APUNTES DE

Introducción a los Sistemas Operativos

2º DE GRADO EN INGENIERÍA INFORMÁTICA

TEMA 2. GESTIÓN DE PROCESOS

CURSO 2021/2022

ÍNDICE GENERAL

CAPÍTULO 2

GESTIÓN DE PROCESOS

2.1 INTRODUCCIÓN

En un sistema de computación, hay muchos recursos que administrar. Entre ellos, uno de los más importantes (si no el que más) es el procesador o CPU (*Central Processing Unit*), ya que ningún programa se puede ejecutar si no se le concede el uso del mismo.

Dado que suele haber bastantes más procesos que procesadores, o que núcleos de procesamiento dentro de una misma CPU[1], la forma en la que se reparte el uso del procesador entre los distintos procesos es fundamental. En unos casos, se buscará la eficiencia en el uso de la CPU, como ocurría en los sistemas antiguos; otras veces, en cambio, se buscará el ejecutar varios programas dando la sensación de que todos se ejecutan a la vez, como suele ocurrir en los sistemas actuales. Esta *planificación de la CPU* será uno de los problemas que trataremos en este tema. Observa que la planificación es necesaria precisamente porque en el sistema generalmente habrá más procesos que núcleos o CPU disponibles. Si siempre hubiera un único proceso en ejecución, no tendría sentido la planificación.

Mantener un registro de distintas actividades paralelas es una tarea difícil. Por ello, para facilitar el uso del paralelismo, los diseñadores de sistemas operativos 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.

2.1.1 Concepto de proceso

Un *proceso* (o proceso secuencial) es básicamente un *programa en ejecución*. Un programa es algo estático, es el contenido de un fichero en disco. Un proceso, en cambio, es algo dinámico que en cada instante de tiempo tiene un estado concreto, el cual viene determinado por el contador de programa y los registros de la CPU usada por el proceso, por el valor de las variables definidas en el programa, etc. Como se puede ver, a diferencia de un programa, que solo ocupa espacio en disco, un proceso necesita disponer de varios recursos del sistema para su funcionamiento: memoria para almacenar el código y los datos del programa correspondiente, memoria también para su pila, CPU para poder ejecutarse, espacio en disco para leer y/o escribir ficheros, etc.

Al igual que un proceso puede constar de varios programas, como ocurre con los procesos en Unix, un mismo programa puede dar lugar a varios procesos, como sucede,

[1] Aunque los sistemas de computación actuales suelen tener varias CPU, o una CPU con varios núcleos de procesamiento, en este tema supondremos que existe una única CPU.

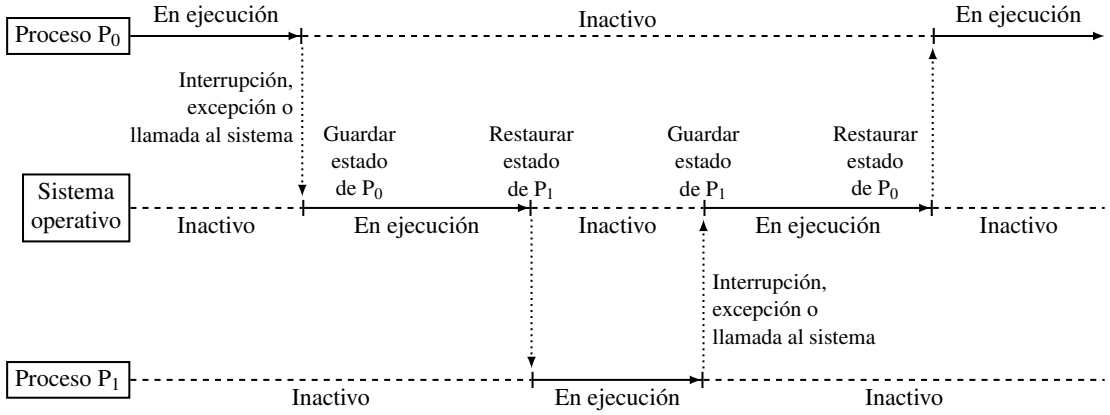


Figura 2.1. La CPU puede cambiar de un proceso a otro con la ayuda del sistema operativo, que guarda el estado de un proceso cuando este deja la CPU y recupera dicho estado cuando el proceso toma de nuevo la CPU. Como se puede observar, el sistema operativo toma el control de la CPU cuando se produce una interrupción hardware, una excepción o una llamada al sistema.

por ejemplo, cuando varios usuarios ejecutan un mismo programa en la misma máquina. Ambas ideas las desarrollaremos más adelante.

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, cuando no es así, pues en un determinado instante la CPU solo puede estar ejecutando el código de un proceso, aunque en un segundo haya podido ejecutar varios procesos. A esta sensación de que varias cosas se ejecutan al mismo tiempo se le denomina *pseudoparalelismo* (ver figura 2.1). El paralelismo real, en cambio, se da cuando hay varias CPU o cuando una CPU dispone de varios núcleos, ya que en un mismo instante de tiempo cada CPU o núcleo puede ejecutar un proceso distinto. En los sistemas actuales, donde puede haber bastantes procesos en ejecución, encontramos tanto pseudoparalelismo como paralelismo real, pues las CPU suelen disponer de varios núcleos y cada uno de ellos suele ser utilizado por varios procesos.

Tal como se ha indicado, permitir la ejecución concurrente de varios procesos es una tarea compleja. Sin embargo, existen muchas razones que hacen que esto merezca la pena, como son:

- 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. Pensemos, por ejemplo, en un ordenador con varias CPU o GPU con una elevada capacidad computacional, decenas de GiB de RAM, etc.

- 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 subtareas para que todas ellas se ejecuten en paralelo. Como es lógico, esta aceleración solo se puede conseguir si el ordenador posee múltiples elementos de procesamiento (varias CPU y/o núcleos) y cada subtarea se ejecuta en un elemento distinto.

- Modularidad, pues, muchas veces, un sistema se construye con más facilidad si se diseña como un conjunto de procesos separados.

- Comodidad, simplemente, para permitir que los usuarios hagan varias cosas a la vez, como editar, imprimir, compilar, navegar, escuchar música, etc.

Los procesos nunca deben programarse con hipótesis implícitas de tiempo, puesto que, al existir varios procesos ejecutándose, no se sabe con certeza cuándo se ejecutará cada proceso ni cuándo se atenderán sus solicitudes. Por ejemplo, un proceso no puede suponer que una operación de disco se va a atender en medio segundo y operar basándose en esa suposición, pues otros procesos (en número indeterminado) pueden haber solicitado también operaciones de disco que se atenderán antes.

Cuando un proceso deja la CPU hay que decidir qué proceso, de entre los procesos que se pueden ejecutar, pasará a la CPU. La decisión de qué proceso escoger se lleva a cabo mediante un *algoritmo de planificación* o, simplemente, *planificador*. Describiremos varios de estos algoritmos en la sección 2.5.

2.1.2 Cambio de proceso, de contexto y de modo

Como estamos viendo, nos interesa repartir la CPU entre los diferentes procesos almacenados en memoria. El hecho de que la CPU deje de ejecutar un proceso y pase a ejecutar otro se denomina *cambio de proceso*[2].

Como hemos visto en la figura 2.1, 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 partir de ese instante. A estos saltos en la ejecución de la CPU, donde se guarda el contexto del proceso que se estaba ejecutando para poder darle de nuevo la CPU más tarde de forma transparente (el proceso nunca verá que se le quitó temporalmente la CPU) los llamaremos *cambios de contexto*.

Un cambio de proceso siempre conlleva uno o más cambios de contexto, pero no al contrario. Por ejemplo, en Unix, cuando un proceso realiza una llamada al sistema, se cambia de contexto cuando se salta del proceso, que se ejecuta en modo usuario, al núcleo del sistema operativo, que se ejecuta en modo núcleo. A pesar de este salto, se supone que la ejecución del código del sistema operativo se produce dentro del mismo proceso que hace la llamada al sistema, por lo que no hay un cambio de proceso. Veremos esto con más detalle posteriormente.

Observa que en una llamada al sistema se produce un *cambio de modo*, pues la CPU pasa de modo usuario a modo núcleo (y viceversa cuando termina la llamada al sistema). Un cambio de modo supone siempre un cambio de contexto (pasamos de ejecutar un código en espacio de usuario a ejecutar otro código en espacio del núcleo), pero no un cambio de proceso, como acabamos de explicar. Un cambio de contexto, en cambio, no implica siempre un cambio de modo. Como veremos en la sección 2.4 cuando expliquemos la implementación de hilos en modo usuario, se puede producir un cambio de contexto de un hilo a otro dentro de un mismo proceso sin que haya ni cambio de proceso ni de modo.

En resumen, es importante tener claras las diferencias entre cambio de proceso, de contexto y de modo. Las relaciones entre ellos dependerán del sistema operativo utilizado.

[2] Algunos autores prefieren denominarlo *cambio de contexto*, lo cual, desde nuestro punto de vista, es incorrecto.

2.1.3 Creación y destrucción de procesos. Jerarquía de procesos

Si un sistema operativo soporta el concepto de proceso, debe proporcionar llamadas al sistema para crear y destruir procesos. Así, para crear un proceso en Unix, existe la llamada al sistema `fork()`[3], que crea una copia idéntica del proceso que hace la llamada. Es decir, tras la llamada al sistema `fork()`, el proceso que hace la llamada (padre) y el nuevo proceso (hijo) son totalmente idénticos: mismo código, mismos datos, misma pila, mismo contador de programa (que apuntará a la instrucción inmediatamente posterior a `fork()`), mismos valores en los registros de la CPU, etc. 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 identifica al nuevo proceso en el sistema (a este número se le llama *identificador de proceso* o *Process IDentifier*, PID). Después del `fork()`, padre e hijo continúan su ejecución en paralelo de forma totalmente independiente. El padre puede crear más hijos y los hijos, a su vez, pueden crear más hijos, lo que permite crear un árbol de procesos de profundidad arbitraria.

Dado que el padre y el hijo son, inicialmente, idénticos, ambos ejecutarán el mismo código y harán exactamente lo mismo. La forma de conseguir que ambos realicen tareas distintas es consultar el valor devuelto por la función `fork()`, ya que, a través de dicho valor, un proceso puede saber quién es (si el padre o el hijo), y hacer una cosa u otra en función de su condición, por ejemplo, a través de una sentencia condicional. Entre las diferentes cosas que pueden hacer, se encuentra la de ejecutar la llamada al sistema `execve()` o cualquier otra de la familia `exec()`[4]. Esta llamada hace que un proceso sustituya su código y datos por los de otro programa (pasado como argumento a la llamada al sistema) y comience la ejecución del nuevo código desde el principio usando una nueva pila. Como todo el contenido del proceso que realiza la llamada se sustituye por uno nuevo, una llamada a `execve()` que tenga éxito nunca regresará. Generalmente, esta llamada la ejecuta el proceso hijo, aunque no es obligatorio que siempre sea así.

Es importante observar que `execve()` no crea un nuevo proceso, sino que, simplemente, cambia el código y los datos por otros *dentro* del mismo proceso. Es decir, con `execve()`, el proceso sufre una *mutación* completa; sigue siendo el mismo proceso, pero con un aspecto totalmente diferente. A pesar de este cambio, hay elementos del proceso que se conservan como, por ejemplo:

- Los ficheros abiertos, aunque hay una forma de especificar el cierre de ciertos descriptores de ficheros al emitir la llamada a `execve()`.
- El tratamiento de las señales[5]. Así, las señales que se ignoren seguirán ignorándose. En cambio, las señales que se capturen dejarán de estarlo, ya que desaparece el código que tenían asociado.
- El PID y otras propiedades del proceso (propietario, etc.).

[3] En Linux, `fork()` es en realidad un envoltorio de la llamada al sistema `clone()` o de su nueva interfaz, llamada `clone3()`.

[4] Estas funciones también son envoltorios de `execve()`, que se proporcionan al programador por conveniencia.

[5] Las señales son como interrupciones que podemos enviar a un proceso. Para cada señal, los procesos suelen tener un comportamiento por defecto, que puede ser ignorar la señal o finalizar su ejecución. Hay algunas señales que un proceso puede «capturar», es decir, puede asociar cierto código a una señal para que se ejecute al recibir la misma.

Gracias a estas dos llamadas al sistema, `fork()` y `execve()`, los sistemas Unix crean todos los procesos y ejecutan todos los programas necesarios. Por ejemplo, el programa 2.1 es un pequeño intérprete de órdenes o *shell* que solicita una orden por teclado (sin argumentos) y la ejecuta usando las dos llamadas al sistema mencionadas. Como podemos ver, el proceso padre va leyendo órdenes y las va ejecutando mediante un proceso hijo. Mientras el hijo ejecuta la orden, el padre simplemente espera a que el hijo termine (usando la llamada al sistema `wait()`) antes de solicitar la siguiente orden. El hijo, por su parte, ejecuta cada orden sustituyendo su imagen actual (la del programa 2.1 mostrado) por la de la orden a ejecutar mediante `execlp()`[6].

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(void)
{
  pid_t pidhijo;
  char orden[80];

  while(1) {
    printf("$ ");
    fgets(orden, 80, stdin);
    orden[strlen(orden)-1]='\0'; // Eliminamos \n.

    pidhijo = fork();
    if (pidhijo == 0 ) {
      execlp(orden, orden, NULL);
      printf("La orden %s falló\n", orden);
      exit(1);
    }
    wait(NULL);
  }

  return 0;
}
```

Programa 2.1. Un pequeño intérprete de órdenes (`sh.c`).

La figura 2.2 muestra gráficamente lo que ocurre en el programa anterior cuando la orden a ejecutar es `ls`. Observa que el PID del proceso hijo no cambia a pesar de que su código sí lo hace tras la llamada al sistema `execlp()`.

En los sistemas Windows, la llamada al sistema para crear procesos es `CreateProcess()`. Esta llamada, en realidad, equivale a ejecutar a la vez `fork()`, para crear el nuevo proceso, y `execve()`, para que el nuevo proceso hijo ejecute un nuevo programa. Por lo tanto, no es posible ejecutar, por un lado, algo equivalente a `fork()` y, por otro, algo equivalente a `execve()`.

Esta forma distinta de crear procesos en Windows también se refleja en su jerarquía de procesos. Así, mientras el sistema operativo construye y gestiona esa jerarquía en los sistemas Unix, manteniendo las relaciones entre padres e hijos y permitiendo ciertas operaciones solo entre ellos, en Windows, un proceso hijo puede pasar a ser hijo de otro

[6]El primer parámetro de `execlp()` es el nombre del fichero que contiene el programa a ejecutar; el segundo parámetro es el nombre del programa que recibirá el proceso (habitualmente, es el mismo que el del fichero); el resto de parámetros si existen, serán cadenas de caracteres que se pasarán como argumentos del programa a ejecutar; el final de la lista se indica con `NULL`.

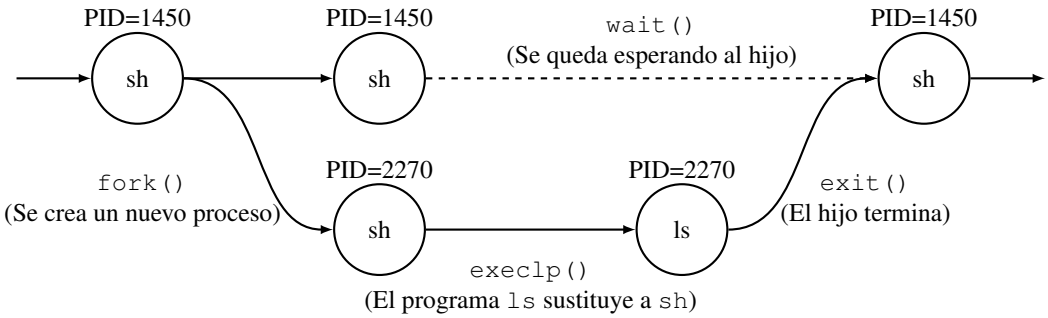


Figura 2.2. Uso conjunto de las llamadas al sistema `fork()` y `exec()` para ejecutar la orden `ls` en el programa 2.1.

proceso que no sea su padre original. Es decir, un proceso puede transferir la «propiedad» de uno de sus hijos a otro proceso. Esto hace que, en Windows, el concepto de jerarquía, al estilo de Unix, desaparezca[7].

De la misma forma que debe ser posible crear un proceso, también debe existir algún mecanismo para poder finalizarlo y liberar así los recursos que pueda estar consumiendo. Un proceso puede terminar de forma *voluntaria* o *involuntaria*. En el primer caso, es el propio proceso el que decide finalizar su ejecución, bien porque ha terminado su tarea o bien porque ha encontrado algún error (un dato inválido, un fichero que no existe, etc.) que le impide continuar su ejecución. Para esta finalización voluntaria, en Unix existe la llamada al sistema `exit()` y en Windows `ExitProcess()`. En la terminación involuntaria, en cambio, un proceso es finalizado por el propio sistema operativo (por ejemplo, cuando el proceso provoca un error fatal como una división por cero, un acceso incorrecto a una zona de memoria, ...), o por otro proceso, siempre que tenga autorización para ello. Para este último caso, es decir, para que un proceso pueda acabar con otro, en Unix existe la llamada al sistema `kill()` y en Windows `TerminateProcess()`.

2.2 ESTADOS DE UN PROCESO

En su concepción más simple, un proceso puede estar en 3 estados posibles (ver figura 2.3):

- *En ejecución*: el proceso está utilizando la CPU.
- *Listo*: el proceso es ejecutable, pero la CPU está siendo usada por otro proceso, por lo que se encuentra detenido de forma temporal a la espera de que le llegue su turno de CPU.
- *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 (por ejemplo, que un hijo termine, que le lleguen los datos de una operación de lectura que ha solicitado, etc.).

Si hay una única CPU (con un único núcleo de procesamiento), solo un proceso puede estar en ejecución; sin embargo, varios procesos pueden estar en estado listo o bloqueado a la misma vez.

[7] En Unix, un proceso también puede pasar a ser hijo de otro proceso, pero solo si su padre desaparece; mientras su padre exista, será su hijo sin poder pasar a ser hijo de otro proceso.

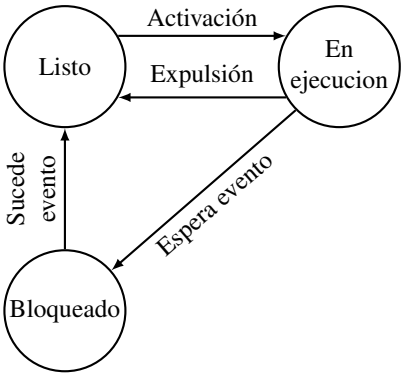


Figura 2.3. Estados básicos de un proceso (en ejecución, bloqueado y listo) y las transiciones entre ellos.

Entre estos estados, existen 4 posibles transiciones, mostradas también en la figura 2.3:

- De «en ejecución» a «bloqueado»: cuando un proceso no puede continuar su ejecución por algún motivo (faltan datos, recurso no disponible, etc.), debe bloquearse.
- De «listo» a «en ejecución» y viceversa: se debe al planificador. Un proceso pasa de estar en ejecución a estar listo cuando es expulsado de la CPU por haber excedido su tiempo de uso de CPU, o cuando se debe ejecutar un proceso más importante. Un proceso listo pasa a ejecutarse cuando la CPU queda libre y, según el planificador, es, de entre todos los procesos listos, al que le corresponde usar la CPU; también un proceso pasa de listo a en ejecución si es más importante que el proceso que actualmente ocupa la CPU, el cual será expulsado. Todas estas decisiones del planificador las veremos con más detalle en la sección 2.5.
- De «bloqueado» a «listo»: el proceso ya dispone de lo que necesitaba (datos, recursos, etc.). Si la CPU está desocupada, pasará a ejecutarse de forma inmediata.

El diagrama de estados anterior es demasiado sencillo para un sistema moderno, ya que existen diversos factores que hacen que un proceso pueda estar en más estados. Por ejemplo, cuando un proceso se crea, es posible que no disponga de todos los recursos necesarios para su ejecución de forma inmediata, por lo que no podrá pasar al estado listo hasta que no disponga de dichos recursos. Esto hace que aparezca el estado «Nuevo», en el que un proceso recién creado permanecerá hasta disponer de todos los recursos necesarios.

De la misma forma, desde que un proceso termina su ejecución hasta que desaparece completamente puede transcurrir un cierto tiempo. Esto da lugar al estado «Saliente». En Unix, por ejemplo, esto ocurre cuando un proceso termina (voluntaria o involuntariamente) y debe esperar a que su proceso padre recoja su estado de salida mediante la llamada al sistema `wait()`.

La figura 2.4 muestra cómo queda la figura 2.3 si incluimos estos dos nuevos estados, así como las transiciones que se pueden dar. Es importante observar que, desde «Nuevo», «Listo», «Bloqueado» y «En ejecución», se puede pasar al estado «Saliente» en cualquier momento. Esto es así porque, por ejemplo, un proceso puede ser «matado» por otro proceso (con permisos suficientes para ello), lo que hará que cambie directamente al estado «Saliente» desde el estado en el que se encontraba. Aunque estas transiciones son posibles, no se muestran en la figura 2.4 por motivos de claridad.

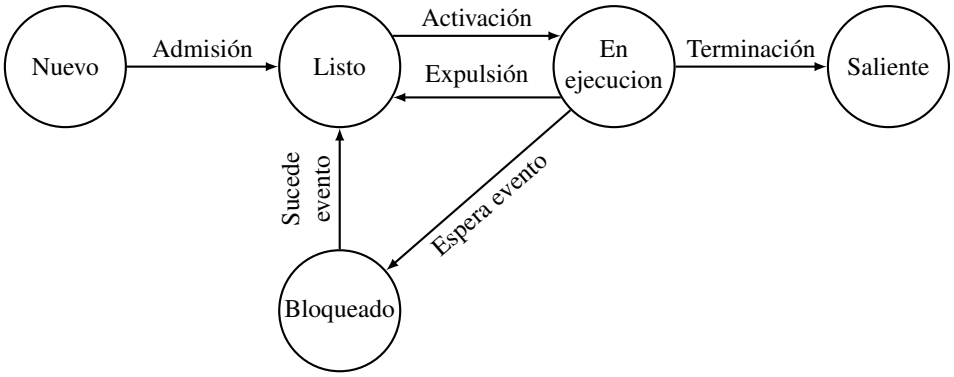


Figura 2.4. Transiciones de estado de los procesos, incluidos los estados «Nuevo» y «Saliente». Las transiciones desde cualquier estado al estado «saliente» existen, pero no se muestran para que la figura quede más «limpia».

Existen 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. Cuando un proceso se suspende, no puede ejecutarse ni verse afectado por la planificación de procesos. Estas operaciones de suspensión y reanudación son útiles por diversas razones:

- El sistema está funcionando mal: se pueden suspender procesos activos hasta corregir el problema y luego reanudarlos.
- Depuración: 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. En caso contrario, se finaliza el proceso.
- El sistema está muy cargado: se puede reducir la carga suspendiendo procesos para así poder dar servicio a los procesos de mayor prioridad, etc.

Ya que los procesos suspendidos no se pueden ejecutar (por lo tanto, no van a usar la CPU), generalmente se guardan en disco para liberar la memoria principal que ocupan, la cual quedará disponible para otros procesos.

En la figura 2.5 podemos ver cómo queda la figura 2.4 si incluimos la suspensión y la reanudación. También podemos ver algunas de las nuevas transiciones que aparecen. De ellas, merece la pena destacar las siguientes:

- De «bloqueado» a «bloqueado suspendido»: cuando interesa pasar un proceso bloqueado a disco para liberar memoria y que otros procesos listos (por ejemplo, los que pueda haber suspendidos en disco) puedan usarla y ejecutarse.
- De «bloqueado suspendido» a «listo suspendido»: cuando el evento por el que se bloqueó el proceso se produce mientras este está suspendido.
- De «listo suspendido» a «listo»: cuando no queden procesos listos, necesitaremos llevar algunos procesos listos de disco a memoria para que la CPU no quede ociosa. También puede suceder que haya quedado memoria libre, o que un proceso listo suspendido deba ejecutarse por tener más prioridad que los procesos que haya en memoria.
- De «listo» a «listo suspendido»: normalmente, se suspenden procesos bloqueados, ya que están consumiendo memoria cuando no se pueden ejecutar, pero, para

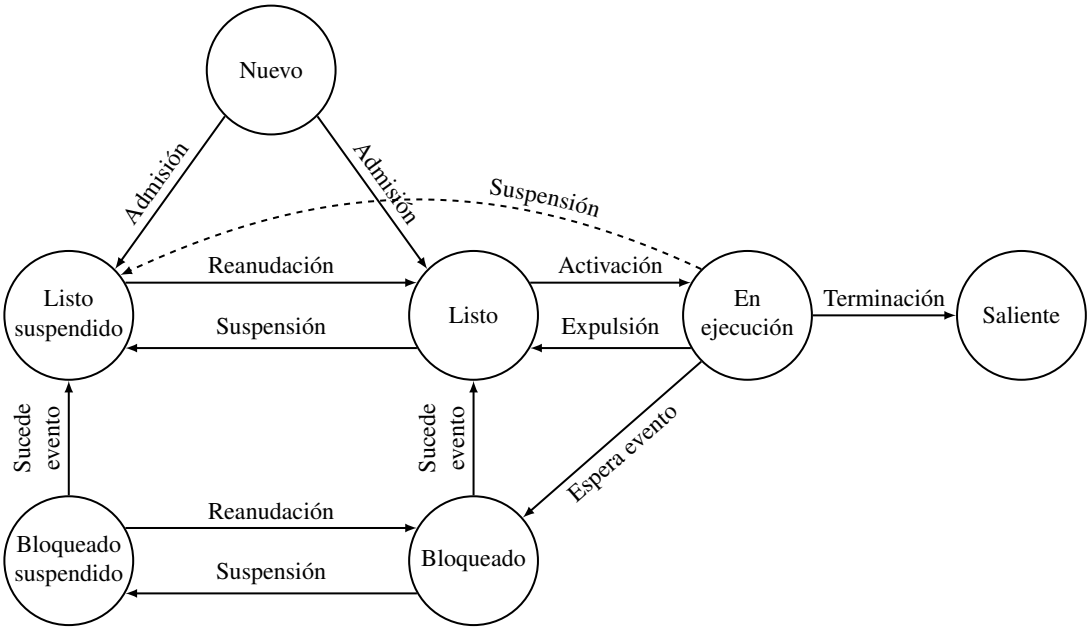


Figura 2.5. Transiciones de estado de los procesos, incluidas la suspensión y la reanudación.

liberar memoria o reducir la carga del sistema, en algunos casos se puede preferir suspender un proceso listo de baja prioridad antes que uno bloqueado de alta prioridad que necesitará ejecutarse lo antes posible una vez pase al estado listo.

- De «nuevo» a «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.
- De «bloqueado suspendido» a «bloqueado»: normalmente, un proceso bloqueado suspendido no se pasa a memoria principal porque, aunque lo haga, no podrá usar la CPU. Sin embargo, si un proceso bloqueado suspendido tiene una prioridad alta, y el sistema operativo sospecha (usando cierta heurística[8]) que el evento que espera sucederá en breve, entonces puede tener sentido mover el proceso bloqueado de disco a memoria.
- De «en ejecución» a «listo suspendido»: puede tener sentido cuando un proceso de prioridad alta despierta de «Bloqueado suspendido» y no hay memoria suficiente si no se pasa inmediatamente a disco precisamente el proceso al que se le quita la CPU.
- De cualquier estado a «saliente»: en cualquier momento, un proceso puede finalizar.

2.3 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. Cada una de estas entradas se llama PCB (*Process Control Block*) o, en

[8]Según una de las acepciones de la RAE: «En algunas ciencias, manera de buscar la solución de un problema mediante métodos no rigurosos, como por tanteo, reglas empíricas, etc.»

Administración de procesos	Administración de memoria	Administración de ficheros
Registros	Dirección del segmento de texto[9]	Directorio raíz
Contador del programa	Dirección del segmento de datos	Directorio de trabajo
Palabra de estado del programa	Dirección del segmento BSS	Descriptores de fichero
Puntero de pila	Dirección del segmento de pila	Identificación de usuario
Estado del proceso		Identificación de grupo
Prioridad		
Parámetros de planificación		
Identificador de proceso		
Proceso padre		
Hora de inicio del proceso		
Tiempo utilizado de CPU		

Figura 2.6. Algunos de los campos más comunes en una entrada de la tabla de procesos.

español, BCP (*Bloque de Control de Proceso*), y en ella se guarda toda la información relacionada con el proceso correspondiente. La figura 2.6 muestra qué información podría contener un PCB. Algunos campos tienen un significado claro. Otros, como la identificación de usuario y de grupo, los veremos en temas posteriores y en prácticas.

En el PCB se debe guardar, al menos, 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 como si nada hubiera pasado (recordemos lo que pasa con el proceso P0 en la figura 2.1). También se suelen guardar en el PCB datos estadísticos y de otro tipo.

2.3.1 Creación de procesos

Tal como hemos indicado, un proceso necesita hacer uso de diferentes recursos para poder ejecutarse. Por eso, la creación de un proceso es quizás una de las operaciones más complicadas, ya que debemos proveer al proceso de todos los recursos que precise en un primer momento. Cuando se crea un proceso, al menos, se debe:

1. Dar nombre al proceso (generalmente, es un número que lo identifica, como el PID en Unix, mencionado anteriormente).

2. Insertarlo en la tabla de procesos, creando su PCB.

3. Determinar su prioridad inicial (hablaremos de la prioridad cuando veamos los algoritmos de planificación).

4. Asignarle recursos iniciales.

Como ya hemos dicho, en el caso de Unix, la creación de procesos se realiza tradicionalmente mediante la llamada al sistema `fork()`. En este caso, los pasos concretos que se siguen son:

[9]En todo proceso en ejecución podemos diferenciar, al menos, tres segmentos o zonas de memoria: la que contiene el código del proceso (también llamado *texto*), la que contiene los datos, y la pila. La zona de datos, a veces, no se ve como una única zona, sino como dos: una para las variables que ya tienen un cierto valor inicial desde que comienza la ejecución del proceso (podemos pensar en variables globales a las que les asignamos un valor dentro del propio código fuente del programa) y otra para las variables que no tienen ningún valor inicial. A esta segunda zona se le llama *BSS* (*Block Starting Symbol*). En los sistemas operativos actuales, los procesos pueden tener bastantes más zonas, como veremos en el tema dedicado a la administración de la memoria.

1. El proceso padre realiza la llamada al sistema `fork()`, 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 de la entrada asignada al padre en la tabla de procesos se copia a la entrada del hijo (recordemos que `fork()` crea un hijo que es idéntico a su padre), con algunas excepciones (PID, información sobre los segmentos de datos y pila, etc.).

4. Se asigna memoria para los segmentos de datos y de pila del hijo y se copian los segmentos del padre en ellos. Como el segmento de código es de solo lectura (por razones de seguridad, un proceso nunca debe modificar su código en memoria), el segmento de código es compartido por padre e hijo[10].

5. Se incrementan los contadores asociados a cualquier fichero abierto por el padre para reflejar que ahora esos ficheros también están abiertos 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, como resultado de la llamada `fork()`.

Llegados a este punto, el hijo está listo para su ejecución.

2.3.2 Estructura de un proceso

Un proceso es, como hemos dicho, un programa en ejecución. Ahora bien, cuando se produce, por ejemplo, una llamada al sistema y la CPU pasa a ejecutar código del sistema operativo para llevarla a cabo, ¿debemos considerar que seguimos dentro del mismo proceso o, por el contrario, debemos pensar que hemos cambiado de proceso? La respuesta a esta pregunta depende de cada sistema operativo y de cómo implemente este los procesos. Por comodidad, vamos a responder a esta pregunta teniendo en cuenta la implementación típica de procesos en Unix.

En Unix, 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 al sistema. Por lo tanto, cada proceso tiene dos partes: la que se ejecuta en espacio de usuario, fuera del sistema operativo, y la que se ejecuta en espacio del núcleo, dentro del sistema operativo. La parte de usuario ejecuta un programa normal y suele diferir de un proceso a otro, puesto que cada uno ejecutará un programa distinto. En cambio, la parte del núcleo es común para todos los procesos, ya que existe una única copia del sistema operativo en memoria. Esta situación se puede observar en la figura 2.7.

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. Por ello, para que los procesos no interfieran entre sí, la parte del núcleo de cada proceso cuenta con su propio contador de programa y su propia pila. El resultado es similar al que se consigue cuando tenemos varios hilos de ejecución dentro de un mismo proceso, como veremos en la sección 2.4.

Es posible que una llamada al sistema haga que un proceso se bloquee dentro del núcleo (por ejemplo, la lectura de un fichero), lo que provocará que dicho proceso ceda

[10] En el tema de administración de memoria veremos cómo los procesos pueden compartir un mismo código e, incluso, otros segmentos, como el de datos.

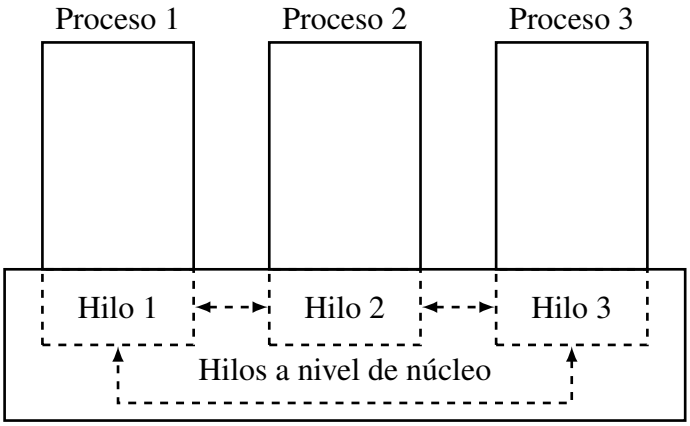


Figura 2.7. Estructura de los procesos en Unix, cada uno con su parte de usuario y su parte del núcleo.

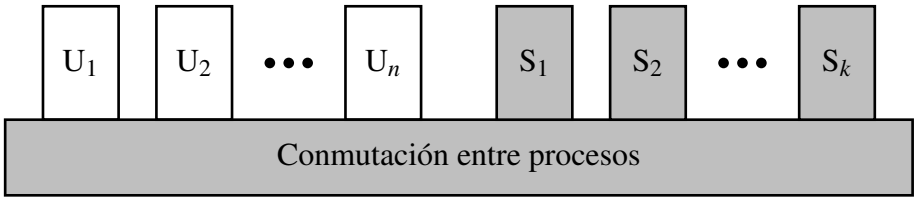


Figura 2.8. Diseño de sistema operativo en el que el propio sistema operativo es una colección de procesos S_i separada de los procesos de usuario U_j.

la CPU a otro proceso. Cuando el proceso se desbloquee, continuará su ejecución dentro del núcleo hasta finalizar el procesamiento de la llamada al sistema y regresar a su parte de usuario.

Esta estructura de un proceso, y su relación con el sistema operativo, es una de las varias posibles. La figura 2.8 muestra otro enfoque, donde el sistema operativo es una colección de procesos de sistema, distintos de los procesos de usuario. A diferencia del enfoque anterior, cada proceso de usuario solo ejecuta código de usuario, mientras que los procesos de sistema ejecutan código del sistema operativo. Este enfoque lo hemos visto también en el tema anterior, pues se corresponde con la estructura de sistemas operativos que sigue el modelo cliente-servidor.

2.3.3 Cambio de proceso

Un aspecto clave del modelo de procesos es el cambio automático y transparente de un proceso a otro. Por motivos de seguridad, este cambio solo lo puede hacer el sistema operativo cuando toma el control de la CPU, lo cual suele ocurrir cuando se produce una interrupción (como una interrupción del reloj), una llamada al sistema solicitando algún tipo de servicio, o una excepción (por ejemplo, una división por cero).

Los pasos que se dan para cambiar de un proceso a otro, y lo que se hace en cada uno de ellos, dependen del hardware y del sistema operativo utilizados, si bien el resultado final es el mismo. A continuación vamos a ver un ejemplo simplificado de los posibles pasos que se podrían dar suponiendo que el sistema operativo toma el control de la CPU por una llamada al sistema. Vamos a tener en cuenta, además, una implementación de procesos como la de Unix, donde cada proceso tiene una parte de usuario y otra de núcleo. En cualquier caso, como decimos, en función del hardware y del sistema operativo concretos que tengamos, el procedimiento final podría ser ligeramente distinto

al descrito aquí:

1. Al producirse la llamada al sistema, el hardware almacena el contador de programa del proceso que la realiza en la pila de este. 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 desde el vector de interrupciones. El nuevo contador debe contener la dirección de inicio de alguna rutina perteneciente al sistema operativo.

3. Un procedimiento en lenguaje ensamblador[11], cuya dirección se ha cargado en el contador de programa en el paso anterior, guarda el «contexto» (resto de registros del procesador diferentes al contador de programa, y otros datos) en el PCB del proceso activo que, como hemos indicado, 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, etc.).

4. Un procedimiento en lenguaje ensamblador configura la nueva pila que va a utilizar el núcleo mientras se atiende la llamada al sistema. Este paso es necesario porque el núcleo del sistema operativo no deja de ser un programa y, como tal, necesita una pila para poder llamar a funciones, pasarles parámetros, definir variables locales en las funciones, etc.

   Como puede haber varios procesos a la vez ejecutando código del sistema operativo, cada uno tendrá una pila para cuando ejecute código en modo núcleo. Esta pila suele ser diferente a la que se emplea en modo usuario. Un motivo es que, en muchos sistemas operativos actuales, los procesos utilizan memoria virtual cuando se ejecutan en modo usuario, pero no cuando lo hacen en modo núcleo (hablaremos de la memoria virtual en un tema posterior).

   Antes de avanzar al paso siguiente, es posible que el procedimiento en ensamblador deba realizar algún que 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[12]. Durante la ejecución de la llamada al sistema, el proceso se puede bloquear, por lo que su estado debe cambiar en consecuencia.

6. Tras la ejecución del procedimiento que lleva a cabo el procesamiento de la llamada al sistema, se vuelve al procedimiento en lenguaje ensamblador. Este consulta entonces si hay que ejecutar el planificador de procesos y, en caso afirmativo, lo ejecuta, tras lo cual se regresará de nuevo al procedimiento en ensamblador. Hay que tener en cuenta que, si el proceso se ha bloqueado durante la ejecución del procedimiento de la llamada al sistema, es posible que el planificador ya haya sido invocado, por lo que puede que ya no sea necesario llamarlo de nuevo.

   Como veremos poco después, el planificador de procesos es un procedimiento en C que decide cuál es el proceso listo que se ejecutará a continuación consultando la información almacenada en la tabla de procesos. Una vez seleccionado un proceso, modifica la información necesaria para que el cambio de proceso sea correcto.

[11] Este procedimiento se suele escribir en ensamblador porque necesita acceder a ciertos elementos del hardware que son específicos de la máquina, como pueden ser los registros internos de la CPU.

[12] Muchos sistemas operativos están escritos en C, de ahí que se haga referencia a este lenguaje de programación.

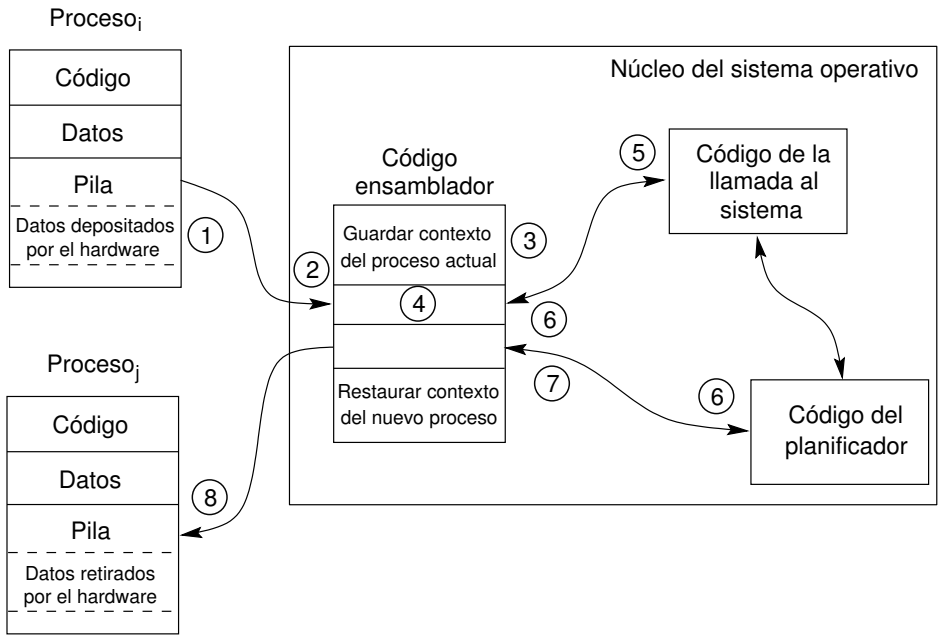


Figura 2.9. Mecanismo de cambio de proceso durante una llamada al sistema.

7. El procedimiento en ensamblador restaura el contexto del proceso a ejecutar, es decir, carga en los registros del procesador los valores que había en el momento en el que se le quitó la CPU al proceso al que se le va a conceder ahora. Entre otras cosas, prepara la pila que el nuevo proceso usará en modo usuario. Esta pila contendrá el valor que se cargará al contador de programa tras finalizar el procesamiento de la llamada al sistema, es decir, contendrá la dirección de la siguiente instrucción a ejecutar tras la llamada al sistema. 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, que será una instrucción del nuevo proceso elegido por el planificador.

Observa que los pasos 1 y 8 son complementarios (en el primero se pasa de modo usuario a núcleo, y en el último de modo núcleo a usuario), al igual que los pasos 3 y 7 (en el 3 se guarda el contexto y en el 7 se restaura).

Los pasos anteriores se muestran de forma gráfica en la figura 2.9. Cuando existen varias CPU, estos pasos se complican, pues un proceso puede cambiar de CPU durante su ejecución. No obstante, la idea es, fundamentalmente, la misma.

Es importante observar que, 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 (en ambos casos, al pasar del programa de usuario al núcleo y viceversa).

2.3.4 Operaciones con procesos

Algunas operaciones que debe ofrecer un sistema operativo que soporte el concepto de proceso son:

- Crear/destruir un proceso.
- Suspender/reanudar un proceso.
- Cambiar la prioridad de un proceso (relacionado con el planificador de procesos).
- Bloquear un proceso.
- Poner un proceso en estado listo.
- Ejecutar un proceso.
- Permitir que un proceso se comunique con otro.

Algunas de estas operaciones son internas del propio sistema operativo (como pasar un proceso al estado listo), mientras que otras las puede utilizar un programa de usuario a través de alguna llamada al sistema (como crear un proceso).

2.4 HILOS

Hasta ahora, hemos presentado el concepto de proceso como algo que engloba las dos siguientes características:

- Unidad de propiedad de recursos: a un proceso se le asigna una porción de memoria, que contiene la imagen del proceso, así como el uso de otros recursos tales como ficheros, dispositivos de E/S, etc.
- Unidad de planificación y ejecución: un proceso es una traza de ejecución a través de uno o más programas. Esta ejecución se puede intercalar con la de otros procesos. Por tanto, un proceso tiene un estado (en ejecución, listo, ...), una prioridad y un contador de programa, y es la entidad que es planificada y ejecutada por el sistema operativo.

En la mayoría de los sistemas operativos estas dos características son, de hecho, la esencia de un proceso. Sin embargo, un análisis más detallado nos hará ver que estas dos características son independientes y que el sistema operativo las podría tratar por separado, como se hace en los sistemas operativos modernos (Windows, Linux, etc.). Para distinguir estas dos características, nos referiremos a la unidad de planificación y ejecución como *hilo* o *proceso ligero* (*Light-Weight Process*, LWP)[13], y seguiremos hablando de proceso o tarea para la unidad de propiedad de recursos.

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. La figura 2.10 muestra estos dos casos.

Para entender mejor el concepto de hilo, analicemos el programa 2.2[14]. Cuando comienza la ejecución del programa, se crea un proceso en el que existe inicialmente un

[13] Un hilo puede recibir más nombres como hebra, subproceso o *thread*.

[14] En Linux, si el programa lo guardamos en un fichero llamado `creahilo.c`, lo podemos compilar desde la línea de órdenes ejecutando `gcc -o creahilo creahilo.c -lpthread`.

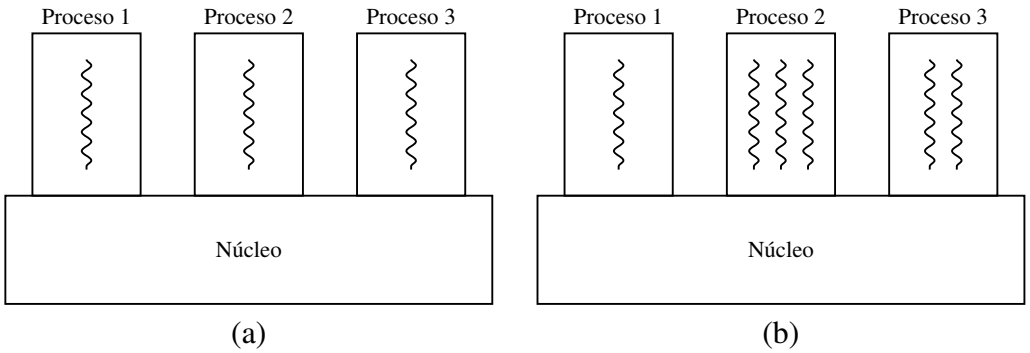


Figura 2.10. Comparación entre procesos e hilos. (a) Procesos tradicionales donde cada proceso tiene un único hilo de ejecución. (b) Procesos modernos con uno o más hilos de ejecución por proceso.

único hilo que llamamos *hilo principal*. Este hilo ejecuta la función `main()`, la cual crea, a su vez, un *hilo hijo* mediante la función `pthread_create()` [15], que es la habitual en muchos sistemas Unix. Uno de los parámetros de `pthread_create` es el nombre de otra función dentro del mismo programa. El hilo hijo comienza su ejecución en esta segunda función que actúa, por tanto, como su función `main()` particular. El programa también hace uso de las llamadas al sistema `getpid()` y `gettid()` para mostrar que existen dos hilos distintos dentro de un mismo proceso, ya que cada hilo mostrará un identificador de hilo (o *thread identifier*, TID) distinto, pero un mismo PID.

```
1  #include <pthread.h>
   #include <stdio.h>
3  #include <unistd.h>
   #include <sys/types.h>
5
   #include <sys/syscall.h>
7
   void * hilo_main (void * arg)
9  {
     fprintf (stderr, "\tEl PID del hilo hijo es: %d\n", (int)
        getpid());
11   fprintf (stderr, "\tEl TID del hilo hijo es: %d\n", (int)
        syscall(SYS_gettid));

13   /* Bucle infinito del hilo hijo. */
     while (1);
15
     return NULL;
17 }

19 int main (void)
   {
21   pthread_t thread;
```

[15] `pthread_create` recibe cuatro parámetros: un puntero a una variable de tipo `pthread_t` que, en caso de éxito, almacenará una referencia al nuevo hilo, un puntero a una estructura de datos de tipo `pthread_attr_t` para especificar atributos para el nuevo hilo (puede ser un puntero nulo, como en nuestro caso, si no queremos especificar ningún atributo), el nombre del procedimiento donde comenzará la ejecución del nuevo hilo (`hilo_main` en nuestro ejemplo), y un puntero a una estructura de datos genérica que se pasará como argumento a dicho procedimiento (en nuestro caso, también es un puntero nulo pues no vamos a pasar ningún argumento al procedimiento principal del hilo).

```
23    fprintf (stderr, "\nEl PID del hilo principal es: %d\n", (
          int)getpid());
      fprintf (stderr, "El TID del hilo principal es: %d\n\n", (
          int)syscall(SYS_gettid));
25
      /* Creamos un hilo cuya ejecución comienza en la función
          hilo_main. */
27    pthread_create(&thread, NULL, &hilo_main, NULL);

29    /* Bucle infinito del hilo padre. */
      while (1);
31
      return 0;
33  }
```

Programa 2.2. Un programa donde el hilo principal crea un hilo hijo.

2.4.1 Elementos por hilo y por proceso

El concepto de hilo, y el hecho de que un proceso pueda tener varios hilos, hace que se produzca una separación entre lo que se considera que sigue perteneciendo a un proceso y lo que se considera que pertenece a un hilo. Los siguientes elementos son algunos de los que se suele asociar a los procesos (ver figura 2.11):

- Zona de memoria o espacio de direcciones que contiene la imagen del proceso.
- Acceso protegido al procesador, otros procesos (mediante mecanismos de comunicación entre procesos), ficheros y recursos de E/S (dispositivos).
- Variables globales, procesos hijos, alarmas y señales.
- Información contable, etc.

Dentro de un proceso, puede haber uno o más hilos, cada uno con (ver figura 2.11):

- Un estado (ejecución, listo, . . .).
- Un contexto del procesador guardado cuando no se está ejecutando el hilo para que pueda continuar su ejecución cuando se le vuelva a asignar la CPU. Este contexto contendrá, al menos, el registro contador de programa y el resto de registros de la CPU. Puesto que un contexto especifica un punto de ejecución dentro de un programa, podemos decir que un hilo es como un contador de programa independiente funcionando dentro de un proceso.
- Una pila de ejecución.
- Algún almacenamiento por hilo para las variables locales. Este almacenamiento puede ser la pila del hilo u otra zona de memoria reservada dentro del proceso para los hilos.
- Acceso a la memoria y recursos del proceso al que pertenece el hilo, compartido con todos los restantes hilos del proceso.

Elementos por hilo	Elementos por proceso
Contador de programa	Espacio de direcciones
Registros de la CPU	Variables globales
Pila	Ficheros abiertos
Estado	Procesos hijos
	Alarmas
	Señales
	Información contable

Figura 2.11. Conceptos habituales por hilo y por proceso.

Como se deduce, 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. Puesto que cada hilo puede tener acceso a cada dirección de memoria de su proceso, un hilo puede leer, escribir o limpiar de manera completa la pila de otro hilo. En otras palabras, no existe protección entre los hilos debido a que, por un lado, es imposible (el hardware y el sistema operativo protegen procesos enteros, no hilos individuales) y, por otro lado, no debe ser necesaria, ya que si un proceso tiene varios hilos es porque el programador lo ha decidido así, y es de esperar que dichos hilos colaboren y no luchen entre sí, como podría pasar con procesos distintos de diferentes usuarios. Para facilitar esta colaboración, y puesto que varios hilos pueden necesitar acceder a y modificar las mismas variables globales, el sistema operativo proporciona distintos mecanismos de sincronización entre hilos, como *mutex* y semáforos (estos mecanismos se estudian en otras asignaturas de la carrera y, por tanto, no se verán aquí).

2.4.2 Aplicaciones de los hilos

Existen varias razones por las que podemos querer usar varios hilos dentro de un proceso en lugar de varios procesos. Algunas de ellas son:

- Puesto que los hilos comparten un mismo espacio de direcciones, se pueden comunicar entre sí sin intervención del núcleo, con lo que la comunicación entre hilos es más rápida que entre procesos.
- Los hilos se pueden bloquear a la espera de que se termine una llamada al sistema, al igual que ocurre con un proceso normal (es decir, con un único hilo). Esto permite acelerar la ejecución de un proceso al solaparse 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.
- Los hilos aportan beneficios desde el punto de vista del rendimiento, pues 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 si no es el último, pues no hay que liberar recursos, y menos tiempo conmutar, es decir, hacer un cambio de contexto entre 2 hilos dentro de un mismo proceso, ya que, entre otras cosas, no hay que cambiar la configuración del hardware para la protección de procesos.
- Si se dispone de varias CPU, o de una CPU con varios núcleos, se puede conseguir paralelismo real dentro de un mismo proceso: puede haber varios hilos del proceso ejecutándose al mismo tiempo en CPU diferentes o en núcleos distintos.

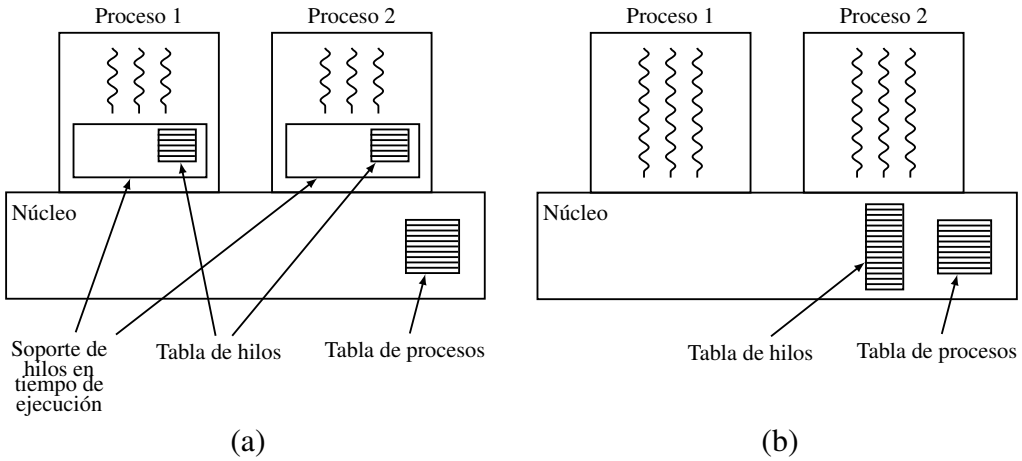


Figura 2.12. (a) Hilos implementados en espacio de usuario. (b) Hilos implementados en espacio del núcleo.

- En ciertos casos, los hilos también facilitan la construcción de programas, ya que, si un programa realiza varias funciones diferentes, cada función puede ser desempeñada por un hilo.

Veamos algunos ejemplos en relación con este último punto. A la hora de implementar un servidor de ficheros, podemos hacer que, cuando llegue una nueva petición, el hilo principal cree un nuevo hilo secundario para atenderla. De esta manera, el hilo principal puede dar paso a varias peticiones creando un hilo secundario para cada una. Los hilos secundarios son los que se bloquean esperando a que termine la operación de disco; el principal no se bloquea. Dependiendo de la funcionalidad a proporcionar, el servidor de ficheros también se podría implementar con procesos hijos y no con hilos, pero, como hemos dicho, es más costoso crear y finalizar procesos que hilos, por lo que el rendimiento ofrecido sería inferior.

Otro ejemplo lo tenemos en un programa en modo gráfico que interactúa directamente con el usuario. En este caso, un hilo puede desplegar menús y leer la entrada del usuario mientras otro hilo ejecuta órdenes previas del usuario y actualiza la ventana de la aplicación. El usuario percibiría un incremento en la velocidad.

Finalmente, un tercer ejemplo es un procesador de texto, en el que un hilo puede activarse cada minuto para escribir en disco las modificaciones que hay en RAM y así evitar pérdidas por un fallo de energía.

2.4.3 Hilos en modo usuario e hilos en modo núcleo

Los hilos pueden ser soportados directamente por el núcleo a través de un conjunto de llamadas al sistema (como ocurre en Windows y Linux) o ser implementados por encima del núcleo mediante un conjunto de llamadas de biblioteca a nivel de usuario. La figura 2.12 muestra gráficamente las dos posibles implementaciones. Cada una de estas opciones tiene sus ventajas e inconvenientes.

En la implementación en modo usuario, las ventajas son:

- El núcleo del sistema operativo no sabe que existen, por lo que se pueden usar en núcleos que no los implementan (es lo que ocurría en las primeras versiones de Linux). En este caso, 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 del sistema operativo para realizarlos.
- Cada proceso puede tener su propio algoritmo de planificación para los hilos.

Sin embargo, también surgen inconvenientes, algunos de ellos importantes:

- Puesto que el sistema operativo no es consciente de la existencia de varios hilos, cualquier llamada al sistema bloqueante realizada por uno de los hilos bloqueará a todo el proceso y sus hilos, puesto que el sistema operativo pasará la CPU a otro proceso. Esto hace que sea imposible solapar la E/S de un proceso con su cómputo. Una posible solución sería utilizar llamadas al sistema no bloqueantes, pero esto dificultaría la programación al tener que llevar un control de posibles tareas pendientes.
- De forma similar, un fallo de página[16] 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 real dentro de un mismo proceso. El sistema operativo repartirá las CPU entre los procesos, y no entre los hilos, por lo que todos los hilos de un proceso usarán la misma CPU.

Debido a los inconvenientes anteriores, los sistemas operativos modernos suelen implementar los hilos dentro del núcleo. Las ventajas que surgen ahora son las siguientes:

- El núcleo mantiene la tabla de hilos y reparte la CPU entre todos ellos. A diferencia de antes, si hay varias CPU, varios hilos de un mismo proceso se pueden ejecutar a la vez en CPU distintas, 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.
- Los fallos de página tampoco representan un problema, pues, al igual que con las llamadas al sistema bloqueantes, se puede pasar a otro hilo del mismo proceso o de otro proceso.

Como vemos, con la implementación de hilos en modo núcleo, los inconvenientes anteriores son ahora ventajas. De la misma manera, las ventajas anteriores desaparecen y surgen algunas aquí como inconvenientes. En concreto, los dos principales inconvenientes de esta implementación de hilos en modo núcleo son:

- Las funciones usadas para la sincronización entre hilos son llamadas al sistema y, por tanto, más costosas. Una forma de aliviar este problema es implementar cuidadosamente las funciones de biblioteca para la sincronización, de tal forma que solo se realice una llamada al sistema cuando sea estrictamente necesario.

[16] Recordemos de Estructura y Tecnología de Computadores que un fallo de página se produce cuando un proceso intenta acceder a una de sus páginas y esta no se encuentra en memoria principal. Esto hace que el proceso no pueda continuar su ejecución hasta que el sistema operativo resuelva el fallo. Hablaremos más sobre esto en el tema de gestión de memoria.

- La creación y destrucción de hilos dentro de un proceso es también más costosa al tener que cambiar al sistema operativo para ello, el cual, además, debe crear y destruir las estructuras de datos asociadas a los hilos. Una posible solución para acelerar la creación y destrucción de hilos sería la reutilización de los mismos o, más bien, de sus estructuras de datos. De esta forma, cuando un hilo termina dentro de un proceso, y siempre que se pueda, no se liberarán completamente las estructuras de datos que el sistema operativo mantiene internamente para el hilo, sino que las mantendrá y reutilizará cuando se cree el siguiente hilo del proceso.

Aunque la planificación se realice basándose en hilos, hay varias acciones que afectan a todos los hilos de un proceso y que el sistema operativo debe manejar por proceso. Por ejemplo, la suspensión de un proceso implica intercambiar de memoria a disco el espacio de direcciones del proceso. Dado que todos los hilos de un proceso comparten el mismo espacio de direcciones, todos los hilos deben pasar al estado «suspendido» al mismo tiempo. De igual modo, la terminación de un proceso provoca la terminación de todos los hilos de ese proceso.

2.5 PLANIFICACIÓN DE PROCESOS

En un momento dado puede haber varios procesos listos para ejecutarse. 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*.

2.5.1 Metas de la planificación

El orden en el que un planificador selecciona los procesos listos para su ejecución viene determinado muchas veces por la meta o metas que se pretenden conseguir. Algunas de estas metas son:

1. *Equidad*: cada proceso obtiene su proporción «justa» de CPU, es decir, el tiempo de CPU asignado a un proceso es directa o inversamente proporcional a una cierta característica del mismo.

   No debemos confundir la equidad con la igualdad, donde todos los procesos recibirían el mismo tiempo de CPU, algo que, en la práctica, sería difícil de conseguir pues los procesos suelen tener comportamientos distintos (unos se bloquean continuamente, otros no, etc.).

2. *Eficiencia*: mantener la CPU ocupada al 100 % realizando trabajo útil, entendido este como el trabajo que realizan los procesos de usuario. Simplificando, podemos definir la eficiencia como

   $$E = \frac{Tiempo\_útil}{Tiempo\_total} \times 100 = \frac{Tiempo\_útil}{Tiempo\_útil + Tiempo\_gestión + Tiempo\_ociosa}.$$

   Como vemos, el tiempo total es la suma de, al menos, tres tiempos: el tiempo útil, el tiempo que la CPU ejecuta código para tareas de gestión del sistema (normalmente, código del sistema operativo) y el tiempo que la CPU queda ociosa por no haber procesos listos que se puedan ejecutar.

3. *Tiempo de espera*: minimizar el tiempo que pasa un proceso en la cola de procesos listos esperando a que se le conceda la CPU.

4. *Tiempo de respuesta*: este es el tiempo que transcurre desde que se solicita la ejecución de una acción (por ejemplo, pulsando la tecla <return> en un programa para que realice algo), hasta que se obtienen los primeros resultados de la acción solicitada. La meta en este caso sería minimizar este tiempo para usuarios interactivos.

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. La meta también sería minimizar este tiempo para los usuarios de trabajos por lotes.

   Recordemos que en el tema 1 hemos visto que en el procesamiento de trabajos por lotes no hay interacción con el usuario, ya que cada programa ejecutado obtiene los datos que necesita de alguna fuente (fichero, tarjeta perforada, etc.) y vuelca los resultados en algún destino (fichero, impresora, tarjeta perforada, etc.).

6. *Rendimiento* o *productividad*: maximizar el nº de tareas procesadas por unidad de tiempo (minutos, horas, etc.).

Podemos ver que hay metas contradictorias, como la 4 y la 5: para minimizar el tiempo de respuesta de los usuarios interactivos, el planificador no debería ejecutar las tareas de procesamiento por lotes salvo de madrugada, cuando los usuarios duermen. Esto se debe a que, si hay una única CPU, debo repartir la misma entre todos los procesos. Si concedo la CPU a los procesos interactivos para reducir su tiempo de respuesta, estaré retrasando los trabajos por lotes, aumentando así su tiempo de regreso.

En definitiva, se puede demostrar que cualquier algoritmo de planificación que favorece a un tipo de tareas perjudica a tareas de otros tipos, por lo que no hay algoritmos de planificación óptimos que permitan alcanzar todas las metas a la vez.

2.5.2 Planificación apropiativa y no apropiativa

Una complicación adicional para los planificadores es que los procesos son impredecibles: algunos esperan mucho en operaciones de E/S; otros utilizan mucha CPU, consumiendo poco tiempo en operaciones de E/S. Por tanto, cuando el planificador cede la CPU a un proceso, no sabe a ciencia cierta cuándo se bloqueará o cuándo terminará.

Principalmente, un planificador puede adoptar dos estrategias de funcionamiento. Una es poder expulsar a procesos ejecutables de la CPU para ejecutar otros procesos. Otra es permitir a un proceso ejecutarse hasta terminar o hasta bloquearse (por ejemplo, por E/S). En el primer caso hablamos de *planificación apropiativa* (porque el SO se puede apropiar de la CPU para dársela a otro) y en el segundo de *planificación no apropiativa* o *planificación de ejecución hasta terminar*.

El tipo de planificación a usar puede depender de los procesos existentes en el sistema. Así, para procesos por lotes, donde no hay una interacción con el usuario, se puede usar tanto una planificación no apropiativa como una apropiativa con largos periodos de CPU (para evitar continuos cambios de proceso, ya que estos cambios desperdician CPU). En cambio, para los procesos interactivos, donde puede haber varios usuarios interactuando con el sistema a la vez, es necesario usar una planificación apropiativa para cambiar la CPU rápidamente de un proceso a otro y así crear la ilusión de que todos los procesos se ejecutan a la vez.

2.5.3 Ciclo de ráfagas de CPU y E/S

El éxito de la planificación de la CPU depende de la siguiente propiedad de los procesos: la ejecución de un proceso consiste en un ciclo de ejecución en la CPU (*ráfaga*

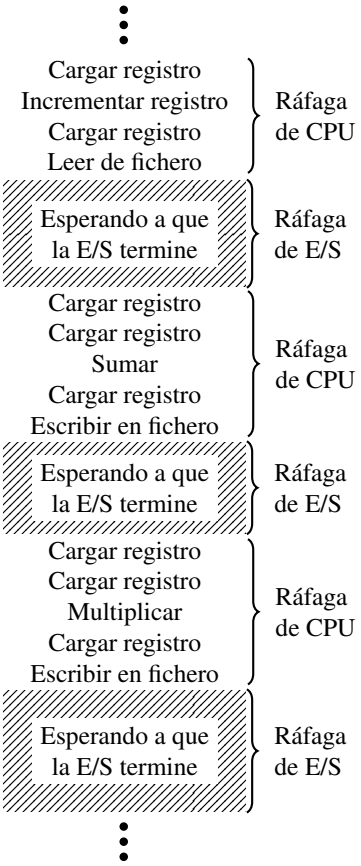


Figura 2.13. Alternancia entre ráfagas de CPU y E/S durante la ejecución de un proceso.

de CPU), un ciclo de espera de E/S (*ráfaga de E/S*), un ciclo de ejecución en la CPU, un ciclo de espera de E/S, y así sucesivamente. Como podemos ver en la figura 2.13, los procesos alternan entre ráfagas de CPU y ráfagas de E/S, *comenzando y terminando con una ráfaga de CPU*.

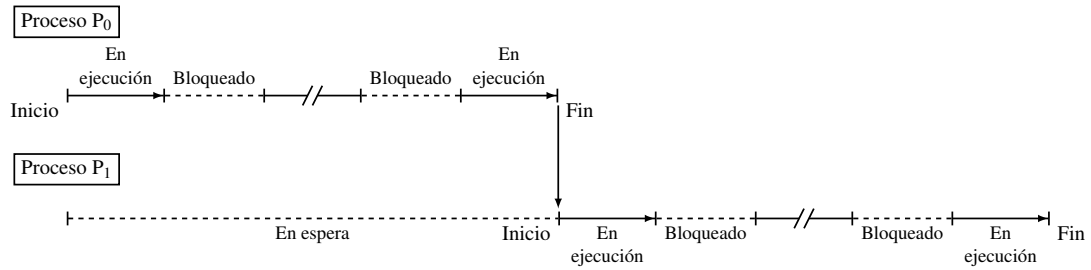
La duración de las ráfagas de un proceso no se conoce a priori, aunque, en algunos casos, se pueden hacer buenas estimaciones de las duraciones a partir de ejecuciones anteriores del proceso (como veremos después).

Esta alternancia entre ráfagas de CPU y de E/S de los procesos es la que hace posible la existencia de la multiprogramación (ver figura 2.14), ya que, mientras unos procesos están en ráfagas de E/S, otro puede estar en una ráfaga de CPU. Esto hace que la CPU no se desperdicie, consiguiendo también una reducción del tiempo total de ejecución de todos los procesos.

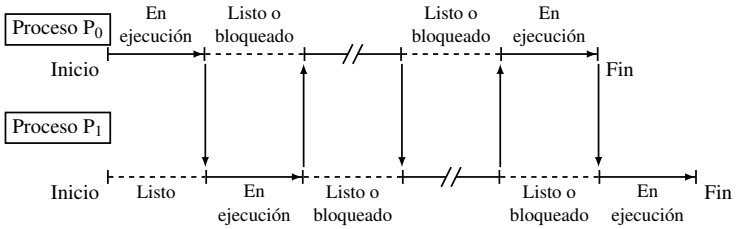
Un *proceso limitado por E/S* es aquel que pasa la mayor parte de su tiempo en espera de una operación de E/S. Normalmente, un proceso de este tipo tendrá muchas ráfagas de CPU aunque breves. Por ejemplo, un intérprete de órdenes o el proceso que comprueba la consistencia de un sistema de ficheros entrarían dentro de esta categoría.

Un *proceso limitado por la CPU* es aquel que necesita usar la CPU la mayor parte de su tiempo. Normalmente, un proceso así tendrá pocas ráfagas de CPU de muy larga duración. Por ejemplo, un proceso para resolver ecuaciones sería de este tipo.

La distribución entre procesos limitados por CPU y procesos limitados por E/S puede ser muy importante a la hora de seleccionar un algoritmo adecuado de planificación, ya que hay algoritmos que favorecen a un tipo de procesos y otros a otro tipo, como veremos a continuación.



(a) Sin multiprogramación



(b) Con multiprogramación

Figura 2.14. Ejecución de procesos con y sin multiprogramación.

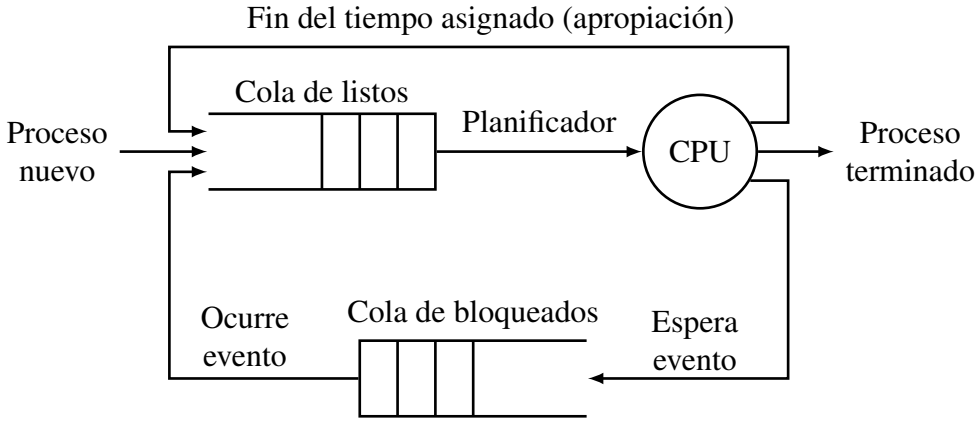


Figura 2.15. Representación de la planificación de procesos mediante un diagrama de colas.

2.5.4 Algoritmos de planificación

La planificación se puede analizar de forma gráfica representando el sistema mediante un diagrama de colas (ver figura 2.15). Cada rectángulo representa una cola, los círculos representan recursos que dan servicio a las colas y las flechas indican el flujo de los procesos en el sistema.

Nosotros nos vamos a centrar en la planificación de la CPU, pero es importante observar que la CPU no es el único recurso que hay que planificar. Muchos dispositivos de E/S pueden ser utilizados por varios procesos al mismo tiempo, por lo que habrá que planificar también las peticiones que llegan a cada uno.

Planificación «Primero en llegar, primero en ser servido» (FCFS)

Es el algoritmo más sencillo de todos. En este esquema, el proceso que primero solicita la CPU es el primero al que se le asigna. El algoritmo FCFS es no apropiativo, por lo que el proceso que toma la CPU no la libera hasta que termina o se bloquea. Este

Proceso	Duración de la ráfaga
P1	24
P2	3
P3	3

Tabla 2.1. Tres procesos y sus duraciones de ráfaga de CPU.

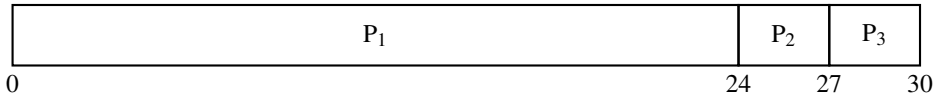


Figura 2.16. Planificación FCFS.

algoritmo es fácil de implementar con una cola FIFO.

En esta política, el tiempo promedio de respuesta puede ser bastante largo. Por ejemplo, supongamos que tenemos 3 procesos con duraciones de la siguiente ráfaga de CPU dadas en tabla 2.1. Si los procesos se ejecutan en el orden P1, P2, P3 (ver figura 2.16), el tiempo promedio de respuesta es (24 + 27 + 30) / 3 = 27. Sin embargo, si el orden es P2, P3, P1, el tiempo promedio de respuesta es (3 + 6 + 30) / 3 = 13, que representa una reducción considerable. El *diagrama de Gantt*[17] para este segundo orden de ejecución se puede ver en la figura 2.17, ya que coincide con el de una planificación SJF. Por tanto, el tiempo promedio de respuesta producido por FCFS no es mínimo y puede variar bastante si los tiempos de ráfagas de CPU de los procesos son muy distintos.

Un problema de la planificación FCFS es que puede producir el denominado *efecto convoy*, que surge de la siguiente manera: un proceso limitado por CPU ocupa la CPU durante un periodo largo de tiempo. Mientras tanto, los procesos limitados por E/S terminan sus peticiones de E/S, pasando entonces a la cola de listos a la espera de que la CPU quede libre. Mientras tanto, los dispositivos de E/S quedan inactivos. Cuando el proceso limitado por CPU pasa a realizar su E/S, los procesos limitados por E/S pueden utilizar la CPU, lo que harán rápidamente al tener ráfagas de CPU breves. Esto provocará ahora que la CPU quede inactiva. Es de esperar que el proceso limitado por CPU termine pronto de realizar su E/S, pues será corta en comparación con la de los procesos limitados por E/S, por lo que ocupará la CPU de nuevo durante un largo periodo, empezando otra vez lo que acabamos de describir.

El resultado final, como podemos ver, es que se desaprovechan los recursos (CPU, dispositivos de E/S, etc.). También podemos ver que el nombre *efecto convoy* viene del hecho de 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), que actúan de vagones.

Planificación «Primero el trabajo más corto» (SJF)

SJF es un algoritmo de planificación no apropiativo que es adecuado para procesos por lotes, donde los tiempos de ejecución se suelen conocer de antemano (o, al menos, una aproximación de los mismos) debido a ejecuciones previas de los mismos. Además, es óptimo para los tiempos de regreso de estos procesos, ya que proporciona el tiempo promedio de regreso mínimo.

[17] Un diagrama de Gantt es una representación gráfica que nos permite ver de forma rápida cuándo empieza y termina cada una de las tareas de un cierto sistema, mostrando así el tiempo dedicado a cada una. En el fondo, un diagrama de Gantt es un cronograma que muestra la evolución de las diferentes tareas.

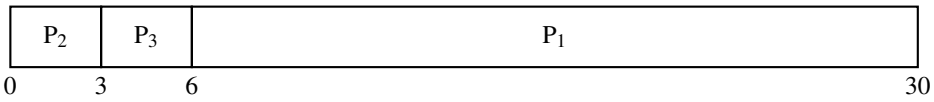


Figura 2.17. Planificación SJF.

El algoritmo consiste en coger, de entre todos los trabajos listos, aquel que menos tiempo tarda en ejecutarse en total (aunque también se podría coger aquel con la siguiente menor ráfaga de CPU). Podemos ver el resultado de esta planificación, en el caso de los 3 procesos anteriores, en la figura 2.17.

Aunque hemos dicho que este algoritmo es adecuado para los procesos por lotes, donde se conocen de antemano los tiempos totales de ejecución, se puede intentar aplicar a los procesos interactivos para obtener el tiempo promedio de respuesta mínimo. Un proceso interactivo consiste, normalmente, en:

$$\text{Esperar orden} \rightarrow \text{Ejecutar orden} \rightarrow \text{Esperar orden} \rightarrow \text{Ejecutar orden} \ldots$$

La ejecución de cada orden se puede considerar como un proceso. Como no se conoce el tiempo exacto de ejecución, debemos hacer estimaciones basándonos en el comportamiento pasado. Para hacer la estimación, podemos utilizar una expresión como:

$$E_t = a \cdot E_{t-1} + (1-a) \cdot T_{t-1}, \text{para } t = 2,3,4,\ldots \tag{2.1}$$

donde a es un parámetro ajustable entre 0 y 1 llamado *coeficiente de credibilidad*, E_t es la estimación a realizar, E_{t-1} es la estimación hecha anteriormente y T_{t-1} es el tiempo real de la ejecución anterior. Por ejemplo, si $a = \frac{1}{2}$ y suponemos que en el instante inicial $E_1 = T_0$ (cuidado, T_0 es una estimación; el primer tiempo real de ejecución es T_1), entonces la secuencia de estimaciones es:

Estimación	Tiempo real
$E_1 = T_0$	T_1
$E_2 = \frac{1}{2} \cdot E_1 + (1-\frac{1}{2}) \cdot T_1 = \frac{T_0}{2} + \frac{T_1}{2}$	T_2
$E_3 = \frac{1}{2} \cdot E_2 + (1-\frac{1}{2}) \cdot T_2 = \frac{T_0}{4} + \frac{T_1}{4} + \frac{T_2}{2}$	T_3
$E_4 = \frac{1}{2} \cdot E_3 + (1-\frac{1}{2}) \cdot T_3 = \frac{T_0}{8} + \frac{T_1}{8} + \frac{T_2}{4} + \frac{T_3}{2}$	T_4
...	...

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 (es decir, tiene más peso el último tiempo de ejecución real) y un valor de a grande hace que se recuerden durante largo tiempo.

Un aspecto importante es que, para que el algoritmo SJF sea óptimo (aun en el caso de procesos por lotes), es necesario disponer de todos los procesos de forma simultánea. Por ejemplo, supongamos que tenemos los siguientes cinco procesos:

Proceso	Tiempo de ejecución	Tiempo de llegada
A	2	0
B	4	0
C	1	3
D	1	3
E	1	3

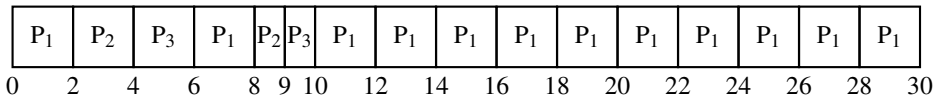


Figura 2.18. Planificación round robin.

El orden de ejecución correcto es A, B, C, D, E, si la planificación se realiza atendiendo únicamente a los procesos que ya han llegado. Según este orden, el tiempo promedio de retorno es (2+6+4+5+6)/5 = 4,6. Si sabemos cómo serán los procesos que van a llegar después, se puede hacer otra planificación, por ejemplo, B, C, D, E, A. En este caso, el tiempo promedio de retorno es (4+2+3+4+9)/5 = 4,4. Observa que el orden no puede ser C, D, E, A, B porque en el instante 0 no ha llegado ni el proceso C, ni D ni E, que llegan en el instante 3.

Planificación «Primero el que tenga el menor tiempo restante» (SRTF)

El algoritmo SJF es no apropiativo. Sin embargo, 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. A la planificación SJF apropiativa se le suele llamar planificación «primero el que tenga el menor tiempo restante» (*Shortest-Remaining-Time First*, SRTF).

Planificación round robin *(RR) o circular*

Esta planificación es de las más antiguas, sencillas, justas y de uso más amplio. En ella los procesos se atiende en el orden de llegada a la cola de procesos listos, como en FCFS, aunque se asigna a cada proceso un intervalo de tiempo de ejecución llamado *quantum*. Ahora se cambia de un proceso a otro cuando el proceso que tiene la CPU:

- Consume su quantum.
- Termina antes de consumir su quantum.
- Se queda bloqueado.

Si un proceso consume su quantum, entonces regresa al final de la cola de procesos listos, donde esperará a que le toque de nuevo su turno de uso de la CPU.

Como se ve, la planificación *round robin* es apropiativa, pues a un proceso se le quita la CPU cuando consume su quantum. Un ejemplo de esta planificación se puede ver en la figura 2.18 para los mismos tres procesos de la tabla 2.1, suponiendo un quantum de 2 unidades de tiempo.

Con esta planificación ningún proceso debe esperar más de $(n-1)\cdot q$ unidades de tiempo antes de recibir el siguiente quantum, suponiendo que hay n procesos listos y un quantum q. Por ello, es importante decidir la longitud del quantum:

- Si el quantum es pequeño, la CPU cambia mucho de proceso, por lo que la CPU se desperdicia en la tarea que supone el cambio de un proceso a otro (guardar y cargar registros y mapas de memoria, actualización de tablas y listas, ...). Incluso, si el quantum es excesivamente pequeño, la CPU puede pasar más tiempo realizando cambios de proceso que ejecutando procesos.

- Si el quantum es grande, la CPU se desperdicia poco al reducirse el número de cambios de proceso, pero los últimos procesos de la lista tardan mucho en ser atendidos. En procesos interactivos, esto puede suponer tiempos de respuesta muy pobres y, por tanto, una mala experiencia de usuario.

Planificación por prioridad

En la planificación *round robin* todos los procesos tienen la misma importancia, pero esa no es la situación habitual, pues suele haber procesos que, por sus características, necesitan ser ejecutados tan pronto sea posible una vez pasen al estado listo.

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 prioridad se suele asignar a un proceso en el momento de su creación, como hemos visto en la sección 2.3.1; normalmente, es un número donde un número alto representa una mayor prioridad, aunque hay sistemas, como Unix/Linux, donde a mayor número, menor prioridad[18].

La asignación de prioridades puede ser *estática* o *dinámica*, es decir, puede no cambiar nunca durante toda la vida de un proceso o puede ir cambiando según ciertos parámetros. Un ejemplo de asignación dinámica lo podemos encontrar cuando queremos favorecer a los procesos limitados por E/S, que pasan gran parte de su tiempo esperando a que la E/S concluya. Estos procesos deben tener gran prioridad para así ejecutarse pronto y enviar su siguiente solicitud de E/S, que puede realizarse en paralelo con el cómputo de otro proceso, lo que mejorará el uso de los recursos del sistema (CPU, dispositivos de E/S, etc.). Para favorecer a estos procesos, podemos hacer que la prioridad dinámica de un proceso sea $\frac{1}{F}$, donde F es la fracción del último quantum utilizado por el proceso[19]. Por ejemplo, si $q = 100$ ms, y un proceso utiliza 2 ms, su prioridad será $50 = \frac{1}{\frac{2}{100}}$. Si utilizara 50 ms, su prioridad sería $2 = \frac{1}{\frac{50}{100}}$.

Además de decidir si las prioridades son estáticas o dinámicas, 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 que se está ejecutando, 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* o *inanición*, 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, por ejemplo, cada interrupción de reloj. Esta opción tiene sentido si la planificación es apropiativa. Además, presenta el problema de que pueden estar llegando continuamente procesos de mayor prioridad que los procesos que llevan tiempo esperando, por lo que estos difícilmente usarán la CPU. Otra posibilidad, que soluciona este problema, es aumentar periódicamente la prioridad de los procesos listos de baja prioridad. Tarde o temprano, estos procesos serán los de mayor prioridad y, por tanto, llegarán a usar la CPU.

[18] En realidad, la planificación de procesos en Linux es diferente de la descrita aquí, ya que la prioridad de un proceso determina la proporción de CPU que le toca y no si se debe ejecutar antes o después que otros. Describiremos con algo más de detalle esta planificación de procesos en Linux en prácticas.

[19] El quantum en este caso es simplemente una cantidad arbitraria de tiempo que usamos para saber si un proceso usa mucha o poca CPU y así bajar o subir su prioridad, respectivamente. Por lo tanto, aunque un proceso consuma su quantum, no se le quita la CPU.

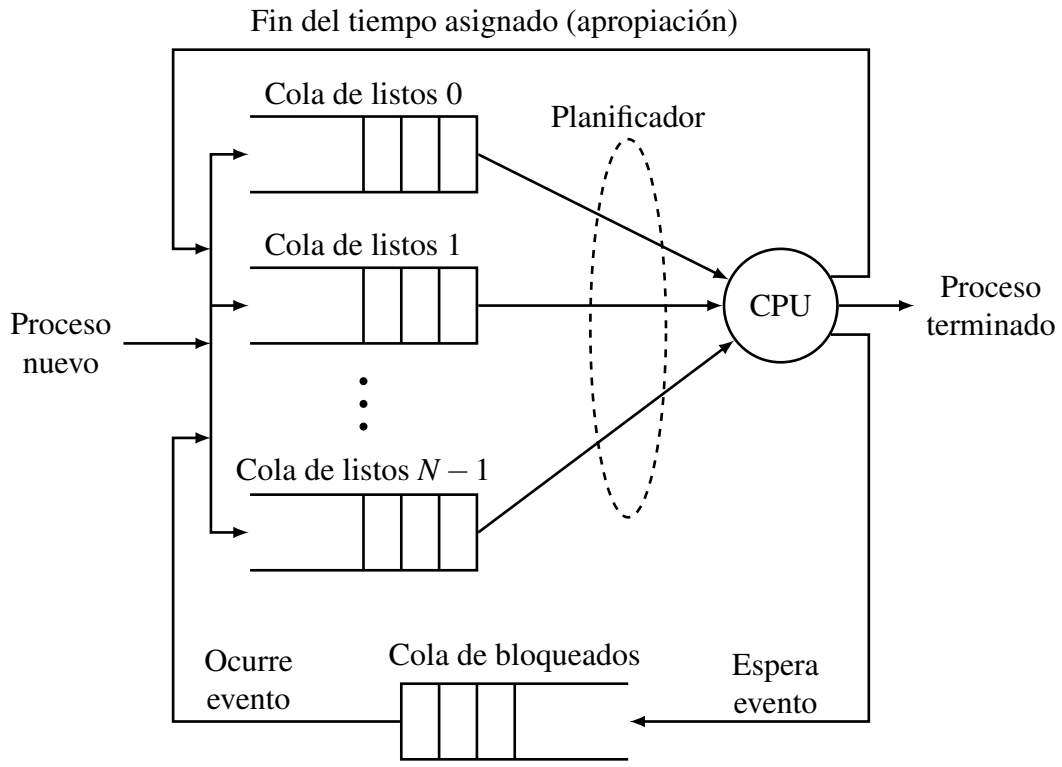


Figura 2.19. Planificación de múltiples colas.

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, como pueden ser: el tipo de proceso (interactivo o por lotes, que tienen requisitos de tiempo de respuesta/retorno bastante diferentes), la importancia del proceso (de sistema o de usuario), la cola que utilizó la última vez, el consumo de CPU del proceso, etc.

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 (que dependerá del tipo de procesos de la cola). Generalmente, entre las colas se suele establecer una planificación apropiativa por prioridad, pero podría ser de otro tipo, como repartir el tiempo de CPU entre las colas y que cada una lo administre a su manera. La figura 2.19 muestra la estructura general de este algoritmo.

Si no queremos que el enfoque anterior de múltiples colas sea inflexible, debemos permitir que los procesos puedan cambiar de una cola a otra, lo que da lugar a la *planificación de múltiples colas con realimentación*. En este caso, habrá que establecer también los criterios para cambiar a un proceso de una cola a otra.

Algunos parámetros de la planificación de múltiples colas con realimentación son los siguientes:

- Número de colas.
- Algoritmo de planificación entre colas.
- Algoritmo de planificación dentro de cada cola.
- Criterio de ascenso y descenso de procesos entre colas.

- Cola inicial de un proceso nuevo, etc.

2.5.5 Planificación a corto, medio y largo plazo

Hasta ahora hemos supuesto que todos los procesos ejecutables se encuentran en la memoria principal. Sin embargo, si no se dispone de suficiente memoria, será necesario que algunos de los procesos ejecutables se mantengan en disco. El problema ahora es cómo hacer la planificación de procesos, ya que el coste de darle la CPU a un proceso que está en disco es considerablemente mayor que el dársela a un proceso que se encuentra en memoria, puesto que habrá que buscar memoria libre, leer el proceso de disco, etc.

Una forma práctica de trabajar con el intercambio de procesos, es decir, cuando unos procesos se encuentran en disco y otros en memoria principal, 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 (lo que hemos visto hasta ahora en este tema), 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 los procesos que hayan permanecido en ella el tiempo suficiente (suspensión de procesos) y para cargar en memoria los procesos que hayan estado en disco demasiado tiempo (reanudación de procesos). Tras esto actúa el PCP hasta que se llame de nuevo al PMP (ver figura 2.20).

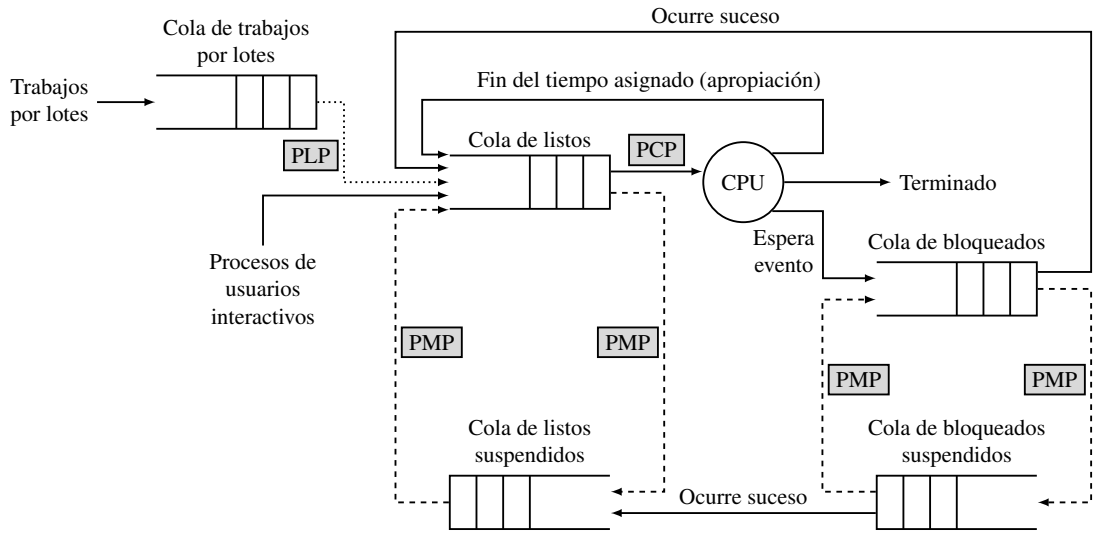


Figura 2.20. Diagrama de colas donde se observa un planificador a corto plazo (PCP), otro a medio plazo (PMP) y un planificador a largo plazo (PLP).

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

- ¿Cuánto tiempo ha transcurrido desde el último intercambio del proceso? Procesos que llevan mucho tiempo en memoria o en disco pueden ser buenos candidatos para ser intercambiados.
- ¿Cuánto tiempo de CPU ha utilizado recientemente el proceso? Si el uso total de la CPU es muy alto, 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.

- ¿Qué tan grande es el proceso? Los procesos pequeños, debido a que no necesitan grandes cantidades de memoria principal, no suelen causar problemas, por lo que podría no ser beneficioso el suspenderlos a disco, ya que liberarían cantidades pequeñas de memoria. Evidentemente, si un proceso pequeño consume mucha CPU, podría ser necesario el suspenderlo para reducir la carga del sistema.
- ¿Qué tan alta es la prioridad del proceso? Los procesos de mayor prioridad deberían permanecer en memoria, mientras que los procesos de baja prioridad podrían intercambiarse entre disco y memoria.

Como se puede ver en la figura 2.20, 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.