

# UT2 - Programación multihilo

---

- [UT2 - Programación multihilo](#)
  - [Objetivos](#)
  - [0. Introducción](#)
    - [Multitarea basada en procesos](#)
    - [Multitarea basada en hilos](#)
    - [Hilos](#)
  - [1. Fundamentos de la programación multihilo](#)
    - [1.1 Programación secuencial \(un hilo\)](#)
    - [1.2 Programación concurrente \(multihilo\)](#)
    - [1.3 Estados de los hilos](#)
    - [Sleep](#)
    - [Join](#)
  - [2. Sincronización de hilos](#)
    - [2.1 Región crítica](#)
    - [2.2 Monitores \(productor-consumidor\)](#)
    - [2.3 Semáforos](#)

## Objetivos

- Conceptos de hilo y asincronía y su utilidad en programación.
- Características y utilidades de los hilos.
- Aplicación de la programación multihilo y técnicas de programación.
- Paquetes y clases más importantes para la programación multihilo.
- Programación de aplicaciones multihilo.

## 0. Introducción

Un **hilo** es 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.

**Hilo vs Proceso:** hemos visto que el sistema operativo gestiona procesos, asignándoles la memoria y recursos que necesiten para su ejecución. En este sentido, el sistema operativo planifica únicamente procesos

Podemos encontrar dos formas de realizar multitarea (multitasking).

- multitarea basada en procesos
- multitarea basada en hilos

### Multitarea basada en procesos

Un proceso es un programa que se está ejecutando. La multitarea basada en procesos es la característica que le permite a la computadora ejecutar dos o más programas concurrentemente.

Los procesos son tareas pesadas que necesitan su propio espacio de direccionamiento. La comunicación entre procesos es más cara y limitada y resulta costoso el cambio de contexto de un proceso a otro.

## Multitarea basada en hilos

La multitarea basada en hilos requiere menos sobrecarga que la multitarea basada en procesos.

Los hilos son más ligeros, ya que comparten la memoria y todos los recursos del proceso al que pertenecen. Por lo tanto la creación de nuevos hilos no supone una reserva adicional de memoria.

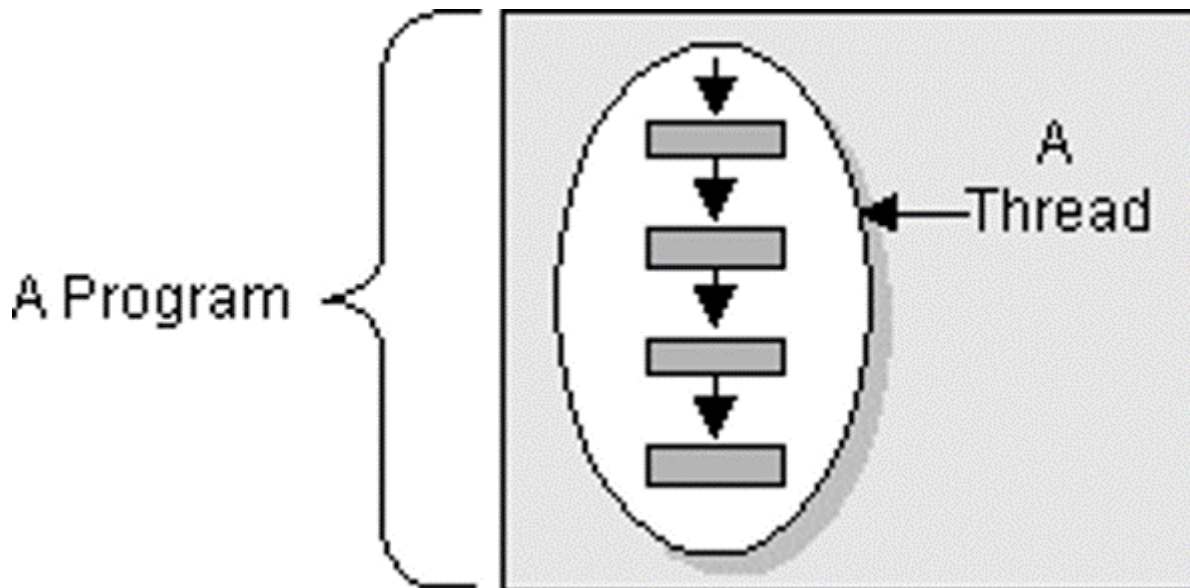
La comunicación entre hilos es ligera ya que no necesitan ningún mecanismo adicional para comunicarse información entre ellos ya que todos pueden ver la información que hay en la memoria del proceso. Se necesitan mecanismos de sincronización adicionales para evitar problemas de acceso.

Ejemplos utilización de hilos:

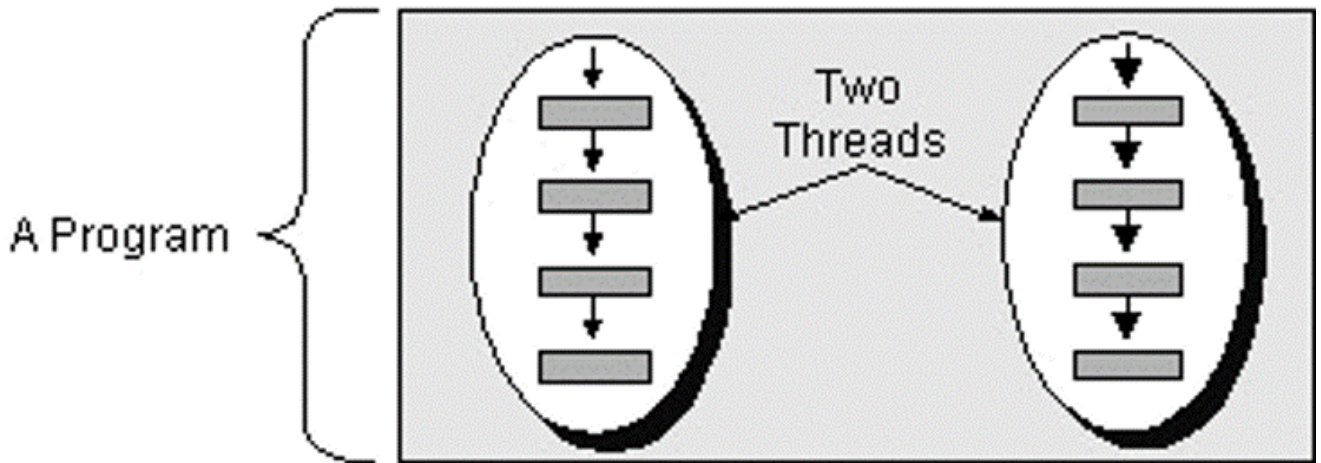
- Pestañas abiertas del navegador Google Chrome
- Word tiene un hilo comprobando automáticamente la gramática a la vez que está escribiendo un documento.
- Diferentes personajes de un videojuego moviéndose e interactuando independientemente.

## Hilos

Un hilo es un único flujo de control dentro de un programa. Algunas veces es llamado contexto de ejecución porque cada hilo debe tener sus propios recursos.



Dos hilos puede ejecutarse a la vez. Dos hilos concurrentes están en progreso, o intentando obtener tiempo de ejecución de la CPU al mismo tiempo, pero no necesariamente de forma simultánea.



Un hilo comparte con otros hilos la sección de código, datos, memoria y los archivos abiertos. Cada hilo tiene un contador de programa, un conjunto de registros de CPU y la [pila de ejecución](#).

## 1. Fundamentos de la programación multihilo

Un programa multihilo contiene dos o más partes que pueden ejecutarse de forma concurrente o secuencial.

Cada parte de ese programa se llama hilo (Thread) y cada hilo establece un camino de ejecución independiente.

Los programas con un único hilo se ejecutan de forma secuencial mientras que los programas multihilo pueden ejecutarse de manera concurrente.

Características principales de los hilos:

- Dependencia del proceso: no se pueden ejecutar independientemente.
- Ligereza: optimizan la utilización de recursos. Podemos crear muchos hilos sin tener pérdidas de memoria.
- Comparten recursos: Dentro de un proceso los hilos comparten espacio de memoria. Esto puede producir **errores de concurrencia**.

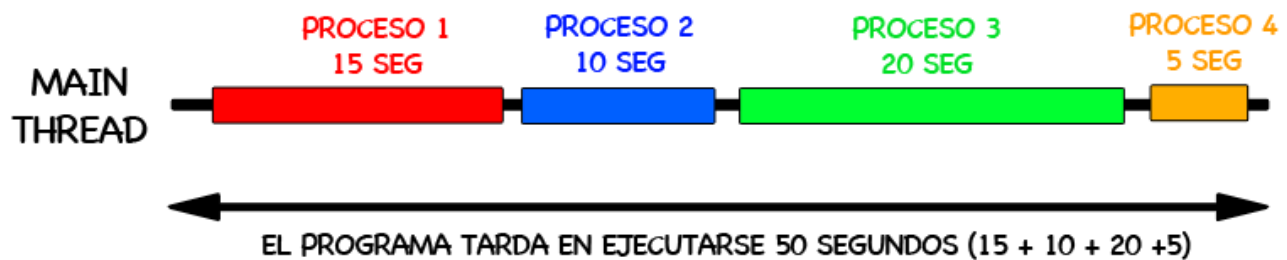
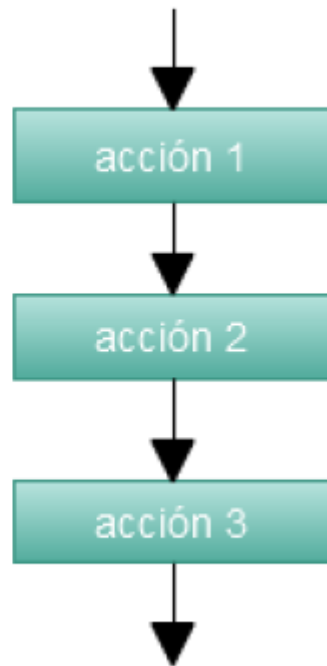
### 1.1 Programación secuencial (un hilo)

## Pseudocódigo

```

inicio
  accion 1
  accion 2
  accion 3
  .
  .
  .
  accion n
fin

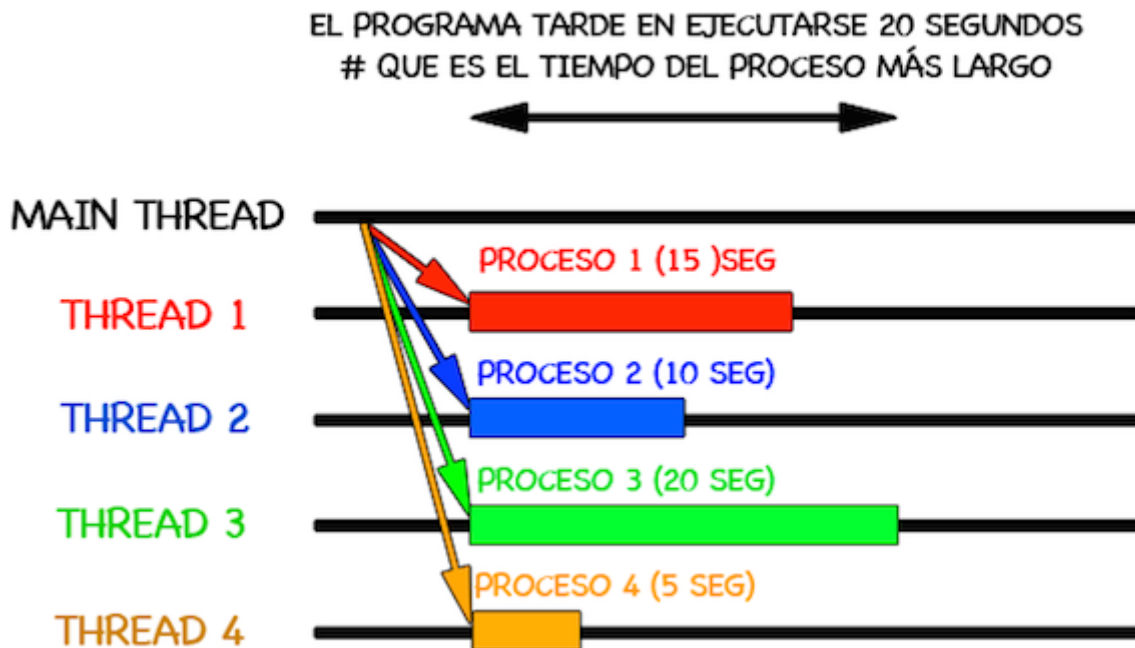
```



**Ejemplo 1:** Veremos un programa en Java que muestra como cualquier programa se ejecuta mediante un hilo. En el programa se declara un objeto de tipo Thread y se recupera la referencia mediante el método `currentThread()`. Una vez creado el hilo se puede pedir el nombre o cambiarlo o dormirlo o ver su prioridad....

### 1.2 Programación concurrente (multihilo)

La concurrencia reúne varios hilos de ejecución.



Cualquier programa a ejecutarse es un proceso que tiene un hilo de ejecución principal, este hilo puede a su vez crear nuevos hilos.

Para crear hilos se puede hacer de dos modos:

- Extendiendo de la clase `java.lang.Thread`.
- Implementando la interfaz `java.lang.Runnable`. Suele utilizar una instancia a la clase `Thread`.

**Ejemplo 2:** Vemos un ejemplo de cómo se crean dos hilos extendiendo de la clase `thread`. La clase que implementa al hilo extiende de `Thread`. Se definen los constructores y un método `run` y se crean los hilos. Para lanzar los hilos se utiliza el método `start`.

Explicación del ejercicio 2:

- Al llamar a `start` el hilo se registra en el planificador de hilos de Java. Este decide que hilo ha de ejecutarse en cada momento.
- El planificador suele utilizar política **FIFO**.
- `Start` no significa que se ejecute el hilo, si no que lo pone en estado preparado-para-ejecución.
- Cada hilo compete para entrar a ejecutarse.
- A priori, no se sabe que hilo va a ejecutarse antes.

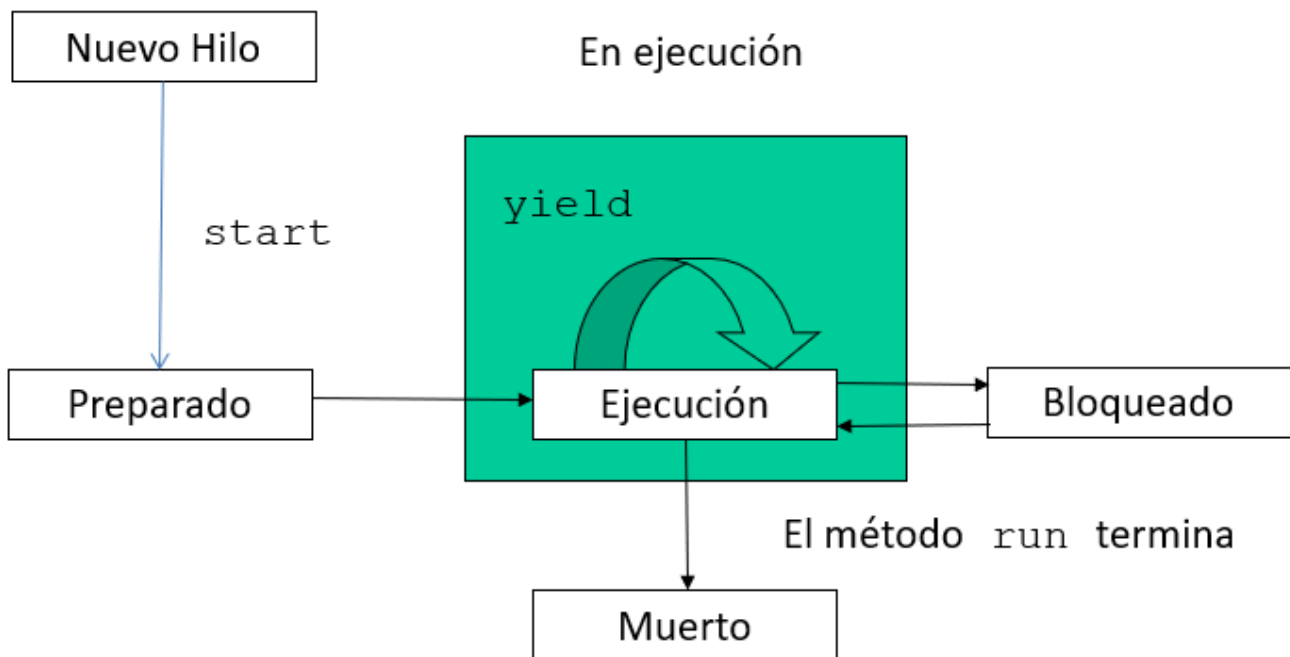
**Ejemplo 3:** Ejemplo de cómo se crean hilos mediante la interfaz `Runnable`. Veremos el mismo ejemplo de antes pero con la interfaz `Runnable`.

Normalmente trabajaremos utilizando directamente la clase `Thread`. `Runnable` se utiliza si la clase ya extiende de otra.

### 1.3 Estados de los hilos

- Estado **nuevo**: el hilo se ha creado, se ha hecho el `new()`, pero no se ha invocado a `start()`.

- Al invocar a `start()` pasa a **preparado** para ejecutarse.
- Al entrar en la CPU, pasa al estado **ejecución**.
- **Bloqueado**: el hilo no se ejecuta porque está a la espera de un evento. Cuando sucede el evento, pasa a preparado para ejecutarse. Se puede bloquear por dos motivos:
  - **Dormido**: se bloquea por un tiempo determinado.
  - **Esperando**: se bloquea esperando la llegada de un evento. Puede ser la recepción de un mensaje, la finalización de una tarea de E/S o acceso a un método sincronizado.
- **Muerto**: Pasa a este estado cuando finaliza su método `run()` o recibe un mensaje `interrupt()`. De este estado no puede ir a ningún otro estado.



## Sleep

### Ejemplo 4

## Join

### Ejemplo 5

## 2. Sincronización de hilos

Todos los hilos de un programa pertenecen a un mismo proceso, por lo que comparten variables y objetos en memoria. Si varios hilos manipulan objetos comunes, puede llevar a resultados erróneos. Para solucionar esto es necesario sincronizar.

Existen varios mecanismos de sincronización:

- Región crítica
- Monitores
- Semáforos

### 2.1 Región crítica

La región crítica de un programa multihilo es el bloque de código que accede a recursos compartidos, por lo que limitaremos el acceso a esta sección a un único hilo en ejecución.

Es importante determinar correctamente la sección crítica para evitar errores de concurrencia. Además es importante hacerlo eficientemente para aprovechar el paralelismo.

**Ejemplo 6a:** programa que crea cuatro hilos que acceden a una variable contador. Cada uno de los hilos aumenta el contador en una unidad. No usa herramientas de sincronización. ¿Funciona correctamente?

Para sincronizar un proceso se utiliza la palabra reservada **Synchronized**. La región crítica (la zona a proteger) se puede crear en el método o en el objeto.

**Ejemplo 6b:** Modificamos el ejemplo anterior y sincronizamos el método protegiendo la región crítica mediante la palabra clave Synchronized.

**Ejemplo 6c:** Ahora sincronizamos un fragmento de código especificando cual es el objeto que queremos que sea la región crítica.

## 2.2 Monitores (productor-consumidor)

Es un método de sincronización que podemos utilizar cuando uno o más hilos producen datos y otros hilos consumen dichos datos. Si los hilos que producen los datos van a diferente velocidad que los que consumen, puede que se salten datos.

En estos ejercicios tendremos un clase común donde se harán las operaciones de producir y consumir.

- Se definirá una clase hilo que ejecuta la operación de producir
- Se definirá una clase hilo que ejecuta la operación de consumir.

El objeto que controla los hilos que acceden a ese objeto mediante estos mecanismos se llama **monitor**.

Utilizaremos la sinstrucciones **wait()**, **notify()** y **notifyAll()** para limitar el acceso a un objeto cuando no se den las condiciones de seguir ejecutándose, y permitir su ejecución cuando las condiciones se cumplan.

Vamos a ver un ejemplo de problema productor-consumidor que sigue el siguiente esquema:



Con este método podemos afrontar problemas en los que hay uno o varios hilos productores, encargados de llenar un recipiente, y uno o varios hilos consumidores.

- Un hilo productor llenará el recipiente siempre que este vacío. Si está lleno, debe bloquearse. El productor saldrá del estado de bloqueo cuando se le notifique que el recipiente está vacío.

```

SI recipiente_lleno ENTONCES
    Esperar_vaciar_recipiente
EN OTRO CASO
    Llenar_recipiente
    Notificar_llenado_Recipiente

```

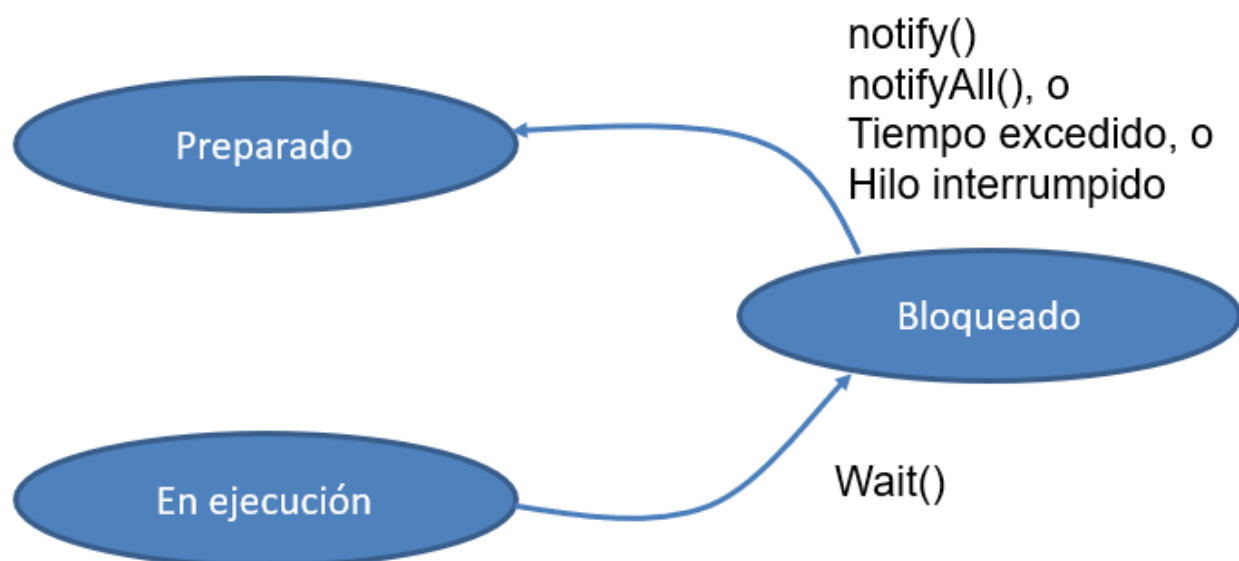
- Un hilo consumidor vaciará el recipiente si está lleno. Si no está lleno, se bloqueará a la espera de que se le notifique que se ha llenado. Una vez vacío el recipiente, debe notificar a los hilos que estaban esperando para llenar el recipiente, que este ha sido vaciado.

```

SI recipiente_vacio ENTONCES
    Esperar_llenar_recipiente
EN OTRO CASO
    Vaciar_recipiente
    Notificar_vaciado_Recipiente

```

El acceso al objeto compartido (recipiente) debe de hacerse de forma sincronizada usando regiones críticas.



Cuando un hilo invoca a `wait()`, queda bloqueado a la espera de un evento. Puede ser la invocación de `notify`, `notifyAll`, superar el tiempo especificado en `Wait` o la interrupción del hilo.

**Ejemplo7\_Monitores:** Tenemos hilos productores y consumidores de datos y un recipiente. Cada hilo va a ejecutar diez veces su acción de llenar o vaciar.

## 2.3 Semáforos

Los semáforos se utilizan para controlar el acceso a un recurso. Se implementan con una cola de tareas o de condición a la cual se añaden los procesos que están en espera del recurso.

Sólo se permiten tres operaciones sobre un semáforo:



- Inicializar: se proporciona un valor inicial al semáforo igual al número de recursos disponibles. Este valor determina el número de hilos que pueden acceder a la vez a una región protegida.  
  
Si inicializamos a 1, solo 1 hilo de cada vez. Si inicializamos a otro valor, se permite más de un hilo a la vez.
- Espera (wait o acquire): El hilo que ejecute esta instrucción disminuye en 1 el valor del semáforo.  
  
Si el valor es menor que 0, no hay permisos disponibles. Se queda bloqueado hasta que haya permisos disponibles.
- Señal (signal o release): Cuando un proceso termina de ejecutar el recurso compartido, avisa de su liberación mediante un signal. Después aumenta el valor del semáforo. Si hubiera hilos bloqueados para entrar, podrán pasar a ejecutarse.

#### Métodos:

- Void Semaphore(int valor): Inicializa el semáforo. Indica el valor inicial del semáforo antes de comenzar su ejecución.
- Void acquire(): implementa la operación wait
- Void release(): implementa la operación signal. Desbloquea un hilo que esté esperando con wait()

#### Flujo de instrucciones de la sincronización con semáforos:

1. Indicar al constructor **Semaphore (int permisos)** el total de permisos que se pueden dar para acceder al mismo tiempo al recurso compartido. Este valor coincide con el número de hilos que pueden acceder a la vez al recurso.
2. Indicar al semáforo mediante el método **acquire()**, que queremos acceder al recurso, o bien mediante **acquire(int permisosAdquirir)** cuántos permisos se quieren consumir al mismo tiempo.
3. Indicar al semáforo mediante el método **release()**, que libere el permiso, o bien mediante **release(int permisosLiberar)**, cuantos permisos se quieren liberar al mismo tiempo.

También podemos utilizar otro constructor **Semaphore (int permisos, boolean justo)** que mediante el parámetro justo permite garantizar que el primer hilo en invocar acquire() será el primero en adquirir un permiso cuando sea liberado. Esto garantiza el orden de adquisición de permisos, según el orden en que se solicitan.

#### Ejemplo 8 (sin semáforos):

Vamos a ver una aplicación ficticia corriendo sin el uso de los semáforos, para luego entender mejor su utilidad:

Tenemos 4 procesos (p1, p2, p3, p4), cada proceso realiza su "tarea" simultáneamente (durante un tiempo indefinido) y posteriormente termina.

Supongamos que necesitamos que se ejecuten primero los procesos P1 y P3, y luego P2 y P4.

Los procesos se ejecutan sin cumplir la condición más importante: que se ejecuten primero los procesos P1 y P3, y posteriormente P2 y P4.

Solucionemos esto con el uso de semáforos Este ejercicio es un ejemplo de uso de semáforos para comunicar hilos.

Aplicaciones de la clase Semaphore:

1. Comunicar hilos (Establecer un orden de ejecución entre los distintos hilos)

Vamos a usar la clase Semaphore, del paquete `java.util.concurrent`. Lo primero es el método `acquire()` de la clase Semaphore. Este método bloquea el semáforo permanentemente (`wait`).

El método `release()` de la clase Semaphore, libera el semáforo para los demás procesos (`signal`).

Ejemplo 9. Uso Semaforos

Modifica el ejemplo 9 para que el orden de ejecución de los hilos sea siempre P1, P4, P2, P3. Utiliza tantos semáforos como creas necesarios.

2. Protección de zonas críticas

Se trata de simular el acceso simultáneo de 4 terminales a un Servidor, pero resuelto ahora con la clase Semaphore en vez de con `synchronized`.

Ejemplo 10 Acceso Servidor

3. Comunicar hilos según el número de permisos

Es un ejemplo de Lectores-Escritores en una BD. Se inician 5 hilos lectores y 2 escritores.

La diferencia con los anteriores es que el semáforo permite hasta cinco accesos simultáneos a la base de datos. Los Escritores necesitan los 5 permisos para acceder, y los lectores solo 1 permiso para acceder.

Ejemplo 11 LectoresEscritores

Hacer hoja de ejercicios