1. Introducción

Como ya conoces los semáforos son primitivas básicas de sincronización basadas en memoria compartida que nos permiten controlar la **exclusión mutua** y la **sincronización** entre procesos.

Java no incluía semáforos, sin embargo con las primitivas de sincronización de Java podemos implementar el comportamiento de los semáforos. Esto supone que cualquier problema que pueda ser resuelto con semáforos también podrá ser resuelto con las primitivas propias de Java. Es posible encontrar librerías que ofrecen diferentes implementaciones para hacer uso de semáforos en Java.

A partir de la versión 1.5 (JSE5.0-2004) Java incluyó la clase java.util.concurrent.Semaphore que nos permite crear semáforos generales, puedes consultar el API de java 5 (o versiones posteriores) para ver los constructores y métodos disponibles.

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Semaphore.html

En Java los equivalentes para las operaciones ***wait()*** y ***singal()*** son ***acquire()*** y ***release()*** respectivamente.

2- Ejercicios

Puedes colaborar y discutir las soluciones con tus compañeros, pero asegúrate de que la solución que plantees la has realizado tú (el grupo, en su caso) y entiendes cómo y porqué. El no contestar adecuadamente a las preguntas del profesor sobre la solución presentada supondrá la no superación de la práctica. Presentar una práctica copiada supone suspender la asignatura.

1. Panel Informativo

En este ejemplo consideraremos un panel informativo como los que solemos encontrar en los aeropuertos con información sobre las próximas llegadas de vuelos

Veremos la programación de este panel en Java. Esto nos permitirá estudiar situaciones simples donde hilos concurrentes interfieren en el uso de una estructura compartida.

En primer lugar descarga el fichero PanelJava.zip y descomprímelo.

Puesto que no tenemos acceso a un panel real de un aeropuerto, en su lugar tenemos aquí una completa implementación, donde el panel es mostrado usando la biblioteca Swing de Java

http://download.oracle.com/javase/6/docs/technotes/guides/swing/

http://en.wikipedia.org/wiki/Swing_%28Java%29

Debemos pensar en el panel como una simulación de un dispositivo hardware de la siguiente forma:

Al panel accederemos mediante la siguiente API sencilla que se encuentra en el archivo **PanelHw.java** :

```
public interface PanelHw {
    public int getRows();
    public int getCols();
    public void write(int row, int col, char c);
}
```

Los dos primeros métodos devuelven el número de filas y columnas del panel (Podría haber paneles de diferentes tamaños). El tercero se utiliza para escribir un carácter en la fila y columna especificada. Consideramos que la primera fila/columna es la 0.

El fichero IPanel.java contiene una implementación de esta API usando Swing. No necesitas conocer el código fuente de este fichero para el ejercicio, éste contiene gran cantidad de detalles de bajo nivel específicos de las librerías GUI de Java que no es necesario conocer. Para ver su utilización comprueba el funcionamiento de Main1.java, un sencillo programa principal que escribe una A y una B en diferentes posiciones del panel, a continuación espera durante tres segundos y, finalmente, elimina la A.

Puesto que la interfaz PanelHw es muy de bajo nivel para utilizar realmente la pantalla, el fabricante nos ofrece también **PanelNivel2.java**, que es mucho más cómoda de utilizar.

```
public interface PanelNivel2 {

    public void clear();
    public void addRow(String str);
    public void deleteRow(int row);

}
```

El método **clear()** borra todo el texto del panel, **ponLinea(str)** añade el string `str` como la última nueva línea después de la última línea existente. Por último **borraLinea(r)** borra la fila r, cambiando todas las filas de abajo a arriba una posición. Además, la intención es que se puedan agregar filas también cuando la pantalla está llena, esta fila no será visible en pantalla hasta que hayan sido eliminadas suficientes filas por encima. Finalmente el último cambio realizado en el panel destacará parpadeando una par de veces.

El fabricante del panel también nos ha proporcionado una implementación de esta interfaz en el archivo **IPanel2.java**, estudia el funcionamiento de esta clase. Puedes ejecutar una demostración sencilla que encontrarás en **Main2** para ver su funcionamiento.

No pretendemos crear un programa completo para la gestión de un aeropuerto usando esta API, pero simplemente ten en cuenta que es posible que este panel pueda ser accedido por un programa multi-hilo en el que varios empleados del aeropuerto actualizan al mismo tiempo el panel. Para probar **IPanel** en esas condiciones, tu tarea consiste en escribir un simple programa multi-hilo.:

Un esqueleto de éste se encuentra en el archivo **Main3.java**, que muestra la estructura de un sencillo programa principal que crea un panel **p** e inicia tres hilos, dos simulan ser operadores que introducen vuelos en el panel, el otro ejecuta **quitaLinea(p).** Debes completar el cuerpo de estos tres procedimientos. Completa **opeardor1 y operador2** con una secuencia de llamadas a **ponLinea(str)** intercaladas con un retardo entre 2 y 5 segundos. De forma similar completa **quitaVuelos(p)** con llamadas a **borraLinea(0) con un intervalo de 4 segundos.**

Tras realizar esto si ejecutamos el programa podremos comprobar que el funcionamiento puede ser "incorrecto", de hecho la clase **IPanel2** no es "*theard safe*", es decir no garantiza un comportamiento correcto cuando sus métodos son accedidos por hilos concurrentes.

DEBES SOLUCIONAR EL PROBLEMA DE DOS FORMAS DISTINTAS:

- A. Desarrollamos la aplicación Main3 pero suponemos que no tenemos acceso a **`IPanel2`**, (sólo a través de su interfaz), así que tenemos que solucionar el problema sin modificar **`IPanel2`**.

  Una forma de hacer esto es identificar las secciones críticas en Main3 y protegerlas utilizando semáforos. Hazlo utilizando instancias de la clase:

  `java.util.concurrent.Semaphore.`

- B. En este segundo caso deseamos modificar **`IPanel2`** para que sea "theard safe" utilizando el modificador *syncronized*.

> El hacer un método ***synchronized*** tiene dos efectos:
>
> - Primero : es imposible que se intercalen dos invocaciones a métodos sincronizados del mismo objeto. (Exclusión mutua)
>
>   Cuando un hilo está ejecutando un método sincronizado de un objeto. Todos los demás hilos que invoquen a métodos sincronizados de ese objeto se bloquean(suspenden su ejecución) hasta que el primer hilo deja el objeto
> - Segundo: Cuando un método sincronizado termina se garantiza que los cambios en el estado del objeto son visibles para todos los Hilos que posteriormente ejecuten un método sincronizado del mismo objeto.

2. Atención de urgencias

Queremos simular nuestro particular servicio de urgencias que tiene las siguientes características:

- Tenemos una sala de espera con 5 sillas, los enfermos llegan y se sientan hasta que pasan a los boxes, donde serán atendidos por uno de los médicos.
- Hay tres boxes, pero sólo tenemos dos médicos, (que evidentemente atenderán a los enfermos de uno en uno) una vez que el médico atiende al enfermo éste se va.
- Las normas de seguridad contra incendios no permiten que haya personas de pie en la sala de espera, por lo tanto si un enfermo llega y no tiene sitio se debe ir a otro centro puesto que no se le permitirá el paso.

Para simular el problema tendremos dos clases de hilos **`enfermos y médicos`**, el método ***main*** construirá e iniciará dos hilos que serán los médicos y 20 enfermos.

Los enfermos realizan estas acciones:

- Entran.
- Pasan al box.
- Son atendidos.
- Se van.

Por su parte los médicos

- Atienden paciente
- Despiden paciente. (le abren la puerta para que se vaya).

Para evitar incidentes los enfermos deben ser atendidos por orden de llegada.

Estudia el problema viendo las exclusiones mutuas y las citas (sincronizaciones) que son necesarias para que la simulación funcione según lo expuesto.

El programa hará visible las acciones que realizan en cada momento los actores (hilos) identificando a los mismos. Utiliza retardos para simular

2B. Atención de urgencias ampliado

En el ejercicio anterior hemos obviado el proceso de clasificación de los enfermos. Modifica el programa realizado para crear otra versión en la que además de los hilos anteriores tendremos un hilo que ejercerá el rol de clasificador que será el encargado de clasificar a los enfermos antes de que éstos pasen a los boxes.

Clasificador:

- Atiende paciente
- Clasifica Paciente. (Para que pueda pasar al box)

Atención: Si los boxes están llenos no se aceptarán más enfermos en clasificación hasta que no haya boxes libres.