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) {
        println("Entra TA T" + numTren);
        sleepRandom(500);
        println("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.

Solución: La idea básica de la solución es poner cada tramo de la vía bajo exclusión mutua. Eso garantiza que no hay dos trenes en el mismo tramo de vía. Para evitar los adelantamientos, basta con salir de la exclusión mutua del tramo anterior cuando ya se ha entrado en la exclusión mutua del tramo siguiente.

```
package ejercicio;

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

public class Ejer13A_Metro {

    private static final int NUM_TRENES = 5;
    private static SimpleSemaphore emTramoA;
    private static SimpleSemaphore emTramoB;
    private static SimpleSemaphore emTramoC;

    public static void tren(int numTren) {

        sleepRandom(500);

        emTramoA.acquire();
        recorrerTramoA(numTren);

        sleepRandom(500);
```

```
        emTramoB.acquire();
        emTramoA.release();
        recorrerTramoB(numTren);

        sleepRandom(500);

        emTramoC.acquire();
        emTramoB.release();
        recorrerTramoC(numTren);

        emTramoC.release();
    }

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

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

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


    public static void main(String args[]){

        emTramoA = new SimpleSemaphore(1);
        emTramoB = new SimpleSemaphore(1);
        emTramoC = new SimpleSemaphore(1);

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

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

```
package ejercicio;

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

public class Ejer13B_Metro {

    private static final int NUM_TRAMOS = 6;
    private static final int NUM_TRENES = 3;
    private static SimpleSemaphore[] emTramos;
```

```
    public static void tren(int numTren) {
        for (int numTramo = 0; numTramo < NUM_TRAMOS; numTramo++) {

            sleepRandom(500);
            emTramos[numTramo].acquire();

            if(numTramo != 0){
                emTramos[numTramo-1].release();
            }

            recorrerTramo(numTren, numTramo);
        }

        emTramos[NUM_TRAMOS-1].release();
    }

    private static void recorrerTramo(int numTren, int numTramo) {
        char tramo = (char)('A' + numTramo);
        println("Entra T"+tramo+" T" + numTren);
        sleepRandom(500);
        println("Sale T"+tramo+" T" + numTren);
    }

    public static void main(String args[]){

        emTramos = new SimpleSemaphore[NUM_TRAMOS];
        for(int i=0; i<NUM_TRAMOS; i++){
            emTramos[i] = new SimpleSemaphore(1);
        }

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

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.

1ª Aproximación Incorrecta - Puede provocar interbloqueo (deadlock) porque nProcesos no está bajo exclusión mutua y se pueden producir errores al contar

```
package ejercicio;

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

public class Ejer14_SincBarrera_Mal1 {

    private static final int NPROCESOS = 3;

    private static volatile int nProcesos;
    private static SimpleSemaphore sb;

    public static void proceso() {
        print("A");
        nProcesos++;
        if (nProcesos < NPROCESOS) {
            sb.acquire();
        } else {
            for (int i = 0; i < NPROCESOS - 1; i++) {
                sb.release();
            }
        }
        print("B");
    }

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

        sb = new SimpleSemaphore(0);

        createThreads(NPROCESOS, "proceso");

        startThreadsAndWait();
    }
}
```

2ª Aproximación Incorrecta - Si se deja la consulta del contador fuera de la Exclusión Mutua, puede ocurrir que dos procesos hagan release().

```
package ejercicio;

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

public class Ejer14_SincBarrera_Mal2 {

    private static final int NPROCESOS = 3;

    private static volatile int nProcesos;
    private static SimpleSemaphore sb;
    private static SimpleSemaphore emNProcesos;

    public static void proceso() {
        print("A");

        emNProcesos.acquire();
        nProcesos++;
        emNProcesos.release();
```

```
        if (nProcesos < NPROCESOS) {
            sb.acquire();
        } else {
            for (int i = 0; i < NPROCESOS - 1; i++) {
                sb.release();
            }
        }
        print("B");
    }

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

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

        createThreads(NPROCESOS, "proceso");

        startThreadsAndWait();
    }
}
```

Solución: Si el proceso no es el último, libera la EM y se bloquea. Si es el último, sale de la EM y desbloquea a los demás procesos.

```
package ejercicio;

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

public class Ejer14_SincBarrera {

    private static final int NPROCESOS = 3;

    private static volatile int nProcesos;
    private static SimpleSemaphore sb;
    private static SimpleSemaphore emNProcesos;

    public static void proceso() {
        print("A");

        emNProcesos.acquire();
        nProcesos++;
        if (nProcesos < NPROCESOS) {
            emNProcesos.release();
            sb.acquire();
        } else {
            emNProcesos.release();
            for (int i = 0; i < NPROCESOS - 1; i++) {
                sb.release();
            }
        }

        print("B");
    }
```

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

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

        createThreads(NPROCESOS, "proceso");

        startThreadsAndWait();
    }
}
```

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 ...

1ª Aproximación Incorrecta La primera aproximación se basa en usar una sincronización de barrera como la que hemos visto en el ejercicio 14.

```
package ejercicio;

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

public class Ejer15_SincBarreraCicl_Mal {

    private static volatile int nProcesos;
    private static SimpleSemaphore sb;
    private static SimpleSemaphore emNProcesos;

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

    public static void procesoB() {
        while(true){
            print("B");
            sincronizacion();
        }
    }
```

```
    public static void procesoC() {
        while(true){
            print("C");
            sincronizacion();
        }
    }

    public static void procesoD() {
        while(true){
            print("D");
            sincronizacion();
        }
    }

    public static void sincronizacion(){

        emNProcesos.acquire();
        nProcesos++;
        if (nProcesos < 4) {
            emNProcesos.release();
            sb.acquire();
        } else {

            println("-");

            nProcesos = 0;
            emNProcesos.release();

            for (int i = 0; i < 3; i++) {
                //sleepRandom(500); Simular condiciones de carrera
                sb.release();
            }
        }
    }

    public static void main(String[] args) {

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

        createThread("procesoA");
        createThread("procesoB");
        createThread("procesoC");
        createThread("procesoD");

        startThreadsAndWait();
    }
}
```

El problema que tiene esta solución es que es posible que un hilo recién despertado muestre su letra y le de tiempo a invocar sb.acquiere() antes de que el hilo despertador haya despertado al resto de sus compañeros. En ese caso, como el semáforo es aleatorio, es posible que ese hilo que ya ha hecho su trabajo pueda desbloquearse de nuevo, volviendo a realizar su trabajo. Si descomentamos la línea con el sleepRandom(500) veremos que la salida será similar a esta:

```
CDAB-
DDDB-
DDBD-
BBDB-
DDBD-
```

2ª Aproximación incorrecta: Para solventar los problemas con la aproximación anterior, podríamos pensar en que la exclusión mutua del contador de procesos no se libere hasta que el proceso "despertador" haya despertado a todos los demás, con eso evitaríamos que un proceso recién despertado pueda invocar sb.acquiere() antes de que todos sus compañeros se hayan despertado. Esta aproximación sería la siguiente:

```
package ejercicio;

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

public class Ejer15_SincBarreraCicl_Mal2 {

    private static volatile int nProcesos;
    private static SimpleSemaphore sb;
    private static SimpleSemaphore emNProcesos;

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

    public static void procesoB() {
        while(true){
            print("B");
            sincronizacion();
        }
    }

    public static void procesoC() {
        while(true){
            print("C");
            sincronizacion();
        }
    }

    public static void procesoD() {
        while(true){
            print("D");
            sincronizacion();
        }
    }

    public static void sincronizacion(){
        emNProcesos.acquire();
        nProcesos++;
        if (nProcesos < 4) {
            emNProcesos.release();
            //sleepRandom(500); Simular condiciones de carrera
            sb.acquire();
        } else {
            println("-");
            nProcesos = 0;
            for (int i = 0; i < 3; i++) {
                sb.release();
            }
            emNProcesos.release();
        }
    }
```

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

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

        createThread("procesoA");
        createThread("procesoB");
        createThread("procesoC");
        createThread("procesoD");

        startThreadsAndWait();
    }
}
```

Pero esta aproximación tampoco es correcta. El problema está en que desde que un proceso se da cuenta de que no es el último (nProcesos < 4) e invoca sb.acquiere() puede pasar mucho tiempo. Y en ese tiempo, otro proceso puede haberse bloqueado, despertado y vuelto a bloquear. Este comportamiento se puede similar descomentando la sentencia con sleepRandom(500) y la salida será similar a esta:

```
CABD-
BCDB-
BAAC-
DCDA-
BCBA-
```

Es muy difícil darse cuenta de que estas soluciones son incorrectas porque si no se intercalan las sentencias de sleep en la mayoría de los casos los programas funcionan correctamente. Pero en los casos extremos (que forzamos con los sleep) los programas se comportan de forma errónea. En los programas en producción, estas situaciones anómalas se suelen producir cuando el sistema está muy cargado, por lo que son muy difíciles de resolver.

Solución: Una de las posibles soluciones consiste en que los hilos que son desbloqueados tengan que notificar al hilo despertador que realmente se han desbloqueado y que han salido de la barrera. Cuando el hilo despertador recibe todas las notificaciones, entonces libera la exclusión mutua de la barrera para comenzar otro ciclo. Para implementar la solución se ha usado un método de la clase SimpleSemaphore que permite hacer release de varios permisos en una única llamada:

```
package ejercicio;

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

public class Ejer15_SincBarreraCicl {

    private static volatile int nProcesos;
    private static SimpleSemaphore sb;
    private static SimpleSemaphore desbloqueo;
    private static SimpleSemaphore emNProcesos;

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

```
    public static void procesoB() {
        while (true) {
            print("B");
            sincronizacion();
        }
    }

    public static void procesoC() {
        while (true) {
            print("C");
            sincronizacion();
        }
    }

    public static void procesoD() {
        while (true) {
            print("D");
            sincronizacion();
        }
    }

    public static void sincronizacion() {

        emNProcesos.acquire();
        nProcesos++;
        if (nProcesos < 4) {
            emNProcesos.release();
            sleepRandom(500); // Simular condiciones de carrera
            sb.acquire();
            desbloqueo.release();
        } else {

            println("-");
            nProcesos = 0;

            sb.release(3);
            desbloqueo.acquire(3);
            emNProcesos.release();
        }
    }

    public static void main(String[] args) {

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

        createThread("procesoA");
        createThread("procesoB");
        createThread("procesoC");
        createThread("procesoD");

        startThreadsAndWait();
    }
}
```

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.

Solución:

La solución se consigue implementando una sincronización de barrera cuando se ha descargado un fichero completo, de forma que el resto de hilos se esperan para volver a descargar un nuevo fichero. Para ello, basta con modificar el método "downloader()" de la solución al Ejercicio 9 para descargar 10 ficheros en vez de 1, reinicializando todas las variables cuando todos los hilos han finalizado su trabajo de descarga.

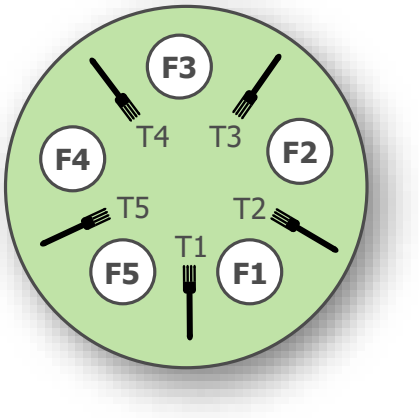
```
package ejercicio;

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

public class Ejer16_Downloader {

    private static final int N_FRAGMENTOS = 10;
    private static final int N_HILOS = 3;

    private static volatile int[] fichero = new int[N_FRAGMENTOS];
    private static volatile int fragPendiente = 0;
    private static volatile int hilosTerminados = 0;

    private static SimpleSemaphore emFragPendiente;

    private static SimpleSemaphore emHilosTerminados;
    private static SimpleSemaphore sb;
    private static SimpleSemaphore desbloqueo;

    private static int descargarDatos(int numFragment) {
        sleepRandom(1000);
        return numFragment * 2;
    }

    private static void mostrarFichero() {
        println("-------------------------------------------------");
        print("File = [");
        for (int i = 0; i < N_FRAGMENTOS; i++) {
            print(fichero[i] + ",");
        }
        println("]");
    }
```

```
    public static void downloader() {

        for (int i = 0; i < 10; i++) {

            descargarFragmentos();

            emHilosTerminados.acquire();
            hilosTerminados++;
            if (hilosTerminados < N_HILOS) {

                emHilosTerminados.release();
                sb.acquire();
                desbloqueo.release();

            } else {

                mostrarFichero();
                fragPendiente = 0;
                hilosTerminados = 0;

                sb.release(N_HILOS-1);
                desbloqueo.acquire(N_HILOS-1);
                emHilosTerminados.release();
            }
        }
    }

    private static void descargarFragmentos() {
        while (true) {

            emFragPendiente.acquire();
            if (fragPendiente == N_FRAGMENTOS) {
                emFragPendiente.release();
                break;
            }

            int fragDescargar = fragPendiente;
            fragPendiente++;
            emFragPendiente.release();

            println(getThreadName() + ": Descargando fragmento " + fragDescargar);

            int downloadedData = descargarDatos(fragDescargar);

            println(getThreadName() + ": Escribiendo fragmento " + fragDescargar);

            fichero[fragDescargar] = downloadedData;
        }
    }

    public static void main(String[] args) {

        emFragPendiente = new SimpleSemaphore(1);

        emHilosTerminados = new SimpleSemaphore(1);
        sb = new SimpleSemaphore(0);
        desbloqueo = new SimpleSemaphore(0);

        createThreads(N_HILOS, "downloader");

        startThreadsAndWait();
    }
}
```

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();
    }
}
```

1º Aproximación Incorrecta: La siguiente puede ser la primera aproximación más evidente, pero no es correcta. Esta aproximación propone un semáforo por cada tenedor y pone cada tenedor bajo exclusión mutua:

```
package ejercicio;

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

public class Ejer17_Filosofos_Mal {

    public static final int N_FILOSOFOS = 5;

    public static SimpleSemaphore[] tenedores = new SimpleSemaphore[N_FILOSOFOS];

    public static void filosofo(int numFilosofo) {

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

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

             tenedores[tIzq].acquire();
             sleepRandom(500); //Simular interbloqueo
             tenedores[tDer].acquire();

             println("Comer");

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

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

        for (int i = 0; i < N_FILOSOFOS; i++) {
            tenedores[i] = new SimpleSemaphore(1);
        }

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

Esta solución no es correcta porque si todos los filósofos avanzan a la vez, cogen el tenedor de su izquierda y se bloquean esperando el de su derecha (interbloqueo).

Solución 1: Para evitar los problemas de esta solución, podemos considerar que existe un "comedor" en el que caben todos los filósofos menos 1. De esta forma, nunca se producirá interbloqueo entre los filósofos porque al menos uno de ellos podrá coger los dos tenedores.

```
package ejercicio;

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

public class Ejer17_Filosofos_1 {

    public static final int N_FILOSOFOS = 5;

    public static SimpleSemaphore[] tenedores = new SimpleSemaphore[N_FILOSOFOS];
    public static SimpleSemaphore comedor;

    public static void filosofo(int numFilosofo) {

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

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

            comedor.acquire();

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

            println("Comer");

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

            comedor.release();

        }
    }
```

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

        comedor = new SimpleSemaphore(N_FILOSOFOS - 1);
        for (int i = 0; i < N_FILOSOFOS; i++) {
            tenedores[i] = new SimpleSemaphore(1);
        }

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

Solución 2: Otra posible solución consiste en que uno de los filósofos sea zurdo y empiece cogiendo los tenedores por la derecha en vez de por la izquierda.

```
package ejercicio;

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

public class Ejer17_Filosofos_2 {

    public static final int N_FILOSOFOS = 5;

    public static SimpleSemaphore[] tenedores = new SimpleSemaphore[N_FILOSOFOS];
    public static SimpleSemaphore comedor;

    public static void filosofo(int numFilosofo, boolean diestro) {

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

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

            if (diestro) {
                tenedores[tIzq].acquire();
                tenedores[tDer].acquire();
            } else {
                tenedores[tDer].acquire();
                tenedores[tIzq].acquire();
            }

            println("Comer");

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

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

        for (int i = 0; i < N_FILOSOFOS; i++) {
            tenedores[i] = new SimpleSemaphore(1);
        }

        for (int i = 0; i < N_FILOSOFOS - 1; i++) {
            createThread("filosofo", i, true);
        }
        createThread("filosofo", N_FILOSOFOS - 1, false);

        startThreadsAndWait();
    }
}
```

Ejercicio 18. Preguntas cortas sobre semáforos

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

s.acquire() – Si el número de permisos del semáforo s es igual a cero el proceso se bloquea en el semáforo s; si no, se decrementa en una unidad el contador de permisos del semáforo s y el proceso prosigue su ejecución.

s.release () – Si hay procesos bloqueados en el semáforo s se desbloquea a uno de ellos; si no, se incrementa en una unidad el contador de permisos del semáforo s. En ambos casos, el proceso prosigue su ejecución.

Ambos métodos son, por definición, instrucciones atómicas.

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

Con semáforos se asocia un semáforo a esa variable, y se inicializa con valor 1. Cuando un proceso debe acceder a la variable, primero ejecuta una operación acquire() sobre el semáforo para obtener el acceso exclusivo sobre ella; después consulta y/o modifica su valor; y finalmente ejecuta una operación release() sobre el semáforo para liberarla.

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 Bloquedados	
	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 Bloquedados	
	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.

2 semaforos. Uno para la exclusión mutua sobre el buffer de datos y otro para sincronizar a los consumidores cuando no hay datos en el buffer.

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?

a) 0 b) 1 c) K

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?

Porque, por definición, son instrucciones atómicas

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();
    }
}
```

Solución:

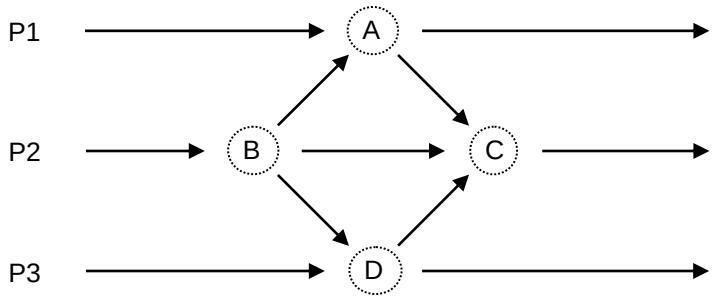
```
package ejercicio;

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

public class Ejer18_7 {

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

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

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

    public static void main(String[] args) {
        nProcesos = 0;
        sb = new SimpleSemaphore(0);
        emNProcesos = new SimpleSemaphore(1);
        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

NO. Sólo puede haber procesos bloqueados en el semáforo cuando el contador vale 0.

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.

Solución:

```
package ejercicio;

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

public class Ejer18_9 {

    private static SimpleSemaphore semA;
    private static SimpleSemaphore semC;
    private static SimpleSemaphore semD;
```

```
    public static void proceso1() {
        semA.acquire();
        print("A");
        semC.release();
    }

    public static void proceso2() {
        print("B");
        semA.release();
        semD.release();
        semC.acquire();
        semC.acquire();
        print("C");
    }

    public static void proceso3() {
        semD.acquire();
        print("D");
        semC.release();
    }

    public static void main(String[] args) {

        semA = new SimpleSemaphore(0);
        semC = new SimpleSemaphore(0);
        semD = new SimpleSemaphore(0);

        createThread("proceso1");
        createThread("proceso2");
        createThread("proceso3");

        startThreadsAndWait();
    }
}
```

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();
    }
}
```

Solución:

a) Si P1 entra primero en su sección crítica, incrementa x y termina. Luego entra P2, incrementa x y termina. b) Si P2 entra primero, entonces incrementa x, ejecuta sincronizacion.acquire() y se bloquea. Entonces P1 nunca podrá entrar en su sección crítica. Por tanto, se produce un interbloqueo de P1 y P2. La solución consiste en que P2 se bloquee fuera de su sección crítica, por ejemplo:

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

Aunque otra solución más sencilla para cumplir con los requisitos del programa puede ser la siguiente:

```
package ejercicio;

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

public class Ejer18_10_2 {

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

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

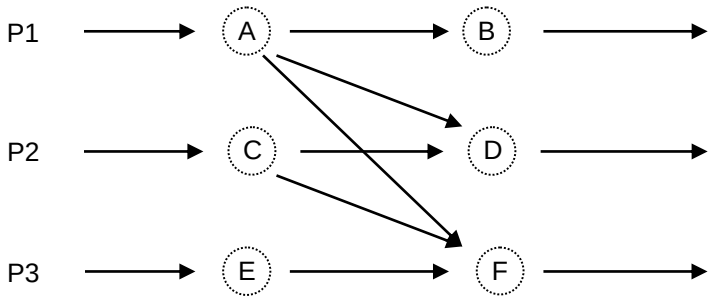
```
    public static void p2() {
        exclusion.acquire();
        x++;
        exclusion.release();
        sincronizacion.acquire();
    }

    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();
    }
}
```

Si el proceso pA ejecuta su primer acquire(), y antes de que ejecute su segundo acquire(), el proceso B ejecuta su primer acquire() (o viceversa) se produce un interbloqueo entre ambos.

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



Solución:

```
package ejercicio;

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

public class Ejer18_12 {

    private static SimpleSemaphore semD;
    private static SimpleSemaphore semF;

    public static void proceso1() {
        print("A");
        semD.release();
        print("B");
    }

    public static void proceso2() {
        print("C");
        semF.release();
        semD.acquire();
        print("D");
    }

    public static void proceso3() {
        print("E");
        semF.acquire();
        semF.acquire();
        print("F");
    }

    public static void main(String[] args) {

        semD = new SimpleSemaphore(0);
        semF = new SimpleSemaphore(0);

        createThread("proceso1");
        createThread("proceso2");
        createThread("proceso3");

        startThreadsAndWait();
    }
```

```
}
```

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.

El último proceso que llega al punto de sincronización hace N release(), cuando sólo hay N-1 procesos bloqueados en el semáforo, por lo que en la siguiente iteración el primer proceso que llegue al punto de sincronización se encuentra el contador del semáforo con valor 1 y no se bloquea esperando a que llegue el último.

Pero además hay otro problema: la liberación de la exclusión mutua en la instrucción I0 permite que un proceso que se desbloquee como resultado de un release() en I1 se adelante a los demás, y antes de que se puedan desbloquear todos, entre de nuevo en la sección crítica, invoque acquire(), y se desbloquee como resultado del siguiente adquire() en I1. Lo cual puede producir la inanición de aquellos procesos que no pueden desbloquearse porque el proceso que se ha adelantado se desbloquea en su lugar.

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.

a) Si P1 ejecuta primero sus dos acquire(), asigna 2 a X. Cuando P1 ejecute sus dos release(), entonces P2 podrá asignar 3 a Y, y los dos procesos terminarán correctamente.

b) Si P2 ejecuta primero sus dos acquire(), asigna 2 a Y. Cuando P2 ejecute sus dos release(), entonces P2 podrá asignar 3 a X, y los dos procesos terminarán correctamente.

c) Si P1 o P2 ejecuta su primer acquire() y después el otro proceso ejecuta el suyo, hay interbloqueo de los dos procesos al ejecutar sus segundos acquire(). La solución consiste en intercambiar el orden de las instrucciones acquire() en uno de los procesos.

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();
    }
}
```

Solución:

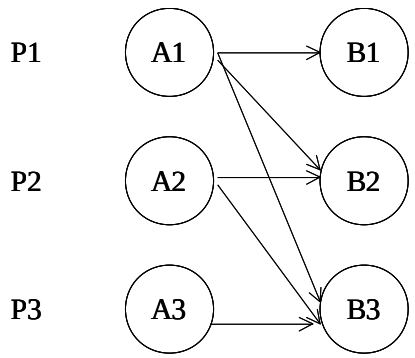
Este problema se denomina exclusión mutua generalizada. Una solución usando semáforos es la siguiente:

```
package ejercicio;

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

public class Ejer18_15 {

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

    private static SimpleSemaphore emGen;

    public static void p() {
        while(true){
            emGen.acquire();
            // Usar recurso
            emGen.release();
            // Hacer otras cosas
        }
    }

    public static void main(String[] args) {
        emGen = new SimpleSemaphore(K);
        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?

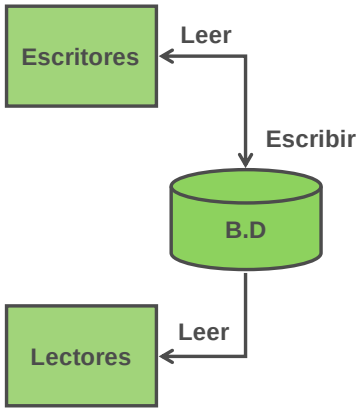
No. Si primero se ejecuta la primera instrucción de P_3, éste queda bloqueado en el semáforo S; si luego se ejecutan todas las instrucciones de P_1, P_3 queda en estado de preparado; si se ejecuta la segunda instrucción de P_3, éste queda de nuevo bloqueado en el semáforo S; el único proceso en estado de preparado será P_2, y quedará también bloqueado en cuanto ejecute su primera instrucción. Por tanto, se produce interbloqueo y no se puede asegurar que B precede a 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.

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.

Concurrencia con Memoria Compartida



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();
    }
}
```

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).

Solución:

Para implementar una solución, nos basamos en los siguientes principios de funcionamiento:

- Cuando un lector intenta entrar en la BD, primero mira para ver si hay algún escritor en ella (trabajando) o algún escritor a la espera de entrar. Si no hay ningún escritor trabajando o a la espera, entonces entra el lector. En caso contrario, se bloquea a la espera de que terminen los escritores.
- Cuando un escritor intenta entrar en la BD, sólo mira por si hay un lector en ella (trabajando), pero no mira si hay algún lector a la espera, porque los escritores tienen prioridad sobre los lectores. Si no hay ningún lector trabajando, entonces entra el escritor. En caso contrario, se bloque a la espera de que terminen los escritores.
- Cuando un lector finaliza el acceso a la base de datos y es el último lector trabajando, desbloquea a los escritores que estuvieran esperando.
- Cuando un escritor finaliza el acceso a la base de datos y es el último escritor trabajando, desbloquea a los lectores que estuvieran esperando.

Con estos principios básicos de funcionamiento, la solución sería:

```
package ejercicio;

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

public class Ejer19_LectoresEscritores_1 {

   // Número de escritores que han conseguido el acceso a la BD
   private static int escritoresBD = 0;

   // Número de lectores que han conseguido el acceso a la BD
   private static int lectoresBD = 0;

   // Lectores esperando a que finalicen los escritores
   private static int lectoresEspera;

   // Escritores esperando a que finalicen los lectores
   private static int escritoresEspera;

   // Exclusión mutura de las variables de control
   private static SimpleSemaphore emControl;

   // Exclusión mutura para el acceso a los escritores
   private static SimpleSemaphore emEscritura;

   // Bloqueo de los lectores cuando hay escritores
   private static SimpleSemaphore esperaFinEscritores;
```

```
// Bloqueo de los escritores cuando hay lectores
private static SimpleSemaphore esperaFinLectores;

public static void inicioLectura() {
    emControl.acquire();
    printlnI("inicioLectura");
    if (escritoresBD == 0 && escritoresEspera == 0) {
        lectoresBD++;
        printlnI("lectTrab="+lectoresBD);
        emControl.release();
    } else {
        lectoresEspera++;
        printlnI("lectEspera="+lectoresEspera);
        emControl.release();
        esperaFinEscritores.acquire();
    }
}

public static void finLectura() {
    emControl.acquire();
    lectoresBD--;
    printlnI("finLectura");
    printlnI("lectTrab="+lectoresBD);
    if (lectoresBD == 0) {
        printlnI("escriEspera="+escritoresEspera);
        for(int i=0; i<escritoresEspera; i++) {
            escritoresBD++;
            esperaFinLectores.release();
        }
        escritoresEspera = 0;
    }
    emControl.release();
}

public static void inicioEscritura() {
    emControl.acquire();
    printlnI("inicioEscritura");
    if (lectoresBD == 0) {
        escritoresBD++;
        printlnI("escTrab="+escritoresBD);
        emControl.release();
    } else {
        escritoresEspera++;
        printlnI("escEspera="+escritoresEspera);
        emControl.release();
        esperaFinLectores.acquire();
    }

    emEscritura.acquire();
    printlnI("Escribiendo...");
}
```

```
    public static void finEscritura() {
        printlnI("FinEscribiendo...");
        emEscritura.release();

        emControl.acquire();
        escritoresBD--;
        printlnI("finEscritura");
        printlnI("escriTrab="+escritoresBD);
        if (escritoresBD == 0) {
            printlnI("lectEspera="+lectoresEspera);
            for(int i=0; i<lectoresEspera; i++){
                lectoresBD++;
                esperaFinEscritores.release();
            }
            lectoresEspera=0;
        }
        emControl.release();
    }

    public static void lector() {
        while (true) {
            inicioLectura();
            printlnI("Leer datos");
            sleepRandom(300);
            finLectura();
            printlnI("Procesar datos");
            sleepRandom(500);
        }
    }

    public static void escritor() {
        while (true) {
            printlnI("Generar datos");
            sleepRandom(2000);
            inicioEscritura();
            printlnI("Escribir datos");
            sleepRandom(500);
            finEscritura();
        }
    }

    public static void main(String[] args) {

        emControl = new SimpleSemaphore(1);
        emEscritura = new SimpleSemaphore(1);
        esperaFinEscritores = new SimpleSemaphore(0);
        esperaFinLectores = new SimpleSemaphore(0);

        createThreads(5, "lector");
        createThreads(3, "escritor");
        startThreadsAndWait();
    }
}
```

No obstante, hay otras soluciones posibles. En este caso se presenta otra solución que utiliza el concepto de procesos activos y procesos trabajando para representar de otra forma los procesos que quieren entrar en la base de datos frente a los que realmente han entrado. El código sería el siguiente:

```
package ejercicio;

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

public class Ejer19_LectoresEscritores_2 {

    // Número de escritores que han iniciado inicioEscritura()
    // Están activos hasta que finalizan finEscritura()
    private static int escritoresActivos = 0;

    // Número de escritores que han conseguido el acceso
    // a la BD (Han terminado inicioEscritura())
    // Están trabajando hasta que finalizan finEscritura()
    // Siempre se cumple escritoresActivos >= escritoresTrabajando
    private static int escritoresBD = 0;

    // Número de lectores que han iniciado inicioLectura()
    // Están activos hasta que finalizan finLectura()
    // Un lector activo puede estar esperando a obtener el
    // acceso a los datos, que se le concederá cuando no
    // haya escritores activos
    private static int lectoresActivos = 0;

    // Número de lectores que han conseguido el acceso a
    // la BD (Han terminado inicioLectura())
    // Están trabajando hasta que finalizan finLectura()
    // Siempre se cumple lectoresActivos >= lectoresTrabajando
    private static int lectoresBD = 0;

    // Exclusión mutura de las variables de control
    private static SimpleSemaphore emControl;

    // Exclusión mutura para el acceso a los escritores
    private static SimpleSemaphore emEscritura;

    // Bloqueo de los lectores cuando hay escritores
    private static SimpleSemaphore esperaFinEscritores;

    // Bloqueo de los escritores cuando hay lectores
    private static SimpleSemaphore esperaFinLectores;

    public static void inicioLectura() {
        emControl.acquire();
        lectoresActivos++;
        if (escritoresBD == 0 && escritoresActivos == 0) {
            lectoresBD++;
            emControl.release();
        } else {
            emControl.release();
            esperaFinEscritores.acquire();
        }
    }









    public static void finLectura() {
        emControl.acquire();
```

```
        lectoresActivos--;
        lectoresBD--;
        if (lectoresBD == 0) {
            while (escritoresActivos > escritoresBD) {
                escritoresBD++;
                esperaFinLectores.release();
            }
        }
        emControl.release();
    }

    public static void inicioEscritura() {
        emControl.acquire();
        escritoresActivos++;
        if (lectoresBD == 0) {
            escritoresBD++;
            emControl.release();
        } else {
            emControl.release();
            esperaFinLectores.acquire();
        }

        emEscritura.acquire();
    }

    public static void finEscritura() {
        emEscritura.release();

        emControl.acquire();
        escritoresActivos--;
        escritoresBD--;
        if (escritoresBD == 0) {
            while (lectoresActivos > lectoresBD) {
                lectoresBD++;
                esperaFinEscritores.release();
            }
        }
        emControl.release();
    }

    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) {
```

```
        emControl = new SimpleSemaphore(1);
        emEscritura = new SimpleSemaphore(1);
        esperaFinEscritores = new SimpleSemaphore(0);
        esperaFinLectores = new SimpleSemaphore(0);

        createThreads(5, "lector");
        createThreads(3, "escritor");
        startThreadsAndWait();
    }
}
```