TRABAJO PSP: BARBERO DORMILÓN.

Hay varios métodos: semáforos, paso por mensaje, **monitores…**

El interbloqueo es un problema que surge en la programación paralela (también llamada concurrente) cuando dos o más procesos o hilos se quedan bloqueados por algún motivo.

El problema del barbero es un problema clásico de sincronización en ocurrencia que requiere exclusión mutua para algunos de los recursos y una condición de espera controlada: un cliente quiere recibir un servicio pero la silla del barbero está ocupada por el barbero, que está durmiendo, así que no puede recibir el servicio a no ser que lo despierte, y no pueden llegar más clientes si las sillas de espera están ocupadas hasta que no se vayan levantando para recibir su servicio.

Este problema refleja la necesidad de sincronización entre productor y consumidor, necesitando coordinar un recurso limitado, que sería la silla del barbero, con una cola de espera cuyo tamaño es finito. Se debe tener en cuenta que hay que evitar condiciones de carrera, como que dos clientes no ocupen la misma silla a la vez.

Habría que programar paralelamente para establecer normas fuertes y que consideren todos esos casos.

https://sarreplec.caib.es/pluginfile.php/10409/mod_resource/content/1/PSP02_Contenidos_Web/75_el_problema_del_interbloqueo_deadlock.html#:~:text=%2D%20El%20problema%20del%20interbloqueo%20(deadlock).,-Ocultar&text=El%20interbloqueo%20o%20bloqueo%20mutuo,se%20bloquean%20o%20esperan%20indefinidamente.&text=Porque%20cada%20hilo%20espera%20a,para%20acceder%20a%20un%20recurso.

::

La programación concurrente mejora la disponibilidad y eficiencia, modela tareas y aisla actividades en hilos de ejecución (aumentando la protección), aunque es compleja y utiliza un mayor número de recursos. El código que ejecuta un thread se define en clases que implementan la interfaz Runnable (public interface Runnable {public void run();}. Tiene métodos como:

- Run() para la actividad del thread.
- Start() para activar el run() y vuelve al llamante.
- Join() que espera a la terminación.
- Interrupt() que sale de un wait, sleep o join.
- IsInterrupted()

Métodos estáticos: sleep(milisegundos), currentThread().

Métodos de la clase Object que controlan la suspensión de threads: wait(), wait(milisegundos), notify(), notifyAll().

```
import java.lang.Math ;

class EjemploThread extends Thread {
        int numero;
        EjemploThread (int n) { numero = n; }

        public void run(){
                try{while(true){
                        System.out.println(numero);
                        sleep((long)(1000*Math.random()));
                    }
                }catch(InterruptedException e){return;} //acaba este thread
        }
        public static void main (String args[]){
                for (int i=0;i<10;i++)
                        new EjemploThread(i).start();
        }
}
```

```
SALIDA
1
4
5
6
2
9
8
4
3
0
7
3
7
2
8
3
1
9
...
```

Un monitor es un objeto que implementa el acceso en exclusión mutua a sus métodos. En Java se aplica a los métodos de la clase declarados como synchronized. Los demás métodos pueden accederse con concurrencia independientemente de si algún thread accede a ellos o a un método synchronized.

```
// ...dentro de código synchronized, ya que el hilo debe ser el propietario del monitor
// del objeto. Liberará el monitor hasta que lo despierten con notify o notifyAll, y
// pueda retomar control del monitor
try {
wait(); // o wait(0);
} catch (InterruptedException e) {
System.out.println(“Interrumpido durante el wait");
}
```

Disciplinas de señalización de monitores:

Signal and exit: después de hacer notify el thread debe salir del monitor y el thread que recibe la señal es el siguiente en entrar en el monitor.

Signal and continue: el thread que recibe la señal no tiene que ser necesariamente el siguiente en entrar en el monitor. *Puede haber intromisión: antes que el thread despertado podría entrar un thread que estuviera bloqueado en la llamada de un método synchronized.

En java, los monitores siguen la disciplina signal and continue. Tampoco se garantiza que el thread que más tiempo lleve esperando sea el que reciba la señal con un notify(). Cuando se usa notifyAll() para pasar todos los threads en wait a la cola de listos para ejecutar, el primer thread en entrar en el monitor no será necesariamente el que más tiempo lleve esperando.

Threads y locks en la JVM:

Cada thread tiene su propia memoria de trabajo. El thread trabaja con copia de las variables en la memoria de trabajo. La memoria principal tiene una copia maestra de cada

variable. La memoria principal puede contener también cerrojos (locks). Todo objeto Java tiene asociado un lock. Los threads pueden competir para adquirir un lock.

Operaciones atómicas en la JVM:

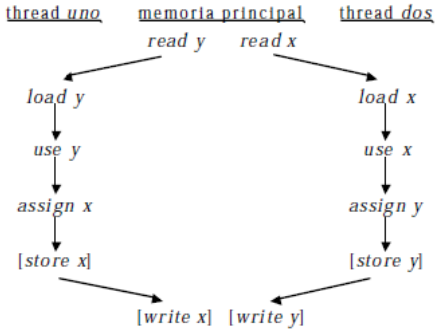
- Ejecutables por threads: use, assign, load, store.
- Ejecutables por la memoria principal: read, write.
- Ejecutables en sincronización por un thread y la memoria: lock (para adquirir un claim en un cerrojo particular), unlock (para liberar un derecho sobre un cerrojo particular).

Reglas sobre cerrojos:

- Un cerrojo solo puede pertenecer a un thread en un momento dado.
- Un cerrojo solo quedará libre cuando el thread haga tantos unlock como lock hubiera hecho.
- Un thread no puede hacer unlock de un cerrojo que no poseyera.

Threads y Locks en la MVJ

```
public class Ejemplo {

    int x = 1, y = 2;

    void uno() {
        x = y;
    }

    void dos() {
        y = x;
    }
}
```



En este ejemplo en memoria principal puede acabar ocurriendo:

- que x acabe valiendo lo que y
- o que y acabe valiendo lo que x
- o que se cambien los valores de x e y

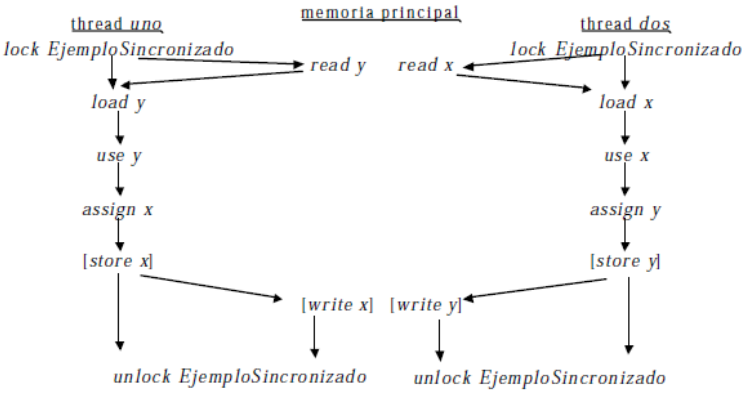
Threads y Locks en la MVJ

```
public class EjemploSincronizado
{
    int x = 1, y = 2;

    synchronized void uno() {
        x = y;
    }

    synchronized void dos() {
        y = x;
    }
}
```

Al usar la operación *unlock* se obliga a escribir en memoria principal, y por haber utilizado *lock* hay menos combinaciones, así que bien:

- o x acaba valiendo lo que y
- o y acaba valiendo lo que x
- y *no* puede haber intercambio de valores

Threads y Locks en la MVJ



Acciones dependientes del estado:

```
public class Semaforo {
        private int cuenta;
        Semaforo(int cuentaInicial) { cuenta = cuentaInicial; }
        public synchronized void P() {
                while ( cuenta <= 0 )
                        try { wait(); }
                        catch(InterruptedException e) {}
                cuenta--;
        }
        public synchronized void V() {
                cuenta++;
                notify();
        }
```

}

(https://grasia.fdi.ucm.es/jpavon/docencia/dso/programacionconcurrentejava.pdf)

::

En el caso de la exclusión mutua es algo más complicado. Para entender este concepto, también llamado mutex (del inglés *mutual exclusion*), primero se debe conocer que existe una parte del código al que llamaremos sección crítica, que es cuando se modifica algo. El término mutex también hace referencia al objeto Mutex, que es una forma de resolver este tipo de problemas de exclusión mutua, algo más complejo.

La exclusión mutua tiene que garantizar que el programa paralelo siempre acceda al dato más actual y no a uno que ya pasó. Por ejemplo, un caso en el que un objeto lee mientras otro modifica. Por tanto, modificar es un proceso delicado, y más si varios elementos se modifican a la vez. La manera más común para solventar este tipo de problemas en programación es establecer colas, por ejemplo, mediante el uso de semáforos. La diferencia entre el mutex y los semáforos es que los mutex están pensados concretamente para la exclusión mutua y los semáforos cuentan recursos y se pueden implementar como un mutex, pero tienen más usos.

::

Fuente de esto: Un hilo es una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un sistema operativo. Los distintos hilos de ejecución comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, la situación de autenticación… Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente. Un hilo es simplemente una tarea que puede ser ejecutada al mismo tiempo que otra tarea. Al igual que los procesos, los hilos poseen un estado de ejecución y pueden sincronizarse entre ellos para evitar problemas de compartición de recursos. Generalmente, cada hilo tiene una tarea específica y determinada, como forma de aumentar la eficiencia del uso del procesador. Los hilos tienen varios estados: ejecución, listo y bloqueado. No tiene sentido asociar estados de suspensión a hilos ya que es un concepto de proceso. Los cambios de estado pueden ser creación (cuando se crea un proceso se crea un hilo para ese proceso, y dentro de ese hilo se pueden crear otros hilos en el mismo proceso), bloqueo (cuando un hilo necesita esperar por un suceso y se bloquea para que el procesador ejecute otro hilo que esté al principio de los Listos mientras que el anterior permanece bloqueado), desbloqueo (cuando el suceso por el que el hilo se bloqueó se produce, el mismo pasa a la final de los Listos) y terminación (cuando finaliza el hilo se liberan sus contextos y columnas).

La creación y arranque de hilos se puede hacer de varias formas:

Una forma es creando un hilo implementando la interfaz Runnable para que sea utilizada por una instancia de Thread.

- La interfaz Runnable proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando dicha interfaz.
- En la utilización de la interfaz Runable, el método run() implementa la operación create conteniendo el código a ejecutar por el hilo.
- Dicho método contendrá el hilo de ejecución. Podemos verlo como el método main() en el hilo.
- El hilo finaliza cuando finaliza el método run().

Creación de un hilo implementando la interfaz Runnable para que sea utilizada por una instancia de Thread.

- Creamos una clase que implemente Runnable.
- Codificamos el método run() en dicha clase.

Opción 1:

- Creamos una instancia de la clase Thread() enviándole al constructor la instancia de la clase que implementa Runnable.
- Llamamos al método start() de la instancia de Thread().

Opción 2:

- En el constructor de la clase que implementa Runnable creamos una instancia de la clase Thread() enviándole una referencia de la propia clase (this).
- Invocamos al método start() de la instancia de Thread() creada.

Creación de un hilo extendiendo la clase Thread.

- Extendiendo de la clase Thread mediante la creación de una subclase.
- La clase Thread es responsable de producir hilos funcionales para otras clases e implementa la interfaz Runnable.

Creación de un hilo usando una clase anónima.

- Extendiendo de la clase Thread enviándole al constructor una clase que implementa la interfaz Runnable.
- A diferencia del caso anterior, la clase Runnable se declara de forma anónima.

Los métodos sincronizados: La palabra reservada synchronized se usa para indicar que ciertas partes del código están sincronizadas, es decir, que solo un subproceso puede acceder a dicho método a la vez. Cada método sincronizado posee una especie de llave que puede cerrar o abrir la puerta de acceso. Cuando un subproceso intenta acceder al método sincronizado mirará a ver si la llave está echada, en cuyo caso no podrá accederlo. Si el método no tiene la llave puesta, entonces el subproceso puede acceder a dicho código sincronizado.

::

Semáforos en java: Un semáforo es una variable especial (o tipo abstracto de datos) que constituye el método clásico para restringir o permitir el acceso a recursos compartidos (por ejemplo, un recurso de almacenamiento del sistema o variables del código fuente) en un entorno de multiprocesamiento (en el que se ejecutarán varios procesos concurrentemente).

java.util.concurrent.Semaphore. Ofrece métodos como acquire() o release() que son como las operaciones wait y signal.

La verificación y modificación del valor, así como la posibilidad de irse a dormir (bloquearse) se realiza en conjunto, como una sola e indivisible acción atómica. El sistema operativo garantiza que al iniciar una operación con un semáforo, ningún otro proceso puede tener acceso al semáforo hasta que la operación termine o se bloquee. Esta atomicidad es absolutamente esencial para resolver los problemas de sincronización y evitar condiciones de competencia.

Si hay n recursos, se inicializará el semáforo al número n. Así, cada proceso, al ir solicitando un recurso, verificará que el valor del semáforo sea mayor de 0; si es así es que existen recursos libres, seguidamente acaparará el recurso y decrementará el valor del semáforo.

Cuando el semáforo alcance el valor 0, significará que todos los recursos están siendo utilizados, y los procesos que quieran solicitar un recurso deberán esperar a que el semáforo sea positivo.

Los semáforos se emplean para permitir el acceso a diferentes partes de programas (llamados secciones críticas) donde se manipulan variables o recursos que deben ser accedidos de forma especial. Según el valor con que son inicializados se permiten a más o menos procesos utilizar el recurso de forma simultánea.

Un tipo simple de semáforo es el binario, que puede tomar solamente los valores 0 y 1. Se inicializan en 1 y son usados cuando sólo un proceso puede acceder a un recurso a la vez. Se les suele llamar mutex.

Los semáforos pueden ser usados para diferentes propósitos, entre ellos:

- Implementar cierres de exclusión mutua o locks.
- Barreras.
- Permitir a un máximo de N threads (hilos) acceder a un recurso, inicializando el semáforo en N.
- Notificación. Inicializando el semáforo en 0 puede usarse para comunicación entre threads sobre la disponibilidad de un recurso.

.....