Ejercicios Tema 2. Parte 2

Concurrencia con Memoria Compartida

Soluciones

Los ejercicios de esta hoja sirven para que los alumnos puedan ejercitar sus conocimientos en el tema de la sincronización con memoria compartida.

Ejercicio 13

Una línea del metro de Madrid está formada por varios tramos de vía que son recorridos secuencialmente en un único sentido por diferentes trenes.

El ciclo de vida de los procesos que controlan los trenes es el siguiente:

```
    public static void tren(int numTren) {

        sleepRandom(500);
        recorrerTramoA(numTren);

        sleepRandom(500);
        recorrerTramoB(numTren);

        sleepRandom(500);
        recorrerTramoC(numTren);
    }
```

El esquema del programa completo es el siguiente:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer13A_Metro_Plantilla {

    private static final int NUM_TRENES = 5;

    public static void tren(int numTren) {

        sleepRandom(500);
        recorrerTramoA(numTren);

        sleepRandom(500);
        recorrerTramoB(numTren);

        sleepRandom(500);
        recorrerTramoC(numTren);
    }

    private static void recorrerTramoA(int numTren) {
        printlnI("Entra TA T" + numTren);
        sleepRandom(500);
        printlnI("Sale TA T" + numTren);
    }
```

```
    private static void recorrerTramoB(int numTren) {
        printlnI("Entra TB T" + numTren);
        sleepRandom(500);
        printlnI("Sale TB T" + numTren);
    }

    private static void recorrerTramoC(int numTren) {
        printlnI("Entra TC T" + numTren);
        sleepRandom(500);
        printlnI("Sale TC T" + numTren);
    }

    public static void main(String args[]) {
        for (int i = 0; i < NUM_TRENES; i++) {
            createThread("tren", i);
        }
        startThreadsAndWait();
    }
}
```

a) Se pide escribir todo el código de sincronización necesario para garantizar que se cumplan las siguientes restricciones:

- Cada tramo sólo pueda estar ocupado por un tren en cada instante
- No deben producirse interbloqueos
- Si un tren puede recorrer un tramo que se encuentra libre, debe hacerlo sin retrasarse innecesariamente
- Los trenes nunca pueden adelantarse unos a otros.
- La implementación se realizará con semáforos.

b) Ahora se pide generalizar la solución anterior para cualquier número de tramos.

Ejercicio 14. Sincronización de barrera

La Sincronización de Barrera es una sincronización condicional en la que los procesos tienen que esperar a que el resto de procesos lleguen al mismo punto para poder continuar su ejecución. Este tipo de sincronización es muy utilizada en programas concurrentes.

Se pide implementar un programa concurrente con las siguientes características:

- El programa tendrá N procesos del mismo tipo.
- Cada proceso escribe la letra 'A', luego la 'B' y terminan
- Los procesos tienen que esperar que todos hayan escrito la letra 'A' antes de poder escribir la 'B'
- La implementación se realizará con semáforos.

La idea básica al implementar la sincronización de barrera es usar un contador de procesos que han llegado a la barrera. Si el contador es menor que el número total de procesos, entonces el proceso se bloquea. Si el contador es igual al número total de procesos, entonces ese proceso desbloquea a los demás para que todos puedan proseguir su ejecución.

Ejercicio 15. Sincronización de barrera cíclica

Se pide implementar un programa concurrente con las siguientes características:

- El programa tendrá 4 procesos que escriben una única letra indefinidamente. Un proceso escribirá la letra A, otra la B, otra la C y otro la D.
- Cada proceso escribe su letra y espera a que los demás hayan escrito también su letra para poder continuar.
- Uno de los procesos tiene que escribir un guión – después de que todos hayan escrito su letra y antes de que empiecen a escribirlas de nuevo.
- La implementación se realizará con semáforos.

La salida por pantalla del programa con 4 procesos sería:

ACDB-BACD-DCBA-ABCD-BCAD ...

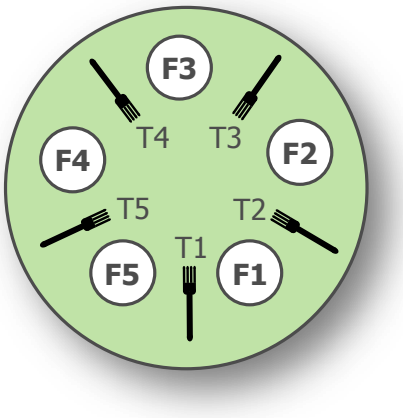
Ejercicio 16. Descarga de múltiples ficheros

Se desea ampliar el ejercicio de descarga de ficheros (Ejercicio 9). Se requiere que cuando los procesos hayan terminado de descargar un fichero, se esperen a que se muestre por pantalla y comiencen a descargar un nuevo fichero, así hasta descargar 10 ficheros. La implementación deberá realizarse íntegramente con semáforos.

Ejercicio 17. Filósofos Comilones

Se pide implementar un programa con semáforos con las siguientes características:

- 5 Filósofos que piensan libremente y comen en una mesa con 5 platos
- Cada filósofo tiene un plato asignado para él sólo
- Hay 5 tenedores, uno entre cada par de platos adyacentes



Cada uno de los 5 filósofos está representado por un proceso. Los filósofos tienen el siguiente ciclo de vida:

- El filósofo inicialmente piensa.
- Se sienta delante de su plato y toma de uno en uno los tenedores situados a ambos lados de su plato.
- Come.

- Cuando finaliza, deja los dos tenedores en su posición original.
- Todo filósofo que come, en algún momento se harta y termina.
- Vuelve a empezar.

Se pide implementar un programa concurrente en Java con SimpleConcurrent que simule la vida de los filósofos. Para ello, se deben cumplir las siguientes restricciones:

- Un filósofo sólo puede comer cuando tenga los dos tenedores.
- Los tenedores se cogen y se dejan de uno en uno.
- Dos filósofos no pueden tener el mismo tenedor simultáneamente (Exclusión Mutua de acceso al tenedor).
- Si varios filósofos tratan de comer al mismo tiempo, uno de ellos debe conseguirlo (Ausencia Interbloqueo).
- Si un filósofo desea comer y tiene competencia, en algún momento lo deberá poder hacerlo (Ausencia de Inanición).
- En ausencia de competencia, un filósofo que quiera comer deberá hacerlo sin retrasos innecesarios (Ausencia retrasos innecesarios).

Se propone el siguiente esquema de código para la resolución del ejercicio:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer17_Filosofos_Plantilla {

    public static final int N_FILOSOFOS = 5;

    public static void filosofo(int numFilosofo) {

        while (true) {
            println("Pensar");
            // Obtener tenedores
            println("Comer");
            // Liberar tenedores
        }
    }

    public static void main(String[] args) {

        for (int i = 0; i < N_FILOSOFOS; i++) {
            createThread("filosofo", i);
        }
        startThreadsAndWait();
    }
}
```

Ejercicio 18. Preguntas cortas sobre semáforos

18.1) Describa la semántica de los métodos acquire() y release() aplicadas sobre semáforos generales.

18.2) ¿Cómo se resuelve el acceso exclusivo de un conjunto de procesos a una variable compartida mediante semáforos?

Concurrencia con Memoria Compartida

18.3) Complete la siguiente tabla cuando el proceso P ejecuta la operación de la columna izquierda sobre el semáforo s.

Operaciones sobre el Semáforo S	Contador de Permisos		Lista de Procesos Bloqueados	
	Antes	Después	Antes	Después
s.acquire()	3		Vacía	
s.acquire()	0		P1	
s.release()	1		Vacía	
s.release()	0		Vacía	
s.release()	0		P1, P2	
s.acquire()	0		Vacía	
s.acquire()	0		P1	
s.acquire()	1		Vacía	
s.release()	0		Vacía	
s.release()	0		P1	
s.release()	1		Vacía	

Solución:

Operaciones sobre el Semáforo s	Contador de Permisos		Lista de Procesos Bloqueados	
	Antes	Después	Antes	Después
s.acquire()	3	2	Vacía	Vacía
s.acquire()	0	0	P1	P1, P
s.release()	1	2	Vacía	Vacía
s.release()	0	1	Vacía	Vacía
s.release()	0	0	P1, P2	P1 o P2
s.acquire()	0	0	Vacía	P
s.acquire()	0	0	P1	P1 y P
s.acquire()	1	0	Vacía	Vacía
s.release()	0	1	Vacía	Vacía
s.release()	0	0	P1	Vacía
s.release()	1	2	Vacía	Vacía

18.4) ¿Cuántos semáforos se precisan para implantar un buffer de tamaño ilimitado? Razone la respuesta.

18.5) Con qué valores se debe inicializar un semáforo para:

a) resolver la sincronización condicional de dos procesos?
b) resolver la exclusión mutua en el acceso a un recurso de N procesos?
c) resolver la exclusión mutua generalizada de N procesos a un recurso que soporta K accesos simultáneos?

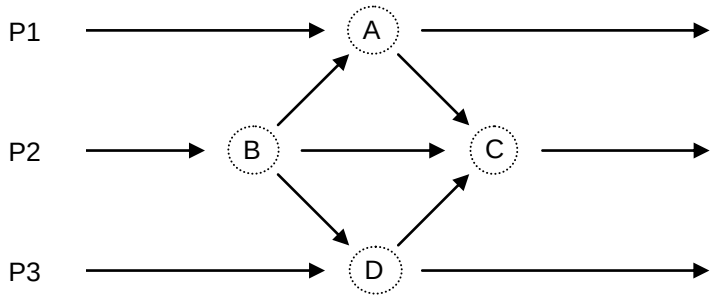
18.6) ¿Por qué las sentencias s.acquire() y s.release() garantizan la exclusión mutua en el acceso al contador de permisos del semáforo s?

18.7) Escribe el código correspondiente al método **puntoSincronización** del siguiente programa utilizando semáforos, para evitar que alguno de los N procesos comience a ejecutar el procedimiento B antes de que alguno de los demás haya finalizado la ejecución del procedimiento A. Incorpora los atributos e inicializaciones necesarios.

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer18_7_Enun {

    private static final int NPROCESOS = 3;

    public static void proceso() {
        print("A");
        puntoSincronizacion();
        print("B");
    }

    private static void puntoSincronizacion() {

    }

    public static void main(String[] args) {
        createThreads(NPROCESOS, "proceso");
        startThreadsAndWait();
    }
}
```

18.8) En un semáforo cuyo contador de permisos es mayor que cero, ¿puede haber algún proceso bloqueado? Justifique su respuesta

18.9) Dado el siguiente diagrama de precedencia de tres procesos:



Escriba en Java con SimpleConcurrent usando Semáforos un programa para cumplir el anterior diagrama de precedencia.

18.10) Describe exactamente todas las situaciones que podrían ocurrir en el siguiente programa. Si adviertes algún error, escribe una solución que garantice que a x se accede bajo exclusión mutua y que P2 nunca termina hasta que ambos procesos hayan incrementado la variable x.

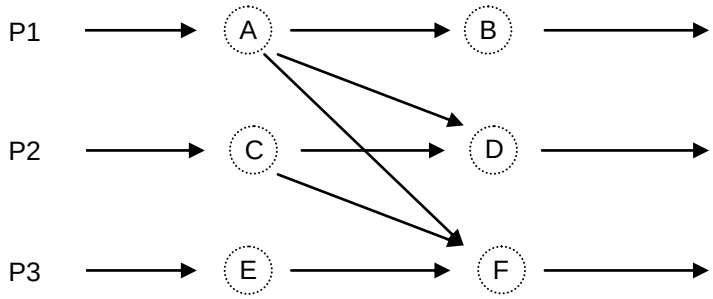
```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;
import es.sidelab.sc.SimpleSemaphore;

public class Ejer18_10_Enun {

    private static int x;
    private static SimpleSemaphore sincronizacion;
    private static SimpleSemaphore exclusion;

    public static void p1() {
        exclusion.acquire();
        x++;
        if (x == 2) {
            sincronizacion.release();
        }
        exclusion.release();
    }

    public static void p2() {
        exclusion.acquire();
        x++;
        if (x == 1) {
            sincronizacion.acquire();
        }
        exclusion.release();
    }

    public static void main(String[] args) {

        x = 0;
        sincronizacion = new SimpleSemaphore(0);
        exclusion = new SimpleSemaphore(1);

        createThread("p1");
        createThread("p2");
        startThreadsAndWait();
    }
}
```

18.11) ¿Hay algún error en el siguiente programa?

```
package ejercicio;

import es.sidelab.sc.SimpleSemaphore;
import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer18_11_Enun {

    private static int x;
    private static int y;
    private static SimpleSemaphore sY;
    private static SimpleSemaphore sX;

    public static void pA() {
        sX.acquire();
        x++;
        sY.acquire();
        y *= x;
        sY.release();
        sX.release();
    }
```

```
    public static void pB() {
        sY.acquire();
        y++;
        sX.acquire();
        x *= y;
        sX.release();
        sY.release();
    }

    public static void main(String[] args) {

        x = 2;
        y = 3;

        sX = new SimpleSemaphore(1);
        sY = new SimpleSemaphore(1);

        createThread("pA");
        createThread("pB");

        startThreadsAndWait();
    }
}
```

4.12) Escribe en Java con SimpleConcurrent el programa que cumpla el siguiente diagrama de precedencia de procesos:



18.13) El siguiente código trata de implantar una sincronización de barrera para un número de procesos determinado por la constante N.

```
package ejercicio;

import es.sidelab.sc.SimpleSemaphore;
import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer18_13_Enun {

    private static final int N = 3;
    private static volatile int nProcesos;
    private static SimpleSemaphore sb;
    private static SimpleSemaphore emNProcesos;

    public static void proceso() {
        while (true) {
            print("A");
            puntoSincronizacion();
            print("B");
        }
    }
```

```
    private static void puntoSincronizacion() {
        emNProcesos.acquire();
        nProcesos++;
        if (nProcesos == N) {
            nProcesos = 0;
            emNProcesos.release(); //I0
            for (int i = 0; i < N; i++) {
                sb.release(); //I1
            }
        } else {
            emNProcesos.release();
            sb.acquire();
        }
    }

    public static void main(String[] args) {
        nProcesos = 0;

        sb = new SimpleSemaphore(0);
        emNProcesos = new SimpleSemaphore(1);

        createThreads(N, "proceso");
        startThreadsAndWait();
    }
}
```

Explica brevemente por qué no se sincronizan adecuadamente los procesos en la sincronización de barrera en este ejemplo. Observa las instrucciones etiquetadas con I0 e I1.

18.14) Dados los dos siguientes procesos concurrentes:

```
    public static void p1(){
        sX.acquire();
        sY.acquire();
        x += y;
        sY.release();
        sX.release();
    }

    public static void p2(){
        sY.acquire();
        sX.acquire();
        y += x;
        sX.release();
        sY.release();
    }
```

Si inicialmente sX y sY tienen el contador de permisos a valor 1, y X e Y tienen valor 1, describe todas las situaciones que podrían ocurrir. Si adviertes algún error, corrígelo.

18.15) Sea el siguiente programa en el que N procesos compiten por utilizar un cierto recurso, de manera que como máximo K de estos procesos puedan usar el recurso concurrentemente. ¿Cómo se denomina este problema clásico? Escriba en Java con SimpleConcurrent una solución a este problema usando semáforos.

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer18_15_Enun {

    private static final int N_PROCESOS = 4;
    private static final int K = 2;

    public static void p() {
        while(true){
            // Obtener recurso
            // Usar recurso
            // Liberar recurso
            // Hacer otras cosas
        }
    }

    public static void main(String[] args) {
        createThreads(N_PROCESOS,"p");
        startThreadsAndWait();
    }
}
```

18.16) Dibuje el diagrama de precedencia del siguiente programa.

```
package exercises.semaphores;

import static simpleconcurrent.SimpleConcurrent.*;
import simpleconcurrent.SimpleSemaphore;

public class Ejer4_16_Enun {

    private static SimpleSemaphore[] sinc = new SimpleSemaphore[2];

    public static void puntoSinc(int numProc) {

        if (numProc == 0) {
            sinc[0].release();
            sinc[1].release();
        } else {
            if (numProc == 1) {
                sinc[1].release();
                sinc[0].acquire();
            } else {
                sinc[1].acquire();
                sinc[1].acquire();
            }
        }
    }

    public static void proceso(int numProc) {
        print("A"+numProc+" ");
        puntoSinc(numProc);
        print("B"+numProc+" ");
    }




    public static void main(String[] args) {
```

```
        sinc[0] = new SimpleSemaphore(0);
        sinc[1] = new SimpleSemaphore(0);

        for (int i = 0; i < 3; i++) {
            createThread("proceso", i);
        }

        startThreadsAndWait();
    }
}
```

18.17) Dados el siguiente programa concurrente:

```
package ejercicio;

import es.sidelab.sc.SimpleSemaphore;
import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer18_17_Enun {

    private static SimpleSemaphore s;

    public static void p1(){
        print("A");
        s.release();
    }

    public static void p2(){
        s.acquire();
        print("B");
        s.release();
        s.release();
    }

    public static void p3(){
        s.acquire();
        s.acquire();
        print("C");
    }

    public static void main(String[] args){

        s = new SimpleSemaphore(0);

        createThread("p1");
        createThread("p2");
        createThread("p3");

        startThreadsAndWait();
    }
}
```

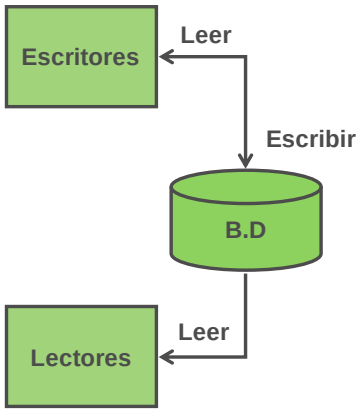
¿Se puede asegurar que siempre la ejecución de A precede a la de B y la de B precede a la de C?

Ejercicio 19. Lectores-Escritores

Se desea implementar un programa concurrente con semáforos con procesos lectores y procesos escritores. En este ejercicio vamos a un ver un ejemplo típico de acceso combinado (concurrente y exclusivo) a variables compartidas. Este tipo de acceso se tiene en los sistemas gestores de bases de datos.

Concurrencia con Memoria Compartida

Los procesos lectores pueden leer información de la base de datos y los procesos escritores pueden escribir información en ella. Para simplificar la resolución del ejercicio, las operaciones de los procesos lectores y escritores se implementarán como escrituras por pantalla.



La solución se basará en el siguiente esquema:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer19_LectoresEscritores_Plantilla {

    public static void inicioLectura() { }

    public static void finLectura() { }

    public static void inicioEscritura() { }

    public static void finEscritura() { }

    public static void lector() {
        while(true){
            inicioLectura();
            println("Leer datos");
            finLectura();
            println("Procesar datos");
        }
    }

    public static void escritor() {
        while (true) {
            println("Generar datos");
            inicioEscritura();
            println("Escribir datos");
            finEscritura();
        }
    }

    public static void main(String[] args) {
        createThreads(5, "lector");
        createThreads(3, "escritor");
        startThreadsAndWait();
    }
}
```

Concurrencia con Memoria Compartida

Se pide implementar los métodos inicioLectura(), finLectura(), inicioEscritura(), finEscritura() de forma que se cumplan las siguientes propiedades:

- Cualquier número de lectores puede acceder a la vez a la BD, siempre que no haya escritores accediendo.
- El acceso a la BD de los escritores es exclusivo. Mientras haya algún lector leyendo, ningún escritor puede acceder a la BD, pero otros lectores sí.
- Se puede tener varios escritores trabajando, aunque estos se deberán sincronizar para que la escritura se lleve a cabo de uno en uno.
- Se da prioridad a los escritores. Ningún lector puede acceder a la BD cuando haya escritores que desean hacerlo (aunque esto pueda causar inanición de Lectores).