Programación multiproceso III

Programación de servicios y procesos

Contenidos:

1) Gestión de procesos. Conceptos básicos: Creación, ejecución y finalización de procesos.
2) Sincronización entre procesos. Exclusión mutua. Condiciones de sincronización.
3) **Recursos compartidos.**
4) **Mecanismos de comunicación y sincronización de procesos (semáforos, monitores, paso de mensajes.)**
5) **Problemas. Inanición, interbloqueos.**

Recursos Compartidos

Las situaciones en las que dos o más procesos tengan que comunicarse, cooperar o utilizar un mismo recurso; implica que deba haber cierto sincronismo entre ellos. O bien unos tienen que esperar que otros finalicen alguna acción o tienen que realizar alguna tarea al mismo tiempo.

El **sincronismo** entre procesos lo hace posible el **SO**, y lo que hacen los l**enguajes de programación** de alto nivel es encapsular los mecanismos de sincronismo que proporciona cada SO en objetos, métodos y funciones. Los lenguajes de programación, proporcionan primitivas de sincronismo entre los distintos hilos que tenga un proceso; estas primitivas del lenguaje, las veremos en la siguiente unidad.

En **programación concurrente**, siempre que accedamos a algún **recurso compartido** (eso incluye a los **ficheros**), deberemos tener en cuenta las condiciones en las que nuestro proceso debe hacer uso de ese recurso: ¿será de forma exclusiva o no?

Recursos Compartidos

En el caso de **lecturas y escrituras en un fichero**, debemos determinar si queremos acceder al fichero como sólo lectura; escritura; o lectura-escritura; y utilizar los objetos que nos permitan establecer los mecanismos de sincronización necesarios para que un proceso pueda bloquear el uso del fichero por otros procesos cuando él lo esté utilizando.

Esto se conoce como el problema de los procesos lectores-escritores. El sistema operativo, nos ayudará a resolver los problemas que se plantean; ya que:

- Si el acceso es de **sólo lectura**: Permitirá que todos los procesos lectores, que sólo quieren leer información del fichero, puedan acceder simultáneamente a él.
- En el caso de **escritura, o lectura-escritura**: El SO nos permitirá pedir un tipo de acceso de forma exclusiva al fichero. Esto significa que el proceso deberá esperar a que otros procesos lectores terminen sus accesos. Y otros procesos (lectores o escritores), esperarán a que ese proceso escritor haya finalizado su escritura.

Debemos tener en cuenta que, nosotros, nos comunicamos con el SO a través de los objetos y métodos proporcionados por un lenguaje de programación; y, por lo tanto, tendremos que consultar cuidadosamente la documentación de las clases que estamos utilizando para conocer todas las peculiaridades de su comportamiento.

Semáforos

Un **semáforo**, es un componente de bajo nivel de abstracción que permite arbitrar los accesos a un recurso compartido en un entorno de programación concurrente.

Al utilizar un **semáforo**, lo veremos como un tipo dato, que podremos instanciar. Ese objeto semáforo podrá tomar un determinado conjunto de valores y se podrá realizar con él un conjunto determinado de operaciones. Un semáforo, tendrá también asociada una **lista de procesos que suspendidos** que se encuentran a la espera de entrar en el mismo.

Dependiendo del conjunto de datos que pueda tomar un semáforo, tendremos:

- **Semáforos binarios:** Aquellos que pueden tomar sólo valores 0 ó 1. Como nuestras luces verde y roja.
- **Semáforos generales**: Pueden tomar cualquier valor Natural (entero no negativo).

En cualquier caso, los valores que toma un semáforo representan:

- Valor igual a 0. Indica que el semáforo está cerrado.
- Valor mayor de 0. El semáforo está abierto.

Semáforos

Para utilizar semáforos, seguiremos los siguientes pasos:

1. Un proceso padre creará e inicializará tanto semáforo.
2. El proceso padre creará el resto de procesos hijo pasándoles el semáforo que ha creado. Esos procesos hijos acceden al mismo recurso compartido.
3. Cada proceso hijo, hará uso de las operaciones seguras wait y signal respetando este esquema:
    1. objSemaforo.**acquire**(); Para consultar si puede acceder a la sección crítica.
    2. Sección crítica; Instrucciones que acceden al recurso protegido por el semáforo objSemaforo.
    3. objSemaforo.**release**(); Indicar que abandona su sección y otro proceso podrá entrar.

El proceso padre habrá creado tantos semáforos como tipos secciones críticas distintas se puedan distinguir en el funcionamiento de los procesos hijos (puede ocurrir, uno por cada recurso compartido).

Semáforos

La ventaja de utilizar **semáforos** es que son fáciles de comprender, proporcionan una **gran capacidad funcional** (podemos utilizarlos para resolver cualquier problema de concurrencia). Pero, su nivel bajo de abstracción, los hace **peligrosos** de manejar y, a menudo, son la causa de muchos errores, como es el interbloqueo. Un simple olvido o cambio de orden conduce a bloqueos; y requieren que la gestión de un semáforo se distribuya por todo el código lo que hace la depuración de los errores en su gestión es muy difícil.

En java, encontramos la clase **Semaphore** dentro del paquete **java.util.concurrent;** y su uso real se aplica a los hilos de un mismo proceso, para arbitrar el acceso de esos hilos de forma concurrente a una misma región de la memoria del proceso. Por ello, veremos ejemplos de su uso en siguiente unidades de este módulo.

Monitores

Un **monitor**, es un componente de alto nivel de abstracción destinado a gestionar recursos que van a ser accedidos de forma concurrente.

Los monitores encierran en su interior los recursos o variables compartidas como componentes privadas y garantizan el acceso a ellas en exclusión mutua.

La declaración de un monitor incluye:

- Declaración de las constantes, variables, procedimientos y funciones que son privados del monitor (solo el monitor tiene visibilidad sobre ellos).
- Declaración de los procedimientos y funciones que el monitor expone (públicos) y que constituyen la interfaz a través de las que los procesos acceden al monitor.
- Cuerpo del monitor, constituido por un bloque de código que se ejecuta al ser instanciado o inicializado el monitor. Su finalidad es inicializar las variables y estructuras internas del monitor.
- El monitor garantiza el acceso al código interno en régimen de exclusión mutua.
- Tiene asociada una lista en la que se incluyen los procesos que al tratar de acceder al monitor son suspendidos.

Monitores

Las ventajas que proporciona el uso de monitores son:

- Uniformidad: El monitor provee una única capacidad, la exclusión mutua, no existe la confusión de los semáforos.
- Modularidad: El código que se ejecuta en exclusión mutua está separado, no mezclado con el resto del programa.
- Simplicidad: El programador o programadora no necesita preocuparse de las herramientas para la exclusión mutua .
- Eficiencia de la implementación: La implementación subyacente puede limitarse fácilmente a los semáforos.

Y, la desventaja:

- Interacción de múltiples condiciones de sincronización: Cuando el número de condiciones crece, y se hacen complicadas, la complejidad del código crece de manera extraordinaria.

Paso de mensajes.

El **paso de mensajes** es una técnica empleada en programación concurrente para aportar sincronización entre procesos y permitir la **exclusión mutua**, de manera similar a como se hace con los semáforos, monitores, etc. Su principal característica es que no precisa de memoria compartida.

Los elementos principales que intervienen en el paso de mensajes son el proceso que envía, el que recibe y el mensaje. Dependiendo de si el proceso que envía el mensaje espera a que el mensaje sea recibido, se puede hablar de paso de mensajes síncrono o asíncrono:

- En el paso de mensajes **asíncrono**, el proceso que envía, no espera a que el mensaje sea recibido, y continúa su ejecución, siendo posible que vuelva a generar un nuevo mensaje y a enviarlo antes de que se haya recibido el anterior. Por este motivo se suelen emplear buzones o colas, en los que se almacenan los mensajes a espera de que un proceso los reciba. Generalmente empleando este sistema, el proceso que envía mensajes sólo se bloquea o para, cuando finaliza su ejecución, o si el buzón está lleno.
- En el paso de mensajes **síncrono**, el proceso que envía el mensaje espera a que un proceso lo reciba para continuar su ejecución. Por esto se suele llamar a esta técnica encuentro, o rendezvous. Dentro del paso de mensajes síncrono se engloba a la llamada a procedimiento remoto (RPC), muy popular en las arquitecturas cliente/servidor.

Problemas. Inanición, interbloqueos.

El acceso a recursos compartidos no está libre de problemas:

- **Inanición**: un proceso/hilo espera indefinidamente para entrar a la sección crítica.
  - Posible escenario: Algunas primitivas de sincronización pueden estar implementadas con criterios de prioridad para seleccionar el siguiente proceso a entrar en la sección crítica.
- **Interbloqueo**: es una situación de reciprocidad en el acceso a recursos múltiples.
  - Un conjunto de procesos está interbloqueado cuando cada proceso está esperando por un recurso que está siendo usado por otro proceso del conjunto.
  - ¡Aunque las primitivas cumplan rigurosamente las propiedades del acceso exclusivo!

Cuando el programa usa recursos “bajo tu control” (stream de datos o acceso a ficheros de uso en varios programas que has programado tu, por ejemplo), los semáforos, monitores o el paso de mensajes deben solucionar estos problemas de inanición o interbloqueo.

Comunicación de procesos con Java

Por defecto el proceso que se crea no tiene su propia terminal o consola. Todas las operaciones E/S serán redirigidas al proceso padre, donde se puede acceder a ellas con los métodos **getInputStream(), getOutpuStream() y getErrorStream()**. El proceso utiliza estos flujos para alimentar la entrada y obtener la salida del proceso.

Cada constructor de **ProcessBuilder** gestiona los siguientes atributos del proceso:

- Un **comando**. En una lista de cadenas que representa el programa y los argumentos (si los hay).
- Un **entorno** (*environment*) con sus variables
- Un **directorio** de trabajo (por defecto el del proceso en curso)
- Una **fuente** o tubería estándar de **entrada**. Se accede a ella por el método **Process.getOutputStream()**. Esta fuente puede ser redirigida con el método **redirectInput()** (si lo hacemos "vaciamos" el stream y el método Process.getOutputStream() devolverá nulo).
- Una **fuente** o tubería estándar de **salida** y otra de **error**. Se accede a ellas por los métodos **Process.getInputStream()** y **Process.getErrorStream()**. Estas fuentes pueden ser redirigidas con los métodos **redirectOutput()** y **redirectError()** (si lo hacemos "vaciamos" el stream correspondiente).

Gestión de procesos en Java.

Tabla 1.3. Métodos de la clase java.lang.ProcessBuilder

Método	Descripción
start	Inicia un nuevo proceso usando los atributos especificados.
command	Permite obtener o asignar el programa y los argumentos de la instancia de ProcessBuilder.
directory	Permite obtener o asignar el directorio de trabajo del proceso.
environment	Proporciona información sobre el entorno de ejecución del proceso.
redirectError	Permite determinar el destino de la salida de errores.
redirectInput	Permite determinar el origen de la entrada estándar.
redirectOutput	Permite determinar el destino de la salida estándar.

https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html

Comunicación de procesos con Java

En un sistema real, probablemente necesitemos guardar los resultados de un proceso en un archivo de log o de errores para su posterior análisis. Afortunadamente lo podemos hacer sin modificar el código de nuestras aplicaciones usando los métodos que proporciona el API de ProcessBuilder para hacer exactamente eso.

Por defecto, tal y como ya hemos visto, los procesos hijos reciben la entrada a través de una tubería a la que podemos acceder usando el OutputStream que nos devuelve Process.getOutputStream().

Sin embargo, tal y como veremos a continuación, esa entrada estándar se puede cambiar y redirigirse a otros destinos como un fichero usando el método redirectOutput(File). Si modificamos la salida estándar, el método getOutputStream() devolverá ProcessBuilder.NullOutputStream.

Comunicación de procesos con Java

Redirección antes de ejecutar

Es importante fijarse en qué momento se realiza cada acción sobre un proceso.

Antes hemos visto que los flujos de E/S se consultan y gestionan una vez que el proceso está en ejecución, por lo tanto los méotodos que nos dan acceso a esos *streams* son métodos de la clase `Process` .

Si lo que queremos es redirigir la E/S, como vamos a ver a continuación, lo haremos mientras preparamos el proceso para ser ejecutado. De forma que cuando se lance sus streams de E/S se modifiquen. Por eso en esta ocasión los métodos que nos permiten redireccionar la E/S de los procesos son métodos de la clase `ProcessBuilder` .

Comunicación de procesos con Java. Ejemplos

Ejemplo de clase Java que llama a un proceso y lee el Stream de salida y lo redirige a un fichero

```
public static void main(String[] args) throws IOException {
    //creamos objeto File al directorio donde esta Ejemplo2
    File directorio = new File("C:\\Users\\jhorn\\eclipse-workspace\\UT2\\UT2_9_CalcularParImpar\\bin");

    //El proceso a ejecutar es Ejemplo2
    ProcessBuilder pb = new ProcessBuilder("java","mispaquetes.Calcular");
    //se establece el directorio donde se encuentra el ejecutable
    pb.directory(directorio);
    File fBat = new File("entrada.txt");
    File fOut = new File("salida.txt");
    File fErr = new File("error.txt");

    pb.redirectInput(fBat);
    pb.redirectOutput(fOut);
    pb.redirectError(fErr);

    //se ejecuta el proceso
    Process p = pb.start();
```

Los ficheros se crean en la carpeta en la que se ejecuta el proyecto. Si queremos definir una carpeta específica debemos hacerlo al definir el fichero

```
File fBat = new File("C:\\Julio","entrada.txt");
File fOut = new File("C:\\Julio","salida.txt");
File fErr = new File("C:\\Julio","error.txt");
```

Comunicación de procesos con Java. Ejercicio

Modifica la clase Java que corresponda para que cuando llamamos a *LeerNombre.java,* la salida la escriba en un fichero llamado “minombre.txt” y la entrada la recoja de un fichero llamado “nombre.txt”