

## TEMA 2 – PROCESOS

### Bibliografía

Este documento se ha elaborado, principalmente, a partir de las referencias bibliográficas que se exponen a continuación, y con propósito meramente académico. El alumno debería completar el contenido de este tema con bibliografía existente en la biblioteca y mediante recursos de la Web, hasta que los conceptos que se indican en el mismo sean afianzados correctamente.

[ARANDA] Joaquin Aranda Alamansa, M<sup>a</sup> Antonia Canto Diaz, Jesús Manuel de la Cruz García, Sebastian Dormido Bencomo, Carolina Mañoso Hierro. Sistemas operativos, teoría y problemas. Editorial Sanz y Torres, S.L, 2002.

[STALLINGS] W. Stallings. Sistemas operativos. 5ª edición, Prentice Hall, Madrid, 2005.

[TANENBAUM] A. S. Tanenbaum. Sistemas operativos modernos, 3ª edición. Prentice Hall, Madrid, 2009.

[CARRETERO] F. Pérez, J. Carretero, F. García. Problemas de sistemas operativos: de la base al diseño, 2ª edición. McGraw-Hill. 2003.

### Contenido

- Procesos y bloque de control de procesos (Capítulo 2, sección 2.3 [Stallings], Capítulo 3, sección 3.1 [Stallings]).
- Creación de un proceso (Capítulo 3, sección 3.4 [Stallings], Capítulo 2, sección 2.1.1 [Tanenbaum]).
- Terminación de procesos (Capítulo 2, sección 2.1.3 [Tanenbaum]).
- Modo usuario y modo núcleo, modos de ejecución (Capítulo 3, sección 3.4 [Stallings]).
- Llamadas al sistema (Capítulo 1, sección 1.5.6 y 1.6 [Tanenbaum]).
- Estados de los procesos (Capítulo 3, sección 3.2 [Stallings]).
- Un modelo de procesos de dos estados (Capítulo 3, sección 3.2 [Stallings]).
- Modelo de proceso de cinco estados (Capítulo 3, sección 3.2 [Stallings]).
- Procesos suspendidos (Capítulo 3, sección 3.2 [Stallings]).
- Procesos y recursos ((Capítulo 3, sección 3.3 [Stallings]).
- Gestión de procesos en UNIX (Capítulo 3, sección 3.5 [Stallings]).
- Hilos (Capítulo 4, sección 4.1 [Stallings]).
- Multiprocesamiento simétrico – SMP (Capítulo 4, sección 4.2 [Stallings]).

## Procesos y bloque de control de procesos

Un proceso es fundamentalmente un programa en ejecución. Según el sistema de que se trate su estructura varía pero, en general se puede decir que consta de código ejecutable, datos y todo lo necesario para su identificación y ejecución dentro del sistema.

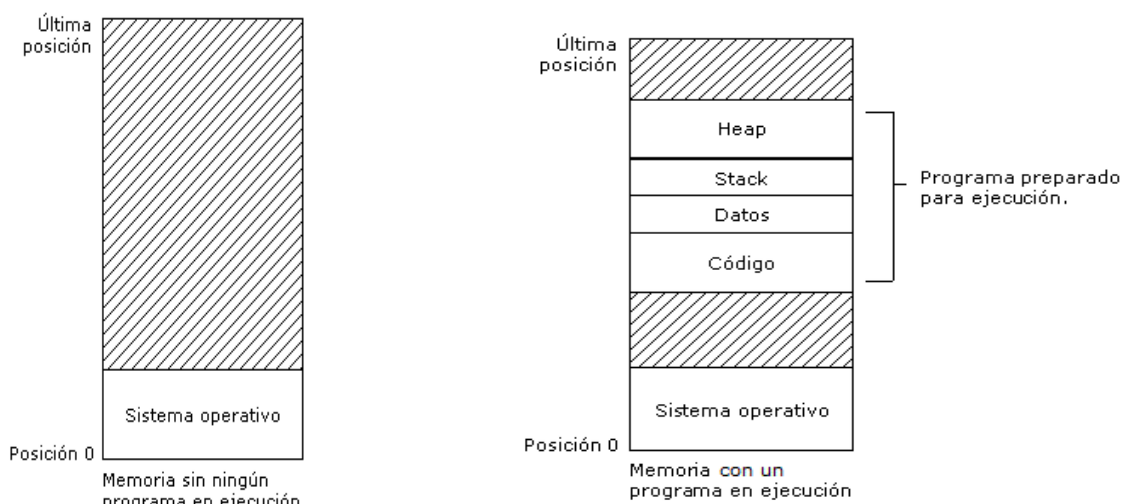
Se han dado muchas definiciones del término proceso, incluyendo:

- Un programa en ejecución.
- Una instancia de un programa ejecutándose en un computador.
- La entidad que se puede asignar o ejecutar en un procesador.
- Una unidad de actividad caracterizada por un solo hilo secuencial de ejecución (más adelante se estudiará lo que es un hilo), un estado actual, y un conjunto de recursos del sistema asociados.


Se puede considerar que un proceso está formado por los siguientes componentes:

- **Código** ejecutable (conjunto de instrucciones en lenguaje máquina).
- Los **datos** asociados que necesita el programa: Variables globales, constantes, variables inicializadas y no inicializadas, variables de solo lectura.
- El **contexto** de ejecución del programa, que se estudia a continuación.
- La **pila** o stack: Registra por bloques llamadas a procedimientos y los parámetros pasados a estos, variables locales de la rutina invocada, y la dirección de la siguiente instrucción a ejecutar cuando termine la llamada. Esta zona de memoria se asigna por el sistema operativo al cargar un proceso en memoria principal. En caso de auto llamadas recursivas podría desbordarse.
- El **montículo** o heap: Zona de memoria asignada por el sistema operativo para datos en tiempo de ejecución, en UNIX se usa para la familia de llamadas *malloc()*. Puede aumentar y disminuir en tiempo de ejecución de un proceso.

A continuación se muestra en una figura la estructura de un proceso en memoria principal. Algunos autores como William Stallings añaden el contexto de ejecución en dicha figura, pero depende del sistema. En muchos casos el contexto de ejecución se almacena en otras estructuras almacenadas en memoria principal, como puede ser la tabla de procesos.



Supongamos que el procesador comienza a ejecutar un código de programa (conjunto de instrucciones), y que nos referiremos a esta entidad en ejecución como un proceso. En cualquier instante puntual del tiempo, mientras el proceso está en ejecución, éste se puede caracterizar por una serie de elementos, cuyo conjunto se llama **contexto de ejecución** (conjunto de datos interno por el cual el sistema operativo es capaz de supervisar y controlar el proceso):

- **Identificación.** Un identificador único asociado al proceso para distinguirlo del resto de procesos. Puede también almacenar el identificador del proceso padre creador de este proceso y el identificador del usuario del proceso, dependiendo del sistema.
  - **Estado.** Si el proceso está actualmente corriendo, está en el estado en ejecución. Hay más tipos de estado, dependiendo del sistema operativo, como por ejemplo Listo, Suspendido, Parado o Zombie. Se estudiará en las siguientes secciones.
  - **Información de planificación:** Nivel de prioridad relativo al resto de procesos. Evento o eventos por los que espera el proceso cuando está bloqueado.
  - **Descripción de los segmentos de memoria asignados al proceso.** Espacio de direcciones o límites de memoria asignado al proceso.
  - **Punteros a memoria.** Incluye los punteros al código de programa y los datos asociados a dicho proceso, además de cualquier bloque de memoria compartido con otros procesos, e incluso si el proceso utiliza memoria virtual. Se almacenan también punteros a la pila y al montículo del proceso.
  - **Datos de contexto.** Estos son datos que están presentes en los registros del procesador cuando el proceso está corriendo. Almacena el valor de todos los registros del procesador, banderas de estado, señales, etc., es decir, todo lo necesario para poder continuar la ejecución del proceso cuando el sistema operativo lo decida.
  - **Información de estado de E/S y recursos asignados:** Incluye las peticiones de E/S pendientes, dispositivos de E/S (por ejemplo, un disco duro) asignados a dicho proceso, una lista de los ficheros en uso por el mismo, etc.
-  • **Comunicación entre procesos.** Puede haber varios indicadores, señales y mensajes asociados con la comunicación entre dos procesos independientes.
- **Información de auditoría.** Puede incluir la cantidad de tiempo de procesador y de tiempo de reloj utilizados, así como los límites de tiempo, registros contables, etc.

La información de la lista anterior se almacena de manera diferente dependiendo del sistema operativo, pero de forma general se hace en una estructura de datos que se llama **Bloque de Control de Proceso** o **BCP** (*process control block* - PCB en Inglés) (Figura 3.1). El punto más significativo en relación al bloque de control de proceso, o BCP, es que contiene suficiente información de forma que es posible interrumpir el proceso cuando está corriendo (en una determinada instrucción de su código) y posteriormente restaurar su estado de ejecución como si no hubiera habido interrupción alguna. El BCP es la herramienta clave que permite al sistema operativo dar soporte a múltiples procesos y proporcionar multiprogramación. En algunos sistemas como UNIX cada proceso tiene asociado un BCP que está almacenado en una lista simplemente enlazada llamada **tabla de procesos**. En otros sistemas el BCP se encuentra almacenado en el propio proceso, junto a los datos, código, pila y montículo. Se llama **imagen del proceso** al conjunto del programa, datos, pila, montículo y BCP (Tabla 3.4). Para ejecutar un proceso, la imagen del proceso completa se debe cargar en memoria principal o al menos en memoria virtual.

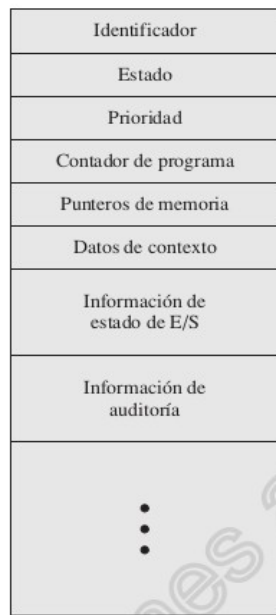


Figura 3.1. Bloque de control de programa (BCP) simplificado.

Ampliando con lo anteriormente comentado, para un computador monoprocesador, en un instante determinado, como máximo un único proceso puede estar corriendo y dicho proceso estará en el estado en ejecución. La conmutación entre unos procesos y otros es lo que da lugar a la multiprogramación, y el programa que se encarga de ello según la política de planificación se llama activador (**dispatcher**). El activador (programa que forma parte del núcleo que está cargado en memoria principal) pasa a ejecución cuando se producen alarmas de temporización (tiempo de CPU asignado a un proceso), cuando hay interrupciones y operaciones de E/S y cuando se ejecuta un proceso por tener una mayor prioridad que otros actualmente ejecutándose o en cola. Dicho esto, el *dispatcher* puede que no se ejecute siempre cuando se termina de ejecutar una rutina ISR y se quiere recuperar un contexto de un determinado proceso, sino que puede dispararse simplemente si el tiempo asignado por el procesador a un proceso termina (ese tiempo también se denomina “rodaja de tiempo”).

## Creación de procesos

Hay cuatro eventos principales que provocan la creación de procesos:

1. El arranque del sistema.
2. La ejecución, desde un proceso, de una llamada al sistema para creación de procesos.
3. Una petición de usuario para crear un proceso.
4. El inicio de un trabajo por lotes (colección o batería de trabajos en cola).

1) Generalmente, cuando se arranca un sistema operativo se crean varios procesos. Algunos de ellos son procesos en primer plano; es decir, procesos que interactúan con los usuarios y realizan trabajo para ellos. Otros son procesos en segundo plano, que no están asociados con usuarios específicos sino con una función específica. Por ejemplo, se puede diseñar un proceso en segundo plano para aceptar el correo electrónico entrante, que permanece inactivo la mayor parte del día pero que se activa cuando llega un mensaje. Se puede diseñar otro proceso en segundo plano para aceptar peticiones entrantes para las páginas Web hospedadas en ese equipo, que despierte cuando llegue una petición para darle servicio. Los procesos que permanecen en segundo plano para manejar

ciertas actividades como correo electrónico, páginas Web, noticias, impresiones, etcétera, se conocen como demonios (*daemons*). Los sistemas grandes tienen comúnmente docenas de ellos. En UNIX podemos utilizar varios comandos para visualizar los procesos en ejecución e incluso los demonios que se están ejecutando en segundo plano (consultar *man*):

```
%> top
```

```
%>service --status-all
```

```
%>ps auxf
```

```
%>ps -ef
```

```
%>ls -l /etc/init.d/
```

2) Además de los procesos que se crean al arranque, posteriormente se pueden crear otros. A menudo, un proceso en ejecución emitirá llamadas al sistema para crear uno o más procesos nuevos, para que le ayuden a realizar su trabajo (*fork()* e *hilos* de UNIX, lo estudiará en las prácticas de la asignatura). En especial, es útil crear procesos cuando el trabajo a realizar se puede formular fácilmente en términos de varios procesos interactivos relacionados entre sí, pero independientes en los demás aspectos. Por ejemplo, si se va a obtener una gran cantidad de datos a través de una red para su posterior procesamiento, puede ser conveniente crear un proceso para obtener los datos y colocarlos en un búffer compartido, mientras un segundo proceso remueve los elementos de datos y los procesa. En un multiprocesador, al permitir que cada proceso se ejecute en una CPU distinta también se puede hacer que el trabajo se realice con mayor rapidez.

3) En los sistemas interactivos, los usuarios pueden iniciar un programa escribiendo un comando o haciendo (doble) clic en un icono. Cualquiera de las dos acciones inicia un proceso y ejecuta el programa seleccionado. Un terminal de UNIX no es más que un proceso esperando un comando, el cual internamente lo ejecutará mediante una llamada a *fork()*.

4) La última situación en la que se crean los procesos se aplica sólo a los sistemas de procesamiento por lotes que se encuentran en las mainframes grandes. Aquí los usuarios pueden enviar trabajos de procesamiento por lotes al sistema (posiblemente en forma remota). Cuando el sistema operativo decide que tiene los recursos para ejecutar un trabajo de la cola, crea un proceso y ejecuta el siguiente trabajo de la cola de entrada.

Técnicamente, en todos estos casos, para crear un proceso hay que hacer que un proceso existente ejecute una llamada al sistema de creación de proceso. Ese proceso puede ser un proceso de usuario en ejecución, un proceso del sistema invocado mediante el teclado o ratón, o un proceso del administrador de procesamiento por lotes. Lo que hace en todo caso es ejecutar una llamada al sistema para crear el proceso. Esta llamada al sistema indica al sistema operativo que cree un proceso y le indica, directa o indirectamente, cuál programa debe ejecutarlo. En UNIX sólo hay una llamada al sistema para crear un proceso: *fork()*. Esta llamada crea un clon exacto del proceso que hizo la llamada. Después de *fork*, los dos procesos (padre e hijo) tienen la misma imagen de memoria, las mismas cadenas de entorno y los mismos archivos abiertos. Eso es todo. Por lo general, el proceso hijo ejecuta después a *execve* o una llamada al sistema similar para cambiar su imagen de memoria y ejecutar un nuevo programa.

Una vez vistos los principales eventos de creación de procesos y las estructuras de datos asociadas a un proceso, ahora se está en posición de describir brevemente los pasos que llevan a la verdadera creación de un proceso. Una vez que el sistema operativo decide, por cualquier motivo, crear un proceso procederá de la siguiente manera:

1. **Asignar un identificador de proceso único al proceso.** En este instante, se añade una nueva entrada a la tabla primaria de procesos, que contiene una entrada por proceso.
2. **Reservar espacio para proceso.** Esto incluye todos los elementos de la imagen del proceso. Para ello, el sistema operativo debe conocer cuánta memoria se requiere para el espacio de

direcciones privado (programas y datos) y para la pila (y montículo) de usuario. Estos valores se pueden asignar por defecto basándose en el tipo de proceso. Si un proceso es creado por otro proceso, el proceso padre puede pasar los parámetros requeridos por el sistema operativo como parte de la solicitud de la creación de proceso. En el caso de que exista una parte del espacio direcciones compartido con otro proceso, se fijan los enlaces apropiados. ~~Por último, se debe reservar el espacio para el bloque de control de proceso (BCP), recuerde que este puede estar en la tabla de procesos o en el mismo espacio de direcciones del proceso, depende del sistema.~~

3. **Inicialización del bloque de control de proceso (BCP).** La parte de identificación de procesos del BCP contiene el identificador del proceso así como otros posibles identificadores, tal como el indicador del proceso padre. En la información de estado de proceso del BCP, habitualmente se inicializa con la mayoría de entradas a 0, excepto el contador de programa (fijado en el punto entrada del programa) y los punteros de pila de sistema (fijados para definir los límites de la pila del proceso). La parte de información de control de procesos se inicializa en base a los valores por omisión, considerando también los atributos que han sido solicitados para este proceso. Por ejemplo, el estado del proceso se puede inicializar normalmente a Listo o Listo/Suspendido. La prioridad se puede fijar, por defecto, a la prioridad más baja, a menos que una solicitud explícita la eleve a una prioridad mayor. Inicialmente, el proceso no debe poseer ningún recurso (dispositivos de E/S, ficheros) a menos que exista una indicación explícita de ello o que haya sido heredados del padre.
4. **Establecer los enlaces apropiados.** Por ejemplo, si el sistema operativo mantiene cada cola del planificador como una lista enlazada, el nuevo proceso debe situarse en la cola de Listos o en la cola de Listos/Suspendidos.
5. **Creación o expansión de otras estructuras de datos.** Por ejemplo, el sistema operativo puede mantener un registro de auditoría por cada proceso que se puede utilizar posteriormente a efectos de análisis de rendimiento del sistema.

## Terminación de procesos

Una vez que se crea un proceso, empieza a ejecutarse y realiza el trabajo al que está destinado. Sin embargo, nada dura para siempre, ni siquiera los procesos. Tarde o temprano el nuevo proceso terminará, por lo general debido a una de las siguientes condiciones:

1. Salida normal (voluntaria).
2. Salida por error (voluntaria).
3. Error fatal (involuntaria).
4. Eliminado por otro proceso (involuntaria).

1) La mayoría de los procesos terminan debido a que han concluido su trabajo. Por ejemplo, cuando un compilador compila un programa añade implícitamente una llamada al sistema para indicar al sistema operativo su terminación normal. Esta llamada es *exit()* en UNIX. En sistemas con entorno gráfico, los procesadores de palabras, navegadores de Internet y programas similares siempre tienen un icono o elemento de menú en el que el usuario puede hacer clic para indicar al proceso que elimine todos los archivos temporales que tenga abiertos y después termine. Esa acción puede llevar una llamada a *exit()*.

2) La segunda razón de terminación es que el proceso descubra un error. Por ejemplo, si un usuario escribe el comando :

```
cc foo.c
```

para compilar el programa “foo.c” y no existe dicho archivo, el compilador simplemente termina.



Note que este error es predecible y tenido en cuenta por el proceso, ya que es el programador el que en su código contempló la opción de terminar el programa en el caso de que no se encontrase el fichero.

3) La tercera razón de terminación es un error fatal producido por el proceso, a menudo debido a un error en el programa. Algunos ejemplos incluyen el ejecutar una instrucción ilegal, hacer referencia a una parte de memoria no existente o la división entre cero. En algunos sistemas (como UNIX), un proceso puede indicar al sistema operativo que desea manejar ciertos errores por sí mismo, en cuyo caso el proceso recibe una señal que puede tratar en vez de terminar el proceso (tratamiento de errores).

4) La cuarta razón por la que un proceso podría terminar es que algún otro proceso ejecute una llamada al sistema que indique que elimine otro proceso o procesos. En UNIX esta llamada es *kill()*.

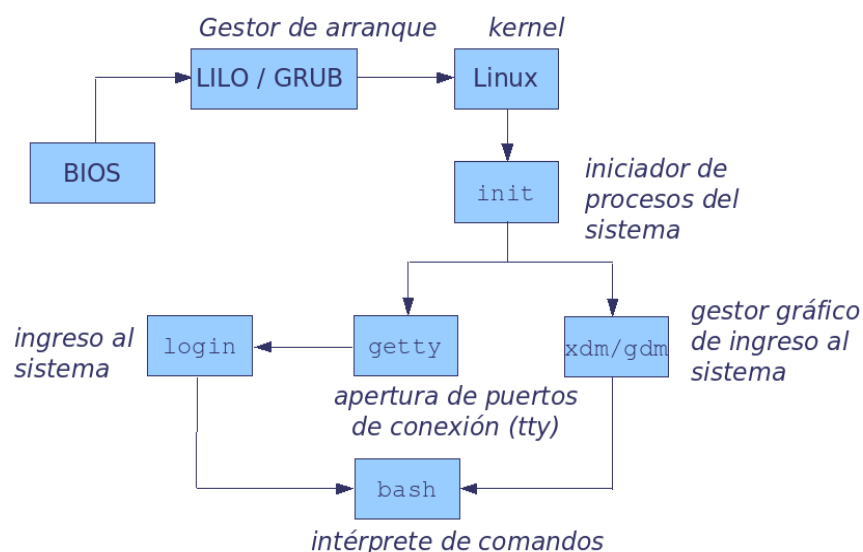
## Jerarquía de procesos en UNIX

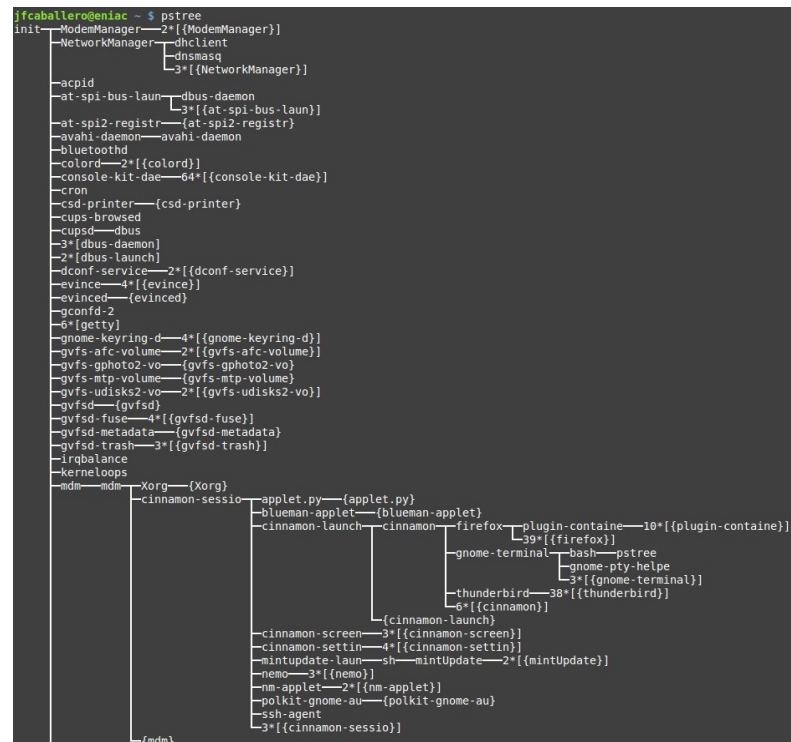
Un proceso en UNIX es un programa que se está ejecutando en un determinado momento en el sistema. Hay dos tipos de procesos: los de usuario y los demonios.

Procesos de usuario: Son aquellos procesos que el usuario utiliza, programas de aplicación o llamadas al sistema mediante *shell*.

Demonios: En el sistema operativo UNIX siempre hay una serie de procesos que se están ejecutando en segundo plano, sin que el usuario se dé cuenta de ello, y hacen que el sistema sea utilizable. No requieren de la intervención del usuario ni interactúan con él, y en general, se usan para programas que funcionan constantemente como por ejemplo servidores de red, mensajería instantánea y correo, sistema de impresión, compartición de ficheros con otros sistemas, servidor ftp, gestor de entorno gráfico, inicialización y control de hardware como tarjeta gráfica, tarjeta de sonido, usb, ventilador y sensores de la CPU, interfaz de red, ratón, gestión de energía, etc.

Cuando el kernel o núcleo del sistema se inicia, es decir, se carga en memoria, y inicialice todos los controladores de dispositivos y establezca las estructuras de datos necesarias, finaliza sus tareas dentro del proceso de arranque del sistema al momento de iniciar la ejecución de *init*. *Init* es el proceso responsable de la inicialización de nuevos procesos a nivel de usuario; o dicho de otra forma, todos los procesos de usuario en el sistema, excepto el proceso *swapper* (las imágenes de los procesos que no han de mantenerse en memoria física – memoria virtual –) descienden del proceso *init*. *Init* se ejecuta como demonio y por lo general tiene PID 1 (identificador de proceso). Con el comando o llamada al sistema “*ps tree*” podemos ver en forma de árbol los procesos que hay ejecutándose en el sistema.





En el momento en el que *init* comienza su ejecución, finaliza el proceso de arranque del sistema, realizando una serie de tareas administrativas. La lista exacta de tareas que *init* realiza puede variar en distintos sistemas GNU/Linux. Normalmente *init* realiza el chequeo de los sistemas de archivos, borra el contenido de `/tmp` e inicia la ejecución de varios servicios. *init* también se encarga de adoptar procesos huérfanos: cuando un proceso inicia un proceso hijo y éste muere antes que el padre, el proceso hijo pasa a ser un hijo de *init* inmediatamente. Al cerrar el sistema, es *init* quien se encarga de matar todos los procesos restantes, desmontar todos los sistemas de archivos y de cualquier otra cosa que haya sido configurado para hacer.

## Modo usuario y modo núcleo, modos de ejecución.

Necesitamos distinguir entre los modos de ejecución del procesador, normalmente asociados con el sistema operativo (modo núcleo, *kernel* o supervisor) y con los programas de usuario (modo usuario).

Si a un usuario, a nivel de aplicación, no le gusta un determinado compilador o no le gusta cómo funciona un programa que ha desarrollado, es libre de hacer cambios en él a su gusto, a través de un lenguaje de programación y sus librerías. El código del propio sistema operativo, su conjunto de instrucciones para gestionar recursos y proporcionar servicios, se encuentra codificado en lenguaje ensamblador (módulos que se pueden cargar dinámicamente convirtiéndose a código máquina) y en lenguaje máquina. Imaginemos que el usuario pudiera hacer cambios en tiempo real de las instrucciones que rigen el funcionamiento del sistema operativo y que interactúan con el hardware, que pudiera cambiar el manejador de interrupciones de reloj, la forma en que produce comunicación con un dispositivo de entrada-salida, es decir, que pudiera cambiar la codificación del sistema operativo o acceder a ella para utilizarla a su manera. Si esto sucediera, se podría hacer un uso indebido del sistema (en última instancia del hardware) produciendo comportamientos inesperados y pudiendo corromper el funcionamiento y gestión de los recursos. De ahí que determinadas instrucciones del sistema operativo solo se pueden ejecutar en modos privilegiados. Éstas incluirían lectura y modificación de los registros de control, instrucciones primitivas de E/S e instrucciones relacionadas con la gestión de memoria, instrucciones sobre señales, planificación, interrupciones,



controladores de dispositivos, etc.

El modo menos privilegiado a menudo se denomina modo usuario, porque los programas de usuario típicamente se ejecutan en este modo. El modo más privilegiado se denomina modo sistema, modo control, modo núcleo o kernel o modo supervisor. Este último término se refiere al núcleo del sistema operativo, que es la parte del sistema operativo que engloba las funciones y servicios más importantes del mismo. Es una parte relativamente pequeña del sistema, pero la más utilizada, por ello suele residir en memoria de forma continua, mientras que el resto del sistema operativo se carga cuando es necesario. La Tabla 3.7 lista las funciones básicas que normalmente se encuentran dentro del núcleo del sistema operativo.

**Tabla 3.7.** Funciones típicas de un núcleo de sistema operativo.

<b>Gestión de procesos</b>
<ul style="list-style-type: none"><li>• Creación y terminación de procesos</li><li>• Planificación y activación de procesos</li><li>• Intercambio de procesos</li><li>• Sincronización de procesos y soporte para comunicación entre procesos</li><li>• Gestión de los bloques de control de proceso</li></ul>
<b>Gestión memoria</b>
<ul style="list-style-type: none"><li>• Reserva de espacio direcciones para los procesos</li><li>• <i>Swapping</i></li><li>• Gestión de páginas y segmentos</li></ul>
<b>Gestión E/S</b>
<ul style="list-style-type: none"><li>• Gestión de <i>buffers</i></li><li>• Reserva de canales de E/S y de dispositivos para los procesos</li></ul>
<b>Funciones de soporte</b>
<ul style="list-style-type: none"><li>• Gestión de interrupciones</li><li>• Auditoría</li><li>• Monitorización</li></ul>

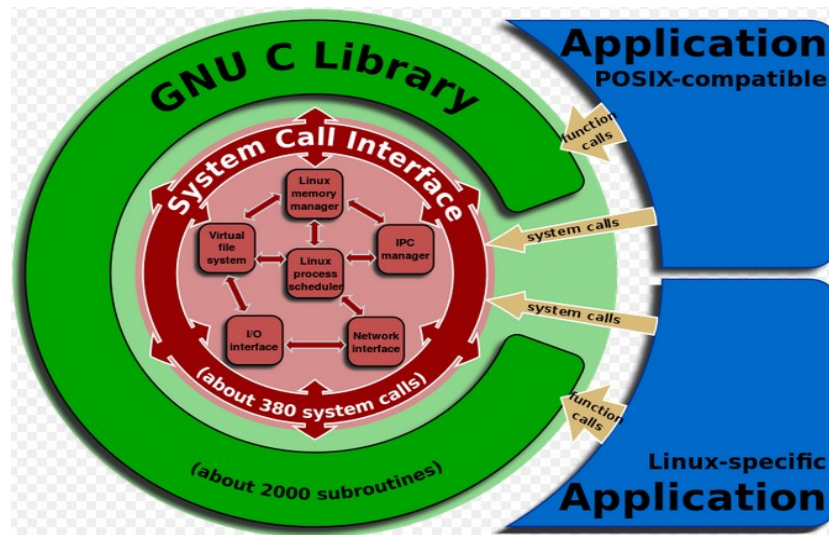
Como se ha comentado, el motivo por el cual se usan los otros modos es claro. Se necesita proteger al sistema operativo y a las tablas clave del sistema, por ejemplo los bloques de control de proceso, de la interferencia con programas de usuario. En modo núcleo, el software tiene control completo del procesador y de sus instrucciones, registros, y memoria.

Dicho esto, aparecen dos cuestiones: ¿cómo conoce el procesador en que modo está ejecutando y cómo este modo puede modificarse? En lo referente la primera cuestión, existe típicamente un bit en la palabra de estado de programa (PSW) que indica el modo de ejecución. Este bit se cambia como respuesta a determinados eventos. Con respecto a la segunda cuestión, cuando un usuario realiza una llamada a un servicio del sistema operativo o cuando una interrupción dispara la ejecución de una rutina del sistema operativo, este modo de ejecución se cambia a modo núcleo y; tras la finalización del servicio, el modo se fija de nuevo a modo usuario .

## Llamadas al sistema

Hemos visto que los sistemas operativos tienen dos funciones principales: proveer abstracciones a los programas de usuario mediante la máquina virtual multinivel (interfaz) y administrar los recursos de la computadora. En su mayor parte, la interacción entre los programas de usuario y el sistema operativo (por ejemplo, crear, escribir, leer y eliminar archivos) se relaciona con la primera función a través de lo que se llaman llamadas al sistema. Las llamadas al sistema disponibles entre

los programas de usuario y el sistema operativo varían de un sistema operativo a otro (UNIX tiene centenares de llamadas para obtener servicios y que el programador puede utilizar por medio de sus programas). En el caso de UNIX, programado en lenguaje C, encontraremos muchas de esas llamadas al sistema en la librería estándar de C, *glibc*. Esta librería nos proporciona llamadas que directamente podemos usar desde un programa escrito en C o incluso desde un terminal o *Shell*. Pero aún tenemos más..., muchas de las llamadas del sistema son complejas de utilizar incluso por un programador, por lo que *glibc* también hace abstracciones a nivel superior de las propias llamadas al sistema o incluso de conjuntos de ellas para que podamos utilizarlas cuando programamos, y así realizar una determinada acción de manera relativamente sencilla. El sistema operativo, por tanto, es el que lleva a cabo en última instancia las llamadas al sistema.



Con el riesgo de confundir un poco las cosas, describiremos brevemente el intérprete de comandos de UNIX, conocido como *Shell*. Aunque no forma parte del sistema operativo (es un programa más que ejecuta el sistema), utiliza con frecuencia muchas características del mismo y, por ende, sirve como un buen ejemplo de la forma en que se pueden utilizar las llamadas al sistema. También es la interfaz principal entre un usuario sentado en su terminal y el sistema operativo, a menos que el usuario esté usando una interfaz gráfica de usuario (que no es más que otro programa en ejecución en espera de eventos nuestros). Cuando cualquier usuario inicia un terminal se inicia un *shell*. El *shell* tiene la terminal como entrada y salida estándar (teclado-pantalla), y empieza por escribir el indicador de comandos (*prompt*) y un carácter tal como un signo de dólar, que indica al usuario que el *shell* está esperando aceptar un comando (argumento de entrada que espera el programa en ejecución *shell*). Por ejemplo, si el usuario escribe

```
jfcaballero@eniac ~ $:date
```

el *shell* crea un proceso hijo y ejecuta el programa “date” como el hijo (se estudiará como crear procesos hijos en las prácticas de la asignatura). Mientras se ejecuta el proceso hijo, el *shell* espera a que termine. Cuando el hijo termina, el *shell* escribe de nuevo el indicador y trata de leer la siguiente línea de entrada o comando.

Resumiendo, el programador o usuario puede comunicarse con el sistema operativo para requerir determinados servicios a través del código fuente de los programas que desarrolla o incluso a través del *shell*, teniendo en cuenta que tanto en el *shell* como en las librerías de funciones que utilizamos para programar puede haber abstracciones más altas de las llamadas que posee el sistema. Esas abstracciones irán bajando “en capas” hasta llegar a nivel ensamblador y código máquina, entrando el sistema en modo núcleo y ejecutando aquellas rutinas codificadas en instrucciones máquina necesarias para que se lleva a cabo la acción que ha pedido el usuario (en modo usuario) a través de

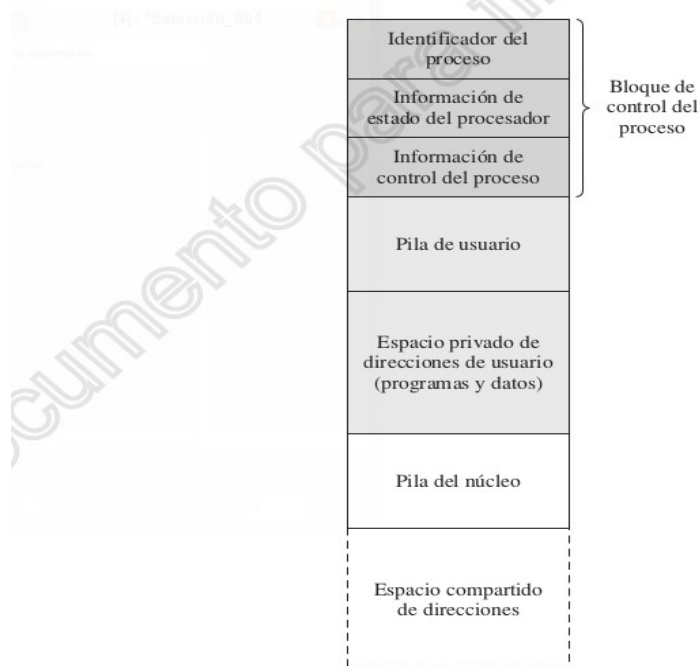
una llamada al sistema.

Las llamadas al sistema son necesarias porque el usuario no puede acceder o no tiene privilegios directos sobre los recursos que gestiona el sistema operativo y se utilizan para solicitar al núcleo del sistema el uso de los mismos (procesador, memoria, ficheros, dispositivo de E/S). Lo que ocurre cuando se invoca a una llamada del sistema desde **un proceso en ejecución** o **desde una rutina de interrupción de E/S** (imagine que ésta última necesita de una o varias llamadas al sistema) es lo siguiente:

1. El programador o usuario realiza la llamada al sistema de forma normal. La llamada junto con los parámetros asociados a la misma se cargarían en la pila (recuérdese que la llamada puede ser similar o igual a la real o puede estar enmascarada a un nivel de abstracción superior).
2. La llamada a biblioteca o llamada al sistema invocada ejecuta una instrucción de interrupción especial del sistema operativo que se llama *trap*, implementada a nivel de lenguaje ensamblador dentro de la propia llamada, y que conlleva un cambio de modo. Al ejecutarse la instrucción *trap* se carga un programa que pasa el sistema operativo de modo usuario a modo núcleo y se busca el lugar del núcleo donde está cargada la rutina a ejecutar (normalmente será un conjunto de instrucciones máquinas cargadas en memoria principal que el procesador tiene que ejecutar y que al usuario se le prohíbe utilizar directamente por seguridad). La ejecución de la instrucción *trap* lleva consigo que sea necesario salvar determinada información de contexto del proceso que realizó la llamada al sistema, pero esto no conlleva un salvado y cambio de contexto completo como ocurre con las interrupciones. Un cambio de modo puede ocurrir sin que se cambie el estado del proceso actualmente en estado “Ejecutando”. En dicho caso, la salvaguarda del estado y su posterior restauración comportan sólo una ligera sobrecarga. Sin embargo, si el proceso actualmente en estado “Ejecutando”, se va a mover a cualquier otro estado (Listo, Bloqueado, etc.), entonces el sistema operativo debe realizar cambios sustanciales en su entorno y se lleva a cabo un cambio de contexto completo, al igual que ocurre con las interrupciones. Considere el ejemplo de que el programa de usuario realiza una llamada al sistema que conlleva una operación de escritura o lectura de E/S (dirección CPU a dispositivo de E/S), en ese caso, se salva el contexto actual y se daría paso, tal y como se explicó en la sección de “E/S mediante interrupciones”, a que el módulo de E/S se encargue de la operación mientras que el procesador realiza otra tarea. Posteriormente, cuando se haya llevado a cabo la operación, se pondrá una señal de interrupción en el procesador (dirección dispositivo de E/S a CPU), se guardará el contexto del proceso que esté en ejecución y se tratará la interrupción.
3. Sigamos en el caso de que no sea necesario cambiar el estado del proceso actual. El programa cargado a partir de la instrucción *trap*, tras identificar que existe una rutina para la llamada realizada y que los parámetros son correctos, procede a ejecutarla.
4. Posteriormente el programa *trap* solicita un código de estado almacenado en un registro, este registro señala si la llamada tuvo éxito o no y ejecuta una instrucción de tipo RETURN FROM TRAP (a nivel núcleo) para regresar el control a la rutina de biblioteca (modo usuario), devolviendo un determinado resultado en el caso de que se tenga que devolver algo, y el éxito o fracaso de la llamada al sistema. En este paso se produce un cambio de modo, pasando de modo núcleo a modo usuario. Esto conlleva a restaurar el salvado parcial de contexto que se hizo del proceso en ejecución, pero comparado con una interrupción la sobrecarga es mínima.
5. Cuando finalizan esas acciones, la rutina de biblioteca a nivel de usuario descarga la pila y comprueba el resultado de la ejecución de la petición al sistema (que llegó hasta el modo núcleo), se lo devuelve al usuario y se continua por la siguiente línea de código del programa (o si estamos usando el *shell* éste queda a la espera del siguiente comando).

Dicho esto no debemos confundir una interrupción de E/S con una interrupción trap. La interrupción trap ocurre siempre que se hace una llamada al sistema (que puede surgir, entre otras, de una operación de E/S, como por ejemplo escribir en disco), es decir, cuando se necesita ejecutar una rutina o servicio propio del sistema operativo. La ejecución de trap (que devolverá si la llamada al sistema se ha hecho correctamente) puede dar lugar a continuar con el mismo proceso o que se cambie a otro diferente. Cuando ocurre una interrupción trap en una llamada al sistema (ya sea a través del código fuente de un programa, desde el shell o a partir de una interrupción de E/S que haga uso de una llamada al sistema), el procesador se pone en modo núcleo y el control se pasa al sistema operativo. Para este fin, parte del contexto se salva y se cambia de modo a una rutina del sistema operativo cargada en memoria principal. Sin embargo, la ejecución continúa dentro del proceso de usuario actual (ojo, no se cambia de proceso). De esta forma, no se realiza un cambio de proceso, sino un cambio de modo dentro del mismo proceso (distinción importante). Si el sistema operativo, después de haber realizado su trabajo, determina que el proceso actual debe continuar con su ejecución (no se ha terminado su rodaja de tiempo o no hay otro proceso con mayor prioridad), entonces el cambio de modo continúa con el programa “interrumpido” (nótese entre comillas dobles), restaurando su salvado de contexto parcial, pero no hemos cambiado de proceso. En caso contrario, si el sistema decide cambiar a otro proceso, el contexto del proceso que estaba en ejecución se guarda por completo, el proceso se bloquea o pasa a estado listo y se carga el contexto del siguiente proceso a ejecutar.

Por último, cuando el sistema operativo está manejando en un momento determinado  $n$  imágenes de procesos, cada imagen incluye no sólo el programa, datos, pila, y BCP, sino que además incluye o hace referencia a áreas de programa, datos y pila para los programas del núcleo. La Figura 3.16 sugiere la estructura de imagen de proceso típica para este caso. Se usa una pila de núcleo separada de los procesos de usuario para manejar llamadas/retornos cuando un proceso está en modo núcleo (trap). El código de sistema operativo y sus datos están en el espacio de direcciones compartidas y se comparten entre todos los procesos, es decir, cualquier proceso de usuario puede hacer referencia o llamar a rutinas en modo núcleo.



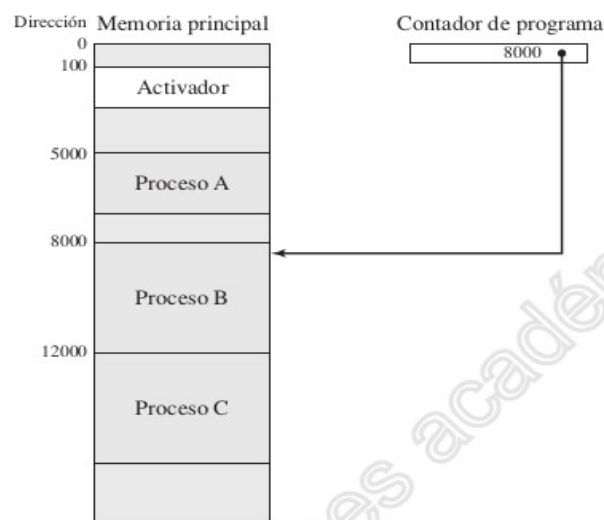
**Figura 3.16.** Imagen de proceso: el sistema operativo ejecuta dentro del espacio de usuario.

## Estados de los procesos

Como se acaba de comentar, para que un programa se ejecute, se debe crear un proceso o tarea para dicho programa. Desde el punto de vista del procesador, él ejecuta instrucciones de su repertorio de instrucciones en una secuencia dictada por el cambio de los valores del registro contador de programa. A lo largo del tiempo, el contador de programa puede apuntar al código de diferentes programas que son parte de diferentes procesos. Desde el punto de vista de un programa individual, su ejecución implica una secuencia de instrucciones dentro de dicho programa.

Se puede caracterizar el comportamiento de un determinado proceso, listando la secuencia de instrucciones que se ejecutan para dicho proceso. A esta lista se la denomina traza del proceso. Se puede caracterizar el comportamiento de un procesador mostrando cómo las trazas de varios procesos se entrelazan.

Considere un ejemplo. La Figura 3.2 muestra el despliegue en memoria de tres procesos. Para simplificar la exposición, se asume que dichos procesos no usan memoria virtual; por tanto, los tres procesos están representados por programas que residen en memoria principal. De manera adicional, existe un pequeño programa activador (*dispatcher*) que intercambia el procesador de un proceso a otro.



**Figura 3.2.** Instantánea de un ejemplo de ejecución (Figura 3.4) en el ciclo de instrucción 13.

La Figura 3.3 muestra las trazas de cada uno de los procesos en los primeros instantes de ejecución. Se muestran las 12 primeras instrucciones ejecutadas por los procesos A y C. El proceso B ejecuta 4 instrucciones y se asume que la cuarta instrucción invoca una operación de E/S, a la cual el proceso debe esperar.

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011
(a) Trazas del Proceso A	(b) Trazas del Proceso B	(c) Trazas del Proceso C

5000 = Dirección de comienzo del programa del Proceso A.

8000 = Dirección de comienzo del programa del Proceso B.

12000 = Dirección de comienzo del programa del Proceso C.

**Figura 3.3.** Trazas de los procesos de la Figura 3.2.



Ahora vea estas trazas desde el punto de vista del procesador. La Figura 3.4 muestra las trazas entrelazadas resultante de los 52 primeros ciclos de ejecución (por conveniencia los ciclos de instrucciones han sido numerados). En este ejemplo, se asume que el sistema operativo sólo deja que un proceso continúe durante seis ciclos de instrucción (rodaja de tiempo), después de los cuales se interrumpe; lo cual previene que un solo proceso monopolice el uso del tiempo del procesador. Como muestra la Figura 3.4, las primeras seis instrucciones del proceso A se ejecutan seguidas de una alarma de temporización (time-out) y de la ejecución de cierto código del activador, que ejecuta seis instrucciones antes de devolver el control al proceso B<sup>1</sup>. Después de que se ejecuten cuatro instrucciones, el proceso B solicita una acción de E/S, para la cual debe esperar. Por tanto, el procesador deja de ejecutar el proceso B y pasa a ejecutar el proceso C, por medio del activador. Después de otra alarma de temporización, el procesador vuelve al proceso A. Cuando este proceso llega a su temporización, el proceso B aún estará esperando que se complete su operación de E/S, por lo que el activador pasa de nuevo al proceso C.

1	5000		27	12004	
2	5001		28	12005	
3	5002				
4	5003		29	100	Temporización
5	5004		30	101	
6	5005		31	102	
			32	103	
7	100	Temporización	33	104	
8	101		34	105	
9	102		35	5006	
10	103		36	5007	
11	104		37	5008	
12	105		38	5009	
13	8000		39	5010	
14	8001		40	5011	
15	8002				Temporización
16	8003		41	100	
		Petición de E/S	42	101	
17	100		43	102	
18	101		44	103	
19	102		45	104	
20	103		46	105	
21	104		47	12006	
22	105		48	12007	
23	12000		49	12008	
24	12001		50	12009	
25	12002		51	12010	
26	12003		52	12011	
					Temporización

100 = Dirección de comienzo del programa activador.

Las zonas sombreadas indican la ejecución del proceso de activación;

la primera y la tercera columna cuentan ciclos de instrucciones;

la segunda y la cuarta columna las direcciones de las instrucciones que se ejecutan

**Figura 3.4.** Taza combinada de los procesos de la Figura 3.2.

## Un modelo de procesos de dos estados

La responsabilidad principal del sistema operativo es controlar la ejecución de los procesos; esto incluye determinar el patrón de entrelazado para la ejecución y asignar recursos a los procesos. El primer paso en el diseño de un sistema operativo para el control de procesos es describir el comportamiento que se desea que tengan los procesos.

Se puede construir el modelo más simple posible observando que, en un instante dado, un proceso está siendo ejecutando por el procesador o no. En este modelo, un proceso puede estar en dos estados: Ejecutando o No Ejecutando, como se muestra en la Figura 3.5a. Cuando el sistema

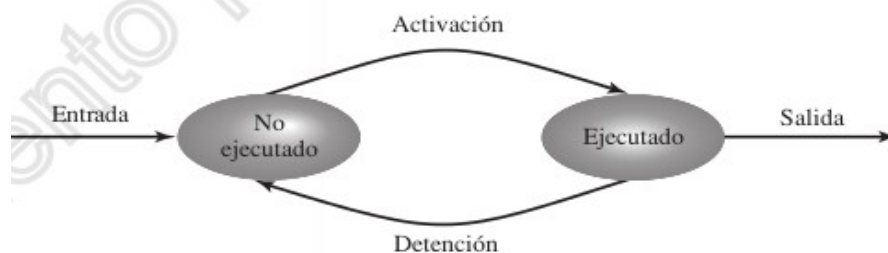
<sup>1</sup> El reducido número de instrucciones ejecutadas por los procesos y por el planificador es irreal; se ha utilizado para simplificar el ejemplo y clarificar las explicaciones.



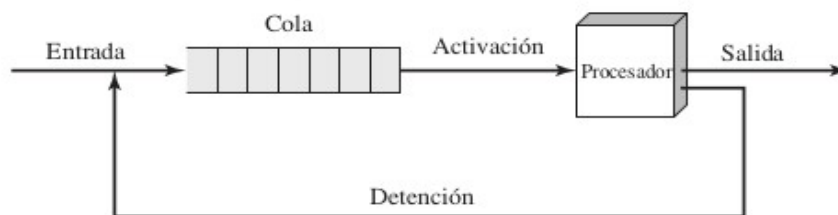
operativo crea un nuevo proceso, crea el bloque de control de proceso (BCP) para el nuevo proceso e inserta dicho proceso en el sistema en estado No Ejecutando. El proceso existe, es conocido por el sistema operativo, y está esperando su oportunidad de ejecutar. De cuando en cuando, el proceso actualmente en ejecución se interrumpirá y una parte del sistema operativo, el activador, seleccionará otro proceso a ejecutar. El proceso saliente pasará del estado Ejecutando a No Ejecutando y pasará a Ejecutando un nuevo proceso.

De este modelo simple, ya se puede apreciar algo del diseño de los elementos del sistema operativo. Cada proceso debe representarse de tal manera que el sistema operativo pueda seguirle la pista. Es decir, debe haber información correspondiente a cada proceso, incluyendo el estado actual y su localización en memoria; esto es el bloque de control de programa. Los procesos que no están ejecutando deben estar en una especie de cola, esperando su turno de ejecución. La Figura 3.5b sugiere esta estructura. Existe una sola cola cuyas entradas son punteros al BCP de un proceso en particular. Alternativamente, la cola debe consistir en una lista enlazada de bloques de datos, en la cual cada bloque representa un proceso.

Podemos describir el comportamiento del activador en términos de este diagrama de colas. Un proceso que se interrumpe se transfiere a la cola de procesos en espera. Alternativamente, si el proceso ha finalizado o ha sido abortado, se descarta (sale del sistema). En cualquier caso, el activador selecciona un proceso de la cola para ejecutar.



(a) Diagrama de transiciones de estados



(b) Modelos de colas

**Figura 3.5.** Modelo de proceso de dos estados.

Por último, en ciertos sistemas operativos, un proceso puede terminarse por parte del proceso que lo creó o cuando dicho proceso padre a su vez ha terminado.

### MODELO DE PROCESO DE CINCO ESTADOS

Si todos los procesos estuviesen siempre preparados para ejecutar, la gestión de colas proporcionada en la Figura 3.5b sería efectiva. La cola es una lista de tipo FIFO y el procesador opera siguiendo una estrategia cíclica (*round-robin* o turno rotatorio) sobre todos los procesos disponibles (cada proceso de la cola tiene cierta cantidad de tiempo, por turnos, para ejecutar y regresar de nuevo a la cola, a menos que se bloquee). Sin embargo, hasta con el sencillo ejemplo que vimos antes, esta implementación es inadecuada: algunos procesos que están en el estado de No Ejecutando están listos para ejecutar, mientras que otros están bloqueados, esperando a que se complete una operación de E/S. Por tanto, utilizando una única cola, el activador no puede seleccionar únicamente los procesos que lleven más tiempo en la cola. En su lugar, debería recorrer la lista buscando los procesos que no estén bloqueados y que lleven en la cola más tiempo.

Una forma más natural para manejar esta situación es dividir el estado de No Ejecutando en dos estados, Listo y Bloqueado. Esto se muestra la Figura 3.6. Para gestionarlo correctamente, se han añadido dos estados adicionales que resultarán muy útiles. Estos cinco estados en el nuevo diagrama son los siguientes:

- **Ejecutando.** El proceso está actualmente en ejecución. Para este capítulo asumimos que el computador tiene un único procesador, de forma que sólo un proceso puede estar en este estado en un instante determinado.
- **Listo.** Un proceso que se prepara para ejecutar cuando tenga oportunidad.
- **Bloqueado.** Un proceso que no puede ejecutar hasta que se cumpla un evento determinado o se complete una operación E/S.
- **Nuevo.** Un proceso que se acaba de crear y que aún no ha sido admitido en el grupo de procesos ejecutables por el sistema operativo. Típicamente, se trata de un nuevo proceso que no ha sido cargado en memoria principal, aunque su bloque de control de proceso (BCP) si ha sido creado.
- **Saliente.** Un proceso que ha sido liberado del grupo de procesos ejecutables por el sistema operativo, debido a que ha sido detenido o que ha sido abortado por alguna razón.

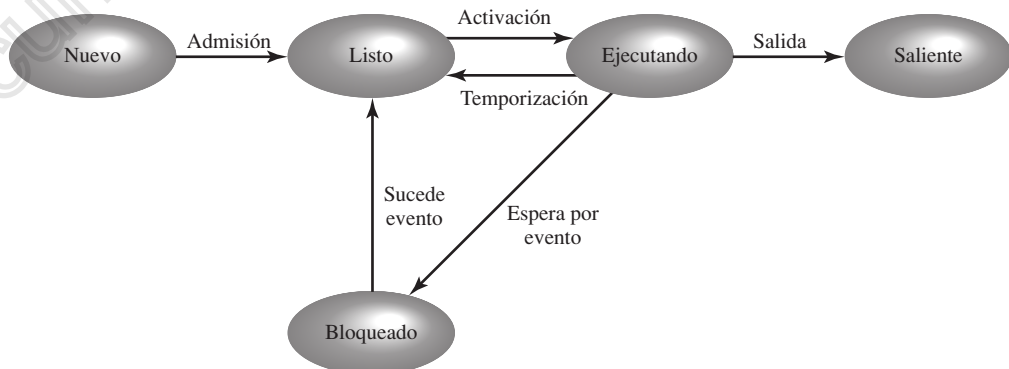




Figura 3.6. Modelo de proceso de cinco estados.

Los estados Nuevo y Saliente son útiles para construir la gestión de procesos. El estado Nuevo se corresponde con un proceso que acaba de ser definido. Por ejemplo, si un nuevo usuario intenta entrar dentro de un sistema de tiempo compartido o cuando se solicita un nuevo trabajo a un sistema de proceso por lotes, el sistema operativo puede definir un nuevo proceso en dos etapas. Primero, el sistema operativo realiza todas las tareas internas que correspondan. Se asocia un identificador a dicho proceso. Se reservan y construyen todas aquellas tablas que se necesiten para gestionar al proceso. En este punto, el proceso se encuentra en el estado Nuevo. Esto significa que el sistema operativo ha realizado todas las tareas necesarias para crear el proceso pero el proceso en sí, aún no se ha puesto en ejecución. Por ejemplo, un sistema operativo puede limitar el número de procesos que puede haber en el sistema por razones de rendimiento o limitaciones de memoria principal. Mientras un proceso está en el estado Nuevo, la información relativa al proceso que se necesite por parte del sistema operativo se mantiene en tablas de control de memoria principal. Sin embargo, el proceso en sí mismo no se encuentra en memoria principal. Esto es, el código de programa a ejecutar no se encuentra en memoria principal, y no se ha reservado ningún espacio para los datos asociados al programa. Cuando un proceso se encuentra en el estado Nuevo, el programa permanece en almacenamiento secundario, normalmente en disco<sup>4</sup>.

De forma similar, un proceso sale del sistema en dos fases. Primero, el proceso termina cuando alcanza su punto de finalización natural, cuando es abortado debido a un error no recuperable, o cuando otro proceso con autoridad apropiada causa que el proceso se aborte. La terminación mueve el proceso al estado Saliente. En este punto, el proceso no es elegible de nuevo para su ejecución. Las tablas y otra información asociada con el trabajo se encuentran temporalmente preservadas por el sistema operativo, el cual proporciona tiempo para que programas auxiliares o de soporte extraigan la información necesaria. Por ejemplo, un programa de auditoría puede requerir registrar el tiempo de proceso y otros recursos utilizados por este proceso saliente con objeto de realizar una contabilidad de los recursos del sistema. Un programa de utilidad puede requerir extraer información sobre el histórico de los procesos por temas relativos con el rendimiento o análisis de la utilización. Una vez que estos programas han extraído la información necesaria, el sistema operativo no necesita mantener ningún dato relativo al proceso y el proceso se borra del sistema.

La Figura 3.6 indica que tipos de eventos llevan a cada transición de estado para cada proceso; las posibles transiciones son las siguientes:

- **Null → Nuevo.** Se crea un nuevo proceso para ejecutar un programa. Este evento ocurre por cualquiera de las relaciones indicadas en la tabla 3.1. 
- **Nuevo → Listo.** El sistema operativo mueve a un proceso del estado nuevo al estado listo cuando éste se encuentre preparado para ejecutar un nuevo proceso. La mayoría de sistemas fijan un límite basado en el número de procesos existentes o la cantidad de memoria virtual que se podrá utilizar por parte de los procesos existentes. Este límite asegura que no haya demasiados procesos activos y que se degrade el rendimiento sistema.
- **Listo → Ejecutando.** Cuando llega el momento de seleccionar un nuevo proceso para ejecutar, el sistema operativo selecciona uno los procesos que se encuentre en el estado Listo. Esta es una tarea la lleva acabo el planificador. El planificador se estudiará más adelante en la Parte Cuatro.
- **Ejecutando → Saliente.** el proceso actual en ejecución se finaliza por parte del sistema operativo tanto si el proceso indica que ha completado su ejecución como si éste se aborta. Véase tabla 3.2. 

<sup>4</sup> En las explicaciones de este párrafo, hemos ignorado el concepto de memoria virtual. En sistemas que soporten la memoria virtual, cuando un proceso se mueve de Nuevo a Listo, su código de programa y sus datos se cargan en memoria virtual. La memoria virtual se ha explicado brevemente en el Capítulo 2 y se examinará con más detalle en el Capítulo 8.

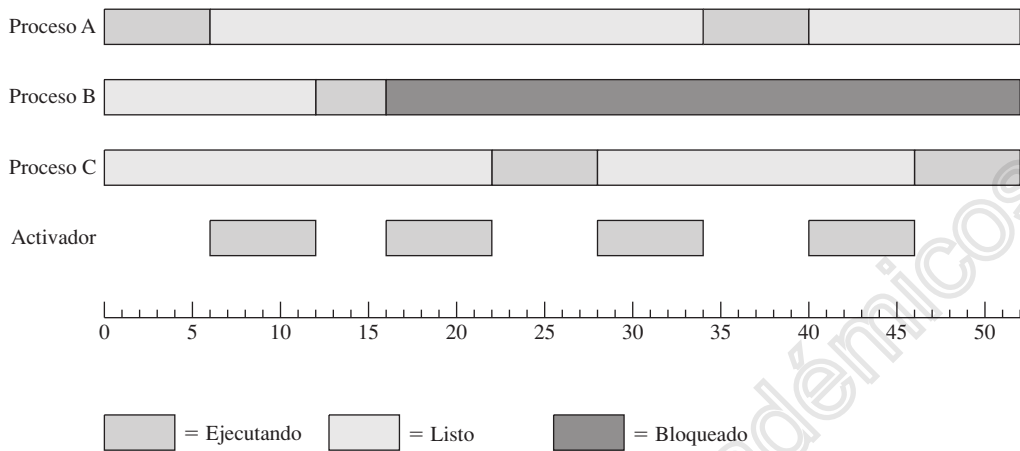
- **Ejecutando → Listo.** La razón más habitual para esta transición es que el proceso en ejecución haya alcanzado el máximo tiempo posible de ejecución de forma ininterrumpida; prácticamente todos los sistemas operativos multiprogramados imponen este tipo de restricción de tiempo. Existen otras posibles causas alternativas para esta transición, que no están incluidas en todos los sistemas operativos. Es de particular importancia el caso en el cual el sistema operativo asigna diferentes niveles de prioridad a diferentes procesos. Supóngase, por ejemplo, que el proceso A está ejecutando a un determinado nivel de prioridad, y el proceso B, a un nivel de prioridad mayor, y que se encuentra bloqueado. Si el sistema operativo se da cuenta de que se produce un evento al cual el proceso B está esperando, moverá el proceso B al estado de Listo. Esto puede interrumpir al proceso A y poner en ejecución al proceso B. Decimos, en este caso, que el sistema operativo ha expulsado al proceso A<sup>5</sup>. Adicionalmente, un proceso puede voluntariamente dejar de utilizar el procesador. Un ejemplo son los procesos que realiza alguna función de auditoría o de mantenimiento de forma periódica.
- **Ejecutando → Bloqueado.** Un proceso se pone en el estado Bloqueado si solicita algo por lo cual debe esperar. Una solicitud al sistema operativo se realiza habitualmente por medio de una llamada al sistema; esto es, una llamada del proceso en ejecución a un procedimiento que es parte del código del sistema operativo. Por ejemplo, un proceso ha solicitado un servicio que el sistema operativo no puede realizar en ese momento. Puede solicitar un recurso, como por ejemplo un fichero o una sección compartida de memoria virtual, que no está inmediatamente disponible. Cuando un proceso quiere iniciar una acción, tal como una operación de E/S, que debe completarse antes de que el proceso continúe. Cuando un proceso se comunica con otro, un proceso puede bloquearse mientras está esperando a que otro proceso le proporcione datos o esperando un mensaje de ese otro proceso.
- **Bloqueado → Listo.** Un proceso en estado Bloqueado se mueve al estado Listo cuando sucede el evento por el cual estaba esperando.
- **Listo → Saliente.** Por claridad, esta transición no se muestra en el diagrama de estados. En algunos sistemas, un padre puede terminar la ejecución de un proceso hijo en cualquier momento. También, si el padre termina, todos los procesos hijos asociados con dicho padre pueden finalizarse.
- **Bloqueado → Saliente.** Se aplican los comentarios indicados en el caso anterior.

Si regresamos a nuestro sencillo ejemplo, Figura 3.7, ahí se muestra la transición entre cada uno de los estados de proceso. La figura 3.8a sugiere la forma de aplicar un esquema de dos colas: la cola de Listos y la cola de Bloqueados. Cada proceso admitido por el sistema, se coloca en la cola de Listos. Cuando llega el momento de que el sistema operativo seleccione otro proceso a ejecutar, selecciona uno de la cola de Listos. En ausencia de un esquema de prioridad, esta cola puede ser una lista de tipo FIFO (*first-in-first-out*). Cuando el proceso en ejecución termina de utilizar el procesador, o bien finaliza o bien se coloca en la cola de Listos o de Bloqueados, dependiendo de las circunstancias. Por último, cuando sucede un evento, cualquier proceso en la cola de Bloqueados que únicamente esté esperando a dicho evento, se mueve a la cola de Listos.

Esta última transición significa que, cuando sucede un evento, el sistema operativo debe recorrer la cola entera de Bloqueados, buscando aquellos procesos que estén esperando por dicho evento. En

---

<sup>5</sup> En general, el término **expulsión** (*preemption*) se define como la reclamación de un recurso por parte de un proceso antes de que el proceso que lo poseía finalice su uso. En este caso, el recurso es el procesador. El proceso está ejecutando y puede continuar su ejecución pero es expulsado por otro proceso que va a entrar a ejecutar.



**Figura 3.7.** Estado de los procesos de la traza de la Figura 3.4.

los sistemas operativos con muchos procesos, esto puede significar cientos o incluso miles de procesos en esta lista, por lo que sería mucho más eficiente tener una cola por cada evento. De esta forma, cuando sucede un evento, la lista entera de procesos de la cola correspondiente se movería al estado de Listo (Figura 3. 8b).

Un refinamiento final sería: si la activación de procesos está dictada por un esquema de prioridades, sería conveniente tener varias colas de procesos listos, una por cada nivel de prioridad. El sistema operativo podría determinar cual es el proceso listo de mayor prioridad simplemente seleccionando éstas en orden.

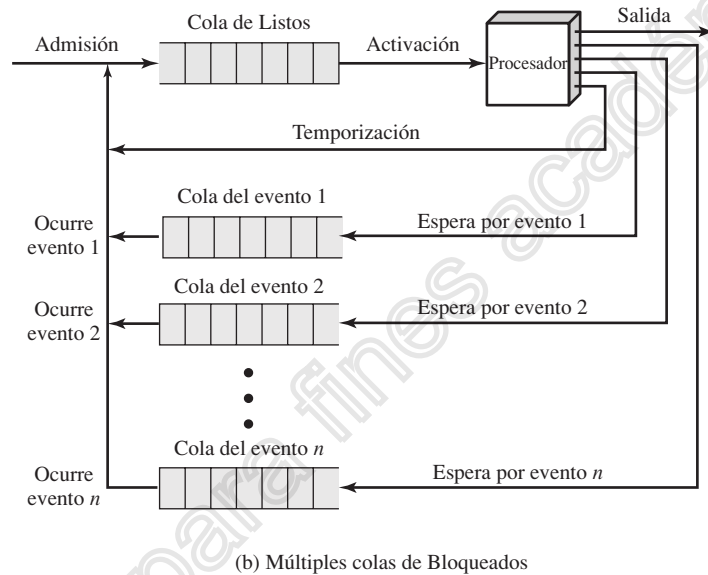
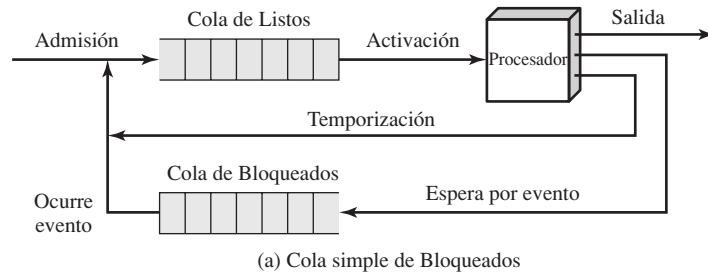
## PROCESOS SUSPENDIDOS

**La necesidad de intercambio o *swapping*.** Los tres principales estados descritos (Listo, Ejecutando, Bloqueado) proporcionan una forma sistemática de modelizar el comportamiento de los procesos y diseñar la implementación del sistema operativo. Se han construido algunos sistemas operativos utilizando únicamente estos tres estados.

Sin embargo, existe una buena justificación para añadir otros estados al modelo. Para ver este beneficio de nuevos estados, vamos a suponer un sistema que no utiliza memoria virtual. Cada proceso que se ejecuta debe cargarse completamente en memoria principal. Por ejemplo, en la Figura 3.8b, todos los procesos en todas las colas deben residir en memoria principal.

Recuérdese que la razón de toda esta compleja maquinaria es que las operaciones de E/S son mucho más lentas que los procesos de cómputo y, por tanto, el procesador en un sistema monoprogramado estaría ocioso la mayor parte del tiempo. Pero los ajustes de la Figura 3.8b no resuelven completamente el problema. Es verdad que, en este caso, la memoria almacena múltiples procesos y el procesador puede asignarse a otro proceso si el que lo usa se queda bloqueado. La diferencia de velocidad entre el procesador y la E/S es tal que sería muy habitual que todos los procesos en memoria se encontrasen a esperas de dichas operaciones. Por tanto, incluso en un sistema multiprogramado, el procesador puede estar ocioso la mayor parte del tiempo.

¿Qué se puede hacer? La memoria principal puede expandirse para acomodar más procesos. Hay dos fallos en esta solución. Primero, existe un coste asociado a la memoria principal, que, desde un



**Figura 3.8.** Modelo de colas de la Figura 3.6.

coste reducido a nivel de bytes, comienza a incrementarse según nos acercamos a un almacenamiento de gigabytes. Segundo, el apetito de los programas a nivel de memoria ha crecido tan rápido como ha bajado el coste de las memorias. De forma que las grandes memorias actuales han llevado a ejecutar procesos de gran tamaño, no más procesos.

Otra solución es el *swapping* (memoria de intercambio), que implica mover parte o todo el proceso de memoria principal al disco. Cuando ninguno de los procesos en memoria principal se encuentra en estado Listo, el sistema operativo intercambia uno de los procesos bloqueados a disco, en la cola de Suspendidos. Esta es una lista de procesos existentes que han sido temporalmente expulsados de la memoria principal, o suspendidos. El sistema operativo trae otro proceso de la cola de Suspendidos o responde a una solicitud de un nuevo proceso. La ejecución continúa con los nuevos procesos que han llegado.

El *swapping*, sin embargo, es una operación de E/S, y por tanto existe el riesgo potencial de hacer que el problema empeore. Pero debido a que la E/S en disco es habitualmente más rápida que la E/S sobre otros sistemas (por ejemplo, comparado con cinta o impresora), el *swapping* habitualmente mejora el rendimiento del sistema.

Con el uso de *swapping* tal y como se ha escrito, debe añadirse un nuevo estado a nuestro modelo de comportamiento de procesos (Figura 3.9a): el estado Suspendido. Cuando todos los procesos en



memoria principal se encuentran en estado Bloqueado, el sistema operativo puede suspender un proceso poniéndolo en el estado Suspendido y transfiriéndolo a disco. El espacio que se libera en memoria principal puede usarse para traer a otro proceso.

Cuando el sistema operativo ha realizado la operación de *swap* (transferencia a disco de un proceso), tiene dos opciones para seleccionar un nuevo proceso para traerlo a memoria principal: puede admitir un nuevo proceso que se haya creado o puede traer un proceso que anteriormente se encontrase en estado de Suspendido. Parece que sería preferible traer un proceso que anteriormente estuviese suspendido, para proporcionar dicho servicio en lugar de incrementar la carga total del sistema.

Pero, esta línea de razonamiento presenta una dificultad: todos los procesos que fueron suspendidos se encontraban previamente en el estado de Bloqueado en el momento de su suspensión. Claramente no sería bueno traer un proceso bloqueado de nuevo a memoria porque podría no encontrarse todavía listo para la ejecución. Se debe reconocer, sin embargo, que todos los procesos en estado Suspendido estaban originalmente en estado Bloqueado, en espera de un evento en particular. Cuando el evento sucede, el proceso no está Bloqueado y está potencialmente disponible para su ejecución.

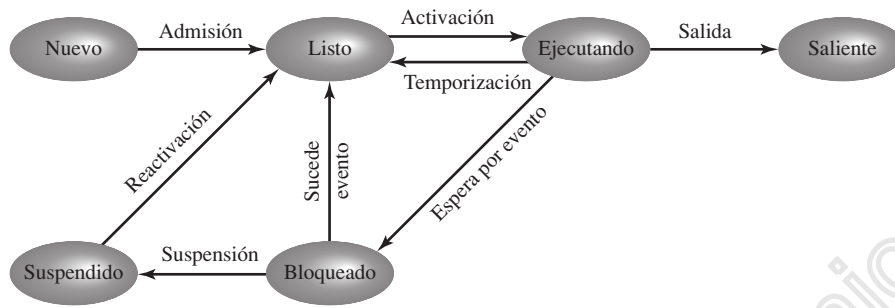
De esta forma, necesitamos replantear este aspecto del diseño. Hay dos conceptos independientes aquí: si un proceso está esperando a un evento (Bloqueado o no) y si un proceso está transferido de memoria a disco (suspendido o no). Para describir estas combinaciones de  $2 \times 2$  necesitamos cuatro estados:

- **Listo.** El proceso está en memoria principal disponible para su ejecución.
- **Bloqueado.** El proceso está en memoria principal y esperando un evento.
- **Bloqueado/Suspendido.** El proceso está en almacenamiento secundario y esperando un evento.
- **Listo/Suspendido.** El proceso está en almacenamiento secundario pero está disponible para su ejecución tan pronto como sea cargado en memoria principal.

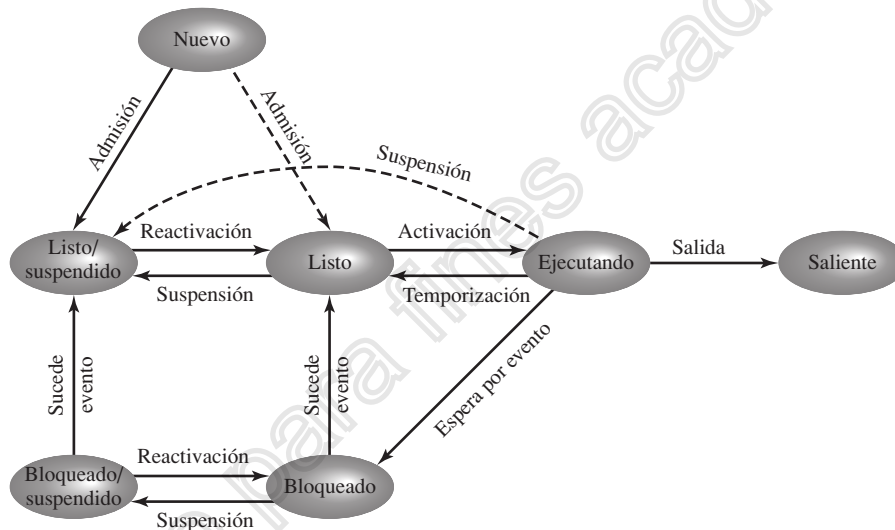
~~Antes de ir a ver un diagrama de transiciones de estado que incluya estos dos nuevos estados Suspendidos, se debe mencionar otro punto. La explicación ha asumido que no se utiliza memoria virtual y que un proceso está completamente en memoria principal, o completamente fuera de la misma. Con un esquema de memoria virtual, es posible ejecutar un proceso que está sólo parcialmente en memoria principal. Si se hace referencia a una dirección de proceso que no se encuentra en memoria principal, la porción correspondiente del proceso se trae a ella. El uso de la memoria principal podría parecer que elimina la necesidad del *swapping* explícito, porque cualquier dirección necesitada por un proceso que la demande puede traerse o transferirse de disco a memoria por el propio hardware de gestión de memoria del procesador. Sin embargo, como veremos en el Capítulo 8 el rendimiento de los sistemas de memoria virtual pueden colapsarse si hay un número suficientemente grande de procesos activos, todos ellos están parcialmente en memoria principal. De esta forma, incluso con un sistema de memoria virtual, el sistema operativo necesita hacer *swapping* de los procesos de forma explícita y completa, de vez en cuando, con la intención de mejorar el rendimiento.~~

Veamos ahora, en la Figura 3.9b, el modelo de transición de estados que ha sido diseñado. (Las líneas punteadas de la figura indican transiciones posibles pero no necesarias.) Las nuevas transiciones más importantes son las siguientes:

- **Bloqueado → Bloqueado/Suspendido.** Si no hay procesos listos, entonces al menos uno de los procesos bloqueados se transfiere al disco para hacer espacio para otro proceso que no se encuentra bloqueado. Esta transición puede realizarse incluso si hay procesos listos disponi-



(a) Con un único estado Suspendido



(b) Con dos estados Suspendido

**Figura 3.9.** Diagrama de transición de estados de procesos con estado suspendidos.

bles, si el sistema operativo determina que el proceso actualmente en ejecución o los procesos listos que desea ejecutar requieren más memoria principal para mantener un rendimiento adecuado.

- **Bloqueado/Suspendido → Listo/Suspendido.** Un proceso en el estado Bloqueado/Suspendido se mueve al estado Listo/Suspendido cuando sucede un evento al que estaba esperando. Nótese que esto requiere que la información de estado concerniente a un proceso suspendido sea accesible para el sistema operativo.
- **Listo/Suspendido → Listo.** Cuando no hay más procesos listos en memoria principal, el sistema operativo necesitará traer uno para continuar la ejecución. Adicionalmente, puede darse el caso de que un proceso en estado Listo/Suspendido tenga mayor prioridad que cualquiera de los procesos en estado Listo. En este caso, el sistema operativo puede haberse diseñado para determinar que es más importante traer un proceso de mayor prioridad que para minimizar el efecto del *swapping*.

- **Listo → Listo/Suspendido.** Normalmente, el sistema operativo preferirá suspender procesos bloqueados que un proceso Listo, porque un proceso Listo se puede ejecutar en ese momento, mientras que un proceso Bloqueado ocupa espacio de memoria y no se puede ejecutar. Sin embargo, puede ser necesario suspender un proceso Listo si con ello se consigue liberar un bloque suficientemente grande de memoria. También el sistema operativo puede decidir suspender un proceso Listo de baja prioridad antes que un proceso Bloqueado de alta prioridad, si se cree que el proceso Bloqueado estará pronto listo.

Otras transiciones interesantes de considerar son las siguientes:

- **Nuevo → Listo/Suspendido y Nuevo a Listo.** Cuando se crea un proceso nuevo, puede añadirse a la cola de Listos o a la cola de Listos/Suspendidos. En cualquier caso, el sistema operativo puede crear un bloque de control de proceso (BCP) y reservar el espacio de direcciones del proceso. Puede ser preferible que el sistema operativo realice estas tareas internas cuanto antes, de forma que pueda disponer de suficiente cantidad de procesos no bloqueados. Sin embargo, con esta estrategia, puede ocurrir que no haya espacio suficiente en memoria principal para el nuevo proceso; de ahí el uso de la transición (Nuevo→Listo/Suspendido). Por otro lado, deseamos hacer hincapié en que la filosofía de creación de procesos diferida, haciéndolo cuanto más tarde, hace posible reducir la sobrecarga del sistema operativo y le permite realizar las tareas de creación de procesos cuando el sistema está lleno de procesos bloqueados.
- **Bloqueado/Suspendido → Bloqueado.** La incursión de esta transición puede parecer propia de un diseño más bien pobre. Después de todo, si hay un proceso que no está listo para ejecutar y que no está en memoria principal, ¿qué sentido tiene traerlo? Pero considérese el siguiente escenario: un proceso termina, liberando alguna memoria principal. Hay un proceso en la cola de Bloqueados/Suspendidos con mayor prioridad que todos los procesos en la cola de Listos/Suspendidos y el sistema operativo tiene motivos para creer que el evento que lo bloquea va a ocurrir en breve. Bajo estas circunstancias, sería razonable traer el proceso Bloqueado a memoria por delante de los procesos Listos.
- **Ejecutando → Listo/Suspendido.** Normalmente, un proceso en ejecución se mueve a la cola de Listos cuando su tiempo de uso del procesador finaliza. Sin embargo, si el sistema operativo expulsa al proceso en ejecución porque un proceso de mayor prioridad en la cola de Bloqueado/Suspendido acaba de desbloquearse, el sistema operativo puede mover al proceso en ejecución directamente a la cola de Listos/Suspendidos y liberar parte de la memoria principal.
- **De cualquier estado → Saliente.** Habitualmente, un proceso termina cuando está ejecutando, bien porque ha completado su ejecución o debido a una condición de fallo. Sin embargo, en algunos sistemas operativos un proceso puede terminarse por el proceso que lo creó o cuando el proceso padre a su vez ha terminado. Si se permite esto, un proceso en cualquier estado puede moverse al estado Saliente.

~~Otros usos para la suspensión de procesos. Hace poco, hemos definido el concepto de procesos suspendidos como el proceso que no se encuentra en memoria principal. Un proceso que no está en memoria principal no está disponible de forma inmediata para su ejecución, independientemente de si está a la espera o no de un evento.~~

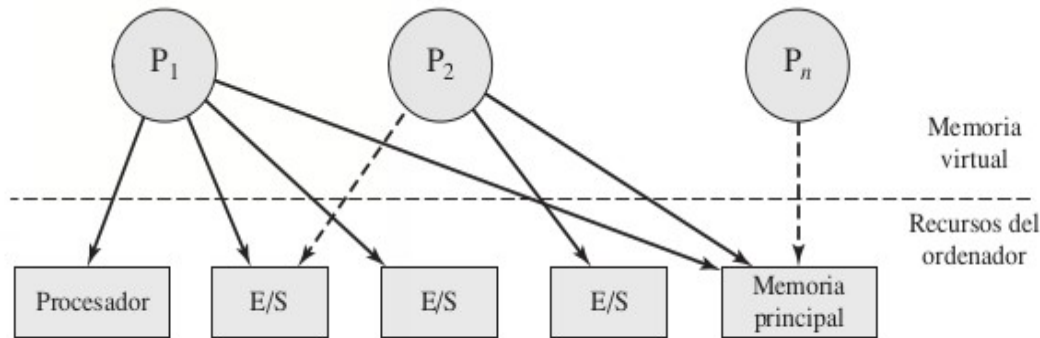
~~Podemos analizar el concepto de procesos suspendidos, definiendo un proceso suspendido como el que cumple las siguientes características:~~

- ~~1. El proceso no está inmediatamente disponible para su ejecución.~~

## Procesos y recursos

El sistema operativo controla los eventos dentro del computador, planifica y activa los procesos para su ejecución por el procesador, reserva recursos para los mismos y responde a las solicitudes de servicios básicos de los procesos de usuario. Fundamentalmente, se piensa en el sistema operativo como en la entidad que gestiona el uso de recursos del sistema por parte de los procesos.

Este concepto se ilustra en la Figura 3.10.



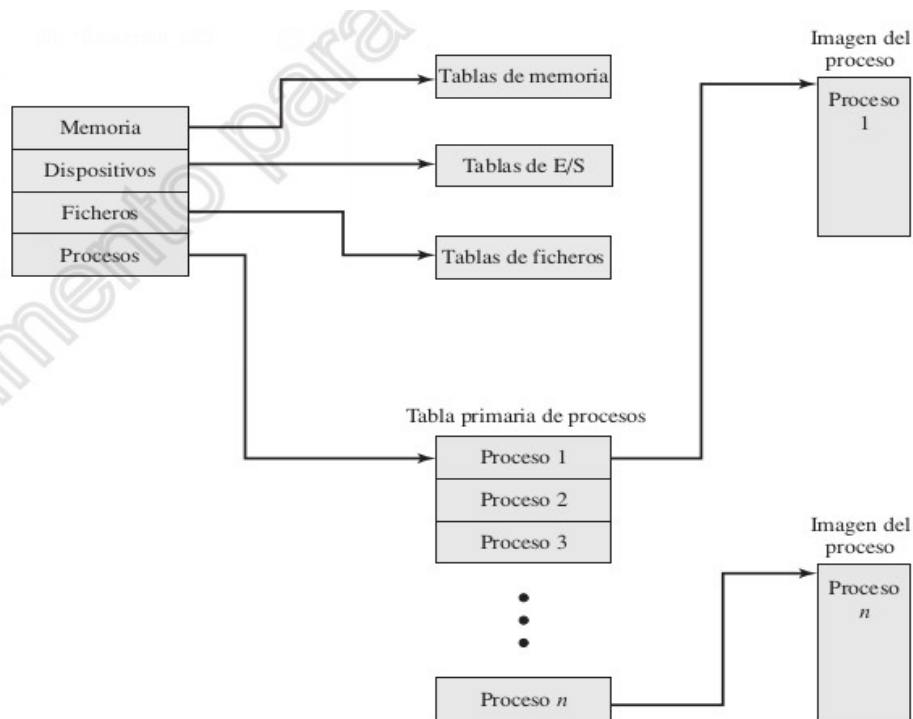
**Figura 3.10.** Procesos y recursos (reserva de recursos en una instantánea del sistema).

En un entorno multiprogramado, hay numerosos procesos ( $P_1, \dots, P_n$ ) creados y residentes en memoria principal y virtual. Cada proceso, durante el transcurso de su ejecución, necesita acceder a ciertos recursos del sistema, incluido el procesador, los dispositivos de E/S y la memoria principal. En la figura, el proceso  $P_1$  está ejecutando en memoria principal, y controla dos dispositivos de E/S. El proceso  $P_2$  está también totalmente en memoria principal pero está bloqueado esperando a disponer de un dispositivo de E/S que está asignado a  $P_1$ . El proceso  $P_n$  se encuentra transferido a disco (swapping) y está por tanto suspendido.

¿Qué información necesita el sistema operativo para controlar los procesos y gestionar los recursos de estos?

## Estructuras de control del sistema operativo

Si el sistema operativo se encarga de la gestión de procesos y recursos, debe disponer de información sobre el estado actual de cada proceso y cada recurso. El mecanismo universal para proporcionar esta información es el siguiente: el sistema operativo construye y mantiene tablas de información sobre cada entidad que gestiona. Se indica una idea general del alcance de esta tarea en la Figura 3.11, que muestra cuatro diferentes tipos de tablas mantenidas por el sistema operativo: memoria, E/S, ficheros y procesos. A pesar de que los detalles difieren de un sistema operativo a otro, fundamentalmente, todos los sistemas operativos mantienen información de estas cuatro categorías.



**Figura 3.11.** Estructura general de las tablas de control del sistema operativo.

Las **tablas de memoria** se usan para mantener un registro tanto de la memoria principal (real) como de la secundaria (virtual). Parte de la memoria principal está reservada para el uso del sistema operativo; el resto está disponible para el uso de los procesos. Los procesos también pueden mantenerse en memoria secundaria utilizando algún tipo de memoria virtual o técnicas de swapping. Las tablas de memoria deben incluir la siguiente información:

- Las reservas de memoria principal por parte de los procesos.
- Las reservas de memoria secundaria por parte de los procesos.
- Todos los atributos de protección que restringe el uso de la memoria principal y virtual, de forma que los procesos puedan acceder a ciertas áreas de memoria compartida.
- La información necesaria para manejar la memoria virtual.

El sistema operativo debe utilizar las **tablas de E/S** para gestionar los dispositivos de E/S y los canales del computador. Pero, en un instante determinado, un dispositivo E/S puede estar disponible o asignado a un proceso en particular. Si la operación de E/S se está realizando, el sistema operativo necesita conocer el estado de la operación y la dirección de memoria principal del área usada como fuente o destino de la transferencia de E/S.

El sistema operativo también puede mantener las **tablas de ficheros**. Estas tablas proporcionan información sobre la existencia de ficheros, su posición en almacenamiento secundario, su estado actual, y otros atributos. La mayoría de, o prácticamente toda esta información, se puede gestionar por el sistema de ficheros, en cuyo caso el sistema operativo tiene poco o ningún conocimiento sobre los ficheros. En otros sistemas operativos, la gran mayoría del detalle de la gestión de ficheros sí que gestiona en el propio sistema operativo .

En último lugar, el sistema operativo debe mantener **tablas de procesos** para gestionar los procesos. A pesar de que la Figura 3.11 muestra cuatro tipos diferentes de tablas debe quedar claro que estas tablas se encuentran entrelazadas y referenciadas entre sí de alguna manera. Memoria, E/S, y ficheros se gestionan por parte de los procesos, de forma que debe haber algunas referencias en estos recursos, directa o indirectamente, desde las tablas de procesos. Los ficheros indicados en las

tablas de ficheros son accesibles mediante dispositivos E/S y estarán, en algún caso, residentes en memoria virtual o principal. Las tablas son también accesibles para el sistema operativo, y además están controladas por la gestión de memoria.

Segundo, ¿cómo puede el sistema operativo crear las tablas iniciales? Ciertamente, el sistema operativo debe tener alguna información sobre el entorno básico, tal como: cuánta memoria física existe, cuáles son los dispositivos de E/S y cuáles son sus identificadores, por ejemplo. Esto es una cuestión de configuración. Esto es, cuando el sistema operativo se inicializa, debe tener acceso a algunos datos de configuración que definen el entorno básico y estos datos se deben crear por medio de algún software de autoconfiguración que el sistema ejecuta a su inicio.



ejecutar. Durante esta fase, es más conveniente ver la ejecución como fuera de cualquiera de los procesos.

De alguna forma, esta visión de un sistema operativo es muy reseñable. De una forma más sencilla, en determinados instantes de tiempo, un proceso salva su estado, elige otro proceso a ejecutar entre aquellos que están listos, y delega el control a dicho proceso. La razón por la cual este esquema no se convierte en una situación arbitraria y caótica es que durante los instantes críticos el código que se está ejecutando es código de compartido de sistema operativo, no código de usuario. Debido al concepto de modo usuario y modo núcleo, el usuario no puede interferir con las rutinas del sistema operativo, incluso cuando se están ejecutando dentro del entorno del proceso del usuario. Esto nos recuerda que existe una clara distinción entre el concepto de proceso y programa y que la relación entre los dos no es uno a uno. Dentro de un proceso, pueden ejecutarse tanto un programas de usuario como programas del sistema operativo. Los programas de sistema operativo que se ejecutan en varios procesos de usuario son idénticos.

**Sistemas operativos basados en procesos.** Otra alternativa, mostrada en la Figura 3.15c, es implementar un sistema operativo como una colección de procesos de sistema. Como en las otras opciones, el software que es parte del núcleo se ejecuta en modo núcleo. En este caso, las principales funciones del núcleo se organizan como procesos independientes. De nuevo, debe haber una pequeña cantidad de código para intercambio de procesos que se ejecuta fuera de todos los procesos.

Esta visión tiene diferentes ventajas. Impone una disciplina de diseño de programas que refuerza el uso de sistemas operativos modulares con mínimas y claras interfaces entre los módulos. Adicionalmente, otras funciones del sistema operativo que no sean críticas están convenientemente separadas como otros procesos. Por ejemplo, hemos mencionado anteriormente un programa de monitorización que recoge niveles de utilización de diferentes recursos (procesador, memoria, canales) y el ratio de progreso de los procesos en el sistema. Debido a que este programa no realiza ningún servicio a ningún otro programa activo, sólo se puede invocar por el sistema operativo. Como proceso, esta función puede ejecutarse a un nivel de prioridad determinado, intercalándose con otros procesos bajo el control del activador. Por último, la implementación del sistema operativo como un grupo de procesos en entornos de multiprocesadores y multicomputadores, en los cuales determinados servicios del sistema operativo se pueden enviar a procesadores dedicados, incrementando el rendimiento.

### 3.5. UNIX SVR4 PROCESS MANAGEMENT

UNIX System V hace uso de una gestión de procesos simple pero muy potente que es fácilmente visible a nivel de usuario. UNIX sigue el modelo mostrado en la Figura 3.15b, en la cual la gran mayoría del sistema operativo se ejecuta dentro del entorno del proceso. De esta forma se necesitan dos modos, usuario y núcleo. UNIX utiliza también dos categorías de procesos, procesos de sistema y procesos de usuario. Los procesos del sistema ejecutan en modo núcleo y ejecuta código del sistema operativo para realizar tareas administrativas o funciones internas, como reserva de memoria o *swapping* de procesos. Los procesos de usuario operan en modo usuario para ejecutar programas y utilidades y en modo núcleo para ejecutar instrucciones que pertenecen al núcleo. Un proceso de usuario entra en modo núcleo por medio de la solicitud de la llamada al sistema, cuando se genera una de excepción (fallo) o cuando ocurre una interrupción.

#### ESTADOS DE LOS PROCESOS

En los sistemas operativos UNIX se utilizan un total de nueve estados de proceso; estos se encuentran recogidos en la Tabla 3.9 y el diagrama de transiciones se muestra en la Figura 3.17 (basada en figura

en [BACH86]). Esta figura es similar a la Figura 3.9b, con dos estados de procesos dormidos, propios de UNIX, correspondientes a dos estados de bloqueo. Las diferencias se pueden resumir en:

- UNIX utiliza dos estados Ejecutando que indican si el proceso está ejecutando en modo usuario o en modo núcleo.
- Se debe realizar la distinción entre dos estados: (Listo para Ejecutar, en Memoria) y (Expulsado). Estos son esencialmente el mismo estado, como indica la línea punteada que los une. La distinción se realiza para hacer énfasis en la forma mediante la cual se llega al estado de Expulsado. Cuando un proceso ejecuta en modo núcleo (como resultado de una llamada al sistema, interrupción de reloj o interrupción de E/S), requerirá un tiempo para que el sistema operativo complete su trabajo y esté listo para devolver el control al proceso de usuario. En este punto, el núcleo decide expulsar al proceso actual en favor de uno de los procesos listos de mayor prioridad. En este caso, el proceso actual se mueve al estado Expulsado. Sin embargo, por cuestiones de activación, estos procesos en el estado Expulsado y todos aquellos en los estados Listo para Ejecutar en Memoria, forman una única cola.



~~La expulsión sólo puede ocurrir cuando un proceso está a punto de moverse de modo núcleo a modo usuario. Mientras los procesos ejecutan al modo núcleo, no pueden ser expulsados. Esto hace que los sistemas UNIX no sean apropiados para procesamiento en tiempo real. Se proporcionará una explicación de los requisitos para procesamiento de tiempo real en el Capítulo 10.~~

En UNIX existen dos procesos con un interés particular. El proceso 0 es un proceso especial que se crea cuando el sistema arranca; en realidad, es una estructura de datos predefinida que se carga en cuanto el ordenador arranca. Es el proceso *swapper*. Adicionalmente, el proceso 0 lanza al proceso 1, que se denomina proceso *init*; todos los procesos del sistema tienen al proceso 1 como antecesor. Cuando un nuevo usuario interactivo quiere entrar en el sistema, es el proceso 1 el que crea un proceso de usuario para él. Posteriormente, el proceso del usuario puede crear varios procesos hijos que conforman una estructura de árbol de procesos, de esta forma cualquier aplicación en particular puede consistir en un número de procesos relacionados entre sí.

Tabla 3.9. Estados de procesos en UNIX.

<b>Ejecutando Usuario</b>	Ejecutando en modo usuario.
<b>Ejecutando Núcleo</b>	Ejecutando en modo núcleo.
<b>Listo para Ejecutar, en Memoria</b>	Listo para ejecutar tan pronto como el núcleo lo planifique.
<b>Dormido en Memoria</b>	No puede ejecutar hasta que ocurra un evento; proceso en memoria principal (estado de bloqueo).
<b>Listo para Ejecutar, en Swap</b>	El proceso está listo para preguntar, pero el <i>swapper</i> debe cargar el proceso en memoria principal antes de que el núcleo pueda planificarlo para su ejecución.
<b>Durmiendo, en Swap</b>	El proceso está esperando un evento y ha sido expulsado al almacenamiento secundario (estado de bloqueo).
<b>Expulsado</b>	El proceso ha regresado de modo núcleo a modo usuario, pero el núcleo lo ha expulsado y ha realizado la activación de otro proceso.
<b>Creado</b>	El proceso ha sido creado recientemente y aún no está listo para ejecutar.
<b>Zombie</b>	El proceso ya no existe, pero deja un registro para que lo recoja su proceso padre.

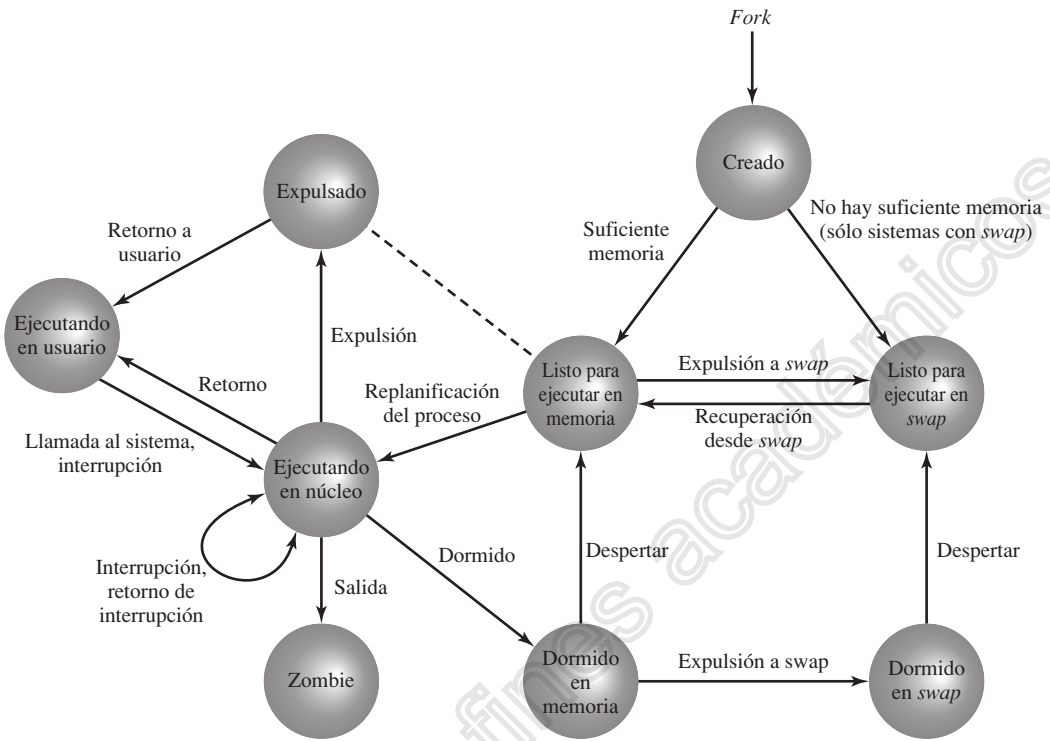


Figura 3.17. Diagrama de transiciones entre estados de procesos UNIX.

## DESCRIPCIÓN DE PROCESOS



Un proceso UNIX es un conjunto de estructuras de datos, más bien complejas, que proporcionan al sistema operativo toda la información necesaria para manejar y activar los procesos. La Tabla 3.10 recoge los elementos de la imagen de proceso, que están organizados en tres partes: contexto a nivel de usuario, contexto de registros, y contexto a nivel de sistema.

El **contexto a nivel de usuario** contiene los elementos básicos de un programa de usuario que se pueden generar directamente desde un fichero objeto compilado. Un programa de usuario se divide en áreas de texto y datos; el área texto es de sólo-lectura y se crea con la intención de contener las instrucciones del programa. Cuando el proceso está en ejecución, el procesador utiliza el área de pila de usuario para gestionar las llamadas a procedimientos y sus retornos, así como los parámetros pasados. El área de memoria compartida es un área de datos que se comparte con otros procesos. **Sólo** existe una única copia física del área de memoria compartida, pero, por medio de la utilización de la memoria virtual, se presenta a cada uno de los procesos que comparten esta región de memoria común dentro de su espacio de dirección. Cuando un proceso está ejecutando, la información de estado de procesador se almacena en el área de **contexto de registros**.

El **contexto a nivel de sistema** contiene la información restante que necesita el sistema operativo para manejar el proceso. Consiste en una parte estática, de tamaño fijo y que permanece como parte del proceso a lo largo de todo su tiempo de vida, y una parte dinámica, que varía de tamaño a lo largo de la vida del proceso. La entrada de la tabla de procesos es un elemento de la parte estática y contiene información de control del proceso que es accesible por parte del núcleo

en todo momento; además, en un sistema de memoria virtual, todas las entradas en la tabla de procesos se mantienen en memoria principal. La Tabla 3.11 muestra los contenidos de la entrada de la tabla de procesos. El área de usuario, o área U, contiene información adicional de proceso que necesita el núcleo cuando está ejecutando en el contexto de este proceso; también se utiliza cuando el proceso se pagina desde/hacia memoria (*swapping*). La Tabla 3.12 muestra los contenidos de dicha tabla.

Tabla 3.10. Imagen de un proceso UNIX.



Contexto a nivel de usuario	
Texto	Instrucciones máquina ejecutables del programa.
Datos	Datos accesibles por parte del programa asociado a dicho proceso.
Pila de usuario	Contiene los argumentos, las variables locales, y los punteros a funciones ejecutadas en modo usuario.
Memoria compartida	Memoria compartida con otros procesos, usada para la comunicación entre procesos.
Contexto de registros	
Contador de programa	Dirección de la siguiente instrucción a ejecutar; puede tratarse del espacio de memoria del núcleo o de usuario de dicho proceso.
Registro de estado del procesador	Contiene el estado del hardware del procesador en el momento de la expulsión; los contenidos y el formato dependen específicamente del propio hardware.
Puntero de pila	Apunta a la cima de la pila de núcleo o usuario, dependiendo del modo de operación en el momento de la expulsión.
Registros de propósito general	Depende del hardware.
Contexto nivel de sistema	
Entrada en la tabla de procesos	Define el estado del proceso; esta información siempre está accesible por parte de sistema operativo.
Área U (de usuario)	Información de control del proceso que sólo se necesita acceder en el contexto del propio proceso.
Tabla de regiones por proceso	Define la traducción entre las direcciones virtuales y físicas; también contiene información sobre los permisos que indican el tipo de acceso permitido por parte del proceso; sólo-lectura, lectura-escritura, o lectura-ejecución.
Pila del núcleo	Contiene el marco de pila de los procedimientos del núcleo cuando el proceso ejecuta en modo núcleo.

**Tabla 3.11.** Entrada en la tabla de procesos UNIX.

Estado del proceso	Estado actual del proceso.
Punteros	Al área U y al área de memoria del proceso (texto, datos, pila).
Tamaño de proceso	Permite al sistema operativo conocer cuánto espacio está reservado para este proceso.
Identificadores de usuario	El <b>ID de usuario real</b> identifica el usuario que es responsable de la ejecución del proceso. El <b>ID de usuario efectivo</b> se puede utilizar para que el proceso gane, de forma temporal, los privilegios asociados a un programa en particular; mientras ese programa se ejecuta como parte del proceso, el proceso opera con el identificador de usuario efectivo.
Identificadores de proceso	Identificador de este proceso; identificador del proceso padre. Estos identificadores se fijan cuando el proceso entra en el estado Creado después de la llamada al sistema <i>fork</i> .
Descriptor de evento	Válido cuando el proceso está en un estado dormido; cuando el evento ocurre, el proceso se mueve al estado Listo para Ejecutar.
Prioridad	Utilizado en la planificación del proceso.
Señal	Enumera las señales enviadas a este proceso pero que no han sido aún manejadas.
Temporizadores	Incluye el tiempo de ejecución del proceso, la utilización de recursos de núcleo, y el temporizador fijado por el usuario para enviar la señal de alarma al proceso.
<b>P_link</b>	Puntero al siguiente enlace en la cola de Listos (válido si proceso está Listo para Ejecutar).
Estado de memoria	Indica si la imagen del proceso se encuentra en memoria principal o secundaria. Si está en memoria, este campo indica si puede ser expulsado a <i>swap</i> o si está temporalmente fijado en memoria principal.

La distinción entre la entrada de la tabla de procesos y el área U refleja el hecho de que el núcleo de UNIX siempre ejecuta en el contexto de un proceso. La mayor parte del tiempo, el núcleo está realizando tareas relacionadas con dicho proceso. Sin embargo, otra parte del tiempo, como cuando el núcleo está realizando tareas de planificación preparatorias para la activación de otro proceso, se necesita la información sobre la totalidad de procesos del sistema. La información en la tabla de procesos es accesible cuando el proceso específico no es el que actualmente está en ejecución.

La tercera parte estática de contexto a nivel de sistema es una tabla de regiones por proceso, que se utiliza para el sistema de gestión de memoria. Por último, la pila del núcleo es una parte dinámica de contexto a nivel de sistema. Esta pila se usa cuando el proceso está ejecutando en modo núcleo y contiene la información que debe salvaguardarse y restaurarse en las llamadas a procedimientos o cuando ocurre una interrupción.

Tabla 3.12. Área U de UNIX.



Puntero a la tabla de proceso	Indica la entrada correspondiente a esta área U.
Identificador de usuario	Identificador de usuario real y efectivo. Utilizado para determinar los privilegios.
Temporizadores	Registro del tiempo que el proceso (y sus descendientes) han utilizado para ejecutar en modo usuario y modo núcleo.
Vector de manejadores de señales	Para cada tipo de señal definida en el sistema, se indica cómo el proceso va a reaccionar a la hora de recibirla (salir, ignorar, ejecutar una función específica definida por el usuario).
Terminal de control	Indica el terminal de acceso ( <i>login</i> ) para este proceso, si existe.
Campo de error	Recoge los errores encontrados durante una llamada al sistema.
Valor de retorno	Contiene los resultados de una llamada al sistema.
Parámetros de E/S	Indica la cantidad de datos transferidos, la dirección fuente (o destino) del vector de datos en el espacio de usuario, y los desplazamientos en fichero para la E/S.
Parámetros en fichero	Directorio actual y directorio raíz, dentro del sistema de ficheros, asociado al entorno de este proceso.
Tabla de descriptores de fichero de usuario	Recoge los ficheros del proceso abierto.
Campos límite	Restringe el tamaño del proceso y el tamaño máximo de fichero que puede crear.
Campos de los modos de permiso	Máscara de los modos de protección para la creación de ficheros por parte de este proceso.

## CONTROL DE PROCESOS

La creación de procesos en UNIX se realiza por medio de la llamada al sistema *fork()*. Cuando con un proceso solicita una llamada *fork*, el sistema operativo realiza las siguientes funciones [BACH86]:

1. Solicita la entrada en la tabla de procesos para el nuevo proceso.
2. Asigna un identificador de proceso único al proceso hijo.
3. Hace una copia de la imagen del proceso padre, con excepción de las regiones de memoria compartidas.
4. Incrementa el contador de cualquier fichero en posesión del padre, para reflejar el proceso adicional que ahora también posee dichos ficheros.
5. Asigna al proceso hijo el estado Listo para Ejecutar.
6. Devuelve el identificador del proceso hijo al proceso padre, y un valor 0 al proceso hijo.

Todo este trabajo se realiza en modo núcleo, dentro del proceso padre. Cuando el núcleo ha completado estas funciones puede realizar cualquiera de las siguientes acciones, como parte de la rutina del activador:



1. Continuar con el proceso padre. El control vuelve a modo usuario en el punto en el que se realizó la llamada *fork* por parte del padre.
2. Transferir el control al proceso hijo. El proceso hijo comienza ejecutar en el mismo punto del código del padre, es decir en el punto de retorno de la llamada *fork*.
3. Transferir el control a otro proceso. Ambos procesos, padre e hijo, permanecen en el estado Listos para Ejecutar.

Puede resultar quizá un poco difícil visualizar este modo de creación de procesos debido a que ambos procesos, padre e hijo, están ejecutando el mismo segmento de código. La diferencia reside en que: cuando se retorna de la función *fork*, el parámetro de retorno se comprueba. Si el valor es 0, entonces este es el proceso hijo, y se puede realizar una bifurcación en la ejecución del programa para continuar con la ejecución de programa hijo. Si el valor no es 0, éste es el proceso padre, y puede continuar con la línea principal ejecución.

### 3.6. RESUMEN

El concepto fundamental dentro de los sistemas operativos modernos es el concepto de proceso. La función principal de un sistema operativo es crear, gestionar y finalizar los procesos. Cuando un proceso está activo, el sistema operativo debe ver cómo reservar tiempo para su ejecución por parte del procesador, coordinar sus actividades, gestionar las demandas que planteen conflictos, y reservar recursos del sistema para estos procesos.

Para realizar estas funciones de gestión de procesos, el sistema operativo mantiene una descripción de cada proceso o imagen de proceso, que incluye el espacio de direcciones dentro del cual el proceso está ejecutando, y el bloque de control de proceso. Este último contiene toda la información que el sistema operativo necesita para gestionar el proceso, incluyendo el estado actual, la reserva de recursos, la prioridad y otros datos de relevancia.

Durante su tiempo de vida, un proceso se mueve a lo largo de diferentes estados. Los más importantes de estos estados son Listo, Ejecutando, y Bloqueado. Un proceso Listo es un proceso que no está actualmente en ejecución pero que está listo para ser ejecutado tan pronto el sistema operativo lo active. Un proceso que está Ejecutando es aquel que está actualmente en ejecución por el procesador. En los sistemas multiprocesador, podrá haber varios procesos en este estado. Un proceso Bloqueado está esperando a que se complete un determinado evento, como una operación de E/S.

Un proceso en ejecución se interrumpe bien por una interrupción, que es un evento que ocurre fuera de proceso y que es recogido por el procesador, o por la ejecución de una llamada al sistema. En cualquier caso, el procesador realiza un cambio de modo, que es la transferencia de control a unas rutinas del sistema operativo. El sistema operativo, después de haber realizado el trabajo necesario, puede continuar con el proceso interrumpido o puede cambiar a otros procesos.

### 3.7. LECTURAS RECOMENDADAS

Todos los libros de texto listados de la Sección 2.9 cubren el material de este capítulo. Se pueden encontrar en [GOOD94] y [GRAY97] unas buenas descripciones de la gestión de procesos en UNIX. [NEHM75] es una descripción interesante de los estados de proceso del sistema operativo y las funciones necesarias para la activación de procesos.

Este capítulo analiza algunos conceptos avanzados relativos a la gestión de procesos que se pueden encontrar en los sistemas operativos modernos. En primer lugar, se muestra cómo el concepto de proceso es más complejo y sutil de lo que se ha visto hasta este momento y, de hecho, contiene dos conceptos diferentes y potencialmente independientes: uno relativo a la propiedad de recursos y otro relativo a la ejecución. En muchos sistemas operativos esta distinción ha llevado al desarrollo de estructuras conocidas como hilos (*threads*). Después de analizar los hilos se pasa a ver el multiprocesamiento simétrico (*Symmetric Multiprocessing*, SMP). Con SMP el sistema operativo debe ser capaz de planificar simultáneamente diferentes procesos en múltiples procesadores. Por último, se introduce el concepto de micronúcleo, una forma útil de estructurar el sistema operativo para dar soporte al manejo de procesos y sus restantes tareas.

#### 4.1. PROCESOS E HILOS

Hasta este momento se ha presentado el concepto de un proceso como poseedor de dos características:

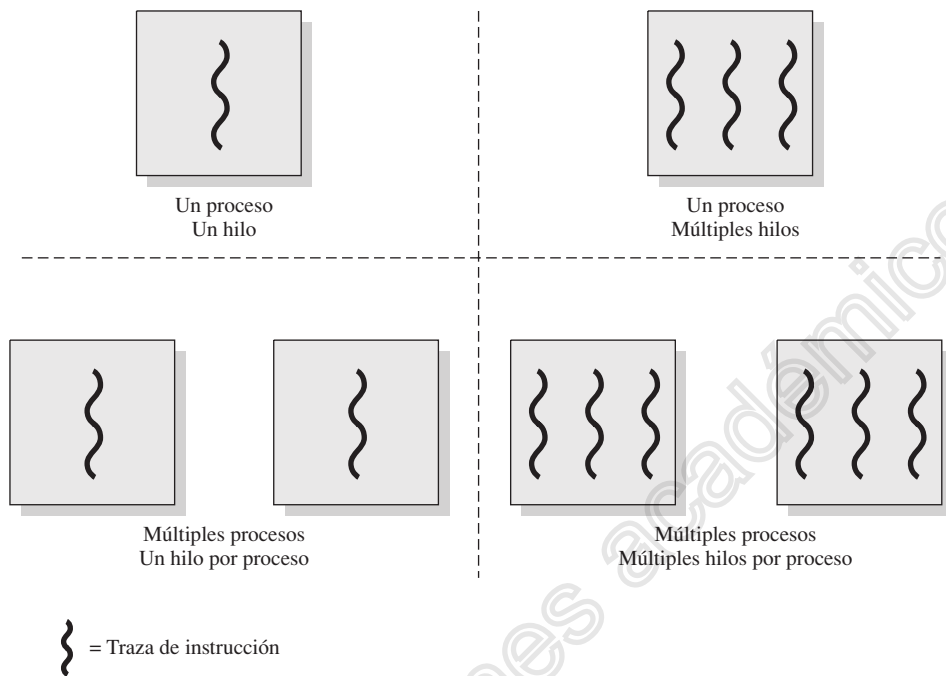
- **Propiedad de recursos.** Un proceso incluye un espacio de direcciones **virtuales** para el manejo de la imagen del proceso; como ya se explicó en el Capítulo 3 la imagen de un proceso es la colección de programa, datos, pila y atributos definidos en el bloque de control del proceso. De vez en cuando a un proceso se le puede asignar control o propiedad de recursos tales como la memoria principal, canales E/S, dispositivos E/S y archivos. El sistema operativo realiza la función de protección para evitar interferencias no deseadas entre procesos en relación con los recursos.
- **Planificación/ejecución.** La ejecución de un proceso sigue una ruta de ejecución (traza) a través de uno o más programas. Esta ejecución puede estar intercalada con ese u otros procesos. De esta manera, un proceso tiene un estado de ejecución (Ejecutando, Listo, etc.) y una prioridad de activación y ésta es la entidad que se planifica y activa por el sistema operativo.

En la mayor parte de los sistemas operativos tradicionales, estas dos características son, realmente, la esencia de un proceso. Sin embargo, debe quedar muy claro que estas dos características son independientes y podrían ser tratadas como tales por el sistema operativo. Así se hace en diversos sistemas operativos, sobre todo en los desarrollados recientemente. Para distinguir estas dos características, la unidad que se activa se suele denominar **hilo** (*thread*), o **proceso ligero**, mientras que la unidad de propiedad de recursos se suele denominar **proceso** o **tarea**<sup>1</sup>.

#### MULTIHILO

Multihilo se refiere a la capacidad de un sistema operativo de dar soporte a múltiples hilos de ejecución en un solo proceso. El enfoque tradicional de un solo hilo de ejecución por proceso, en el que no se identifica con el concepto de hilo, se conoce como estrategia monohilo. Las dos configuraciones que se muestran en la parte izquierda de la Figura 4.1 son estrategia monohilo. Un ejemplo de siste-

<sup>1</sup> Ni siquiera se puede mantener este grado de consistencia. En los sistemas operativos para *mainframe* de IBM, los conceptos de espacio de direcciones y tarea, respectivamente, más o menos se corresponden a los conceptos de proceso e hilo que se describen en esta sección. Además, en la literatura, el término *proceso ligero* se utiliza para (1) equivalente al término *hilo*, (2) un tipo particular de hilo conocido como hilo de nivel de núcleo, o (3) en el caso de Solaris, una entidad que asocia hilos de nivel de usuario con hilos de nivel de núcleo.



**Figura 4.1.** Hilos y procesos [ANDE97].

ma operativo que soporta un único proceso de usuario y un único hilo es el MS-DOS. Otros sistemas operativos, como algunas variedades de UNIX, soportan múltiples procesos de usuario, pero sólo un hilo por proceso. La parte derecha de la Figura 4.1 representa las estrategias multihilo. El entorno de ejecución de Java es un ejemplo de sistema con un único proceso y múltiples hilos. Lo interesante en esta sección es el uso de múltiples procesos, cada uno de los cuales soporta múltiples hilos. Este enfoque es el de Windows, Solaris, Mach, y OS/2 entre otros. En esta sección se ofrece una descripción general del mecanismo multihilo; más adelante en este capítulo se discuten los detalles de los enfoques de Windows, Solaris y Linux.

En un entorno multihilo, un proceso se define como la unidad de asignación de recursos y una unidad de protección. Se asocian con procesos los siguientes:

- Un espacio de direcciones virtuales que soporta la imagen del proceso.
- Acceso protegido a procesadores, otros procesos (para comunicación entre procesos), archivos y recursos de E/S (dispositivos y canales).

Dentro de un proceso puede haber uno o más hilos, cada uno con:

- Un estado de ejecución por hilo (Ejecutando, Listo, etc.).
- Un contexto de hilo que se almacena cuando no está en ejecución; una forma de ver a un hilo es como un contador de programa independiente dentro de un proceso.
- Una pila de ejecución.
- Por cada hilo, espacio de almacenamiento para variables locales.

- Acceso a la memoria y recursos de su proceso, compartido con todos los hilos de su mismo proceso.

La Figura 4.2 muestra la diferencia entre hilos y procesos desde el punto de vista de gestión de procesos. En un modelo de proceso monohilo (es decir, no existe el concepto de hilo), la representación de un proceso incluye su bloque de control de proceso y el espacio de direcciones de usuario, además de las pilas de usuario y núcleo para gestionar el comportamiento de las llamadas/retornos en la ejecución de los procesos. Mientras el proceso está ejecutando, los registros del procesador se controlan por ese proceso y, cuando el proceso no se está ejecutando, se almacena el contenido de estos registros. En un entorno multihilo, sigue habiendo un único bloque de control del proceso y un espacio de direcciones de usuario asociado al proceso, pero ahora hay varias pilas separadas para cada hilo, así como un bloque de control para cada hilo que contiene los valores de los registros, la prioridad, y otra información relativa al estado del hilo.

De esta forma, todos los hilos de un proceso comparten el estado y los recursos de ese proceso, residen en el mismo espacio de direcciones y tienen acceso a los mismos datos. Cuando un hilo cambia determinados datos en memoria, otros hilos ven los resultados cuando acceden a estos datos. Si un hilo abre un archivo con permisos de lectura, los demás hilos del mismo proceso pueden también leer ese archivo.

Los mayores beneficios de los hilos provienen de las consecuencias del rendimiento:

1. Lleva mucho menos tiempo crear un nuevo hilo en un proceso existente que crear un proceso totalmente nuevo. Los estudios realizados por los que desarrollaron el sistema operativo Mach muestran que la creación de un hilo es diez veces más rápida que la creación de un proceso en UNIX [TEVA87].
2. Lleva menos tiempo finalizar un hilo que un proceso.
3. Lleva menos tiempo cambiar entre dos hilos dentro del mismo proceso.
4. Los hilos mejoran la eficiencia de la comunicación entre diferentes programas que están ejecutando. En la mayor parte de los sistemas operativos, la comunicación entre procesos inde-

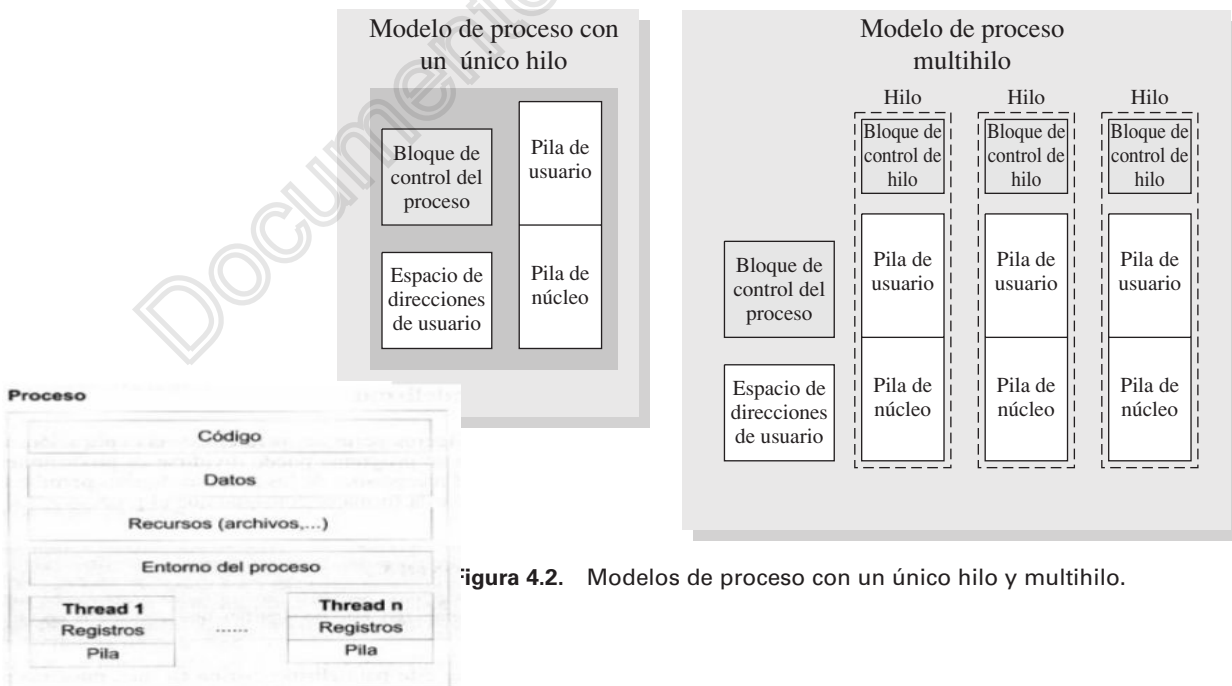


Figura 4.2. Modelos de proceso con un único hilo y multihilo.

pendientes requiere la intervención del núcleo para proporcionar protección y los mecanismos necesarios de comunicación. Sin embargo, ya que los hilos dentro de un mismo proceso comparten memoria y archivos, se pueden comunicar entre ellos sin necesidad de invocar al núcleo.

De esta forma, si se desea implementar una aplicación o función como un conjunto de unidades de ejecución relacionadas, es mucho más eficiente hacerlo con un conjunto de hilos que con un conjunto de procesos independientes.

Un ejemplo de una aplicación que podría hacer uso de hilos es un servidor de archivos. Cada vez que llega una nueva petición de archivo, el programa de gestión de archivos puede ejecutar un nuevo hilo. Ya que un servidor manejará muchas peticiones, se crearán y finalizarán muchos hilos en un corto periodo de tiempo. Si el servidor ejecuta en una máquina multiprocesador, pueden estar ejecutando simultáneamente múltiples hilos del mismo proceso en diferentes procesadores. Además, ya que los procesos o los hilos en un servidor de archivos deben compartir archivos de datos y, por tanto, coordinar sus acciones, es más rápido usar hilos y memoria compartida que usar procesos y paso de mensajes para esta coordinación.

A veces los hilos son también útiles en un solo procesador ya que ayudan a simplificar la estructura de programas que realizan varias funciones diferentes.

[LETW88] ofrece cuatro ejemplos de uso de hilos en un sistema de multiprocesamiento de un solo usuario:

- **Trabajo en primer plano y en segundo plano.** Por ejemplo, en un programa de hoja de cálculo, un hilo podría mostrar menús y leer la entrada de usuario, mientras otro hilo ejecuta los mandatos de usuario y actualiza la hoja de cálculo. Esta forma de trabajo a menudo incrementa la velocidad que se percibe de la aplicación, permitiendo al programa solicitar el siguiente mandato antes de que el mandato anterior esté completado.
- **Procesamiento asíncrono.** Los elementos asíncronos de un programa se pueden implementar como hilos. Por ejemplo, se puede diseñar un procesador de textos con protección contra un fallo de corriente que escriba el *buffer* de su memoria RAM a disco una vez por minuto. Se puede crear un hilo cuyo único trabajo sea crear una copia de seguridad periódicamente y que se planifique directamente a través del sistema operativo; no se necesita código adicional en el programa principal que proporcione control de tiempo o que coordine la entrada/salida.
- **Velocidad de ejecución.** Un proceso multihilo puede computar una serie de datos mientras que lee los siguientes de un dispositivo. En un sistema multiprocesador pueden estar ejecutando simultáneamente múltiples hilos de un mismo proceso. De esta forma, aunque un hilo pueda estar bloqueado por una operación de E/S mientras lee datos, otro hilo puede estar ejecutando.
- **Estructura modular de programas.** Los programas que realizan diversas tareas o que tienen varias fuentes y destinos de entrada y salida, se pueden diseñar e implementar más fácilmente usando hilos.

En un sistema operativo que soporte hilos, la planificación y la activación se realizan a nivel de hilo; de aquí que la mayor parte de la información de estado relativa a la ejecución se mantenga en estructuras de datos a nivel de hilo. Existen, sin embargo, diversas acciones que afectan a todos los hilos de un proceso y que el sistema operativo debe gestionar a nivel de proceso. Suspender un proceso implica expulsar el espacio de direcciones de un proceso de memoria principal para dejar hueco a otro espacio de direcciones de otro proceso. Ya que todos los hilos de un proceso comparten el mismo

espacio de direcciones, todos los hilos se suspenden al mismo tiempo. De forma similar, la finalización de un proceso finaliza todos los hilos de ese proceso.

## FUNCIONALIDADES DE LOS HILOS

Los hilos, al igual que los procesos, tienen estados de ejecución y se pueden sincronizar entre ellos. A continuación se analizan estos dos aspectos de las funcionalidades de los hilos.

**Estados de los hilos.** Igual que con los procesos, los principales estados de los hilos son: Ejecutando, Listo y Bloqueado. Generalmente, no tiene sentido aplicar estados de suspensión a un hilo, ya que dichos estados son conceptos de nivel de proceso. En particular, si se expulsa un proceso, todos sus hilos se deben expulsar porque comparten el espacio de direcciones del proceso.

Hay cuatro operaciones básicas relacionadas con los hilos que están asociadas con un cambio de estado del hilo [ANDE97]:

- **Creación.** Cuando se crea un nuevo proceso, también se crea un hilo de dicho proceso. Posteriormente, un hilo del proceso puede crear otro hilo dentro del mismo proceso, proporcionando un puntero a las instrucciones y los argumentos para el nuevo hilo. Al nuevo hilo se le proporciona su propio registro de contexto y espacio de pila y se coloca en la cola de Listos.
- **Bloqueo.** Cuando un hilo necesita esperar por un evento se bloquea, almacenando los registros de usuario, contador de programa y punteros de pila. El procesador puede pasar a ejecutar otro hilo en estado Listo, dentro del mismo proceso o en otro diferente.
- **Desbloqueo.** Cuando sucede el evento por el que el hilo está bloqueado, el hilo se pasa a la cola de Listos.
- **Finalización.** Cuando se completa un hilo, se liberan su registro de contexto y pilas.

Un aspecto importante es si el bloqueo de un hilo implica el bloqueo del proceso completo. En otras palabras, si se bloquea un hilo de un proceso, ¿esto impide la ejecución de otro hilo del mismo proceso incluso si el otro hilo está en estado de Listo? Sin lugar a dudas, se pierde algo de la potencia y flexibilidad de los hilos si el hilo bloqueado bloquea al proceso entero.

Volveremos a este tema a continuación cuando veamos los hilos a nivel de usuario y a nivel de núcleo, pero por el momento consideraremos los beneficios de rendimiento de los hilos que no bloquean al proceso completo. La Figura 4.3 (basada en una de [KLEI96]) muestra un programa que realiza dos llamadas a procedimiento remoto (RPC)<sup>2</sup> a dos máquinas diferentes para poder combinar los resultados. En un programa de un solo hilo, los resultados se obtienen en secuencia, por lo que el programa tiene que esperar a la respuesta de cada servidor por turnos. Reescribir el programa para utilizar un hilo diferente para cada RPC, mejora sustancialmente la velocidad. Observar que si el programa ejecuta en un uniprocador, las peticiones se deben generar en secuencia y los resultados se deben procesar en secuencia; sin embargo, el programa espera concurrentemente las dos respuestas.

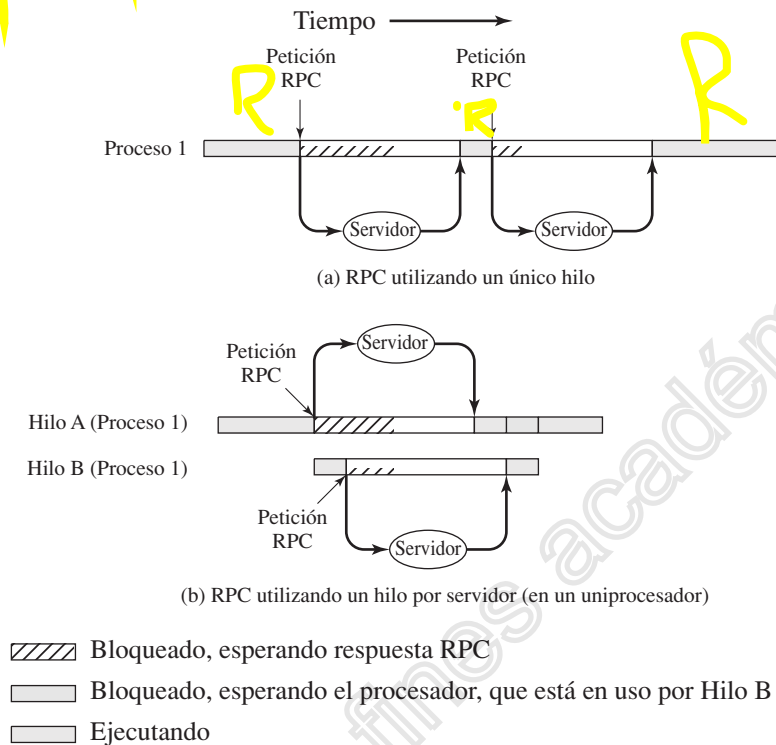
En un uniprocador, la multiprogramación permite el intercalado de múltiples hilos con múltiples procesos. En el ejemplo de la Figura 4.4, se intercalan tres hilos de dos procesos en un procesa-

---

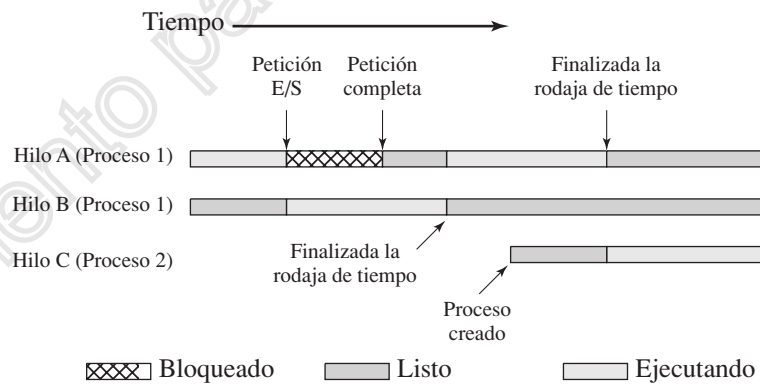
<sup>2</sup> RPC es una técnica por la que dos programas, que pueden ejecutar en diferentes máquinas, interactúan utilizando la sintaxis y la semántica de las llamadas a procedimiento. Tanto el programa llamante como el llamado se comportan como si el otro programa estuviera ejecutando en la misma máquina. Los RPC se suelen utilizar en las aplicaciones cliente/servidor y se analizan en el Capítulo 13.



R. Fun



**Figura 4.3.** Llamadas a Procedimiento Remoto (RPC) utilizando hilos.



**Figura 4.4.** Ejemplo multihilo en un uniprocador.

dor. La ejecución pasa de un hilo a otro cuando se bloquea el hilo actualmente en ejecución o su porción de tiempo se agota<sup>3</sup>.

<sup>3</sup> En este ejemplo, el hilo C comienza a ejecutar después de que el hilo A finaliza su rodaja de tiempo, aunque el hilo B esté también listo para ejecutar. La elección entre B y C es una decisión de planificación, un tema que se aborda en la Parte Cuatro.

## Hilos a nivel de usuario y a nivel de núcleo

Existen dos amplias categorías de implementación de hilos: hilos de nivel de usuario (*user-level threads*, ULT) e hilos de nivel de núcleo (*kernel-level threads*, KLT). Los últimos son también conocidos en la literatura como hilos soportados por el núcleo (*kernel-supported threads*) o procesos ligeros (*lightweight processes*).

### Hilos a nivel de usuario

En un entorno ULT, la aplicación o programa de usuario gestiona todo el trabajo de los hilos y el núcleo no es consciente de la existencia de los mismos (un hilo a nivel de usuario mantiene todo su estado en el espacio de usuario). Como consecuencia de ello, el hilo no utiliza recursos del kernel para su gestión, y se puede conmutar entre hilos sin cambiar el espacio de direcciones. Principalmente se utilizaban en las aplicaciones que se ejecutaban en sistemas operativos que no podían planificar hilos. La Figura 4.6a muestra el enfoque:

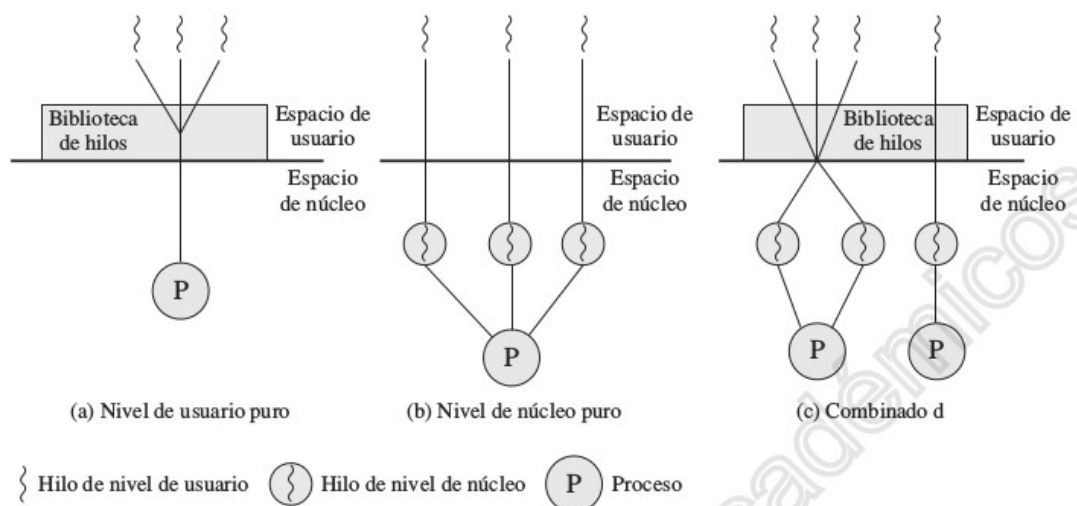


Figura 4.6. Hilos de nivel de usuario y de nivel de núcleo.

Cualquier aplicación puede programarse para ser multihilo a través del uso de una biblioteca de hilos, que es un paquete de rutinas para la gestión de ULT (e.j threads de Mach, las cuales ofrecen las funciones para la creación, sincronización y planificación de hilos). La biblioteca de hilos contiene código para la creación y destrucción de hilos, para paso de mensajes y datos entre los hilos, para planificar la ejecución de los hilos, y para guardar y restaurar el contexto de los hilos.

Por defecto, una aplicación comienza con un solo hilo y ejecutando en ese hilo. Esta aplicación y su hilo se localizan en un solo proceso gestionado por el núcleo. En cualquier momento que la aplicación esté ejecutando (el proceso está en estado Ejecutando), la aplicación puede crear un nuevo hilo a ejecutar dentro del mismo proceso. La creación se realiza llamando a la utilidad de creación en la biblioteca de hilos que se esté usando, y es el programa a nivel de aplicación (en el espacio de usuario) el que gestiona la planificación de cada hilo. Toda esto tiene lugar dentro de un solo proceso, el núcleo no es consciente de esta actividad, los hilos son invisibles para él. El núcleo continúa planificando el proceso como una unidad y asigna al proceso un único estado (Listo, Ejecutando, Bloqueado, etc.).

El uso de ULT en lugar de KLT, presenta las siguientes **ventajas**:

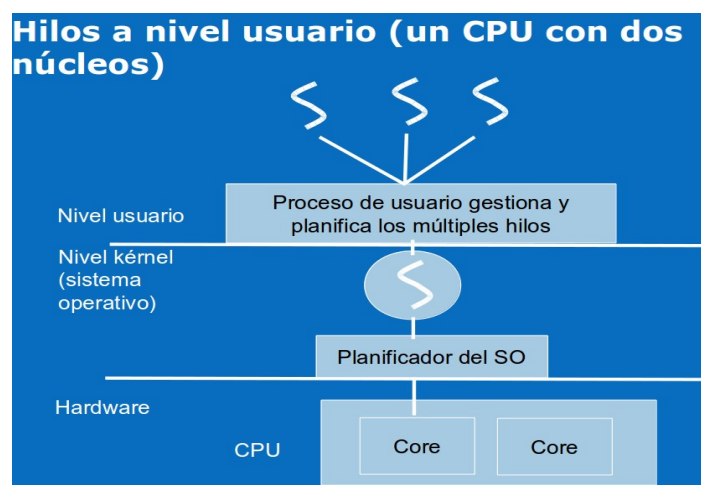
1. Un cambio de hilo por parte de la aplicación no requiere privilegios de modo núcleo porque todas

las estructuras de datos de gestión de hilos están en el espacio de direcciones de usuario de un solo proceso. Por consiguiente, el proceso principal no necesita cambiar a modo núcleo y ejecutar un *dispatcher* para realizar la gestión de los hilos, esto ahorra tiempo de cómputo (usuario a núcleo; núcleo a usuario). Es decir, cuando se acabe la ejecución de un hilo, el procesador seguirá ejecutando el proceso (para núcleo todavía no ha terminado aunque de ese hilo en concreto si se haya ejecutado su última instrucción), pero será la aplicación la que asigne qué hilo del proceso será el siguiente a mandar al procesador. Es local al proceso donde se guarda la información de contexto de cada hilo. Entre otras cuestiones, no se necesita un *trap* ni una conmutación de contexto (ya que está cargado el contexto del proceso principal, que es el que ve el núcleo), la memoria caché no necesita vaciarse, etc. Esto hace que la planificación de hilos sea muy rápida, pudiendo crear una aplicación con varios hilos en vez de varios procesos que conllevarían el uso del núcleo.

2. Los ULT pueden ejecutar en cualquier sistema operativo, acepte o no hilos. No se necesita ningún cambio en el nuevo núcleo para dar soporte a los ULT. La biblioteca de los hilos es un conjunto de utilidades a nivel de aplicación que pueden usar todas las aplicaciones del sistema.

Hay dos **desventajas** de los ULT en comparación con los KLT:

1. Los hilos a nivel de usuario no se pueden ejecutar mientras el kernel está ocupado, por ejemplo, con paginación o E/S. En un sistema operativo típico muchas llamadas al sistema son bloqueantes. De forma genérica, cuando un proceso realiza una operación de E/S bloqueadora, como puede ser una lectura del teclado, éste pasa a estado bloqueado mientras se espera esa entrada y el sistema operativo pasa a ejecutar otro proceso. Si esto lo aplicamos a hilos, cuando un ULT realiza una llamada al sistema bloqueante, no sólo se bloquea ese hilo, sino que se bloquean todos los hilos del proceso.



2. En una estrategia ULT, una aplicación multihilo no puede sacar ventaja del multiproceso (tienen que ceder la CPU entre ellos, habiendo conmutaciones en el mismo proceso). El núcleo asigna un proceso a un solo procesador al mismo tiempo. Por consiguiente, en un determinado momento sólo puede ejecutar un hilo de los que tenga el proceso.

Hay formas de afrontar estos dos problemas.

a) Ambos problemas pueden superarse escribiendo una aplicación de múltiples procesos en lugar de múltiples hilos. Pero este enfoque elimina la principal ventaja de los hilos: cada cambio es un cambio de proceso en lugar de un cambio de hilo, lo que genera una gran sobrecarga.

b) Otra forma de solucionar el problema de hilos que se bloquean es una técnica denominada jacketing (revestimiento o envoltura). El objetivo de esta técnica es convertir una llamada al sistema que podría ser bloqueante en una llamada al sistema no bloqueante. Por ejemplo, en

lugar de llamar directamente a una rutina del sistema de E/S, un hilo puede llamar a una rutina jacket de E/S a nivel de aplicación. Con esta rutina jacket, el código verifica si el dispositivo de E/S está ocupado. Si lo está, el hilo entra en estado Bloqueado y pasa el control (a través de la biblioteca de hilos) a otro hilo, con lo que el núcleo no lo percibe y sigue la ejecución por donde la aplicación a nivel de usuario decida. Cuando este hilo recupera de nuevo el control porque el núcleo ha terminado de ejecutar otro hilo del proceso, chequea de nuevo el dispositivo de E/S y así sucesivamente. Este método requiere que se reescriban partes de la biblioteca de llamadas al sistema operativo, es ineficiente y nada elegante, pero hay muy poca opción.

Ejemplo de sistemas que usan este tipo de hilos: Primeras versiones de Solaris y UNIX,

Algunos ejemplos de bibliotecas ULT son: Solaris Green Threads, GNU Portable Threads<sup>1</sup> (antiguos hilos basados en POSIX), FSU Threads, Apple Computer Thread Manager, REAL basic's cooperative threads.

### **Hilos a nivel de núcleo**

En un entorno KLT, el núcleo gestiona todo el trabajo de gestión de hilos. No hay código de gestión de hilos en la aplicación (planificación), solamente una interfaz de programación de aplicación (API) para acceder a las utilidades de hilos del núcleo (creación, terminación, comunicación entre hilos).

La Figura 4.6b representa el entorno KLT puro. Cualquier aplicación puede programarse para ser multihilo. Todos los hilos de una aplicación se mantienen en un solo proceso. El núcleo mantiene información de contexto del proceso como una entidad y de los hilos individuales del proceso. La planificación realizada por el núcleo se realiza a nivel de hilo. Este enfoque resuelve los dos principales inconvenientes del enfoque ULT. Primero, el núcleo puede planificar simultáneamente múltiples hilos de un solo proceso en múltiples procesadores (o múltiples núcleos). Segundo, cuando un hilo se bloquea, el kernel, según lo que decida, puede ejecutar otro hilo del mismo proceso (si hay uno listo) o un hilo de un proceso distinto. Con los hilos de nivel usuario (ULT), el sistema en tiempo de ejecución ejecuta hilos de su propio proceso hasta que el kernel le quita la CPU (o cuando ya no hay hilos para ejecutar).

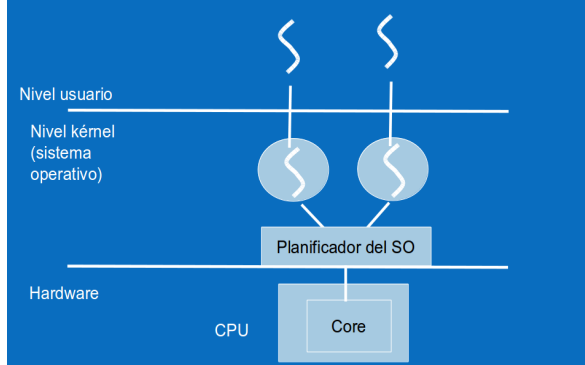
La principal desventaja del enfoque KLT en comparación con el enfoque ULT es que la transferencia de control de un hilo a otro del mismo proceso requiere un cambio de modo al núcleo, así como las operaciones de creación y terminación de hilos, con lo que se incurrirá en una mayor sobrecarga. Como ya sabemos, cuando el sistema operativo decide que la rodaja de tiempo de un proceso ha terminado o tiene que bloquearlo por una interrupción, hay que buscar otro proceso de la lista de Listos, y esto supone guardar el contexto actual y cargar otro nuevo contexto de proceso a partir del **dispatcher**, que se ejecuta en modo núcleo.

Ejemplos de sistemas operativos que soportan hilos a nivel de núcleo (recuerde que lo hilos a nivel de usuario teóricamente se puede soportar por cualquier sistema operativo): Windows, Linux, OS/2, Mac OS, Solaris. Algunas bibliotecas que los implementan son: Light Weight Kernel Threads, M:N threading, Native POSIX Thread Library para Linux (*glibc*), Mach C-threads, Win32 threads.

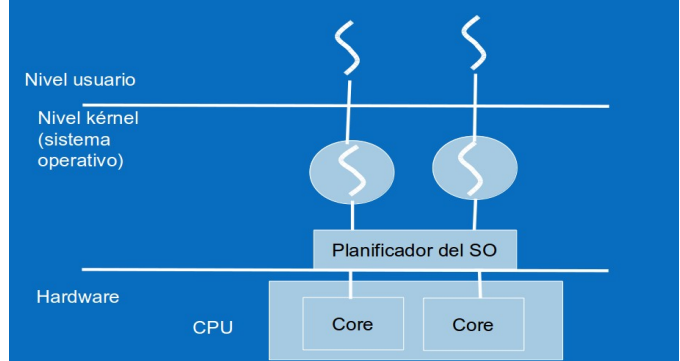
---

1 [https://www.gnu.org/software/pth/pth-manual.html#threading\\_background](https://www.gnu.org/software/pth/pth-manual.html#threading_background)

### Hilos a nivel kernel (con un CPU con un solo núcleo)



### Hilos a nivel kernel (con un CPU con doble núcleo)



## Enfoques combinados

Algunos sistemas operativos proporcionan utilidades combinadas ULT/KLT (Figura 4.6c). En un sistema combinado, la creación de hilos se realiza por completo en el espacio de usuario, como la mayor parte de la planificación y sincronización de hilos dentro de una aplicación. Los múltiples ULT de una aplicación se asocian en un número (menor o igual) de KLT. El programador debe ajustar el número de KLT para una máquina y aplicación en particular para lograr los mejores resultados posibles.

En los enfoques combinados, múltiples hilos de la misma aplicación pueden ejecutar en paralelo en múltiples procesadores, y una llamada al sistema bloqueante no necesita bloquear el proceso completo. Si el sistema está bien diseñado, este enfoque debería combinar las ventajas de los enfoques puros ULT y KLT, minimizando las desventajas.

Ejemplos de sistemas operativos que soportan hilos a nivel de núcleo: Solaris 2.x.

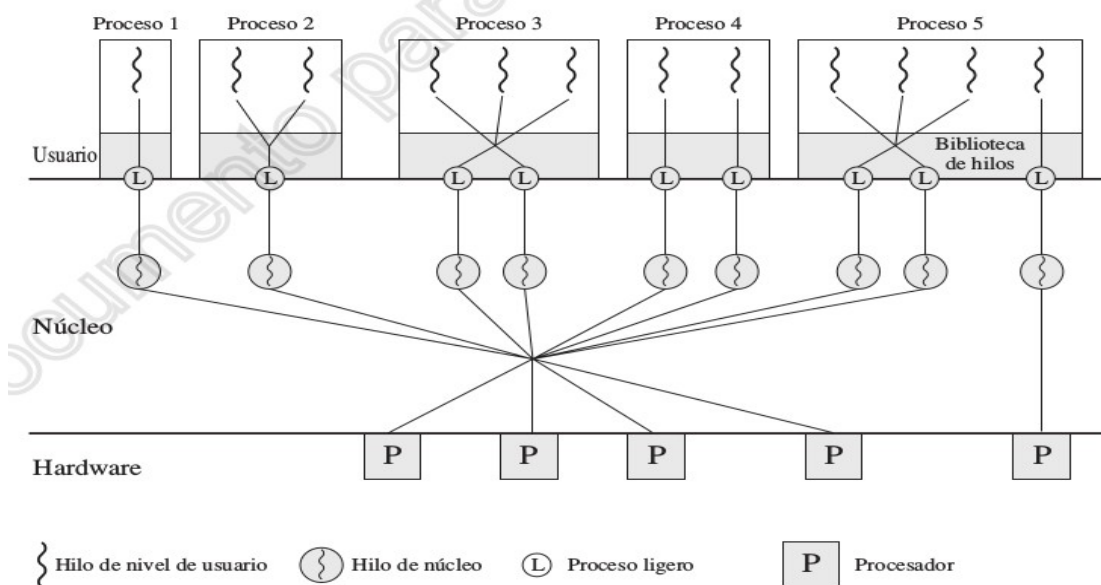


Figura 4.15. Ejemplo de arquitectura multihilo de Solaris.

~~El enfoque de Clouds proporciona una forma eficiente de aislar a los usuarios y programadores de los detalles del entorno distribuido. La actividad del usuario puede ser representada como un solo hilo, y el movimiento de ese hilo entre máquinas puede ser gestionado por el sistema operativo gracias a información relativa al sistema, tal como la necesidad de acceder a un recurso remoto o el equilibrio de carga.~~

## 4.2. MULTIPROCESAMIENTO SIMÉTRICO

Tradicionalmente, el computador ha sido visto como una máquina secuencial. La mayor parte de los lenguajes de programación requieren que el programador especifique algoritmos como una secuencia de instrucciones. Un procesador ejecuta programas a través de la ejecución de instrucciones máquina en secuencia y de una en una. Cada instrucción se ejecuta como una secuencia de operaciones (ir a buscar la instrucción, ir a buscar los operandos, realizar la operación, almacenar resultados).

~~Esta visión del computador nunca ha sido totalmente cierta. A nivel de micro-operación, se generan múltiples señales de control al mismo tiempo. El pipeline de instrucciones, al menos en lo relativo a la búsqueda y ejecución de operaciones, ha estado presente durante mucho tiempo. Éstos son dos ejemplos de realización de funciones en paralelo.~~

A medida que ha evolucionado la tecnología de los computadores y el coste del hardware ha descendido, los diseñadores han visto cada vez más oportunidades para el paralelismo, normalmente para mejorar el rendimiento y, en algunos casos, para mejorar la fiabilidad. En este libro, examinamos los dos enfoques más populares para proporcionar paralelismo a través de la réplica de procesadores: multiprocesamiento simétricos (SMP) y *clusters*. Los SMP se abordan en esta sección; los *clusters* se examinan en la Parte Seis.

### ARQUITECTURA SMP



Es útil ver donde encaja la arquitectura SMP dentro de las categorías de procesamiento paralelo. La forma más común de categorizar estos sistemas es la taxonomía de sistemas de procesamiento paralelo introducida por Flynn [FLYN72]. Flynn propone las siguientes categorías de sistemas de computadores:

- **Única instrucción, único flujo de datos – *Single instruction single data (SISD) stream*.** Un solo procesador ejecuta una única instrucción que opera sobre datos almacenados en una sola memoria.
- **Única instrucción, múltiples flujos de datos – *Single instruction multiple data (SIMD) stream*.** Una única instrucción de máquina controla la ejecución simultánea de un número de elementos de proceso. Cada elemento de proceso tiene una memoria de datos asociada, de forma que cada instrucción se ejecuta en un conjunto de datos diferente a través de los diferentes procesadores. Los procesadores vectoriales y matriciales entran dentro de esta categoría.
- **Múltiples instrucciones, único flujo de datos – *Multiple instruction single data (MISD) stream*.** Se transmite una secuencia de datos a un conjunto de procesadores, cada uno de los cuales ejecuta una secuencia de instrucciones diferente. Esta estructura nunca se ha implementado.
- **Múltiples instrucciones, múltiples flujos de datos – *Multiple instruction multiple data (MIMD) stream*.** Un conjunto de procesadores ejecuta simultáneamente diferentes secuencias de instrucciones en diferentes conjuntos de datos.

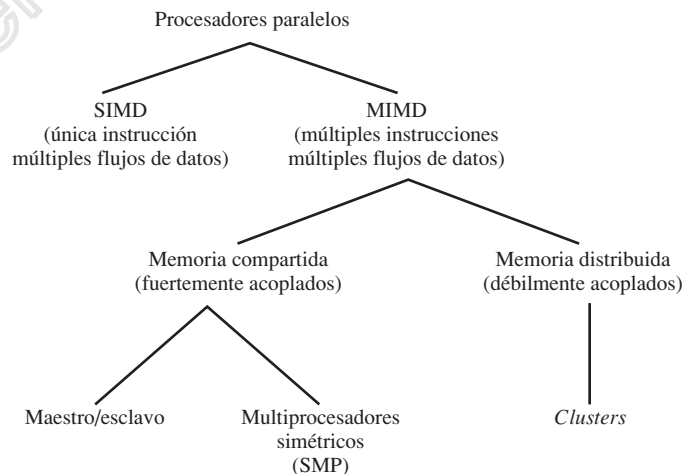


Con la organización MIMD, los procesadores son de propósito general, porque deben ser capaces de procesar todas las instrucciones necesarias para realizar las transformaciones de datos apropiadas. MIMD se puede subdividir por la forma en que se comunican los procesadores (Figura 4.8). Si cada procesador tiene una memoria dedicada, cada elemento de proceso es en sí un computador. La comunicación entre los computadores se puede realizar a través de rutas prefijadas o bien a través de redes. Este sistema es conocido como un **cluster**, o multicomputador. Si los procesadores comparten una memoria común, entonces cada procesador accede a los programas y datos almacenados en la memoria compartida, y los procesadores se comunican entre sí a través de dicha memoria; este sistema se conoce como **multiprocesador de memoria compartida**.

Una clasificación general de los multiprocesadores de memoria compartida se basa en la forma de asignar procesos a los procesadores. Los dos enfoques fundamentales son maestro/esclavo y simétrico. Con la arquitectura **maestro/esclavo**, el núcleo del sistema operativo siempre ejecuta en un determinado procesador. El resto de los procesadores sólo podrán ejecutar programas de usuario y, a lo mejor, utilidades del sistema operativo. El maestro es responsable de la planificación de procesos e hilos. Una vez que un proceso/hilo está activado, si el esclavo necesita servicios (por ejemplo, una llamada de E/S), debe enviar una petición al maestro y esperar a que se realice el servicio. Este enfoque es bastante sencillo y requiere pocas mejoras respecto a un sistema operativo multiprogramado uniprocador. La resolución de conflictos se simplifica porque un procesador tiene el control de toda la memoria y recursos de E/S. Las desventajas de este enfoque son las siguientes:

- Un fallo en el maestro echa abajo todo el sistema.
- El maestro puede convertirse en un cuello de botella desde el punto de vista del rendimiento, ya que es el único responsable de hacer toda la planificación y gestión de procesos.

En un **multiprocesador simétrico (Symmetric Multiprocessor, SMP)**, el núcleo puede ejecutar en cualquier procesador, y normalmente cada procesador realiza su propia planificación del conjunto disponible de procesos e hilos. El núcleo puede construirse como múltiples procesos o múltiples hilos, permitiéndose la ejecución de partes del núcleo en paralelo. El enfoque SMP complica al sistema operativo, ya que debe asegurar que dos procesadores no seleccionan un mismo proceso y que no se pierde ningún proceso de la cola. Se deben emplear técnicas para resolver y sincronizar el uso de los recursos.



**Figura 4.8.** Arquitectura de procesadores paralelos.

El diseño de SMP y *clusters* es complejo, e involucra temas relativos a la organización física, estructuras de interconexión, comunicación entre procesadores, diseño del sistema operativo y técnicas de aplicaciones software. Nuestra preocupación aquí, y más adelante cuando hablemos de los *clústers* (Capítulo 13), se centra en los aspectos de diseño del sistema operativo, aunque en ambos casos veremos brevemente temas de organización.

## ORGANIZACIÓN SMP

La Figura 4.9 muestra la organización general de un SMP. Existen múltiples procesadores, cada uno de los cuales contiene su propia unidad de control, unidad aritmético-lógica y registros. Cada procesador tiene acceso a una memoria principal compartida y dispositivos de E/S a través de algún mecanismo de interconexión; el bus compartido es común a todos los procesadores. Los procesadores se pueden comunicar entre sí a través de la memoria (mensajes e información de estado dejados en espacios de memoria compartidos). Los procesadores han de poder intercambiarse señales directamente. A menudo la memoria está organizada de tal manera que se pueden realizar múltiples accesos simultáneos a bloques separados.

En máquinas modernas, los procesadores suelen tener al menos un nivel de memoria cache, que es privada para el procesador. El uso de esta cache introduce nuevas consideraciones de diseño. Debido a que la cache local contiene la imagen de una porción de memoria principal, si se altera una palabra en una cache, se podría invalidar una palabra en el resto de las caches. Para prevenir esto, el resto de los procesadores deben ser alertados de que se ha llevado a cabo una actualización. Este problema se conoce como el problema de coherencia de caches y se suele solucionar con técnicas hardware más que con el sistema operativo<sup>6</sup>.

## CONSIDERACIONES DE DISEÑO DE SISTEMAS OPERATIVOS MULTIPROCESADOR

Un sistema operativo SMP gestiona los procesadores y otros recursos del computador, de manera que el usuario puede ver al sistema de la misma forma que si fuera un sistema uniprocador multiprogramado. Un usuario puede desarrollar aplicaciones que utilicen múltiples procesos o múltiples hilos dentro de procesos sin preocuparse de si estará disponible un único procesador o múltiples procesadores. De esta forma, un sistema operativo multiprocador debe proporcionar toda la funcionalidad de un sistema multiprogramado, además de características adicionales para adecuarse a múltiples procesadores. Las principales claves de diseño incluyen las siguientes características:

- **Procesos o hilos simultáneos concurrentes.** Las rutinas del núcleo necesitan ser reentrantes para permitir que varios procesadores ejecuten el mismo código del núcleo simultáneamente. Debido a que múltiples procesadores pueden ejecutar la misma o diferentes partes del código del núcleo, las tablas y la gestión de las estructuras del núcleo deben ser gestionadas apropiadamente para impedir interbloqueos u operaciones inválidas.
- **Planificación.** La planificación se puede realizar por cualquier procesador, por lo que se deben evitar los conflictos. Si se utiliza multihilo a nivel de núcleo, existe la posibilidad de planificar múltiples hilos del mismo proceso simultáneamente en múltiples procesadores. En el Capítulo 10 se examina la planificación multiprocador.

---

<sup>6</sup> En [STAL03] se proporciona una descripción de esquemas de coherencia de cache basados en el hardware.

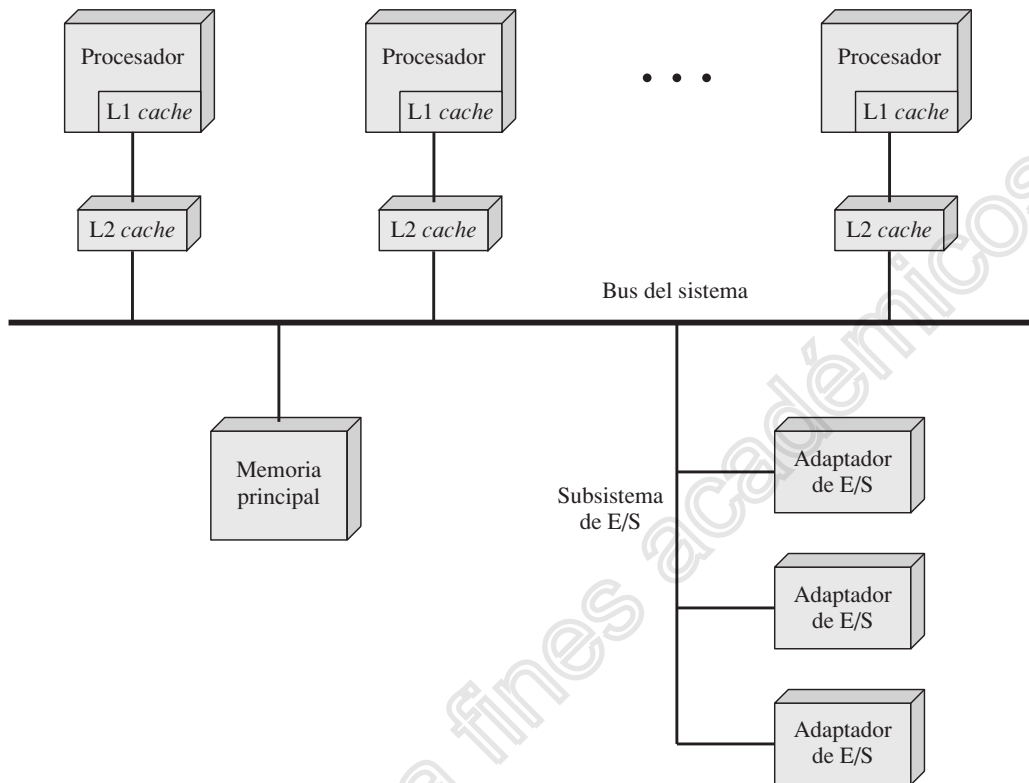


Figura 4.9. Organización de los multiprocesadores simétricos.

- **Sincronización.** Con múltiples procesos activos, que pueden acceder a espacios de direcciones compartidas o recursos compartidos de E/S, se debe tener cuidado en proporcionar una sincronización eficaz. La sincronización es un servicio que fuerza la exclusión mutua y el orden de los eventos. Un mecanismo común de sincronización que se utiliza en los sistemas operativos multiprocesador son los cerrojos, descritos en el Capítulo 5.
- **Gestión de memoria.** La gestión de memoria en un multiprocesador debe tratar con todos los aspectos encontrados en las máquinas uniprocador, que se verán en la Parte Tres. Además, el sistema operativo necesita explotar el paralelismo hardware existente, como las memorias multipuerto, para lograr el mejor rendimiento. Los mecanismos de paginación de los diferentes procesadores deben estar coordinados para asegurar la consistencia cuando varios procesadores comparten una página o segmento y para decidir sobre el reemplazo de una página.
- **Fiabilidad y tolerancia a fallos.** El sistema operativo no se debe degradar en caso de fallo de un procesador. El planificador y otras partes del sistema operativo deben darse cuenta de la pérdida de un procesador y reestructurar las tablas de gestión apropiadamente.

~~Debido a que los aspectos de diseño de un sistema operativo multiprocesador suelen ser extensiones a soluciones de problemas de diseño de uniprocadores multiprogramados, no los trataremos por separado. En su lugar, los aspectos específicos a los temas multiprocesador se tratarán en su contexto apropiado a lo largo del libro.~~