




# Programación de Servicios y Procesos

## 1.3. Procesos en el SO

---



I.E.S.  
Doctor Balmis

Apuntes de PSP creados por Vicente Martínez bajo licencia [CC BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/) 

# 1.3. Procesos en el Sistema Operativo

- 1.3.1. El kernel del SO
- 1.3.2. Control de procesos en GNU/Linux
  - Comandos para saber el pid de los procesos
  - Comandos para ver los procesos activos
  - Control de procesos
- 1.3.3. Estados de un proceso
- 1.3.4 Planificación de procesos
- 1.3.5. Algoritmos de planificación de procesos
  - FCFS - First Come First Served
  - SJF - Shortest Job First
  - Planificación por prioridad
  - Round Robin
  - Procesos con operaciones de E/S o bloqueos

## 1.3.1. El kernel del SO

---

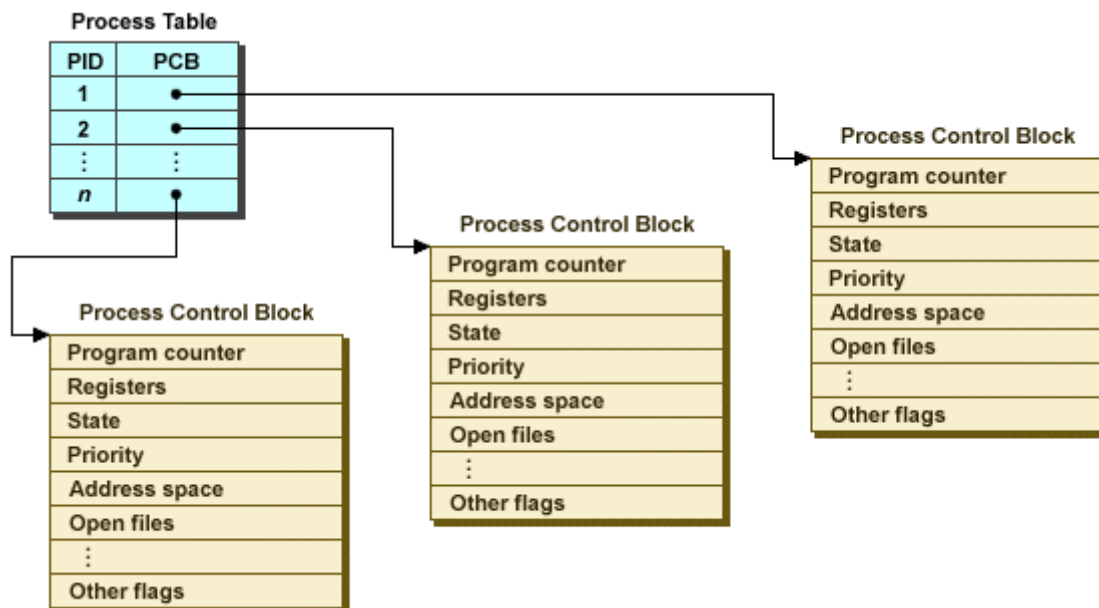
El `kernel` o `núcleo de un SO` se encarga de la funcionalidad básica del sistema, el responsable de la gestión de los recursos del ordenador, se accede al núcleo a través de las llamadas al sistema, es la parte más pequeña del sistema en comparación con la interfaz. El resto del sistema operativo se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



## 1.3.2. 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.

```
1 # pidof systemd
2 1
3 # pidof top
4 2060
5 # pidof httpd
6 2103 2102 2101 2100 2099 1076
7 # Process pid
8 echo $$
9 2109
10 # Process parent pid
11 echo $PPID
12 2106
```

sh

## 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.

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

sh



## 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 la 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          1 root        20   0    892    576   516  S   0.0   0.0   0:00.04 init
10         7 root        20   0    892     80    20  S   0.0   0.0   0:00.00 init
11         8 root        20   0    892     80    20  S   0.0   0.0   0:00.01 init
12         9 vicente    20   0  10032   5032  3324  S   0.0   0.0   0:00.11 bash
13        73 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 gestionarla.

```
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 killall

<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:~$
```

sh



## 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

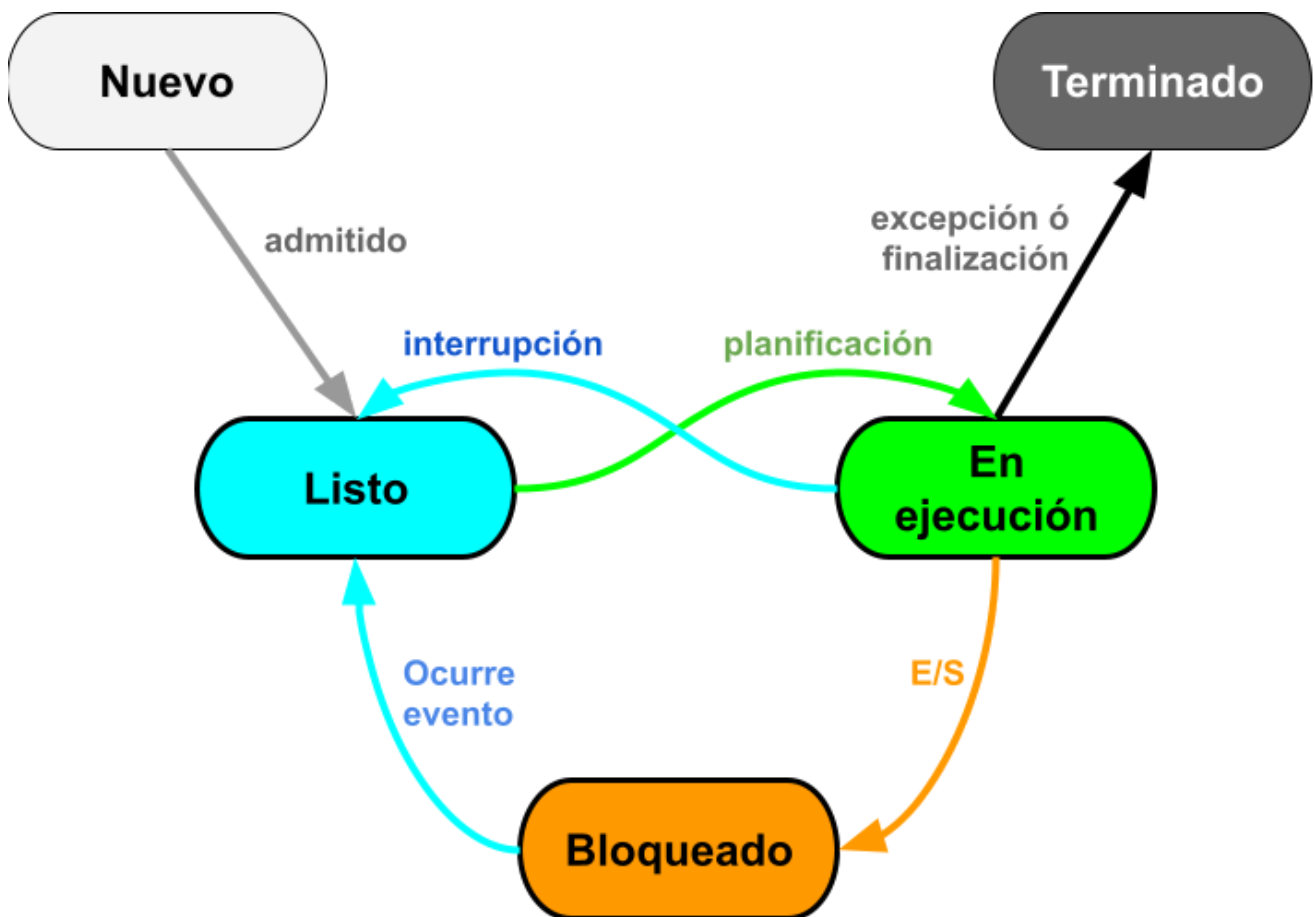
*tasklist /svc /fi "imagenname eq svchost.exe"* Con esta instrucción sabremos que servicios se están ejecutando bajo el proceso svchost.exe, es el nombre de proceso de host genérico para servicios que se ejecutan desde bibliotecas de vínculos dinámicos (DLL), hay tantos para evitar riesgos ya que si estuviera todo en uno u nposible fallo podría colapsar el sistema.

### 1.3.3. 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 sistemas 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 operativo todavía no le asigna 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 necesita 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 que 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.

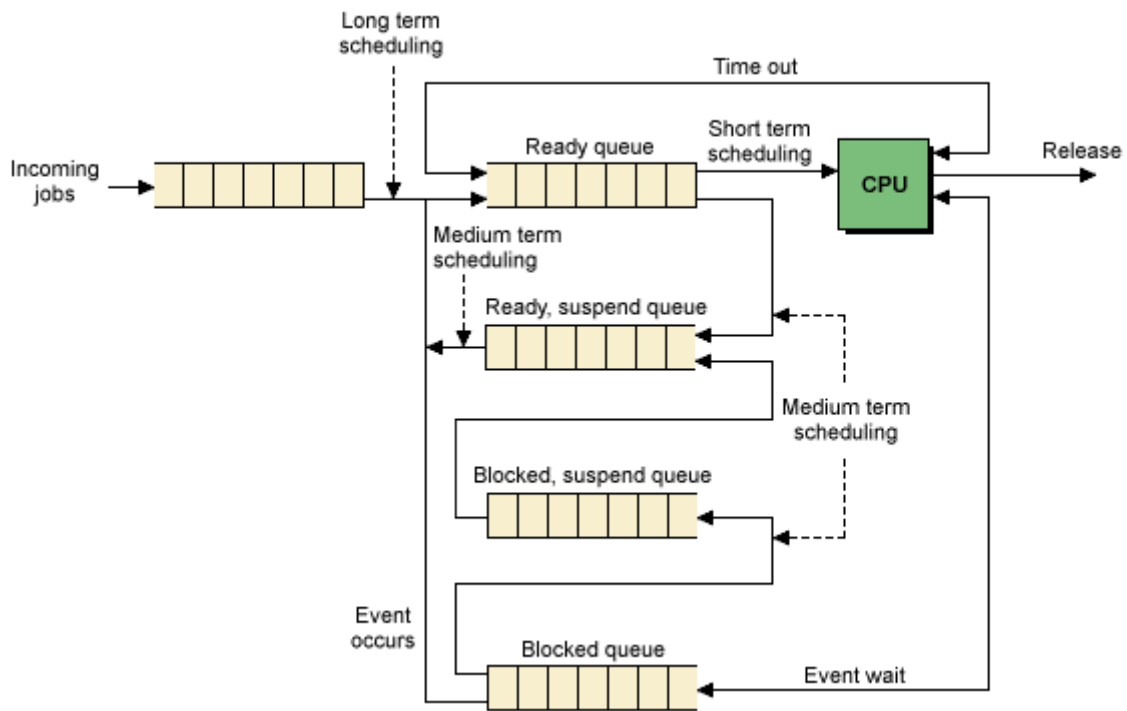
## 1.3.4 Planificación de procesos

---

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.

## 1.3.5. 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

En sistema con 1 unidad de proceso

$$\frac{\text{instante de tiempo en el que acaba el ultimo proceso}}{\# \text{ instantes de tiempo que el procesador esta ocupado}} \times 100$$

En un sistema con N unidades de proceso

$$\frac{\text{instante de tiempo en el que acaba el ultimo proceso} * N}{\sum_{n=1}^N \# \text{ instantes de tiempo que el procesador}_n \text{ esta ocupado}} \times 100$$

- **Rendimiento/Productividad (throughput):** número de procesos que se completan por unidad de tiempo

$$\frac{\text{numero de procesos}}{\text{instante de tiempo en el que acaba el ultimo proceso}}$$

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

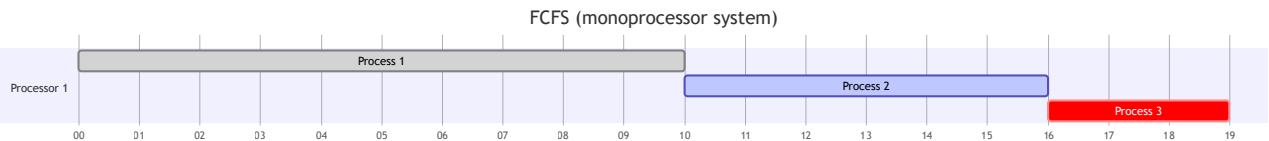
### FCFS - First Come First Served

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

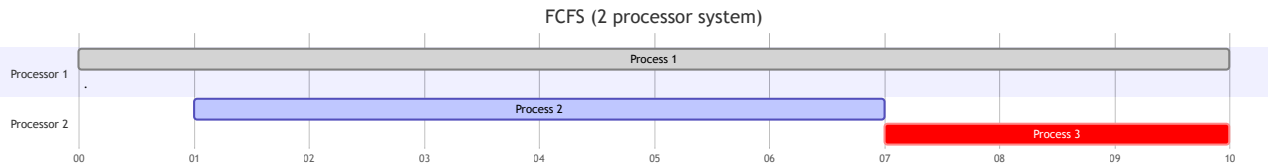
preparados permanecerán en el orden en que lleguen hasta que les toque su ejecución. Este método se conoce también como FIFO (Fist In, First Out).

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.



Procesos	Tiempo espera	Tiempo de retorno	% uso CPU	Productividad
P1	0	10		
P2	9	15		
P3	14	17		
Medias	8,3	14	100%	0,15



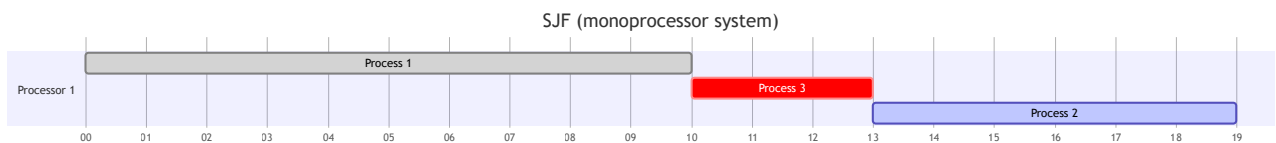
Procesos	Tiempo espera	Tiempo de retorno	% uso CPU	Productividad
P1	0	10		

Procesos	Tiempo espera	Tiempo de retorno	% uso CPU	Productividad
P2	0	6		
P3	5	8		
Medias	1,6	6	95%	0,3

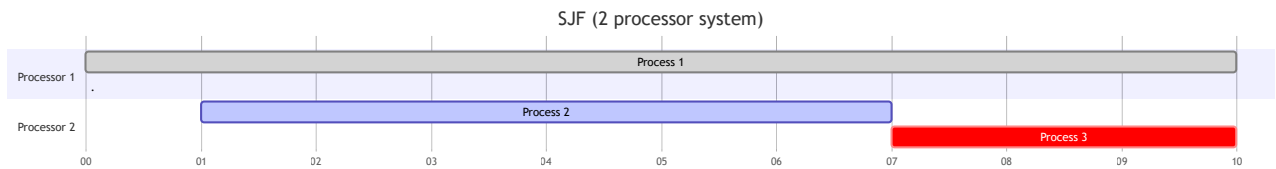
## SJF - Shortest Job First

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 .



Procesos	Tiempo espera	Tiempo de retorno	% uso CPU	Productividad
P1	0	10		
P2	12	18		
P3	8	11		
Medias	7,3	13	100%	0,15

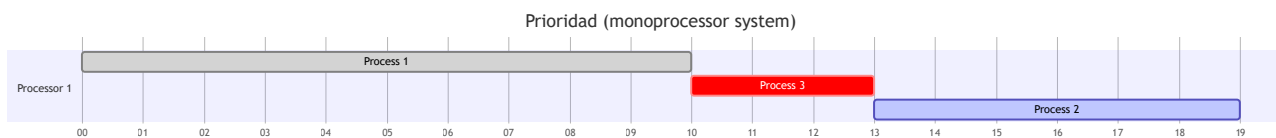


Procesos	Tiempo espera	Tiempo de retorno	% uso CPU	Productividad
P1	0	10		
P2	0	6		
P3	5	8		
Medias	1,6	6	95%	0,3

## Planificación por prioridad

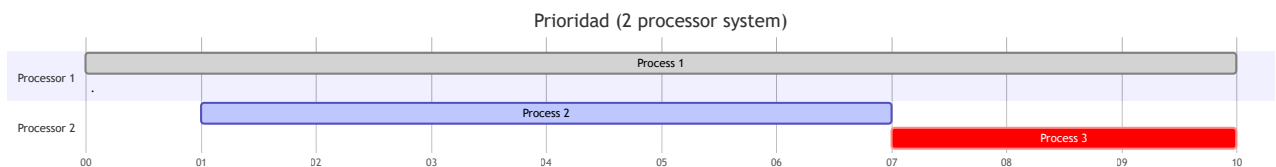
Cada proceso tiene una prioridad, ejecutándose primero el que tenga mayor prioridad, independientemente de su llegada y en caso de que las prioridades sean iguales utilizará el método FIFO anterior, es decir, el orden según entrada.

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



Procesos	Tiempo espera	Tiempo de retorno	% uso CPU	Productividad
P1	0	10		
P2	12	18		

Procesos	Tiempo espera	Tiempo de retorno	% uso CPU	Productividad
P3	8	11		
Medias	6,6	13	100%	0,15



Procesos	Tiempo espera	Tiempo de retorno	% uso CPU	Productividad
P1	0	10		
P2	0	6		
P3	5	8		
Medias	1,6	6	95%	0,3

## Round Robin

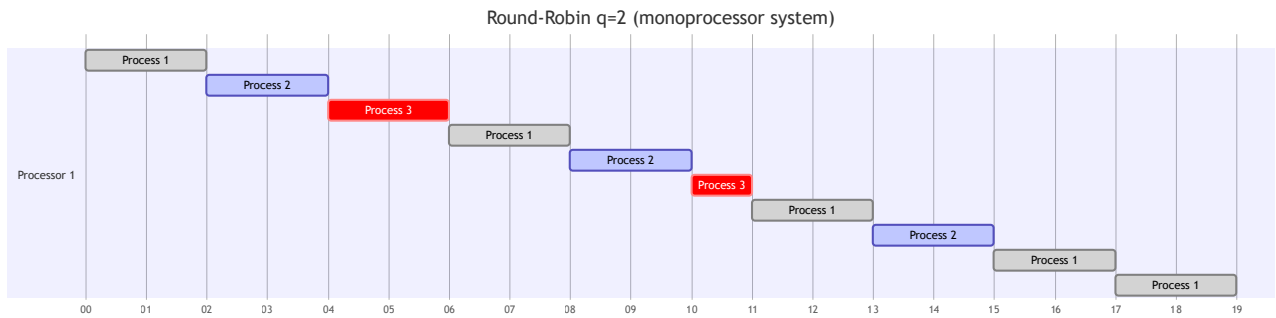
Este algoritmo de planificación es uno de los más complejos y difíciles de implementar, asigna a cada proceso un tiempo equitativo tratando a todos los procesos por igual y con la misma prioridad.

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`.

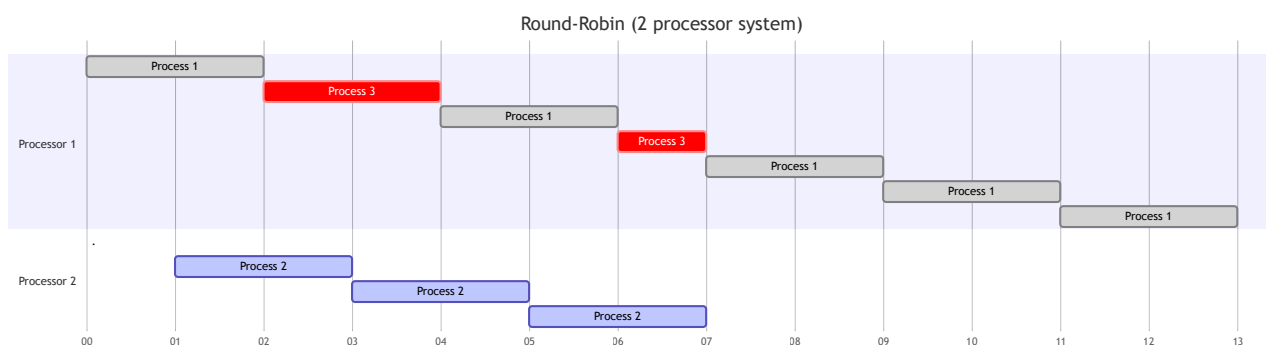
Se pueden dar dos casuísticas con este método :

- 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.



Procesos	Tiempo espera	Tiempo de retorno	% uso CPU	Productividad
P1	9	19		
P2	8	14		
P3	6	9		
Medias	7,6	14	100%	0,15



Procesos	Tiempo espera	Tiempo de retorno	% uso CPU	Productividad
P1	3	13		
P2	0	5		
P3	2	5		
Medias	1,6	7,6	73%	0,23



### 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

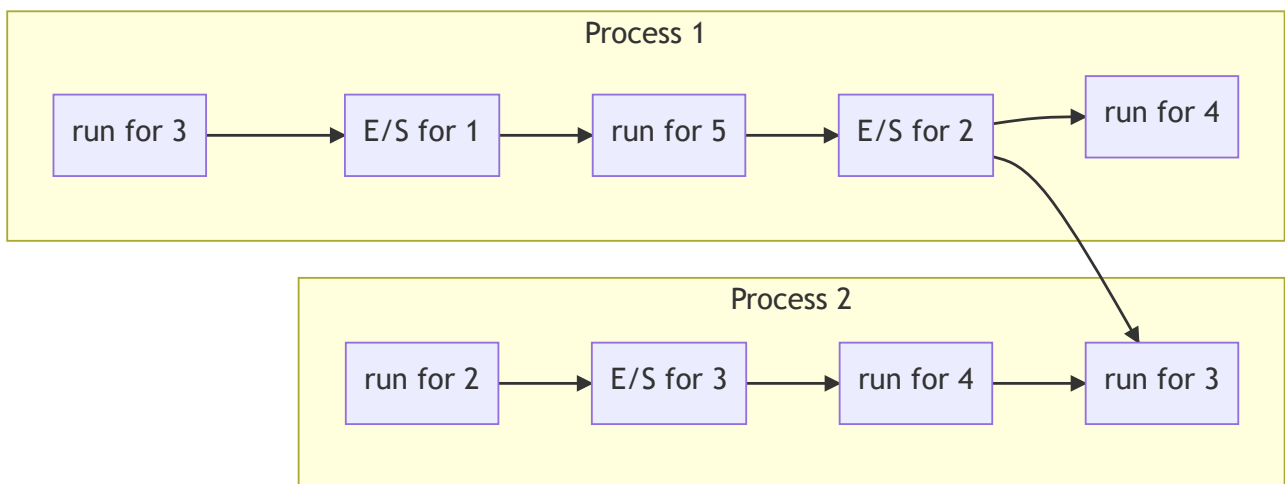
En los ejemplos anteriores hemos visto que todos los procesos pasan su tiempo en el procesador, pero esto no es un reflejo de la realidad, más bien al contrario. Los procesos en



determinados momentos deben dejar el procesador para esperar una entrada de usuario, leer o almacenar información en disco, o simplemente esperar a que otro proceso termine una acción y le envíe 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 procesos en 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.

The Gantt chart shows the execution of two tasks, Task 1 and Task 2, over a timeline from 00 to 20. Task 1 (light blue background) includes processes P1 runs 1 and P1 runs 2, which have multiple execution periods and wait states (E/S). Task 2 (light purple background) includes processes P2 runs 1 and P2 runs 2, also with multiple execution periods and wait states (E/S and Locked waiting). The chart illustrates how the expiration of a semaphore (q) allows processes to proceed, even if they are waiting for the semaphore to be released by another process.

Task	Process	State / Event	Start Time	End Time
Task 1	P1 runs 2	Execution	00	01
	P1 runs 2	Wait (E/S)	01	04
	P1 runs 1	Execution	04	05
	P1 runs 1	Wait (E/S)	05	06
	P1 runs 2	Execution	06	08
	P1 runs 2	Wait (E/S)	08	10
	P1 runs 2	Execution	10	12
	P1 runs 2	Wait (E/S)	12	14
	P1 runs 2	Execution	14	16
	P1 runs 2	Wait (E/S)	16	18
Task 2	P2 runs 2	Execution	01	04
	P2 runs 2	Wait (E/S)	04	06
	P2 runs 2	Execution	06	09
	P2 runs 2	Wait (E/S)	09	10
	P2 runs 2	Execution	10	12
	P2 runs 2	Wait (E/S)	12	14
	P2 runs 2	Execution	14	16
	P2 runs 2	Wait (E/S)	16	18

### Task 1

## Task 2

P1 runs 2 (exits because of q expiration)

P1 runs 1 (exits and has to wait 1 for E/S)

E/S

P1 runs 2 (exits because of q expiration)

P1 runs 2 (exits because of q expiration)

P1 runs 2 (exits and has to wait 1 for E/S)

E/S

P1 runs 2 (exits because of q expiration)

P1 runs 2 (exits because of q expiration)

P2 runs 2 (exits and has to wait 3 for E/S)

E/S

P2 runs 2 (exits because of q expiration)

**P2 runs 2 (exits because of q expiration)**

Locked waiting

P2 runs 2 (exits because of q expiration)

P2 runs 2 (ends execution)

00

05

10

15

20