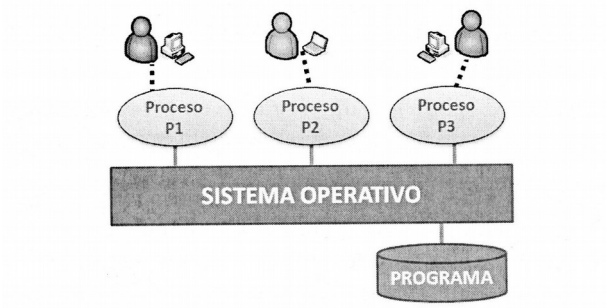
2.- Programación concurrente

Podemos definir la concurrencia como la existencia simultánea de varios procesos en ejecución

2.1.- Programa y proceso

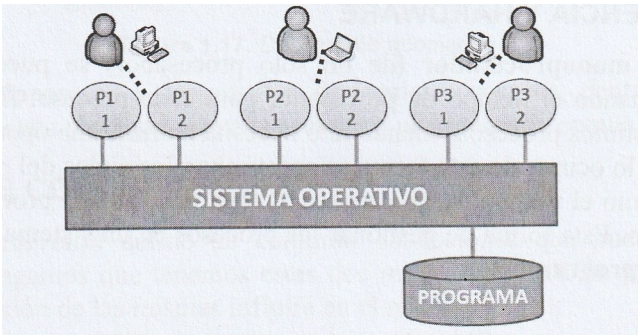
Podemos definir programa como un conjunto de instrucciones que se aplican a un conjunto de datos de entrada para obtener una salida. Un proceso es algo activo que cuenta con una serie de recursos asociados, en cambio un programa es algo pasivo. Un programa al ponerse en ejecución puede dar lugar a más de un proceso , cada uno ejecutando una parte del programa. Ejemplo, el navegador web, por un lado está controlando las acciones del usuario con la interfaz, por otro hace las peticiones al servidor web . Por tanto, cada vez que se ejecuta este programa crea 2 procesos.



En la figura observamos que tenemos un programa almacenado en disco y 3 instancias del mismo ejecutándose, por ejemplo, por 3 usuarios diferentes. Cada instancia del programa es un proceso, por tanto, existen 3 procesos independientes ejecutándose al mismo tiempo sobre el sistema operativo, tenemos 3 procesos concurrentes.

Decimos que dos procesos son concurrentes cuando la primera instrucción de uno de ellos se ejecuta después de la primera instrucción del otro y antes de la última. Es decir , existe un solapamiento o intercalado en la ejecución de sus instrucciones. No hay que confundir el solapamiento con la ejecución simultánea de las instrucciones, en este caso estaríamos en una situación de programación paralela, aunque a veces el hardware subyacente (más de un procesador) si permitirá la ejecución simultánea.

Supongamos ahora que el programa anterior al ejecutarse da lugar a 2 procesos más ,cada uno ejecutando una parte del programa, entonces La figura anterior se convierte en la siguiente figura:



Ya que un programa puede estar compuesto por diversos procesos, una definición más acertada de proceso es la de una actividad asíncrona susceptible de ser asignada a un procesador.

Cuando varios procesos se ejecutan concurrentemente puede haber procesos que colaboren para un determinado fin (por ejemplo, P1.1 y P1.2), y otros que compitan por los recursos del sistema (por ejemplo P2.1 y P3.1) . Estas tareas de colaboración y competencia por los recursos exigen mecanismos de comunicación y sincronización entre procesos.

2.1.1.-Características de la programación concurrente.

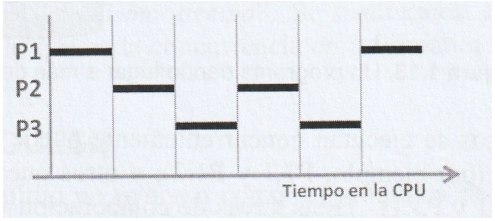
La programación concurrente es la disciplina que se encarga del estudio de las notaciones que permiten especificar la ejecución concurrente de las acciones de un programa, así como las técnicas para resolver los problemas inherentes a la ejecución concurrente (comunicación y sincronización).

2.1.2.- Beneficios de la programación concurrentes

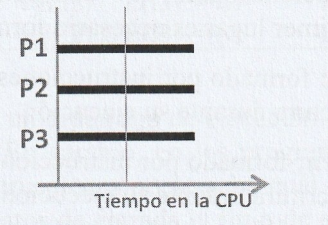
- **Mejor aprovechamiento de la CPU** (Un proceso puede aprovechar ciclos de CPU mientras otro realiza operaciones de entrada/salida)
- **Velocidad de ejecución** (Al subdividir un programa en procesos, éstos se pueden "repartir" entre procesadores o gestionar en un único procesador según importancia.
- **Solución a problemas de naturaleza concurrentes**.
  - Sistemas de control: son sistemas donde hay que capturar datos a través de sensores , análisis y actuación en función del análisis (sistemas de tiempo real)
  - Tecnologías web: los servidores web son capaces de atender múltiples peticiones de usuarios concurrentemente.
  - Aplicaciones basadas en GUI (Interfaz Gráfica de Usuario): el usuario puede interactuar con la aplicación mientras la aplicación está realizando otra tarea.
  - Simulación
  - Sistemas Gestores de Bases de Datos.

2.1.3.- Concurrencia y Hardware

En un sistema de un solo procesador se puede tener una ejecución concurrente gestionando el tiempo de procesador para cada proceso. El S.O. va alternando el tiempo entre los distintos procesos.



En un sistema multiprocesador podemos tener un proceso en cada procesador . Esto permite que exista paralelismo real entre los procesos.



2.1.4.- Programas concurrentes

Un programa concurrente define un conjunto de acciones que pueden ser ejecutadas simultáneamente.

x=x+1; y=x+1;	La primera instrucción se debe ejecutar antes de la segunda.

Dada estas dos instrucciones de un programa, podemos apreciar que el orden de la ejecución influye en el resultado final.

En cambio , si tenemos estas otras , el orden de ejecución es indiferente.

x=1;
y=2;
z=3;
	El orden no interviene en el resultado final.

2.1.5.- Problemas inherentes a la programación concurrente.

Nos podemos encontrar con dos problemas a la hora de crear un programa concurrente.

- **Exclusión mutua.** En programación concurrente es muy típico que varios procesos accedan a la vez a una variable compartida para actualizarla. Esto se debe evitar , ya que puede producir inconsistencia de datos: uno puede estar actualizando la variable a la vez que otro la puede estar leyendo. Para conseguir la exclusión mutua de los procesos respecto a la variable compartida se propuso la "región crítica" . Cuando dos o más procesos comparten una variable , el acceso a dicha variable debe efectuarse siempre dentro de la región crítica asociada a la variable . Solo uno de los procesos podrá acceder para actualizarla y los demás deberán esperar.
- **Condiciones de sincronización**. Hace referencia a la necesidad de coordinar los procesos con el fin de sincronizar sus actividades. Puede ocurrir que un proceso P1 llegue a un estado X que no pueda continuar su ejecución hasta que otro proceso P2 haya llegado a un estado Y de su ejecución . La programación concurrente proporciona mecanismos para bloquear procesos a la espera de que ocurra un evento y para desbloquearlos cuando este ocurra.

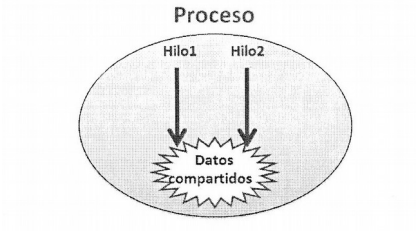
Algunas herramientas para manejar la concurrencia son : la región crítica, los semáforos , región crítica condicional , buzones , sucesos, monitores y sincronización por rendez-vous.

2.2. Programación concurrente con Java

Al igual que el sistema operativo puede ejecutar varios procesos concurrentemente, dentro de un proceso podemos encontrarnos con varios hilos en ejecución.

2.2.1.- Conceptos básicos

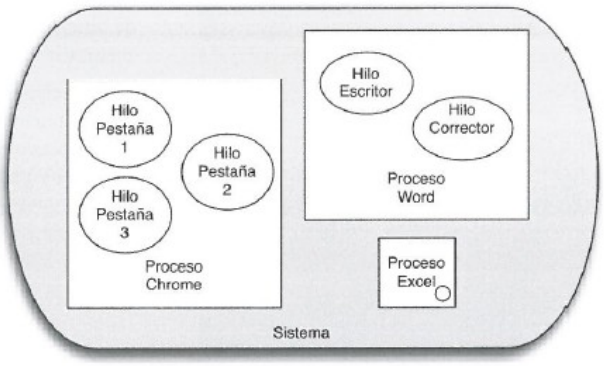
Hilo : los hilos o threads, son la unidad básica de utilización de la CPU , y más concretamente de un core del procesador . Así un thread se puede definir como la secuencia de código que está en ejecución, pero dentro del contexto de un proceso y que ejecuta sus instrucciones de forma independiente.



Hilo o Procesos: El sistema operativo gestiona procesos, asignándoles la memoria y recursos que necesiten para su ejecución. En este sentido , el sistema operativo planifica únicamente procesos.

Los hilos se ejecutan dentro del contexto de un proceso, por lo que dependen de un proceso para ejecutarse. Mientras que los procesos son independientes y tienen espacios de memoria diferentes, dentro de un mismo proceso pueden coexistir varios hilos ejecutándose que compartirán la memoria de dicho proceso . Esto sirve para que el mismo programa en ejecución (proceso) pueda realizar diferentes tareas (hilos) al mismo tiempo. Un proceso siempre tendrá por lo menos , un hilo en ejecución que es el encargado de la ejecución del proceso.

Ejemplo:
Gracias a los hilos podemos tener diferentes pestañas abiertas en el navegador, cada una cargando a la vez una página web diferente, o Microsoft Word puede tener un hilo comprobando automáticamente la gramática, a la vez que se está escribiendo un documento.

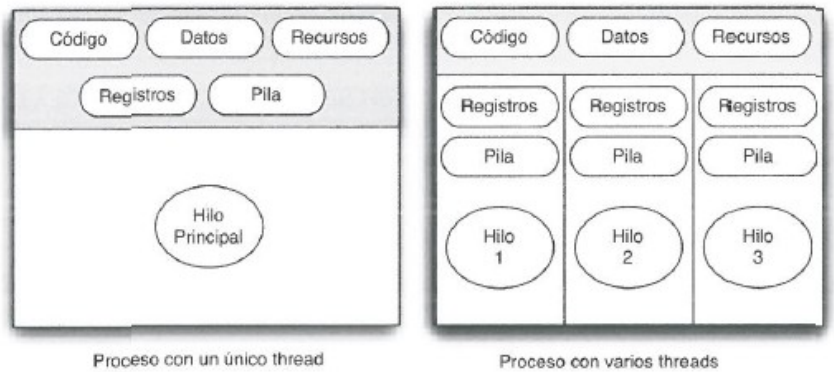


Ventajas de la multitarea frente a la multiprogramación de procesos

- **Capacidad de respuesta**. Los hilos permiten a los procesos continuar atendiendo peticiones del usuario aunque alguna de las tareas que esté realizando el programa sea muy larga. Este modelo se utiliza mayoritariamente en la programación de un servicio en servidores. Un hilo se encarga de recibir todas las peticiones del usuario y por cada petición se lanza un nuevo hilo para responderla. De esta forma se están tratando varias peticiones al mismo tiempo.
- **Compartición de recursos.** Por defecto, los threads comparten la memoria y todos los recursos del proceso al que pertenecen. No necesitan ningún medio adicional para comunicarse información entre ellos ya que todos pueden ver la información que hay en la memoria del proceso. Sin embargo, debido a que todos los hilos pueden acceder y modificar los datos al mismo tiempo, se necesitan medios de sincronización adicionales para evitar problemas en el acceso.
- ***Como los hilos utilizan la misma memoria del proceso del cual depende , la creación de nuevos hilos no supone ninguna reserva adicional de memoria por parte del sistema operativo***. Es más barato en términos de uso de memoria y otros recursos crear nuevos threads que crear nuevos procesos.
- **Paralelismo real**. La utilización de threads permite aprovechar la existencia de más de un núcleo en el sistema en arquitecturas multicore. Sabemos que el sistema operativo planifica procesos de forma concurrente , intercambiando los procesos uno por otro mediante un cambio de contexto. En este sentido , solamente puede haber un proceso en ejecución en un momento dato independientemente del número de núcleos del procesador ya que solo existe una memoria (cuando se desea acceder a un dato se realiza mediante la traducción de las referencias a direcciones de memoria que se encuentran en el código del proceso a las direcciones efectivas en memoria principal). Es decir , la gestión de procesos no puede aprovecharse de la existencia de varios núcleos. Sin embargo , varios hilos pueden ejecutarse en el contexto de un mismo proceso. Cuando ese proceso está en ejecución , los hilos programados del mismo pueden utilizar todos los núcleos del procesador de forma paralela, permitiendo que se ejecuten varias instrucciones (una por cada thread y núcleo) a la vez.

2.2.2.- Recursos compartidos por hilos

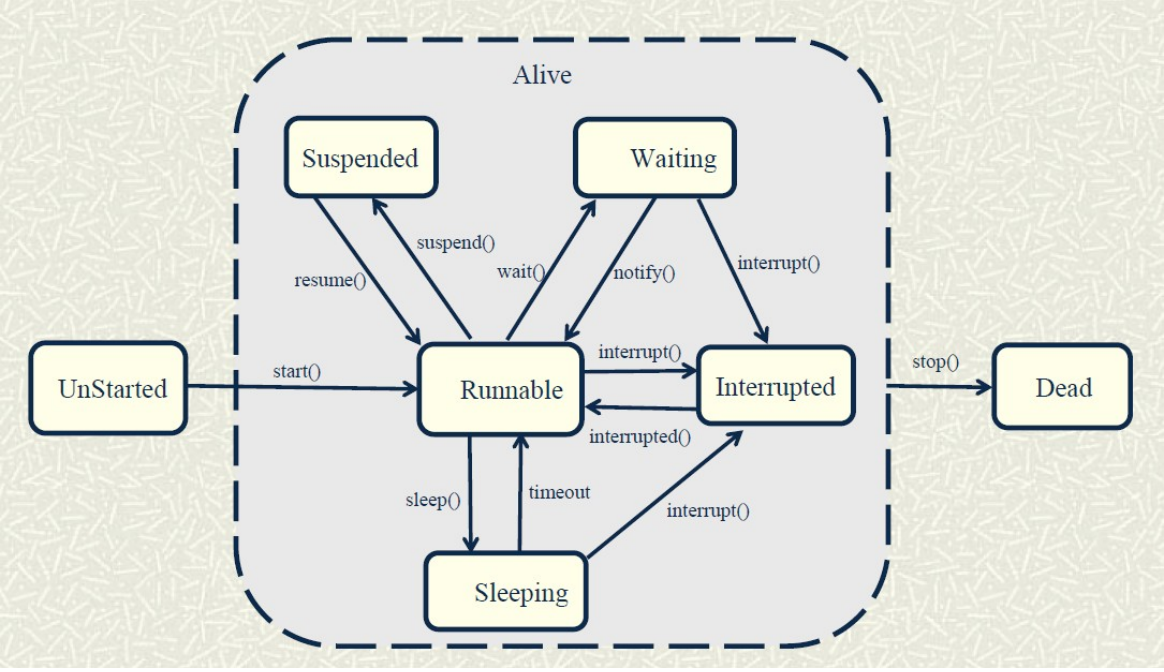
Un hilo es muy similar a un proceso pero con la diferencia de que un hilo se ejecuta dentro del contexto de un proceso. Los procesos mantienen su propio espacio de direcciones y recursos de ejecución mientras que los hilos dependen del proceso , por lo que comparten con otros hilos la sección de código, datos y otros recursos. Únicamente cada hilo tiene su propio contador de programa, conjunto de registros de la CPU y pila para indicar por dónde se está ejecutando.



2.2.3.- Estados de un hilo.

Al igual que los procesos, los hilos pueden cambiar de estado a lo largo de su ejecución. Su comportamiento dependerá del estado en el que se encuentren:

- **Nuevo: el hilo** está preparado para su ejecución pero todavía no se ha realizado la llamada correspondiente en la ejecución del código del programa. Los hilos se inicialilzan en la creación del proceso correspondiente, ya que forman parte de su espacio de memoria pero no empiezan a ejecutar hasta que el proceso lo indica.
- **Listo**: el proceso no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse . El planificador del sistema operativo es el encargado de elegir cuándo el proceso pasa a ejecutarse.
- **Pudiendo ejecutar (Runnable)**: el hilo está preparado para ejecutarse y puede estar ejecutandose. A todos los efectos sería como si el thread estuviera en ejecución, pero debido al número limidtado de núcleos no se puede saber si se está ejecutando o esperando debido a que no hay hardware suficiente para hacerlo.
  Para simplificar, se puede considerar que todos los threads del proceso se ejecutan al mismo tiempo (en paralelo) sabiendo que los hilos deben compartir los recursos del sistema.
- **Bloqueado:** el hilo está bloqueado por diversos motivos (esperando por una operación de E/S, sincronización con otros hilos, dormido, suspendido) esperando que el suceso suceda para volver al estado Runnable. Cuando ocurre el evento que lo desbloquea, el hilo pasaría directamente a ejecutarse.
- **Terminado:** el hilo ha finalizado su ejecución. Sin embargo, frente a los procesos que liberan los recursos que mantenían cuando finalizan, el hilo no libera ningún recurso ya que pertenecen al proceso y no a él mismo.
  Para terminar un hilo, él mismo puede indicarlo o puede ser el propio proceso el que lo finalice mediante la llamada correspondiente.



2.2.4.- Operaciones básicas con los hilos

2.2.4.1.- Creación y arranque de hilos (operación create)

Los threads coomparten tanto el espacio de memoria del proceso como los recursos asociados (entorno de ejecución), siendo su creación más eficiente que la creación de procesos. Estos últimos tienen que crear estructuras especiales en el sistema además de hacer la reserva de memoria y recursos correspondiente.Cualquier programa a ejecutarse es un proceso que tiene un hilo de ejecución principal. Este hilo puede a su vez crear nuevos hilos que ejecutarán código diferente o tareas, es decir el camino de ejecución no tiene por qué ser el mismo.

A la hora de crear los nuevos hilos de ejecución dentro de un proceso, existen dos formas de hacerlo en Java: ***implementando la interfaz Runnable***, o ***extendiendo de la clase Thread*** mediante la creación de subclase.

La interfaz Runnable proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando dicha interfaz. Java proporciona soporte para hilos a través de esta interfaz. La interfaz de Java necesaria para cualquier clase que implemente hilos es **Runnable.**

Las clases que implementan la interfaz Runnable proporcionan una forma de realizar la operación **create**, encargada de crear nuevos hilos. La operación ***create*** , encargada de crear nuevos hilos. La operación **create** inicia un ***thread*** de la clase correspondiente, pasándolo del estado "***nuevo***" al de "***pudiendo ejecutar***".
En la utilización de la ***interfaz Runnable*** , el método ***run()*** implementa la operación **create** conteniendo el código a ejecutar por el hilo. Dicho método contendrá el hilo de ejecución , es decir viene a ser como el método **main ()** en el hilo. Esto implica que el hilo finaliza su ejecución cuando finaliza el método ***run()***. A menudo se denomina a ese método "***el cuerpo del hilo***".

Además de la ***interfaz Runnable***, otra clase principal para el uso de hilos es la ***clase Thread***. La clase **Thread** es responsable de producir hilos funcionales para otras clases e implementa la interfaz ***Runnable***. La interfaz ***Runnable*** debería ser utilizada si la clase solamente va a utilizar la funcionalidad ***run()*** de los hilos. En otro caso se debería derivar de la clase ***Thread*** modificando los métodos que se consideren. Eso si, hay que tener en cuenta que Java no soporta herencias múltiples de forma directa , es decir , no se puede derivar una clase de varias clases padres. Para poder añadir la funcionalidad de hilo a una clase que deriva de otra clase , siendo esta distinta de Threadm se debe utilizar la interfaz ***Runnable.***

Para añadir la funcionalidad de hilo a una clase medianate la clase ***Thread*** simplemente se deriva de dicha clase ignorando el método ***run()*** (proveniente de la interfaz Runnable). La clase ***Thread*** define también un método para implementar la operación ***create*** para comenzar la ejecución del hilo, Este método es ***start()***, que comienza la ejecución del hilo de la clase correspondiente. Al ejecutar ***start()***, la maquina virtual de java llama el método ***run()*** del hilo que contiene el código de la tarea.

Para crear un hilo utilizando la interfaz Runnable se debe crear una nueva clase que implemente la interfaz , teniendo que implementar únicamente el método ***run()*** *con la tarea a realizar* . Además se debe crear una instancia de la clase *Thread* dentro de la nueva clase, la cual representará el hilo a

ejecutar . Como dicho hilo pertenece a la clase *Thread* se debe de utilizar ***start()*** para ponerlo en ejecución o arrancarlo.

```
public class Nuevo Thread implements Runnable {

        Thread hilo;
        public void run() {
                // Código a ejecutar por el hilo
        }
}
```

Ejemplo de creación de un hilo implementando la interfaz Runnable.

```
class HolaThread implements Runnable{

        Thread t;
        HolaThread () {
                t = new Thread (this, "Nuevo Thread");
                System.out. println("Creado hilo: " + t);
                t.start();  //Arranca el nuevo hilo de ejecución. Ejecuta run
        }

        public void run () {
                System.out.println("Hola desde el hilo creado !");
                System.out.println ("Hilo finalizado");
        }
}

class RunThread {

        public static void main (String args [] )  {

                new HolaThread ();  // Crea un nuevo hilo de ejecución
                System.out.println ("Hola desde el hilo principal");
                System.out.println ("Proceso acabado");
        }
}
```

El otro mecanismo de creación de hilos consiste en la creación previa de una subclase de la clase Thread, la cual podemos instanciarla después. Es necesario sobrecargar el método run() con la implementación de lo que se desea que el hilo ejecute. Este método, nunca se ejecuta de forma directa , sino que se llama al método start() de dicha clase para arrancar el hilo. En este caso se heredan los métodos y variables de la clase padre.

```
public class NuevoThread extends Thread {

        public void run () {
                // código a ejecutar por el hilo
        }
}
```

Ejemplo de creación de un hilo extendiendo la clase Thread

```
class HolaThread extends Thread {

        public void run () {
                System.out.println ("Hola desde el hilo creado !");
        }
}

public class RunThread {

        public static void main (String args []) {
                new HolaThread().start(); //Crea y arranca un nuevo hilo de ejecución
                System.out.println("Hola desde el hilo principal !");
                System.out.println ("Proceso acabado");
        }
}
```

A la hora de optar por una u otra opción hay que saber que la utilización de la ***interfaz Runnable*** es más general, ya que el objeto puede ser una subclase de una clase distinta de ***Thread,*** pero no tiene ninguna otra funcionalidad además ***run()*** que la incluida por el programador.

La segunda opción es más fácil de utilizar , ya que tiene definida una serie de métodos útiles para la administración de hilos, pero está limitada porque las clases creadas como hilos deben ser descendientes únicamente de la clase Thread.

Con la implementación de la interfaz Runnable, se podría extender la clase hilo creada , si fuese necesario. Hay que recordar que siempre se arrancan los hilos ejecutando ***start()***, la cual llamará al método ***run()*** correspondiente.

2.2.4.2.- Espera de hilos (operaciones join y sleep)

Si todo fue bien en la operación ***create,*** el objeto de la clase debería contener un hilo cuya ejecución depende del método ***run()*** especificado para ese objeto.

Como varios hilos comparten el mismo procesador, si alguno no tiene trabajo que hacer, es bueno suspender su ejecución para que haya un mayor tiempo de procesador disponible para el resto de hilos del sistema. La ejecución del hilo se puede suspender mediante la operación **join** esperando hasta que el hilo correspondiente finalice su ejecución.

Además se puede dormir un hilo mediante la operación ***sleep*** por un periodo especificado, teniendo en cuenta que el tiempo especificado puede no ser preciso, ya que depende de los recursos proporcionados por el sistema operativo subyacente.

2.2.4.3.- Interrupción

Ambas operaciones de espera pueden ser interrumpidas, si otro hilo interrumpe al hilo actual mientras está suspendido por dichas llamadas.

Una interrupción es una indicación a un hilo que debería dejar de hacer lo que esté haciendo para hacer otra cosa.

Un hilo envía una interrupción mediante la invocación del método ***interrupt()*** en el objeto del hilo que se quiere interrumpir. El hilo interrumpido recibe la ***excepción InterruptedExcepion*** si están ejecutando un método que la lance (por ejemplo , los que implementan las operaciones join y sleep), pero podrían haber sido interrumpidos por interrupt() mientras tanto. En este sentido se debe invocar periódicamente el método interrupted() para saber si se ha recibido una interrupción.

Una vez comprobado si el hilo ha sido interrumpido se puede o bien finalizar su ejecución , o lanzar InterruptedException para manejarla en una sentencia catch centralizada en aplicaciones complejas.

Ejemplo de gestión de interrupciones

```
public void run() {
        for (int i=0; i < Ndatos; i++) {
                try{
                        System.out.println ("Esperando a recibir dato !");
                        Thread.sleep (500);
                } catch (InterruptedException e) {
                        System.out.println ("Hilo interrumpido ");
                        return;
                }
                // Gestiona dato i
        }
        System.out.println ("Hilo finalizado correctamente ");
}
```

NOTA: Queda a criterio del programador decidir exactamente cómo un hilo responde a una interrupción, pero en muchos de los casos lo que se hace es finalizar su ejecución.

Como no se puede controlar cuándo un thread finaliza su ejecución, al depender de su código en el método run(), se puede utilizar el método isAlive() para comprobar si el método no ha finalizado su ejecución antes de trabajar con él.

2.3.- Clase Thread

A la hora de utilizar hilos en Java se utiliza la clase Thread.

2.3.1.- Constructores de la clase Thread

- **Thread()**
- **Thread**(Runnable *threadOb*)
- **Thread**(Runnable *threadOb*, String *threadName*)
- **Thread**(String *threadName)*
- **Thread**(ThreadGroup *groupOb***,** Runnable *threadOb)*
- **Thread**(ThreadGroup *groupOb***,** Runnable *threadOb*, String *threadName*);
- **Thread**(ThreadGroup *groupOb,* String *threadName*)

2.3.2.- Métodos de la clase Thread: Inicio y finalización

- void **run**()
  - Contiene el código que ejecuta el thread.
- void **start**()
  - Inicia la ejecución del thread.
- void **stop**()
  - Termina la ejecución del thread (*)
- static Thread **currentThread**()
  - Retorna el objeto Thread que ha ejecutado este método.
- final void **sleep**(long milliseconds) **throws** InterruptedException
  - Suspende la ejecución del thread por el número de milisegundos especificados.

2.3.3.- Métodos de la clase Thread: Control de finalización

- final boolean **isAlive**()
  - Retorna true si el thread se encuentra en el estado Alive (en alguno de sus subestados), esto es, ya ha comenzado y aun no ha terminado.
- final void **join**() throws InterruptedException
  - Suspende el thread que invoca hasta que el thread invocado haya terminado.
- final void **join**(long *milliseconds*) throws InterruptedException
  - Suspende el thread que invoca hasta que el thread invocado haya terminado o hasta que hayan transcurrido los milisegundos.

2.3.4.- Métodos de la clase Thread: Interrupción

- void **interrupt()**
  - El thread pasa a estado Interrupted. Si está en los estados Waiting, Joining o Sleeping termina y lanza la excepción InterruptedException. Si está en estado Runnable, continua ejecutándose aunque cambia su estado a Interrupted.
- static boolean **interrupted**()
  - Procedimiento estático. Retorna true si el thread que lo invoca se encuentra en estado Interrupted. Si estuviese en el estado Interrupted se pasa al estado Runnable.
- boolean **isInterrupted**()
  - Retorna true si el objeto thread en que se invoca se encontrase en el estado Interrupted. La ejecución de este método no cambia el estado del thread.

2.3.5.- Métodos suspend() y resume() de la clase Thread

- El método **suspend()** permiten parar reversiblemente la ejecución de un Thread. Se reanuda cuando sobre él se ejecute el método **resume()**.

- Ejecutar suspend sobre un thread suspendido o ejecutar **resume()** sobre un thread no suspendido no tiene ningún efecto.

- public final void **suspend()**
  - Suspende la ejecución del thread. Los objetos sobre los que tenía derecho de acceso quedan cerrados para el acceso de otros threads hasta que tras la aplicación de **resume(),** los libere.

- public final void **resume()**
  - Reanuda el thread si ya estuviera suspendido.

2.3.6.- Métodos de la clase Thread: Control de planificación

Constantes

- MAX_PRIORITY //Máxima prioridad asignable al thread.
- MIN_PRIORITY //Mínima prioridad asignable al thread.
- NORM_PRIORITY //Prioridad por defecto que se asigna.

Métodos

- final void **setPriority**(int *priority*)
  - Establece la prioridad de Thread.
- final int **getPriority**()
  - Retorna la prioridad que tiene asignada el thread.
- static void **yield**()
  - Se invoca el planificador para que se ejecute el thread en espera que corresponda.

2.3.7.- Métodos de la clase Thread: Debuging y control

- String **toString**()
  - Retorna un string con el nombre, prioridad y nombre del grupo del Thread.
- final String **getName**()
  - Retorna el nombre del Thread
- final void **setName**(String *threadName*)
  - Establece el nombre del Thread.
- static int **activeCount**()
  - Retorna el número de threads activos en el grupo al que pertenece el thread que invoca.
- final ThreadGroup **getThreadGroup**()
  - Retorna el grupo al que pertenece el thread que invoca.

2.3.8.- Métodos de la clase Thread: Daemon

Existen dos tipos de thread “user” y “daemon “. Su diferencia radica en su efecto sobre la finalización de la aplicación a la que pertenecen.

- Una aplicación termina solo si han terminado todos los threads de tipo user.

- Cuando todos los threads de tipos user han terminado los thread de tipo Daemon son terminados con independencia de cual sea su estado y la aplicación es finalizada.

Metodos relativos al tipo de Thread.

- final void **setDaemon**(boolean *state*)
  - Convierte un thread en daemon. Solo es aplicable después de haber creado el thread y antes de que se haya arrancado (start).
- final boolean **isDaemon**()
  - Retorna true si el thread es de tipo daemon.

2.4.- Planificación de hilos

Cuando se trabaja con varios hilos, en ocasiones es necesario pensar en la planificación de threads, para asegurarse de que cada hilo tiene una oportunidad justa de ejecutarse.

El planificador es quien determina qué proceso es el que se ejecuta en un determinado momento en el procesador, el hilo que se ejecutará estará en función del número de núcleos disponibles y del algoritmo de planificación que se esté utilizando.

Java , por defecto , utiliza un planificador apropiativo. Si en un momento dato un hilo que tiene una prioridad mayor a cualquier otro hilo que se está ejecutando , pasa al estado Runnable, entonces el sistema elige a este nuevo hilo para su ejecución. Si los hilos tienen la misma prioridad , será el planificador el que asigne a uno u otro el núcleo correspondiente para su ejecución utilizando tiempo compartido, en caso de que el sistema operativo subyacente lo permita.

La prioridad de los hilos se puede establecer utilizando el método setPriority () de la clase Thread. Cuando se crea un hilo, este hereda la prioridad del proceso que lo creó , pero puede modificarse dicha prioridad en cualquier momento. Los procesos en Java no pueden cambiar de prioridad , ya que Java le deja esa planificación al sistema operativo, pero en cambio si permite que cambie la prioridad de los thread que se ejecutan.

Utilización de prioridades para gestionar hilos:

```
class CounterThread extends Thread{

        String name;

        public CounterThreard (String name){
                super();
                this.name= name;
        }

        public void run(){
                int count = 0;
                while (true) {
                        try{
```

```
                        sleep (10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    if (count == 1000)
                        count = 0;
                    System.out.println (name+ " : " + count++);
                }
        }
}

public class Prioridad{
        public static void main (String [] args) {
            CounterThread thread1 = new CounterThread ( "thread1");
            thread1.setPriority(10);
            CounterThread thread2 = new CounterThread ( "thread2");
            thread1.setPriority(1);
            thread2.start();
            thread1.start();
        }
}
```

NOTA: *Los sistemas operativos no están obligados a tener en cuenta la prioridad de hilo ya que trabajan a nivel de procesos en sus algoritmos de planificación.*

2.5.- Sincronización de hilos

Los threads se comunican principalmente mediante el intercambio de información a través de variables y objetos en memoria. Como todos los threads pertenecen al mismo proceso , pueden acceder a toda la memoria asignada a dicho proceso y utilizar las variables y objetos del mismo para compartir información, siendo este el método de comunicación más eficiente. Sin embargo, cuando varios hilos manipulan concurrentemente objetos conjuntos, puede llevar a resultado erróneos o a la paralización de la ejecución. La solución es la sincronización

2.5.1.- Problemas de sincronización

2.5.1.1.- Condición de carrera

Se dice que existe una condición de carrera si el resultado de la ejecución de un programa depende del orden concreto en que se realicen los accesos a memoria.

Ejemplo:

Dos hilos, *sumador* y *restador*, se ejecutan al mismo tiempo sobre la misma variable:

- *Sumador*: cuenta++;
- *Restador*: cuenta--;

En código maquina eso se representa por:

- registroX = cuenta
- registroX = registroX (operación: suma o resta) 1
- cuenta = registroX

Supongamos que *cuenta* vale 10, y los dos hilos están ejecutándose. Puede suceder que ambos lleguen a la instrucción que modifica *cuenta* a la vez y se ejecute lo siguiente:

- T0: *sumador* registro1 = cuenta {registro1 = 10}
- T1: *sumador* registro1 = registro1 + 1 {registro1 = 11}
- T2: *restador* registro2 = cuenta {registro2 = 10}
- T3: *restador* registro2 = registro2 - 1 {registro2 = 9}
- T4: *sumador* cuenta = registro 1 {cuenta = 11}
- T5: *restador* cuenta = registro2 {cuenta = 9}

Se ha llegado al valor de cuenta = 9, el cual es incorrecto. En función del orden en que se ejecuten cada una de las sentencias del código máquina el resultado podría ser tanto 9 como 10 u 11. Si el resultado depende del orden en la ejecución en concreto realizada, existirá una condición de carrera. Debido a que el resultado es impredecible y podría devolver el resultado esperado (en este caso 10) las condiciones de carrera son complicadas de detectar y de solucionar.

2.5.1.2.- Inconsistencia de memoria

Una inconsistencia de memoria se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser el mismo dato. Las causas de los errores de coherencia son complejas y van desde la no liberación de datos obsoletos hasta el desbordamiento del buffer entre otros.

Nota: El problema conocido como buffer overflow es una vulnerabilidad, este fallo de coherencia de memoria es muy conocido por los hackers para tomar el control de la maquina

Ejemplo de inconsistencia en memoria

Dos hilos, *sumador* e *impresor*, realizan el siguiente código partiendo de cuenta=0

- *Sumador*: cuenta++;
- *Impresor*: System.out.println(cuenta).

Si las dos sentencias se ejecutaran por el mismo hilo, se podría asumir que el valor impreso sería 1. Pero si se ejecutan en hilos separados, el valor impreso podría ser 0, ya que no hay garantía para el *impresor* de que el *sumador* haya realizado ya su operación a menos que el programador haya establecido una relación de ocurrencia entre estas dos sentencias.

2.5.1.3.- Inanición

Se conoce como inanición al fenómeno que tiene lugar cuando a un proceso o hilo se le deniega continuamente el acceso a un recurso compartido. Este problema se produce cuando un proceso nunca llega a tomar el control de un recurso debido a que el resto de procesos o hilos siempre toman el control antes que él por diferentes motivos.

2.5.1.4.- Interbloqueo

Un interbloqueo se produce cuando dos o más procesos o hilos están esperando indefinidamente por un evento que solo puede generar un proceso o hilo bloqueado.

Ejemplo de interbloqueo

Un ejemplo de un interbloqueo sería:

H0	H1
bloquear (S);	bloquear (Q);
bloquear (Q);	bloquear (S);
...	...
desbloquear (S);	desbloquear (Q);
desbloquear (Q);	desbloquear (S);

2.5.1.5.- Bloqueo activo

Un bloqueo activo es similar a un interbloqueo , excepto que el estado de los dos procesos envueltos en el bloqueo activo cambia constantemente con respecto al otro. En un bloqueo activo , los procesos no están realmente bloqueados, sino que es una forma de inanición debido a que un proceso no deja avanzar al otro.

2.5.2.- Mecanismos de sincronización

Las condiciones de carrera y las inconsistencias de memoria se producen porque se ejecutan varios hilos concurrentemente pudiendo ser ordenados de forma diferentes a la esperada. La solución pasa por provocar que cuando los hilos accedan a datos compartidos, los accesos se produzcan de forma ordenada o síncrona.

2.5.2.1.- Condiciones de Bernstein

Bernstein definió unas condiciones para que dos conjuntos de instrucciones se puedan ejecutar concurrentemente.

Agrupó las instrucciones en dos conjuntos de instrucciones

a) ***Conjunto de lectura:*** formado por instrucciones que cuentan con variables a las que se accede en modo lectura durante su ejecución
b) ***Conjunto de escritura:*** formado por instrucciones que cuentan con variables a las que se accede en modo escritura durante su ejecución

Para que dos conjuntos se puedan ejecutar concurrentemente se deben de cumplir las siguientes condiciones:

1) La intersección entre las variables leídas por un conjunto de instrucciones Li y las variables escritas por otro conjunto Lj debe de ser vacio, es decir, no debe de haber variables comunes.

2) La intersección entre las variables de escritura de un conjunto de instrucciones Li y las variables leidas por otro conjunto de instrucciones Lj debe ser nulo, es decir , no debe de haber variables comunes.
3) La intersección entre las variables de escritura de un conjunto de instrucciones Li y las variables escritas de un conjunto Lj debe ser vacío , no debe de haber variables comunes.

2.5.2.2.- Operación atómica

Una operación atómica es una operación que sucede toda al mismo tiempo. Es decir, siempre se ejecutará de forma continuada sin ser interrumpida, por lo que ningún otro proceso o hilo puede leer o modificar datos relacionados mientras se esté realizando la operación. Es como si se realizara en un único paso.

2.5.2.3.- Sección crítica

Se denomina sección crítica a una región de código en la cual se accede de forma ordenada a variables y recursos compartidos, de forma que se puede diferenciar de aquellas zonas de código que se pueden ejecutar de forma asíncrona.

2.5.2.4.- Semaforos

Los semáforos se pueden utilizar para controlar el acceso a un determinado recurso formado por un número finito de instancias. Un semáforo se representa como una variable entera donde su valor representa el número de instancias libres o disponibles en el recurso compartido y una cola donde se almacenan los procesos o hilos bloqueados esperando para usar el recurso. En la fase de **inicialización**, se proporciona un valor inicial al semáforo igual al número de recursos inicialmente disponibles. Posteriormente , se puede acceder y modificar el valor del semáforo mediante dos operaciones atómicas.

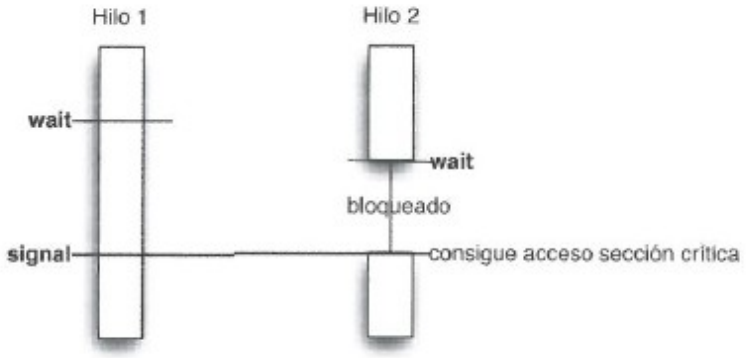
- ***Wait (espera)***. Un proceso que ejecuta esta operación disminuye el número de instancias disponible en 1 , ya que se supone que la va a utilizar. Si el valor es menor que 0, significa que no hay instancias disponibles. En ese sentido , el proceso queda en estado Bloqueado hasta que el semáforo se libere cuando haya instancias. El valor negativo del semáforo especifica cuántos procesos están bloqueados esperando por el recurso.

```
wait (Semaphore S) {
        S.valor- -;
        if (S.valor < 0) {
                //Añadir el proceso o hilo a la lista S.cola
                // Bloquear tarea
        }
}
```

- ***signal (señal)*** . Un proceso, cuando termina de usar la instancia del recurso compartido correspondiente, avisa de su liberación mediante la operación signal. Para ello aumenta el valor de instancias disponible en el semáforo. Si el valor es negativo o menor que 0, significará que hay procesos en estado Bloqueado por lo que despertará a uno de ellos obteniéndolo de la cola. Si existen varios procesos esperando, solamente uno de ellos pasará a estado Runnable. Esto ocurre cuando el número de recursos es limitado.

Por ejemplo , una plaza de parking. Si un coche abandona el parking , solamente otro puede ocupar la plaza dejada sin ser necesario avisar a todos los coches. El hilo que se despierta cuando se hace un signal es aleatorio y depende de la implementación de los semáforos y del sistema operativo subyacente.

```
Signal (Semaphore S) {
        S.valor++;
        if(S.valor <= 0) {
                //Sacar una tarea P de la lista S.cola
                //Despertar a P
        }
}
```



Un semáforo binario , también llamado mutex (**Mut**ual **Ex**clusión) es un indicador de condición que registra si un único recurso está disponible o no. Un mutex solo puede tomar los valores 0 y 1. Si el semáforo vale 1, entonces el recurso está disponible y se accede a la zona de compartición del recurso mientras que si el semáforo es 0 , el hilo debe esperar.

En Java, la utilización de semáforos se realiza mediante el paquete ***java.util.concurrent*** y su clase ***Semaphore*** correspondiente.

Ejemplo de utilización de un único semáforo con valor 1 para crear una sección crítica.

```
import java.util.concurrent.Semaphore;

class Acumula {
        public static int acumulador = 0;
}

class Sumador extends Thread {

        private int cuenta;
        private Semaphore sem;
```

```
        Sumador (int hasta, int id, Semaphore sem) {
                this.cuenta = hasta;
                this.sem = sem;
        }

        public void sumar () {
                Acumula.acumulador++;
        }

        public void run () {

                for (int i=0; i < cuenta; i++) {
                        try{
                                sem.acquire();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                        sumar ();
                        sem.release ();
                }
        }
}

public class SeccionCriticaSemaforos {

        private static Sumador sumadores[ ];
        private static Semaphore semaphore = new Semaphore (1);

        public static void main (String [] args) {

                int n_sum = Integer.parseInt (args [0] );
                sumadores = new Sumador [n_sum];
                for (int i=0; i < n_sum; i++){
                        sumadores [i] = new Sumador ( 100000000 , i , semaphore);
                        sumadores[i].start ();
                }

                for (int i=0; i < n_sum; i++) {
                        try{
                                sumadores[i].join();
                        } catch ( InterruptedException e) {
                                e.printStackTrace();
                        }
                }
                System.out.println (" Acumulador: " + Acumula.acumulador);
        }
}
```

2.5.2.4.1.- Clase Semaphore

Semaphore(int permits)
Crea un semáforo con un cierto número de permisos.

public void acquire(intpermits) throws InterruptedException
Solicita unos permisos. Si no hay suficientes, queda esperando. Si los hay, se descuentan del semáforo y se sigue.Si llega una interrupción, se aborta la espera.

public void acquire() throws InterruptedException
acquire(1)

public void acquireUninterruptibly(int permits)
Solicita unos permisos. Si no hay suficientes, queda esperando. Si los hay, se descuentan del semáforo y se sigue.Si llega una interrupción, se ignora y se sigue esperando.

public void acquireUninterruptibly()
acquireUninterruptibly(1)

release(int permits)
Libera unos permisos que retornan a la cuenta del semáforo. Si hay threads esperando, se reactivan.

release()

2.5.2.5.- Monitores

Un monitor es un conjunto de métodos atómicos que proporcionan de forma sencilla exclusión mutua a un recurso. Los métodos indicados permiten que cuando un hilo ejecute uno de los mismos, solamente ese hilo pueda estar ejecutando un método del monitor.

Funciona de forma similar a los semáforos binarios, pero proporciona una mayor simplicidad ya que lo único que tiene que hacer el programador es ejecutar una entrada del monitor. Mientras que un monitor no puede ser utilizado incorrectamente, los semáforos dependen del programador ya que debe proporcionar la correcta secuencia de operaciones para no bloquear el sistema.

*Para utilizar un monitor en Java se utiliza la palabra clave **synchronized** sobre una región de código para indicar que se debe ejecutar como si de una sección crítica se tratase.*

Existen dos formas de utilizar la palabra clave synchronized :

- Los métodos
- Las sentencias sincronizadas

2.5.2.5.1.- Métodos sincronizados

Los métodos sincronizados es un mecanismo sencillo de construir una sección crítica de forma sencilla. La ejecución por parte de un hilo de un método sincronizado de un objeto Java imposibilita que se ejecute a la vez otro método sincronizado del mismo objeto por parte de otro hilo, cumpliendo con los requisitos de las secciones críticas.

Cuando un hilo invoca un método sincronizado, adquiere automáticamente el monitor que el sistema crea específicamente para todo el objeto que contiene ese método. Solo lo libera cuando el método finaliza o si lanza una excepción no capturada. Ningún otro hilo podrá ejecutar ningún método sincronizado del mismo objeto mientras el monitor de ese objeto no sea liberado, es decir , el cerrojo está cerrado. Esto implica que se bloqueen los métodos que afectan a los recursos compartidos del objeto (indicados son synchronized)

Ejemplo de métodos sincronizados

Para crear un método sincronizado, solo es necesario añadir la palabra clave synchronized en la declaración del método, sabiendo que los constructores ya son síncronos por defecto (no pueden ser marcados como synchronized) ya que solo el hilo que lo llama tiene acceso a ese objeto mientras lo está creando.

```
public class Contador{

	private int c = 0;

	public void Contador (int num) {
		this.c = num;
	}

	public synchronized void increment () {
		c++;
	}

	public synchronized void decrement () {
		c - - ;
	}

	public synchronized int value () {
		return c;
	}
}
```

2.5.2.5.2. Sentencias sincronizadas

Los métodos sincronizados utilizan un monitor que afecta a todo el objeto correspondiente, bloqueando todos los métodos sincronizados del mismo. Esto provoca, por ejemplo , que la ejecución de métodos de solo lectura (sin modificación de datos compartidos) no pueda paralelizarse si existe algún método sincronizado que si modifique los datos y se quiera mantener el orden entre modificaciones y lecturas.

La utilización de **sysnchronized** en una sentencia o región específica de código es mucho más versátil y permite una sincronización de grano fino. Esta funcionalidad permite especificar el objeto que proporciona el monitor en vez de ser el objeto por defecto que se está ejecutando como ocurre en métodos sincronizados. De esta forma, se pueden crear nuevos objetos que se compartirán entre

los hilos. Al bloquearse únicamente los hilos al acceder a esos nuevos objetos, lo que se consigue es bloquear a los hilos únicamente en las secciones de código especificadas por el programado (secciones críticas)

Ejemplo de uso de sentencias sincronizadas

```
class GlobalVar {
        public static int c1 = 0;
        public static int c2 = 0;
}

class DosMutex extends Thread {

        private Object mutex1 = new Object ();
        private Object mutex2 = new Object ();

        public void inc1 () {
                synchronized (mutex1) {
                GlobalVar.c1++;
                }
        }

        public void inc2 () {
                synchronized (mutex2) {
                GlobalVar.c2++;
                }
        }

        public void run () {
                inc1 ();
                inc2 ();
        }
}

public class MutualExclusion {

        public static void main (String [] args ) throws InterruptedException {
                int N = Integer.parseInt (args [0]);
                DosMutex hilos [];
                System.out.println ("Creando "+ N + " hilos " );

                hilos = new DosMutex [N];
                for (int i= 0; i < N ; i++)
                {
                        hilos[i] = new DosMutex ();
                        hilos[i].start ();
                }
                for (int i = 0 ; i < N ; i++)
```

```
                {
                        hilos[i].join();
                }
                System.out.println ("C1 = " + GlobalVar.c1);
                System.out.println ("C2 = " + GlobalVar.c2);
        }
}
```

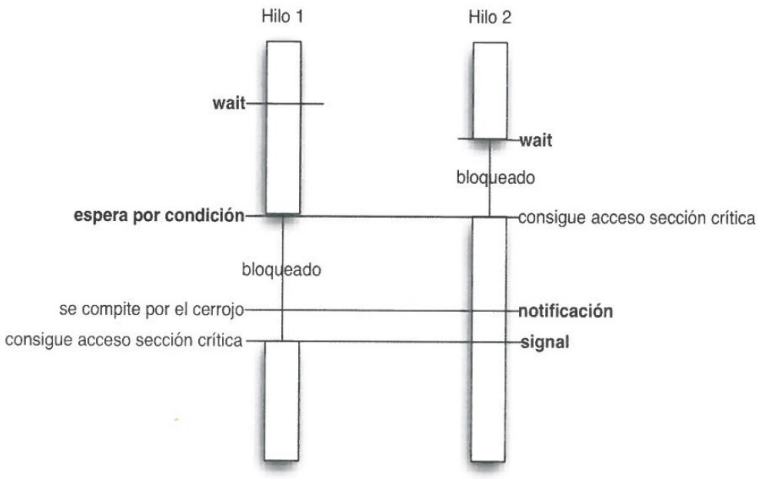
Observación: *Un hilo no puede acceder a una sección protegida por un monitor que ha conseguido otro hilo para ejecutar. Sin embargo, un hilo puede acceder a una sección de código de un monitor que ya posee. Permitir que un hilo pueda adquirir un monitor que ya tiene es lo que se denomina* ***sincronización reentrante****. Esto describe una situación en la cual un hilo está ejecutando código sincronizado , que directa o indirectamente , invoca un método que también contiene código sincronizado, y ambos conjuntos de código utilizan el mismo monitor.*

2.5.2.6.- Condiciones

En ocasiones, el hilo que se está ejecutando dentro de una sección crítica no puede continuar, porque no se cumple cierta condición que solo podría cambiar otro hilo desde dentro de su correspondiente sección crítica. En este caso es preciso que el hilo que no puede continuar libere temporalmente el cerrojo que protege la sección crítica mientras espera a que alguien le avise de la ocurrencia de dicha condición. Este proceso debe ser atómico y cuando el hilo retorna la ejecución lo hace en el mismo punto donde lo dejó (dentro de la sección crítica). En definitiva, una condición es una variable que se utiliza como mecanismo de sincronización en un monitor.

Para implementar condiciones se pueden utilizar operaciones conocidas. Llamar a la operación wait libera automáticamente el cerrojo sobre la sección crítica que se está ejecutando. Para avisar de la ocurrencia de la condición por la que espera, se puede utilizar signal. Sin embargo, esta operación no provoca que los hilos notificados empiecen a ejecutarse en ese mismo instante, sino que el hilo notificado no puede ejecutarse hasta que el hilo que lo notificó no libere el cerrojo de la sección crítica por la cual el hilo notificado está esperando.

Si no se tiene en cuenta que el hilo que esperaba no empieza a ejecutarse inmediatemente cuando recibe la notificación se pueden provocar comportamientos anómalos. Como se ve en la siguiente figura:



El hilo no se pone en ejecución hasta que consigue el cerrojo correspondiente , por el cual pelea en igualdad de condiciones con el resto de hilos. Esto podría provocar que otro hilo ejecute antes y le robe los recursos necesarios (condición) por los que el otro hilo esperó. Por eso mismo, la comprobación de la condición de espera no se debe realizar en una sentencia "if", sino en un "while". Cuando el hilo vuelva ejecutar después de realizar la operación wait siempre debe comprobar si la condición por la que esperó y le notificaron se cumple a su vuelta para poder seguir con la ejecución.

La implementación de condiciones en Java se realiza mediante la utilización de la clase Object. Cualquier hilo puede utilizar la operación signal en cualquier objeto en el cual otro hilo ejecutó la operación wait. Es decir los wait y signal tienen relación únicamente sobre el mismo objeto. Un hilo que espere con un wait sobre un objeto no se despertará por signal de otros hilos en otros objetos. Los métodos que corresponden a la implementación de las operaciones wait y signal para condiciones son wait() y notify() . Dichos métodos siempre se deben ejecutar sobre un bloque sincronizado, es decir , dentro de un monitor.

```
Synchronized public void comprobación_ejecucion()
{
        //Sección critica

        while (condicion)  //no pueda continuar
        {
                wait ();
        }

        // Sección crítica
}

synchronized public void aviso_condicion ()
{
        // Sección critica
        if (condicion_se_cumple)
        notify();
        //Sección critica
}
```

2.5.2.6.1. Clase Object

A la hora de usar condiciones en Java se utiliza la clase Object. Los métodos deben de ejecutarse desde bloques sincronizados.

Esta clase define los siguientes métodos, que están disponibles para todos los objetos.

Método	Descripción
Object clone()	Crea un nuevo objeto, igual al que se duplica
boolean equals (Object objeto)	Determina si un objeto es igual a otro
void finalize()	Llama antes de que un objeto no utilizado sea reciclado.
Class getClass()	Obtiene la clase de un objeto en tiempo de ejecución.
int hashCode()	Devuelve el código asociado al objeto que realiza la llamada.
void notify()	Reiniciar la ejecución de un hilo en espera en el objeto que realiza la llamada.
void notifyAll()	Reiniciar la ejecución de todos los hilos en espera en el objeto que realiza la llamada.
String toString()	Devuelve una cadena que describe el objeto
void wait() void wait (long milisegons) void wait (long milisegons,int nanosegons)	Espera en otro subproceso de ejecución

Los métodos ***getClass***, ***notify***, ***notifyAll*** y ***wait*** están declarados como ***final***, el resto pueden sobrescribir.

Ejemplo de utilización de condiciones. La condición de clase comenzada no sería necesaria ya que se podría gestionar fácilmente con wait y notifyAll de forma sencilla.

```
class Bienvenida{

    boolean clase_comenzada;

    public Bienvenida () {
        this.clase_comenzada = false;
    }

    // Hasta que el profesor no salude no empieza la clase;
    // por lo que los alumnos esperan con un wait

    public synchronized void saludarProfesor () {
        try{
            while (clase_comenzada ==false) {
                wait();
            }
            System.out.println (" Buenos días , profesor " );
        } catch ( InterruptedException es) {
            ex.printStackTrace();
        }
    }

// Cuando el profesor saluda avisa a los alumnos con notifyall de su llegada
```

```
        public synchronized void llegadaProfesor (String nombre) {
                System.out.println ("Buenos días a todos. Soy el profesor " + nombre);
                this.clase_comenzada = true;
                notifyAll ();
        }
}

class Alumno extends Thread{

        Bienvenida saludo;

        public Alumno (Bienvenida bienvenida) {
                this.saludo = bienvenida;
        }

        public void run () {
                System.out.println ("Alumno llegó " );
                try{
                        Thread.sleep (1000);
                        saludo.saludarProfesor ();
                } catch (InterruptedException ex) {
                        System.err.println (" Hilo alumno interrumpido ! ");
                        System.exit (-1);
                }
        }
}

class Profesor extends Thread {
        String nombre;
        Bienvenida saludo;

        public Profesor (String nombre, Bienvenida bienvenida) {
                this.nombre = nombre;
                this.saludo = bienvenida;
        }

        public void run () {
                System.out. Println ( nombre + "  llegó  ");
                try{
                        Thread.sleep (1000);
                        saludo.llegadaProfesor(nombre);
                } catch (InterruptedException ex) {
                        System.err.println (" Hilo profesor interrumpido! ");
                        System.exit (-1);
                }
        }
}

public class ComienzoClase {

        public staric void main (String [] args) {
                // Objeto compartido
                Bienvenida b = new Bienvenida ();

                int n alumnos = Integer.parseInt (args [0]);
                for (int i=0; i < n_alumnos; i++) {
                        new Alumno(b).start();
```

```
                }
                Profesor profesor = new Profesor ("Osvaldo Ramirez" , b);
                profesor.start();
        }
}
```

2.6.- Programación de aplicaciones Multihilo

La programación multihilo permite la ejecución de varios hilos al mismo tiempo, tantos hilos como núcleos tenga el procesador, para realizar una tarea común. A la hora de realizar una programación multihilo cooperativo, se deben seguir las mismas fases que para la programación de aplicaciones multiproceso.

Los pasos a seguir son los siguientes:

- **Descomposición funcional.** Es necesario identificar previamente las diferentes tareas que deben realizar la aplicación y las relaciones existentes entre ellas.
- **Partición.** La comunicación entre hilos se realiza principalmente a través de memoria. Los tiempos de acceso son muy rápidos por lo que no supone una pérdida de tiempo significativa. Hay que tener en cuenta que la existencia de secciones críticas bloquea a los procesos para su sincronización, provocando una pérdida del rendimiento que se puede conseguir. Es conveniente minimizar la dependencia de sincronización existente entre los hilos, para aumentar la ejecución paralela o ejecución asincrona.
- **Implementación**. Se utiliza la clase Thread o la interfaz Runnable como punto de partida.

En el paquete java.util.concurrent aparecen clases de más alto nivel , que se abstraen del bajo nivel mostrado en este tema.

Ver el siguiente tutorial para comprender los nuevos conceptos probando su funcionalidad.

http://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html

https://dis.um.es/~bmoros/Tutorial/parte10/cap10-1.html