CAPÍTULO 2. PROCESOS VS. HILOS

El objetivo de este capítulo es el de discutir con un poco más de profundidad todo lo relacionado con el concepto de proceso que sea relevante para el estudio de la programación concurrente. En esta discusión aparecerá un nuevo concepto que es el de hilo o hebra[1] y que resulta fundamental para entender cómo es la programación concurrente en Java.

En primer lugar veremos el ciclo de vida de un proceso y las nociones de planificación de procesos y cambio de contexto. Posteriormente, se verá la disposición en memoria de las estructuras de datos relacionadas con un proceso. Por último nos adentraremos en el mundo de los hilos y su relación con los procesos y, particularmente, nos centraremos en el estudio de la gestión de hilos en Java.

2.1 Procesos

2.1.1 Ciclo de vida de un proceso

En la Figura 1 puede apreciarse el ciclo de vida que suele seguir un proceso. Este ciclo de vida es prácticamente estándar en todos los SSOO.

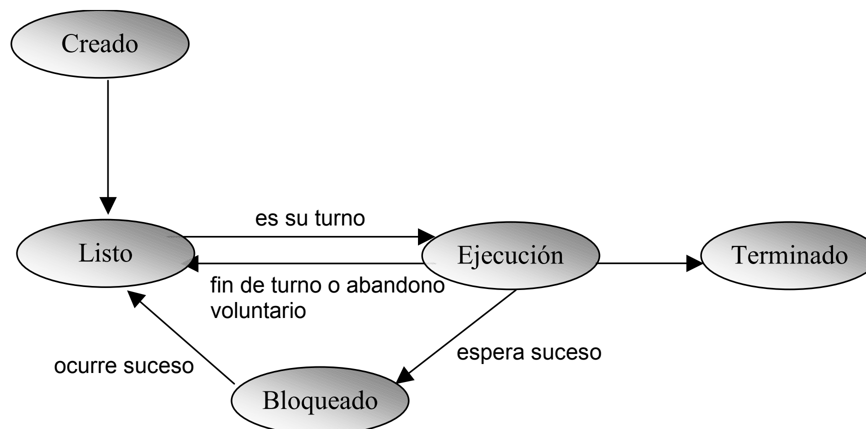

Figura 1 Estados de un proceso.

En un principio, un proceso no existe. En algún momento es *creado*. La forma de crearlo variará en función del lenguaje que se esté utilizando. Una vez creado el proceso, éste pasa al estado denominado *Listo*. Este estado significa que el proceso está en condiciones de hacer uso de la CPU en cuanto se le dé la oportunidad. El encargado de darle la oportunidad de usar la CPU es el denominado **planificador de procesos** o **scheduler**, que suele formar parte del SO. Como lo normal es que haya más procesos que procesadores, no todos los procesos que pueden hacer uso de la CPU en un momento determinado pueden hacerlo realmente. Esos procesos permanecen en el

[1] Suelen ser las traducciones más aceptadas del término inglés *thread*. A lo largo de este libro utilizaremos el término hilo. Usaremos Thread cuando hagamos referencia a la clase con ese nombre en Java.

estado listo hasta que el planificador decide darles tiempo de CPU. Cuando el planificador decide dar tiempo de CPU a un proceso, éste pasa al estado de *Ejecución*.

Como puede comprobarse en la Figura 1, un proceso también puede pasar de Ejecución a Listo. Esta decisión la toma el planificador. El planificador sigue algún tipo de política de planificación para asignar la CPU a los distintos procesos. Una forma bastante justa y extendida de hacerlo es mediante la asignación de **rodajas de tiempo** a cada uno de los procesos, de tal forma que cuando un proceso cumple su tiempo de permanencia en el procesador, éste es desalojado y pasado al estado listo. En este estado esperará una nueva oportunidad para pasar a Ejecución. También es posible que un proceso abandone voluntariamente la CPU y pase de esta forma al estado listo.

También en la Figura 1 puede apreciarse la presencia del estado *Bloqueado*. Un proceso puede pasar de Ejecución a Bloqueado cuando ha de esperar porque ocurra un determinado evento o suceso. Ejemplos de eventos pueden ser la espera por la terminación de una operación de Entrada/Salida, la espera por la finalización de una tarea por parte de otro proceso, un bloqueo voluntario durante un cierto período de tiempo, etc. Una vez que ocurre el evento por el que se está esperando, el proceso pasa al estado Listo. Al acto de cambiar un proceso de estado se le denomina **cambio de contexto**.

2.1.2 Disposición en memoria de un proceso

En un Sistema Operativo tradicional, la memoria suele estar dividida en dos partes: un **espacio de usuario** donde suele encontrarse la mayoría de la información relativa a los procesos de usuario y un **espacio del núcleo** donde reside el código y las estructuras de datos propios del SO.

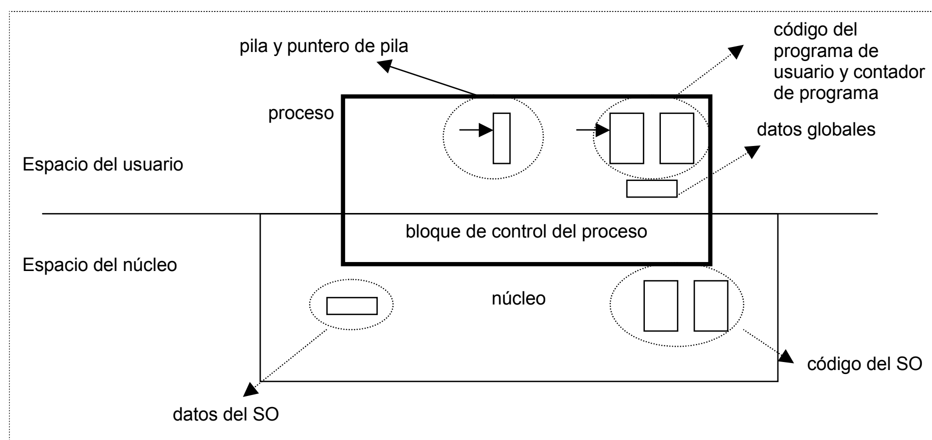

Figura 2 Mapa de memoria de un proceso para un SO multitarea.

Cuando estamos ante un SO multitarea, la información relativa a un proceso suele estar dividida entre los dos espacios (Figura 2). En el espacio de usuario se encuentra información propia del proceso tales como el código del proceso, el contador de programa, sus variables, su pila y su puntero de pila. Sin embargo, el SO necesita tener información del estado de los procesos para poder realizar apropiadamente los cambios de contexto. Esta información, que suele conocerse con el nombre de **bloque de control del proceso (PCB)**,

suele residir en el espacio del núcleo. En la Figura 3 puede verse la estructura de un proceso en el SO Unix. La información contenida en la estructura del proceso puede variar de un SO a otro, pero sustancialmente suele ser la misma.

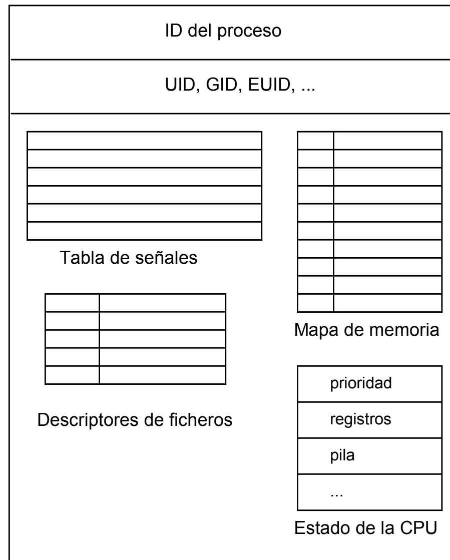

Figura 3 Estructura tradicional de un proceso en UNIX.

Cuando tenemos más de un proceso, se tiene algo como lo representado en la Figura 4. En el espacio del núcleo estará el planificador de procesos que será el encargado de decidir cuándo hacer los cambios de contexto. Cuando se hace el cambio de contexto hay que recuperar la estructura del proceso que se quiere poner en el estado Ejecución y actualizar convenientemente los registros del procesador para que el nuevo proceso tome el control del mismo. Los cambios de contexto son costosos desde el punto de vista del tiempo de ejecución pues consumen un tiempo considerable.

Cada proceso de los representados en la figura tiene su propio contador de programa, su propia pila, etc. Cada proceso suele tener un solo hilo de ejecución; se dice entonces que son monohilo.

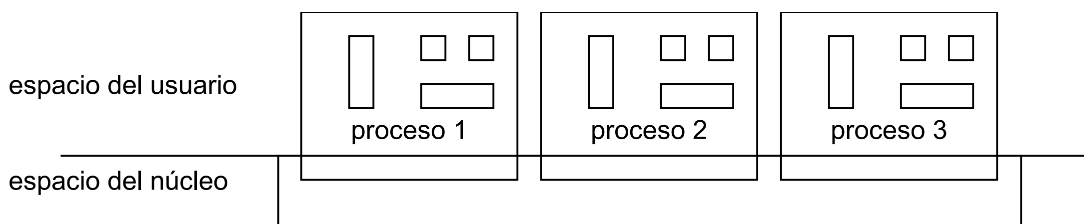

Figura 4 Varios procesos en un SO multitarea.

2.2 Hilos

Aunque el concepto de hilo lleva existiendo varias décadas, no ha sido hasta los 90 cuando ha alcanzado una cierta mayoría de edad. Mientras que a comienzos de los 90 el uso de hilos se circunscribía a la investigación en universidades y al desarrollo en sectores industriales específicos, a finales de los 90 y en el nuevo milenio, el uso de hilos está ampliamente extendido. La

incorporación del concepto de hilo al lenguaje Java ha supuesto un empuje definitivo a su uso.

Pero, ¿qué es un hilo?. De la misma manera que un Sistema Operativo puede ejecutar varios procesos al mismo tiempo bien sea por concurrencia o paralelismo, dentro de un proceso puede haber varios hilos de ejecución. Por tanto, **un hilo puede definirse como cada secuencia de control dentro de un proceso que ejecuta sus instrucciones de forma independiente**.

En la Figura 5 puede verse cómo sobre el hardware subyacente (una o varias CPU's) se sitúa el Sistema Operativo. Sobre éste se sitúan los procesos (P_i) que pueden ejecutarse concurrentemente y dentro de estos se ejecutan los hilos (h_j) que también se pueden ejecutar de forma concurrente dentro del proceso. Es decir, tenemos concurrencia a dos niveles, una entre procesos y otra entre hilos de un mismo proceso. Si por ejemplo tenemos dos procesadores, se podrían estar ejecutando al mismo tiempo el hilo 1 del proceso 1 y el hilo 2 del proceso 3. Otra posibilidad podría ser el hilo 1 y el hilo 2 del proceso 1.

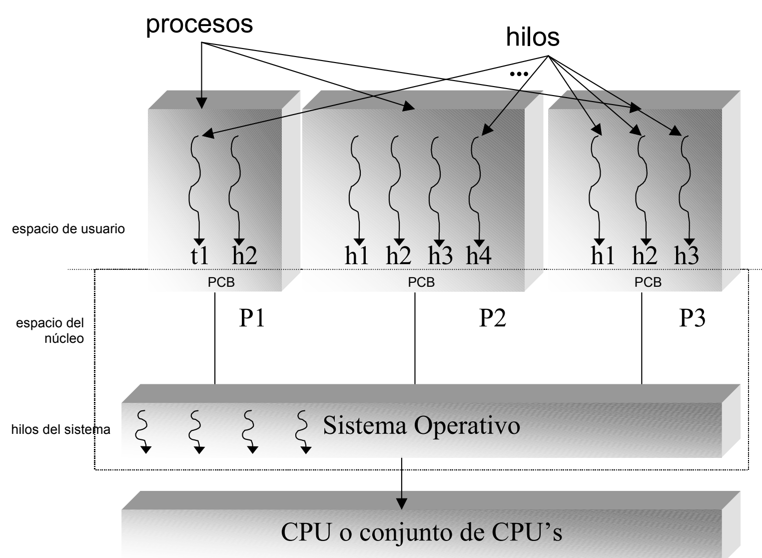

Figura 5 Concurrencia a dos niveles: procesos e hilos.

Los procesos son **entidades pesadas**. La estructura del proceso está en la parte del núcleo y, cada vez que el proceso quiere acceder a ella, tiene que hacer algún tipo de llamada al sistema, consumiendo tiempo extra de procesador. Por otra parte, los cambios de contexto entre procesos son costosos en cuanto a tiempo de computación se refiere. Por el contrario, la estructura de los hilos reside en el espacio de usuario, con lo que un hilo es una **entidad ligera**. Los hilos comparten la información del proceso (código, datos, etc). Si un hilo modifica una variable del proceso, el resto de hilos verán esa modificación cuando accedan a esa variable. Los cambios de contexto entre hilos consumen poco tiempo de procesador, de ahí su éxito.

Podemos hablar de dos niveles de hilos: aquellos que nosotros usaremos para programar y que pueden crearse desde lenguajes de programación como Java y aquellos otros hilos del propio SO que sirven para dar soporte a nuestros hilos de usuario y que son los **hilos del sistema** según la Figura 5. Cuando nosotros programamos con hilos, hacemos uso de un API (Application

Program Interface) proporcionado por el lenguaje, el SO o un tercero mediante una librería externa. El implementador de este API será el que haya usado los hilos del sistema para dar soporte de ejecución a los hilos que nosotros creamos en nuestros programas. En la Figura 6 puede observarse este hecho aplicado a Java.

Nuestro interés radica en aquellos hilos del espacio de usuario, es decir, aquellos hilos que uno crea para la construcción de sus programas. Sin embargo, aunque no es el objetivo principal de este libro, nos adentraremos un poco en la forma en la que pueden ser implementados los hilos del sistema para entender ciertas cosas que ocurren con los hilos de Java, pues el comportamiento de éstos a veces depende de la forma en la que se gestione la librería de hilos del sistema.

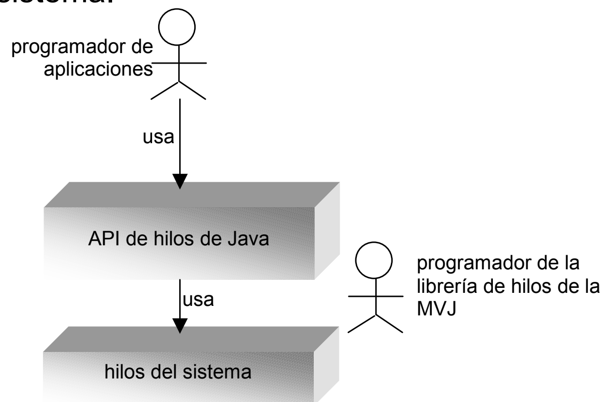

Figura 6 Dos niveles de hilos.

2.2.1 Estándares de hilos

Cada SO implementa los hilos del sistema como quiere, aunque se puede hablar de la existencia de tres estándares. Existen tres librerías nativas diferentes de hilos que compiten hoy en día para ser las más usadas: **Win32, OS/2 y POSIX**. Las dos primeras son propietarias y sólo corren bajo sus respectivas plataformas (NT, Win95, OS/2). La especificación POSIX (IEEE 1003.1c) conocida como *pthreads* [Butenhof, 1997] está pensada para todas las plataformas y está disponible para la mayoría de las implementaciones UNIX y Linux, así como VMS y AS/400.

Por su parte, los hilos de Java están implementados en la MVJ (Máquina Virtual Java) que a su vez está construida sobre la librería de hilos nativas de la correspondiente plataforma (ver Figura 6). Java sólo ofrece su API para manejar hilos, independientemente de la librería subyacente. De esta forma es mucho más fácil utilizar las hilos de Java. Sin embargo, y como se verá posteriormente, hay algunos asuntos importantes a tener en cuenta para que un programa multihilo en Java sea independiente de la plataforma.

2.2.2 Implementación de hilos

Hay fundamentalmente 2 formas de implementar hilos. La primera es escribir una **librería al nivel de usuario**. Todas las estructuras y el código de la librería estarán en espacio de usuario. La mayoría de las llamadas que se hagan desde la librería se ejecutarán en el espacio de usuario y no hará más uso de las rutinas del sistema que cualquier otra librería o programa de usuario. La

segunda forma es una **implementación al nivel de núcleo**. La mayoría de las llamadas de la librería requerirán llamadas al sistema.

Algunas de las implementaciones del estándar POSIX son del primer tipo, mientras que tanto OS/2 como Win32 son del segundo tipo. Ambos métodos pueden ser usados para implementar exactamente el mismo API. En el caso de Java, el implementador de la MVJ es el que tendrá que pelearse con el API de esa librería. El programador de aplicaciones usará el API implementado por la librería de hilos de la MVJ. Al programador de aplicaciones le debe dar igual cómo esté implementada la librería de la MVJ.

Para una discusión más a fondo de la implementación de hilos se sugiere consultar [Lewis and Berg, 2000].

2.2.3 Planificación de hilos

Los SSOO modernos como Solaris definen el concepto de procesador lógico de tal forma que donde se ejecutan los hilos de usuario es en un procesador lógico. En la terminología de Solaris a estos procesadores lógicos se les denomina LWP (Light-Weight Processes)

De esta forma vamos a tener una planificación a dos niveles. Un primer nivel para asignar los hilos de usuario a los procesadores lógicos y otro para asignar los procesadores lógicos al procesador o procesadores físicos. En la Figura 7 pueden observarse estos dos niveles según la filosofía de Solaris. Los hilos de usuario compiten por los procesadores lógicos (primer nivel de planificación) mientras que estos a su vez competirán por los hilos del sistema que son los que directamente se ejecutarán en los procesadores físicos (segundo nivel de planificación)

Hay principalmente tres técnicas distintas para hacer la planificación de hilos sobre los recursos del núcleo (indirectamente sobre las distintas CPU's). Se describen a continuación.

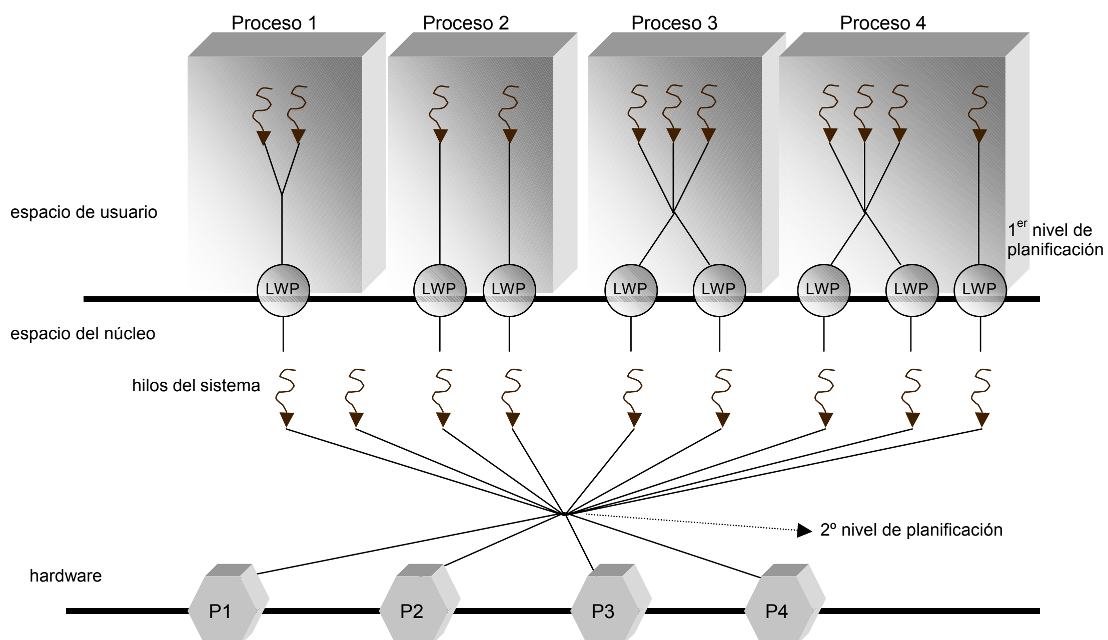

Figura 7 Planificación de procesos e hilos en Solaris.

Muchos hilos en un procesador lógico

Conocido como el modelo *muchos-a-uno*. Todos los hilos creados en el espacio de usuario hacen turnos para ir ejecutándose en el único procesador lógico asignado a ese proceso. De esta forma un proceso no toma ventaja de una máquina con varios procesadores físicos. Otra desventaja es que cuando se hace una llamada bloqueante, p. ej. de E/S, todo el proceso se bloquea. Es algo parecido a lo que ocurría con los procesos en Pascal-FC pero aquí al nivel de hilo. Como ventaja podemos decir que en este modelo la creación de hilos y la planificación se hacen en el espacio de usuario al 100%, sin usar los recursos del núcleo. Por tanto es más rápido. Sería el caso del proceso 1 en la Figura 7.

Un hilo por procesador lógico

También llamado como modelo *uno-a-uno*. Aquí se asigna un procesador lógico para cada hilo de usuario. Este modelo permite que muchos hilos se ejecuten simultáneamente en diferentes procesadores físicos. Sin embargo, tiene el inconveniente de que la creación de hilos conlleva la creación de un procesador lógico y por tanto una llamada al sistema. Como cada procesador lógico toma recursos adicionales del núcleo, uno está limitado en el número de hilos que puede crear. Win32 y OS/2 utilizan este modelo. Algunas implementaciones de POSIX como los LinuxThreads de Xavier Leroy también lo usan. Cualquier MVJ basada en estas librerías también usa este modelo, así pues Java sobre Win32 lo usa. Sería el caso del proceso 2 en la Figura 7.

Muchos hilos en muchos procesadores lógicos

Llamado modelo *muchos-a-muchos* (estricto). Un número de hilos es multiplexado en un número de procesadores lógicos igual o menor. Numerosos hilos pueden correr en paralelo en diferentes CPUs y las llamadas al sistema de tipo bloqueante no bloquean al proceso entero. Sería el caso del proceso 3 en la Figura 7.

El modelo de dos niveles

Llamado modelo *muchos-a-muchos* (no estricto). Es como el anterior pero ofrece la posibilidad de hacer un enlace de un hilo específico con un procesador lógico. Probablemente sea el mejor modelo. Varios sistemas operativos usan este modelo (Digital UNIX, Solaris, IRIX, HP-UX, AIX). En el caso de Java, las MVJ sobre estos SSOO tienen la opción de usar cualquier combinación de hilos enlazadas o no. La elección del modelo de hilos es una decisión al nivel de implementación de los escritores de la MVJ. Java en sí mismo no tiene el concepto de procesador lógico. Sería el caso del proceso 4 en la Figura 7.

Con esto daríamos por terminada la discusión sobre hilos en general y nos adentramos en el mundo de los hilos en Java, es decir, en el mundo de los hilos desde una perspectiva de programador de aplicaciones y no de Sistema Operativo.

2.3 Hilos en java

Java proporciona un API para el uso de hilos. Este API es bastante simple (aproximadamente una veintena de métodos) comparado con otras librerías de hilos como Posix que alcanza el medio centenar de métodos.

Como prácticamente todo en Java, los hilos se representan mediante una clase, la clase Thread. Esta clase se encuentra en el paquete java.lang.Thread. Los métodos de esta clase junto con algunos métodos de la clase Object son los que permiten un manejo prácticamente completo de hilos en Java.

En esta primera incursión en el mundo de los hilos en Java veremos la diferencia entre hilos y objetos, las dos formas distintas que hay de crear hilos en Java, el ciclo de vida de un hilo y las prioridades de los hilos.

2.3.1 Hilos y objetos

Nada más empezar un programa en Java, existe un hilo de ejecución que es el indicado por el método main, es el denominado hilo principal. Si no se crean más hilos desde el hilo principal tendremos algo como lo representado en la Figura 8 donde sólo existe un hilo de ejecución que va recorriendo los distintos objetos participantes en el programa según se vayan produciendo las distintas llamadas entre métodos de los mismos.

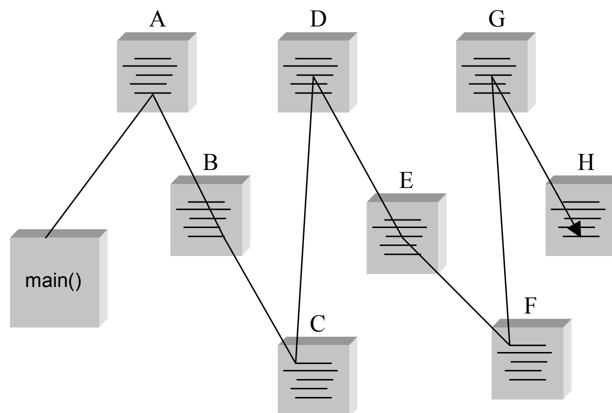

Figura 8 Varios objetos y un solo hilo.

Sin embargo, si desde el hilo principal se crean por ejemplo dos hilos tendremos algo como lo representado en la Figura 9, en la que se puede ver cómo el programa se está ejecutando al mismo tiempo por tres sitios distintos: el hilo principal más los dos hilos creados. Los hilos pueden estar ejecutando código en diferentes objetos, código diferente en el mismo objeto o incluso el mismo código en el mismo objeto y al mismo tiempo.

Es necesario en este punto ver la diferencia entre objeto e hilo. Un objeto es algo estático, tiene una serie de atributos y métodos. Quien realmente ejecuta esos métodos es el hilo de ejecución. En ausencia de hilos sólo hay un hilo que va recorriendo los objetos según se van produciendo las llamadas entre métodos de los objetos. Podría darse el caso del objeto E en la Figura 9, donde es posible que tres hilos de ejecución distintos estén ejecutando el mismo método al mismo tiempo.

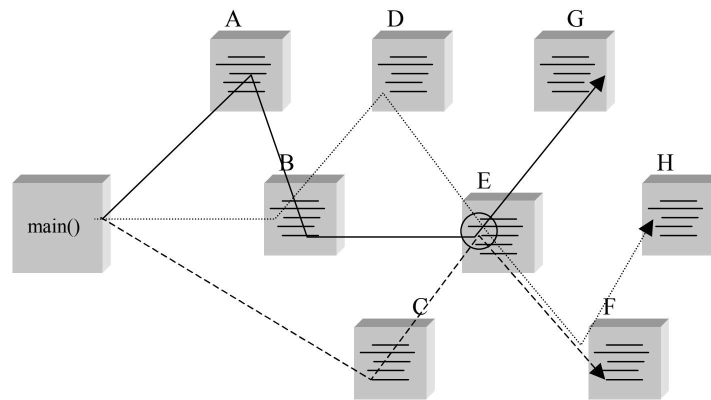

Figura 9 Tres hilos "atravesando" varios objetos.

2.3.2 Creación de hilos

Existen 2 formas de trabajar con hilos en cuanto a su creación se refiere:

- Heredando de la clase Thread
- Implementando la interfaz Runnable

Heredando de Thread y redefiniendo el método run

La clase Thread tiene un método especial cuya signatura es public void run (). Cuando queramos crear una clase cuyas instancias queremos que sean un hilo tendremos que heredar de la clase Thread y redefinir este método. Dentro de este método se escribe el código que queremos que se ejecute cuando se lance a ejecución el hilo. Se podría decir que main() es a un programa lo que run() es a un hilo.

Veamos un ejemplo en el que creamos una clase denominada ThreadConHerencia. Si queremos que los objetos pertenecientes a esta clase sean hilos entonces debemos extender la clase Thread. El constructor de la clase ThreadConHerencia recibe como parámetro una palabra a imprimir. En el método run() especificamos el código que ha de ejecutarse al lanzar el hilo: la impresión de la palabra 10 veces.

```
class ThreadConHerencia extends Thread {
  String palabra;


public ThreadConHerencia (String _palabra) {
  palabra = _palabra;
}

public void run( ) {
  for (int i=0; i<10; i++)
    System.out.print (palabra);
}
```

Hasta aquí lo único que hemos hecho es crear una clase. Los objetos pertenecientes a esa clase serán hilos. En las siguientes líneas, desde el programa principal creamos dos objetos que son hilos, a y b.

```
public static void main(String[] args) {
  Thread a = new ThreadConHerencia ("hiloUno");
  Thread b = new ThreadConHerencia ("hiloDos");
```

Hasta aquí los hilos están creados, pero no han sido puestos en ejecución. Para ponerlos en ejecución hay que invocar el método start(). Este método pertenece a la clase Thread. Se encarga de hacer algunas inicializaciones propias del hilo y posteriormente invoca al método run(). Es decir, invocando start, se ejecutan en orden los métodos start (heredado) y run (redefinido)

Las líneas siguientes habría que añadirlas al programa principal y ya tendríamos nuestro primer programa con hilos en Java.

```
  a.start();
  b.start();
  System.out.println ("Fin del hilo principal")
}
```

Una posible salida para este programa sería:

hiloDos hiloDos hiloUno hiloDos Fin del Hilo principal hiloUno hiloUno hiloUno hiloUno hiloUno hiloDos hiloDos hiloDos hiloDos hiloDos hiloUno hiloUno hiloUno hiloUno hiloDos hiloDos hiloDos

Puede apreciarse cómo se intercalan las salidas de los tres hilos. Recordemos que tenemos tres hilos: el principal más los dos creados. Y entre ellos se pueden intercarlar de cualquier forma, por tanto no es extraño ver cómo el hilo principal acaba antes que los otros o cómo el hilo dos comienza antes que el uno.

Implementando la interfaz Runnable

Volveremos a implementar el mismo ejemplo pero usando esta otra forma de crear hilos. En las siguientes líneas de código creamos una clase denominada ThreadConRunnable. Ahora, en vez de heredar de la clase Thread, implementamos la interfaz Runnable. Esta interfaz sólo tiene un método con la signatura public void run(). Este método es el que, como mínimo, tenemos que implementar en esta clase.

```
public class ThreadConRunnable implements Runnable {
  String palabra;

  public ThreadConRunnable (String palabra {
    palabra = _palabra;
  }

  public void run() {
    for (int i = 0;i<10;i++) {
      System.out.println (palabra);
  }
```

Hasta aquí simplemente hemos creado una clase. Al contrario que antes, los objetos de esta clase no serán hilos pues no hemos heredado de la clase

Thread. Si queremos que un objeto de la clase ThreadConRunnable se ejecute en un hilo independiente, entonces tenemos que crear un objeto de la clase Thread y pasarle como parámetro el objeto donde queremos que empiece su ejecución ese hilo. En las siguientes líneas de código se crean dos objetos de la clase ThreadConRunnable (a y b) y dos hilos (t1 y t2) a los que se le pasa como parámetro los objetos a y b. Posteriormente hay que invocar el método start() de la clase Thread que ya se encarga de invocar al método run() de los objetos a y b respectivamente.

```
  public static void main (String args[]) {
    ThreadConRunnable a = new ThreadConRunnable ("hiloUno");
    ThreadConRunnable b = new ThreadConRunnable ("hiloDos");
    Thread t1 = new Thread (a);
    Thread t2 = new Thread (b);
    t1.start();
    t2.start();
    System.out.println (“Fin del hilo principal”)
  }
}
```

Una posible salida de este ejemplo sería la misma de antes.

Comparando ambos métodos

La segunda forma de crear hilos puede parecer un poco más confusa. Sin embargo, es más apropiada y se utiliza más que la primera. Esto se debe a que como en Java no hay herencia múltiple, el utilizar la primera opción hipoteca nuestra clase para que ya no pueda heredar de otras clases. Sin embargo, con la segunda opción puedo hacer que el método run() de una clase se ejecute en un hilo y además esta clase herede el comportamiento de otra clase. En la medida de lo posible, deberíamos usar esta forma.

2.3.3 Objeto autónomo en un hilo

Independientemente del método que usemos para la creación de hilos, se puede apreciar cómo el hecho de que el objeto que implementa el método run se ejecute en un hilo depende del método cliente, en nuestro caso del método main. Es decir, el cliente, en nuestro caso el programa principal, tiene que lanzar a ejecución el nuevo hilo. Incluso en el caso de la implementación de la interfaz Runnable también tiene que crearlo explícitamente. Puede que a veces sea esto lo que queramos, que el cliente tenga el control de la creación y lanzamiento de hilos. Sin embargo, a veces es preferible un objeto autónomo que automáticamente se ejecute en un nuevo hilo, sin intervención del cliente. No en vano, esto es lo que ocurre con algunos objetos de Java. Sin nuestra intervención se ejecutan en un hilo independiente.

Veamos un ejemplo en el que al construir un objeto éste se ejecuta automáticamente en un nuevo hilo. Para ello, en el constructor creamos el hilo y lo lanzamos.

```
class AutoThread implements Runnable {
  private Thread me_;
```

```
  public AutoThread() {
     me_ = new Thread(this);
     me_.start();
  }

  public void run() {
    if (me_==Thread.currentThread())
      for (int i=0;i<10;i++)
        System.out.println ("Dentro de Autothread.run()");
  }

  public static void main(String[] args) {
    AutoThread miThread = new AutoThread();
    for (int i = 0;i<10;i++)
      System.out.println ("Dentro de main()");
  }
```

Como vemos, en la implementación del método run() hay que controlar quién es el hilo que se está ejecutando. Para ello nos servimos del método currentThread() de la clase Thread. Este método nos devuelve una referencia al hilo que está ejecutando ese código. Esto se hace para evitar que cualquier método de un hilo distinto haga una llamada a run() directamente, como ocurre en el siguiente caso.

```
  public static void main (String[] args) {
    AutoThread miThread = new AutoThread();
    miThread.run();
    while (true)
      System.out.println (“Dentro del main()”);
  }
}
```

2.3.4 Estados de un hilo en Java

El ciclo de vida de un hilo comprende diversos estados por los que puede ir pasando. La Figura 10 muestra estos estados y los métodos que provocan el paso de un estado a otro.

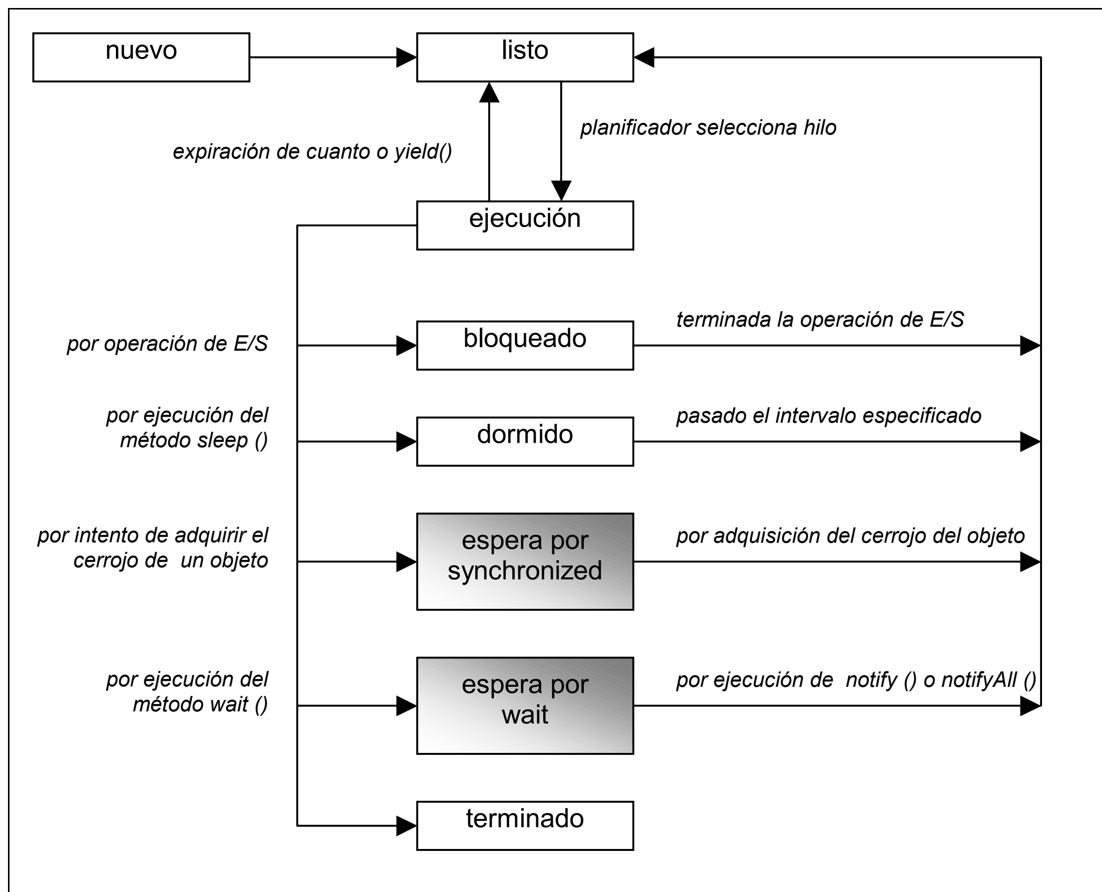

Figura 10 Estados de un hilo en Java.

Nuevo es el estado en el que se encuentra un hilo cuando se crea con el operador new. Al lanzarlo con start() pasa al estado *listo*, es decir, es susceptible de acaparar el procesador si el planificador se lo permite. Cuando obtiene el procesador pasa al estado *ejecución*. Una vez que el hilo está en ejecución puede pararse por diversos motivos:

- Por hacer una operación de E/S. Saldrá de este estado al completarse dicha operación.
- Por ejecutar el método sleep (milisegundos). Saldrá de este estado al cumplirse el tiempo especificado como parámetro.
- Por intentar adquirir el cerrojo de un objeto. Hablaremos del cerrojo de un objeto en el capítulo 4.
- Por ejecutar el método wait(). Saldrá de este estado cuando se ejecute el método notify() o notifyAll() por parte de otro hilo. Hablaremos de esto en el capítulo 4.

Como puede apreciarse, el ciclo de vida de un hilo es muy parecido al ciclo de vida de un proceso. El planificador de los hilos de nuestras aplicaciones Java se encuentra implementado en la MVJ. A continuación hacemos algunas consideraciones sobre cómo se hace esa planificación en Java, es decir, cómo es el paso del estado listo a ejecución y viceversa. Los dos estados sombreados serán tratados con profundidad en el tema 4.

2.3.5 Planificación y prioridades

Java fue diseñado principalmente para obtener una gran portabilidad. Este objetivo hipoteca de alguna forma también el modelo de hilos a usar en Java pues se deseaba que fuera fácil de soportar en cualquier plataforma y, como se ha visto, cada plataforma hace las cosas de una forma diferente. Esto hace que el modelo de hilos sea demasiado generalista. Sus principales características son:

- Todos los hilos de Java tienen una prioridad y se supone que el planificador dará preferencia a aquel hilo que tenga una prioridad más alta. Sin embargo, no hay ningún tipo de garantía de que en un momento determinado el hilo de mayor prioridad se esté ejecutando.
- Las rodajas de tiempo pueden ser aplicadas o no. Dependerá de la gestión de hilos que haga la librería sobre el que se implementa la máquina virtual java.

Ante una definición tan vaga como esta uno se puede preguntar cómo es posible escribir código portable y, lo que es peor, el comportamiento que tendrá la aplicación dependiendo del sistema subyacente. Ante esto, lo que el programador no debe hacer es ningún tipo de suposición y ponerse siempre en el peor de los casos:

- Se debe asumir que los hilos pueden intercalarse en cualquier punto en cualquier momento.
- Nuestros programas no deben estar basados en la suposición de que vaya a haber un intercalado entre los hilos. Si esto es un requisito en nuestra aplicación, deberemos introducir el código necesario para que esto sea así.

Prioridades

Las prioridades de cada hilo en Java están en el rango de 1 (*MIN_PRIORITY)* a 10 (*MAX_PRIORITY*). La prioridad de un hilo es inicialmente la misma que la del hilo que lo creó. Por defecto todo hilo tiene la prioridad 5 (*NORM_PRIORITY*). El planificador siempre pondrá en ejecución aquel hilo con mayor prioridad (teniendo en cuenta lo dicho en el punto anterior) Los hilos de prioridad inferior se ejecutarán cuando estén bloqueados los de prioridad superior.

Las prioridades se pueden cambiar utilizando el método setPriority (nuevaPrioridad). La prioridad de un hilo en ejecución se puede cambiar en cualquier momento. El método getPriority() devuelve la prioridad de un hilo.

El método yield()hace que el hilo actualmente en ejecución ceda el paso de modo que puedan ejecutarse otros hilos listos para ejecución. El hilo elegido puede ser incluso el mismo que ha dejado paso, si es el de mayor prioridad.

En el siguiente programa se puede ver cómo se crean dos hilos (t1 y t2) y al primero de ellos se le cambia la prioridad. Esto haría que t2 acaparase el procesador hasta finalizar pues nunca será interrumpido por un hilo de menor prioridad.

```
public class A implements Runnable {

  String palabra;
```

```
  public A (String _palabra)  {
    palabra = _palabra;
  }

  public void run () {
    for (int i=0;i<100;i++)
      System.out.println (palabra);
  }

  public static void main (String args[]) {
    A a1 = new A("a1");
    A a2 = new A("a2");
    Thread t1 = new Thread (a1);
    Thread t2 = new Thread (a2);
    t1.start();
    t1.setPriority(1);
    System.out.println ("Prioridad de t1: "+t1.getPriority());
    t2.start();
    System.out.println ("Prioridad de t2: "+t2.getPriority());
  }
}
```

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.

2.3.6 La clase Thread

A continuación mostramos los atributos y métodos de la clase Thread tratados en este capítulo y algunos más que consideramos de interés. No pretendemos dar un listado exhaustivo con el API completo. En el capítulo 4 se verán algunos métodos más, no de la clase Thread, pero sí relacionados con ella.

Atributos	
public static final int	**MIN_PRIORITY** La prioridad mínima que un hilo puede tener.
public static final int	**NORM_PRIORITY** La prioridad por defecto que se le asigna a un hilo.
public static final int	**MAX_PRIORITY**

	La prioridad máxima que un hilo puede tener.

Constructores	
public	**Thread** () Crea un nuevo objeto Thread. Este constructor tiene el mismo efecto que Thread (null, null, gname), donde ***gname*** es un nombre generado automáticamente y que tiene la forma "Thread-"+n, donde n es un entero asignado consecutivamente.
public	**Thread** (String name) Crea un nuevo objeto Thread, asignándole el nombre ***name***.
public	**Thread** (Runnable target) Crea un nuevo objeto Thread. ***target*** es el objeto que contiene el método run() que será invocado al lanzar el hilo con start().
public	**Thread** (Runnable target, String name) Crea un nuevo objeto Thread, asignándole el nombre ***name***. ***target*** es el objeto que contiene el método run() que será invocado al lanzar el hilo con start().

Métodos	
public static Thread	**currentThread** () Retorna una referencia al hilo que se está ejecutando actualmente.
public static void	**dumpStack** () Imprime una traza del hilo actual. Usado sólo con propósitos de depuración..
public String	**getName** () Retorna el nombre del hilo.
int	**getPriority** () Retorna la prioridad del hilo.
public final boolean	**isAlive** () Chequea si el hilo está vivo. Un hilo está vivo si ha sido lanzado con start y no ha muerto todavía.
public final void	**isDaemon** () Devuelve verdadero si el hilo es daemon.
public final void	**join** () throws InterruptedException Espera a que este hilo muera.
public final void	**join** (long millis) throws InterruptedException Espera como mucho ***millis*** milisegundos para que este hilo muera.
public final void	**join** (long millis, int nanos) throws InterruptedException Permite afinar con los nanosegundos ***nanos*** el tiempo a esperar.
public void	**run** () Si este hilo fue construido usando un objeto que implementaba Runnable, entonces el método run de ese

	objeto es llamado. En cualquier otro caso este método no hace nada y retorna.
public final void	**setDaemon** (boolean on) Marca este hilo como daemon si el parámetro ***on*** es verdadero o como hilo de usuario si es falso. El método debe ser llamado antes de que el hilo sea lanzado.
public final void	**setName** (String name) Cambia el nombre del hilo por ***name***.
public final void	**setPriority** (int newPriority) Asigna la prioridad ***newPriority*** a este hilo.
public static void	**sleep** (long millis) throws InterruptedException Hace que el hilo que se está ejecutando actualmente cese su ejecución por los milisegundos especificados en ***millis***. Pasa al estado dormido. El hilo no pierde la propiedad de ningún cerrojo que tuviera adquirido con *synchronized*.
public static void	**sleep** (long millis, int nanos) throws InterruptedException Permite afinar con los nanosegundos ***nanos*** el tiempo a estar dormido.
public void	**start** () Hace que este hilo comience su ejecución. La MVJ llamará al método run de este hilo.
public String	**toString** () Devuelve una representación en forma de cadena de este hilo, incluyendo su nombre, su prioridad y su grupo.
public static void	**yield** () Hace que el hilo que se está ejecutando actualmente pase al estado listo, permitiendo a otro hilo ganar el procesador.

2.4 Resumen

En este capítulo hemos visto los conceptos fundamentales sobre procesos e hilos que necesitamos desde el punto de vista de la programación concurrente. Con respecto a los procesos, hemos visto su ciclo de vida y su disposición en memoria, adentrándonos en el modelo de procesos de Pascal-FC y la gestión y planificación de procesos que hace. Con respecto a los hilos, hemos visto los diferentes estándares que se pueden encontrar (Win32, OS/2 y posix), dos posibles implementaciones (a nivel de usuario o a nivel de núcleo) y cómo es la planificación de hilos en un SO moderno como Solaris. Posteriormente nos hemos adentrado en el mundo de los hilos en Java. Hemos visto la diferencia entre hilo y objeto, diferentes formas de crear hilos, el ciclo de vida de un hilo y su planificación y prioridades. Finalmente se han mostrado los métodos más usados e importantes de la clase Thread de Java.

Con todo lo visto en este capítulo y en el anterior ya estamos en disposición de analizar en mayor profundidad los problemas inherentes a la programación concurrente y sus posibles soluciones utilizando distintas primitivas de sincronización.