IES SERRA PERENXISA

Unidad didáctica 1
Programación de procesos

Curso 2024/2025
“Programación de servicios y procesos”
CFGS Desarrollo de Aplicaciones Multiplataforma

Basado en los materiales formativos de FP propiedad del Ministerio de Educación, Cultura y Deporte.
Adaptado por: José Miguel Blázquez – Versión: 1.3

Índice

1. Introducción: aplicaciones, ejecutables y procesos

A simple vista, parece que con los términos aplicación, ejecutable y proceso, nos estamos refiriendo a lo mismo. Pero hay sustanciales diferencias entre ellos y debemos tenerlas claras.

*Una **aplicación** es un tipo de programa informático, diseñado como herramienta para resolver de manera automática un problema específico del usuario.*

Debemos darnos cuenta de que sobre el hardware del equipo, todo lo que se ejecuta son programas informáticos, que, ya sabemos, que se llama software. Con la definición de aplicación anterior, buscamos diferenciar las aplicaciones, de otro tipo de programas informáticos, como pueden ser: los sistemas operativos, las utilidades para el mantenimiento del sistema, o las herramientas para el desarrollo de software. Por lo tanto, son aplicaciones, aquellos programas que nos permiten editar una imagen, enviar un correo electrónico, navegar en Internet, editar un documento de texto, chatear, etc.

Recordemos, que un **programa** es el conjunto de instrucciones que ejecutadas en un ordenador realizarán una tarea o ayudarán al usuario a realizarla. Nosotros, como programadores y programadoras, creamos un programa, escribiendo su código fuente; con ayuda de un compilador, obtenemos su código binario o interpretado. Este código binario o interpretado, lo guardamos en un fichero. Este fichero, es un fichero ejecutable, llamado comúnmente: ejecutable o binario.

*Un **ejecutable** es un fichero que contiene el código binario o interpretado que será ejecutado en un ordenador.*

Ya tenemos más clara la diferencia entre aplicación y ejecutable. Ahora, ¿qué es un proceso?

*De forma sencilla, un **proceso** es un programa en ejecución.*

Pero, es más que eso, un proceso en el sistema operativo (SO) es una unidad de trabajo completa; y el SO gestiona los distintos procesos que se encuentren en ejecución en el equipo. En siguientes apartados de esta unidad trataremos más en profundidad todo lo relacionado con los procesos y el SO. Lo más importante, es que diferenciemos que un ejecutable es un fichero y un proceso es una entidad activa, el contenido del ejecutable, ejecutándose.

Un **proceso** existe mientras que se esté ejecutando una aplicación. Es más, la ejecución de una aplicación puede implicar que se arranquen varios procesos en nuestro equipo; y puede estar formada por varios ejecutables y librerías.

*Completando, ahora, la definición inicial que dimos de **aplicación,** ya sabemos que, al instalarla en el equipo, podremos ver que puede estar formada por varios **ejecutables** y librerías. Siempre que lancemos la ejecución de una aplicación se creará, al menos, un **proceso** nuevo en nuestro sistema.*

2. Gestión de procesos

Como sabemos, en nuestro equipo se están ejecutando al mismo tiempo muchos procesos. Por ejemplo, podemos estar escuchando música con nuestro reproductor multimedia favorito; al mismo tiempo estamos programando con NetBeans, e incluso tenemos el documento PDF abierto para ver los contenidos de esta unidad.

Independientemente de que el microprocesador de nuestro equipo sea más o menos moderno (con uno o varios núcleos de procesamiento), lo que nos interesa es que actualmente, nuestros SO son **multitarea**; como son, por ejemplo, Windows y GNU/Linux.

Ser **multitarea** es, precisamente, permitir que **varios procesos** puedan ejecutarse al mismo tiempo haciendo que todos ellos compartan el núcleo o núcleos del procesador. Pero, ¿cómo lo hacen? Imaginemos que nuestro equipo es como nosotros mismos cuando tenemos más de una tarea que realizar. Podemos ir realizando cada tarea una detrás de otra, o, por el contrario, ir realizando un poco de cada tarea e ir alternando entre ellas. Al final tendremos realizadas todas las

tareas, pero, a otra persona que nos esté mirando desde fuera le parecerá que, de la primera forma vamos muy lentos (y más si está esperando el resultado de una de las tareas que tenemos que realizar); sin embargo, de la segunda forma le parecerá que estamos muy ocupados, pero que poco a poco estamos haciendo lo que nos ha pedido. Pues bien, el micro es nuestro cuerpo y el SO es el encargado de decidir, por medio de la gestión de procesos, si lo hacemos todo de golpe o una tarea detrás de otra y cómo se ha de secuenciar todo.

2.1. Introducción a la gestión de procesos

En nuestros equipos ejecutamos distintas aplicaciones. Como sabemos, un microprocesador es capaz de ejecutar miles de millones de instrucciones básicas en un segundo (por ejemplo, un i7 puede llegar hasta los 3,4 GHz). Un micro, a esa velocidad, es capaz de realizar muchas tareas, y nosotros (muy lentos para él), apreciaremos que solo está ejecutando la aplicación que nosotros estamos utilizando. Al fin y al cabo, al micro lo único que le importa es ejecutar instrucciones y dar sus resultados, no tiene conocimiento de si pertenecen a uno u otro proceso, para él son instrucciones. Es el SO el encargado de decidir qué proceso debe entrar a ejecutarse o esperar. Lo veremos más adelante, pero se trata de una fila en la que cada proceso coge un número y va tomando su turno de servicio durante un periodo de tiempo en la CPU; pasado ese tiempo, vuelve a ponerse al final de la fila, esperando a que llegue de nuevo su turno.

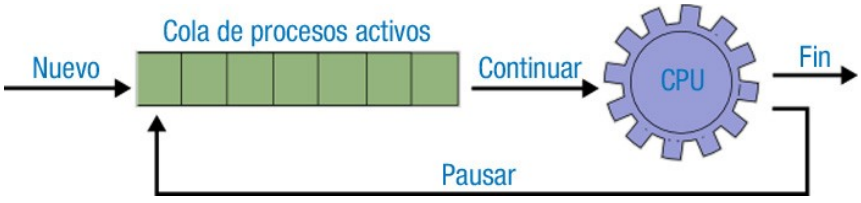
Vamos a ver cómo el SO es el encargado de gestionar los procesos, qué es realmente un programa en ejecución y qué información asocia el SO a cada proceso. También veremos qué herramientas tenemos a nuestra disposición para poder obtener información sobre los procesos que hay en ejecución en el sistema y qué uso están haciendo de los recursos del equipo.

Los nuevos micros, con varios núcleos, pueden, casi totalmente, dedicar una CPU a la ejecución de uno de los procesos activos en el sistema. Pero no nos olvidemos de que además de estar activos los procesos de usuario, también se estará ejecutando el propio SO, por lo que seguirá siendo necesario repartir los distintos núcleos entre los procesos que estén en ejecución.

2.2. Estados de un proceso

Si el sistema tiene que repartir el uso del microprocesador entre los distintos procesos, ¿qué le sucede a un proceso cuando no se está ejecutando? Y, si un proceso está esperando datos, ¿por qué el equipo hace otras cosas mientras que un proceso queda a la espera de datos?

Veamos con detenimiento cómo el SO controla la ejecución de los procesos. Ya comentamos en el apartado anterior que el SO es el encargado de la gestión de procesos. 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:

1. Los procesos nuevos entran en la cola de procesos activos en el sistema.
2. Los procesos van avanzando posiciones en la cola de procesos activos hasta que les toca el turno para que el SO les conceda el uso de la CPU.
3. El SO 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.
4. 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 procesos:

- Cuando un proceso necesita datos de un archivo o de una entrada de datos del usuario; o, tiene que imprimir o grabar datos; cosa que llamamos 'el proceso está en una operación de entrada/salida' (E/S), 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 SO 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).

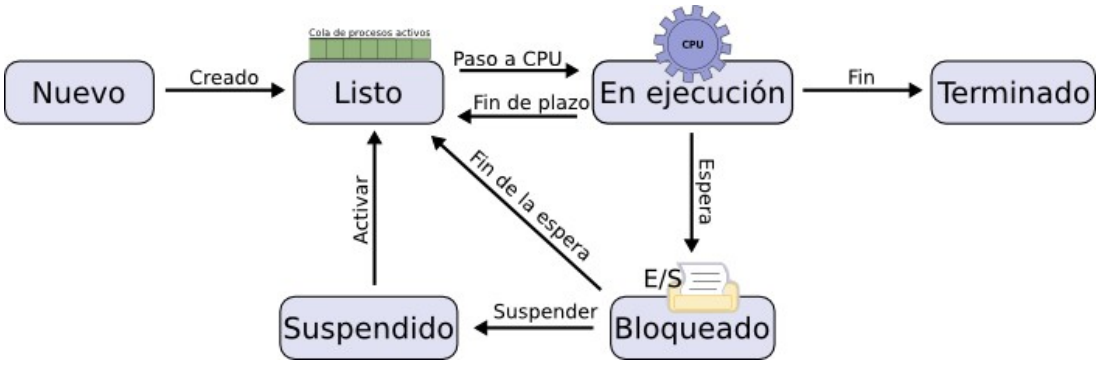
- Recordemos que, cuando la memoria RAM se llena, algunos procesos deben pasar a disco (o almacenamiento secundario) y dejar espacio en RAM para la ejecución de otros procesos. Todo proceso en ejecución tiene que estar cargado en la RAM física del equipo o memoria principal, así como todos los datos que necesite.

- 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. Por ejemplo, el propio SO es un conjunto de procesos en ejecución. Se le da prioridad, evidentemente, a los procesos del SO frente a los de usuario.

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

- ✔ **Nuevo**. Proceso nuevo, creado.
- ✔ **Listo**. 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 finalice una E/S.
- ✔ **Suspendido**. Proceso que se ha llevado a la memoria virtual para liberar, un poco, la RAM del sistema.
- ✔ **Terminado**. Proceso que ha finalizado y ya no necesitará más la CPU.

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



2.3. Planificación de procesos por el Sistema Operativo

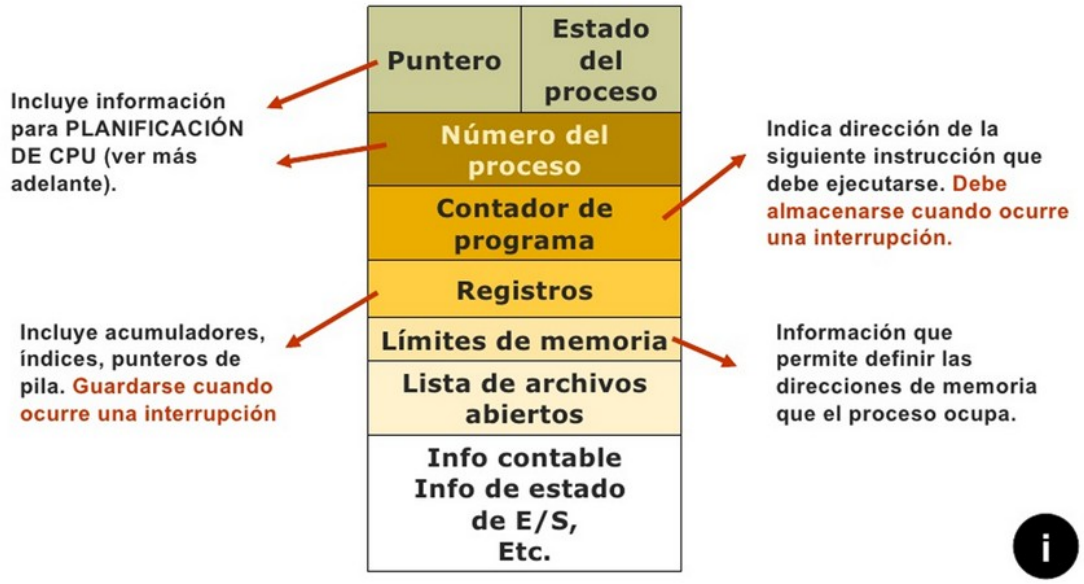
Entonces, ¿un proceso sabe cuando tiene o no la CPU? ¿Cómo se decide qué proceso debe ejecutarse en cada momento? Hemos visto que un proceso, desde su creación hasta su fin (durante su vida), pasa por muchos estados. Esa transición de estados es transparente para él, todo lo realiza el SO. Desde el punto de vista de un proceso, él siempre se está ejecutando en la CPU sin esperas. Dentro de la gestión de procesos vamos a destacar dos componentes del SO que llevan a cabo toda la tarea: el cargador y el planificador.

El **cargador** es el encargado de crear los procesos. Cuando se inicia un proceso (para cada uno), el cargador realiza las siguientes tareas:

1. **Carga el proceso en memoria principal.** Reserva un espacio en la RAM para el proceso. En ese espacio copia las instrucciones del fichero ejecutable de la aplicación, las constantes y deja un espacio para los datos (variables) y la pila (llamadas a funciones). Un proceso, durante su ejecución, no podrá hacer referencia a

direcciones que se encuentren fuera de su espacio de memoria; si lo intentara, el SO lo detectará y generará una excepción (produciendo, por ejemplo, los típicos pantallazos azules de Windows).

2. **Crea una estructura de información llamada PCB** (Bloque de Control de Proceso). La información del PCB es única para cada proceso y permite controlarlo. Esta información también la utilizará el planificador. Entre otros datos, el PCB estará formado por:
   - **Identificador** del **proceso** o **PID**. Es un número único para cada proceso, como un DNI de proceso.
   - **Estado actual** del proceso: en ejecución, listo, bloqueado, suspendido, finalizando.
   - **Espacio** de **direcciones** de **memoria** donde comienza la zona de memoria reservada al proceso y su tamaño.
   - **Información** para la **planificación**: prioridad, quamtum, estadísticas, ...
   - **Información** para el **cambio** de **contexto**: valor de los registros de la CPU, entre ellos el contador de programa y el puntero a pila. Esta información es necesaria para poder cambiar de la ejecución de un proceso a otro.
   - **Recursos utilizados**. Ficheros abiertos, conexiones, …



Una vez que el proceso ya está cargado en memoria será el **planificador** el encargado de tomar las decisiones relacionadas con la ejecución de los procesos: qué proceso se ejecuta y durante cuánto tiempo.

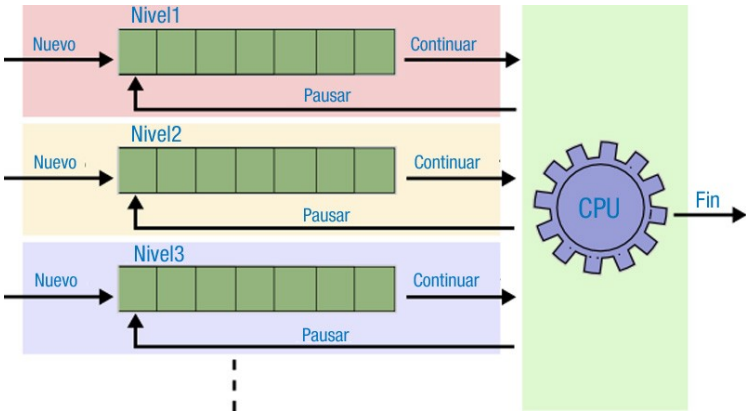
El **planificador** es otro proceso que, en este caso, es parte del SO. La política en la toma de decisiones del planificador se denomina algoritmo de planificación. Los más importantes son:

- **Round-Robin**. Este algoritmo de planificación favorece la ejecución de procesos interactivos. Es aquél en el que cada proceso puede ejecutar sus instrucciones en la CPU durante un ***quantum***. Si no le ha dado tiempo a finalizar en ese ***quantum***, se coloca al final de la cola de procesos listos y espera a que vuelva su turno de procesamiento. Así, todos los procesos en el sistema van ejecutándose poco a poco.

- Por **prioridad**. En el caso de Round-Robin todos los procesos son tratados por igual. Pero existen procesos importantes que no deberían esperar a que finalicen otros procesos de menor importancia. En este algoritmo se asignan prioridades a los distintos procesos, y la ejecución de estos se hace de acuerdo a esa prioridad asignada. Por ejemplo: el propio planificador tiene mayor prioridad en ejecución que los procesos de usuario, ¿no crees?

- **Múltiples colas de prioridad**. Es una combinación de los dos anteriores y el implementado en los sistemas operativos actuales. Todos los procesos de una misma prioridad estarán en la misma cola. Cada cola será

gestionada con el algoritmo Round-Robin. Los procesos de colas de inferior prioridad no pueden ejecutarse hasta que no se hayan vaciado las colas de procesos de mayor prioridad.



En la **planificación** (**scheduling**) de procesos se busca conciliar los siguientes objetivos:

- ✔ **Equidad**. Todos los procesos deben poder ejecutarse.
- ✔ **Eficacia**. Mantener ocupada la CPU un 100 % del tiempo.
- ✔ **Tiempo** de **respuesta**. Minimizar el tiempo de respuesta al usuario.
- ✔ **Tiempo** de **regreso**. Minimizar el tiempo que deben esperar los usuarios de procesos por lotes para obtener sus resultados.
- ✔ **Rendimiento**. Maximizar el número de tareas procesadas por hora.

Para entenderlo mejor puedes emplear un simulador de algoritmos de planificación de la CPU para **Android**, se llama: "**CPU Simulator (CPU Scheduling)**".

2.4. Cambio de contexto en la CPU

Un proceso es una unidad de trabajo completa. El sistema operativo es el encargado de gestionar los procesos en ejecución de forma eficiente, intentando evitar que haya conflictos en el uso que hacen de los distintos recursos del sistema. Para realizar esta tarea de forma correcta, se asocia a cada proceso un conjunto de información (PCB) y de unos mecanismos de protección (un espacio de direcciones de memoria del que no se puede salir y una prioridad de ejecuión).

Imaginemos que, en nuestro equipo, en un momento determinado, podemos estar escuchando música, editando un documento, al mismo tiempo, programando y navegando en Internet. En este caso, tendremos ejecutándose en el sistema cuatro aplicaciones distintas, que pueden ser: el reproductor multimedia VLC, el editor de textos writer de OpenOffice, el Visual Studio Code y el navegador Firefox. Todos ellos ejecutados sin fallos y cada uno haciendo uso de sus datos.

El sistema operativo (el **planificador**), al realizar el cambio una aplicación a otra tiene que guardar el estado en el que se encuentra el microprocesador y cargar el estado en el que estaba el microprocesador cuando cortó la ejecución de otro proceso para continuar con ese. Pero, ¿qué es el estado de la CPU?

Una CPU, además de circuitos encargados de realizar las operaciones con los datos (llamados circuitos operacionales), tiene unas pequeños espacios de memoria (llamados registros), en los que se almacenan temporalmente la información que, en cada instante, necesita la instrucción que esté procesando la CPU. El conjunto de registros de la CPU es su estado.

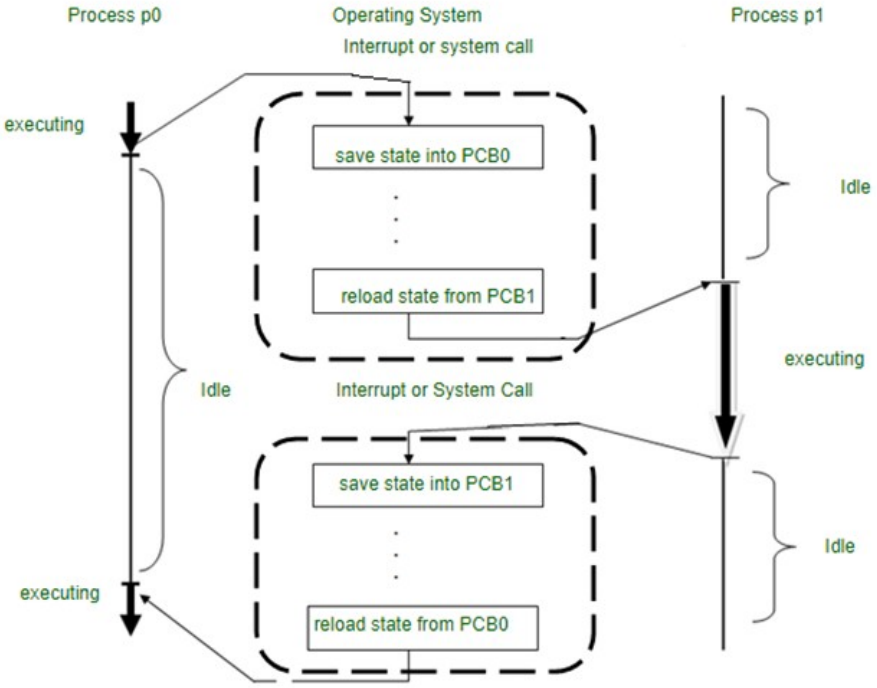
Entre los registros destacamos el Registro **Contador** de **Programa** y el **puntero** a la **pila:**

- ➢ **El Contador de Programa**, en cada instante almacena la dirección de la siguiente instrucción a ejecutar. Recordemos, que cada instrucción a ejecutar, junto con los datos que necesite, es llevada desde la memoria

principal a un registro de la CPU para que sea procesada; y, el resultado de la ejecución, dependiendo del caso, se vuelve a llevar a memoria (a la dirección que ocupe la correspondiente variable). Pues el Contador de Programa, apunta a la dirección de la siguiente instrucción que habrá que traer de la memoria, cuando se termine de procesar la instrucción en curso. Este Contador de Programa nos permitirá continuar en cada proceso por la instrucción en dónde lo hubiéramos dejado todo.

- **El Puntero a Pila**, en cada instante apunta a la parte superior de la pila del proceso en ejecución. En la pila de cada proceso es donde será almacenado el contexto de la CPU. Y de donde se recuperará cuando ese proceso vuelva a ejecutarse.

La CPU realiza un cambio de contexto cada vez que cambia la ejecución de un proceso a otro distinto. En un cambio de contexto hay que guardar el estado actual de la CPU y restaurar el estado de CPU del proceso que va a pasar a ejecutar.



2.5. Servicios e hilos

En este apartado haremos una breve introducción a los conceptos **servicio** e **hilo**, ya que los trataremos en profundidad en el resto de unidades de este módulo.

El ejemplo más claro de uso de **hilos** o **threads** es un juego. El juego es la aplicación y, mientras que nosotros controlamos uno de los personajes, los 'malos' también se mueven, interactúan por el escenario y quitan vida. Cada uno de los personajes del juego es controlado por un hilo. Todos los hilos forman parte de la misma aplicación (el juego) pero cada uno actúa siguiendo un patrón de comportamiento. El comportamiento es el algoritmo que cada uno de ellos ejecutará. Sin embargo, todos esos hilos comparten la información de la aplicación: el número de vidas restantes por personaje, la puntuación obtenida hasta ese momento, la posición en la que se encuentra el personaje del usuario y el resto de personajes, etc. Como sabemos, esas información son variables. Pues bien, un proceso no puede acceder directamente a la información de otro proceso. Pero, los hilos de un mismo proceso están dentro de él, por lo que comparten la información de las variables de ese proceso.

Realizar cambios de contexto entre hilos de un mismo proceso es más rápido y menos costoso que el cambio de contexto entre procesos, ya que sólo hay que cambiar el valor del registro contador de programa de la CPU y no todos los valores de los registros de la CPU, entre otras cosas.

*Un **proceso**, estará formado por, al menos, un hilo de ejecución.*

*Un **proceso** es una **unidad** de ejecución **pesada**. Si el proceso tiene varios hilos, cada hilo, es una **unidad** de ejecución **ligera**.*

*Un **servicio** es un proceso que queda a la espera de que otro le pida que realice una tarea. Normalmente, los servicios son cargados durante el arranque del sistema operativo.*

Por ejemplo, tenemos el servicio de impresión con su típica cola de trabajos a imprimir. Nuestra impresora imprime todo lo que recibe del sistema; pero se debe tener cuidado, ya que, si no se le envían los datos de una forma ordenada, la impresora puede mezclar las partes de un trabajo con las de otro, incluso dentro del mismo folio. El servicio de impresión es el encargado de ir enviando los datos de forma correcta a la impresora para que el resultado sea el esperado. Además, las impresoras no siempre tienen suficiente memoria para guardar todos los datos de impresión de un trabajo completo, por lo que el servicio de impresión se los dará conforme vaya necesitándolos. Cuando finalice cada trabajo puede notificárselo al usuario. Si en la cola de impresión no hay trabajos pendientes, el servicio de impresión quedará a la espera y podrá avisar a la impresora para que quede en *standby*.

Como este, hay muchos servicios activos o en ejecución en el sistema, y no todos son servicios del sistema operativo, también hay servicios de aplicación, instalados por el usuario y que pueden lanzarse al arrancar el sistema operativo o no, dependiendo de su configuración o cómo los configuremos. Ejemplos: servidor web, servidor de base de datos...

2.6. Creación de procesos

En muchos casos necesitaremos que una aplicación lance varios procesos o llame a otras aplicaciones. Esos procesos pueden realizar cada uno una tarea distinta o todos la misma. Por ejemplo, imaginemos un editor de texto plano sencillo. Estamos acostumbrados a que los distintos ficheros abiertos se muestren en pestañas independientes, pero ¿cómo implementamos eso?

Las clases que vamos a necesitar para la creación de procesos son:

- ✔ Clase **`java.lang.Process`**. Proporciona los objetos Process, por los que podremos controlar los procesos creados desde nuestro código.
- ✔ Clase **`java.lang.Runtime`**. Clase que permite lanzar la ejecución de un programa en el sistema. Sobre todos son interesantes los métodos **`exec()`** de esta clase, por ejemplo:
  - **`Runtime.exec(String comando);`** devuelve un objeto **Process** que representa al proceso en ejecución que está realizando la tarea del "comando".
- ✔ Clase **`java.lang.ProcessBuilder`**. Permite lanzar procesos del sistema operativo.

La ejecución del método **`exec()`** puede lanzar las excepciones:

- ➢ **`SecurityException`**, si hay administración de seguridad y no tenemos permitido crear subprocesos.
- ➢ **`IOException`**, si ocurre un error de E/S.
- ➢ **`NullPointerException`** e **`IllegalArgumentException`**, si "comando" es una cadena nula o vacía.

Veamos un par de ejemplos de código fuente:

```
/* Clases necesarias para poder crear procesos. */
import java.lang.Process;
import java.lang.Runtime;
/**/

    @Action
```

```
    public void crearNuevoProceso() {
        Process nuevoProceso; //Definimos una variable de tipo Process
        try{
            //Obtenemos el nombre del SO
            String osName = System.getProperty("os.name");

            if (osName.toUpperCase().contains("WIN")){ //Windows
                // modificar ruta o dará error ejecución
                nuevoProceso = Runtime.getRuntime().exec("comando o ejecutable");
            }else{ //Linux – modificar ruta o error ejecución
                nuevoProceso = Runtime.getRuntime().exec("comando o ejecutable");
            }

        }catch (SecurityException ex){
            System.out.println("Error: no hay permisos para crear el proceso.");
        }catch (Exception ex){
            System.out.println("Ha ocurrido un error, descripción: "+
                    ex.toString());
        }
}
```

Ejemplo con ProcessBuilder:

```
import java.io.IOException;

public class Proceso{
 public static void main(String[] args) throws IOException{
          ProcessBuilder pb = new ProcessBuilder("notepad");
          pb.start();
 }
}
```

Investiga las diferencias entre lanzar un proceso con ProcessBuilder.start y Runtime.exec

2.7. Comandos para la gestión de procesos

El comienzo en el mundo de los comandos puede resultar aterrador ya que hay muchísimos, ¡es imposible aprendérselos todos! Bueno, no nos alarmemos, con este par de trucos podremos defendernos:

1. El **nombre** de los comandos suele estar relacionado con la tarea que realizan, sólo que expresado en inglés o utilizando siglas. Por ejemplo: **tasklist** muestra un listado de los procesos en sistemas Windows; y en GNU/Linux obtendremos el listado de los procesos con **ps**, que son las siglas de 'process status'.

2. Su **sintaxis** siempre tiene la misma forma:

   nombreDelComando *[opciones]*

Las opciones dependen del comando en si. Podemos consultar el manual del comando antes de utilizarlo. En GNU/Linux lo podemos hacer con "**man** nombreDelComando"; y en Windows, con "nombreDelComando /?"

Ahora, los comandos que nos interesa conocer para la gestión de procesos son:

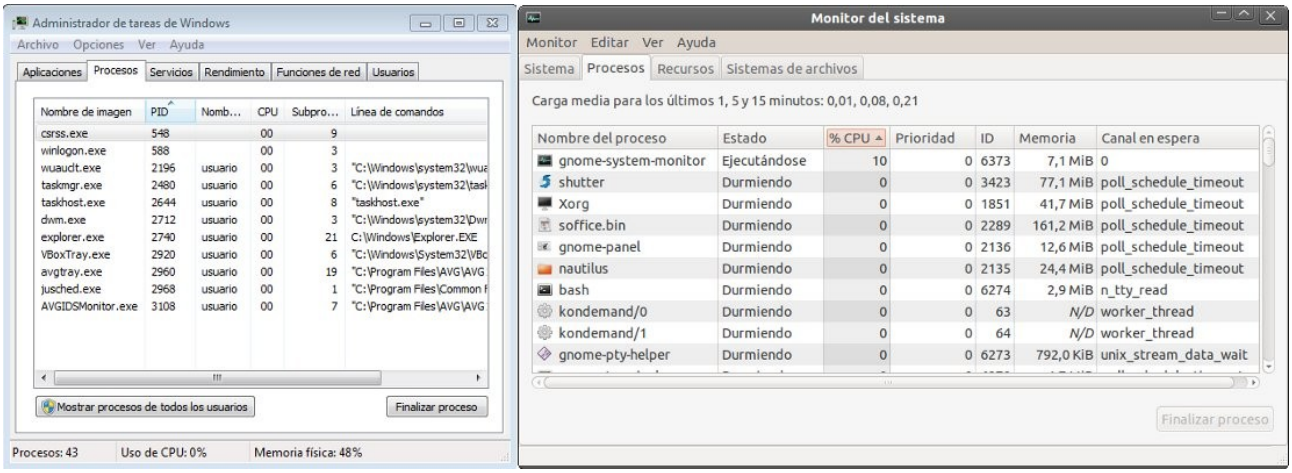
1. **Windows**. Este sistema operativo es conocido por sus interfaces gráficas, el intérprete de comandos conocido como Símbolo del sistema, no ofrece muchos comandos para la gestión de procesos. Tendremos:

   - **tasklist**. Lista los procesos presentes en el sistema. Mostrará el nombre del ejecutable; su correspondiente Identificador de proceso; y, el porcentaje de uso de memoria; entre otros datos.
   - **taskkill**. Mata procesos. Con la opción /PID especificaremos el Identificador del proceso que queremos matar.

2. **GNU/Linux**. En este sistema operativo se puede realizar cualquier tarea en modo consola, además de que los desarrolladores y desarrolladoras respetan en la implementación de las aplicaciones que sus configuraciones se guarden en archivos de texto plano. Esto es muy útil para las administradoras y administradores de sistemas.

   - **ps**. Lista los procesos presentes en el sistema. Con la opción "aux" muestra todos los procesos del sistema independientemente del usuario que los haya lanzado.
   - **pstree**. Muestra un listado de procesos en forma de árbol, mostrando qué procesos han creado otros. Con la opción "AGu" construirá el árbol utilizando líneas guía y mostrará el nombre de usuario propietario del proceso.
   - **kill**. Manda señales a los procesos. La señal -9, matará al proceso. Se utiliza "kill -9 <PID>".
   - **killall**. Mata procesos por su nombre. Se utiliza como "killall nombreDeAplicacion".
   - **nice**. Cambia la prioridad de un proceso. "nice -n 5 comando" ejecutará el comando con una prioridad 5. Por defecto la prioridad es 0. Las prioridades están entre -20 (más alta) y 19 (más baja).

También contamos con herramientas gráficas para gestionar procesos, tanto en Windows como en Linux:



3. Programación concurrente

Hasta ahora hemos programado aplicaciones secuenciales u orientadas a eventos. Siempre hemos pensado en nuestras aplicaciones como si se ejecutaran de forma aislada en la máquina. De hecho, el SO garantiza que un proceso no accede al espacio de trabajo (zona de memoria) de otro, esto es, unos procesos no pueden acceder a las variables de otros procesos. Sin embargo, los procesos, en ocasiones, necesitan comunicarse entre ellos o necesitan acceder al mismo recurso (fichero, dispositivo, etc.). En esas situaciones hay que controlar la forma en la que esos procesos se comunican o acceden a los recursos para que no haya errores, resultados incorrectos o inesperados.

Podemos ver la **concurrencia** como una carrera en la que todos los corredores corren al mismo tiempo buscando un mismo fin, que es ganar la carrera. En el caso de los procesos, competirán por conseguir los recursos que necesiten.

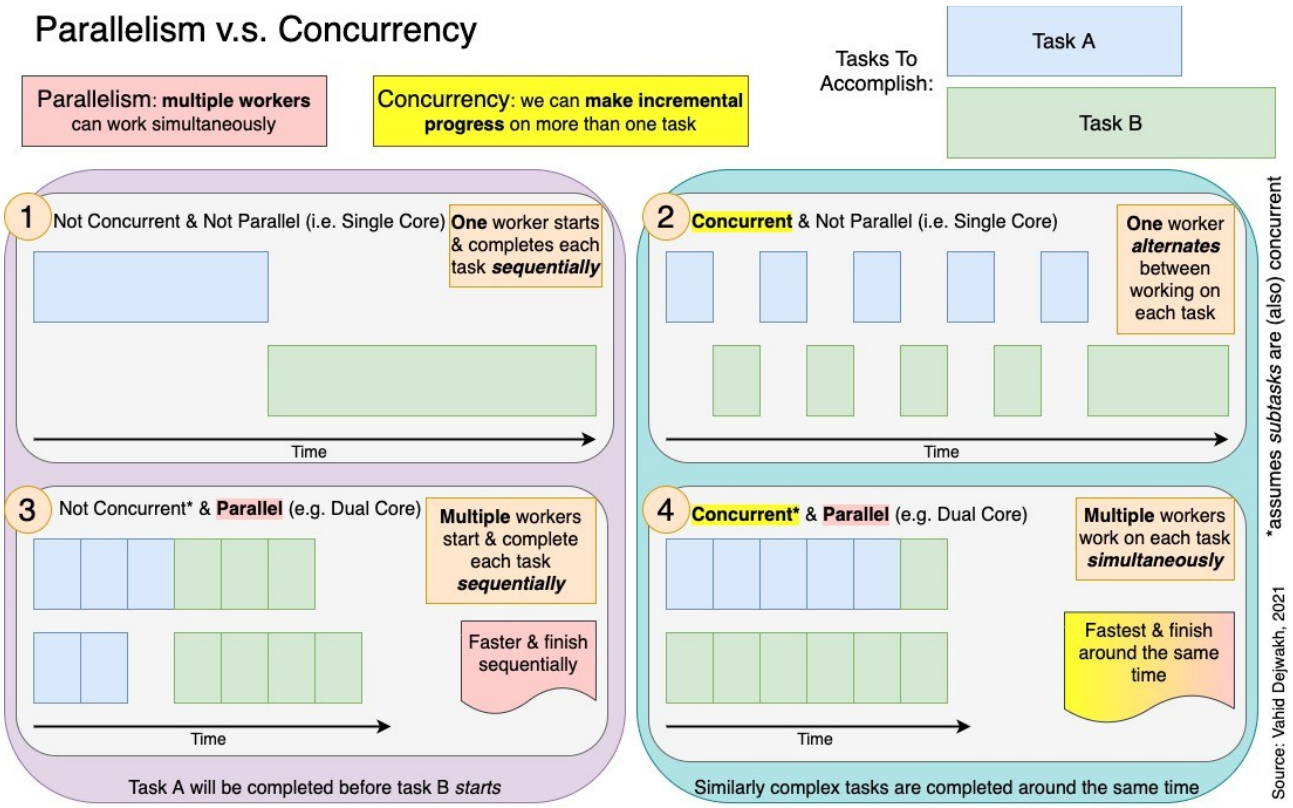
La definición de **concurrencia** no es algo sencillo. En el diccionario, concurrencia es la coincidencia de varios sucesos al mismo tiempo. Nosotros podemos decir que dos procesos son concurrentes cuando la primera instrucción de un proceso se ejecuta después de la primera y antes de la última de otro proceso.

Por otro lado, hemos visto que los procesos activos se ejecutan alternando sus instantes de ejecución en la CPU. Y, aunque nuestro equipo tenga más de un núcleo, los tiempos de ejecución de cada núcleo se repartirán entre los distintos procesos en ejecución. La planificación alternando los instantes de ejecución en la gestión de los procesos hace que los procesos se ejecuten de forma concurrente. O lo que es lo mismo: **multiproceso** = **concurrencia**.

*La **programación concurrente** proporciona mecanismos de comunicación y sincronización entre procesos que se ejecutan de forma simultanea en un sistema informático. La programación concurrente nos permitirá definir qué instrucciones de nuestros procesos se pueden ejecutar de forma simultánea con las de otros procesos, sin que se*

produzcan errores; y cuáles deben ser sincronizadas con las de otros procesos para que los resultados de sean correctos.

En el resto de la unidad pondremos especial cuidado en estudiar cómo solucionar los conflictos que pueden surgir cuando dos o más procesos intentan acceder al mismo recurso de forma concurrente. Veamos una imagen que nos va a permitir entender gráficamente la concurrencia, y, también, introducir el concepto de paralelismo que definiremos más adelante.



De forma genérica llamaremos a los procesos que se ejecuten a la vez, ya sea de forma real o simulada, **procesos concurrentes.** La ejecución de procesos concurrentes, aunque sea de forma simulada, aumenta el rendimiento del sistema informático ya que aprovecha más el tiempo del procesador. Es el sistema operativo el encargado de gestionar la ejecución concurrente de diferentes procesos contra un mismo procesador ya menudo nos referimos a él con el nombre de **multiprogramación**.

3.1. Importancia de la concurrencia

Las **principales razones** por las que se utiliza una estructura **concurrente** son:

1. **Optimizar** la utilización de los recursos. Podremos simultanear las operaciones de E/S en los procesos, y, así, la CPU estará menos tiempo ociosa. Un equipo informático es como una cadena de producción, obtenemos más productividad realizando las tareas concurrentemente.
2. Proporcionar **interactividad** a los usuarios (y animación gráfica). Todos nos hemos desesperado esperando que nuestro equipo finalizara una tarea. Esto se agravaría sino existiera el multiprocesamiento, sólo podríamos ejecutar procesos por lotes.
3. Mejorar la **disponibilidad**. Un servidor que no realice tareas de forma concurrente no podrá atender peticiones de clientes simultáneamente.
4. Conseguir un **diseño** conceptualmente más comprensible y mantenible. El diseño concurrente de un programa nos llevará a una mayor **modularidad** y claridad. Se diseña una solución para cada tarea que tenga que realizar la aplicación (no todo mezclado en el mismo algoritmo). Cada proceso se activará cuando sea necesario realizar cada tarea.

5. Aumentar la **protección**. Tener cada tarea aislada en un proceso permitirá depurar la seguridad de cada proceso y, poder finalizarlo en caso de mal funcionamiento sin que suponga la caída del sistema.

Los anteriores pueden parecer los motivos para utilizar concurrencia en sistemas con un solo procesador. Los actuales avances tecnológicos hacen necesario tener en cuenta la concurrencia en el diseño de las aplicaciones para aprovechar su potencial. Los nuevos **entornos hardware** son:

- **Microprocesadores** con **múltiples núcleos** que comparten la memoria principal del sistema.
- **Entornos multiprocesador** con **memoria compartida**. Todos los procesadores utilizan un mismo espacio de direcciones a memoria, sin tener conciencia de dónde están instalados físicamente los módulos de memoria.
- **Entornos distribuidos**. Conjunto de equipos heterogéneos o no, conectados por red y/o Internet.

Los **beneficios** que obtendremos al adoptar un modelo de programa concurrente son:

- Estructurar un programa como conjunto de procesos concurrentes que interactúan, aporta gran claridad sobre lo que cada proceso debe hacer y cuando debe hacerlo.
- Puede conducir a una reducción del tiempo de ejecución. Cuando se trata de un entorno monoprocesador, permite solapar los tiempos de E/S o de acceso al disco de unos procesos con los tiempos de ejecución de CPU de otros procesos. Cuando el entorno es multiprocesador, la ejecución de los procesos es realmente simultánea en el tiempo (paralela), y esto reduce el tiempo de ejecución del programa.
- Permite una mayor flexibilidad de planificación. Procesos de alta prioridad pueden ser ejecutados antes de otros procesos menos urgentes.
- La concepción concurrente del software permite un mejor modelado previo del comportamiento del programa, y en consecuencia un análisis más fiable de las diferentes opciones que requiera su diseño.

3.2. Condiciones de competencia

Acabamos de ver que tenemos que desechar la idea de que nuestra aplicación se ejecutará de forma aislada. Y que, de una forma u otra, va a interactuar con otros procesos. Distinguimos los siguientes tipos básicos de interacción entre procesos concurrentes:

- **Independientes**. Sólo interfieren en el uso de la CPU.
- **Cooperantes**. Un proceso genera la información o proporciona un servicio que otro necesita.
- **Competidores**. Procesos que necesitan usar los mismos recursos de forma exclusiva.

En el segundo y tercer caso, necesitamos componentes que nos permitan establecer acciones de sincronización y comunicación entre los procesos.

Un proceso entra en **condición de competencia** con otro, cuando ambos necesitan el mismo recurso, ya sea forma exclusiva o no; por lo que será necesario utilizar mecanismos de sincronización y comunicación entre ellos.

Un ejemplo sencillo de procesos **cooperantes**, es "un proceso recolector y un proceso productor". El proceso recolector necesita la información que el otro proceso produce. El proceso recolector quedará bloqueado mientras que no haya información disponible.

El proceso **productor** puede escribir siempre que lo desee (es el único que produce ese tipo de información). Por supuesto, podemos complicar esto, con varios procesos recolectores para un sólo productor; y si ese productor puede dar información a todos los recolectores de forma simultánea o no; o a cuántos procesos recolectores puede dar servicio de forma concurrente para determinar si los recolectores tendrán que esperar su turno o no. Pero ya abordaremos las soluciones a estas situaciones más adelante.

En el caso de procesos **competidores**, vamos a comenzar viendo unas definiciones:

- Cuando un proceso necesita un recurso de forma exclusiva, es porque mientras que lo esté utilizando él, ningún otro puede utilizarlo. Se llama región de **exclusión mutua** o **región crítica** al conjunto de instrucciones en las

que el proceso utiliza un recurso y que se deben ejecutar de forma exclusiva con respecto a otros procesos competidores por ese mismo recurso.

- Cuando más de un proceso necesitan el mismo recurso, antes de utilizarlo tienen que pedir su uso, una vez que lo obtienen, el resto de procesos quedarán bloqueados al pedir ese mismo recurso. Se dice que un proceso hace un lock (bloqueo) sobre un recurso cuando ha obtenido su uso en exclusión mutua.

- Por ejemplo dos procesos, compiten por dos recursos distintos, y ambos necesitan ambos recursos para continuar. Se puede dar la situación en la que cada uno de los procesos bloquee uno de los recursos, lo que hará que el otro proceso no pueda obtener el recurso que le falta; quedando bloqueados un proceso por el otro sin poder finalizar. Deadlock o interbloqueo, se produce cuando los procesos no pueden obtener, nunca, los recursos necesarios para continuar su tarea. El interbloqueo es una situación muy peligrosa, ya que puede llevar al sistema a su caída o cuelgue.

4. Comunicación entre procesos

Como ya hemos comentado en más de una ocasión a lo largo de esta unidad, cada proceso tiene su espacio de direcciones privado al que no pueden acceder el resto de procesos. Esto constituye un mecanismo de seguridad; imagina qué locura si tienes un dato en tu programa y cualquier otro programa puede modificarlo de cualquier manera. Por supuesto, nos damos cuenta de que, si cada proceso tiene sus datos y otros procesos no pueden acceder a ellos directamente, cuando otro proceso los necesite, tendrá que existir alguna forma de comunicación entre ellos.

Comunicación entre procesos: un proceso da o deja información; recibe o recoge información.

Los lenguajes de programación y los sistemas operativos nos proporcionan primitivas de sincronización que facilitan la interacción entre procesos de forma sencilla y eficiente. Una primitiva hace referencia a una operación de la cual conocemos sus restricciones y efectos, pero no su implementación exacta. Veremos que usar esas primitivas se traduce en utilizar objetos y sus métodos, teniendo muy en cuenta sus repercusiones reales en el comportamiento de nuestros procesos.

Clasificaremos las interacciones entre los procesos y el resto del sistema (recursos y otros procesos), como estas tres:

- **Sincronización**: Un proceso puede conocer el punto de ejecución en el que se encuentra otro en ese determinado instante.
- **Exclusión mutua**: Mientras que un proceso accede a un recurso, ningún otro proceso accede al mismo recurso o variable compartida.
- **Sincronización condicional**: Sólo se accede a un recurso cuando se encuentra en un determinado estado interno.

Si pensamos en la forma en la que un proceso puede comunicarse con otro veremos que existen 2 mecanismos básicos de comunicación, que son:

- **Intercambio de mensajes**. Tendremos las primitivas enviar (send) y recibir (receive o wait) información.
- **Recursos (o memoria) compartidos**. Las primitivas serán escribir (write) y leer (read) datos en o de un recurso. Sólo cuando los procesos se encuentran en la misma máquina.

En el caso de comunicar procesos dentro de una misma máquina, el **intercambio de mensajes** se puede realizar de dos formas:

- Utilizar un **buffer** de memoria.
- Utilizar un **socket**.

La diferencia entre ambos está en que un **socket** se utiliza para intercambiar información entre procesos en **distintas máquinas a través de la red,** y un **buffer** de memoria crea un canal de comunicación entre dos procesos utilizando la

memoria principal del sistema. Actualmente, es más común el uso de sockets que buffers para comunicar procesos. Trataremos en profundidad los sockets en posteriores unidades.

Con respecto a las lecturas y escrituras debemos recordar que serán bloqueantes. Es decir, un proceso quedará bloqueado hasta que los datos estén listos para poder ser leídos. Una escritura bloqueará al proceso que intenta escribir hasta que el recurso esté preparado para poder escribir. Aunque esto está relacionado con el acceso a **recursos compartidos** cosa que estudiaremos en profundidad en el apartado de **Regiones Críticas**.

5. Sincronización entre procesos

En las situaciones en las que 2 o más procesos tengan que comunicarse, cooperar o utilizar un mismo recurso; implicará que deba haber cierto **sincronismo** entre ellos. O bien, unos tienen que esperar que otros finalicen alguna acción; o, tienen que realizar alguna tarea al mismo tiempo.

En este apartado veremos distintas problemáticas, primitivas y soluciones de sincronización necesarias para resolverlas. También es cierto que el sincronismo entre procesos lo hace posible el SO, y lo que hacen los lenguajes de programación de alto nivel es encapsular los mecanismos de sincronismo que proporciona cada SO en objetos, métodos y funciones. Los lenguajes de programación proporcionan primitivas de sincronismo entre los distintos hilos que tenga un proceso; estas primitivas del lenguaje las veremos en la siguiente unidad.

5.1. Regiones críticas

Una **región crítica** es el conjunto de instrucciones en las que un proceso accede a un recurso compartido. Para que la definición sea correcta, añadiremos que, las instrucciones que forman esa región crítica, se ejecutarán de forma indivisible o atómica y de forma exclusiva con respecto a otros procesos que accedan al mismo recurso compartido al que se está accediendo.

Al identificar y definir nuestras regiones críticas en el código tendremos en cuenta:

- Se protegerán con secciones críticas sólo aquellas instrucciones que acceden a un recurso compartido.
- Las instrucciones que forman una sección crítica serán las mínimas. Incluirán sólo las instrucciones imprescindibles que deban ser ejecutadas de forma atómica.
- Se pueden definir tantas secciones críticas como sean necesarias.
- Un único proceso entra en su sección crítica. El resto de procesos esperan a que éste salga de su sección crítica, porque encontrarán el recurso bloqueado. El proceso que está en su sección crítica, es el que ha bloqueado el recurso.
- Al final de cada sección crítica el recurso debe ser liberado para que puedan utilizarlo otros procesos.

Algunos lenguajes de programación permiten definir bloques de código como secciones críticas. Estos lenguajes cuentan con palabras reservadas específicas para la definición de estas regiones. En Java veremos cómo definir este tipo de regiones a nivel de hilo en posteriores unidades. Los algoritmos que controlan esto, es decir, que evitan el acceso a una región crítica por más de un hilo o proceso y que garantizan que únicamente un proceso estará haciendo uso de este recurso y el resto que quieren utilizarlo estarán a la espera de que sea liberado, se llaman algoritmos de exclusión mutua.

***Exclusión mutua** (**MUTEX**, mutual exclusion en inglés) es el tipo de sincronización que impide que dos procesos ejecuten simultáneamente una misma sección crítica.*

5.2. Categoría de procesos cliente-suministrador

Vamos introducir los procesos que podremos clasificar dentro de la categoría **cliente-suministrador**.

- **Cliente**. Es un proceso que requiere o solicita información o servicios que proporciona otro proceso.

- **Suministrador**. Probablemente, te suene más el término servidor; pero, no queremos confundirnos con el concepto de servidor en el que profundizaremos en próximas unidades. Suministrador hace referencia a un concepto de proceso más amplio; un suministrador "proporciona" información o servicios; ya sea a través memoria compartida, un fichero, red, o cualquier otro recurso.

- **Información o servicio es perecedero.** La información desaparece cuando es consumida por el cliente; y, el servicio es prestado en el momento en el que cliente y suministrador están sincronizados.

Entre un **cliente** y un **suministrador** (ojo, empecemos con un proceso de cada) se establece sincronismo entre ellos por medio de intercambio de mensajes o a través de un recurso compartido. Entre un cliente y un servidor, la comunicación se establece de acuerdo a un conjunto de mensajes a intercambiar con sus correspondientes reglas de uso; llamado **protocolo**. Podremos implementar nuestros propios protocolos, o, protocolos existentes (ftp, http, telnet, smtp, pop3, …); pero aún tenemos que ver algunos conceptos más antes de implementar protocolos.

Entre procesos **cliente** y **suministrador** debemos disponer de mecanismos de **sincronización** que permitan que:

- Un **cliente** no debe poder leer un dato hasta que no haya sido completamente suministrado. Así nos aseguraremos de que el dato leído es correcto y consistente.

- Un **suministrador** irá produciendo su información, que en cada instante, no podrá superar un volumen de tamaño máximo establecido; por lo que el suministrador, no debe poder escribir un dato si se ha alcanzado ese máximo. Esto es así, para no desbordar al cliente.

Lo más sencillo es pensar que el suministrador sólo produce un dato que el cliente tiene que consumir. ¿Qué sincronismo hace falta en esta situación?

- El cliente tiene que esperar a que el suministrador haya generado el dato.

- El suministrador genera el dato y de alguna forma avisa al cliente de que puede consumirlo.

Podemos pensar en dar una solución a esta situación con programación secuencial. Incluyendo un bucle en el cliente en el que esté testeando el valor de una variable que indica que el dato ha sido producido. Ese bucle hace que esta solución sea poco eficiente, ya que el proceso cliente estaría consumiendo tiempo de CPU sin realizar una tarea productiva; lo que conocemos como **espera activa**. Además, si el proceso suministrador quedara bloqueado por alguna razón, ello también bloquearía al proceso cliente.

En los próximos apartados, vamos a centrarnos en los mecanismos de programación concurrente que nos permiten resolver estos problemas de sincronización entre procesos de forma eficiente, llamados primitivas de programación concurrente: **semáforos** y **monitores**; y son estas primitivas las que utilizaremos para proteger las **secciones críticas** de nuestros procesos.

5.3. Semáforos

Veamos una primera solución eficiente a los problemas de sincronismo entre procesos que acceden a un mismo recurso compartido.

Podemos entender varios procesos que quieren acceder al mismo recurso como si fueran coches que necesitan pasar por un cruce de calles. En nuestros cruces, los semáforos nos indican cuándo podemos pasar y cuándo no. Nosotros, antes de intentar entrar en el cruce, primero miramos el color en el que se encuentra el semáforo, y si está en verde (abierto), pasamos. Si el color es rojo (cerrado) quedamos a la espera de que ese mismo semáforo nos indique que podemos pasar. Este mismo funcionamiento es el que van a seguir nuestros semáforos en programación concurrente. Y son una solución eficiente porque los procesos quedarán bloqueados (y no en espera activa) cuando no puedan acceder al recurso, y será el semáforo el que vaya desbloqueándolos cuando puedan pasar.

"*Un **semáforo** es un componente de bajo nivel de abstracción que permite arbitrar los accesos a un recurso compartido en un entorno de programación concurrente.*"

Ventajas: son fáciles de comprender y proporcionan una gran capacidad funcional (podemos utilizarlos para resolver cualquier problema de concurrencia).

Inconvenientes: Su nivel bajo de abstracción los hace peligrosos de manejar y, a menudo, son la causa de muchos errores como es el interbloqueo. Un simple olvido o cambio de orden conduce a bloqueos; y requieren que la gestión de un semáforo se distribuya por todo el código lo que hace que la depuración de los errores en su gestión sea muy difícil. Es decir, su correcto uso recae en la responsabilidad del programador.

En java, encontramos la clase **Semaphore** dentro del paquete **java.util.concurrent**; y su uso real se aplica a los hilos de un mismo proceso para arbitrar el acceso de esos hilos semafde forma concurrente a una misma región de la memoria del proceso, como por ejemplo una variable compartida.

5.4. Monitores

Los monitores nos ayudan a resolver los inconvenientes del uso de semáforos. El problema en el uso de semáforos es que, recae sobre el programador o programadora la tarea implementar el correcto uso de cada semáforo para la protección de cada recurso compartido; y, sigue estando disponible el recurso para utilizarlo sin la protección de un semáforo.

Los monitores son como guardaespaldas encargados de la protección de uno o varios recursos específico,; pero encierran esos recursos de forma que el proceso sólo puede acceder a ellos a través de los métodos que el monitor expone. Viene a ser un concepto similar a los getters y setters para las variables protegidas en orientación a objetos.

*"Un **monitor** es un componente de alto nivel de abstracción destinado a gestionar recursos que van a ser accedidos de forma concurrente."*

Los monitores encierran en su interior los recursos o variables compartidas como componentes privadas y garantizan el acceso a ellas en exclusión mutua.

En **Java no existen los monitores como una clase especial** sino que cualquier clase que definamos se puede comportar como un monitor encapsulando en su interior los datos a proteger. Para ello, emplearemos la palabra reservada "***synchronized***" al definir métodos públicos de dicha clase. Además, debemos usar los métodos "wait()", "notifiy()" y "notifyAll()".

Las **ventajas** que proporciona el uso de **monitores** son:

- **Uniformidad**: El monitor provee una única capacidad, la exclusión mutua, no existe la confusión de los semáforos.
- **Modularidad**: El código que se ejecuta en exclusión mutua está separado, no mezclado con el resto del programa.
- **Simplicidad**: El programador o programadora no necesita preocuparse de las herramientas para la exclusión mutua .
- **Eficiencia** de la implementación: La implementación subyacente puede limitarse fácilmente a los semáforos.

Y la **desventaja** es la interacción de múltiples condiciones de sincronización: Cuando el número de condiciones crece, y se hacen complicadas, la complejidad del código crece de manera extraordinaria.

5.5. Memoria compartida

Una forma natural de comunicación entre procesos es la posibilidad de disponer de zonas de memoria compartidas (variables, buffers o estructuras). Además, los mecanismos de sincronización en programación concurrente que hemos visto: regiones críticas, semáforos y monitores; tienen su razón de ser en la existencia de recursos compartidos, incluida la memoria compartida.

Cuando se crea un proceso el sistema operativo le asigna los recursos iniciales que necesita, siendo el principal recurso la zona de memoria en la que se guardarán sus instrucciones, datos y pila de ejecución. Pero, como ya hemos comentado anteriormente, los sistemas operativos modernos implementan mecanismos que permiten proteger la zona de memoria de cada proceso siendo ésta privada para cada uno de forma que otros no podrán acceder a ella.

Con esto, podemos pensar que no hay posibilidad de tener comunicación entre procesos por medio de memoria compartida. Pues no es así. En la actualidad, la programación multihilo (que abordaremos en la siguiente unidad y, se refiere, a tener varios flujos de ejecución dentro de un mismo proceso, compartiendo entre ellos la memoria asignada al proceso), nos permitirá examinar al máximo esta funcionalidad.

Pensemos ahora en problemas que pueden resultar complicados si los resolvemos con un sólo procesador, por ejemplo: la ordenación de los elementos de un array. Ordenar un array pequeño no supone mucho esfuerzo; pero si el array es muy grande, si disponemos de varios procesadores y somos capaces de partir el array en trozos (convertir un problema grande en varios más pequeños) de forma que cada procesador se encargue de ordenar cada parte, conseguiremos resolver el problema en menos tiempo; eso sí, teniendo en cuenta la complejidad de dividir el problema y asignar a cada procesador el conjunto de datos (o zona de memoria) que tiene que manejar y la tarea o proceso a realizar (y finalizar con la tarea de combinar todos los resultados para obtener la solución final).

En este caso, tenemos sistemas multiprocesador como los actuales microprocesadores de varios núcleos, o los supercomputadores formados por múltiples ordenadores completos (e idénticos) trabajando como un único sistema. En ambos casos contaremos con ayuda de sistemas específicos (sistemas operativos o entornos de programación) preparados para soportar la carga de computación en múltiples núcleos y/o equipos.

6. Requisitos: seguridad, vivacidad, eficiencia y reusabilidad

Como cualquier aplicación, los programas concurrentes deben cumplir una serie de **requisitos** de **calidad**. En este apartado veremos algunos aspectos que nos permitirán desarrollar proyectos concurrentes con, casi, la completa certeza de que estamos desarrollando software de calidad.

Todo **programa concurrente** debe satisfacer dos tipos de **propiedades**:

- ✔ **Seguridad** ("safety"): relativas a que en cada instante de la ejecución no debe haberse producido algo que haga entrar al programa en un estado erróneo:
  - Dos procesos **no deben entrar simultáneamente en una sección crítica**.
  - Se **respetan las condiciones de sincronismo**, como: el consumidor no debe consumir datos antes de que el productor los haya producido; y, el productor no debe producir datos mientras el buffer esté lleno.

- ✔ **Vivacidad** ("liveness"): cada sentencia que se ejecute conduce en algún modo a un avance constructivo para alcanzar el objetivo funcional del programa. Son, en general, muy dependientes de la política de planificación que se utilice. Ejemplos de propiedades de vivacidad son:
  - No deben producirse **bloqueos activos** (livelock). Conjuntos de procesos que ejecutan de forma continuada sentencias que no conducen a un progreso constructivo.
  - **Aplazamiento indefinido** (starvation): consiste en el estado al que puede llegar un programa que aunque potencialmente puede avanzar de forma constructiva. Esto puede suceder, como consecuencia de que no se le asigna tiempo de procesador en la política de planificación; o, porque en las condiciones de sincronización hemos establecido criterios de prioridad que perjudican siempre al mismo proceso.
  - **Interbloqueo** (deadlock): se produce cuando los procesos no pueden obtener, nunca, los recursos necesarios para finalizar su tarea. Vimos un ejemplo de esta situación en el apartado 4.2 Condiciones de competencia.

Resulta evidente que también nos preocuparemos por diseñar nuestras aplicaciones para que sean **eficientes**:

- No utilizarán más **recursos** de los **necesarios**.
- Buscaremos la **rigurosidad** en su **implementación**: toda la funcionalidad esperada de forma correcta y concreta.

Y, en cuanto a la reusabilidad, debemos tenerlo ya muy bien aprendido:

- ✔ Implementar el código de forma **modular**: definiendo clases, métodos, funciones, ...
- ✔ **Documentar** correctamente el código y el proyecto.

6.1. Arquitecturas y patrones de diseño

La **Arquitectura del Software**, también denominada arquitectura lógica, es el **diseño de más alto nivel** de la estructura de un sistema. Consiste en un **conjunto de patrones y abstracciones** coherentes con base a las cuales se pueden resolver los problemas. A semejanza de los planos de un edificio o construcción, estas indican la **estructura**, **funcionamiento** e **interacción** entre las **partes** del **software**.

La arquitectura de software, tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar un cierto número de elementos arquitectónicos de forma adecuada para **satisfacer** la **mayor funcionalidad** y **requerimientos** de desempeño de un sistema, así como requerimientos no funcionales, como la **confiabilidad, escalabilidad, portabilidad, y disponibilidad; mantenibilidad, auditabilidad, flexibilidad e interacción**.

Generalmente, no es necesario inventar una nueva arquitectura de software para cada sistema de información. Lo habitual es adoptar una arquitectura conocida en función de sus ventajas e inconvenientes para cada caso en concreto. Así, las arquitecturas más universales son:

- ✔ **Monolítica**. El software se estructura en grupos funcionales muy acoplados.
- ✔ **Cliente-servidor**. El software reparte su carga de cómputo en dos partes independientes: consumir un servicio y proporcionar un servicio.
- ✔ **Arquitectura de tres niveles**. Especialización de la arquitectura cliente-servidor donde la carga se divide en capas con un reparto claro de funciones: una capa para la presentación (interfaz de usuario), otra para el cálculo (donde se encuentra modelado el negocio) y otra para el almacenamiento (persistencia). Una capa solamente tiene relación con la siguiente.

Los patrones de diseño se definen como soluciones de diseño que son válidas en distintos contextos y que han sido aplicadas con éxito en otras ocasiones:

- ✔ **Ayudan** a "**arrancar**" en el diseño de un programa complejo.
  - Dan una descomposición de objetos inicial "bien pensada".
  - Pensados para que el programa sea escalable y fácil de mantener.
  - Otra gente los ha usado y les ha ido bien.

- ✔ **Ayudan** a **reutilizar** técnicas.
  - Mucha gente los conoce y ya sabe como aplicarlos.
  - Están en un alto nivel de abstracción.
  - El diseño se puede aplicar a diferentes situaciones.

Existen dos modelos básicos de **programas concurrentes**:

- ➢ Un programa resulta de la actividad de objetos activos que interaccionan entre si directamente o a través de recursos y servicios pasivos.
- ➢ Un programa resulta de la ejecución concurrente de tareas. Cada tarea es una unidad de trabajo abstracta y discreta que idealmente puede realizarse con independencia de las otras tareas.

No es obligatorio utilizar patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. Como podemos ver en la siguiente web, lo normal es que encontremos la definición de los patrones de diseño en UML:

`https://sourcemaking.com/design_patterns`

Aunque los diagramas UML son fáciles de entender, y los trabajaste en el módulo “Entornos de Desarrollo”; en el siguiente enlace encontrarás un tutorial sencillo: **`https://www.docirs.cl/uml.asp`**

6.2. Dificultades en la depuración

Cuando estamos programando aplicaciones que incluyen mecanismos de sincronización, y acceden a recursos de forma concurrente junto con otras aplicaciones, podemos enfrentarnos, a la hora de depurarlas, a varios problemas:

- Los mismos problemas de depuración de una aplicación secuencial.
- Además de nuevos errores de temporización y sincronización propios de la programación concurrente.

Los programas secuenciales presentan una línea simple de control de flujo. Las operaciones de este tipo de programas están estrictamente ordenados como una secuencia temporal lineal.

- ✔ El comportamiento del programa es solo función de la naturaleza de las operaciones individuales que constituye el programa y del orden en que se ejecutan.
- ✔ En los programas secuenciales, el tiempo que tarda cada operación en ejecutarse no tiene consecuencias sobre el resultado.
- ✔ Para validar un programa secuencial se necesitaremos comprobar:
  - La correcta respuesta a cada sentencia.
  - El correcto orden de ejecución de las sentencias.

Para **validar** un programa **concurrente** se requiere comprobar los **mismos aspectos** que en los programas **secuenciales**, **además** de los siguientes **nuevos aspectos**:

- ✔ Las sentencias se pueden validar individualmente solo si no están involucradas en el uso de recursos compartidos.
- ✔ Cuando existen recursos compartidos, los efectos de interferencia entre las sentencias concurrentes pueden ser muy variados y la validación es muy difícil. Comprobaremos la corrección en la definición de las regiones críticas y que se cumple la exclusión mutua.
- ✔ Al comprobar la correcta implementación del sincronismo entre aplicaciones; que es forzar la ejecución secuencial de tareas de distintos procesos, introduciendo sentencias explícitas de sincronización. Tendremos en cuenta que el tiempo no influye sobre el resultado.

El problema es que las herramientas de depuración no nos proporcionan toda la funcionalidad que quisiéramos para poder depurar nuestros programas concurrentes.

¿Con qué herramientas contamos para depurar programas concurrentes?

- El depurador del IDE NetBeans. sí está preparado para la depuración concurrente de hilos dentro de un mismo proceso. En esta unidad estamos tratando procesos independientes pero en la siguiente veremos hilos.
- Hacer volcados de actividad en un fichero de log o en pantalla de salida (nos permitirá hacernos una idea de lo que ha estado pasando durante las pruebas de depuración).
- Plugins de depuración para entornos como Visual Studio, etcétera...
- Herramientas de depuración específicas...

Una de las nuevas situaciones a las que nos enfrentamos es que, a veces, los errores que parecen estar sucediendo, pueden desaparecer cuando introducimos código para tratar de identificar el problema.

Nos damos cuenta de la complejidad que entraña depurar el comportamiento de aplicaciones concurrentes, es por ello que, al diseñarlas, tendremos en cuenta los patrones de diseño existentes que ya resuelven errores comunes de la concurrencia. Podemos verlos como 'recetas' que nos permiten resolver los problemas 'tipo' que se presentan en determinadas condiciones de sincronismo y/o en los accesos concurrentes a un recurso.

7. Programación paralela y distribuida

Dos procesos se ejecutan de forma paralela si las instrucciones de ambos se están ejecutando realmente de forma simultánea. Esto sucede en la actualidad en sistemas que poseen más de un núcleo de procesador.

La **programación paralela** y **distribuida** considera los aspectos conceptuales y físicos de la **computación paralela**; siempre con el objetivo de mejorar las prestaciones aprovechado la ejecución simultánea de tareas. Tanto en la programación paralela como en la distribuida existe ejecución simultánea de tareas que resuelven un problema común. La **diferencia** entre ambas es:

- La **programación paralela** se centra en procesadores multinúcleo (en nuestros PC y servidores); o sobre los llamados supercomputadores, fabricados con arquitecturas específicas, compuestos por gran cantidad de equipos idénticos interconectados entre sí y que cuentan son sistemas operativos propios.

- La **programación distribuida**, en sistemas formados por un conjunto de ordenadores heterogéneos interconectados entre sí, por redes de comunicaciones de propósito general: redes de área local, metropolitana; incluso, a través de Internet. Su gestión se realiza utilizando componentes, protocolos estándar y sistemas operativos de red.

En la **computación paralela** y **distribuida**:

- Cada procesador tiene asignada la tarea de resolver una porción del problema.
- En programación paralela, los procesos pueden intercambiar datos a través de direcciones de memoria compartidas o mediante una red de interconexión propia.
- En programación distribuida, el intercambio de datos y la sincronización se realizará mediante intercambio de mensajes.
- El sistema se presenta como una unidad ocultando la realidad de las partes que lo forman.

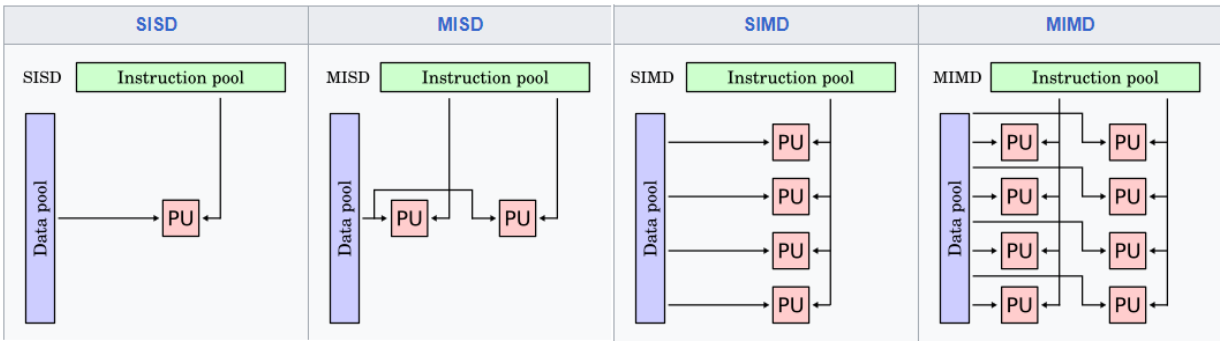
7.1. Conceptos básicos de computación paralela y distribuida

Comencemos revisando algunas clasificaciones de sistemas distribuidos y paralelos:

- En función de los **conjuntos** de **instrucciones** y **datos** (conocida como la **taxonomía** de **Flynn**):

  - La **arquitectura secuencial** la denominaríamos **SISD** (single instruction single data). Antiguos sistemas monoprocesador.

  - Las **diferentes arquitecturas paralelas** (o **distribuidas**) en diferentes grupos: **SIMD** (single instruction multiple data), **MISD** (multiple instruction single data; poco usada) y **MIMD** (multiple instruction multiple data), con algunas variaciones como la SPMD (single program multiple data).



- Por **comunicación** y **control**:

- Sistemas de **multiprocesamiento simétrico** (SMP). Son **MIMD** con **memoria compartida** . Los procesadores se comunican a través de esta memoria compartida; este es el caso de los microprocesadores de múltiples núcleos de nuestros Pcs.

- **Sistemas MIMD** con **memoria distribuida** . La memoria está distribuida entre los procesadores (o nodos) del sistema, cada uno con su propia memoria local, en la que poseen su propio programa y los datos asociados. Una red de interconexión conecta los procesadores (y sus memorias locales), mediante enlaces (links) de comunicación, usados para el intercambio de mensajes entre los procesadores. Los procesadores intercambian datos entre sus memorias cuando se pide el valor de variables remotas. Tipos específicos de estos sistemas son:

  - **Clusters**. Consisten en una colección de ordenadores (no necesariamente homogéneos) **conectados por red** para trabajar concurrentemente en tareas del mismo programa. Aunque la interconexión puede no ser dedicada.

  - **Grid**. Es un cluster cuya interconexión se realiza a través de **internet**.

Veamos una introducción a algunos **conceptos básicos** que debemos conocer para **desarrollar** en **sistemas distribuidos**:

- **Distribución**: construcción de una aplicación por partes, a cada parte se le asigna un conjunto de responsabilidades dentro del sistema.

- **Nodo de la red**: uno o varios equipos que se comportan como una unidad de asignación integrada en el sistema distribuido.

- Un **objeto distribuido** es un módulo de código con plena autonomía que se puede instanciar en cualquier nudo de la red y a cuyos servicios pueden acceder clientes ubicados en cualquier otro nudo.

- **Componente**: Elemento de software que encapsula una serie de funcionalidades. Un componente, es una unidad independiente, que puede ser utilizado en conjunto con otros componentes para formar un sistema más complejo (concebido por ser reutilizable). Tiene especificado: los servicios que ofrece; los requerimientos que necesarios para poder ser instalado en un nudo; las posibilidades de configuración que ofrece; y, no está ligado a ninguna aplicación, que se puede instanciar en cualquier nudo y ser gestionado por herramientas automáticas. Sus características:
  - **Alta cohesión**: todos los elementos de un componente están estrechamente relacionados.
  - **Bajo acoplamiento**: nivel de independencia que un componente respecto a otros.

- **Transacciones**: Conjunto de actividades que se ejecutan en diferentes nudos de una plataforma distribuida para ejecutar una tarea de negocio. Una transacción finaliza cuando todas las parte implicadas clientes y múltiples servidores confirman que sus correspondientes actividades han concluido con éxito.

- **Propiedades ACID** de una transacción:

  - **Atomicidad**: Una transacción es una unidad indivisible de trabajo, Todas las actividades que comprende deben ser ejecutadas con éxito.
  - **Consistencia**: Después de que una transacción ha sido ejecutada, la transacción debe dejar el sistema en estado correcto. Si la transacción no puede realizarse con éxito, debe restaurar el sistema al estado original previo al inicio de la transacción.
  - **Aislamiento**: La transacciones que se ejecutan de forma concurrente no deben tener interferencias entre ellas. La transacción debe sincronizar el acceso a todos los recursos compartidos y garantizar que las actualizaciones concurrentes sean compatibles entre si.
  - **Durabilidad**: Los efectos de una transacción son permanentes una vez que la transacción ha finalizado con éxito.

- ✔ **Gestor de transacciones**. Controla y supervisa la ejecución de transacciones asegurando propiedades ACID.

7.2. Tipos de paralelismo

Las mejoras arquitectónicas que han sufrido los computadores se han basado en la obtención de rendimiento explotando los diferentes niveles de paralelismo.

En un sistema podemos encontrar los siguientes **niveles** de **paralelismo**:

- A nivel de **bit**. Conseguido incrementando el tamaño de la palabra del microprocesador. Realizar operaciones sobre mayor número de bits. Esto es, el paso de palabras de 8 bits, a 16, a 32 y en los microprocesadores actuales 64 bits.

- A nivel de **instrucciones**. Conseguida introduciendo ***pipeline*** en la ejecución de instrucciones máquina en el diseño de los microprocesadores.

- A nivel de **bucle**. Consiste en dividir las interacciones de un bucle en partes que se pueden realizar de manera complementaria. Por ejemplo, un bucle de 0 a 100; puede ser equivalente a dos bucles, uno de 0 a 49 y otro de 50 a 100.

```
for (int i = 0; i < = 100; i++)
      L[i] = L[i] + 10;
```

¿Podría este bucle ser paralelizado?

- A nivel de **procedimientos**. Identificando qué fragmentos de código dentro de un programa pueden ejecutarse de manera simultánea sin interferir la tarea de una en la otra.

- A nivel de **tareas dentro de un programa**. Tareas que cooperan para la solución del programa general (utilizado en sistemas distribuidos y paralelos).

- A nivel de **aplicación dentro de un ordenador**. Se refiere a los conceptos que vimos al principio de la unidad, propios de la gestión de procesos por parte del sistema operativo multitarea: planificador de procesos, Round-Robin, quamtum, etc.

En sistemas distribuidos hablaremos de la granularidad, que es una medida de la cantidad de computación de un proceso software. Se considera como el segmento de código escogido para su procesamiento paralelo; tenemos:

- Paralelismo de **Grano Fino**: No requiere tener mucho conocimiento del código y la paralelización se obtiene de forma casi automática. Permite conseguir buenos resultados en eficiencia en poco tiempo. Por ejemplo: la descomposición de bucles.

- Paralelismo de **Grano Grueso**: Es una paralelización de alto nivel que engloba al grano fino. Requiere mayor conocimiento del código puesto que se paraleliza mayor cantidad de él. Consigue mejores rendimientos que la paralelización fina, ya que intenta evitar los overhead (exceso de recursos asignados y utilizados) que se suelen producir cuando se divide el problema en secciones muy pequeñas. Un ejemplo de paralelismo grueso lo obtenemos descomponiendo el problema en dominios (dividiendo conjunto de datos a procesar, acompañando a estos, las acciones que deban realizarse sobre ellos).

7.3. Infraestructuras para programación distribuida

Las **aplicaciones distribuidas** requieren que componentes que se ejecutan en diferentes procesadores se comuniquen entre sí. Los modelos de infraestructura que permiten la implementación de esos componentes, son:

- Uso de **Sockets**: Facilitan la generación dinámica de canales de comunicación. Es actualmente la base de la comunicación. Pero al ser de muy bajo nivel de abstracción, no son adecuados a nivel de aplicación.
- **Remote Procedure Call** (RPC): Abstrae la comunicación a nivel de invocación de procedimientos. Es adecuada para programación estructurada basada en librerías.
- **Invocación remota** de **objetos**: Abstrae la comunicación a la invocación de métodos de objetos que se encuentran distribuidos por el sistema distribuido. Los objetos se localizan por su identidad. Es adecuada para aplicaciones basadas en el paradigma OO.
- RMI (**Remote Method Invocation**) es la solución Java para la comunicación de objetos Java distribuidos. Presenta un inconveniente, y es el paso de parámetros por valor implica tiempo para hacer la serialización, enviar los objetos serializados a través de la red y luego volver a recomponer los objetos en el destino.
- **CORBA** (Common Object Request Brocker Architecture). Para facilitar el diseño de aplicaciones basadas en el paradigma Cliente/Servidor. Define servidores estandarizados a través de un modelo de referencia, los patrones de interacción entre clientes y servidores y las especificaciones de las APIs.
- **MPI** ("Message Passing Interface", Interfaz de Paso de Mensajes) es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.
- La **Interfaz de Paso de Mensaje**s es un protocolo de comunicación entre computadoras. Es el estándar para la comunicación entre los nodos que ejecutan un programa en un sistema de memoria distribuida. Las llamadas de MPI se dividen en **cuatro** clases:
  - Llamadas utilizadas para inicializar, administrar y finalizar comunicaciones.
  - Llamadas utilizadas para transferir datos entre un par de procesos.
  - Llamadas para transferir datos entre varios procesos.
  - Llamadas utilizadas para crear tipos de datos definidos por el usuario.
- **Máquina paralela virtual**. Paquetes software que permite ver el conjunto de nodos disponibles como una máquina virtual paralela ; ofreciendo una opción práctica, económica y popular hoy en día para aproximarse al cómputo paralelo .

7.4. Comparando la programación concurrente, la paralela y la distribuida

*Hablamos de **programación concurrente** cuando se ejecutan en un dispositivo informático de forma simultánea diferentes tareas (procesos).*

*Cuando la programación concurrente se realiza en un sistema multiprocesador hablamos de **programación paralela**.*

La programación paralela está **muy ligada a la arquitectura del sistema** de computación. Pero cuando programamos en lenguajes de alto nivel debemos abstraernos de la plataforma en la que se ejecutará. Los lenguajes de alto nivel nos proveen de librerías que nos facilitan esta abstracción y nos permiten utilizar las mismas operaciones para programar de forma paralela que acabarán implementadas para usar una plataforma concreta.

Un tipo especial de programación paralela es la llamada **programación distribuida**. Esta programación se da en sistemas informáticos distribuidos. Un sistema distribuido está formado por un conjunto de ordenadores que pueden estar situados en sitios geográficos diferentes unidos entre sí a través de una red de comunicaciones. Un ejemplo de sistema distribuido puede ser el de un banco con muchas oficinas en el mundo, con un ordenador central por oficina para guardar las cuentas locales y realizar las transacciones locales. Este ordenador puede comunicarse con los otros ordenadores centrales de la red de oficinas. Cuando se realiza una transacción no importa dónde se encuentra la cuenta o el cliente.

*La **programación distribuida** es un tipo de programación concurrente en la que los procesos son ejecutados en una red de procesadores autónomos o en un sistema distribuido. Es un sistema de computadores independientes que desde el punto de vista del usuario del sistema se ve como una única computadora.*