PSP

Programación Concurrente

Unidad 1

Jesús Alberto Martínez
versión 0.1

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

Importante	Atención	Interesante

Unidad 1. Programación Concurrente

1 Introducción ... 3
1.1 Objetivos ... 3
2 Procesos, Programas, hilos ... 4
2.1 Procesos y programas ... 4
2.2 Programación concurrente ... 5
¿Para qué? ... 6
Comunicación y sincronización entre procesos ... 6
2.3 Servicios e hilos ... 7
Programa secuencial (Arquitectura Von Newmann) ... 8
Programa concurrente ... 9
Hilos vs procesos ... 11
3 Concurrencia ... 14
3.1 Concurrencia vs Paralelismo ... 14
Monoproceso ... 14
Multiprogramación ... 14
Paralelismo ... 15
3.2 Sistemas distribuidos ... 16
3.3 Ventajas e inconvenientes ... 16
3.4 Condiciones de Bernstein ... 18
4 Procesos en el Sistema Operativo ... 20
4.1 El kernel del SO ... 20
4.2 Control de procesos en GNU/Linux ... 21
Comandos para saber el pid de los procesos ... 22
Comandos para ver los procesos activos ... 22
Control de procesos ... 23
4.3 Estados de un proceso ... 24
4.4 Planificación de procesos ... 26
4.5 Algoritmos de planificación de procesos ... 28
FCFS - First Come First Served ... 28
SJF - Shortest Job First ... 29
Planificación por prioridad ... 29
Round Robin ... 30
Procesos con operaciones de E/S o bloqueos ... 31

1 Introducción

En este primer tema vamos a conocer los conceptos básicos relacionados con la programación concurrente, así como la mayoría de la terminología que vamos a trabajar y utilizar durante todo el curso.

En un mundo en el que cada vez los dispositivos electrónicos son cada vez más potentes, y veloces, el software debe ser capaz de aprovechar las características que le ofrecen tanto el hardware como los sistemas operativos.

Son muchas las tareas que requieren de un procesamiento rápido de cantidades ingentes de datos. Un par de ejemplos los tenemos en las aplicaciones `Big Data` e `Inteligencia artificial`. Estos dos campos son unos de los máximos exponentes en cuanto a programación concurrente.

> 📣 ¿Qué es para ti concurrencia?

1.1 Objetivos

Los objetivos que alcanzaremos tras esta unidad son:

- Diferenciar entre programa y proceso
- Comprender qué es la concurrencia
- Conocer el concepto, diferencias y relación existente entre las dos unidades básicas de ejecución: procesos e hilos.
- Tener nociones sobre programación concurrente
- Entender el funcionamiento concurrente del SO y del hardware

2 Procesos, Programas, hilos

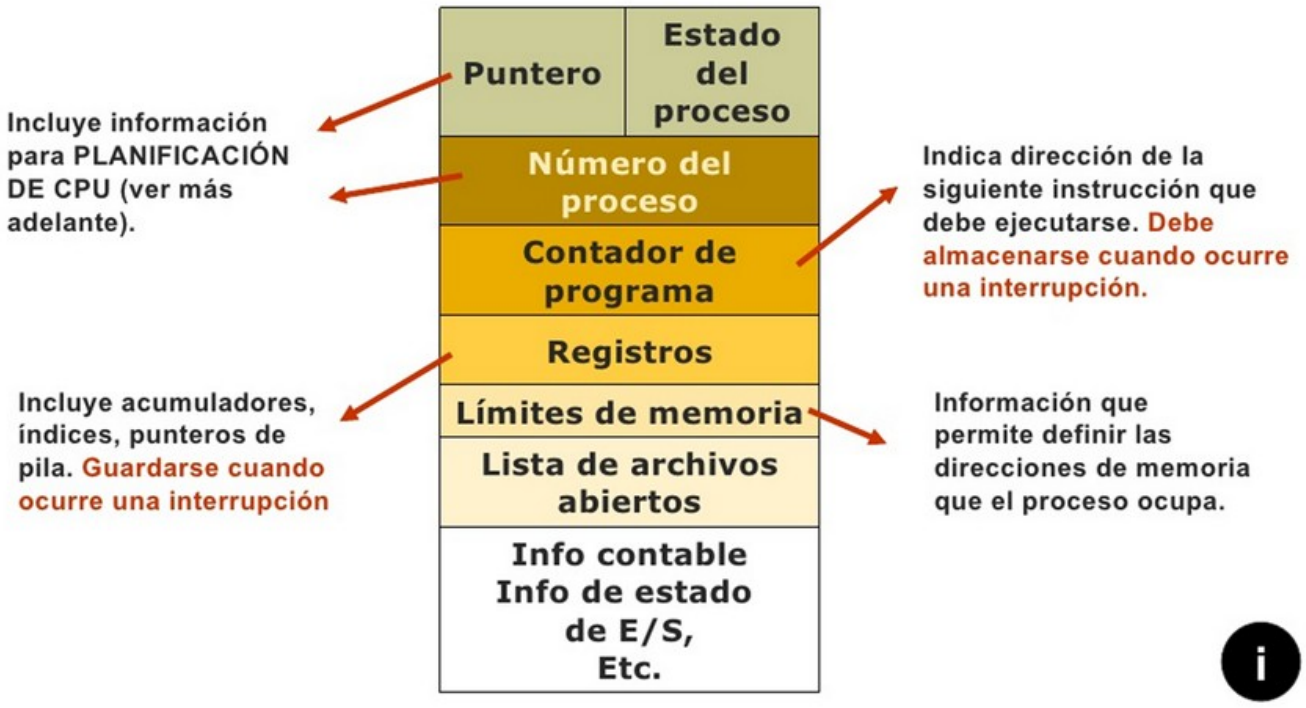
2.1 Procesos y programas

Un `programa` no es más que un conjunto de instrucciones u órdenes que le indican a un dispositivo qué acciones debe realizar con los datos recibidos.

> **Caja negra**
>
> Según la visión de un sistema como caja negra, un programa le indica al sistema cómo obtener unos datos de salida a partir de unos datos de entrada.

Sin embargo, un `proceso` es un programa en ejecución. Esto es, un proceso es una entidad activa y un programa es una entidad pasiva.

El proceso, por tanto, está representado por el contador del programa, el valor de los registros, la pila, el *código ejecutable*, su estado, ... todo lo necesario para la ejecución del programa por parte del SO.



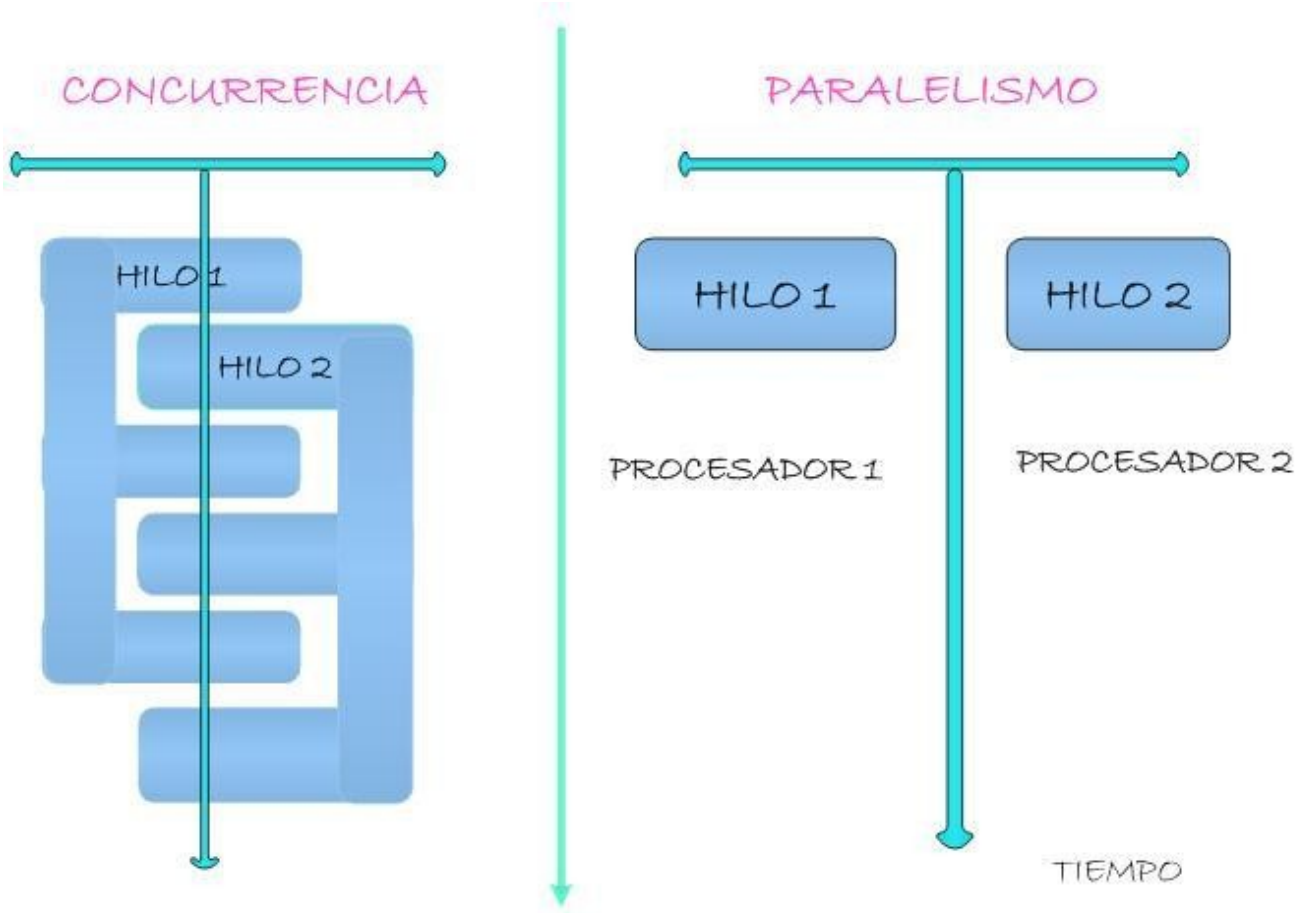
Cada proceso es una entidad independiente. Cuando un programa se ejecuta, el sistema operativo crea un proceso. Si ese mismo programa se vuelve a ejecutar, se crearía un proceso

distinto, teniendo en memoria dos instancias del mismo programa. Pero es importante recalcar que los dos procesos son independientes.

2.2 Programación concurrente

Podemos decir que dos procesos son concurrentes cuando la primera instrucción de uno de los procesos se ejecuta después de la primera y antes de la última de otro proceso.

La planificación alternando los instantes de ejecución, **`multitarea`**, hace que los procesos se ejecuten de forma concurrente. También la disponibilidad de varias unidades de proceso, **`multiproceso`**, permite la ejecución simultánea o paralela de procesos en el sistema.



> **Concurrencia**
>
> A los dos escenarios descritos anteriormente es a lo que vamos a denominar, de forma general, **concurrencia**.

¿Para qué?

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

- Optimizar la utilización de los recursos: Podremos simultanear las operaciones de E/S en los procesos. La CPU estará menos tiempo ociosa.
- Proporcionar interactividad a los usuarios (y animación gráfica).
- Mejorar la disponibilidad: Servidor que no realice tareas de forma concurrente, no podrá atender peticiones de clientes simultáneamente.
- Conseguir un diseño conceptualmente más comprensible y mantenible: El diseño concurrente de un programa nos llevará a una mayor modularidad y claridad.
- 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.

Además, 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.
- Entornos distribuidos y servicios cloud.

Comunicación y sincronización entre procesos

Cuando varios procesos se ejecutan concurrentemente puede haber procesos que colaboren para un determinado fin mientras que puede haber otros que compitan por los recursos del sistema.

En ambos casos se hace necesaria la introducción de mecanismos de comunicación y sincronización entre procesos.

> **Programación concurrente**
>
> Precisamente del estudio de esos **mecanismos de sincronización y comunicación** trata la programación concurrente y este módulo de PSP.

Si pensamos en la forma en la que un proceso puede comunicarse con otro, se nos plantean estas dos:

- Intercambio de mensajes: Es la forma que se utiliza habitualmente cuando los procesos se encuentran en máquinas diferentes. Los procesos intercambian información siguiendo un protocolo previamente establecido.
- Recursos (o memoria) compartidos: Sólo se puede utilizar cuando los dos procesos se encuentran en la misma máquina y permite la sincronización de los procesos en función del valor o estado de un recurso compartido.

También podemos ver el tipo de comunicación en función de la sincronía que mantengan los procesos durante la comunicación:

- Síncrona: El emisor queda bloqueado hasta que el receptor recibe el mensaje. Ambos se sincronizan en el momento de la recepción del mensaje.
- Asíncrona: El emisor continúa con su ejecución inmediatamente después de emitir el mensaje, sin quedar bloqueado.

2.3 Servicios e hilos

Un programa, como ya hemos dicho, se compone de un conjunto de sentencias (operaciones y verificaciones) y un flujo de ejecución. La línea de flujo de control establece, de acuerdo con la estructura del propio programa y de los datos que maneja, el orden en que deben ejecutarse las sentencias.

Atendiendo al número de líneas de flujo de control que tiene un programa, los procesos pueden ser:

- Secuenciales: Poseen un único flujo de control (monohilo)
- Concurrentes: Poseen varios hilos de ejecución (multihilo).

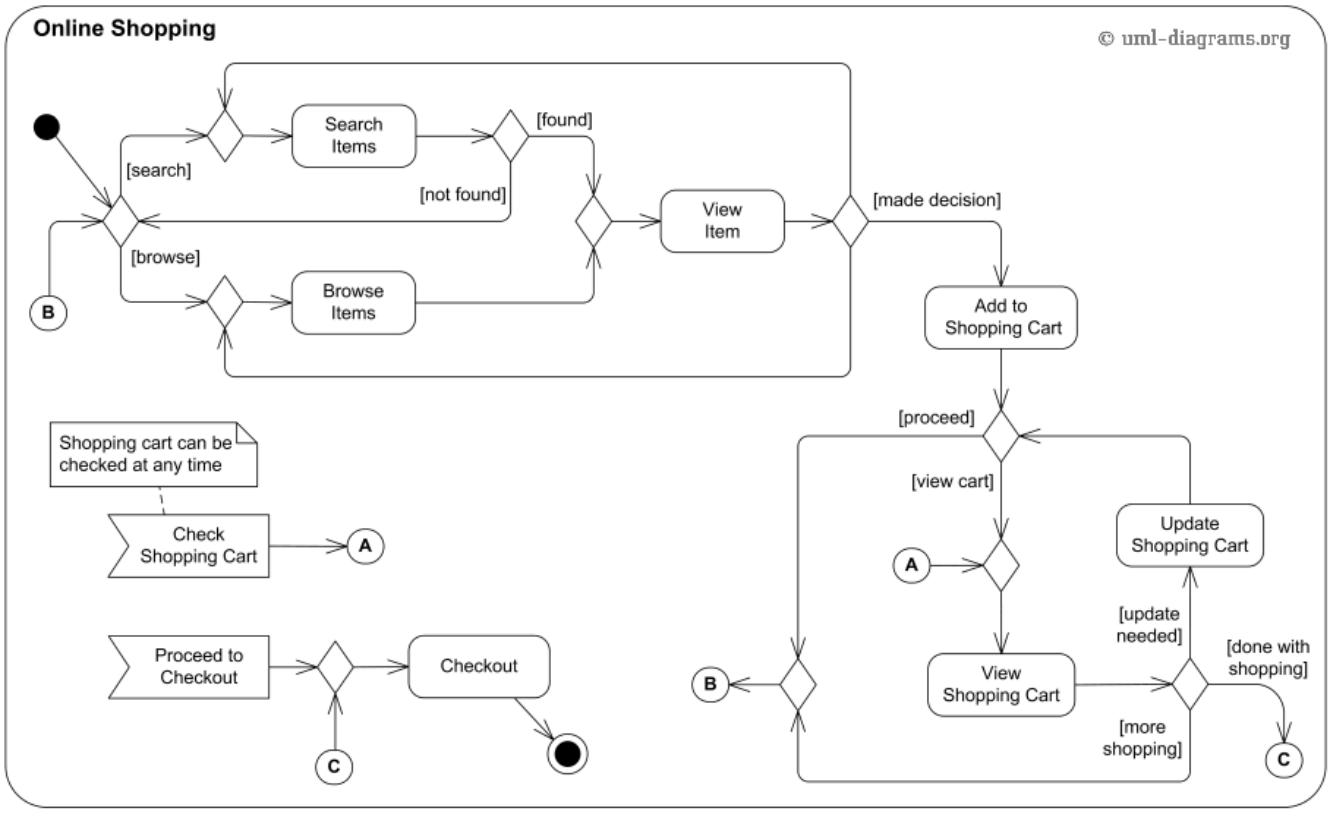
Programa secuencial (Arquitectura Von Newmann)

Cuando empezamos a programar, usamos el estilo de programación clásico, en el que se sigue el modelo conceptual de Von Newmann

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 (determinado por el conjunto de entradas que recibe).

En los programas secuenciales, el tiempo que tarda cada operación en ejecutarse no tiene consecuencias sobre el resultado.



La comprobación del correcto funcionamiento (`verificación` o `depuración`) de un programa secuencial es sencilla:

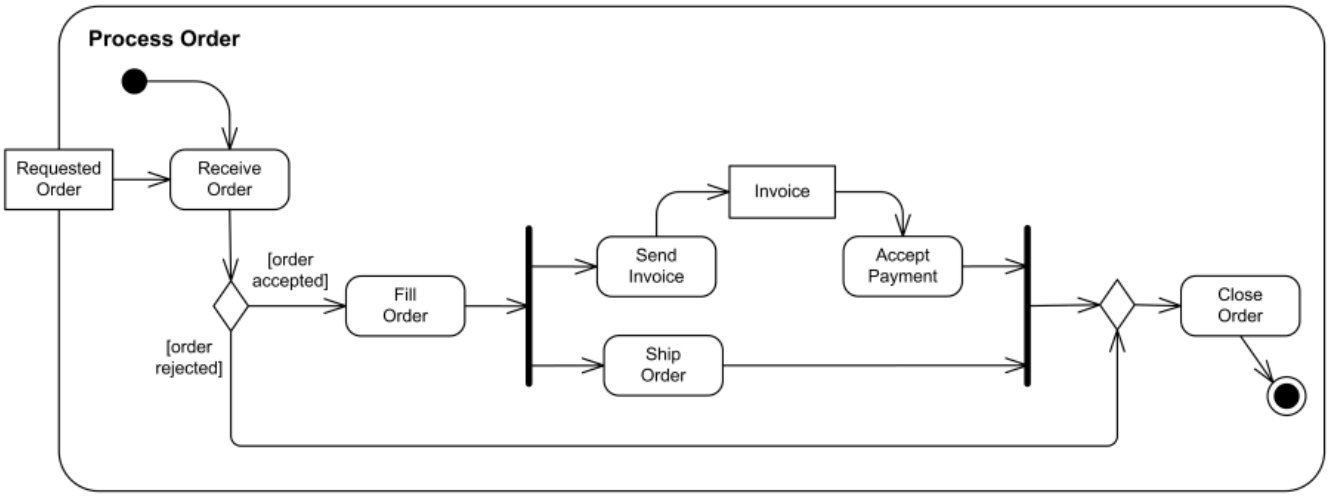
- Cada sentencia produce la respuesta correcta.
- Las sentencias se ejecutan en el orden esperado.

De aquí surgen algunos de los métodos básicos de pruebas de sistemas, como el de *caja blanca*.

Programa concurrente

En los programas concurrentes existen múltiples líneas de control de flujo. Las sentencias que constituyen el programa no se ejecutan siguiendo un órden que corresponda a una secuencia temporal lineal.

En los programas concurrentes el concepto de secuencialidad entre sentencias continua siendo muy importante. Sin embargo en los programas concurrentes es de orden parcial, mientras que, tal y como hemos comentado anteriormente, en los programas secuenciales era de orden estricto.



En los programas concurrentes la *secuencialización* entre procesos concurrentes se llama **sincronización**.

Este orden parcial implica que los programas concurrentes no tienen porqué ser deterministas, es decir, que ante el mismo conjunto de datos de entrada no siempre se va a obtener el mismo resultado.

> **Indeterminismo**
>
> Que existan diferentes salidas para el mismo conjunto de datos de entrada no significa necesariamente que un programa concurrente sea incorrecto.

Si observamos el siguiente ejemplo de pseudocódigo

```
1  public class TestClass {
2      int x;
3
4      public void testMethod1() {
5          for (int i=1; i <= 5; i++) {
6              x++;
7          }
8      }
9      public void testMethod2() {
10         for (int j=1; j <= 5; j++) {
11             x++;
12         }
13     }
14     public void sequential() {
15         x = 0;
16         testMethod1();
17         testMethod2();
18         System.out.println(x);
19     }
20     public void parallel() {
21         x = 0;
22         // cobegin-coend means that both methods are run simultaneously
23         // These sentences doesn't exist in Java. They are used for
24         // sample purposes
25         cobegin
26             testMethod1();
27             testMethod2();
28         coend
29         System.out.println(x);
30     }
31 }
```

¿Qué valor tendrá x tras ejecutar el método sequential?
¿Qué valor tendrá x tras ejecutar el método parallel?

Reseña histórica

La naturaleza y los modelos de interacción entre procesos de un programa concurrente fueron estudiados y descritos por **Dijkstra** (1968), Brinch **Hansen** (1973) y **Hoare** (1974).

Estos trabajos constituyeron los principios en que se basaron los sistemas operativos multiproceso de la década de los 70 y 80.

El indeterminismo inherente a los programas concurrentes hace que su análisis y validación sea más complejo. No obstante, para la comprobación del correcto funcionamiento (**verificación** o **depuración**) de un programa concurrente se requiere comprobar los mismos aspectos que en los programas secuenciales, pero con los siguientes nuevos aspectos:

- Las sentencias se pueden validar individualmente solo si no están acopladas por variables compartidas.
- Cuando existen variables compartidas, los efectos de interferencia entre las sentencias concurrentes pueden ser muy variados y la validación es muy difícil. `cuidado`
- Siempre que la secuencialidad entre tareas se lleve a cabo por sentencias explícitas de **sincronización**, el tiempo es un elemento que no influye sobre el resultado

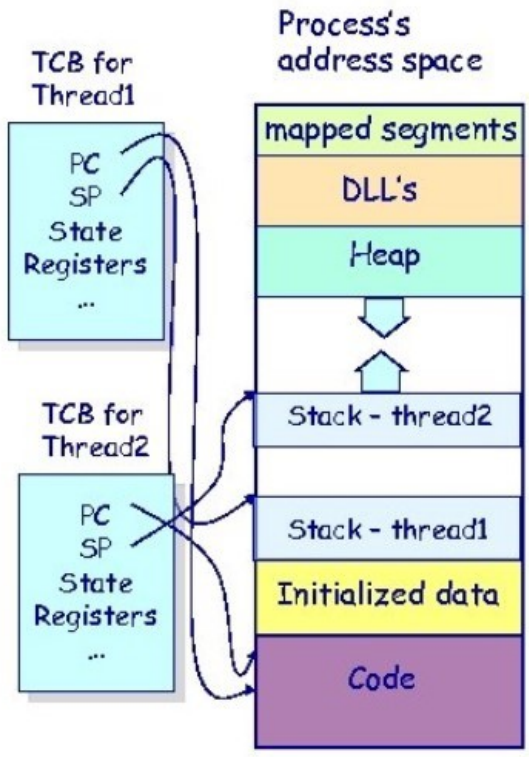
> **Importante**
>
> Estos tres aspectos que se acaban de describir forman la base de toda la programación concurrente.
>
> 👁 Conocerlos, entenderlos y saber aplicarlos es a lo que dedicaremos una parte importante de este curso.

Hilos vs procesos

Un hilo no es más que cada una de esas líneas de flujo que puede tener un proceso ejecutándose de forma concurrente. Un procesos es una unidad pesada de ejecución.

Así, un proceso estará formado por, al menos, un hilo de ejecución, el hilo principal. Si el proceso tiene varios hilos, cada uno es una unidad de ejecución ligera.

Procesos	Hilos
Constan de uno o más hilos	Un hilo siempre existe dentro de un proceso
Son independientes unos de otros	Comparten los recursos del proceso de forma directa
Son gestionados por el SO	Son gestionados por el proceso
Se comunican a través del SO	La comunicación la controla el proceso



En la imagen anterior se puede observar la relación existente entre la creación de un thread y la de su proceso asociado.

- El proceso define un espacio de memoria en el que reside. Los hilos comparten ese espacio de memoria. Dentro de ese espacio de memoria cada hilo tiene su espacio reservado, pero todos pueden compartir la memoria global del proceso y los recursos (ficheros, sockets, etc.) que el proceso tenga abiertos.
- Como ya hemos visto, cada proceso tiene su PCB con información relativa al proceso.
- Los hilos, de forma similar, tienen su TCB (Thread Control Block) en el que guardan la información específica de cada hilo (Contador de programa, puntero a la pila, estado del thread, registros y un puntero al PCB).

Servicios

Un servicio es un proceso que, normalmente, es cargado durante el arranque del sistema operativo. Al no necesitar interacción con el usuario, los servicios suelen ejecutarse en forma de **demonios*, quedando su ejecución en *segundo plano*.

Recibe el nombre de servicio ya que es un proceso que queda a la espera de que otro le pida que realice una tarea. Como deben atender las solicitudes de varios procesos, los servicios suelen ser programas multihilo.

3 Concurrencia

Según el diccionario de la RAE una de las aceptaciones de concurrencia es

> Coincidencia, concurso simultáneo de varias circunstancias.

Si cambiamos circunstancias por `procesos`, ya tendríamos una definición cercana a lo que significa concurrencia en el mundo digital

Si nos fijamos, no es la primera vez que surge la palabra `proceso` en este texto, y es que los procesos son una pieza fundamental del puzle, por no decir la parte más importante.

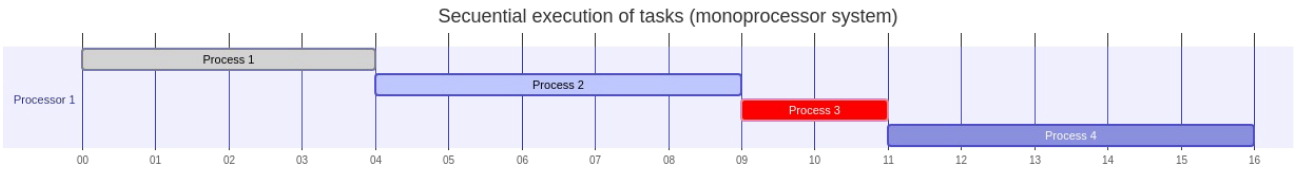
3.1 Concurrencia vs Paralelismo

Ahora que ya sabemos qué es un proceso, vamos a ver la relación que éstos tienen con el hardware en el que se ejecutan.

Monoproceso

Por mucho que tengamos varios procesos procesos ejecutándose a la vez, si sólo tenemos un microprocesador para atenderlos a todos, estas tareas nunca van a poder ejecutarse a la vez.

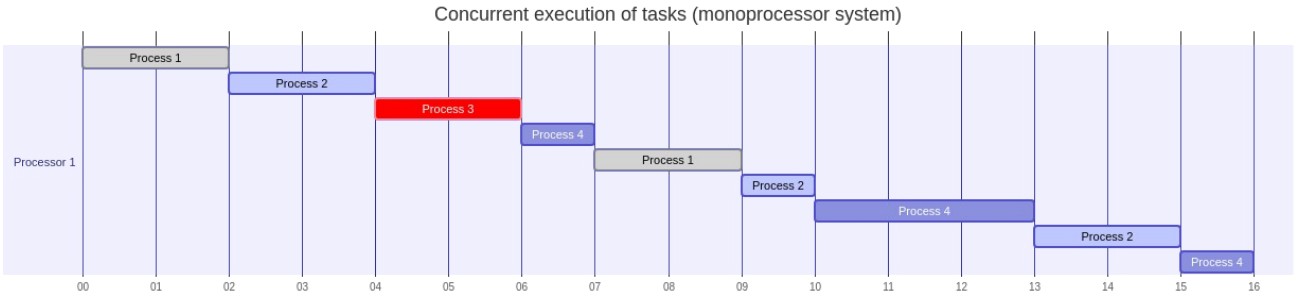
Una posibilidad sería la ejecución secuencias de las tareas en el sistema. Se empieza a ejecutar una tarea y, hasta que esta no finaliza, el sistema no empieza a ejecutar la siguiente. Esto se correspondería con sistemas que sólo son capaces de hacer una tarea a la vez, algo raro hoy en día.



Multiprogramación

Para que los procesos no tengan que esperar a que todos los demás se ejecuten, los sistemas aprovechan y exprimen los recursos al máximo, permitiendo la ilusión de que varios procesos se ejecutan de forma simultánea. Esto es lo que se conoce como `multitarea`.

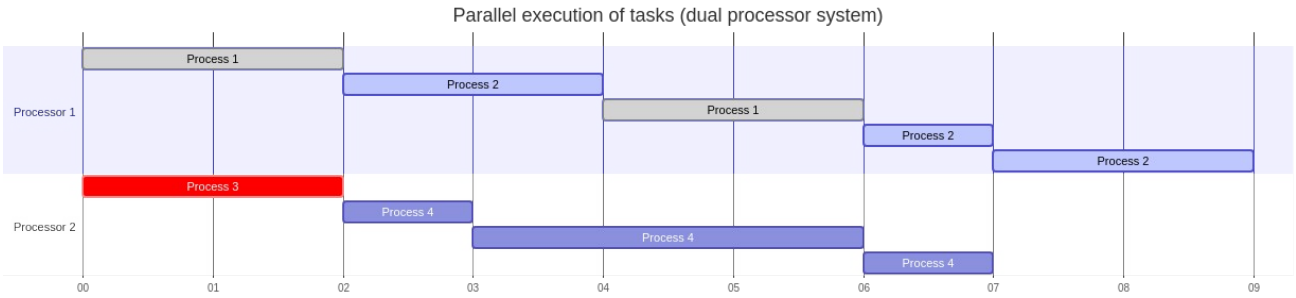
En estos sistemas, se aprovecha el diseño de los sistemas operativos modernos, y de las operaciones que realizan los procesos que no requieren el uso del procesador (esperar a una operación de E/S, una interacción con el usuario, la recepción de información desde la red, etc.) para poder ejecutar otros procesos. La ejecución se multiplexa en el tiempo.



Como se puede observar en las dos imágenes anteriores (aunque se trata sólo de un modelo), el tiempo de uso total del procesador es igual en ambos casos, es decir, que el sistema tardará el mismo tiempo en completar todas las tareas. Sin embargo, la sensación es que todas las tareas se están realizando a la vez.

Paralelismo

Con el avance de la tecnología ahora la gran mayoría de dispositivos, desde los equipos de escritorio, portátiles, dispositivos móviles, ... hasta los dispositivos IoT, 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 es a lo que llamamos `paralelismo`.



En este caso, a mayor número de unidades de proceso, menor tiempo tardarán las tareas en completarse y mayor será la sensación de rapidez que notará el usuario. Este es uno de los retos de los sistemas operativos, planificar adecuadamente las tareas para minimizar los tiempos de ejecución, de espera y el uso de los recursos del sistema, el procesador principalmente.

? núcleos vs hilos

Si habéis comprado un procesador hace poco, o estáis al día en cuanto al hardware, sabréis que una de las características de los procesadores es su **número de núcleos** (4, 8, 16).
Pero además, al número de núcleos lo acompaña otra característica que es el número de **hilos o threads**, que suele ser el doble que el de núcleos.
¿Qué implicación tienen los threads de un procesador con respecto a la concurrencia? ¿Si un equipo tiene 8 núcleos / 16 hilos significa eso que puede ejecutar 16 procesos a la vez?

3.2 Sistemas distribuidos

> "Un sistema distribuido es una colección de computadores independientes que aparecen ante los usuarios como un único sistema coherente"
>
> "Andrew S. Tanembaum"

Posiblemente el ejemplo más famoso y conocido de sistema distribuido sea `Internet`. Internet aparece ante los usuarios como un enorme repositorio de documentos, es decir, como un único sistema capaz de proveer casi cualquier tipo de información o servicio que se necesite. No obstante, sabemos que está compuesta por millones de equipos ubicados en localizaciones diferentes e interconectados entre sí.

Nace de la necesidad de compartir recursos. Actualmente el máximo exponente de este tipo de sistemas es el `Cloud Computing` o servicios en la nube. Un sistema es distribuido cuando los componentes software están distribuidos en la red, se comunican y coordinan mediante el paso de mensajes.

Las características de este tipo de sistemas son::

- Concurrencia: ejecución de programas concurrentes.
- Inexistencia de un reloj global. Implica sincronizarse con el paso de mensajes.
- Fallos independientes: cada componente del sistema puede fallar sin que perjudique la ejecución de los demás.

3.3 Ventajas e inconvenientes

Ventajas del procesamiento paralelo:

- Ejecución simultánea de tareas.
- Disminuye el tiempo total de ejecución
- Resuelve problemas complejos y de grandes dimensiones.
- Utilización de recursos no locales distribuidos en la red
- Disminución de costos, aprovechando los recursos distribuidos, no es necesario gastar en un único supercomputardor, se puede alcanzar el mismo poder de computación con equipos más modestos distribuidos.

Inconvenientes del procesamiento paralelo:

- Los compiladores y entornos de programación para sistemas paralelos son más complicados de desarrollar.
- Los programas paralelos son más difíciles de escribir
- Hay mayor consumo de energía
- Mayor complejidad en el acceso a datos
- Complejidad a la hora de la comunicación y sincronización de las diferentes subtareas. **cuidado**

Ventajas de la programación distribuida

- Se comparten recursos y datos
- Crecimiento bajo demanda
- Mayor flexibilidad para distribuir la carga
- Alta disponibilidad
- Soporte de aplicaciones distribuidas
- Filosofía abierta y heterogénea

> **? Escalado de sistemas**
> Con escalado nos referimos a la posibilidad de incrementar las capacidades de un sistema. Investiga las diferencias, ventajas e inconvenientes del `escalado horizontal` y el `escalado vertical`.

Inconvenientes de la programación distribuida

- Aumenta la complejidad
- Se necesita software nuevo especializado
- Problemas derivados de las comunicaciones (perdidas, saturaciones, etc.)
- Problemas de seguridad, ataques DDoS

Ejemplos de utilización de la programación paralela y distribuida

- Estudios meteorológicos
- Estudios del genoma humano
- Modelado de la biosfera
- Predicciones sísmicas
- Simulación de moléculas

> **Ejemplo de programación paralela y distribuida**
>
> Búsqueda de inteligencia extraterrestre - Proyecto SETI

3.4 Condiciones de Bernstein

Una vez que sabemos qué es un programa concurrente y las distintas arquitecturas hardware que pueden soportarlo, vamos a ver qué partes de un programa se pueden ejecutar de forma concurrente y cuáles no.

Si observamos 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).

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

Sin embargo, en un código como el siguiente el órden en el que se ejecuten las instrucciones no influye en el resultado final (valor de las variables). En este caso se pueden ejecutar las tres sentencias a la vez incrementando la velocidad de procesamiento.

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

A.J. Bernstein definió unas condiciones para determinar si dos conjuntos de instrucciones Si y Sj se pueden ejecutar concurrentemente.

Para poder determinar si dos conjuntos de instrucciones se pueden ejecutar concurrentemente, se definen en primer lugar los siguientes conjuntos

- L(Sk) = {a1, a2, a3, ...} como el conjunto de lectura formado por todas las variables cuyos valores se leen durante la ejecución de las instrucciones del conjunto k.
- E(Sk) = {b1, b2, b3, ...} como el conjunto de escritura formado por todas las variables cuyos valores se actualizan durante la ejecución de las instrucciones del conjunto k.

Para que dos conjuntos de instrucciones Si y Sj se puedan ejecutar concurrentemente, se deben cumplir estas tres condiciones de forma simultánea.

- L(Si) ∩ E(Sj) = ∅
- E(Si) ∩ L(Sj) = ∅
- E(Si) ∩ E(Sj) = ∅

? Cuales de estas instrucciones se pueden ejecutar de forma concurrente

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

`Primero deberíamos obtener los conjuntos L y E para cada sentencia`

> L(S1) = {x, y} E(S1) = {a}
>
> L(S2) = {z} E(S2) = {b}
>
> L(S3) = {a, b} E(S3) = {c}
>
> L(S4) = {c} E(S4) = {w}

Y ahora aplicarlas entre cada par de sentencias

> L(S1) ∩ E(S2) = ∅ E(S1) ∩ L(S2) = ∅ E(S1) ∩ E(S2) = ∅ // Sí se pueden ejecutar concurrentemente

> L(S1) ∩ E(S3) = ∅ E(S1) ∩ L(S3) = {a} ≠∅ E(S1) ∩ E(S3) = ∅ // NO se pueden ejecutar concurrentemente
>
> L(S1) ∩ E(S4) = ∅ E(S1) ∩ L(S4) = ∅ E(S1) ∩ E(S4) = ∅ // Sí se pueden ejecutar concurrentemente
>
> L(S2) ∩ E(S3) = ∅ E(S2) ∩ L(S3) = {b] ≠E(S2) ∩ E(S3) = ∅ // NO se pueden ejecutar concurrentemente
>
> L(S2) ∩ E(S4) = ∅ E(S2) ∩ L(S4) = ∅ E(S2) ∩ E(S4) = ∅ // Sí se pueden ejecutar concurrentemente
>
> L(S3) ∩ E(S4) = ∅ E(S3) ∩ L(S4) = {c} ≠∅ E(S3) ∩ E(S4) = ∅ // NO se pueden ejecutar concurrentemente

4 Procesos en el Sistema Operativo

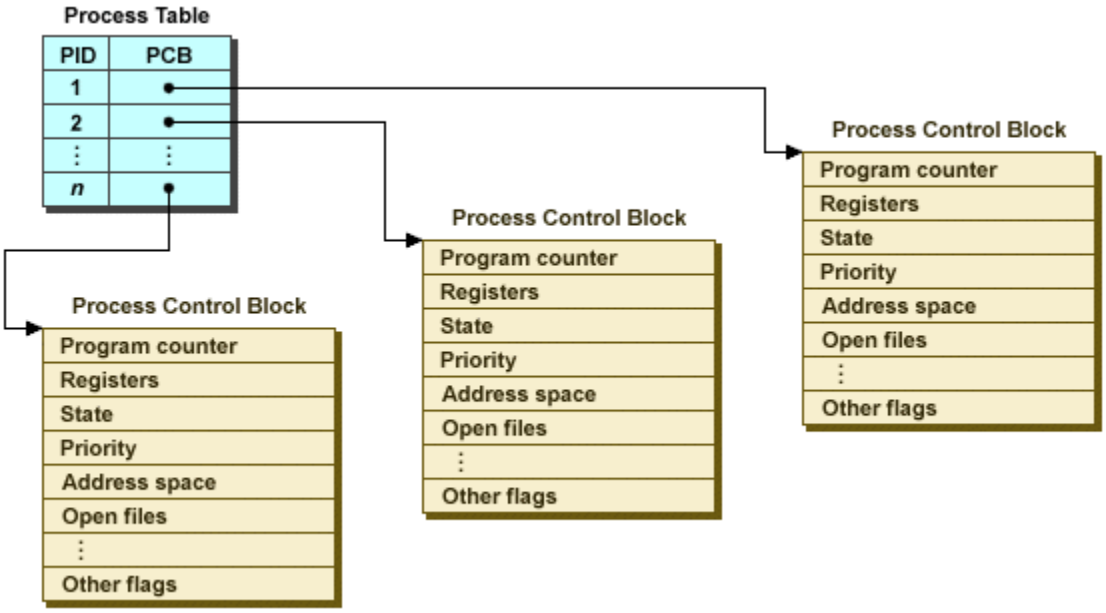
4.1 El kernel del SO

El **`kernel`** o **`núcleo de un SO`** se encarga de la funcionalidad básica del sistema, el responsable de la gestión de los recursos del ordenador, se accede al núcleo a través de las llamadas al sistema, es la parte más pequeña del sistema en comparación con la interfaz. El resto del sistema operativo se le denomina como programas del sistema.

Todos los programas que se ejecutan en el ordenador se organizan como un conjunto de procesos. Es el sistema operativo el que decide parar la ejecución , por ejemplo, porque lleva mucho tiempo en la CPu, y decide cuál será el siguiente proceso que pasará a ejecutarse.

Cuando se suspende la ejecución de un proceso, luego deberá reiniciarse en el mismo estado en el que se encontraba antes de ser suspendido. Esto implica que debemos almacenar en algún sitio la información referente a ese proceso para poder luego restaurarla tal como estaba antes. Esta información se almacena en el **`PCB`** (Bloque de control de procesos).

Estos **`cambios de contexto`**, que es como se conoce al reemplazo de un proceso por otro, son bastante costosos (en tiempo y recursos) por toda la información que hay que guardar. Ya veremos más adelante que existe otra unidad de ejecución, los **`hilos`**, que solucionan en parte este problema.



4.2 Control de procesos en GNU/Linux

Los sistemas Linux identifican a los procesos por su PID (Process ID) así como por su PPID (Parent PID). De esta forma, los procesos pueden clasificarse en:

- Procesos padre: Son procesos que crean otros procesos durante su ejecución
- Procesos hijos: son procesos creados por otros procesos

Cuando se arranca el sistema, el kernel lanza el proceso **init** que es la madre de todos los demás procesos. Al ser el primero que se lanza es el único que no tiene padre. El proceso init se encarga de gestionar todos los demás procesos que se van ejecutando en el SO.

> **proceso init**
>
> El proceso init tiene el pid 1 y, como ya hemos dicho no tiene padre.
>
> Este proceso se utiliza como padre "adoptivo" para todos aquellos procesos que se quedan huérfanos.

Comandos para saber el pid de los procesos

El comando **`pidof cmdname`** nos dice el nombre de todos los procesos asociados a ese comando. Es importante recordar que cada vez que ejecutamos un comando, se crea un nuevo proceso.

Las variables $$ y $PPID nos indican el pid del proceso actual y su ppid respectivamente.

```
# pidof systemd
1
# pidof top
2060
# pidof httpd
03 2102 2101 2100 2099 1076
# Process pid
echo $$
2109
# Process parent pid
echo $PPID
2106
```

Comandos para ver los procesos activos

El principal comando para conocer los procesos que se están ejecutando en un equipo es el comando **ps**. Con este comando podemos ver parte de la información asociada a un proceso.

El comando ps tiene múltiples opciones que nos permiten ver más o menos información de los procesos, así como los procesos de nuestro usuario o del resto de usuarios, estadísticas sobre el uso de recursos de cada proceso, etc.

```
vicente@Desktop-Vicente:~$ ps -AF
UID          PID  PPID  C    SZ   RSS PSR STIME TTY          TIME CMD
root           1     0  0   223   576   5 11:00 ?        00:00:00 /init
root           7     1  0   223    80   3 11:00 ?        00:00:00 /init
root           8     7  0   223    80   1 11:00 ?        00:00:00 /init
vicente        9     8  0  2508  5032   4 11:00 pts/0    00:00:00 -bash
vicente       70     9  0  2650  3224   5 11:06 pts/0    00:00:00 ps -AF
vicente@Desktop-Vicente:~$ ps -auxf
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0    892   576 ?        Sl   11:00   0:00 /init
root           7  0.0  0.0    892    80 ?        Ss   11:00   0:00 /init
root           8  0.0  0.0    892    80 ?        S    11:00   0:00  \_ /init
vicente        9  0.0  0.0  10032  5032 pts/0    Ss   11:00   0:00      \_ -bash
vicente       72  0.0  0.0  10832  3408 pts/0    R+   11:09   0:00          \_ ps -
auxf
```

> 📣 Useful 'ps' examples for Linux process monitoring
>
> https://www.tecmint.com/ps-command-examples-for-linux-process-monitoring/

El otro comando que nos permite ver la información, en este caso en tiempo real, de los procesos que se están ejecutando en l máquina junto con los recursos que están consumiendo, es el comando **`top`**.

```
 1  vicente@Desktop-Vicente:~$ ps -AF
 2  top - 11:14:52 up 14 min,  0 users,  load average: 0.00, 0.00, 0.00
 3  Tasks:   5 total,   1 running,   4 sleeping,   0 stopped,   0 zombie
 4  %Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
 5  MiB Mem :  12677.3 total,  12556.4 free,     70.6 used,     50.3 buff/cache
 6  MiB Swap:   4096.0 total,   4096.0 free,      0.0 used.  12433.8 avail Mem
 7
 8    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 9  root      20   0     892    576    516 S   0.0   0.0   0:00.04 init
10  root      20   0     892     80     20 S   0.0   0.0   0:00.00 init
11  root      20   0     892     80     20 S   0.0   0.0   0:00.01 init
12  vicente   20   0   10032   5032   3324 S   0.0   0.0   0:00.11 bash
13  vicente   20   0   10856   3664   3148 R   0.0   0.0   0:00.00 top
```

> 📣 'top' examples in Linux
>
> https://www.tecmint.com/12-top-command-examples-in-linux/

Control de procesos

Linux tiene varios comandos para controlar los procesos, entre los que cabe destacar el comando **`kill`**.

La forma de controlar los procesos es enviándoles señales. Hay multitud de señales que se pueden enviar a un proceso. Sin embargo, para responder a una señal, los procesos deben estar programados para gestionarlas.

```
1  # Get Firefox PID after it freezes
2  $ pidof firefox
3  2687
4  # Send the SIGKILL (9) signal to end the process immediately
5  $ kill 9 2687
```

> How to control Linux process Using kill, pkill and kilall
>
> https://www.tecmint.com/how-to-kill-a-process-in-linux/

Otra forma de influir en la ejecución de los procesos es mediante la prioridad. En los sistemas Linux todos los procesos tienen una cierta prioridad. Esto influye a la hora de obtener tiempo de CPU por lo que podemos conseguir que un proceso se ejecute más o menos rápido que los demás.

Un usuario con privilegios de ***root*** puede modificar los valores de prioridad de los procesos. Este valor lo podemos ver en la columna NI (nice) del comando top. Este valor influye en la columna PR que indica la prioridad que le da el sistema a un proceso.

El rango de asignación de prioridad disponible es de -20 a 19 , siendo -20 la mayor prioridad y 19 la menor. Con el comando **`nice`** podemos asegurarnos que en momentos de usos elevados de CPU los procesos adecuados reciban el mayor % de la misma.

```
1  vicente@Desktop-Vicente:~$ nice
2  0
3  vicente@Desktop-Vicente:~$ nice -n 10 bash
4  vicente@Desktop-Vicente:~$ nice
5  10
6  vicente@Desktop-Vicente:~$
```

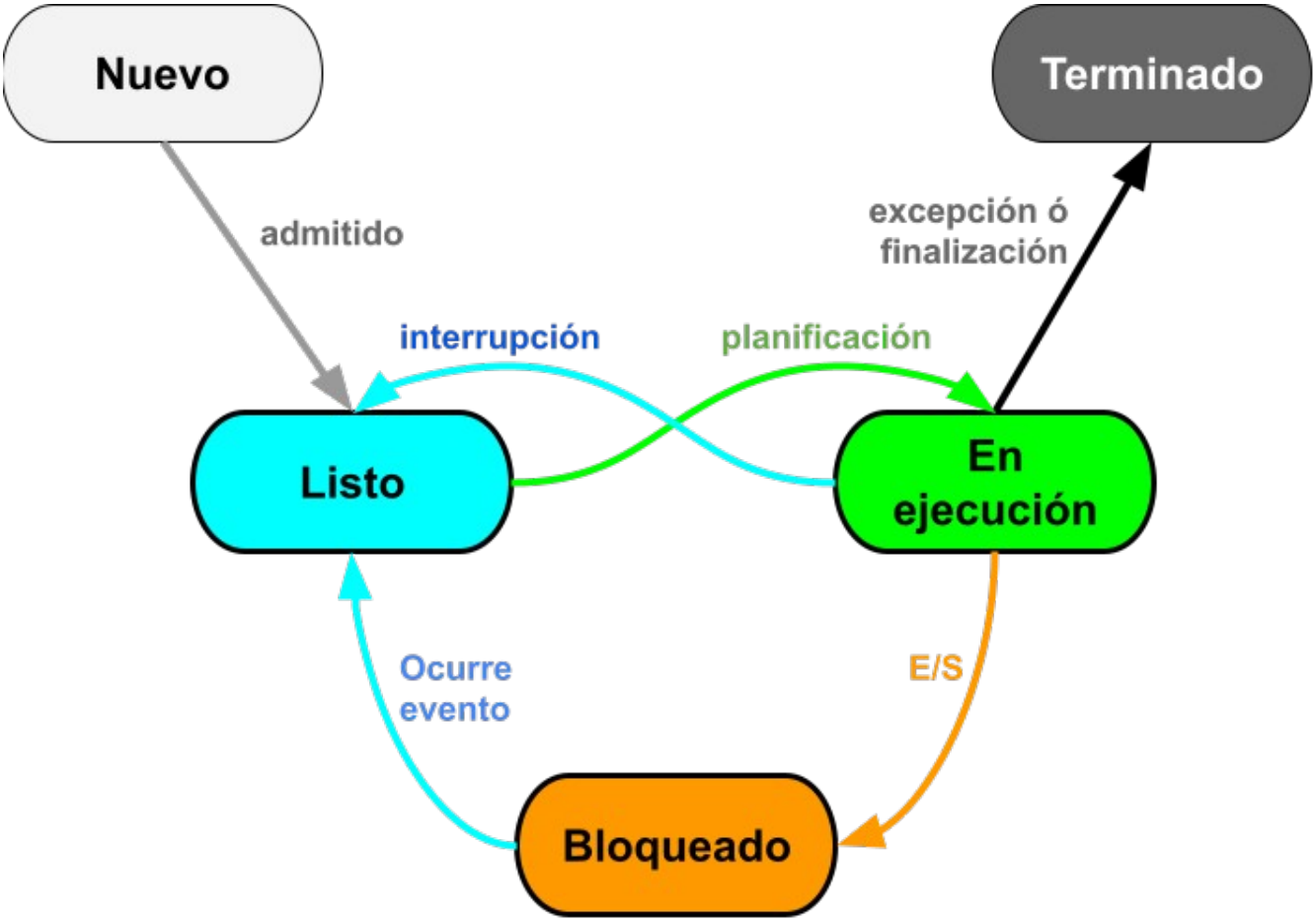
> **Control de procesos en Windows**
>
> En los sistemas operativos Windows, la mayoría de estas acciones se pueden realizar desde el administrador de tareas, aunque también tenemos los comandos **tasklist** y **taskkill** para hacerlo desde consola
>
> *tasklist /svc /fi "imagename eq svchost.exe"* Con esta instrucción sabremos que servicios se están ejecutando bajo el proceso svchost.exe, es el nombre de proceso de host genérico para servicios que se ejecutan desde bibliotecas de vínculos dinámicos (DLL), hay tantos para evitar riesgos ya que si estuviera todo en uno un posible fallo podría colapsar el sistema.

4.3 Estados de un proceso

El siguiente diagrama muestra los tres posibles estados en los que se puede encontrar un proceso. Las líneas que conectan los estados representan las posibles transiciones que se pueden dar.

En todo momento un procesos estará en una de los tres estados. Como ya hemos visto, en los sitemas monoprocesador, un único proceso podrá estar en estado de ejecución en un momento dado. El resto de procesos estará o bien en espera o bien bbloqueados.

Para cada uno de los estados se gestiona una lista de procesos que administra el kernel del SO. Los procesos permanecerán en la cola hasta que se produzca algún evento.



- **Nuevo**. El fichero es creado a partir de un ejecutable.
- **Listo**. Está parado temporalmente y listo para ejecutarse cuando se le dé la oportunidad. El sistema oprativo todavía no le asigno un procesador para ejecutarse. El planificador del S.O. será el responsable de seleccionar el proceso para que pase a estado de ejecución.
- **En ejecución**. Está usando el procesador. El sistema operativo utiliza el mecanismo de interrupciones para controlar su ejecución. Si el proceso necesitase un recurso, incluyendo la realización de operaciones de E/S, llamará a la llamada al sistema

correspondiente. Si un proceso se ejecuta durante el máximo tiempo permitido por la política del sistema, salta un temporizador que lanza una interrupción. Si el sistema es de tiempo compartido, lo para y lo pasa a estado de listo.

- **Bloqueado**. El proceso se encuentra bloqueado esperando a aque ocurra algún suceso. Por ejemplo puede estar esperando a que termine alguna operación de E/S, o bien a sincronizarse con otro proceso. Cuando ocurre el evento que lo desbloquea, el proceso queda pendiente de ser planificado por el S.O. no pasa directamente a ejecución.
- **Terminado**. El proceso termina y libera su imagen de memoria. Es el propio proceso el que debe llamar al sistema para indicar que ha terminado, aunque el sistema puede finalizarlo con una excepción (que es una interrupción especial).

Transiciones entre estados:

- **De ejecución a bloqueado**: un proceso pasa de ejecución a bloqueado cuando espera la ocurrencia de un evento externo.
- **De bloqueado a listo**: cuando ocurre el evento externo que esperaba
- **De listo a ejecución**: cuando el sistema le otorga un tiempo de CPU.
- **De ejecución a listo**: cuando se le acaba el tiempo asignado por el S.O.

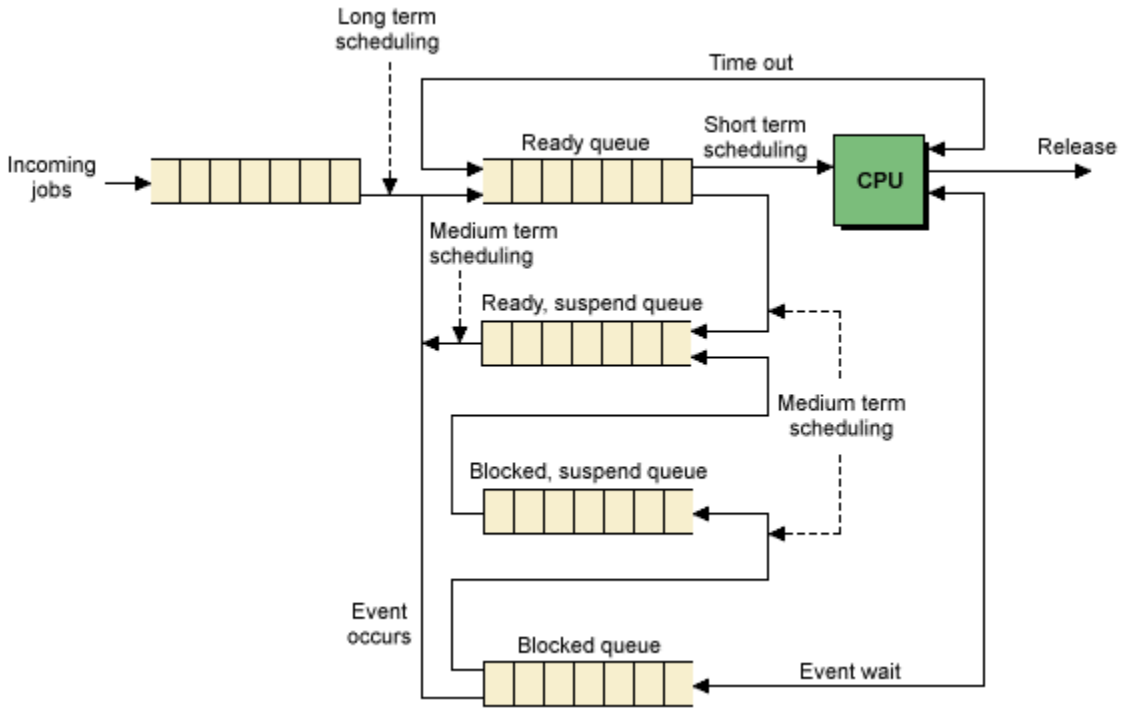
4.4 Planificación de procesos

Uno de los objetivos de los sistemas operativos es la multiprogramación, es decir, admitir varios procesos en memoria para maximizar el uso del procesador. Esto funciona ya que los procesos se irán intercambiando el uso del procesador para su ejecución de forma concurrente. Para ello, el sistema operativo organiza los procesos en varias colas pasándolos de unas colas a otras

•Cola de procesos: contiene todos los procesos del sistema

•Cola de procesos preparados: todos los procesos listos esperando para ejecutarse.

•Varias colas de dispositivos: procesos que están esperando alguna operación de E/S.



El planificador es el encargado de seleccionar los movimientos de los procesos entre las distintas colas. Existe una planificación a corto plazo y otra a largo plazo, veamos cada una:

- Corto plazo: selecciona los procesos de la cola de preparados para pasarlos a ejecución, se invoca con mucha frecuencia, del orden de milisegundos, por lo que el algoritmo debe ser muy sencillo.
    - Planificación sin desalojo: un proceso en ejecución sólo se saca si termina o bien se queda bloqueado.
    - Planificación apropiativa: solo se saca un proceso de ejecución si termina, se bloquea o por último aparece un proceso con mayor prioridad.
    - Tiempo compartido: cada cierto tiempo (cuanto), se desaloja un proceso y se mete otro, Se considera que todos los procesos tienen la misma prioridad.
- Largo plazo: selecciona que procesos nuevos pasan a la cola de preparados. Hace un control del grado de multiprogramación del proceso para tomar sus decisiones.

Cambios de contexto

El cambio de contexto que se hace al cambiar un proceso es tiempo perdido, ya no se hace trabajo útil. Cambiar el estado del proceso, el estado del procesador (cambio valores de registro) e información de la gestión de memoria, por muy rápido que se haga si se hace con mucha frecuencia puede provocar una ralentización del sistema, por eso tener muchos programas abiertos provoca una disminución importante en el rendimiento del sistema.

4.5 Algoritmos de planificación de procesos

Los algoritmos de planificación se utilizan para intentar mejorar el rendimiento del sistema y, por ende, la experiencia de usuario.

Para establecer parámetros objetivos que permitan comparar los diferentes resultados, vamos a tomar como referencia los siguientes criterios:

- **Tiempo de espera**: tiempo que un proceso permanece en la cola de preparados o de bloqueados esperando a ser ejecutado.
- **Tiempo de retorno**: tiempo transcurrido entre la llegada de un proceso y su finalización.
- **Uso de CPU**: % de tiempo que la CPU está siendo utilizada
- **Rendimiento/Productividad (throughput)**: número de procesos que se completan por unidad de tiempo

Procesos	LLegada	Tiempo uso CPU	Prioridad
P1	0	10	5
P2	1	6	10
P3	2	3	7

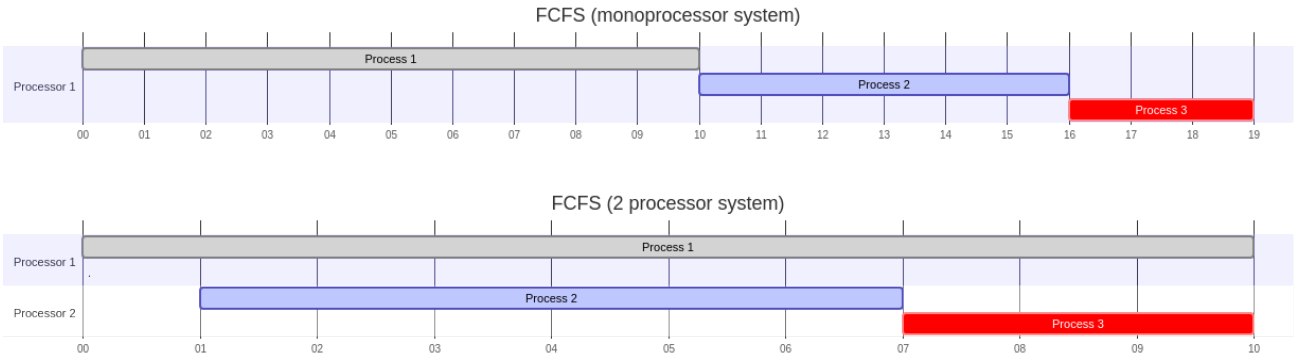
Con esta información, vamos a ver cómo se comportan los diferentes algoritmos

FCFS - First Come First Served

En esta política de planificación, el procesador ejecuta cada proceso hasta que termina o pasa al estado de bloqueado, por tanto, los procesos que están en la cola de procesos preparados permanecerán en el orden en que lleguen hasta que les toque su ejecución. Este método se conoce también como FIFO (Fist In, First Out).

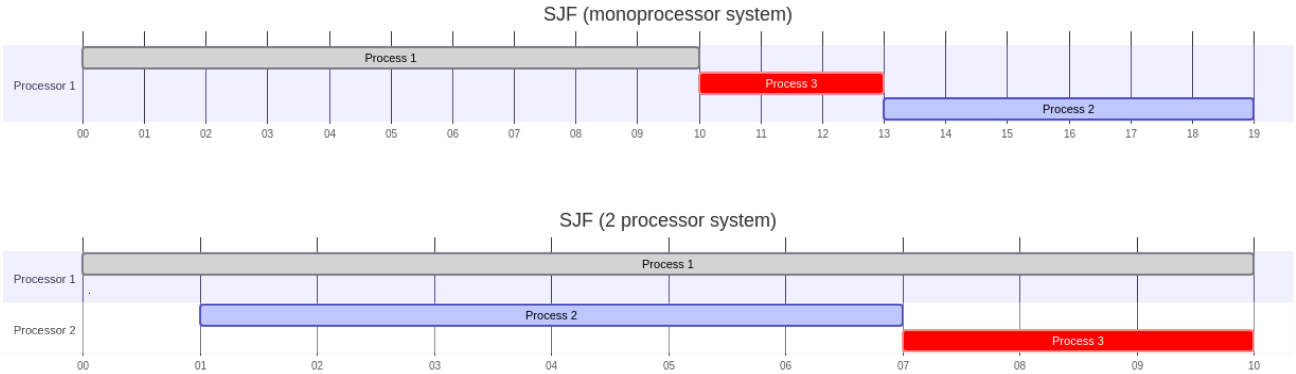
Se trata de una política muy simple y sencilla de llevar a la práctica, pero muy pobre en cuanto a su comportamiento.

La cantidad de tiempo de espera de cada proceso depende del número de procesos que se encuentren en la cola en el momento de su petición de ejecución y del tiempo que cada uno de ellos tenga en uso al procesador, y es independiente de las necesidades del propio proceso.



SJF - Shortest Job First

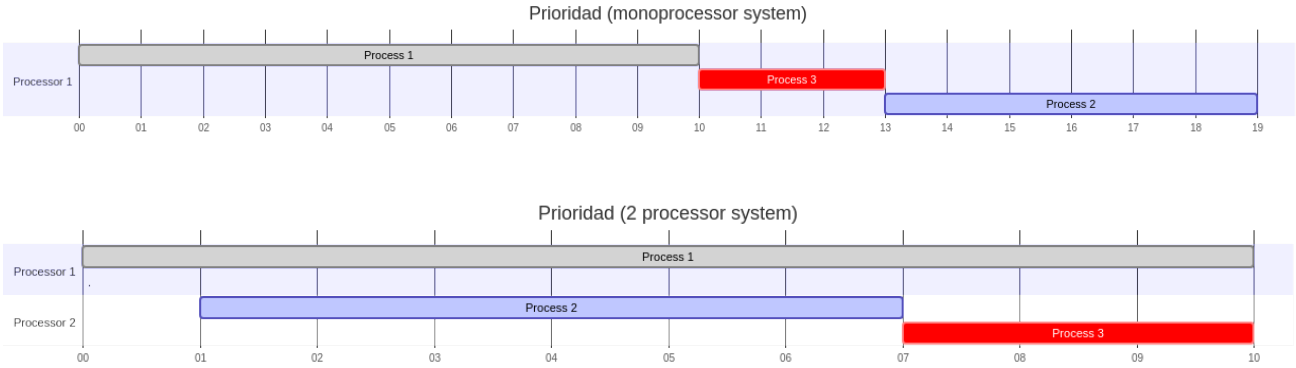
Este algoritmo siempre prioriza los procesos más cortos primero independientemente de su llegada y en caso de que los procesos sean iguales utilizara el método FIFO anterior, es decir, el orden según entrada. Este sistema tiene el riesgo de poner siempre al final de la cola los procesos más largos por lo que nunca se ejecutarán, esto se conoce como `inanición`.



Planificación por prioridad

Cada proceso tiene una prioridad, ejecutándose primero el que tenga mayor prioridad, independientemente de su llegada y en caso de que las prioridades sean iguales utilizará el método FIFO anterior, es decir, el orden según entrada.

Como ocurría con SJF, con este algoritmo son los procesos de prioridad más baja los que tienen riesgo de inanición.



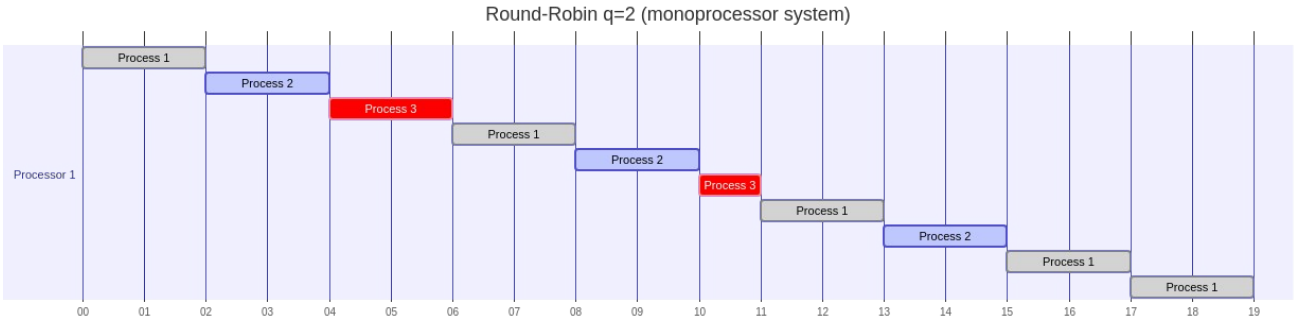
Round Robin

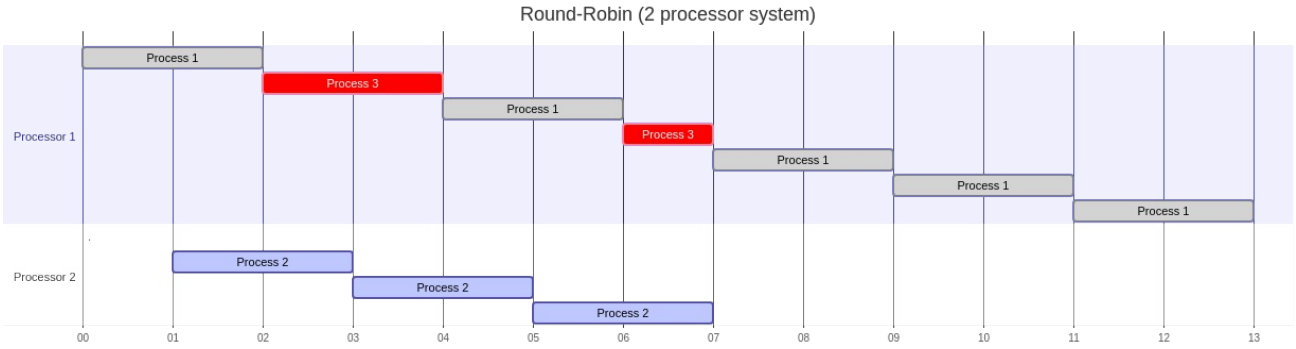
Este algoritmo de planificación es uno de los más complejos y difíciles de implementar, asigna a cada proceso un tiempo equitativo tratando a todos los procesos por igual y con la misma prioridad.

Este algoritmo es circular, volviendo siempre al primer proceso una vez terminado con el último. Para controlar que todos los procesos tienen su tiempo de CPU este método asigna a cada proceso un intervalo de tiempo llamado `quantum`.

Se pueden dar dos casuísticas con este método :

- El proceso, o lo que le queda por ejecutar, es menor que el quantum: Al terminar antes se planifica un nuevo proceso.
- El proceso, o lo que le queda por ejecutar, es mayor que el quantum: Al terminar el quantum se expulsa el proceso dando paso al siguiente proceso en la lista. Al terminar la iteración se volverá para terminar el primer proceso expulsado.





Planificador combinado

En realidad, no se usa una única estrategia de planificación, sino que lo más común es que se combinen varias de ellas. De hecho en Round-Robin hemos usado también FCFS.

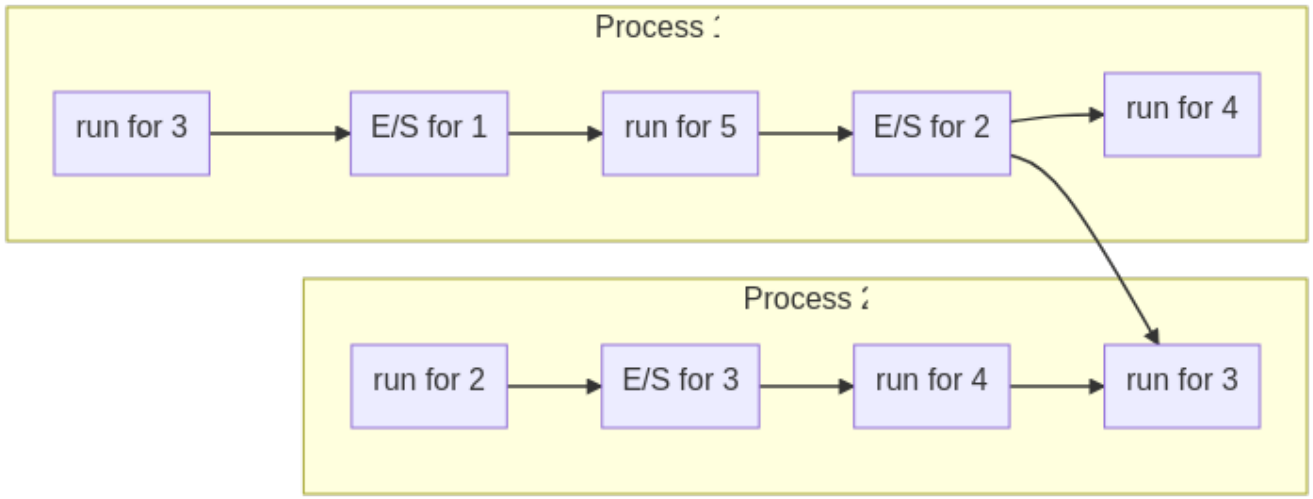
¿Te atreves a ver cómo sería una planificación Round-Robin con prioridad? Ten en cuenta que funcionará con el quantum y a la hora de escoger el siguiente proceso a ejecutar, se basará en la prioridad de los que haya en la lista.

Procesos con operaciones de E/S o bloqueos

En los ejemplos anteriores hemos visto que todos los procesos pasan su tiempo en el procesador, pero esto no es un reflejo de la realidad, más bien al contrario. Los procesos en determinados momentos deben dejar el procesador para esperar una entrada de usuario, leer o almacenar información en disco, o símplemente esperar a que otro proceso termine una acción y le envie un dato que necesita para continuar.

En esos instantes, el proceso deja el procesador libre para que otros puedan hacer uso de él. En el momento en que ha terminado su espera o bloqueo, se vuelve a poner en cola de preparado para seguir ejecutándose.

En el siguiente gráfico tenemos una especificación de la actividad de 2 procesosen el que, antes de realizar el último paso de ambos, debe haber acabado la operación de E/S que realiza el proceso1.



Veamos cómo se materializa esto en una ejecución de los procesos, suponiendo que ambos llegan a la vez a la cola.