

Tema 1. Introducción a concurrencia

Contenido

| | | |
|-------|--|----|
| 1 | Procesos | 2 |
| 1.1 | Aspectos básicos de los procesos | 2 |
| 1.1.1 | Concepto de proceso..... | 2 |
| 1.2 | Estados de un proceso..... | 2 |
| 1.3 | Bloque de control de proceso (PCB) | 4 |
| 2 | Planificación de procesos | 5 |
| 2.1 | Planificación en sistemas de tiempo compartido * | 5 |
| 2.2 | Colas de planificación | 5 |
| 2.3 | Concepto de cambio de contexto (<i>context switch</i>) | 7 |
| 2.4 | Concepto de <i>swapping</i> (intercambio) | 7 |
| 3 | Operaciones sobre procesos | 7 |
| 3.1 | Creación de procesos | 7 |
| 3.2 | Terminación de procesos | 9 |
| 4 | Cooperación entre procesos | 10 |
| 5 | Tipos de sistemas | 11 |
| 6 | Concurrencia..... | 15 |
| 6.1 | Programación Paralela y Concurrente Como diferenciarlas ?..... | 15 |
| 6.2 | Tipos de programación concurrente..... | 16 |
| 6.2.2 | Programación paralela..... | 19 |
| 6.2.3 | Relación Programación concurrente y paralela. | 21 |
| 6.3 | Procesamiento distribuido | 21 |
| 6.4 | Hilos | 24 |
| 6.4.1 | Estados de un hilo | 27 |
| 6.4.2 | Proceso o Hilo ? | 30 |
| 6.5 | Multithreading o multihilo..... | 32 |
| 6.6 | Tecnología Hyperthreading. Procesadores multinúcleo | 34 |
| 7 | Principios de la programación concurrente | 37 |
| 7.1 | Principios de concurrencia | 37 |
| 7.1 | Programas concurrentes | 37 |
| 7.2 | CONDICIONES DE BERNSTEIN | 37 |
| 7.3 | Problemas de los programas concurrentes | 39 |
| 7.3.1 | Conceptos | 39 |

| | | |
|-------|--------------------------------------|----|
| 7.3.2 | Violación de la exclusión mutua..... | 41 |
| 7.3.3 | Deadlock..... | 43 |
| 7.3.4 | Starvation | 45 |

1 Procesos

1.1 Aspectos básicos de los procesos

1.1.1 Concepto de proceso

* Definición informal: un proceso es un programa en ejecución

Un programa ejecutable es un conjunto de instrucciones y datos almacenados en un fichero. Cuando lo que tiene ese programa se carga en la memoria y se pone en ejecución, se convierte en un proceso.

* Definición técnica: un **proceso** es una entidad formada por los siguientes elementos principales:

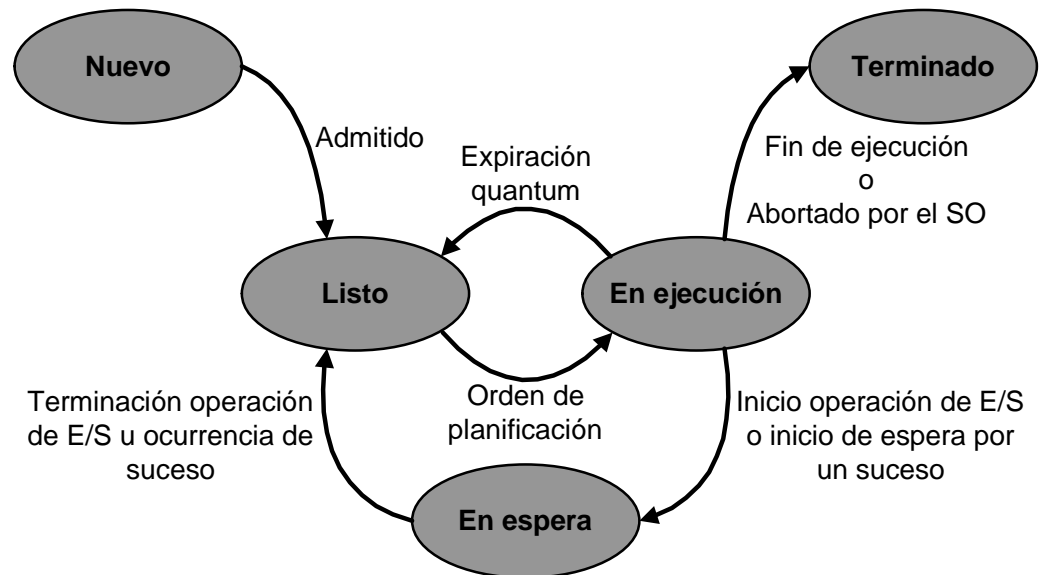
- Una imagen binaria de un programa, cargada total o parcialmente en la memoria física. La imagen binaria está formada por las instrucciones y datos del programa.
- Un área de memoria para almacenar datos temporales, conocida como pila.

La imagen binaria y la pila son el programa en si mismo, pero para que el SO pueda controlar el programa hacen falta una serie de estructuras de datos. Las estructuras fundamentales son:

- La tabla de páginas para traducir las direcciones virtuales generadas por el proceso en las direcciones físicas en la que se encuentra almacenado.
- Una estructura de control, conocida como PCB, para que el sistema operativo pueda controlar su ejecución.

1.2 Estados de un proceso

Un proceso pasa por varios estados durante su ejecución. Los estados posibles para un proceso se muestran en la figura siguiente:



En la figura anterior los nodos (nuevo, listo, etc.) representan los estados y los arcos, las acciones o eventos que llevan a un cambio de estado.

* Definición de los estados:

- **Nuevo:** El proceso se acaba de crear, pero aún no ha sido admitido en el grupo de procesos ejecutables por el sistema operativo.

Habitualmente en un sistema operativo multitarea como Windows, nada más que un proceso se crea, éste resulta admitido, pasando al estado listo. Sin embargo, esto no tiene por qué ser siempre así. Por ejemplo, en una situación de sobrecarga temporal del sistema, el SO puede decidir retardar la admisión de los procesos nuevos. Así se alivia la carga del sistema, ya que hasta que un proceso no es admitido, éste no compite por los recursos del sistema.

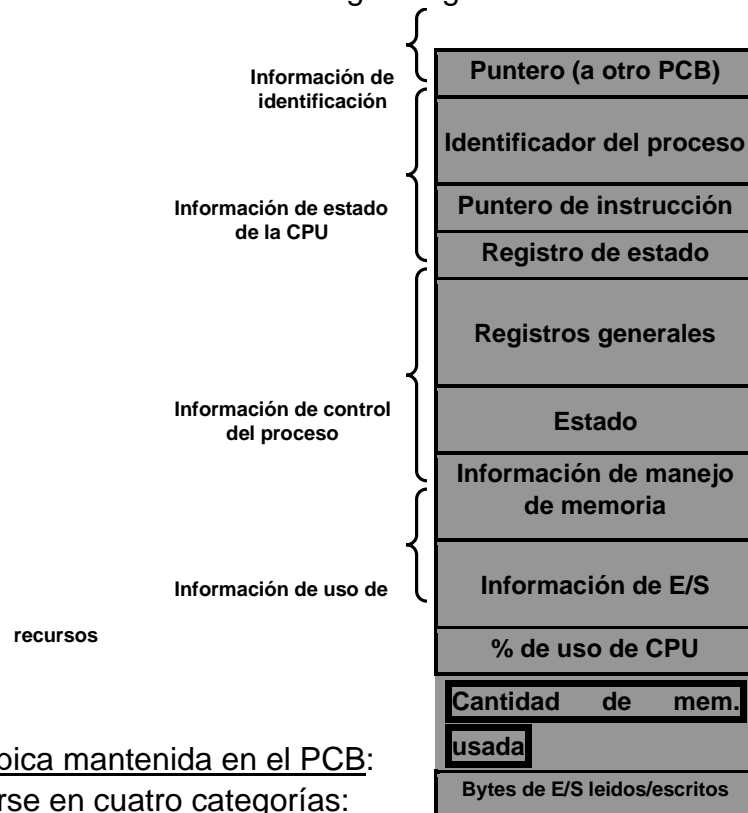
- **Listo:** El proceso está esperando ser asignado al procesador para su ejecución.

Una CPU clásica (con un solo núcleo) solo se puede dedicar en cada momento a un proceso. Los procesos que están preparados para ejecutarse permanecen en estado listo hasta que se les concede la CPU. Entonces pasan al estado “En ejecución”.

- **En ejecución:** El proceso tiene la CPU y ésta ejecuta sus instrucciones.
- **En espera:** El proceso está esperando a que ocurra algún suceso, como por ejemplo la terminación de una operación de E/S.
- **Terminado:** El proceso ha sido sacado del grupo de procesos ejecutables por el sistema operativo. Después de que un proceso es marcado como terminado se liberarán los recursos utilizados por ese proceso, por ejemplo, la memoria.

1.3 Bloque de control de proceso (PCB)

- ✓ PCB = Process Control Block
- ✓ Definición: Es una estructura de datos que permite al sistema operativo controlar diferentes aspectos de la ejecución de un proceso.
- ✓ Estructura típica del PCB de un proceso:
El PCB se organiza en un conjunto de campos en los que se almacena información de diversos tipos. Los campos típicamente mantenidos en el PCB de un proceso se muestran en la figura siguiente:



- ✓ Información típica mantenida en el PCB:
Puede clasificarse en cuatro categorías:
 - Información de identificación
Esta información está integrada básicamente por el **identificador del proceso (PID)**, que es un número que identifica al proceso. Este número es diferente para todos los procesos que se encuentran en ejecución.
 - Información de estado de la CPU
Se trata de un conjunto de campos que almacenan el estado de los registros de la CPU cuando el proceso es suspendido.
 - Información de control del proceso
Se trata de un conjunto de información que es utilizada por el sistema operativo para controlar diversos aspectos de funcionamiento del proceso. Pertenecen a esta categoría de información los siguientes campos:
 - **Estado del proceso:** Listo, en ejecución, etc.
 - **Información de manejo de memoria:** Como por ejemplo, la dirección física de memoria en la que se ubica la tabla de páginas del proceso.

- **Información de E/S:** Lista de ficheros abiertos, ventanas utilizadas, etc.
- Información de uso de recursos
Se trata de un conjunto de información relativa a la utilización realizada por el proceso de los recursos del sistema, como por ejemplo, el porcentaje de utilización de la CPU, la cantidad de memoria usada o los bytes de E/S escritos y leídos por el proceso.

2 Planificación de procesos

El objetivo de los sistemas multitarea es mantener múltiples programas en ejecución simultáneamente, pero como la CPU sólo puede ejecutar un programa de cada vez, hay que decidir quién se ejecuta en cada momento.

Se denomina planificación (*scheduling*) al mecanismo utilizado por el sistema operativo para determinar qué proceso (entre los presentes en el sistema) debe ejecutarse en cada momento.

2.1 Planificación en sistemas de tiempo compartido *

Los sistemas operativos más importantes del mercado actual (Windows, Linux, MacOS y todas las versiones de Unix) se consideran sistemas operativos de tiempo compartido.

- ✓ Objetivo prioritario de estos sistemas: Garantizar que el tiempo de respuesta de los programas se mantiene en unos valores admisibles para los usuarios.

Cuando un usuario interacciona con un programa y le da una orden, quiere que el programa responda en un tiempo razonable. Para conseguir esto hay que hacer que el resto de programas que se encuentren en ejecución no monopolicen la CPU. Para ello, hay que ir repartiendo la CPU entre todos los programas, y además muy rápidamente, para que cada programa tenga una fracción del recurso CPU cada muy poco tiempo.

- ✓ Esquema de funcionamiento: A cada proceso en ejecución se le asigna un *quantum*, que representa el tiempo máximo que puede estar ocupando la CPU. Entonces un proceso abandona la CPU, o bien cuando se bloquea por una operación de E/S (pasando al estado “en espera”), o bien cuando expira su *quantum* (pasando al estado “listo”).

2.2 Colas de planificación

Son unas estructuras de datos que organizan los PCBs de los procesos que se encuentran cargados en el sistema en función de su estado.

- ✓ El SO planifica los procesos en función de la información mantenida en estas colas.
- ✓ Estas estructuras se forman enlazando los PCBs de los procesos mediante punteros.
- ✓ Existen dos tipos de colas:

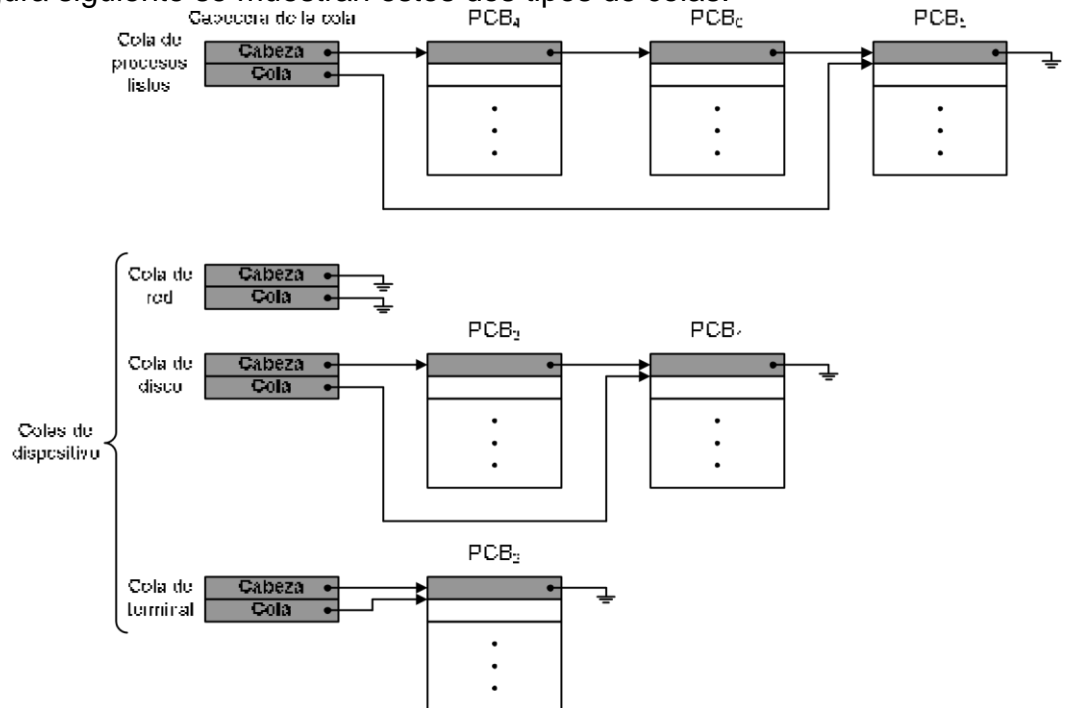
- **Cola de procesos listos:** Contiene a los procesos que se encuentran en el estado “listo”.

Debe indicarse una vez más que estos procesos son los que están preparados para ser asignados a la CPU.

- **Cola de dispositivo:** Contiene los procesos que están esperando por un determinado dispositivo. Estos procesos se encuentran en el estado “En espera”. Cada dispositivo tiene una cola asignada.

Hay muchos dispositivos, como por ejemplo el disco, que son intensivamente utilizados por muchos procesos. Los procesos deben esperar ordenadamente para poder utilizar este recurso.

En la figura siguiente se muestran estos dos tipos de colas:



Las colas se forman enlazando mediante punteros los PCBs de los procesos. El primer campo del PCB es un puntero que se usa para formar estas colas.

La cola tiene una cabecera que contiene dos punteros (llamados cabeza y cola) que se usan para apuntar al primer y último proceso de la cola.

2.3 Concepto de cambio de contexto (*context switch*)

- ✓ Es el hecho de abandonar la ejecución de un proceso y poner en marcha otro proceso.
- ✓ El cambio de contexto requiere salvar el estado que tienen los registros de la CPU justo antes de que ésta abandone el proceso que se saca de ejecución. Así, después, se podrá reanudar la ejecución de este proceso, justo en el punto en el que se suspendió su ejecución. El estado de los registros de la CPU se salva en el PCB del proceso.

2.4 Concepto de *swapping* (intercambio)

Se trata de un mecanismo que permite sacar procesos de ejecución, salvándolos en el disco, para luego volver a ponerlos en ejecución cuando sea requerido.

El objetivo del “*swapping*” es aliviar al sistema, cuando su carga de trabajo es demasiado alta, suspendiendo temporalmente en el disco unidades de trabajo (procesos). Cuando la carga del sistema baja, se ponen de nuevo en ejecución los procesos temporalmente suspendidos. Al final se conseguirá mejorar el rendimiento global del sistema multitarea.

Debe observarse la clara diferencia existente entre los mecanismos del “cambio de contexto” y del swapping.

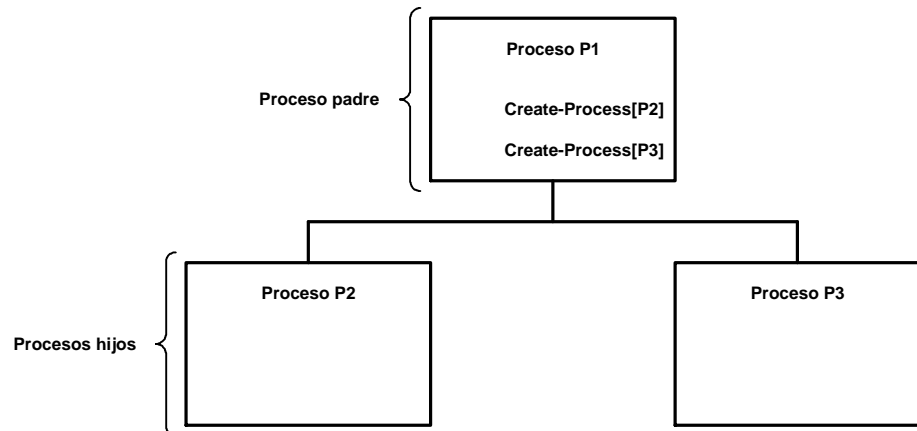
3 Operaciones sobre procesos

Los procesos tienen que poder ser creados y eliminados dinámicamente en el sistema. Debido a ello, el sistema debe proporcionar facilidades para llevar a cabo estas acciones con los procesos. Las funcionalidades básicas se indican a continuación.

3.1 Creación de procesos

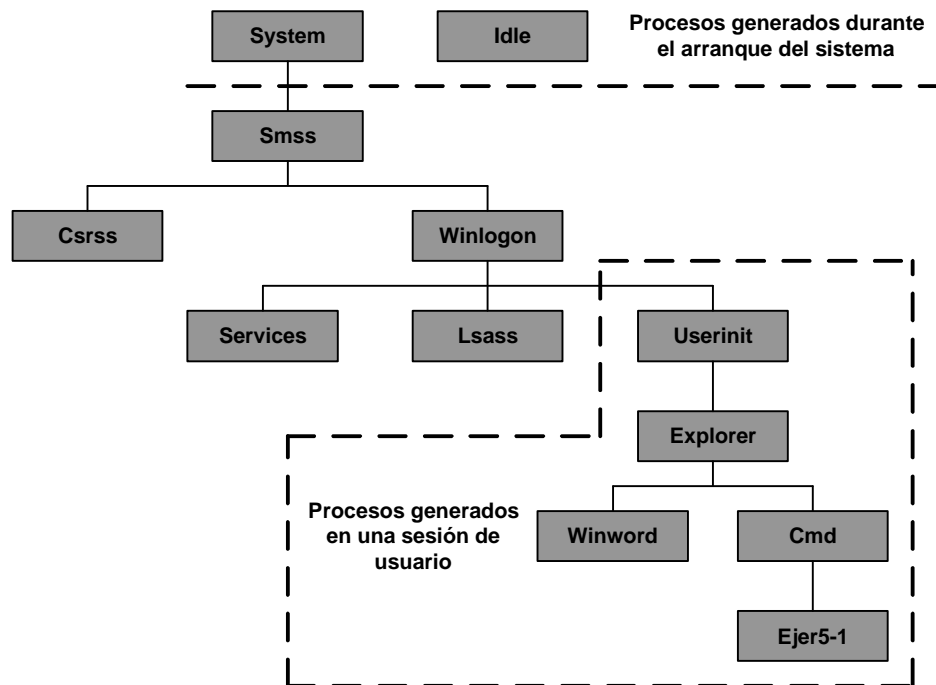
- ✓ Todo sistema operativo debe proporcionar un servicio Create-Process, que será utilizado por un proceso para crear otro proceso.
- ✓ Al proceso que solicita el servicio Create-Process se le denomina proceso padre, y al proceso que es creado mediante este servicio, proceso hijo.

Ejemplo:



- ✓ Este mecanismo de generación de procesos tiene como consecuencia que las relaciones de parentesco entre los procesos existentes en un sistema tenga estructura de árbol.

- ✓ Ejemplo de árbol de procesos típico de una plataforma Windows



Siempre debe haber un proceso raíz, que será creado durante el arranque del sistema. En el caso de Windows este proceso se llama System.

También se crea durante el arranque el proceso idle, que es un proceso que no hace nada. Este proceso es el que se ejecuta cuando la CPU no tiene ningún otro proceso para ejecutar.

A partir de System se van generando otra serie de procesos (Smss, Csrss, Winlogon, etc.) que llevan a cabo labores vitales en el sistema y que por tanto permanecen en ejecución mientras el sistema esté en funcionamiento.

Winlogon controla el inicio de las sesiones de usuario. Cuando un usuario se autentica en el sistema, Winlogon genera el proceso Userinit para cargar el perfil de usuario y

poner en marcha el proceso Explorer que proporciona la interfaz con usuario. A partir del Explorer se irán generando nuevos procesos según sea requerido por el usuario.

Ejemplo de creación de un proceso en Java. Donde creamos un proceso block de notas.

```
package proceso;

import java.io.IOException;

public class Ejemplo1 {
    public static void main(String[] args) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("NOTEPAD");
        Process p = pb.start();
    }
} //Ejemplo1
```

3.2 Terminación de procesos

Un proceso puede terminar por sí mismo, o bien puede ser terminado por otro proceso, que generalmente sólo puede ser su proceso padre.

Un proceso termina por sí mismo llamando a un servicio del sistema, denominado normalmente Exit o Exit-Process.

En los programas sencillos que llevamos a cabo en las prácticas, normalmente no hacemos ninguna llamada al sistema para terminar el proceso, pero ésta es siempre insertada por el sistema de desarrollo. Debe tenerse en cuenta que en la imagen binaria de un programa hay bastante más código que el que explícitamente escribe el usuario.

Un proceso puede terminar la ejecución de un proceso hijo llamando a un servicio del sistema, conocido normalmente como Abort o Terminate-Process.

El que un proceso haga que termine otro proceso es una situación extraordinaria, normalmente ligada a la ocurrencia de errores. Cuando las cosas van bien, los procesos terminan por sí mismos.

Terminando procesos en Java con `System.exit(0);` en un programa Java.

Terminando procesos de la clase ProcessBuilder `p.destroy();` matamos el proceso que hemos creado

```
package proceso;

import java.io.IOException;

public class Ejemplo1 {
    public static void main(String[] args) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("NOTEPAD");
        Process p = pb.start();

        p.destroy(); // Terminar proceso process builder notepad
        System.exit(0); // Termina proceso padre programa java
    }
} //Ejemplo1
```

4 Cooperación entre procesos

En la mayoría de las ocasiones los procesos son entidades totalmente aisladas: llevan a cabo su trabajo sin tener que comunicarse con otros procesos o programas. Todos los programas realizados en las prácticas de la asignatura son así. Son programas muy simples que no necesitan comunicarse con otros programas. Sin embargo las cosas en la realidad no son tan sencillas. En muchas ocasiones, los programas o procesos necesitan intercambiar información entre sí. Pongamos dos ejemplos:

- 1) En una plataforma Windows, el intercambio de información a través del portapapeles.*
- 2) Chatear a través de la red. Hay dos procesos (dos navegadores) que intercambian información.*

Son dos casos totalmente diferentes, pero son dos ejemplos claros de programas que cooperan entre sí.

La cooperación entre procesos requiere que estos se comuniquen. A continuación se indican los mecanismos básicos de comunicación:

✓ Memoria compartida

- Se basa en que los procesos que desean comunicarse compartan una misma región de memoria física. Para llevar a cabo la comunicación, uno escribe y otro lee de la región de memoria compartida.
- Los procesos utilizan servicios del sistema operativo para compartir la región.

✓ Paso de mensajes

- Los procesos utilizan una pareja de servicios del sistema operativo para comunicarse. Estos servicios son conocidos habitualmente como *Send* y *Receive*.
- Para llevar a cabo la comunicación un proceso ejecuta la función *Send* y el otro *Receive*, intercambiando de esta forma un bloque de información que recibe el nombre de mensaje.

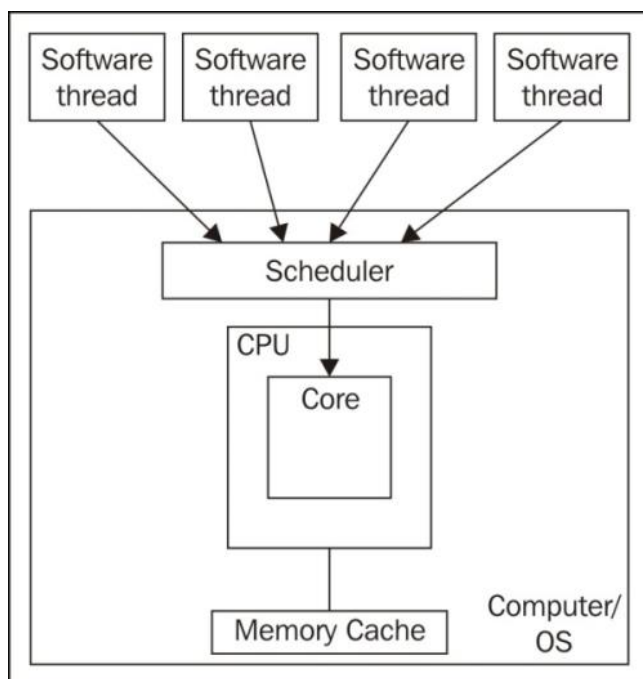
NOTA:

Los párrafos escritos sobre fondo gris y recuadrados mediante línea discontinua contienen información complementaria al resto del contenido de estos apuntes. De cara al examen, los conocimientos fundamentales que el alumno debe adquirir son los que se encuentran fuera de estos recuadros.

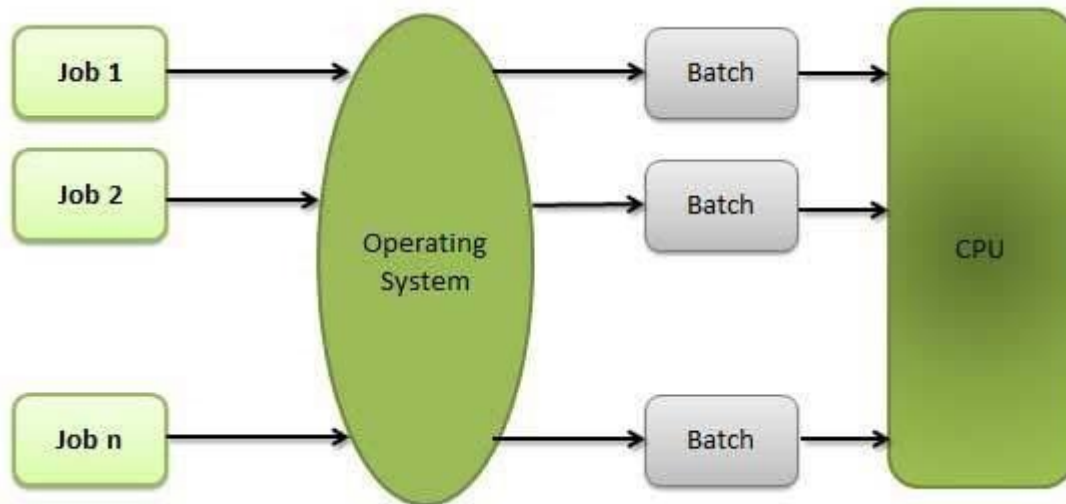
5 Tipos de sistemas

Son entornos que configurados de distinta manera presentan uno o multiples procesadores que ejecutan procesos e hilos que pueden cooperar. Presentan todas las dificultades vistas anteriormente en el tema.

Sistemas monoprocesador: presentan las características de que poseen un único procesador y todos los programas que ejecutamos en ellos deben compartir la tarea. Con tres posibilidades:

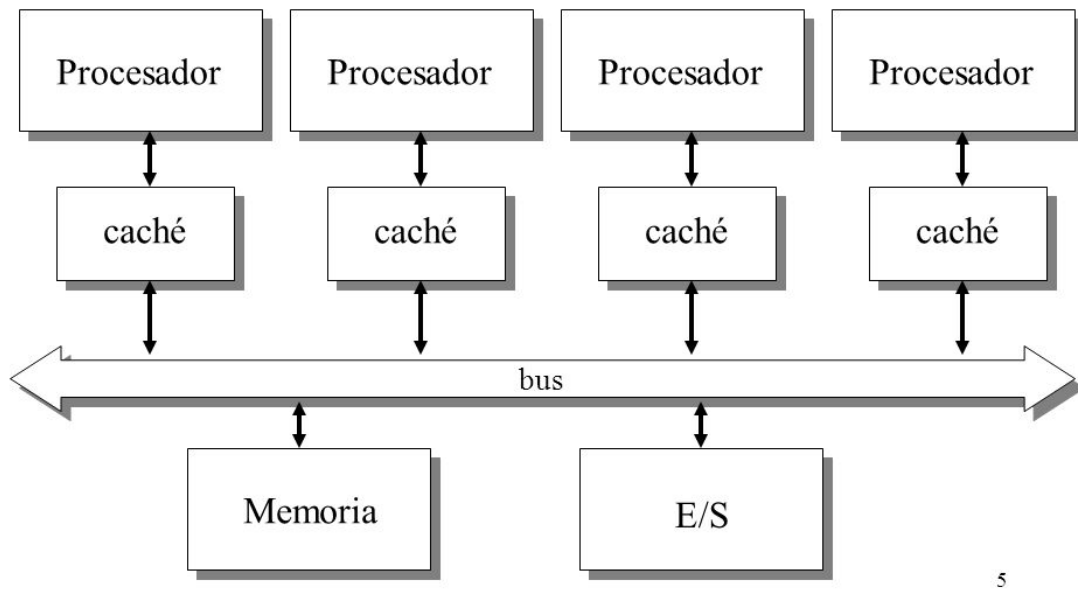


- Un solo programa a la vez en el **procesador, monotarea**. Por ejemplo, **MSDOS**, cuando ejecutaba un programa **el sistema operativo perdía el control de ordenador en favor del programa**.
- **Múltiples programas compartiendo el procesador**, multitarea dando la sensación de ejecutarse a la vez. Lo veremos más adelante.
- **Ejecución de un programa detrás de otro. Sólo un proceso a la vez**, cuando acaba un proceso empieza otro, lo que se conoce como procesamiento por lotes. Los procesos llamados trabajos se **colocan en una cola y se ejecutan uno detrás de otro**.



Sistemas multiprocesador de memoria compartida. En estos **sistemas todos los procesadores comparten una memoria común**, aunque **cada uno pueda tener su propia caché**. Es el caso, por ejemplo de Java Hyperthreading de los Intel Core. Estos procesadores implementan instrucciones Lock para bloquear posiciones de memoria RAM. De esta manera **cuando un proceso o hilo invoca a Lock/Unlock** tenemos que el resto de los **procesos o hilos que acceden a esa posición de memoria se queden bloqueados** esperando a que la **memoria sea liberada**. Java implementa esta opción con las **operaciones synchronize** y con las **operaciones de tipos Atómico** que veremos en este tema.

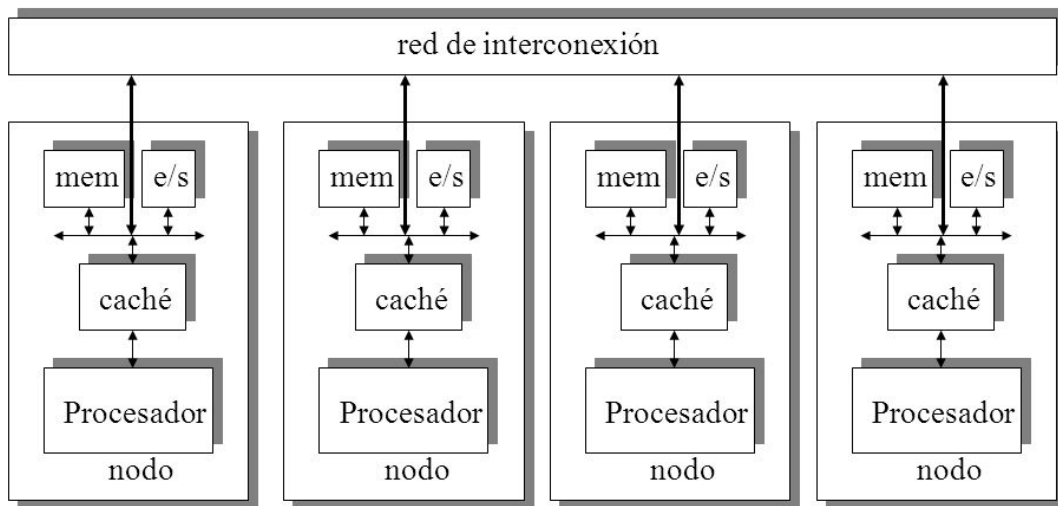
Arquitectura centralizada de memoria compartida



5

Sistemas procesadores de memoria distribuida, cada procesador tiene su memoria RAM propia. Los procesadores colaboran y se **comunican entre ellos con paso de mensajes**. Para comunicarse utilizan un bus o microred de altas prestaciones como podéis ver en la figura siguiente:

Arquitectura de memoria distribuida



Los sistemas vistos hasta ahora cuentan con un reloj del sistema para sincronizarse, lo que facilita la tarea.

Sistemas distribuidos

"Sistemas cuyos **componentes hardware y software**, que están en **computadoras conectadas en red**, se comunican y coordinan sus acciones mediante el paso de mensajes, para el logro de un objetivo. Se establece la comunicación mediante un protocolo preestablecido". Como veréis sólo permite la sincronización por paso de mensajes. En el tema programaremos este tipo de entornos a través del protocolo TCP/IP.

CARACTERÍSTICAS.

- **Concurrencia.** - Esta **característica de los sistemas distribuidos permite que los recursos disponibles en la red** puedan ser utilizados simultáneamente por los usuarios y/o agentes que interactúan en la red.
- **Carencia de reloj global.** - Las **coordinaciones para la transferencia de mensajes** entre los diferentes componentes para la realización de una tarea, no tienen una temporización general, está más bien distribuida en los componentes.
- **Fallos independientes de los componentes.** - Cada **componente del sistema pudiera fallar de manera independientemente**, y los demás continuar ejecutando sus acciones. Esto permite el logro de las tareas con mayor efectividad, pues el sistema en su conjunto continúa trabajando.



6 Concurrencia

6.1 Programación Paralela y Concurrente Como diferenciarlas ?

La concurrencia y el paralelismo han sido extensamente discutidos, pero al momento de definirlos, varios autores salen con su propia definición y esta variedad no hacen tan fácil la tarea de entender. En términos sencillos, se puede explicar así:

Un programa es concurrente si puede soportar dos o más acciones en progreso.

Un programa es paralelo si puede soportar dos o más acciones ejecutándose simultáneamente."

CONCURRENTE



PARALELA



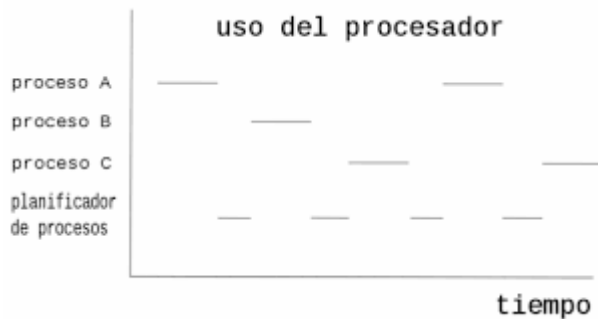
La palabra clave es **en progreso**. Un programa es concurrente por que maneja varias tareas al mismo tiempo, define acciones que pueden ser ejecutadas al mismo tiempo. Y para que un programa sea paralelo, no solo debe ser concurrente, sino que tambien debe estar diseñado para correr en un medio con hardware paralelo (GPU's, procesadores multi-core, etc).

Puede ser visto como que la concurrencia es la propiedad de un programa, mientras que el paralelismo es la forma en la que se ejecuta un programa concurrente.

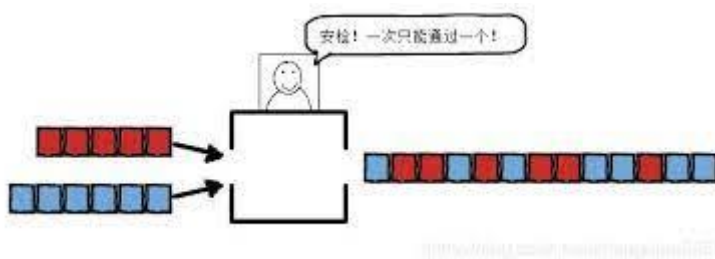
6.2 Tipos de programación concurrente

6.2.1.1 Multiprogramación

Es una **técnica de multiplexación que permite de múltiples procesos en un único procesador**. Es importante resaltar que los procesos nunca corren en paralelo en el procesador, ya que en cada instante de tiempo solo se ejecuta un proceso en el procesador.



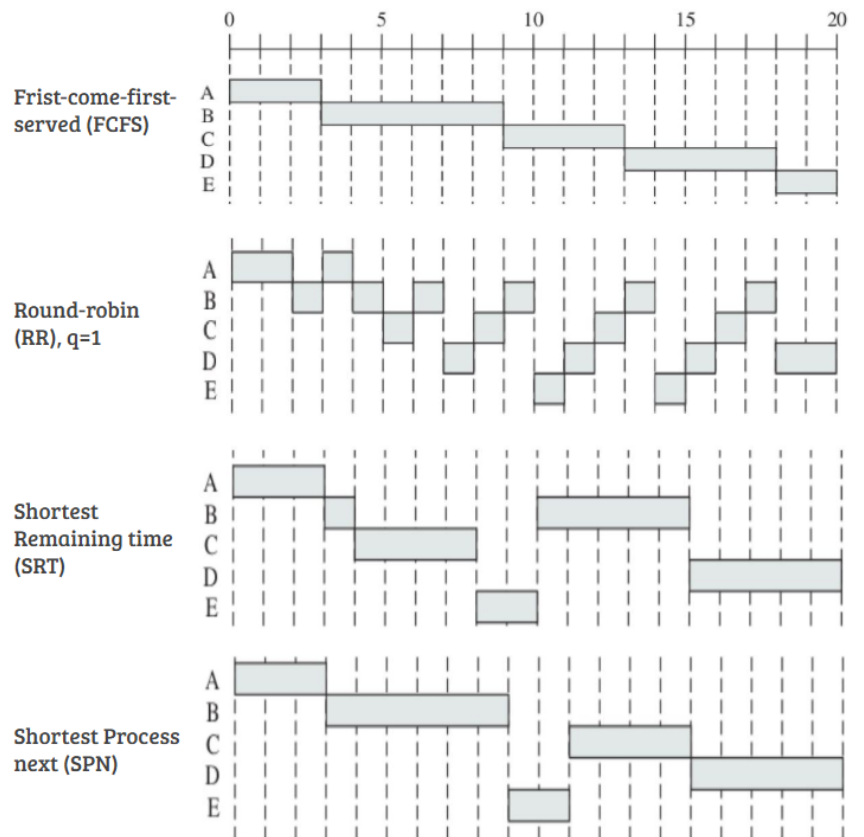
La idea es que una serie de procesos se reparten el tiempo de procesador. **Sólo puede haber un proceso en el procesador a la vez**, porque hay sólo uno, de manera que los procesos van entrando en el procesador **ejecutando unas instrucciones y saliendo para que entren otros**. Se reparten la CPU, hasta que terminan de ejecutarse.



6.2.1.2 Planificación De Procesos

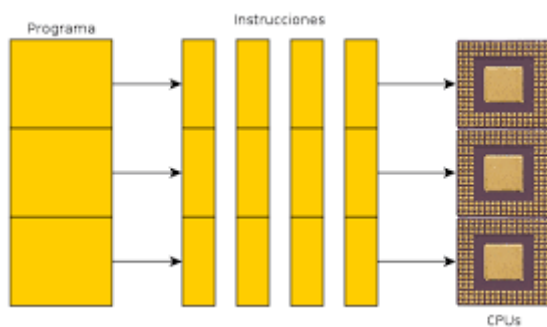
Estrategia de los sistemas operativos con la que se es posible compartir la **CPU** entre los diferentes procesos alojados en memoria. La **calendarización** es un manejo de colas con el objetivo de maximizar el uso de recursos y minimizar retardos. Existen múltiples algoritmos, pero no son el objetivo de este curso. Para realizar programación multitarea hacemos uso de estas capacidades que habitualmente nos proporciona el sistema operativo, desde Windows 95.

| PROCESO | TIEMPO DE ARRIBO | TIEMPO DE SERVICIO |
|---------|------------------|--------------------|
| A | 0 | 3 |
| B | 2 | 6 |
| C | 4 | 4 |
| D | 6 | 5 |
| E | 8 | 2 |

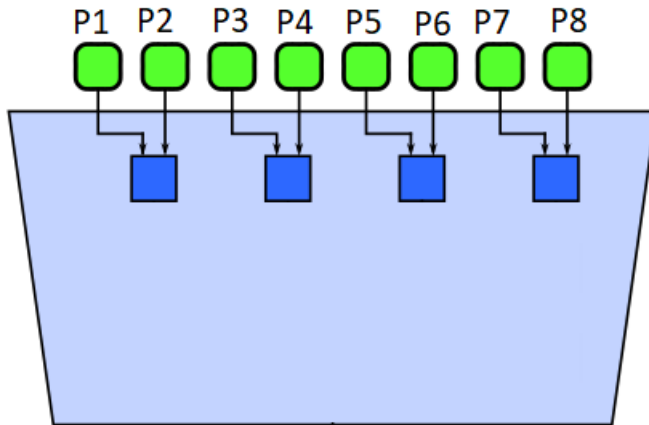


6.2.1.3 Multiproceso

Es una **técnica en la cual se hace uso de dos o más procesadores en una computadora para ejecutar uno o varios procesos**. En este caso los procesos se ejecutan en procesadores independientes de manera simultanea.



Nota: Tened en cuenta que **podemos tener la combinación de ambas, multiproceso y multitarea**, de manera que en cada procesador de nuestro sistema multiprocesador se ejecuten varios procesos.

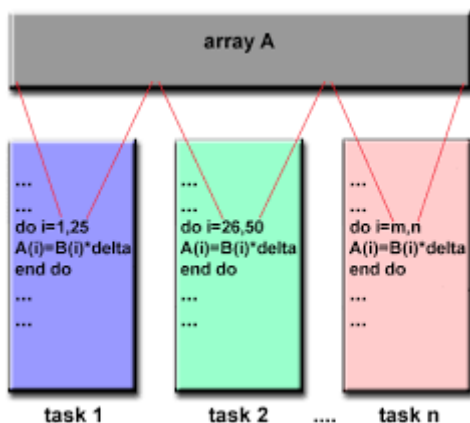


A partir de Windows 8, **Microsoft nos proporciona herramientas para hacer Multitarea pero también programación concurrente y paralela.**

6.2.2 Programación paralela

La **programación paralela** se utiliza para resolver problemas en los que los recursos de una sola máquina no son suficientes. La finalidad de paralelizar un algoritmo es disminuir el tiempo de procesamiento mediante la distribución de tareas entre los procesadores disponibles.

Imaginad que queremos tratar un array con programación paralela. Si tenemos 40 procesos y el array tiene 1000 de valores, podemos hacer que cada proceso se encargue de 25 valores, y luego que junten resultados.



Cuando **terminen todas las tareas, se juntan normalmente con un Join para ofrecer un resultado común y a la vez.** Este join que veremos en el tema 2 con el pool de hilos paralelo hace que todas las tareas esperen porque

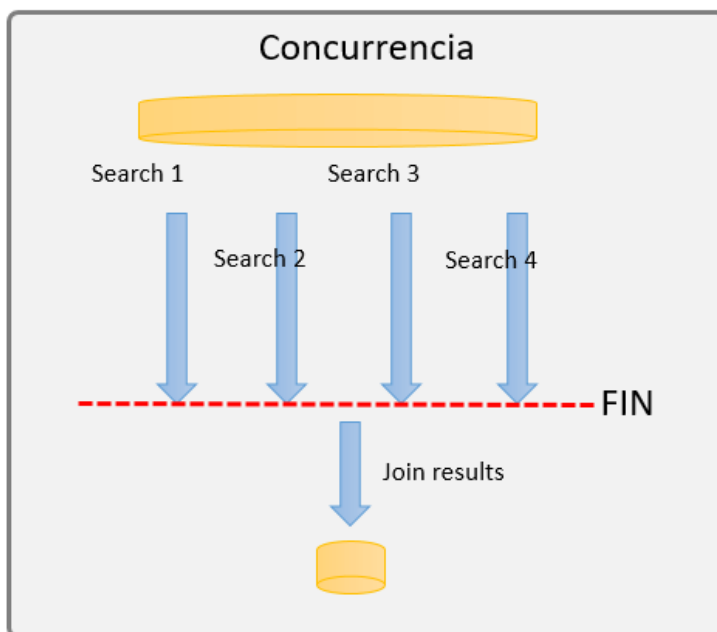
colaboran. De un punto de vista totalmente purista sólo hay programación paralela si todas las tareas se ejecutan a la vez en distintos procesadores y colaboran entre ellas, como en la imagen anterior donde todas hacen un trabajo en común sobre el array. Es valido para otro tipo de tratamientos.

```
task1.join(); task2.join(); ..... task.join();
```

Otro ejemplo de paralelismo puro sería

Ejemplo de paralelismo puro:

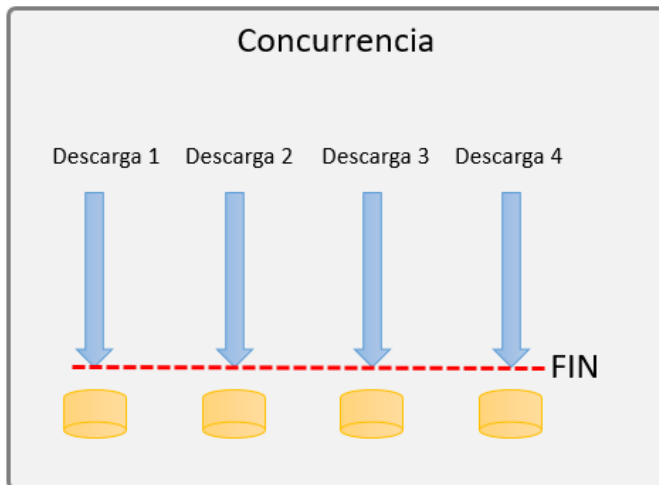
Imagina la clásica página de viajes, donde nos ayudan a **buscar el vuelo más barato o las mejores promociones**, para hacer esto, **la página debe de buscar al momento en cada aerolínea el vuelo más barato, con menos conexiones, etc.** Para esto puedo hacerlo de dos formas, buscar secuencialmente en cada aerolínea las mejores promociones (muy tardado) o utilizar el paralelismo para buscar al mismo tiempo las mejores promociones en todas las aerolíneas.



Ejemplo de concurrencia:

Imagina una aplicación de descarga de música, en la cual puedes **descargar un número determinado de canciones al mismo tiempo**, cada canción es independiente de la otra, **por lo que la velocidad y el tiempo que tarde en descargarse cada una no afectara al resto de canciones.** Esto lo podemos ver como un proceso concurrente, ya que cada descarga es un proceso totalmente independiente del resto. **Como veis la concurrencia también es posible en entornos multiprocesador.** La diferencia desde un punto de vista teórico es que aquí las tareas, aunque se puedan ejecutar cada

una en un procesador son concurrentes porque no colaboran. En entornos prácticos llamaríamos a ambos casos programación paralela.

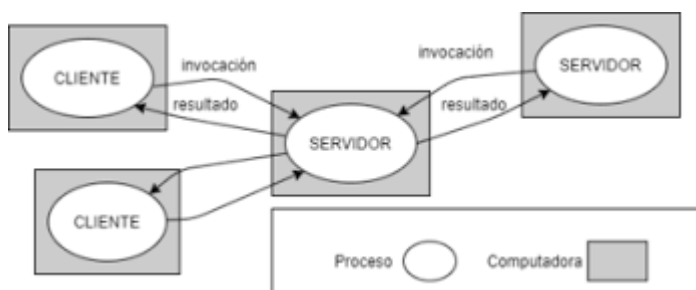


6.2.3 Relación Programación concurrente y paralela.

Múltiples procesos pueden ser ejecutados al mismo tiempo. En otras palabras, **la programación concurrente se comporta igual que la paralela cuando tenemos un sistema multiprocesador** en el cual cada unidad de procesamiento ejecuta un proceso o hilo. En la imagen inferior podemos ver que el la unidad de procesamiento 2 y 3 se comportan igual.

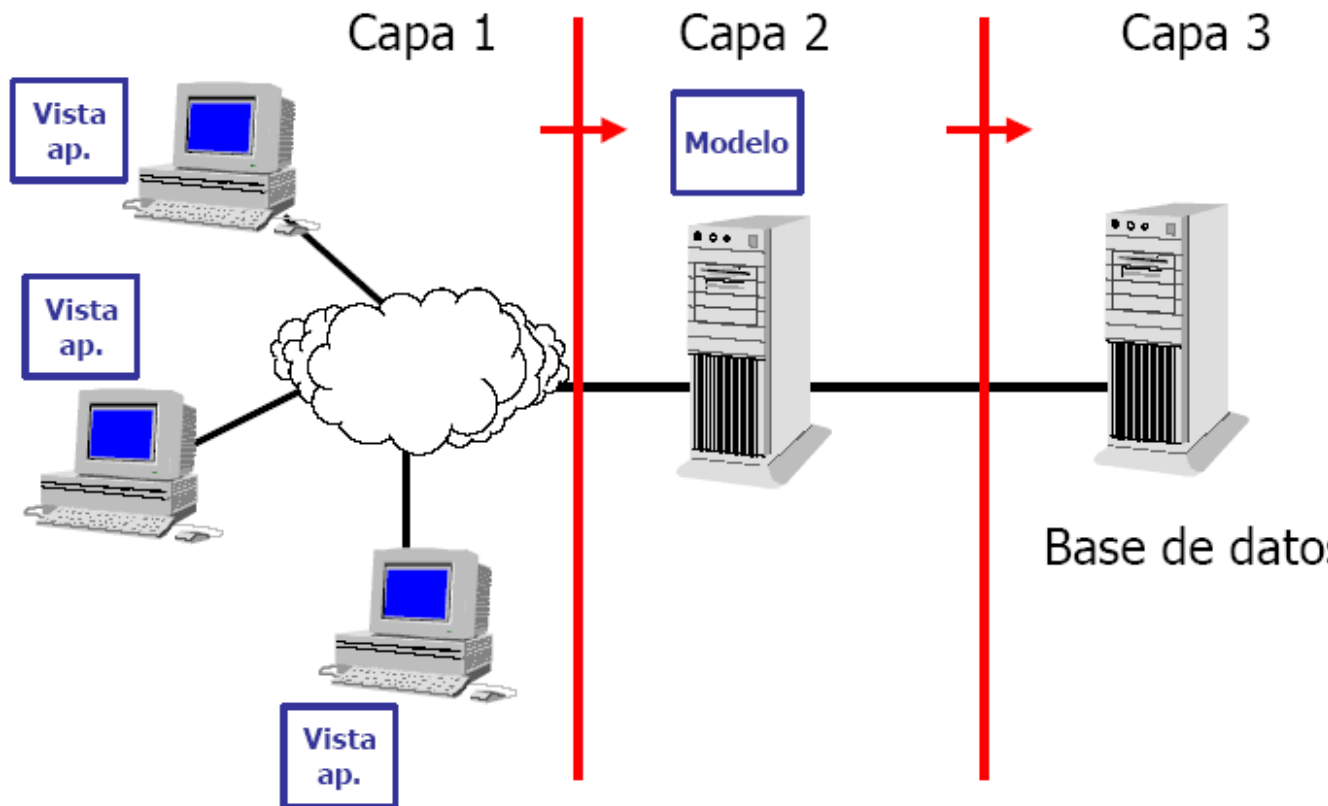
6.3 Procesamiento distribuido

Es cuando uno o varios procesos son ejecutados en una o más computadores. Un ejemplo muy típico son los modelos cliente servidor. Tenemos **procesos ejecutándose en unos ordenadores clientes que se comunican con procesos** ejecutándose en un servidor que puede estar a grandes distancias. Se usa **algún de tipo de protocolo de red habitualmente TCP/IP. Un ejemplo es una aplicación web.**



Otro ejemplo **que esta en desuso es el procesamiento central (Host).**- se refiere a **uno de los primeros modelos de computadoras interconectadas, llamados centralizados, donde todo el procesamiento de la organización se llevaba a cabo en una sola computadora**, normalmente un Mainframe, y los usuarios empleaban sencillas computadoras personales.

Grupo de Servidores.- Otro modelo que **entró a competir con el anterior, también un tanto centralizado, son un grupo de computadoras actuando como servidores**, normalmente de archivos o de impresión, poco inteligentes para un número de minicomputadores que hacen el procesamiento conectados a una red de área local.



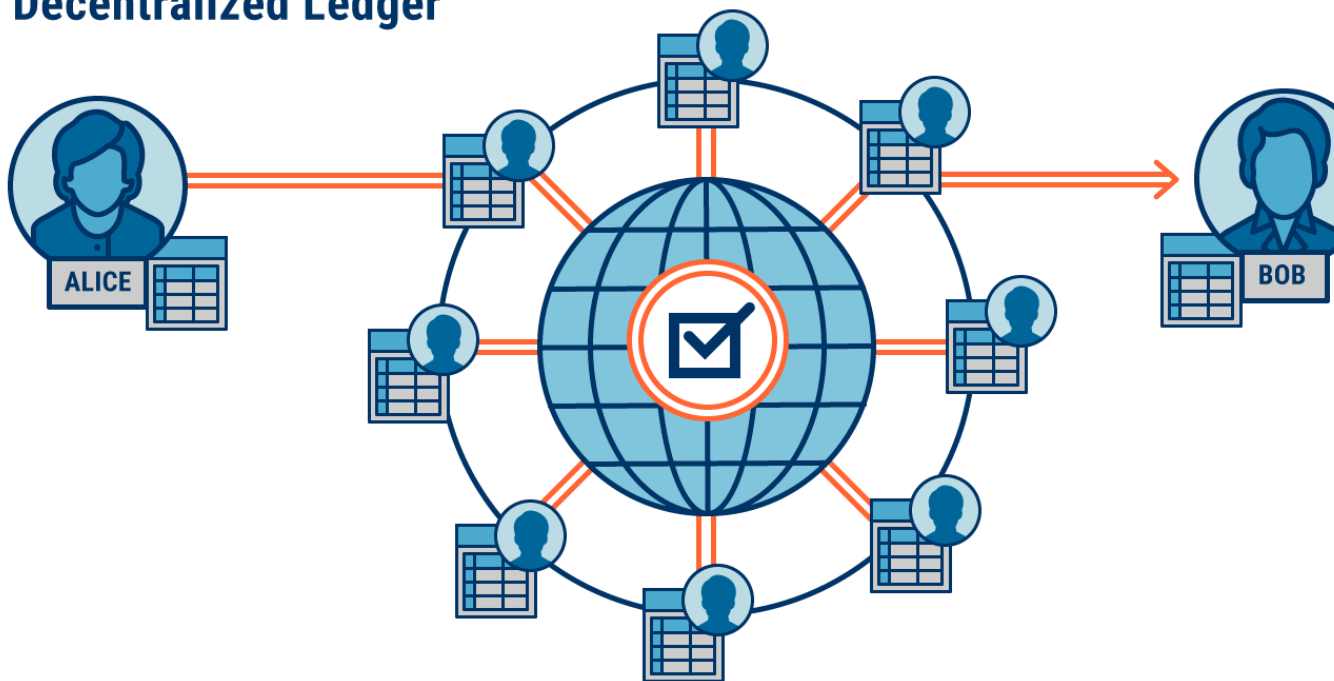
Blockchain es otro ejemplo de programación distribuida. **Múltiples servidores se ponen de acuerdo para guardar la información de una transacción y ofrecer información segura a los clientes.**

Blockchain, a veces conocida como Tecnología de Contabilidad Distribuida (DLT), **hace que la historia de cualquier activo digital sea inalterable y**

transparente mediante el uso de la descentralización (entorno distribuido) y el hash criptográfico.

Una analogía **simple para comprender la tecnología blockchain es un Documento de Google**. Cuando creamos un documento y lo **compartimos con un grupo de personas, el documento se distribuye en lugar de copiarlo o transferirlo**. Esto crea una **cadena de distribución descentralizada que da a todos acceso al documento al mismo tiempo**. Nadie **está bloqueado a la espera de cambios de otra parte**, mientras que **todas las modificaciones al documento se registran en tiempo real**, lo que hace que los cambios sean completamente transparentes.

Decentralized Ledger



CBINSIG

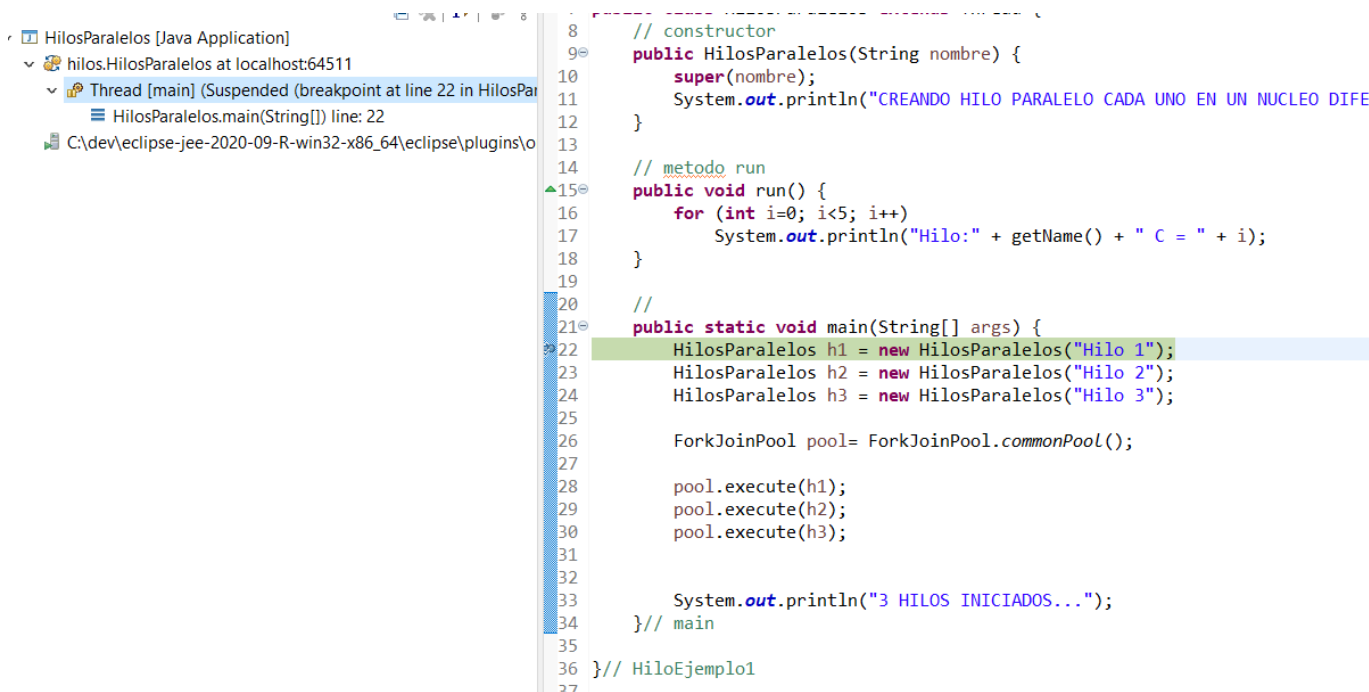
Blockchain se fundamenta en:

- Una **cadena de bloques es una base de datos que almacena bloques cifrados de datos y luego los encadena para formar una única fuente cronológica** de verdad para los datos.
- Los **activos digitales se distribuyen en lugar de copiarse o transferirse, creando un registro inmutable** de un activo.

- El **activo está descentralizado**, lo que permite el **acceso completo en tiempo real** y la **transparencia** al público.
- Un **libro mayor transparente de cambios** preserva la **integridad del documento**, lo que **crea confianza** en el activo.
- Las **medidas de seguridad inherentes de Blockchain** y el **libro mayor público** lo convierten en una **tecnología principal** para casi todos los sectores.

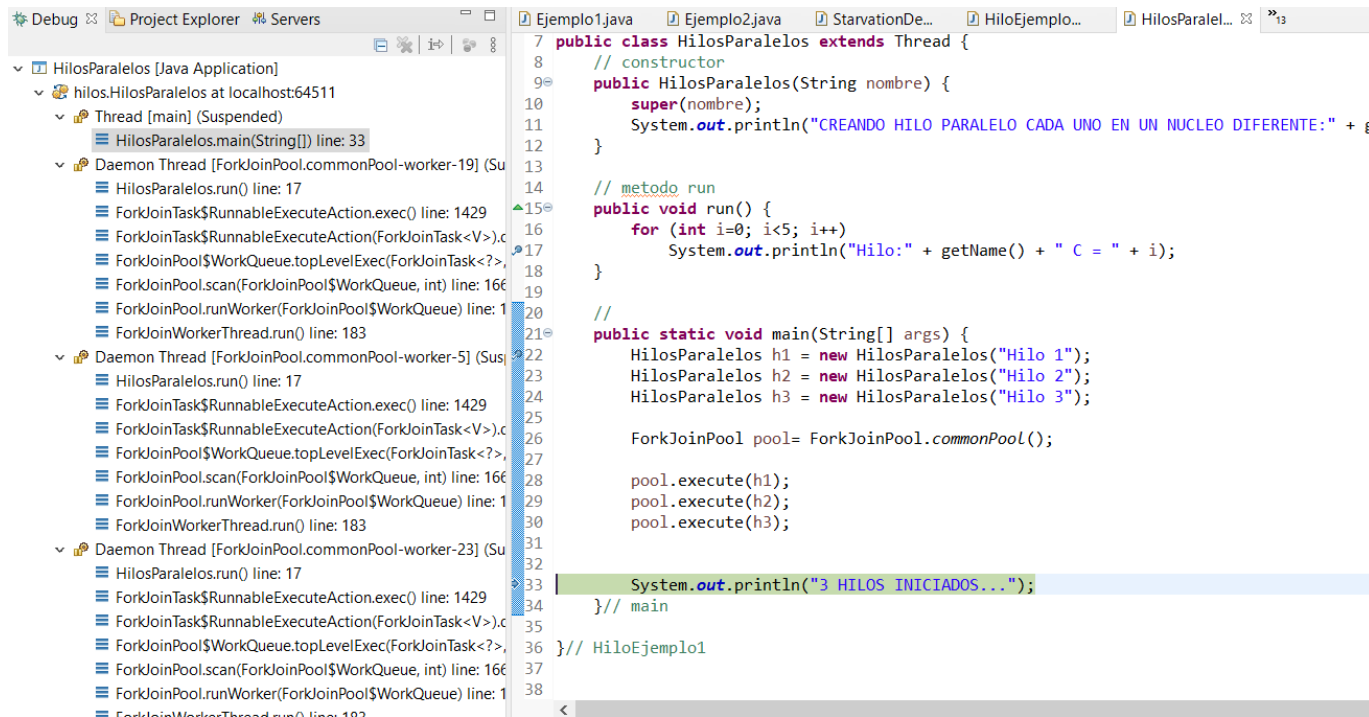
6.4 Hilos

Lo primero ha tener en cuenta es que **cualquier programa tiene un hilo de ejecución principal**. Si depuro un programa Java os encontrareis en el área de depuración de Eclipse con el hilo principal Main. Para probarlo podéis usar el ejemplo HilosParalelos.java de la **sección Multithreading o Multihilo**.

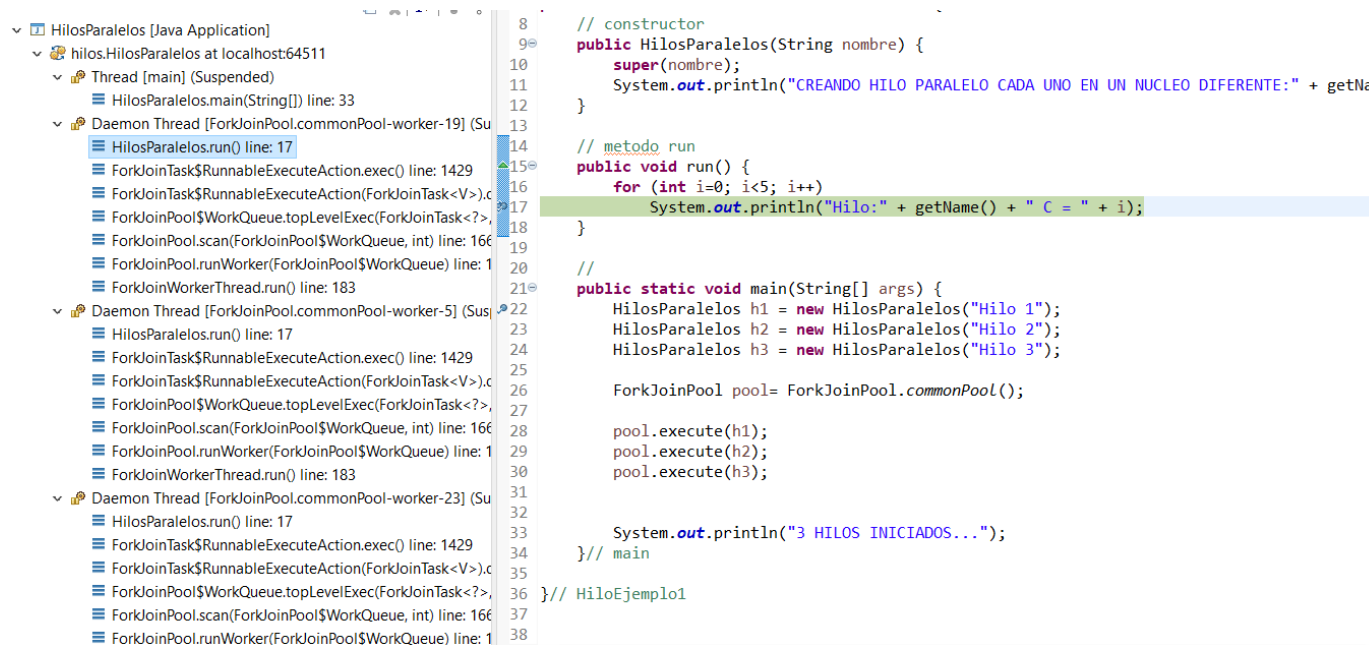


```
8 // constructor
9 public HilosParalelos(String nombre) {
10     super(nombre);
11     System.out.println("CREANDO HILO PARALELO CADA UNO EN UN NUCLEO DIFE
12 }
13
14 // metodo run
15 public void run() {
16     for (int i=0; i<5; i++)
17         System.out.println("Hilo:" + getName() + " C = " + i);
18 }
19
20 //
21 public static void main(String[] args) {
22     HilosParalelos h1 = new HilosParalelos("Hilo 1");
23     HilosParalelos h2 = new HilosParalelos("Hilo 2");
24     HilosParalelos h3 = new HilosParalelos("Hilo 3");
25
26     ForkJoinPool pool= ForkJoinPool.commonPool();
27
28     pool.execute(h1);
29     pool.execute(h2);
30     pool.execute(h3);
31
32     System.out.println("3 HILOS INICIADOS...");
33 } // main
34 } // HiloEjemplo1
35
36
37
```

Pero Java como otros entornos de programación nos permite abrir múltiples hilos de ejecución dentro de un mismo proceso, es como llevar varias líneas de ejecución en un programa y cada una puede hacer una labor diferente. Cuando lanzo los tres hilos en el programa anterior, tengo cuatro hilos de ejecución el principal y los tres hilos que pertenecen al mismo proceso, programa. Lo podéis ver en la siguiente imagen, donde hemos parado al hilo principal.

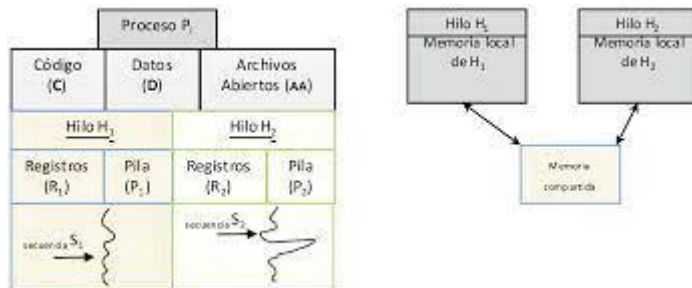


Si cambio de hilo voy a la ejecución de ese hilo que lo hemos parado en el método run. Como veis cada hilo esta haciendo una cosa diferente dentro de un programa, con la ventaja de que pueden compartir variables, ya que los hilos pertenecen al proceso y comparten toda la información, entre procesos sobre todo en sistemas SandBox no es así.

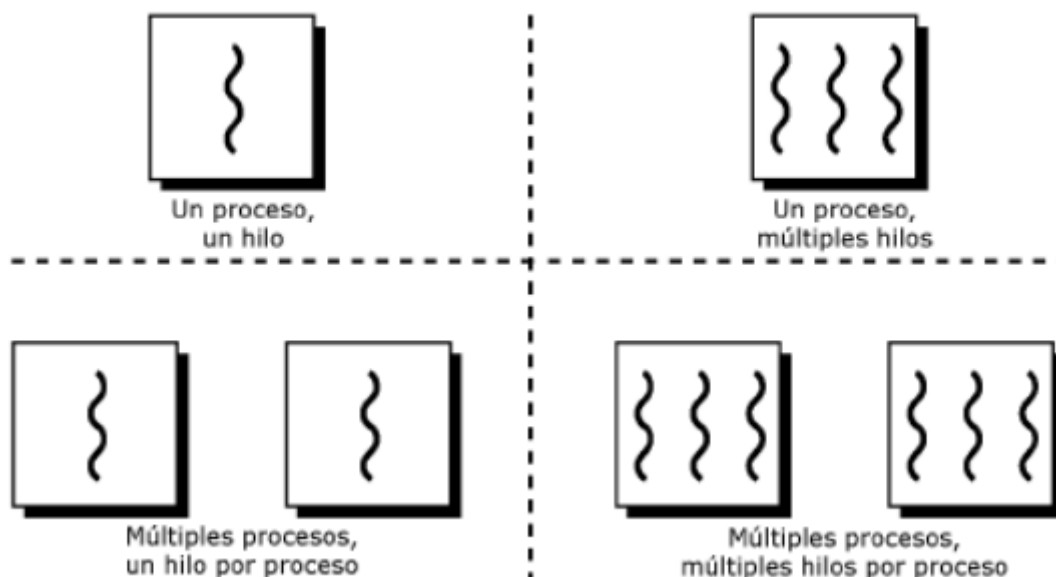


Se puede definir como una unidad básica de ejecución del Sistema Operativo para la utilización del CPU. Este es quien va al procesador y realiza todos los cálculos para que el programa se pueda ejecutar. Es necesario, ya que debe contar con al menos un hilo para que cualquier programa sea

ejecutado. Cada **hilo** tiene: **id de hilo**, **contador de programa**, **registros**, **stack**. Dentro de las características que se encuentran se tiene que cada hilo tiene información del código de máquina que se va a ejecutar en el procesador, sus respectivos datos, el acceso a los archivos, el registro y su respectivo stack en donde se guarda toda la información necesaria del hilo como son las variables locales, variables de retorno o algo a lo que se acceda en tiempo de ejecución.



Los hilos que pertenecen a un mismo proceso comparten: sección de código, sección de datos, entre otros recursos del sistema. El multithreading es la capacidad para poder proporcionar múltiples hilos de ejecución al mismo tiempo. Una aplicación por lo general es implementada como un proceso separado con muchos hilos de control. Dentro de las semejanzas de hilo y multihilo es que poseen un solo bloque de control de proceso (PCB) y un solo espacio de direcciones de proceso. Por el contrario, en las diferencias es que mientras un hilo tiene una pila para sus registros y stack, el multihilo tiene una pila para cada hilo con subbloques de control internos, incluyendo la pila de registros y sus respectivos stacks. Se dice que los hilos de ejecución que comparten recursos agregando estos recursos da como resultado el proceso.



Ejemplo de hilo en java:

```
package hilos;

public class HiloEjemplo1 extends Thread {
    // constructor
    public HiloEjemplo1(String nombre) {
        super(nombre);
        System.out.println("CREANDO HILO:" + getName());
    }

    // metodo run
    public void run() {
        for (int i=0; i<5; i++)
            System.out.println("Hilo:" + getName() + " C = " + i);
    }

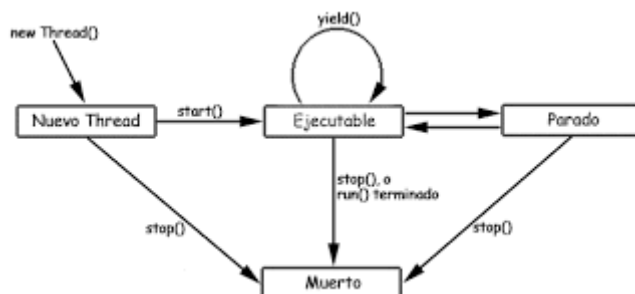
    //
    public static void main(String[] args) {
        HiloEjemplo1 h1 = new HiloEjemplo1("Hilo 1");
        HiloEjemplo1 h2 = new HiloEjemplo1("Hilo 2");
        HiloEjemplo1 h3 = new HiloEjemplo1("Hilo 3");

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

        System.out.println("3 HILOS INICIADOS...");
    } // main
} // HiloEjemplo1
```

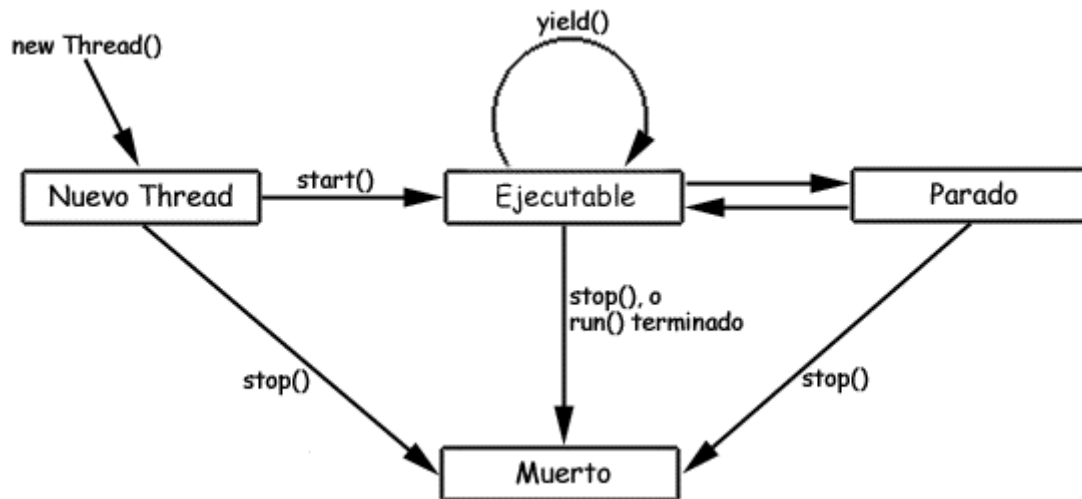
6.4.1 Estados de un hilo

Los hilos igual que los procesos también pueden cambiar de estado y pararse o bloquearse por operaciones de E/S o porque lo forcemos nosotros con los métodos wait() u otras herramientas de sincronización.



Vamos a ver **con método Java de la clase Thread como pasamos de un estado a otro**. En cualquier caso lo repasaremos en detalle en el siguiente tema.

Durante el ciclo de vida de un hilo, éste se puede encontrar en diferentes estados. La figura siguiente muestra estos estados y los métodos que provocan el paso de un estado a otro. Este diagrama no es una máquina de estados finita, pero es lo que más se aproxima al funcionamiento real de un hilo.



Nuevo Thread

La siguiente sentencia crea un nuevo hilo de ejecución pero no lo arranca, lo deja en el estado de *Nuevo Thread*:

```
Thread MiThread = new MiClaseThread();  
Thread MiThread = new Thread( new UnaClaseThread,"hiloA" );
```

Cuando un hilo está en este estado, es simplemente un objeto *Thread* vacío. El sistema no ha destinado ningún recurso para él. Desde este estado solamente puede arrancarse llamando al método `start()`, o detenerse definitivamente, llamando al método `stop()`; la llamada a cualquier otro método carece de sentido y lo único que provocará será la generación de una excepción de tipo *IllegalThreadStateException*.

Ejecutable

Ahora obsérvense las dos líneas de código que se presentan a continuación:

```
Thread MiThread = new MiClaseThread();  
MiThread.start();
```

La llamada al método `start()` creará los recursos del sistema necesarios para que el hilo puede ejecutarse, lo incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método `run()` del hilo de ejecución. En este momento se encuentra en el estado *Ejecutable* del diagrama. Y este estado es *Ejecutable* y no *En Ejecución*, porque cuando el hilo está aquí no está corriendo.

Muchos ordenadores tienen solamente un procesador lo que hace imposible que todos los hilos estén corriendo al mismo tiempo. Java implementa un tipo de *scheduling* o lista de procesos, que permite que el procesador sea compartido entre todos los procesos o hilos que se encuentran en la lista. Sin embargo, para el propósito que aquí se persigue, y en la mayoría de los casos, **se puede considerar que este estado es realmente un estado *En Ejecución***, porque la impresión que produce ante el usuario es que todos los procesos se ejecutan al mismo tiempo.

Cuando el hilo se encuentra en este estado, todas las instrucciones de código que se encuentren dentro del bloque declarado para el método *run()*, se ejecutarán secuencialmente.

Parado

El hilo de ejecución entra en estado *Parado* cuando alguien llama al método *suspend()*, cuando se llama al método *sleep()*, cuando el hilo está bloqueado en un proceso de entrada/salida o cuando el hilo utiliza su método *wait()* para esperar a que se cumpla una determinada condición. Cuando ocurra cualquiera de las cuatro cosas anteriores, el hilo estará *Parado*.

Por ejemplo, en el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
    MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
    ;
}
```

la línea de código que llama al método *sleep()*:

```
MiThread.sleep( 10000 );
```

hace que el hilo se duerma durante 10 segundos. Durante ese tiempo, incluso aunque el procesador estuviese totalmente libre, *MiThread* no correría. Después de esos 10 segundos, *MiThread* volvería a estar en estado *Ejecutable* y ahora sí que el procesador podría hacerle caso cuando se encuentre disponible.

Para cada una de los cuatro modos de entrada en estado *Parado*, hay una forma específica de volver a estado *Ejecutable*. Cada forma de recuperar ese estado es exclusiva; por ejemplo, si el hilo ha sido puesto a dormir, una vez transcurridos los milisegundos que se especifiquen, él solo se despierta y vuelve a estar en estado *Ejecutable*. Llamar al método *resume()* mientras esté el hilo durmiendo no serviría para nada.

Los métodos de recuperación del estado *Ejecutable*, en función de la forma de llegar al estado *Parado* del hilo, son los siguientes:

- Si un hilo está dormido, pasado el lapso de tiempo
- Si un hilo de ejecución está suspendido, después de una llamada a su método ***resume()***

- Si un hilo está bloqueado en una entrada/salida, una vez que el comando de entrada/salida concluya su ejecución
- Si un hilo está esperando por una condición, cada vez que la variable que controla esa condición varíe debe llamarse al método ***notify()*** o ***notifyAll()***

Muerto

Un **hilo de ejecución se puede morir de dos formas: por causas naturales o porque lo maten (con *stop()*)**. Un hilo muere normalmente cuando concluye de forma habitual su **método *run()***. Por ejemplo, en el siguiente trozo de código, el bucle while es un bucle finito -realiza la iteración 20 veces y termina-:

```
public void run() {
    int i=0;
    while( i < 20 ) {
        i++;
        System.out.println( "i = "+i );
    }
}
```

Un hilo que contenga **a este método *run()***, morirá **naturalmente** después de que se complete el bucle y *run()* concluya.

También se puede matar en **cualquier momento un hilo, invocando a su método *stop()***. En el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
    MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
    ;
}
MiThread.stop();
```

se crea y arranca el hilo *MiThread*, se duerme durante 10 segundos y en el momento de despertarse, la llamada a su método *stop()*, lo mata. **El problema es que *stop()* está deprecado**.

El método *stop()* envía un objeto *ThreadDeath* al hilo de ejecución que quiere detener. Así, cuando un hilo es parado de este modo, muere asincrónicamente. El hilo morirá en el momento en que reciba ese objeto *ThreadDeath*.

Los applets utilizarán el método *stop()* para **matar a todos sus hilos cuando el navegador con soporte Java** en el que se están ejecutando **le indica al applet que se detengan**, por ejemplo, cuando se minimiza la ventana del navegador o cuando se cambia de página.

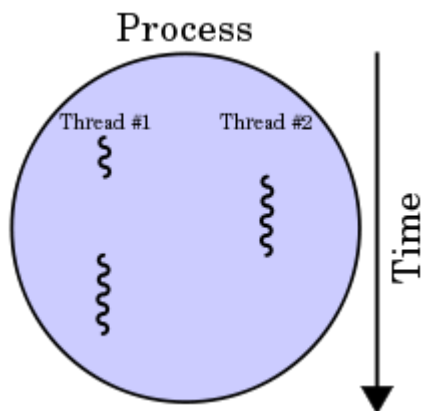
6.4.2 Proceso o Hilo ?

Primero, **los procesos son más pesados de crear y, por lo tanto, utilizan más capacidades de computadora que los hilos**. Esto permite entonces de **hacer tareas más pesadas**. En segundo lugar, los procesos son totalmente

independientes entre ellos. No comparten ningún dato entre ellos; Mientras que, los hilos pueden compartir información fácilmente.

Desafortunadamente, esto crea un problema: **si dos hilos trabajan con los mismos datos, se puede crear errores** como, por ejemplo, la valor cambia sin entender por qué. De hecho, **cuando se realizan dos tareas al mismo tiempo, si modificamos los datos en un hilo y hacemos lo mismo en la otra**, será imposible predecir qué hilo modificarán primero la variable.

Para esto, hay un **sistema para "bloquear" los datos en un hilo. Una vez que este hilo desbloqueará los datos**, el otro hilo recibirá una señal de que puede modificarlo. Para resumir, **cuando se trabaja en los mismos datos con dos subprocesos diferentes**, es necesario **bloquear los datos en un hilo en el momento de modificarlo**, y luego desbloquearlo.



| # | Procesos | Hilos |
|---|---|---|
| 1 | Son programas en ejecución. | Son segmentos de procesos. |
| 2 | Tardan más tiempo en terminar. | Tardan menos tiempo en terminar. |
| 3 | Toma más tiempo crearlos. | Toma menos tiempo crearlos. |
| 4 | Toma más tiempo hacer el cambio de contexto. | Toma menos tiempo hacer el cambio de contexto. |
| 5 | Son menos eficientes en términos de comunicación. | Son más eficientes en términos de comunicación. |
| 6 | Consumen más recursos. | Consumen menos recursos. |
| 7 | Son aislados. | Comparten memoria. |

| # | Procesos | Hilos |
|---|--|---|
| 8 | Son denominados "heavy weight process". | Son denominados "light weight process". |
| 9 | Los procesos tienen su propio PCB, Stack y espacio en memoria. | Los hilos tienen el PCB de sus padres, su propio Thread Control Block (TCB) y Stack y un espacio de memoria compartido. |

6.5 Multithreading o multihilo

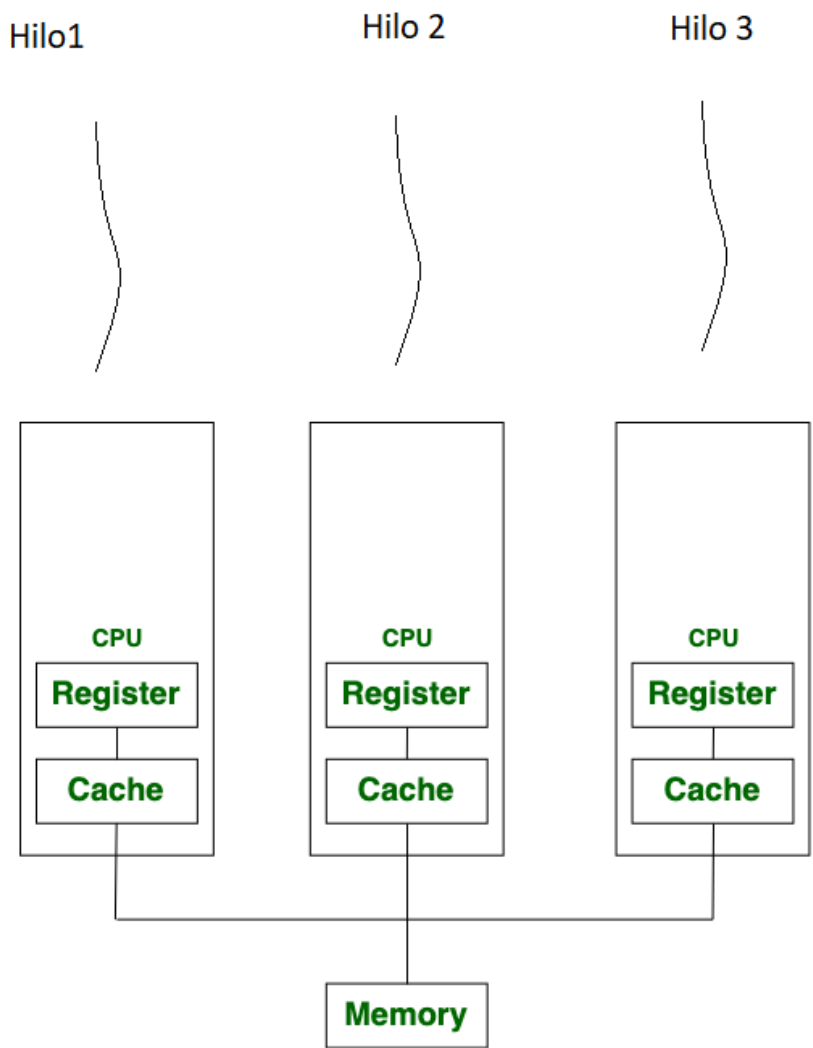
Consiste en la ejecución de múltiples hilos en un programa, y cada uno de ellos realizará una tarea y **podrán comunicarse y sincronizarse entre ellos en caso de usar recursos comunes o intentar completar una tarea común**.

Tenemos dos opciones en **Multihilo**:

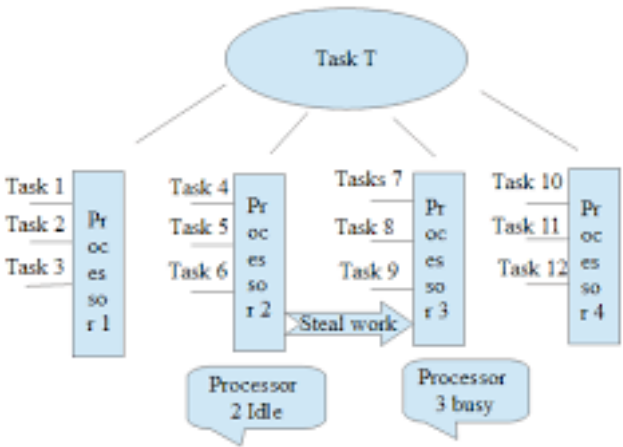
El multitarea multihilo: En un entorno multitarea basado en hilos, el hilo es la unidad más pequeña de código distribuible. Esto significa **que un solo programa puede realizar dos o más tareas a la vez**. Por ejemplo, un editor de texto puede formatear texto al mismo tiempo que está imprimiendo, siempre que estas dos acciones se realicen mediante dos hilos separados. Es el ejemplo **HiloEjemplo1.java** que hemos visto anteriormente.



El multihilo multiprocesador: un **proceso tiene múltiples hilos y cada uno se puede ejecutar en procesadores diferentes de nuestro sistema**. En la imagen siguiente podéis ver como cada hilo de un mismo proceso va a un procesador diferente, ya que todos los procesadores comparten memoria debido a la tecnología hyperthreading. Igualmente, los procesadores multinúcleo ofrecen posibilidades de bloque y sincronización de la memoria.



Multiprocessing



Un ejemplo de **hilos paralelos en Java** sería el siguiente usando el pool de hilos paralelos **ForkJoinPool**:

HilosParalelos.java

```
package hilos;

import java.util.concurrent.ForkJoinPool;

public class HilosParalelos extends Thread {
    // constructor
    public HilosParalelos(String nombre) {
        super(nombre);
        System.out.println("CREANDO HILO PARALELO CADA UNO EN UN NUCLEO
DIFERENTE:" + getName());
    }

    // metodo run
    public void run() {
        for (int i=0; i<5; i++)
            System.out.println("Hilo:" + getName() + " C = " + i);
    }

    //
    public static void main(String[] args) {
        HilosParalelos h1 = new HilosParalelos("Hilo 1");
        HilosParalelos h2 = new HilosParalelos("Hilo 2");
        HilosParalelos h3 = new HilosParalelos("Hilo 3");

        ForkJoinPool pool= ForkJoinPool.commonPool();

        pool.execute(h1);
        pool.execute(h2);
        pool.execute(h3);

        System.out.println("3 HILOS INICIADOS...");
    } // main
} // HiloEjemplo1
```

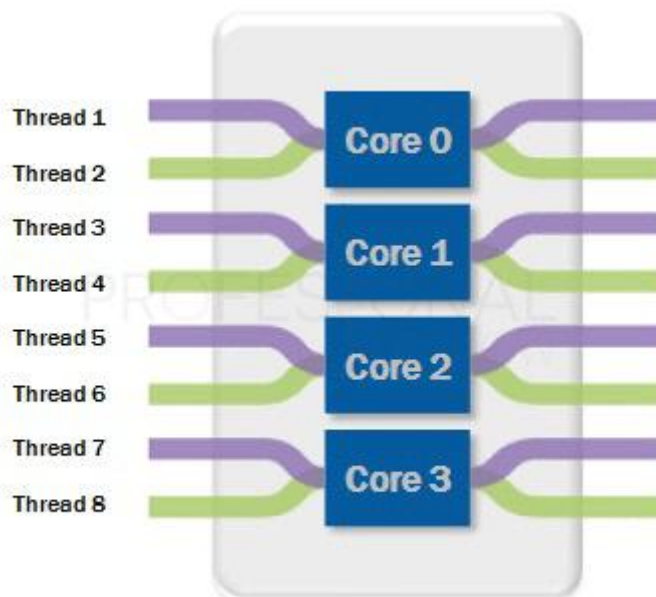
6.6 Tecnología Hyperthreading. Procesadores multinúcleo

La posibilidad de tener múltiples núcleos en nuestro procesador de nuestra ordenador nos abre las puertas a el multihilo paralelo. La lógica parece muy sencilla, meter núcleos y aumentar la cantidad de procesos simultáneos. Pero al principio esto fue un verdadero quebradero de cabeza para los fabricantes de hardware y **sobre todo para los creadores de software**.

Y es que los programas estaban diseñados (compilados) solamente para funcionar con un núcleo. No solo necesitamos que un procesador físicamente sea capaz de hacer múltiples operaciones simultáneas, también **necesitamos que el programa que genera estas instrucciones, pueda hacerlo**

comunicándose con cada uno de los núcleos disponibles. Incluso los sistemas operativos tuvieron que cambiar su arquitectura para ser capaces de utilizar de forma eficiente varios núcleos de forma simultánea.

Los Threads 1 del procesador no son los hilos de los programas, no confundir. **Puedo tener 6 hilos de un proceso** repartidos los dos Threads del Core 0 de manera multitarea.

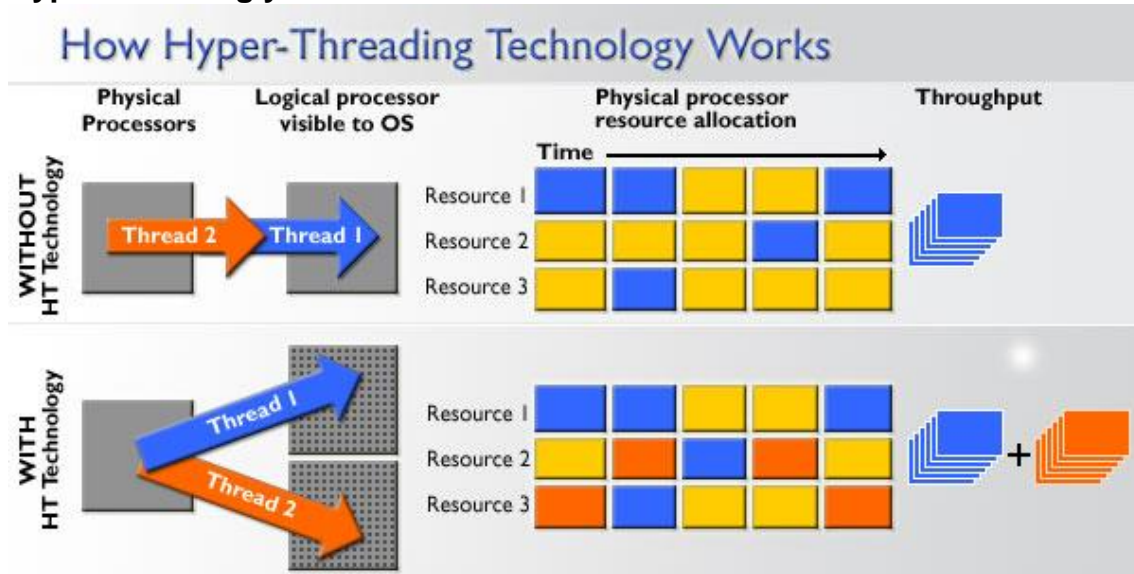


De esta forma, los programadores se pusieron manos a la obra y comenzaron a compilar los nuevos programas con **soporte multinúcleo**, de forma que actualmente, un programa es capaz de utilizar de forma eficiente todos los núcleos que haya disponibles en el ordenador. Multiplicando así los **hilos de ejecución** a la cantidad necesaria. Porque si, además de núcleos, también apareció el concepto de hilo de ejecución.

En un **procesador multinúcleo** es fundamental la paralelización de los procesos que ejecuta un programa, esto implica que **cada núcleo consigue ejecutar una tarea de forma paralela a otro**, y de forma consecutiva, una detrás de otra. A este método de crear distintas tareas de forma simultánea de un programa, se le llama **hilos de procesos, hilos de trabajo, subprocesos o simplemente Threads en inglés**. Tanto el sistema operativo como los programas, deben ser capaces de **crear hilos de procesos paralelos** para

aprovechar toda la potencia del procesador. Esto es algo que **los programas de diseño, edición de vídeo o CAD hacen muy bien**, mientras que los juegos, aunque les queda un camino por recorrer.

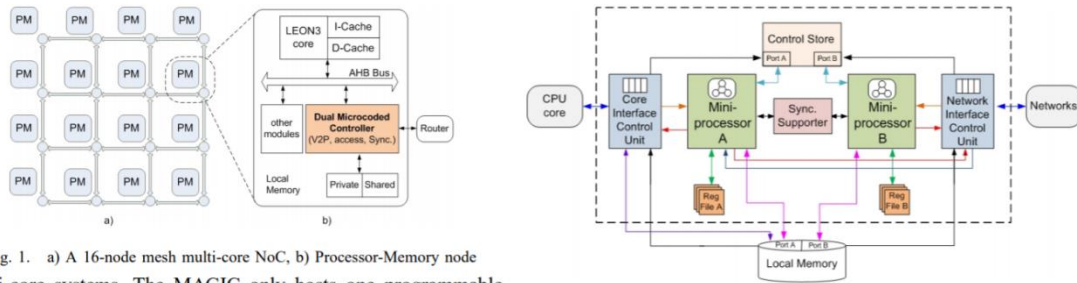
HyperThreading y SMT



En consecuencia de lo anterior, aparecen las tecnologías propias de los fabricantes de procesadores. La más famosa entre ellas es el **HyperThreading** que **Intel** comenzó a utilizar en sus procesadores, y más tarde lo haría **AMD** en los suyos con la tecnología CMT primero, y luego con una evolución a **SMT (Simultaneous Multi-Threading)**.

Esta tecnología **consiste en la existencia de dos núcleos en uno solo**, pero no serán núcleos reales, **sino lógicos**, algo que en programación se denomina hilos de procesamiento o **threads**. Ya hemos hablado antes de ello. **La idea es dividir**, una vez más, **la carga de trabajo entre núcleos segmentando** cada una de las tareas a realizar en subprocesos para que se vayan ejecutando cuando un núcleo esté libre.

Existen procesadores que cuentan con solo dos núcleos, por ejemplo, pero tienen 4 threads gracias a estas tecnologías. Intel la utiliza principalmente **en sus procesadores de alto rendimiento Intel Core** y en las CPU de los portátiles, mientras que AMD la ha implementado **en toda su gama de procesadores Ryzen**. Importante que los núcleos comparten memoria local facilita la comunicación entre hilos de un proceso.



7 Principios de la programación concurrente

En este apartado explicaremos brevemente los principios y problemas de la programación concurrente que usaremos a lo largo del curso.

7.1 Principios de concurrencia

7.1 Programas concurrentes

Un programa concurrente define un conjunto de acciones que pueden ser ejecutadas simultáneamente. Supongamos que tenemos estas dos instrucciones en un programa, está claro que el orden de la ejecución de las mismas influirá en el resultado final:

| | |
|----------|-------------------------------|
| $x=x+1;$ | La primera instrucción se |
| debe; | ejecutar antes de la $y=x+1;$ |
| | segunda. |

En cambio, si tenemos estas otras, el orden de ejecución es indiferente:

| | |
|----------|------------------------------|
| $x=x+1;$ | El orden no interviene en el |
| | resultado final. |
| $y=2;$ | |
| $z=3;$ | |

7.2 CONDICIONES DE BERNSTEIN

Bernstein definió unas condiciones para que dos conjuntos de instrucciones se puedan ejecutar concurrentemente. En primer lugar, es necesario formar 2 conjuntos de instrucciones:

- Conjunto de lectura: formado por instrucciones que cuentan con variables a las que se accede en modo lectura durante su ejecución.
- Conjunto de escritura: formado por instrucciones que cuenta con variables a las que se accede en modo escritura durante su ejecución.

Por ejemplo, sean las siguientes instrucciones:

I.

| | |
|----------------|------------|
| Instrucción 1: | $x := y+1$ |
| Instrucción 2; | $y := x+2$ |
| Instrucción 3: | $z := a+b$ |

Los conjuntos de lectura y escritura estarían formados por las variables siguientes:

| | Conjunto de lectura -L | Escritura - E |
|---------------------|------------------------|---------------|
| Instrucción 1- I1 : | y | x |
| Instrucción 2-I2 : | x | z |
| Instrucción 3-I3 : | a,b | z |

Se pueden expresar de la siguiente manera:

| | |
|-----------------|---------------|
| $L(I1)=\{y\}$ | $E(I1)=\{x\}$ |
| $L(I2)=\{x\}$ | $E(I2)=\{y\}$ |
| $L(I3)=\{a,b\}$ | $E(I3)=\{z\}$ |

Para que dos conjuntos se puedan ejecutar concurrentemente se deben cumplir estas 3 condiciones:

- La intersección entre las variables leídas por un conjunto de instrucciones I_i y las variables escritas por otro conjunto I_j debe ser vacío, es decir, no debe haber variables comunes:

$$L(I_i) \cap E(I_j) = \emptyset$$

- La intersección entre las variables de escritura de un conjunto de instrucciones I_i y las variables leídas por otro conjunto I_j debe ser nulo, es decir, no debe haber variables comunes:

$$E(I_i) \cap L(I_j) = \emptyset$$

- Por último, la intersección entre las variables de escritura de un conjunto de instrucciones I_i y las variables de escritura de un conjunto I_j debe ser vacío, no debe haber variables comunes:

$$E(I_i) \cap E(I_j) = \emptyset$$

En el ejemplo anterior tenemos las siguientes condiciones, donde se observa que las instrucciones I_1 e I_2 no se pueden ejecutar concurrentemente porque no cumplen las 3 condiciones:

| Conjunto I_1 e I_2 | Conjunto I_2 e I_3 | Conjunto I_1 e I_3 |
|-------------------------------------|----------------------------------|----------------------------------|
| $L(I_1) \cap E(I_2) \neq \emptyset$ | $L(I_2) \cap E(I_3) = \emptyset$ | $L(I_1) \cap E(I_3) = \emptyset$ |
| $E(I_1) \cap L(I_2) \neq \emptyset$ | $E(I_2) \cap L(I_3) = \emptyset$ | $E(I_1) \cap L(I_3) = \emptyset$ |
| $E(I_1) \cap E(I_2) = \emptyset$ | $E(I_2) \cap E(I_3) = \emptyset$ | $E(I_1) \cap E(I_3) = \emptyset$ |

En los programas secuenciales hay un orden fijo de ejecución de las instrucciones, siempre se sabe por dónde va a ir el programa. En cambio, en los programas concurrentes hay un orden parcial. Al haber solapamiento de instrucciones no se sabe cuál va a ser el orden de ejecución, puede ocurrir que ante unos mismos datos de entrada el flujo de ejecución no sea el mismo. Esto da lugar a que los programas concurrentes tengan un comportamiento indeterminista donde repetidas ejecuciones sobre un mismo conjunto de datos puedan dar diferentes resultados.

7.3 Problemas de los programas concurrentes

7.3.1 Conceptos

A la hora de crear un programa concurrente podemos encontrar con dos conceptos que pueden producir problemas en nuestros programas o efectos indeseados:

- **Exclusión mutua.** En programación concurrente es muy típico que **varios procesos accedan a la vez a una variable compartida para actualizarla**. Esto se debe evitar, ya que puede producir inconsistencia de datos: uno puede estar actualizando la variable a la vez que otro la puede estar leyendo. Por ello es necesario conseguir la exclusión mutua de los procesos respecto a la variable compartida. Para ello se propuso la región crítica. Cuando dos o más procesos comparten una variable, el acceso a dicha variable debe efectuarse siempre dentro de la región crítica asociada a la variable. Solo uno de los procesos podrá acceder para actualizarla y los demás deberán esperar, el tiempo de estancia es finito.

Imaginad dos procesos o hilos que acceden a la misma variable

Alguien Tiene a X?
Mientras tienen espero
espero wait
Sino

Tengo a X -> mensaje

X= 6

Proceso 1

X= X+1
Aviso de que suelto a X

Alguien Tiene a X?
Mientras tienen

Sino

Tengo a X -> mensaje

X=7

Proceso 2

X=X+3
Aviso de que suelto a X

Si se realiza en orden el resultado final será X=10. Pero si los dos procesos leen a la vez la variable, el resultado puede ser indeseado. Proceso 1 lee 6 y Proceso 2 lee 6 también porque llegan a la vez. El resultado puede ser 7 si Proceso 1 acaba después o 9 Si Proceso 2 acaba después.

- **Condición de sincronización.** Hace referencia a la necesidad de coordinar los procesos con el fin de sincronizar sus actividades. Puede ocurrir que un proceso P1 llegue a un estado X que no pueda continuar su ejecución hasta que otro proceso P2 haya llegado a un estado Y de su ejecución. La programación concurrente proporciona mecanismos para bloquear procesos a la espera de que ocurra un evento y para desbloquearlos cuando este ocurra.

La idea en el problema anterior es que Proceso 1, podría ser Hilo1 también coja la variable de exclusión mutua X y le diga a Proceso2, Espera que termine la operación para leer X. Hacer la operación atómica haciendo que los procesos se comuniquen

X= 6

Proceso 1
Espera(X)
X= X+1
Libera(X)

Proceso 2
Espera(X)
X=X+3
Libera(X)

Proceso 1 ahora le dice que X es sólo para él, y Proceso 2 cuando llega espera y no lee. Cuando Proceso 1 termina, y X=7, entonces Proceso 2, lee y suma 3. Resultado X= 10 correcto.

Algunas herramientas para manejar la concurrencia son: la región crítica, los semáforos, región crítica condicional, buzones, sucesos, monitores y sincronización por rendez-vous. Pero si no usamos bien estas herramientas los procesos se puede quedar todos bloqueados y nunca acabar.

Al lidiar con varias tareas al tiempo los programas concurrentes pueden presentar varios problemas que vamos a intentar ejemplificar en estos apuntes que son:

- **Violación de la exclusión mutua**
- **Deadlock**
- **Starvation o aplazamiento indefinido**
- **Unfairness o Injusticia**

7.3.2 Violación de la exclusión mutua

Violación de la exclusión mutua : Como se mencionó anteriormente, es cuando más de un hilo trata de ejecutar la sección crítica de un programa y lo logra, obteniendo así resultados indeseados. Por ejemplo, tenemos una expresión de tipo:

x = x + 1

Donde x tiene un valor inicial de 6, entonces al haber violación de exclusión mutua, varios hilos podrían tomar una copia local de x, añadir 1 y todos devuelven 7 a x, lo cual es algo que no se quiere.

Un ejemplo que replica este problema en Java, un contador que al final debería dar 1000000, pero siempre arroja diferentes resultados:

ExclusionMutua.java

```
package problemasconcurrencia;
```

```
public class ExclusionMutua {
    public static void main(String[] args) throws InterruptedException {
        class Counter {
            int counter = 0;

            public void increment() {
                counter++;
            }

            public int get() {
                return counter;
            }
        }
        final Counter counter = new Counter();

        class CountingThread extends Thread {
            public void run() {
                for (int x = 0; x < 500000; x++) {
                    counter.increment();
                }
            }
        }

        CountingThread t1 = new CountingThread();
        CountingThread t2 = new CountingThread();
        t1.start();
        t2.start();
        t1.join();
        t2.join();

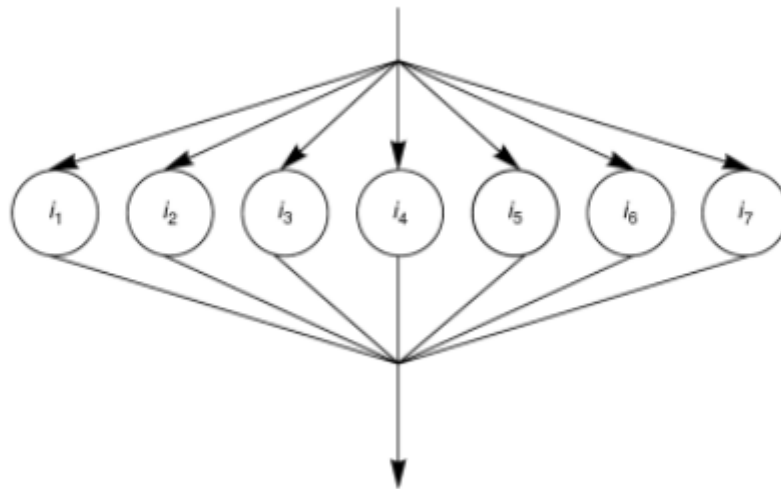
        System.out.println("Con exclusión mutua debería dar 1000000");
        System.out.println("Al violar la exclusión mutua el resultado
es: " + counter.get()); // debería dar 1000000
    }
}
```

Y el resultado es:

Con exclusión mutua debería dar 1000000

Al violar la exclusión mutua el resultado es: 894273

Esto se debe a que **no hay un orden total como sucede en la programación secuencial**, hay un orden parcial, **lo que genera un orden de precedencia de las instrucciones que se deben ejecutar muy diferente**, de tal manera que en dicho árbol se muestra un orden de ejecución simultáneo para todas las instrucciones del programa.



7.3.3 Deadlock

- **Deadlock** : También conocido como abrazo mortal, ocurre cuando un proceso espera un evento que nunca va a pasar. Aunque se puede dar por comunicación entre procesos, es más frecuente que se de por manejo de recursos. En este caso, deben cumplirse 4 condiciones para que se de un "deadlock" :
 - 1) Los procesos deben reclamar un acceso exclusivo a los recursos.
 - 2) Los procesos deben retener los recursos mientras esperan otros.
 - 3) Los recursos pueden no ser removidos de los procesos que esperan.
 - 4) Existe una cadena circular de procesos donde cada proceso retiene uno o más recursos que el siguiente proceso de la cadena necesita.

En este ejemplo crearemos dos clases dentro de nuestro programa, primero crearemos nuestro **main()** pero primero crearemos dos objetos llamados *Lock1* y *Lock2* de tipo **Object** los cuales los utilizaremos en las otras clases, volviendo al **main()** crearemos dos objetos de tipo **ThreadDemo1** y **ThreadDemo2** llamados *T1* y *T2* respectivamente, para luego iniciarlo.

Lo siguiente será una clase una llamada **ThreadDemo1** el cual extiende a la clase **Thread**, en este caso tendremos un método llamado **run()** al cual le crearemos un bloque **synchronized** para *Lock1*, dentro de este bloque primero mostraremos un mensaje relacionado al Thread 1 y el Lock 1, después tendremos un bloque **try/catch** en el cual pausaremos 10 segundos al Thread de turno, después mostraremos otro mensaje, para luego crear otro bloque **synchronized** pero para el objeto *Lock2* donde mostraremos un mensaje, la clase **ThreadDemo2** es similar al anterior pero la única diferencia es que el primer bloque **synchronized** es para el objeto *Lock2* y el

segundo **synchronized** sera para el objeto *Lock1* la funcionalidad es la misma pero solo cambian ligeramente los mensajes, para cada uno de los threads creados, si lo compilamos y ejecutamos nos ocurrira lo siguiente:

TestThread.java

```
package problemasconcurrencia;

public class TestThread
{
    public static Object Lock1 = new Object();
    public static Object Lock2 = new Object();

    public static void main(String[] args)
    {
        ThreadDemo1 T1 = new ThreadDemo1();
        ThreadDemo2 T2 = new ThreadDemo2();
        T1.start();
        T2.start();
    }

    private static class ThreadDemo1 extends Thread
    {
        public void run()
        {
            synchronized(Lock1)
            {
                System.out.println("Thread 1: "
                    + "Reteniendo a Lock1...");
                try { Thread.sleep(10); }
                catch (InterruptedException e) {}
                System.out.println("Thread 1: "
                    + "Esperando a Lock2...");
                synchronized (Lock2)
                {
                    System.out.println("Thread 1: "
                        + "Reteniendo a Lock1 y Lock2");
                }
            }
        }
    }

    private static class ThreadDemo2 extends Thread
    {
        public void run()
        {
            synchronized (Lock2)
            {
                System.out.println("Thread 2: "
                    + "Reteniendo a Lock2...");
                try { Thread.sleep(10); }
                catch (InterruptedException e) {}
                System.out.println("Thread 2: "
                    + "Esperando por Lock1...");
                synchronized(Lock1)
                {
                    System.out.println("Thread 2: "
```

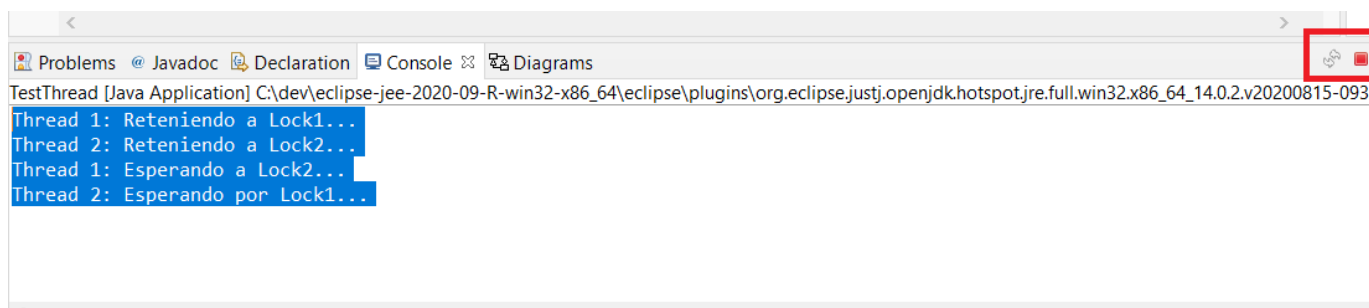
PSP Introducción a concurrencia.

```
        + "Reteniendo a Lock1 y Lock2");  
    }  
    }  
}
```

Y el resultado es:

```
Thread 1: Reteniendo a Lock1...  
Thread 2: Reteniendo a Lock2...  
Thread 1: Esperando a Lock2...  
Thread 2: Esperando por Lock1...
```

Fijaos como el proceso no acaba y el programa se queda bloqueado



Video explicativo de deadlock:

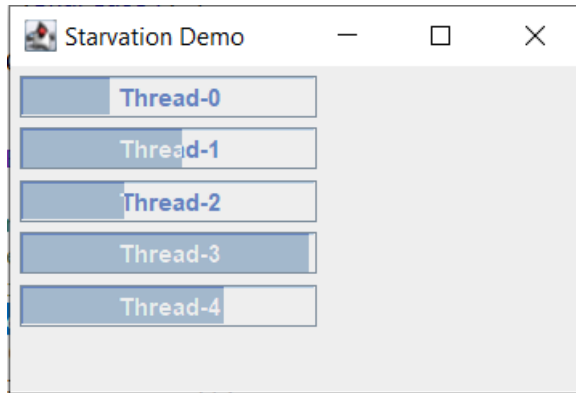
<https://www.youtube.com/watch?v=UwJ1wFxlCRk>

7.3.4 Starvation

- **Aplazamiento indefinido:** O también conocido como "**starvation**" o "**lockout**", se da cuando **el algoritmo que maneja los recursos no tiene en cuenta el tiempo que lleva esperando ese proceso**. Para solucionar esto entre procesos que compiten se puede manejar de tal manera que **mientras más espere un proceso, más alta sera su prioridad**, aunque una solución más sencilla y aplicable a un rango mayor de circunstancias es tratar con los procesos estrictamente en su orden de espera.

En el ejemplo el Hilo 0, Thread-0 tiene prioridad mínima le cuesta avanzar en su tarea. Si hubiera 50 hilos no haría su tarea nunca.

+



```
package problemasconcurrency;

import java.awt.Dimension;
import java.awt.FlowLayout;

import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JProgressBar;

public class StarvationDemo {
    private static Object sharedObj = new Object();

    public static void main (String[] args) {
        JFrame frame = createFrame();
        frame.setLayout(new FlowLayout(FlowLayout.LEFT));

        for (int i = 0; i < 5; i++) {
            ProgressThread progressThread = new ProgressThread();
            frame.add(progressThread.getProgressComponent());

            if (i==0) {
                progressThread.setPriority(Thread.MIN_PRIORITY);
            } else {
                progressThread.setPriority(Thread.MAX_PRIORITY);
            }

            progressThread.start();
        }

        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    private static JFrame createFrame () {
        JFrame frame = new JFrame("Starvation Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(300, 200));
    }
}
```

```
        return frame;
    }

    private static class ProgressThread extends Thread {
        JProgressBar progressBar;

        ProgressThread () {
            progressBar = new JProgressBar();
            progressBar.setString(this.getName());
            progressBar.setStringPainted(true);
        }

        JComponent getProgressComponent () {
            return progressBar;
        }

        @Override
        public void run () {

            int c = 0;
            while (true) {
                synchronized (sharedObj) {
                    if (c == 100) {
                        c = 0;
                    }
                    progressBar.setValue(++c);
                    try {
                        //sleep the thread to simulate long running task
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

- **Injusticia** : También conocido como "unfairness", **ocurre cuando el programa no tiene mecanismos para asegurar que se da un progreso "parejo" en las tareas concurrentes**, este es un aspecto que el diseñador debe tener en cuenta al desarrollar el programa, cualquier descuido a la "justicia" de un programa podría generar un **aplazamiento indefinido**. El problema que hemos visto anteriormente.

Para hacer la competencia justa, y tener un programa fairness hacemos que todos los hilos tengan la misma prioridad y compartan de manera equitativa el objeto sharedObject con el método `sharedObj.wait(100);`. Cada Hilo espera y libera el elemento compartido durante.

FairnessDemo.java

```
package problemasconurrencia;
```

```
import java.awt.Dimension;
import java.awt.FlowLayout;

import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JProgressBar;

public class FairnessDemo {
    private static Object sharedObj = new Object();

    public static void main (String[] args) {
        JFrame frame = createFrame();
        frame.setLayout(new FlowLayout(FlowLayout.LEFT));

        for (int i = 0; i < 5; i++) {
            ProgressThread progressThread = new ProgressThread();
            frame.add(progressThread.getProgressComponent());
            progressThread.start();
        }

        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    private static JFrame createFrame () {
        JFrame frame = new JFrame("Fairness Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(300, 200));
        return frame;
    }

    private static class ProgressThread extends Thread {
        JProgressBar progressBar;

        ProgressThread () {
            progressBar = new JProgressBar();
            progressBar.setString(this.getName());
            progressBar.setStringPainted(true);
        }

        JComponent getProgressComponent () {
            return progressBar;
        }

        @Override
        public void run () {

            int c = 0;
            while (true) {

                synchronized (sharedObj) {
                    if (c == 100) {
                        c = 0;
                    }

                    progressBar.setValue(++c);
                    try {
                        //simulate long running task with wait..
                    } catch (InterruptedException e) {}
                }
            }
        }
    }
}
```



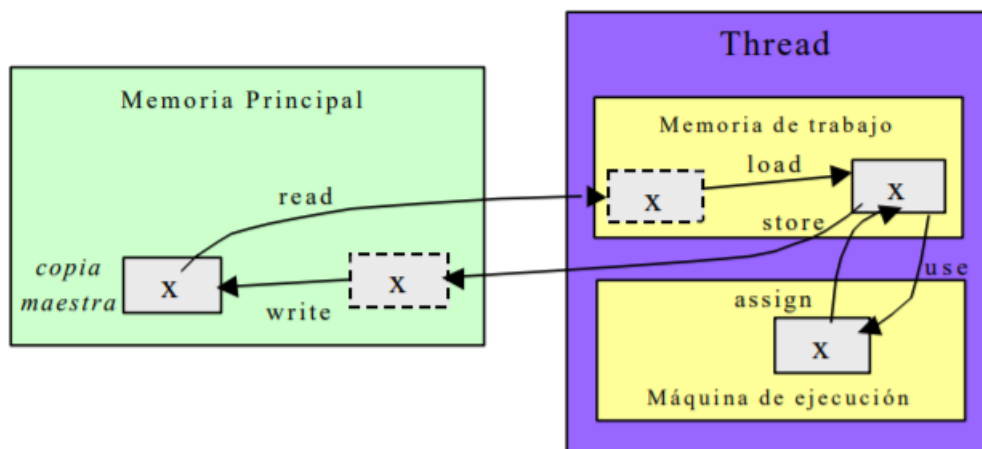
```
        // releasing the lock for long running task gives
        //fair chances to run other threads
        sharedObj.wait(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

8 Sincronización de hilos y procesos en Java. Técnicas

8.1 Memoria compartida

Para memoria compartida java usa varias técnicas que veremos en el tema dos.

- Podemos usar Synchronize sobre objetos de manera que sólo un hilo es dueño del objeto, de esa zona de memoria a la vez. Junto a los métodos wait y notify si queremos bloquear y desbloquear hilos para que no ocupen tiempo de CPU.
- Podemos usar tipos Atómicos sin bloquear hilos.



8.2 Memoria distribuida

Veremos en el tema 3 que cuando dos hilos o procesos no comparten memoria en Java, la solución más eficiente es realizar la sincronización con Sockets, usando conexiones TCP o UDP.

