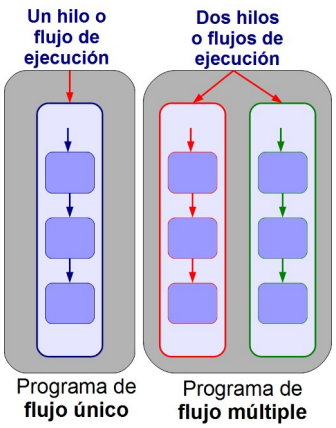
1. Introducción

Seguro que en más de una ocasión mientras te descargabas una imagen desde tu navegador web, seguías navegando por Internet e incluso iniciabas la descarga de un nuevo archivo, y todo esto ejecutándose el navegador como un único proceso, es decir, teniendo un único ejemplar del programa en ejecución.



Pues bien, ¿Cómo es capaz de hacer el navegador web varias tareas a la vez? Seguro que estarás pensando en la **programación concurrente**, y así es, pero con un nuevo enfoque de la concurrencia, denominado "**programación multihilo**". Justo lo que vamos a estudiar en esta unidad.

Los programas realizan actividades o tareas, y para ello pueden seguir uno o más flujos de ejecución. Dependiendo del número de flujos de ejecución, podemos hablar de dos tipos de programas:

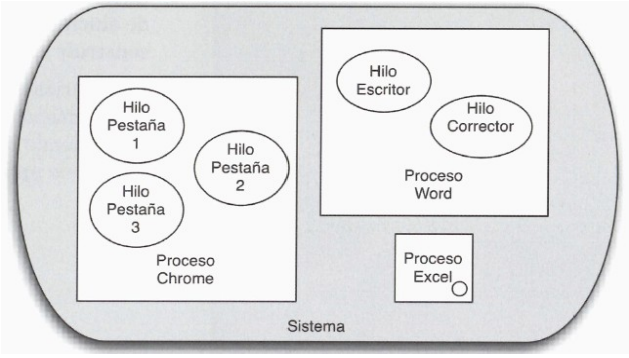
- **Programa de flujo único.** Es aquel que realiza las actividades o tareas que lleva a cabo una a continuación de la otra, de manera **secuencial**, lo que significa que cada una de ellas debe concluir por completo, antes de que pueda iniciarse la siguiente.

- **Programa de flujo múltiple.** Es aquel que coloca las actividades a realizar en diferentes flujos de ejecución, de manera que cada uno de ellos se inicia y termina por separado, pudiéndose ejecutar éstos de manera concurrente.

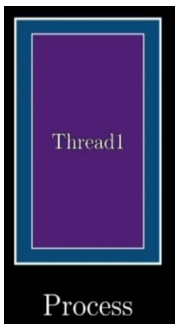
La **programación multihilo** o **multithrearing** consiste en desarrollar programas o aplicaciones de flujo múltiple. Cada uno de esos flujos de ejecución es un **thread** o **hilo**.

En el ejemplo anterior sobre el navegador web, un hilo se encargaría de la descarga de la imagen, otro de continuar navegando y otro de iniciar una nueva descarga. La utilidad de la programación multihilo resulta evidente en este tipo de aplicaciones. El navegador puede realizar "a la vez" estas tareas, por lo que no habrá que esperar a que finalice una descarga para comenzar otra o seguir navegando.

Cuando decimos "a la vez" recuerda que nos referimos a que las tareas se realizan concurrentemente, pues el que las tareas se ejecuten realmente en paralelo dependerá del Sistema Operativo y del número de procesadores del sistema donde se ejecute la aplicación. En realidad, esto es transparente para el programador y usuario, lo importante es la sensación real de que el programa realiza de forma simultánea diferentes tareas.



2. Conceptos sobre hilos

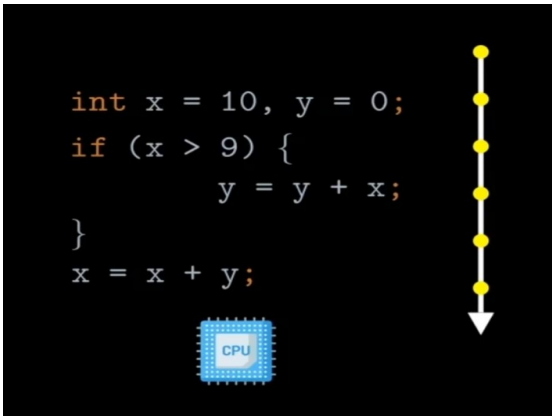


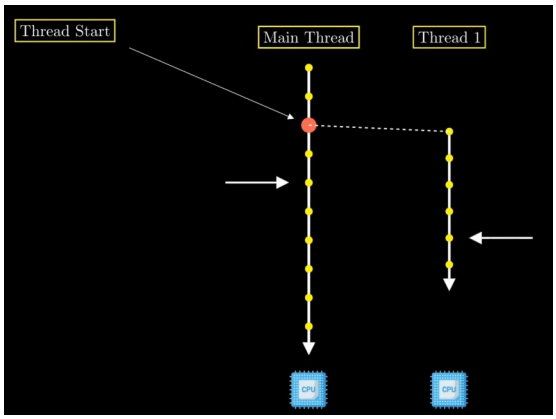
Pero ¿qué es realmente un hilo? Un **hilo** es un flujo de control secuencial independiente dentro de un proceso y está asociado con una secuencia de instrucciones, un conjunto de registros y una pila.

Cuando se ejecuta un programa, el Sistema Operativo crea un proceso y también crea su primer hilo, **hilo primario**, el cual puede a su vez crear hilos adicionales.

Desde este punto de vista, un proceso no se ejecuta, sino que solo es el espacio de direcciones donde reside el código que es ejecutado mediante uno o más hilos.

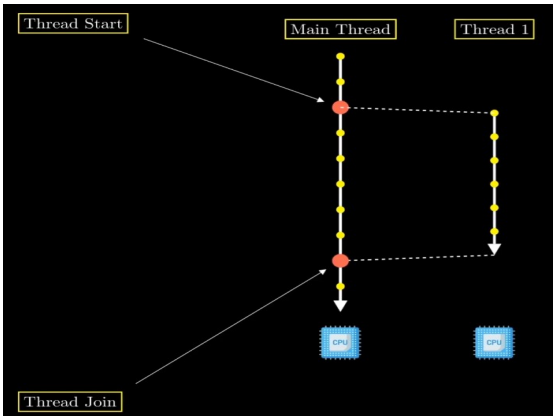
Flujo único	Flujo múltiple: Dos hilos independientes

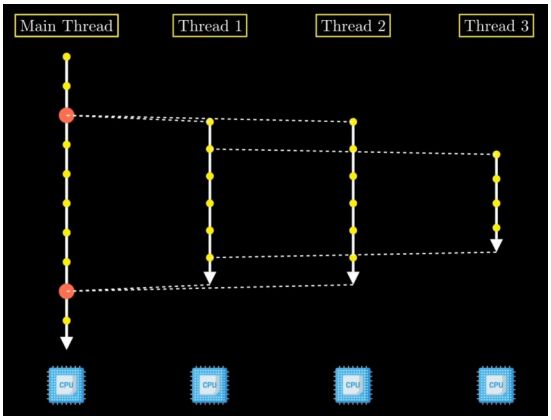




Cuando un programa Java se lanza (se convierte en un proceso) empieza a ejecutarse por su método ***main()*** que lo ejecuta el thread principal (**Main Thread**), un hilo especial creado por la JVM para ejecutar la aplicación.

Flujo múltiple: Dos hilos dependientes	Flujo múltiple: Cuatro hilos dependientes

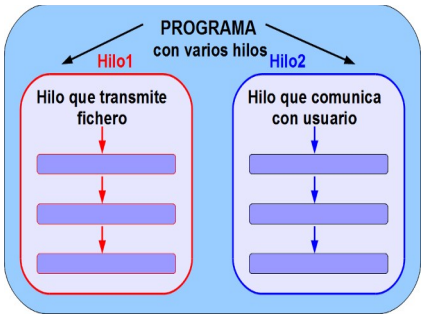




Desde un proceso se pueden crear e iniciar tantos threads como sean necesarios. Estos hilos ejecutarán partes del código de la aplicación en paralelo con el thread principal.

Por lo tanto podemos hacer las siguientes **observaciones**:

- Un hilo no puede existir independientemente de un proceso. Por tanto, si el proceso finaliza por alguna circunstancia, todos sus hilos también lo hacen.
- Un hilo no puede ejecutarse por si solo.
- Dentro de cada proceso puede haber varios hilos ejecutándose.

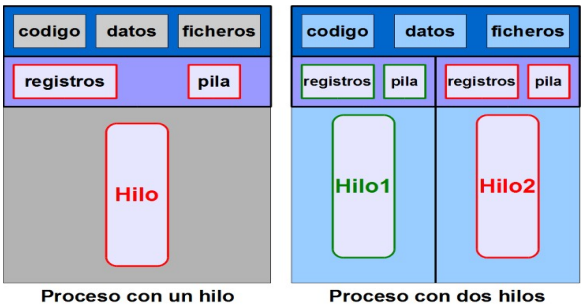


Un único hilo es similar a un programa secuencial, por si mismo no nos ofrece nada nuevo. Es la habilidad de ejecutar varios hilos dentro de un proceso lo que ofrece algo nuevo y útil ya que cada uno de estos hilos puede ejecutar actividades diferentes al mismo tiempo. Así en un programa un hilo puede encargarse de la comunicación con el usuario, mientras que otro hilo transmite un fichero, otro puede acceder a recursos del sistema (cargar sonidos, leer ficheros, …), etc.

2.1. Recursos compartidos por los hilos

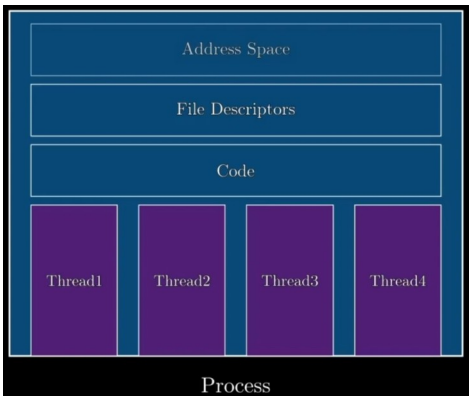
Un hilo lleva asociados los siguientes elementos:

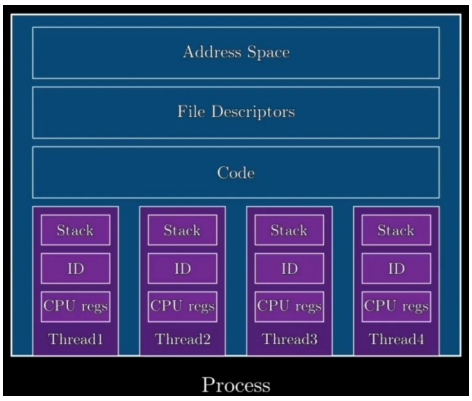
- Un identificador único que lo identifica dentro del proceso.
- Un contador de programa propio.
- Un conjunto de registros de la CPU.
- Una pila (usada para almacenar variables locales y parámetros de métodos).

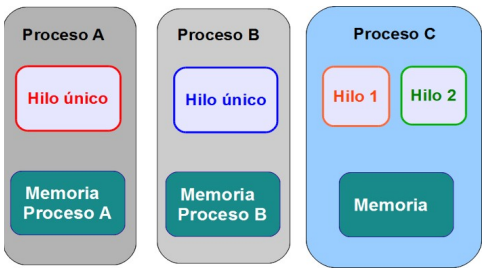


Por otra parte, **un hilo puede compartir con otros hilos del mismo proceso los siguientes recursos:**

- Código.
- Espacio de direcciones usado por el proceso.
- Datos (como variables globales).
- Otros recursos del sistema operativo, como los ficheros abiertos.

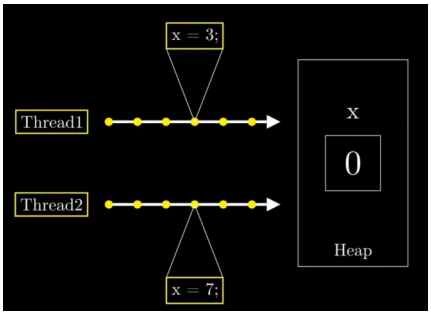






Seguro que te estarás preguntando "si los hilos de un proceso comparten el mismo espacio de memoria, ¿qué pasa si uno de ellos la corrompe?" La respuesta es que los otros hilos también sufrirán las consecuencias. Recuerda que en el caso de procesos, el sistema operativo normalmente protege a un proceso de otro y si un proceso corrompe su espacio de memoria los demás no se verán afectados

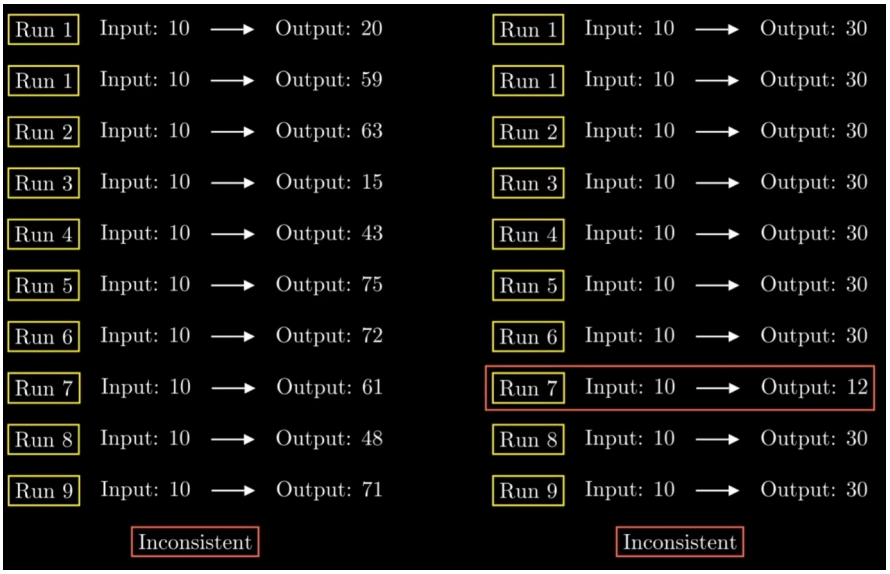
El hecho de que los hilos compartan recursos (por ejemplo, pudiendo acceder a las mismas variables) implica que sea necesario **utilizar esquemas de bloqueo y sincronización,** lo que puede hacer más difícil el desarrollo de los programas y así como su depuración.



Por ejemplo, si dos hilos quieren, en el mismo instante de tiempo, actualizar con diferentes valores una variable x compartida, es necesario garantizar el orden de la operación para tener un **comportamiento consistente**. De lo contrario se tendría un resultado impredecible, algo que debe evitarse cuando se implementan aplicaciones multihilo ya que provocará que se tengan diferentes salidas en diferentes ejecuciones.

COMPORTAMIENTO INCONSISTENTE

COMPORTAMIENTO INDETERMINISTA



Realmente, es en la sincronización de hilos donde reside el arte de programar con hilos ya que de no hacerlo bien, podemos crear una aplicación totalmente ineficiente o inútil, como por ejemplo, programas que tardan horas en procesar servicios, que se bloquean con facilidad o que intercambian datos de manera equivocada.

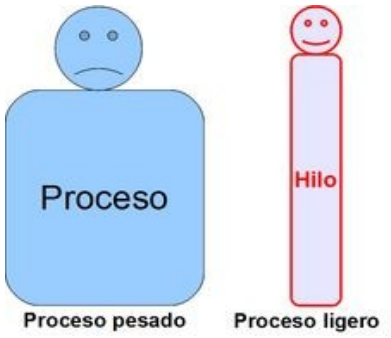
Profundizaremos más adelante en la sincronización, comunicación y compartición de recursos entre hilos dentro del contexto de Java.

Pero ¿qué ventajas aportan los hilos y cuando deben utilizarse?

2.2. Ventajas y usos de hilos

Como consecuencia de compartir el espacio de memoria, los hilos aportan las siguientes **ventajas sobre los procesos**:

- Se consumen menos recursos en el lanzamiento y la ejecución de un hilo que en el lanzamiento y ejecución de un proceso.
- Se tarda menos tiempo en crear y terminar un hilo que un proceso.
- La conmutación entre hilos del mismo proceso o **cambio de contexto** es bastante más rápida que entre procesos.



Es por esas razones, por lo que a los hilos se les denomina también **procesos ligeros**.

Y ¿**cuándo se aconseja utilizar hilos**? Se aconseja utilizar hilos en una aplicación cuando:

- La aplicación maneja entradas de varios dispositivos de comunicación.
- La aplicación debe poder realizar diferentes tareas a la vez.
- Interesa diferenciar tareas con una prioridad variada. Por ejemplo, una prioridad alta para manejar tareas de tiempo crítico y una prioridad baja para otras tareas.
- La aplicación se va a ejecutar en un entorno multiprocesador.

Por ejemplo, imagina la siguiente situación:

Debes crear una aplicación que se ejecutará en un servidor para atender peticiones de clientes. Esta aplicación podría ser un servidor de bases de datos, o un servidor web.

- Cuando se ejecuta el programa éste abre su puerto y queda a la escucha, esperando recibir peticiones.
- Si cuando recibe una petición de un cliente se pone a procesarla para obtener una respuesta y devolverla, cualquier petición que reciba mientras tanto no podrá atenderla, puesto que está ocupado.

La solución será construir la aplicación con múltiples hilos de ejecución.

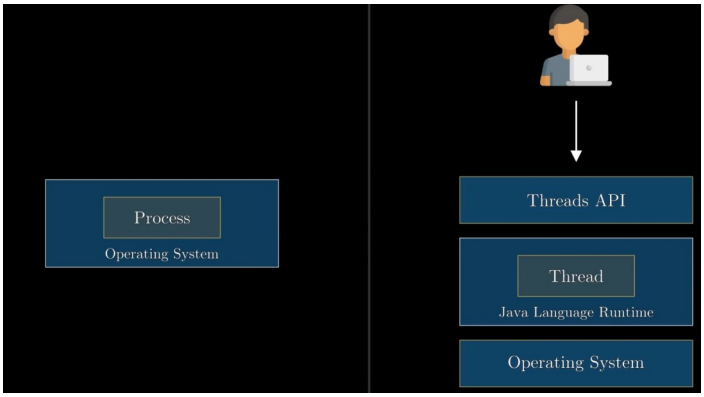
- En este caso, al ejecutar la aplicación se pone en marcha el hilo principal, que queda a la escucha.
- Cuando el hilo principal recibe una petición, creará un nuevo hilo que se encarga de procesarla y generar la consulta, mientras tanto el hilo principal sigue a la escucha recibiendo peticiones y creando hilos.
- De esta manera un gestor de bases de datos puede atender consultas de varios clientes, o un servidor web puede atender a miles de clientes.
- Si el número de peticiones simultáneas es elevado, la creación de un hilo para cada una de ellas puede comprometer los recursos del sistema. En este caso, lo resolveremos mejor con un pool de hilos (contenedor de hilos).

Resumiendo, los hilos son idóneos para programar aplicaciones de entornos interactivos y en red, así como simuladores y animaciones.

2.3. Threads Vs Procesos

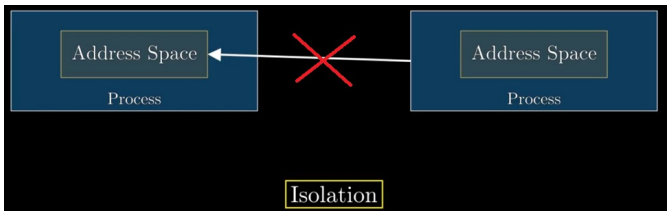
Primera diferencia: Quién los gestiona

Los procesos son objetos con los que se trabaja a nivel de sistema operativo. En cambio, las hebras son objetos con los que se trabaja a nivel de ***Java Runtime*** (se ejecuta un nivel por encima del Sistema Operativo). Por tanto, será el ***Java Runtime*** el encargado de gestionar los hilos y todas sus complejidades.



Segunda diferencia: Aislamiento.

Cada proceso está asociado a una aplicación en ejecución, teniendo cada uno de ellos su propio espacio de direcciones al cual no va a poder acceder ningún otro proceso (**aislamiento**).

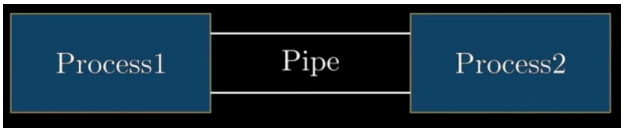


Tercera diferencia: La comunicación.

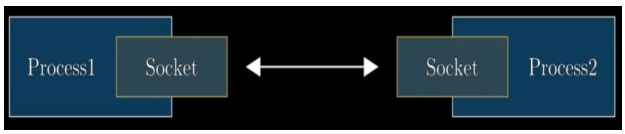
¿Cómo pueden comunicarse los procesos? Como se ha visto en la unidad anterior, si ambos procesos están ejecutándose en la misma máquina, una de las formas más fáciles de lograrlo es mediante el uso de ficheros. Por ejemplo, el proceso 1 produce unos datos que necesita pasar el proceso 2. Para ello los escribe en un fichero y el proceso 2, los lee desde fichero.

En esta solución, ambos procesos necesitan saber el nombre del fichero, tener los permisos correspondiente y acceso exclusivo para que uno no intente leer antes de que el otro haya escrito.

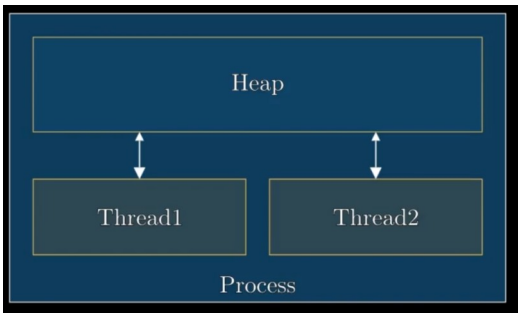
También se puede usar el concepto de Pipe visto en la unidad anterior.



En cambio, si los dos procesos están en máquinas diferentes se puede recurrir al uso de sockets que serán vistos en la unidad siguiente.



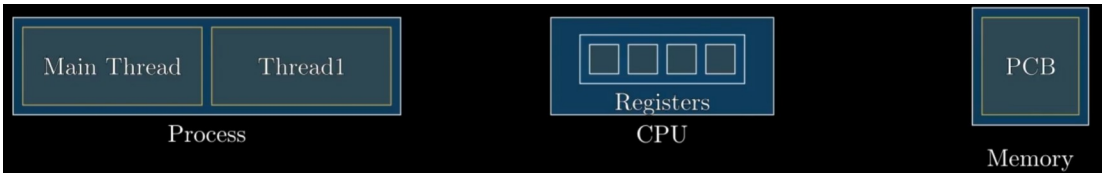
A la hora de llevar a cabo una **comunicación entre hilos**, esta se puede llevar a cabo de un modo muy simple debido a que los hilos pueden compartir el espacio de direcciones del proceso que los crea:



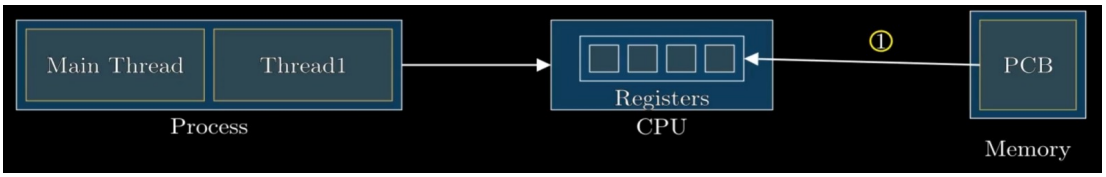
Por tanto, no son necesarios métodos sofisticados ni complejos para efectuar el intercambio de datos ya que los distintos hilos de un mismo proceso comparten la memoria asignada al proceso. Sin embargo, los **hilos deben coordinarse para el acceso a los contenidos de la memoria y a los ficheros, lo cual hace que esa coordinación y sincronización sea la parte complicada de uso**.

Cuarta diferencia: El cambio de contexto entre procesos es más costoso que el cambio de contexto entre hilos.

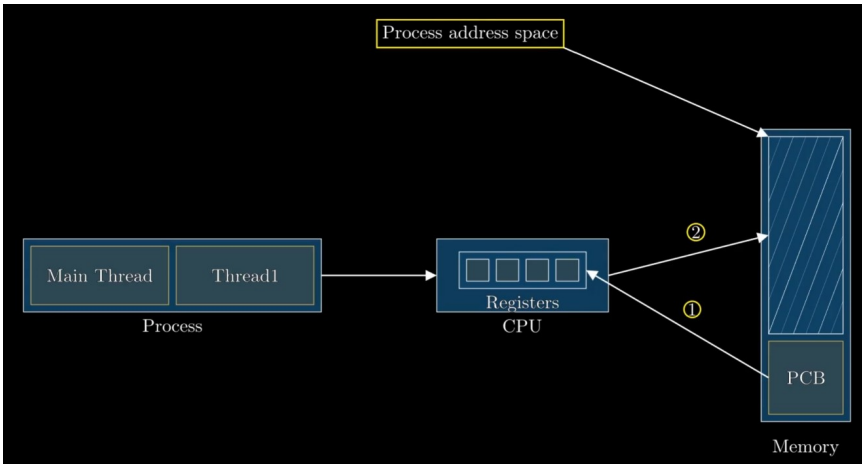
Partimos de una situación en la que se tiene un proceso con su hilo principal y que crea un nuevo hilo.



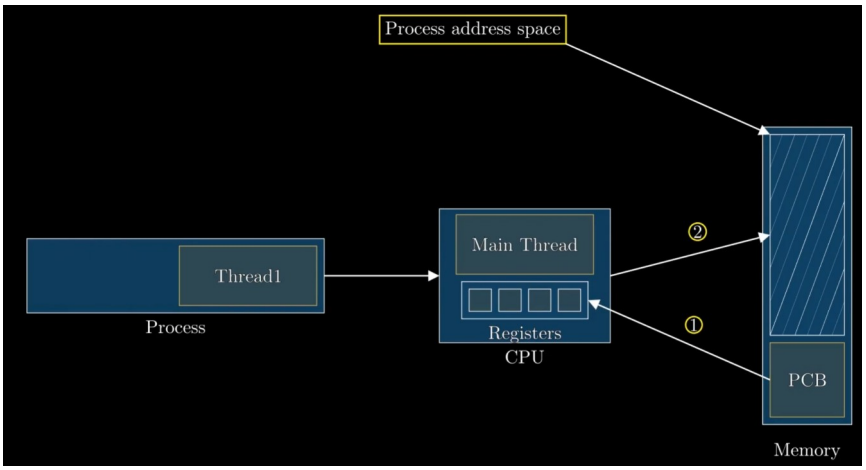
Este proceso está activo y por tanto, cuando el scheduler del sistema operativo le asigna la CPU, se cargan en los registros de la CPU toda la información necesaria leída del PCB.



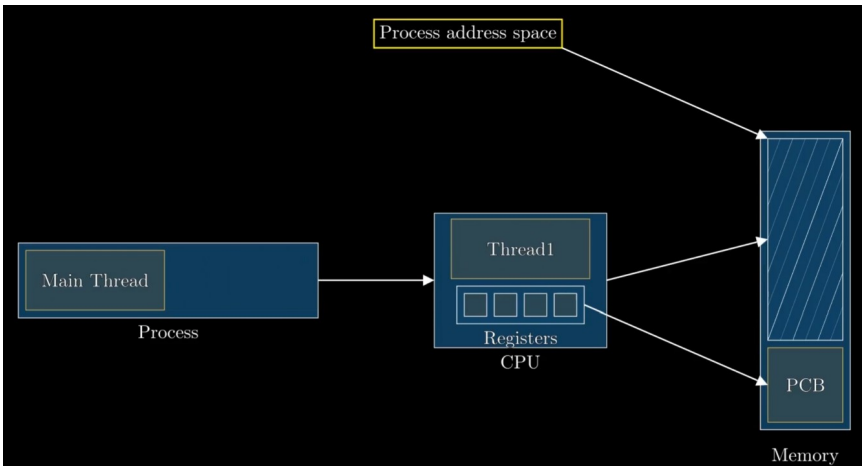
Además, se reserva espacio en memoria para cargar todo el espacio de direcciones del proceso.



A continuación el hilo principal comienza su ejecución.



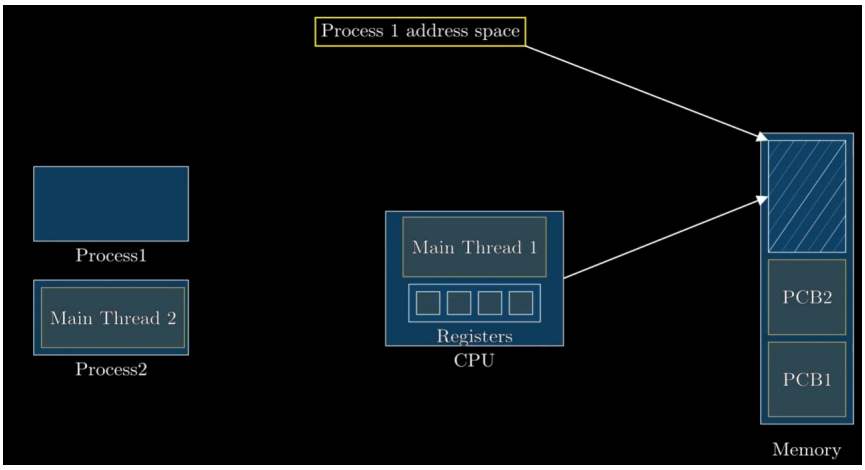
Cuando la CPU vaya a ejecutar el hilo Thread1, el sistema operativo debe actualizar el PCB del proceso, sacar del procesador al hilo principal y asignar el procesador al hilo Thread1.



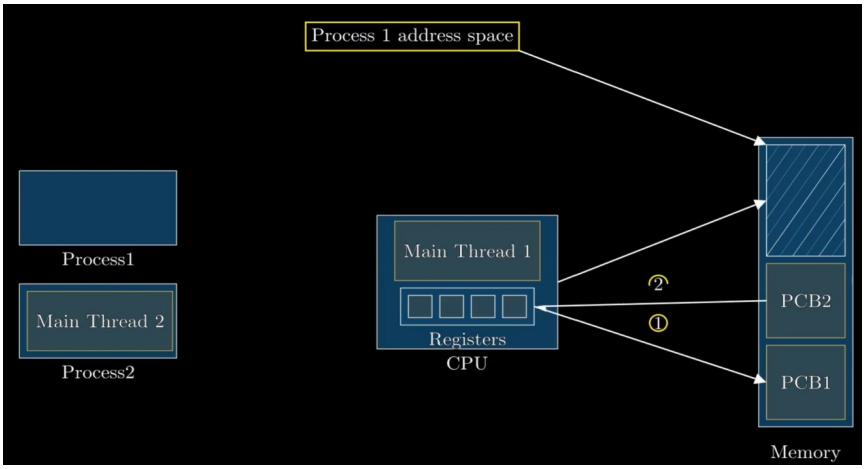
Esto que se acaba de realizar, es lo que se conoce como **cambio de contexto del hilo**.

Supongamos ahora que, en lugar de un proceso con dos hilos, tenemos dos procesos con sus respectivos hilos principales. Ambos procesos están activos, y por consiguiente, a cualquiera de los dos se le podrá asignar la CPU.

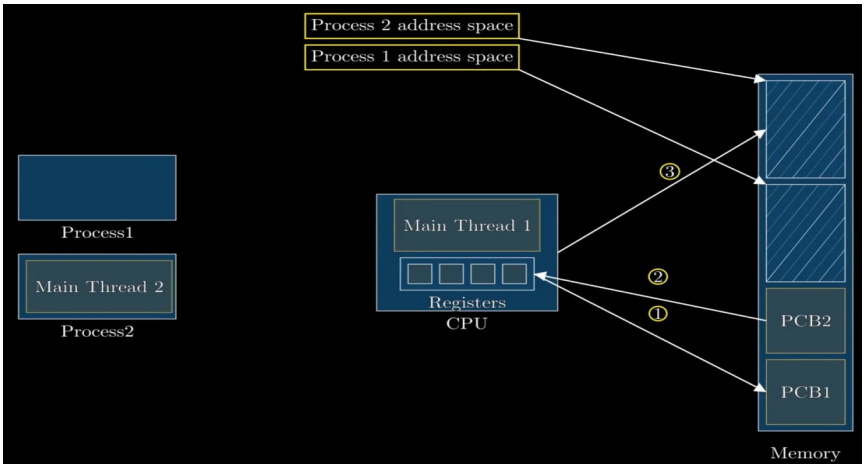
El scheduler del sistema operativo decide asignar la CPU al primero de los procesos:



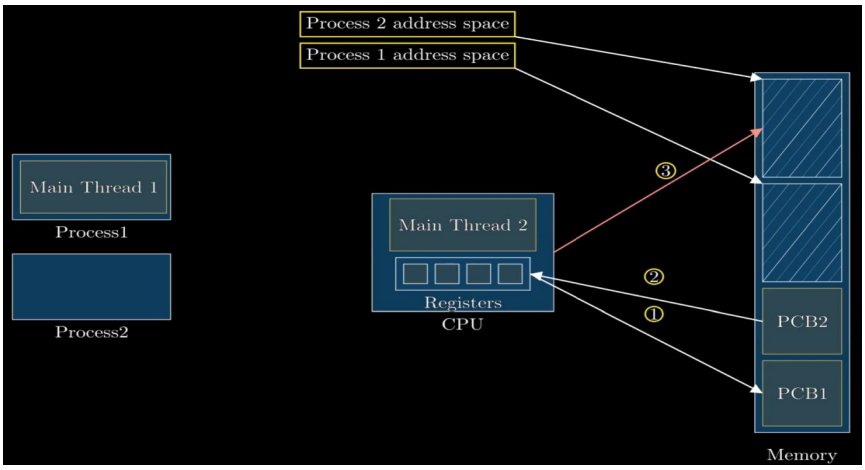
Si el scheduler le arrebata la CPU al proceso 1 para asignarla al proceso 2, debe haber un cambio de contexto a nivel de proceso. Por tal motivo, debe guardar todo el contexto del proceso 1 en su PCB (PCB1). Además, también deberá cargar en los registros de la CPU toda la información necesaria del proceso 2, leyéndola del PCB2.



Además, necesita cargar en memoria el espacio de direcciones del proceso 2:



Finalmente, el hilo principal del proceso 2 comenzará/continuará con su ejecución:



Se puede observar que el cambio de contexto entre procesos, supone una sobrecarga en comparación con el cambio de contexto entre hilos.

3. Multihilos en Java. Librería y clases

Java da soporte al concepto de hilo desde el propio lenguaje, con algunas **clases e interfaces** definidas en el paquete java.lang y con **métodos** específicos para la manipulación de hilos en la clase Object. A partir de la versión Java 5.0, se incluye el paquete java.util.concurrent con nuevas utilidades para desarrollar aplicaciones multihilo e incluso aplicaciones con un alto nivel de concurrencia.

3.1. Utilidades de concurrencia del paquete java.lang

Dentro del **paquete** java.lang disponemos de una interfaz y las siguientes clases para trabajar con hilos:

- **Clase Thread**. Es la clase responsable de producir hilos funcionales para otras clases y proporciona gran parte de los métodos utilizados para su gestión.

  Tiene algunos métodos estáticos importantes, entre los que se pueden destacar:

  - ***currentThread***: Nos devuelve un objeto de tipo ***Thread*** a través del cual se puede acceder a toda la información sobre la hebra que está actualmente ejecutando el código.
  - ***sleep***: Provoca que el hilo se duerma (se suspenda) una determinada cantidad de tiempo en milisegundos. Necesita tratar la excepción ***InterruptedException***.
  - ***getName***: Nos devuelve el nombre del hilo. Por defecto, el nombre del hilo principal se llama ***main***. En caso de no asignarle ningún nombre, a los hilos que se vayan creando se les asignará un nombre por defecto con el formato Thread-xx, siendo xx un número que empieza en 0.
  - ***setName***: Permite cambiar el nombre del hilo.
  - ***start***: Método que inicia o arranca el hilo.

  Ver: MetodosEstaticosThreads.

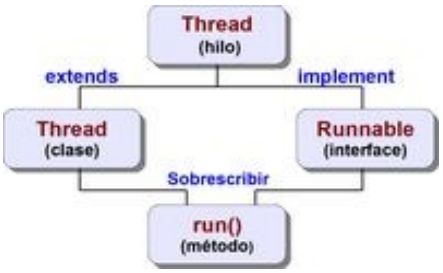
- **Interfaz Runnable**. Proporciona la capacidad de añadir la funcionalidad de hilo a una clase simplemente implementando la interfaz, en lugar de derivándola de la clase thread.
- **Clase ThreadDeath.** Es una clase de error, deriva de la clase Error, y proporciona medios para manejar y notificar errores.
- **Clase ThreadGroup.** Esta clase se utiliza para manejar un grupo de hilos de modo conjunto, de manera que se pueda controlar su ejecución de forma eficiente.
- **Clase Object.** Esta clase no es estrictamente de apoyo a los hilos, pero proporciona unos cuantos métodos cruciales dentro de la arquitectura multihilo de Java. Estos métodos son wait(), notify() y notifyAll().

4. Creación de hilos

En Java, un **hilo** se representa mediante una instancia de la clase java.lang.thread. Este **objeto thread** se emplea para iniciar, detener o cancelar la ejecución del hilo de ejecución.

Los hilos o threads se pueden **implementar o definir de dos formas**:

- Extendiendo la clase Thread.
- Implementando la interfaz Runnable.



En **ambos casos, se debe proporcionar una definición del método** *run(),* ya que este método es el que contiene el código que ejecutará el hilo, es decir, su comportamiento.

El procedimiento de construcción de un hilo es independiente de su uso, pues una vez creado se emplea de la misma forma.

Constructores	
Thread()	Crea una hebra que hereda de Thread y su nombre es asignado por Java.
Thread(Runnable target)	Crea una hebra que implementa la interfaz Runnable y su nombre es asignado por Java.
Thread(String name)	Crea una hebra que hereda de Thread y cuyo nombre es name.
Thread(Runnable target, String name)	Crea una hebra que implementa la interfaz Runnable y cuyo nombre es name

Entonces, ¿**cuando utilizar uno u otro procedimiento?** No hay nada que indique que una forma es mejor que otra. Ambos métodos son similares y el resultado es el mismo.

- Extender la clase Thread es el procedimiento más sencillo, pero no siempre es posible. Si la clase ya hereda de alguna otra clase padre, no será posible heredar también de la clase Thread (recuerda que Java no permite la herencia múltiple), por lo que habrá que recurrir al otro procedimiento.
- El método preferido debería ser implementar Runnable, y pasarle la instancia al constructor de Thread. Implementar Runnable siempre es posible, es el procedimiento más general y también el más flexible.

RECUERDA: Cuando la Máquina Virtual Java (JVM) arranca la ejecución de un programa, ya hay un hilo ejecutándose, denominado **hilo principal del programa**, controlado por el método main(), que se ejecuta cuando comienza el programa y es el último hilo que termina su ejecución, ya que cuando este hilo finaliza, el programa termina.

4.1. Creación de hilos extendiendo de la clase Thread

Para **definir** y **crear un hilo extendiendo la clase Thread**, haremos lo siguiente:

- Implementar una nueva clase que herede de la clase Thread.
- Redefinir en la nueva clase el método run() con el código asociado al hilo. Son las sentencias que ejecutará el hilo.
- Declarar y crear un objeto de la nueva clase Thread. Éste será realmente el hilo.

Una vez creado el hilo, para **ponerlo en marcha o iniciarlo**:

- Invocar al método start() del objeto thread (el hilo que hemos creado).

Ver: CreacionHilosHeredando

4.2. Creación de hilos implementado la interfaz Runnable

Para **definir y crear hilos implementando la interfaz Runnable** seguiremos los siguientes pasos:

- Implementar una nueva clase que implemente Runnable.
- Redefinir en la nueva clase el método run() con el código asociado al hilo. Son las sentencias que ejecutará el hijo
- Declarar un objeto de la clase Thread y se llama al constructor de la clase Thread pasando al constructor un objeto cuya clase implementa Runnable. Este será realmente el hilo.

Una vez creado el hilo, para ponerlo en marcha o iniciarlo:

- Invocar al método start() del objeto thread (el hilo que hemos creado).

Ver: CreacionHilosRunnable; EjemploRunnable;

4.3. Creación de hilos mediante clases anónimas que implementan la interfaz Runnable

Para ello seguiremos los siguientes pasos:

- Declarar y crear un objeto de tipo Runnable.
- Redefinir el método run() con el código asociado al hilo.
- Declarar un objeto de la clase Thread y se instancia llamando al constructor de la clase Thread pasando al constructor, el objeto runnable previamente declarado y creado.

Una vez creado el hilo, para ponerlo en marcha o iniciarlo:

- Invocar al método start() del objeto thread (el hilo que hemos creado).

Ver: CreacionHilosAnonimoRunnable

4.4. Creación de hilos mediante expresiones lambda

Permite hacer lo mismo que en el caso anterior pero de una forma más elegante. Para ello seguiremos los siguientes pasos:

- Declarar un objeto de tipo Runnable y se le asocia la expresión lambda que incluye el código asociado al hilo.
- Declarar un objeto de la clase Thread y se instancia llamando al constructor de la clase Thread pasando al constructor, el objeto runnable previamente declarado y creado.

Una vez creado el hilo, para ponerlo en marcha o iniciarlo:

- Invocar al método start() del objeto thread (el hilo que hemos creado).

Esto tiene sentido usarlo en el caso de que la tarea la vamos a usar en un solo punto de la aplicación y no se va a volver a instanciar.

Ver: CreacionHilosExpresionLambda;

4.5. Iniciar un hilo

Cuando se crea un nuevo hilo mediante el método new(), esto no implica que el hilo ya se pueda ejecutar.

Para que el hilo se pueda ejecutar, debe estar en el estado "Ejecutable", y para conseguir ese estado es necesario **iniciar o arrancar el hilo** mediante el método start() de la clase Thread.

En realidad el método start() **realiza las siguientes tareas:**

- Crea los recursos del sistema necesarios para ejecutar el hilo.
- Se encarga de **llamar a su método** run().

Es por esto último que cuando se invoca a start() se suele decir que el hilo está **Runnable**, pero esto no significa que el hilo esté ejecutándose en todo momento, ya que **un hilo "Ejecutable" puede estar "Runnable" o "Running"** según tenga o no asignado tiempo de procesamiento.

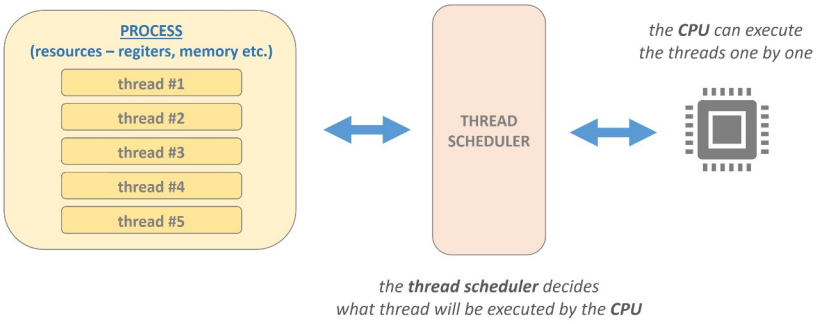
Algunas **consideraciones importantes** que debes tener en cuenta son las siguientes:

- Puedes invocar directamente al método run(), por ejemplo poner hilo1.run(); y se ejecutará el código asociado a run() dentro del hilo actual (como cualquier otro método), pero no comenzará un nuevo hilo.
- Una vez que se ha llamado al método start() de un hilo, no puedes volver a realizar otra llamada al mismo método. Si lo haces, obtendrás una excepción IllegalThreadStateException.
- El orden en el que inicies los hilos mediante start() no influye en el orden de ejecución de los mismos, lo que pone de manifiesto que **el orden de ejecución de los hilos es no-determinístico** (no se conoce la secuencia en la que serán ejecutadas la instrucciones del programa).

Ver: CreacionVariosHilos

5. Prioridades de hilos

En Java, **cada hilo tiene una prioridad** representada por un valor de tipo entero **entre 1 y 10**. Cuanto mayor es el valor, mayor es la prioridad del hilo.



Por defecto, el **hilo principal** de cualquier programa, o sea, el que ejecuta su método main() siempre es creado con prioridad 5.

El resto de **hilos secundarios** (creados desde el hilo principal, o desde cualquier otro hilo en funcionamiento), **heredan la prioridad que tenga en ese momento su hilo padre**.

En la clase Thread se definen 3 constantes para manejar estas prioridades:

- MAX_PRIORITY (= 10). Es el valor que simboliza la máxima prioridad.
- MIN_PRIORITY (=1). Es el valor que simboliza la mínima prioridad.
- NORM_PRIORITY (= 5). Es el valor que simboliza la prioridad normal, la que tiene por defecto el hilo donde corre el método main().

Además en cualquier momento se puede **obtener** y **modificar la prioridad de un hilo**, mediante los siguientes métodos de la clase Thread:

- getPriority(). Obtiene la prioridad de un hilo. Este método devuelve la prioridad del hilo.
- setPriority(int prioridad). Modifica la prioridad de un hilo. Este método toma como argumento un entero entre 1 y 10, que indica la nueva prioridad del hilo.

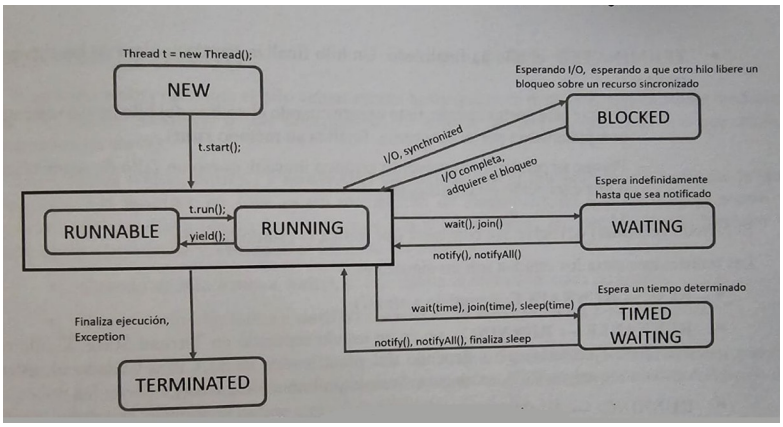
Por tanto, Java tiene 10 niveles de prioridad que no tienen por qué coincidir con los del sistema operativo sobre el que está corriendo. Por ello, lo mejor es que utilices en tu código sólo las constantes MAX_PRIORITY, NORM_PRIORITY y MIN_PRIORITY.

Podemos conseguir **aumentar el rendimiento de una aplicación multihilo** gestionando adecuadamente las prioridades de los diferentes hilos, por ejemplo utilizando una prioridad alta para tareas de tiempo crítico y una prioridad baja para otras tareas menos importantes.

Ver: PrioridadesHilos.

Ejemplo en el que se declara un hilo cuya tarea es llenar un vector con 20000 caracteres. Se inician 15 hilos con prioridades diferentes, 5 con prioridad máxima, 5 con prioridad normal y 5 con prioridad mínima. Al ejecutar el programa comprobarás que los hilos con prioridad más alta tienden a finalizar antes.

6. Estados de un hilo



El **ciclo de vida de un hilo** comprende los diferentes estados en los que puede estar desde que se crea hasta que finaliza.

- **NEW**

  Estado de un hilo cuando se crea con el operador *new*. En este estado el hilo aún no se ha iniciado, es decir, no se ha comenzado a ejecutar el código del método *run()* del hilo.

- **RUNNABLE**

  Estado de un hilo cuando está listo para ejecutarse, es decir, está esperando a que se le asigne la CPU. Pasará al estado de RUNNING cuando el scheduler de hilos lo selecciona y lo asigna a uno de los núcleos de la CPU, pasando a ejecutar las sentencias de su método *run()*.

- **BLOCKED**

  En este estado el hilo está bloqueado esperando a que otro hilo libere un recurso por el que compiten. También se encuentra en este estado aquellos hilos que están esperando a que se complete una operación de E/S.

- **WAITING**

  En este estado el hilo espera indefinidamente hasta que otro hilo realice una acción concreta. Pasará al estado de RUNNABLE cuando otro hilo realice una notificación (*notify*).

- **TIMED_WAITING**

  En este estado se encuentran aquellos hilos que están esperando un tiempo específico a que otro hilo realice una acción. Pasará al estado RUNNABLE cuando otro hilo realice una notificación (*notify*) o se haya completado el tiempo de espera.

- **TERMINATED**

  Estado que alcanza un hilo cuando finaliza. Un hilo finaliza por cualquiera de las siguientes razones:

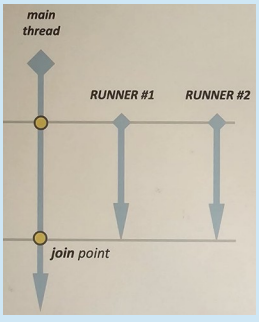
  - ➜ Cuando el código del hilo se ha ejecutado completamente, es decir, cuando se ha finalizado la ejecución de todas las sentencias de su método *run()*.
  - ➜ Cuando se produce algún error inusual que provoca una excepción no controlada.

Se usará el método *getState()* para, en cualquier momento, consultar el estado en el que se encuentra un hilo.

Ver: EstadosHilo: Programa cuyo hilo principal lanza un hilo secundario que realiza una cuenta atrás desde 10 hasta 1. Desde el hilo principal se verificará la muerte del hilo secundario mediante la función isAlive(). Además mediante el método getState() de la clase Thread vamos obteniendo el estado del hilo secundario. Se usa también el método join() que espera hasta que el hilo termina.

7. Métodos de la clase java.lang.Thread

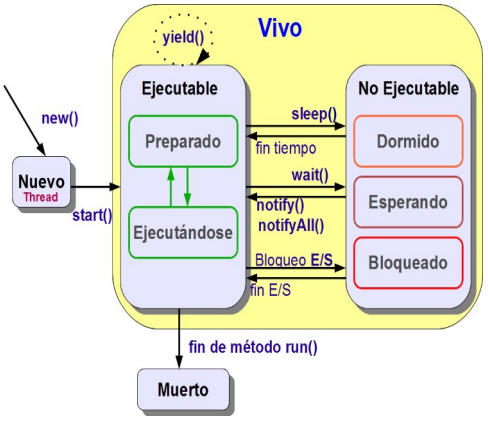
Veamos algunos de los métodos de la clase Thread más utilizados:

Método	Descripción
boolean isAlive()	Comprueba si un thread está vivo o no. Un hilo se considera que está vivo desde que se llama a su método start() hasta que finaliza su ejecución. Por tanto, este método devolverá: • **False**: Estamos ante un nuevo hilo recién "creado" o ante un hilo "muerto". • **True**: sabemos que el hilo se encuentra en un estado diferente a ***NEW*** y a ***TERMINATED***.
sleep(long ms) sleep (long milisegundos, int nanosegundos)	Cambia el estado del thread a ***TIMED_WAITING*** durante los ms indicados.
String toString()	Devuelve una representación legible de un thread: nombre, priority, nombre_del_grupo.
~~long getId()~~	Deprecated. Devuelve el identificador del thread.
void yield()	Hace que el hilo pare su ejecución instantáneamente volviendo a la cola y permitiendo que otros hilos y/o procesos se ejecuten. Sin embargo, el funcionamiento de Thread.yield() no está garantizado. Puede que después de que un hilo invoque a Thread.yield() y pase a "RUNNABLE", éste vuelva a ser elegido para ejecutarse. No garantiza que el hilo actual deje de ejecutarse. El scheduler puede ignorarlo.
void join()	Se llama desde otro hilo y hace que el hilo que lo invoca se bloquee hasta que el thread termine. Es parecido a p.waitFor() para los procesos. 

7.1. Detener temporalmente un hilo

¿Qué significa que un hilo se ha detenido temporalmente? Significa que **el hilo ha pasado a un estado "No Ejecutable"**, concretamente al estado ***TIMED_WAITING*** o ***WAITING***.

Un hilo pasará al estado ***TIMED_WAITING (hilo dormido)*** por alguna de estas circunstancias:



- Se ha invocado al método sleep() de la clase Thread, indicando el tiempo que el hilo permanecerá deteniendo. Transcurrido ese tiempo, el hilo se vuelve "Ejecutable", en concreto pasa al estado ***RUNNABLE***.

  Hay dos formas de llamar al método sleep():

  - La primera le pasa como argumento un entero (positivo) que representa el tiempo en milisegundos que el hilo permanecerá dormido.
  - La segunda le agrega un segundo argumento entero (esta vez, entre 1 y 999999), que representa un tiempo extra en nanosegundos (Un nanosegundo es la milmillonésima parte de un segundo, (10-9 s)) que se sumará al primer argumento:

- Cualquier llamada a sleep() puede provocar una excepción, que el compilador de Java nos obliga a controlar ineludiblemente mediante un bloque try-catch.

Un hilo pasará al estado ***WAITING (hilo esperando)*** cuando:

- El hilo ha detenido su ejecución mediante la llamada al método wait(), y no se reanudará, no pasará a ***RUNNABLE***, hasta que otro hilo produzca una llamada al método notify() o notifyAll(). Estudiaremos detalladamente estos métodos de la clase Object cuando veamos la sincronización y comunicación de hilos.

Un hilo pasará al estado BLOCKED (hilo bloqueado) cuando:

- El hilo está pendiente de que finalice una operación de E/S en algún dispositivo, o a la espera de algún otro tipo de recurso(ha sido bloqueado por el sistema operativo). Cuando finaliza el bloqueo, vuelve al estado ***RUNNABLE***.

Ver: JoinSleep; Viaje

7.2. Hilos demonio (Daemon Thread)

Todos los hilos que hemos visto hasta ahora son llamados hilos de usuario (**User Threads**). Sin embargo, también existen los conocidos como hilos de sistema (**System Threads**) que la JVM crea automáticamente para manejar tareas internas como gestionar recursos, gestionar señales del sistema operativo, finalización de objetos, etc.

Ver: HilosJVM

Además, en una aplicación se pueden tener hilos demonio (**Daemon Thread**) que sí los puede crear el desarrollador de la aplicación.

Un **Daemon Thread** es un hilo background cuyo propósito es asistir a otros hilos (por ejemplo, liberación de recursos, tareas de limpieza, timers, logging en segundo plano).

La característica esencial de estos hilos es que la JVM los puede finalizar automáticamente si no queda ningún **User Thread** vivo.

Los **Daemon Thread** se crean a partir de hilos de usuario, siendo convertidos en hilos daemon usando el método setDaemon(true). Esto debe hacerse antes de llamar al método start() del hilo. Si se intenta después de la llamada al método start() se lanzará un IllegalThreadStateException.

Por otro lado, para saber si un hilo es un hilo de usuario o un hilo demonio se puede utilizar el método isDaemon().

Ver: UserThread; DaemonThread

8. Sincronización y comunicación de hilos

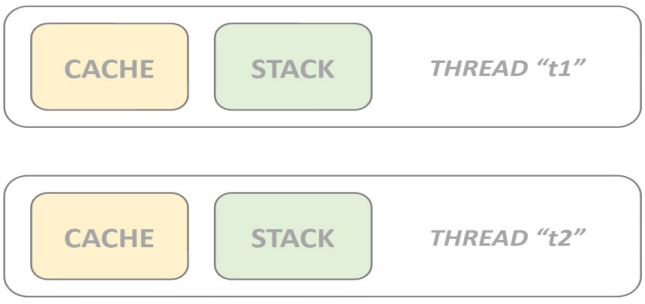
Los ejemplos realizados hasta ahora utilizan hilos independientes, es decir, una vez iniciados los hilos, éstos no se relacionan con los demás y no acceden a los mismos datos u objetos, por lo que no hay conflictos entre ellos.

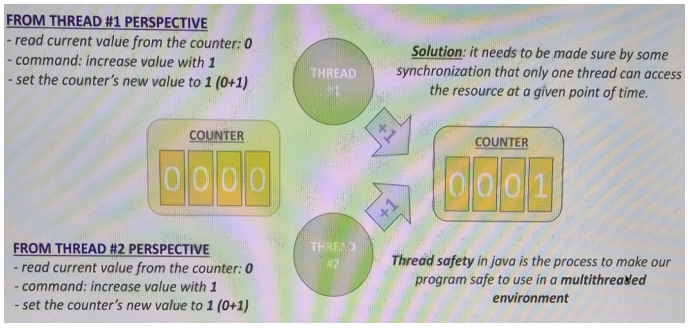
Sin embargo, hay ocasiones en las que distintos hilos de un programa necesitan establecer alguna relación entre sí y **compartir recursos o información**. Se pueden presentar las siguientes situaciones:

- Dos o más hilos **compiten por obtener un mismo recurso**, por ejemplo dos hilos que quieren escribir en un mismo fichero o acceder a la misma variable para modificarla.
- Dos o más hilos **colaboran para obtener un fin común** y para ello, necesitan comunicarse a través de algún recurso. Por ejemplo un hilo produce información que utilizará otro hilo.

En cualquiera de estas situaciones, es necesario que los hilos se ejecuten de manera controlada y coordinada, para evitar posibles interferencias que pueden desembocar en programas que se bloquean con facilidad y que intercambian datos de manera equivocada.

Ver: ProblemaSincronizacion





¿Cómo conseguimos que los hilos se ejecuten de manera coordinada? Utilizando sincronización y comunicación de hilos:

- **Sincronización**. Es la capacidad de informar de la situación de un hilo a otro. El objetivo es establecer la secuencialidad correcta del programa.
- **Comunicación**. Es la capacidad de transmitir información desde un hilo a otro. El objetivo es el intercambio de información entre hilos para operar de forma coordinada.

En Java la sincronización y comunicación de hilos se consigue mediante:

- **Monitores**. Se crean al marcar bloques de código con la palabra synchronized.
- **Semáforos**. Podemos implementar nuestros propios semáforos, o bien utilizar la clase Semaphore incluida en el paquete java.util.concurrent.
- **Notificaciones**. Permiten comunicar hilos mediante los métodos wait(), notify() y notifyAll() de la clase java.lang.Object.
- Otras Clases del paquete java.util.concurrent: Además, de Semaphore, Java proporciona unas **clases de sincronización** que permiten la sincronización y comunicación entre diferentes hilos de una aplicación multithreadring: CountDownLatch, CyclicBarrier y Exchanger.

Ver: ProblemaSincronizacionResuelto

8.1. Información compartida entre hilos

Las **secciones críticas** son aquellas secciones de código que no pueden ejecutarse concurrentemente, debido a que en ellas se encuentran recursos o información que comparten diferentes hilos.

Un ejemplo sencillo que ilustra lo que puede ocurrir cuando varios hilos actualizan una misma variable es el clásico "ejemplo de los jardines". En él, se pone de manifiesto el problema conocido como la "**condición de carrera**", que se produce cuando varios hilos acceden a la vez a un mismo recurso, por ejemplo a una variable, cambiando su valor y obteniendo de esta forma un valor no esperado de la misma.

En el ejemplo del "problema de los jardines", el recurso que comparten diferentes hilos es la variable contador **cuenta**. Las secciones de código donde se opera sobre esa variable son dos secciones críticas, los métodos incrementaCuenta() y decrementaCuenta().

La forma de proteger las secciones críticas es mediante sincronización. La **sincronización** se consigue mediante:

- **Exclusión mutua**. Asegurar que un hilo tiene acceso a la sección crítica de forma exclusiva y por un tiempo finito.
- **Por condición**. Asegurar que un hilo no progrese hasta que se cumpla una determinada condición.

En Java, la sincronización para el acceso a recursos compartidos se basa en el concepto de monitor.

8.2. Monitores. Métodos synchronized

En Java, cada objeto tiene asociado un bloqueo intrínseco llamado **monitor**. Para **crearlo**, hay que **marcar un bloque de código con la palabra synchronized**, pudiendo ser ese bloque:

- Un **método completo**.
- Cualquier **segmento de código**.

Añadir synchronized a un método significará que:

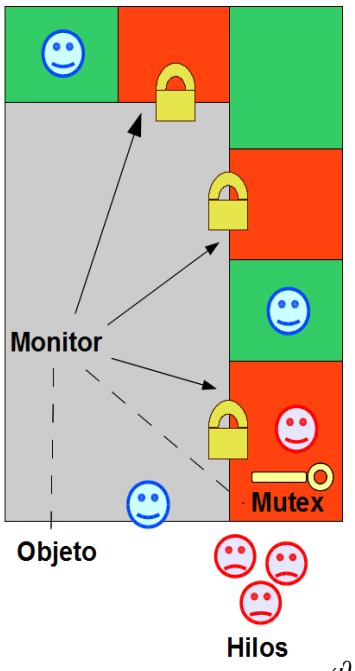
- Hemos creado un monitor asociado al objeto.
- Sólo un hilo puede ejecutar el método synchronized de ese objeto a la vez.
- Los hilos que necesitan acceder a ese método synchronized permanecerán bloqueados en una cola de hilos que está asociada al objeto (**BLOCKED**).
- Cuando el hilo finaliza la ejecución del método synchronized, los hilos en espera de poder ejecutarlo se desbloquearán. La JVM seleccionará a uno de ellos.

sintaxis para declarar **un método** synchronized	sintaxis para declarar **un segmento de código** synchronized

public **synchronized** tipodato metodo()
{
//sentencias ;
}
	public **synchronized** tipodato metodo()
	{
	//sentencias;
	synchronized (objeto)
	{
	//sentencias;
	}
	//sentencias;
	}

Y **funciona de la siguiente forma**:

- Un monitor está asociado con un objeto específico y **solo se asocia un monitor por objeto**, aunque éste tenga más de un bloque synchronized.
- Sólo un hilo puede tener el mutex o candado de un objeto en un momento dado.
- El resto de hilos que también necesitan hacerse con el mutex de ese objeto para acceder a los bloques synchronized, permanecerán bloqueados a la espera de que se libere el mutex del objeto.
- Cuando el hilo finaliza la ejecución de un bloque synchronized libera el mutex.
- Al liberarse el mutex, todos los hilos que estaban en espera para obtenerlo, se reactivarán y la JVM cederá el mutex a uno de ellos.
- El monitor sólo permitirá que un hilo pueda ejecutar un bloque synchronized a la vez. Por tanto, si existen varios bloques synchronized dentro de un objeto, sólo uno de ellos podrá ejecutarse al mismo tiempo.



Por tanto, **el hilo que quiere entrar en un método o bloque synchronized se puede encontrar** con las siguientes situaciones:

- Mutex libre. El hilo tomará el mutex, ejecutará el método y lo liberará cuando haya finalizado la ejecución de dicho método.
- Mutex en posesión de otro hilo. El hilo se bloqueará en espera de que el otro hilo lo libere.

Y ¿**qué bloques interesa marcar** como synchronized? Precisamente los que se correspondan con secciones críticas y contengan el código o datos que comparten los hilos.

En el ejemplo anterior, "Problema de los jardines" se debería sincronizar tanto el método *incrementaCuenta()*, como el *decrementaCuenta()* tal y como ves en el siguiente código, ya que estos métodos contienen la variable *cuenta*, la cual es modificada por diferentes hilos. Así mientras un hilo ejecuta el método *incrementaCuenta()* del objeto *jardin,* ningún otro hilo podrá ejecutarlo.

```
public synchronized void incrementaCuenta() {
    //método que increamenta en 1 la varibale cuenta
    System.out.println("hilo " + Thread.currentThread().getName()
            + "----- Entra en Jardín");
    //muestra el hilo que entra en el método
    cuenta++;
    System.out.println(cuenta + " en jardín");
    //cuenta cada acceso al jardín y muestra el número de accesos
}
```

Ver: JardinesSincronizado ; SincronizacionFrases; SincronizaMetodo

Ver presentación: new_threads.pptx

El problema de usar synchronized es que si tenemos varios métodos independientes que tienen que ser sincronizados y varios hilos, cada uno usando uno de esos métodos independientes, al existir un solo monitor por objeto, si uno de los hilos accede al primer método sincronizado, el otro hilo no podrá acceder al segundo método sincronizado aunque sea independiente. Esto provocará que la aplicación sea más lenta y no se esté maximizando el rendimiento del equipo.

Por lo tanto, cuando dentro de una clase se declaran múltiples métodos synchronized, todos los hilos competirán por el mismo monitor.

Solución: Declarar synchronized un segmento de código en lugar de un método completo.

En sistemas con muchos hilos o alto volumen de peticiones, el rendimiento se degrada significativamente.

Ver: ProblemaSincronizacionMultiple; ProblemaSincronizacionMultipleConEstado