1. Aplicación, Programa, Ejecutable y Proceso.

A simple vista, parece que con los términos **aplicación, programa**, **ejecutable** y **proceso**, nos estamos refiriendo a lo mismo. Pero, no olvidemos que en los módulos de primero hemos aprendido a diferenciarlos.

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 (código máquina) o interpretado, lo guardamos en un fichero. Este fichero es llamado comúnmente ejecutable, es decir, 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 programa, 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: no se refiere únicamente al código y a los datos del programa en ejecución, sino que también incluye todo lo necesario para la ejecución del mismo:

- ✔ Código del programa.
- ✔ Los datos y la pila del programa.
- ✔ Contador de programa, para controlar por donde se va ejecutando el programa.
- ✔ Puntero a la pila del programa.
- ✔ Imagen de la memoria que afecta el proceso.
- ✔ Estado del procesador: valores de los registros del procesador.
- ✔ Identificador del proceso.
- ✔ Estado del proceso.
- ✔ Información sobre la gestión de la memoria.
- ✔ Prioridad del proceso.
- ✔ Información sobre el estado de la E/S.
- ✔ Tiempo de CPU consumido.
- ✔ …

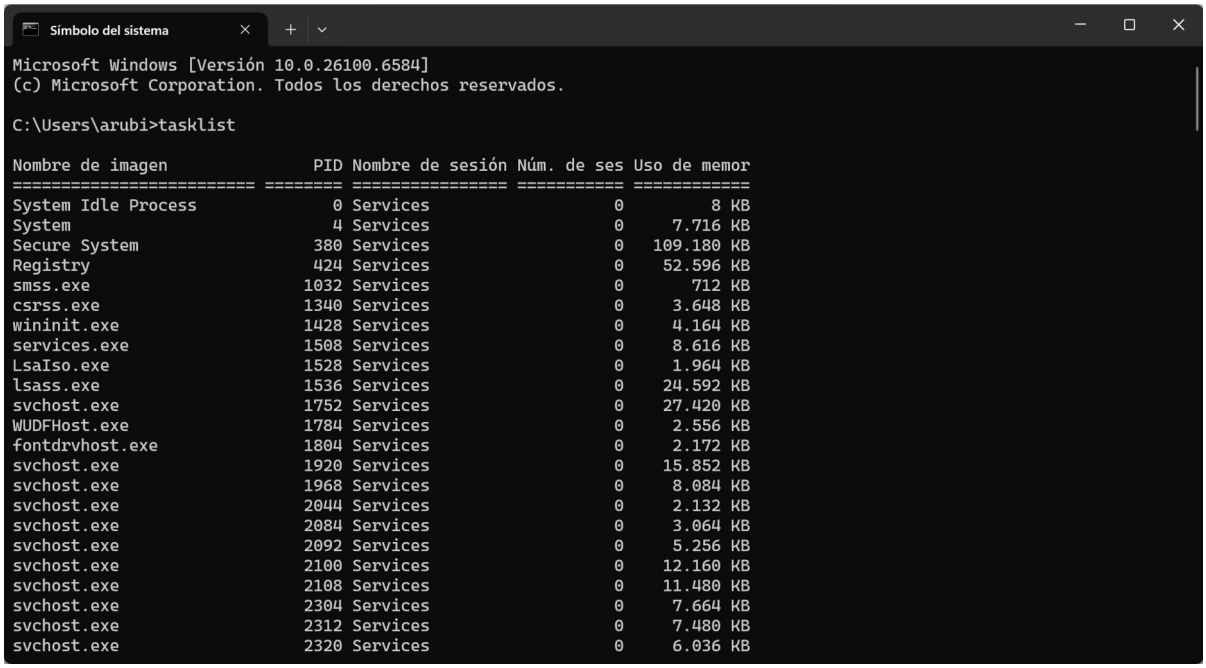
Toda esta información será guardada en una estructura de datos llamada **Bloque de Control de Procesos (BCP)**.

¿Y quien se encarga de su gestión? El SO. En siguientes apartados de esta unidad trataremos más en profundidad todo lo relacionado con los procesos y el SO.

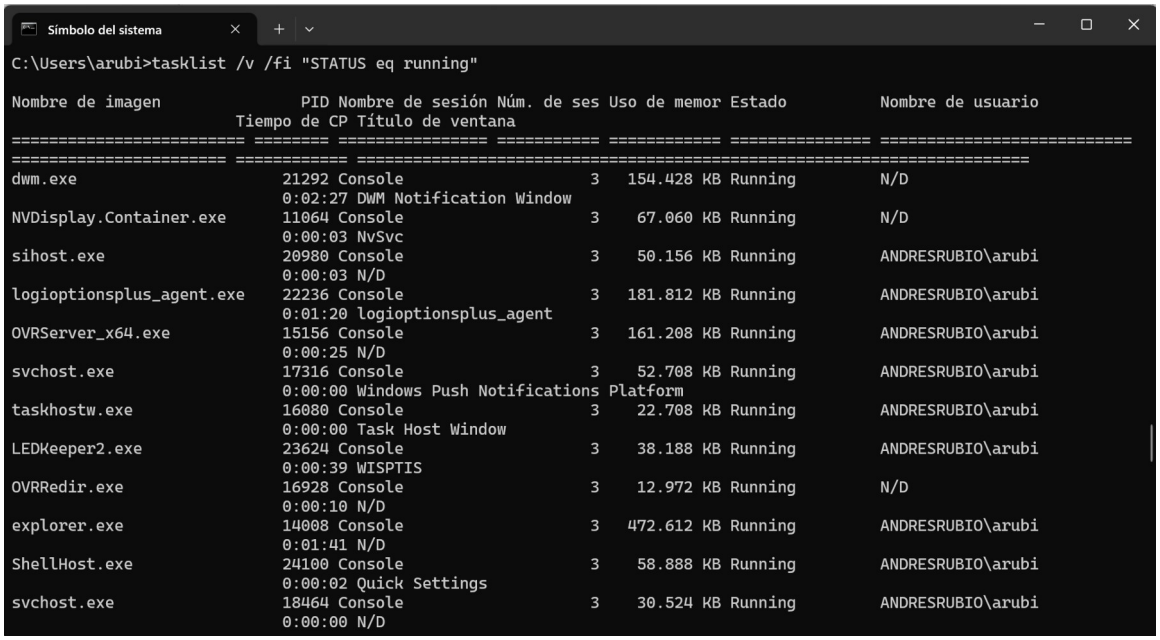
Es importante destacar que los procesos son entidades independientes aunque sean diferentes ejecuciones de un mismo programa, ya que pueden contener distintos datos (distintas imágenes de memoria) y distintos momentos de ejecución (diferentes contadores de programa). Ejemplo: Dos instancias del programa Word ejecutándose a la vez, modificando cada una un fichero diferente. Para que los datos de uno no interfieran con los del otro, cada proceso se ejecuta en su propio espacio de direcciones de memoria.

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.

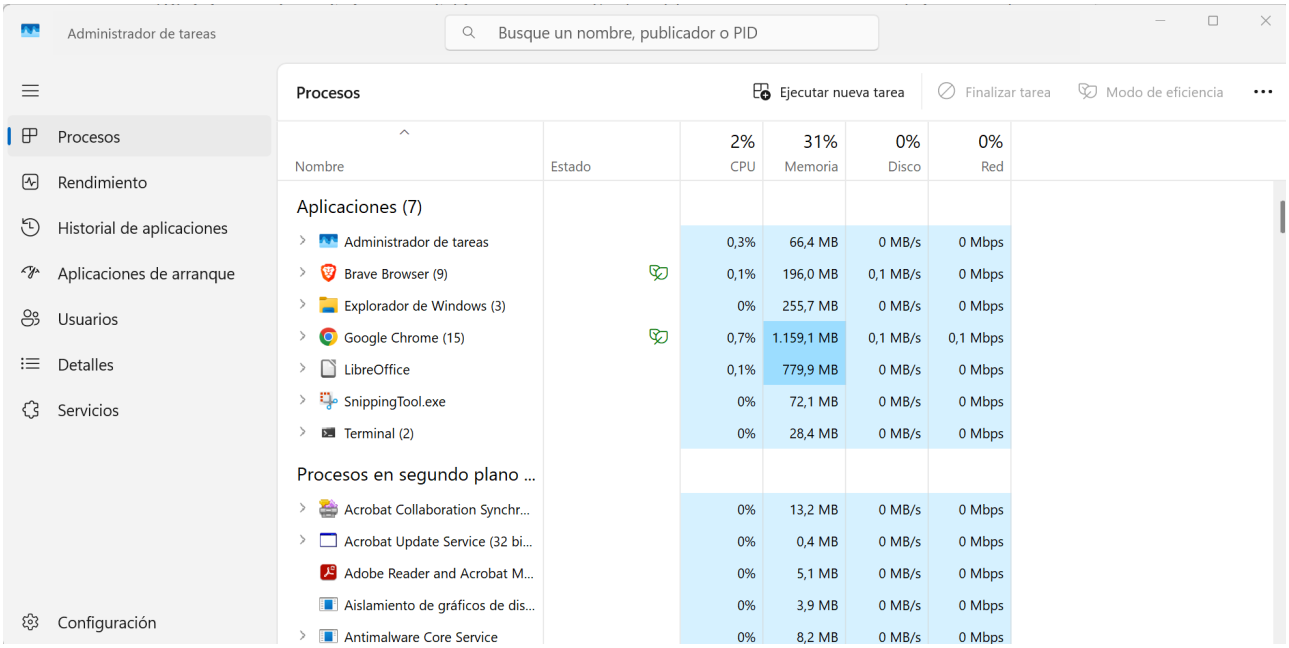
En sistemas operativos Windows se puede usar desde la línea de comandos la orden **tasklist** para ver los procesos que se están ejecutando. En otros sistemas operativos como **Linux** esto mismo lo obtenemos mediante el comando **ps**.



O si queremos ver los procesos que están ejecutándose, podemos añadir una serie de parámetros al comando **tasklist**:



También se pueden ver los procesos existentes a través de alguna herramienta gráfica proporcionada por el sistema operativo o por algún software externo. En el caso de **Windows** tenemos el **Administrador de tareas**.



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 ***IntelliJ***, tenemos el navegador web 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. 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? 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. Al final tendremos realizadas todas las tareas, pero para 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.

2.1. 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 del 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 una entrada de datos que deba suministrar el usuario, tiene que imprimir o grabar datos, es decir, el proceso está en una operación de entrada/salida (E/S para abreviar), dicho 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 uno de ellos está esperando una E/S, otros procesos pueden pasar a la CPU y ejecutar sus instrucciones. Cuando termina la E/S de 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).
- Sólo mencionar (o recordar), que cuando la memoria RAM del equipo está llena, algunos procesos deben pasar a disco (o almacenamiento secundario) para dejar espacio en RAM que permita la ejecución de otros procesos.

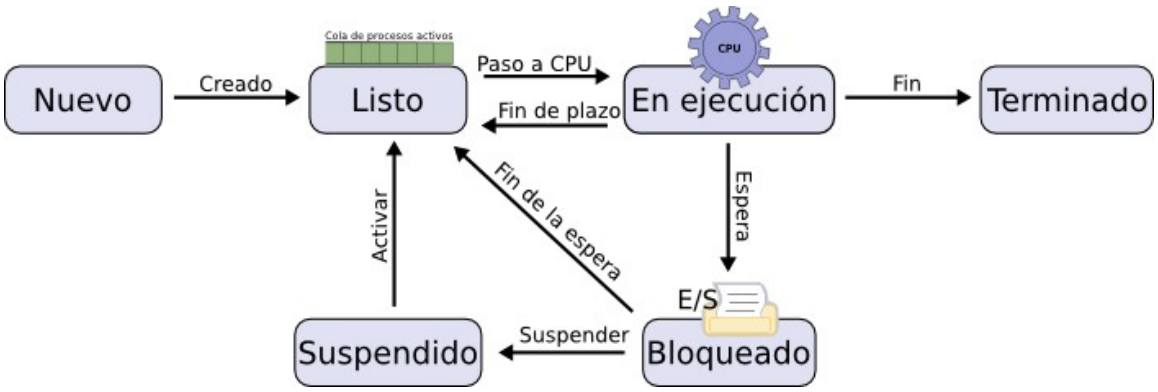
Importante: Todo proceso en ejecución, tiene que estar cargado en la RAM física del equipo o memoria principal, así como todos los datos que necesite.

- 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 programa, y por lo tanto un proceso o un conjunto de procesos en ejecución. Se le da prioridad, evidentemente, a los procesos del SO, frente a los procesos de usuario.

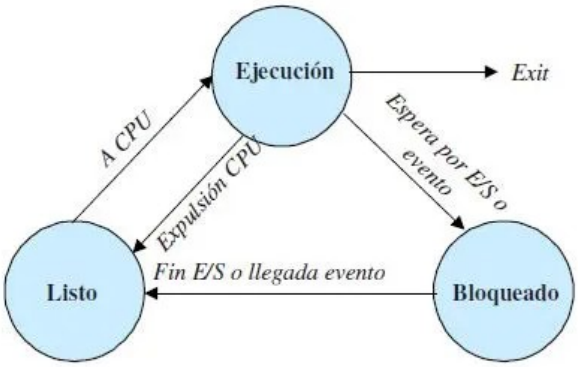
Con todo lo anterior, podemos quedarnos con los siguientes **estados en el ciclo de vida de un proceso:**

1. Nuevo. Proceso nuevo, creado.
2. Listo. Proceso que está esperando la CPU para ejecutar sus instrucciones.
3. En ejecución. Proceso que actualmente, está en turno de ejecución en la CPU.
4. Bloqueado. Proceso que está a la espera de que finalice una E/S.
5. Suspendido. Proceso que se ha llevado a la memoria virtual para liberar, un poco, la RAM del sistema.
6. 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:



Un esquema más simplificado de los estados de un proceso se puede ver en la siguiente imagen:



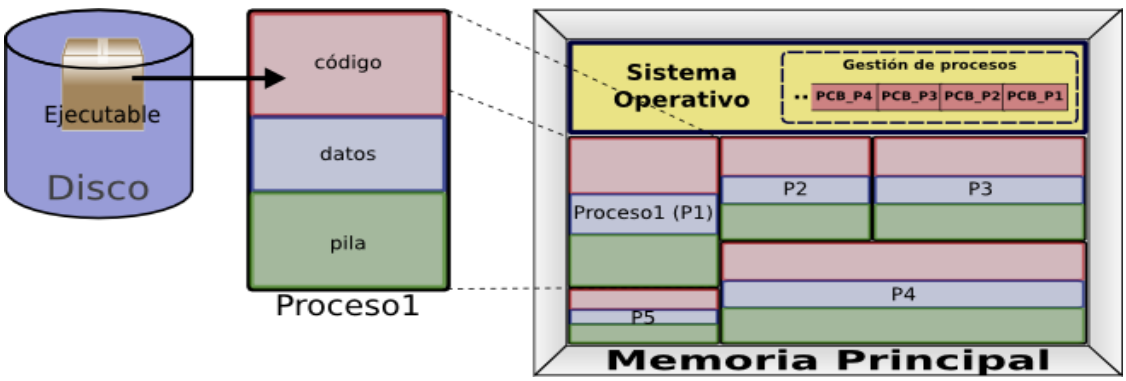
2.2. Planificación de procesos por el Sistema Operativo

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 proceso) 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).



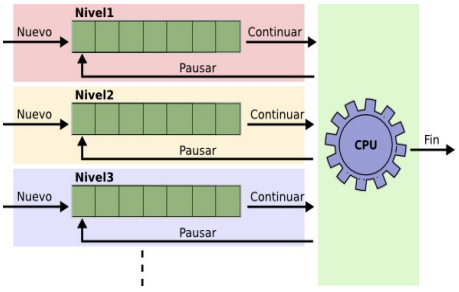
2) **Crea** una **estructura de información** llamada **PCB** (Bloque de Control de Proceso). La

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. Se encarga de decidir qué proceso se ejecuta y cuánto tiempo se ejecuta. El planificador es otro proceso que, en este caso, es parte del SO. La política en la toma de decisiones del planificador se denominan: 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 quamtum. Si no le ha dado tiempo a finalizar en ese quamtum, se coloca al final de la cola de procesos listos, y espera a que vuelva su turno de procesamiento. Así, todos los procesos listos 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 recibir su tiempo de procesamiento 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. Es una combinación de los dos anteriores y es 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. Hay que evitar la **inanición**.
- Eficacia y rendimiento. Mantener ocupada la CPU un 100% del tiempo, maximizando el número de tareas procesadas.
- Tiempo de respuesta. Minimizar el tiempo de respuesta al usuario.

En el siguiente enlace puedes ver una simulación del algoritmo de planificación Round-Robin. En él podrás ver cómo los procesos van tomando su turno de ejecución en la CPU hasta su finalización.

https://www.youtube.com/watch?v=TfwLl9AWvNA

3. Creación y gestión de procesos con Java

Java dispone en el paquete **java.lang** de varias clases para la gestión de procesos. Las clases que vamos a necesitar para la creación de procesos son ***ProcessBuilder, Runtime y Process***. Las dos primeras clases permiten lanzar programas externos (por ejemplo, ejecutar un comando del SO o lanzar otro programa Java).

ProcessBuilder

Permite crear un proceso en el sistema desde nuestro código Java a través de la llamada al método ***start***. Por tanto, a través de la llamada al método ***start*** se crea un nuevo proceso en el sistema con los atributos definidos en la instancia de ***ProcessBuilder***.

Su sintaxis es: *ProcessBuilder(String... command)*

```
ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");
Process proceso = pb.start();
```

Si llamamos varias veces al método start, se crearán tantos nuevos procesos como llamadas hagamos, todos ellos con los mismos atributos.

La ejecución del método ***start*** puede lanzar una excepción de tipo ***IOException***.

Runtime

Permite lanzar la ejecución de un programa en el sistema desde nuestro código Java.

De esta clase es interesante el método ***exec( )***, cuya sintaxis es ***exec(String comando)*** que devuelve un objeto de tipo ***Process***. Para poder ejecutarlo es necesario obtener previamente el objeto **Runtime** de la aplicación Java que actualmente se está ejecutando. Para ello se usa el método ***getRuntime***.

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 o IllegalArgumentException:*** si el parámetro comando es una cadena nula o vacía.

Process

Proporciona un objeto Proceso a través del cual podemos controlar los procesos creados desde nuestro código.

Hay que tener presente que el proceso que se crea (subproceso o proceso hijo) no tiene su propia terminal o consola. Por tal motivo, todas las salidas del proceso creado serán redirigidas al proceso padre, pudiendo acceder a las mismas usando los métodos ***getOutputStream***, ***getInputStream*** y ***getErrorStream***.

El proceso padre utilizará estos flujos para proporcionar datos al subproceso/proceso hijo y para obtener la salida del subproceso/proceso hijo.

En la siguiente tabla se muestran los métodos más destacados de la clase ***Process***:

Métodos	Misión
InputStream getInputStream()	Devuelve el flujo de entrada conectado a la salida normal del subproceso. Nos permite leer el stream de salida del subproceso, es decir, lo que el comando que ejecutamos escribió en la consola. **BufferedReader br = new BufferedReader(new InputStreamReader(p.getInputStream()));** **String linea;** **while ((linea = br.readLine()) != null)** **{** **System.out.println("OUTPUT: " + linea);** **}**
int waitFor()	Provoca que el proceso actual espere hasta que el subproceso representado por el objeto Process finalice. Devuelve 0 si ha finalizado correctamente. **int code = p.waitFor();** **System.out.println("Terminó con: " + code);** Una variante de waitFor **boolean finished = p.waitFor(5, java.util.concurrent.TimeUnit.SECONDS);** **if (!finished)** **{** **System.out.println("Proceso tardó demasiado, se fuerza su fin.");** **p.destroyForcibly();** **}**
InputStream getErrorStream()	Devuelve el flujo de entrada conectado a la salida de error del subproceso.

	Nos va a permitir poder leer los posibles errores que se produzcan al lanzar el subproceso. **BufferedReader err = new BufferedReader(new InputStreamReader(p.getErrorStream()));** **String lineaError;** **while ((lineaError = err.readLine()) != null)** **{** **System.err.println("ERROR: " + lineaError);** **}**
OutputStream getOutputStream()	Devuelve el flujo de salida conectado a la entrada normal del subproceso. Nos va a permitir escribir en el stream de entrada del subproceso, así podemos enviar datos al subproceso que se ejecute. **OutputStream os = p.getOutputStream();** **os.write("Hola proceso\n".getBytes());** **os.flush();** **os.close();**
void destroy()	Elimina el subproceso de un modo no forzado. **Process p = Runtime.getRuntime().exec("ping google.com");** **p.destroy();** Una variante de este método es destroyForcibly que fuerza a que el proceso finalice.
int exitValue()	Devuelve el valor de salida del subproceso. Si devuelve 0 quiere decir que terminó con éxito. Si el proceso aún no terminó, lanza **IllegalThreadStateException**. **int code = p.exitValue();** **System.out.println("Código de salida: " + code);**
boolean isAlive()	Comprueba si el subproceso representado por Process está vivo. **if (p.isAlive()) {** **System.out.println("El proceso aún se está ejecutando...");** **}**
long pid()	Devuelve el PID del proceso. **long pid = p.pid();**

```
System.out.println("PID: " + pid);
```

ProcessHandle.Info info()	Devuelve una instantánea de información sobre el proceso.	
	Método	**Descripción**
	command()	Ruta completa del comando que inició el proceso.
	arguments	Lista de argumentos con los que se inició el proceso.
	startInstant()	Fecha y hora de inicio del proceso.
	totalCpuDuration()	Tiempo total de CPU consumido por el proceso.
	user()	Usuario que ejecutó el proceso.

Ver ejemplos 1 y 2.

3.1. Leer datos enviados por los subprocesos

Ver ejemplo 3

Tal y como hemos comentado anteriormente, hay que tener presente que cuando se crea un proceso, debido a que no tiene su propia terminal o consola, todas las salidas del proceso creado serán redirigidas al proceso padre. Por tanto, si es necesario leer dicha salida se tienen dos opciones:

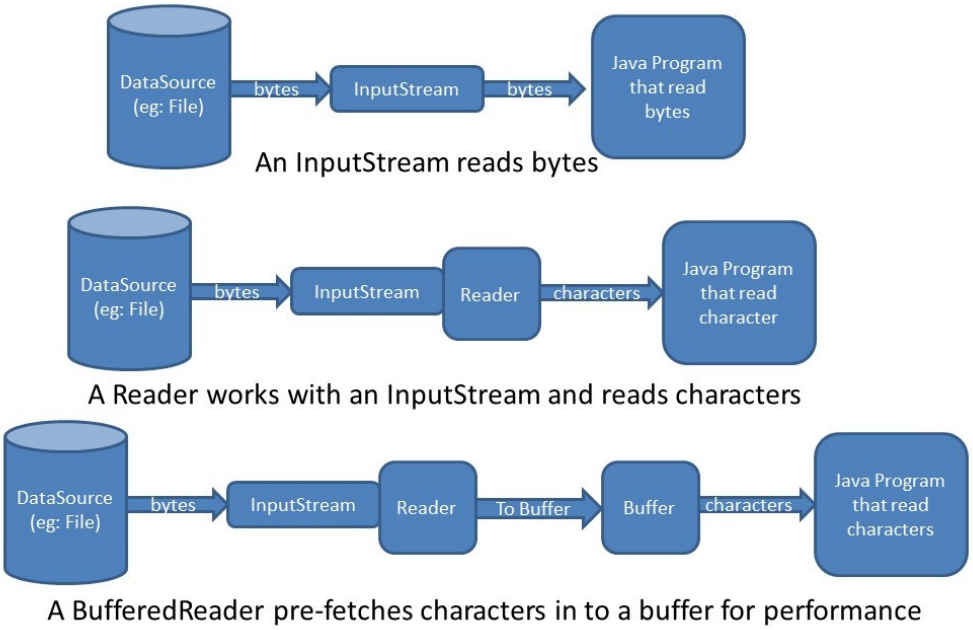
1. Usar el método ***getInputStream*** de la clase ***Process***.

   Ver ejemplos 4 y 5

2. Usar el método ***InheritIO*** de la clase **ProcessBuilder**.

   Este método permite que la salida del proceso creado aparezca directamente en la consola del proceso padre sin necesidad de capturarla manualmente.

   Ver ejemplo 6



3.2. Lectura de posibles errores al lanzar subprocesos

La clase ***Process*** posee el método ***getErrorStream*** que va a devolver un stream para leer los posibles errores que se produzcan al lanzar el proceso. Por ejemplo, si en el Ejemplo4.java cambiamos los argumentos y escribimos algo incorrecto como lo siguiente:

```
proceso1 = new ProcessBuilder("CMD", "/C", "DIRR").start();
```

La comprobación del resultado de salida devolverá 1, indicando que el proceso no ha finalizado correctamente.

```
exitVal = proceso1.waitFor();
```

En este caso el valor almacenado en exitVal será 1, en lugar de 0.

Por lo tanto, es necesario hacer un tratamiento de errores que nos indique el error existente en la creación del subproceso. Para esto, se hará uso del método ***getErrorStream*** que devolverá el flujo sobre el cual se irán escribiendo los errores generados por el subproceso. Además, sobre dicho flujo lo más adecuado es montar un ***InputStreamReader*** que convierte el ***InputStream*** en un ***Reader*** para poder trabajar con caracteres en lugar de con bytes. Además, esto se suele envolver en un buffer lector (***BufferedReader***) para facilitarnos la lectura de los errores que se produzcan.

```
InputStream error=proceso1.getErrorStream();
BufferedReader br=new BufferedReader(new InputStreamReader(error));

 String linea=null;
while ((linea=br.readLine())!=null)
      System.out.println("ERROR: " + linea);
```

Ver ejemplos: SubprocesoErroneoConFlujo y SubprocesoErroneoSinFlujo

3.3. Enviar datos al subproceso

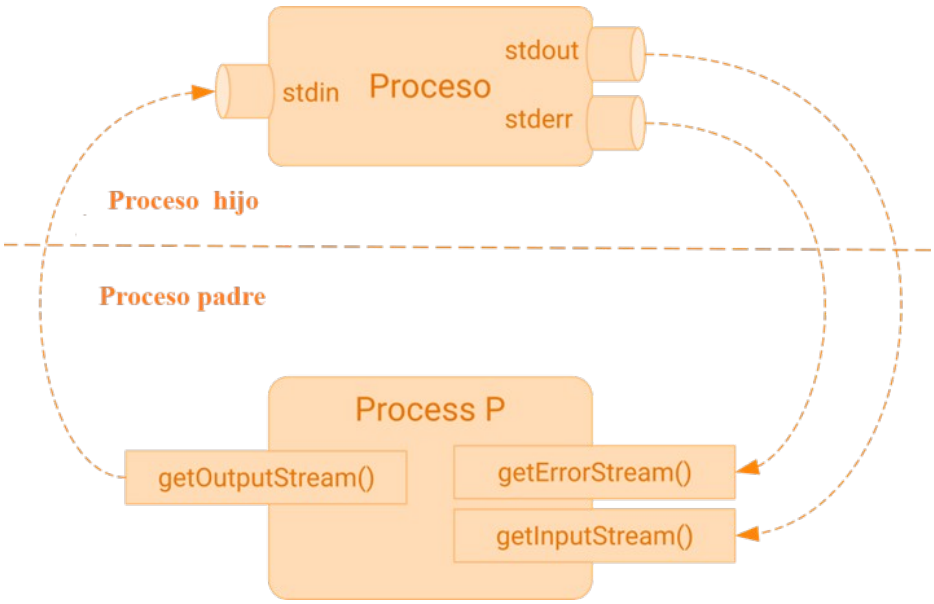
En el caso de tener que crear un proceso que necesita información de entrada, se tienen dos posibles opciones:

1. Pasar los datos de entrada a través de los parámetros de la aplicación (**args**).

2. Hacer uso del método ***getOutputStream*** para escribir en el flujo que comunica el proceso principal con el subproceso creado.

Para trabajar con este flujo se suele crear un ***OutputStreamWriter*** que convierte el ***OutputStream*** en un ***Writer*** para poder manejar caracteres en lugar de bytes. Además, al igual que en el caso anterior, se suele envolver en un buffer de escritura (***BufferedWriter***) para mejorar el rendimiento y escribir texto de forma más eficiente. También se puede usar un ***PrintWriter*** en lugar de un ***BufferedReader***.

Ver: GetOutputStreamConBuffered, GetOutputStreamSinBuffered, OtroGetOutputStreamConBuffered



3.4. Redirección de Errores y de las Entradas y Salidas Estándar

En una situación real, lo más probable es que sea necesario guardar los resultados de un subproceso en un archivo de log o de errores para su posterior análisis. La ***API*** de ***ProcessBuilder*** nos proporciona los métodos para poder lograrlo sin necesidad de modificar el código de nuestras aplicaciones.

Para ello se usarán los ***diferentes métodos redirect*** que van a permitir cambiar la entrada estándar, la salida estándar y los errores de los subprocesos a otros destinos como puede ser un fichero.

```
pBuilder.redirectInput(new File(entrada.txt));
pBuilder.redirectOutput(new File(salida.txt));
pBuilder.redirectError(new File(error.txt));
```

En estos casos hay que resaltar que si se modifican dichas entradas y salidas, los correspondientes métodos ***getInputStream***, ***getOutputStream*** y ***getErrorStream*** devolverán ***NullInputStream***, ***NullOutputStream*** y ***NullErrorStream*** respectivamente.

Además, estas redirecciones de la E/S, hay que hacerla desde el proceso padre mientras se prepara el subproceso para ser ejecutado, es decir, antes de llamar al método ***start***.

Ver: RedireccionSalida.

La solución anterior genera un problema: los ficheros se sobreescriben. Para solucionarlo se ha de usar el método `Redirect.appendTo` del siguiente modo:

```
pBuilder.redirectOutput(ProcessBuilder.Redirect.appendTo(salida.txt));
```

Ver: RedireccionEntradaSalidaError

Para el redireccionamiento de la entrada y de la salida de un proceso, también se puede usar la clase ***ProcessBuilder.Redirect***. El redireccionamiento puede ser uno de los siguientes:

- ***Redirect.INHERIT***: Indica que la fuente de entrada o salida del proceso será la misma que la del proceso actual.
- ***Redirect.from(File)***: Redirige la entrada del proceso al fichero indicado.
- ***Redirect.to(File)***: Redirige la salida del proceso al fichero indicado.
- ***Redirect.appendTo(File)***: Redirige la salida del proceso al fichero indicado pero sin sobreescribir.

Ver: ProcesoPadreProcessBuilderRedirect

3.5. Configuraciones adicionales de un proceso

Algunos de los atributos importantes que se pueden configurar para un proceso son:

- **Establecer el directorio de trabajo donde el proceso se ejecutará**

  Por defecto, el directorio de trabajo de un proceso se establece al valor de la variable del sistema ***user.dir***. Este directorio es el punto de partida para acceder a ficheros, imágenes y todos los recursos que necesite nuestra aplicación.

  Salvo que se cambie, el valor de la variable ***user.dir*** en un proyecto Java se corresponde con el path donde se encuentra almacenado el proyecto.

  Esto se puede cambiar llamando al método ***directory*** de la clase ***ProcessBuilder***, pasándole un objeto de tipo ***File***.

  También se puede usar el método ***directory()*** para consultar el valor del directorio de trabajo. El valor devuelto por el método ***directory*** no solo será el directorio de trabajo del proceso principal, también lo será para todos los subprocesos creados por él.

Hay que destacar que cuando el directorio de trabajo es el mismo que el valor de la variable ***user.dir***, ***directory*** devolverá un valor ***null***.

```
// Cambia el directorio de trabajo a la carpeta personal del usuario
pbuilder.directory(new File(System.getProperty("user.home")));
```

- **Configurar o modificar variables de entorno para el proceso con el método environment()**

```
// Obtenemos las variables de entorno para el proceso
Map<String, String> environment = pbuilder.environment();
// Obtenemos la variable de entorno path
String systemPath = environment.get("Path")
```

Ver: DirectorioTrabajo

3.6. Lanzar una clase Java como proceso desde otra clase Java en el mismo proyecto

Para lograr que desde un proceso principal se llame a otra clase Java previamente compilada es necesario establecer correctamente el directorio de trabajo del proceso principal.

En ningún momento, cuando se está programando un proceso, se debe pensar si va a ser lanzado como padre o como hijo. Es más, todos los programas implementados son lanzados como hijos por el IDE y eso no hace que cambiemos nuestra forma de programarlos.

Un proceso que se vaya a lanzar como hijo debería funcionar perfectamente como proceso independiente y puede ser ejecutado directamente sin tener que hacerle ningún tipo de cambio.

Ver: Lanzador y Lanzado.

En el ejemplo ***Lanzador/Lanzado*** se han utilizado los parámetros del subproceso para llevar a cabo la comunicación desde el proceso Lanzador (proceso padre) al subproceso Lanzado (proceso hijo).

Sin embargo, dado el que el proceso ***Lanzado*** carece de consola, al usar la salida estándar (System.out) para mostrar la información generada, dicha información se pierde si el proceso Lanzador no la lee. Para ello, se utilizarán los Streams previamente comentados: ***InputStream*** para leer la salida estandar generada por el proceso hijo, ***OutputStream*** para escribir en la entrada estandar del proceso hijo y ***ErrorStream*** para leer los errores existentes.
Por lo tanto, el proceso padre puede usar el flujo de salida ***OutputStream*** para enviar datos al proceso hijo. El proceso hijo los leerá usando su entrada estándar (***System.in***). Por otro lado, el proceso padre puede usar el flujo ***InputStream*** para recibir datos del proceso hijo que serán escritos por este en su salida estándar.

Ver: ProcesoPadre y ProcesoHijo para envío de datos.

3.7. Lanzar un proyecto Java como proceso desde otro proyecto Java

En este caso vamos a tener un fichero .jar de un proyecto Java previamente compilado y vamos a lanzar el proceso correspondiente desde otro proyecto Java que estamos desarrollando.

Concretamente, el proceso a lanzar se llama ***PintaDatos.jar***. Se trata de un proceso que recibe datos por parámetro y los muestra en la consola.

En este caso como el proyecto jar no recibe datos por su entrada estándar, se lo tenemos que enviar a través de ProcessBuilder cuando establecemos el comando que se va a asociar al proceso:

```
pb=new ProcessBuilder(“java”,”-jar”,”jars/PintaDatos.jar”,”uno”,”dos”,”tres”);
```

Muy importante indicar correctamente donde se encuentra el fichero jar que se desea ejecutar ya que de lo contrario obtendremos un error.

Ver: lanzaJar

3.8. Subproceso con parámetros dinámicos

Hay ocasiones en los que el subproceso a lanzar no recibe una cantidad fijada de parámetros, es decir, unas veces puede ejecutarse pasándole 2 parámetros, otras veces ninguno, otras veces 5,... A esto se le conoce como subproceso con parámetros dinámicos.

Esto nos genera una dificultad cuando se crea el objeto ***ProcessBuilder*** puesto que no se puede saber a priori los parámetros a pasar y por consiguiente, no se puede crear el objeto. Como ejemplo se tiene el proyecto ***PintaDatos.jar*** que recibe un número dinámico de parámetros. En el apartado anterior se le han pasado tres parámetros, pero ¿y si el usuario en una ejecución posterior quiere pasarle ahora 2 parámetros?. Se tendría que cambiar la sentencia de creación del objeto ProcessBuilder por la siguiente:

```
pb=new ProcessBuilder(“java”,”-jar”,”jars/PintaDatos.jar”,”uno”,”dos”);
```

Teniendo nuevamente que recompilar el código del proyecto.

Para solventar esto ***ProcessBuilder*** nos ofrece la posibilidad de agregar parámetros dinámicamente usando el método `pb.command.add();` quedando el caso anterior de la siguiente manera:

```
pb=new ProcessBuilder(“java”,”-jar”,”jars/PintaDatos.jar”);
pb.command.add(“uno”);
pb.command.add(“dos”);
```

Por tanto, se podría incluir en el proceso padre un bucle while en el que se pidan al usuario todos los parámetros que se quieren pasar al subproceso y luego añadirlos dinámicamente usando el método `pb.command.add();`

Ver: lanzaJarDinamico

Del mismo modo, puede ocurrir que los argumentos a pasar al subproceso se encuentren en un fichero. En este caso, primero se debe leer el fichero creando una lista con los datos del fichero y luego se pasan al objeto ***ProcessBuilder*** usando `pb.command.addAll(argumentos);`

Ver: LanzaJarDinamicoParametrosEnFichero

4. Programación concurrente vs Programación Paralela

La definición de **concurrencia**, no es algo sencillo. En el diccionario, concurrencia es la **coincidencia de varios sucesos al mismo tiempo**.

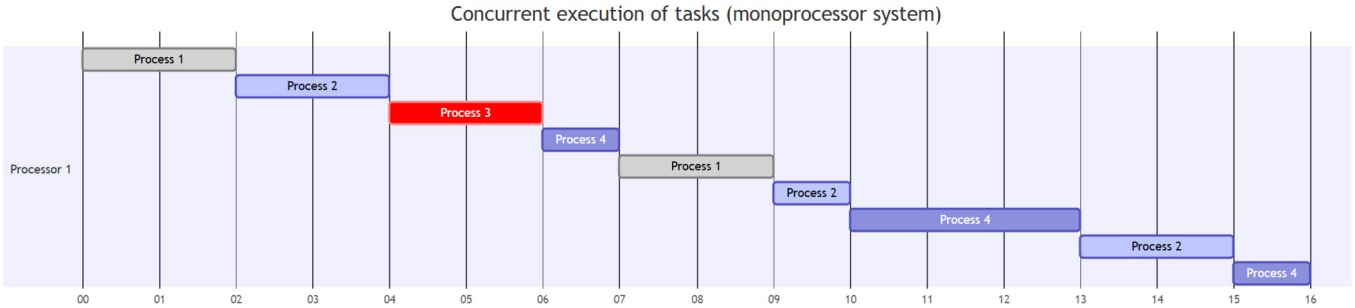
La **programación concurrente** es un paradigma de programación que permite que **varias tareas o procesos se ejecuten de forma solapada en el tiempo**, pudiendo progresar de manera independiente y potencialmente interactuando entre sí.

No implica necesariamente que las tareas se ejecuten **simultáneamente** (como en sistemas multiprocesador o multinúcleo), sino que **sus ejecuciones se intercalan** o **superponen** para aprovechar mejor los recursos del sistema y mejorar la eficiencia o la capacidad de respuesta.

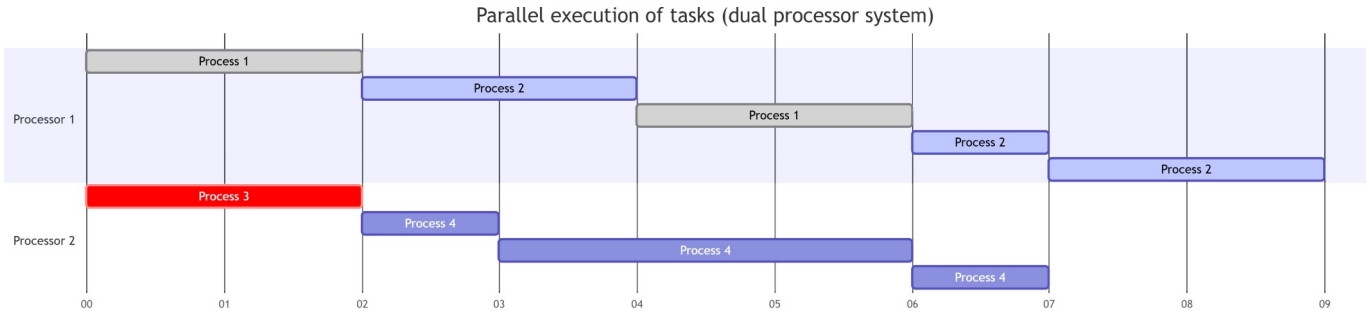
Sin embargo, esto puede originar situaciones de competencia por recursos y la programación concurrente los va a solventar proporcionando mecanismos de comunicación y sincronización entre los procesos que se ejecutan de forma simultanea en un sistema informático.

Por tanto, 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 un sistema monoprocesador, si se tienen varios procesos activos, estos nunca se van a poder ejecutar a la vez. Lo único que se puede hacer es que el Sistema Operativo vaya alternando el uso de la CPU entre los distintos procesos. Esta forma de gestionar los procesos en un sistema monoprocesador recibe el nombre de **multiprogramación**.



Con el avance de la tecnología la gran mayoría de dispositivos actuales, tienen capacidades de multiproceso, es decir, tienen más de un procesador para poder realizar varias tareas a la vez de forma real, no simulada. A este tipo de ejecución se le llama **paralelismo**.



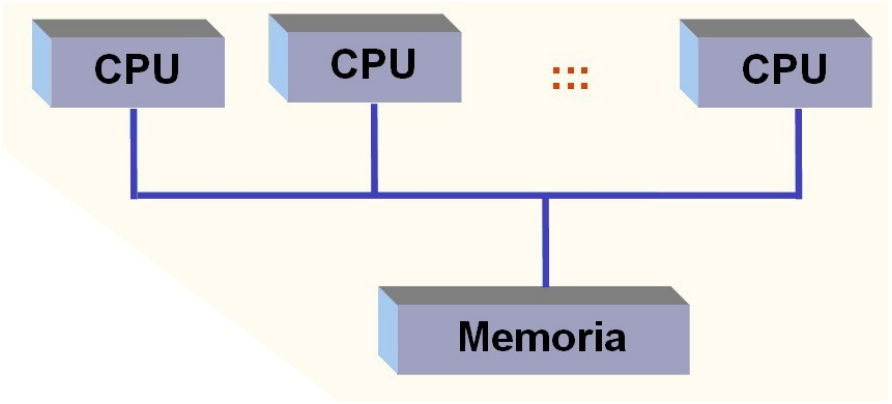
Como se puede apreciar en la imagen, los proceso finalizan antes y por tanto, el usuario obtendrá los resultados más rápidos.

Por tanto, un **programa paralelo** es un tipo de programa concurrente diseñado para ejecutarse en un sistema **multiprocesador**, pudiendo tener un proceso en cada procesador.

En el caso de tener un sistema multiprocesador y varios procesos trabajando juntos para resolver un problema, cada procesador estaría realizando una parte del problema y sería necesario (muy posiblemente) el intercambio de información entre ellos. Según como se realice este intercambio, se tienen distintos modelos de programación paralela:

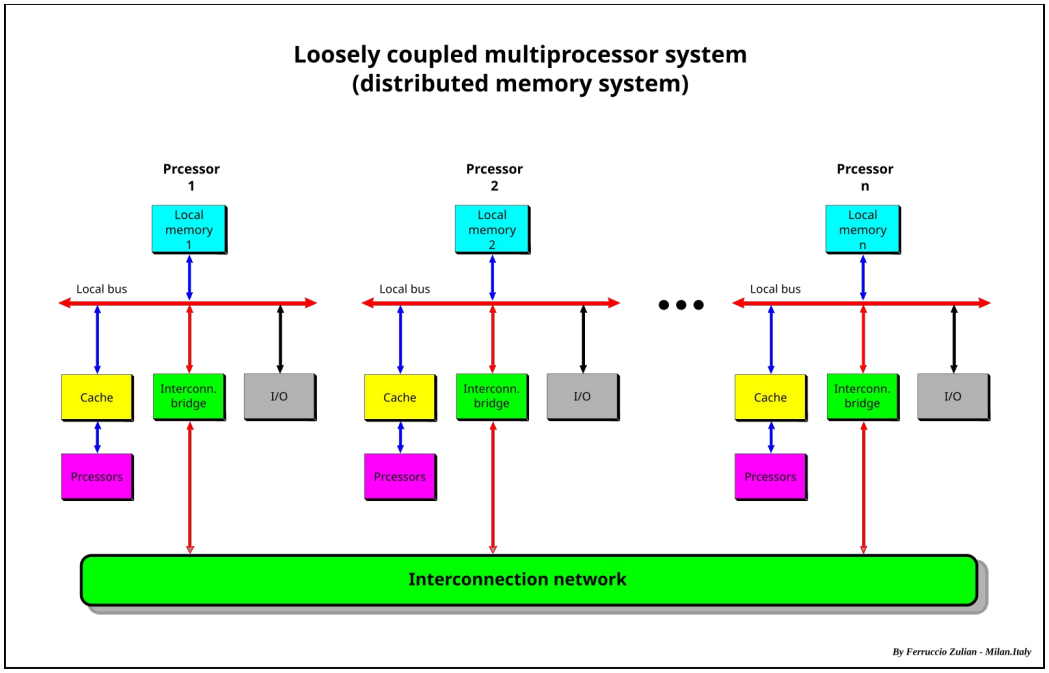
- **Modelo de memoria compartida (sin hilos)**

  Los procesadores comparten físicamente la memoria, es decir, todos acceden al mismo espacio de direcciones. Un valor escrito por un procesador puede ser leído por otro directamente. Por tanto, la comunicación entre procesos se realizará a través de **variables compartidas**.



- **Modelo de paso de mensajes (memoria distribuida)**

  Cada procesador dispone de su propia memoria independiente del resto y solo accesible por él. Por tanto, para realizar el intercambio de información es necesario que cada procesador le pida los datos que necesita al procesador que los tiene, haciendo éste el envío correspondiente. Este modelo se usa en **sistemas distribuidos** y la comunicación se realiza mediante **paso de mensajes**.



- **Modelo basado en hilos (memoria compartida con hilos)**

  Dentro de un mismo proceso los hilos comparten la memoria y los recursos. Además, los hilos pueden ejecutarse en núcleos separados dentro de la misma máquina.

- **Modelo híbrido**

  Combinación de los modelos de memoria compartida y distribuida.

- **Modelo SPMD (Single Program Multiple Data)**

  Un solo programa se ejecuta en múltiples procesadores con diferentes datos.

- **Modelo MPMD (Multiple Program Multiple Data)**

  Diferentes programas se ejecutan en paralelo con diferentes conjuntos de datos.

4.1. Problemas inherentes a la programación concurrente

A la hora de crear un programa concurrente hay que tener presente dos problemas que pueden presentarse:

- **Exclusión mutua**

  En programación concurrente es muy común que varios procesos accedan a la vez a la misma variable compartida para actualizarla. Esto debe evitarse ya que de no hacerlo, provocará inconsistencia de datos: uno puede estar actualizando la variable a la vez que otro la puede estar

leyendo. Por ello debe garantizarse la **exclusión mutua** mediante el establecimiento de **regiones críticas** que garantizará que solo uno de los procesos pueda acceder a la variable mientras el resto deben esperar un tiempo finito.

- **Condición de sincronización**

  Consiste en la necesidad de coordinar procesos para sincronizar sus actividades. Puede suceder que un proceso P1 llegue a una determinada sentencia y no puede continuar su ejecución porque necesita que otro proceso P2 haya realizado una determinada tarea. Para esto la programación concurrente proporciona mecanismos de bloqueo de procesos a la espera de que ocurra un determinado suceso, y mecanismos de desbloqueo una vez haya ocurrido el suceso.

4.2. Condiciones de Bernstein

En los programas secuenciales siempre hay un orden fijo de ejecución de las instrucciones. En cambio, en los programas concurrentes, al haber solapamiento de instrucciones no se sabe cuál va a ser el orden de ejecución, pudiendo ocurrir que ante unos mismos datos de entrada el resultado obtenido no sea el mismo ya que el flujo de ejecución ha cambiado.

Esto da lugar a que los programas concurrentes tengan un comportamiento indeterminista donde repetidas ejecuciones del mismo programa sobre un mismo conjunto de datos pueda generar diferentes resultados de salida.

Si se observa el siguiente código, se puede apreciar el orden en el que se ejecuten las instrucciones no influye en el resultado final (valor de las variables). Por tanto, se pueden ejecutar las tres sentencias a la vez incrementando la velocidad de procesamiento.

```
x = 1;
y = 2;
z = 3;
```

Sin embargo, para el siguiente código, queda claro que la primera instrucción se debe ejecutar antes que la segunda para que el resultado sea siempre el mismo (para los mismos datos de entrada).

```
x = x + 1;
y = x + 1;
```

Bernstein estableció unas serie de condiciones para que dos conjuntos de instrucciones se puedan ejecutar de forma concurrente.

Para ello es necesario formar dos conjuntos de instrucciones:

1. Conjunto de lectura: Aquellas sentencias que cuentan con variables a las que se accede en modo lectura. $L(S_k) = \{a_1, a_2, a_3, ...\}$

2. Conjunto de escritura: Aquellas sentencias que cuentan con variables a las que se accede en modo escritura. $E(S_k) = \{b_1, b_2, b_3, ...\}$

Para que dos conjuntos de instrucciones se puedan ejecutar de forma concurrente se deben cumplir tres condiciones:

1. La intersección entre las variables leídas por un conjunto de instrucciones y las variables escritas por otro conjunto debe ser vacía.

$$L(S_i) \cap E(S_j) = \emptyset$$

2. La intersección entre las variables escritas por un conjunto de instrucciones y las variables leídas por otro conjunto debe ser vacía.

$$E(S_i) \cap L(S_j) = \emptyset$$

3. La intersección entre las variables escritas por un conjunto de instrucciones y las variables escritas por otro conjunto debe ser vacía.

$$E(S_i) \cap E(S_j) = \emptyset$$

Por ejemplo, dada las sentencias indicadas a continuación, ¿cuales de ellas se pueden ejecutar concurrentemente?

```
a = x + y;
b = z - 1;
c = a - b;
w = c + 1;
```

Primero deberíamos obtener los conjuntos L y E para cada sentencia

$L(S_1) = \{x, y\}$ $E(S_1) = \{a\}$

$L(S_2) = \{z\}$ $E(S_2) = \{b\}$

$L(S_3) = \{a, b\}$ $E(S_3) = \{c\}$

$L(S_4) = \{c\}$ $E(S_4) = \{w\}$

Y ahora aplicarlas entre cada par de sentencias:

S_1 y S_2

$L(S_1) \cap E(S_2) = \emptyset$ $E(S_1) \cap L(S_2) = \emptyset$ $E(S_1) \cap E(S_2) = \emptyset$ // Sí se pueden ejecutar concurrentemente

S_1 y S_3

$L(S_1) \cap E(S_3) = \emptyset$ $E(S_1) \cap L(S_3) = \{a\} \neq \emptyset$ $E(S_1) \cap E(S_3) = \emptyset$ // NO se pueden ejecutar concurrentemente

S_1 y S_4

$L(S_1) \cap E(S_4) = \emptyset$ $E(S_1) \cap L(S_4) = \emptyset$ $E(S_1) \cap E(S_4) = \emptyset$ // Sí se pueden ejecutar concurrentemente

S_2 y S_3

$L(S_2) \cap E(S_3) = \emptyset$ $E(S_2) \cap L(S_3) = \{b\} \neq \emptyset$ $E(S_2) \cap E(S_3) = \emptyset$ // NO se pueden ejecutar concurrentemente

S_2 y S_4

$L(S_2) \cap E(S_4) = \emptyset$ $E(S_2) \cap L(S_4) = \emptyset$ $E(S_2) \cap E(S_4) = \emptyset$ // Sí se pueden ejecutar concurrentemente

S_3 y S_4

$L(S_3) \cap E(S_4) = \emptyset$ $E(S_3) \cap L(S_4) = \{c\} \neq \emptyset$ $E(S_3) \cap E(S_4) = \emptyset$ // NO se pueden ejecutar concurrentemente

5. Programación paralela

Paradigma de programación enfocado a desarrollar sistemas distribuidos, escalables y tolerantes a fallos.

Los sistemas distribuidos son aquellos en los que los componentes hardware y software localizados en ordenadores unidos por una red, se comunican y coordinan mediante el paso de mensajes.

Una arquitectura típica para el desarrollo de sistemas distribuidos es la arquitectura **cliente-servidor**.