

Programación de Sistemas y Concurrency

Tema 4: Soporte a la Concurrency en Lenguajes y Sistemas operativos

Grado en Ingeniería Informática

Grado en Ingeniería del Software

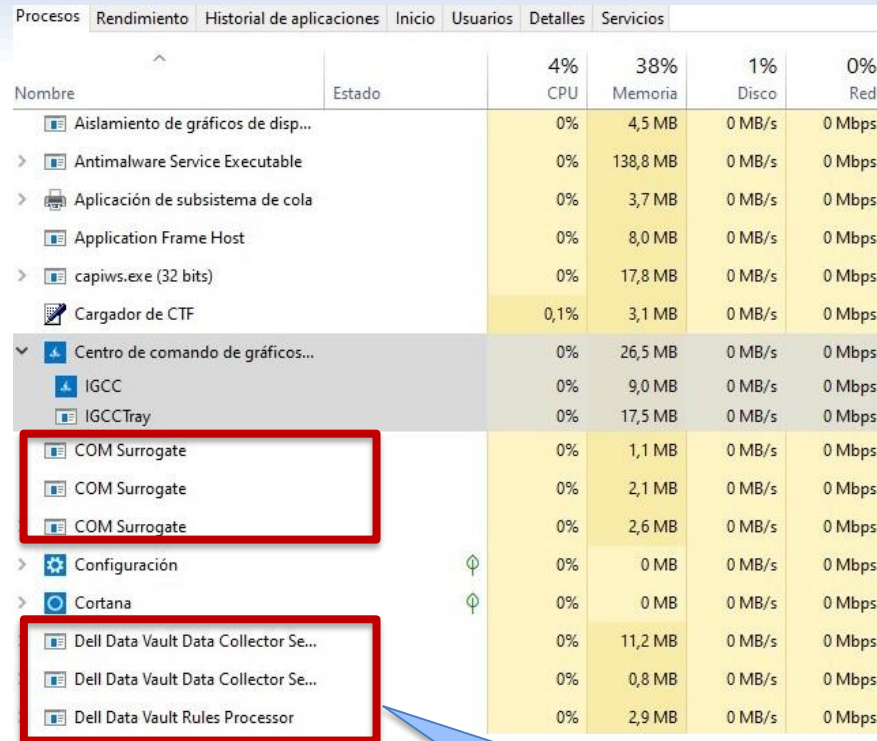
Grado en Ingeniería de Computadores

Contenido

- Procesos
 - Concepto
 - Características
- Modelo de hebras en Java

Concepto de proceso

- Un **proceso** es una instancia de un programa en ejecución
- Un **proceso** consta de:
 - Código del programa
 - Datos del programa
 - Pila de ejecución
 - recursos del SO (CPU, memoria, etc.)
- Un **proceso ligero o hebra**
 - Es la unidad básica de utilización de la CPU
 - Tiene asignado un procesador lógico
 - Dentro de un proceso podemos tener varias hebras en ejecución

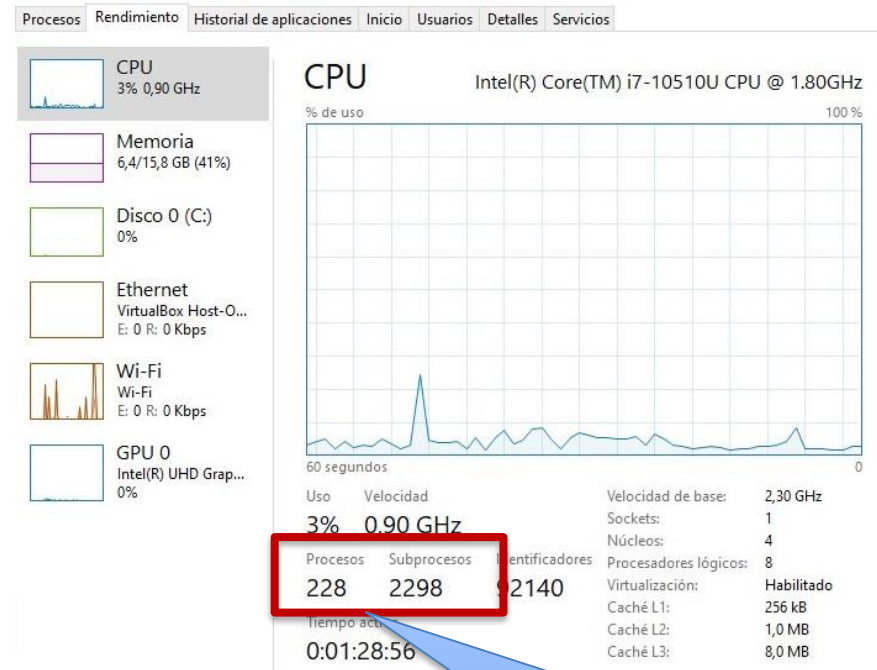


Procesos	Rendimiento	Historial de aplicaciones	Inicio	Usuarios	Detalles	Servicios
Nombre	Estado	4% CPU	38% Memoria	1% Disco	0% Red	
Aislamiento de gráficos de disp...		0%	4,5 MB	0 MB/s	0 Mbps	
> Antimalware Service Executable		0%	138,8 MB	0 MB/s	0 Mbps	
> Aplicación de subsistema de cola		0%	3,7 MB	0 MB/s	0 Mbps	
Application Frame Host		0%	8,0 MB	0 MB/s	0 Mbps	
> capiws.exe (32 bits)		0%	17,8 MB	0 MB/s	0 Mbps	
Cargador de CTF		0,1%	3,1 MB	0 MB/s	0 Mbps	
▼ Centro de comando de gráficos...		0%	26,5 MB	0 MB/s	0 Mbps	
IGCC		0%	9,0 MB	0 MB/s	0 Mbps	
IGCCTray		0%	17,5 MB	0 MB/s	0 Mbps	
COM Surrogate		0%	1,1 MB	0 MB/s	0 Mbps	
COM Surrogate		0%	2,1 MB	0 MB/s	0 Mbps	
COM Surrogate		0%	2,6 MB	0 MB/s	0 Mbps	
> Configuración		0%	0 MB	0 MB/s	0 Mbps	
> Cortana		0%	0 MB	0 MB/s	0 Mbps	
Dell Data Vault Data Collector Se...		0%	11,2 MB	0 MB/s	0 Mbps	
Dell Data Vault Data Collector Se...		0%	0,8 MB	0 MB/s	0 Mbps	
Dell Data Vault Rules Processor		0%	2,9 MB	0 MB/s	0 Mbps	

Pueden existir varios procesos ejecutando el mismo programa

Concepto de proceso

- Un **proceso** es una instancia de un programa en ejecución
- Un **proceso** consta de:
 - Código del programa
 - Datos del programa
 - Pila de ejecución
 - recursos del SO (CPU, memoria, etc.)
- Un **proceso ligero o hebra**
 - Es la unidad básica de utilización de la CPU
 - Tiene asignado un procesador lógico
 - Dentro de un proceso podemos tener varias hebras en ejecución



Un proceso puede tener varias hebras (o subprocesos)

Concepto de proceso

- P, Q y R son procesos
 - $P \parallel Q \parallel R$ significa que la ejecución de P, Q y R puede solaparse en el tiempo
- Cada proceso P, Q, R
 - representa una **hebra de ejecución** secuencial,
 - tiene asociado un procesador lógico que le permite evolucionar en su código
- El orden de la composición $P \parallel Q$ no es relevante:
 - \parallel es conmutativo

Características de los procesos

- Estructura estática vs dinámica
- Jerarquía entre procesos
- Inicialización
- Terminación
- Representación de los procesos en los lenguajes de programación

Estructura estática vs dinámica

- Los lenguajes de programación pueden soportar una estructura de procesos
 - **Estática:** el número de procesos en ejecución está determinado en compilación
 - **Dinámica:** los procesos pueden crearse dinámicamente, por lo que el número total de procesos sólo puede conocerse en ejecución
 - Aunque la estructura sea dinámica, el número máximo de procesos está limitado por la de memoria de la máquina.

Jerarquías entre procesos

- Un proceso puede crear otros procesos (podemos anidar unos procesos dentro de otros)
- Existe una **relación padre/hijo** entre el proceso desde el que se crea a otro proceso, y el nuevo proceso creado
- La consecuencia más importante de esta relación es que un proceso padre no puede terminar su ejecución hasta que no terminen todos sus hijos
 - El padre puede contener recursos que pueden estar siendo utilizados por sus hijos.
- Siempre existe un nivel de anidamiento mínimo: el programa principal es padre de todos los procesos del sistema creados.

Inicialización

- Para darle valores iniciales a un proceso pueden utilizarse dos métodos:
 - **Paso de parámetros:** como en los procedimientos o métodos
 - **Paso de mensajes:** el proceso padre, cuando crea un subproceso le envía un mensaje para inicializar sus variables
- Distintos valores iniciales pueden dar lugar a distintos comportamientos del proceso, es natural introducir el concepto de *tipo de proceso*.
 - Podemos definir un patrón de comportamiento común y luego instanciarlo varias veces para obtener distintos procesos que aunque comparten el código, pueden comportarse de forma distinta dependiendo de sus valores iniciales

Terminación

- Un proceso puede acabar su ejecución por distintas causas:
 - cuando se acaba su código (terminación con éxito)
 - cuando tiene un error durante su ejecución (terminación con fallo)
 - por la ejecución de una instrucción de autodestrucción (suicidio)
 - cuando otro proceso aborta su ejecución
- Un proceso puede no terminar
 - siendo éste su comportamiento esperado
 - como consecuencia de un error en su diseño

Creación y representación

- La creación de un proceso (el momento en que empieza a ejecutarse) puede ser
 - **Explícita:** se llama al proceso, como si fuera un procedimiento
 - **Implícita:** el proceso empieza a ejecutarse en cuanto lo hace el módulo o estructura en la que se encuentra anidado.
- La representación de los procesos puede ser
 - **Explícita:** el lenguaje proporciona un constructor específico para definir a los procesos
 - **Implícita:** Cualquier constructor del lenguaje puede ser considerado como un proceso (procedimiento, función, instrucción simple).

Creación y representación

- Creación explícita, representación implícita: C + OpenMP
- Creación implícita, representación explícita: Ada

```
#include <omp.h> //biblioteca OpenPM
#include <stdio.h>
int main(int argc, char *argv[]){
    /*código secuencial*/
    #pragma omp parallel{
        printf("Esta region se ejecuta de forma paralela");
    }
    /* código secuencial */
    return 0;
}
```

```
procedure Proc
process P;
begin ...end;

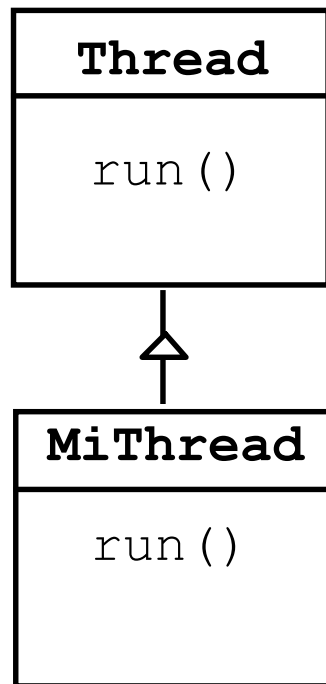
process Q;
begin ...end;
begin
    {Proc, P y Q están ejecutándose}
    ...
end;
```

El modelo de proceso en Java

- Estructura dinámica de procesos.
 - Cualquier proceso puede crear uno nuevo
 - No hay límite en el número de procesos, salvo el impuesto por la memoria de la máquina
- Jerarquía arbitraria
 - Puede existir cualquier nivel de anidamiento entre procesos
- Inicialización mediante paso de parámetros en el constructor
- Terminación
 - Soporta todos los tipos de terminación, aunque el uso de la instrucción abort está desaconsejado
- Creación y representación explícitas
 - Existe un tipo proceso/hebra
 - Para que los procesos/hebras se ejecuten hay que llamarlos explícitamente

El modelo de proceso/hebra en Java

- La clase **Thread** proporciona el soporte para manejar las hebras (que hemos llamado procesos hasta ahora)
- Las hebras pueden crearse y destruirse de forma dinámica (durante la ejecución del programa)

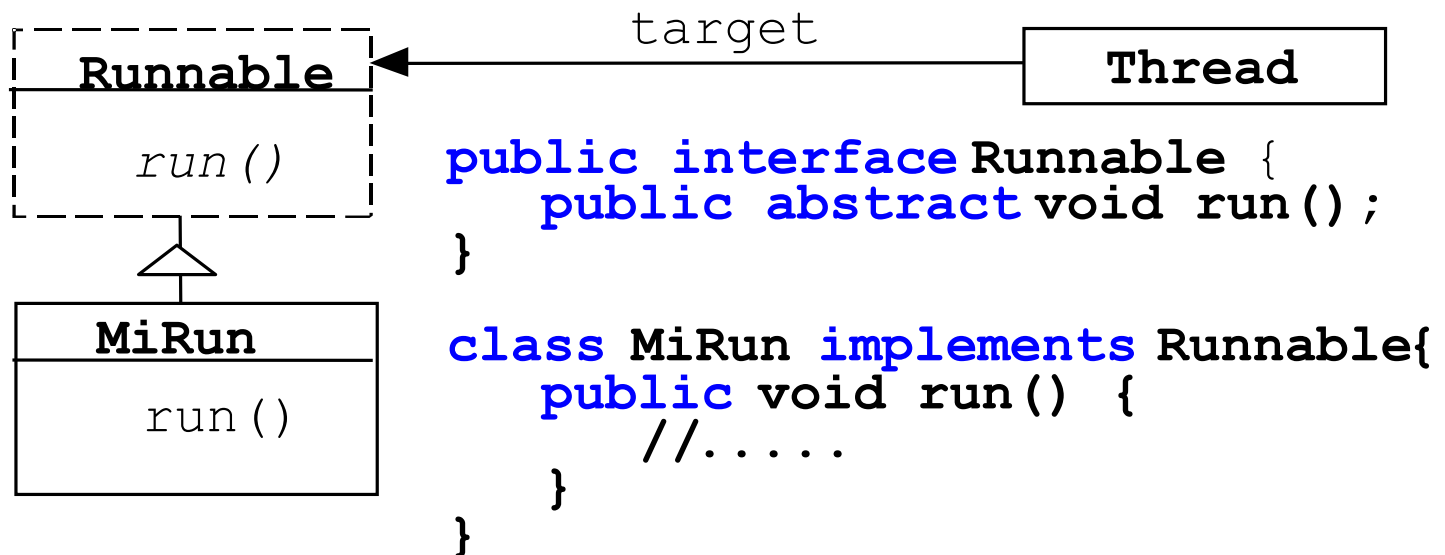


- La clase **Thread** representa una hebra que ejecuta las instrucciones de su método **run()**
- Podemos definir el comportamiento de una hebra mediante herencia y sobrescribiendo el método **run()** en la clase derivada

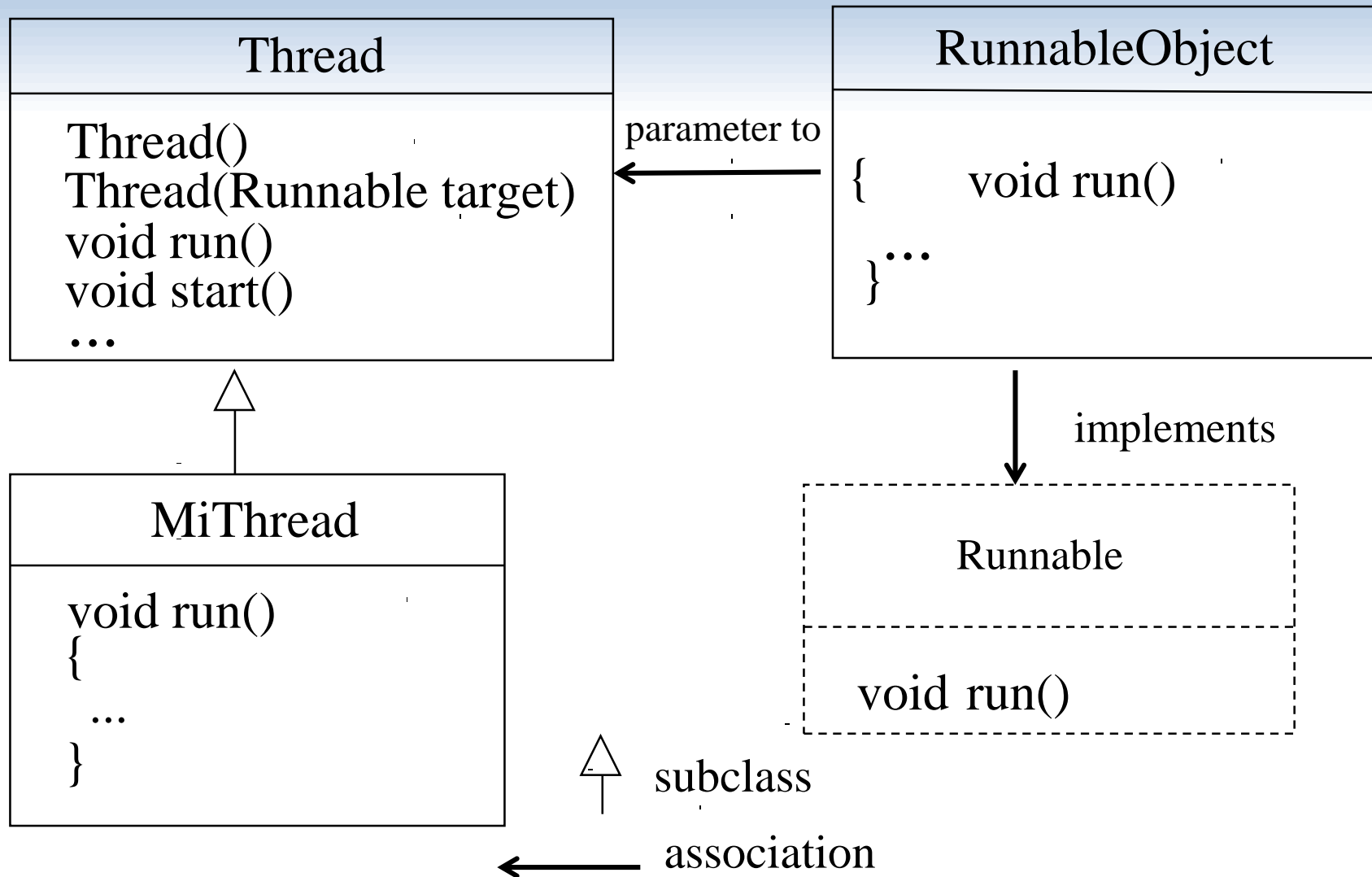
```
class MiThread extends Thread {
    public void run() {
        //.....
    }
}
```

El modelo de proceso/hebra en Java

- Como Java no permite **la herencia múltiple**, si queremos que un objeto **hebra herede de otra superclase, y de la clase Thread**, el lenguaje proporciona **la interfaz Runnable**, que obliga a implementar el método **run()** y que permite que una clase que no deriva de **Thread** sea una hebra.
- Uno de los constructores de la clase Thread permite pasarle como parámetro un objeto Runnable



El modelo de proceso/hebra en Java



La classe Thread

```
public class Thread extends Object
    implements Runnable {
    public Thread();
    public Thread(String name);
    public Thread(Runnable target);
    public Thread(Runnable target,
        String name);

    public void run();
    public void start();
    ...
}
```

La clase Thread

- Hay dos opciones para crear una (hebra) Thread:
 - **Opción 1:** crear una clase que herede (extienda) la clase Thread y sobrescribir el método run() con el comportamiento deseado.
 - **Opción 2:** crear un objeto que implemente la interfaz Runnable, y pasar su referencia a un objeto de la clase Thread a través del constructor.

Un ejemplo simple usando Runnable (Opción2)

- Definimos el comportamiento de la hebra usando una clase Esc que implementa la interfaz Runnable
- Implementamos el método run() de la clase Esc
- Pasamos un objeto de la clase Esc al constructor de Thread

```
public class Esc implements Runnable{  
    private char miLetra;  
    public Esc(char l){  
        miLetra = l;  
    }  
    public void run() {  
        for (int i = 0; i<100; i++){  
            System.out.print(miLetra);  
        }  
    }  
}
```

Todavía no hay
ninguna hebra
en ejecución

```
public static void main(String[] args){  
    Thread a = new Thread(new Esc('A'));  
    Thread b = new Thread(new Esc('B'));  
    Thread c = new Thread(new Esc('C'));  
    a.start();b.start();c.start();  
}
```

- El constructor de Esc recibe una letra
- El método run de Esc muestra por pantalla 100 veces la letra que recibió como parámetro
→ conseguimos comportamientos diferentes

Un ejemplo simple usando Runnable (Opción2)

```
public class Esc implements Runnable{  
    private char miLetra;  
    public Esc(char l){  
        miLetra = l;  
    }  
    public void run() {  
        for (int i = 0; i<100; i++){  
            System.out.print(miLetra);  
        }  
    }  
}
```

La llamada a
start() hace que
se ejecute el
método run()

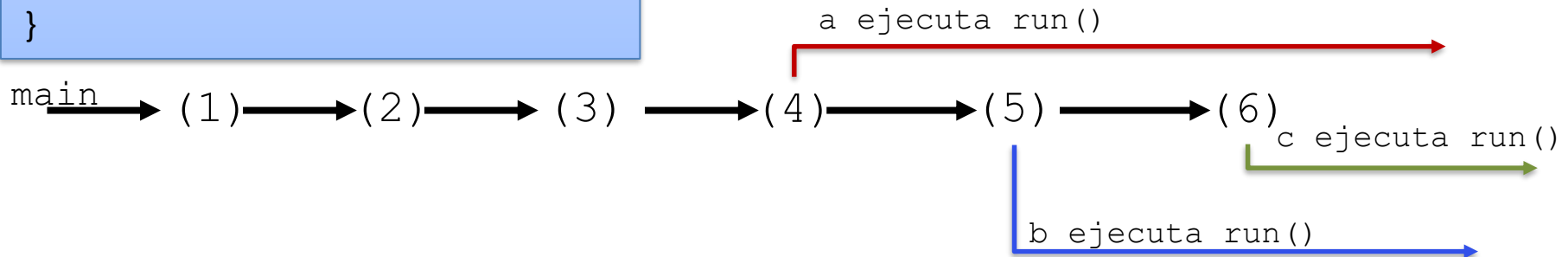
```
public static void main(String[] args){  
    Thread a = new Thread(new Esc('A'));  
    Thread b = new Thread(new Esc('B'));  
    Thread c = new Thread(new Esc('C'));  
    a.start();b.start();c.start();  
}
```

Las hebras a, b y c
comienzan su ejecución
cuando se llama a start()

Un ejemplo simple usando Runnable (Opción2)

```
public class Esc implements Runnable{  
    private char miLetra;  
    public Esc(char l){  
        miLetra = l;  
    }  
    public void run() {  
        for (int i = 0; i<100; i++){  
            System.out.print(miLetra);  
        }  
    }  
}
```

```
public static void main(String[] args){  
    Thread a = new Thread(new Esc('A'));  
    Thread b = new Thread(new Esc('B'));  
    Thread c = new Thread(new Esc('C'));  
    a.start();b.start();c.start();  
}
```

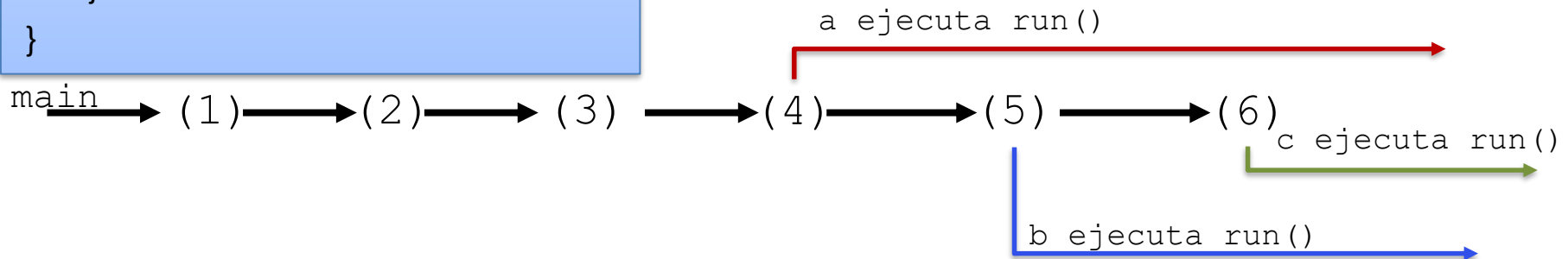


Cuando el método main llama a los métodos start de las tres hebras, las ejecuciones de main, a, b y c prosiguen de forma asíncrona

Un ejemplo simple usando Runnable (Opción2)

```
public class Esc implements Runnable{  
    private char miLetra;  
    public Esc(char l){  
        miLetra = l;  
    }  
    public void run() {  
        for (int i = 0; i<100; i++){  
            System.out.print(miLetra);  
        }  
    }  
}
```

```
public static void main(String[] args){  
    Thread a = new Thread(new Esc('A'));  
    Thread b = new Thread(new Esc('B'));  
    Thread c = new Thread(new Esc('C'));  
    a.start();b.start();c.start();  
}
```



La salida es una combinación de las letras A, B y C. Una posible traza sería:
BCCCCCCCCCAAAAAAAAAAABBBBBBBBBB

Otra alternativa usando Thread (Opción1)

- Definimos el comportamiento de la hebra usando una clase Esc que extiende (hereda) de Thread
- Sobreescribimos el método run de la clase derivada

```
public class Esc extends Thread{  
    private char miLetra;  
    public Esc(char l){  
        miLetra = l;  
    }  
    public void run() {  
        for (int i = 0; i<10; i++){  
            System.out.print(miLetra);  
        }  
    }  
}
```

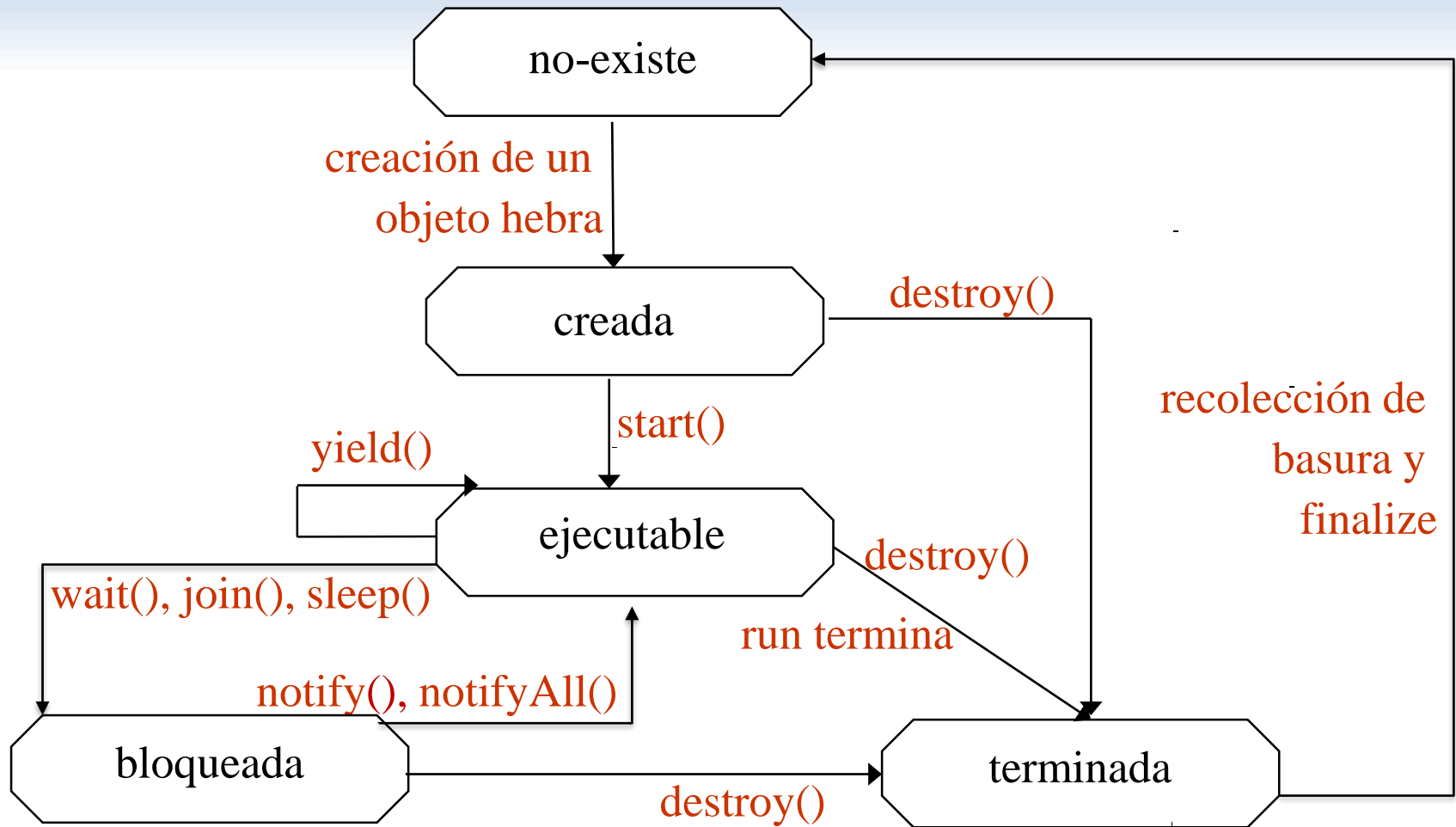
```
public static void main(String[] args){  
    Esc a = new Esc('A');  
    Esc b = new Esc('B');  
    Esc c = new Esc('C');  
    a.start();b.start();c.start();  
}
```

Advertencia

El método **run()** **no debe llamarse directamente** desde la aplicación. El sistema lo llama cuando se llama a start().

Si el método run() es llamado explícitamente, entonces el código se ejecuta de forma secuencial, no concurrentemente.

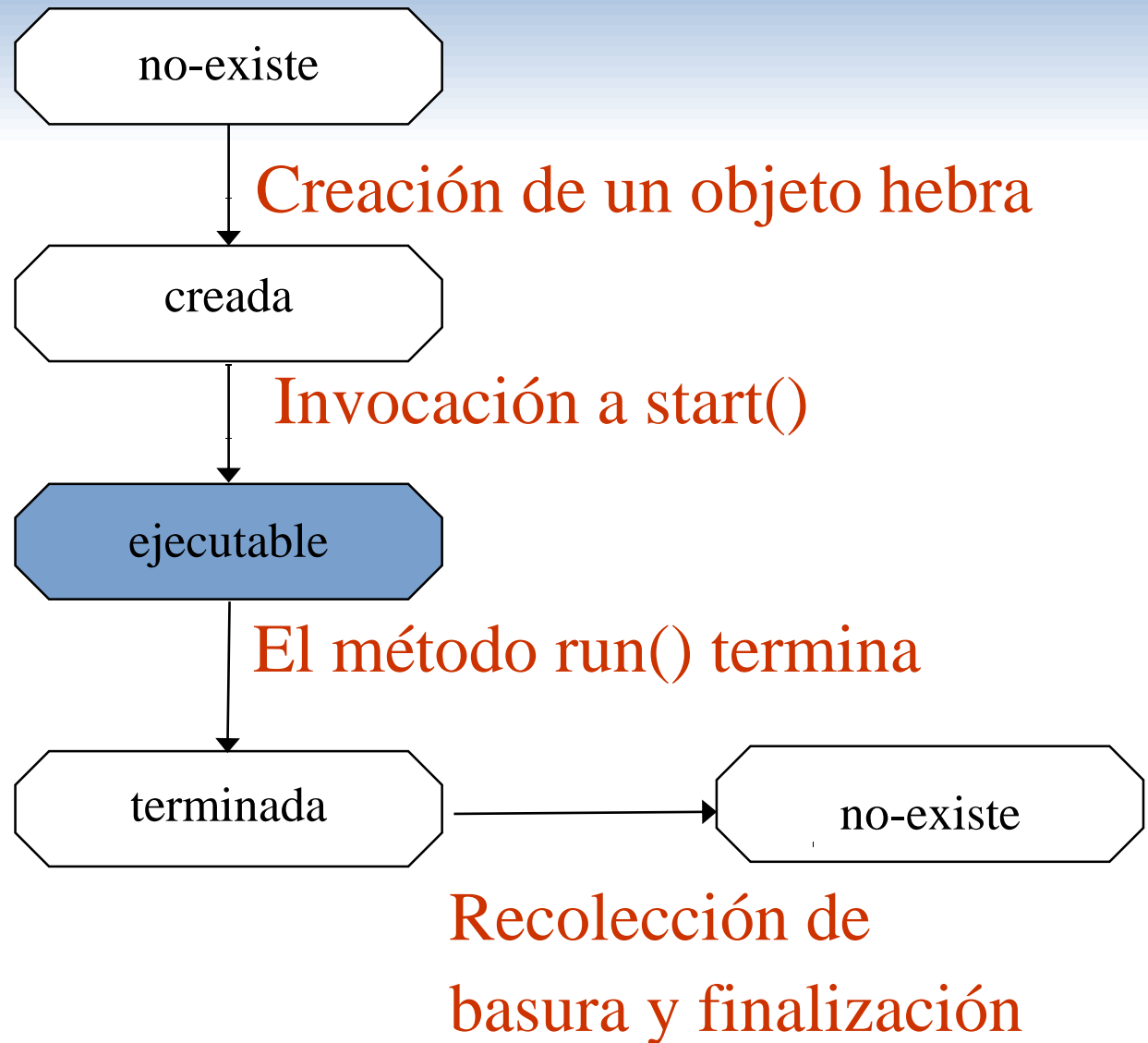
Ciclo de vida de una hebra



Ciclo de vida de una hebra

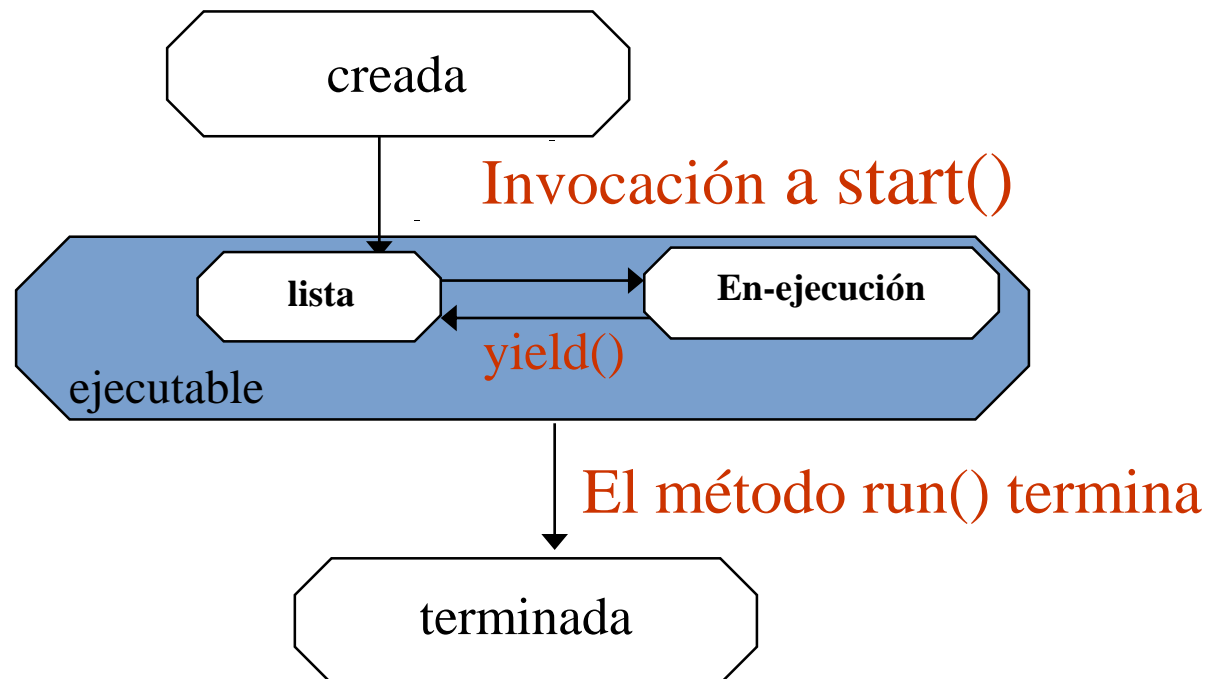
- A continuación veremos los distintos métodos de la clase **Thread**, organizados por los estados en los que se puede encontrar la hebra:
 - **Estado ejecutable**
 - Una hebra en este estado no estará siempre ejecutándose, podrá estar en ejecución o lista para ejecutarse
 - **Estado bloqueado**
 - Hay distintas situaciones en las que la hebra tendrá que bloquearse, por ejemplo, para esperar a que los recursos compartidos a los que quiere acceder estén en el estado apropiado
 - **Estado terminada**
 - La hebra podrá terminar por distintos motivos, de forma correcta cuando termine su funcionamiento o debido a algún error

Ciclo de vida: estado ejecutable



Estado ejecutable

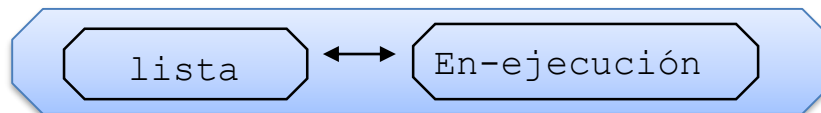
- En un sistema monoprocesador, o en el que hay más hebras que procesadores sólo una, de entre las hebras ejecutables, puede estar ejecutándose, el resto debe esperar.



Estado ejecutable

- Una hebra pasa de lista a en-ejecución cuando el procesador se queda disponible
- Una hebra pasa del estado en-ejecución a lista
 - En un sistema de tiempo compartido (time-sharing) cuando se le ha acabado el tiempo de ejecución.
 - Cuando se hace ejecutable una hebra con más **prioridad**.
 - El cambio de una hebra por otra en el procesador se denomina “cambio de contexto” y es un proceso con una carga computacional.
 - Hay que guardar los valores de todos los registros de la CPU, para que cuando la hebra vuelva al procesador, el cambio de contexto no haya interferido en su ejecución.
 - El cambio de contexto sólo puede realizarse entre la ejecución de dos instrucciones máquina, nunca durante la ejecución de una de ellas.

ejecutable



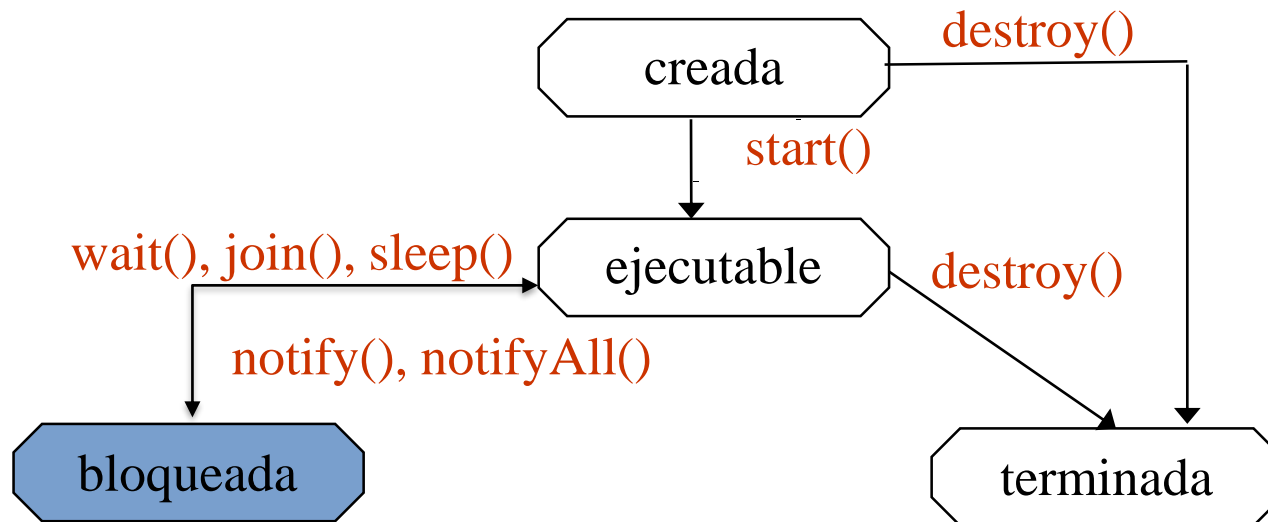
Clase Thread

```
public class Thread extends Object implements Runnable {  
    ...  
    public static void yield();  
}
```

- Usando `yield()` la hebra indica que está dispuesta a ceder su uso actual del procesador.
 - El planificador podría ignorar esta sugerencia. Es decir con solo ejecutar este método no se pasa automáticamente al estado lista
 - Su uso lo veremos en detalle en el próximo tema

Ciclo de vida: estado bloqueado

- Una hebra se encuentra en un estado bloqueado cuando no puede continuar su ejecución hasta que ocurra “algún evento”.
 - Por ejemplo, que un recurso que necesita quede disponible
- Este estado es imprescindible para la **sincronización** de las hebras



Class Thread

```
public class Thread extends Object implements Runnable{
    ...
    public static void sleep(long millis)
        throws InterruptedException;
    public static void sleep(long millis, int nanos)
        throws InterruptedException;

    public final void join()
        throws InterruptedException;
    public final void join(long millis)
        throws InterruptedException;
    public final void join(long millis, int nanos)
        throws InterruptedException;

    public final boolean isAlive();
}
```


El método sleep

- La hebra actual puede parar su ejecución durante un tiempo determinado llamando al método **sleep**.

```
//Clase que genera 20 números aleatorios entre 0 y 99,  
//esperando 4s (aproximadamente) entre número y número  
class DormirHebra {  
  
    public static void main(String args[]) throws InterruptedException{  
        java.util.Random rnd = new java.util.Random();  
        int valor;  
  
        for (int i =0; i< 20; i++){  
            valor = rnd.nextInt(100);  
            System.out.println("Valor: " + valor);  
            Thread.sleep(4000);  
        }  
    }  
}
```

El método join

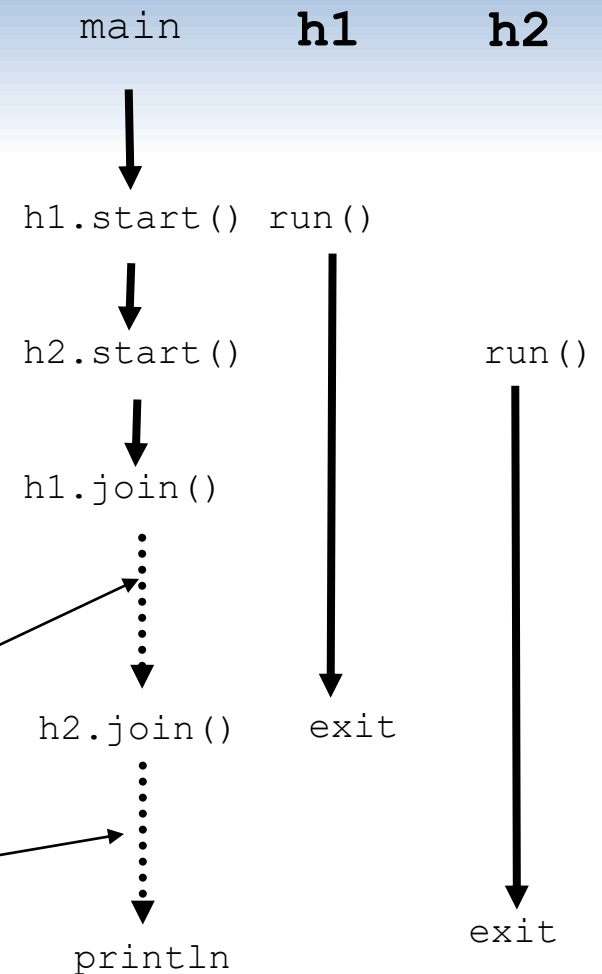
- Una hebra puede esperar (en estado bloqueado) a que otra hebra termine llamando a método **join** sobre esta otra hebra.

```
class Hebra extends Thread{
    private int[] v;
    private int inic,fin;
    private int id;
    private boolean[] b;
    public Hebra(int[] v, int inic, int fin, boolean[] b, int id)
    {
        this.inic = inic; this.fin = fin;
        this.id= id;
        this.v= v;
        this.b = b;
    }
    public void run(){
        int i = inic;
        boolean escero = true;
        while (escero && i < fin){
            escero = v[i] == 0;
            i++;
        }
        b[id] = escero;
    }
}
```

```
public static void main(String[] args) {
    Random r = new Random();
    int[] v;
    boolean[] b = {true,true};
    v= new int[r.nextInt(10)];
    for (int i = 0; i<v.length;i++){
        v[i]= r.nextInt(2);
        System.out.print(v[i]+" ");
    }
    Hebra h1 = new Hebra(v,0,v.length/2,b,0);
    Hebra h2 = new Hebra(v,v.length/2,v.length,b,1);
    h1.start();
    h2.start();
    try{
        h1.join();
        h2.join();
    } catch (InterruptedException ie){ }
    System.out.println(b[0] && b[1]);
}
```

El método join

```
public static void main(String[] args) {  
    .....  
    Hebra h1 = new Hebra(v,0,v.length/2,b,0);  
    Hebra h2 = new Hebra(v,v.length/2,v.length,b,1);  
    h1.start();  
    h2.start();  
    try{  
        h1.join();  
        h2.join();  
    } catch (InterruptedException e){ }  
    System.out.println(b[0] && b[1]);  
}
```



main bloqueado

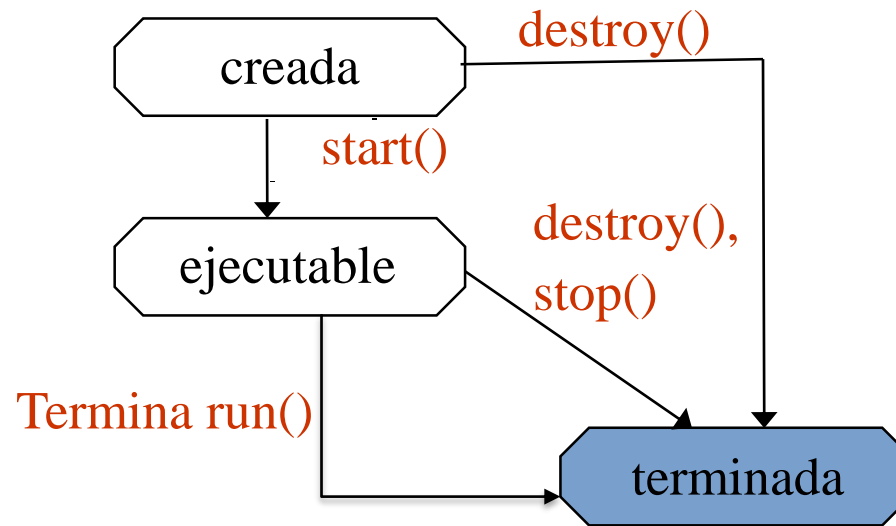
El método isAlive

- Podemos utilizar el método **isAlive** para saber si una hebra todavía no ha terminado su ejecución
- Pero a diferencia de **join**, **isAlive** consume ciclos del procesador

```
public static void main(String[] args) {  
    .....  
    Hebra h1 = new Hebra(v,0,v.length/2,b,0);  
    Hebra h2 = new Hebra(v,v.length/2,v.length,b,1);  
    h1.start();  
    h2.start();  
    while (h1.isAlive()); //espera activa  
    while (h2.isAlive()); //espera activa  
    System.out.println(b[0] && b[1]);  
}  
}
```

Ciclo de Vida: estado terminada

- Una hebra puede terminas su ejecución por varios motivos.



```
public class Thread extends Object implements Runnable{  
    ...  
    public void destroy();           // DEPRECATED  
    public final void stop();        // DEPRECATED  
}
```

Terminación de una hebra

- Cuando el método **run** termina su ejecución
 - Normalmente
 - Como resultado de una excepción no manejada
- Cuando la hebra llama al método **stop**, el método run termina, pero antes:
 - Se liberan los **locks** que tuviera cogidos
 - Se ejecutan las cláusulas **finally** que quedan
 - A continuación el objeto hebra puede ser limpiado por el **recolector de basura** de java
 - stop no es seguro, porque puede dejar al resto de las hebras en un estado inconsistente. En la actualidad **se desaconseja su uso**
- Cuando se llama al método **destroy**
 - En este caso, la hebra termina abruptamente, sin realizarse ninguna acción adicional. Como es un método muy inseguro algunas máquinas javas no lo implementan. **Se desaconseja su uso**

Otras características de las hebras en Java

- Hay métodos de la clase **Thread** que no se han explicado y que se explicarán cuando se usen en temas posteriores
- Otras características de las hebras que se explican a continuación son:
 - Hebras Demonio
 - Hebras vs. Objetos
 - Identidad de una hebra

Hebras Demonio

- En java las hebras pueden ser de dos tipos: hebras de **usuario** o hebras **demonio** (**daemon**)
- Las **hebras demonio** son hebras que proporcionan servicios al resto de las hebras del sistema y, por lo tanto, deben permanecer vivas mientras esté viva cualquier otra hebra
- Cuando todas las hebras usuario han terminado, las hebras demonio pueden acabar y el programa principal también termina

Hebras Demonio

```
public class Thread extends Object implements Runnable{
    ...
    public final boolean isDaemon();
    public final void setDaemon(boolean on);
}
```

- El método `setDaemon` permite convertir en demonio una hebra. Debe llamarse antes de que la hebra comience a ejecutarse (*antes de llamar a `start()`*)
- El método `isDaemon` nos dice si la hebra es demonio (true) o usuario (false)

Hebras vs Objetos

- En java **una instancia** de la clase **Thread** es un **objeto**, y como tal podemos pasar una referencia a ese objeto a métodos.
- Cualquier hebra que tiene una referencia a otra hebra puede ejecutar cualquier método del objeto Thread de la otra hebra.
- El **objeto** Thread **no** es la **hebra**.
- El objeto Thread equivale al conjunto de datos y métodos que encapsulan información sobre la hebra.
- Cualquier otra hebra puede acceder a esos datos y métodos.
- Por lo tanto, el código de la clase a la que pertenece un objeto no nos dice nada sobre las hebras que están ejecutando sus métodos, o examinando sus datos.
- Incluso si un objeto es de una clase que extiende de Thread, puede haber miles de hebras ejecutando su código.

Identidad de una hebra

- Podemos saber qué objeto hebra en particular está ejecutando un método utilizando el método **currentThread** de **Object**.
- Como este método es estático, hay sólo uno para todas las instancias, y podemos llamarlo utilizando la clase **Thread**.

```
public class Thread extends Object
    implements Runnable {
    ...
    public static Thread currentThread();
    ...
}
```

Ejemplo: Paso de una referencia Thread a otra hebra

```
class UnaHebra extends Thread{
    private int id;

    public UnaHebra(int id){
        this.id = id;
    }

    public void quienSoy(){
        System.out.println(Thread.currentThread());
    }

    public String toString(){
        return "Hebra " +id;
    }

    public void run(){
        for (int i = 0; i<Iter ; i++){
            quienSoy();
        }
    }
}
```

```
class OtraHebra extends Thread{
    private int id;
    private UnaHebra h;
    public OtraHebra(int id, UnaHebra h){
        this.id = id;
        this.h = h;
    }

    public String toString(){
        return "OtraHebra " +id;
    }

    public void run(){
        for (int i = 0; i<Iter; i++){
            h.quienSoy();
        }
    }
}
```

```
public static void main(String[] args) {
    UnaHebra h= new UnaHebra(0);
    OtraHebra o = new OtraHebra(1,h);
    h.start();
    o.start();
}
```

objetos

hebras

Ejecución de un programa concurrente

- Ejecutar un programa concurrente es más complejo que uno secuencial.

1. Hay que representar internamente cada objeto hebra:

identificador
estado
entorno volátil: vbles locales, registros cpu,...
enlaces

Ejecución de un programa concurrente

- Ejecutar un programa concurrente es más complejo que uno secuencial.
 1. Hay que representar internamente cada objeto hebra
 2. Hay que decidir en cada momento a qué hebra le toca ejecutarse (scheduler o planificador)
 - El criterio que utiliza el planificador se denomina **política de planificación**
- Los programas deben ejecutarse correctamente con independencia de la política de planificación subyacente.
 - En principio, no debería ser necesario conocer cómo la máquina ejecuta nuestro código.
 - Sin embargo, algunas propiedades de justicia (que veremos más adelante) pueden verse afectadas por esta política.

Ejecución de una programa concurrente

- La especificación del lenguaje Java **no define** una forma particular de planificar las hebras. Sólo indica que las hebras deberían planificarse utilizando una **política basada en las prioridades** de las hebras (las hebras con mayor prioridad deberían ejecutarse antes).
- Sin embargo, algunas implementaciones de la máquina virtual podrían no seguir esta recomendación, o llevarla a cabo de forma diferente. Como consecuencia

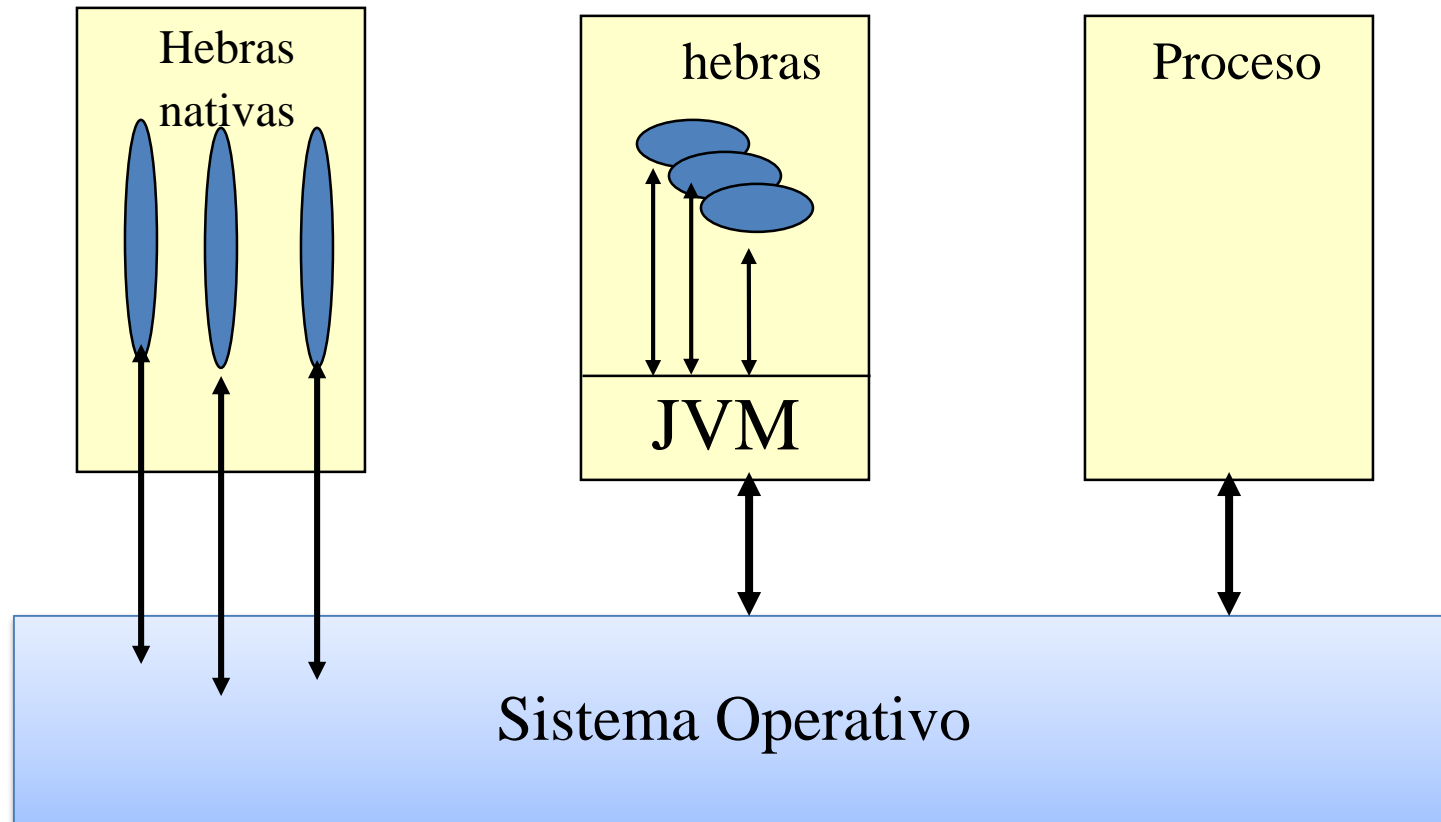
el orden de ejecución de las hebras no está garantizado, depende de la plataforma de ejecución

- Además, si todas las hebras tienen la misma prioridad, esta recomendación no implica nada.

Ejecución de un programa concurrente

- En Java las hebras pueden ser de dos tipos:
 - **Green threads**: son hebras que están implementadas sobre la máquina virtual, que emula la ejecución concurrente implementando su propio planificador. En la actualidad, muy pocas implementaciones de java utilizan este tipo de hebras.
 - **Native threads**: son hebras que el sistema operativo reconoce como tales. Hay una relación 1-1 entre las hebras Java y las que el sistema operativo ejecuta. En este caso, la planificación de las hebras es realizada directamente por el sistema subyacente.

Procesos vs Hebras



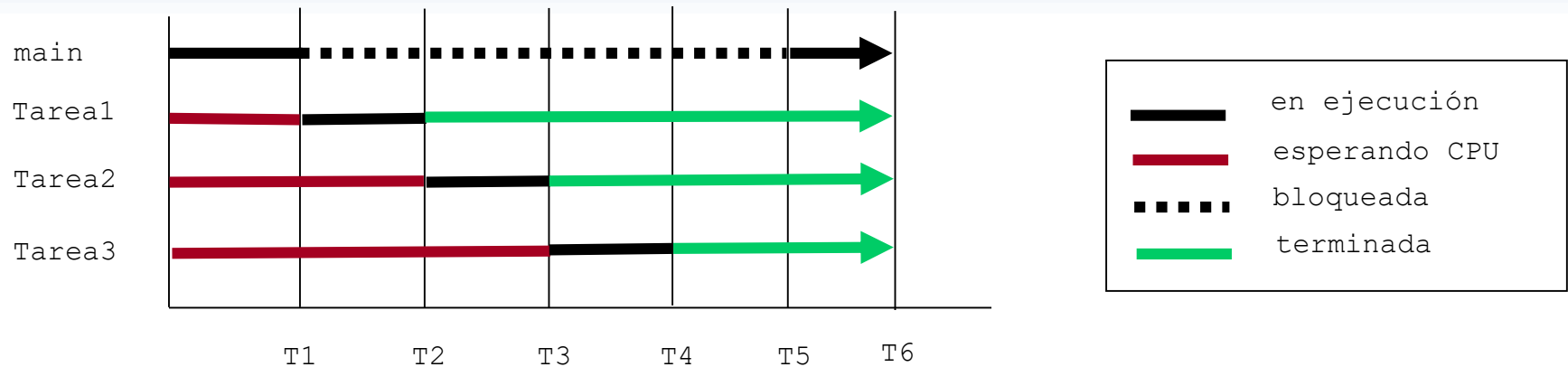
Tipos de Planificación: Ejemplo

```
public class Task extends Thread{
    private long n;
    private int id;
    public Task(long n,int id){
        this.n = n;
        this.id = id;
    }
    private long fib(long n){
        if (n == 0) return 0L;
        if (n == 1) return 1L;
        return fib(n-1)+fib(n-2);
    }
    public void run(){
        Date d = new Date();
        DateFormat df = new SimpleDateFormat("HH:mm:ss:SSS");
        long inicio = System.currentTimeMillis();
        d.setTime(inicio);
        System.out.println("Empieza la tarea "+id+" a las "+df.format(d));
        fib(n);

        long fin = System.currentTimeMillis();
        d.setTime(fin);
        System.out.println("La tarea "+id+" termina a las "+
            df.format(d)+" . Ha tardado "+(fin-inicio)+" milisegundos");
    }
    ....
}
```

```
public static void main(String[] args)
{
    Task[] t = new Task[4];
    for (int i = 0;i<4;i++){
        t[i] = new Task(40,i);
        t[i].start();
    }
}
```

Tipos de Planificación: ejecución 1



Empieza la tarea 0 a las 12:40:39:312

La tarea 0 termina a las 12:40:42:125. Ha tardado 2813 milisegundos

Empieza la tarea 1 a las 12:40:42:125

La tarea 1 termina a las 12:40:44:921. Ha tardado 2796 milisegundos

Empieza la tarea 2 a las 12:40:44:921

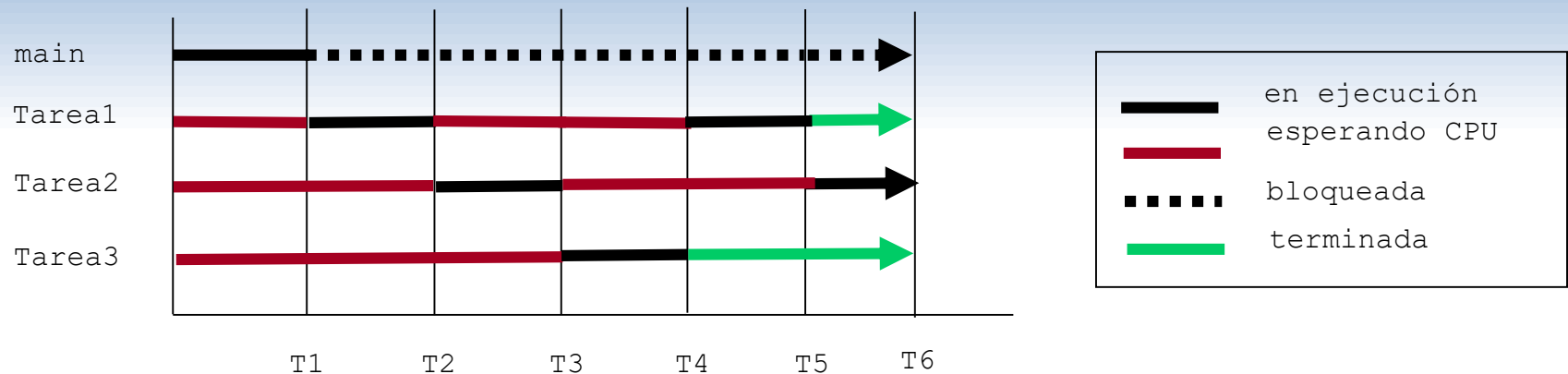
La tarea 2 termina a las 12:40:47:703. Ha tardado 2782 milisegundos

Empieza la tarea 3 a las 12:40:47:718

La tarea 3 termina a las 12:40:50:500. Ha tardado 2782 milisegundos

La ejecución parece secuencial porque el planificador ha dejado ejecutarse cada tarea todo el tiempo que ha necesitado

Tipos de Planificación: ejecución 2



Empieza la tarea 0 a las 12:51:13:031

Empieza la tarea 2 a las 12:51:13:031

Empieza la tarea 3 a las 12:51:13:046

Empieza la tarea 1 a las 12:51:13:046

La tarea 3 termina a las 12:51:18:687. Ha tardado 5641 milisegundos

La tarea 1 termina a las 12:51:18:750. Ha tardado 5704 milisegundos

La tarea 0 termina a las 12:51:18:765. Ha tardado 5734 milisegundos

La tarea 2 termina a las 12:51:18:781. Ha tardado 5750 milisegundos

La ejecución se intercala. El planificador asigna un tiempo de ejecución a cada hebra y cuando se acaba cambia de hebra.

Esta planificación se llama time-slicing, y al cambio de hebra en la CPU se le llama cambio de contexto (context switch)

Referencias

- Concurrency: State Models & Java Programs
Jeff Magee, Jeff Kramer, Ed. Willey
- Concurrent Programming
Alan Burns, Geoff Davies, Ed. Addison Wesley
- Concurrent and Real Time Programming in Java
Andy Wellings, Ed. Willey