Unidad Didáctica 1:

INTRODUCCIÓN A LA PROGRAMACIÓN MULTIPROCESO

1. Introducción a los sistemas multitarea

Un ordenador actual de potencia media con cuatro núcleos y correctamente configurado es capaz de realizar simultáneamente varias tareas.

Este hecho sugiere una pregunta, ¿Cómo puede un ordenador ejecutar a la vez más tareas que el número de unidades de proceso que tiene disponibles? La respuesta es gracias a la multitarea, que es la capacidad de ejecutar varias tareas simultáneamente por parte de un computador.

La capacidad de realizar multitarea de un sistema depende principalmente del procesador y del sistema operativo. El primero tiene la capacidad física y el segundo la capacidad lógica. Unas determinadas características del sistema operativo son necesarias para que exista multitarea mientras que las necesidades del procesador son menos restrictivas: un procesador simple con un sistema operativo adecuado puede disponer de multitarea; un procesador con múltiples núcleos con un sistema operativo inadecuado no aprovechara los recursos disponibles.

Alrededor de la multitarea existe una serie de conceptos que, en ocasiones, provocan confusión, ya que la frontera que los delimita es difusa. Concurrencia, paralelismo, proceso o hilo son algunos de los términos cuyo significado es fundamental para comprender la multitarea.

1.1. Programas. Ejecutables. Procesos. Servicios

El sistema operativo es el elemento del ordenador que coordina el funcionamiento del resto de componentes de este, tanto software como hardware. Es a el a quien se indica que se quiere hacer, o siendo mas precisos, que programas se desean ejecutar.

Para llegar a poder ejecutar un programa primero hay que obtenerlo o, en el caso de los programadores, crearlo, para posteriormente proceder a su ejecución. El proceso de creación por parte del programador y de ejecución por parte del usuario final de un programa compilado es el siguiente:

- El programador escribe el código fuente, y lo almacena.
- El programador compila el código fuente utilizando un compilador, generando un programa ejecutable.

- El usuario ejecuta el programa ejecutable, generando un proceso.

Por lo tanto, se puede afirmar que un programa, al ser ejecutado por un usuario, genera un proceso en el sistema operativo: un proceso es un programa en ejecución.

Por su parte, un servicio es también un programa cuya ejecución se realiza en segundo plano y que no requiere la interacción del usuario. Normalmente, se arranca de forma automática por el sistema operativo y esta en constante ejecución.

1.2. Computación concurrente, paralela y distribuida

La multitarea, como indica su nombre, es la capacidad de realizar varias tareas simultáneamente, frente a la restricción de la monotarea, en donde las tareas se ejecutan detrás de otra. Cuando se trabaja en un sistema multitarea, varias tareas avanzan a la vez, aunque pueden hacerlo de diferentes maneras.

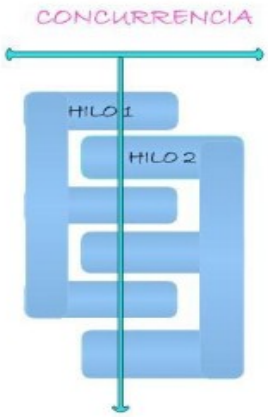
Un sistema basado en un único procesador con un único núcleo es capaz de realizar multitarea mediante el uso de la concurrencia. En la computación concurrente, los tiempos de CPU se reparten entre los distintos procesos según una planificación dirigida por el SO. Si la CPU es suficientemente rápida, el número de procesos limitado y el planificador tienen un buen diseño, todas las tareas avanzaran a la vez (aparentemente) pese a que un ciclo de CPU en un momento dado solo se puede ejecutar una instrucción. En una unidad de tiempo de computación solo avanza un proceso. La velocidad en la asignación de la CPU a los distintos procesos logra que no se perciba el cambio, pero, aunque este se apreciase, seguiría siendo multitarea, ya que todas las tareas avanzan sin tener que esperar unas a que las otras terminen. Este tipo de computación se denomina concurrente.

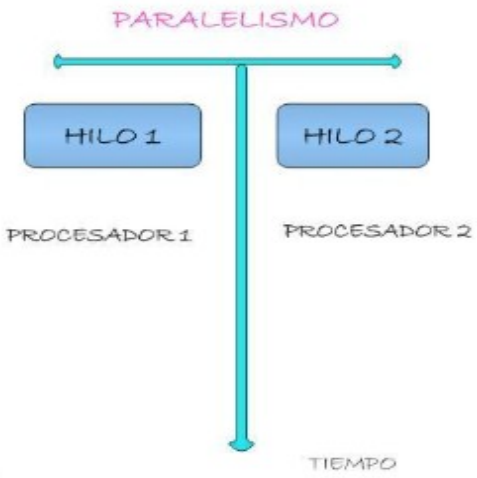
Los sistemas basados en varios procesadores o en procesadores de varios núcleos aportan una mejora sustancial: permiten ejecutar varias instrucciones en único ciclo de reloj. Esta capacidad hace posible ejecutar en paralelo varias instrucciones, lo que da origen al termino procesamiento paralelo. En este tipo de procesamiento, los procesos se dividen en pequeñas subtareas (hilos) que se ejecutan en los diferentes núcleos, consiguiendo una reducción en los tiempos de ejecución de los procesos.

La programación distribuida es otro de los paradigmas de la programación multiproceso. En este tipo de arquitectura, la ejecución del software se distribuye entre varios ordenadores, consiguiendo así disponer de una potencia de procesamiento mucho mas elevada y escalable.

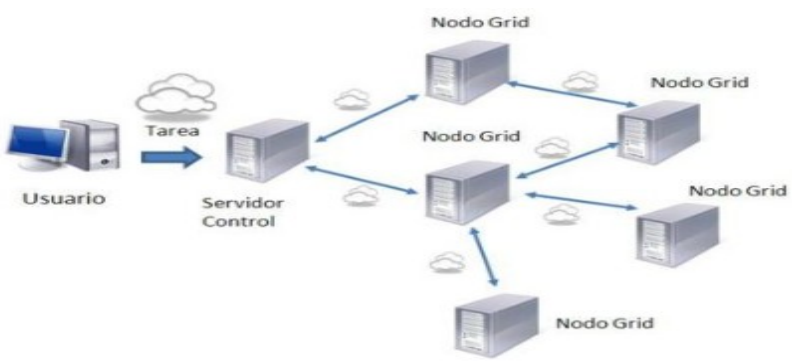
Como resumen, se pueden definir los conceptos de procesamiento concurrente y paralelo de la siguiente manera:

- **Procesamiento concurrente**. Es aquel en el que varios procesos se ejecutan en una misma unidad de proceso de manera alterna, provocando el avance simultaneo de los mismos evitando la secuencialidad.
- **Procesamiento paralelo**. Es aquel en el que divisiones de un proceso se ejecutan de forma simultánea en los diversos núcleos de ejecución de un procesador o en diversos procesadores.
- **Procesamiento distribuido**. Es aquel en el que un proceso se ejecuta en unidades de computación independientes conectadas y sincronizadas.





DISTRIBUIDA



1.3. Hilos

Un programa básico esta compuesto por una serie de sentencias que se ejecutan de manera secuencial y síncrona: hasta que no se completa la ejecución de la primera de las sentencias no se comienza con la ejecución de la segunda, y así sucesivamente hasta terminar la ejecución del programa completo.

En muchos casos, es necesaria la secuencialidad y sincronía. En otros casos en cambio, un algoritmo podría trocearse en varias unidades pequeñas , ejecutar cada una por separado, juntar los resultados sin que importe el orden en el que se obtengan y generara el resultado final. Esta técnica se conoce como programación multihilo.

Los hilos de ejecución son fracciones de programa que, si cumplen con determinadas características, pueden ejecutarse simultáneamente gracias al procesamiento paralelo.

Los programas que se ejecutan en un único hilo se denominan programas monohilo, mientras que los que se ejecutan en varios hilos se conocen como programas multihilo.

2. PROCESOS: conceptos teóricos

Se puede definir un proceso como un programa en ejecución. Algunos ejemplos podrían ser las instancias de un navegador web, de un procesador de textos, de un entorno de desarrollo o de una maquina virtual de Java.

Cada proceso está compuesto por:

- Las instrucciones que se van a ejecutar.
- El estado del propio proceso.
- El estado de la ejecución, principalmente recogido en los registros del procesador.
- El estado de la memoria.

Los procesos están constantemente entrando y saliendo del procesador. Se denomina contexto a toda la información que determina el estado de un proceso en un instante dado.
Sacar un proceso del procesador para meter a otro se conoce como cambio de contexto. Esta acción implica capturar el estado de la CPU y de sus registros, de la memoria y de la propia ejecución del proceso saliente para restaurar la información equivalente del proceso entrante y poder continuar en el punto exacto en el que realizo el cambio de contexto anterior. Los cambios de contexto en si mismos implican un consumo de recursos importante.

Los pasos que se realizan para un cambio de contexto:

- Guardar el estado del proceso actual.
- Determinar el siguiente proceso que se va a ejecutar.
- Recuperar y restaurar el estado del siguiente proceso.
- Continuar con la ejecución del siguiente proceso.

El SO es el encargado de la gestión de los procesos, quedando en la responsabilidad del programador el crear los programas que van a dar lugar a los procesos y en manos de los usuarios ejecutarlos.

2.1. **Gestión y estados de los procesos**.

En la actualidad prácticamente todos los sistemas de computación(ordenares, tablets, relojes inteligentes, móviles, videoconsalas, etc..) son multiproceso, por lo que tienen la capacidad de mantener en ejecución simultáneamente varios programas o procesos.
Los procesos necesitan recursos y estos son limitados. El procesador, la memoria, el acceso a los sistemas de almacenamiento son algunos de ellos.

Entonces, ¿Cómo se consigue la correcta convivencia entre los distintos procesos que compiten entre si por estos recursos?. La respuesta esta en el sistema operativo y , mas concretamente, en el planificador de procesos.
El planificador es uno de los componentes fundamentales del SO, ya que determina la calidad del multiproceso del sistema, y como consecuencia, la eficiencia en el aprovechamiento de los recursos.

Los objetivos del planificador son los siguientes:

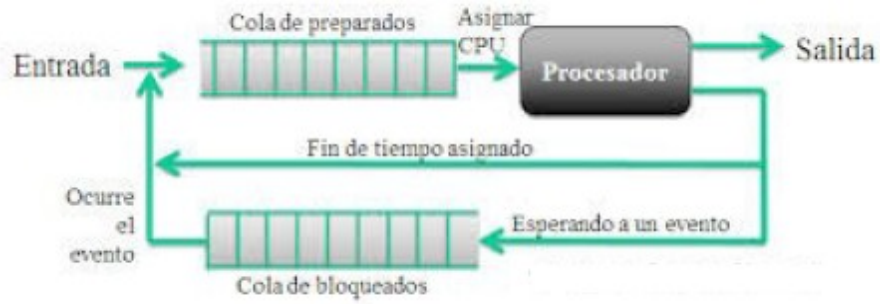
- Maximizar el rendimiento del sistema.
- Maximizar la equidad en el reparto de los recursos.
- Minimizar los tiempos en espera.
- Minimizar los tiempos de respuesta.

Se puede sintetizar que el objetivo del planificador es conseguir que todos los procesos terminen lo antes posible aprovechando al máximo los recursos del sistema. La tarea, como se puede suponer, es compleja.

Cada SO utiliza sus propias estrategias de gestión de recursos, recogidos en los algoritmos de planificación(FIFO,LIFO,Round Robin, Short Job First, etc..)

Para realizar la gestión de los procesos, el planificador debe conocer el estado en que se encuentran. En un momento dado, un proceso se encuentra en un estado de un conjunto de estados posibles.

En el siguiente gráfico, podemos ver un esquema muy simple de cómo podemos planificar la ejecución de varios procesos en una CPU.



En este esquema, podemos ver:
Los procesos nuevos, entran en la cola de procesos activos en el sistema.
Los procesos van avanzando posiciones en la cola de procesos activos, hasta que les toca el turno para que el planificador les conceda el uso de la CPU.
El planificador concede el uso de la CPU, a cada proceso durante un tiempo determinado y equitativo, que llamaremos quantum. Un proceso que consume su quantum, es pausado y enviado al final de la cola.
Si un proceso finaliza, sale del sistema de gestión de procesos.

Esta planificación que hemos descrito, resulta equitativa para todos los procesos (todos van a ir teniendo su quantum de ejecución). Pero se nos olvidan algunas situaciones y características de nuestros los procesos:

Cuando un proceso, necesita datos de un archivo o una entrada de datos que deba suministrar el usuario; o, tiene que imprimir o grabar datos; cosa que llamamos 'el proceso está en una operación de entrada/salida' (E/S para abreviar). El proceso, queda bloqueado hasta que haya finalizado esa E/S. El proceso es bloqueado, porque, los dispositivos son mucho más lentos que la CPU, por lo que, mientras que uno de ellos está esperando una E/S, otros procesos pueden pasar a la CPU y ejecutar sus instrucciones.

Cuando termina la E/S que tenga un proceso bloqueado, el planificador, volverá a pasar al proceso a la cola de procesos activos, para que recoja los datos y continúe con su tarea (dentro de sus correspondientes turnos).
Todo proceso en ejecución, tiene que estar cargado en la RAM física del equipo o memoria principal.
Hay procesos en el equipo cuya ejecución es crítica para el sistema, por lo que, no siempre pueden estar esperando a que les llegue su turno de ejecución, haciendo cola.

Con todo lo anterior, podemos quedarnos con los siguientes estados en el ciclo de vida de un proceso:

- **Nuevo**. Proceso nuevo, creado, técnicamente no se considera un estado como tal.
- **Listo**. El proceso está en memoria. Proceso que está esperando la CPU para ejecutar sus instrucciones.
- **En ejecución**. Proceso que actualmente, está en turno de ejecución en la CPU.
- **Bloqueado**. Proceso que está a la espera de que ocurra un evento externo ajeno al planificador.
- **Terminado**. Proceso que ha finalizado y ya no necesitará más la CPU.

El siguiente gráfico, nos muestra las distintas transiciones que se producen entre uno u otro estado:

Ni el programador ni el usuario tiene control directo sobre los estados de un proceso, ya que el ciclo de vida es responsabilidad exclusiva del planificador.

2.2. Comunicación entre procesos.

Por definición, los procesos de un sistema son elementos estancos. Cada uno tiene su espacio de memoria, su tiempo de CPU asignado por el planificador y su estado de los registros. No obstante, los procesos deben poder comunicarse entre si, ya surgen dependencias entre ellos en lo referente a entradas y salidas de datos.

La comunicación entre procesos se denomina IPC(Inter-Process Communication) y existen diversas alternativas para llevarla a cabo:

- Utilización de sockets. Los sockets son mecanismos de comunicación de bajo nivel.
- Utilización de flujos de entrada y salida. Los procesos pueden interceptar los flujos de entrada y salida estándar, por lo que pueden leer y escribir información unos en otros.
- RPC. Llamada a procedimiento remoto. Consiste en realizar llamadas a métodos de otros procesos que, potencialmente pueden estar ejecutándose en otras maquinas. En Java este tipo de llamada se realiza mediante tecnología llamada RMI.
- Mediante el uso de sistemas de persistencia. Consiste en realizar escrituras y lecturas desde los distintos procesos en cualquier tipo de sistema de persistencia, como los ficheros o las bases de datos.

2.3. Sincronización entre procesos

Todos los sistemas en los que participan múltiples actores de manera concurrente están sometidos a ciertas condiciones que exigen que exista sincronización entre ellos. Por ejemplo, puede que sea necesario saber si un proceso ha terminado satisfactoriamente para ejecutar el siguiente que se

encuentra en un flujo de procesos o, en caso de que haya ocurrido un determinado error, ejecutar otro proceso alternativo.

Es el planificador del sistema operativo el encargado de decidir en qué momento tiene acceso a los recursos un proceso, pero a nivel general, la decisión de crear y lanzar un proceso es humana expresada a través de un algoritmo.

Para gestionar un flujo de trabajo de procesos en el que la ejecución de de sus integrantes depende del resultado de otros procesos se necesita disponer de los siguientes mecanismos:

- Ejecución: Un mecanismo para ejecutar procesos desde un proceso.
- Espera: Un mecanismo puede bloquear la ejecución de un proceso a la espera de que otro termine.
- Generación de código de terminación: Un mecanismo de comunicación que permita indicar a un proceso como ha terminado la ejecución mediante un código.
- Obtención del código de terminación: Un mecanismo que permita a un proceso obtener el código de terminación de otro proceso.

Clases y métodos de Java que proporcionan soporte a la gestión de los procesos

Mecanismo	Clase	Método
Ejecución	Runtime	exec()
Ejecución	ProcessBuilder	start()
Espera	Process	waitFor()
Generación de código de terminación	System	exit(valor_del _retorno)
Obtención de código de terminación	Process	waitFor()

Ejemplo ProcessBuilder para lanzar el notepad:

```
ProcessBuilder build = new ProcessBuilder("notepad.exe");
```

3. PROGRAMACION DE APLICACIONES MULTIPROCESO EN JAVA

En Java, la creación de un proceso se puede realizar de dos maneras diferentes:

- Utilizando la clase java.lang.Runtime
- Utilizando la clase java.lang.ProcessBuilder

3.1. Creación de procesos con Runtime

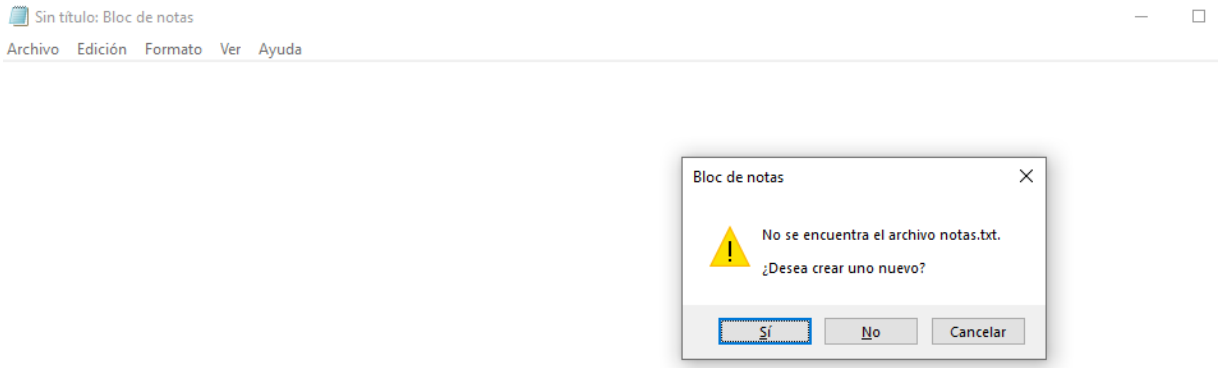
Toda aplicación Java tiene una única instancia de la clase Runtime que permite que la propia aplicación interactúe con su entorno de ejecución a través del método estático getRuntime. Este método proporciona un canal de comunicación entre la aplicación y su entorno, posibilitando la interacción con el sistema operativo a través del método exec.

Ejemplo Runtime para lanzar el notepad:

```
Runtime.getRuntime().exec("Notepad.exe");
```

Se pueden indicar parámetros a través del método exec, ya que puede recibir una cadena de caracteres y en dicha cadena, separadas por espacios se diferencian los parámetros.

```
Runtime.getRuntime().exec("Notepad.exe notas.txt");
```



Podemos crear el proceso proporcionando un array de objetos String con el nombre del programa y los parámetros.

```
String [] infoproceso={"Notepad.exe","notas.txt"};
Runtime.getRuntime().exec(infoproceso);
```

El siguiente nivel consiste en gestionar el proceso lanzado. Para ello, se debe obtener la referencia a la instancia de la clase Process proporcionada por el método exec. Es este objeto el que proporciona los métodos para conocer el estado de la ejecución del proceso.

```
String [] infoproceso={"Notepad.exe","notas.txt"};
Process proceso=Runtime.getRuntime().exec(infoproceso);
```

Si se necesita esperar a que el proceso ejecutado termine y conocer el estado en que ha finalizado dicha ejecución, se puede utilizar el método waitFor. Este método suspende la ejecución del programa que ha arrancado el proceso quedando a la espera de que este termine, proporcionando además el código de finalización.

```
String [] infoproceso={"Notepad.exe","notas.txt"};
Process proceso=Runtime.getRuntime().exec(infoproceso);
int codigoRetorno=proceso.waitFor();
System.out.println("Fin de la ejecución :"+ codigoRetorno);
```

Para los comandos de Windows que no tienen ejecutable es necesario utilizar el comando CMD.EXE(Abrir una instancia del intérprete de comandos).
Por ejemplo para hacer un DIR desde un programa de Java haríamos:

```
Runtime r=Runtime.getRuntime();
String comando="CMD /C DIR";
Process p;
p=r.exec(comando);
```

La clase Process representa al proceso en ejecución y permite obtener información sobre este.

Métodos de la clase java.lang.Process

Método	Descripción
Destroy()	Destruye el proceso sobre el que se ejecuta.
exitValue()	Devuelve el valor de retorno del proceso cuanto este finalizaa. Sirve para controlar el estado de la ejecución.
getErrorStream()	Proporciona un InputStream conectado a la salida de error del proceso.
getInputStream()	Proporciona un InputStream conectado a la salida normal del proceso.
getOutputStream()	Proporciona un OutputStream conectado a la entrada normal del proceso.
isAlive()	Determina si el proceso está o no en ejecución.
waitFor()	Detiene la ejecución del programa que lanza el proceso a la espera de que este último termine.

La salida del DIR no la obtenemos ya que se redirige a nuestro programa Java y no a la pantalla. Para obtenerla utilizaríamos el método getInputStream() de la clase Proccess.

3.2. Creación de procesos con ProcessBuilder

La clase ProcessBuilder permite, al igual que Runtime, crear procesos.
La creación mas sencilla de un proceso se realiza con un único parámetro en el que se indica el programa a ejecutar. Es importante saber que esta construcción no supone la ejecución del proceso.

```
new ProcessBuilder("notepad.exe")
```

La ejecución del proceso se realiza a partir de la invocación al método start:

```
new ProcessBuilder("notepad.exe").start();
```

El constructor de ProcessBuilder admite parámetros que serán entregados al proceso que se crea.

```
new ProcessBuilder("notepad.exe", "datos.txt").start();
```

Al igual que ocurre con el método exec de la clase Runtime, el método start de ProcessBuilder proporciona un proceso como retorno, lo que posibilita la sincronización y gestión de este.

```
Process proceso= new ProcessBuilder("notepad.exe", "datos.txt").start();
int valor= proceso.waitFor();
System.out.println("Valor retorno:" + valor);
```

El método start permite crear múltiples subprocesos a partir de una única instancia de ProcessBuilder. El siguiente código crea cuatro instancias del bloc de notas:

```
ProcessBuilder pBuilder=   new ProcessBuilder("notepad.exe");
      for (int i = 0; i <4 ; i++) {
          pBuilder.start();
      }
```

Además del método start, la clase ProcessBuilder dispone de métodos para consultar y gestionar algunos parámetros relativos a la ejecución del proceso.

Métodos de la clase java.lang.ProcessBuilder

Método	Descripción
start()	Inicia un nuevo proceso con los atributos indicados.
command()	Permite obtener o asignar el programa y los argumentos de la

	instancia de ProcessBuilder
directory()	Permite obtener o asignar el directorio de trabajo del proceso.
environment()	Proporciona información sobre el entorno de ejecución del proceso.
redirectError()	Permite determinar el destino de la salida de errores.
redirectInput()	Permite determinar el origen de la entrada estándar.
redirectOutput()	Permite determinar el destino de la salida estándar.

Ejemplo: de utilización de acceso a la información del entorno de ejecución, el método enviroment devuelve un objeto Map con la información proporcionada por el sistema operativo:

```
ProcessBuilder pB=new ProcessBuilder("notepad.exe");
java.util.Map<String,String> env=pB.environment();
 System.out.println("Num procesadores:"+ env.get("NUMBER_OF_PROCES-
SORS"));
```