Hilos en java

Ing. Patricia Del Valle Morales

Introducción

➢ Procesos

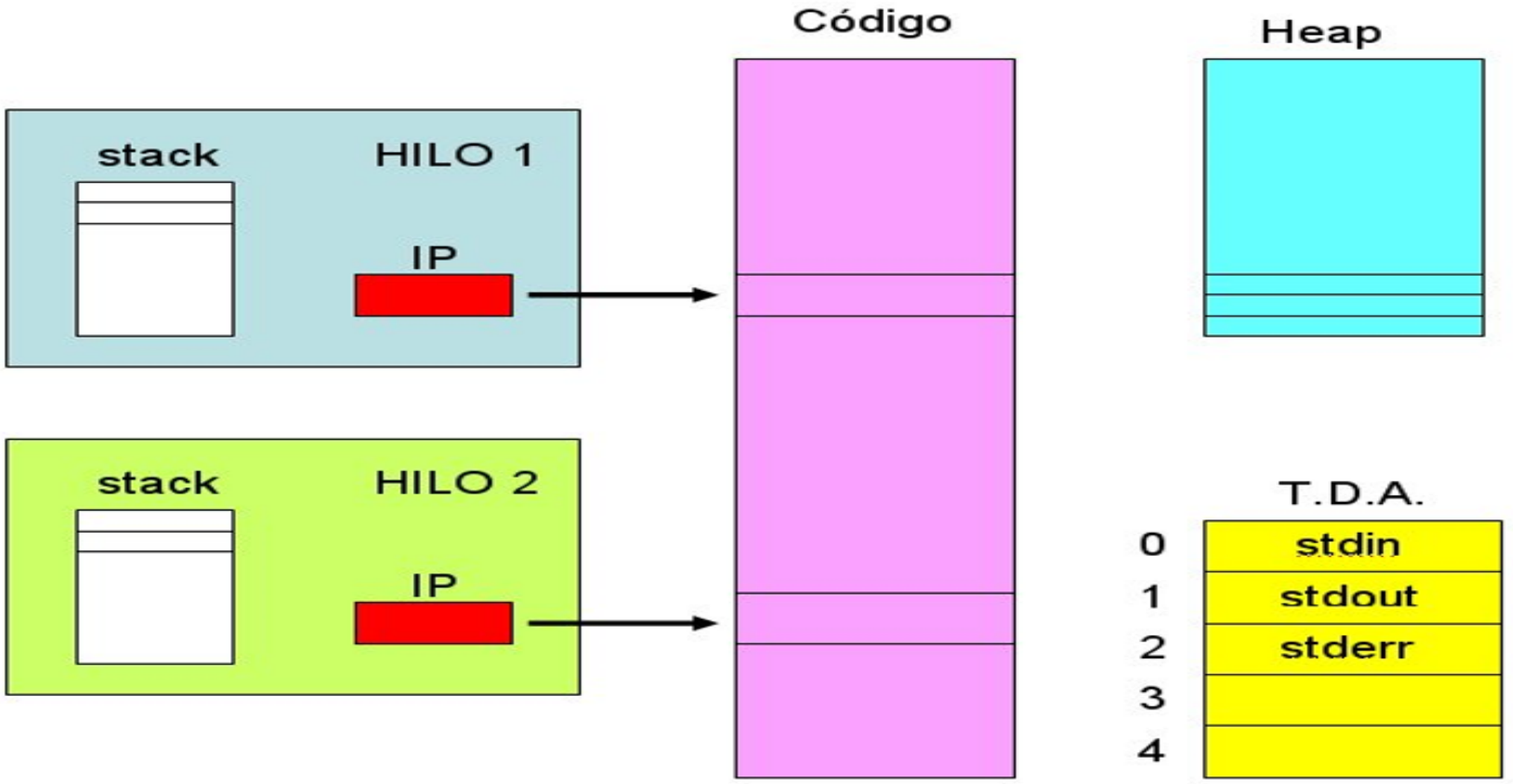
Los **procesos** creados con la llamada *fork* son denominados como ***procesos pesados***; estos procesos son programas completos e independientes con sus propias **variables**, **stack**, y **memoria reservada**, lo único que se comparte con el proceso original es el **código**.

Cuando creamos un nuevo proceso con fork, creamos una nueva secuencia que se ejecuta concurrentemente con el proceso padre, pero no comparten las zonas de datos y la comunicación entre ellos es muy limitada. Por ello aparecen las pipes y los otros mecanismos de comunicación entre procesos.

➢ Hilos

Los **hilos** son otra forma de crear la posibilidad de concurrencia de actividades; sin embargo, la gran diferencia es que los hilos **comparten** el **código** y el acceso a algunos **datos** en forma similar a como un objeto tiene acceso a otros objetos. En Java un hilo es un objeto con capacidad de correr en forma concurrente el método **run**(). En cierta manera es como tener dos "program counters" para un mismo código

Esquema general de un hilo:



Estructura de un hilo

Un hilo (proceso ligero) es una unidad básica de utilización de la CPU, y consiste en un contador de programa, un juego de registros y un espacio de pila.

Los hilos dentro de una misma aplicación **comparten**:

- La sección de código.
- La sección de datos.
- Los recursos del SO (archivos abiertos y señales).

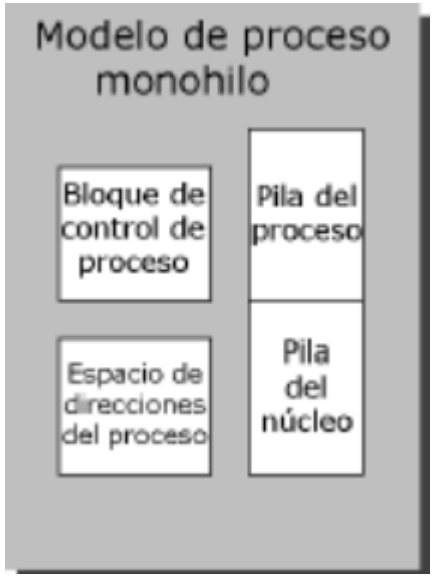
Un proceso tradicional o pesado es igual a una tarea con un solo hilo.

Los hilos permiten la ejecución concurrente de varias secuencias de instrucciones asociadas a diferentes funciones dentro de un mismo proceso, **compartiendo** un mismo espacio de **direcciones** y las mismas **estructuras de datos del núcleo**.

Proceso Monohilo vs Multihilos

En un SO con procesos **monohilo** , en el que no existe el concepto de hilo, la representación de un proceso incluye:

- ✓ su PCB,
- ✓ un espacio de direcciones del proceso,
- ✓ una pila de proceso y una pila núcleo.

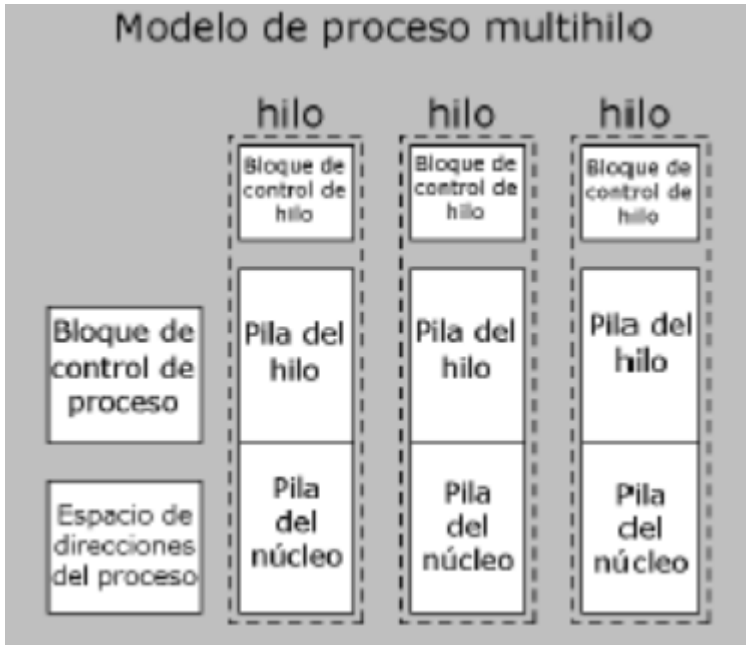


En un SO con procesos **multihilo**, sólo hay:

- ✓ un PCB y un espacio de direcciones asociados al proceso,

Sin embargo, ahora hay:

- ✓ pilas separadas para cada hilo y
- ✓ bloques de control para cada hilo.



Recursos compartidos y no compartidos

Los hilos permiten la ejecución concurrente de varias secuencias de instrucciones asociadas a diferentes funciones dentro de un mismo proceso, compartiendo un mismo espacio de direcciones y las mismas estructuras de datos del núcleo.

Recursos compartidos entre los hilos:

- Código (instrucciones).
- Variables globales.
- Ficheros y dispositivos abiertos.

Recursos no compartidos entre los hilos:

- Contador del programa (cada hilo puede ejecutar una sección distinta de código).
- Registros de CPU.
- Pila para las variables locales de los procedimientos a las que se invoca después de crear un hilo.
- Estado: distintos hilos pueden estar en ejecución, listos o bloqueados esperando un evento.

Concurrencia

Ing. Patricia Del Valle Morales

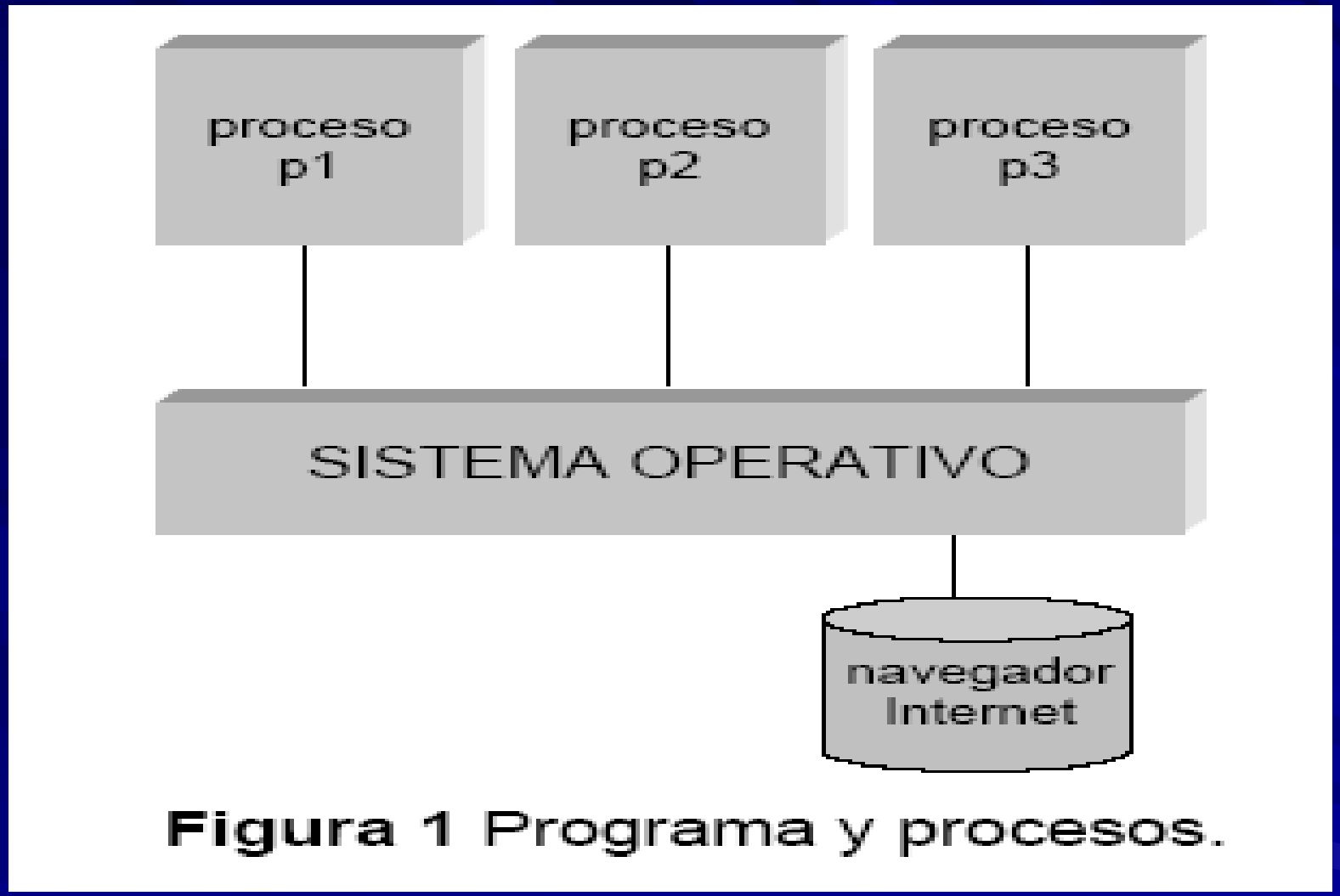
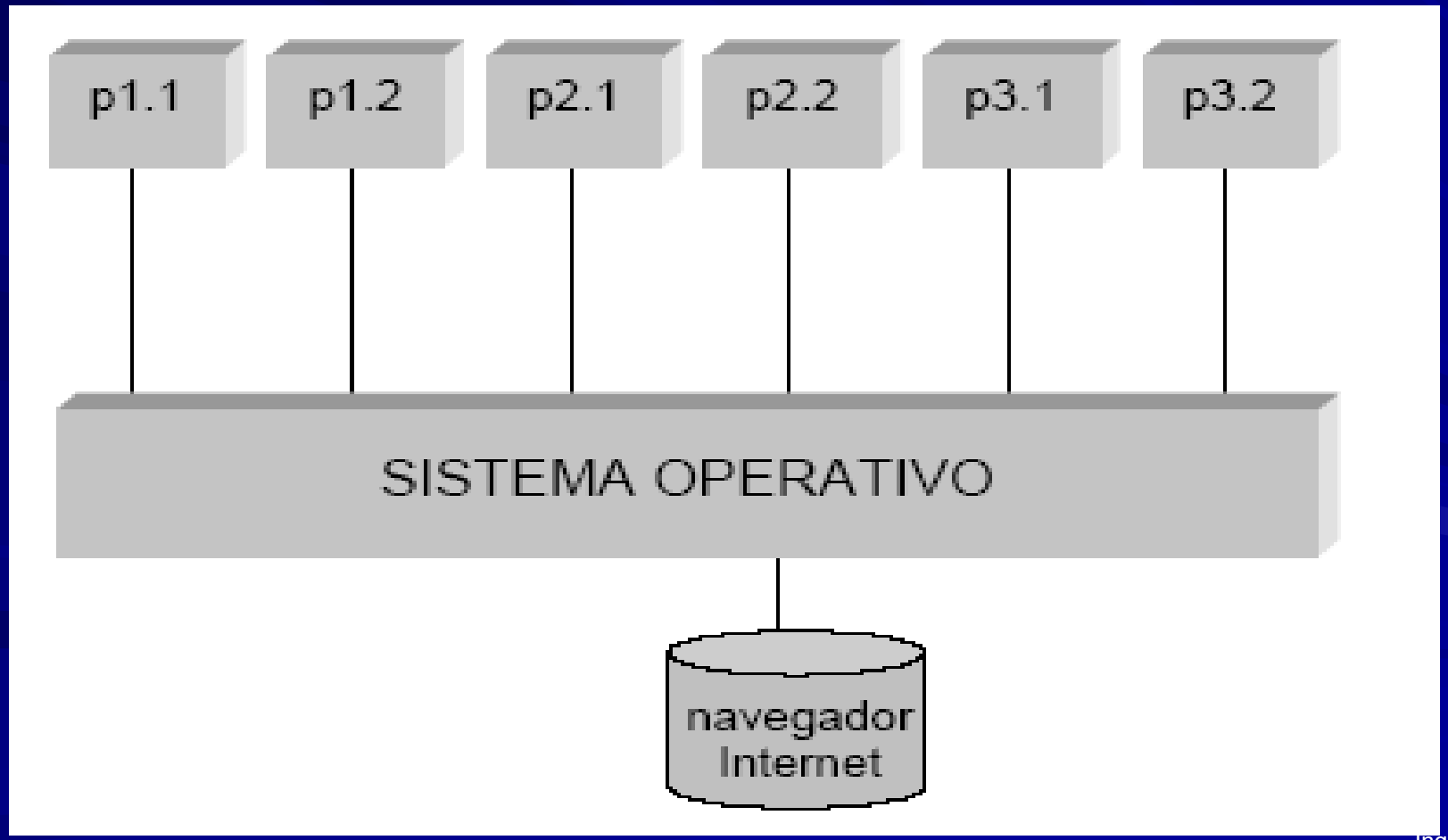


Figura 1 Programa y procesos.

Primer nivel de concurrencia, donde existen 3 procesos independientes ejecutándose al mismo tiempo sobre el Sistema Operativo.

No necesariamente un proceso es todo el programa en ejecución, sino que, al ejecutarse, puede dar lugar a más de un proceso, cada uno de ellos ejecutando una parte del programa.



¿Qué es la programación concurrente?

Son los mecanismos de **comunicación** y **sincronización** entre procesos para llevar a cabo las tareas de colaboración y competencia por los recursos del sistema.

Beneficios de la programación concurrente

1. Velocidad de ejecución
2. Solución de problemas de naturaleza concurrente.

a) Tecnologías web

- ✓ *Servidores web* que son capaces de atender concurrentemente múltiples conexiones de usuarios.
- ✓ *Programas de chat* que permiten mantener la conversación de varios usuarios.
- ✓ *Servidores de correo* que permiten que múltiples usuarios puedan mandar y recibir mensajes al mismo tiempo.
- ✓ *Los Navegadores* que permiten que un usuario pueda estar haciendo una descarga mientras navega por otras páginas, o se ejecuta un applet de Java, etc.

b) Sistemas de control

✓La recolección de datos puede hacerse ser desde diversas entidades físicas como por ejemplo edificios o estancias dentro de edificios.

✓Tanto la captura de datos, el análisis y posterior actuación son candidatos a ser procesos distintos y de naturaleza concurrente.

¿Qué pasaría si el problema no plantea como una solución concurrente?

c) Aplicaciones basadas en interfaces de usuarios

- La concurrencia en este tipo de aplicaciones va a permitir que el usuario pueda interactuar con la aplicación aunque ésta esté realizando alguna tarea que consume mucho tiempo de procesador.
- Un proceso controla la interfaz mientras otro hace la tarea que requiere un uso intensivo de la CPU.
- Esto facilitará que tareas largas puedan ser abortadas a mitad de ejecución.

d) Ejemplos propuestos por los alumnos

¿Qué se puede ejecutar concurrentemente?

Condiciones de Bernstein

Para poder determinar si dos conjuntos de instrucciones se pueden ejecutar de forma concurrente, se definen en primer lugar los siguientes conjuntos:

L (Sk) = {a1, a2, ... , an}, como el **conjunto de lectura** del conjunto de instrucciones Sk y que está formado por todas las variables cuyos valores son referenciados (se leen) durante la ejecución de las instrucciones en Sk.

E (Sk) = {b1, b2, ... , bm}, como el **conjunto de escritura** del conjunto de instrucciones Sk y que está formado por todas las variables cuyos valores son actualizados (se escriben) durante la ejecución de las instrucciones en Sk.

Para que dos conjuntos de instrucciones Si y Sj se puedan ejecutar concurrentemente, se tiene que cumplir que:

1. $L(Si) \cap E(Sj) = \emptyset$
2. $E(Si) \cap L(Sj) = \emptyset$
3. $E(Si) \cap E(Sj) = \emptyset$

Ejemplo:

S1 → a := x+y;
S2 → b := z-1;
S3 → c := a-b;
S4 → w := c+1;

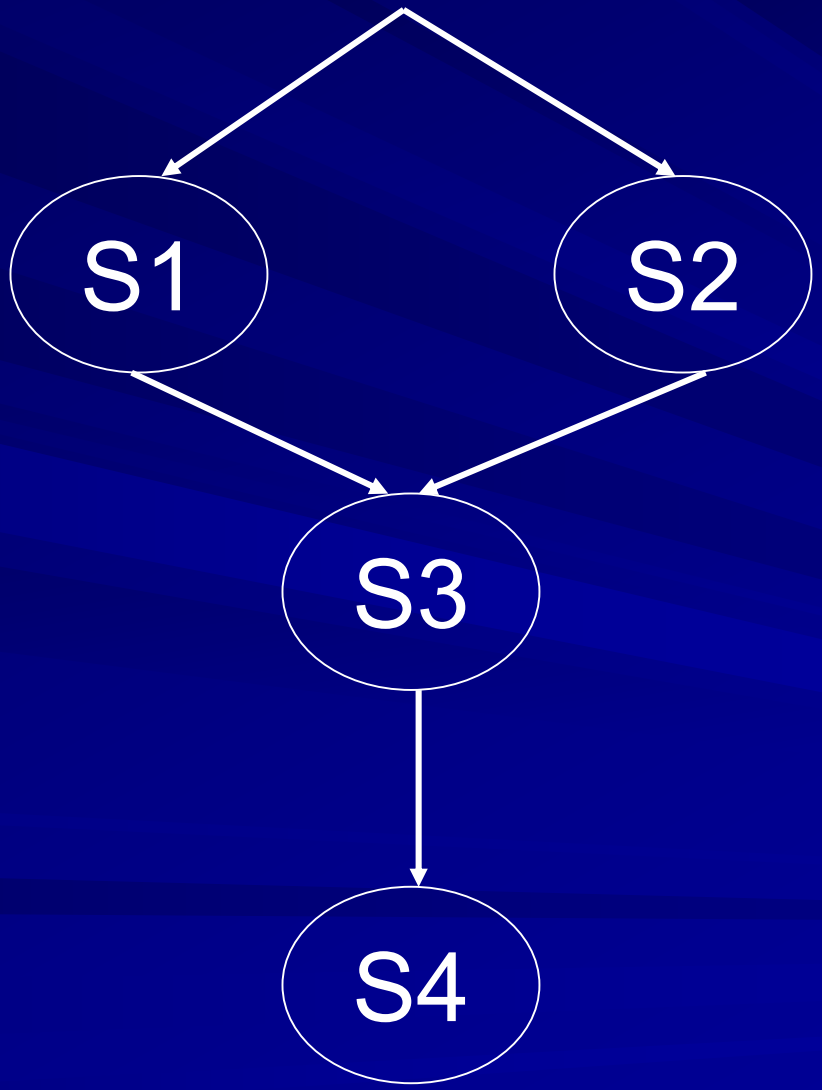
- Calculamos los conjuntos de lectura y escritura:

L (S1) = {x, y} E (S1) = {a}
L (S2) = {z} E (S2) = {b}
L (S3) = {a, b} E (S3) = {c}
L (S4) = {c} E (S4) = {w}

- Aplicamos las condiciones de Bernstein a cada par de sentencias:

	S_1	S_2	S_3	S_4
S_1	--	Sí	No	Sí
S_2	--	--	No	Sí
S_3	--	--	--	No
S_4	--	--	--	--

Grafos de precedencia



Problema

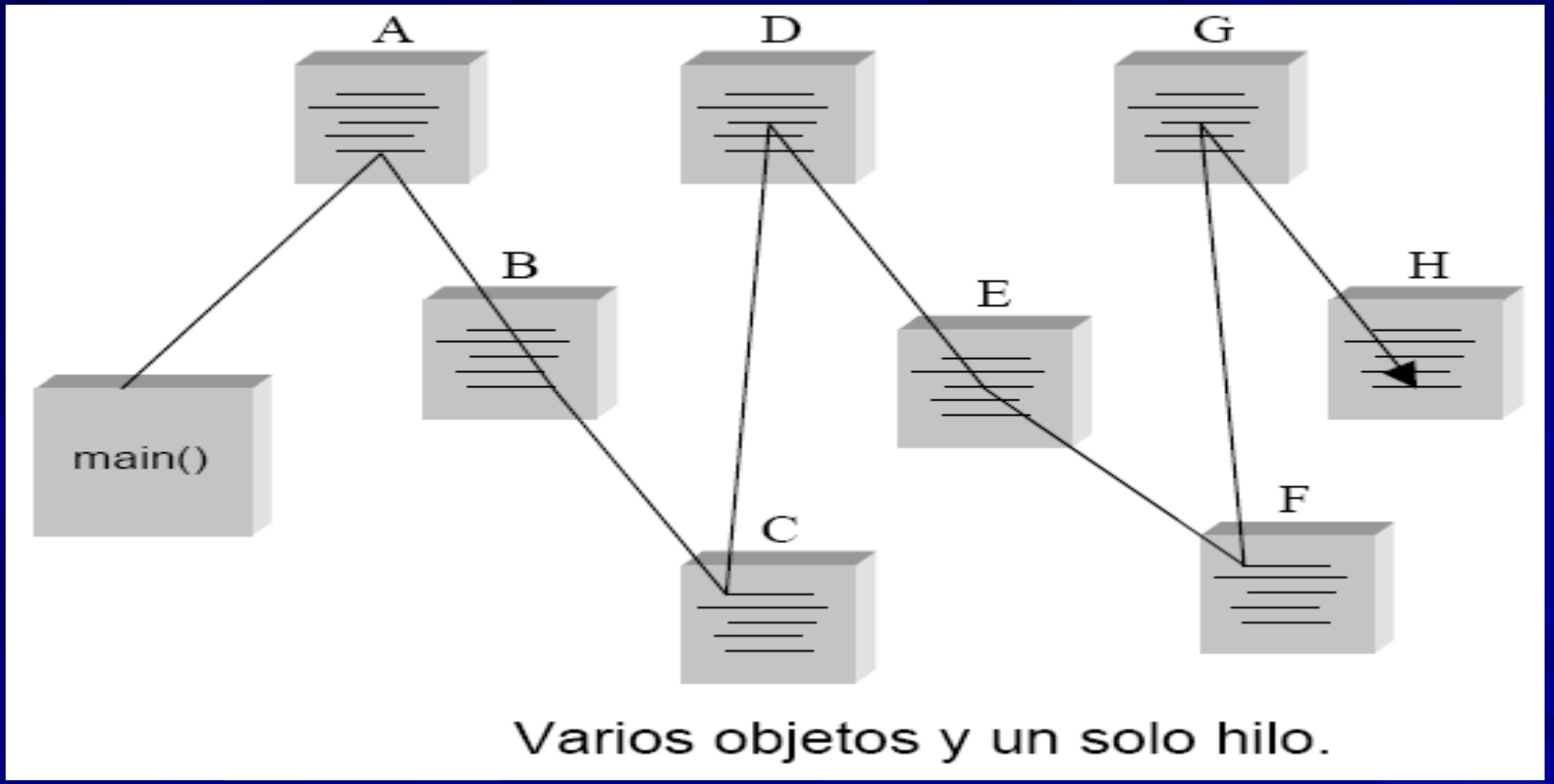
- Usando las condiciones de Bernstein, construir el grafo de precedencia del siguiente trozo de código.

```
S1: cuad := x*x;
S2: m1 := a*cuad;
S3: m2 := b*x;
S4: z := m1 + m2;
S5: y := z + c;
```

Hilos en java

Ing. Patricia Del Valle Morales

Hilos y objetos



- Sólo hay un hilo que va recorriendo los objetos según se van produciendo las llamadas entre métodos de los objetos.

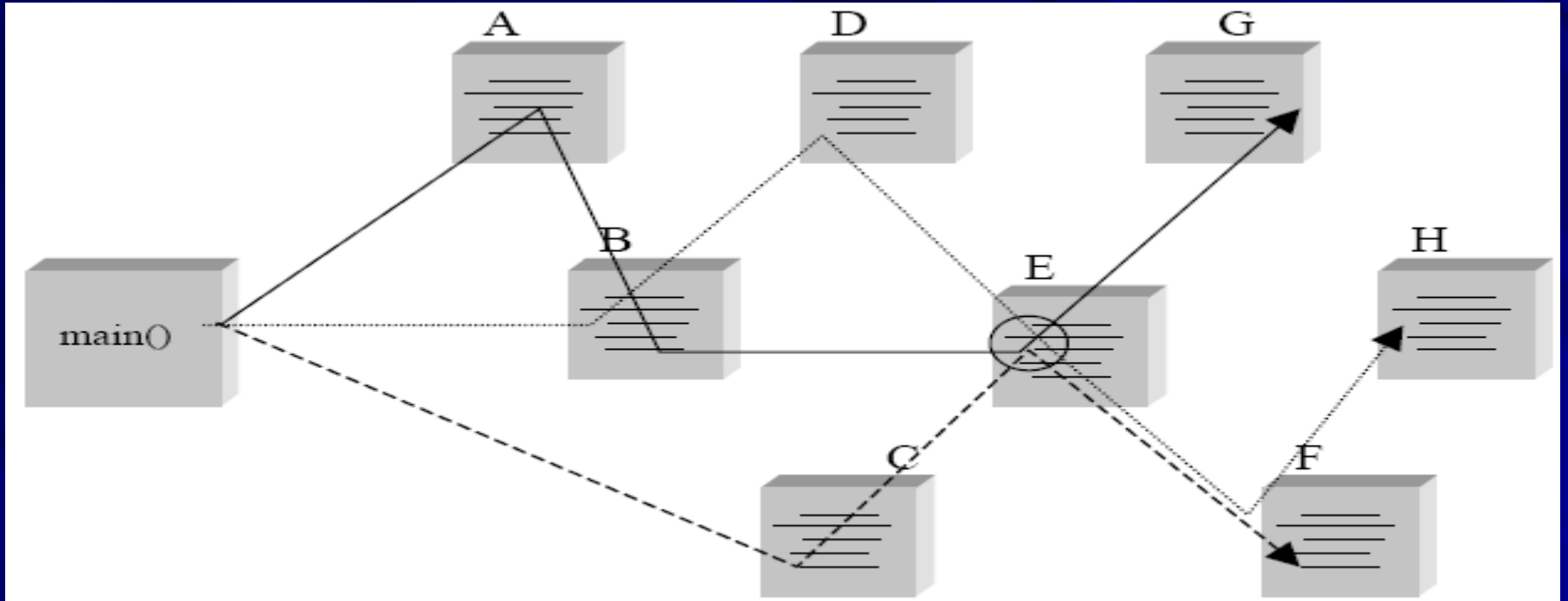


Figura 9 Tres hilos "atravesando" varios objetos.

- 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.**

Creación de hilos

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

a) Heredando de la clase Thread

b) Implementando la interfaz Runnable

Clase thread (atributos)

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**

Clase thread (constructores)

	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().

Clase thread (Métodos)

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.

Clase thread (Métodos)

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

Clase thread (Métodos)

	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.

Creación de threds

a) Clase Thread

1. Se crea una subclase de la clase Thread
2. Se define un método run() para ella
3. Se crea una instancia de ella
4. Y se ejecuta el método start().

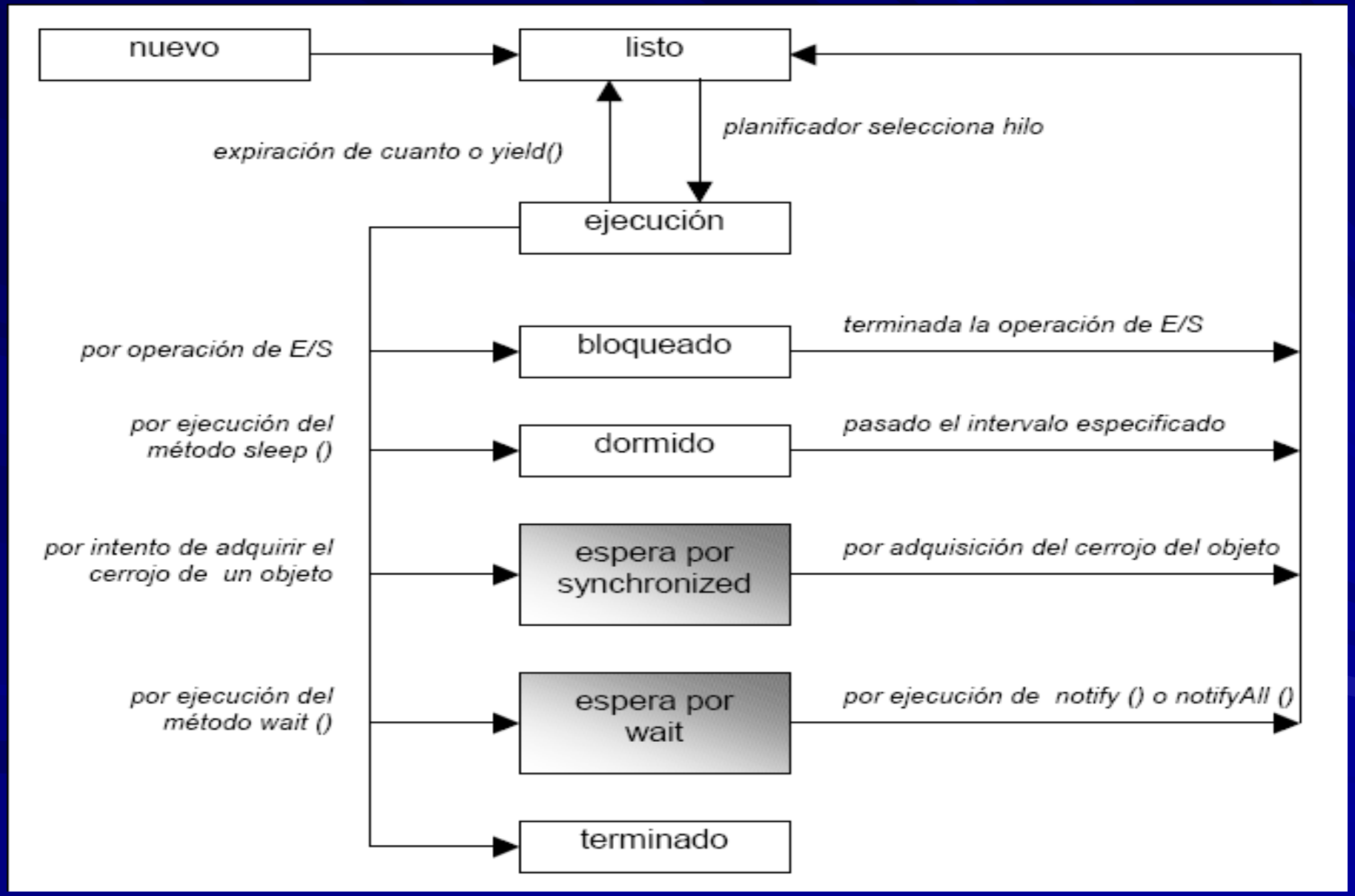
b) Interfaz Runnable

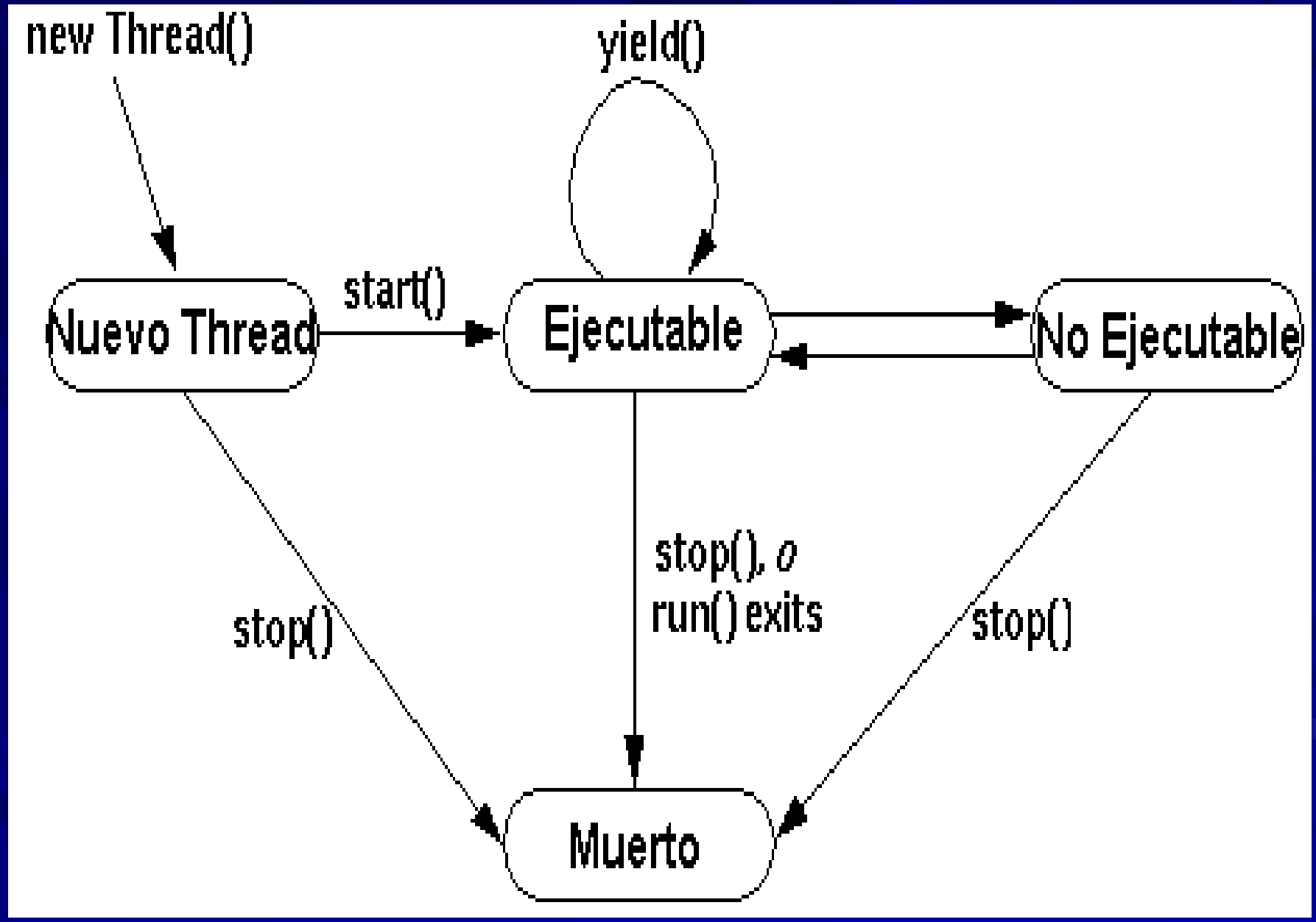
1. Se crea una clase que implementa la interfaz Runnable
2. Se define un método run() en ella
3. Se crea una instancia con esta interfaz Runnable como argumento
4. Y se ejecuta el método start().

Creación de un hilo con Thread

```
class MyThread extends Thread {
    …….

            public void run() {
            ……..
            //trabajo del hilo
            ……
            }
}

……

Thread t = new MyThread();
t.start();
```

Estados de un hilo en Java




new Thread()
yield()
start()
Nuevo Thread
Ejecutable
No Ejecutable
stop()
stop(), o
run() exits
stop()
Muerto

- *Nuevo:*

  Es el estado en el que se encuentra un hilo cuando se crea con el operador new.

- *Listo:*

  Pasa a este estado al llamar al método *start().*

- *Ejecución:*

  Pasa al estado de ejecución cuando el planificador le asigna el procesador

Planificación y prioridades

- 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

- 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.

- Se debe asumir que los hilos pueden intercalarse en cualquier punto en cualquier momento.

- 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.

- Los hilos de prioridad inferior se ejecutarán cuando estén bloqueados los de prioridad superior.

- Las prioridades se pueden cambiar utilizando el método setPriority (nuevaPrioridad).
- 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.
- El método stop() detiene la ejecución del hilo.