

TEMA 2: GESTIÓN DE PROCESOS

INTRODUCCIÓN

En un sistema de computación hay muchos recursos que administrar. Entre ellos, el procesador o la CPU, ya que ningún programa se puede ejecutar si no se le concede el uso de este.

Dado que suele haber bastantes más procesos que procesadores, la forma en la que se reparte el uso del procesador entre los distintos procesos es fundamental, se llama **planificación de la CPU**.

Mantener un registro de distintas actividades paralelas es una tarea difícil. Para facilitar el uso del paralelismo han desarrollado el **modelo de procesos**, en el cual todo el software ejecutable del ordenador, incluido el propio sistema operativo, se organiza en torno al concepto de proceso.

CONCEPTO DE PROCESO

Un proceso es básicamente un programa en ejecución. Un programa, en cambio es algo **estático**, es el contenido de un fichero en disco. Un **proceso**, en cambio, es **algo dinámico** que en cada instante tiene un estado concreto, el cual viene determinado por el contador de programa y los registros de la CPU. Un proceso necesita disponer de memoria para almacenar el **código** y los **datos** del programa, memoria también para su pila, CPU para poder ejecutarse, espacio en disco para leer y escribir, etc...

Al igual que un proceso puede constar de varios programas, un mismo programa puede dar lugar a varios procesos.

Cuando solo hay una CPU y esta se pasa rápidamente de un proceso a otro, tenemos la **sensación** de que los procesos se ejecutan al mismo tiempo (**pseudoparalelismo**), cuando no es así, pues en un determinado instante la CPU solo puede estar ejecutando el código de un proceso. El **paralelismo real**, en cambio, se da cuando hay varias CPU o cuando una CPU dispone de varios núcleos.

Permitir la ejecución concurrente de varios procesos es una tarea compleja. Sin embargo, existen muchas razones que hacen que esto merezca la pena:

- **Compartir recursos físicos**, pues muchas veces los recursos hardware son caros o su disponibilidad en el sistema está limitada, por lo que nos podemos ver obligados a compartirlos en un entorno multiusuario.
- **Compartir recursos lógicos**, como una base de datos o cualquier otro elemento de información.
- **Acelerar los cálculos**, ya que, si queremos acelerar una tarea, podemos dividirla en subtarefas para que todas ellas se ejecuten en paralelo.
- **Modularidad**, ya que un sistema se construye con más facilidad si se diseña como un conjunto de procesos separados.
- **Comodidad** para el usuario.

Los procesos nunca deben programarse con hipótesis implícitas de tiempo, al existir varios procesos ejecutándose, no se sabe con certeza cuándo se ejecutará cada proceso ni cuándo se atenderán sus solicitudes. Cuando un proceso deja la CPU hay que decidir qué proceso, pasará mediante un **algoritmo de planificación**.

CAMBIO DE PROCESO, DE CONTEXTO Y DE MODO

El hecho de que la CPU deje de ejecutar un proceso y pase a ejecutar otro se denomina **cambio de proceso**.

Para cambiar de proceso, el control de la CPU debe pasar al sistema operativo, que guardará el estado o contexto del proceso que estaba usando la CPU, seleccionará un nuevo proceso y restaurará su estado o contexto para que este otro proceso haga uso de la CPU. A estos saltos los llamaremos

cambio de contexto. Un cambio de proceso siempre conlleva uno o más cambios de contexto, pero no al contrario.

Un **cambio de modo** es cuando pasa de modo usuario a modo núcleo y viceversa. Un cambio de modo supone siempre un cambio de contexto, pero no un cambio de proceso. En cambio, un cambio de contexto no implica siempre un cambio de modo.

CREACIÓN Y DESTRUCCIÓN DE PROCESOS. JERARQUIA DE PROCESOS

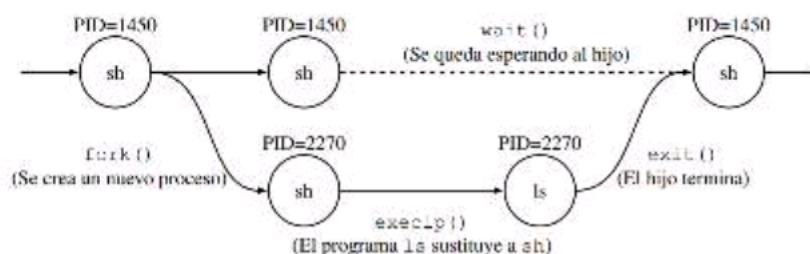
Si es un sistema operativo soporta el concepto de proceso, debe proporcionar llamadas al sistema para crear y destruir proceso. En Unix, existe `fork()`, que crea una **copia idéntica del proceso padre** que hace la llamada. Podemos decir que el **proceso hijo es un clon del proceso padre**. La única **diferencia** entre uno y otro es el valor devuelto por la función `fork()`, que en el **hijo es 0** y en el **padre** es un número que se le llama identificador de proceso (**PID**) que identifica el proceso hijo.

Después del `fork()`, el **padre e hijo continúan su ejecución en paralelo** de forma totalmente independiente. Dado que el padre y el hijo son idénticos, ambos ejecutarán el mismo código. La forma de conseguir que **ambos realicen tareas distintas**, a través de otra función como `execve()` o `exec()`, que no crea un nuevo proceso, sino que simplemente cambia el código (mutación) y los datos por otros dentro del mismo proceso. A pesar de este cambio, hay elementos que conservan como los ficheros abiertos, el tratamiento de las señales y el PID y otras propiedades del proceso.

En los sistemas Windows, la llamada al sistema para crear procesos y para que el nuevo proceso hijo ejecute un nuevo programa es `CreateProcess()`.

Además, en Windows, un proceso hijo puede pasar a ser hijos de otro proceso que no sea su padre original, a través de transferir la propiedad. Esto hace que en Windows el concepto de jerarquía desaparezca.

Un proceso puede terminar de forma **voluntaria** o **involuntaria**. La **finalización voluntaria** es el propio proceso el que decide finalizar su ejecución, bien porque ha terminado su tarea o bien porque **ha encontrado algún error** que le impide continuar su ejecución. Para esta finalización voluntaria, en Unix existe `exit()` y en Windows `ExitProcess()`. En una **terminación involuntaria**, un proceso es **finalizado por el propio sistema operativo o por otro proceso**, siempre que tenga autorización para ello. Para esto en Unix existe `kill()` y en Windows `TerminateProcess()`.



ESTADOS DE UN PROCESO

Un proceso puede estar en 3 estados posibles:

- En **ejecución**, el proceso está utilizando la CPU (solo puede haber uno a la vez).
- En **listo**, el proceso es ejecutable, pero está a la espera de que llegue su turno de CPU.
- En **bloqueado**, el proceso no se puede ejecutar, aunque se le asigne la CPU, porque se encuentra a la espera de que ocurra algún evento externo.

Lo ideal es un proceso en ejecución y varios en listo y varios en bloqueados.



Entre estos estados, existen 3 posibles transiciones:

- **en ejecución → bloqueado**: cuando un proceso **no puede continuar su ejecución** por algún motivo debe bloquearse.
- **en ejecución → listo o listo → en ejecución**: un proceso **pasa de estar en ejecución a estar listo cuando es expulsado de la CPU por haber excedido su tiempo de uso**, o cuando se debe ejecutar un proceso más importante. Un proceso **listo pasa a ejecutarse cuando la CPU queda libre** y, según el planificador, al que le corresponde usar la CPU; **también si es más importante.**
- **bloqueado → listo**: el proceso **ya dispone de lo que necesitaba**. Si la CPU está desocupada, pasará a ejecutarse de forma inmediata.

El diagrama de estados anterior es demasiado sencillo. Cuando un proceso se crea, hace que aparezca el estado *nuevo* en el que un proceso recién creado permanecerá hasta disponer de todos los recursos necesarios.

Y desde que un proceso termina su ejecución hasta que desaparece completamente puede transcurrir un cierto tiempo. Esto da lugar al estado *saliente*.

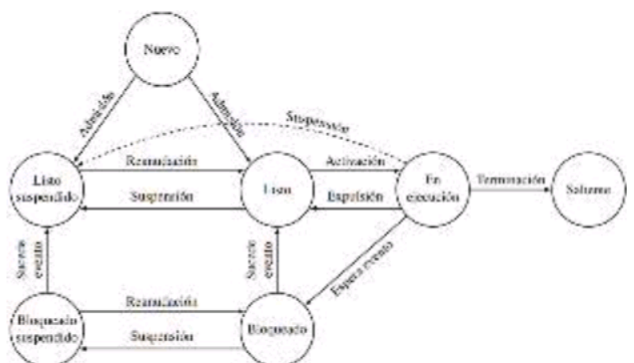
Existe más situaciones que pueden dar lugar a otros estados. Así, un proceso puede ser suspendido durante un tiempo y después ser reanudado por otro proceso. Estas operaciones son útiles por:

- Se pueden suspender procesos activos hasta corregir el problema y luego reanudarlos.
- Si se piensa que los resultados de un proceso son incorrectos, se puede suspender dicho proceso. Si después se comprueba que los resultados son correctos, se puede reanudar.
- Se puede reducir la carga suspendiendo procesos para así poder dar servicio a los procesos de mayor prioridad.

Los procesos suspendidos generalmente se guardan en disco para liberar la memoria principal que ocupan, la cual quedará disponible para otros procesos.

Para añadir estos dos nuevos estados hay 8 posibles transiciones más:

- **bloqueado** → **bloqueado suspendido**: cuando interesa para liberar memoria y que otros procesos listos pueden usarla y ejecutarse.
- **bloqueado suspendido** → **listo suspendido**: cuando el evento por el que se bloqueó el proceso se produce mientras este está suspendido.
- **listo suspendido** → **listo**: cuando no queden procesos listos, que haya quedado memoria libre, o que un proceso listo suspendido deba ejecutarse por tener más propiedad que los procesos que haya en memoria.
- **listo** → **listo suspendido**: se suspenden ya que están consumiendo memoria.
- **nuevo** → **listo suspendido**: cuando se crea un proceso, no hay memoria principal disponible y el nuevo proceso no tiene prioridad suficiente como para expulsar a otro de memoria principal.
- **bloqueado suspendido** → **bloqueado**: si un proceso bloqueado suspendido tiene una prioridad alta, y el sistema operativo sospecha que el evento que espera sucederá en breve, entonces puede tener sentido mover el proceso bloqueado de disco a memoria.
- **En ejecución** → **listo suspendido**: cuando un proceso de prioridad alta despierta de **bloqueado suspendido** y no hay memoria suficiente si no se pasa inmediatamente a disco.
- **cualquier estado** → **saliente**: en cualquier momento, un proceso puede finalizar.



IMPLEMENTACIÓN DE PROCESOS

El sistema operativo debe tener conocimiento de qué procesos existen y en qué estado se encuentra cada uno. Para ello, existe una **tabla de procesos con una entrada por proceso** llamados **PCB** (*Process Control Block*) o **BCP** (*Bloque de Control de Proceso*), y en ella se guarda toda la información relacionada con el proceso correspondiente.

En el PCB se debe guardar, todo aquello que sea necesario para que, después de perder la CPU por cualquier motivo y recuperarla más tarde, el proceso pueda continuar su ejecución.

CREACIÓN DE PROCESOS

Cuando se crea un proceso, al menos, se debe:

1. **Dar nombre al proceso** (generalmente es el **PID**).
2. **Insertarlo en la tabla de procesos**, creando su **PCB**.
3. Determinar su **prioridad** inicial.
4. Asignarle **recursos iniciales**.

Si la creación de procesos se realiza mediante la llamada al sistema **fork()**: *en Unix* :

1. Lo que produce el **salto al núcleo** del sistema operativo.
2. El **núcleo busca una entrada libre** en la tabla de procesos para el proceso hijo y le asigna un PID.
3. Toda la información del **padre se copia a la entrada del hijo** con algunas excepciones.
4. Se **asigna memoria** para los segmentos de datos y de pila del hijo.
5. Se **incrementan los contadores** asociados a cualquier fichero abierto en un nuevo proceso,
6. **Se le asigna al proceso hijo el estado listo**. Se devuelve el PID del hijo al padre y el valor 0 al hijo.

ESTRUCTURA DE UN PROCESO

Cuando un proceso realiza una **llamada al sistema**, se considera que ese mismo proceso pasa de modo usuario a **modo núcleo** y ejecuta el código del sistema operativo que **atiende dicha llamada**. Por lo tanto, **cada proceso tiene dos partes**: la que se ejecuta en espacio de usuario, y la que se ejecuta en espacio del núcleo.

Dado que varios procesos pueden estar haciendo llamadas al sistema al mismo tiempo, **es posible que varios procesos se encuentren a la vez ejecutando su parte del núcleo**, es decir, **ejecutando la misma copia de código**.

Es posible que una llamada al sistema haga que un proceso se bloquee dentro del núcleo. Cuando se desbloquee, continuará su ejecución dentro del núcleo hasta finalizar y regresar a su parte de usuario. Otro enfoque, donde el sistema operativo es una colección de procesos de sistema, distintos de los procesos de usuario se corresponde con la estructura de sistemas operativos que sigue el *modelo cliente-servidor*.

CAMBIO DE PROCESO

Un aspecto clave es que el **cambio** tiene que ser **automático y transparente de un proceso a otro**. Por seguridad, este cambio **solo lo puede hacer el sistema operativo** cuando se produce una interrupción (por un error o por un evento de E/S), una llamada al sistema solicitando algún tipo de servicio, o una excepción (un fallo de página pasaría a estado *bloqueado* y una división por cero pasaría a estado *saliente* porque lo mata).

Los pasos que se dan para cambiar de un proceso a otro son:

1. Al producirse la llamada al sistema, el hardware **almacena el contador de programa** del proceso que la realiza **en la pila**. Esta pila es la que utiliza el propio proceso cuando se ejecuta en modo usuario.
2. El hardware **pasa a modo núcleo** y carga el nuevo contador de programa que debe contener la dirección de inicio de alguna rutina.
3. **Se guarda el resto de los registros y otros datos en el PCB** del proceso activo que es el que realiza la llamada al sistema. También es posible que se **actualice otra información del PCB** (estado del proceso, tiempo de uso de CPU).
4. Un procedimiento en lenguaje ensamblador configura la nueva pila que va a utilizar el núcleo mientras se atiende la llamada al sistema. Como puede haber varios procesos, cada uno tendrá una pila para cuando ejecute código en modo núcleo. Suele ser diferente a la que se emplea en modo usuario. Antes de avanzar al paso siguiente, es posible que el procedimiento en ensamblador deba realizar algún otro trabajo dependiente de la arquitectura.
5. **Tras comprobar que la llamada al sistema existe, el procedimiento en ensamblador llama al procedimiento en C que la implementa.** Durante la ejecución el proceso se puede bloquear.
6. Tras la ejecución, **se vuelve al procedimiento en lenguaje ensamblador.** Esta **consulta** si hay que ejecutar el **planificador de procesos** y, en caso afirmativo, lo ejecuta. Si el proceso se ha bloqueado es posible que el planificador ya haya sido invocado, por lo que no sea necesario llamarlo de nuevo.
7. **El procedimiento en ensamblador carga en los registros del procesador los valores que había en el momento en el que se le quitó la CPU.** Entre otras cosas, prepara la pila que el nuevo proceso usará. Si la CPU cambia de proceso, entonces tanto el estado del proceso que sale como el estado del proceso que entra deben cambiar en consonancia. A esta parte del sistema operativo que entrega el control de la CPU al proceso seleccionado por el planificador se le denomina despachador.
8. **Se cambia a modo usuario** y se regresa de la llamada al sistema, lo que hace que se desapile la dirección de la siguiente instrucción a ejecutar.

Cuando finaliza el tratamiento de la llamada al sistema y se invoca al planificador, este puede decidir darle la CPU al mismo proceso que la tenía. En este caso, no se habrá producido un cambio de proceso sino varios cambios de contexto y de modo.

OPERACIONES CON PROCESOS

Un **sistema operativo** debe poder crear, destruir, suspender, reanudar, bloquear, ejecutar un proceso; además, debe poder cambiar su prioridad, cambiarlo de estado y permitirle que se comunique con otro proceso.

HILOS

Nos referimos a la unidad de planificación y ejecución como **hilo** (es una traza de ejecución a través de uno o más programas, un proceso tiene un estado, una prioridad y un contador de programa, y es la entidad que es planificada y ejecutada por el sistema operativo).

El **uso más importante** del concepto de hilo se da en aquellos casos en los que **varios hilos** pueden **existir dentro de un mismo proceso**. Por contra, los **procesos tradicionales son procesos con un único hilo**.

ELEMENTOS POR HILO Y POR PROCESO

Los hilos tienen cada uno:

- Zona de memoria o espacio de direcciones que contiene la imagen del proceso.
- Acceso protegido al procesador, otros procesos, ficheros y recursos de E/S.
- Variables globales, procesos hijos, alarmas y señales.
- Información contable.
- Un estado.
- Un contexto del procesador con al menos el registro contador de programa y el resto de los registros de la CPU.
- Una pila en ejecución.
- Algún almacenamiento por hilo para las variables locales, puede ser la pila del hilo u otra zona de memoria reservada.
- Acceso a la memoria y recursos del proceso al que pertenece el hilo, compartiendo con todos los restantes hilos del proceso.

Los distintos hilos de un proceso no son tan independientes como procesos diferentes. Todos los hilos tienen el mismo espacio de direcciones de memoria, lo que quiere decir que comparten también las mismas variables globales. Un hilo puede leer, escribir o limpiar de manera completa la pila de otro hilo; no existe protección entre los hilos.

APLICACIONES DE LOS HILOS

Puesto que los hilos comparten un mismo espacio de direcciones, se pueden comunicar entre sí sin intervención del núcleo; es más rápido que entre procesos.

Los hilos se pueden bloquear a la espera de que se termine una llamada al sistema. Esto permite solapar la E/S del proceso con su propio cómputo, ya que unos hilos del proceso pueden estar bloqueados en E/S mientras otros están haciendo uso de la CPU.


Es más rápido crear un nuevo hilo en un proceso existente que crear un nuevo proceso al que hay que asignarle nuevos recursos. También lleva menos tiempo finalizar un hilo, pues no hay que liberar recursos.

Si se dispone de varias CPU, se puede conseguir paralelismo real dentro de un mismo proceso.

Los hilos facilitan la construcción de programas, ya que, si un programa realiza varias funciones diferentes, cada función puede ser desempeñada por un hilo.

HILOS EN MODO USUARIO E HILOS EN MODO NÚCLEO

Hilos implementados en espacio de usuario:

- | | |
|----------------|--|
| Ventajas | <ul style="list-style-type: none">• El núcleo del sistema operativo no sabe que existen. Existe una tabla de hilos privada en cada proceso para realizar los cambios de contexto entre sus hilos.• Los cambios de contexto entre hilos son mucho más rápidos, ya que no es necesario pasar al núcleo.• Cada proceso puede tener su propio algoritmo de planificación para los hilos. |
| Inconvenientes | <ul style="list-style-type: none">• Cualquier llamada al sistema bloqueante realizada por uno de los hilos bloqueará a todo el proceso y sus hilos.• De forma similar, un fallo de página bloquea a todo el proceso y a sus hilos.• La CPU asignada al proceso tiene que repartirse entre todos sus hilos.• Si hay varias CPU, no es posible obtener paralelismo. |
- 

Hilos implementados en espacio núcleo:

- | | |
|----------------|--|
| Ventajas | <ul style="list-style-type: none">• El núcleo mantiene la tabla de hilos y reparte la CPU entre todos ellos. Si hay varias CPU, varios hilos de un mismo proceso se pueden ejecutar a la vez, consiguiendo paralelismo real.• Las llamadas al sistema bloqueantes no suponen ningún problema. Si un hilo se bloquea, el núcleo puede dar la CPU a otro hilo del mismo proceso o de otro proceso. Igual que para los fallos de página. |
| Inconvenientes | <ul style="list-style-type: none">• Las funciones usadas para la sincronización entre hilos son llamadas al sistema y son más costosas. Una forma de aliviar este problema es implementar las funciones de biblioteca para la sincronización, y que solo se realice una llamada al sistema cuando sea estrictamente necesario.• La creación y destrucción de hilos dentro de un proceso es también más costosa. Una posible solución sería la reutilización de estos o de sus estructuras de datos. |

La suspensión de un proceso implica intercambiar de memoria a disco el espacio de direcciones del proceso, todos los hilos deben pasar al estado *suspendido*. De igual modo la terminación de un proceso provoca la terminación de un proceso provoca la terminación de todos los hilos de ese proceso.

PLANIFICACIÓN DE PROCESO

La parte del sistema operativo que decide cuál ejecutar se llama *planificador* y el algoritmo que utiliza para tomar la decisión se llama *algoritmo de planificación*.

METAS DE LA PLANIFICACIÓN

El orden en el que un planificador selecciona los procesos listos viene determinado muchas veces por la meta o metas que se pretenden conseguir.

1. **Equidad**: **cada proceso obtiene su proporción justa de CPU**; no debemos confundir la equidad con la igualdad.
2. **Eficiencia**: **mantener la CPU ocupada al 100%** o lo más cerca posible.

$$E = \frac{\text{Tiempo útil}}{\text{Tiempo total}} \times 100 = \frac{\text{Tiempo útil}}{\text{Tiempo útil} + \text{Tiempo gestión} + \text{Tiempo ociosa}}$$

3. **Tiempo de espera**: **minimizar el tiempo que pasa un proceso en la cola de listos**.
4. **Tiempo de respuesta**: este es el tiempo que transcurre **desde que se solicita la ejecución** de una acción, **hasta que se obtienen los primeros resultados** de la acción solicitada.
5. **Tiempo de regreso o de retorno**: es el **tiempo que transcurre desde que se entrega un trabajo para que sea procesado hasta que se obtienen sus resultados**.
6. **Rendimiento o productividad**: **maximizar el nº de tareas procesadas por unidad de tiempo**.

PLANIFICACIÓN APROPIATIVA Y NO APROPIATIVA

Un planificador puede adoptar dos estrategias de funcionamiento. La *planificación apropiativa* consiste, en **poder expulsar a procesos ejecutables de la CPU** para ejecutar otros procesos (se usa para procesos interactivos). Y la *planificación no apropiativa*, consiste en **permitir a un proceso ejecutarse hasta terminar** o hasta bloquearse (se usa para lotes de trabajos).

CICLOS DE RÁFAGAS DE CPU Y E/S

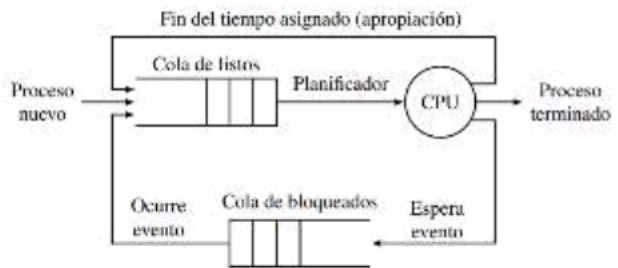
El éxito de la planificación de la CPU depende de la *ráfaga de CPU* y la *ráfaga de E/S*. Los procesos alternan entre estas ráfagas, **comenzando y terminando con una ráfaga de CPU**.

La duración de las ráfagas de un proceso no se conoce, aunque se pueden hacer estimaciones a partir de ejecuciones anteriores.

Esta alternancia entre ráfagas de CPU y de E/S de los procesos es la que hace posible la existencia de la multiprogramación.

Un proceso **limitado por E/S** es aquel que pasa la mayor parte de su tiempo en espera de una operación de E/S. Un proceso de este tipo tendrá **muchas ráfagas de CPU, aunque breves**.

Un **proceso limitado por la CPU** es aquel que necesita usar la CPU la mayor parte de su tiempo. **Un proceso así tendrá pocas ráfagas de CPU de muy larga duración**.



ALGORITMOS DE PLANIFICACIÓN

PLANIFICACIÓN PRIEMRO EN LLEGAR, PRIMERO EM SER SERVIDO (FCFS)

El proceso que **primero solicita la CPU es el primero al que se le asigna**. El algoritmo **FCFS** es **no apropiativo**, algoritmo es **fácil de implementar** con una cola FIFO.

En esta política, el **tiempo promedio de respuesta puede ser bastante largo/alto**.

Un problema de la planificación FCFS es que puede producir el denominado **efecto convoy**. Lo que pasa es que se desaprovechan los recursos. También podemos ver que el nombre *efecto convoy* viene de hecho que hay un **proceso grande y pesado (el limitado por CPU), que actúa de locomotora, seguido continuamente por un montón de procesos pequeños y ligeros (los limitados por E/S)**.

PLANIFICACIÓN PRIMERO EL TRABAJO MÁS CORTO (SJF)

SJF es un algoritmo de planificación **no apropiativo** donde **los tiempos de ejecución se aproximan**. El algoritmo consiste en **coger, de entre todos los trabajos listos, aquel que menos tiempo tarda en ejecutarse en total**.

Como no se conoce el tiempo exacto de ejecución, debemos hacer *estimaciones* basándonos en el comportamiento pasado.

$$E_t = a \cdot E_{t-1} + (1 - a) \cdot T_{t-1}$$

donde a es un parámetro llamado coeficiente de credibilidad que oscila entre 0 y 1.

A esta técnica de estimación se conoce como **maduración**. Un valor de a pequeño hace que se olviden con rapidez los tiempos de las ejecuciones anteriores y un valor de a grande hace que se recuerden durante largo tiempo.

PLANIFICACIÓN PRIMERO EL QUE TENGA EL MENOR TIEMPO RESTANTE (SRTF)

El algoritmo **SJF** se puede hacer apropiativo cuando se quita la CPU a un proceso para dársela a otro proceso listo con tiempo total de ejecución o ráfaga de CPU menor que lo que resta del proceso que se está ejecutando.

PLANIFICACIÓN ROUND ROBIN (RR) O CIRCULAR

En esta planificación los procesos **se atienden en el orden de llegada a la cola de procesos listos**, aunque se asigna a cada proceso un intervalo de tiempo de ejecución llamado quantum. La **planificación round robin** es apropiativa. Se **cambia de un proceso a otro cuando se consume su quantum, termina antes o se queda bloqueado**.

Es importante decidir la *longitud del quantum* ya que si el **quantum es pequeño** puede **pasar más tiempo realizando cambios** de proceso que ejecutando procesos. Pero si el **quantum es grande** los **últimos procesos** de la lista **tardan mucho en ser atendidos**. Esto puede suponer tiempos de respuesta muy pobres y, por tanto, una mala experiencia de usuario.

PLANIFICACIÓN POR PRIORIDAD

Lo habitual no es que los procesos tengan la misma importancia por eso en la **planificación por prioridad** cada proceso tiene asociada una prioridad y el proceso ejecutable de mayor prioridad es el que toma la CPU.

La asignación de prioridades puede ser **estática o dinámica**. También hay que decidir si la planificación por prioridad es **apropiativa o no**, dependiendo de si al llegar un nuevo proceso listo de prioridad mayor, al nuevo proceso se le da la CPU o no.

Un problema serio de los algoritmos de planificación por prioridades es el del **bloqueo indefinido**, que surge cuando los procesos de baja prioridad nunca consiguen el control de la CPU. Para evitar que los procesos de alta prioridad se ejecuten de forma indefinida, el planificador puede disminuir poco a poco la prioridad del proceso en ejecución. Otra posibilidad, que soluciona este problema, es aumentar periódicamente la prioridad de los procesos listos de baja prioridad.

PLANIFICACIÓN DE MÚLTIPLES COLAS CON REALIMENTACIÓN

Es la planificación más general de todas, **pero también la más compleja**. La idea es **tener varias colas de procesos listos**. Un proceso irá a una cola u otra en función de ciertos criterios.

A la hora de asignar la CPU a un proceso, hay que decidir de qué cola se selecciona el proceso y qué proceso, dentro de esa cola, hay que coger. Por lo tanto, debe haber una planificación entre colas y también una planificación dentro de cada cola. Debemos **permitir que los procesos puedan cambiar de una cola a otra**, en este caso habrá que establecer también los criterios para cambiar a un proceso de una cola a otra.

Para esta planificación debes tener el número de colas, el algoritmo de planificación dentro de cada cola o entre estas y el criterio de ascenso y descenso de procesos entre colas.

PLANIFICACIÓN A CORTO, MEDIO Y LARGO PLAZO

Si no se dispone de suficiente memoria, será necesario que algunos de los procesos ejecutables se mantengan en disco. El problema ahora es el coste de darle la CPU a un proceso que está en disco ya que es mayor que el dársela a un proceso que se encuentra en memoria.

Una forma práctica de trabajar con esto es por medio de un planificador de dos niveles: un **planificador a corto plazo (PCP)**, que es el que planifica los procesos que se encuentran en memoria y un **planificador a medio plazo (PMP)**, que es el que planifica el intercambio de procesos entre la memoria principal y el disco.

Periódicamente se llama al PMP para eliminar de memoria procesos que lleven mucho tiempo y para cargar en memoria los procesos que hayan estado en disco demasiado tiempo. Tras esto actúa el PCP hasta que se llame de nuevo al PMP.

Algunos de los criterios que puede utilizar el PMP para tomar decisiones sobre un proceso son:

- Los que llevan tiempo en memoria o en disco pueden ser candidatos para ser intercambiados.
- Puede interesar suspender a disco procesos que consuman mucha CPU. Si el sistema está poco cargado y hay procesos que consumen poca CPU, estos procesos pueden ser suspendidos a disco para poder traer a memoria otros procesos que puedan hacer un mayor uso de la CPU.
- Los procesos pequeños podrían no ser beneficioso el suspenderlos, pero si un proceso pequeño consume mucha CPU, podría ser necesario suspenderlo.
- Los procesos de mayor prioridad deberían permanecer en memoria, mientras que los procesos de baja prioridad podrían intercambiarse entre disco y memoria.

También es posible un **planificador a largo plazo (PLP)** que decida, de entre los trabajos por lotes preparados, cuál entra al sistema para su ejecución.