Hilos

Concepto de concurrencia

Supongamos una máquina dotada de un único procesador, encargado de administrar todas las operaciones. Alguien desea leer un documento y es este procesador el que interactuando con los controladores del disco debe controlar los movimientos del cabezal, encontrar y leer los distintos sectores, colocar la información en el bus de datos para luego recibirla y procesarla.

Naturalmente no es así como funcionan las máquinas de hoy. En el disco duro existe un procesador de menor inteligencia que el procesador central, que se encarga de mover el cabezal, rastrear sectores y manejar el bus de datos. En conclusión llegamos a que es imposible que un procesador maneje todas las operaciones que se realizan en unos cuantos milisegundos dentro de una computadora brindando un tiempo de respuesta aceptable.

Los procesadores más pequeños que se encuentran en los dispositivos o en las placas adaptadoras se llaman controladores y no son un invento nuevo. En los años 50 se desarrollaron equipos que tenían, además de un procesador central, otros capaces de manejar dispositivos de entrada salida y bus de datos, para lograr un mejor rendimiento del procesador. Con esta configuración un equipo es capaz de correr dos o más procesos al mismo tiempo, mientras el procesador central está ocupado con cierta tarea, el controlador del disco está moviendo el cabezal, el cañón del monitor está dibujando la imagen, el coprocesador matemático está haciendo algunos cálculos, etc. y de esa forma se ahorra mucho tiempo, porque los procesos corren en el mismo momento, y no es necesario terminar de imprimir para leer el disco o calcular una suma.

Con el mismo concepto, se han creado grandes equipos, que además de sus controladores, tienen más de un procesador central, de esta manera distintos programas pueden ejecutarse al mismo tiempo, uno en cada procesador y además hacer uso de todos los recursos del equipo por medio de sus controladores. Esto último se denomina PARALELISMO EN HARDWARE, más de un procesador permite ejecutar más de un proceso al mismo tiempo. Se dice que dos o más procesos son concurrentes si están construidos de manera tal que pueden ejecutarse al mismo tiempo, compartiendo recursos y datos.

En todos los paradigmas analizados hasta el momento, el seguimiento del programa ha sido siempre secuencial, es decir que se ejecuta una instrucción debajo de la otra. Si un proceso llama a una función, este se interrumpe, comienzan a ejecutarse secuencialmente una a una las instrucciones de la función hasta que esta retorna un valor que es tomado por el proceso llamador para continuar con la instrucción siguiente.

Lenguajes como JAVA permiten que dos procesos o más se ejecuten simultáneamente, de esta manera, los procesos pueden interactuar compartiendo datos y recursos, como funciones y procedimientos comunes, pero no necesariamente debe interrumpirse uno para que otro comience hasta finalizar de devolver el control.

Es muy importante que se entienda que pese a que paralelismo y concurrencia son dos conceptos muy relacionados puede existir uno sin el otro. Puede haber paralelismo en el hardware sin

concurrencia, por ejemplo, cuando se corren dos o más procesos en paralelo, en diferentes procesadores, pero no están preparados para compartir datos, no interactúan, son aplicaciones distintas que se diseñaron para correr independientemente una de la otra, aunque un equipo con más de un procesador pueda soportarlas al mismo tiempo. Por otro lado puede haber concurrencia sin paralelismo, si una aplicación está compuesta por procesos que no necesariamente deben ejecutarse en forma secuencial, comparten datos, interactúan entre si y pueden hacerlo simultáneamente, pero al ser soportadas por un único procesador se ejecutan en forma secuencial.

Hilos de ejecución (Threads)

Un hilo de ejecución, en los sistemas operativos, es similar a un proceso en cuanto a que ambos representan una secuencia simple de instrucciones ejecutada en paralelo con otras secuencias. Los hilos permiten dividir un programa en dos o más tareas que corren simultáneamente, por medio de la multiprogramación. En realidad, este método permite incrementar el rendimiento de un procesador de manera considerable. En todos los sistemas de hoy en día los hilos son utilizados para simplificar la estructura de un programa que lleva a cabo diferentes funciones.

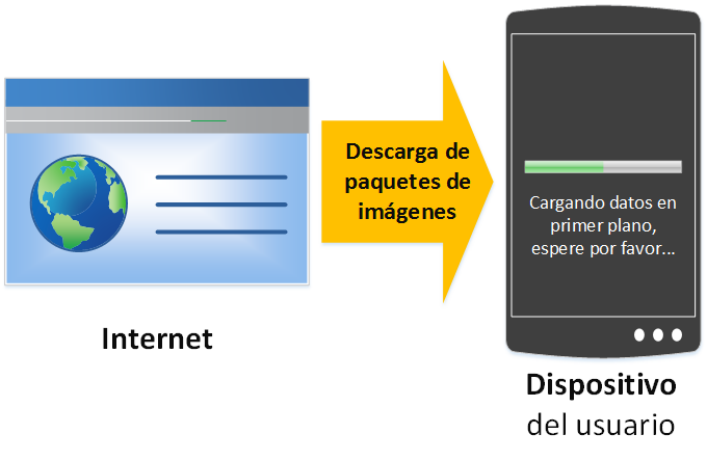
Todos los hilos de un proceso comparten los recursos del proceso. Residen en el mismo espacio de direcciones y tienen acceso a los mismos datos. Cuando un hilo modifica un dato en la memoria, los otros hilos utilizan el resultado cuando acceden al dato. Cada hilo tiene su propio estado, su propio contador, su propia pila y su propia copia de los registros de la CPU. Los valores comunes se guardan en el bloque de control de proceso (PCB), y los valores propios en el bloque de control de hilo (TCB).

Multihilo (multithreading)

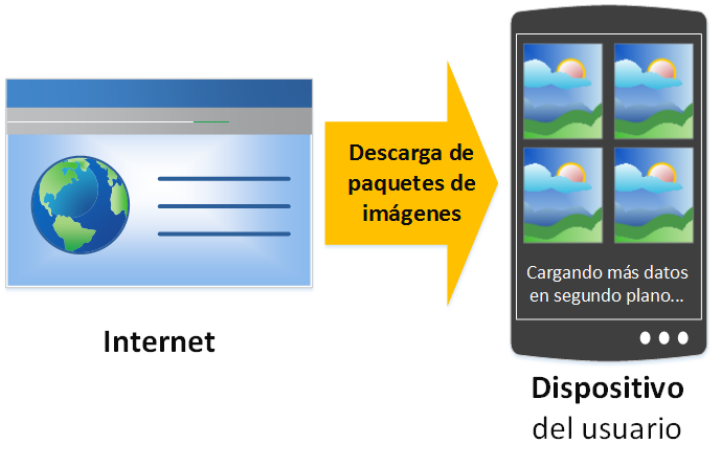
La técnica multihilo permite ejecutar varios hilos a la vez; es decir, de forma concurrente y por tanto permite hacer programas que se ejecuten en menor tiempo y sean más eficientes. Evidentemente no se pueden ejecutar infinitos hilos de forma concurrente ya que el hardware tiene sus limitaciones, pero raro es a día de hoy los ordenadores que no tengan más de un núcleo por tanto en un procesador con dos núcleos se podrían ejecutar dos hilos a la vez y así nuestro programa utilizaría al máximo los recursos hardware.

Ejemplo: Si se desea ver un listado de 100 imágenes que se descargan desde Internet, como usuario ¿Cuál de las dos opciones siguientes elegiría?:

1) Descargar las 100 imágenes, haciendo esperar al usuario con una pantalla de “cargando” hasta finalizar dicha tarea.



2) Que el usuario pueda ir viendo las imágenes que ya se han descargado mientras paralelamente se descargan el resto.



La opción 2 es lo que todo usuario quiere de una aplicación, tener que esperar no es una opción. Volviendo al punto de vista del desarrollador, tampoco es una opción. Un buen desarrollador de aplicaciones hace bien las cosas y se decanta por la opción B –la opción A no existe.

Desde el punto de vista del usuario existen dos áreas bien diferenciadas, que el desarrollador ha de tener en cuenta:

Primer plano: Aquí se ejecuta únicamente un hilo llamado *"hilo principal"*. Aquí se ha programado siempre, sin conocimiento de estar trabajando ya con hilos. Es el hilo que trabaja con las vistas, es decir, con la interfaz gráfica que ve el usuario: botones, ventanas emergentes, campos editables, etc. También, puede ser usado para hacer cálculos u otros procesamientos complejos, aunque estos deberían de evitarse hacerse en este hilo a toda costa – salvo si es imposible que se hagan en otro hilo. Aquí es donde el usuario interacciona de manera directa. El desarrollador ha de tener especial cuidado al trabajar con el hilo principal, ya que si la aplicación va lenta es porque el primer plano va lento. También es importante saber, que una mala gestión del primer plano por parte del desarrollador, será castigada por el sistema operativo (por ejemplo: en Android si el hilo principal de una aplicación es bloqueado más de 5 segundos, la aplicación se cerrará mostrando una ventana de forzar cierre; y comportamientos parecidos en otros sistemas operativos en donde se muestra le famoso mensaje "la aplicación no responde, con opción de cerrar el programa.

Segundo plano (o en inglés background): Se ejecuta el resto de los hilos. El segundo plano tiene la característica de darse en el mismo momento que el primer plano. Aquí los hilos deberían de llevar las ejecuciones pesadas de la aplicación. El segundo plano el usuario no lo ve, es más, ni le interesa, para el usuario no existe.

Lo principal es claro: no se debe interrumpir al usuario nunca, por lo que no se debe hacer cosas que consuman muchos recursos en el hilo principal.

Para poder entender mejor la importancia y la función de los hilos se hará una analogía del ejemplo anterior en donde una persona será un hilo, una fábrica que representa a internet, una tienda que

representa la visualización de la aplicación y el usuario final que es el que desea descargas las imágenes de internet.

Hilo que lo hace todo
encargado de descargar
los datos de Internet
Y también encargado
dibujar las descargas en
la pantalla

Este trabajador (el hilo), hace su trabajo de la mejor manera que sabe, pero tiene demasiadas responsabilidades. Para lo que le han contratado es para llevar paquetes desde una fábrica hasta una tienda. Además, este trabajador (hilo) es el hilo principal de la aplicación.

Fábrica (Representa todo lo que consuma tiempo de proceso, como consultas a Internet, cálculos, etc)
Fabrica paquetes para la tienda

Internet

Hilo que lo hace todo
Encargado de descargar las imágenes de Internet.
Y también de dibujar las descargas en la pantalla

Tienda (Representa la pantalla del dispositivo)
Aquí los paquetes se muestran a los clientes en el escaparate para que los puedan consumir

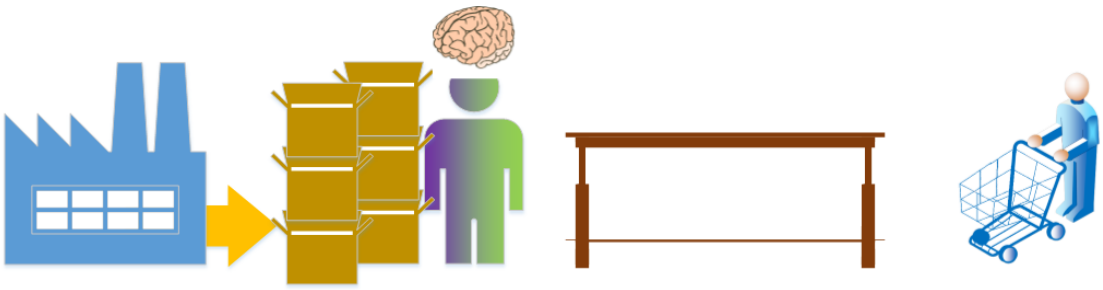
Pantalla del dispositivo

Usuario

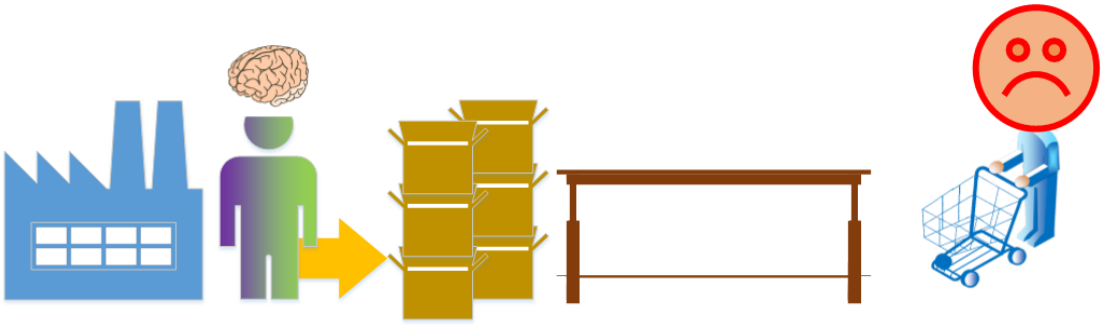
A continuación se detalla todo el proceso de este trabajador (hilo) en la vida de un programa.

Ejemplo sin concurrencia ni paralelismo

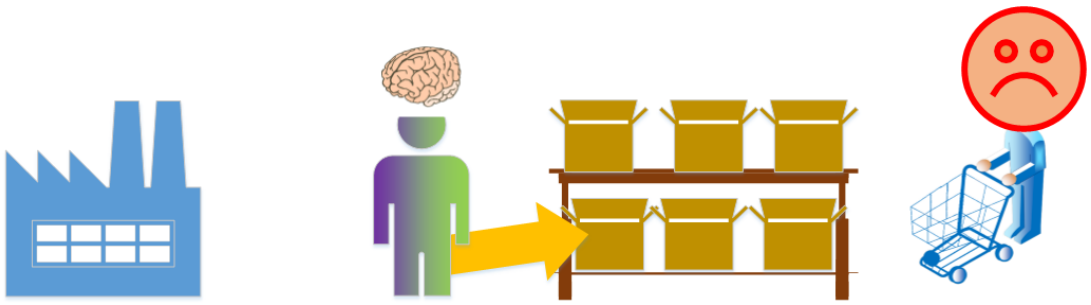
1) En primer lugar el trabajador busca todos los paquetes de datos con las “imágenes” de una fábrica que se llama “Internet”.



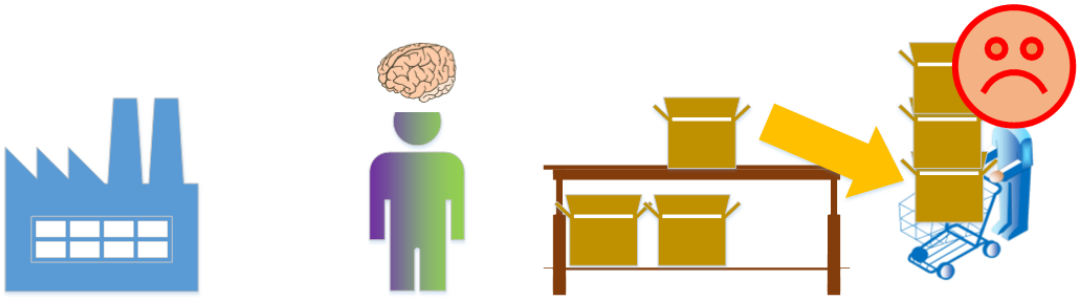
2) Luego lleva todos estos paquetes en camión y los descarga a una tienda llamada “dispositivo del usuario”.



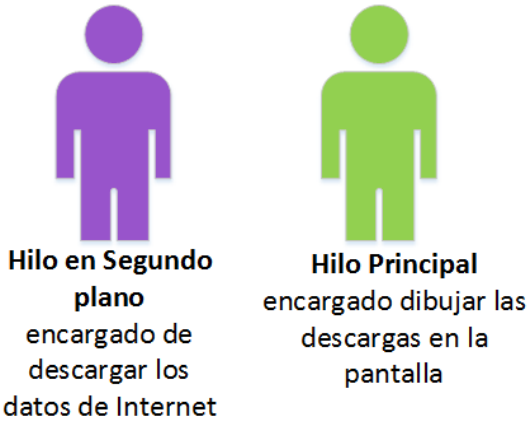
3) Luego coloca todos y cada uno de los paquetes que ha trasportado en el mostrador de la tienda, que podemos tomarlo como la interfaz del programa.



4) Todo esto se hace con la finalidad de que el cliente o “usuario” los consuma. Eso sí, solo en este momento se le permite al usuario poder consumirlos –ya que es el momento que tiene algo que consumir, antes no había nada- con lo que el resto del tiempo estuvo esperando.



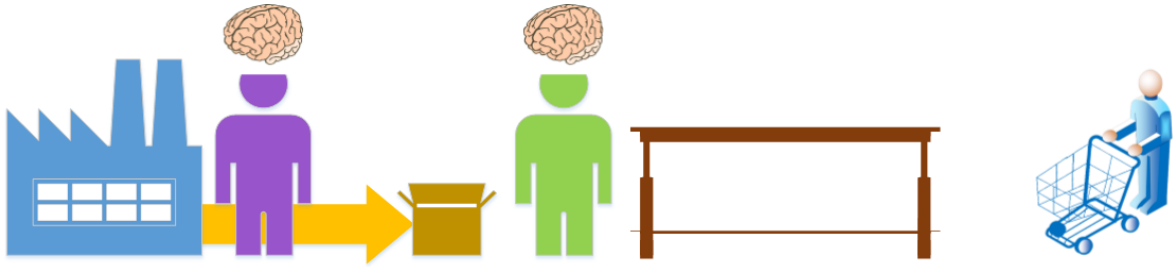
En el ejemplo el trabajador tiene dos colores para representar la gran carga de trabajo que tiene este hilo. Habrá que investigar si se puede hacer de él dos hilos diferentes ¿Será necesario contratar a un segundo trabajador?



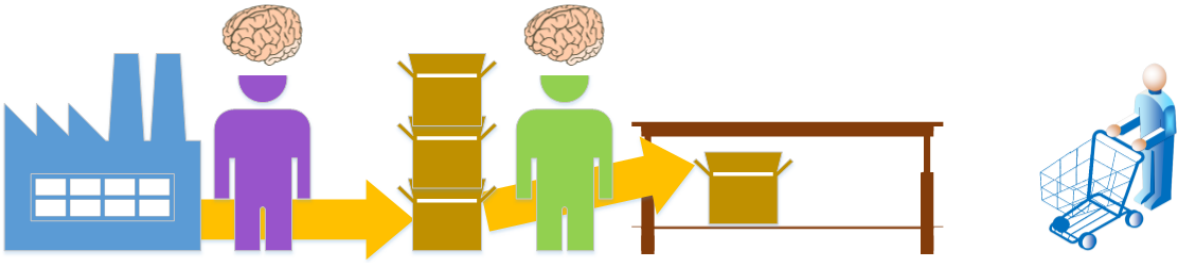
Teniendo dos trabajadores es lógico pensar que todo vaya mejor y más rápido. De esta manera se dispondrá de un trabajador principal cuya dedicación es exclusiva para el usuario. Y otro trabajador en segundo plano, que se dedica a descargar los paquetes de datos, cuya función no la siente el usuario y nunca sabrá que está ahí trabajando para él.

Ejemplo con concurrencia y paralelismo

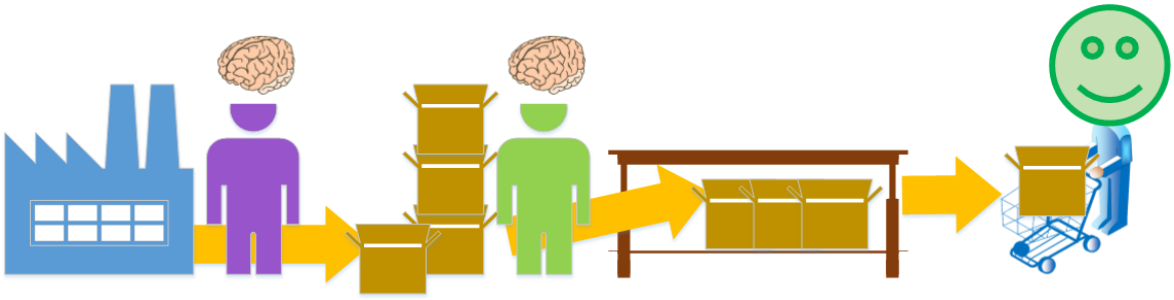
1) Para empezar, el trabajador en segundo plano toma algún paquete de datos de la fábrica que llamamos "Internet", lo lleva en camión y lo descarga en la tienda. Donde el trabajador principal está esperando a que le llegue algún paquete para poder realizar su labor.



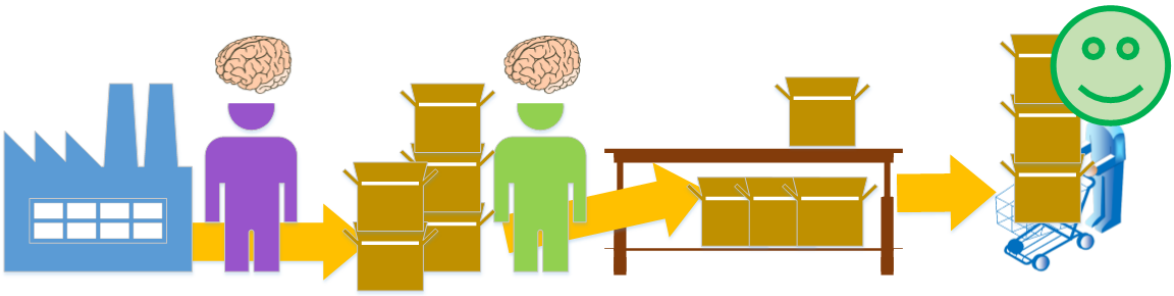
2) El trabajador principal ya tiene algo que hacer. Toma el paquete que le ha llegado y lo coloca en el mostrador de la tienda. Mientras tanto, el trabajador en segundo plano no ha terminado su trabajo, con lo que continúa llevando paquetes a la tienda desde la fábrica. Cabe notar que el cliente/usuario ya tiene al menos un paquete que consumir en la tienda.



3) El proceso continúa. El trabajador en segundo plano lleva a la tienda los paquetes, para que el trabajador en primer plano los coloque en el mostrador. Por lo que, el cliente/usuario puede ir consumiendo cuanto desee de lo que exista ya en el mostrador.



4) Esto durará hasta que se cumpla alguna condición de finalización o interrupción de algún trabajador, las cuales pueden ser muchas (por ejemplo: apagar el dispositivo, que se hayan acabado los paquetes que descargar, etc.).



Con los hilos aparece la verdadera magia: la concurrencia y con ello se posibilita la programación en paralelo.

En el primer ejemplo de la fábrica y la tienda (sin concurrencia ni paralelismo), en el que solo había un hilo principal que lo hace todo, se tiene que todo se hace en un tiempo X que se estima largo. Se puede representar por la siguiente barra de tiempo desde el principio de la ejecución del hilo hasta el fin:

En comparación con este segundo ejemplo, en el que existen dos hilos, uno principal y otro secundario, que trabajan de manera simultánea, es deducible que el tiempo que tarda en hacerse todo el proceso es bastante menor, ya que se sobrexponen los tiempos de ambos hilos. En la siguiente representación se expone como el hilo principal se ejecuta de manera “paralela” al hilo en segundo plano y que es menos a la anterior barra de tiempo:

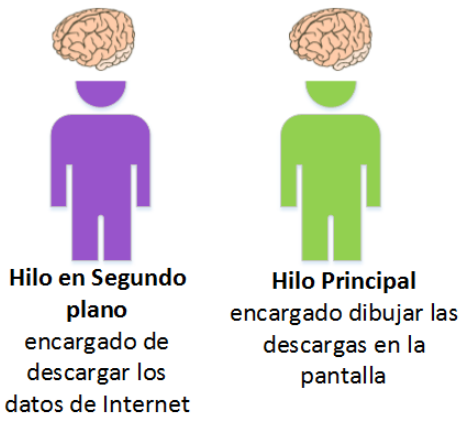
Remitiendo a la definición de paralelismo, se basa (en teoría) en el hecho de poder ejecutar varios procesos en varios núcleos del procesador, o en este caso varios hilos en varios núcleos del procesador, pero que sucede si se desean crear 20 hilos, ¿Se necesitarían 20 núcleos?

Para explicar esto se retomará el ejemplo de los trabajadores. En este caso los trabajadores representan hilos, y el núcleo de un procesador sería el cerebro de cada trabajador. Un trabajador siempre tendrá una misión que hacer determinada, pero sin un cerebro no puede hacer nada.

Así, si se repasa el primer ejemplo de la fábrica y la tienda con un solo trabajador, con un cerebro, le es más que suficiente para ejecutar todo de manera correcta. Le sobraría con un procesador con un núcleo, pues más núcleos no los aprovecharía.

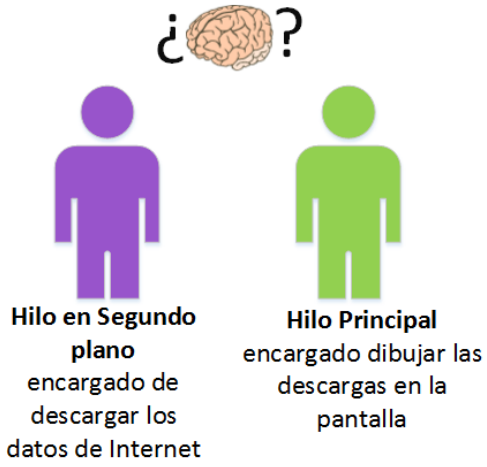
Por lo que un cerebro (núcleo) procesará el trabajo tan rápido como pueda en el tiempo (esta rapidez se mide con la velocidad de reloj de un núcleo de un procesador, es decir, los Hercios; si por ejemplo, un procesador posee una velocidad de 3.4GHz, quiere decir que cada núcleo tiene una velocidad de 3.4GHz, y esa velocidad refiere a los datos que puede procesar por segundo, en este caso 3.400.000.000 (tres mil cuatrocientos millones) de datos por segundo.

Al repasar el segundo ejemplo es más complejo, se supone que se tienen dos núcleos, y que cada núcleo se corresponde a cada hilo que se ejecuta.



Así, con dos cerebros se puede trabajar en paralelo como se indicó anteriormente.

Volviendo a la cuestión en la cual si se desea crear 20 hilos se necesitarían 20 núcleos, la respuesta es no. Para poder razonarlo se hará un ejemplo más acotado donde la pregunta es, que sucede si tengo 2 hilos (como en el ejemplo 2 trabajadores), pero tengo un procesador de un único núcleo.



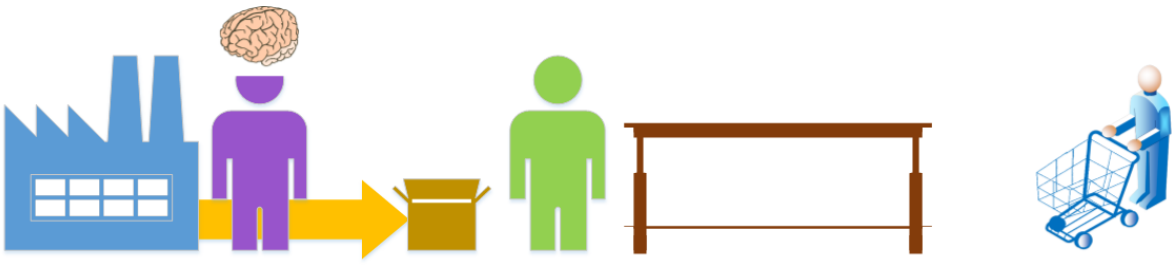
Es evidente que no hay suficientes cerebros para tantos trabajadores. En algún momento uno de ellos no tendrá cerebro y se tendrá que quedar idiota –detenido completamente- mientras el otro con cerebro realiza sus labores. Lo razonable y justo es que compartan el cerebro un rato cada uno para poder trabajar los dos. Y así es como realmente trabaja el procesador para ejecutar varios hilos "en paralelo". El procesador se encarga de decidir cuánto tiempo va a usar cada hilo el único núcleo que existe. Dicho de otro modo, el procesador partirá los hilos en cachos muy pequeños y los juntará salteados en una línea de tiempo. Al procesarse los dos hilos de manera salteada, y gracias a la gran velocidad con la que procesan datos los procesadores hoy día, dará la sensación de paralelismo, pero no es paralelismo. Al observar la siguiente línea de tiempo, se ve que la longitud es tan larga como si se hiciera todo en un hilo.

No es paralelo pero sí es concurrente: dos o más componentes entre ellos independientes (por ejemplo, hilos con funcionalidades separadas) se ayudan para solucionar un problema común (en estos ejemplos, los hilos trabajan juntos para mostrar al usuario unos paquetes descargados de Internet). Lo concurrente se recomienda ejecutarse en paralelo para sacar el máximo partido de la concurrencia, pero como se puede observar no es obligatorio. En el primer ejemplo con un solo hilo se comprueba que no existe ni la concurrencia ni el paralelismo. Y en el segundo ejemplo, con dos hilos y dos núcleos, se da tanto la concurrencia como el paralelismo.

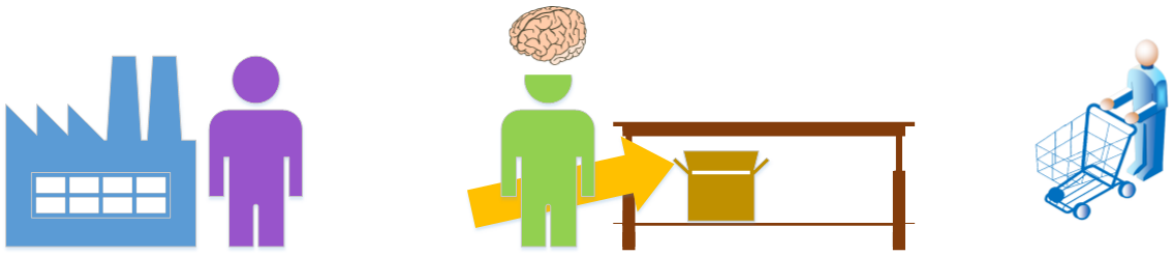
Entonces, si al ejecutar hilos concurrentemente sin paralelismo se tarda el mismo tiempo que al hacerlo sin concurrencia ni paralelismo ¿Cuál es la ventaja? La ventaja es clara y se ve, y es que si se ejecuta toda la aplicación en un solo hilo sin concurrencia ni paralelismo, el usuario deberá esperar que se descargue todo el contenido para poder acceder a él, mientras que si se ejecuta la aplicación en 2 hilos con concurrencia, habrá algún instante en el que el usuario tenga con que actuar.

Ejemplo de concurrencia sin paralelismo

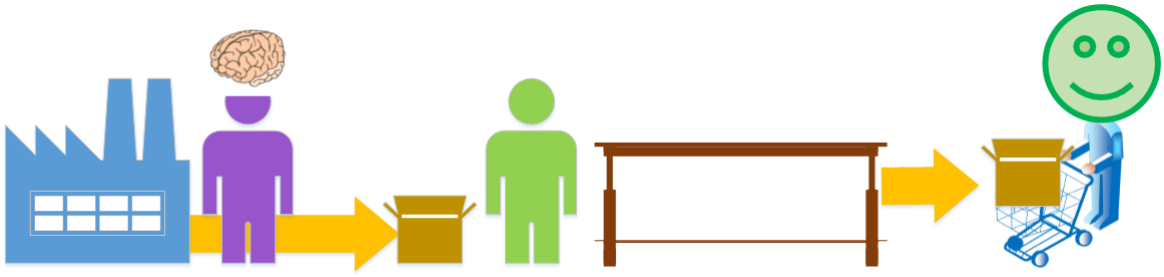
1) El procesador decide que empiece poseyendo al cerebro el trabajador en segundo plano, para que descargue algún paquete de datos. Mientras, el trabajador principal no podrá hacer nada porque no tiene cerebro.



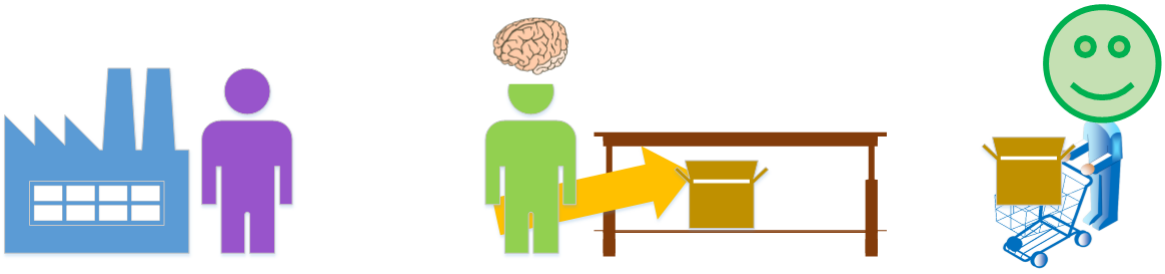
2) El procesador ha decidido que ha tenido mucho tiempo al trabajador en segundo plano y le pasa el turno al trabajador principal. Este, mientras pueda usar al cerebro, colocará todos los paquetes de datos que pueda en el mostrador.



3) El procesador cambia el turno al trabajador secundario, que realiza su trabajo. El trabajador principal no hace nada. Y ya por lo menos el cliente/usuario dispone de algún dato.



4) Este vaivén de turnos cerebrales se repite.



Se aprecia en este ejemplo, que no es tan rápido como disponer de dos núcleos, pero tampoco tan lento como trabajar con un solo hilo que lo haga todo -en poco tiempo el usuario ya tiene algo que hacer.

En este ejemplo se ha simplificado el orden de turnos del procesador. En la mayoría de los casos, el procesador no espera a que se termine la acción completa antes de ofrecer el turno del núcleo a otro hilo. Es decir, el procesador no esperaría a que el hilo que descarga los datos, descargue el 100% del paquete entero para pasarle el núcleo al otro hilo. Sino que le puede quitar el núcleo al hilo secundario que descarga, por ejemplo, al 23,567% de la descarga. Y lo mismo con el hilo principal, podría quitarle el núcleo a mitad de la acción de colocar el paquete en el escaparate. Eso sí, al volver a tener el turno del procesador continuará desde donde se quedó. Las únicas acciones que no puede dividir el procesador son aquellas sentencias que se hacen llamar operaciones atómicas: son aquellas que no son divisibles por el procesador y se tienen que ejecutar de manera completa.

En la aplicación anterior, no se ve el hilo de ejecución que corre el programa. Sin embargo, Java posibilita la creación y control de hilos de ejecución explícitamente. La utilización de hilos (threads) en Java, permite una enorme flexibilidad a los programadores a la hora de plantearse el desarrollo de aplicaciones. La simplicidad para crear, configurar y ejecutar hilos de ejecución, permite que se puedan implementar muy poderosas y portables aplicaciones/applets que no se puede con otros lenguajes de tercera generación. En un lenguaje orientado a Internet como es Java, esta herramienta es vital. Al usar un navegador se ve el uso de múltiples hilos. Se observa que se puede navegar en una página al mismo tiempo que las imágenes siguen cargándose, o realizar un comentario en un video que todavía no ha cargado, utilizar múltiples pestañas para abrir varias páginas, etc. Esto no significa que las páginas utilicen múltiples hilos, sino que el navegador es multihilo o multithreaded. En este caso, el navegador por ejemplo, está trayéndose las imágenes en un hilo de ejecución y soportando el desplazamiento de la página en otro hilo diferente. Las aplicaciones (y applets) multihilo utilizan muchos contextos de ejecución para cumplir su trabajo. Hacen uso del hecho de que muchas tareas contienen subtareas distintas e independientes. Se puede utilizar un hilo de ejecución para cada subtarea. Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multihilo permite que cada thread comience y termine tan pronto como sea posible. Este comportamiento presenta una mejor respuesta a la entrada en tiempo real.

La clase Thread y la interfaz Runnable

La interfaz **java.lang.Runnable** permite definir las operaciones que realiza cada *thread*. Esta interfaz se define con un solo método público llamado **run** que puede contener cualquier código, y que será el código que se ejecutará cuando se lance el *thread*. De este modo para que una clase realiza tareas concurrentes, basta con implementar *Runnable* y programar el método **run**.

La clase **Thread** crea objetos cuyo código se ejecute en un hilo aparte. Permite iniciar, controlar y detener hilos de programa. Un nuevo *thread* se crea con un nuevo objeto de la clase java.lang.Thread. Para lanzar hilos se utiliza esta clase a la que se la pasa el objeto **Runnable**.

Cada clase definida con la interfaz Runnable es un posible objetivo de un thread. El código de **run** será lo que el thread ejecute.

La clase Thread implementa la interfaz Runnable, con lo que si creamos clases heredadas, estamos obligados a implementar *run*. La construcción de objetos Thread típica se realiza mediante un constructor que recibe un objeto Runnable.

```
1    hilo = new Thread(objetoRunnable);
2    hilo.start(); //Se ejecuta el método run del objeto Runnable
```

Métodos de la clase Thread

Métodos	Uso
void **interrupt()**	Solicita la interrupción del hilo
static boolean **interrupted()**	**true** si se ha solicitado interrumpir el hilo actual del programa
static void **sleep(int** milisegundos**)**	Pausa el Thread actual durante un cierto número de milisegundos
static void **sleep(int** milis, **int** nanos**)**	Pausa el Thread actual durante un cierto número de milisegundos y nanosegundos
boolean **isAlive()**	Devuelve **true** si el hilo está vivo
boolean **isInterrupted()**	Devuelve **true** si se ha pedido interrumpir el hilo
static Thread **currentThread()**	Obtiene un objeto Thread que representa el hilo actual
void **setDaemon(boolean** b**)**	Establece (en el caso de que b sea **true**) el Thread como servidor. Un thread servidor puede pasar servicios a otros hilos. Cuando sólo quedan hilos de este tipo, el programa finaliza
void **setPriority(int** prioridad**)**	Establece el nivel de prioridad del Thread. Estos niveles son del 1 al 10. Se pueden utilizar estas constantes también: • **Thread.NORMAL_PRIORITY**. Prioridad normal (5) • **Thread.MAX_PRIORITY**. Prioridad alta (10). • **Thread.MIN_PRIORITY**. Prioridad mínima (1).
void **start()**	Lanza la ejecución de este hilo, para ello ejecuta el código del método **run** asociado al *Thread*
static void **yield()**	Hace que el Thread actual deje ejecutarse a Threads con niveles menores o iguales al actual.

Creación de hilos

Existen dos formas para la creación de hilos:

1) Mediante clases que implementan Runnable
2) Mediante clases derivadas de Thread

1- Mediante clases que implementan Runnable

Consiste en:

1. Crear una clase que implemente **Runnable**
2. Definir el método **run** y en él las acciones que tomará el hilo de programa
3. Crear un objeto **Thread** tomando como parámetro un objeto de la clase anterior
4. Ejecutar el método **start** del **Thread** para ejecutar el método **run**

El código de run se ejecuta hasta que es parado el Thread. La clase Thread dispone de el método **stop()** para definitivamente la ejecución del thread. Sin embargo no es recomendable su utilización ya que puede frenar inadecuadamente la ejecución del hilo de programa. De hecho este método se considera obsoleto y **no debe utilizarse jamás**. La interrupción de un hilo de programa se tratará más adelante. Ejemplo de creación de hilos:

```
class animacionContinua implements Runnable{
 ...
   public void run() {
      while (true) {
        //código de la animación
        ...
      }
   }
}
```

```
class Principal {
 ...
  public static void main() {
     animacionContinua a1 = new animacionContinua();
     Thread thread1 = new Thread(a1);
     thread1.start();
  }
...
}
```

Se considera una solución más adaptada al uso de objetos en Java hacer que la propia clase que implementa **Runnable**, lance el **Thread** en el propio constructor.

```
class animacionContinua implements Runnable{
Thread thread1;
  animacionContinua() {
     thread1 = new Thread(this);
     thread1.start();
  }
```

```
7       … // método run
8     }
```

2- mediante clases derivadas de Thread

Es la opción más utilizada. Como la clase **Thread** implementa la interfaz **Runnable**, ya cuenta con el método **run**. Pasos:

1. Crear una clase basada en **Thread**
2. Definir el método **run**
3. Crear un objeto de la clase
4. Ejecutar el método **start**

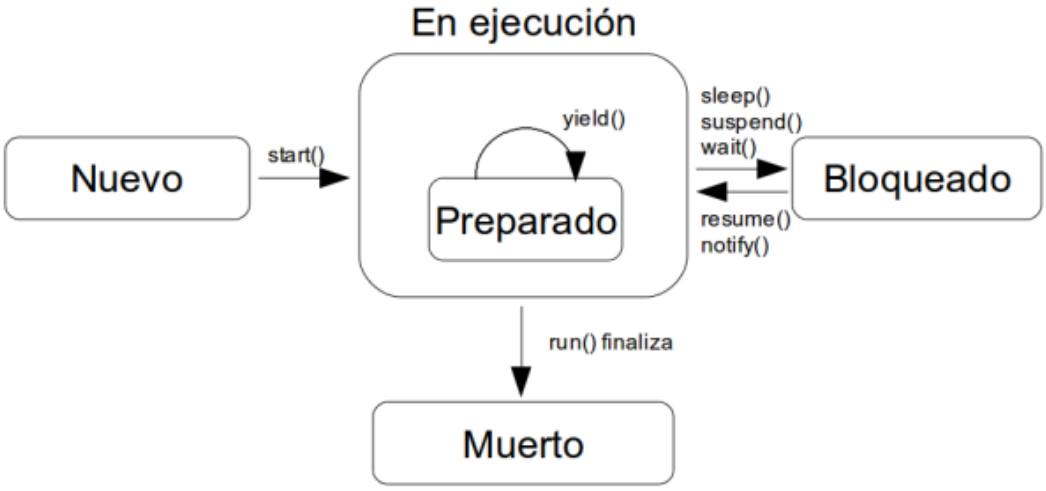
Ejemplo:

```
1   class animacionContinua extends Thread {
2    ...
3      public void run() {
4         while (true) {
5           //código de la animación
6           ...
7         }
8      }
9   }
```

```
1   class Principal {
2    ...
3     public static void main() {
4        animacionContinua a1 = new animacionContinua();
5        thread1.start();
6     }
7   ...
8    }
```

Estado de un hilo en Java

Un hilo tiene su propio ciclo de vida, durante el cual puede encontrarse en diferentes estados. Java controla el estado de los hilos mediante el llamado planificador de hilos, que se encargará de



gestionar qué hilo debe ejecutarse en cada momento y en qué estado deben encontrarse el resto de hilos.

Estado nuevo

Es el estado en el que se encuentra un thread en cuanto se crea. En ese estado el thread no se está ejecutando. En ese estado sólo se ha ejecutado el código del constructor del Thread.

Estado de ejecución

Ocurre cuando se llama al método start. No tiene por qué estar ejecutándose el thread, eso ya depende del propio sistema operativo. Es muy conveniente que salga de ese estado a menudo (al estado de bloqueado), de otra forma se trataría de un hilo egoísta que impide la ejecución del resto de threads. La otra posibilidad de abandonar este estado es debido a la muerte del thread.

Estado bloqueado

Un thread está bloqueado cuando:

- Se llamó al método **sleep**
- Se llamó a una operación de entrada o salida
- Se llamó al método **wait**
- Se intentó bloquear otro thread que ya estaba bloqueado

Se abandona este estado y se vuelve al de ejecutable si:

- Se pasaron los milisegundos programados por el método **sleep**
- Se terminó la operación de entrada o salida que había bloqueado al thread
- Se llamó a **notify** tras haber usado **wait**
- Se liberó al thread de un bloqueo

Estado muerto

Significa que el método finalizó. Esto puede ocurrir si:

- El flujo de programa salió del método run
- Por una excepción no capturada

Se puede comprobar si un thread no está muerto con el método **isAlive** que devuelve **true** si el thread no está muerto.

Ejemplo:

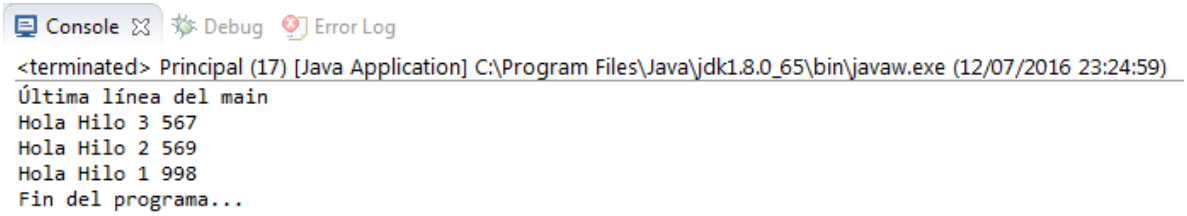
```
1   public class Hilo extends Thread {
2
3     private String nombre;
4     private int retardo;
5
6     public Hilo (String nombre, int retardo) {
7       this.nombre = nombre;
8       this.retardo = retardo;
```

```
  }

  public void run() {
    try {
      sleep(retardo);
    } catch (InterruptedException e) {;}
  System.out.println("Hola " +  nombre + " " + retardo);
  }
}
```

```
public class FinDelPrograma extends Thread {

  private int retardo;

  public FinDelPrograma(int retardo) {
    this.retardo = retardo;
  }

  public void run() {
    try {
      sleep(retardo);
    } catch (InterruptedException e) {;}
      System.out.println("Fin del programa...");
  }
}
```

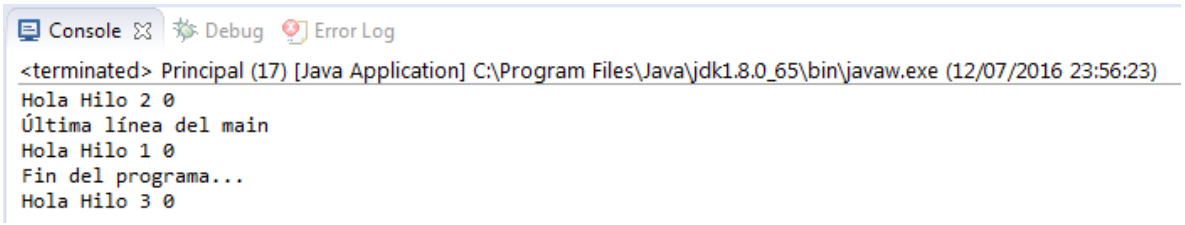
```
public class Principal {

  public static void main (String[] args){

     Hilo h1, h2, h3;
     FinDelPrograma h4;

     h1 = new Hilo("Hilo 1", (int) (Math.random()*2000));
     h2 = new Hilo("Hilo 2", (int) (Math.random()*2000));
     h3 = new Hilo("Hilo 3", (int) (Math.random()*2000));
     h4 = new FinDelPrograma(2100);

     h1.start();
     h2.start();
     h3.start();
     h4.start();

     System.out.println("Última línea del main");

   }
}
```

Este ejemplo consta de 3 clases: La clase principal que contiene el método **main** y dos clases que extienden de Thread. La clase **Hilo** informa su nombre y el tiempo que tardo en ejecutarse, y la clase **FinDelPrograma** informa el fin del programa luego de un determinado tiempo. El resultado de ejecutar el programa es el siguiente:



El último println del main con el mensaje *"Última línea del main"* es la primera que se ejecuta ya que las demás lo hacen con un tiempo de retardo. Luego los 3 hilos creados de la clase **Hilo** se ejecutarán en un tiempo aleatorio y finalmente el hilo de la clase **FinDelPrograma** se ejecutará en último lugar.

Si se modifica este ejemplo y se ponen todos los tiempos en 0 todos los hilos, incluido el principal (main), se ejecutarán sin ningún patrón predecible por el programador. Uno de los resultados puede ser el siguiente:



Planificación de hilos

Un tema fundamental dentro de la programación multihilo es la planificación de los hilos. Este concepto se refiere a la política a seguir de qué hilo toma el control del procesador y de cuándo. Obviamente en el caso de que un hilo este bloqueado esperando una operación de IO debería dejar el control del procesador y que este control lo tomara otro hilo que si pudiera hacer uso del tiempo de CPU. ¿Pero qué pasa si hay más de un hilo esperando? ¿A cuál de ellos se le debe otorgar el control del procesador? Para determinar esta cuestión cada hilo posee su propia prioridad, un hilo de prioridad más alta que se encuentre en el estado listo entrará antes en el estado de ejecución que otro de menor prioridad. Cada hilo hereda la prioridad del hilo que lo ha creado, habiendo 10 niveles enteros de prioridad desde 1 a 10, siendo 1 la menor prioridad y 10 la mayor prioridad.

```
h1.setPriority(10);  //Le concede la mayor prioridad
h2.setPriority(1);   //Le concede la menor prioridad
```

También existen constantes definidas para la asignación de prioridad estas son:

```
MIN_PRIORITY    = 1
NORM_PRIORITY = 5
MAX_PRIORITY    = 10
```

Entrando en cuestiones de diseño, se cita el modelo de Seehein en el que la aplicación está dividida en al menos 3 hilos, uno de lógica de negocio, otro de interfaz de usuario y otro de acceso a datos. En este modelo, debería concederse a la interfaz de usuario la más alta prioridad ya que se encontrará la mayor parte del tiempo inactiva esperando una acción por parte del usuario y queremos una respuesta rápida con el fin de proporcionar sensación de viveza de la aplicación.

Hilos daemon

Antes de lanzar un hilo, éste puede ser definido como un Daemon, indicando que va a hacer una ejecución continua para la aplicación como tarea de fondo. Entonces la máquina virtual abandonará la ejecución cuando todos los hilos que no sean Daemon hayan finalizado su ejecución. Los hilos Daemon tienen la prioridad más baja. Se usa el método setDaemon(true) para marcar un hilo como hilo demonio y se usa getDaemon() para comprobar ese indicador. Por defecto, la cualidad de demonio se hereda desde el hilo que crea el nuevo hilo. No puede cambiarse después de haber iniciado un hilo. La propia MVJ pone en ejecución algunos hilos daemon cuando ejecutamos un programa. Entre ellos cabe destacar el garbage collector o recolector de basura, que es el encargado de liberar la memoria ocupada por objetos que ya no están siendo referenciados.

Hay que señalar que cuando en la JVM solo se ejecuten hilos que sean demonios esta se dentendrá y acabará la ejecución del programa,dado que al ser hilos de servicio y no haber otros hilos a los que prestar ese servicio se considera que no queda más que hacer.

Control de Hilos

El control de hilos, se basa en mover hilos de estado a estado. La idea es controlar los hilos disparando estados de transición. Esta sección examina varias formas del estado en ejecución (Running). Estas formas son

- Yielding
- Suspendido y luego resumido
- Dormido y luego despertado
- Bloqueado y luego continuando
- Esperando y luego ser anunciado

Yielding (ceder el paso)

Este método pasa al hilo que lo llama al estado de ejecución a listo, permitiendo que el resto de los hilos compitan por el uso de CPU. Es importante en hilos de una alta prioridad y que podrían bloquear el sistema al no dejar al resto de hilos ejecutarse nunca.

Sin embargo, si ningún otro hilo está esperando, entonces el hilo que está cediendo pasara a ejecutarse inmediatamente.

El método yield() es un método estático de la clase Thread . Este siempre origina que el hilo ejecutándose corrientemente ceda el paso a otros hilos. Por ejemplo, considere un applet que calcula una imagen de pixeles 300 x 300 usando un algoritmo ray-tracing. El applet podría tener un botón "Calcular" y un botón de "Interrupir". El evento generado por el botón "Calcular" crearía e

iniciaría un hilo separado, el cual llamaría un método traceRaysQ. Una primera parte para este código podría verse como sigue:

```
1  private void traceRays() {
2    for (int j=0; j<300; j++) {
3      for (int i=0; i<300;i++) {
4        computeOnePixel(i, j);
5      }
6    }
7  }
```

Hay 90,000 valores color de pixeles que calcular. Si toma 0.1 segundo calcular el valor del color de un pixel, entonces tomaría dos horas y media calcular la imagen completa.

Suponga que después de media hora el usuario ve la imagen parcialmente y se da cuenta que algo está mal. (Quizás el punto de vista o el factor de zoom son incorrectos.) El usuario hará click en el botón "Interrumpir" de esta forma no tiene sentido seguir calculando la imagen inservible. Desafortunadamente, el hilo que maneja la entrada GUI podría quizás no conseguir una oportunidad de ejecutarse hasta que el hilo que está ejecutando traceRays() suelte la CPU. De esta manera el botón "Interrumpir" podría no tener efecto durante las otras dos horas.

Si las prioridades son implementadas en el programador, entonces bajando la prioridad del hilo ray-tracing tendrá el efecto deseado, asegurando que el hilo GUI correrá cuando este tenga algo útil que hacer. Sin embargo, esto no es confiable entre plataformas (aunque es una buena acción de cualquier forma, dado que este trabajará en muchas plataformas). La aproximación confiable es tener el hilo ray-tracing periódicamente cediendo. Si no hay entradas pendientes cuando el hilo yield es ejecutado, entonces el hilo raytracing no dejará la CPU. Si, de otro modo, hay una entrada para ser procesada, el hilo input-listening tendrá la oportunidad de ejecutarse.

El hilo ray-tracing puede obtener su prioridad como se muestra a continuación:

```
rayTraceThread.setPriority(Thread.NORM_PRIORITY-1);
```

El método traceRays() listado anteriormente puede ceder después que cada valor de pixel es calculado, después de la línea 4. La versión revisada se ve como sigue:

```
1  private void traceRays() {
2    for (int j=0; j<300; j++) {
3      for (int i=0; i<300;i++) {
4        computeOnePixel(i, j);
5        Thread.yield();
6      }
7    }
8  }
```

Suspendido (Suspending)

El método suspend() es distinto de stop(). suspend() toma el hilo y provoca que se detenga su ejecución sin destruir el hilo de sistema subyacente, ni el estado del hilo anteriormente en

ejecución. Si la ejecución de un hilo se suspende, puede llamarse a resume() sobre el mismo hilo para lograr que vuelva a ejecutarse de nuevo. Puede resultar útil suspender la ejecución de un hilo sin marcar un límite de tiempo. Si, por ejemplo, está construyendo un applet con un hilo de animación, seguramente se querrá permitir al usuario la opción de detener la animación hasta que quiera continuar. No se trata de terminar la animación, sino desactivarla. Para este tipo de control de los hilos de ejecución se puede utilizar el método suspend().

resume()

El método resume() se utiliza para revivir un hilo suspendido. No hay garantías de que el hilo comience a ejecutarse inmediatamente, ya que puede haber un hilo de mayor prioridad en ejecución actualmente, pero resume() ocasiona que el hilo vuelva a ser un candidato a ser ejecutado.

Dormido (Sleeping)

Un hilo dormido pasa el tiempo sin hacer nada y sin usar la CPU. Una llamada al método sleep() requiere que el hilo se esté ejecutando corrientemente para cesar su ejecución por una cantidad específica de tiempo. Hay dos formas de llamar este método, dependiendo si se quiere especificar el periodo para dormir con una precisión en milisegundos o nanosegundos:

- public static void sleep(long milliseconds) throws InterruptedException
- public static void sleep(long mili, int nano) throws InterruptedException

Note que sleep(), como yield(), es estático. Ambos métodos operan sobre el hilo que se está ejecutando corrientemente. Note que cuando el hilo ha finalizado el estado sleeping, este no continúa la ejecución. Como podría esperarse, este entra al estado listo y solamente se ejecutará cuando el programador de hilos se lo permita. Por esta razón, podría esperarse que una llamada a sleep() bloqueará un hilo al menos por el tiempo requerido, pero este podría bloquearse por mucho más tiempo. Esto sugiere piense con cuidado lo que haría en dado momento su diseño antes de esperar cualquier resultado de la precisión de la versión nanosegundos del método sleep().

Bloqueado (Blocking)

Muchos métodos que ejecutan entrada o salida deben esperar por alguna ocurrencia en el mundo exterior antes que ellos puedan proceder; este comportamiento es conocido como bloqueado (blocking). Un buen ejemplo es la lectura de un socket:

```
1   try {
2     Socket sock = new Socket("magnesium", 5505);
3     InputStream istr = sock.getInputStream();
4     int b = istr.read();
5   }
6   catch (IOException ex) {
7     // Manejador para la excepción
8   }
```

Si no está familiarizado con sockets en Java y funcionalidad de flujo, no se preocupe, el problema no es complicado. Esta muestra como en la Linea 4 lee un byte de una entrada de flujo que es

conectada al puerto 5505 sobre una maquina llamada "magnesium." Actualmente, la línea 4 trata de leer un byte. Si un byte está disponible (Si magnesium tiene previamente escrito un byte), entonces la línea 4 puede retornar inmediatamente y la ejecución puede continuar. Si magnesium no tiene escrito nada, sin embargo, la llamada a read() tiene que esperar. Si magnesium está ocupada haciendo otras cosas y le toma una hora y media para escribir un byte de nuevo, entonces el llamado read() tendrá que esperar por una hora y media. Claramente, esto podría ser un serio problema si el hilo en ejecución llama a read() en la línea 4 permaneciendo en el estado ejecución (Running) por toda la media hora . Note que esto podría pasar. En general, si un método necesita esperar una cantidad de tiempo indeterminada hasta que algún evento de I/O tome lugar, entonces un hilo en ejecución podría pasar fuera del estado de ejecución. Todos los métodos de I/O de Java tienen este comportamiento. Un hilo de este tipo pasa al estado bloqueado.

Estado Bloqueado (Blocked)

En general, si usted ve un método con un nombre que sugiere que este podría no hacer nada hasta que algo esté listo, por ejemplo esperar por una entrada, se esperaría que la llamada al hilo quedara bloqueada, manteniendo este estado hasta que, por ejemplo, se produzca una entrada.

interrupt()

El último elemento de control que se necesita sobre los hilos de ejecución es su detención. En programas simples es posible dejar que el hilo termine "naturalmente". En programas más complejos se requiere gestionar esta finalización. Para ello se definió el método stop(). Debido a serios defectos, fue desestimado (Ya JDK 1.3 lo señala como error de compilación). En su lugar debe usarse interrupt(), que permite la cancelación cooperativa de un hilo. Si se necesita, se puede comprobar si un hilo está vivo o no; considerando vivo un hilo que ha comenzado y no ha sido detenido.

Métodos de instancia

Se los llama así por estar directamente relacionados con objetos. Algunos de ellos son:

setName(String)

Este método permite identificar al hilo con un nombre mnemónico. De esta manera se facilita la depuración de programas multihilo. El nombre mnemónico aparecerá en todas las líneas de trazado que se muestran cada vez que el intérprete Java imprime excepciones no capturadas.

getName()

Este método devuelve el valor actual, de tipo cadena, asignado como nombre al hilo en ejecución mediante setName().

getId()

Devuelve un tipo de dato long que es interpretado como identificador único del hilo.

isAlive();

Este método devolverá true en caso de que el hilo t1 esté vivo, es decir, ya se haya llamado a su método run() y no haya sido parado con un stop() ni haya terminado el método run() en su ejecución. En el ejemplo no hay problemas de realizar una parada incondicional, al estar todos los hilos vivos.

getPriority()

Devuelve un número entero que indica la prioridad del hilo.

setPriority(int newPriority)

Permite asignar o cambiar la prioridad del hilo.

setDaemon(boolean on)

Permite indicar mediante un booleano si el hilo se trata de un demonio o un hilo de usuario. En el caso de que el valor booleano sea **true** se indicará que el hilo es un demonio, en caso contrario se indicará que es un hilo de usuario.

isDaemon()

Método que devuelve un booleano indicando si el hilo es demonio o no.

isInterrupted()

Método que devuelve un booleano indicando si el hilo ha sido interrumpido.

Para más información se puede consultar la documentación oficial de Oracle
https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

Grupos de hilos

Todo hilo de ejecución en Java forma parte de un grupo denominado ThreadGroup que define e implementa la capacidad de un grupo de hilos. Los grupos de hilos permiten que sea posible recoger varios hilos de ejecución en un solo objeto y manipularlo como un grupo, en vez de individualmente. Cuando se crea un nuevo hilo, se coloca en un grupo, bien indicándolo explícitamente, o bien dejando que el sistema lo coloque en el grupo por defecto. Una vez creado el hilo y asignado a un grupo, ya no se podrá cambiar a otro grupo. Si no se especifica un grupo en el constructor, el sistema coloca el hilo en el mismo grupo en que se encuentre el hilo de ejecución que lo haya creado, y si no se especifica en grupo para ninguno de los hilos, entonces todos serán miembros del grupo "main", que es creado por el sistema cuando arranca la aplicación Java. La división de hilos en grupos permite una gestión y un tratamiento homogéneo a todos los hilos del grupo como si se tratara de uno solo, pudiendo cambiar su prioridad, detenerlos, interrumpirlos, enumerarlos, obtener información de su estado, etc. La clase Thread proporciona constructores en los que se puede especificar el grupo del hilo que se está creando en el mismo momento de instanciarlo, y también

métodos como setThreadGroup(), que permiten determinar el grupo en que se encuentra un hilo de ejecución.

La clase Thread proporciona los siguientes constructores para la utilización de hilos:

- Thread()
- Thread(Runnable target)
- Thread(Runnable target, String name)
- Thread(String name)
- Thread(ThreadGroup group, Runnable target)
- Thread(ThreadGroup group, Runnable target, String name)
- Thread(ThreadGroup group, Runnable target, String name, long stackSize)
- Thread(ThreadGroup group, String name)

Ejemplo de utilización:

```
1  ThreadGroup group = new ThreadGroup("MiGrupo");
2  Thread h1 = new Thread(group,new HiloRunnable());
3  Thread h2 = new Thread(group,new HiloRunnable());
4  Thread h3 = new Thread(group,new HiloRunnable());
5  group.list(); //muestra información de todos los hilos del grupo
```

Sincronización

Cuando dos hilos necesitan utilizar el mismo recurso (objeto), aparece la posibilidad de operaciones entre -lazadas que pueden corromper los datos. Estas acciones potencialmente interferentes se denominan secciones críticas o regiones críticas. La interferencia se puede evitar sincronizando el acceso a esas regiones críticas. La acción equivalente cuando hay múltiples hilos es adquirir un bloqueo de un objeto. Los hilos cooperan acordando el protocolo de que antes de que ciertas acciones puedan ocurrir en un objeto se debe adquirir el bloqueo del objeto. Adquirir el bloqueo de un objeto impide que cualquier otro hilo adquiera ese bloqueo, hasta que el que posee el bloqueo lo libere. Si se hace correctamente, los múltiples hilos no realizarán acciones que puedan interferir entre sí. Todos los objetos tienen un bloqueo asociado, que puede ser adquirido y liberado mediante el uso de métodos y sentencias synchronized. El término código sincronizado describe a cualquier código que esté dentro de un método o sentencia synchronize. Un ejemplo podría ser muchos hilos que intentan acceder a una impresora, esto pone en peligro la estabilidad del sistema y la consistencia de la información.

Métodos synchronized

Una clase cuyos objetos se deben proteger de interferencias en un entorno con múltiples hilos declara generalmente sus métodos modificadores como synchronized. Si un hilo invoca a un método synchronized sobre un objeto, en primer lugar se adquiere el bloqueo de ese objeto, se ejecuta el cuerpo del método y después se libera el bloqueo. Otro hilo que invoque un método synchronized sobre ese mismo objeto se bloqueará hasta que el bloqueo se libere.

La sincronización fuerza a que la ejecucion de los dos hilos sea mutuamente exclusiva en el tiempo. Los accesos no sincronizados no respetan ningún bloqueo, proceden independientemente de cualquier bloqueo