PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Tipos de programación

Instalación del entorno de desarrollo Netbeans

(netbeans 8.2 ya no está en su página oficial)

Instalación de Netbeans en windows:

https://netbeans-ide.informer.com/download/#downloading

Instalación en Linux junto con el JDK

https://www.youtube.com/watch?v=kgBk8bjow5c

Antes de instalar NetBeans, deberemos tener instalada la JDK.

Programa, proceso, servicio

• **Programa**: Nos podemos referir a un programa como toda la información, tanto código como datos, almacenada en disco de una aplicación y que nos resolverá un problema concreto.

• **Proceso**: Un proceso se va a crear **cuando ejecutamos un programa**, por lo que podremos decir, de una forma muy simplificada, que un programa es, en cierto modo, un proceso. Así pues, podemos definir un proceso como un **programa en ejecución**. El concepto de proceso no se refiere solo al código y a los datos, sino que incluye todo lo necesario para que este se ejecute. Es muy importante tener en cuenta que un proceso es una entidad independiente de todo lo demás, aunque se ejecute en un mismo programa.

• **Servicios**: Los servicios son procesos que **no son interactivos**, pero que se están **ejecutando continuamente**. Son **controlados por el sistema**, sin ninguna intermediación por parte del usuario. Este tipo de procesos proporciona un servicio básico para el resto de procesos. El sistema operativo tiene sus propios servicios y los **arrancará automáticamente** en su inicialización. Los servicios van a pasar totalmente desapercibidos al usuario. Sin embargo, aunque se encuentren permanentemente en ejecución, se suele decir que se ejecutan en segundo plano (background). Cada **proceso** puede tener **uno o varios servicios**. Los Servicios también son conocidos como ***demonios***.

Concepto de aplicación

*Una **aplicación** es un **programa informático** que está diseñado para ser usado como una herramienta que permita la resolución de un problema concreto. Una vez que la aplicación está instalada, puede ser que la compongan diferentes ejecutables y bibliotecas, creando como mínimo un nuevo proceso cuando la lancemos.*

Hilos y ejecutable

• **Hilos**: Los hilos, hebras, o thread son una tarea que podemos **ejecutar en paralelo** a otras, es decir: todos los hilos se estarán ejecutando al mismo tiempo, por lo que la tarea que ejecutan se podrá realizar mucho más rápido. Todos y cada uno de los hilos **comparten una serie de recursos**, como pueden ser: el espacio de memoria, los archivos abiertos, los puertos para la comunicación en red o una base de datos.

Todos los **hilos que comparten los mismos recursos** forman parte de un **mismo proceso**, lo que implica que cualquier hilo puede acceder a estos y modificarlos, pero hay que tener en cuenta que esta no es una tarea banal, ya que pueden producirse infinidad de errores. Hay que tener muy presente que, si un hilo modifica, por ejemplo, un dato en la memoria, los otros hilos posteriores a él ya accederán a ese dato modificado.

Un proceso que esté formado por varios hilos va a seguir en ejecución mientras alguno de sus hilos siga activo. Hasta que todos los hilos finalizan, no finaliza el proceso, y todos los recursos que estuviesen siendo utilizados son liberados. Un **hilo** siempre se va a **ejecutar dentro del** contexto de un proceso, que se conoce como **proceso padre**. Los programas que se ejecutan mediante **varios hilos** se denominan de **flujo múltiple**, mientras que los de **flujo único** son los que tienen **un solo hilo**. La utilización de **hilos** nos va a permitir simplificar el diseño de aplicaciones que deben realizar varias **funciones de forma simultánea**.

• **Ejecutable**: *Los ejecutables son los **ficheros con los que podemos instalar las aplicaciones**.* Podemos definir un ejecutable como un fichero que contiene toda la **información** que será necesaria para **crear un proceso** partiendo de los datos almacenados **de un programa**, es decir, vamos a llamar ejecutable a todo aquel fichero que nos permita **poner el programa en ejecución** como un proceso. Hay **diferentes tipos de ejecutables**, lo cual es cierto. Dentro de este tipo de ficheros, podemos encontrar:

- los **binarios**, que son ficheros que contienen **código** que va a ser **ejecutado directamente por el procesador**;
- los **interpretados**, que son ficheros que se ejecutarán en una **máquina virtual**;
- las **bibliotecas**, que son un conjunto de funciones que pueden ser **utilizadas por otros programas**.

Programación concurrente

Tipo de programación que nos va a permitir tener en ejecución al mismo tiempo **varias tareas** que **pueden ser interactivas**. Nos va permitir realizar varias tareas al mismo tiempo como, por ejemplo, escuchar música, ver la pantalla del ordenador, imprimir documentos, etc. Proporciona **mecanismos de comunicación y sincronización** entre **procesos**.

Las tareas de la programación concurrente se pueden **ejecutar en**:

- **Multiprogramación** (ejecución en un **único procesador**): Cuando hablamos de multiprogramación, aunque para el usuario que está frente al ordenador pueda parecer que se están ejecutando varios procesos al mismo tiempo (música, imprimir, escribir...), como solamente existe un único procesador, solamente se va a poder estar ejecutando un único proceso en un momento determinado de tiempo. Para poder cambiar entre los diferentes procesos que se deben ejecutar, el sistema operativo se encargará de **cambiar el proceso** que está actualmente en ejecución **cada cierto período** de tiempo, en el orden de **milisegundos**. Todo esto permite que en un segundo se ejecuten múltiples procesos, creando la impresión en el usuario de que muchos programas se están ejecutando al mismo tiempo, cuando no es cierto. Este complejo proceso no mejora el tiempo de ejecución global de los programas, ya que todos estos se ejecutan intercambiándose unos por otros en el procesador, aunque sí que va a permitir que varios programas tengan la apariencia de que se ejecutan al mismo tiempo.

- **Programación paralela**: Este tipo de programación permite **mejorar el rendimiento** de un **programa** si este se ejecuta de **forma paralela en diferentes núcleos**, ya que permite que se ejecuten varias **instrucciones a la vez**.

- **Multitarea** (**varios núcleos** en un **mismo procesador**): Cuando hablamos de multitarea, nos referimos a la existencia de varios núcleos en un procesador, apareciendo con esto los Dual **Cores**, Quad Cores, etc. Cada uno de estos núcleos podría estar ejecutando una instrucción diferente al mismo tiempo.

Ventajas / Inconvenientes de la programación concurrente	
Ventajas	**Inconvenientes**
Hacer un mejor uso de los procesadores de nuestro equipo	Se aumenta el número de veces que se produce un cambio de contexto
Velocidad de respuesta mayor	Pueden producirse muchos problemas de sincronización
Posibilidad de compartir recursos	Si los recursos son limitados afectará al rendimiento general
Uso más eficiente de memoria	Mayor complejidad a la hora de desarrollar un código
Permite desarrollo de aplicaciones que no se vean afectadas en tiempo real	Mayor complejidad a la hora de depurar un programa

Programación paralela

Es un tipo de **programación concurrente** que se diseñó para ejecutarse únicamente en un **sistema multiprocesador**, es decir, con **más de un núcleo** en su microprocesador. Este tipo de programación permite la ejecución **simultánea** de **una o varias tareas de un proceso**.

La programación paralela nos va a permitir mejorar enormemente el rendimiento de nuestros programas, si estos se ejecutan en diferentes núcleos, puesto que cada una de las ejecuciones en cada núcleo será una tarea del mismo programa. Estas podrán cooperar entre sí, por lo que resuelven el problema muchísimo más rápido. La programación paralela también se conoce como **multitarea, multihilo**, **multihebra** o **multithreading**.

La mayor ventaja que nos ofrece la programación paralela es que va a conseguir aumentar el rendimiento de nuestros programas, siempre y cuando sepamos implementar de forma correcta esta técnica, ya que podremos hacer que cada una de las tareas que creemos se ejecute en un procesador diferente.

Al igual que muchas otras técnicas, la programación paralela tiene una serie de inconvenientes. El mayor de ellos es la **complejidad** del diseño de sus algoritmos, concretamente, en la parte de **comunicación** de los **distintos procesos** o hilos. Aunque para resolver este problema tenemos varias opciones, como la utilización de **semáforos**, **interbloqueos**, **algoritmos de exclusión mutua**, etc., su implementación sigue siendo compleja.

En la actualidad, podemos encontrar las siguientes **arquitecturas** que utilizan **programación paralela**:

- **Sistemas multinúcleo**: Aquí podemos considerar a los microprocesadores actuales, ya que **todos** tienen varios núcleos.
- **Microprocesadores específicos**: Como procesadores gráficos, procesadores para videojuegos, procesadores embebidos, etc.

Tipos de paralelismos:

Paralelismo de grano fino: **No requiere mucho conocimiento** de código, ya que se realiza casi automáticamente. Se da cuando las subtareas deben comunicarse **muchas veces por segundo**. Se utilizará para cosas de poco trabajo para prevenir un interbloqueo.

Paralelismo de grano grueso: Es de alto nivel y **requiere de mayor conocimiento** de código ya que la mayor parte del mismo se va a paralelizar. Obtendrá mejores resultados que la de grano fino. Se da cuando las tareas **no** se comunican **muchas veces por segundo**. Será mucho más dificil de implementar. Se puede ver como la acumulación de granos finos que permitan funcionar de forma independiente.

Programación distribuida

Se da en **sistemas distribuidos**, que son un **conjunto de ordenadores** interconectados que comparten un **mismo estado**, dando la impresión de que son un único sistema cuyo objetivo es compartir recursos. El sistema distribuido más conocido por todos es la **red Internet**.

La programación distribuida siempre ha estado enfocada a desarrollar software para sistemas distribuidos, abiertos, escalables, transparentes y muy tolerantes a fallos. Este tipo de programación utiliza la **arquitectura hardware cliente-servidor**.

En la actualidad, encontraremos sistemas donde se utiliza arquitectura y programación distribuida en los siguientes casos:

- **Redes**: En las redes de **ámbito local**, se conectan **varios microprocesadores** a través de **una red** de conexión de alta velocidad, lo cual va a formar lo que conocemos como **clúster**, que son **varios ordenadores** que funcionan como un **único ordenador**.

- **Supercomputadores**: Estos computadores son **sistemas computacionales muy potentes** y se utilizan para tareas que necesitan una **enorme capacidad de cálculo**, como militares, meteorología, simulación de moléculas, etc.

- **Grid Computing**: En este tipo de computación distribuida, van a poder usarse **ordenadores muy potentes conectados** en red entre sí.

- **Cloud Computing**: El cloud computing o cómputo en la red son sistemas donde podremos tener **varios recursos** (**uno** de los más conocidos es el **espacio en disco**). Las máquinas que ofrecen ese servicio pueden estar en otra parte del mundo y están **interconectadas**.

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Procesos

Tipos de procesos de acuerdo al **modo de ejecución**:

- **Por lotes**: Este tipo de procesos están formados por una **serie de tareas** que se van a realizar. El usuario que las ejecuta únicamente está interesado en su **resultado final**, y no en su ejecución (antivirus).
- **Interactivos**: En este tipo de procesos, habrá una interacción del usuario y del propio proceso, que puede **pedir al usuario datos** necesarios para su ejecución (editor de imágenes).
- **En tiempo real**: estos consisten en ciertas tareas en las que el **tiempo de respuesta** por parte del **sistema es crucial** (conducción automática de un coche).

Tipos de procesos de acuerdo al **origen de la ejecución**,

- Procesos en **modo Kernel**: Son los procesos que ejecuta el propio **núcleo del sistema** operativo y que el usuario no podrá controlar (gestión de **memoria**, la gestión de **tráfico de red**, etc.)
- Procesos en **modo usuario**: estos los **lanza el propio usuario**. Son los procesos más «normales» o usuales.

Los procesos están formados por una **estructura llamada PCB** (**Bloque de Control de Proceso**). Toda la información que se guarda en el PCB es única y nos permitirá controlar al proceso.

Los PCB están **formados por**:

- **PID** o Identificador de Proceso: Es un número único para cada proceso que nos permitirá identificarlo de forma única.
- **Estado** actual: Este será el estado actual en el que se encuentra el proceso, puede ser listo, bloqueado, ejecutándose, etc.
- **Espacio de direcciones** de direcciones de memoria: Esto nos indicará dónde comienza la zona de memoria que está reservada para cada proceso.
- **Información** importante para la **planificación**: Aquí podremos encontrar toda la información que nos permitirá planificar el proceso, como puede ser el tiempo de quantum, la prioridad, etc.

- **Información** de cambio de **contexto**: Aquí tendremos toda la información que tendremos que guardar o recuperar cuando ejecutemos un cambio de contexto, valor de los registros de la CP de la CPU, contador del programa, etc.
- **Recursos** utilizados por el proceso: Aquí tendremos los recursos que utiliza el proceso, como puede ser tráfico de red, ficheros, etc.

Estados de un proceso

El planificador (***scheduler***), **crea** y **pone** en **ejecución** los procesos del sistema operativo. Para el correcto funcionamiento de los procesos existe lo que conocemos como el **ciclo de vida** de un proceso, el cual, se compone de una **serie de estados** por los que pasará el proceso a lo largo de su ejecución, teniendo que cumplir ciertos requisitos para pasar de un estado a otro.

Los **estados** por los que puede pasar un proceso son:

• **Nuevo**: El **proceso sea crea** a partir de un **fichero ejecutable**, aunque aún no haya sido admitido en el grupo de procesos ejecutables por el sistema operativo.

• **Listo**: El proceso está **creado**, pero aún no está listo para ejecutarse, ya que el **planificador** del sistema operativo **no lo ha seleccionado** para entrar a ejecución.

• En **ejecución**: El **planificador** del sistema operativo, siguiendo un algoritmo de planificación concreto, ha **elegido al proceso** para que se ejecute. Este **se ejecutará** de forma continua hasta que se cumpla su **tiempo máximo de ejecución** o hasta que el **planificador lo saque** de este estado, pasando de '**Nuevo' a 'Listo**'. Si el proceso necesita algún **recurso**, lo podrá pedir mediante **interrupciones**.

• **Bloqueado / En espera**: El proceso ha **pedido algún recurso** mediante una interrupción y está esperando que ese recurso le sea concedido. Cuando **se le conceda, se desbloqueará** y volverá a pasar al estado '**Listo**'.

• **Terminado**: El proceso ha finalizado su ejecución y **libera todos los recursos** que estaba acaparando, esperando que el sistema operativo lo destruya.

SWAPING

El **SWAPING** (intercambio) viene dado por una **pequeña partición en el disco duro** que automáticamente el sistema operativo creará cuando es instalado y que nos servirá para estos casos.

Cuando el ordenador se quede sin RAM disponible, este **espacio de disco** actuará como un **fragmento más de RAM** y se **guardarán ahí los procesos** que estén **bloqueados**, que podremos **recuperar más adelante** de una forma segura.

Gestión de procesos

El **sistema operativo** es el encargado principal de toda la **gestión de los procesos**, que casi siempre sigue las órdenes del usuario. Al abrir un programa, es el sistema operativo el responsable de **crear y poner en ejecución** el proceso que corresponde al programa en sí.

Cuando el **procesador**, mediante la **orden del planificador** del sistema operativo, pasa a ejecutar un **nuevo proceso**, es el **sistema operativo** quien debe **guardar todo el contexto** que tiene el proceso actual y quien debe restaurar el contexto que tenía el proceso que el planificador de procesos ha decidido ejecutar. El sistema operativo también es el responsable de **controlar toda la comunicación** y sincronización entre los procesos, además de la **eliminación y terminación** estos.

En **la multiprogramación** los **procesos irán intercambiándose** con los que están en ejecución, para que todos usen el procesador de igual forma, teniendo una ejecución concurrente de los procesos.

El sistema operativo organiza los procesos en **varias colas**:

- Una cola de procesos que van a contener **todos los procesos.**
- Una cola de procesos **preparados**, que contiene todos los procesos en **estado 'Listo**'.
- **Varias colas** que contienen los procesos que están en **estado 'Bloqueado**' y que están esperando alguna operación de entrada/salida.

Eventos que van a provocar la **creación de un proceso son:**

1. El **arranque** del sistema.
2. La ejecución, desde un proceso, de una **llamada al sistema para la creación de otro** proceso.
3. Una **petición de usuario** para crear un proceso.
4. El inicio de un **fichero por lotes**.

Tipos de Planificación de procesos

Un **planificador** (**scheduler**) de procesos se encargará de seleccionar los procesos que van a ejecutarse en el procesador, imponiendo un **orden de ejecución** entre los procesos de las colas.

Hay dos tipos de planificación de procesos:

A corto plazo: Este tipo de planificadores seleccionan qué proceso de la cola de procesos preparados va a pasar a ejecución. Estos se invocan **muy frecuentemente**. Deben de ser **muy rápidos** en la toma de decisiones. Los algoritmos serán muy sencillos. Ejemplos:

- **Sin desalojo o cooperativa**: Se cambia el proceso en ejecución si se ha **bloqueado o terminado**.
- **Apropiativa**: Cambiará el proceso en ejecución si en cualquier momento otro proceso con **mayor prioridad** puede ejecutarse.
- De **Tiempo compartido**: Cada cierto tiempo (quantum), quitan el proceso que estaba en ejecución y seleccionan otro proceso para que se ejecute. Todas las **prioridades** son consideradas **iguales**.

A largo plazo: Seleccionan los procesos nuevos que deben pasar a la **cola de procesos preparados**. Estos planificadores son invocados con poca frecuencia, ya que son **muy lentos**.

Los planificadores no se encargan de crear las estructuras que componen los procesos, únicamente se encargan de **decidir**, mediante un algoritmo, **qué proceso se puede ejecutar** en qué momento. Las **estructuras de los procesos se crearán solo una vez**, que es cuando el proceso se crea.

Algoritmos de planificación

Planificar los procesos que se van a ejecutar es la práctica de un **conjunto de políticas y mecanismos del propio sistema operativo**, incorporadas en un **módulo** que ya conocemos, llamado **planificador** de procesos.

Decidide **qué procesos** (que estén listos para ser ejecutados), deben iniciarse **y en qué orden** deben hacerlo. El panificador de procesos debe dar **servicio a todos los procesos** que estén en ejecución en un momento dado y **no dejar a ninguno de ellos** sin ejecutarse. Para ello el planificador puede utilizar distintos **algoritmos de planificación**:

- **Primero en llegar** (**FCFS o FIFO**): Este tipo de planificador es uno de los más simples y hará que se vayan ejecutando los procesos **según se creen**, **sin** ningún otro **criterio**.
- Prioridad al **más corto**: Este tipo de planificador ordenará los procesos según el tiempo de ejecución que necesiten e irá pasando a ejecución priorizando el que **menos tiempo necesite**. Con esta planificación, se produce lo que se conoce como **inanición**, ya que, si hay un proceso que necesita mucho tiempo de ejecución y siempre entran otros que necesiten menos, **nunca se ejecutará**.
- Prioridad al **más largo**: Este tipo de planificador ordenará los procesos según el tiempo de ejecución que necesiten e irá pasando a ejecución priorizando el que **más tiempo necesite**. Al igual que con la anterior, con esta planificación también se produce **inanición**, ya que, si hay un proceso que necesita muy poco tiempo de ejecución y siempre entran otros que necesiten más tiempo, **nunca se ejecutará**.
- **Round Robin**: Este tipo de planificación tiene un tiempo de ejecución llamado **quantum**, que es el **tiempo que dejará ejecutarse a cada proceso**, sacándolos de ejecución una vez pasado el quantum de tiempo. **Todos los procesos se ejecutarán poco a poco**.
- Planificación por **prioridad**: Este tipo de planificación utiliza la prioridad de ejecución que tienen los procesos, ejecutando antes el más prioritario. También puede producir **inanición** a procesos con poca prioridad.
- Planificación de **colas múltiples**: Este tipo de planificador es una **mezcla de todos los anteriores**.

Cambios de contexto

Cuando el planificador de procesos decide sacar un proceso del **estado 'Ejecutándose**' y volver a ponerlo en el **estado 'Listo'** (lo cual ocurre muy frecuentemente), el sistema operativo debe ser capaz de **guardar el contexto** del proceso actual y restaurar el contexto del proceso que el planificador ha elegido ejecutar. Es en este momento cuando se produce un **cambio de contexto**.

Esto significa que deben guardarse todos los datos de la situación del proceso que se estaba ejecutando. Este cambio de contexto se producirá también cuando hay una interrupción, es decir, cuando un proceso que se estaba ejecutando necesita un recurso y produce la interrupción para que se le proporcione.

Cada vez que se produzca un cambio de contexto, el sistema operativo debe **guardar**:

- **Estado** en el que se encontrada el **proceso**.
- **Estado** del **procesador**, debiendo guardar todos los **valores** que tenían los diferentes **registros** del **procesador** en el momento del cambio de contexto.
- Información de gestión de **memoria**, debiendo guardarse el espacio de memoria que tenía **reservado el proceso**.
- **Contador de programa**, que es el encargado de indicar por **dónde va ejecutándose el proceso**, lo cual nos permitirá, más adelante, seguir ejecutando la instrucción del proceso desde donde se quedó.
- **Puntero de pila**, que se encarga de apuntar a la parte **superior de la pila de ejecución** del **proceso**.

Podemos decir que el **cambio de contexto** es «**tiempo perdido**» o tiempo no aprovechado, debido a que el procesador **no hace un** «**trabajo útil**» durante ese tiempo.

Una vez el proceso vuelva a entrar al **estado 'Ejecutándose'**, todos los **datos** que han sido **salvados** se deberán volver a **restaurar** para que el proceso pueda seguir ejecutándose como si nada hubiese pasado.

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Gestión de procesos

Creación de procesos (Java)

Utilizaremos la clase **Process**.

Tenemos los siguientes métodos:

- **ProcessBuilder.start()**
- **Runtime.exec()**

Crean un proceso nativo del sistema operativo donde se está ejecutando la máquina virtual de Java.

Devolverán un objeto de la clase **Process**.

Ejemplo de creación de proceso en java:

Este ejemplo está realizado en un sistema Linux, y el usuario es “marc”.

```
(...)
static ProcessBuilder pb;
public static void main(String[] args) {
        pb = new ProcessBuilder().command("touch", "/home/marc/"
        + "Escritorio/prueba.txt");
        try {
                /*
                Este método ProcessBuilder.start() iniciará un proceso nuevo. Este
                va a ejecutar el comando y los argumentos que le indiquemos en el
                método command() arriba implementado, y se va a ejecutar en el
                directorio de trabajo que le indiquemos con el método directory().
                En este caso no usaremos el método directory(), ya que incluimos la
                información en el método command().
                Además, podrá utilizar las variables de entorno del sistema
                operativo que estén definidas en environment().
                */
                Process process = pb.start();
                int retorno = process.waitFor();
                System.out.println("El proceso touch devuelve: " + retorno);

                /* Segundo proceso, con Runtime.exec, cambiamos nombre al archivo
                recién creado anteriormente.
                */
                Process procesoMv = null;
                // Este método exec() es el de 1 parámetro tipo String.
                procesoMv = Runtime.getRuntime().exec("mv /home/marc/"
                + "Escritorio/prueba.txt /home/marc/Escritorio/prueba2.txt");
                int waitFor = procesoMv.waitFor();
                System.out.println("El proceso mv devuelve: " + waitFor);
```

```
                /* Ahora ejecutaremos otro método exec() de 3 parámetros, y usaremos
                el tercero para indicar el directorio de ejecución:
                Runtime.exec(String[] cmdarray, String[] envp, File dir)
                Este método ejecutará el comando que le especifiquemos con sus
                correspondientes argumentos en el parámetro cmdarray, en un proceso
                hijo que será totalmente independiente. Además, se ejecutará el
                entorno de trabajo indicado en el parámetro envp, y en el directorio
                de trabajo especificado en el parámetro dir.
                */
                Process procesoMkdir = null;
                // Creamos el argumento cmd para usar mkdir y crar un nuevo directorio
                String[] cmd = {"mkdir", "nuevo_directorio"};
                /*
                Cada unos de los índices de este array cmd será un comando o
                argumento, como si cortáramos en trozos el comando de la consola
                y lo crearemos en el directorio de trabajo /Escritorio.
                */
                File dir = new File("/home/marc/Escritorio/");
                procesoMkdir = Runtime.getRuntime().exec(cmd, null, dir);
                int waitForMkdir = procesoMv.waitFor();
                System.out.println("El proceso mkdir devuelve: " + waitForMkdir);
                } catch (IOException ex) {
                        Logger.getLogger(CreacionProcesosLinux.class.getName()).
                        log(Level.SEVERE, null, ex);
                } catch (InterruptedException ex) {
                        Logger.getLogger(CreacionProcesosLinux.class.getName()).
                        log(Level.SEVERE, null, ex);
                }
        }
```

Estos dos métodos comprobarán que el comando que les hayamos indicado sea un **comando o fichero ejecutable válido** en el sistema operativo que estemos utilizando. A fin y al cabo, el hecho de crear un nuevo proceso va a **depender del sistema operativo** que estemos utilizando, y pueden ocurrir diversos **problemas** como, por ejemplo:

- No encontrar el **ejecutable** debido a la **ruta** indicada.
- No tener **permisos** de ejecución.
- No ser un **ejecutable válido** en el sistema.

En la mayoría de los casos, se lanzará una excepción, dependiendo del sistema operativo que estemos utilizando, aunque siempre va a ser una **subclase de IOException**.

Lanzar un método dentro de un proceso

Podemos lanzar, en un proceso, un **método de una clase** java creada por nosotros.

Lo haremos de la siguiente manera:

1- Crearemos una clase y el método o métodos que queremos **ejecutar en forma de proceso** independiente.

2- Crearemos un método main desde el cual llamaremos al método que queremos ejecutar. Si tiene parámetros, utilizaremos el parámetro del **método main String[] args** para **enviar** los **parámetros** que necesitemos al crear el ProcessBuilder (en el siguiente ejemplo serán los últimos 2 parámetros del constructor de ProcessBuilder).

3- Crearemos (en este ejemplo) la **función** ejecutarClaseProceso(Class clase, int n1, int n2), dentro de la clase LanzarMetodoDesdeProceso, el cual nos permitirá **ejecutar una clase en un proceso independiente**.

Ejemplo de lanzamiento de un método dentro de un proceso en java:

Clase **Sumador.java** (nos servirá para realizar la suma):

```
public class Sumador {
	/**
	* Este método suma todos los números que hay en un intervalo.
	*
	* @param numero1 Inicio del intervalo.
	* @param numero2 Fin del intervalo.
	* @return
	*/
	public static int sumar(int numero1, int numero2) {
		int suma = 0;
		for (int i = numero1; i <= numero2; i++) {
		suma += i;
		}
		return suma;
	}
	public static void main(String[] args) {
		// Obtenemos los argumentos pasados al crear el proceso:
		int numero1=Integer.valueOf(args[0]);
		int numero2=Integer.valueOf(args[1]);
		int suma = sumar(numero1, numero2);
	}
}
```

Clase **LanzarMetodoDesdeProceso.java** (utilizará la clase Sumador para ejecutar un proceso; Esta es la clase principal del proyecto):

```
public class LanzarMetodoDesdeProceso {
	public static void main(String[] args) {
		try {
			Scanner scanner = new Scanner(System.in);
			int numero1, numero2;
			System.out.println("Dame un primer número inicial para el intervalo");
			numero1 = scanner.nextInt();
			System.out.println("Dame el número final del intervalo");
			numero2 = scanner.nextInt();
			/**
			* Ejecutamos el proceso con el método ejecutarClaseProceso,
			* definido en esta misma clase, abajo, y obtenemos el valor de
			* salida recuperado en una variable. Este método hará 2 cosas:
			* Ejecutará el proceso, y obtendrá el resultado del proceso
			* (resultado satisfactorio con valor 0).
			*/
			int valorSalidaProceso = ejecutarClaseProceso(Sumador.class,
			numero1, numero2);
			if (valorSalidaProceso == 0) {
			System.out.println("Proceso ejecutado correctamente";
			} else {
			System.out.println("Ocurrió un error al ejecutar el proceso...");
			}
		} catch (IOException | InterruptedException ex) {
			System.err.println("Error: " + ex.toString());
			System.exit(-1);
		}
	}

	/**
	* Como podemos ver, al método de la clase principal le pasamos 3 atributos:
	* @param clase El nombre de la clase que queremos ejecutar.
	* Parámetros de tipo int, que serán los parámetros que necesite la clase para
	* ejecutar su método. Estos parámetros podrán cambiar y podrán ser tantos como
	* necesitemos (en este ejemplo serán 2, para hacer una suma):
	* @param n1
	* @param n2
	* @return Retorn un int, la suma del intervalo.
	* @throws IOException
	* @throws InterruptedException
	*/
```

```
        public static int ejecutarClaseProceso(Class clase, int n1, int n2)
                throws IOException, InterruptedException {
                // defino en la variable javaHome dónde está el home de java
                String javaHome = System.getProperty("java.home");
                /**
                * defino en la variable javaBin dónde está el binario ejecutable de la
                * máquina virtual java
                */
                String javaBin = javaHome
                + File.separator + "bin"
                + File.separator + "java";
                /**
                * Defino la variable classPath que es el nombre completo de la clase
                * java
                */
                String classPath = System.getProperty("java.class.path");
                // Obtenemos el nombre canónico de la clase que se ejecutará
                String ClassName = clase.getCanonicalName();
                /**
                * Creamos el proceso a ejecutar. Los dos últimos parámetros son los
                * parámetros de la clase main de la clase.
                *
                * Cuando estamos creando el proceso con ProcessBuilder, los dos últimos
                * parámetros que pasamos son los dos enteros citados, ya que son los
                * que necesitará la clase que queremos ejecutar. Aquí podremos pasarle
                * todos los parámetros que necesitemos (o ninguno, si se da el caso),
                * que deben ser, obligatoriamente, del tipo String, ya que, en el main
                * de la clase, utilizaremos el array de Strings que recibe para
                * recuperar todos los parámetros que estamos pasando en este punto.
                */
                ProcessBuilder builder = new ProcessBuilder(javaBin, "-cp",
                classPath, ClassName, String.valueOf(n1),
                String.valueOf(n2));
                Process proceso = builder.start();
                // Con waitFor espero a que se ejecute el proceso
                proceso.waitFor();
                // Devolvemos el valor de salida del proceso, si es 0 será ejecutado ok
                return proceso.exitValue();
        }
}
```

Nota: si lanzamos procesos muy grandes, podemos hacer que el ordenador se quede sin memoria RAM.

Terminar un proceso

Vamos a poder **finalizar** en java, cuando necesitemos, un **proceso hijo que haya creado**.

Esto lo podremos hacer ejecutando el método **destroy**, que pertenece a la **clase Process**. Este método va a eliminar el **proceso hijo** que indiquemos **liberando**, a su vez, todos los **recursos** que estuviera utilizando, dejándolos disponibles para que el sistema operativo pueda asignarlos de nuevo. En Java, todos los recursos correspondientes al proceso que hayamos eliminado serán liberados por el **recolector de basura** cuando se ejecute la próxima vez.

En caso de que **no forcemos la finalización de la ejecución** del proceso hijo, este se **ejecutará de forma normal y completa**, terminando y **liberando sus recursos al finalizar**. Esto **también** se puede producir cuando el **proceso hijo** ejecuta la **operación exit()** para forzar la **finalización de su ejecución**.

Ejemplo de creación de proceso y su finalización con destroy:

```
public class DestroyProceso {
	public static void main(String[] args) {
		// Creamos la ruta para la ejecución del proceso del navegador chrome:
		String RUTA_PROCESO
			= "C:\\Program Files\\Google\\Chrome\\Application\\chrome";
		// Creamos el proceso:
		ProcessBuilder pb = new ProcessBuilder(RUTA_PROCESO);
		// Creamos objeto Scanner para leer datos del teclado
		Scanner teclado = new Scanner(System.in);

		// Lanzamos el proceso
		Process proceso;
		try {
			proceso = pb.start();
			System.out.println("¿Deseas terminar el proceso chrome? (s/n)");
			if (teclado.nextLine().charAt(0) == 's') {
				// Destruimos el proceso
				proceso.destroy();
			}
		} catch (IOException ex) {
			System.err.println("Error: " + ex.toString());
			System.exit(-1);
		}
	}
}
```

Primero lanzamos el proceso de forma normal y, en caso de que queramos terminar su ejecución, llamaremos al método destroy.

Comunicación entre procesos

Para leer y mostrar información, contamos con:

- La **entrada** de datos **estándar** (*stdin*): Obtener los datos necesarios para que nuestro proceso se ejecute de teclado, de un fichero, etc.
- La **salida estándar** (*stdout*): Mostrar información en un fichero, en un socket...
- La **salida de error** (*stderr*): Mostrar la información de los errores que ocurran en la pantalla, en un fichero, etc.

En Java el **proceso hijo** que creemos no va a tener su propia interfaz de comunicación, por lo que **no podrá comunicarse con el proceso padre directamente**. Deberemos **redireccionar todas sus salidas y entradas** (*stdin*, *stdout* y *stderr*) mediante:

- **OutputStream**: Flujo de **salida**. Está conectado a la salida estándar del proceso hijo. Se redirecciona con el método ***redirectOutput***.
- **InputStream**: Flujo de **entrada**. Está conectado a la entrada estándar del proceso hijo. Se redirecciona con el método ***redirectInput***.
- **ErrorStream**: Flujo de **salida** para los errores. Está conectado a la salida estándar de errores del proceso hijo. Se redirecciona con el método ***redirectError***.

Una vez hayamos **redirigido todos estos flujos**, podremos hacer una **comunicación** entre los procesos padre e hijo.

Si queremos **redireccionar** a la salida/entrada estándar, deberemos utilizar ***Redirect.INHERIT***.

En nuestro de ejemplo anterior, en la clase sumador, agregaríamos:

```
public static void main(String[] args) {
        // obtenemos los argumentos pasados al crear el proceso
        int numero1 = Integer.valueOf(args[0]);
        int numero2 = Integer.valueOf(args[1]);
        // Utilizaremos la salida estándar para imprimir el resultado:
        System.out.println("La suma iterativa desde el número " + numero1
                + " hasta el número " + numero2 + " es: "
                + sumar(numero1, numero2));
}
```

Y a nuestra clase LanzarMetodoDesdeProceso, en su método ejecutarClaseProceso, agrtegaríamos:

```
// Indicaremos las redirecciones de salida, para mostrar resultados
builder.redirectError(ProcessBuilder.Redirect.INHERIT);
/**
```

```
* Si deseamos redireccionar a un archivo: builder.redirectError(new
* File("errores.txt"));
*/
builder.redirectOutput(ProcessBuilder.Redirect.INHERIT);
```

Invocar al bash del sistema operativo

Podremos crear procesos de la **bash o consola** del sistema operativo.

Mediante el bash podremos ejecutar multitud de comandos que nos permitirán realizar operaciones muy potentes.

Para ejecutar el bash de **Windows:**

1- Deberemos crear un proceso con la ruta CMD, siendo **CMD** el proceso que se encargará de **ejecutar el bash**.

2- Obtendremos los flujos de entrada y salida del proceso que ejecutará el bash, para poder **interactuar** con él de forma correcta.

3- Con el **proceso lanzado**, podremos indicarle que **ejecute comandos** mediante la orden println.

4- Después de cada comando, deberemos **limpiar el flujo del proceso**, para evitar errores, mediante el método **flush** (limpiado, vaciado...).

El proceso se estará **ejecutando en segundo plano** e irá ejecutando todos y cada uno de los comandos que le indiquemos, que pueden ser estos tan complejos como necesitemos.

Hay que tener mucho cuidado si queremos ejecutar el bash, ya que, según el **sistema operativo** que estemos utilizando, tanto la forma de **invocarlo** como los **comandos** a ejecutar cambiarán.

Ejemplo de lanzamiento de bash o consola dentro de un proceso en java:

```
public static void main(String[] args) {
        // Se crea el proceso hijo:
        // → CMD es el programa que ejecuta la consola bash en Windows
        // → Para usarlo en Linux, bastará con crear el ProcessBuilder con el
        // String “/bin/bash” en vez de “CMD”
        ProcessBuilder builder_echo = new ProcessBuilder("CMD");
        Process proceso_echo;
        try {
                proceso_echo = builder_echo.start();
                final Scanner scanner = new Scanner(
        // Aquí obtenemos el flujo de entrada al proceso desde la entrada Scanner
                        proceso_echo.getInputStream());
                new Thread() {
```

```
                @Override
                public void run() {
                    while (scanner.hasNextLine()) {
                        System.out.println(scanner.nextLine());
                    }
                }
            }.start();
            // Obtengo la salida del proceso hijo:
            PrintWriter salida
                = new PrintWriter(proceso_echo.getOutputStream());
            /**
            * Usamos el objeto PrintWriter salida, que escribe en el flujo de
            * salida del proceso:
            */
            for (int i = 0; i < 10; i++) {
                salida.println("echo Iteración " + i);
                salida.flush();
            }
            // Cerramos los flujos
            salida.close();
        } catch (IOException ex) {
            System.err.println("Excepción de E/S: " + ex.toString());
        }
    }
```

Bibliografía:

Running Bash commands in Java. Stack Overflow. https://stackoverflow.com/questions/26830617/running-bash-commands-in-java

Sincronización entre procesos

Zona crítica

Puede que estemos lanzando más de un proceso al mismo tiempo. En ese caso, no tenemos forma de saber qué línea de código estará ejecutando cada uno de ellos en un momento dado. Si **dos o más procesos necesitan acceder** a una **variable** en memoria es posible que alguno de ellos la modifique y los demás ya no puedan ver su valor original, sino el valor que ya ha sido modificado por el otro proceso que accedió a ella en primer lugar.

Estas situaciones las vamos a encontrar en todos y cada uno de los programas que tengan más de un proceso en activo. Estas zonas críticas son muy **peligrosas** y deben protegerse mediante una serie mecanismos.

Mecanismos más comunes para **controlar** zonas críticas:

- Semáforos,
- Colas de mensajes,
- Pipes o tuberías,
- Bloques de memoria compartida.

En java, para sincronizar un **bloque de código entre varios procesos: incluiremos delante** la palabra reservada **synchronized: así** la propia máquina virtual de Java hace que ese **código sea seguro en la ejecución** de dos o más **procesos**.

Ejemplo de uso de **syncronized** en un bloque de código:

```
int i = 0, j = 0;
syncronized(ClaseASincronizar.class) {
        if (<2) {
                j++ ;
                i++ ;
        }
        System.out.println("ok") ;
        i = i * 2 ;
        j-- ;
}
```

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Introducción a la programación paralela o multihilo

Hilos de ejecución en un proceso

Este tipo de programación es un tipo de programación concurrente capaz de ejecutar al **mismo tiempo varias tareas o hilos**. Un **ejemplo** es la descarga de un archivo en el navegador mientras oímos música.

Un **hilo**, también conocido como **hebra**, es un **trozo de código** que se ejecuta **dentro de un proceso**, pero que puede ser ejecutado **en paralelo con otros hilos**.

El proceso podrá crearlos y lanzarlos, pero estos solo podrán utilizar los **recursos** de los que el **propio proceso** disponga. ¡Cuidado! **Diferentes hilos** de un proceso pueden necesitar **acceder a los mismos recursos** y ocasionar «**inconsistencias**» en nuestros programas.

Los **procesos activos** seguirán en ejecución **hasta que** todos y cada uno de los **hilos** que han lanzado **terminen** de ejecutarse. Entonces el **proceso podrá ser destruido** por el sistema operativo y todos los **recursos serán liberados**.

Cuando ejecutamos un programa, se crean también tanto un proceso como un hilo primario.

Sobre los hilos, debemos tener en cuenta que:

- Los hilos **no pueden existir de forma independiente a un proceso**.
- Los hilos **no** se podrán **ejecutar por sí solos**.
- Un **proceso** podrá tener tantos **hilos** ejecutándose como necesitemos.

Diferencia entre proceso e hilo

*Los **hilos** se podrán ejecutar sólo dentro de un proceso; van a **depender de un proceso para ejecutarse**. Los **procesos** son **independientes unos de otros**, teniendo **zonas de memoria distintas**, cosa que en los **hilos no** es así.*

Ventajas / desventajas
hilos

Ventajas aparecen una vez tenemos más de uno (**programación multihilo**). En este caso, dentro de **un mismo proceso**, tendremos **múltiples hilos** en ejecución, los cuales estarán realizando **actividades totalmente distintas**, **pudiendo o no cooperar** entre ellos.

Ventajas (en **comparación con los procesos**):

• **Compartir recursos**: **Compartir** tanto la **memoria** como los **recursos** dentro del proceso que los crea. **Operaciones** mucho más **rápidas**.

• Más **eficiente** y **ahorro** de **memoria**: **Misma zona de memoria**, **no va a suponer una reserva adicional** de esta.

• **Capacidad de respuesta**: Permitirá al proceso atender **otras peticiones** que le envíe el usuario a través de los **otros hilos**.

• **Paralelismo real**: En microprocesador **multinúcleo**, los **hilos** se podrán ejecutar en un **núcleo diferente cada uno:** usar el procesador de forma paralela (**varias instrucciones al mismo tiempo**).

Desventajas:

• **No todos los lenguajes** de programación soportan hilos.

• Hay que **controlar todos los problemas** (**comunicación** y **sincronización**). Los problemas que se darán cuando estos comparten recursos son:

- ➔ **inanición**,
- ➔ **bloqueo** activo,
- ➔ acceso a los **recursos críticos**,
- ➔ zonas de **exclusión mutua**,
- ➔ condiciones de **carrera**,
- ➔ errores de **inconsistencia** en la memoria compartida,
- ➔ etc.

Si hay más de un proceso con varios **hilos** creados y ejecutándose, estos **nunca «verán» los hilos de otro proceso**. En cambio, cuando un **proceso crea otro proceso hijo**, ocurre totalmente lo contrario, ya que este podrá **interactuar con otros procesos** que existan.

- ➔ Si es un **único problema con varias partes** identificables, podremos **usar hilos**;
- ➔ Si son varios **problemas diferentes** dentro del mismo, podremos **usar procesos**.

Recursos compartidos por los hilos

Elementos de un hilo:

- Un **identificador** único (identificado de forma rápida).
- **Contador** de programa (podrá **ejecutar** su código de forma **independiente**). Nota: El **contador de programa no se comparte** entre los hilos.
- **Registros** asociados (realizar todas las operaciones aritmético-lógicas que necesite de forma independiente).
- Una **pila propia** (ejecutar llamadas a funciones que necesite de forma independiente).

Compartir recursos con otros hilos:

- El **código** a ejecutar.
- **Variables globales** que se encontrarán en la **zona crítica**.
- Recursos del **sistema operativo** (ficheros, sockets, bases de datos, etc).

Compartir recursos es **muy peligroso** (acceder a una misma variable global y cambiar el valor de dicha variable: el primer hilo accede a la variable y cambia su valor; los demás hilos, cuando accedan a ella, no accederán al valor anterior, sino al cambiado... 😮).

Se soluciona utilizando un esquema de **bloqueo y sincronización** entre varios **hilos**. Su **implementación no es nada sencilla** y complicarán bastante nuestros programas.

Ejemplo de varios hilos:

Cuando una ejecución de código es automática, respondiendo a alguna acción del usuario, normalmente ese código estará en un hilo, porque se podrá ejecutar al mismo tiempo que otro código. Como por ejemplo ocurre al escribir en un editor de texto y el editor corrige la ortografía.

Estados de un hilo

Los hilos tienen un **ciclo de vida**.

Los hilos que crea el **sistema** son **hilos demonio** o de sistema.

El **comportamiento** de cada hilo **dependerá del estado** en el que se encuentre (va a **definir la operación** que está realizando).

Los **estados** por los que puede pasar un hilo son los mismos que los de un proceso, pero con peculiaridades:

- **Nuevo**: El hilo ya ha sido creado, **listo para ejecutarse**, aunque no haya sido elegido para empezar a ejecutarse.
- **Listo o ejecutable**: Cuando indiquemos que está listo para poder ejecutarse.
- **En ejecución**: Está listo para ejecutarse y el sistema operativo lo ha **seleccionado** para que se ejecute.
- **Bloqueado**: Cuando **necesite** algunos **recursos** de entrada/salida que debe proporcionarle el usuario o el propio sistema operativo. **No** se le va a asignar **tiempo de CPU** al hilo. Estará **listo para volver a ser usado**.
- **Finalizado**: Termina su ejecución, en espera de que el sistema operativo lo destruya y libere sus recursos.

Clases para hilos en Java

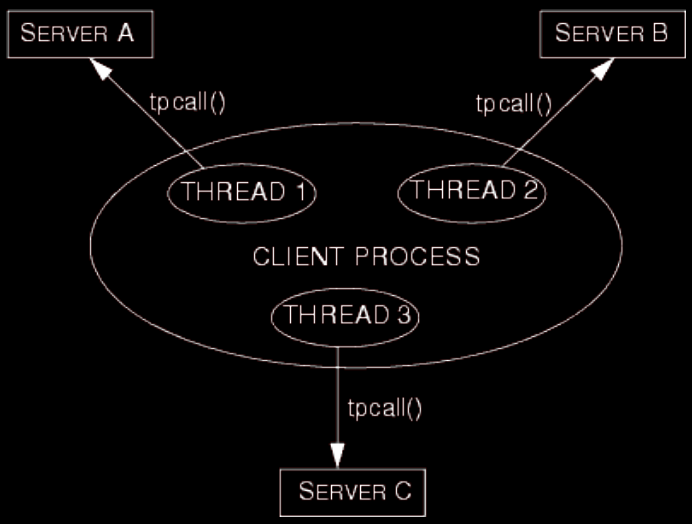
Dentro del paquete **java.lang**:

- Clase **Thread**: Crear hilos totalmente funcionales a los que podremos **asignar el código** que queramos para que lo ejecuten. Las **clases que queramos que sean hilos** deberán heredar de esta.
- Interfaz **Runnable**: Añadir la **funcionalidad de hilo** a cualquier otra clase por **implementar esta interfaz**.
- Clase **ThreadDeath**: **Manejar y notificar errores** en el uso de las hebras. Hereda de la clase Error.
- Clase **ThreadGroup**: Manejaremos un **grupo de hilos** de forma conjunta, haciendo que se ejecuten de una forma bastante **más eficiente**.

Algunos **métodos** de la clase **Thread**:

- **new()**: Este método **creará** un hilo.
- **start()**: (Preparado) Indica que el hilo está **listo para ejecutarse**.
- **run()**: (En ejecución) Ejecuta el hilo, que **empezará a ejecutarse** de forma paralela.
- **sleep()**: (Bloqueos) Hace que el hilo **se bloquee** una cantidad de tiempo dada en **milisegundos** .
- **wait()**: (Semáforos) Hace que el hilo **espere** la ejecución de **otra tarea** para volver a ejecutarse .
- **getState()**: Este método nos devolverá el **estado** en el que se encuentra actualmente el hilo.
- **public boolean isInterrupted()**: podemos **llamarlo varias veces** de la clase hebra **para ver si ha sido interrumpido**. Prueba si este hilo ha sido interrumpido. Este método **no afecta el estado interrumpido** del hilo. Una interrupción de subproceso ignorada porque un subproceso no estaba activo en el momento de la interrupción se reflejará en que este método devuelva falso.

Ejemplo de uso de hilos:



La programación multihilo no es solo cosa de Java. Cualquier lenguaje de programación potente de hoy en día la va a soportar::

- C,
- C++,
- C#,
- Python,
- Kotlin,
- Swift
- etcétera.

Ejemplo de hilos
“caja”

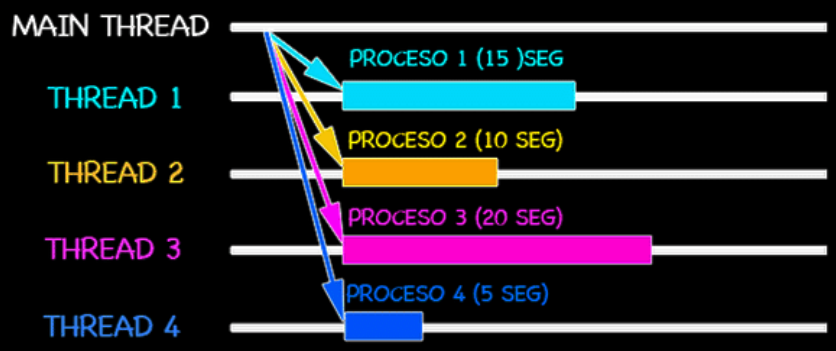
Una cola en un supermercado y el cobro por parte de los empleados de caja (proceso).

Una **serie de clientes** van a **pagar** con sus carros llenos de productos (hilos).

Sin programación **multihilo**, únicamente tendríamos **una caja abierta**, en donde se iría pasando, uno a uno, todos los artículos de todos los clientes. Muy lento.

Con multihilo: más eficiente y rápido:

> **4 cajas** con 4 empleados, en lugar de solamente 1. Cada empleado de caja pasará, uno a uno, todos los productos de cada cliente.
>
> Los clientes **van a tardar menos** en ser atendidos.
>
> El **proceso se ejecuta mucho más rápido**.



PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Manejo de hilos

Creación de hilos

Thread

En el lenguaje de programación Java, un **hilo** será creado mediante una instancia de la clase **Thread**, que nos permitirá **gestionar** completamente nuestros hilos.

Para crear una clase con la **funcionalidad de hilo** en Java, deberemos seguir los siguientes pasos:

- Crear una clase que **herede** de la clase **Thread**.
- **Reescribir el método run** en el que deberemos poner todo el **código que queramos que ejecute** nuestro hilo.
- Crear un **objeto** de la nueva clase, que será el **hilo** con el que trabajaremos.

```
public class Hilo1 extends Thread {
    @Override
    public void run() {
        // Aquí va el código del hilo
    }

}
```

Dentro de la clase, podremos crear todas las **variables y métodos que necesitemos**, incluidos constructores, métodos get, set y toString, ya que lo que estamos creando es una **clase sin ningún tipo de peculiaridad**.

Mediante los **constructores**, podremos crear los hilos con los **datos que necesitemos**, y con los métodos **get y set**, podremos **obtener o modificar** sus variables sin ningún tipo de problema.

Dentro del **método run** deberemos crear un **bucle infinito**, ya que la **hebra estará ejecutándose** de forma constante hasta que **decidamos finalizarla** o hasta que finalice el propio programa.

```
public class Hilo1 extends Thread {
	@Override
	public void run() {
		// Con el valor true creamos un bucle infinito en while
		while (true) {

		// Aquí va el código a ejecutar...

		}
	}
}
```

Runnable

El lenguaje de programación Java también nos permite crear hilos implementando la **interfaz Runnable**, la cual nos ofrecerá, igualmente, **gestionar completamente los hilos** de nuestros programas.

Pasos para crear un hilo implementando Runnable:

- Crear una **clase** que **implemente** la interfaz **Runnable**.
- Reescribir el método **run**, en el que deberemos poner todo el **código** que queramos que **ejecute** nuestro hilo.
- Crear un **objeto** de la nueva clase, que será el hilo con el que trabajaremos.

Tendremos que **reescribir el método run** de forma **obligatoria**, cosa que no ocurría al heredar de Thread. Si, en este caso, **no reescribimos** el método, **no tendrá funcionalidad**.

Al igual que ocurría con el método anterior, en esta clase **podremos crear todos los atributos y métodos** que vayamos a **necesitar**, incluyendo constructores, métodos get, set y toString.

También, en el método run, deberemos crear un **bucle infinito** para que la **hebra** se quede en **segundo plano**, **ejecutándose constantemente** hasta que decidamos finalizarla o hasta que finalice el propio programa.

```
public class Hilo_Runnable implements Runnable {

	@Override
	public void run() {

		while (true) {
			// Aquí estará el código a ejecutar en el hilo
		}
	}
}
```

Nota:

El lenguaje de programación **Java** ofrece **dos mecanismos para crear hilos**, cosa que no ocurre en otros lenguajes de programación como, por ejemplo, el **C++**, que **solo ofrece una** forma de crearlos.

Java no nos permite utilizar la herencia múltiple, ya que solo puede heredar de una única clase. Si queremos que una clase que ya hereda de otra sea una hebra, no podremos hacerlo. Por este motivo, tenemos la posibilidad de implementar la **interfaz Runnable** para que **cualquier clase que necesitemos tenga la funcionalidad de esta hebra**, ya que, en Java, podremos implementar todas las interfaces que necesitemos.

Inicialización y ejecución de hilos

Mediante la herencia de Thread, **no bastará con crear el objeto con new()**.

Para que este hilo sea ejecutado y pase a segundo plano, debemos hacer que **pase al estado Ejecutándose** y, para ello, deberemos **iniciarlo** mediante el **método start()**. Hay que tener mucho cuidado de **no utilizar el método run()**, ya que no hará que se ejecute el hilo de forma paralela. El método **start()** hará que se **ejecute el método run() de forma paralela**.

El **método start()** va a realizar las siguientes **tareas**:

- **Reservar** todos los **recursos** necesarios para la correcta ejecución del hilo.
- **Llamar al método run()** y hacer que se ejecute de forma **paralela**.

```
public static void main(String[] args) {
            // creamos el hilo
            Hilo_Thread hiloThread = new Hilo_Thread();
            // y lo lanzamos de forma paralela con run():
            hiloThread.start();
        }
```

Implementando la interfaz Runnable, debemos seguir los siguientes **pasos**:

- Crear un **objeto** de la clase hilo.
- Crear un **objeto de la clase Thread** mediante el **objeto anterior**.
- Llamar al método **run()** y hacer que se **ejecute de forma paralela**.

Hay que tener en cuenta que, una vez hemos **llamado al método start()** de un hilo, **no podremos hacerlo de nuevo**, ya que el hilo se **está ejecutando**. En caso de hacerlo, tendremos una excepción del tipo IllegalThread**State**Exception.

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

            // Vamos acrear un hilo Thread:
            Hilo_Thread hiloThread = new Hilo_Thread();
            // Y lo lanzamos con start() que también lanza de forma paralela
            run()
            hiloThread.start();
            // ahora vamos a crear un hilo Runnable:
            Hilo_Runnable hiloRunnable = new Hilo_Runnable();
            // Creamos una hebra Thread a partir del hilo Runnable
            Thread hebraThread = new Thread(hiloRunnable);
            // y lanzamos la hebra Thread, que será un Runnable:
            hebraThread.start();

        }
```

Suspensión y finalización de hilos

Si necesitamos que nuestros hilos **ejecuten una acción cada cierto tiempo** deberemos **detener** el hilo de alguna forma.

El lenguaje de programación Java nos proporciona los siguientes métodos para **suspender hilos** de forma segura:

- Método **sleep()**:

  Conseguiremos que nuestro hilo «se duerma» una cierta cantidad de **milisegundos**, que tendremos que pasarle como parámetro. En realidad, el hilo pasará al estado **bloqueado** durante esa cantidad tiempo, volviendo a ejecutarse después. En caso de que necesitemos **reactivar el hilo antes** de que se cumpla el tiempo de bloqueo, podemos hacerlo mediante el método **interrupt().**

- Método **wait()**:

  Conseguiremos que el hilo se quede **bloqueado hasta nuevo aviso**.

  Para **desbloquear** el hilo:

  NotifiyAll(): notifica a **todos y cada uno** de los hilos que estén **bloqueados**.

  Notify(): podemos pasarle como **parámetro** una cantidad de **milisegundos** para que el hilo esté bloqueado, desbloqueándose cuando se cumpla esa cantidad de tiempo o cuando le llegue una **señal notify**, lo que antes ocurra.

Otra operación que podemos realizar es la de **finalizar un hilo** (que en condiciones normales finalizaría al terminar su tarea) directamente **por nosotros mismos**.

Para esto, podemos utilizar los siguientes métodos:

- Método **isAlive()**:

  Nos indica si un hilo está vivo o no. Devolverá '**true**' en caso de que el hilo esté en un **estado diferente al de finalizado** y '**false**' si está **finalizado**.

- Método **stop()**:

  Finaliza un hilo, pero es extremadamente **peligroso** utilizarlo, ya que puede dar situaciones de **interbloqueo de hilos** y hacer que nuestra **aplicación se bloquee** sin posibilidad de arreglo.

Planificación de hilos

Con **varios hilos** ejecutándose en segundo plano, **no** vamos a **poder controlar** cuál de ellos se ejecutará, ya que será el propio **planificador** del sistema operativo que estemos usando quien **decida** qué hilo se ejecutará en cada momento. Puede que tengamos **diferentes salidas de datos** por pantalla en las diferentes ejecuciones del programa.

Si **queremos** hacer que una **acción** se realice **después de finalizar un hilo**, podemos hacer que se **espere a que el hilo acabe** utilizando el método **join()**. A este método podremos incluirle como parámetro la cantidad de tiempo en **milisegundos** que queremos **esperar para que acabe el hilo** o, si queremos esperar a que **acabe** el hilo **de forma «natural»**, sin ninguna restricción de tiempo, no le pasaremos **ningún parámetro**.

Debemos tener **mucho cuidado** con el método join(), ya que, si se produce un **error de bloqueo** con los hilos anteriores, **no** habrá forma de que ese código **se ejecute**.

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

		Hilo hilo1 = new Hilo("Hilo 1", 1000);
		Hilo hilo2 = new Hilo("Hilo 2", 1000);
		Hilo hilo3 = new Hilo("Hilo 3", 1000);

		try {

			hilo1.start(); // iniciamos el hilo: estado listo o runnable
			hilo1.join(); // hacemos que se espere
			hilo2.start(); // start llama a run(), y el hilo se ejecuta
			hilo2.join();
			hilo3.start();
			hilo3.join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
```

Usos de hilos:

El **uso más común** que le vamos a dar a los hilos está relacionado con el **tiempo**. **Mostrar el propio tiempo** en una aplicación (**cronómetro, cuenta atrás...)**. O mostrar fecha y hora.

Para mostrar la hora actual del sistema, esta se tendrá que ir actualizando segundo a segundo o minuto a minuto. Esto no es más que un hilo ejecutándose de forma paralela, que **consulta la hora del sistema cada intervalo de tiempo** configurado para actualizarla en la interfaz de la aplicación.

Prioridad de hilos

En Java, los hilos van a tener una **prioridad de 1 a 10 (**1 la menor prioridad y 10 la máxima).

Si **no indicamos** nada, tendremos, por defecto, una prioridad media, o de **5**, en nuestros hilos.

La clase **Thread** tiene definidas **tres constantes** que van a representar los **niveles de prioridad** relativos a los hilos:

- **MIN_PRIORITY** tiene un valor de 1.
- **MAX_PRIORITY** tiene un valor de 10.
- **NORM_PRORITY** tiene un valor de 5 (**por defecto**).

Para la **gestión de las prioridades** de los hilos::

- Método **getPriority():** Podremos obtener la prioridad que tiene un hilo.
- Método **setPriority()**: Podremos cambiar en cualquier momento la **prioridad de un hilo** de nuestra aplicación. Deberemos pasársela al **método como parámetro**, que puede valer:
  - **Thread.NORM_PRORITY**
  - **Thread.MIN_PRIORITY**
  - **Thread.MAX_PRIORITY**

Cuando todos los hilos tengan la **misma prioridad**, el comportamiento será exactamente el mismo al que estamos acostumbrados. En este caso, si queremos que el **planificador** del sistema operativo **equilibre la ejecución** de los hilos, podremos usar el método **yield()** (ceder el paso) de la clase Thread.

```
public static void main(String[] args) {
        Hilo hilo1 = new Hilo("Hilo 1", 1000);
        Hilo hilo2 = new Hilo("Hilo 2", 1000);
        Hilo hilo3 = new Hilo("Hilo 3", 1000);

        /*
         *  Daremos prioridades a los hilos, aunque dependerá del
         *  planificador del sistema el orden final. Es prioridad,
         *  no planificación, recordemos. Para planificar debemos usar join()
         *  y así esperará el hilo hasta terminar su ejecución para ejecutar
         *  el siguiente hilo deseadop.
         */
        hilo1.setPriority(Thread.MAX_PRIORITY);
        hilo2.setPriority(Thread.NORM_PRIORITY);
        hilo3.setPriority(Thread.MIN_PRIORITY);

        hilo1.start();
        hilo2.start();
        hilo3.start();
    }
```

Nota:

Peligros de cambiar la prioridad: Cuando cambiamos las prioridades en los hilos tenemos que tener muchísimo cuidado, ya que podemos conseguir que haya **hilos con la mínima prioridad** y éstos **no se lleguen a ejecutar nunca**, produciendo una **inanición** de los mismos.

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONESMULTIPLATAFORMA

Sincronización de varios hilos
Monitores y Semáforos

Vamos a ver problemas comunes de la **sincronización de hilos** en programas paralelos o concurrentes.

Vamos a ver qué son los **monitores** y los **semáforos**.

Veremos el **problema** del **productor/consumidor**.

Sincronizaremos métodos de forma nativa **en Java**.

Problemas asociados a la sincronización

Cuando los hilos intentan acceder a los recursos que comparten, se darán **problemas**:

- **Condición de carrera**: El resultado de ejecutar el programa dependerá del **orden** en que se realicen los accesos a los recursos.
- **Inconsistencia de memoria**: Diferentes **hilos** que tienen una **visión diferente** de un mismo dato.
- **Inanición**: a un **hilo** se le **denega continuamente** el acceso a un recurso compartido al que quiere tener acceso, porque otros hilos toman el control antes que él. Es complicado de detectar.
- **Interbloqueo (Deadlock)**: Dos o más **hilos** estén **esperando** que suceda un **evento** que solo puede generar un **hilo** que se encuentra **bloqueado**.
- **Bloqueo activo**: Tenemos **dos hilos** que están cambiando continuamente de estado y terminan por **bloquearse mutuamente**. Es un tipo de inanición, porque un proceso no deja avanzar al otro, y viceversa.

Exclusión mutua

*Es una **forma de resolver los problemas** asociados a la sincronización. Consiste en que únicamente se va a permitir **acceder a recursos compartidos** a **un solo proceso**, excluyendo temporalmente a todos los demás, de forma que **garantice la integridad** del sistema.*

Monitores

Es una de varias opciones para **evitar gran parte de los problemas en la sincronización**.

Se encargan de **gestionar el acceso a los recursos** compartidos o críticos.

Sección crítica o **zona de exclusión mutua:** Fragmento de **código** que **engloba a estos recursos**. Es una **región** de código a la que **se accede de forma ordenada** a los recursos compartidos. Tiene que ser forzosamente **excluyente para los hilos**: si un hilo se encuentra ejecutando su fragmento de código de sección crítica en un momento dado, ***ningún otro hilo*** podrá **entrar** a esa zona hasta que este no finalice. El lenguaje de programación Java nos proporciona el **modificador synchronized**, que, cuando lo **aplicamos a un método**, nos va a garantizar que este se va a **ejecutar de forma excluyente**.

A una **clase** que tiene un **método con el modificador synchronized**, la llamaremos **monitor**, debido a que, dentro de este método, se va a estar **monitoreando algún recurso crítico**.

Tenemos que tener muy claro que los monitores son los encargados de implementar las secciones críticas. La **implementación** de monitores debe **tener forzosamente mecanismos** que nos permitan llevar a cabo la **sincronización** y que **actúen antes y después de entrar** en la sección crítica. Estos fragmentos de código son los que vamos a tener dentro de la palabra reservada synchronized.

Para poder programar los monitores en Java vamos a utilizar, **además** de synchronized, **los métodos**:

- **wait()**: Si no se cumple la condición de exclusión, tendremos que **esperar**.
- **notify()**: Cuando se **concluya la tarea** que se encuentra en la sección crítica, podamos **salir** de ella **avisando al hilo** que se encuentre **esperando** para que pueda entrar.
- **notifyAll()**: Igual que notify() pero se **notificará** a **todos los hilos** que están **esperando**.

Clase Monitor:

```
package monitor;
public class ClaseMonitorSynchronizer {
	private int valor = 0;
	public synchronized void incrementar() {
		valor++;
	}
	public synchronized void decrementar() {
		valor--;
	}
	public synchronized int getValor() {
		return valor;
	}
}
```

Prácticamente todos los lenguajes de programación, para que no tengamos la opción de crear y ejecutar la concurrencia de hilos. La idea será la misma: **bloquear** una cantidad de **hilos** para que vayan **accediendo**, **de uno en uno**, a las **zonas críticas**, por lo que habrá que tener en consideración **cómo se utilizarán estos mecanismos**, no cuáles son.

Ejemplos de lenguajes de programación que no son Java y que ofrecen mecanismos de sincronización de hilos pueden ser **C, C++ y Python**.

Problema del
Productor Consumidor

Consiste en tener **dos «agentes»** que **comparten un almacén o buffer** con un tamaño limitado:

Productor: Coloca o **produce información** en el buffer.

Consumidor: E**xtrae la información** para realizar una operación con ella.

Almacén lleno: el productor deberá «**dormirse**». El **consumidor** «**despertará**» **al productor** cuando haya **extraído** algún elemento del almacén, ya que será entonces cuando el productor pueda producir otro elemento y colocarlo en este.

Almacén vacío: si el **consumidor** necesita **extraer** un elemento, debe «**dormirse**» hasta que el **productor** coloque **elementos** en el almacén, «**despertando**» en ese momento al **consumidor**.

Para solucionar el problema del productor/consumidor, necesitaremos tener **dos hilos**, que cubrirán el papel de **agentes**:

Productor, encargado de **generar** elementos y de guardarlos,

Consumidor, encargado de **obtenerlos**.

En este problema pueden ocurrir variedad de situaciones:

- Tanto el productor como el consumidor quieren **acceder a la vez** al almacén.
- El productor genera a **distinta velocidad** que el consumidor consume.
- El productor ha **llenado el almacén** y **no puede producir más** hasta que el consumidor consuma algo.
- El **consumidor no puede consumir** nada porque el **almacén está vacío**.

Este problema se puede extrapolar a tener un productor y varios consumidores, y varios productores y consumidores.

Productor (genera información) → datos → Consumidor (aprovecha información)

La forma más eficiente y sencilla de resolver el problema del productor es mediante **monitores**. Para ello, necesitaremos las siguientes **clases**:

- **Productor**: Esta será la clase que se encargará de **producir** los elementos que se guardarán en el almacén y que el consumidor obtendrá.
- **Consumidor**: Esta **consumirá** los elementos que se guardarán en el almacén y que el productor creará.
- **Buffer**: Será la encargada de **almacenar** los elementos que producirá el productor y que, más adelante, consumirá el consumidor. Dentro de esta clase, vamos a tener dos métodos (aunque pude teenr más):
  - Método **put**: Este método será el encargado de **introducir un elemento dentro del buffer**. Deberá estar **sincronizado** para que tanto productor como consumidor **no accedan al mismo tiempo**.
  - Método **get**: Este método será el encargado de **obtener un elemento** que se encuentre dentro del buffer. Igualmente, deberá estar **sincronizado** para que tanto productor como consumidor **no accedan al mismo tiempo**.

Ejemplo Productor Consumidor sin monitor

Esta clase representa el almacén, es decir, el recurso que se produce y se consume (Versión **no sincronizada**):

```
public class Buffer
{
    private int contenido;
    /**
     * Obtiene el contenido del buffer
     * @return Contenido del buffer
     */
    /**
     * Notamos como estos métodos a continuación no están synchronized:
     */
    public int get()
    {
        return contenido;
    }

    /**
     * Inserta un valor dentro del buffer
     * @param value Valor para insertar
     */
    public void put(int value)
    {
        contenido = value;
    }
}
```

Esta clase representa el productor:

```
public class Productor extends Thread
{
    private Buffer almacen;
    private int dormir;
    /**
     * Constructor del productor
     * @param almacen Buffer donde se producirán los recursos
     * @param dormir Tiempo que dormirá el productor
     */
    public Productor(Buffer almacen, int dormir)
    {
        this.almacen = almacen;
        this.dormir = dormir;
    }
```

```
    public void run()
    {
        for (int i = 0; i < 10; i++)
        {
            almacen.put(i);
            System.out.println("Productor pone: " + i);
            try
            {
                sleep(dormir);
            }
            catch (InterruptedException e)
            {
                System.err.println("Error en el productor: " + e.toString());
            }
        }
    }
}
```

Esta clase representa al consumidor:

```
public class Consumidor extends Thread
{
    private Buffer almacen;
    private int dormir;
    /**
     * Constructor del consumidor
     * @param almacen Buffer de donde se obtendrán los recursos
     * @param dormir Tiempo que dormirá el consumidor
     */
    public Consumidor(Buffer almacen, int dormir)
    {
        this.almacen = almacen;
        this.dormir = dormir;
    }
    public void run()
    {
        int valor = 0;
        for (int i = 0; i < 10; i++)
        {
            valor = almacen.get();
            System.out.println("Consumidor saca: "+ valor);
            try
            {
                sleep(dormir);
```

```
                }
                catch (InterruptedException e)
                {
                    System.err.println("Error en el consumidor: " + e.toString());
                }
            }
        }
    }
```

Esta es la clase pricncipal con el método principal que ejecutará los hilos:

```
    public static void main(String[] args)
    {
        final int DORMIR_PRODUCTOR = 1000, DORMIR_CONSUMIDOR = 2000;

        Buffer almacen = new Buffer();
        Productor productor = new Productor(almacen, DORMIR_PRODUCTOR);
        Consumidor consumidor = new Consumidor(almacen, DORMIR_CONSUMIDOR);
        productor.start();
        consumidor.start();
    }
```

El resultado es el siguientye:

```
Productor pone: 0
Consumidor saca: 0
Productor pone: 1
Consumidor saca: 1
Productor pone: 2
Productor pone: 3
Consumidor saca: 3
Productor pone: 4
Productor pone: 5
Consumidor saca: 5
Productor pone: 6
Productor pone: 7
Consumidor saca: 7
Productor pone: 8
Productor pone: 9
Consumidor saca: 9
Consumidor saca: 9
Consumidor saca: 9
Consumidor saca: 9
Consumidor saca: 9
```

El funcionamiento de este problema será muy sencillo:

- Se **lanzarán** tanto productor como consumidor.
- Si el **buffer** está **vacío**, el **consumidor no** podrá **obtener** ningún valor hasta el que el productor cree y almacene uno. En este caso, será cuando el **productor se active** y coloque un elemento dentro del almacén.
- Si el **buffer** está lleno, el **productor no** podrá **introducir** ningún valor hasta que el consumidor obtenga y consuma el valor que hay dentro del buffer. En este caso, será cuando el **consumidor se active** y obtenga el valor que se encuentre dentro del almacén.

Ejemplo Productor Consumidor con monitor

Este ejemplo es idéntico al anterior, **solo cambia la clase Buffer (el monitor de la zona crítica)**, cuyos métodos ahora sí están sincronizados:

Esta clase representa el almacén, es decir, el recurso que se produce y se consume (Versión **sincronizada**):

```
public class Buffer {
    private int contenido;
    private boolean disponible = false;

    /**
     * Obtiene el contenido del buffer
     * @return Contenido del buffer
     */
    public synchronized int get()
    {
        // Mientras el buffer no esté disponible
        while (disponible == false)
        {
            try
            {
                // me espero a que produzcan
                wait();
            }
            catch (InterruptedException e) {}
        }
        // Cuando vuelve a estar disponible, notifico que está disponible
        disponible = false;
        notify();
        return contenido;
    }
```

```java
    /**
     * Inserta un valor dentro del buffer
     * @param value Valor para insertar
     */
    public synchronized void put(int value)
    {
        // Mientras el buffer esté disponible
        while (disponible == true)
        {
            try
            {
                // me espero a que consuman
                wait();
            }
            catch (InterruptedException e) {}
        }
        // Cuando vuelve a estar disponible, notifico que está disponible
        contenido = value;
        disponible = true;
        notify();
    }
}
```

Semáforos

Un semáforo es **otro de los mecanismos** que nos va a proporcionar el lenguaje de programación Java para solucionar problemas de sincronización de varios hilos.

Semáforo binario: Indicador de condición de entrada que **gestiona** si un **recurso** de la sección crítica de nuestro código está **disponible o no**. Es **binario** porque va a tener **dos posibles valores**, 'disponible' o 'no disponible'.

En Java, podremos utilizar la clase **java.util.concurrent.Semaphore**, la cual ya está integrada en la JDK de Java y es totalmente funcional (*https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html).*

En los problemas habrá una **sección crítica** que necesitamos proteger, la cual puede consistir en una o varias **variables compartidas** que los **hilos van a necesitar**. Los semáforos sincronizan todos esos hilos para que **no se produzca inanición** ni **accesos indebidos**.

Tienen **dos métodos**:

- Método **acquire()** (adquirir): **Cerrar la sección crítica** y que **ningún** otro **hilo** pueda **acceder** a ella.
- Método **release()**: **Abrir la sección crítica** y que otro **hilo** pueda **acceder** a ella.

Podremos **indicarle el número de hilos** que podrán entrar **a la vez** en la sección crítica. En caso de **no indicar nada**, será un semáforo binario y únicamente podrá acceder a la sección crítica **un hilo**.

Ejemplo de semáforo en Java:

```
public static void main(String[] args) {
		Semaphore semaphore = new Semaphore(1, true);
		try {
			// Protejo la sección crítica
			semaphore.acquire();
		} catch (InterruptedException ex) {
			System.out.println("Error: " + ex.toString());
		}

	}
```

Otros Problemas clásicos de sincronización

• **Problema de los fumadores**: **Tres hilos**, que serán los «**fumadores**», y **un hilo**, al que podremos considerar el «**estanquero**», que dará material a los fumadores. Cada fumador está continuamente deseando fumar un cigarrillo. Sin embargo, para fumar, necesita **tres ingredientes**: tabaco, papel y fósforos. Cada uno de los tres **fumadores tendrá un material** de los necesarios, por lo que necesitan que el **estanquero produzca los dos** que les faltan. Cuando un **fumador consigue** los ingredientes que le faltan, **fuma**, **avisando** al **productor** cuando termina, quien **coloca otros dos** de los tres ingredientes, repitiéndose el ciclo.

• **Los filósofos**: **Cinco filósofos** que se pasan la vida **pensando y comiendo**. Los filósofos están sentados en una mesa circular. Delante de **cada filósofo** hay **un plato**, y hay cinco platos en total. Cuando un filósofo está **pensando**, **no** necesita **comer**, y cuando **come**, **no piensa**. Cuando a un filósofo le da hambre y quiere comer, **necesitará 2 cubiertos**, por tanto, tratará de coger los dos cubiertos más cercanos a él, pero solo podrá usar cada cubierto **si no lo tiene el filósofo contiguo**. Cada cubierto, por tanto, es compartido por 2 filósofos. Cuando **termina de comer**, vuelve a dejar sus dos cubiertos en la mesa y comienza a **pensar** de nuevo.

• **El barbero**: Peluquería que regenta **un barbero que tiene una silla de peluquero y X sillas para que se sienten los clientes que están en espera. Si no** hay clientes, el barbero se sentará en su silla y **dormirá**, pero, cuando **llega un cliente** a la peluquería, el barbero se **despierta** y lo **atiende**. En el caso de que lleguen más clientes y el barbero esté ocupado, estos deberán **esperar sentados**, siempre que haya sillas disponibles, **o salirse** de la peluquería, si no las hay.

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONESMULTIPLATAFORMA

Introducción a la comunicación entre aplicaciones

Modelo OSI

Después del surgimiento de Internet, sobre el año **1980**, las redes comenzaron a crecer de forma exponencial, ya que muchas empresas lo consideraron una gran oportunidad para expandir sus negocios. Como las redes estaban empezando a surgir, cada una de estas contaban con sus **propias especificaciones** para funcionar, haciendo que la **comunicación** entre todas las redes fuese una **tarea prácticamente imposible**.

Debido a esto, la **Organización Internacional para la Estandarización** (**ISO**) empezó a estudiar la forma de arreglar este problema, investigando cómo podían hacer que las redes se comunicaran entre sí. Todo esto llevó a una serie de reglas, que se estandarizarían bajo el nombre de **modelo OSI**, las cuales dictaban cómo debían **funcionar las redes** y cómo debían ser sus **comunicaciones**, haciendo que la interconexión de ellas fuera un paso trivial, ya que **todas trabajaban de la misma forma**.

El **modelo OSI** está formado por **siete capas** que dividen todo el funcionamiento y los estados por los que deben pasar los datos para viajar de una red a otra.

Las **capas del modelo OSI son**:

- **Capa física**: Esta capa será la encargada de manejar la **topología de la red** y las **conexiones del ordenador**, por ejemplo. Podemos decir que gestiona todo el **hardware** necesario.
- **Capa de enlace de datos**: Será la encargada de realizar el **direccionamiento físico** de los datos que se envíen. Detectará errores, controlará el flujo y hará que los paquetes lleguen ordenados.
- **Capa de red**: Esta capa **enrutará las redes**. Su objetivo principal es **hacer que** los datos **lleguen** desde su origen a su destino.
- **Capa de transporte**: Como su nombre indica, llevará a cabo el transporte de los datos.
- **Capa de sesión**: Su cometido pasa por **mantener la conexión** entre dos equipos, reanudándola en caso de interrupción.
- **Capa de presentación**: Será la responsable de la **presentación de la información**. Aquí se tratan temas como **semántica y sintaxis** de los **paquetes** que se transmiten.
- **Capa de aplicación**: Encargada de **acceder a los servicios** que proveen las demás capas. Aquí es donde se definen los **protocolos que usaremos**.

El modelo TCP/IP

Cualquier **red está separada** en **capas**, conocidas también como **layers**, en inglés. La comunicación entre dichas capas estará **controlada por protocolos**, los cuales indicarán **cómo tienen que ser**, de qué datos tienen que constar, cómo deberán enviar dichos datos, cómo deberán recibirlos, etc.

El **protocolo más común** es el dictado por el **modelo TCP/IP**, el cual consta de las **siguientes capas**:

- **Capa de aplicación**: Esta capa está compuesta por **aplicaciones de red**, las cuales usarán los niveles más inferiores para poder **transferir mensajes** entre ellas mismas. Algunos ejemplos de **protocolos** que trabajan en esta capa son:
  - **HTTP**: Este protocolo es el encargado de definir la manera en la que se van a **comunicar** los **servidores y los navegadores web**.
  - **SMTP**: Este protocolo es el encargado de definir la manera en la que se gestiona el **correo** electrónico.
  - **DNS**: Es el que **traduce a direcciones IP** los **nombres de los dispositivos** que se encuentra en la red.
  - **FTP**: Posibilita las **transferencias de ficheros**.
  - **NFS (Network File System)**: Permitirá que podamos **compartir ficheros** en diferentes **ordenadores de una red**.
  - **TELNET**: Posibilitará la **conexión remota** de **terminales**.
- **Capa de transporte**: Esta capa está compuesta por todos aquellos **elementos software** cuya función es crear el **canal de comunicación**, **descomponer** el **mensaje** que hayamos enviado en diferentes **paquetes** y gestionar la transmisión del mismo entre el **emisor y el receptor**. Aquí es donde actuarán los **protocolos TCP y UDP**.
- **Capa de Internet**: Esta capa está compuesta por todos aquellos **elementos software** encargados de **dirigir los paquetes** por la red; además, se asegurarán de que dichos paquetes **lleguen a su destino**.
- **Capa de red**: La forman todos aquellos **elementos hardware** de comunicaciones, tarjetas de red, cables, etc., y es la encargada de **trasmitir todos los paquetes** de información. Debemos tener en cuenta que los **protocolos** tienen que **conocer los detalles físicos** de la **red** para un correcto envío de los paquetes.

No confundir OSI con TCP/IP

Cada uno de ellos trabaja de una forma diferente al otro, realizando tareas **diferentes en cada** una de sus **capa**s.

Además, el **modelo TCP/IP** suele confundirse con el protocolo **TCP**, que en realidad es un **protocolo de transporte** que pertenece a ambos modelos.

El **modelo TCP/IP** **tiene 4 capas** mientras que el modelo OSI tiene 7 capas:

- La capa **Aplicación** corresponde a Aplicación, Presentacion y Sesión del modelo OSI.
- La capa de **Transporte** es igual a la del modelo OSI.
- La capa de **Red** corresponderá a las capas Fisica, Enlace de datos y Red del modelo OSI.

Cómo se identifican las aplicaciones en diferentes PC

Las **aplicaciones** las vamos a tener en **diferentes ordenadores** que están **conectados en red**, seguramente una red local, y cada uno de estos ordenadores se va a **identificar de forma única** en dicha red, mediante su **dirección IP**.

A través de las diferentes **direcciones IP** de los ordenadores, podremos **realizar las comunicaciones**, ya que, cuando enviemos un mensaje a otra aplicación, lo tendremos que enviar a la dirección IP del ordenador donde se encuentra, y, de la misma forma, cuando nos envíen a nosotros un mensaje, lo deberán hacer a la dirección IP que tenga nuestro ordenador.

Protocolos de comunicaciones

Protocolo TCP

Ya hemos visto que en la **capa de transporte** del modelo TCP/IP y en la capa de transporte del modelo OSI, es donde se realiza todo lo relacionado con la **transferencia de datos** y **corrección de errores** de estos, entre el **emisor** y el **receptor** de la comunicación.

Como seguro que habéis adivinado, la misión es proporcionar un **transporte de información confiable** entre el emisor y el receptor, que sea totalmente independiente de la capa física que se esté utilizando en la misma.

Esto puede realizarse, por ejemplo, mediante el protocolo TCP (Protocolo de Control de Transmisión o **Transmission Control Protocol** en inglés), que es un protocolo orientado a conexión que **creará un flujo de transmisión** de datos entre el origen y el destino, que garantizará que la información se entregue sin errores. Este protocolo **parte el mensaje** que se quiere enviar **en paquetes**, enviándolos por el **canal de comunicación**.

A estos **paquetes** les irá asignando un **número** para que, una vez lleguen a su destino, puedan ser **reconstruidos**, volviendo a tener de una pieza el mensaje que se envió.

Otra tarea del protocolo TCP será **controlar el flujo del canal** de comunicación. Con esto, controlará si hay más o menos tráfico, haciendo que la **red no se sature** y evitando que un receptor que sea lento en el proceso de recepción de paquetes quede saturado por un emisor que sea muy rápido enviando los mensajes.

Este protocolo es fiable, lo cual va a hacer que se **garantice la llegada de los paquetes** al receptor.

Al contar con todas estas características, el protocolo TCP **no es un protocolo sencillo de implementar**, ya que tiene que **cubrir muchos aspectos en el transporte** de la información.

Ejemplos de **protocolos que usan TCP** son **HTTP**, **FTP**, **Telnet**, etc.

Protocolo UDP

Además del protocolo TCP, contamos con otro protocolo llamado **UDP** (Protocolo de Datagramas de Usuario o **User Datagram Protocol** en inglés), el cual es un protocolo que **no está orientado a conexión**. Esto implica que **no** va a tener ningún tipo de **sincronización para el envío** de mensajes entre el **emisor** y el **receptor**.

Este protocolo se utiliza, principalmente, para **aplicaciones que no van a necesitar asignación de control** de **secuencia** ni de control de **flujo** en las transmisiones que se hagan.

Su funcionamiento se basa en la **partición del mensaje** que se quiere enviar en **datagramas**, enviándolos por el canal de comunicación, **sin control** ninguno, pudiendo **llegar o no** al destinatario. Al contrario de lo que pasaba con el protocolo TCP, en **UDP no hay una numeración** de cada datagrama para que más adelante se puedan unir y reconstruir el mensaje original.

Este protocolo se va a utilizar en **aplicaciones en las que prime más la velocidad** de entrega de paquetes, las conocidas como **aplicaciones en tiempo real**, como pueden ser aplicaciones de **streaming** o transmisión de **voz**.

Este protocolo **no es fiable**, ya que no garantiza la llegada de todos y cada uno de los datagramas enviados por el emisor del mensaje al receptor.

Además, **no realiza ningún tipo de control de flujo** en las comunicaciones.

Al no realizar ni control de flujo ni de errores, el protocolo UDP es **mucho más rápido que TCP** en la transmisión de los datos de la comunicación. Además, esto también implica que será **mucho menos complejo de implementar**.

Este protocolo se encuentra en la **capa de transporte**.

El protocolo UDP es parecido a un río, caudaloso, veloz, pero no fiable.

Ejemplos de protocolos que usan UDP son **DHCP**, **BOOTP**, **DNS**, etc.

Los puertos

Ya hemos visto que, para lograr que la **información vaya de un extremo a otro** de la comunicación, deberemos utilizar la **capa de red** y enviar esos paquetes o datagramas a la dirección IP del destinatario.

Ciertamente, el proceso no es tan sencillo, ya que tendremos que **indicar a dónde van dirigidos** dichos paquetes.

Para esto, podemos definir una serie de **direcciones de transporte** en las que los **procesos de nuestras aplicaciones** puedan estar de forma constante **a la escucha**, por si llega algún paquete o datagrama. Esto es lo que conocemos como **puertos**.

La gran mayoría de puertos se va a **asignar de forma aleatoria**, sin ningún tipo de orden, simplemente se asignará el **primero que se encuentre libre**.

No obstante, hay ciertas **aplicaciones** que sí tienen **un puerto ya asignado** para su funcionamiento, los cuales **no podrán ser usados** en las asignaciones aleatorias o en asignaciones manuales que hagamos nosotros, ya que eso **llevaría a diversos errores** en la **transmisión** de la información a través de la red.

Quien se encarga de **asignar los puertos predefinidos** a las **aplicaciones que así lo necesiten** es la **Autoridad de Asignación de números de Internet** o IANA, por sus siglas (**Internet Assigned Numbers Authority)**.

La **IANA** tiene definidos los siguientes **rangos de puertos**:

- **Puertos conocidos**: Estos puertos son los que están reservados para aplicaciones estándar y van desde el **puerto 0 al 1023**. Algunos ejemplos de estos son el **puerto 21** para el protocolo **FTP**, el puerto **80** para el protocolo **HTTP**, etc.
- **Puertos que están registrados**: Estos puertos han sido asignados para servicios o aplicaciones específicas y van desde el **puerto 1024 al 49151**. Este será el rango de puertos que deberemos utilizar para desarrollar **nuestras aplicaciones**.
- **Puertos dinámicos**: Estos puertos no están registrados para ningún servicio, sino que su uso es para atender a **conexiones temporales** entre diferentes aplicaciones y van desde el **puerto 49151 al 65535**.

El esquema cliente/servidor

Cuando desarrollamos aplicaciones que hacen **uso de las comunicaciones** en red, no podemos hacerlo sin ningún tipo de orden y enviar los mensajes sin más: debemos **seguir un protocolo estándar** para este tipo de aplicaciones.

Estas aplicaciones se van a desarrollar mediante lo que conocemos como el **protocolo cliente/servidor**. Este protocolo es muy sencillo de comprender, ya que vamos a tener dos ordenadores diferentes: uno hará de servidor y otro de cliente. Básicamente, el **servidor** será el que **provea de servicios** a los ordenadores **que se los piden**, los cuales serán los **clientes**.

Una vez comprendida esta idea, **cualquier aplicación** que use las **comunicaciones** en red estará **dividida en estas dos partes**: un **servidor** que **escucha peticiones y provee** de servicios y **uno o varios clientes** que se conectarán al servidor para **pedir** un servicio.

Bibliotecas para networking en Java

El lenguaje de programación Java nos ofrece las siguientes **clases y bibliotecas** para poder desarrollar **aplicaciones con comunicaciones en red**:

• **InetAddress**: Esta clase nos va a permitir **encontrar un nombre de dominio a partir de su dirección IP**, y viceversa. Los objetos de esta clase tendrán **dos elementos**: el **nombre del equipo** y la **dirección IP**.

• **Socket**: Esta clase realiza o implementa la **comunicación bidireccional** entre un programa cliente y otro programa servidor, es decir, va a permitir tanto el **envío** como la **recepción** de mensajes. Si usamos objetos de esta clase, nuestros programas Java podrán comunicarse a través de la red de forma **independiente de la plataforma**. Esta clase se utilizará para **clientes TCP**.

• **ServerSocket**: Nos ayudará a **implementar un socket** que puede ser utilizado por los **servidores** para **escuchar y aceptar peticiones** de conexión de clientes. Esta clase se utilizará para **servidores TCP y UDP**.

• **DatagramSocket**: Con ella podremos **implementar clientes** que utilicen datagramas, siendo no fiables y no ordenados. Esta clase ofrece una **comunicación muy rápida**, ya que **no hay que establecer la conexión** entre cliente y servidor. Esta clase se utilizará para **clientes UDP**.

• **DatagramPacket**: Representará **un datagrama**, y contiene toda la información necesaria por el mismo: **longitud de paquete**, **direcciones IP** y **número de puerto**.

• **MulticastSocket**: Utilizada para crear una versión '**multicast**' (multi-lanzamiento en español) de la clase **DatagramSocket**. Mediante esta clase podremos enviar **mensajes a múltiples clientes o servidores**.

Clases para networking en Java		
ServerSocket	Nos va a permitir crear un servidor.	new ServerSocket(PUERTO) accept() writeUTF(mensaje) readUTF() close()
Socket	Nos va a permitir crear un cliente TCP.	new Socket(HOST , PUERTO) getOutputStream() getInputStream() writeUTF(mensaje) readUTF() close()
DatagramSocket / DatagramPacket	Nos va a permitir crear un cliente UDP.	new DatagramSocket(PUERTO) (Socket) receive() (Packet) getAddress() (Packet) getPort() (Packet) send() (Socket) receive() (Socket) close() (Socket)

Comunicación TCP y UDP: