Programación de Hilos

Tabla de contenido

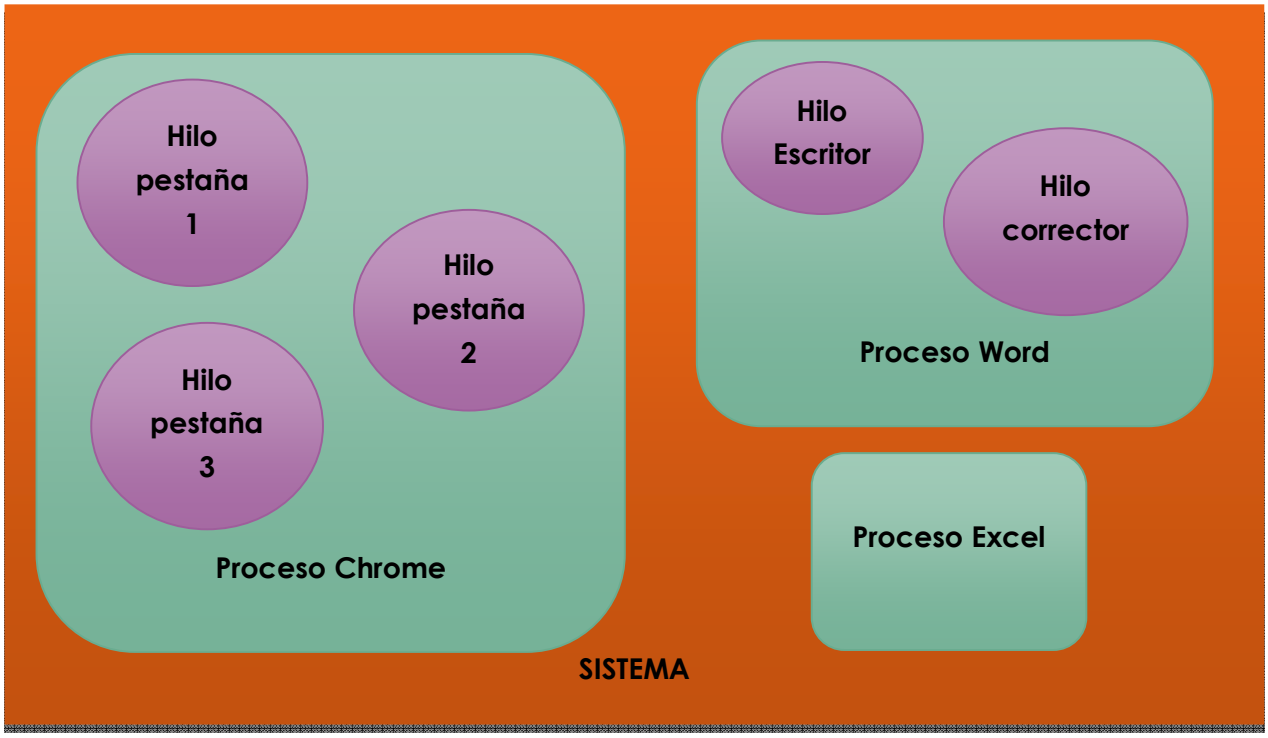
1. Conceptos básicos

Hilo:thread en inglés, son una secuencia de código que está ejecutándose en la CPU, más concretamente un núcleo (core), pero dentro del contexto de un programa.

El SO solo es capaz de gestionar procesos de manera que les asigna memoria y los recursos que necesiten.

Los hilos, al encontrarse dentro del contexto del programa, dependen de un proceso para poder ejecutarse.

Como se vio en el tema anterior los procesos son independientes y por ello no comparten su espacio en memoria pero dentro de un mismo proceso puede haber varios hilos en ejecución y estos sí que tienen acceso al espacio de memoria del proceso al que pertenecen. Cada proceso tendrá al menos **un hilo en** ejecución el cual será el encargado de la ejecución del proceso.

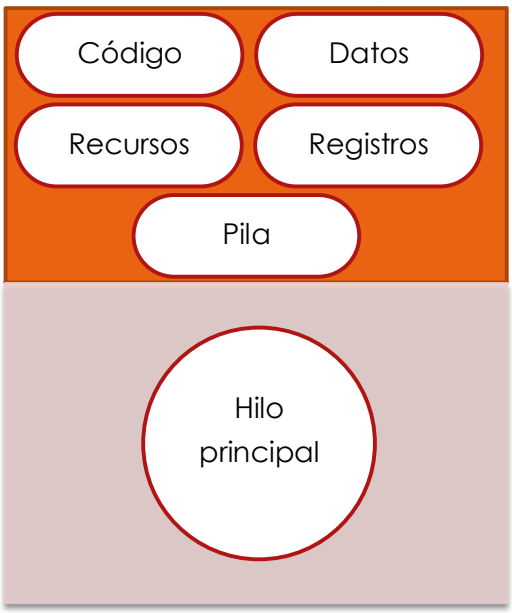


Ventajas de la multitarea frente a la mmultiprogramación

- **Mayor capacidad** de respuesta debido a que puede haber un hilo atendiendo peticiones mientras otro realiza otra tarea más larga. Se usa en la programación de servicios en los servidores, un hilo se encarga de recibir peticiones y por cada petición se abre otro hilo para atenderla.
- **Compartición de recursos** ya que todos los hilos de un proceso tienen acceso a los recursos del proceso por lo que no se necesita ningún mecanismo adicional. Lo que se tendrá que evitar son los problemas derivados de que varios hilos accedan a la vez a un recurso por ejemplo la memoria, así habrá que prestar mayor atención a la **sincronización** entre hilos.
- Como todos los hilos de un proceso utilizan el mismo espacio de memoria para crear hilos nuevos no hay que reservar memoria. Hablando de uso de memoria y recusos es **más barato** crear hilos que procesos.
- En sistemas con varios núcleos reales se consigue un **paralelismo real** en la ejecución de hilos.

2. Recursos compartidos por hilos

Realmente los hilos son muy parecidos a los procesos, de tal manera cada hilo tiene su propio contador de programa, sus registros de la CPU y la pila de ejecución. Pero los hilos de un mismo proceso comparten el código, los datos y los recursos de ese proceso.



Proceso con un único hilo

Proceso con varios hilos

3. Estados de un hilo

Los hilos como los procesos pueden cambiar de estado durante su ejecución, su comportamiento dependerá del estado:

- **Nuevo**: el hilo se encuentra preparado para ser ejecutado pero aún no se le ha llamado. Los hilos se inician en la creación del proceso pero no empiezan hasta que el proceso lo indique.

- **Listo**: El hilo no se está ejecutando pero está preparado, esperando a que el proceso entre a un procesador, es el planificador del SO el que se encarga de elegir qué proceso es el siguiente.

- **Pudiendo ejecutar (runnable)**: El hilo está preparado para ejecutarse o incluso en ejecución. No se puede saber debido a que el hardware no informa de esta situación. Simplificando se considera que todos loshilos de un proceso se ejecutan de manera paralela.

- **Bloqueado**: el hilo está bloqueado, por ejemplo, esperando una operación E/S o una sincronización…

- **Terminado**: El hilo ha finalizado pero no linbera recursos ya que no le pertenencen a él si no al proceso que lo creó. Puede terminar por sí mismo o porque el proceso que lo creó lo finalice.

4. Gestión de hilos

De manera general, un hilo es un proceso que se está ejecutando en un momento determinado en la CPU. Hay programas que por su sencillez sólo utilizan un hilo de ejecución, mientras que otros programas más complejos utilizan varios hilos de ejecución. A los hilos se les denomina procesos ligeros, en contraposición a los procesos llamados procesos pesados.

Un ejemplo de la necesidad de la programación multihilo sería el de un software de gestión de un negocio en el que se tiene que poder añadir nuevos productos en segundo plano mientras se registran compras.

En Java un hilo es un objeto **Thread**.

4.1. Operaciones básicas

4.1.1. Creación y arranque de hilos - create

Los hilos comparten el espacio en memoria y los recursos (entorno de ejecución) del proceso. Cualquier programa que se ejecuta es un proceso que tiene un hilo de ejecución principal. Este hilo a su vez puede crear nuevos hilos que ejecutarán código o tareas diferentes. El camino que siguen los diferentes hilos no tiene por qué ser el mismo.

En Java existen dos formas de crear nuevos hilos:

- Extendiendo de la clase **Thread**.
- Implementar la interfaz **Runnable**.

EN las dos opciones se tendrá que crear una clase nueva. Esa clase nueva contendrá el método **run()**. Este método contendrá el código que ejecutará el hilo.

Independientemente de la opción que se elija se tendrá que crear un objeto tipo **Thread** para poder ejecutar un hilo. En la primera opción es sencillo ya que la clase creada simplemente hereda de la clase Thread. En la segunda opción se creará un objeto Runnable y a continuación un objeto Thread al que se le pase como parámetro el objeto Runable creado anteriormente.

Ejemplo – Creación de hilo con la interfaz **Runnable**.

```
class RunnableClass  implements Runnable {
public void run(){
for(int i=1; i<=5; i++) {
System.out.println("Ejecutando "+ Thread.currentThread().getName() +":"+ i);
}
}
}

public class Hilos {
public static void main(String args[]) {
RunnableClass rc = new RunnableClass();
Thread hilo1 = new Thread(rc);
Thread hilo2 = new Thread(rc);
Thread hilo3 = new Thread(rc);
hilo1.setName("Hilo1");
hilo2.setName("Hilo2");
hilo3.setName("Hilo3");
hilo1.start();
hilo2.start();
hilo3.start();
}
}
```

Ejemplo – Creación de hilo con la clase **Thread**.

```
class HelloThread extends Thread {
public void run() {
System.out.println("Hola desde el hilo creado!");
   }
}

public class RunThread {
public static void main (String args[ ]) {
HelloThread hilo = new HelloThread();// Se crea un nuevo hilo de ejecución
      hilo.start()                                    // se arranca el hilo creado anteriormente
System.out.println("Hola desde el hilo principal!");
System.out.println("Proceso acabando");
}
}
```

La interfaz Runnable solo debería usarse si únicamente se va a usar la funcionalidad **run** de los hilos, para cualquier otro caso debe derivarse de **Thread** modificando los métodos que se necesite.

A la hora de elegir una u otra opción hay que tener en cuenta que Runnable es más general ya que el objeto no tiene por qué ser subclase de Thread pero solo se puede usar la funcionalidad que se incluya en el método run().Mediante el uso de la clase Thread, además de que es más fácil, también se dispone de una serie de métodos para la administración de hilos, aunque como ya se sabe, la clase solo puede ser subclase de Thread.

Hay que tener en cuenta que Java **no permite la herencia múltiple directa**. Por ejemplo, si se quiere añadir la funcionalidad de hilo a una clase que ya extiende a otra que no sea Thread deberá utilizarse la interfaz Runnable (simulando una herencia múltiple).

Ejercicio 1 – Crea un programa que cree un hilo que realice el cálculo de los primeros N números de la sucesión de Fibonacci.

La sucesión de Fibonacci comienza con los números 0 y 1 y el siguiente elemento es la suma de los dos elementos anteriores. Así la sucesión de Fibonacci sería 0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

El parámetro N se pedirá y recogerá en el hilo principal e irá indicado cuando se llamé al constructor de la clase Thread correspondiente.

Ejercicio 2– Crea un programa que cree entre 1 y 10 hilos.

Cada hilo tendrá que mostrar por pantalla:

- Mensaje indicando que empieza.
- El número de hilo por orden de creación.
- Mensaje de que va a finalizar.

Se le preguntará al usuario cuántos hilos se van a crear.

4.1.2. Espera de hilos **sleep y join**

sleep – Si se necesita que un hilo espere durante un tiempo se puede utilizar el método **sleep(tiempo_ms)**. Este método puede lanzar una excepción de tipo **InterruptedException** que es obligatorio capturar. Hay que tener en cuenta que el tiempo que se indique puede no ser preciso ya que depende del planificador del SO en el que se ejecute el código.

Es importante capturar la excepción mediante un boque **try-catch**. La última instrucción del bloque de instrucciones catch siembre debería ser un **return** para finalizar el hilo ya que al en caso contrario el hilo continuará su ejecución sin esperar el tiempo correspondiente lo que podría ocasionar comportamientos inesperados.

```
class HiloEspera extends Thread {
public void run() {
System.out.println("Soy el "+ Thread.currentThread().getName() +" empezando.");
try {
this.sleep(10000);
        // También se puede hacer así y lo aplicará al hilo actual.
        // Thead.sleep(3000);
} catch (InterruptedException e) {
System.out.println(Thread.currentThread().getName() +" interrumpido.");
return;
}
System.out.println(Thread.currentThread().getName() +" acabado.");
}
}
```

Ejercicio 3 – Crea un programa que cree un hilo. Este hilo mostrará su nombre e indicará que está empezando. A continuación, se esperará 3 segundos. Antes de acabar deberá mostrar que va a acabar.

```
import java.util.Random;

public class Hilo implements Runnable {
   private int espera;

   public static void main(String[] args) {
      // Lanzamos dos hilos de forma concurrente que duren un tiempo aleatorio:
      Random aleatorio = new Random(1725);
      for (int i=0; i<2; i++) {
         // Un hilo tendrá un tiempo de ejecución comprendido entre los 0 y 10 segundos.
         new Thread(new Hilo(aleatorio.nextInt(5000))).start();
      }
   }

   public Hilo(int espera) {
      this.espera = espera;
   }

   // Método que contiene las acciones que hará el hilo cuando se ejecute.
   @Override
   public void run() {
String nombre = Thread.currentThread().getName();
      System.out.println("Soy el hilo "+ nombre +" y he iniciado mi ejecución.");
      System.out.println("Soy el hilo "+ nombre +" y voy a parar mi ejecución "+ espera +" ms.");
      try {
         Thread.sleep(espera);
      } catch (InterruptedException e) {
        System.err.println("Soy el hilo "+ nombre +" y me han interrumpido.");
      }

      System.out.println("Soy el hilo "+ nombre +" y continúo mi ejecución.");
      System.out.println("Soy el hilo "+ nombre +" y he finalizado mi ejecución.");
   }
}
```

Ejercicio 4 – Crea un programa que cree dos hilos.

Después de empezar un hilo hijo debe esperar 3 segundos y el otro hilo hijo debe esperar 5 segundos.

El hilo principal después de crear a los dos hijos debe esperar 4 segundos.

Cada hijo debe de mostrar por pantalla su nombre y cuánto tiempo va a esperar

Se puede forzar la interrupción de un hilo mediante el método **interrupt()**.

```
class HiloEspera extends Thread {
@Override
public void run() {
System.out.println("Soy el "+ Thread.currentThread().getName() +" empezando.");
try {
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println(Thread.currentThread().getName() +" interrumpido.");
return;
}
System.out.println(Thread.currentThread().getName() +" acabado.");
}
}

public class CreaHilos {
public static void main(String args[]) {
HiloEsperahilo1 = new HiloEspera();
HiloEsperahilo2 = new HiloEspera();
hilo1.setName("hilo 1");
hilo2.setName("hilo 2");
hilo1.start();
hilo2.start();
      try {
         Thread.currentThread().sleep(5000);
      } catch (InterruptedException e) {
         System.out.println(Thread.currentThread().getName() +" interrumpido. ");
         return;
      }
hilo1.interrupt();
}
}
```

Join – Mediante el método **join()** se puede forzar a un hilo padre a esperar a que su hijo finalice. Al igual que el método sleep() se puede lanzar una excepción de tipo **InterruptedException** que es obligatorio capturar. Hay dos maneras de usar join():

- hiloHijo.join(); → Espera a que el hijo acabe
- hiloHijo.join(5000); → Espera 5 segundos a que el hijo acabe y si no continúa.

Ejercicio 5 – Modifica el programa anterior para que el hilo principal no espere 4 segundos si no que se espere a que acaben los dos hilos que crea. El hilo principal ha de mostrar por pantalla cuando empieza y cuando acaba.

4.2. Hilo daemon

En Java existen dos tipos de hilos, **user** y **daemon**. Por defecto todos los hilos que se crean son de tipo **user**. Para cambiar el tipo de hilo a **daemon** se usa el método **setDaemon(boolean)**.

El tipo de hilo influye a la hora de finalizar un programa:

- Una aplicación termina únicamente si todos los hilos de tipo user han finalizado.
- Si existe algún hilo de tipo daemon y acaban todos los hilos user, todos los hilos daemon que existan finalizarán independientemente del estado de estos y la aplicación se finalizará. Como la aplicación se termina no se lanzará ninguna excepción.

Se puede comprobar si un hilo es de tipo daemon con el método **isDaemon()**.

Los hilos de tipo daemon se suelen utilizar para dar servicio a hilos de tipo user, por eso cuando terminan todos los hilos de tipo user ya no tiene sentido la existencia de hilos tipo daemon.

Ejercicio 6 – Comprueba el funcionamiento de los hilos tipo **daemon**creando una copia del ejercicio 4 y añadiendo un hilo nuevo que será de tipo daemon y esperará 20 segundos.

4.3. Planificación de hilos

En Java no se puede cambiar la prioridad de los procesos ya que eso es controlado por el SO. Al contrario, sí que se puede cambiar la prioridad de los hilos que se ejecutan. Mediante las prioridades se puede planificar qué procesos se ejecutan para asegurarse que todos los hilos tengan la oportunidad de entrar a ejecutarse.

Java por defecto utiliza una planificación apropiativa por prioridades, por lo que si un hilo tiene mayor prioridad pasará al estado **runnable**. En caso de haber varios hilos con la misma prioridad será el planificador el que elija qué hilo pasa a ejecutarse.

Para establecer la prioridad de los hilos se dispone del método **setPriority()** de la clase Thread. Las prioridades se indican con un número entero normalmente comprendido entre 1 y 10, estos valores están definidos en dos constantes de la clase Thread: MIN_PRIORITY y MAX_PRIORITY. Cuando un hilo es creado, este hereda directamente la misma prioridad del hilo que lo crea.

Los SO no están obligados a tener en cuenta la prioridad de los hilos al trabajar a nivel de procesos.

```
class HiloPrioridad extends Thread {
public void run() {
System.out.println("Soy el "+ Thread.currentThread().getName() +" empezando.");
      for(int i=0; i<10; i++)
         System.out.println("Soy el "+ Thread.currentThread().getName() +" iteración "+ i);
System.out.println("Soy el "+ Thread.currentThread().getName() +" acabando.");
}
}
public class Prioridad {
public static void main(String[] args) {
HiloPrioridad thread1 = new HiloPrioridad();
thread1.setName("Hilo 1");
thread1.setPriority(1);
HiloPrioridad thread2 = new HiloPrioridad();
thread2.setName("Hilo 2");
thread2.setPriority(Thread.MAX_PRIORITY);
thread1.start();
thread2.start();
}
}
```

Ejercicio 7 – Prueba el ejemplo anterior en tu equipo. Ejecuta el programa varias veces cambiando la prioridad de los hilos y comprueba si se nota este cambio.

Haz que se creen cuatro hilos más y vuelve a hacer las pruebas anteriores y comprueba si se notan los cambios de prioridad.

¿A qué se debe este comportamiento cuando se cambian las prioridades?

Ejercicio 8 – Crea un programa que cree 10 hilos de manera recursiva, cada hilo creará un hilo hijo y así sucesivamente. Cada hilo padre tiene que esperar a que su hijo acabe. El último hilo creado ha de esperar 5 segundos. Todos los hilos han de informar de quién son, que van a empezar, que van a esperar y que han acabado.

5. Sincronización de hilos

Los hilos se pueden comunicar intercambiando información a través de los objetos almacenados en la memoria. Como los hilos pertenecen a un proceso, pueden acceder todos a los objetos del mismo que haya en memoria. Este método de comunicación es muy eficiente.

Compartir la memoria por los hilos puede producir resultados erróneos si los hilos acceden a la vez. Para solucionar estos problemas se utiliza la **sincronización**.

5.1. Problemas de la sincronización

- **Condición de carrera**: se define así cuando el resultado de la ejecución de un programa depende del orden concreto en el que se realizan los accesos a memoria.

  Ejemplo:supongamos que se tienen dos hilos, H1 que suma (cuenta++) y H2 que resta (cuenta--) y cuenta inicialmente vale 10.

  En una ejecución óptima primero sumaría y luego restaría o viceversa, pero como resultado siempre daría que cuenta vale 10.

Una **operación atómica** es aquella que en código máquina solo se traduce en una única instrucción. El problema es que ++ y -- no son operaciones atómicas y se traducen en código máquina de la siguiente forma:

registroX = cuenta;	registroX = cuenta;
registroX = registroX + 1	registroX = registroX - 1
cuenta = registroX	cuenta = registroX

Si estas instrucciones se ejecutan en el siguiente orden:

H1: registro1 = cuenta	registro1 = 10
H1: registro1 = registro1 + 1	registro1 = 11
H2: registro2 = cuenta	registro2 = 10
H2: registro2 = registro2 - 1	registro2 = 9
H1: cuenta = registro1	cuenta = 11
H2: cuenta = registro2	cuenta = 9

Como se puede ver el resultado es erróneo.

- **Inconsistencia de memoria**: se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser el mismo dato. Puede ocurrir desde la no liberación de datos obsoletos hasta por el desbordamiento de un buffer (**buffer overflow**).

  Ejemplo: supongamos que se tienen dos hilos, H1 que suma (cuenta++) y H2 que imprime (System.out.println(cuenta)) y cuenta inicialmente vale 0.

  Si las dos sentencias se ejecutaran en el mismo hilo, primero la suma y luego la impresión, el valor impreso sería 1.

  Si esas instrucciones se ejecutan en hilos diferentes no hay garantía de que primero se sume y que luego se imprima por lo que el valor impreso podría ser 0 lo cual es un resultado no deseado

- **Inanición**: es el fenómeno que ocurre cuando a un proceso o hilo se le deniega el acceso a un recurso compartido debido a que otros procesos o hilos siempre toman el control de ese recurso antes por diferentes motivos.

  Esto puede producirse si un hilo tiene una prioridad inferior a la del resto de hilos.

- **Interbloqueo**: se produce cuando dos o más procesos o hilos se encuentran esperando indefinidamente por un evento que ha de generar otro proceso o hilo que se encuentra bloqueado o que se encuentra esperando un recurso que tiene asignado otro proceso o hilo que está bloqueado.

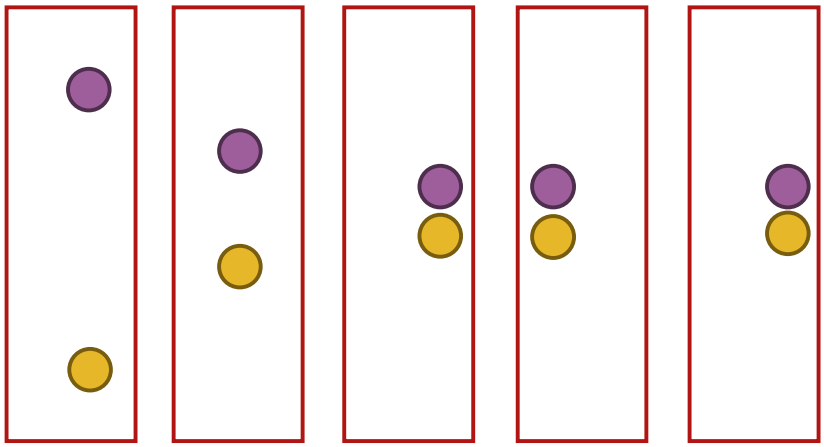
  Ejemplo: Supongamos que se tienen 2 hilos (H1 y H2) y dos recursos (R1 y R2). Los dos hilos necesitan tener asignados los dos recursos para trabajar. La secuencia de instrucciones de cada hilo podría ser la siguiente:

H1	H2
Petición R1	Petición R2
Petición R2	PeticiónR1
Instrucciones	Instrucciones
Desbloquear R1	Desbloquear R2
Desbloquear R2	Desbloquear R1

  Si los hilos se ejecutan de manera paralela y ejecutan su primera instrucción a la vez se bloquearían entre ellos.

- **Bloqueo activo**: es similar al interbloqueo, pero los procesos o hilos involucrados van cambiando su estado respecto al otro volviendo a bloquearse mutuamente.

  Ejemplo: para entender problema lo mejor es pensar en el mundo real. Imaginémonos que por un pasillo avanzan dos personas en sentido opuesto. Cuando llegan a la altura del otro una persona se aparta hacia su derecha y la otra hacia su izquierda de manera que no pueden pasar ninguna. Inmediatamente la primera persona se aparta a su izquierda y la segunda se aparta hacia su derecha volviendo a bloquearse el paso.



En el ejemplo anterior de los dos hilos, imaginémonos que cada uno de los hilos intenta conseguir los dos recursos que necesita, en el caso de tener uno asignado y no conseguir el otro procede a liberar el recurso y lo vuelve a intentar. Hasta que no tenga los dos recursos no continuará.

5.2. Mecanismos de sincronización

Los problemas anteriormente citados ocurren debido a que varios hilos se ejecutan concurrentemente y no tener control sobre su orden, lo que desemboca en una ordenación no deseada. Para solucionarlo se debería controlar que los accesos a datos compartido se realicen de manera ordenada o **síncrona**. Y cuando estén ejecutando código independiente de los datos compartidos que se ejecuten en paralelo libremente también llamado ejecución **asíncrona**.

5.2.1. Condiciones de Berstein

Las condiciones de Berstein definen las condiciones que permiten averiguar si dos segmentos de código (i y j por ejemplo) se pueden ejecutar en paralelo de manera asíncrona en diferentes hilos.

- **Dependencia de flujo**. Todas las variables de entrada del segmento j han de ser diferentes de las variables de salida del segmento i. En caso contrario el segmento j depende de la ejecución de i.

- **Antidependencia**. Todas las variables de entrada del segmento i han de ser diferentes de las variables de salida del segmento j. En caso contrario el segmento i es dependiente de la ejecución de j. (caso contrario al anterior).

- **Dependencia de salida**. Todas las variables de salida del segmento i deben ser diferentes a las variables de salida del segmento j. En caso contrario los dos segmentos podrían escribir en el mismo lugar provocando resultados inesperados.

El código que no cumpla estas condiciones debe de ejecutarse de manera síncrona.

5.2.2. Operaciones atómicas.

Una **operación atómica**, como ya se explicó anteriormente, se ejecutará completamente sin ser interrumpida. De esta manera ningún proceso o hilo puede interferir en su ejecución.

Por ejemplo, se pueden declarar las variables como **volatile** para que el sistema no actualice la variable en un registro si no directamente en memoria en un único paso. Esta solución es una buena ayuda para el acceso a las variables, pero hay otros casos en los que es necesario aportar atomicidad a una parte del código formada por varias instrucciones.

Imagina que vas a sacar dinero al cajero. Las operaciones sacarDinero y apuntarEnCuenta deberían poder ejecutarse en una única instrucción.

5.2.3. Sección crítica

Una **sección crítica** es aquella en la que se accede a variables y recursos compartidos de forma ordenada por lo que las instrucciones han de ejecutarse de manera síncrona. Este concepto se puede aplicar tanto a procesos como hilos si acceden datos o recursos compartidos.

- Si ya hay un proceso o hilo ejecutando su sección crítica y otro quiere ejecutar la suya, este último se bloqueará hasta que el primero termine con su sección crítica.
- De esta manera se establece un antes-después en el orden de ejecución de secciones críticas, así los cambios son visibles desde el resto de procesos o hilos.

La sección crítica tiene un problema y este es el diseñar un protocolo de ejecución de la sección crítica. Para que esté bien diseñado ha de cumplir:

- **Exclusión mutua**. Si un proceso o hilo está ejecutando su sección crítica, ningún otro puede pasar a ejecutar la suya.
- **Progreso**. Si no hay ningún proceso o hilo ejecutando su sección crítica y hay varios esperando a ejecutar la suya alguno debe de empezar. No pueden estar esperando todos si ninguno está en su sección crítica.
- **Espera limitada**. Debe haber un número limitado de procesos o hilos que entran a su sección crítica cuando un proceso o hilo pide ejecutar la suya. En caso contrario se podría producir inanición.

Resumiendo, el código correspondiente a la sección crítica debe terminar en un tiempo determinado y se debe evitar la inanición. Para implementar la sección crítica hay diferentes mecanismos cuya implementación depende de cada SO. Java provee estos mecanismos, pero ocultando al programador esas diferencias que existen a nivel de SO, estos mecanismos son los **semáforos** y los **monitores**.

5.2.4.Semáforos

Los **semáforos** son mecanismos que permiten que un número limitado de procesos accedan de manera ordenada a un recurso. Se representa con una variable entera cuyo valor es el número de huecos disponibles en el recurso compartido y una cola donde almacenar a los procesos bloqueados que esperan el acceso.

En la inicialización se proporciona un valor que será el número de huecos disponibles. Existen dos operaciones atómicas para acceder y modificar el valor del semáforo.

- **wait**. Mediante esta operación se disminuye el número de huecos en uno, si el valor es menor que cero es que no hay hueco y el proceso debería bloquearse hasta que haya hueco. El valor negativo indica cuántos procesos están bloqueados.

- **signal**. Cuando un proceso termina de usar el recurso utiliza esta operación para avisar. Con esta operación se aumenta en uno el número de huecos. Si el valor es menor o igual a cero es que hay algún proceso bloqueado esperando y habrá que despertarlo. El

proceso que se despierta es aleatorio y depende de la implementación del semáforo y del SO donde se ejecute.

En sistemas monoprocesador se usa la inhibición de interrupciones para que sean operaciones atómicas. En sistemas multiprocesador hay que usar instrucciones especiales (TestAndSet o Swap) o soluciones software (algoritmo de Peterson).

MUTEX (**MUT**ual **EX**clusion), son semáforos binarios, sólo pueden tener valor 0 (recurso ocupado) o 1 (recurso disponible). Son un mecanismo que permiten la sincronización con hilos. Son como la llave de una habitación, solo hay una llave y el que tiene la llave puede acceder.

En Java el uso de semáforos se realiza mediante la clase **Semaphore** del paquete **java.util.concurrent**.

Ejemplo – Utilización de semáforos para restringir el acceso a variables.

```
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[];   // Array de hilos
    private static Semaphore semaphore = new Semaphore(1);
    public static void main (String[] args) {
            int n_sum = 10;
        sumadores = new Sumador[n_sum];
        for(int i=0; i<n_sum;i++) {
            sumadores[i] = new Sumador(10000000, i, semaphore);
sumadores[i].start();
            try {
               sumadores[i].join();
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
        }
        System.out.println("Acumulador: "+ Acumula.acumulador);
    }
}
```

5.2.5. Monitores

Un **monitor** es un conjunto de **métodos atómicos** que proporcionan exclusión mutua a un recurso. Estos métodos proporcionan que cuando un hilo ejecuta uno de esos métodos únicamente él pueda ejecutar un método del monitor. Se podría decir que los monitores son como un **cerrojo (lock)** que permite el acceso o no.

Su funcionamiento es muy similar al de los semáforos binarios, pero con mayor simplicidad ya que el programador solo tiene que ejecutar una entrada al monitor, por lo que un monitor nunca se puede utilizar de manera incorrecta. sin embargo, los semáforos dependen de que el programador introduzca las sentencias en el orden correcto para no bloquear el sistema.

En Java todos los objetos derivados de la clase **Object** (casi todos), tienen asociados dos monitores: uno afecta a la propia clase en sí y otro que afecta a los métodos definidos como **synchronized**.

Para usar un monitor en Java se utiliza la palabra clave **synchronized** sobre una región del código. De esta manera esa sección se ejecutará como si fuera una **sección crítica**. **synchronized** se puede usar en métodos y en **sentencias sincronizadas**.

Métodos sincronizados:

Un método declarado synchronized garantiza que no se pueda ejecutar de manera concurrente a él otro método declarado sinchronized del mimo objeto.

Cuando un hilo invoca un método synchronized intenta tomar el **cerrojo** del objeto al que pertenezca el método. Si está libre lo toma y ejecuta el método. Si el **cerrojo** está ocupado el hilo se suspenderá hasta que se libere el cerrojo.

En caso de que el método sea **static synchronized** el cerrojo será de la clase y afectará a todos los métodos **static synchronized** de esa clase.

Para crear un método sincronizado se ha de añadir la palabra **synchronized** en su declaración. Es importante saber que los constructores por defecto son síncronos por lo que no se pueden marcar con **synchronized**.

- Cuando un hilo toma el cerrojo de un objeto puede acceder a otro método synchronized de dicho objeto sin necesidad de soltar el cerrojo.
- Hay un cerrojo por cada instancia del objeto.
- Cuando se extienda una clase y se sobrescriba un método synchronized no es obligado que el método nuevo sea synchronized.

Ejemplo

```
public class Contador {
   private int e=0;
   public void Contador(int num) {
      this.c=num;
   }
  public synchronized void increment () {
      e++;
   }
public synchronized void decrement () {
e--;
   }
public synchronized int value() {
      return c;
   }
}
```

Ejemplo

```
public class Main {
   public static void main(String[] args) {
      VerificarCuenta vc = new VerificarCuenta();
      Thread Luis = new Thread(vc, "Luis");
      Thread Manuel = new Thread(vc, "Manuel");
      Luis.start();
      Manuel.start();
   }
}

class CuentaBanco {
```

```
    private int balance = 50;
    public CuentaBanco(){
    }
    public int getBalance(){
        return balance;
    }
    public void retiroBancario(int retiro){
        balance = balance - retiro;
    }
}

class VerificarCuenta implements Runnable{
    private CuentaBanco cb = new CuentaBanco();
    private synchronized void HacerRetiro(int cantidad) throws InterruptedException{
        if(cb.getBalance() >= cantidad){
            System.out.println(Thread.currentThread().getName() + " está realizando un retiro de: "
                    + cantidad + ".");
            Thread.sleep(1000);
            cb.retiroBancario(cantidad);
            System.out.println(Thread.currentThread().getName() + ": Retiro realizado.");
            System.out.println(Thread.currentThread().getName() + ": Los fondos son de: " +
cb.getBalance());
        } else {
            System.out.println("No hay suficiente dinero en la cuenta para realizar el retiro Sr."
                    + Thread.currentThread().getName());
            System.out.println("Su saldo actual es de "+cb.getBalance());
            Thread.sleep(1000);
        }
    }
    @Override
    public void run() {
        for (int i = 0; i <= 3; i++) {
            try {
                this.HacerRetiro(10);
                if(cb.getBalance() < 0){
                    System.out.println("La cuenta está sobregirada.");
                }
            } catch (InterruptedException e) {
             e.printStackTrace();
            }
        }
    }
}
```

Ejercicio 9 – Crea una clase en Java que utilice 5 hilos hijos para contar el número de vocales que hay en un determinado texto. Cada hilo se encargará de contar una vocal diferente. Todos los hilos actualizan el número de vocales en una variable común. Hay que utilizar métodos sincronizados para evitar condiciones de carrera.

Ejercicio 10 – Crea una clase en Java que haga lo siguiente. El hilo principal escribirá en un archivo llamado vetusta.txt el texto "Vetusta Morla - La deriva" y a continuación creará dos hilos. Una vez creados los dos hilos esperará a que los dos acaben antes de cerrar el archivo de texto.

El primer hilo mediante un método llamado estrofa1 escribirá en el archivovetusta.txt el texto:

> He tenido tiempo de desdoblarme
>
> y ver mi rostro en otras vidas.
>
> Ya tiré la piedra al centro del estanque.

El segundo hilo mediante un método llamado estrofa2 escribirá en el archivo vetusta.txt el texto:

> He enterrado cuentos y calendario,
>
> ya cambié el balón por gasolina.
>
> Ha prendido el bosque al incendiar la orilla.

Ten en cuenta que los métodos estrofa1 y estrofa2 tendrán que ser synchronized.

Cuando lo tengas realiza una prueba quitando la palabra **synchronized** de los métodos y observa el texto del archivo.

Sentencias sincronizadas:

Los métodos sincronizados utilizan un monitor que afecta a todo el objeto correspondiente de manera que se bloquean todos los métodos sincronizados. De esta manera no se podrían ejecutar métodos de solo lectura de manera paralela.

Si se utiliza **synchronized** en una sentencia o región específica se conseguirá una sincronización más detallada.

Para usar sentencias sincronizadas se utiliza monitor de un objeto o lo que es lo mismo el monitor de la instancia de la clase (no el que afecta a los métodos **synchronized**).

Ejemplo – Uso de sentencias sincronizadas.

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

class TwoMutex 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 = 20;
TwoMutex hilos[];
```

```
        System.out.println("Creando "+ N +" hilos");
        hilos = new TwoMutex[N];
        for (int i=0; i<N; i++) {
hilos[i] = new TwoMutex();
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);
    }
}
```

En el ejemplo anterior se han creado dos objetos auxiliares para usar su monitor así se pueden ejecutar de manera concurrente los dos métodos (**inc1** e **inc2**) debido a que actúan sobre variables diferentes.

A continuación, se puede ver el mismo ejemplo anterior, pero con un solo monitor.

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

class TwoMutex extends Thread{
    GlobalVar globalVar = new GlobalVar();

public TwoMutex (GlobalVar globalVar) {
            this.globalVar = globalVar;
    }

    public void inc1() {
synchronized(globalVar) {
            globalVar.c1++;
        }
    }
    public void inc2() {
        synchronized (globalVar) {
            globalVar.c2++;
        }
    }
    public void run () {
            inc1();
            inc2();
    }
}

public class MutualExclusion {
    public static void main(String[] args) throws InterruptedException {
```

```
        int N = 10000;
        TwoMutex hilos[];
        GlobalVar globalVar = new GlobalVar();
        System.out.println("Creando "+ N +" hilos");
        hilos = new TwoMutex[N];
        for (int i=0; i<N; i++) {
            hilos[i] = new TwoMutex(globalVar);
            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);
    }
}
```

Prueba a comentar la parte **synchronized**, ejecuta el programa varias vecesy observa el resultado.

5.2.6. Condiciones

Hay ocasiones en las que un hilo que está ejecutando su sección crítica no puede continuar debido a que no se cumple **una determinada condición** que solo puede cambiar dentro de la sección crítica de otro hilo. Si el hilo que está esperando la condición no libera el **cerrojo** que protege la sección crítica hasta que la condición cambie.

Este proceso debe ser **atómico** y cuando el hilo retome su ejecución debe ser en el mismo punto de la sección crítica en el que se quedó. Básicamente se usa la condición como variable que sincroniza un monitor.

Hay que tener en cuenta el hecho de que el hilo que está esperando comience a ejecutarse nada más recibir la notificación para evitar comportamientos extraños.

wait() esta operación libera el cerrojo de la sección crítica que se está ejecutando.

Debe utilizarse dentro de un método synchronized para que la suspensión del hilo y la liberación del cerrojo sea **operaciones atómicas**. De la misma manera la activación del hilo y la toma del cerrojo también ha de ser una **operación atómica**.

```
        synchronized void EjecutaSiCondicion() {
                while (!condición) wait();
                // Instrucciones que se ejecutan cuando se cumpla la condición
```

```
        }
```

notify() sirve para avisar a los procesos que la condición ya se cumple. Esta operación no provoca que los hilos notificados empiecen en ese preciso instante.

Debe utilizarse dentro de un método synchronized tal y como se ha indicado anteriormente. Debido a que puede haber más de un hilo esperando sobre ese objeto se pueden utilizar dos métodos: **notify()** solo un hilo que no se sabe cuál será se despertará y **notifyAll()** todos se despertarán y cada uno decidirá si le afecta la notificación o no.

```
        synchronized void CambiaCondicion() {
                // Instrucciones que pueden hacer que la condición se cumpla
                notifyAll;
        }
```

Ejemplo – Condiciones.

```
// código hilo 1
public synchronized guardedJoy() {
while(!joy) {
try {
wait();
} catch (InterruptedException e) {}
 }
System.out.println("Joy and efficiency have been achieved!");
}

//código hilo 2
public synchronized notifyJoy() {
joy = true;
notifyAll();
}
```

Ejemplo – Así podría ser un semáforo en java.

```
public class Semaphore{
int value;

public Semaphore(int initialValue){
value = initialValue;
}

synchronized public void signal(){
      value= value+1;
```

```
notify();
}

synchronized public void await() throw InterruptedException{
while (value == 0) wait();
      value = value - 1;
   }
}
```

Ejemplo – Así podría ser un semáforo binario (MUTEX) en java.

```
public class Semaphore{
int value = 1;

synchronized public void signal(){
      value= value+1;
notify();
}
synchronized public void await() throw InterruptedException{
while (value == 0) wait();
      value = value - 1;
   }
}
```

Ejemplo – Simular el comienzo de una clase con profesor y alumnos. Hasta que el profesor no salude no empezará la clase.

```
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(Thread.currentThread().getName() +" Buenos dias, profesor.");
                } catch (InterruptedException e) {
                        e.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 "+ Thread.currentThread().getName() +" llegó.");
			try {
				Thread.sleep(1000);
				saludo.saludarProfesor();
			} catch (InterruptedException e) {
				System.err.println("Thread 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 e) {
				System.err.println("Thread profesor interrumpido!");
				System.exit(-1);
			}
		}
}

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

			int n_alumnos = 10;
			for (int i=0 ; i<n_alumnos; i++) {
```

```
                    Alumno alumno = new Alumno(b);
                    alumno.setName("alumno-"+ i);
                    alumno.start();
                }
                Profesor profesor = new Profesor("Manolo Gómez", b);
                profesor.start();
        }
}
```

Sincronizar, ¿Cuándo?

Se sincronizarán los hilos siempre que dos o más hilos accedan al mismo objeto o atributo y alguno de los hilos vaya a modificarlo.

No hay que sobre sincronizar ya que causa retrasos innecesarios, habrá que detectar las instrucciones concretas que se desea sincronizar y en vez de poner que todo el método esté sincronizado que solo lo estén esas instrucciones concretas.

No es necesario sincronizar cuando se acceden a variables locales ya que estas variables pertenecen al ámbito del hilo y no hay posibilidad de error.