1.- Sincronización y comunicación de hilos.
1.1.- La clase Semaphore.
1.2.- Información compartida entre hilos.
1.3.- Monitores. Métodos synchronized.
1.4.- Monitores. Segmentos de código synchronized.
1.5.- Comunicación entre hilos con métodos de java.lang.Object.
1.6.- El problema del interbloqueo (deadlock).

1.- Sincronización y comunicación de hilos.

Hay ocasiones en las que distintos hilos de un programa necesitan establecer alguna relación entre sí y compartir recursos o información. Se pueden presentar las siguientes situaciones:

Dos o más hilos compiten por obtener un mismo recurso, por ejemplo dos hilos que quieren escribir en un mismo fichero o acceder a la misma variable para modificarla.

Dos o más hilos colaboran para obtener un fin común y para ello, necesitan comunicarse a través de algún recurso. Por ejemplo un hilo produce información que utilizará otro hilo.

En cualquiera de estas situaciones, es necesario que los hilos se ejecuten de manera controlada y coordinada, para evitar posibles interferencias que pueden desembocar en programas que se bloquean con facilidad y que intercambian datos de manera equivocada.

¿Cómo conseguimos que los hilos se ejecuten de manera coordinada? Utilizando sincronización y comunicación de hilos:

Sincronización. Es la capacidad de informar de la situación de un hilo a otro. El objetivo es establecer la secuencialidad correcta del programa.

Comunicación. Es la capacidad de transmitir información desde un hilo a otro. El objetivo es el intercambio de información entre hilos para operar de forma coordinada.

En Java la sincronización y comunicación de hilos se consigue mediante:

Monitores. Se crean al marcar bloques de código con la palabra synchronized.

Semáforos. Podemos implementar nuestros propios semáforos, o bien utilizar la clase Semaphore incluida en el paquete java.util.concurrent.

Notificaciones. Permiten comunicar hilos mediante los métodos wait(), notify() y notifyAll() de la clase java.lang.Object.

Por otra parte, Java proporciona en el paquete java.util.concurrent varias clases de sincronización que permiten la sincronización y comunicación entre diferentes hilos de una aplicación multithreadring, como son: Semaphore, CountDownLatch, CyclicBarrier y Exchanger.

1.1.- La clase Semaphore.

La clase Semaphore del paquete java.util.concurrent, permite definir un semáforo para controlar el acceso a un recurso compartido.

Para crear y usar un objeto Semaphore haremos lo siguiente:

Indicar al constructor Semaphore (int permisos) el total de permisos que se pueden dar para acceder al mismo tiempo al recurso compartido. Este valor coincide con el número de hilos que pueden acceder a la vez al recurso.

Indicar al semáforo mediante el método acquire(), que queremos acceder al recurso, o bien mediante acquire(int permisosAdquirir) cuántos permisos se quieren consumir al mismo tiempo.

Indicar al semáforo mediante el método release(), que libere el permiso, o bien mediante release(int permisosLiberar), cuantos permisos se quieren liberar al mismo tiempo.

Hay otro constructor Semaphore (int permisos, boolean justo) que mediante el parámetro justo permite garantizar que el primer hilo en invocar adquire () será el primero en adquirir un permiso cuando sea liberado. Esto es, garantiza el orden de adquisición de permisos, según el orden en que se solicitan.

¿Desde dónde se deben invocar estos métodos? Esto dependerá del uso de Semaphore.

Si se usa para proteger secciones críticas, la llamada a los métodos acquire() y release() se hará desde el recurso compartido o sección crítica, y el número de permisos pasado al constructor será 1.

Si se usa para comunicar hilos, en este caso un hilo invocará al método acquire() y otro hilo invocará al método release() para así trabajar de manera coordinada. El número de permisos pasado al constructor coincidirá con el número máximo de hilos bloqueados en la cola o lista de espera para adquirir un permiso.

1.2.- Información compartida entre hilos.

Las secciones críticas son aquellas secciones de código que no pueden ejecutarse concurrentemente, pues en ellas se encuentran los recursos o información que comparten diferentes hilos, y que por tanto pueden ser problemáticas.

Un ejemplo sencillo que ilustra lo que puede ocurrir cuando varios hilos actualizan una misma variable es el clásico "ejemplo de los jardines". En él, se pone de manifiesto el problema conocido como la "condición de carrera", que se produce cuando varios hilos acceden a la vez a un mismo recurso, por ejemplo a una variable, cambiando su valor y obteniendo de esta forma un valor no esperado de la misma. En el siguiente enlace te facilitamos este ejemplo detallado.

En el ejemplo del "problema de los jardines", el recurso que comparten diferentes hilos es la variable contador cuenta. Las secciones de código donde se opera sobre esa variable son dos secciones críticas, los métodos incrementaCuenta() y decrementaCuenta().

La forma de proteger las secciones críticas es mediante sincronización. La sincronización se consigue mediante:

Exclusión mutua. Asegurar que un hilo tiene acceso a la sección crítica de forma exclusiva y por un tiempo finito.

Por condición. Asegurar que un hilo no progrese hasta que se cumpla una determinada condición.

En Java, la sincronización para el acceso a recursos compartidos se basa en el concepto de monitor.

1.3.- Monitores. Métodos synchronized.

En Java, un monitor es una porción de código protegida por un mutex o lock. Para crear un monitor en Java, hay que marcar un bloque de código con la palabra **synchronized**, pudiendo ser ese bloque:

Un método completo.

Cualquier segmento de código.

Añadir synchronized a un método significará que:

Hemos creado un monitor asociado al objeto.

Sólo un hilo puede ejecutar el método synchronized de ese objeto a la vez.

Los hilos que necesitan acceder a ese método synchronized permanecerán bloqueados y en espera.

Cuando el hilo finaliza la ejecución del método synchronized, los hilos en espera de poder ejecutarlo se desbloquearán. El planificador Java seleccionará a uno de ellos.

Y ¿qué bloques interesa marcar como synchronized? Precisamente los que se correspondan con secciones críticas y contengan el código o datos que comparten los hilos.

En el ejemplo anterior, "Problema de los jardines" se debería sincronizar tanto el método incrementaCuenta(), como el decrementaCuenta() tal y como ves en el siguiente código, ya que estos métodos contienen la variable cuenta, la cual es modificada por diferentes hilos. Así mientras un hilo ejecuta el método incrementaCuenta() del objeto jardin, jardin.incrementaCuenta(), ningún otro hilo podrá ejecutarlo.

1.4.- Monitores. Segmentos de código synchronized.

Hay casos en los que no se puede, o no interesa sincronizar un método. Por ejemplo, no podremos sincronizar un método que no hemos creado nosotros y que por tanto no podemos acceder a su código fuente para añadir synchronized en su definición. La forma de resolver esta situación es poner las llamadas a los métodos que se quieren sincronizar dentro de segmentos sincronizados de la siguiente forma: synchronized (objeto){ // sentencias segmento; }

En este caso el funcionamiento es el siguiente:

El objeto que se pasa al segmento, es el objeto donde está el método que se quiere sincronizar.

Dentro del segmento se hará la llamada al método que se quiere sincronizar.

El hilo que entra en el segmento declarado synchronized se hará con el monitor del objeto, si está libre, o se bloqueará en espera de que quede libre. El monitor se libera al salir el hilo del segmento de código synchronized.

Sólo un hilo puede ejecutar el segmento synchronized a la vez.

En el ejemplo del problema de los jardines, aplicando este procedimiento, habría que sincronizar el objeto que denominaremos jardin y que será desde donde se invoca al método incrementaCuenta(), método que manipula la variable cuenta que modifican diferentes hilos:

Observa que:

Ahora el método incrementaCuenta() no será synchronized (se haría igual para decrementaCuenta()),

Se está consiguiendo un acceso con exclusión mutua sobre el objeto jardin, aún cuando su clase no contiene ningún segmento ni método synchronized.

1.5.- Comunicación entre hilos con métodos de java.lang.Object.

La comunicación entre hilos la podemos ver como un mecanismo de auto-sincronización, que consiste en logar que un hilo actúe solo cuando otro ha concluido cierta actividad (y viceversa). Java soporta comunicación entre hilos mediante los siguientes métodos de la clase java.lang.Object.
wait(). Detiene el hilo (pasa a "no ejecutable"), el cual no se reanudará hasta que otro hilo notifique que ha ocurrido lo esperado.

wait(long tiempo). Como el caso anterior, solo que ahora el hilo también puede reanudarse (pasar a "ejecutable🗣) si ha concluido el tiempo pasado como parámetro.
notify(). Notifica a uno de los hilos puestos en espera para el mismo objeto, que ya puede continuar.

notifyAll(). Notifica a todos los hilos puestos en espera para el mismo objeto que ya pueden continuar.

La llamada a estos métodos se realiza dentro de bloques synchronized.

¿Cómo funcionan realmente estos métodos? En el siguiente documento tienes la explicación de cómo funcionan estos métodos y un ejemplo de su uso.

Explicación del funcionamiento de los métodos de comunicación entre hilos. (0.15 MB)

Dos problemas clásicos que permiten ilustrar la necesidad de sincronizar y comunicar hilos son:

El problema del Productor-Consumidor. Del que has visto un ejemplo anteriormente y que permite modelar situaciones en las que se divide el trabajo entre los hilos. Modela el
acceso simultáneo de varios hilos a una estructura de datos u otro recurso, de manera que unos hilos producen y almacenan los datos en el recurso y otros hilos (consumidores)
se encargan de eliminar y procesar esos datos.

El problema de los Lectores-Escritores. Permite modelar el acceso simultáneo de varios hilos a una base de datos, fichero u otro recurso, unos queriendo leer y otros escribir o modificar los datos.

1.6.- El problema del interbloqueo (deadlock).

El interbloqueo o bloqueo mutuo (deadlock) consiste en que uno a más hilos, se bloquean o esperan indefinidamente.

¿Cómo se llega a una situación de interbloqueo? A dicha situación se llega

Porque cada hilo espera a que le llegue un aviso de otro hilo que nunca le llega.

Porque todos los hilos, de forma circular, esperan para acceder a un recurso.

El problema del bloqueo mutuo, en las aplicaciones concurrentes, se podrá dar fundamentalmente cuando un hilo entra en un bloque synchronized, y a su vez llama a otro bloque synchronized, o bien al utilizar clases de java.util.concurrent que llevan implícita la exclusión mutua.

En el siguiente enlace puedes consultar un artículo y ejemplo detallado sobre el interbloqueo.

El problema de los interbloqueos.

En el siguiente enlace encontrarás un ejemplo que produce interbloqueo. Observa que no finaliza la ejecución del programa y que en la barra de estado del IDE NetBeans (a la derecha) aparece la indicación de programa bloqueado. Habrá que finalizar manualmente el programa.

Otro problema, menos frecuente, es la inanición (starvation), que consiste en que un hilo es desestimado para su ejecución. Se produce cuando un hilo no puede tener acceso regular a los recursos compartidos y no puede avanzar, quedando bloqueado. Esto puede ocurrir porque el hilo nunca es seleccionado para su procesamiento o bien porque otros hilos que compiten por el mismo recurso se lo impiden.

Está claro que los programas que desarrollemos deben estar exentos de estos problemas, por lo que habrá que ser cuidadosos en su diseño.