Gestión de procesos
Una visión interna

Septiembre de 2010

Autor: Fernando Pérez Costoya

Sobre este documento

Como su título indica, este documento analiza la gestión de procesos desde un punto de vista interno. Aunque está concebido como un documento autónomo y autocontenido, puede servir de complemento al capítulo de procesos del libro "Sistemas Operativos: una visión aplicada", presentando un enfoque algo diferente, más avanzado y orientado hacia aspectos de diseño. En cualquier caso, hay que resaltar que se trata de un documento que está ideado para su **libre distribución** y que no tiene ninguna vinculación con el libro anteriormente citado, más allá de que está creado por uno de los autores del mismo.

Prefacio

El proceso es la principal abstracción creada por el sistema operativo, alrededor de la cual se articula toda su funcionalidad. Suele ser, por ello, el primer concepto que se estudia dentro de esta materia. Esta abstracción permite la ejecución segura y eficiente de múltiples actividades concurrentes sobre un único computador, lo que redunda en un uso más eficiente del mismo y en un mejor servicio a los usuarios.

El objetivo de este documento es estudiar en detalle cómo se implementa la abstracción de proceso. Se trata de un tema que está fuertemente vinculado con la manera como el sistema operativo maneja los distintos eventos internos del procesador (interrupciones, excepciones y llamadas al sistema). Por tanto, una parte importante de la exposición abordará este aspecto.

La presentación tiene un carácter avanzado y, por tanto, se asume que el lector ya posee previamente conocimientos sobre aspectos básicos de la gestión de procesos, tales como el concepto de proceso e hilo, los fundamentos de la multiprogramación, de la planificación de procesos y de los mecanismos de interacción entre los procesos, así como el API de procesos de algún sistema operativo. El tratamiento que se le otorga al tema en esta disertación tiene un mayor nivel de profundidad que el que se le suele proporcionar en los libros de texto generales de sistemas operativos, estando más cercano al enfoque utilizado en libros dedicados a estudiar los aspectos internos de un determinado sistema operativo. Sin embargo, a diferencia de estos libros, que tienen un planteamiento orientado a las características específicas de cada sistema operativo, el enfoque de este capítulo es neutral, independiente de cualquier sistema operativo, intentando plantear aquellos aspectos comunes, aplicables a cualquier sistema operativo de propósito general. Esta neutralidad no quita, en cualquier caso, que la exposición se ilustre con diversos ejemplos reales de distintos sistemas operativos, tanto de sistemas Windows como de la familia UNIX.

Con respecto al contenido del capítulo, hay que comentar que se ha tomado la decisión de no introducir el concepto de hilo desde el principio del tema, usando para la primera parte de la exposición un modelo de procesos convencional con un solo flujo de ejecución por proceso. Aunque la opción de presentar el concepto de hilo desde el principio es igualmente razonable, se ha preferido esta alternativa que se corresponde con cómo ha sido la evolución histórica de los sistemas operativos, lo que consideramos que permite al lector apreciar mejor qué aporta al sistema operativo la incorporación de esta nueva abstracción.

Otro aspecto que conviene resaltar es que el estudio realizado en este documento se ha extendido también a sistemas multiprocesadores, mostrando las peculiaridades de la gestión de procesos en este tipo de sistemas, que cada vez están más presentes en todos los ámbitos de aplicación de la informática, especialmente con la aparición de los procesadores multi-núcleo. Asimismo, se recogen algunas consideraciones de diseño específicas de los sistemas de tiempo real, aunque siempre dentro del contexto de los sistemas no críticos.

En cuanto a la organización del documento, el capítulo comienza con una introducción que repasa el concepto de proceso. A continuación, se estudia cómo maneja el sistema operativo los distintos eventos del procesador. La tercera sección se centra en el objetivo básico del documento: la implementación del modelo de procesos. A continuación, se estudia cómo se llevan a cabo las diversas operaciones de gestión de los procesos, tales como la creación y destrucción de un proceso. La siguiente sección estudia el problema de la sincronización tanto dentro del propio sistema operativo como entre los procesos. Por último, se analiza cómo se implementa la abstracción de hilo, comparando distintas alternativas.

A continuación, se presenta el índice que se seguirá en esta presentación:

- *Introducción*
- *Gestión interna de eventos*
- *Implementación del modelo de procesos*
- *Operaciones sobre los procesos*
- *Sincronización*
- *Implementación de hilos*

1 Introducción

En un sistema con multiprogramación, el sistema operativo debe repartir los recursos disponibles entre los distintos programas activos en el sistema, de manera que cada programa ejecute como si tuviera su propio computador. La ejecución concurrente de programas en un procesador proporciona tres beneficios:

- Permite un uso más eficiente del procesador, ya que casi siempre tendrá cosas que hacer.
- Proporciona un mejor servicio a múltiples usuarios, puesto que se puede intercalar la ejecución de actividades asociadas a distintos usuarios.
- Posibilita el desarrollo de aplicaciones concurrentes con múltiples flujos de ejecución, que en algunos casos pueden conducir a una solución más adecuada y eficiente que la obtenida usando un programa secuencial, incluso en sistemas con un único procesador.

Además de dar servicio a los múltiples programas que se están ejecutando en cada momento, el sistema operativo debe gestionar los diversos elementos hardware presentes en el sistema. Esto requiere ejecutar múltiples actividades independientes de forma concurrente. Sin embargo, la arquitectura von Neumann está concebida para la ejecución de una única actividad en cada procesador. Por tanto, el sistema operativo debe crear el artificio necesario para intercalar en cada procesador la ejecución de aquellas actividades presentes en el sistema que se hayan asignado al mismo (que serían todas en el caso de un sistema con un único procesador).

En un sistema de propósito general, la ejecución de un programa en un procesador se puede ver entremezclada con actividades tales como las siguientes:

- Las rutinas de tratamiento de interrupción de los distintos dispositivos.
- Las llamadas al sistema que realiza el programa a lo largo de su ejecución.
- El tratamiento de las excepciones que puede causar el programa durante su ejecución.
- La ejecución de otros programas, consecuencia de la multiprogramación.

El sistema operativo debe multiplexar los recursos hardware entre las distintas actividades del sistema repartiendo de forma transparente y segura dichos recursos entre las mismas. Dado que los dos principales recursos del computador son el procesador y la memoria, a continuación, se analiza cómo se realiza el reparto de cada uno de ellos.

Por lo que se refiere al procesador, se debe hacer un reparto del tiempo del procesador entre las actividades involucradas, de forma transparente, sin que se vean afectadas por el mismo. Para ello, en las cuatro situaciones planteadas, cuando se interrumpe una actividad que estaba ejecutándose, se debe almacenar el estado de los registros del procesador en ese punto y restaurarlos cuando se reanude dicha actividad interrumpida.

En cuanto a la memoria, en vez de una multiplexación temporal como ocurre con el procesador, en este caso, como se estudia en el tema de gestión de memoria, se trata de una multiplexación espacial: el sistema operativo asignará a cada programa un espacio de memoria suficiente para su ejecución, pero de manera que el programa no se vea afectado de ninguna forma por este hecho. De esta manera, cada vez que ejecute un programa todas las direcciones de memoria que genere serán interpretadas dentro de su mapa de memoria. Así, por ejemplo, la dirección de memoria 100 que utiliza un programa durante su ejecución no tiene ninguna relación con la dirección 100 que usa otro programa. Cuando hay un cambio del programa en ejecución, se instala el mapa del nuevo programa. Sin embargo, esta situación de cambio de programa sólo se produce en el último de los cuatro casos planteados. En los tres primeros, en los que la ejecución de un programa se ve detenida por el tratamiento de una interrupción, una excepción o una llamada al sistema, no existe tal cambio, por lo que no es necesario instalar un nuevo mapa de memoria, siendo suficiente con hacer un reparto del tiempo del procesador, ya que la nueva actividad comparte el resto de los recursos con aquélla que ha sido interrumpida (como se analizará más adelante, estamos asumiendo un modelo de ejecución de "Múltiples espacios de direcciones con mapa de sistema único", que es el más habitual; con otro tipo de modelo, cambiaría el modo de multiplexar la memoria).

El éxito de la multiprogramación reside en gran parte en toda esta transparencia: el programador no se ve afectado en ninguna fase del desarrollo de una aplicación (programación,

compilación, montaje y ejecución) por el hecho de que esta aplicación vaya a ejecutarse conviviendo con otras.

Mediante este mecanismo de reparto transparente de recursos, el sistema operativo logra crear una abstracción que podría considerarse como un "procesador virtual": se le proporciona a cada programa una máquina von Neumann propia (más lenta, debido a la multiplexación del procesador real, y con menos memoria, debido al reparto de la memoria real), sin que su ejecución se vea interferida por la existencia de otros programas que están ejecutando en otros "procesadores virtuales". Esta abstracción de procesador virtual, que proporciona un entorno de ejecución independiente para cada programa, es a la que se le denomina "proceso".

Nótese que esta definición de proceso es más abstracta y genérica que la que se suele usar en cursos de introducción de sistemas operativos, en los que se suele definir el proceso como un programa en ejecución. De hecho, es importante enfatizar sobre la diferencia entre el programa, que es una entidad pasiva que implementa algún tipo de algoritmo, y el proceso, que se corresponde con una entidad activa que lleva a cabo una determinada tarea que, evidentemente, está muy condicionada por el programa que está ejecutando el proceso, pero no sólo por ello. La labor realizada por un programa también está determinada por otros factores tales como cuáles son los argumentos y el entorno que recibe o quién es el usuario que lo ha activado. De hecho, en un sistema con multiprogramación, en un momento dado, pueden existir múltiples copias activas del mismo programa (por ejemplo, el intérprete de mandatos) y, por tanto, múltiples procesos ejecutando el mismo programa, que en cada caso pueden estar llevando a cabo una labor diferente.

La distinción entre programa y proceso es especialmente apreciable en el API de gestión de procesos de UNIX, donde existen dos llamadas al sistema separadas vinculadas con la creación de procesos y la ejecución de programas, respectivamente. Como ya conoce el lector, estas llamadas son las siguientes:

- La llamada `fork` permite crear un nuevo proceso que es una copia del proceso que invocó la llamada y que, por tanto, ejecutará el mismo programa.
- La llamada `exec` permite que un proceso solicite la ejecución de un nuevo programa. El proceso deja de ejecutar el programa en curso y pasa a ejecutar el programa especificado en la llamada. Por tanto, es el mismo proceso ejecutando otro programa.

Se puede apreciar, por tanto, que en los sistemas UNIX, mediante el uso de `exec`, un proceso puede ejecutar varios programas durante su vida, lo cual refuerza la distinción entre proceso, como entorno o contexto de ejecución, y programa, como conjunto de instrucciones y datos que representan el algoritmo que está realizando actualmente el proceso.

Evidentemente, existen numerosos sistemas operativos en los que no existe una separación de servicios como en UNIX, sino que existe una única llamada que crea un nuevo proceso que ejecuta un nuevo programa. Este es el caso de Windows, donde el servicio `CreateProcess` crea un nuevo proceso que ejecuta el código del programa especificado como argumento. En estos sistemas la relación entre proceso y programa es "para toda la vida", aunque en ellos siga existiendo, obviamente, la clara distinción entre proceso y programa. Digamos que el caso de UNIX es un ejemplo especialmente ilustrativo de esta distinción. Observe que en un sistema UNIX, para lograr crear un nuevo proceso que ejecute un nuevo programa, hay que recurrir al uso combinado de una llamada `fork`, para crear un nuevo proceso, seguida de una llamada `exec` en el nuevo proceso, para solicitar la ejecución del nuevo programa.

Una vez identificadas las distintas actividades que se llevan a cabo de forma concurrente en un sistema e introducida la definición de proceso como la abstracción que crea un procesador virtual que aísla la ejecución de un programa de los otros programas activos en el sistema, en el resto del capítulo se estudia cómo se implementa esta abstracción: cómo el sistema operativo va repartiendo los recursos entre los distintos programas al ritmo que le marcan las interrupciones, excepciones y llamadas al sistema que se van produciendo en el procesador. Dado que la gestión de estos tres tipos de eventos (en la presentación se usará el término *evento* para hacer referencia genérica a cualquiera de esas tres entidades, a saber, interrupción, excepción y llamada al sistema) es la base sobre la que se construye la abstracción de proceso, antes de analizar cómo

se implementa el modelo de procesos, se estudia en la siguiente sección de qué manera el sistema operativo lleva a cabo dicha gestión.

2 Gestión interna de eventos

El manejo de las interrupciones, llamadas al sistema y excepciones se realiza mediante la estrecha colaboración del sistema operativo y del hardware del procesador, pudiendo ser diferente el reparto del trabajo entre ambos dependiendo de cada tipo de sistema. Por ello, en esta sección se lleva a cabo, en primer lugar, un breve repaso de cómo maneja el procesador las interrupciones, llamadas al sistema y excepciones, para, a continuación, analizar cómo gestiona el sistema operativo estos eventos, que, evidentemente, es la parte que más nos interesa en este tema.

2.1 Gestión hardware de eventos

Esta revisión no se centrará en ningún procesador específico, sino que se planteará en términos generales, aplicables a cualquier tipo de procesador, y obviando detalles no relevantes para el sistema operativo. Antes de analizar cómo se gestionan en el nivel hardware las interrupciones, excepciones y llamadas al sistema, se presenta el concepto de modo de operación del procesador.

Modo de operación del procesador

Como el lector ya conoce previamente, un requisito ineludible para construir un sistema operativo multiusuario es que el procesador posea distintos modos de ejecución con diferentes privilegios. Deben distinguirse al menos dos modos de ejecución: uno restrictivo, denominado modo *usuario*, y otro con todos los privilegios, llamado, usualmente, modo *sistema* o *núcleo*.

Cuando el procesador está en modo usuario, hay una serie de restricciones que limitan al programa en ejecución su capacidad de actuar sobre el sistema, en aras de mantener la seguridad en el mismo. Por un lado, no están disponibles todas las instrucciones del procesador. Existen algunas instrucciones que se consideran privilegiadas y que, por tanto, no están disponibles en este modo de ejecución. Algunos ejemplos ilustrativos de este tipo de instrucciones serían la instrucción que permite prohibir las interrupciones en el sistema o la que permite parar la ejecución del procesador (*HALT*). En el caso de un procesador con mapas de memoria y de entrada/salida separados (como, por ejemplo, la familia Pentium de Intel), las instrucciones de entrada/salida también serán privilegiadas, para evitar el acceso indiscriminado a los dispositivos. Asimismo, hay un acceso restringido a los registros del procesador (por ejemplo, en este modo no se podrá acceder al registro que mantiene la dirección de comienzo de la tabla de páginas activa) y a la memoria del sistema, quedando sólo accesibles las direcciones lógicas de usuario. Nótese que la información del modo de ejecución del procesador y del estado de las interrupciones está normalmente almacenada en un registro de estado que, evidentemente, sólo es manipulable estando en modo sistema.

En modo sistema, no hay ninguna restricción y, por tanto, está disponible todo el juego de instrucciones y hay acceso ilimitado a todos los registros, dispositivos de entrada/salida y memoria del sistema.

Aunque este primer apartado se centra en aspectos hardware, se puede anticipar que, como se verá más adelante y como el lector podrá suponer a priori, el sistema operativo se las arreglará para asegurarse de que cuando se estén ejecutando programas de usuario, el procesador esté en modo usuario, y que cuando se trate del código del sistema operativo, el procesador tenga modo sistema.

Es interesante resaltar que hay procesadores que distinguen más de dos modos de ejecución. Así, por ejemplo, los procesadores Pentium de Intel proporcionan 4 modos de ejecución diferentes, que van desde un modo con privilegio mínimo hasta uno con todos los privilegios, pasando por dos modos intermedios que progresivamente van teniendo mayores privilegios. Aunque, a priori, esta riqueza en modos de ejecución parece interesante (así, el diseñador de un sistema operativo podría plantearse una arquitectura por capas tal que ciertas partes del código del sistema operativo que, por sus características, no requieran un privilegio total puedan

ejecutarse en un modo intermedio), los sistemas operativos de propósito general no suelen utilizarla, usando sólo el modo de privilegio mínimo como modo usuario y el de máximo privilegio como modo sistema. Aunque parezca sorprendente, esta infrautilización de la funcionalidad proporcionada por el hardware es habitual en el sistema operativo. En muchas ocasiones, el diseñador del sistema operativo prefiere usar sólo la funcionalidad mínima requerida, que estará presente en cualquier procesador, en lugar de aprovecharse de las funcionalidades extendidas que puede proporcionar un determinado procesador, siempre que no haya una repercusión significativa en las prestaciones del sistema. Esta estrategia facilita el transporte de un sistema operativo a distintas plataformas hardware. Un ejemplo adicional de esta política "de mínimos" se produce en la gestión de memoria. La teoría de gestión de memoria establece que un sistema de paginación es suficiente para llevar una gestión segura y eficiente de la memoria del sistema, aunque la técnica de segmentación paginada, presente, por ejemplo, en los procesadores Pentium de Intel, proporciona algunas ventajas, liberando al software de ciertas operaciones. Aplicando esta misma estrategia, muchos sistemas operativos, como, por ejemplo, Linux, que ejecutan sobre estos procesadores de Intel, prefieren no hacer uso de la segmentación proporcionada por la unidad de gestión de memoria del procesador.

Cambio de modo de ejecución

El procesador va transitando de forma controlada entre los dos modos de ejecución, de manera que se asegure un buen funcionamiento del sistema. A continuación, se analiza cómo se realizan estas transiciones, identificando qué eventos las generan y qué acciones hardware conlleva el cambio en el nivel de procesador.

Cuando se inicia la ejecución del procesador, éste comienza en modo sistema. Además, en este estado inicial las interrupciones están inhibidas y, generalmente, está desactivado el hardware de traducción de memoria (las direcciones lógicas que se generan en la unidad de control salen al bus sin que la unidad de gestión de memoria las manipule). En caso de un sistema multiprocesador, normalmente, en el estado inicial sólo está activo un procesador (que suele denominarse "procesador maestro") y con una configuración tal que las interrupciones, que inicialmente están inhibidas, se dirigirán a ese procesador.

Como se verá en secciones posteriores, en la fase inicial toma control el sistema operativo que, una vez realizadas las correspondientes iniciaciones, cede el control, mediante un artificio, al primer proceso de usuario que ejecutará con el procesador en modo usuario, las interrupciones habilitadas y el hardware de traducción de memoria activado.

A partir de este momento, se puede considerar que la situación habitual es que el procesador esté en modo usuario, con las interrupciones habilitadas, y sólo transitará a modo sistema debido a la aparición de algún evento "excepcional" (interrupción, excepción o llamada al sistema) en el procesador. Una vez tratado el evento, o el conjunto de eventos que se han producido de forma anidada, se retornará al modo usuario.

Los procesadores gestionan normalmente una pila (y, por tanto, un registro de puntero de pila) para cada modo de ejecución. Cuando se produce un evento estando el procesador en modo usuario, el hardware empieza a usar automáticamente el puntero de pila de modo sistema (o *puntero de pila de sistema*, para abreviar), salvando en ella información del estado de la ejecución actual, para poder restaurarla después del tratamiento del evento. Además, se pone al procesador en modo sistema y se salta a la rutina de tratamiento de dicho evento, terminándose con ello el tratamiento hardware del evento. Para determinar cuál es la rutina de tratamiento, se suele usar el mecanismo de los vectores de interrupción, que, como el lector ya conocerá, consiste en que cada evento tiene asociado un número (un *vector*) que el procesador usa para acceder a la posición correspondiente de una tabla donde se almacena la dirección de la rutina de tratamiento de cada evento.

Con respecto a la pila de sistema, algunos procesadores usan dos pilas de sistema: una pila, que mantiene el nombre de pila de sistema, para tratar eventos síncronos (que, como se verá más adelante, corresponden a excepciones y llamadas al sistema) y otra pila, que se suele denominar pila de interrupciones, para gestionar eventos asíncronos (que corresponden a interrupciones). Más adelante, se analiza el uso de estas dos pilas de sistema, aunque mientras no se diga lo contrario, se presupone un procesador con una única pila de sistema.

En cuanto a la información que el hardware salva en la pila de sistema cuando se produce un evento, va a depender de las características específicas de cada procesador. Como mínimo, debe almacenarse el registro de estado, ya que en este registro se almacena el modo de ejecución y el nivel de interrupción previos, y el contador de programa de la ejecución interrumpida. Hay que resaltar que aquellos registros que no salve el hardware automáticamente y que se puedan ver afectados por la ejecución de la rutina de tratamiento del evento, deberán salvarse por software, esto es, deberá almacenarlos en la pila la propia rutina. En algunos procesadores se puede especificar qué registros del procesador se salvan automáticamente en la pila cuando se produce un evento, liberando de esta labor al sistema operativo. Aunque en principio puede parecer una ventaja, algunos sistemas operativos prefieren especificar en estos casos que el hardware guarde la mínima información en la pila y salvar el resto por software, sacrificando una posible mayor eficiencia por una mayor flexibilidad, ya que el sistema operativo conoce mejor el uso real de los registros.

Puede producirse un anidamiento en el tratamiento de los eventos en el caso de que, durante el tratamiento de un evento, se active otro más prioritario. En esta situación, el procesador ya estaba previamente en modo sistema. Por tanto, no se producirá un cambio de modo ni de pila: sólo se salvará la información habitual en la pila activa, que es la de modo sistema. En caso de que se trate de un procesador con dos pilas de sistema, si se estaba ejecutando la rutina de tratamiento de una llamada al sistema o una excepción, usando para ello la pila de sistema, y se produce una interrupción, el procesador comenzará a usar la pila de interrupción.

Cuando termina una rutina de tratamiento, la ejecución de la instrucción de retorno de la rutina causa que se recupere de la pila de sistema la información que restaura el modo de ejecución previo a la activación de esta rutina. En el caso de que no haya anidamiento, se retornará a modo usuario. Ésta es la forma habitual de pasar de modo sistema a modo usuario.

A continuación, se distinguen y caracterizan, desde el punto de vista del tratamiento hardware, los tres tipos de eventos que ponen en modo sistema al procesador: interrupciones, excepciones y llamadas al sistema.

Interrupciones

Se trata de eventos de carácter asíncrono que notifican al procesador que se ha producido alguna circunstancia que debe atender. Generalmente, están asociadas a dispositivos de E/S. Los controladores de los dispositivos generan interrupciones para notificar algún evento del dispositivo que necesita ser tratado, como la finalización de una operación de E/S. Además de interrupciones asociadas a dispositivos, también suelen existir interrupciones vinculadas con situaciones de error críticas en el sistema, tales como fallos del hardware del equipo. Generalmente, este tipo de interrupciones no pueden ser inhibidas debido a su carácter crítico (en algunos procesadores, se las califica como “no *enmascarables*”).

Existe una gran variedad de esquemas hardware de gestión de interrupciones. En esta sección sólo se pretende repasar las características generales de este tipo de esquemas, centrándose únicamente en aquellos aspectos relevantes al sistema operativo.

Generalmente, hay múltiples líneas de interrupción, pudiendo haber varios dispositivos conectados a la misma línea. En algunos sistemas, estas líneas están conectadas directamente a entradas del procesador disponibles para este fin. En otros procesadores, como los de la familia Pentium de Intel, están conectadas al procesador a través de un controlador de interrupciones, que se encarga de su gestión. En cualquier caso, cada línea tiene asociado un vector de interrupción que identifica la rutina de tratamiento, siendo esta asociación, habitualmente, configurable.

Dado que hay múltiples fuentes de interrupciones, debe existir un mecanismo capaz de arbitrarlas. Hay una gran variedad de alternativas, pero, desde el punto del sistema operativo, interesa distinguir entre sistemas que proporcionan un esquema de prioridades entre las interrupciones y aquéllos que no lo hacen.

En el esquema donde el procesador no distingue prioridades entre las interrupciones, que se corresponde, por ejemplo, con la familia Pentium de Intel, el procesador puede tener en cada momento inhibidas o habilitadas las interrupciones. Se dispone de instrucciones para cambiar este estado, que, inicialmente, es de inhibición. Cuando se acepta una determinada interrupción,

el procesador inhibe automáticamente las interrupciones hasta que termine la rutina de tratamiento o hasta que el código de la rutina las habilite explícitamente. Generalmente, en este tipo de esquemas existe además la posibilidad de, estando habilitadas la interrupciones, poder *enmascarar* (inhibir) selectivamente una o más líneas de interrupción.

En los esquemas basados en niveles de prioridad, usados, por ejemplo, por los procesadores SPARC, cada línea de interrupción tiene asignada una prioridad, lo que permite discriminar entre distintos grados de urgencia a la hora de tratar las diversas interrupciones. En cada momento el procesador tiene asociado un nivel de interrupción determinado, de manera que sólo admitirá interrupciones que tengan un nivel superior al del procesador. Cuando el procesador acepta una interrupción, el nivel de interrupción del procesador se eleva automáticamente al correspondiente a dicha interrupción y, al terminar la rutina de tratamiento, se restaurará el nivel previo. Existen también instrucciones que permiten cambiar explícitamente el nivel de interrupción del procesador, modificando así qué interrupciones quedan habilitadas y cuáles inhibidas. Inicialmente, el procesador comienza a ejecutar con el nivel máximo, lo que implica que todas las interrupciones están inhibidas.

Las interrupciones en un multiprocesador

La gestión de las interrupciones en un multiprocesador presenta numerosos aspectos específicos que hacen que su complejidad sea apreciablemente mayor que en un sistema monoprocesador. La primera cuestión es cómo se determina a qué procesador o procesadores debe enviarse una determinada interrupción. La mayoría de los multiprocesadores permite configurar la manera como se distribuye cada interrupción, proporcionando modalidades como las siguientes: envío siempre al mismo procesador, envío a un subconjunto (*multicast*) o a todos los procesadores (*broadcast*), envío a un procesador siguiendo un turno rotatorio, envío al procesador que ejecute el proceso de menor prioridad (el procesador dispone de un registro que se carga con un valor que representa la prioridad del proceso que está ejecutando en el mismo) o envío al mismo procesador al que se le transmitió una interrupción de ese mismo tipo la última vez que se produjo, siempre que no haya pasado un determinado umbral de tiempo (el objetivo de esta modalidad es intentar aprovechar la información de la rutina de interrupción que pueda quedar todavía en la memoria caché de ese procesador).

Otro aspecto que conviene resaltar es que en un sistema multiprocesador, aunque se inhiba una interrupción en un procesador, puede activarse en otro cuyo estado interno lo permita. Esto dificulta considerablemente lograr una correcta sincronización en el sistema operativo, como se hará patente en la sección dedicada a este tema.

Un último aspecto específico de los multiprocesadores que conviene resaltar por su relevancia para el sistema operativo es la interrupción entre procesadores (IPI, *InterProcessor Interrupt*). Este mecanismo permite enviar una interrupción desde un procesador a otro para forzar a que este último realice una determinada operación. La IPI se usa en múltiples labores, como, por ejemplo, planificación de procesos o gestión de memoria.

Excepciones

Corresponden a situaciones de carácter excepcional que detecta el procesador durante la ejecución de una instrucción. Algunos ejemplos de este tipo de eventos son una división por cero o la ejecución de una instrucción privilegiada en modo usuario. Se trata, por tanto, de eventos de carácter síncrono, ya que el instante en el que se producen está claramente determinado: en el momento de ejecutarse la instrucción que los causa.

Aunque la mayoría de las excepciones corresponden a situaciones de error, no siempre es así. Algunas excepciones son la base para implementar funcionalidades normales, libres de error. Así, por ejemplo, a partir de la excepción del fallo de página, se articula todo el sistema de memoria virtual basado en paginación por demanda. Asimismo, existen diversas excepciones destinadas a facilitar el proceso de depuración de programas (por ejemplo, la excepción correspondiente a la ejecución paso a paso de instrucciones o la generada al alcanzarse un punto de ruptura).

Como ocurre con las interrupciones, se pueden producir estando el procesador previamente en cualquier modo de ejecución. El procesador pasa a modo sistema, en caso de que no lo estuviera previamente, pero no se altera el nivel de interrupción (o el estado de las interrupciones, si se trata de un sistema sin prioridades) previo. Generalmente, cada excepción tiene un vector fijo. Así, por ejemplo, en los procesadores Pentium de Intel, la excepción de dividir por cero tiene asociado el vector 0.

Llamadas al sistema

Este tipo de eventos se producen mediante una instrucción no privilegiada que causa la llamada al sistema. De esta forma, los programas de usuario invocan los servicios del sistema operativo. En los procesadores Pentium de Intel, se trata de la instrucción *INT*. En los últimos modelos de este procesador, se han añadido las instrucciones *SYSENTER*, para realizar la llamada, y *SYSEXIT*, para terminarla, que son más eficientes al conllevar una menor sobrecarga.

Se trata de eventos de carácter síncrono, ya que se producen cuando se ejecuta la instrucción correspondiente. El procesador previamente está en modo usuario, transitando con la ejecución de esta instrucción a modo sistema, manteniéndose todas las interrupciones habilitadas. En algunos procesadores, este tipo de eventos tiene un vector fijo, mientras que en otros, como en Intel cuando se usa la instrucción *INT*, el vector es un operando de la instrucción. En el caso de la instrucción *SYSENTER*, no se usa el mecanismo de los vectores para determinar la dirección de la rutina de tratamiento, sino que se utiliza un registro privilegiado donde se almacena la dirección de la rutina de tratamiento de ese evento.

2.2 Gestión de eventos por el sistema operativo

El sistema operativo es un sistema dirigido por eventos. Por tanto, tiene el mismo tipo de estructura que cualquier otro sistema de estas mismas características. Supóngase, por ejemplo, una aplicación que proporciona una interfaz de usuario gráfica que sigue un modelo dirigido por eventos. La estructura habitual de estas aplicaciones es la mostrada en el programa 1.

Programa 1 Esquema básico de un programa que sigue un modelo dirigido por eventos.

```
void tratar_boton_OK(){
    ........
}
void tratar_boton_Cancel(){
    ........
}
........
int main(){
    inicia_estructuras_de_datos();
    establecer_manejador(boton_Cancel, tratar_boton_Cancel);
    establecer_manejador(boton_OK, tratar_boton_OK);
    ........
    presentar_ventana_inicial();
    pausa();     // se queda indefinidamente bloqueado a la
                 // espera de eventos
}
```

Como se puede apreciar en el fragmento de código anterior, el programa comienza iniciando sus estructuras de datos y estableciendo las rutinas que manejarán los distintos eventos. Al terminar esta fase inicial, la función principal ya ha terminado su cometido y, por tanto, detiene su ejecución. A partir de este momento, se ejecutarán las rutinas manejadoras según vayan produciéndose los distintos eventos. Por tanto, el programa sólo mantiene el control activo de la ejecución durante la rutina inicial. A partir de ese momento, son los eventos los que determinan la ejecución del programa, que tiene, por tanto, un carácter pasivo: se ejecutan las rutinas que activan los sucesivos eventos.

Se puede hacer un paralelismo entre ese ejemplo y la estructura del sistema operativo, debido a que en ambos casos subyace un modelo dirigido por eventos. En el caso del sistema operativo, los eventos son los generados por el procesador y la tabla de vectores de interrupción es la estructura de datos que mantiene cuáles son las rutinas manejadoras.

Cuando se inicia la ejecución del procesador, éste comienza en modo sistema ejecutándose el programa que carga el sistema operativo (*boot*). Una vez terminado el proceso de carga, tomará control el sistema operativo, que se encontrará con el procesador en el estado "seguro" previamente descrito (modo sistema, interrupciones inhibidas y hardware de traducción de memoria desactivado).

De manera similar al ejemplo planteado, en esta fase inicial, el sistema operativo tiene el control de la ejecución y realiza todas las iniciaciones, tanto del hardware (por ejemplo, controladores de dispositivos y hardware de gestión de memoria) como de sus propias estructuras de datos (por ejemplo, la tabla de procesos).

En el caso de un multiprocesador, el sistema operativo debe encargarse también de activar el resto de los procesadores del sistema. Asimismo, debe configurar el esquema de distribución de interrupciones. Habitualmente, en un sistema operativo de propósito general se suele usar una distribución siguiendo un turno rotatorio, aunque puede haber otros esquemas convenientes dependiendo de las características de cada interrupción. Por ejemplo, podría ser interesante configurar la interrupción de reloj en modo *broadcast*, de manera que la actualización de la contabilidad sobre el uso del procesador por parte de cada uno de los procesos que están ejecutando en los distintos procesadores se haga en paralelo, facilitando así la capacidad de crecimiento del sistema.

Como parte del proceso de iniciación, se rellena la tabla de vectores de interrupción de manera que cada posición contenga la dirección de la rutina del sistema operativo que lleva a cabo el tratamiento de la misma. Nótese que esta última acción se corresponde con establecer el manejador para cada tipo de evento. Es importante resaltar que el sistema operativo debe rellenar todas las posiciones de la tabla, incluso aquéllas que se corresponden con interrupciones de dispositivos que no existen en el sistema. En caso de que no se hiciera esto, si ocurre una interrupción indebida por algún error en el hardware del sistema, se ejecutaría en modo sistema un código impredecible, puesto que la posición del vector no ha sido rellenada con un valor razonable, con unas consecuencias inimaginables. Normalmente, todas las posiciones correspondientes a interrupciones que no deberían producirse se rellenan con la dirección de una misma rutina de interrupción, que usualmente genera un mensaje de error por la consola. Hay que resaltar que, en el caso de que el sistema operativo proporcione algún mecanismo de carga dinámica de manejadores (*drivers*), algunas de las posiciones a las que durante la fase inicial se les asocia esta rutina de error pueden rellenarse posteriormente con una rutina de tratamiento específica cuando se cargue el manejador correspondiente.

En esta fase inicial, el sistema operativo crea el proceso inicial (y posiblemente, como se verá más adelante, varios procesos de núcleo). Dado que se pretende que cuando se empiece a ejecutar dicho proceso lo haga en modo usuario y con las interrupciones habilitadas, el sistema operativo prepara el proceso de manera que el valor inicial del registro de estado cuando empiece a ejecutarse dicho proceso especifique la situación deseada (modo usuario e interrupciones habilitadas). Más adelante, se analizará en detalle este artificio, que es el que se usa para activar por primera vez cada proceso.

Una vez terminada esta fase inicial, el sistema operativo cede el control al primer proceso de usuario. Volviendo al ejemplo de aplicación dirigida por eventos planteado, este punto se corresponde con el final de la rutina inicial de la aplicación y, como ocurría con dicho ejemplo, a partir de este punto el sistema operativo se convierte en una entidad pasiva: sólo se ejecutará código del sistema operativo cuando se produzca alguno de los eventos del procesador estudiados previamente.

El modo de operación que se acaba de describir asegura que nunca se ejecute en modo sistema el código de programas de usuario. En el arranque, con el procesador en modo sistema, el mecanismo de *boot* garantiza que sea el sistema operativo el que se esté ejecutando. Una vez arrancado el primer proceso de usuario, el procesador sólo puede pasar a modo sistema cuando ocurre un evento, pero, si la tabla de vectores de interrupción se ha rellenado correctamente de

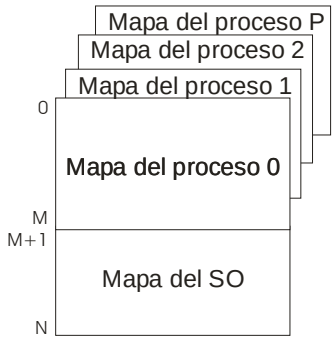


Figura 1 *Múltiples espacios de direcciones con mapa de sistema único.*

manera que contenga las direcciones de rutinas del sistema operativo y esta tabla no se puede modificar estando el procesador en modo usuario, está garantizado que no se puede ejecutar código de programas de usuario con el procesador en modo sistema.

Antes de analizar la gestión de eventos por parte del sistema operativo, es conveniente estudiar cómo es el modelo de memoria que el sistema operativo ofrece a los procesos. Evidentemente, el tema de la gestión de memoria es suficientemente complejo para tener que dedicarle un capítulo completo. Sin embargo, en este punto, se considera oportuno introducir algunos conceptos de gestión de memoria que repercuten directamente en cómo se implementa el modelo de procesos.

Modelo de memoria del proceso

El modelo de memoria proporcionado por la mayoría de los sistemas operativos provee de un espacio de direcciones independiente a cada proceso integrando dentro del mismo el espacio de direcciones del sistema, que es común a todos los procesos, como refleja la figura 1.

Con este esquema, cada programa genera direcciones en un mismo rango de 0 a M durante su ejecución (de 0 a 3G-1, en el caso de Linux sobre un Pentium de 32 bits). Estas direcciones son traducidas por el hardware de gestión de memoria a las direcciones de memoria reales asignadas al programa para su ejecución. De esta forma, un programa no se ve afectado por la convivencia con otros: el programa no debe tener que volver a compilarse o enlazarse aunque ejecute cada vez en direcciones de memoria diferentes. Aunque en el ejecutable del programa se haga referencia a una determinada dirección (por ejemplo, la 100), cuando se produzca la ejecución del programa, la dirección que finalmente llega a la memoria debe corresponder a la dirección física asignada al proceso durante esta ejecución, que será diferente de la correspondiente a esa misma dirección (la 100) de cualquier otro proceso. El programa durante su ejecución, al estar el procesador en modo usuario, sólo podrá acceder a las direcciones que le corresponden (las de su mapa), pero no a las del sistema operativo o a las de otro proceso. Cuando hay un cambio de proceso, se debe activar su mapa de memoria para, de esta forma, conseguir que sus accesos a memoria sean a las direcciones que tiene asignadas.

El sistema operativo tiene asignado un rango disjunto de direcciones para su mapa (de 3G a 4G-1, en Linux sobre un Pentium de 32 bits). De esta forma, no es necesario cambiar de mapa cuando se transita de modo usuario a sistema o viceversa. Al estar el procesador en modo sistema, cuando ejecuta el sistema operativo puede acceder a todo el rango de direcciones: tanto a su mapa, cuando usa direcciones en el rango de M+1 a N, como al del proceso en ejecución, si usa direcciones entre 0 y M. Para entender esa última posibilidad, téngase en cuenta que cuando un proceso realiza una llamada al sistema, el código que ejecuta dicha llamada debe poder acceder al mapa del proceso para poder leer o escribir información en el mismo. Considere, por ejemplo, una llamada al sistema `read`, que especifica como parámetro la dirección del *buffer* de lectura. Nótese que con este esquema el sistema operativo no tiene acceso directo a los mapas de los procesos que no están ejecutando.

Este esquema con múltiples espacios de direcciones y un mapa de sistema único es el más habitual y el que asumiremos en todo este capítulo. Sin embargo, existen otras estrategias mucho menos utilizadas, que comentaremos brevemente a continuación:

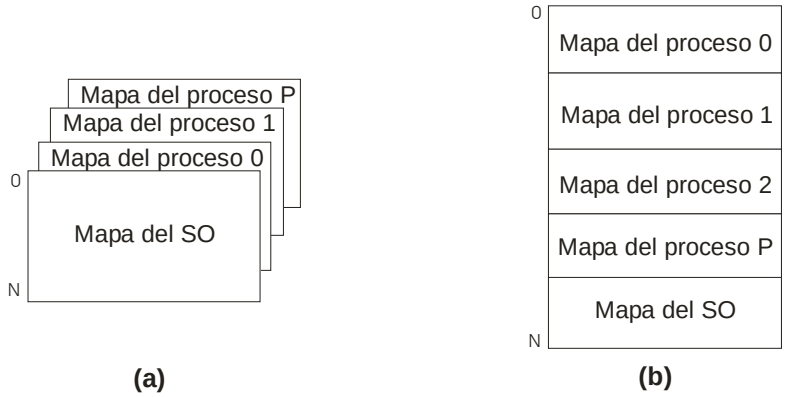


Figura 2 *Modelos de memoria:* ***(a)*** *Espacio de sistema independiente* ***(b)*** *Espacio de direcciones único.*

- Uso de un espacio propio independiente para el mapa del sistema operativo. Como se puede apreciar en la parte izquierda de la figura 2, con este esquema el mapa del sistema operativo usa el mismo rango de direcciones que el de los procesos, lo que permite que ambos espacios sean de mayor tamaño (el lector interesado puede consultar información sobre la versión de Linux sobre Pentium denominada 4G/4G, por el tamaño de los mapas, que sigue este esquema). Esta estrategia resulta menos eficiente puesto que requiere cambiar de mapa cada vez que se pasa de modo usuario a sistema o viceversa y dificulta el acceso desde el código de tratamiento de una llamada al mapa del proceso que la invocó.
- Uso de un espacio de direcciones único para los procesos y el sistema operativo. Los sistemas operativos que siguen este esquema se denominan SASOS (*Single Address Space Operating System*) y están actualmente poco extendidos, puesto que requieren un hardware de gestión de memoria específico para poder implementarse eficientemente. Obsérvese que en ese caso no sería necesario modificar el mapa de memoria en ninguna circunstancia, puesto que tanto los procesos como el sistema operativo tienen su propio rango de direcciones. Sin embargo, sí habría que cambiar la protección de la memoria cada vez que se cambie de proceso para asegurar que un proceso sólo accede al rango de direcciones que le corresponde.

Esquemas de gestión de eventos

Con independencia de cómo se lleva a cabo el tratamiento de cada tipo de evento y previo a su estudio, es conveniente analizar qué distintos esquemas se presentan a la hora de organizar el código de tratamiento de los diversos tipos de eventos. Como se verá a lo largo del capítulo, este aspecto tendrá una importante repercusión en la estructura y modo de operación del sistema operativo.

Como se explicó previamente, una vez pasada su fase de iniciación, la ejecución del código del sistema operativo se realiza a través del tratamiento de los distintos eventos del procesador. Por tanto, en una primera aproximación, el modo de operación del sistema operativo sería el siguiente:

- Estando en ejecución un programa y, por tanto, el procesador en modo usuario, se genera un evento transitando el procesador a modo sistema.
- El código de tratamiento del evento forma parte del sistema operativo. La pila del sistema servirá como soporte de los registros de activación asociados a la ejecución del código de tratamiento. Durante la ejecución del mismo, se pueden generar y procesar otros eventos anidados, que usarán también la pila de sistema como soporte (y la de interrupciones, si hay pilas de sistema separadas).
- Al terminar la rutina de tratamiento del evento, se retorna al modo usuario continuando la ejecución del programa en el punto donde se había quedado. En un sistema multiprogramado, en algunos casos se retorna a un proceso distinto del que fue interrumpido, realizándose de esta forma la multiplexación de procesos.

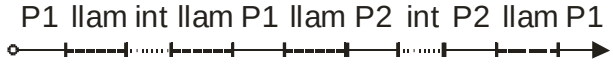


Figura 3 *Traza de ejecución de procesos.*

En este esquema hay un tratamiento homogéneo de todos los eventos. Además, se cumple que no puede haber un cambio de proceso en mitad del tratamiento de un evento, pudiendo darse sólo al final del mismo. La figura 3 ilustra este modelo mostrando un ejemplo de ejecución en las que se alternan fases con el procesador en modo usuario ejecutando programas (dibujadas con líneas continuas) y fases de tratamiento de eventos que corresponden a llamadas al sistema e interrupciones (dibujadas con líneas discontinuas). En la misma se puede apreciar, en primer lugar, cómo el tratamiento de una interrupción se anida en el de una llamada. Más adelante en la traza, se puede observar un cambio del programa en ejecución, que podría deberse, por ejemplo, a que el proceso P1 ha realizado una llamada al sistema que requiere leer de un dispositivo.

Sin embargo, como ocurre en muchas ocasiones, la realidad es un poco más compleja y el tratamiento de los eventos no es tan homogéneo, distinguiéndose apreciablemente entre cómo se gestionan las llamadas al sistema y las excepciones frente a las interrupciones.

Para poder apreciar mejor estas diferencias, utilizaremos un ejemplo muy simplificado del código de un hipotético manejador de un dispositivo de entrada, que opera en modo carácter y usa interrupciones, para un sistema operativo con multiprogramación. Téngase en cuenta que en este punto sólo se muestra un esquema simplificado, sin entrar en detalles de aspectos tales como la salvaguarda y recuperación del estado de la ejecución de un programa, que se analizarán en detalle más adelante.

A continuación, se muestra el programa 2, que constituye una primera versión del manejador planteado como ejemplo. En el mismo, sólo aparecen aquellas operaciones que son relevantes a la hora de comparar los distintos esquemas de gestión de eventos. Téngase en cuenta que este código estaría incluido dentro del sistema operativo y su rutina `lectura` se activaría indirectamente cuando un programa realiza una llamada al sistema que solicita leer de este dispositivo.

Programa 2 Primera versión del manejador del dispositivo.

```
char *dir_buf; // guarda la dirección donde se copiará el dato leído

// Función que realiza la lectura de un carácter copiándolo a "dir"
// que será la dirección especificada por el programa en la lectura.
int lectura(char *dir) {
    dir_buf = dir;
    out(R_CONTROL, LECTURA); // programa el dispositivo
    Retorna cediendo el control a otro proceso;
}
// rutina de interrupción del dispositivo
void interrupcion() {
    *(dir_buf) = in(R_DATOS); // lee el carácter y lo copia
    Marca que el proceso lector ya puede ejecutar;
    Si dicho proceso es más prioritario, retorna al mismo;
    En caso contrario retorna al proceso interrumpido;
}
```

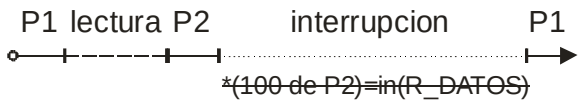


Figura 4 *Traza de ejecución de procesos errónea.*

El esquema planteado parece válido a primera vista. Sin embargo, esta solución no es correcta, ya que cuando la rutina de interrupción hace referencia a la dirección almacenada en `dir_buf`, el sistema de gestión de memoria la interpreta dentro del contexto del proceso actual, que, evidentemente, no es el que realizó la llamada, puesto que dicho proceso está parado esperando a que se complete la operación. Como resultado, se estarán copiando los datos leídos en una zona impredecible del mapa del proceso que estuviera ejecutando en ese momento[1]. Recuerde que cada vez que se cambia de proceso se activa el mapa de memoria del nuevo proceso para que éste continúe su ejecución de forma transparente. La figura 4 ilustra este problema. En ella, el proceso P1, que es más prioritario que P2, ha solicitado una lectura especificando en la llamada una variable `dir` que se corresponde con la dirección 100. Sin embargo, el valor se acaba copiando en la dirección 100 de P2.

Para resolver este problema es necesario que sea el propio proceso lector el que realice la copia. Sin embargo, eso requiere que una vez terminada la rutina de interrupción del dispositivo, prosiga la llamada de lectura y que ésta realice la copia, como se muestra en el programa 3.

Programa 3 Segunda versión del manejador del dispositivo.

```
char buf; // guarda el dato leído
// Función que realiza la lectura de un carácter copiándolo a "dir"
int lectura(char *dir) {
    out(R_CONTROL, LECTURA); // programa el dispositivo
    Guarda estado en ese punto y cede control a otro proceso;
    *dir = buf; // continuará ejecutando esta sentencia
}
// rutina de interrupción del dispositivo
void interrupcion() {
    buf = in(R_DATOS); // lee el carácter y lo copia
    Marca que el proceso lector ya puede ejecutar;
    Si dicho proceso es más prioritario, cede control al mismo;
    En caso contrario retorna al proceso interrumpido;
}
```

Con este esquema, si la ejecución de una llamada debe esperar a que se complete una determinada operación, se almacena el estado de la ejecución justo en ese punto. De esta manera, cuando vuelva a ejecutar el proceso en el momento en que ya se haya finalizado la operación, la llamada proseguirá desde el punto donde se quedó detenida. Toda la información de la ejecución hasta ese momento estará almacenada en la pila del sistema (por ejemplo, el parámetro `dir`), permitiendo que cuando una llamada se reanude después de una parada, pueda hacerlo de forma transparente.

[1] Si se tratara de un sistema operativo de tipo SASOS, la dirección de copia sería válida, incluso aunque el proceso que solicitó el dato no esté en ejecución, puesto que los rangos de direcciones que usan los procesos son disjuntos. Sin embargo, la operación seguiría siendo incorrecta puesto que en ese momento está activa la protección del proceso que está en ejecución, por lo que se produciría un error de acceso.

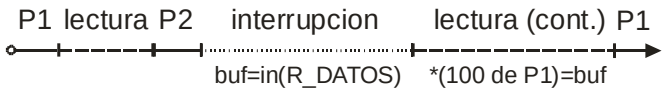


Figura 5 *Traza de ejecución de procesos correcta.*

La figura 5 muestra la traza de ejecución resultante cuando se usa este esquema. En ella se puede apreciar cómo la llamada continúa después de completarse la interrupción. En este caso, la copia final del dato se realiza cuando está instalado el mapa de memoria del proceso lector (P1), con lo que el resultado es correcto.

Aunque en este ejemplo sencillo la llamada sólo tiene dos fases, en general, una misma llamada puede tener un número de fases considerable. Así, por ejemplo, la llamada de apertura de un fichero cuyo nombre de ruta se extienda por múltiples directorios podrá tener varias paradas (o bloqueos) por cada uno de los directorios, cuyo descriptor (*inodo* en el caso de UNIX) y contenido habrá que leer del disco. Tomando como base el ejemplo planteado, su puede complicar un poco la llamada de lectura para mostrar un ejemplo de llamada con múltiples fases, como se puede apreciar en el programa 4, en el que se muestra una versión del manejador donde en la función de lectura recibe como parámetro el tamaño que se desea leer. Obsérvese como en este caso la llamada queda detenida dentro de una función auxiliar. Cuando se reanude la ejecución de la llamada, seguirá dentro de dicha función auxiliar en el mismo estado que se encontraba antes de ceder el control, puesto que en la pila del sistema están almacenados los registros de activación correspondientes a las distintas funciones invocadas por el código de la llamada.

Programa 4 Tercera versión del manejador del dispositivo.

```
char buf; // guarda el dato leído
// Realiza la lectura de "tam" caracteres
int lectura(char *dir, int tam) {
    while (tam--)
        lee_caracter(dir++);
}
// rutina de interrupción del dispositivo
void interrupcion() {
    buf = in(R_DATOS); // lee el carácter y lo copia
    Marca que el proceso lector ya puede ejecutar;
    Si dicho proceso es más prioritario, cede control al mismo;
    En caso contrario retorna al proceso interrumpido;
}
// Función auxiliar que copia un carácter
int lee_caracter(char *dir) {
    out(R_CONTROL, LECTURA); // programa el dispositivo
    Guarda estado en ese punto y cede control a otro proceso;
    *(dir) = buf; // continuará ejecutando esta sentencia
}
```

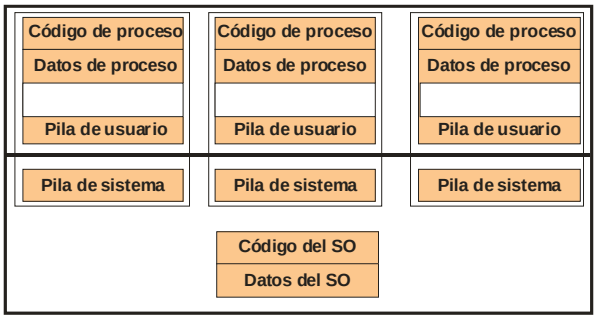


Figura 7 *Organización del sistema basada en el modelo de procesos.*

En la figura 6 se muestra un ejemplo de ejecución donde el proceso P1 ha solicitado leer a partir de la dirección 100 un tamaño de 3 caracteres. En la misma se puede apreciar cómo la llamada de lectura se desarrolla en cuatro fases.

Con este esquema la ejecución de una llamada al sistema solicitada por un determinado proceso puede quedarse bloqueada varias veces hasta que se completa, manteniéndose en la pila del sistema del proceso la información necesaria para que se reanude la llamada de forma transparente. Dado que mientras tanto ejecutarán otros procesos que también realizarán llamadas al sistema, habrá que asignar una pila de sistema a cada proceso, como se puede apreciar en la figura 7.

En este esquema sí que puede haber un cambio de proceso en la mitad del tratamiento de un evento (concretamente, de una llamada al sistema o excepción; como se verá más adelante, el tratamiento de una interrupción nunca puede quedarse a medias por un cambio de proceso). Por consiguiente, en un momento dado puede haber en el sistema un número considerable de rutinas de tratamiento de eventos pendientes de completarse (incluso tantas como procesos). En consecuencia, la ejecución del sistema operativo deja de ser algo lineal, y éste pasa a ser una especie de gran biblioteca de funciones, de carácter pasivo, ejerciendo los procesos como las únicas entidades activas del sistema. Debido a ello, a este esquema se le suele denominar **modelo de procesos**.

El modelo de procesos es el que se utiliza en prácticamente todos los sistemas operativos actuales (por ejemplo, en las familias de sistemas operativos UNIX y Windows), ya que facilita el desarrollo del código del sistema operativo. Sin embargo, existe un modelo alternativo denominado **modelo de interrupciones**, que tiene un modo de operación más parecido al planteado en la primera versión del manejador.

Para entender las posibles ventajas de este segundo modelo, hay que exponer el principal inconveniente del modelo anterior. El modelo de procesos requiere que cada proceso tenga asignada una pila de sistema. En principio, no parece mucho gasto, pero si pensamos en sistemas con decenas de miles de procesos, el coste en memoria puede ser significativo, comprometiendo la capacidad de crecimiento del sistema. Téngase en cuenta que, como se estudia en el tema de gestión de memoria, la pila del sistema de un proceso pertenece a la memoria del sistema y en algunos sistemas operativos se requiere que esté siempre residente en memoria. Otros sistemas operativos, sin embargo, permiten, cuando las circunstancias así lo aconsejen, copiar el contenido de la pila de sistema de un proceso al disco liberando la memoria ocupada por la misma (así ocurre en Windows, pasando el hilo afectado al estado de "en transición"), aliviando, por tanto, este problema.

El modelo de interrupciones plantea que el tratamiento de todos los eventos, incluidas las llamadas al sistema, se complete sin realizarse ningún cambio de proceso (es decir, que tenga una sola fase) en mitad del mismo. Por tanto, cuando hay un cambio de proceso, se puede usar

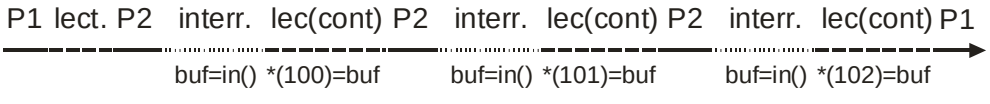


Figura 6 *Traza de ejecución de procesos con una llamada que conlleva múltiples fases.*

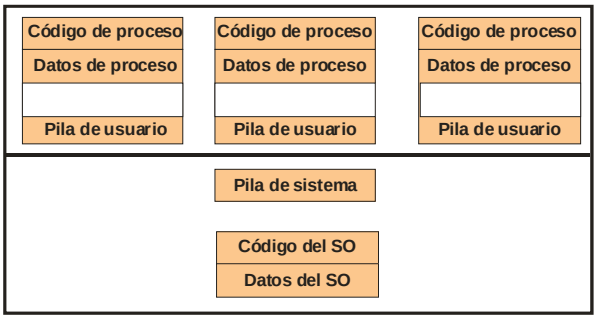


Figura 8 *Organización del sistema basada en el modelo de interrupciones.*

la misma pila de sistema para la ejecución de las llamadas al sistema que realice este segundo proceso, puesto que no hay ninguna información en la misma que haya que preservar. Obsérvese que, además de requerir menos memoria, se consigue un mejor rendimiento del sistema de memorias caché (y de la TLB), puesto que en lo que respecta a la pila del sistema, después de un cambio de proceso, se sigue accediendo a las mismas direcciones de memoria.

En la figura 8 se muestra la organización del sistema operativo siguiendo este modelo. Téngase en cuenta que en un multiprocesador, debería haber una pila de sistema por cada procesador.

Sin embargo, retomando el ejemplo planteado, queda pendiente una cuestión: si no hay una segunda fase de la operación de lectura y desde la interrupción no se puede acceder al mapa del proceso lector, ¿cuándo y cómo se copia el valor leído?

La solución es una técnica que algunos autores denominan **continuaciones**: en vez de "congelarse" la ejecución de la llamada, ésta termina, liberando la pila del sistema, pero estableciendo que la próxima vez que ejecute el proceso se activará la función de continuación especificada. Esta función está incluida dentro del código del sistema operativo y se ejecutará en modo sistema en el contexto del proceso que la activó.

Dado que se pierde el contenido de la pila de sistema, además de la identificación de la función de continuación, habrá que almacenar en una estructura de datos (como el lector ya habrá imaginado, esa información se vincula al BCP del proceso, pero todavía no se ha hablado de esa estructura de datos) aquellos valores que se necesitan para completar la operación.

El programa 5 ilustra un hipotético uso de esta técnica para el código del manejador de dispositivo planteado como ejemplo en esta sección. Observe que es necesario salvar tanto el valor `dir` como el de `tam` para que la rutina de continuación pueda acceder al mismo. La figura 9 muestra un ejemplo de traza de ejecución siguiendo este modelo que corresponde a la lectura de un único carácter.

Programa 5 Cuarta versión del manejador del dispositivo.

```
char buf; // guarda el dato leído
// Realiza la lectura de "tam" caracteres
int lectura(char *dir, int tam) {
    out(R_CONTROL, LECTURA); // programa el dispositivo
    Establece que cuando vuelva a ejecutar el proceso se active
    "cont_lectura" y guarda "dir" y "tam" en una estructura de datos;
    Retorna cediendo el control a otro programa en ejecución;
}
// rutina de continuación
int cont_lectura() {
    char *dir_aux;
    int tam_aux;
```

```
    Recupera en "dir_aux" y "tam_aux" los valores almacenados en la
    estructura de datos;
    *(dir_aux++) = buf;
    if (--tam_aux>0) {
        out(R_CONTROL, LECTURA); // programa el dispositivo
        Establece que cuando vuelva a ejecutar el proceso se active
        "cont_lectura" y guarda "dir_aux" y "tam_aux";
        Retorna cediendo el control a otro programa en ejecución;
    }
    else
        Retorna al proceso que hizo la llamada al sistema;
}
// rutina de interrupción del dispositivo
void interrupcion() {
    buf = in(R_DATOS); // lee el carácter y lo copia
    Marca que el proceso lector ya puede ejecutar;
    Si dicho proceso es más prioritario, retorna a su continuación;
    En caso contrario retorna al proceso interrumpido;
}
```

Nótese que cuando se trata de una llamada con varias fases, la función de continuación puede especificar a su vez otra función de continuación, que en algunos casos podría ser la misma como ocurre en el ejemplo, para poder completar el trabajo.

Aunque esta técnica elimina la necesidad de tener una pila de sistema por cada proceso, conlleva un modelo de programación poco natural, basado en funciones que se ejecutan de forma deslavazada y que se dejan unas a otras la información necesaria para lograr una cierta continuidad lógica. Es evidente que se trata de un modelo de programación propenso a errores.

Para ilustrar este argumento, considérese llamadas al sistema relativamente complejas como la creación de un fichero o la ejecución de un programa (en UNIX, `creat` y `exec`, respectivamente), que requieren la traducción de una ruta que corresponde al nombre del archivo involucrado. Este proceso de traducción es iterativo (una iteración por cada componente de la ruta), con varios posibles bloqueos por iteración. En un modelo de procesos, ese esquema iterativo se convierte directamente en un bucle convencional. Con el modelo de interrupciones, habrá que diseñar una continuación que se invoca a sí misma para implementar la iteración como una especie de recursividad. Asimismo, si se pretende factorizar el código que realiza la traducción de una ruta englobándolo en una función que será utilizada por ambas llamadas al sistema, en el caso de un modelo de interrupciones, habrá que usar algún artificio para saber al completar la traducción de la ruta si hay que proseguir con la creación de un fichero o con la ejecución de un programa. En un modelo de procesos, sin embargo, el mero retorno de la función común de traducción vuelve al punto donde se invocó la función, que proseguirá con la ejecución de la llamada.

Además de ofrecer un modelo de programación inadecuado, el modelo de interrupciones, como se podrá comprender según avance el tema, impide implementar mecanismos más sofisticados como conseguir que un núcleo sea expulsivo o permitir que se produzcan fallos de

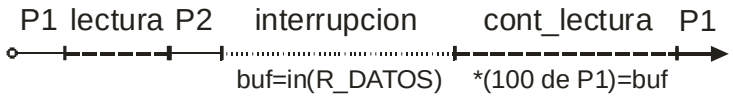


Figura 9 *Traza de ejecución de procesos con un modelo de interrupciones.*

página en modo sistema, puesto que en ambos casos es necesario que cada proceso tenga su propia pila de sistema.

Por todo ello, esta técnica sólo se utiliza en algunos sistemas operativos con arquitectura de tipo micronúcleo (por ejemplo, en el sistema QNX), donde las llamadas al sistema son muy sencillas. Sin embargo, ni siquiera es la opción preferente en este tipo de sistemas, puesto que el micronúcleo L4, reconocido como uno de los sistemas más eficientes dentro de este tipo de arquitecturas, utiliza un modelo de procesos.

Un caso interesante es el del sistema operativo Mach, que permite que convivan ambos modelos: en cada operación de bloqueo se especifica si se usará una continuación (modelo de interrupciones) o el proceso reanudará su ejecución desde el punto donde estaba (modelo de procesos). Otra característica interesante del modelo de interrupciones de Mach es el mecanismo denominado *transferencia de pila* (*stack handoff*). Como se ha explicado previamente, cuando se usa un modelo de interrupciones, en un cambio de proceso, se reutiliza la pila de sistema sobrescribiendo su contenido, puesto que ya no es relevante. Sin embargo, en Mach esto no siempre es así. Cuando hay un cambio de proceso siguiendo el modelo de interrupciones, tal que el proceso que cede el procesador es el emisor de un mensaje y el que lo retoma es el receptor del mismo, se usa la pila de sistema común para transferir información del mensaje optimizando, de esta forma, la operación de transferencia.

En resumen, a todos los efectos, el modelo de procesos es el claro ganador en esta confrontación y, por tanto, será el usado en el resto del capítulo.

Modos de ejecución de un proceso

Con un sistema operativo basado en el modelo de procesos, cuando se produce el tratamiento de un evento, hay un cambio de modo del procesador, pero no hay un cambio del proceso que está ejecutando. La rutina de tratamiento se ejecuta en el contexto del proceso que esté ejecutando actualmente. Así, el mapa de memoria activo en el sistema se corresponderá con el de ese proceso en ejecución. Esto es así, incluso aunque el evento no esté directamente relacionado con el proceso en ejecución. Por ejemplo, mientras se está ejecutando un proceso, puede llegar una interrupción de disco correspondiente a una operación iniciada previamente por otro proceso. Sin embargo, la rutina de interrupción de disco ejecutará en el contexto del proceso en ejecución, aunque no esté vinculado con ese evento, puesto que es el contexto que está actualmente activo.

Como se explicó previamente, los procesos son las únicas entidades activas del sistema, mientras que el sistema operativo, excepto en la fase inicial de arranque, tiene un carácter pasivo, como una especie de gran biblioteca que proporciona servicios a los procesos y trata las interrupciones de los dispositivos. Una vez terminada la fase de arranque del sistema operativo, siempre hay un proceso en ejecución, aunque en algunos casos sea el proceso nulo. Bajo este prisma, dado que el código del sistema operativo, excepto en la fase inicial, se ejecuta siempre en el contexto de un proceso, se puede considerar que dicho código lo ejecuta el propio proceso. Se podría hacer un símil con un programa que llama a una función de biblioteca: dado el carácter pasivo de la biblioteca, se puede considerar que es el programa el que ejecuta el código de la función, aunque, estrictamente, dicha función no pertenezca al programa. En consecuencia, un proceso irá transitando a lo largo de su ejecución entre dos modos de ejecución:

- *proceso en modo usuario*: el proceso está ejecutando el código del programa correspondiente con el procesador en modo usuario, usando, por tanto, la pila de usuario, y estando las interrupciones habilitadas.
- *proceso en modo sistema* (o núcleo): el proceso está ejecutando código del sistema operativo con el procesador en modo sistema, usando, por tanto, la pila de sistema. Esta situación se debe a que se ha producido un evento que ha causado un cambio de modo. Según el modelo planteado, es el proceso en modo sistema quien realiza el tratamiento del evento. Como se comentó previamente, el evento puede estar vinculado con el proceso (por ejemplo, una llamada al sistema o una excepción de fallo de página causada por el propio proceso) o no (por ejemplo, una interrupción de disco que indica el final de una operación arrancada previamente por otro proceso).

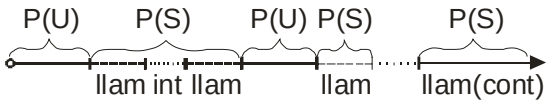


Figura 10 *Traza de ejecución de un proceso alternando modo usuario y modo sistema.*

En la figura 10 se muestra la traza de ejecución de un proceso P que realiza una llamada al sistema que tiene una única fase, durante cuyo tratamiento se produce una interrupción, y, a continuación, realiza una llamada al sistema que requiere un bloqueo (es decir, tiene varias fases). En la figura se puede apreciar que bajo este modelo, toda la ejecución está vinculada a un proceso, ya sea en modo usuario (denotado en la figura como *P(U)*), cuando ejecuta el programa correspondiente, o en modo sistema (mostrado en la figura como *P(S)*), cuando el sistema operativo trata los eventos acaecidos.

Tratamiento de eventos síncronos y asíncronos

Como se ha comentado previamente, el tratamiento de un evento por parte de un proceso en modo sistema va a ser significativamente diferente dependiendo de si es de tipo síncrono o asíncrono.

Un evento síncrono (llamada al sistema o excepción) está vinculado con el proceso en ejecución. Se trata, al fin y al cabo, de una solicitud de servicio por parte del proceso en ejecución (en una excepción el proceso también está "solicitando", aunque sea implícitamente, que el sistema operativo trate esa condición excepcional). Por tanto, es razonable que desde la rutina de tratamiento del evento se realicen operaciones que afecten directamente al proceso en ejecución. Así, desde la rutina de tratamiento se puede acceder al mapa del proceso para leer o escribir información en él. Por ejemplo, una llamada al sistema de lectura de un fichero o dispositivo requiere que su rutina de tratamiento acceda al mapa de memoria del proceso en ejecución para depositar en él los datos solicitados. Asimismo, como se analizó previamente, en el tratamiento de este tipo de eventos puede producirse una parada o bloqueo, con el consiguiente cambio de proceso. Nótese que en este modelo el bloqueo en el tratamiento de un evento se corresponde con el bloqueo del proceso, puesto que es el que se encarga de ejecutar el código de tratamiento. Previamente, en el ejemplo de un manejador de dispositivo se mostró un caso de una llamada al sistema con un bloqueo. Un ejemplo en lo que se refiere al tratamiento de excepciones sería un fallo de página, cuyo tratamiento no puede completarse hasta que finalice la operación del disco que lee la página requerida.

Cuando se realiza el tratamiento de un evento asíncrono (interrupción), como se explicó previamente, aunque se lleve a cabo en el contexto del proceso en ejecución (o sea, el proceso en ejecución ejecute en modo sistema el tratamiento del evento), el proceso no está vinculado con dicho evento. Por tanto, no tiene sentido que se realicen desde la rutina de tratamiento operaciones que afecten directamente al proceso en ejecución, como, por ejemplo, acceder a su mapa de memoria. Asimismo, en el tratamiento de este tipo de eventos no puede producirse un bloqueo, ya que se quedaría bloqueado el proceso en ejecución, que no tiene ninguna vinculación con el evento. Si se analiza la esencia de las interrupciones, esta característica es bastante razonable: una interrupción indica el final de algo, por lo que no parece que tenga sentido que su tratamiento tenga que esperar por algún otro evento.

Como se aprecia en la figura 11, esta distinción en el tratamiento de estos dos tipos de eventos marca incluso una división, aunque sólo sea conceptual, del sistema operativo en dos capas:

- Capa superior: rutinas del sistema operativo vinculadas con el tratamiento de los eventos síncronos (llamadas y excepciones). Se podría decir que esta capa está más "próxima" a los procesos.
- Capa inferior: rutinas del sistema operativo vinculadas con el tratamiento de interrupciones de dispositivos. Se podría decir que esta capa está más "próxima" a los dispositivos.

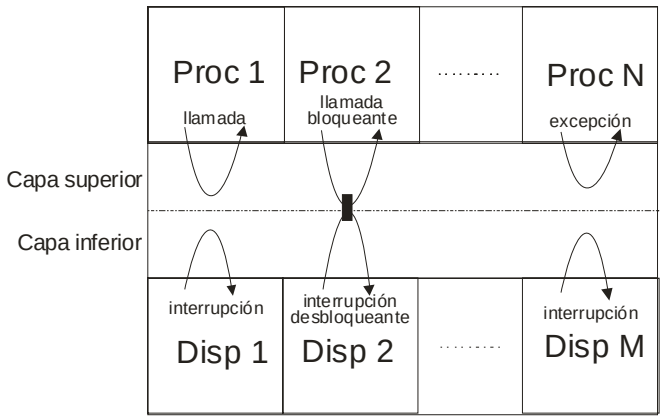


Figura 11 *Relación entre tipos de eventos y organización del sistema operativo.*

Como se explicó anteriormente, cada proceso tiene asociada una pila de sistema que servirá como soporte para la ejecución de las rutinas de tratamiento de los eventos que ocurran durante su ejecución. Cada vez que haya un cambio de proceso, se cambiará de pila de sistema. El tamaño de esta pila debe ser suficiente para almacenar toda la información requerida en el caso del máximo anidamiento de eventos. Este caso corresponde a una situación donde durante una llamada al sistema se produce una excepción y, durante el tratamiento de la misma, se producen todas las interrupciones presentes en el sistema anidadas entre sí. Téngase en cuenta que el espacio de memoria asociado a la pila de sistema de un proceso debe formar parte de la memoria del sistema operativo, inaccesible desde el modo usuario. Gracias a ello y a que el procesador distinga entre la pila de usuario y de sistema, se puede asegurar que no se manipula malintencionadamente el contenido de la pila del sistema, lo que podría causar la ruptura de la seguridad del sistema.

En procesadores que gestionan dos pilas de sistema, los eventos síncronos usan la pila de sistema, propiamente dicha, mientras que los asíncronos utilizan la pila de interrupción. Por tanto, en la primera pila sólo se podrá anidar una llamada al sistema y una excepción, mientras que en la segunda todas las interrupciones disponibles en el sistema. Es interesante resaltar que, dado que durante el tratamiento de una interrupción no puede producirse un cambio de proceso, sólo es necesario tener una pila de interrupción en el sistema (realmente, una por procesador), como se puede apreciar en la figura 12. Dado que este esquema conlleva un menor gasto de memoria, algunos sistemas operativos lo implementan, incluso aunque el procesador no lo proporcione. Se puede implementar por software, incluyendo un cambio de pila al principio de la rutina de tratamiento de un evento asíncrono.

En los siguientes apartados se analiza cómo el sistema operativo maneja los distintos eventos del procesador: interrupciones, excepciones y llamadas al sistema. Además, se presenta un nuevo tipo de interrupción, la interrupción software, y se muestra cómo se usa para diferir la ejecución de operaciones.

Tratamiento de interrupciones

Como se comentó previamente al analizar el tratamiento hardware de las interrupciones, hay dos enfoques básicos a la hora de gestionar múltiples interrupciones, dependiendo de si se implementa un esquema de prioridad, o uno sin prioridad y con enmascaramiento de interrupciones. En este punto, es conveniente resaltar que, con independencia del tipo de gestión que realice el procesador subyacente, el sistema operativo puede establecer su propio modelo, implementándolo por software. Así, por ejemplo, Linux utiliza un modelo sin prioridades, mientras que Windows usa un enfoque basado en prioridades. Cuando se ejecuta Windows sobre un procesador Pentium de Intel, que ofrece una estrategia sin prioridades, es el sistema operativo el que se tiene que encargar de construir por software el modelo de prioridades, asignando la prioridad a las distintas líneas de interrupción según su propio criterio. En cambio, el modelo de interrupciones de Linux encaja con el de este procesador, no requiriendo una adaptación del mismo. Sin embargo, cuando Linux se ejecuta sobre el procesador SPARC, que

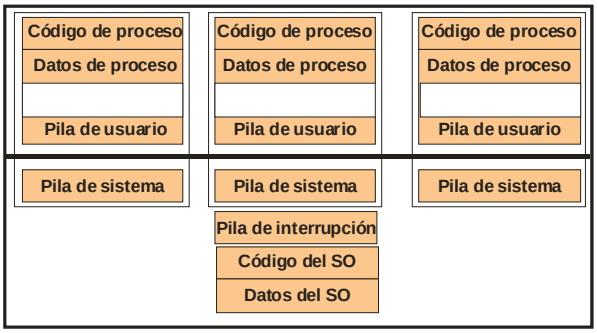


Figura 12 *Organización del sistema basada en el modelo de procesos usando dos pilas de sistema.*

implementa prioridades, el propio sistema operativo debe de encargarse de crear un modelo sin prioridades sobre el hardware subyacente.

A la hora de comparar estos dos modelos de gestión de interrupciones, a priori, puede parecer más razonable el enfoque que distingue niveles de interrupción, ya que modela más fielmente los distintos grados de urgencia en el tratamiento de los dispositivos. Sin embargo, la aplicación de este modelo a un procesador que no lo implemente es bastante artificial, prácticamente arbitraria. Además, como se analizará cuando se estudie el concepto de interrupción software, en la mayoría de los sistemas operativos, las operaciones que se realizan directamente dentro de la rutina de interrupción de cualquier dispositivo quedan reducidas a la mínima expresión, por lo que prácticamente se eliminan los problemas de poder perder interrupciones. Por tanto, ambos modelos son razonables y queda a criterio del diseñador del sistema operativo cuál usar.

En cualquier caso, es importante resaltar ese criterio de diseño reseñado: Se debe intentar minimizar la duración de las rutinas de interrupción y, en general, mantener al procesador ejecutando con todas las interrupciones habilitadas la mayor parte del tiempo, incluso aunque el procesador esté en modo sistema. Con ello, se consigue reducir la **latencia** en el tratamiento de las interrupciones, es decir, el tiempo que transcurre desde que se activa una interrupción hasta que comienza a ejecutarse su rutina de tratamiento. Este objetivo es especialmente relevante en los sistemas de tiempo real, y se retomará más adelante en esta presentación.

Con independencia del tipo de interrupción, se pueden clasificar las operaciones vinculadas con la misma dependiendo de su grado su urgencia y el tipo de conflictos de concurrencia que pueden suceder durante su ejecución:

- Operaciones críticas, que hay que llevarlas a cabo de forma inmediata y que deben realizarse con todas las interrupciones inhibidas. Se trata de acciones necesarias para el buen funcionamiento del sistema y tal que su resultado puede verse comprometido si se admite otra interrupción. Un ejemplo podría ser una operación que actúe sobre el propio controlador de interrupciones del sistema. En algunos sistemas no existen operaciones de estas características.
- Operaciones urgentes, cuyo resultado puede ser erróneo si se produce una interrupción del mismo tipo, pero que no se ven afectadas por otro tipo de interrupciones. Deberán ejecutarse de manera que se impidan interrupciones del mismo tipo mientras se llevan a cabo y tendrán que procesarse con una cierta urgencia para evitar que se pueda perder información. Considere, por ejemplo, la operación de leer el código de la tecla del registro de datos del dispositivo cuando se produce una interrupción de teclado. Si esta operación no se realiza con urgencia, puede ocurrir que en el intervalo que transcurre hasta que se lleve a cabo la operación, se produzca una segunda pulsación en el teclado que escriba un nuevo valor en el registro de datos perdiéndose el valor previo. Además, se debe ejecutar con las interrupciones del mismo tipo inhabilitadas para evitar condiciones de carrera, como se analizará en la sección dedicada a la sincronización.
- Operaciones no urgentes, que pueden ejecutarse con todas las interrupciones habilitadas, ya que no entran en conflicto con sus rutinas de tratamiento. Habitualmente, estas operaciones se realizan fuera del ámbito de la rutina de interrupción, de forma

diferida, usando el concepto de interrupción software que se analiza en la sección etiquetada con esa denominación.

En un esquema de interrupciones sin prioridades, la rutina de tratamiento inicia su ejecución con las interrupciones inhibidas, por lo que en esta primera parte de la rutina de tratamiento se incluirían las operaciones críticas. A continuación, se habilitarían las interrupciones pero enmascarando las interrupciones del mismo tipo durante toda la duración de la rutina, incluyendo en la misma las operaciones urgentes. Las operaciones no urgentes se realizarían fuera de la rutina de interrupción, dentro del tratamiento de una interrupción software.

Con un esquema de interrupciones con niveles, se cumple automáticamente el requisito de que mientras se ejecuta la rutina de interrupción quedan inhibidas las interrupciones del mismo tipo, ya que la rutina ejecuta teniendo sólo habilitadas las interrupciones de nivel superior.

Como se comentó previamente, en un multiprocesador, no basta con inhibir o enmascarar interrupciones, puesto que esas operaciones sólo afectan al procesador local. En la sección dedicada a la sincronización se estudiará qué técnicas se utilizan en este tipo de sistemas.

En cuanto al tratamiento que realiza el sistema operativo de una determinada interrupción, va a depender, evidentemente, de las características específicas de la misma. Como es fácil de entender, no tiene nada que ver, por ejemplo, el procesamiento de la interrupción de reloj con el de la interrupción de disco. Todas estas particularidades se estudian dentro del tema que trata el sistema de entrada/salida. Por ello, este apartado simplemente plantea algunas ideas generales sobre el tratamiento de las interrupciones por parte del sistema operativo. En cualquier caso, a pesar de esta diversidad, existe una parte del tratamiento que es común a todas las interrupciones.

En cualquier sistema operativo, la organización del esquema de tratamiento de las interrupciones debe tener en cuenta los siguientes factores:

- Se debe minimizar el código en ensamblador, manteniendo el mismo dentro de pequeñas rutinas, que realizan su breve labor, y rápidamente invocan funciones escritas en un lenguaje de alto nivel.
- Se debe intentar factorizar la mayor parte del código de tratamiento creando funciones genéricas.
- Se debe permitir que haya varias rutinas de interrupciones asociadas a la misma línea.

Teniendo en cuenta estos requisitos, a continuación, se explica el esquema de gestión de interrupciones que se corresponde, a grandes rasgos, con el utilizado por la mayoría de los sistemas operativos:

- En cada vector de interrupción se instala la dirección de una pequeña rutina codificada en ensamblador que realiza las operaciones que requieren estar programadas a este nivel. Habitualmente, la función se implementa como una macro, que es igual para todas las líneas, diferenciándose únicamente en el número de línea. Cuando se produce la interrupción, el procesador realiza su labor y pasa el control a esta rutina, que salva en la pila (en la de sistema, puesto que es la que está activa) los registros que no se hayan salvado de forma automática e invoca una rutina escrita en un lenguaje de alto nivel, pasándole como parámetro la identidad de la línea de interrupción.
- La rutina de alto nivel, común para todas las interrupciones, además de otras labores, invoca la rutina, o rutinas, específicas asociadas a dicha línea de interrupción. Como se comentó previamente, el sistema operativo permite asociar múltiples rutinas de interrupción a cada línea. El sistema operativo mantendrá una estructura de datos vinculada a cada línea de interrupción que, entre otras cosas, almacenará la lista de rutinas de interrupción asociadas a esa línea.
- Cada rutina específica forma parte del código del manejador del dispositivo y, por tanto, su contenido depende de las características del mismo.
- Cuando terminan todas las rutinas específicas, y, acto seguido, la rutina genérica, prosigue la rutina en ensamblador restaurando los registros almacenados en la pila al principio de la rutina y ejecutando la instrucción de retorno de interrupción.

La figura 13 ilustra este esquema de gestión de interrupciones, característico de los sistemas operativos de propósito general. En ella se representan con líneas discontinuas las relaciones

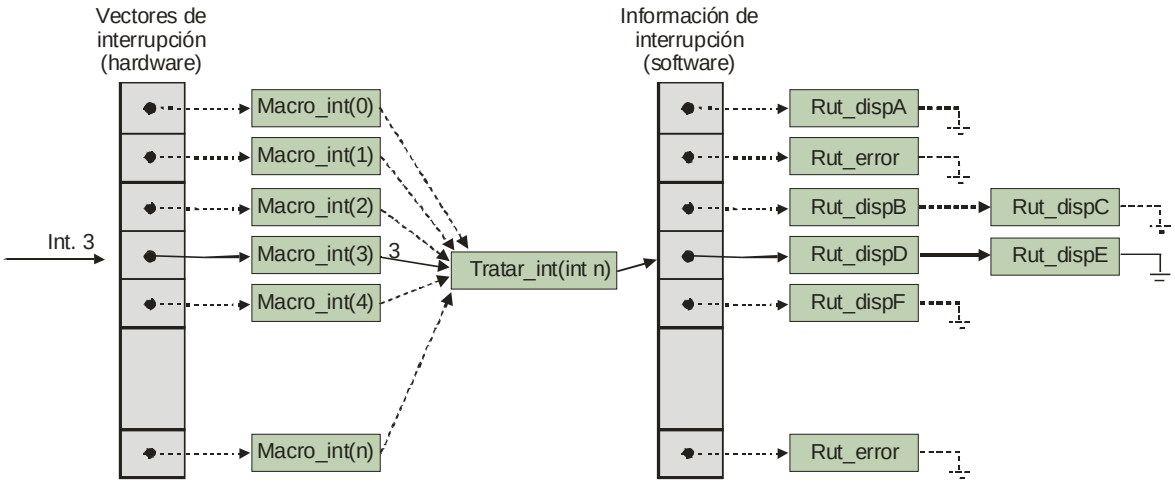


Figura 13 *Esquema de gestión de interrupciones.*

entre las diversas funciones y con líneas continuas aquellas funciones que se activan cuando llega una interrupción por la línea 3. Observe que algunas posiciones no tienen instalada una función de tratamiento, por lo que hacen referencia a una función de error.

Tratamiento de excepciones

El tratamiento de una excepción tiene ciertas similitudes con el de una interrupción: en el vector correspondiente está instalada una pequeña rutina en ensamblador que salva los registros en la pila del sistema e invoca la rutina de alto nivel asociada a la excepción.

Las excepciones indican frecuentemente que se ha producido un error en la ejecución de un programa, por lo que comenzaremos analizando ese caso.

Cuando se produce una excepción que informa de un error, hay que tomar medidas drásticas para resolverlo. El tratamiento que se le da a una excepción va a depender de en qué modo estaba ejecutando el proceso que produjo la excepción.

En el caso de que el modo previo fuera usuario, se estaba ejecutando un programa. Teniendo en cuenta que se trata de un programa erróneo, un posible tratamiento es abortar la ejecución del mismo para permitir que el programador lo depure. Sin embargo, esa suele ser la última solución. La mayoría de los sistemas operativos permiten que el programa especifique cómo desea manejar una excepción. Asimismo, si la ejecución del programa está bajo la supervisión de un depurador, éste deberá ser notificado de la aparición del evento. Recapitulando, el sistema operativo suele llevar a cabo una estrategia como la siguiente:

- Si el programa está siendo depurado, se detiene la ejecución del proceso y se notifica al proceso depurador que ha ocurrido la excepción para que la trate como considere oportuno facilitando al programador la depuración del programa erróneo.
- Si no está siendo depurado, se comprueba si el programa ha declarado que quiere manejar la excepción, en cuyo caso se activa la ejecución de ese tratamiento.
- En caso contrario, se aborta el proceso.

A continuación, se describe cómo es el tratamiento de las excepciones en los sistemas UNIX. En este sistema operativo, el manejo de las excepciones está englobado dentro del mecanismo general de las señales. Cada excepción tiene asociada una señal y la rutina de tratamiento de la excepción se encarga de generarla. Por ejemplo, una excepción aritmética se corresponde con la señal SIGFPE, la ejecución de una instrucción no permitida en modo usuario causa la señal SIGILL, y un punto de ruptura, que es un caso de excepción no vinculada con una situación de error, se proyecta en la señal SIGTRAP. Como conoce el lector, el programa puede establecer qué tipo de tratamiento desea llevar a cabo con respecto a una determinada señal (mediante la llamada `sigaction`), existiendo tres posibilidades: capturarla para tratarla dentro del programa mediante la rutina especificada, ignorarla o mantener el comportamiento por defecto, que consiste generalmente en abortar el proceso. En cualquier caso, sea cual sea el tratamiento, si el sistema operativo detecta que el proceso está siendo supervisado por un

depurador (que habrá usado para ello la llamada `ptrace`) cuando le llega una señal, detiene la ejecución del proceso y le envía la señal SIGCLD al depurador para notificarle del hecho.

En Windows, se proporciona un esquema de gestión de excepciones más sofisticado, que no va a ser tratado en esta presentación. Simplemente, se resalta en este punto que se trata de un manejo de excepciones estructurado, similar al disponible en lenguajes como C++ y Java.

En caso de que el modo de ejecución previo fuera sistema, el error estaba en el código del propio sistema operativo. Se trata, por tanto, de una situación crítica, como queda reflejado en el hecho de que en algunos entornos se denomine a esta situación "pánico en el sistema". Dependiendo de la gravedad de la situación, puede bastar con abortar el proceso que está actualmente en ejecución o puede ser más conveniente detener de forma ordenada, si es posible, la ejecución del sistema operativo para evitar que pueda corromperse alguna información del sistema debido a este error. En cualquier caso, el tratamiento incluye habitualmente imprimir en la consola información que posibilite una posible depuración posterior del problema (por ejemplo, se imprimirá el valor de los registros del procesador en el momento de producirse la excepción). Hay que resaltar que el tratamiento que hace el sistema operativo de situaciones en las que detecta alguna incoherencia en alguna estructura de datos interna es similar al realizado con las excepciones.

Como se ha comentado previamente, no todas las excepciones están vinculadas con condiciones de error. Además de las excepciones vinculadas con la depuración, es de especial interés la excepción de fallo página. El tratamiento de la misma se estudiará en el tema de gestión de memoria, pero en este punto se puede resaltar que es considerablemente distinto al resto de las excepciones:

- En modo usuario, una excepción de fallo de página puede corresponderse con un acceso a memoria inválido (en UNIX, una señal SEGV). Sin embargo, puede no estar asociado a una situación errónea, sino indicar que una página no está residente y es, por tanto, la base de la paginación por demanda.
- En modo sistema, una excepción de fallo de página puede indicar un error en el código del sistema operativo, con la consiguiente situación de pánico. Sin embargo, puede no tratarse de un error, sino de un acceso desde el sistema operativo a una página no residente del mapa de usuario del proceso. Este acceso debería realizarse desde el tratamiento de un evento síncrono, ya que, como se comentó previamente, desde una rutina de interrupción no se debe acceder al mapa de usuario del proceso. Por tanto, una excepción durante la ejecución de una rutina de interrupción indica un error en el código del sistema operativo.

Tratamiento de llamadas

Generalmente, los sistemas operativos dedican sólo un vector para todas las llamadas al sistema. Puede parecer una estrategia un poco extraña: normalmente, en un sistema suele haber vectores libres, ¿por qué no dedicar uno a cada llamada? Como ya se ha comentado con anterioridad, el diseñador de un sistema operativo debe usar siempre una política de "requisitos mínimos". Exigir que el procesador tenga tantos vectores libres como llamadas al sistema tiene el sistema operativo puede implicar que dicho sistema operativo no se pueda transportar a ciertos procesadores. Dado que no parece razonable tener un vector para cada llamada, la solución más sencilla es usar un único vector para todas las llamadas. Así, por ejemplo, en plataformas con el procesador Pentium de Intel, si se usa la instrucción *INT* para realizar la llamada, tanto Linux (vector 80) como Windows (vector 2e) usan este criterio. Además, en algunos procesadores ni siquiera se usa el mecanismo de vectores, como ocurre con las instrucciones *SYSENTER* y *SYSEXIT* del Pentium.

Al usar un único vector, o incluso no usar ninguno, no se puede aplicar este mecanismo del procesador para activar la rutina de tratamiento de una determinada llamada. Es necesario determinar por software de qué llamada se trata y cuál es su rutina de tratamiento. La estrategia utilizada por la mayoría de los sistemas operativos imita hasta cierto punto el mecanismo de vectores del procesador, como se explica a continuación:

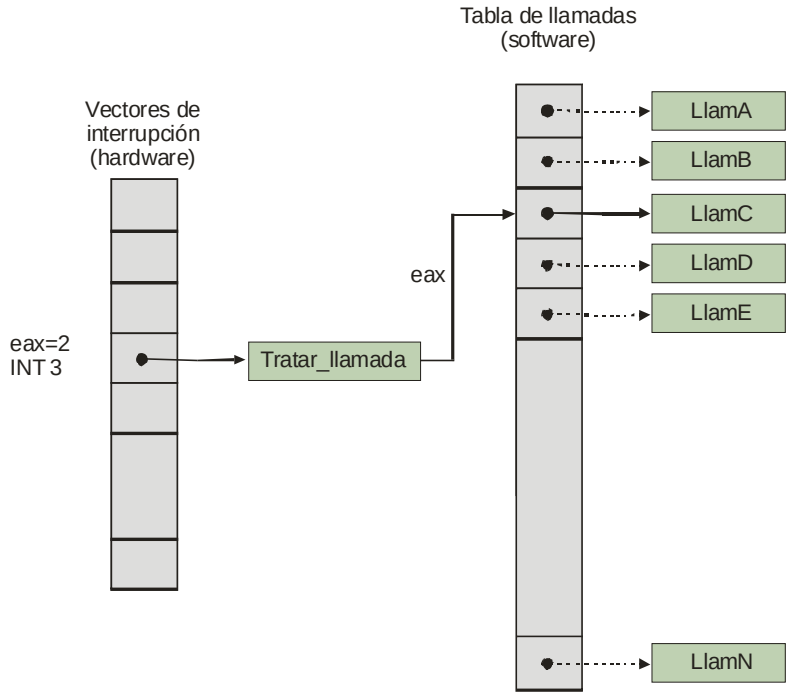


Figura 14 *Esquema de tratamiento de llamadas al sistema.*

- El sistema operativo mantiene una tabla con las rutinas de tratamiento de cada una de las llamadas al sistema. Nótese que es similar a la tabla de vectores de interrupción, pero en este caso se trata de una estructura de datos gestionada totalmente por software. Además, las rutinas incluidas en esta tabla, a diferencia de lo que ocurre con las de la tabla de vectores de interrupción, son funciones que terminan con una instrucción de retorno convencional y no con una de retorno de interrupción.
- Cada llamada al sistema tiene asignado un número (su "vector") que debe corresponder a la posición de la tabla de llamadas que contiene su rutina de tratamiento. En Linux, por ejemplo, la llamada *fork* tiene asociado el valor 5 y en Windows la llamada *NtWriteFile* tiene el 14. A la hora de hacer una llamada al sistema, el proceso debe especificar de alguna forma cuál es el número de la llamada que desea invocar. Normalmente, se usa un registro general del procesador para esta labor. Por ejemplo, en plataformas Intel, tanto Linux como Windows usan el registro *eax* para pasar el número de la llamada. La figura 14 ilustra el esquema de tratamiento de llamadas, mostrando un ejemplo de llamada donde se usa el vector 3 para todas las llamadas al sistema y se está invocando la número 2.
- A diferencia de los otros eventos del procesador, la llamada al sistema requiere parámetros y devuelve un resultado. Con respecto a este último, algunos sistemas operativos usan un registro como medio para devolver el resultado. Por ejemplo, en Linux se usa el mismo registro *eax* en el que se indicó el número de la llamada.
- En cuanto a los parámetros, hay típicamente dos opciones:
    - Usar registros generales para transferir los parámetros. Este es el caso de Linux que, en procesadores Pentium de Intel, usa los siguientes 5 registros para esta labor: *ebx*, *ecx*, *edx*, *esi* y *edi*. Esta solución es sencilla aunque requiere que se disponga de varios registros y obliga a que, si una llamada tiene más de cinco parámetros (por ejemplo, la llamada *mmap*), se deba usar en este caso un método alternativo usando la pila de usuario, que se corresponde precisamente con la otra opción.
    - Usar la pila de usuario para pasar los parámetros. Este es el método usado por Windows. Con esta estrategia pueden ocurrir situaciones más complejas como, por ejemplo, que la página de la pila de usuario donde residen los parámetros no esté residente en memoria cuando el sistema operativo acceda a estos parámetros. Para simplificar el tratamiento, en Windows, la rutina que trata el evento de llamada al

sistema comienza copiando los parámetros de la pila de usuario a la de sistema, de manera que se puedan acceder sin problemas en el resto de la llamada.

Hechas estas consideraciones previas, queda claro cómo debe ser la estructura de la rutina del sistema operativo que trata el evento de llamada al sistema.

- En primer lugar, al igual que ocurre con las otras rutinas que tratan eventos, debe salvar los registros correspondientes en la pila de sistema. Nótese que si se ha optado por una estrategia de pasar los parámetros de la llamada en registros, con esta operación los parámetros quedan en la pila del sistema.
- Si los parámetros se pasan en la pila de usuario, para simplificar el tratamiento, los copia a la pila del sistema. Si se compara con el punto anterior, se puede apreciar que sea cual sea la estrategia usada para pasar los parámetros, éstos quedan finalmente en la pila de sistema.
- Obtiene del registro convenido (por ejemplo, el *eax* en Intel) el número de la llamada y lo usa para indexar en la tabla de llamadas al sistema, invocando la rutina almacenada en la posición de la tabla que se corresponde con el número recibido.
- Esta rutina específica podrá obtener sus parámetros en la pila de sistema.
- Al terminar la rutina específica, se reanuda la rutina de tratamiento del evento restaurando los registros almacenados en la pila al principio de la rutina, almacenando el resultado en el registro convenido y ejecutando la instrucción de retorno del tratamiento del evento.

Hay que tener en cuenta que algunos de los parámetros de la llamada pueden ser punteros: direcciones de memoria del mapa de usuario del proceso a las que el sistema operativo tiene que acceder para obtener o depositar información. Posibles ejemplos de este tipo de parámetros serían: la dirección de la cadena de caracteres que corresponde al nombre de un fichero en una llamada de apertura del mismo o la dirección de una zona donde se quiere que el sistema operativo deje los datos en una llamada de lectura de un fichero.

Ya sea por error o intencionadamente, el programa que solicita la llamada puede especificar valores erróneos para esos parámetros de tipo puntero. El sistema operativo debe detectar esta situación y tratarla adecuadamente, devolviendo un error en la llamada. Hay dos estrategias para llevar a cabo esta detección de error:

- Una política que podríamos calificar como “pesimista”, que es la utilizada en Windows. Con esta estrategia, se comprueba que los valores de los punteros son correctos (corresponden a direcciones que pertenecen a una región de memoria del proceso, que, además, tiene los permisos adecuados para el tipo de operación que se pretende realizar) antes de procesar la llamada al sistema.
- Una política de tipo “optimista”, que corresponde a la usada en Linux. En este caso, no se comprueba la correcta accesibilidad de los parámetros de tipo puntero que pueda tener una llamada, sino que se comienza a procesarla suponiendo que son correctos. Si alguno no lo es, se producirá una excepción de acceso a memoria. El código de tratamiento de esta excepción se ha extendido para que tenga en cuenta este caso particular: si se produce una excepción de memoria y se detecta que el proceso estaba previamente en modo sistema, se comprueba si el proceso estaba en ese momento accediendo a los parámetros. Si es así, se termina la llamada al sistema devolviendo un error. En caso contrario, se trata de una situación de pánico, que ya se analizó previamente. Queda un último aspecto por aclarar: ¿cómo se determina que el proceso en modo sistema estaba accediendo a un parámetro de tipo puntero en el momento de producirse la excepción? Para ello, en Linux se usa una tabla donde se almacenan las direcciones de todas las instrucciones dentro del código del sistema operativo que realizan accesos de este tipo. Dentro del código de tratamiento de la excepción de acceso a memoria se comprueba si el valor del contador de programa en el momento de producirse la excepción se corresponde con algún valor almacenado en la tabla. Si es así, se trata de una excepción debido a un parámetro de tipo puntero erróneo. En caso contrario, se trata de una situación de pánico en el sistema. Para terminar la explicación de en qué consiste esta estrategia, hay que hacer una aclaración. Realmente, sí es necesario realizar una comprobación a priori: hay que asegurarse de que no son direcciones de sistema puesto que en ese caso el sistema operativo estaría accediendo o modificando su propio mapa.

Comparando las dos estrategias, la optimista es más eficiente, ya que cuando los parámetros son correctos, lo que ocurrirá la mayoría de las veces, no incurre en ninguna sobrecarga para comprobarlo. Sin embargo, puede complicarse la programación de las llamadas al sistema que tengan parámetros de este tipo, ya que la excepción producida por un parámetro erróneo se produce en medio de la llamada al sistema y hay que cancelar cuidadosamente todo el trabajo que ya haya hecho la llamada hasta ese momento.

Invocación de llamadas

Como se ha explicado en el apartado previo, para realizar una llamada al sistema, un programa tiene que realizar las siguientes operaciones: escribir el número de la llamada en el registro convenido, asegurarse de que los parámetros de la llamada estén en la pila de usuario o en los registros, dependiendo del método usado, y ejecutar la instrucción que causa el evento de llamada al sistema. Al terminar el tratamiento de la llamada, el programa debe recoger del registro convenido el resultado de la misma.

Evidentemente, no tiene sentido obligar a que un programa tenga que hacer todas esas operaciones, que, además habría que hacerlas usando ensamblador, cada vez que quiera hacer una llamada al sistema. En todos los sistemas operativos se utiliza una solución similar: proporcionar una serie de rutinas de biblioteca que ofrecen una interfaz funcional para realizar llamadas al sistema. Se trata de funciones de envoltura que simplemente incorporan la lógica planteada en el párrafo anterior.

A continuación, se plantea un ejemplo hipotético en el que se usa el registro 0 para especificar el código de la llamada y para recoger el resultado. En la primera versión, se supone que el parámetro se pasa en el registro 1:

```
close(int desc){
        MOVE R0, #NUM_CLOSE
        MOVE R1, desc
        INT 0x80
        Valor devuelto está en R0
        RET
}
```

En esta segunda, los parámetros se pasan en la pila de usuario. Nótese que no es necesario realizar ninguna operación explícita para incluir los parámetros en la pila, puesto que ya están en ella como resultado de la invocación de la función de envoltura.

```
close(int desc){
        MOVE R0, #NUM_CLOSE
        INT 0x80
        Valor devuelto está en R0
        RET
}
```

El sistema operativo sólo necesita saber en qué desplazamiento con respecto a la cima de la pila de usuario se encuentran los parámetros. Puede determinarse un cierto desplazamiento fijo sabiendo cómo es el patrón de llamadas. Sin embargo, es más flexible usar una estrategia como la que utiliza Windows, donde se le pasa en el registro *ebx* cuál es la posición de los parámetros dentro de la pila de usuario.

Es interesante resaltar que en sistemas UNIX, en caso de error, la mayoría de las llamadas devuelven un –1 y actualizan la variable global *errno* con un valor que indica el tipo de error. Normalmente, es la función de envoltura la que se encarga de hacer esto. La llamada al sistema real suele devolver un solo valor en un registro que, en caso de error, consiste en un número negativo cuyo valor absoluto indica el tipo de error. En ese caso, la función de envoltura almacena en *errno* el valor absoluto de ese código de error y devuelve un valor de –1.

Interrupciones software

Como se irá mostrando a lo largo de este capítulo, en el código del sistema operativo es bastante frecuente encontrarse con situaciones en las que se requiere diferir la realización de un conjunto de operaciones asociadas a un evento, de forma que se ejecuten en un ámbito distinto al de la rutina de tratamiento de dicho evento, cuando se cumplan las condiciones requeridas por las mismas. A continuación, vamos a plantear dos ejemplos de este tipo de situaciones.

Como se explicó en la sección sobre el tratamiento de las interrupciones, suele existir una serie de operaciones "no urgentes" asociadas con una interrupción, que se pueden ejecutar fuera del ámbito de la propia rutina de interrupción, en un contexto en el que todas las interrupciones estén habilitadas. En el caso, por ejemplo, de una interrupción del teclado, una vez leído el código de la tecla, el resto de las operaciones (principalmente, la traducción de ese código a un carácter teniendo en cuenta la configuración del teclado y el estado de las teclas modificadoras), que son más pesadas, pueden ejecutarse posteriormente, en un contexto con las interrupciones habilitadas, pero sólo cuando se hayan completado todas las rutinas de interrupción que puedan estar anidadas en ese momento.

Un segundo ejemplo aparece cuando se produce una interrupción de un disco y en la rutina de tratamiento se desbloquea un proceso que es más prioritario que alguno de los que está actualmente en ejecución (sólo uno, en el caso de un monoprocesador). En esta situación, hay que realizar una operación de cambio de proceso expulsando al proceso menos prioritario, pero esta expulsión no se puede llevar a cabo hasta que se completen todas las rutinas de interrupción que puedan estar anidadas en ese momento (y, en algunos casos, como se verá en la sección dedicada a la planificación, hasta que se complete la llamada al sistema en curso, en caso de que la hubiera), puesto que de no ser así éstas quedarían incompletas.

Este tipo de comportamiento se podría programar específicamente en cada uno de los escenarios que sigan este esquema, guardando para ello información de que hay operaciones diferidas pendientes de ejecutar en alguna variable del sistema operativo que se consultaría en los puntos adecuados para comprobar si hay trabajo pendiente, llevándolo a cabo en caso de que lo hubiera. Sin embargo, dada su frecuente aparición, sería conveniente tener un soporte directo del mismo.

Si analizamos el modo de operación de los ejemplos, podemos ver que es similar a lo que ocurre cuando se produce una interrupción de baja prioridad de un dispositivo mientras se está sirviendo una de mayor prioridad: el trabajo de baja prioridad no se realiza hasta que se completen todas las rutinas de mayor prioridad que estén anidadas en ese momento. Por tanto, lo que se requiere para dar cobertura a esta necesidad es un mecanismo que permita solicitar una interrupción de baja prioridad pero por software, surgiendo así el concepto de **interrupción software**.

Como su nombre indica, se trata de un mecanismo que permite activar una interrupción por software, frente a las interrupciones convencionales, que son generadas por dispositivos hardware. Evidentemente, el mecanismo de llamada al sistema se puede considerar una interrupción software. Sin embargo, asumiendo esa acepción, en esta exposición vamos a usar este término para denominar a otro mecanismo, que también permite activar una interrupción por software, pero estando previamente en modo sistema en vez de en modo usuario, como ocurre en el caso de las llamadas al sistema. Esta distinción de uso conlleva un modo de operación con diferencias apreciables. Las llamadas al sistema son síncronas: dado que se producen con el procesador en modo usuario, se tratan inmediatamente. Las interrupciones software tienen un carácter asíncrono. Se invocan en modo sistema por lo que, si el estado actual de las interrupciones del procesador lo impide, su tratamiento se retardará hasta que las circunstancias lo permitan.

Este mecanismo permite implementar de una manera conceptualmente sencilla la técnica de diferir la realización de un conjunto de operaciones asociadas a un evento, de forma que se ejecuten en un ámbito distinto al de la rutina de tratamiento de dicho evento, cuando se cumplan las condiciones requeridas por las mismas. Con el mecanismo de interrupción software, en los ejemplos planteados bastaría con activar una interrupción software dentro de la rutina de interrupción del teclado o del disco, respectivamente, e incluir en la rutina de tratamiento de la misma las operaciones no urgentes del teclado, en un caso, y el cambio de proceso, en el otro.

Hay que resaltar que la interrupción software no es un mecanismo imprescindible para un sistema operativo, pero la existencia del mismo proporciona una técnica común para resolver todas las situaciones donde se requiere la ejecución diferida de operaciones, sean del tipo que sean, en vez de tener que implementar una solución específica para cada caso.

Aunque algunos procesadores ofrecen algún tipo de soporte para este mecanismo, la mayoría no lo hace, siendo implementado completamente por el sistema operativo. En cualquier caso, sea cual sea su implementación, su modo de operación es el mismo, tal como se explica en esta sección, y en el resto del capítulo se considera que existe en el sistema un mecanismo de interrupción software.

Retomando los dos ejemplos planteados, existen algunas diferencias significativas en el uso de la interrupción software en cada caso. En el primero, el ámbito de la interrupción software es global, afectando a todo el sistema con independencia de qué proceso esté ejecutando. En cuanto se cumplan las condiciones requeridas, se ejecutará la rutina de tratamiento de la interrupción software. En el segundo caso, sin embargo, la interrupción software va dirigida a un proceso específico, que puede estar ejecutando en un procesador distinto a aquél donde se activa la interrupción software. Incluso, como se verá más adelante cuando se presenten otras situaciones en las que se usan interrupciones software de proceso, el proceso al que va dirigida una interrupción software de este tipo podría no estar ni siquiera en ejecución en ese instante. Por tanto, este tipo de interrupción software está ligada a un proceso y su rutina de tratamiento se ejecutará cuando se cumplan las condiciones requeridas, pero sólo en el momento en que esté en ejecución el proceso vinculado con la misma.

Se distinguen, por tanto, dos tipos de interrupciones software: a la primera la denominaremos *interrupción software de sistema*, mientras que a la segunda *interrupción software de proceso*. En las siguientes secciones se estudian con más detalle estos dos tipos de interrupción software.

Interrupciones software de sistema

El mecanismo de interrupción software de sistema, dado su carácter global e independiente del proceso que está en ejecución, va a utilizarse básicamente para diferir la ejecución de las operaciones no urgentes asociadas a las distintas interrupciones de los dispositivos (reloj, red, terminal, disco, etc.).

Antes de pasar a explicar esta técnica, hay que recordar un criterio fundamental de diseño del sistema operativo: se debe intentar minimizar la duración de las rutinas de interrupción y, en general, mantener el procesador ejecutando con todas las interrupciones habilitadas la mayor parte del tiempo. Alcanzar este objetivo tiene como consecuencia un sistema mucho más ágil y con un mejor tiempo de respuesta. Para ilustrarlo con un ejemplo concreto, considere que durante un tiempo apreciable está inhibida la interrupción del ratón: el usuario notará que el sistema no le responde con la suficiente rapidez a las peticiones que realiza con este dispositivo.

El encaje de la interrupción software de sistema con el resto del sistema de interrupciones va a depender del modelo de interrupciones que use el sistema.

En el caso de un sistema que gestiona niveles de prioridad, esta interrupción tendría un nivel menor que todas las interrupciones de los dispositivos y, por lo demás, se integraría de forma natural en este tipo de esquema de gestión de interrupciones, pudiéndose inhabilitar selectivamente haciendo que el nivel de interrupción sea igual que el nivel asignado a este tipo de interrupción.

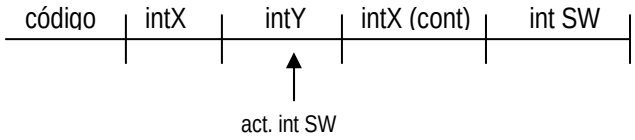
En un sistema con un esquema sin niveles de prioridad, la interrupción software de sistema sí tendría prioridad, inferior a las de los dispositivos, no pudiendo, por tanto, interrumpir la ejecución de la rutina de tratamiento de una interrupción convencional. La prohibición de las interrupciones incluiría también a la interrupción software de sistema. Además, debería existir una operación para poder inhabilitar específicamente la interrupción software de sistema manteniendo habilitadas las demás interrupciones (en Linux, por razones históricas, esta función se denomina se denomina `local_bh_disable`).

El programa 6 muestra un ejemplo del uso de la interrupción software de sistema. En el mismo, se está ejecutando inicialmente código con todas las interrupciones habilitadas, pudiendo tratarse de un programa o del tratamiento de un evento síncrono. A continuación, durante la ejecución de ese código, se activa una interrupción y comienza a ejecutar su rutina de

tratamiento (`intX`). En mitad de la misma, se activa otra rutina de interrupción (`intY`). Supóngase que la primera interrupción no tiene asociadas operaciones diferidas, pero la segunda sí las incluye. Dentro de esta última rutina (`intY`) se activa una interrupción software de sistema, cuya ejecución, dado que tiene menos prioridad que la asociada a ambas rutinas de interrupción, se aplazará hasta que terminen. En ese momento, al retornar de la rutina de interrupción `intX`, se activa la rutina de tratamiento de la interrupción software de sistema donde se llevarán a cabo las operaciones pendientes de ejecutar.

Programa 6 Ejemplo de uso de la interrupción software de sistema.

```
intX() {
      .......
      .......
}

intY() {
      .......
      activa_int_software_sistema();
      .......
}

// rutina de tratamiento de la interrupción software de sistema
tratamiento_int_software_sistema() {
      Realiza la operación diferida
}
```



Para implementar este mecanismo por software, se puede reflejar la activación de la interrupción software de sistema en una variable global (en un sistema multiprocesador, debería haber una variable por procesador). Cada vez que termine una rutina de tratamiento de una interrupción hardware, se debe comprobar si se va a retornar a modo usuario o a la rutina de tratamiento de un evento síncrono (para ello, basta con examinar el registro de estado almacenado en la pila de sistema). En el caso de que así sea y la variable indique que está activada una interrupción software de sistema, en vez de retornar, se ejecuta la rutina de tratamiento de esa interrupción software con todas las interrupciones hardware habilitadas. A continuación, en el programa 7, se muestra de forma simplificada cómo se llevaría a cabo la implementación de este mecanismo de interrupción software de sistema tomando como base el ejemplo anterior.

Programa 7 Implementación de una interrupción software.

```
intX() {
      .......
      rutina_fin_int();
}
intY() {
      .......
      activa_int_software_sistema();
      .......
      rutina_fin_int();
}
// rutina de tratamiento de la interrupción software de sistema
tratamiento_int_software_sistema() {
      Realiza la operación diferida
}

// código de implementación de interrupción software de sistema
activa_int_software_sistema () {
```

```
        int_SW_pendiente = true;
        .......
}
rutina_fin_int () {
        Si (int_SW_pendiente Y
              se va a retornar a modo usuario O
              al tratamiento de evento síncrono) {
                    int_SW_pendiente = false;
                    Habilitar interrupciones;
                    /* invoca rutina de tratamiento de int. SW */
                    tratamiento_int_software_sistema();
        }
}
```

Algunos sistemas proporcionan varios tipos de interrupciones software de sistema que corresponden a cada uno de los distintos dispositivos del sistema que requieren este mecanismo. Estas distintas interrupciones software de sistema pueden poseer prioridades diferentes. Para implementar esta funcionalidad, bastaría con dedicar un bit de una variable global a cada interrupción software y, al retornar a modo usuario o al tratamiento de un evento síncrono, comprobar el estado de cada bit en el orden definido por las prioridades de las respectivas interrupciones software.

En algunos sistemas sólo hay un único tipo de interrupción software de sistema, pero se gestiona una lista de tareas pendientes asociadas a dicha interrupción (de manera similar a cómo se asociaban distintas rutinas de tratamiento a la misma interrupción hardware), que puede estar ordenada por prioridad. De esta manera, la interrupción hardware de un determinado dispositivo simplemente encola su trabajo diferido en esa lista, almacenando normalmente la identificación de una función que llevará a cabo sus operaciones pendientes, así como un dato que se le pasará como parámetro a la función cuando sea invocada. En la rutina de tratamiento de la interrupción software de sistema se vaciará la cola de trabajos invocando a cada una de las funciones encoladas.

El sistema operativo Windows sigue este modelo proporcionando una única interrupción software de sistema denominada DPC (*Deferred Procedure Call*).

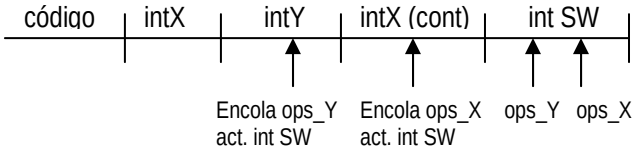
En cambio, en Linux existen múltiples tipos de interrupción software de sistema, denominadas *softirqs*, cada una con una prioridad diferente (actualmente hay 10 tipos, que implementan las operaciones diferidas vinculadas a dispositivos como el temporizador o la red). Sin embargo, dado que el número de *softirqs* es fijo (hay que recompilar el sistema operativo para cambiarlo) y existe una gran variedad de dispositivos, dos de las *softirqs* no están dedicadas a ningún dispositivo en concreto, sino a poder encolar trabajos pendientes en ellas (denominados *tasklets*), de manera similar a lo que ocurre con la DPC de Windows.

En el programa 8 se muestran, de forma esquemática y simplificada, estos dos esquemas (interrupción software de sistema única versus múltiple) en una situación en la que dos rutinas de interrupción anidadas tienen asociadas operaciones diferidas.

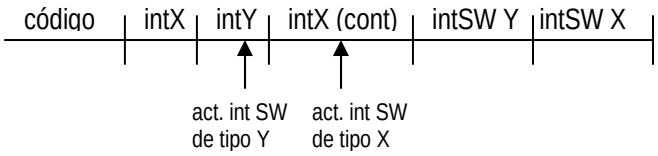
Programa 8 Operaciones de interrupción diferidas mediante interrupción software de sistema.

```
// versión con interrupción software de sistema única

intX() {
        Operaciones críticas y urgentes
        Encola operación (ops_X, datos_X)
        activa_int_software_sistema();
}

intY() {
        Operaciones críticas y urgentes
        Encola operación (ops_Y, datos_Y)
        activa_int_software_sistema();
```

```
}

ops_X() {
      ........
}
ops_Y() {
      ........
}
tratamiento_int_software_sistema() {
      Mientras (haya operaciones diferidas) {
            Extraer operación → (func, datos)
            func(datos);
      }
}
```




```
// versión con múltiples interrupciones software de sistema

intX() {
      Operaciones críticas y urgentes
      activa_int_software_sistema(DISPO_X);
}

intY() {
      Operaciones críticas y urgentes
      activa_int_software_sistema(DISPO_Y);
}
trat_int_SW_sist_X() {
      .....
}
trat_int_SW_sist_Y() {
      .....
}
```



Analizando la traza de ejecución de ambos ejemplos, se puede interpretar que con este mecanismo se prioriza a las operaciones críticas y urgentes de una interrupción frente a las no urgentes de otra, aunque esta última tenga mayor prioridad, lo cual es bastante razonable.

Es interesante resaltar que mientras haya operaciones pendientes de realizar, éstas se seguirán ejecutando, no pudiendo continuar la ejecución de los procesos. En algunos sistemas se ha detectado que en ciertas ocasiones la tasa de interrupciones software puede ser excesiva, "congelando" durante un intervalo la ejecución de los procesos (por ejemplo, en situaciones con un tráfico muy elevado en una red de altas prestaciones). Para paliar este problema, algunos sistemas operativos, como Linux, establecen un cierto umbral a partir del cual algunas de esas operaciones pendientes pasan a ejecutarse en el ámbito de un proceso de núcleo (concepto que se explicará más adelante) de una prioridad media, que intercalará su ejecución con el resto de los procesos (ese proceso de núcleo de Linux se denomina *ksoftirqd*).

Para concluir esta sección, es importante resaltar que, dado el carácter asíncrono y global de las interrupciones software de sistema, que no están vinculadas con ningún proceso, desde una rutina de interrupción software de sistema no se debe acceder al mapa del proceso actualmente en ejecución, ni cambiar el proceso en ejecución.

Dado que se trata de un nuevo tipo de evento, en algunos sistemas operativos se llegan a usar tres pilas para gestionar los eventos del sistema:

- La pila de sistema para los eventos síncronos, habiendo una por cada proceso.
- La pila de interrupciones hardware, necesitándose una por cada procesador.
- La pila de interrupciones software de sistema para tratar este tipo de eventos, requiriéndose sólo una por cada procesador, dado que no puede haber un cambio de proceso dentro del tratamiento de una interrupción software de sistema.

Interrupciones software de proceso

Como se comentó al principio de la sección, existe otra clase de interrupción software, que va dirigida a un determinado proceso (*interrupción software de proceso*), en vez de al sistema global. Cuando se activa una interrupción software de este tipo dirigida a un proceso, sólo se tratará cuando se cumplan las condiciones requeridas (por ejemplo, cuando se vaya a retornar a modo usuario o al tratamiento de un evento síncrono) y, además, esté en ejecución dicho proceso.

La implementación de este tipo de interrupción requiere que haya una variable por proceso (en el BCP), en vez de una en el sistema, donde se anote que hay una interrupción de este tipo pendiente. Cuando se satisfagan las condiciones requeridas, se consultará el valor de dicha variable, pero del proceso que está en ejecución y, si está activa, se tratará la interrupción software de proceso correspondiente.

Nótese que, dado que una interrupción software de proceso, como su nombre indica, está asociada a un determinado proceso y no a todo el sistema, desde su rutina de tratamiento se puede acceder al mapa del proceso actualmente en ejecución y se puede producir un cambio de proceso durante el tratamiento de la misma, puesto que se sabe a priori qué proceso estará en ejecución cuando se active su rutina de tratamiento. Asimismo, en caso de existir pilas de sistema y de interrupción separadas, este tipo de evento utilizará la pila de sistema del proceso.

Entre las diversas aplicaciones de este mecanismo, se pueden destacar las tres siguientes:

- La realización de labores de planificación, en concreto, para llevar a cabo la operación de expulsión de un proceso del uso del procesador (cambio de contexto involuntario). Como se analizará en la sección que estudia cómo se realizan los cambios de contexto, para forzar la expulsión del proceso se le enviará una interrupción software de proceso destinada para ese propósito.
- La terminación involuntaria de un proceso. Como se verá en la sección correspondiente, para abortar un proceso se le envía una interrupción software de proceso dedicada a tal fin. La utilización de una interrupción software de proceso para este propósito permite que la terminación del proceso no se haga de forma abrupta, dándole la posibilidad de que complete todo el trabajo de sistema que pudiera estar realizando en ese momento.
- La implementación de operaciones de entrada/salida asíncronas. Sin entrar en detalles sobre este tipo de operaciones, que quedan fuera del alcance de esta exposición, es conveniente explicar por qué motivo es útil este mecanismo para aplicarlo a ese tipo de operaciones. En una operación asíncrona, ésta transcurre mientras el proceso puede seguir ejecutando. Cuando se complete una operación asíncrona de lectura, por ejemplo, debe ser el propio proceso el que realice la copia de los datos leídos desde algún *buffer* del sistema a la zona correspondiente del mapa del proceso. Para conseguir que el proceso realice esta operación final de copia, se le envía una interrupción software de proceso al completarse la operación de entrada/salida.

El sistema operativo implementará distintos tipos de interrupciones software de proceso para las diversas labores que así lo requieran. Además, proporcionará operaciones para inhabilitar temporalmente cada uno de los tipos de interrupción software de proceso disponibles. Con respecto a la convivencia entre las interrupciones software de sistema y las de proceso, las primeras tienen prioridad sobre las segundas, puesto que en un sistema operativo de propósito general es más importante dar servicio a los dispositivos que a los procesos (en Windows la prioridad de DPC es mayor que la de APC, que presentaremos a continuación).

Este mecanismo está presente, de una forma u otra, en cualquier sistema operativo. En Windows, está integrado en el mecanismo general de niveles interrupción: la interrupción software llamada *Dispatch* está vinculada con la planificación de procesos, mientras que la denominada APC (*Asynchronous Procedure Call*) se usa, entre otras labores, para la terminación involuntaria de procesos y para la implementación de operaciones asíncronas. En este sistema para inhibir un determinado tipo de interrupción software de proceso basta con fijar el nivel de interrupción al valor correspondiente.

En cuanto a Linux, aunque no se usa este término explícitamente, la implementación de los cambios de contexto involuntarios y del mecanismo de señales se puede considerar un ejemplo de este tipo de interrupciones software de proceso. En este sistema, existen operaciones específicas para inhabilitar temporalmente la funcionalidad correspondiente a este mecanismo (`preempt_disable` inhabilita temporalmente la posibilidad de que se produzca la expulsión de un proceso en ejecución y `preempt_enable` la rehabilita).

Como se verá más adelante, en algunos casos se requiere que la interrupción software de proceso tenga un comportamiento **expulsivo** (como las interrupciones software de sistema), es decir, que, una vez activada y estando en ejecución el proceso destinatario de la misma, su tratamiento se dispare en cuanto se complete el anidamiento de interrupciones que puede estar activo. En cambio, en otras situaciones (como en la terminación involuntaria de un proceso, que se estudiará más adelante), es necesario un comportamiento **no expulsivo**, es decir, que hay que esperar también hasta que se complete el tratamiento de los eventos síncronos que pueden estar anidados (dicho de otra forma, la comprobación de si hay una interrupción software de proceso pendiente sólo se realiza cuando el proceso va a retornar a modo usuario, y no cuando retorna al tratamiento de un evento síncrono).

En cuanto a su uso en un sistema multiprocesador, hay que tener en cuenta que el proceso destinatario de una interrupción software de proceso puede estar ejecutando en un procesador distinto a aquél donde se ha activado la interrupción software (considérese el ejemplo de un proceso que está ejecutando en el procesador 1 y en ese instante un usuario teclea `Control-C`, que es la secuencia de caracteres que, por defecto, en UNIX aborta a los procesos que ejecutan en primer plano, y tal que la rutina de interrupción del teclado se ejecuta en el procesador 2). En ese caso, la operación de dirigir una interrupción software de proceso al proceso afectado, debe incluir también el envío de una IPI ideada para tal fin al procesador donde ejecuta para forzar el tratamiento de la interrupción software de proceso.

Por último, es interesante resaltar que, con frecuencia, la solicitud de activación de una interrupción software de proceso se realiza dentro de la rutina de tratamiento de una interrupción software de sistema. Retomando el ejemplo del proceso que se aborta cuando un usuario teclea `Control-C`, en este caso la interrupción del teclado propiamente dicha realizará sólo las operaciones urgentes, activando una interrupción software de sistema para completar el resto de las actividades. Será dentro de esa rutina donde se detecte que se trata de un `Control-C`, activándose a su vez una interrupción software de proceso dirigida al proceso afectado.

Vida de un proceso

En este punto de la exposición ya se puede plantear un primer esbozo, que más tarde se irá completando, de cómo es la evolución de un proceso a lo largo de su ejecución:

- El proceso arranca en modo sistema, gracias a un pequeño artificio que se explicará más adelante cuando se estudie cómo se crea un proceso, y, enseguida, pasa a modo usuario.
- A partir de ese momento, el proceso continuará su ejecución en modo usuario excepto cuando realiza una llamada al sistema, genera una excepción o se produce una interrupción. En los tres casos, el proceso pasa a ejecutar en modo sistema la rutina de tratamiento correspondiente y, una vez terminada dicha rutina y otras que se hayan podido anidar, vuelve a modo usuario.
- Finalmente, el proceso finaliza su ejecución, ya sea de forma voluntaria, invocando la llamada al sistema correspondiente, o involuntaria, debido a una excepción o por ser abortado. Sea cual sea la forma de terminar, el proceso entra en modo sistema, si es que no lo estaba previamente, y termina.

Esta descripción refleja cómo evoluciona un proceso desde que se crea hasta que termina su ejecución. Sin embargo, para completar toda la panorámica, falta explicar con más detalle cómo convive la ejecución de un proceso con la de otros procesos activos, aunque en las secciones previas ya se ha vislumbrado algo de este asunto: en el tratamiento de un evento que causa que un proceso pase a modo sistema puede ocurrir que éste ceda el uso del procesador a otro proceso deteniendo su ejecución hasta que, posteriormente, otro proceso le devuelva el uso del

procesador y pueda así reanudar su ejecución continuando con el tratamiento del evento que quedó interrumpido.

Procesos/hilos de núcleo

El patrón de ejecución descrito es aplicable a los procesos convencionales, los procesos de usuario. Sin embargo, en la mayoría de los sistemas operativos existe otro tipo de procesos que se suelen denominar procesos o hilos de núcleo. En cuanto a la segunda denominación, proviene de que no tienen un código o datos propios, sino que ejecutan el código del sistema operativo y usan las regiones de datos del mismo, por lo que encajan con el concepto de hilo.

Se trata de procesos que durante toda su vida ejecutan código del sistema operativo en modo sistema. Generalmente, se crean en la fase inicial del sistema operativo, aunque también pueden ser creados más adelante por un manejador u otro módulo del sistema operativo. Este tipo de procesos lleva a cabo labores del sistema que se realizan mejor usando una entidad activa, como es el proceso, que bajo el modelo de operación orientado a eventos de carácter pasivo con el que se rige la ejecución del sistema operativo. El uso de un proceso permite realizar operaciones, como bloqueos, que no pueden hacerse en el tratamiento de un evento asíncrono del sistema.

Este tipo de procesos no tienen un mapa de memoria de usuario asociado, de manera que cuando se ejecutan está activo el mapa de usuario del último proceso de usuario que ejecutó en ese procesador. Por tanto, un proceso de núcleo no puede acceder a direcciones de usuario al tener asociado un mapa de usuario que no le pertenece. Esta característica también tiene repercusiones a la hora de transferir el uso del procesador a un proceso de este tipo. Este cambio de proceso es más eficiente que uno convencional puesto que no es necesario cambiar de mapa de memoria de usuario, que como se verá más adelante, es una operación costosa.

Habitualmente, estos procesos se dedican a labores vinculadas con la gestión de memoria (los "demonios de paginación" presentes en muchos sistemas operativos) y al mantenimiento de la caché del sistema de ficheros.

Hay que resaltar que, aunque en la mayoría de los casos estos procesos de núcleo tienen una prioridad alta, debido a la importancia de las labores que realizan, no siempre es así. Un ejemplo evidente es el proceso nulo, que es un proceso de núcleo pero que, sin embargo, tiene prioridad mínima, para de esta forma sólo conseguir el procesador cuando no hay ningún otro proceso activo en el sistema. En un multiprocesador, hay un proceso nulo por cada procesador.

Es importante notar la diferencia que hay entre estos procesos de núcleo y los procesos de usuario creados por el *super-usuario* del sistema. Los procesos de *super-usuario* son procesos convencionales: ejecutan en modo usuario el código del ejecutable correspondiente. No pueden, por tanto, realizar operaciones privilegiadas. Su "poder" proviene de que, al tener como propietario al *super-usuario*, no tienen restricciones a la hora de realizar llamadas al sistema aplicadas a cualquier recurso del sistema. Algunos ejemplos de procesos de *super-usuario* son los que proporcionan servicios de red y *spooling* (los típicos "demonios" de los sistemas UNIX). Nótese que, aunque muchos de estos procesos se crean también en el arranque del sistema, hay una diferencia significativa con los procesos de núcleo: los procesos de *super-usuario* iniciales los crea el proceso inicial (en UNIX, *init*), mientras que los procesos de núcleo iniciales los crea el propio sistema operativo en su fase de arranque. Téngase en cuenta que, en el caso de un sistema operativo con una estructura de tipo micronúcleo, los procesos que proporcionan las funcionalidades básicas del sistema, como, por ejemplo, la gestión de ficheros, no son procesos de núcleo, sino que tienen las mismas características que los procesos de *super-usuario*.

Aunque, en principio, cualquier módulo del sistema puede crear un proceso de núcleo, éste no es el método más recomendado. En la mayoría de los sistemas operativos, existen colas predefinidas de trabajos, a veces con distintas prioridades, servidas cada una de ellas por uno o más procesos (hilos) de núcleo "trabajadores", cuya única misión es ejecutar estas peticiones. Por tanto, un módulo que quiera realizar una operación en el contexto de un proceso de núcleo no necesita crearlo sino que puede encolar su petición para que la ejecute uno de los procesos trabajadores de núcleo.

Recapitulación de la gestión de eventos

Una vez analizado el tratamiento que realiza el sistema operativo de los distintos eventos, se puede concluir que el trabajo que lleva a cabo el sistema operativo se encuadra en una de las siguientes cinco categorías: tratamiento de llamadas al sistema, de excepciones, de interrupciones hardware y de interrupciones software (de sistema y de proceso), así como ejecución de procesos de núcleo. Por tanto, cuando se desarrolla un módulo del sistema operativo, el diseñador debe decidir a qué contexto asociar las distintas funcionalidades del módulo.

En el caso de un evento síncrono, la decisión no presenta ninguna complicación: la funcionalidad asociada a una llamada al sistema o a una excepción se incluirá en el tratamiento de dicho evento.

La dificultad se presenta para determinar dónde incluir aquellas acciones vinculadas a un evento asíncrono, como ocurre con una interrupción: ¿se incluyen en la propia rutina de interrupción, en la de una interrupción software de sistema o en el contexto de un proceso de núcleo? A continuación, se plantean qué criterios utilizar para tomar esta decisión:

- Si se trata de una operación crítica o urgente, se incluirá en la propia rutina de interrupción.
- En el caso de una operación que sea no urgente y no requiera operaciones de bloqueo, se englobará en la rutina de la interrupción software de sistema.
- Si la acción puede requerir bloqueos, se ejecutará dentro de un proceso de núcleo, que proporciona el contexto para realizar estas operaciones. En este caso, se presentan dos alternativas: crear un proceso de núcleo dedicado al fin perseguido o, la más recomendada, incluir el procesamiento requerido dentro de las colas predefinidas de trabajos del sistema.

En cualquier caso, es importante resaltar que, siempre que sea posible, lo más recomendable es que una determinada funcionalidad pueda realizarse en modo usuario, ya sea como una biblioteca o como un proceso de usuario, puesto que facilita su desarrollo y mantenimiento.

Para concluir esta sección y a modo de recapitulación, en la tabla 1 se muestra desde qué contextos se puede acceder a direcciones de usuario y desde cuáles se puede realizar un cambio del proceso en ejecución. Obsérvese que, dado que un acceso a una dirección de usuario puede causar un fallo de página, todos los contextos que permiten realizar este tipo de accesos deben también habilitar la posibilidad de cambiar de proceso, aunque lo contrario no es cierto, como se puede apreciar en el caso de los procesos de núcleo.

3 Implementación del modelo de procesos

En la sección previa se ha estudiado cómo gestiona el sistema operativo los distintos eventos que se producen en el procesador y cómo éstos van marcando la evolución de un proceso, desde que se crea hasta que se destruye. El proceso cambia de modo usuario a modo sistema cada vez que se produce un evento y retorna a modo usuario cuando termina el tratamiento del evento. Se producen cambios de modo, pero toda la ejecución se ejecuta en el contexto del mismo proceso.

En un sistema con multiprogramación se va multiplexando la ejecución de los distintos procesos activos. El sistema operativo cederá el procesador a otro proceso cuando el que está en ejecución no puede continuar puesto que debe esperar hasta que se produzca cierto evento (por ejemplo, hasta que termine una operación del disco o hasta que un usuario introduzca información con el teclado) o cuando considere que es más conveniente que ejecute otro proceso (por ejemplo, debido a que se ha activado un proceso más prioritario o a que el proceso

	Referencia dir. usuario	**Cambio de proceso**
Llamada al sistema	SÍ	SÍ
Excepción (p. ej. Fallo de página)	SÍ	SÍ
Interrupciones de dispositivo	NO	NO
Interrupciones software de sistema	NO	NO
Interrupciones software de proceso	SÍ	SÍ
Proceso/*thread* de núcleo	NO	SÍ

Tabla 1 *Posibilidad de referencia al mapa de usuario y cambio de proceso desde distintos contextos.*

en ejecución lleva demasiado tiempo ejecutando). ¿Cómo encaja esta multiplexación de procesos con el modelo planteado en la sección anterior?, ¿en qué momento de la ejecución de un proceso se produce el cambio del proceso asignado al procesador?

Esta sección se dedicará precisamente a responder a estas cuestiones, pero, a continuación, se anticipa cuál es la idea subyacente, que ya apareció en la sección previa. Cuando un proceso pasa a modo sistema para realizar el tratamiento de un evento, puede suceder que el proceso ceda el uso del procesador a otro proceso deteniendo su ejecución en ese punto, sin haber terminado el tratamiento del evento. El otro proceso continuará su ejecución desde el punto donde se había quedado, que será también en medio de una rutina de tratamiento de un evento. Pasado un cierto tiempo, otro proceso, nuevamente dentro de la rutina de tratamiento de un evento, cederá el procesador al primer proceso, que así reanudará su ejecución continuando con la rutina de tratamiento del evento que quedó interrumpida. Como se analizó en la sección previa, ese es el "truco" en el que se fundamenta la multiplexación de procesos en un sistema operativo que se basa en el modelo de procesos: el tratamiento de un evento por parte de un proceso en modo sistema puede que no se haga de un tirón, sino que se realice en varias fases debido a que se hagan uno o más cambios de proceso durante su procesamiento, intercalándose, por tanto, la ejecución de múltiples procesos.

Para poder detener temporalmente la ejecución de un proceso, cediéndole el procesador a otro proceso, y luego, posteriormente, poder reanudarla, de forma transparente, justo en el punto donde se quedó (como ya sabemos, en medio de la ejecución de una rutina de tratamiento de un evento), es necesario que el sistema operativo mantenga toda la información sobre el estado de la ejecución de ese proceso. A esa información de estado se la considera el contexto de ejecución del proceso y, por ello, al cambio de proceso se le denomina también *cambio de contexto*: en un cambio de proceso el procesador pasa de ejecutar en el contexto de un proceso a hacerlo en el contexto del otro (un ejemplo de uno de los elementos que se ve afectado por este cambio es el mapa de memoria activo, que pasa a ser el del proceso al que se le cede el procesador). Precisamente, este mecanismo de gestionar el contexto de cada programa activo e "instalarlo" en el procesador cuando va a ejecutar el programa es el que crea el "procesador virtual" que se corresponde con la abstracción de proceso.

3.1 Contexto de un proceso

Para crear la abstracción de proceso, el sistema operativo debe mantener toda la información específica del estado de cada uno de los programas que están ejecutándose en el sistema, o sea, su contexto de ejecución. Como parece lógico, toda la información sobre un determinado proceso se agrupa en una estructura de datos, denominada *Bloque de Control de Proceso* (BCP), y el conjunto de los BCP de todos los procesos conforman la tabla de procesos.

En este punto, para de alguna forma desmitificar cómo se programa un sistema operativo, se considera importante resaltar que el sistema operativo, a pesar de sus peculiaridades, es un programa, y, por tanto, las estructuras de datos que utiliza no tienen ninguna característica extraña: son las mismas que usa cualquier otro programa. Así, en el caso del BCP, si se trata de un sistema operativo escrito en C, como ocurre en la mayor parte de los casos, estará definido como un tipo *struct* y la tabla de procesos podrá ser, en principio, un vector de elementos de tipo BCP.

Diseño de la tabla de procesos

Siguiendo con esta argumentación, hay que destacar que el programador del sistema operativo se verá enfrentado al mismo tipo de alternativas que cualquier programador de aplicaciones a la hora de diseñar la estructura de datos que se corresponde con la tabla de procesos. Así, en una primera versión del sistema operativo, la tabla de procesos puede ser un vector de tamaño fijo de estructuras de BCP, como puede apreciarse en la figura 15.

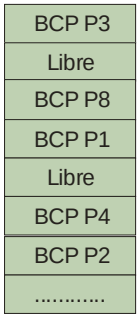


Figura 15 *Tabla de procesos como un vector de estructuras.*

Sin embargo, el programador puede intentar reducir el gasto de memoria al darse cuenta de que, dado que el BCP tiene un tamaño considerable (puede estar cerca de los 1000 bytes) y generalmente la tabla de procesos no está llena, puede ser mejor reservar en memoria dinámica (el sistema operativo también gestiona y usa su propia memoria dinámica) el BCP cuando se crea un proceso. Con este nuevo diseño, la tabla de procesos se convierte en un vector de punteros a BCP, como se muestra en la figura 16. Ésta es la solución usada en la versión 2.2 del núcleo de Linux. La parte negativa de esta estrategia estaría en la sobrecarga introducida por las operaciones de reserva y liberación de cada BCP al crearse y destruirse un proceso, respectivamente, aunque se puede usar una caché de BCP, de manera que se puedan reutilizar.

Esta solución sigue teniendo un problema: la tabla tiene un tamaño fijo, que obliga a recompilar el código del sistema operativo cada vez que se pretenda reajustar esta constante. Además, conlleva una mala utilización de la memoria. Si en un momento dado hay pocos procesos, se está desperdiciando el espacio reservado para cada entrada de la tabla que no está siendo utilizada (el espacio que ocupa un puntero), espacio que podría haberse dedicado a almacenar otro tipo de información, como, por ejemplo, la correspondiente a la caché del sistema de ficheros. Por otro lado, si se llena la tabla de procesos, no se podrán crear nuevos procesos, aun cuando se disponga de memoria libre.

Para solucionar este problema, la tabla de procesos se puede definir como una lista enlazada de BCP, como se ilustra en la figura 17. De esta forma, el tamaño de la "tabla" de procesos se ajusta al número de procesos existentes en el sistema y puede aprovecharse la memoria disponible para reservar espacio para nuevos BCP. Además, esta estrategia elimina la necesidad de buscar una posición libre en el vector cada vez que se crea un nuevo proceso. En el lado negativo, al convertir un vector en una lista, se pierde la capacidad de indexación que proporciona el vector. Sin embargo, aunque parezca sorprendente, esta pérdida no afecta a la gestión de procesos, ya que en la práctica nunca se requiere este acceso indexado.

Esta es la solución planteada en Linux a partir de la versión 2.4. Se usa una lista doblemente enlazada para mantener los BCP de todos los procesos que hay en el sistema. En la cabeza de

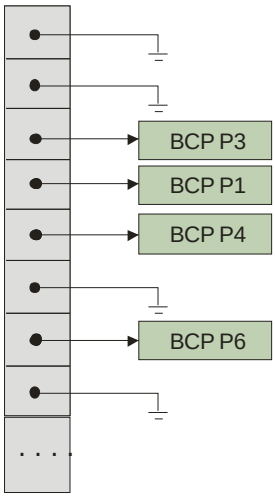


Figura 16 *Tabla de procesos como un vector de punteros a estructuras.*

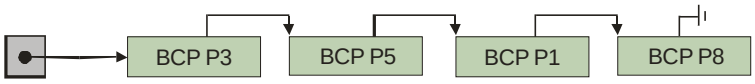


Figura 17 *Tabla de procesos como una lista de estructuras.*

esta lista está el proceso 0 (el proceso de núcleo denominado *swapper*).

Con respecto a la pérdida de la capacidad de indexación debido al uso de una lista, podría parece a priori que hay operaciones que sí requieren esta capacidad. Concretamente, parece necesitarse en aquellas operaciones que tienen como destinatario un proceso especificado por su identificador de proceso (PID, *Process Identifier*), como, por ejemplo, la llamada *kill*, que manda una señal a un proceso identificado por su *pid*. Si se usa un vector y el *pid* correspondiera con la posición del BCP en el vector, que sería un valor que cumple la propiedad de ser único para cada proceso, la localización del proceso sería inmediata. En cambio, usando una lista, habría que recorrerla completa para buscarlo, lo que sería ineficiente, sobretodo en sistemas con un gran número de procesos. Sin embargo, hay que hacer una puntualización sobre el problema planteado. Esta correspondencia directa entre *pid* y posición en el vector no siempre es factible. Así, en los sistemas UNIX el *pid* se asigna de manera creciente, incrementándole cada vez en una unidad, por lo que no se puede corresponder con la posición de su BCP en el vector, y, por tanto, no se puede aplicar la indexación.

En cualquier caso, queda pendiente una cuestión: ¿cómo conseguir deducir cuál es el BCP de un proceso a partir de su *pid*? La solución que se usa en Linux es aplicar una técnica habitual para este tipo de problemas: construir con los BCP una tabla *hash* accedida con el *pid*.

Hay que resaltar que esta disquisición sobre cómo diseñar la tabla de procesos ilustra el tipo de problemática con la que se enfrenta el diseñador de un sistema operativo, que es de la misma índole que la que se encuentra cualquier diseñador de aplicaciones. Además, permite apreciar una tendencia general en el diseño de sistemas operativos: sustituir las estructuras de datos de tamaño fijo (además del caso de la tabla de procesos, hay otros ejemplos como la caché del sistema de ficheros) por estructuras dinámicas.

Listas de recursos

En la última solución planteada en la sección anterior para el diseño de la tabla de procesos, se utilizaba una lista de BCP. En general, en un sistema operativo habrá numerosas listas de distintos recursos y, por tanto, para evitar redundancias, se suele definir un único tipo genérico de manejo de listas.

En Linux, este tipo genérico permite definir listas circulares doblemente enlazadas. Como curiosidad, se puede resaltar que cada elemento de la lista (por ejemplo, un BCP en la lista de procesos existentes) no apunta a la dirección de comienzo del siguiente elemento (es decir, a la dirección de inicio del siguiente BCP), sino a la dirección donde está definido el campo correspondiente a la lista dentro de ese elemento, lo que facilita el uso de un mismo tipo de datos para todas las listas.

El Bloque de Control del Proceso (BCP)

Centrándose en el BCP, su contenido va a depender de las características específicas de cada sistema operativo. Así, por ejemplo, en un sistema operativo UNIX, en el BCP habrá que guardar información sobre la disposición del proceso con respecto a cada señal (qué tratamiento ha asociado a la señal, si la tiene bloqueada, etc.). Sin embargo, en un sistema operativo que no implemente este mecanismo de señales, evidentemente, no existiría esta información en el BCP. Por tanto, manteniendo el enfoque neutral de la presentación, a continuación se va a mostrar qué tipo de información suele estar presente en un BCP. Nótese que muchos de estos campos están vinculados con otros componentes del sistema operativo (por ejemplo, con la gestión de memoria o con el sistema de ficheros) y, por tanto, el estudio detallado de los mismos se realiza en los temas correspondientes.

- El estado del proceso. A este campo se le dedica la próxima sección.
- Una zona para guardar una copia de los registros del procesador cuando el proceso cede el uso del procesador, de manera que se puedan restaurar estos valores cuando el proceso

vuelva a ejecutar. Se trata de un campo cuyo contenido es dependiente del procesador que se esté usando.

- Información de planificación, tal como la prioridad del proceso o cuánto tiempo le resta al proceso en ejecución para consumir su turno.
- Información sobre el mapa de memoria del proceso. Incluirá, entre otras informaciones, una lista de las regiones de memoria que tiene el mapa del proceso, así como una referencia a la tabla de páginas del proceso, que deberá estar activa cuando el proceso esté en ejecución. Una de estas regiones será la pila de usuario del proceso.
- La dirección de la pila de sistema del proceso. Dado el limitado nivel de anidamiento de rutinas que se produce durante la ejecución de un proceso en modo sistema, suele ser suficiente con reservar una página para esta pila. Linux presenta una estrategia interesante a la hora de reservar la pila de sistema del proceso: usa una zona contigua para almacenar en la parte de direcciones más bajas el BCP y en la de direcciones más altas la pila del sistema, que, como es habitual, crecerá hacia direcciones más bajas, o sea, hacia el BCP. Observe que, con esta disposición, a partir del valor del puntero de pila de sistema durante la ejecución de un proceso, se puede deducir directamente dónde está almacenado su BCP. Si se configura Linux sobre un Pentium de manera que se use pila de sistema y de interrupción separadas, sólo hay que reservar 4K bytes para la zona que almacena el BCP y la pila de sistema, requiriéndose 8K bytes en caso de pila única.
- Información de ficheros, de E/S y de otros recursos del sistema que está usando el proceso. En el caso de Windows, se corresponde con la tabla de *handles* abiertos, ya que en este sistema prácticamente todos los recursos se acceden a través de este mecanismo. En los sistemas UNIX, se trata de la lista de ficheros que ha abierto el proceso, que no sólo incluye ficheros propiamente dichos, sino también dispositivos de E/S y mecanismos de comunicación y sincronización.
- Información de contabilidad sobre el uso de recursos por parte del proceso, así como límites sobre el uso máximo de los mismos. Por ejemplo, con respecto al tiempo de procesador consumido por el proceso, se especificaría cuánto ha sido hasta el momento, y cuál es el máximo permitido.
- Información sobre quién es el "dueño" del proceso. Corresponde al usuario que causó que se creara este proceso. Esta información será la que consulte el sistema operativo a la hora de determinar si el proceso tiene permiso para realizar una determinada operación (por ello, en algunos entornos esta información se denomina "los credenciales" del proceso). En el caso de los sistemas UNIX, esta información corresponde al *uid* (identificador de usuario) y *gid* (identificador de grupo) del usuario, aunque la cosa es algo más complicada por el mecanismo del *setuid* característico de UNIX, que el lector ya conocerá, pero que esta presentación no tratará por su carácter específico. En Windows, esta información corresponde al *token* de acceso del proceso, que describe el perfil de seguridad del mismo.
- El identificador del proceso (*pid*). Un número que identifica de forma única a cada proceso. En los sistemas UNIX este valor va incrementándose según van creándose procesos. Esto permite que haya una relación entre los valores de los identificadores de los procesos y el tiempo cuando se crearon. Además del *pid* del proceso, en el BCP también se incluye información adicional que identifica al proceso, como el nombre del programa que está ejecutando, así como sus argumentos.
- Varios campos para construir distintas listas de BCPs, entre otras las siguientes:
    - En el caso de no usar un vector para la tabla de procesos, sino una lista, será necesario incluir punteros para construirla.
    - En Linux, punteros para construir las listas *hash* comentadas en la sección anterior.
    - Punteros para mantener las relaciones "familiares" de los procesos. En Linux, un proceso tiene un puntero a su proceso padre, a su hijo más reciente y punteros para formar una lista con todos sus hermanos.

  - Punteros para mantener listas de procesos que están en el mismo estado. Por ejemplo, la lista de procesos listos para ejecutar.

Una vez descrita la información que suele aparecer en el BCP de cualquier sistema operativo, a continuación, se presentan algunas consideraciones sobre esta estructura de datos:

- Habitualmente no toda la información descrita está incluida directamente en el BCP, sino que en ocasiones se trata de punteros a estructuras de datos externas al BCP, sobretodo si se trata de información que puede ocupar un tamaño variable.
- En principio, como ocurre con cualquier estructura de tipo registro (*struct* en C), el orden de declaración de los campos de la misma es intrascendente. Sin embargo, en este caso, habitualmente hay que ser especialmente cuidadoso a la hora de modificar el tipo de datos del BCP, ya que esta estructura de datos es accedida desde rutinas en ensamblador, que pueden presuponer un determinado desplazamiento para cada campo de la estructura. Asimismo, a veces, se definen de forma contigua aquellos campos que se utilizan generalmente de forma simultánea para intentar que al acceder a uno de ellos se traiga el otro automáticamente a la línea de caché correspondiente.
- En un sistema con hilos habría información de los mismos en el BCP, pero en esta exposición se ha optado por diferir la introducción de este concepto hasta la parte final del capítulo. Así, por ejemplo, en Windows, que implementa hilos, el BCP, denominado bloque *EPROCESS*, incluye la cabecera de una lista de los bloques de control de los hilos asociados al proceso.
- Si se trata de un sistema operativo basado en un micronúcleo, la información del proceso estará repartida entre los distintos componentes del sistema, es decir, cada componente almacenará un BCP con las características del proceso que conciernen a ese componente. Así, el micronúcleo guardará en el BCP de cada proceso información requerida por la gestión de bajo nivel que lleva este módulo (por ejemplo, almacenará la copia de los registros del procesador), mientras que el sistema de ficheros, por ejemplo, almacenará en su BCP información de los ficheros que está usando ese proceso, y, de manera similar, el resto de los componentes. Sin ser Windows un sistema basado en un micronúcleo, como ya conoce el lector, está dividido en varios componentes: el ejecutivo, que implementa la funcionalidad del sistema operativo, y los subsistemas, que ofrecen distintas interfaces a los procesos. Por tanto, en Windows también la información del BCP está repartida: el ejecutivo almacena la mayor parte de la información del proceso en el bloque *EPROCESS*, pero cada subsistema almacena la información del proceso que le concierne en el Bloque de Entorno del Proceso.

Estado de un proceso

Este campo del BCP refleja las distintas situaciones por las que transita un proceso a lo largo de su vida con respecto al uso del procesador. Cada sistema operativo requiere sus estados específicos que reflejan las peculiaridades del mismo. Sin embargo, existen tres estados básicos aplicables a la gestión de procesos de cualquier sistema operativo:

- *En ejecución*: el proceso está actualmente usando el procesador, ya sea en modo usuario o en modo sistema. En un sistema habrá tantos procesos en este estado como procesadores. Este estado se puede considerar, por tanto, doble, distinguiéndose entre estar en ejecución en modo usuario y en modo sistema.
- *Listo para ejecutar*. El proceso podría estar en ejecución, pero no tiene un procesador asignado en estos momentos. Como se verá más adelante y ya se ha anticipado, aunque sea parcialmente, un proceso en este estado se habrá quedado parado en modo sistema en la rutina de tratamiento de un evento, o bien nunca habrá ejecutado.
- *Bloqueado*. El proceso no puede continuar debido a que está a la espera de que se produzca un determinado evento. Por ejemplo, el proceso puede estar esperando que se termine una operación de entrada/salida. También en este caso el proceso se habrá quedado parado en modo sistema en la rutina de tratamiento de un evento. Realmente, se trata de un estado múltiple, ya que se podría distinguir un estado de bloqueo

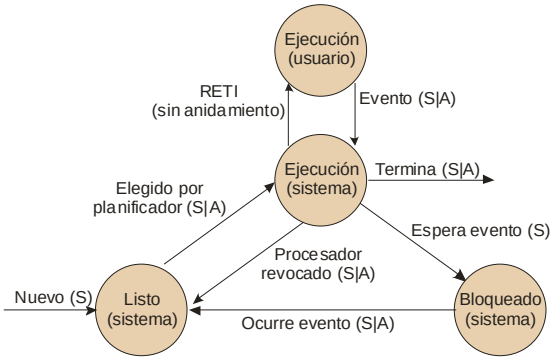


Figura 18 *Diagrama de estados de un proceso.*

independiente por cada uno de los posibles motivos por los que se puede bloquear un proceso.

El proceso va transitando entre estos tres estados. Hay que resaltar que para que se produzca cualquier transición en el estado de un proceso siempre tiene que haber una activación del sistema operativo (o sea, tiene que producirse un evento del procesador), aunque, por otra parte, hay que resaltar que no toda activación causará un cambio de estado. Así, por ejemplo, una interrupción de reloj puede cambiar el estado de un proceso que estaba dormido esperando un determinado plazo de tiempo, que transitará de bloqueado a listo. Asimismo, una interrupción de reloj puede implicar el final del turno de un proceso que está en ejecución, cambiando su estado a listo para ejecutar. Sin embargo, habrá muchas activaciones de la interrupción de reloj que no cambien el estado de ningún proceso.

La figura 18 muestra el diagrama de estados de un proceso identificando qué tipo de evento (Síncrono, *S*, o asíncrono, *A*) puede causar cada transición. Téngase en cuenta que si se trata de un sistema con hilos, el diagrama de estados se aplicará a cada hilo.

A continuación, se comentan las distintas transiciones presentes en el diagrama de estados:

- Inicialmente, cuando se crea un proceso, su estado inicial es listo para ejecutar. Como se analizará más adelante, el proceso empezará en modo sistema, aunque, en cuanto empiece a ejecutar transitará rápidamente a modo usuario. Se crea, de manera artificial, una situación inicial similar a la que existe mientras se está tratando un evento del procesador, como se verá cuando se estudie cómo se crea un proceso.
- Del estado de listo para ejecutar, el proceso transita al estado de en ejecución cuando otro proceso que está en ejecución tratando un evento del procesador decide ceder el procesador y selecciona a este proceso para esta cesión. Esta selección la realizará el planificador del sistema operativo. El proceso reanudará su ejecución continuando la rutina de tratamiento donde se quedó parado, por tanto, en modo sistema.
- Estando en ejecución, el proceso puede transitar a otros estados pero, como todas las transiciones, siempre ocurrirá debido a que se produce un evento del procesador:
    - Transitará a bloqueado si dentro de la rutina de tratamiento del evento se detecta que el proceso no puede continuar hasta que se produzca un determinado suceso.
    - Transitará a listo si dentro de la rutina de tratamiento del evento se detecta que el proceso no debe continuar, ya sea porque se detecta que hay otro proceso más urgente o porque se determina que lleva demasiado tiempo en ejecución.
    - Concluirá su ejecución de forma voluntaria o involuntaria si el evento correspondiente indica que debe terminar su ejecución.
- El proceso transita de bloqueado a listo cuando se produce un evento del procesador, que será manejado por el proceso en ejecución, que causa que ocurra el suceso por el que estaba esperando el proceso. El proceso pasa en este momento a ser candidato a ser elegido por el planificador.

Es importante resaltar que un proceso que no esté en ejecución, en estado listo o bloqueado, estará parado en una rutina del sistema operativo, por tanto, en modo sistema. Por otro lado, el proceso en ejecución podrá estar en modo usuario o sistema, pudiendo cambiar de estado sólo cuando está en modo sistema.

En esta sección se ha planteado un diagrama de estados mínimo. En un sistema real, hay estados adicionales. Así, en sistemas con memoria virtual, como se estudia en ese tema, existe un mecanismo de suspensión de procesos que, en situaciones de hiperpaginación, suspende la ejecución de uno o más procesos, eliminando de memoria física las páginas de los mismos. Se introducen, por tanto, dos nuevos estados:

- *Suspendido y listo*: Un proceso en estado de listo ha sido suspendido, para intentar paliar problemas de hiperpaginación. Mientras está en este estado, el proceso no es elegible por el planificador. Cuando el sistema operativo lo considere oportuno, el proceso volverá a estado de listo, siendo de nuevo candidato a usar el procesador.
- *Suspendido y bloqueado*: Un proceso en estado bloqueado ha sido suspendido, para intentar paliar problemas de hiperpaginación. Normalmente, es preferible suspender procesos que estén bloqueados. Cuando ocurra el evento por el que estaba esperando el proceso, transitará al estado de suspendido y listo. En ningún caso saldrá del estado de suspensión sin pasar por ese estado de suspendido y listo.

Además, en cada sistema operativo existen estados específicos. A continuación, como ejemplo, se comentan algunas características específicas en cuanto al diagrama de estados del proceso presentes en los sistemas UNIX:

- Existe un estado denominado *zombi*, que corresponde a un proceso que ha terminado su ejecución pero cuyo proceso padre todavía no ha ejecutado la llamada *wait*.
- Aparece el estado *stopped*, al que un proceso transita cuando recibe la señal SIGSTOP (u otras señales que tienen ese mismo efecto) o cuando, estando bajo la supervisión de un depurador, recibe cualquier señal.
- El estado de bloqueo está dividido en dos estados independientes:
    - Espera interrumpible: además de por la aparición del suceso que está esperando, un proceso puede salir de este estado por una señal dirigida al mismo. Considere, por ejemplo, un proceso que está bloqueado esperando por que le llegue información por la red o del terminal. Es impredecible cuando puede llegar esta información y debe de ser posible hacer que este proceso salga de este estado para, por ejemplo, abortar su ejecución.
    - Espera no interrumpible: en este caso, sólo la aparición del suceso puede hacer que este proceso salga de este estado. Debe de tratarse de un suceso que es seguro que ocurrirá y, además, en un tiempo relativamente acotado, como, por ejemplo una interrupción del disco.
- Normalmente, en el campo del BCP que guarda el estado no se distingue explícitamente entre el estado listo y en ejecución. Esa diferencia se deduce a partir de otras informaciones adicionales (el BCP del proceso que está actualmente en ejecución suele estar apuntado por una variable que hace referencia al mismo para poder acceder eficientemente a sus campos).

Colas de procesos

Una de las estructuras de datos que más utiliza un sistema operativo para la gestión de procesos es la cola, o lista, de procesos. El sistema operativo enlaza en una lista todos los procesos que están en el mismo estado. Estas listas, evidentemente, estarán basadas en el tipo de lista genérico del sistema operativo correspondiente.

En todos los sistemas existe una lista de procesos listos que, normalmente, incluye el proceso en ejecución (o los procesos en ejecución, en el caso de un multiprocesador), aunque en teoría sean dos estados distintos. Normalmente, existe una variable global que hace referencia al proceso (o procesos) en ejecución.

En cuanto a los procesos bloqueados existen distintas alternativas a la hora de organizarlos en colas:

- Puede haber una única cola que incluya a todos los procesos bloqueados, con independencia de cuál sea el motivo del bloqueo. Se trata de una solución poco eficiente, ya que desbloquear a un proceso esperando por un evento requiere recorrer toda la cola.

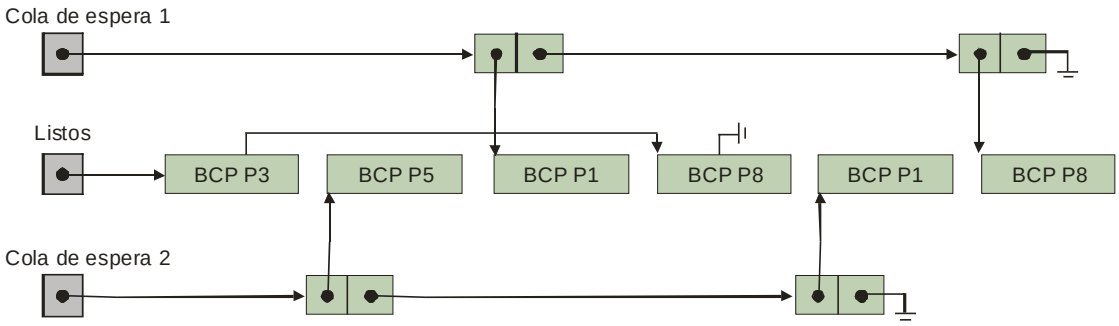


Figura 19 *Colas de procesos en Linux.*

- Usar una cola por cada posible condición de bloqueo, por ejemplo, procesos bloqueados esperando datos de un determinado terminal o esperando que se libere un mutex. Esta es la solución usada, por ejemplo, en Linux. El único inconveniente, aunque tolerable, es el gasto de datos de control requeridos por cada cola.
- Utilizar una solución intermedia de manera que se hagan corresponder múltiples eventos con una misma cola. Esta es la solución usada en UNIX System V.

Dado que cada proceso sólo puede estar en un estado en cada momento, basta con tener un puntero en el BCP (o dos, en el caso de que las colas tengan un enlace doble) para insertar el proceso en la cola asociada al estado correspondiente.

En Linux, sin embargo, el puntero incluido en el BCP se usa sólo para insertar el proceso en la cola de listos. Las colas de bloqueados, denominadas en Linux *wait queues*, incluyen sus propios punteros, no usando ningún puntero del BCP para formar la lista, como se puede apreciar en la figura 19. Por tanto, como se verá más adelante, un proceso puede estar en la cola de listos y en la cola de un determinado evento (aunque, como se analizará entonces, se trata de una situación momentánea) o en las colas de espera de varios eventos.

Hay que resaltar que la cola de procesos es una de las estructuras de datos más importante y más utilizada en el sistema operativo. Cada cambio de estado requiere la transferencia del BCP de la cola que representa el estado previo a la cola correspondiente al nuevo estado.

3.2 Cambio de contexto

La operación básica en la que se fundamenta la multiplexación de procesos es el cambio de contexto, o sea, la operación que cambia el proceso asignado al procesador. Un cambio de contexto corresponde a una activación del sistema operativo (esto es, el tratamiento de un evento del procesador) que provoca los dos siguientes cambios de estado:

- El proceso en ejecución, que está en modo sistema tratando un evento del procesador, transita al estado de listo o de bloqueado.
- Un proceso seleccionado por el planificador pasa del estado de listo a estar en ejecución, con lo que reanudará su ejecución en el punto que la dejó: en medio de una rutina de tratamiento de un evento del procesador. Nótese que, dado el artificio que se usa para crear un proceso, cuando se le cede el procesador a un proceso que nunca ha ejecutado, equivale, a todos los efectos, a que éste reanudará su ejecución dentro de una rutina de tratamiento de un evento del procesador.

Téngase en cuenta que para que haya un cambio de contexto tiene que haber una activación del sistema operativo que cause este par de cambios de estado, pero no todo cambio de estado implica un cambio de contexto. Así, por ejemplo, cuando se produce una interrupción de reloj que despierta un proceso que estaba dormido un determinado plazo de tiempo, hay un cambio de estado del proceso, que pasa de bloqueado a listo, pero no hay un cambio de contexto a no ser que el proceso que se desbloquea sea más prioritario que el proceso en ejecución.

La operación de cambio de contexto es dependiente tanto del procesador, de hecho, parte de ella hay que hacerla en ensamblador, como del sistema operativo. A continuación, se muestra de forma genérica qué operaciones conlleva un cambio de contexto:

- El proceso en ejecución (*P*) está realizando en modo sistema el tratamiento de un evento del procesador y decide que no debe continuar su ejecución, cambiando su estado a listo

o a bloqueado. En caso de que pase a bloqueado, se moverá el BCP de la cola de listos a la cola de bloqueo correspondiente.

- El proceso *P* invoca a la función de planificación, que selecciona un proceso (*Q*) entre los procesos listos existentes. Asimismo, se actualiza la variable global que hace referencia al proceso en ejecución.
- El proceso *P* salva su propio contexto, almacenando el contenido de todos los registros del procesador que se precise guardar en el campo correspondiente de su BCP. El valor que se guarda en el caso del contador de programa está manipulado de manera que al volver a ejecutar *P* se salte la restauración que viene a continuación.
- El proceso *P* restaura el contexto de *Q* lo que implica activar su mapa de memoria (como se estudia en el tema de gestión de memoria, esta operación es costosa) y restaurar la copia de los registros almacenada en el BCP de *Q*. En ese momento habrá comenzado a ejecutar *Q*, continuando desde el punto donde se quedó la última vez: justo en un cambio de contexto donde cedió el procesador a otro proceso, o sea, en una situación similar a la que tiene en este momento *P*.

El proceso *P* volverá a ejecutar cuando otro proceso en el tratamiento de un evento ceda el procesador y el planificador elija a *P* como destinatario de la cesión. Téngase en cuenta que, en el caso de que *P* hubiera dejado el procesador por haber pasado al estado de bloqueado, no podrá ser elegido por el planificador hasta que se produzca una activación del procesador que lo pase al estado de listo. A partir de ese momento, pasaría a ser candidato a ser elegido por el planificador. Nótese, por tanto, que la ejecución de un proceso consiste en una serie de rachas de ejecución que van desde un cambio de contexto a otro.

Es importante remarcar que el cambio de contexto es una operación relativamente costosa (típicamente, dura una decenas de microsegundos) y no es realmente trabajo útil, sobretodo en el caso de que el proceso en ejecución pase al estado de listo, ya que en esa situación se ha incurrido en la sobrecarga del cambio de contexto cuando se podría haber continuado con el proceso en ejecución. Por tanto, hay que intentar mantener la frecuencia de cambios de contexto dentro de unos límites razonables.

Por último, hay que resaltar la dificultad que conlleva entender lo que ocurre cuando se produce un cambio de contexto: la ejecución da un salto repentino dentro del código del sistema operativo desde el punto donde se queda el proceso *P* hasta el punto donde se reanuda *Q*. Dada esta complejidad, a continuación, se intenta aclarar qué sucede en un cambio de contexto usando para ello un ejemplo.

Supóngase que se han incluido en la función que encapsula el cambio de contexto algunas sentencias que imprimen información de los procesos involucrados, tal como muestra el programa 9.

Programa 9 Ejemplo aclaratorio del cambio de contexto.

```
BCP *ant, *post;
void cambio (BCP *prev, BCP *sig) {
    printf("prev %d sig %d \n", prev->pid, sig->pid);
    ant = prev;
    post = sig;
    cambio_contexto(&prev->contexto, &sig->contexto);
    printf("prev %d sig %d ant %d post %d\n",
                prev->pid, sig->pid, ant->pid, post->pid);
}
```

Suponiendo que se ejecutan continuamente, y de forma cíclica, tres procesos P1 (pid=1), P2 (pid=2), P3 (pid=3), en ese orden, ¿qué información se imprimiría durante el intervalo de tiempo que va desde que P1 vuelve a ejecutar, después de que P3 le ceda el control, hasta el momento en que nuevamente P3 le va a ceder el control (es decir, un ciclo completo de ejecución)?

Para responder a esta cuestión hay que tener en cuenta la manera en que se lleva a cabo un cambio de contexto. Cuando un proceso quiere ceder el procesador y ejecuta la rutina de cambio de contexto, su ejecución queda detenida en esa operación, guardándose su estado de manera que, cuando en su momento reanude su ejecución, lo haga justo a partir de ese punto, completando la segunda parte de la rutina de cambio.

Otro aspecto que hay que tener en cuenta es cómo afecta el cambio de contexto a los datos que gestiona el sistema operativo. Concretamente, dado que la ejecución de la rutina de cambio no se hace de manera continua en el tiempo, sino que puede transcurrir un intervalo de tiempo impredecible hasta que se reanude la ejecución del proceso y se complete la rutina, ¿qué habrá ocurrido con los valores de las variables locales de la rutina, así como con las variables globales usadas en la misma?, ¿seguirán manteniendo sus valores cuando se reanude la ejecución del proceso en el ámbito de la segunda parte de la rutina de cambio?

Para responder a estas preguntas, hay que observar que las variables locales de la rutina se almacenan en la pila de sistema del proceso que la ejecuta, mientras que las variables globales del sistema operativo se almacenan en la región de datos del mismo. Por tanto, dado que en un cambio de contexto se deja de usar la pila de sistema del proceso saliente para comenzar a utilizar la del proceso entrante, el valor de las variables locales al reanudarse la ejecución de la rutina será el mismo que tenían antes del cambio de contexto. No ocurre así con las variables globales cuyo valor se corresponderá con su última actualización, que, evidentemente, podrá haber sido posterior al cambio de contexto.

Teniendo en cuenta estas consideraciones, la salida generada por la traza de ejecución planteada en el enunciado sería la siguiente:

1. P1 reanuda su ejecución: prev 1 sig 2 ant 3 post 1
2. P1 invoca cambio(P1, P2): prev 1 sig 2
3. P2 reanuda su ejecución: prev 2 sig 3 ant 1 post 2
4. P2 invoca cambio(P2, P3): prev 2 sig 3
5. P3 reanuda su ejecución: prev 3 sig 1 ant 2 post 3
6. P2 invoca cambio(P3, P1): prev 3 sig 1

Nótese que a continuación se repetiría el paso 1 y así sucesivamente. Si se observa una ejecución completa de la rutina de cambio por parte de un proceso (por ejemplo, el proceso P1), se puede apreciar que realmente están involucrados tres procesos (el cambio de contexto es cosa de tres): el proceso que invoca la rutina para ceder el procesador (identificado por `prev`, siendo P1 en el ejemplo), el proceso al que se le cede el control (identificado por `sig`, siendo P2 en el ejemplo), y el proceso que posteriormente le devolverá el control al proceso que invocó la rutina de cambio para que complete su ejecución (identificado por `ant`, siendo P3 en el ejemplo).

Tipos de cambio de contexto

Como se explicó en la sección previa, en un cambio de contexto el proceso en ejecución puede transitar al estado de listo o al de bloqueado. Es importante distinguir entre estos dos tipos de cambio de contexto:

- Cambio de contexto voluntario: Se produce cuando el proceso en ejecución pasa al estado de bloqueado debido a que tiene que esperar por algún tipo de evento. Sólo pueden ocurrir dentro de una llamada al sistema o en el tratamiento de la excepción correspondiente al fallo de página. No pueden darse nunca en una rutina de interrupción, ya que ésta no está relacionada con el proceso que está actualmente ejecutando. Obsérvese que el proceso deja el procesador puesto que no puede continuar. Por tanto, el motivo de este cambio de contexto es un uso eficiente del procesador.
- Cambio de contexto involuntario: Se produce cuando el proceso en ejecución tiene que pasar al estado de listo ya que debe dejar el procesador por algún motivo (por ejemplo, debido a que se le ha acabado su rodaja de ejecución o porque hay un proceso más urgente listo para ejecutar). Nótese que en este caso el proceso podría seguir ejecutando, pero se realiza el cambio de contexto para realizar un mejor reparto del procesador.

Cambio de contexto voluntario

Con respecto a los cambios de contexto voluntarios, el programador del sistema operativo va a realizar una programación que podríamos considerar normal, pero va a incluir operaciones de cambio de contexto cuando así se requiera, debido a que bajo ciertas condiciones el proceso no pueda continuar su ejecución hasta que se produzca un cierto evento. Así, por ejemplo, el pseudo-código de una llamada que lee del terminal podría ser como el siguiente:

```
leer() {
      ..............................
      Si no hay datos disponibles
        proc_anterior = proc_actual;
        proc_anterior-> estado= BLOQUEADO;
        Mover proc_anterior de listos a lista del terminal;
        proc_actual = planificador();
        cambio_contexto(proc_anterior, proc_actual);
      ..............................
}
```

Dado que este código se repite mucho dentro del sistema operativo con la única diferencia de a qué lista se mueve el proceso, normalmente se suele codificar una función para encapsular esta funcionalidad (podría llamarse, por ejemplo, `bloquear`). El ejemplo previo quedaría de la siguiente forma:

```
Si no hay datos disponibles
      bloquear();
```

Por su parte, la función `bloquear` podría ser de la siguiente forma:

```
Bloquear (cola de bloqueo) {
  p_proc_actual->estado=BLOQUEADO;
  Inhibir todas las interrupciones;
  Mover BCP de proceso actual a cola de bloqueo;
  Restaurar el estado de interrupciones previo;
  cambio de contexto a proceso elegido por planificador;
}
```

Obsérvese que el código de la función `bloquear` debe inhibir todas las interrupciones, puesto que está modificando la lista de procesos listos y es prácticamente seguro que todas las rutinas de interrupción manipulan esta lista, puesto que normalmente desbloquean a algún proceso. Volveremos a este asunto en la sección dedicada a la sincronización.

Téngase en cuenta que, como se comentó previamente, este tipo de cambios de contexto sólo pueden darse dentro de una llamada (o en el tratamiento de la excepción de fallo de página), por lo que sólo se podrá llamar a `bloquear` desde el código de una llamada. En consecuencia, no se debe invocar desde una rutina de interrupción ya que el proceso en ejecución no está relacionado con la interrupción.

Como se analizó al presentar el modelo de procesos, una determinada llamada puede incluir varias invocaciones de `bloquear`, puesto que podría tener varias condiciones de bloqueo. A continuación, se muestra una hipotética llamada al sistema que incluye varios puntos en los que, dependiendo de distintas condiciones, se realiza un cambio de contexto de tipo voluntario:

```
sis_llamada_X () {
      .........
      Si (condición_1) Bloquear(cola de evento 1);
      ......... ← Cuando se desbloquee seguirá por aquí

      Mientras (! condición_2) {
            Bloquear(cola de evento 2);
```

```
                ......... ←  seguirá por aquí
        }
    }
```

Cuando el proceso por fin vuelva a ejecutar lo hará justo después de la llamada al cambio de contexto dentro de la rutina `bloquear`.

Como se comentó previamente, también puede haber cambios de contexto voluntarios en el tratamiento de la excepción de fallo de página:

```
    fallo_página () {
        .........
        Si (no hay marco libre
            y página expulsada modificada) {
              Programar escritura en disco;
              Bloquear(cola de espera del disco);
              ......... ←   seguirá por aquí
        }
        // Traer nueva página
        Programar lectura de disco;
        Bloquear(cola de espera del disco);
        ......... ←   seguirá por aquí
    }
```

Cuando se produzca el evento que estaba esperando el proceso, éste se desbloqueará pasando al estado de listo para ejecutar. Nótese que es un cambio de estado, pero no un cambio de contexto.

De la misma manera que ocurre con la función `bloquear`, habitualmente, las operaciones que hay que llevar a cabo para desbloquear un proceso se suelen encapsular en una función `desbloquear` que generalmente recibe como parámetro la lista donde está el proceso que se quiere desbloquear.

```
Desbloquear (cola de bloqueo) {
    Seleccionar proceso en la cola (primero si política FIFO);
    Estado de proceso elegido  = LISTO;
    Inhibir todas las interrupciones;
    Mover BCP de elegido de cola del evento a cola de listos;
    Restaurar el estado de interrupciones previo;
}
```

Observe que la rutina `desbloquear` podría ser invocada tanto desde una llamada al sistema como desde una interrupción y que, dado que manipula la lista de procesos listos, debería ejecutarse gran parte de la misma con las interrupciones inhabilitadas.

Retomando el ejemplo anterior del terminal, la rutina de interrupción del terminal incluiría algo como lo siguiente:

```
int_terminal() {
    ..............................

    Si hay procesos esperando
          desbloquear(lista del terminal);
    ..............................
}
```

En el caso de Linux, las llamadas de bloqueo, *sleep_on* (o *wait_event*, más recomendable porque presenta menos problemas de condiciones de carrera), y desbloqueo, *wake_up*, presentan distintas modalidades, que, de forma simplificada, se comentan a continuación:

- En la llamada de bloqueo se puede especificar si la espera será interrumpible por una señal o no, así como establecer un plazo máximo de espera.

- En la llamada de desbloqueo se puede especificar, entre otras cosas, a cuántos procesos se les desbloqueará (sólo uno, un número determinado o todos) y si sólo se tendrán en cuenta los que están en espera interrumpible o todos.

Asimismo, es interesante resaltar que, dado que, como se comentó previamente, las colas de bloqueo de Linux (*wait queues*) incluyen sus propios punteros a BCP, no usando ningún campo interno del BCP, la propia rutina de bloqueo puede encargarse de eliminar al proceso de la cola de bloqueo, lo que genera un código más homogéneo y simple, como se puede apreciar en los siguientes fragmentos simplificados.

```
sleep_on (cola de bloqueo) {
   Incluir BCP de proceso actual en cola de bloqueo;
   Eliminar BCP de proceso actual de la lista de listos;
   Cambio de contexto al elegido por el planificador;
   Eliminar BCP de proceso actual en cola de bloqueo;
}

wake_up (cola de bloqueo) {
   Incluir BCP de proceso desbloqueado a cola de listos;
}
```

Observe cómo es más sencillo con este esquema el desbloquear a un proceso cuando desde la rutina que causa el desbloqueo no se conoce en qué cola de bloqueo se encuentra el proceso.

Cambio de contexto involuntario

Una llamada al sistema o una rutina de interrupción pueden desbloquear a un proceso más importante que alguno de los que está ejecutando actualmente o pueden indicar que el proceso actual ha terminado su turno de ejecución. En esta situación, el proceso afectado por la expulsión, que puede ser el mismo que trata el evento que causa la expulsión o uno diferente ejecutando en otro procesador, puede estar involucrado en la ejecución anidada de actividades del sistema operativo.

Para asegurar el buen funcionamiento del sistema, hay que diferir el cambio de contexto involuntario hasta que terminen todas las rutinas de tratamiento de eventos que pudieran estar anidadas (al menos las asociadas a interrupciones, como se analizará en esta sección). La cuestión es cómo implementar este esquema de cambio de contexto retardado. La solución más elegante es utilizar el ya conocido mecanismo de interrupción software de proceso descrito en secciones previas.

Con la interrupción software de proceso el cambio de contexto involuntario es casi trivial: cuando dentro del código de una llamada o una interrupción se detecta que hay que realizar un cambio de contexto involuntario, se activa la interrupción software de proceso dedicada a este fin dirigiéndola al proceso que se debe expulsar. Dado que se trata de una interrupción de nivel mínimo, su rutina de tratamiento no se activará hasta que el estado de las interrupciones del procesador sea el adecuado. Si había un anidamiento de rutinas de interrupción, tanto hardware como software de sistema, todas ellas habrán terminado antes de activarse la rutina de tratamiento de esta interrupción software de planificación. Esta rutina de tratamiento se encargará de llevar a cabo el cambio de contexto involuntario, que se producirá justo cuando se pretendía: en el momento en que han terminado todas las rutinas de interrupción de los dispositivos así como de las interrupciones software de sistema. Nótese que en el caso de un monoprocesador, la interrupción software de planificación va siempre dirigida al único proceso en ejecución, que será precisamente el que trató el evento que causa su propia expulsión.

A continuación, se muestran dos escenarios en los que puede producirse un cambio de contexto involuntario.

En el primero, el cambio de contexto se activa por la finalización del turno de ejecución (la rodaja de ejecución) del proceso:

```
int_reloj () {
      .......
```

```
        Si (--p_proceso_actual->rodaja == 0)
              Activa int.SW de planificación dirigida a p. actual;
        .......
    }
```

En el segundo, una rutina, que podría corresponder a una llamada al sistema o a una interrupción, causa el desbloqueo de un proceso más prioritario.

```
    rutina_X () {
        .......
        Si (condición) {
              Desbloquear(cola de evento);
              Si prioridad desbloqueado > alguno en ejecución
                    Activa int.SW planif dirigida a p. a expulsar;
        }
        .......
    }
```

En el tratamiento de la interrupción software de planificación, se llevaría a cabo el cambio de contexto involuntario:

```
tratamiento_int_software_planificacion() {
    Llamar al planificador;
    Realizar cambio de contexto involuntario al proceso elegido;
}
```

En el segundo ejemplo planteado, si se trata de un multiprocesador, es posible que el proceso desbloqueado expulse a un proceso que esté ejecutando en otro procesador (expulsará al proceso en ejecución que tenga menos prioridad). Por tanto, la interrupción software de proceso estará dirigida a dicho proceso de baja prioridad, enviándose también una IPI al procesador donde ejecuta dicho proceso.

Surge en este punto una decisión de diseño importante: en el caso de que dentro de este anidamiento en el tratamiento de eventos haya una llamada al sistema y/o el tratamiento de una excepción (que, evidentemente, tendrán que estar en el nivel más externo del anidamiento), ¿se difiere el cambio de contexto involuntario hasta que termine también el tratamiento de los posibles eventos síncronos anidados o sólo hasta que termine la ejecución de las rutinas de tratamiento de interrupción? Dicho de otro modo, ¿se usa una interrupción software de proceso no expulsiva o expulsiva, respectivamente, para diferir el cambio de contexto involuntario? Esta decisión tiene bastante trascendencia, dando lugar, respectivamente, a dos tipos de sistemas operativos: sistemas con un núcleo no expulsivo frente a sistemas con un núcleo expulsivo. A continuación, se analizan estas dos estrategias.

Núcleo no expulsivo

Con este modelo, el cambio de contexto se difiere hasta que, además de las rutinas de interrupción anidadas, termine también la llamada al sistema y/o la rutina de tratamiento de excepción en ejecución, si las hubiese. El diseño original de UNIX responde a este modelo, y así se comportaba Linux hasta que apareció la versión 2.6. Este esquema, como se verá más adelante, reduce apreciablemente los problemas de sincronización, creando un sistema más fiable y con menor sobrecarga.

Un proceso que realiza una llamada al sistema continuará haciéndolo hasta que se bloquee (o finalice su ejecución), realizando un cambio de contexto voluntario, o hasta completar la llamada. Mientras realiza la llamada, se procesarán las interrupciones que sucedan pero éstas no producirán un cambio de contexto. El resultado es que el sistema operativo, en cuanto a la ejecución de llamadas al sistema, se comporta como un monitor, eliminando la necesidad de sincronizaciones complejas y creando un sistema operativo robusto y fiable.

Sin embargo, la concurrencia del sistema operativo queda limitada ya que no se pueden ejecutar llamadas concurrentemente, lo que afecta negativamente al tiempo de respuesta del

sistema. En concreto, la **latencia de activación** de un proceso, es decir, el tiempo que transcurre desde que se produce el evento que desbloquea un proceso hasta que éste reanuda su ejecución, es, en el peor caso, bastante considerable, e incluso intolerable para un sistema con un perfil de tiempo real o, en general, que requiera un buen tiempo de respuesta (por ejemplo, aplicaciones multimedia).

Téngase en cuenta que cuando un proceso de prioridad máxima está bloqueado esperando una interrupción y ésta se produce, tendrá que esperar para poder ejecutar, en el peor de los casos, a que termine la llamada al sistema en ejecución en ese instante (más exactamente la fase correspondiente de la llamada al sistema), así como el tratamiento de los posibles fallos de página que se produzcan durante la misma (siempre que éstos no impliquen un bloqueo, puesto que en ese caso ya entraría a ejecutar ese proceso de prioridad máxima), junto con el anidamiento de todas las rutinas de interrupción hardware y software existentes en el sistema. Para calcular la mayor latencia (el peor **tiempo de respuesta**) habría que usar el tiempo máximo que puede requerir la ejecución de cada una de estas operaciones y, además, habría que añadirle la posible latencia de la interrupción, así como el tiempo del propio cambio de contexto. Esta espera tendrá lugar incluso aunque el proceso que estaba realizando la llamada al sistema sea de prioridad mínima, produciéndose una **inversión de prioridades** (un proceso de alta prioridad listo para ejecutar debe esperar a que otro de prioridad menor complete una cierta actividad antes de proseguir su ejecución). Hay que resaltar que se trata de una inversión de prioridades con una duración máxima acotada, aunque con un valor que puede ser considerable.

De estos tiempos que forman parte de la latencia de activación, el más crítico es el de la llamada al sistema, puesto que generalmente es significativamente más largo que el de los restantes eventos. Es por ello que en este tipo de núcleos hay que evitar incluir llamadas al sistema que tengan fases de ejecución largas. Así, por ejemplo, no sería una buena estrategia de cara a mantener una latencia razonable en un núcleo no expulsivo incluir dentro del sistema operativo llamadas al sistema que proporcionan servicios gráficos. Este tipo de funciones requieren operaciones largas, complejas y no bloqueantes, que manipulan *buffers* de memoria que contienen información gráfica para realizar transformaciones, cálculos de visibilidad, etc.

En cualquier caso, aunque se evite la inclusión de este tipo de llamadas en el núcleo, pueden producirse ocasionalmente situaciones de larga latencia en otro tipo de llamadas presentes indefectiblemente en cualquier sistema operativo. Supóngase, por ejemplo, una llamada de lectura de un fichero que solicita un número elevado de bloques, tal que se da la circunstancia de que todos ellos ya están en memoria, en la caché del sistema de ficheros, y, además, ocurre que las páginas del *buffer* de lectura especificado en la llamada están también residentes en memoria. Esta llamada transcurrirá en una sola fase (es decir, sin bloqueos) y su duración puede ser considerablemente larga, puesto que implica copiar todos los bloques solicitados desde la caché del sistema de ficheros al *buffer* de la aplicación. Para reducir la latencia en estas situaciones patológicas, en un núcleo no expulsivo se podría incluir explícitamente en ciertos puntos del código de aquellas llamadas al sistema donde potencialmente pudiera darse este problema (por ejemplo, en cada iteración del bucle de lectura que trata un bloque), donde las condiciones fueran adecuadas (como se verá más adelante, en un contexto atómico, donde se estuviera ejecutando siempre en el contexto de una llamada al sistema, con las interrupciones habilitadas y sin estar en posesión de un *spinlock* en el caso de un multiprocesador), una comprobación de si hay un proceso más prioritario listo para ejecutar y, en caso afirmativo, hacer un cambio de contexto al mismo. El método denominado *fixed preemption points*, que aparece en la versión *System V Release 4.2* de UNIX, y la técnica llamada *Voluntary Kernel Preemption* (que esconde el punto de comprobación en la llamada `might_sleep`), que se incorporó a Linux antes de que éste se convirtiera en un núcleo realmente expulsivo, son dos ejemplos que siguen básicamente el esquema descrito en este párrafo.

Dado que con este esquema el cambio de contexto involuntario se debe diferir hasta que termine la llamada al sistema en curso, en el caso de que hubiera alguna, hay que usar una interrupción software de proceso no expulsiva para tal fin.

Hay que tener en cuenta que, puesto que con este esquema la llamada al sistema interrumpida continúa aunque se haya activado la interrupción software de planificación, podría ocurrir que la llamada no termine sino que el proceso se bloquee realizando un cambio de

contexto voluntario (dicho de manera informal, al proceso que queríamos quitarle el procesador, lo ha dejado voluntariamente antes). En este caso, cuando el proceso se desbloquee posteriormente y complete la llamada al sistema (o el tratamiento de la excepción) no tendría sentido volver a llamar al planificador. Para evitarlo, habría que desactivar la interrupción software de planificación asociada a un proceso, en caso de que la hubiera, cuando éste realiza un cambio de contexto voluntario.

Núcleo expulsivo

En el esquema expulsivo, el cambio de contexto se difiere sólo hasta que terminen las rutinas de interrupción. Windows y la versión 2.6 de Linux son de este tipo. Como se apreciará en la sección que estudia la sincronización, esta estrategia genera problemas de sincronización más complejos ya que se puede producir un cambio de contexto involuntario en mitad de una llamada, pudiendo, por tanto, haber llamadas concurrentes. Esto requiere esquemas de sincronización más sofisticados, y, en general, menos eficientes, aumentando las posibilidades de que haya errores en el código del sistema operativo. Sin embargo, puede mejorar considerablemente el tiempo de respuesta del sistema.

En este tipo de núcleos, la latencia de activación de un proceso no se ve afectada por el tiempo de tratamiento de los eventos síncronos, sino sólo por los asíncronos. Por tanto, para el cálculo de la **peor latencia** posible, sólo hay que tener en cuenta el tiempo máximo que puede requerir la ejecución de todas las rutinas de interrupción hardware y software existentes en el sistema y, además, añadirle la posible latencia de la interrupción, así como el tiempo del propio cambio de contexto.

Como se analizará en la sección sobre la sincronización, para evitar condiciones de carrera entre llamadas concurrentes en este tipo de núcleos, en ciertos fragmentos del código de una llamada se inhabilita la expulsión de procesos (es decir, se inhibe la interrupción software de planificación). Este tiempo de inhibición, concretamente, su valor máximo, también hay que incluirle en el cálculo del peor tiempo de latencia para este tipo de núcleos.

En cualquier caso, la reducción en la latencia, tanto en su valor medio como en el máximo, es muy significativa, disminuyendo apreciablemente la duración de los intervalos en los que se produce inversión de prioridades. Los primeros experimentos con la versión expulsiva de Linux mostraron unas mejoras en la latencia media y máxima de al menos un orden de magnitud

Con este esquema hay que usar una interrupción software de proceso expulsiva para realizar los cambios de contexto involuntarios diferidos. Por lo demás, el tratamiento de la interrupción software de planificación sería el mismo que en un núcleo no expulsivo.

Analizando las diferentes situaciones donde se producen cambios de contexto involuntarios, podría parecer que el mecanismo de la interrupción software de proceso no es estrictamente necesario en todas ellas. Concretamente, la interrupción software de planificación parece innecesaria en el caso de que el cambio de contexto involuntario se desencadene dentro de una llamada al sistema al desbloquear un proceso que debe expulsar justamente al que está ejecutando la llamada: dado que no hay anidamiento de eventos y un proceso debe pasarle directamente el procesador al otro, ¿por qué motivo no se realiza un cambio de contexto *in situ*, eliminando la sobrecarga introducida por la gestión de la interrupción software, como se hace cuando hay un cambio de contexto voluntario, tal como muestra el siguiente fragmento de código?

```
llamada_X () {
      .......
      Si (condición) {
            Desbloquear(cola de evento);
            Si prioridad desbloqueado Q > alguno en ejecución
                  P = proceso en ejecución de menor prioridad;
                  Si proc. a expulsar P == proc. realiza llamada
                        cambio de contexto de P a Q
                  Sino
                        Activa int. SW plan. dirigida a P;
      }
```

```
        . . . . . .
}
```

El principal problema de este cambio de contexto directo es que, antes de realizarlo, habrá que comprobar explícitamente que en ese punto no está inhabilitada la expulsión de procesos (es decir, que no está inhibida la interrupción software de planificación). Utilizando siempre la interrupción software de planificación esta comprobación es automática.

En cuanto a la menor eficiencia de este tipo de núcleos frente a los no expulsivos, como se verá en la sección dedicada a la sincronización, esto se debe principalmente a la necesidad de incorporar mecanismos de sincronización para evitar condiciones de carrera en la ejecución concurrente de llamadas. Además de por aspectos vinculados con la sincronización, un núcleo expulsivo tiene mayor sobrecarga porque genera, en promedio, un número mayor de cambios de contexto, como se puede apreciar en los dos siguientes ejemplos:

- Un núcleo expulsivo se puede *ahorrar* un cambio de contexto involuntario cuando se da una situación en la que se debe expulsar un proceso que está realizando una llamada al sistema bloqueante, ya que el proceso deja voluntariamente el procesador antes de ser expulsado.
- En situaciones donde estando en ejecución un proceso de baja prioridad realizando una llamada al sistema se produce una secuencia de eventos que van desbloqueando a procesos de prioridad creciente, en un núcleo no expulsivo puede ocurrir que se produzcan todos ellos durante la llamada al sistema y, por tanto, al final de la misma, sólo haya un cambio de contexto involuntario al proceso de mayor prioridad de la secuencia. Sin embargo, en un núcleo expulsivo se producirá en este caso una secuencia de cambios de contexto involuntarios.

Núcleo para sistemas de tiempo real

Los sistemas de tiempo real deben garantizar que se cumplen los plazos de tiempo requeridos por una determinada aplicación, incluso en el peor de los casos. Para satisfacer este requisito, un sistema debe acotar su peor tiempo de latencia de interrupción y de activación.

En principio, podría parecer que el uso de un sistema con un núcleo expulsivo que garantice bajas latencias en el peor de los casos es suficiente para cumplir estas necesidades. Sin embargo, no es así y se requiere una importante transformación en el modo de gestión de los eventos y los procesos que realiza un sistema de propósito general, como se explica a continuación.

El modelo de operación de un sistema de propósito general asume que el tratamiento de cualquier interrupción, aunque sea de mínima prioridad, es más urgente que la ejecución de cualquier proceso, aunque sea de máxima prioridad. En un sistema de tiempo real esto no es así y, por tanto, hay que integrar de alguna forma las prioridades de las interrupciones y de los procesos.

En principio podría parecer que basta con integrar en un esquema único las prioridades de las interrupciones y de los procesos, y añadir un mecanismo al sistema operativo de propósito general que asegure que mientras ejecuta un proceso no se active el tratamiento de las interrupciones que se consideren menos urgentes que el mismo. Sin embargo, el asunto es más complicado.

Para entenderlo, considere el siguiente ejemplo: estando activo el tratamiento de una interrupción de baja prioridad, se ve interrumpido por una de mayor prioridad que desbloquea un proceso que se ha definido como más urgente que la interrupción de baja prioridad. En ese ejemplo, se debería ceder el control a ese proceso, sin retrasarle hasta que se complete la rutina de tratamiento de la interrupción de prioridad más baja (o interrupciones, en caso de haber anidamiento); pero si así lo hacemos, ¿quién, cómo y cuándo se completarán esas rutina(s) de interrupción inacabadas(s)? En el caso de un sistema de propósito general, no hay forma de hacerlo, porque se implementa un modelo en el que bajo ninguna circunstancia la ejecución de una rutina de interrupción queda inacabada para dar paso a otro proceso.

Una posible solución para dotar al procesamiento de una interrupción de un contexto independiente, que pueda ser interrumpido por la ejecución de un proceso para ser posteriormente completado, es hacer que el tratamiento de la misma lo realice un proceso/*thread* de núcleo. Cada nivel de interrupción tendría asignado un proceso de núcleo por cada

procesador, con su propia pila de núcleo como todos los procesos, de manera que cuando se produjera una interrupción de ese nivel se activaría el proceso de núcleo asociado. El propio mecanismo de planificación de procesos definiría un esquema integrado de niveles de prioridad que englobaría también a las interrupciones.

En el ejemplo planteado previamente, el proceso desbloqueado tendrá más prioridad que el proceso de núcleo que ejecuta el tratamiento de la interrupción de baja prioridad, por lo que pasará a ejecutar inmediatamente. El tratamiento de dicha interrupción de baja prioridad se completará cuando el proceso de núcleo que la sirve sea el proceso listo en el sistema con mayor prioridad.

Hay que resaltar que también hay que reconvertir a procesos de núcleo las interrupciones software de sistema, dado que al fin y al cabo son prolongaciones de las interrupciones de los dispositivos, existiendo un proceso de núcleo por cada interrupción software de sistema y por cada procesador.

Obsérvese que con esta solución, la activación de una rutina de interrupción requiere un cambio de contexto, lo que, en principio, parece ineficiente. Téngase en cuenta, sin embargo, que se trata de un cambio de contexto a un proceso de núcleo, que es mucho más eficiente.

En cuanto a la peor latencia de activación posible, que, al fin y al cabo, es el dato que más nos interesa, en un núcleo de estas características, cuando se produce una interrupción de alta prioridad que desbloquea a un proceso de máxima prioridad, en el peor de los casos, habrá que esperar la latencia de la interrupción más el tiempo de dos cambios contexto: uno involuntario, de la actividad en ejecución al proceso de núcleo asociado a la interrupción, y otro voluntario, del proceso de núcleo al proceso de máxima prioridad. Nótese que estos dos cambios de proceso sólo requieren un cambio de mapa de proceso. Obsérvese también que la peor latencia es independiente del número de interrupciones presentes en el sistema y de la duración de sus rutinas.

Dentro de Linux, la línea de desarrollo `PREEMPT_RT`, que no está incluida por el momento en el núcleo oficial, intenta incorporar funcionalidades de tiempo real al núcleo de Linux usando una solución similar a la descrita en esta sección.

4 Operaciones sobre los procesos

En las secciones anteriores hemos llegado a conocer cómo se desarrolla la vida de un proceso, alternando fases en modo usuario y en modo sistema. Para completar la panorámica, falta analizar cómo nace un proceso y cómo, por ley de vida, fenece. En este análisis seguiremos manteniendo el enfoque neutral de esta exposición, no entrando en los detalles específicos de cómo se crean y destruyen los procesos en un determinado sistema operativo, sino destacando aquellos aspectos generales, que se pueden aplicar a cualquier sistema operativo.

4.1 Creación de un proceso

Todos los sistemas operativos multiprogramados ofrecen servicios para que un proceso pueda crear otros procesos. Normalmente, se trata de una función en la que, además de otros posibles datos específicos de cada sistema operativo, se indica el fichero ejecutable que contiene el programa que se desea ejecutar, así como los argumentos que se pretende suministrar al mismo. Observe que el hecho de que un proceso cree a otro y éste, a su vez, pueda hacer lo mismo, y así sucesivamente, da lugar a una jerarquía de procesos. En algunos sistemas operativos esta jerarquía es puramente coyuntural, sin ninguna trascendencia sobre el modo de operación del sistema. Sin embargo, en otros, como se verá al analizar cómo terminan los procesos, es un factor que influye considerablemente en todas las operaciones vinculadas con los procesos.

Los sistemas operativos de la familia UNIX, como se comentó en la introducción del tema, presentan un modelo de creación de procesos alternativo, distinguiendo entre una operación que duplica el proceso actual (`fork`), creando un nuevo proceso que ejecuta el mismo programa, y una que causa que el proceso que la invoca pase a ejecutar el programa especificado (`exec`).

En primer lugar, analizaremos qué operaciones conlleva el modelo que podíamos denominar tradicional y, a continuación, veremos cuáles son las operaciones requeridas en el modelo de creación de procesos de UNIX.

Para entender la problemática asociada a la creación de un proceso, hay que considerar, en primer lugar, cómo debe ser su contexto inicial. La ejecución de un nuevo programa debe comenzar por su punto de entrada, cuya dirección estará almacenada en el fichero ejecutable, usando la pila de usuario de este nuevo proceso, y de forma que el procesador esté en modo usuario y con las interrupciones habilitadas. En principio, parece que sería suficiente con almacenar en el BCP del nuevo proceso estos valores, de manera que cuando un proceso realice un cambio de contexto al nuevo proceso, éste ejecute en las condiciones requeridas. Esta estrategia, sin embargo, tiene algunas deficiencias:

- Falta de homogeneidad en los cambios de contexto. Hasta este punto, todos los cambios de contexto analizados intercambiaban un proceso en modo sistema por otro que también ejecutaba en ese mismo modo. Con esta estrategia se rompería esta uniformidad, ya que en el caso de un cambio de contexto a un nuevo proceso, éste empezaría directamente en modo usuario. Asimismo, se dejaría de cumplir que todos los cambios de modo de sistema a usuario se realizan mediante una instrucción de retorno de tratamiento de un evento.
- Carencia de atomicidad en el cambio de contexto. Para dar paso al nuevo proceso en el cambio de contexto hay que actualizar registros del procesador como el contador de programa y el registro de estado, pero para ello se utilizarán varias instrucciones, lo que puede causar problemas de coherencia al poder ocurrir interrupciones entre las mismas. Sería preferible actualizar estos valores de forma atómica.

Para paliar estos problemas se utiliza, habitualmente, un artificio como el que se explica a continuación:

- Se incluye en la pila de sistema del nuevo proceso la misma información que almacena el procesador cuando se produce un evento, aunque éste no haya existido realmente. El contador de programa almacenado en la pila corresponderá a la dirección inicial del programa y el registro de estado indicará que se ejecute en modo usuario y con las interrupciones habilitadas.
- El contador de programa almacenado en el BCP del nuevo proceso hará referencia a una función del sistema operativo que actuará como una especie de lanzadera, mientras que el registro de estado inicial especificará una ejecución en modo sistema. Esta función de lanzadera terminará con una instrucción de retorno de tratamiento de un evento.
- En un cambio de contexto que activa un nuevo proceso, se comenzará a ejecutar la función de lanzadera, en modo sistema y con la pila de sistema del nuevo proceso activada. Esta función de lanzadera termina con un retorno de tratamiento de evento, que activa la ejecución del programa en modo usuario, utilizando la pila de usuario del nuevo proceso. A todos los efectos, es como si se hubiera reanudado la ejecución de un programa desde el punto donde se detuvo, aunque realmente nunca haya ejecutado previamente.

Con esta estrategia, se consigue homogeneidad en los cambios de contexto, ya que todos transitan de un proceso en modo sistema a otro que está en ese mismo modo, y en los cambios a modo usuario, puesto que todos se producen con la instrucción de retorno de tratamiento de un evento. Asimismo, se logra la atomicidad requerida, al conseguir modificar el contador de programa y el registro de estado con una única instrucción (la de retorno de tratamiento de un evento).

En la zona central y derecha de la figura 20, se puede apreciar la disposición inicial del contexto de un proceso, tal como se acaba de describir: el BCP hace referencia a una función lanzadera que ejecuta en modo sistema, y tal que al completarse (RETI) retorna al punto inicial del programa (`start`), especificado en el fichero ejecutable, ejecutando en modo usuario. Aunque con esto se completa la parte que concierne al sistema operativo, hay también algún artificio en la parte de usuario. En la zona izquierda de la figura se muestra que el punto de arranque de un programa no se corresponde con la función inicial del mismo tal como puede creer el programador (en este caso, la función `main` al tratarse de un programa en C), sino con una función de biblioteca incluida por el montador. Su propósito es llamar a la función inicial del programa propiamente dicha y asegurarse de que se invoque la función que notifica al

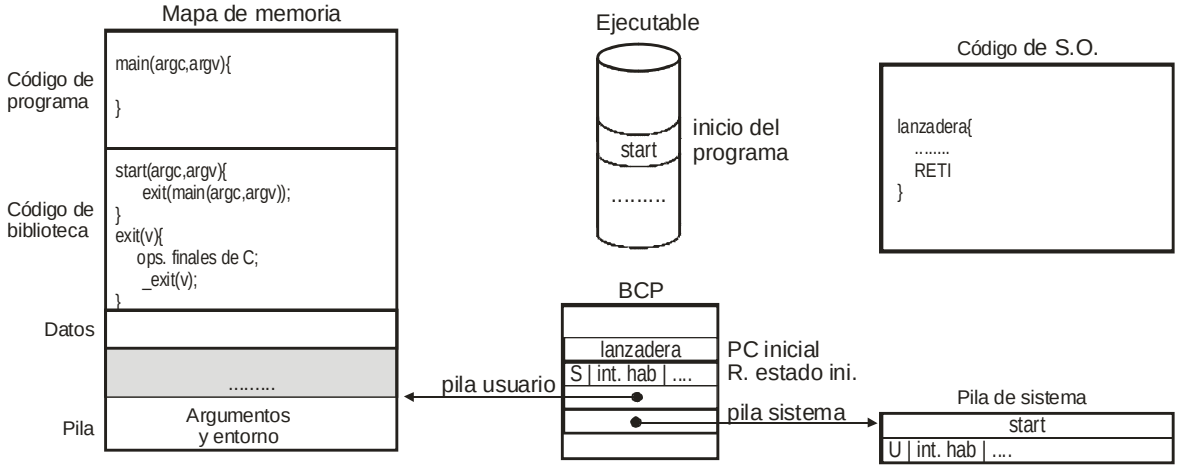


Figura 20 *Disposición inicial en la creación de un proceso.*

sistema operativo que el proceso ha terminado (`exit`, si se trata de un programa C, tal como analizaremos en la siguiente sección) en caso de que el programa se complete sin hacerlo. Si el programa usa bibliotecas dinámicas, el arranque del proceso es todavía más complejo, como se analiza al estudiar la gestión de memoria.

Operaciones implicadas en la creación de un proceso

La creación de un nuevo proceso es una de las funciones más complejas del sistema operativo y requiere la colaboración de varios componentes del mismo para completar la labor requerida. En esta sección se especifican las operaciones de alto nivel presentes, de una forma u otra, en todos los sistemas operativos. A continuación, se enumeran dichas operaciones:

- Asigna un identificador al proceso. Generalmente, se busca el primer valor que esté libre y que sea mayor que el último asignado.
- Reserva un BCP para el nuevo proceso. Si la tabla de procesos es un vector, habrá que buscar una posición libre. Si se trata de una lista, habrá que reservar una zona de memoria para el BCP e insertarlo en la lista.
- Lee la cabecera del fichero ejecutable y crea el mapa de memoria inicial del proceso basándose en el contenido del mismo. Esta operación se analiza en detalle al estudiar la gestión de memoria.
- Crea la pila de usuario incluyendo en ella los argumentos y las variables de entorno especificados al crear el proceso.
- Reserva espacio para la pila de sistema del proceso. Inicia esa pila de manera que contenga la dirección de inicio del programa y un valor que se corresponderá con el registro de estado inicial del proceso, tal como se ha explicado previamente. En muchos sistemas es suficiente con reservar un solo marco de página de memoria para esta pila.
- Inicia los diversos campos del BCP con los valores correspondientes. Algunos serán copia de los valores del proceso padre, como, por ejemplo, el identificador de usuario. En otros se fijará un valor inicial razonable. Así, por ejemplo, todos los campos cuyo cometido sea almacenar información del gasto de recursos de un proceso se iniciarán a 0. Asimismo, se fijará un valor inicial de los registros tal que cuando se active el proceso por primera vez comience en la rutina de lanzadera, en modo sistema y usando la pila de sistema del proceso.
- Pone el proceso en estado de listo para ejecutar y lo incluye en la cola de procesos en ese estado. Téngase en cuenta que habrá que incluir el BCP en otras listas, tales como las vinculadas con las relaciones de parentesco o las requeridas para una búsqueda rápida a partir del identificador de proceso.

En caso de tratarse de un sistema operativo con una arquitectura de tipo micronúcleo, todos los componentes del sistema recibirán la notificación de que se está creando un nuevo proceso y cada uno de ellos creará la parte de contexto del proceso que le corresponde. Así, por ejemplo,

cuando al sistema de ficheros se le comunica que se está creando un nuevo proceso, habilita su propio BCP para el nuevo proceso donde sólo almacenará información vinculada con el uso de ficheros por parte del mismo (ficheros abiertos por el proceso, directorio actual del mismo, etc.). Téngase en cuenta que Windows, aunque no tenga una arquitectura de este tipo, sí tiene una organización con un ejecutivo y múltiples subsistemas, por lo que sigue esta misma estrategia de propagar la notificación de la creación de un proceso a todos los componentes del sistema, cada uno de los cuales habilitará su propio tipo de BCP para almacenar la información que le incumbe.

Además de proporcionar a las aplicaciones servicios para crear procesos, el sistema operativo ofrece a sus propios módulos una función interna para crear procesos (hilos) de núcleo, en la que se especifica, habitualmente, la dirección de la función del sistema operativo que llevará a cabo la labor correspondiente, así como un argumento para la misma. Este tipo de procesos usará un BCP y una pila de sistema, pero no tendrá asociado un mapa de memoria, ni una pila de usuario. Asimismo, no estará vinculado a un ejecutable y su ejecución se restringirá a la función del código del sistema operativo especificada en su creación. Comparándolo con los procesos convencionales, en ambos casos se comienza por una función de código del sistema operativo. Pero en el caso de los procesos normales, se trata de la función de lanzadera que de forma inmediata retorna a modo usuario, mientras que en los procesos de núcleo se corresponde con la función especificada que ejecuta en modo sistema durante toda la vida del proceso.

Operaciones requeridas en el modelo de creación de procesos de UNIX

En esta sección se pretende mostrar cómo se trasladan las operaciones identificadas en la sección anterior al modelo de procesos de UNIX, analizando cómo se reparten entre las llamadas `fork` y `exec`. Hay que reiterar que no se trata de un estudio de ese sistema operativo y, por tanto, no se entrará a describir de forma detallada qué operaciones conllevan estas dos llamadas al sistema de UNIX. Se recomienda al lector interesado que revise alguno de los múltiples libros dedicados a los aspectos internos de algún sistema operativo de la familia UNIX.

Comencemos con las operaciones asociadas a la llamada `fork`:

- Asigna un identificador al proceso.
- Reserva un BCP para el nuevo proceso.
- Duplica el mapa del proceso. Esta operación de duplicado debe optimizarse para conseguir un buen rendimiento en los sistemas UNIX, como se estudia en el tema de gestión de memoria.
- Reserva espacio para la pila de sistema del proceso. Inicia esa pila de manera que cuando se ejecute por primera vez este proceso comience en modo usuario, justo después de la llamada `fork`, habiendo devuelto un 0 en la misma.
- Duplica el contenido del BCP del padre, iniciando específicamente sólo aquellos campos que tendrán valores diferentes en el hijo, como, por ejemplo, el identificador de proceso. Asimismo, se fijará un valor inicial de los registros tal que cuando se active el proceso por primera vez comience en una rutina de lanzadera, en modo sistema y usando la pila de sistema del proceso. En la figura 21 se muestra la disposición inicial del nuevo proceso.
- Pone el proceso en estado de listo para ejecutar y lo incluye en la cola de procesos en ese estado, así como en otras listas de procesos del sistema operativo.

A continuación, se enumeran las operaciones vinculadas con la llamada `exec`:

- Libera el mapa de memoria del proceso. Dado que hay que cambiar de programa, se debe abandonar el mapa actual del proceso.
- Lee la cabecera del fichero ejecutable y crea un nuevo mapa de memoria del proceso basándose en el contenido del mismo.
- Crea la pila de usuario incluyendo en ella los argumentos y las variables de entorno especificados en la llamada.
- Modifica algunos campos del BCP, como, por ejemplo, los correspondientes al estado de las señales capturadas, que pasan a tener asociada la acción por defecto.

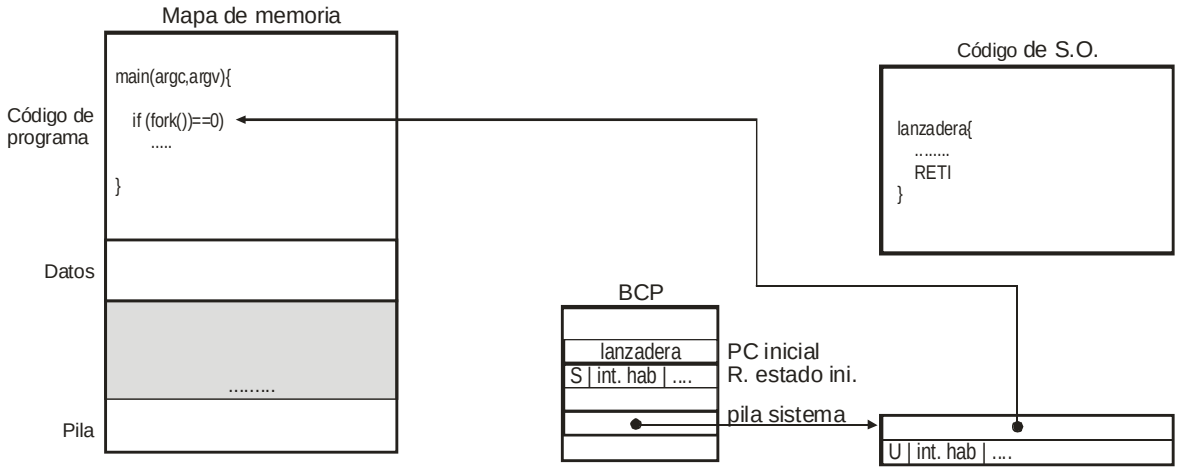


Figura 21 *Disposición inicial en la creación de un proceso con* `fork`.

- Modifica el contenido de la pila de sistema del proceso de manera que cuando se termine la llamada y se retorne a modo usuario la ejecución comience por el punto de entrada del programa y usando la nueva pila de usuario, como se puede apreciar en la figura 22.

4.2 Terminación de un proceso

Tarde o temprano, todos los procesos acaban terminando (como muy tarde, durante el proceso de apagado del sistema). En esta sección se analiza qué operaciones conlleva la terminación de un proceso. En primer lugar, se estudia el problema de forma general y, a continuación, se particulariza para el caso de UNIX, donde las operaciones de terminación se reparten entre las llamadas `_exit` y `wait`.

Un primer aspecto que conviene distinguir es de qué manera termina el proceso: de forma voluntaria, invocando una llamada al sistema concebida para tal fin, o involuntaria, abortado por el sistema operativo.

En cuanto a la primera forma de terminar, como ya se pudo apreciar en las figuras 20 y 22, normalmente, el programa (o la función `start`) invoca la función del lenguaje correspondiente destinada a tal fin (en el caso del lenguaje C, se trata de la función `exit`), en vez de invocar directamente la llamada al sistema (`_exit`, en el caso de UNIX). Ésta es la forma recomendada de terminar un programa, puesto que cada lenguaje puede necesitar realizar ciertas acciones ajenas al sistema operativo antes de que concluya el programa (por ejemplo, en C, se deben volcar los *buffers* de la biblioteca de entrada/salida estándar y ejecutar las funciones `atexit`).

Con respecto al segundo tipo de muerte, la involuntaria, se pueden presentar diversos casos, algunos de los cuales se muestran a continuación:

- El proceso ejecutando en modo usuario produce una excepción en el procesador debido a

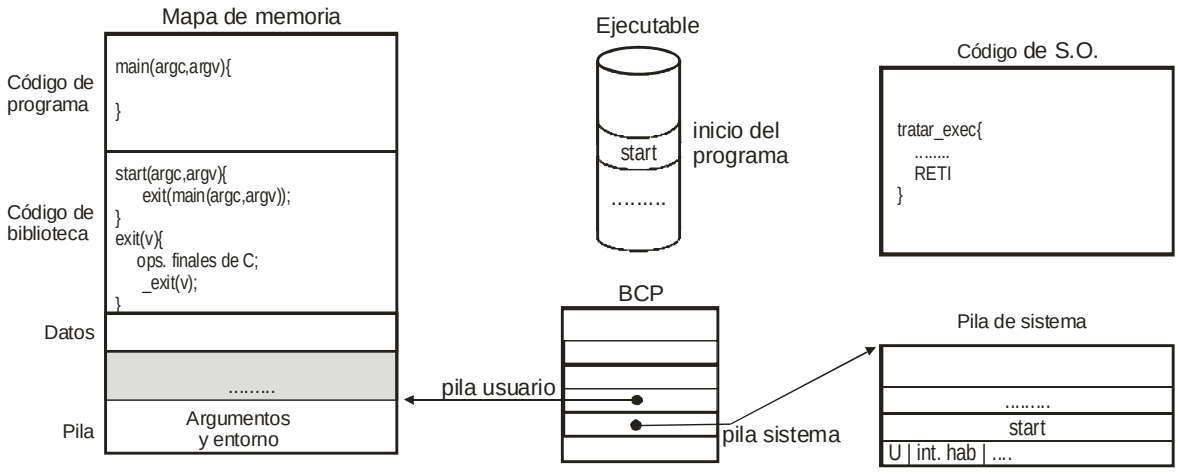


Figura 22 *Disposición inicial en la activación de un programa mediante* `exec`.

un error en su código.
- También puede terminarse incondicionalmente la ejecución de un proceso por alguna otra condición de error, no vinculada directamente con una excepción. Así, el sistema operativo puede abortar un proceso si supera algún límite máximo en el uso de recursos (por ejemplo, el uso máximo del procesador por parte de un proceso).
- Un proceso aborta la ejecución de otro (en UNIX, mediante la llamada `kill`).
- Un usuario aborta la ejecución de un proceso mediante el teclado (por ejemplo, usando `Control-C`, que es la combinación por defecto para abortar a un proceso en los sistemas UNIX).

Hay que resaltar que la mayoría de los sistemas operativos permiten que el proceso pueda declarar que, ante una de estas situaciones, no sea abortado inmediatamente, sino que pueda ejecutar una especie de últimos deseos para terminar su labor de forma coherente.

En el caso de UNIX, el manejo de todas estas situaciones excepcionales está englobado dentro de un mecanismo más general: las señales. Este mecanismo proporciona un medio de interacción entre el sistema operativo y un proceso, o, directamente, entre los procesos. Asimismo, ofrece la posibilidad de que el propio programa establezca un tratamiento específico para la señal: capturarla, lo que permite realizar esos últimos deseos, ignorarla, haciendo que el proceso no se vea afectado por esa señal, o mantener la opción por defecto, que para la mayoría de las señales se corresponde con abortar el proceso.

Operaciones implicadas en la terminación de un proceso

Sea de forma voluntaria o involuntaria, el acto de terminar un proceso conlleva las siguientes operaciones de alto nivel:
- Libera el mapa de memoria del proceso.
- Cierra los ficheros abiertos por el proceso y libera todos los recursos que estuviera usando (por ejemplo, semáforos).
- Elimina el BCP de la cola de procesos listos, y de las otras listas donde estuviera incluido.
- Libera el BCP.
- Libera la pila de sistema del proceso.
- Activa el planificador y realiza un cambio de contexto voluntario al proceso seleccionado por el mismo.

Un aspecto conflictivo en esa lista de operaciones es cómo poder liberar la pila del sistema mientras se está ejecutando la rutina que finiquita al proceso que la necesita como soporte de su ejecución. Una posible solución es que, cuando un proceso termina, sea el proceso al que se le cede el procesador el encargado de liberar la pila de sistema del proceso terminado, puesto que en ese momento ya no se está usando. Esta estrategia se concretaría de la siguiente forma:
- En la rutina que termina un proceso se marcaría el estado del mismo como terminado, pero no se liberaría la pila del sistema ni el BCP.
- Justo después del cambio de contexto, se comprobaría si el proceso que acaba de dejar el procesador ya había terminado. En caso afirmativo, se liberará en ese momento tanto la pila de sistema como el BCP del proceso que acaba de fallecer.

Para ilustrar el uso de esta técnica, el programa 10 retoma el ejemplo del programa 9 e incluye en el mismo cómo se llevaría a cabo la liberación de recursos pendiente del proceso recién terminado. Observe que para hacer referencia al proceso que acaba de terminar, es necesario usar la variable `ant`.

Programa 10 Liberación de recursos del proceso terminado por parte del nuevo proceso.

```
BCP *ant, *post;
void cambio (BCP *prev, BCP *sig) {
    ant = prev;
    post = sig;
    cambio_contexto(...);
    if (ant->estado == TERMINADO) {
```

```
            liberar_pila(ant->pila_sistema);
            liberar_BCP(ant);
        }
}
```

Operaciones implicadas en la terminación de un proceso en UNIX

En el modelo de procesos de UNIX juega un papel muy importante la jerarquía de procesos, de manera que un proceso no termina completamente su ejecución, quedándose en el estado de zombi, hasta que el proceso padre recoge su estado de finalización mediante la llamada al sistema `wait`. Dado este requisito, nunca puede quedarse un proceso sin padre, por lo que cuando termina un proceso, para mantener la coherencia de la jerarquía, es necesario que sus hijos sean adoptados por otro proceso, concretamente, por el proceso `init`. Dados estos condicionantes previos, a continuación, se muestra el reparto de las operaciones de finalización de un proceso entre las dos llamadas al sistema involucradas: `_exit` y `wait`.

La mayoría de las operaciones están asociadas a la llamada `_exit`:

- Libera el mapa de memoria del proceso.
- Cierra los ficheros abiertos por el proceso y libera todos sus recursos.
- Elimina el BCP de la cola de procesos listos, y de las otras listas donde estuviera incluido.
- Pone al proceso en el estado de zombi.
- Establece que el nuevo padre de los hijos del proceso que termina, en caso de que los tuviera, pasa a ser el proceso `init`.
- Activa el planificador y realiza un cambio de contexto voluntario al proceso seleccionado por el mismo.

La llamada `wait` del padre completará el trabajo:

- Espera a que haya algún hijo en estado de zombi.
- Recoge la información de finalización del hijo.
- Libera su BCP.
- Libera su pila de sistema.

Nótese que gracias al reparto de trabajo entre ambas llamadas, se resuelve fácilmente el problema de liberar la pila de sistema del proceso que termina, puesto que de ello se encarga el proceso padre.

Escenarios de terminación de un proceso

La terminación de un proceso es una acción delicada, sobretodo, en el caso de que sea involuntaria. Para apreciar este problema, imaginemos que un proceso está en medio de una llamada al sistema como, por ejemplo, la que crea un proceso. Si se aborta ese proceso en medio de esa llamada, las estructuras de datos del sistema operativo pueden quedar incoherentes (por ejemplo, se podría haber interrumpido la llamada después de haber reservado un BCP y haber construido parte del mapa de memoria del nuevo proceso). Ante esa situación, se presentan dos opciones:

- Abortar la llamada pero dejando previamente el estado del sistema coherente, eliminando limpiamente el trabajo que llevaba hecho hasta ese momento (en el ejemplo planteado, se liberaría el BCP y la parte del mapa que ya estaba creada).
- Dejar que se complete la llamada antes de abortarle.

A continuación, se analizan las distintas situaciones que se pueden presentar a la hora de terminar un proceso en un sistema operativo con un núcleo expulsivo, dado que este tipo de sistemas presenta una mayor complejidad. En el programa 11, se incluye un posible tratamiento de cada una de ellas. A priori, se puede resaltar que el aspecto clave de la terminación involuntaria es que debe ser el propio proceso afectado el que se destruya a sí mismo, usándose una interrupción software de proceso no expulsiva cuando sea necesario.

1. Un proceso termina de forma voluntaria invocando la llamada al sistema disponible para tal fin. Las operaciones asociadas a la finalización del proceso (englobadas en la función

hipotética `fin_proceso` en el programa 11) se realizan dentro del tratamiento de la llamada.

2. Un proceso termina de forma involuntaria debido a una excepción. De manera similar al punto anterior, las operaciones de finalización del proceso se realizan dentro de la rutina de tratamiento del evento.
3. Mientras ejecuta un proceso, se produce un evento asíncrono que implica la finalización involuntaria de dicho proceso. Un ejemplo sería una interrupción de reloj que detecta que el proceso en ejecución ha sobrepasado su límite de uso de procesador. En este caso, y en los siguientes, se puede producir la situación problemática planteada al principio de la sección, es decir, que el proceso que se debe abortar esté en medio de una llamada. Dado que en este caso no es posible saber en qué punto se encontraba la llamada interrumpida, la única alternativa es diferir la terminación del proceso hasta que se complete la llamada. Para ello, se usa una interrupción software de proceso no expulsiva dedicada a este fin dirigida al proceso actualmente en ejecución. En el tratamiento del evento asíncrono que provoca la finalización del proceso, se marca el proceso como terminado y se activa una interrupción software de estas características, siendo en el tratamiento de dicha interrupción cuando se lleva a cabo la terminación real. Téngase en cuenta que podría haber problemas si el proceso a abortar, en vez de completar la llamada, se bloquea, puesto que seguirá vivo. Esta circunstancia se retomará en el siguiente punto.
4. Mientras ejecuta un proceso, se produce un evento que provoca que otro proceso aborte. Un ejemplo sería una situación en la que el primer proceso invoca una llamada al sistema para abortar al segundo (en UNIX, la llamada `kill`, que será el ejemplo planteado en el programa 11). El proceso que debe abortarse puede estar en distintas circunstancias:
   - En estado de listo para ejecutar habiendo sido expulsado previamente en un cambio de contexto involuntario. Dado que al tratarse de un núcleo expulsivo puede estar parado en medio de una llamada, no se puede abortar inmediatamente. Además, tampoco es posible en este caso conocer en qué punto se encontraba la llamada interrumpida, por lo que la única alternativa es diferir la terminación del proceso hasta que se complete la llamada. Como en el caso anterior, esto se puede conseguir activando una interrupción software de terminación en la rutina de tratamiento del evento que causa el aborto, que estará dirigida, en este caso, al proceso que se pretende abortar. De esta forma, cuando vuelva a ejecutar dicho proceso, en el momento que vaya a retornar a modo usuario, se tratará la interrupción software de terminación, como en el caso anterior, y concluirá. Vuelve a haber problemas si el proceso se bloquea antes de completar la llamada, como ocurría en el punto previo.
   - En estado de bloqueado o de listo pero habiendo estado previamente bloqueado. El proceso estará parado en medio de una llamada al sistema, pero, en este caso, en un punto conocido de la misma, precisamente donde se quedó bloqueado, por lo que se puede abortar la ejecución de la llamada, aunque dejando previamente coherente el estado del sistema. En este caso, en la rutina de tratamiento del evento que causa el aborto bastaría con marcar el proceso como terminado y, en caso de que estuviera bloqueado, moverlo desde la cola de procesos bloqueados correspondiente hasta la de procesos listos. Cuando vuelva a ejecutar el proceso que se pretende abortar, justo después del cambio de contexto, dentro de la rutina de bloqueo, comprobará su estado y, si detecta que se ha abortado, se puede finiquitar a sí mismo en ese punto, habiendo dejado previamente un estado coherente en el sistema. Para tratar la situación planteada en el tercer punto y en el primer caso de este punto (un proceso abortado que se bloquea, en vez de terminar la llamada al sistema que está ejecutando), se puede incluir una comprobación del estado del proceso también justo antes del cambio de contexto, realizando el mismo tratamiento que en la comprobación posterior. Nótese que, estrictamente, en este caso no es necesario generar una interrupción software de terminación vinculada al proceso que se quiere abortar en la rutina de tratamiento del evento que causa el aborto, ya que la

terminación del proceso puede realizarse en la propia llamada. Sin embargo, como se puede apreciar en el programa 11, por uniformidad con el otro caso, también se puede usar una interrupción software de terminación y dejar que el proceso se destruya a sí mismo en el tratamiento de la misma. Observe en dicho programa como la función `kill` envía la interrupción software al proceso abortado con independencia de cuál sea su estado.

- En ejecución, circunstancia que puede darse sólo en un multiprocesador, y que, por simplicidad, no se incluye en el programa 11. En este caso, en la rutina de tratamiento del evento que causa el aborto, además de dirigir una interrupción software de terminación al proceso afectado, al detectar que dicho proceso a abortar está ejecutando en otro procesador, se envía una IPI dispuesta para tal propósito al procesador involucrado.

Programa 11 Tratamiento de la finalización de un proceso bajo diversas circunstancias.

```
// Situación 1: terminación voluntaria
tratar_exit() {
    ......
    fin_proceso();
}
// Situación 2: terminación involuntaria por excepción
tratar_excepcion() {
    ......
    fin_proceso();
}
// Situación 3: terminación involuntaria por evento asíncrono
int_reloj() {
    ......
    Si (proceso_actual ha sobrepasado límite de tiempo de UCP){
          proceso_actual->estado=TERMINADO;
          activar_int_SW_terminacion(proceso_actual);
    }
    ......
}
tratamiento_int_software_terminacion() {
    ......
    Si (proceso_actual->estado==TERMINADO)
          fin_proceso();
    ......
}
// Situación 4: terminación involuntaria por otro proceso
kill (int pid) { // por simplicidad pid no corresponde a pr. actual
    ......
    proceso_abortado=buscarBCP(pid); // obtiene BCP del pid dado
    Si (proceso_abortado->estado==BLOQUEADO)
          mover BCP de abortado de cola de bloqueados a listos;
    proceso_abortado->estado=TERMINADO;
    activar_int_SW_terminacion(proceso_abortado);
    ......
}
int bloquear(cola de bloqueo) {
    ......
    Si (proceso_actual->estado==TERMINADO)
          return -1;
    cambio_contexto(...);
    Si (proceso_actual->estado==TERMINADO)
          return -1;
    ......
}
// ejemplo de llamada con posible bloqueo
```

```
leer_tubería() {
    ......
    Si (tubería_vacia) {
          ret=bloquear(cola de tubería);
          if (ret==-1) {
                dejar estado coherente;
                // se podría ejecutar fin_proceso aquí, pero por
                // uniformidad se puede hacer en el tratamiento
                // de la interrupción software no expulsiva.
                return -1; // al retornar se trata la int. software.
          }
    }
    ......
}
```

Como se comentó previamente, la mayoría de los sistemas operativos permite que un proceso pueda especificar que no quiere verse abortado cuando ocurra una determinada circunstancia, sino que desea ejecutar una determinada función para tratar esa situación. En el caso de UNIX, esto se corresponde con la captura de la señal asociada a esa circunstancia. En ese caso, el tratamiento sería similar al planteado en el programa 11, pero en vez de invocarse la función de terminación (`fin_proceso`) en cada una de las distintas situaciones, se manipulará la pila de usuario del proceso, de manera que cuando vuelva a ejecutar en modo usuario, se activará la rutina de tratamiento de la señal antes de proseguir desde el punto en el que vio interrumpido.

Nótese que en este punto de la exposición se ha completado la revisión de los tres usos básicos del mecanismo de la interrupción software:

- Diferir la ejecución de las operaciones que no son urgentes de una rutina de interrupción. Esta labor se lleva a cabo mediante una interrupción software de sistema dedicada a este fin. Obsérvese que esta interrupción software de sistema es expulsiva con independencia de que el núcleo sea o no expulsivo.
- Gestionar los cambios de contexto involuntarios. Para ello, se usa una interrupción software de proceso dedicada a esta labor que será expulsiva o no dependiendo del tipo de núcleo utilizado.
- Implementar la terminación involuntaria de los procesos. Para llevar a cabo esta labor, se utiliza una interrupción software de proceso destinada a este fin. Esta interrupción software tendrá carácter no expulsivo, con independencia de que el núcleo sea o no expulsivo.

4.3 Recapitulación sobre la vida de un proceso

Una vez estudiado cómo se crea un proceso y de qué manera termina, ya se dispone de toda la información necesaria para entender cómo se desarrolla la vida de un proceso.

- La vida de un nuevo proceso comienza cuando hay otro proceso que realiza un cambio de contexto, ya sea voluntario o involuntario, y cede el procesador al nuevo proceso. Este nuevo proceso comienza en modo sistema ejecutando una pequeña rutina del sistema operativo que se completa enseguida retornando a modo usuario, empezando en ese punto la ejecución del programa. El caso de un proceso de núcleo es la excepción, puesto que su ejecución se mantiene siempre en modo sistema.
- El proceso proseguirá en modo usuario hasta que se produzca un evento, presentándose distintos casos:
    - Si no implica cambio de contexto, al completarse la rutina de tratamiento vuelve a modo usuario, continuando la ejecución del programa en el punto donde se quedó.
    - Si se produce un cambio de contexto voluntario, que sólo puede suceder si el evento es síncrono, el proceso se queda parado en la rutina de bloqueo, cediendo el procesador a otro proceso. Cuando estando en ejecución otro proceso se produzca el evento que desbloquee el primer proceso, éste seguirá

detenido en el mismo punto, puesto que todavía no ha vuelto a ejecutar, pero ya será un candidato a ser elegido por el planificador. Finalmente, habrá un proceso que realice un cambio de contexto, que podrá ser voluntario o involuntario, y le ceda el control. El proceso continuará su ejecución desde el punto donde se quedó (en la rutina de bloqueo durante el tratamiento de un evento síncrono). Puede ocurrir que complete el tratamiento del evento y retorne a modo usuario, o que se vuelva a bloquear de nuevo en el tratamiento de ese mismo evento, repitiéndose el proceso. Observe que en ese último caso el proceso ha ejecutado todo el tiempo en modo sistema entre esos dos bloqueos.

  - Si hay un cambio de contexto involuntario, se habrá producido durante el tratamiento de una interrupción software de planificación. Se le ha retirado al proceso el uso del procesador, pero, al mantenerse en estado de listo para ejecutar, puede ser seleccionado en cualquier momento por otro proceso que realice un cambio de contexto, ya sea voluntario o involuntario. Cuando prosiga su ejecución, completará la rutina de interrupción software de planificación y continuará la ejecución en el punto donde se había quedado. Si se trata de un sistema operativo con un núcleo que no es expulsivo, seguirá ejecutando el programa en modo usuario. En caso de ser un núcleo expulsivo, al completarse la rutina de interrupción software de planificación, el proceso puede proseguir con la ejecución en modo usuario del programa o con la rutina de tratamiento de un evento síncrono, dependiendo que en qué punto se hubiera visto interrumpida su ejecución.

- Finalmente, se produce un evento, que hace transitar al proceso a modo sistema, y durante cuyo tratamiento se lleva a cabo la terminación del proceso, ya sea de forma voluntaria o involuntaria.

Obsérvese como el proceso se inicia ejecutando la parte final del tratamiento de un evento que nunca existió y termina llevando a cabo el tratamiento de un evento que no se completa.

Para completar esta sección, y como repaso del uso de los distintos tipos de interrupciones software, en la tabla 2 se muestra qué comportamiento, expulsivo o no expulsivo, tienen las distintas interrupciones software dependiendo del tipo de núcleo que se utilice.

A continuación, se plantean algunos ejemplos de trazas de ejecución de procesos que permiten ilustrar cómo se lleva a cabo la gestión de los eventos y la secuenciación de los procesos, distinguiendo entre un núcleo expulsivo y uno que no lo es.

Primer ejemplo

La figura 23 muestra cómo sería la secuencia de ejecución de la siguiente traza, tanto para un núcleo no expulsivo, en la parte superior, como para uno expulsivo, en la parte inferior:

- P tiene mayor prioridad que Q. P está ejecutando y Q es un nuevo proceso.
- P llamará a leer del terminal, cuyo buffer está vacío.
- Q alternará fases de usuario con llamadas no bloqueantes. Supóngase, por ejemplo, que invoca la llamada `getpid` de UNIX, que simplemente retorna el identificador del proceso.
- En medio de la llamada `getpid` se producirá una interrupción de la red, que no tiene una interrupción software de sistema asociada, y justo a la mitad de la misma, una interrupción del terminal, que tampoco la tiene.
- Después de leer del terminal, el proceso termina voluntariamente (en UNIX, se trata de

	Núcleo no expulsivo	**Núcleo expulsivo**
Interrupción software de sistema	Expulsiva	Expulsiva
Interrupción software de planificación	No expulsiva	Expulsiva
Interrupción software de terminación	No expulsiva	No expulsiva

Tabla 2 *Tipos de interrupciones software y su utilización dependiendo del tipo de núcleo.*

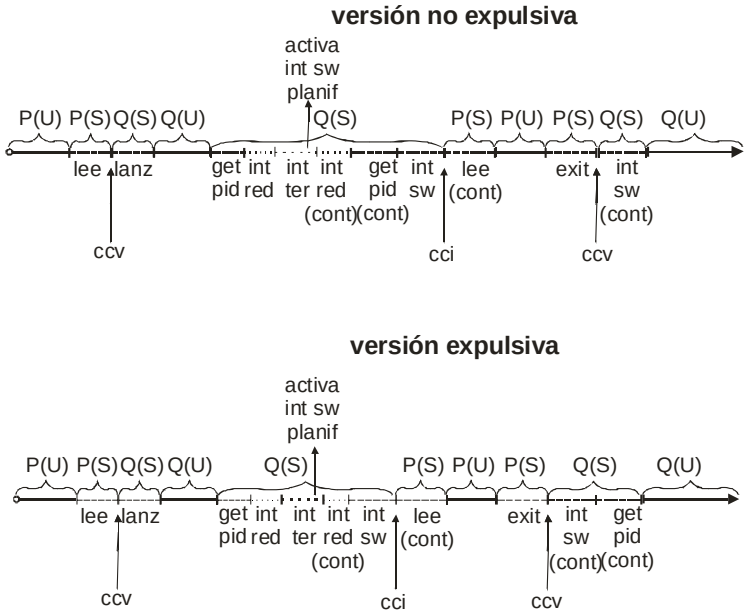


Figura 23 *Secuencia de ejecución que corresponde a la traza del primer ejemplo.*

una llamada a `exit`).

Segundo ejemplo

En la figura 24 se puede observar la secuencia de ejecución de la siguiente traza:

- P tiene mayor prioridad que Q. P está ejecutando y Q es un nuevo proceso.
- P causa un fallo de página, que requiere escribir una página modificada a disco y, a continuación, leer de disco la nueva página.
- Q alternará fases de usuario con llamadas a `getpid`.
- La primera interrupción de disco, que no tiene una interrupción software de sistema asociada, ocurre estando P en modo usuario. La segunda en medio de `getpid`.
- Después del fallo de página, P llama a `getpid`.
- En medio de la llamada un usuario aborta el proceso tecleando `Ctrl-C`, que suele ser la combinación de teclas usada en UNIX para esta operación. La interrupción del terminal tampoco tiene una interrupción software de sistema asociada.

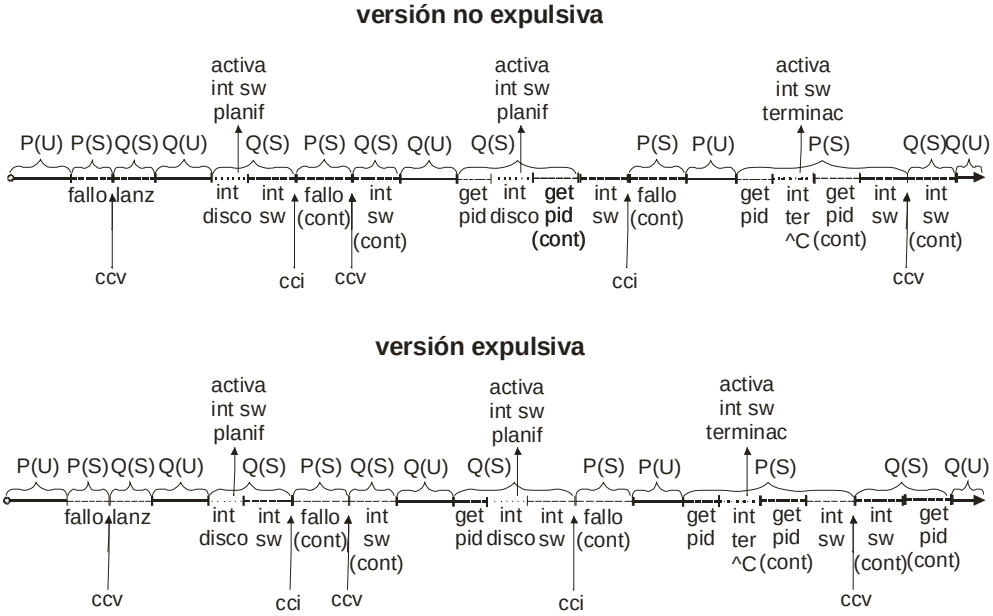


Figura 24 *Secuencia de ejecución que corresponde a la traza del segundo ejemplo.*

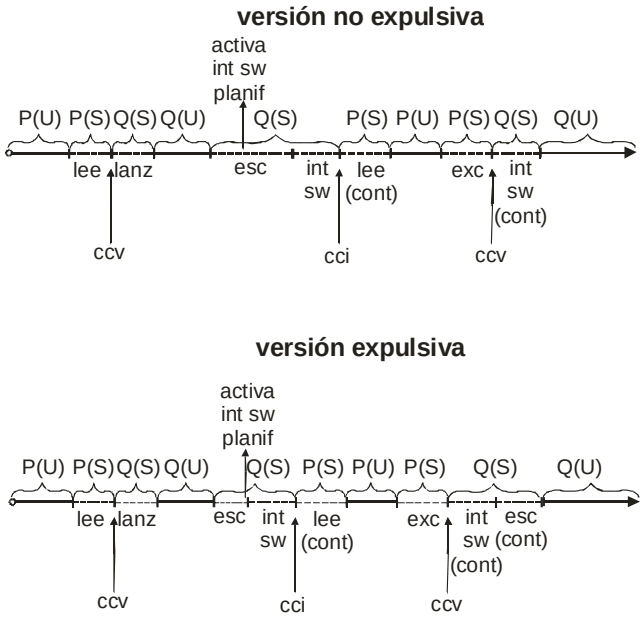


Figura 25 *Secuencia de ejecución que corresponde a la traza del tercer ejemplo.*

Tercer ejemplo

La figura 25 representa la secuencia de ejecución de la siguiente traza:

- P tiene mayor prioridad que Q. P está ejecutando y Q es un nuevo proceso.
- P lee de una tubería vacía.
- Q escribe en la tubería.
- Después de leer, P realiza una división por cero.

Cuarto ejemplo

La figura 26 representa la secuencia de ejecución de la siguiente traza:

- P tiene mayor prioridad que Q. P está ejecutando y Q es un nuevo proceso.

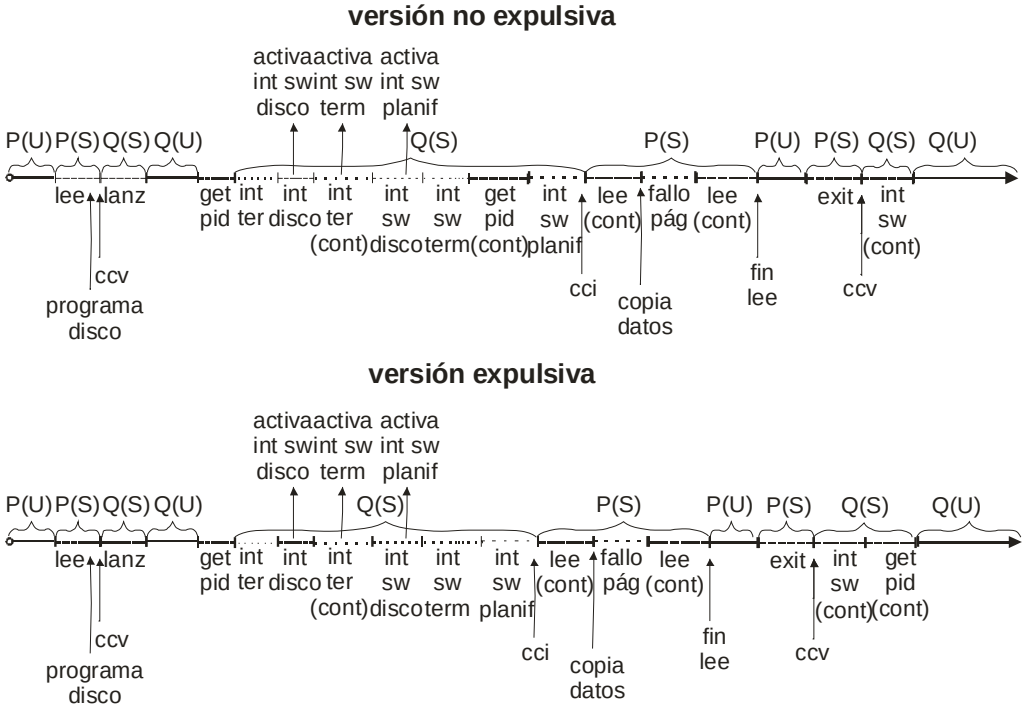


Figura 26 *Secuencia de ejecución que corresponde a la traza del cuarto ejemplo.*

- P lee un bloque de un fichero especificando en la operación de lectura una variable global sin valor inicial a la que nunca se ha accedido y ocupa sólo una página. Además, en el sistema hay marcos de página libres. Por tanto, cuando se acceda a esa página se producirá un fallo de página que no requerirá ninguna operación sobre el disco.
- Q ejecuta todo el tiempo cálculos en modo usuario, excepto una llamada `getpid`.
- Antes de llegar la interrupción del disco, llega una interrupción del terminal justo cuando Q está ejecutado la llamada al sistema, que corresponde a la pulsación de una tecla cuya lectura no ha solicitado ningún proceso.
- Tanto la interrupción del terminal como la del disco usan interrupciones software de sistema para diferir las operaciones menos urgentes, como, por ejemplo, el desbloqueo de un proceso.
- Después de leer del terminal, el proceso termina voluntariamente.

Quinto ejemplo

La figura 27 representa la secuencia de ejecución de una traza con 4 procesos en un sistema con dos procesadores:

- P tiene mayor prioridad que Q, y éste mayor que R, que es más prioritario que S. P está ejecutando en el primer procesador y Q en el segundo, mientras que R y S son nuevos procesos.
- P lee de una tubería vacía.
- Q escribe en la tubería.
- R y S están ejecutando siempre en modo usuario.
- Después de hacer sus operaciones, P y Q terminan voluntariamente

Nótese que la traza es igual para un núcleo no expulsivo que para uno que lo sea, puesto que en ningún momento se requiere expulsar a un proceso estando éste en modo sistema.

5 Sincronización

Uno de los aspectos más importantes en el desarrollo de una aplicación concurrente es asegurar una correcta sincronización entre las distintas partes de la misma. En esta sección se asume que el lector ya tiene conocimientos previos sobre el problema de la sincronización entre procesos, así como de los diversos mecanismos desarrollados para solventarlo. El objetivo de esta sección es estudiar en profundidad el problema de la sincronización interna del propio sistema

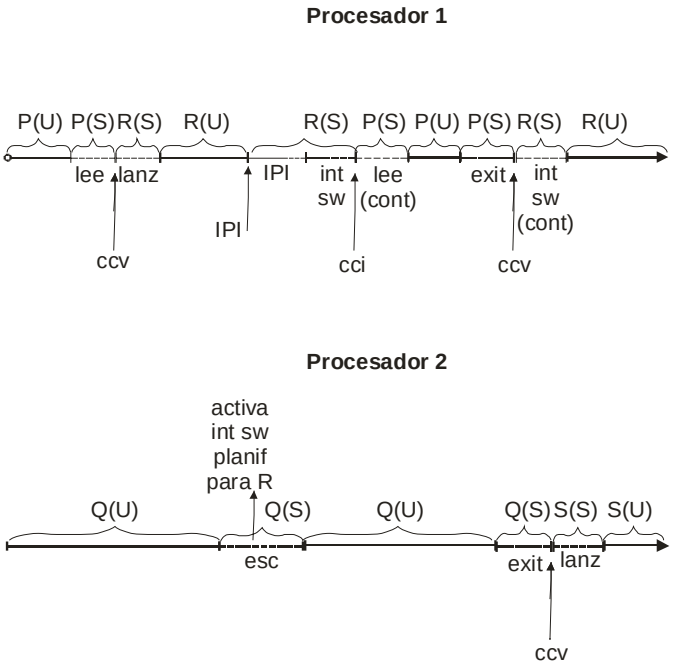


Figura 27 *Secuencia de ejecución que corresponde a la traza del quinto ejemplo.*

operativo. No obstante, al final de la misma, se exponen algunas consideraciones sobre cómo el sistema operativo exporta sus mecanismos internos para crear servicios de sincronización para los procesos de usuario.

5.1 Sincronización dentro del sistema operativo

El sistema operativo es un programa con un alto grado de concurrencia y de asincronía. En cada momento se pueden producir múltiples eventos de forma asíncrona, lo que da lugar a problemas de sincronización muy complejos. Este hecho ha causado que los sistemas operativos sean tradicionalmente un módulo software con una tasa de errores apreciable. Hay que resaltar que la mayoría de los problemas clásicos de sincronización que aparecen en la literatura sobre la programación concurrente provienen del ámbito de los sistemas operativos.

En un sistema operativo se presentan varios tipos de problemas de concurrencia internos:

- Los debidos a la activación y tratamiento de eventos asíncronos.
- Los causados por la ejecución intercalada de procesos en un procesador.
- En el caso de un multiprocesador, los específicos de este tipo de sistemas.
- Si se trata de un sistema con un perfil de tiempo real, los característicos del mismo.

El primer tipo de problemas sucede cuando se produce, y procesa, una interrupción, mientras se está ejecutando código del sistema operativo vinculado con el tratamiento de una llamada, una excepción u otra interrupción. Se ejecutará la rutina de tratamiento correspondiente, que puede entrar en conflicto con la labor que estaba realizando el flujo de ejecución interrumpido. Téngase en cuenta que también puede estar involucrada en este tipo de conflictos una interrupción software (de proceso o de sistema), ya sea como el evento que interrumpe (por ejemplo, una llamada al sistema interrumpida por una interrupción software de proceso expulsiva o de sistema) o como el interrumpido (por ejemplo, el tratamiento de una interrupción software detenido por una interrupción de un dispositivo).

Consideremos, por ejemplo, una llamada o rutina de interrupción que manipula la lista de listos, que es interrumpida por una interrupción que también modifica esta estructura de datos. Puede ocurrir una condición de carrera que deje corrupta la lista.

Supóngase una hipotética función que inserta un BCP al final de la cola de listos, siendo ésta una lista con enlace simple y con una cabecera que incluye un puntero al primer y al último elemento:

```
insertar_ultimo(lista, BCP){
    lista->ultimo->siguiente = BCP;
    lista->ultimo = BCP;
}
```

Considere que dentro de una llamada al sistema se desbloquea un proceso y se usa la rutina de inserción anterior para incorporarlo en la cola de listos. Podría ser, por ejemplo, una llamada de escritura en una tubería, que estaba previamente vacía habiendo uno o más procesos lectores esperando por datos.

```
escritura_tuberia() {
    .....
    Si (vacia Y procesos esperando) {
          Seleccionar un proceso P;
          insertar_ultimo(lista_listos, P);
    }
    .....
}
```

Asimismo, una rutina de interrupción, como, por ejemplo, la del terminal, puede desbloquear un proceso que estuviera esperando ese evento e insertarlo en la cola de listos usando esa misma función:

```
int_terminal(){
    .....
    Si (procesos esperando) {
          Seleccionar un proceso Q;
```

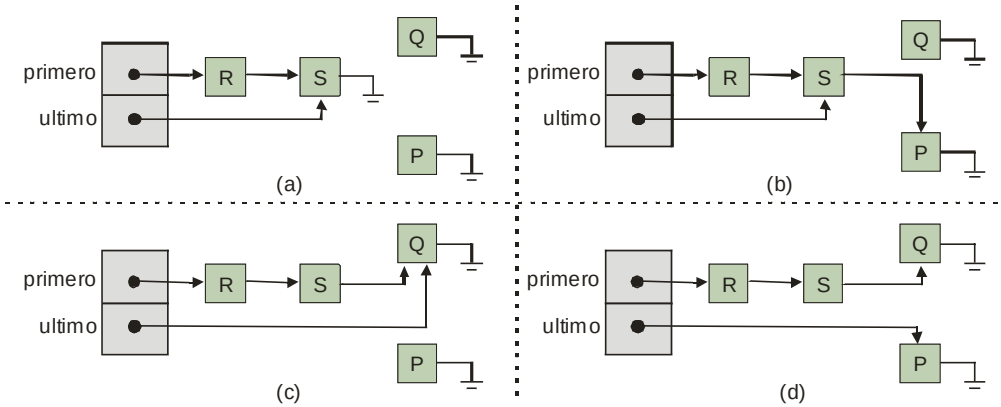


Figura 28 *Problemas de sincronización entre una llamada y una interrupción.*

```
            insertar_ultimo(lista_listos, Q);
        }
        .....
    }
```

La figura 28 muestra qué problema puede surgir si se activa una interrupción en mitad de una llamada. Consideremos que inicialmente hay dos procesos listos (figura 28a). Supóngase que, mientras se está ejecutando la rutina *insertar_ultimo*, dentro de la escritura en la tubería, para insertar un proceso *P* al final de la cola de listos, habiéndose ejecutado sólo la primera sentencia (figura 28b), llega una interrupción del terminal que desbloquea a un proceso *Q* incorporándolo a la lista de listos (figura 28c). Cuando retorna la rutina de interrupción, se completa la inserción realizada dentro de la escritura en la tubería, pero la lista de listos queda corrupta, como se puede apreciar en la figura 28d:

- El puntero al último elemento que hay en la cabecera de la lista quedará apuntando a *P*.
- Sin embargo, ningún elemento de la lista hará referencia a *P* como su sucesor.

El segundo tipo de problemas aparece cuando se produce un cambio de contexto a otro proceso mientras se está realizando una llamada al sistema (o, en general, el tratamiento de un evento síncrono). Este proceso a su vez puede ejecutar una llamada al sistema que entre en conflicto con la llamada previamente interrumpida. Esta situación puede causar una condición de carrera. Obsérvese que se puede considerar que se trata de un problema convencional de sincronización de procesos, aunque con la peculiaridad de que los procesos ejecutan en modo sistema. Por ello, como se verá más adelante, para afrontar este tipo de conflictos se usan mecanismos convencionales, como son los semáforos.

A continuación, se plantean dos ejemplos. En el primero, el cambio de contexto conflictivo es de tipo involuntario, mientras que en el segundo es voluntario.

El primer ejemplo plantea cómo la ejecución concurrente de dos llamadas que intentan crear un proceso en un sistema que usa una tabla de procesos implementada como un vector podría acabar de manera que ambas obtengan el mismo BCP libre. Supóngase que el siguiente fragmento corresponde al código de la llamada al sistema que crea un proceso:

```
crear_proceso(...) {
    ...........
    pos=BuscarBCPLibre();
    tabla_procesos[pos].ocupada = true;
    ...........
}
```

Podría ocurrir un cambio de contexto involuntario (por ejemplo, por fin de rodaja) justo después de que la función *BuscarBCPLibre* se haya ejecutado pero antes de poner la entrada correspondiente de la tabla de procesos como ocupada. El proceso activado por el cambio de contexto involuntario podría invocar también la llamada *crear_proceso*, con lo que habría dos llamadas ejecutándose concurrentemente. Esta segunda invocación de *crear_proceso* causaría una condición de carrera, ya que seleccionaría la misma posición de la tabla de procesos que la llamada interrumpida.

En cuanto a un ejemplo donde haya problemas de sincronización vinculados a un cambio de contexto voluntario, considere una llamada de escritura en un fichero que se queda bloqueada durante su ejecución (en un sistema UNIX podría quedarse bloqueada, por ejemplo, mientras lee un bloque de punteros indirectos):

```
write(...) {
    ...........
    Por cada bloque afectado
          Escribir el bloque (operación que puede causar bloqueos)
    ...........
}
```

Si se permite que mientras tanto proceda una llamada de lectura sobre el mismo fichero que afecte a una zona solapada, ésta puede leer parte de la información ya actualizada por la escritura y parte que todavía tiene el contenido previo. También habría problemas si estando bloqueada una operación de escritura sobre un fichero, se produce una segunda operación de escritura sobre el mismo. Sería necesario algún tipo de sincronización si se pretende que estas llamadas tengan un comportamiento atómico.

Hay que resaltar que este tipo de problemas de sincronización debidos a la ejecución concurrente de procesos no sólo se da en el caso de que los procesos estén haciendo llamadas al sistema, sino también en la ejecución del tratamiento de excepciones y si se trata de procesos de núcleo.

Con respecto a los problemas específicos presentes en los multiprocesadores, en este tipo de sistemas, como ya se ha comentado en secciones previas, existe paralelismo real, con lo que aumenta apreciablemente el número de situaciones conflictivas, no siendo válidas, además, algunas de las estrategias utilizadas en sistemas monoprocesador, como inhabilitar las interrupciones para eliminar condiciones de carrera.

Por último, en cuanto a los problemas particulares de los sistemas de tiempo real, en este tipo de sistemas, de cara a garantizar el cumplimiento de los plazos requeridos por las distintas aplicaciones, es necesario controlar la inversión de prioridades no acotada asociada con el uso de los mecanismos de sincronización.

De los escenarios planteados en esta sección, se deduce la necesidad de crear secciones críticas dentro de la ejecución del sistema operativo. Evidentemente, éste no es un problema nuevo. Es el mismo que aparece en cualquier programa concurrente, que se resuelve con mecanismos de sincronización como semáforos o *mutex*. Sin embargo, en este caso se presentan dos dificultades adicionales:

- Estos mecanismos de sincronización se basan en que sus operaciones se ejecutan de forma atómica. ¿Cómo se puede lograr esto dentro del sistema operativo?
- Desde una rutina de interrupción, como se ha explicado reiteradamente, no se puede usar una primitiva que cause un bloqueo.

A continuación, se estudia el problema en un sistema monoprocesador distinguiendo si se trata de un sistema con un núcleo expulsivo o no expulsivo. Después, se analizará qué dificultades adicionales se presentan cuando se usa un sistema multiprocesador y, por último, en un sistema de tiempo real.

Sincronización en un núcleo no expulsivo

Como se comentó en la sección anterior, el modo de operación de los sistemas con un núcleo no expulsivo facilita considerablemente el tratamiento de los problemas de sincronización al limitar el grado de concurrencia en el sistema. A continuación, se analizan los dos escenarios problemáticos planteados para un monoprocesador, es decir, el tratamiento de un evento asíncrono y la ejecución intercalada de procesos.

En primer lugar, se trata el problema de concurrencia que surge cuando una rutina del sistema operativo es interrumpida por una rutina de interrupción. En un sistema operativo que usa un esquema de interrupciones con niveles de prioridad, la solución habitual es elevar el nivel de interrupción del procesador durante el fragmento correspondiente para evitar la activación de la rutina de interrupción conflictiva.

Es importante resaltar que se debería elevar el nivel interrupción del procesador justo lo requerido y minimizar el fragmento durante el cual se ha elevado explícitamente dicho nivel para intentar, de esta forma, minimizar la latencia de las interrupciones y, de manera indirecta, la de activación de los procesos. Así, por ejemplo, en un fragmento de código del sistema operativo donde se manipula el *buffer* de un terminal, bastaría con inhibir la interrupción del terminal justo en el fragmento requerido, pudiendo seguir habilitadas otras interrupciones, como, por ejemplo, la del reloj.

```
leer_terminal(...) {
    ...........
    nivel_anterior = fijar_nivel_int(NIVEL_TERMINAL);
    extrae datos del buffer del terminal;
    fijar_nivel_int(nivel_anterior);
    ...........
}
```

En caso de que el conflicto lo genere una interrupción software de sistema habrá que establecer el nivel que evite las interrupciones de este tipo. Si hay varios tipos de interrupciones software de sistema con distintas prioridades, se elevará el nivel al valor que corresponda para eliminar el conflicto. Nótese que sólo las interrupciones software de sistema pueden causar este tipo de problemas. Las interrupciones software de proceso, al no tener carácter expulsivo en este tipo de núcleos, no provocarán esta clase de conflictos.

Si se usa un esquema sin prioridades, habrá que inhabilitar todas las interrupciones en el fragmento afectado, siempre minimizando la longitud del fragmento involucrado:

```
leer_terminal(...) {
    ...........
    estado_anterior = inhibir_int();
    extrae datos del buffer del terminal;
    restaurar_estado_int(estado_anterior);
    ...........
}
```

Si el conflicto está generado por una interrupción software de sistema, se utilizará la función disponible en ese sistema sin niveles de prioridad para inhabilitar sólo este tipo de interrupciones (en Linux, `local_bh_disable`). También en este caso puede haber varios tipos de interrupciones software de sistema con distintas prioridades. Para eliminar un conflicto, se inhibirán las interrupciones software de sistema del nivel que corresponda.

En el ejemplo planteado en la sección anterior, donde la estructura conflictiva era la cola de procesos listos, aunque se use un sistema con niveles de prioridad de interrupciones, habrá que elevar el nivel al máximo para prohibir todas las interrupciones, puesto que se trata de una estructura de datos que manipulan todas ellas.

```
insertar_ultimo(lista, BCP){
    nivel_anterior = fijar_nivel_int(NIVEL_MAXIMO);
    lista->ultimo->siguiente = BCP;
    lista->ultimo = BCP;
    fijar_nivel_int(nivel_anterior);
}
```

Hay que resaltar que, dado que la inhibición de las interrupciones pertinentes se usa para asegurarse de que el fragmento correspondiente a la sección crítica se ejecuta de forma atómica con respecto a las mismas, parece razonable que no debería haber un cambio de contexto voluntario (involuntario no es posible por la propia idiosincrasia de este tipo de núcleos) durante dicha sección crítica. Aunque a priori parece relativamente sencillo garantizar este requisito cuando se está desarrollando el código del sistema operativo, no lo es tanto porque hay numerosas operaciones internas que pueden esconder un bloqueo (copiar datos desde/hacia un *buffer* de usuario del proceso, reservar un *buffer* de memoria del sistema,...).

En cuanto a los problemas de sincronización debidos a los cambios de contexto involuntarios, con este tipo de núcleo, como se explicó en la sección previa, no existe este

problema ya que no se permite la ejecución concurrente de llamadas: una llamada al sistema continúa su ejecución hasta que termina o causa el bloqueo del proceso. Como ya se expuso en dicha sección, pueden activarse rutinas de tratamiento de interrupción, pero no causarán un cambio de contexto. Por tanto, no existirá este problema de sincronización en este tipo de núcleos no expulsivos. En el ejemplo planteado, la rutina *crear_proceso* podrá marcar como ocupada la entrada encontrada sin ninguna interferencia.

Con respecto a los problemas de sincronización debidos a los cambios de contexto voluntarios, el programador debe ser consciente de que si un proceso se bloquea, cuando continúe su ejecución posteriormente, la situación en el sistema puede haber cambiado significativamente. Una condición que se cumplía antes del bloqueo puede haberse dejado de satisfacer cuando se reanuda la ejecución, debido a que ha podido cambiar por la ejecución de otros procesos en ese intervalo de tiempo. La llamada al sistema debería comprobarla de nuevo y tomar las acciones correctivas oportunas (por ejemplo, volver a bloquearse esperando que el estado sea el adecuado).

Asimismo, si una llamada bloqueante necesita asegurarse de que otras llamadas no utilizan un determinado recurso mientras el proceso está bloqueado en esa llamada, deberá controlarlo de alguna forma. La solución habitual es implementar internamente un mecanismo de tipo semáforo. Obsérvese que, dadas las características del modelo no expulsivo, la implementación de mecanismos de sincronización, tales como los semáforos, es directa. Así, las llamadas al sistema que implementen las operaciones del semáforo, no necesitarán ninguna técnica para asegurar su comportamiento atómico con respecto a las llamadas al sistema, puesto que éstas son atómicas entre sí mismas.

En el ejemplo planteado en la sección anterior, donde había que asegurar que mientras se ejecuta una llamada de escritura de un fichero no haya lecturas o escrituras simultáneas sobre el mismo, se puede incluir un semáforo en el *inodo* del fichero para lograr este comportamiento, como se muestra en el siguiente fragmento:

```
write(...) {
    ...........
    bajar(inodo.sem)
    ...........
    Por cada bloque afectado
          Escribir el bloque (operación que puede causar bloqueos)
    ...........
    subir(inodo.sem);
    ...........
}
```

Para terminar la sección, y como resumen de los problemas de sincronización presentes en este tipo de núcleos y de las técnicas usadas para solventarlos, en la tabla 3 se muestran todos los casos posibles (se han obviado las interrupciones software de proceso y los procesos de

Rutina a estudiar	**Rutina conflictiva**				
	Llamada	**Excep**	**I. SW sis**	**I. dis. mn**	**I. dis. mx**
Llamada	Semáf. si SC con blq	Semáf. si SC con blq	Inhibir int SW sis	Inhibir int. mín	Inhibir int. máx
Excepción	Semáf. si SC con blq	Semáf. si SC con blq	Inhibir int SW sis	Inhibir int. mín	Inhibir int. máx
Int. SW sistema	—	—	—	Inhibir int. mín	Inhibir int. máx
Int. dispo. mínima	—	—	—	—	Inhibir int. máx
Int. dispo. máxima	—	—	—	—	—

Tabla 3 *Posibles conflictos de sincronización y su solución en un núcleo no expulsivo.*

núcleo para no complicar excesivamente la tabla y puesto que, en este tipo de núcleos, presentan el mismo tipo de conflictos que las llamadas al sistema y las excepciones). Cada fila representa una determinada actividad del sistema operativo que estamos analizando desde el punto de vista de qué tipo de sincronización requiere, mientras que cada columna indica qué tipo de actividad entra en conflicto con la primera al existir una sección crítica entre las mismas, apareciendo en la celda correspondiente qué técnica se usa en la primera para resolver el problema, en caso de que pueda existir.

En este tipo de núcleos no expulsivos, las llamadas al sistema y las excepciones no pueden ejecutar concurrentemente, por lo que no presentan problemas de sincronización excepto cuando se requiera crear una sección crítica que incluya un bloqueo, en cuyo caso se usa un semáforo para solventarlo, como aparece en la tabla.

Nótese que si hubiera más de dos tipos de actividades involucradas en un problema de sincronización, en cada una de ellas se usaría la técnica requerida por la actividad conflictiva más exigente. Así, por ejemplo, si existe una sección crítica entre una llamada, una interrupción software de sistema y una interrupción de un dispositivo, tanto en la llamada como en la interrupción software de sistema se usaría la técnica adecuada para protegerse de la interrupción del dispositivo (en el caso que nos ocupa, tratándose de un monoprocesador con un núcleo no expulsivo, se inhibirían las interrupciones del dispositivo), mientras que en la interrupción del dispositivo se debería utilizar el mecanismo requerido para protegerse de la interrupción software de sistema (en este caso, no se requiere ninguno).

Sincronización en un núcleo expulsivo

Como se analizó anteriormente, este tipo de sistema presenta más dificultades a la hora de lograr una correcta sincronización dado que posee una mayor concurrencia.

En cualquier caso, la solución frente al problema que surge cuando una rutina del sistema operativo es interrumpida por una rutina de interrupción, es la misma que para un núcleo no expulsivo: elevar el nivel de interrupción del procesador (o inhibir todas las interrupciones en un sistema sin niveles) durante el fragmento correspondiente para evitar la activación de la rutina de interrupción conflictiva, minimizando la duración de esta inhibición para reducir las latencias de interrupción y de activación. Dado que la interrupción software de planificación es expulsiva, a diferencia de lo que ocurría en un núcleo no expulsivo, las llamadas al sistema y las excepciones pueden necesitar inhibirla durante secciones críticas afectadas por la misma.

En este tipo de núcleo, además de evitar los cambios de contexto voluntarios mientras se mantienen inhibidas interrupciones, hay que eliminar también la posibilidad de que se produzcan cambios involuntarios, lo cual es directo puesto que al inhibir las interrupciones de un dispositivo se inhabilitan automáticamente todas la de un nivel inferior, entre ellas la interrupción software de planificación.

El problema principal de este tipo de núcleo reside en la sincronización entre llamadas concurrentes que presenten una sección crítica entre ellas (ocurriría lo mismo con las excepciones o con los procesos de núcleo). Dado que la ejecución de llamadas al sistema por parte de varios procesos puede verse entremezclada de forma impredecible, hay que asegurarse de que no se produzcan problemas de condiciones de carrera estableciendo las secciones críticas que se requieran. Para ello, se deben evitar los cambios de contexto involuntarios durante el fragmento conflictivo, inhibiendo las interrupciones software de planificación durante el mismo. Aplicándolo al ejemplo planteado previamente y suponiendo un esquema con niveles de prioridad de interrupciones, quedaría un esquema como el siguiente:

```
crear_proceso(...) {
    ...........
    nivel_anterior = fijar_nivel_int(NIVEL_INT_SW_PLANIFICACION);
    pos=BuscarBCPLibre();
    tabla_procesos[pos].libre = false;
    fijar_nivel_int(nivel_anterior);
    ...........
}
```

En caso de un sistema sin niveles, habría que inhabilitar de forma específica este tipo de interrupción software.

```
crear_proceso(...) {
    ...........
    inhibir_int_SW_planificacion();
    pos=BuscarBCPLibre();
    tabla_procesos[pos].libre = false;
    habilitar_int_SW_planificacion();
    ...........
}
```

La solución planteada resuelve el problema, pero puede ser inadecuada si la sección crítica es larga (en el ejemplo, es posible que *BuscarBCPLibre* consuma un tiempo apreciable). En ese caso, se empeoraría la latencia de activación de los procesos, como ya se analizó anteriormente. Si un proceso urgente se desbloquea mientras otro poco prioritario está en la sección crítica, no comenzará a ejecutar hasta que este segundo proceso concluya la sección crítica. Nótese que, si hay secciones críticas largas, el núcleo tiende a ser no expulsivo. Asimismo, hay que resaltar que esta estrategia afecta a todos los procesos, con independencia de si el proceso ejecuta código que entre en conflicto con el que mantiene la sección crítica. Además, esta solución no sería válida si la sección crítica puede incluir el bloqueo del proceso, como ocurre en los problemas de sincronización vinculados a un cambio de contexto voluntario, analizados en secciones precedentes mediante el ejemplo de la escritura en un fichero. Como ya se explicó previamente, es conveniente resaltar nuevamente que a veces es difícil para el programador del código del sistema operativo cerciorarse de si puede o no haber bloqueos en un determinado fragmento de código, puesto que, aunque éstos no aparezcan directamente en el código involucrado, pueden estar escondidos en alguna función invocada desde ese código.

La solución habitual en estos dos casos (sección crítica larga y/o con bloqueos) es implementar un semáforo, o algún mecanismo de sincronización equivalente, tal como se analizó en el caso de un núcleo no expulsivo. Sin embargo, en este caso, lograr la atomicidad en las operaciones del semáforo no es algo directo, debido al carácter expulsivo del núcleo. Para conseguir la atomicidad requerida, hay que recurrir precisamente al mecanismo anterior: elevar el nivel de interrupción de manera que se inhiban las interrupciones software de planificación. Esta prohibición sólo afectaría a las operaciones sobre el semáforo, que son muy breves, pudiendo ejecutarse la sección crítica con todas las interrupciones habilitadas. Con esta técnica, el ejemplo anterior quedaría de la siguiente forma:

```
crear_proceso(...) {
    ...........
    bajar(semaforo_tabla_procesos);
    /* Sección crítica ejecuta con inter. habilitadas */
    pos=BuscarBCPLibre();
    tabla_procesos[pos].libre = false;
    subir(semaforo_tabla_procesos);
    ...........
}
```

Usando esta estrategia, sólo se ejecuta con las interrupciones software de planificación inhibidas durante muy poco tiempo (el correspondiente a las operaciones del semáforo). Asimismo, la sección crítica propiamente dicha sólo involucra a los procesos afectados por la misma, pudiendo, además, incluir el bloqueo del proceso, como en el ejemplo de la escritura en un fichero.

De todas formas, hay que tener en cuenta que si la sección crítica es muy corta y sin bloqueos, será más adecuado utilizar directamente la estrategia de inhabilitar las interrupciones software de planificación, puesto que la solución basada en semáforos incluye una cierta sobrecarga debida a las operaciones vinculadas con los mismos. Además, conlleva cambios de contexto por los bloqueos que se producen al competir por el semáforo, que no se generan en la solución que no lo utiliza.

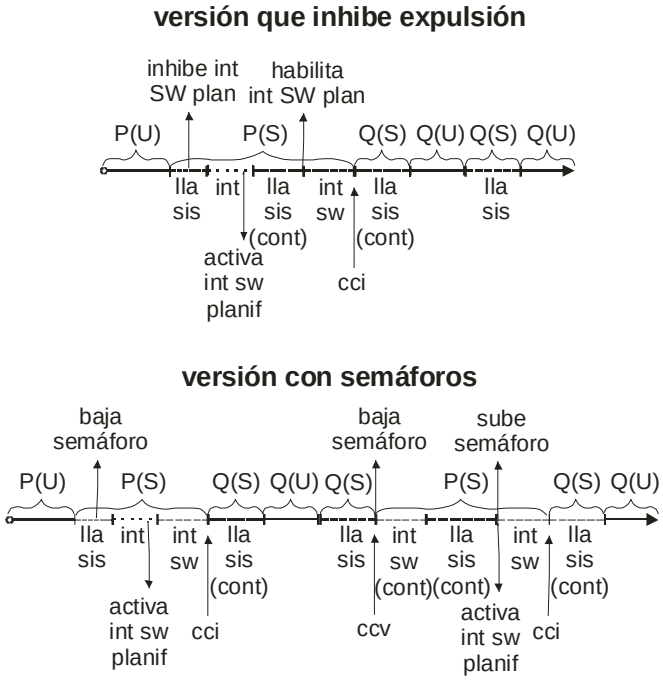


Figura 29 *Eficiencia de la solución basada en inhibir las expulsiones vs. el uso de semáforos.*

En la figura 29 se compara la eficiencia de ambas estrategias con un ejemplo donde un proceso P está en la sección crítica de una llamada cuando llega una interrupción que desbloquea a un proceso Q más prioritario que, cuando reanude su ejecución y vuelva a modo usuario, invocará esa misma llamada conflictiva. Con la solución basada en inhibir las expulsiones, ese cambio de contexto involuntario a Q se retrasa hasta que P completa la sección crítica de la llamada rehabilitando las expulsiones. En el esquema que usa semáforos el cambio de contexto a Q es inmediato, pero este proceso se queda bloqueado al intentar entrar en la sección crítica volviendo a ejecutar P. Cuando P completa la sección crítica abre el semáforo, desbloqueando a Q y realizando un cambio de contexto involuntario al mismo. Como se puede apreciar, el uso del semáforo ha significado dos cambios de contexto adicionales, además de la sobrecarga de las operaciones sobre el mismo.

El diseñador del sistema operativo tiene que determinar qué semáforos se deben usar para proteger las distintas estructuras de datos del sistema operativo. Para ello, debe establecer un compromiso con respecto a la granularidad del semáforo, es decir, en cuanto a la cantidad de información que protege un determinado semáforo: cuanta más información sea protegida por un semáforo, menos concurrencia habrá en el sistema, pero menos sobrecarga causada por la gestión de los semáforos. A continuación, se plantea un ejemplo hipotético para ilustrar este aspecto.

Supóngase un sistema UNIX en el que se requiere controlar el acceso mediante semáforos a dos estructuras de datos en memoria vinculadas con el sistema de ficheros: por ejemplo, la tabla intermedia de ficheros, donde se almacenan los punteros de posición de todos los ficheros abiertos en el sistema, y la tabla de *inodos*, que guarda en memoria los *inodos* de los ficheros que están siendo utilizados en el sistema. Considere en este punto las dos alternativas posibles.

Si se establecen sendos semáforos para el control de acceso a cada estructura, podrán ejecutar concurrentemente dos llamadas si cada una de ellas sólo involucra a una de las estructuras. Sin embargo, si una llamada determinada requiere el uso de ambas estructuras, como, por ejemplo, la apertura de un fichero, deberá cerrar y abrir ambos semáforos, con la consiguiente sobrecarga.

En caso de que se defina un único semáforo para el control de acceso a ambas estructuras, se limita la concurrencia, puesto que dos llamadas, tal que cada una de ellas sólo usa una de las estructuras de datos, no podrán ejecutarse concurrentemente. Sin embargo, una llamada que usa ambos recursos tendrá menos sobrecarga, ya que será suficiente con cerrar y abrir un único semáforo.

Rutina a estudiar	Rutina conflictiva				
	Llamada	**Excep**	**I. SW sis**	**I. dis. mn**	**I. dis. mx**
Llamada	Semáf. o Inh. expuls.	Semáf. o Inh. expuls.	Inhibir int SW sis	Inhibir int. mín	Inhibir int. máx
Excepción	Semáf. o Inh. expuls.	Semáf. o Inh. expuls.	Inhibir int SW sis	Inhibir int. mín	Inhibir int. máx
Int. SW sistema	—	—	—	Inhibir int. mín	Inhibir int. máx
Int. dispo. mínima	—	—	—	—	Inhibir int. máx
Int. dispo. máxima	—	—	—	—	—

Tabla 4 *Posibles conflictos de sincronización y su solución en un núcleo expulsivo.*

Del análisis previo se concluye que sería conveniente usar dos semáforos si hay un número significativo de llamadas que usan de forma independiente cada recurso, mientras que sería adecuado usar un único semáforo si la mayoría de las llamadas usa ambos recursos conjuntamente. Nótese que llevándolo a un extremo, se puede establecer un semáforo para todo el sistema operativo, lo que haría que el sistema se comportara como un núcleo no expulsivo.

Se debe incidir en que los semáforos no son aplicables a los problemas de sincronización generados por el tratamiento de un evento asíncrono puesto que una rutina de interrupción no se puede bloquear, como se ha comentado reiteradamente a lo largo de esta presentación.

Como última consideración, y para comparar con la sincronización en sistemas no expulsivos, se debe resaltar que en los sistemas expulsivos el análisis de los problemas de sincronización que debe realizar el diseñador del sistema operativo se complica de manera apreciable. A los problemas ya existentes en los sistemas no expulsivos, hay que añadir los que conlleva la ejecución concurrente de llamadas, que, en principio, requeriría analizar las posibles interferencias entre todas las llamadas al sistema.

Para terminar la sección, se presenta la tabla 4, que es una adaptación de la presentada al final de la sección dedicada a la sincronización en núcleos no expulsivos, y sirve como resumen de los problemas de sincronización presentes en este tipo de núcleos.

Sincronización en multiprocesadores

Los problemas de sincronización dentro del sistema operativo se complican considerablemente en un sistema multiprocesador. Hay que tener en cuenta que en este tipo de sistemas la concurrencia, que en un monoprocesador implica la ejecución alternativa de distintos procesos, se corresponde con la ejecución en paralelo de los procesos. Evidentemente, esto dificulta la correcta sincronización y hace que algunas estrategias utilizadas en uniprocesadores no sean válidas para sistemas multiprocesador.

En un multiprocesador, se producen situaciones conflictivas que no pueden aparecer en un sistema monoprocesador. Así, mientras se está ejecutando una rutina de interrupción en un procesador, se pueden estar ejecutando paralelamente en otros procesadores llamadas al sistema, tratamientos de excepciones y rutinas de interrupción de la misma o menor prioridad, en caso de tratarse de un sistema con niveles de interrupción.

Por otro lado, las técnicas de sincronización basadas en elevar el nivel de interrupción para inhibir el tratamiento de determinadas interrupciones, o bien inhabilitar todas las interrupciones en un sistema sin niveles, no son directamente aplicables, ya que, aunque se impida que se ejecute dicha rutina de interrupción en ese procesador, esta rutina podrá ejecutarse en cualquier otro procesador cuyo estado actual lo permita. Asimismo, la estrategia de impedir los cambios de contexto involuntarios durante una llamada al sistema no evita que se ejecuten concurrentemente llamadas al sistema en un multiprocesador.

Las principales técnicas de sincronización utilizadas en multiprocesadores se basan en el mecanismo del *spin-lock*. Este mecanismo se basa en el uso de una variable como un cerrojo. La

variable se consulta en un bucle con espera activa hasta que tenga un valor que indique que el cerrojo está abierto. Esta consulta se realiza mediante una instrucción atómica de consulta y actualización de memoria, que suele estar disponible en el juego de instrucciones de cualquier procesador. Esta instrucción asegura que la operación de consulta y modificación de una posición de memoria por parte de un procesador se realiza sin interferencia de ningún otro procesador. Un ejemplo de este tipo sería la instrucción *TestAndSet*, que, de manera atómica, devuelve el valor almacenado en una variable escribiendo, a continuación, un 1 en la misma. Usando esta instrucción se puede construir un *spin-lock* siguiendo un esquema similar al siguiente:

```
/* cierra el cerrojo */
spin_lock(int *cerrojo) {
   /* espera hasta que el cerrojo valga 0 y escribe un 1 */
   while (TestAndSet(*cerrojo) == 1);
}
/* abre el cerrojo */
spin_unlock(int *cerrojo) {
    *cerrojo = 0;
}
```

Basándose en este esquema, se puede construir una sección crítica de la siguiente forma:

```
int mi_cerrojo = 0;
spin_lock(&mi_cerrojo);
Sección Crítica
spin_unlock(&mi_cerrojo);
```

En este esquema básico, con diversas variedades, se basa la sincronización en multiprocesadores. Nótese que, sin embargo, este mecanismo de sincronización carece de sentido en un uniprocesador: un proceso haciendo espera activa sobre un cerrojo consumirá su turno de ejecución sin poder obtenerlo, puesto que para ello debe ejecutar el proceso que lo posee. En muchos sistemas operativos, tanto en su versión para sistemas monoprocesador como en la destinada a multiprocesadores se mantienen los *spin-locks*. Sin embargo, en la versión para sistemas monoprocesador, las primitivas de gestión del cerrojo se definen como operaciones nulas. Un programador de un sistema operativo siempre debería desarrollar el código pensando en la configuración más compleja posible (multiprocesador con un núcleo expulsivo) y las herramientas de generación del sistema operativo deberían eliminar automáticamente, sin ninguna sobrecarga, todas las operaciones que son innecesarias para la configuración real del sistema.

Dado que el bucle de espera activa sobre un cerrojo conlleva un malgasto de ciclos de procesador, no debería realizarse un cambio de contexto, ni voluntario ni involuntario, en el procesador que ejecuta el proceso que mantiene posesión del cerrojo, pues alargaría la espera activa de los procesadores en los que se están ejecutando procesos que intentan tomar posesión del mismo. En cambio, sí se deben permitir cambios de contexto involuntarios en el procesador que está realizando la espera activa.

Además del modelo básico descrito, existen otras variedades, que intentan mejorar la eficiencia de este mecanismo o ampliar su funcionalidad. Aunque queda fuera del objetivo de esta exposición analizar en detalle este mecanismo, a continuación, se comentan brevemente algunas modalidades de cerrojos:

- *spin-locks* con reintentos. Para mitigar la congestión que se produce en el sistema de memoria cuando hay múltiples procesadores esperando, con este tipo de *spin-locks* si el cerrojo no está disponible, el procesador espera un breve plazo de tiempo antes de volver a intentarlo.
- *spin-locks* con cola. Cuando se detecta que el cerrojo no está disponible, el procesador realiza la espera activa sobre una variable específica asociada a ese procesador, de manera que el procesador que libera el cerrojo puede elegir a qué procesador cedérselo.

Permiten disminuir la congestión en el sistema de memoria y controlar el orden con el que los procesos en espera van obteniendo el cerrojo.

- *spin-locks* de lectura/escritura. Permiten que varios procesos puedan entrar de forma simultánea en una sección crítica si sólo van a leer la estructura de datos protegida por el cerrojo, pero sólo dejan entrar a un proceso en el caso de que vaya a modificarla. Proporcionan mayor paralelismo, siempre que se usen en situaciones donde haya lectores, aunque tienen una mayor sobrecarga. Puede haber modalidades en las que se les otorga prioridad a los lectores, mientras que en otras se les da a los escritores.

Una vez presentando el modo de operación del *spin-lock*, que es el mecanismo sobre el que se construyen todos los esquemas de sincronización de los multiprocesadores, se analizan, a continuación, los dos tipos de problemas de sincronización identificados: los vinculados con el tratamiento de eventos asíncronos y los generados por la existencia de llamadas al sistema concurrentes.

Antes de analizar cómo resolver el primer tipo problema de sincronización, hay que recordar que en un sistema multiprocesador la ejecución de una rutina de interrupción puede convivir con la de cualquier otra rutina del sistema operativo. Por tanto, el análisis de los problemas de sincronización se hace más complejo en este tipo de sistema. Asimismo, es conveniente reseñar que en un multiprocesador es necesario incluir mecanismos de sincronización en toda rutina afectada, a diferencia de lo que ocurre en un sistema uniprocesador donde, como se analizó previamente, sólo se incluye en la que puede ser interrumpida.

La estrategia para resolver este tipo de problemas se basa en los *spin-locks*. Las rutinas afectadas usarán un cerrojo de este tipo para asegurarse de que no se ejecutan concurrentemente. Además, la rutina que puede ser interrumpida, ya sea una rutina de interrupción, una llamada al sistema o el tratamiento de una excepción, debe elevar el nivel de interrupción en el procesador (o inhibir todas las interrupciones, en caso de que se trate de un procesador con un esquema sin niveles) donde ejecuta para asegurarse de que no se producen interbloqueos. Para entender este problema hay que plantearse una situación como la siguiente: una rutina en posesión de un cerrojo es interrumpida por una rutina que también lo requiere, quedándose esta última indefinidamente en un bucle de espera que congela la ejecución del procesador (a partir de ese momento, lo único que podría ejecutar son rutinas de interrupción de mayor prioridad).

En un sistema con niveles de prioridad de interrupciones, el nivel de interrupción que debe fijarse mientras se está usando un determinado cerrojo se corresponderá con el de la rutina de tratamiento de interrupción de mayor prioridad que puede utilizar ese cerrojo. Observe que las interrupciones software se integrarían de forma natural en este modelo como niveles adicionales.

A continuación, se muestra una implementación hipotética de esta técnica.

```
int spin_lock_interrupcion(int *cerrojo, int nivel) {
    nivel_anterior = fijar_nivel_int(nivel);
    spin_lock(cerrojo);
    return nivel_anterior;
}
spin_unlock_interrupcion(int *cerrojo, int nivel) {
    spin_unlock(cerrojo);
    fijar_nivel_int(nivel);
}
```

Aplicando este mecanismo al ejemplo de sincronización en el acceso a la cola de procesos listos para ejecutar planteado previamente, que requeriría elevar el nivel de interrupción del procesador al máximo ya que generalmente se accede desde todas las rutinas de interrupción, resultaría un esquema similar al siguiente:

```
insertar_ultimo(lista, BCP){
    int niv_previo;
    niv_previo=spin_lock_interrupcion(&lista->cerrojo,NIVEL_MAXIMO);
    lista->ultimo->siguiente = BCP;
    lista->ultimo = BCP;
```

```
    spin_unlock_interrupcion(&lista->cerrojo, niv_previo);
}
```

Observe que se ha incluido un cerrojo en la cabecera de la lista para controlar el acceso a la misma. Esa es la estrategia habitual a la hora de organizar el esquema de coherencia de un determinado componente del sistema operativo: incluir cerrojos en las estructuras de datos que sea necesario proteger.

En el caso de un sistema que no gestione niveles de prioridad entre las interrupciones, para resolver este tipo de esquemas de sincronización, se prohíben todas las interrupciones mientras se mantiene el cerrojo.

```
int spin_lock_interrupcion(int *cerrojo, int estado) {
    estado_anterior = inhibir_int();
    spin_lock(cerrojo);
    return estado_anterior;
}
spin_unlock_interrupcion(int *cerrojo, int estado) {
    spin_unlock(cerrojo);
    restaurar_estado_int(estado);
}
```

Nótese que en un sistema uniprocesador, dado que las operaciones del *spin-lock* se redefinen como nulas, queda como resultado la misma solución que se explicó para un sistema monoprocesador: inhibir la interrupción que pueda causar un conflicto.

Para terminar el análisis de los problemas de sincronización que conlleva el tratamiento de un evento asíncrono, en este punto se va a considerar un caso especial: la sincronización entre dos o más activaciones de la misma interrupción que se producen en distintos procesadores. Como ya se analizó cuando se estudió cómo trata el sistema operativo las interrupciones, las propias características intrínsecas de las operaciones que conlleva una interrupción requieren que en cada momento sólo esté activo el tratamiento de una misma interrupción. Ese requisito se satisface de manera relativamente simple en un monoprocesador, gracias al cambio de nivel de prioridad de interrupciones del procesador, en un sistema con niveles, o mediante el enmascaramiento, en un sistema que no los tenga. Sin embargo, en un multiprocesador esta solución no es suficiente y, adicionalmente, se requiere el uso de un *spin-lock* para que la nueva activación de la interrupción espere a que se complete el tratamiento de la interrupción en curso antes de proseguir. Una solución alternativa interesante consiste en que cuando la nueva activación detecta que ya hay un tratamiento en curso, anota en las estructuras de datos del sistema operativo que se ha producido esta segunda interrupción del mismo tipo y termina, siendo la misma rutina de tratamiento que está activa la encargada de procesar la segunda interrupción (la rutina de interrupción es un bucle que se repite hasta que no queden interrupciones del mismo tipo pendientes de tratar). Con este esquema puede mejorar el rendimiento del sistema ya que aprovecha la información de la rutina de tratamiento que ya está presente en la caché del primer procesador.

A continuación, se analiza el problema de sincronización que puede aparecer cuando se ejecutan concurrentemente llamadas al sistema, excepciones o procesos de núcleo. Nuevamente, en este caso la solución está basada en *spin-locks*, aunque, como se hizo en el estudio de este problema para un sistema uniprocesador, se va a realizar un análisis distinguiendo si se trata de un sistema con un núcleo expulsivo o no.

En el caso de un núcleo no expulsivo bastaría con que las llamadas al sistema que necesitan sincronizar ciertas partes de su ejecución usaran *spin-locks* para hacerlo. No sería necesario en este caso modificar el nivel (o estado) de interrupción, por lo que se usarían directamente las primitivas de gestión de este tipo de cerrojos (`spin_lock` y `spin_unlock`). Dado que en un sistema uniprocesador estas primitivas se convierten en operaciones nulas, no tendrán ningún efecto, lo cual no es sorprendente ya que, como se analizó anteriormente, para un núcleo no expulsivo en un sistema uniprocesador no existe este problema al no permitirse la ejecución concurrente de llamadas.

Si se trata de un núcleo expulsivo, además de usar *spin-locks*, es preciso asegurarse de que no se produce un cambio de contexto involuntario mientras se está en posesión de un cerrojo ya

que, como se explicó previamente, podría producirse un interbloqueo si el proceso que entra a ejecutar intenta obtener ese mismo cerrojo.

En sistemas que proporcionan niveles de prioridad para las interrupciones, se usarán las primitivas de *spin-locks* que alteran el nivel de las interrupciones (a las que previamente se las denominó `spin_lock_interrupcion` y `spin_unlock_interrupcion`), asociándoles como nivel de interrupción aquél que impide que se produzcan cambios de contexto involuntarios, es decir, el que inhibe las interrupciones software de planificación.

```
crear_proceso(...) {
    ...........
    nivel_anterior=spin_lock_interrupcion(cerrojo_tabla_proc,
                          NIVEL_INT_SW_PLANIFICACION);
    pos=BuscarBCPLibre();
    tabla_procesos[pos].libre = false;
    spin_unlock_interrupcion(cerrojo_tabla_proc, nivel_anterior);
    ...........
}
```

En caso de que se trate de un sistema sin niveles de prioridad de interrupciones, se usarán las funciones específicas para tal fin.

Nuevamente, si se redefinen como nulas las operaciones directas sobre el cerrojo, el resultado es el mismo que un sistema uniprocesador, es decir, se soluciona el problema inhibiendo la interrupción software de planificación en los fragmentos conflictivos.

Como ocurría con los semáforos para los sistemas uniprocesador con núcleos expulsivos, hay que establecer la granularidad del *spin-lock*. Se trata de la misma deliberación: intentar maximizar el paralelismo haciendo que la estructura de datos protegida por un *spin-lock* sea pequeña, pero suficientemente grande para que la sobrecarga por el uso de los distintos *spin-locks* implicados sea tolerable. Nótese que llevado a un extremo, se podría usar un único cerrojo para todo el sistema operativo. Ésta era la solución usada en la versión 2.0 de Linux, que utilizaba un cerrojo global, denominado *kernel_flag*, que aseguraba que en cada momento sólo un único procesador ejecutara en modo sistema el código del sistema operativo. Se trataba de una solución de compromiso que impedía cualquier tipo de paralelismo en la ejecución del sistema operativo y que, evidentemente, se ha ido mejorando en las sucesivas versiones de Linux, "rompiendo" ese único cerrojo en múltiples cerrojos que controlan el acceso a las distintas estructuras del sistema operativo.

Sea cual sea el modelo de núcleo, surge en este punto la misma cuestión que apareció en el análisis de la sincronización entre llamadas para un sistema monoprocesador con un núcleo expulsivo: ¿qué ocurre si la sección crítica es muy larga o requiere que el proceso pueda bloquearse durante la misma? Y la respuesta es la misma que en el análisis previo: se deberán implementar semáforos (o un mecanismo equivalente) para resolver estas deficiencias. Nótese que en este caso, una sección crítica muy larga, además de afectar directamente al tiempo de respuesta de los procesos, causa una disminución del paralelismo del sistema ya que puede haber uno o más procesadores haciendo espera activa mientras dura la sección crítica ejecutada en otro procesador.

Dadas las similitudes con en el caso de un núcleo expulsivo para un sistema uniprocesador, se pueden aplicar algunas de las consideraciones expuestas en el análisis realizado para ese caso:

- En cuanto a la implementación del semáforo, sus operaciones conseguirán atomicidad gracias al uso de los *spin-locks*, con las interrupciones software de planificación inhibidas en el caso de un núcleo expulsivo.
- Con el uso del semáforo, la sección crítica se ejecuta con todas las interrupciones habilitadas y sin mantener la posesión de ningún cerrojo, proporcionando un tiempo de respuesta y un nivel de paralelismo adecuados, y permitiendo que haya bloqueos durante la sección crítica. En cualquier caso, si la sección crítica es muy breve y no conlleva bloqueos, puede ser más adecuado usar directamente un *spin-lock* en vez de un semáforo, para evitar la sobrecarga asociada a las operaciones del semáforo.

- Nuevamente, hay que lograr un compromiso a la hora de fijar la granularidad del semáforo: la cantidad de información protegida por un semáforo debe de ser suficientemente pequeña para asegurar un nivel de paralelismo adecuado en la ejecución de llamadas al sistema, pero intentando limitar el número de semáforos que debe obtener el proceso durante una llamada al sistema para, de esta forma, acotar la sobrecarga debida a la sincronización.

Del análisis realizado en esta sección, se puede apreciar que los sistemas operativos para uniprocesadores con un modelo de núcleo expulsivo están mucho mejor preparados para afrontar el reto de adaptarse a sistemas multiprocesadores, dado que muchos de los desafíos que plantea la existencia de un paralelismo real, como el tratar con la sincronización de llamadas al sistema concurrentes, ya se abordan en los sistemas expulsivos.

Para completar esta sección, de manera similar a lo que hizo en las secciones previas, se presenta en la tabla 5 una recopilación de los posibles conflictos de sincronización para un multiprocesador, así como las soluciones a los mismos. Por simplicidad, sólo se muestra el caso de que el núcleo sea expulsivo: el no expulsivo sólo cambiaría en que no es necesario inhabilitar la expulsión en ningún caso. Es interesante resaltar en la tabla que en un multiprocesador para proteger una sección crítica hay que incluir medidas protectoras en todas las rutinas involucradas en la misma y no sólo en las de menor prioridad, como ocurre en un sistema monoprocesador.

Contextos atómicos

Dado que en este punto ya se han presentado todos los mecanismos de sincronización, se considera conveniente completar el análisis de qué propiedades se deben satisfacer para que en un determinado contexto de ejecución sea posible realizar un cambio de contexto y cuáles para que sea factible acceder a direcciones del mapa de usuario del proceso.

Para facilitar la explicación, se va a introducir el concepto de contexto atómico, tomado prestado de Linux. Se considera que en un determinado instante de la ejecución de una actividad de sistema se está en un contexto atómico si en ese momento se cumple **alguna** de estas condiciones:

- Es una rutina de interrupción de un dispositivo.
- Es una rutina de interrupción software de sistema.
- En ese momento están prohibidas las interrupciones de los dispositivos.
- En ese instante están inhibidas las interrupciones software de sistema.
- En ese punto está deshabilitada la expulsión de procesos (es decir, no está permitida la interrupción software de planificación).
- En ese momento se está en posesión de un *spinlock*.

A partir de esa definición, se puede establecer directamente bajo qué condiciones se permite que en un determinado punto se lleve a cabo un cambio de contexto: sólo se podrá llevar a cabo si en ese instante **no** se está ejecutando en un contexto atómico. Dicho de manera positiva, la

Rutina a estudiar	**Rutina conflictiva**				
	Llamada	**Excep**	**I. SW sis**	**I. dis. mn**	**I. dis. mx**
Llamada	Sem. o spin sin expul.	Sem. o spin sin expul.	Spin + Inh. int SW sis	Spin + Inh. int mín	Spin + Inh. int máx
Excepción	Sem. o spin sin expul.	Sem. o spin sin expul.	Spin + Inh. int SW sis	Spin + Inh. int mín	Spin + Inh. int máx
Int. SW sistema	Spin	Spin	Spin	Spin + Inh. int mín	Spin + Inh. int máx
Int. dispo. mínima	Spin	Spin	Spin	Spin	Spin + Inh. int máx
Int. dispo. máxima	Spin	Spin	Spin	Spin	Spin

Tabla 5 *Posibles conflictos de sincronización y su solución en multiprocesador con un núcleo expulsivo.*

condición seria la siguiente: únicamente se podrá realizar un cambio de contexto si se está ejecutando una llamada al sistema, una excepción, una interrupción software de proceso o un proceso de núcleo y, además, no está inhabilitado ningún tipo de interrupción ni se está en posesión de un *spinlock*.

En cuanto a la posibilidad de acceder a direcciones del mapa de usuario desde un determinado contexto, sólo se podrá llevar a cabo si en ese punto no se está ejecutando en un contexto atómico ni dentro de un proceso de núcleo.

Sincronización en núcleos para sistemas de tiempo real

Como ya se ha analizado previamente, el principal reto de este tipo de sistemas frente a los convencionales es controlar la inversión de prioridades, es decir, las situaciones patológicas durante las cuales está en ejecución un determinado proceso mientras que no puede hacerlo uno de mayor prioridad que está listo para ejecutar. Téngase en cuenta que la inversión de prioridades desvirtúa la noción de prioridad, que es clave para un sistema de tiempo real, y puede hacer que una aplicación de estas características fracase.

En la sección dedicada al estudio de los cambios de contexto involuntarios, se mostró que para evitar la inversión de prioridades acotada asociada a los núcleos convencionales, tanto expulsivos como, sobretodo, no expulsivos, una solución habitual en los núcleos para sistemas de tiempo real es procesar las interrupciones, tanto de los dispositivos como las software de sistema, en el contexto de procesos de núcleo.

Por lo que se refiere a los mecanismos de sincronización, algunos de los utilizados en los núcleos convencionales no son adecuados, al menos de forma general, para los núcleos con un perfil de tiempo real:

- Deshabilitar temporalmente la expulsión de procesos para atajar los problemas de sincronización debidos a la ejecución concurrente de procesos en un procesador es una solución que causa inversión de prioridades en la que, además, se ven afectados procesos que no están involucrados en el problema de sincronización que se pretende resolver.
- Inhabilitar interrupciones para evitar problemas de sincronización debidos al tratamiento de eventos asíncronos en un procesador aumenta el tiempo de latencia de interrupción y, en consecuencia, el de activación de un proceso, con la consiguiente inversión de prioridades.
- El uso de *spinlocks* para controlar los problemas de sincronización en multiprocesadores causa que en un procesador donde un proceso mantiene un *spinlock* no pueda haber expulsiones, provocando el mismo comportamiento que en el primer punto analizado.

Dadas estas consideraciones, la solución habitual es usar un mecanismo de tipo semáforo para afrontar todos los problemas de sincronización presentes en este tipo de sistemas, puesto que este mecanismo afecta únicamente a los procesos involucrados en el problema. Sobre el uso de semáforos, hay que hacer algunas aclaraciones:

- Se puede usar este mecanismo, aunque sea bloqueante, para todo tipo de situaciones puesto que toda actividad del sistema se ejecuta en el ámbito de un proceso.
- Para construir el propio mecanismo de semáforo, se requerirá el uso de las técnicas que se han considerado inadecuadas para este tipo sistemas (inhibir interrupciones, deshabilitar expulsiones y uso de *spinlocks*), pero se trata de un uso totalmente restringido y optimizado al máximo para que la inversión de prioridades inherente a estos mecanismos sea tolerable.
- El uso de semáforos para secciones críticas muy breves puede causar sobrecarga frente a soluciones más ligeras, como los *spinlocks*. Sin embargo, hay que tener en cuenta que el objetivo de un sistema de tiempo real no es hacer las cosas lo más rápido posible sino garantizar los plazos que requiere una aplicación.

Centrándonos en los semáforos, o mecanismo equivalente, el primer requisito que deben satisfacer es que respeten las prioridades de los procesos, de manera que, si se abre un semáforo donde hay varios procesos esperando, se desbloquee el más prioritario. Este requisito es fácilmente alcanzable y lo cumplen la mayoría de los sistemas de propósito general.

Una cuestión mucho más peliaguda es controlar la inversión de prioridades que puede provocar este mecanismo, que, además, puede no estar acotada en el tiempo, lo que hace que el problema sea todavía más grave. Veámoslo con un ejemplo:

- El proceso P de baja prioridad obtiene un semáforo.
- A continuación, se activa el proceso Q, de alta prioridad, que se bloquea al intentar obtener el mismo semáforo.
- Acto seguido, entra a ejecutar el proceso R, de prioridad media, que no tiene nada que ver con el semáforo, no permitiendo, lógicamente, la progresión de P, pero, indirecta y erróneamente desde el punto de vista del cumplimiento de las prioridades, tampoco la del proceso Q.

El ejemplo planteado muestra una situación de inversión de prioridades, que, además, no está acotada en el tiempo: se podría suceder indefinidamente la ejecución de procesos de prioridad intermedia, que impedirían la ejecución del proceso P, y con ello la de Q.

Para resolver este problema, en el ejemplo planteado, se necesitaría que, una vez que Q se bloquea esperando por el semáforo, la prioridad de ejecución de P aumentara heredándola de Q. De esta forma, los procesos de prioridad intermedia no afectarían a P hasta que libere el semáforo y desbloquee a Q, momento en el que P recobraría su prioridad original.

En esa idea se basa la técnica de **herencia de prioridades**, que es la solución más habitual en los núcleos de tiempo real pata afrontar este tipo de problemas de inversión de prioridades no acotada. Con este esquema, la prioridad de un proceso en un momento dado sería el valor máximo entre la suya propia y la de los procesos que están esperando por semáforos que posee dicho proceso. Nótese que este ajuste no es trivial y se producen cadenas de dependencias de semáforos en las que hay que realizar una propagación transitiva de la prioridad, como se puede observar en el siguiente ejemplo:

- El proceso P de baja prioridad obtiene el semáforo S1.
- A continuación, se activa el proceso Q, de prioridad media, que obtiene el semáforo S2 y luego se bloquea al intentar obtener el semáforo S1. En ese momento, el proceso P hereda la prioridad de Q.
- Acto seguido, entra a ejecutar el proceso R, de prioridad alta, que se bloquea al intentar obtener el semáforo S2. Por herencia de prioridades, la prioridad de Q, aunque esté bloqueado, pasa a ser la de R, lo que se propaga a P, que también hereda la prioridad del proceso R.

5.2 Sincronización de procesos

Aunque en este capítulo se ha asumido que el lector ya conoce los servicios de sincronización que ofrece el sistema operativo a las aplicaciones y, por tanto, no se ha incidido en ese aspecto, en esta sección se exponen algunas consideraciones sobre la relación entre los mecanismos internos que usa el sistema operativo para su correcta sincronización y los ofrecidos a las aplicaciones.

En principio, parecería razonable que se usarán directamente los mismos mecanismos tanto para la sincronización interna como para la externa. Al fin y al cabo, si un mecanismo sirve para sincronizar la ejecución de procesos cuando ejecutan en modo sistema, debería valer también para hacerlo cuando ejecutan en modo usuario. ¿Por qué no usar para sincronizar las aplicaciones directamente los mismos semáforos que se utilizan internamente? Sin embargo, hay que hacer ciertas matizaciones a esa posibilidad.

En primer lugar, hay que tener en cuenta que internamente se usa directamente la dirección del objeto de sincronización (ya sea un semáforo o mecanismo equivalente) como su identificador. Aunque se podría exportar directamente ese valor de manera que las aplicaciones lo utilizarán como identificador del elemento de sincronización, parece más razonable usar alguna forma de identificador más orientado a las aplicaciones, como podría ser un nombre de ruta. El sistema operativo debería gestionar la correspondencia entre los identificadores que usan las aplicaciones y el identificador interno del objeto de sincronización.

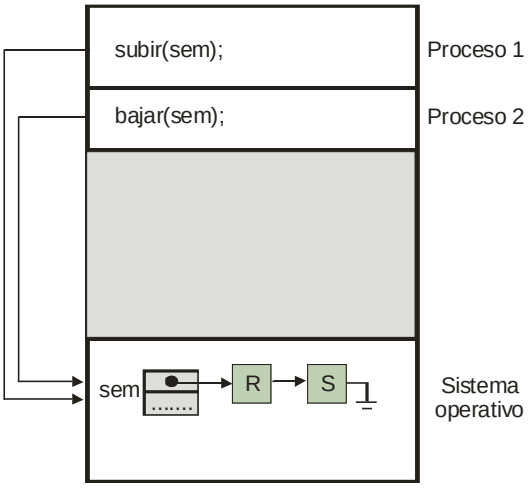


Figura 30 *Mecanismo de sincronización con soporte del sistema operativo.*

Por otro lado, en muchas ocasiones la funcionalidad de los mecanismos de sincronización que se ofrecen a las aplicaciones no puede definirse a criterio del diseñador del sistema operativo, sino que debe satisfacer algún estándar, que posiblemente recoja una funcionalidad relativamente compleja que dificulte su uso como mecanismo de sincronización interno. Para apreciar esta desavenencia, sólo es necesario estudiar cómo son los semáforos definidos en UNIX System V. Su excesiva complejidad desaconseja utilizarlos como mecanismo de sincronización interno del sistema operativo, puesto que conllevarían una sobrecarga apreciable.

En cualquier caso, aunque los mecanismos de sincronización ofrecidos a las aplicaciones no sean el resultado de exportar directamente los mecanismos internos, se construyen basándose en los mismos, de manera que un elemento de sincronización externo tendrá asociado uno o más elementos internos para construir su funcionalidad.

Estos elementos de sincronización destinados al uso por parte de las aplicaciones requieren la activación del sistema operativo para cada una de las operaciones (en el caso de un semáforo de usuario, `subir` y `bajar` serían llamadas al sistema), con la consiguiente sobrecarga. La figura 30 muestra este esquema de sincronización. En ocasiones, sin embargo, puede requerirse una sincronización más ligera, que pueda construirse en gran parte en modo usuario y que sólo requiera una pequeña intervención por parte del sistema operativo.

El primer aspecto que hay que tener en cuenta para lograr este tipo de sincronización más ligera es que todo elemento de sincronización se basa en que los usuarios del mismo puedan compartir el almacenamiento en memoria que le da soporte (por ejemplo, si se trata de un semáforo, el contador y la cola asociados al mismo). En el caso de un elemento de sincronización implementado dentro del sistema operativo, este requisito se satisface automáticamente puesto que todos los procesos en modo sistema comparten el mapa de memoria del sistema operativo. Sin embargo, si se pretende construir un elemento de sincronización basado principalmente en operaciones en modo usuario, será necesario que los procesos que vayan a usar este mecanismo compartan una zona de memoria de usuario donde se aloje el almacenamiento requerido por el elemento de sincronización. Por tanto, ya sea haciendo de forma explícita que los procesos involucrados establezcan una zona de memoria compartida o realizándolo de forma implícita dentro del código de usuario que implementa el elemento de sincronización, habrá que conseguir satisfacer este requisito. Nótese que en el caso de que el elemento de sincronización esté destinado a la sincronización de hilos del mismo proceso, el compartimiento requerido se produce de forma automática, puesto que, como se analizará en al próxima sección, todos los hilos de un proceso comparten su mapa de memoria.

Una vez obtenido el soporte de almacenamiento compartido requerido, la implementación del mecanismo ligero de sincronización seguiría una pauta similar a la que se describe a continuación, en la que se ha supuesto que se trata de un semáforo (véase la figura 31):

- La operación de bajar el semáforo estaría implementada en una biblioteca de usuario y actualizaría el contador del semáforo, que estaría almacenado en una zona compartida de

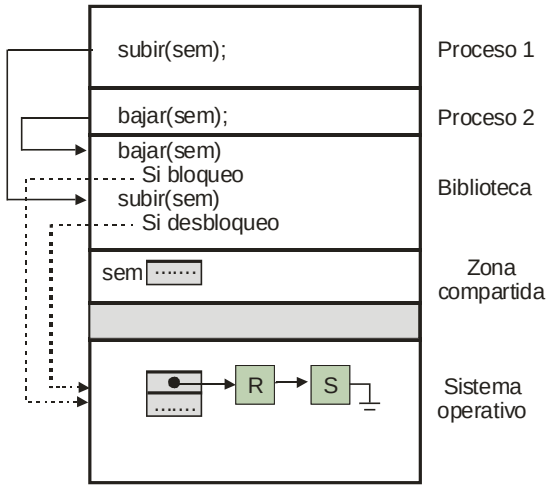


Figura 31 *Mecanismo de sincronización con soporte de usuario y del sistema operativo.*

memoria de usuario, siguiendo el comportamiento habitual de los semáforos. Sólo en el caso de que como resultado de la operación se requiera bloquear el proceso, se invocaría una llamada al sistema creada para tal fin, de manera que el proceso se quede bloqueado en un elemento de sincronización interno asociado al semáforo de usuario.

- La operación de subir el semáforo estaría también implementada en una biblioteca de usuario y actualizaría el contador del semáforo conforme al comportamiento habitual de este mecanismo. Sólo en el caso de que como resultado de la operación se requiera desbloquear un proceso, se invocaría una llamada al sistema creada para este fin, que produciría el desbloqueo de un proceso que estaba bloqueado en el elemento de sincronización interno asociado al semáforo de usuario.

Para concluir esta sección, hay que resaltar que, en el caso de un sistema de tiempo real, los mecanismos de sincronización para procesos en modo usuario también estarían integrados en el esquema de herencia de prioridades requerido por este tipo de sistemas.

6 Implementación de hilos

De forma intencionada, a lo largo de este capítulo se ha estudiado la gestión de procesos sin introducir el concepto de hilo, retrasando hasta este punto la presentación del mismo. Con ello se pretende que el lector pueda apreciar mejor lo que aporta esta abstracción al modelo de procesos.

Para entender la importancia de esta nueva entidad, hay que contemplar cómo se vivía antes de que apareciese en el mundo de los sistemas operativos. El modelo de procesos "tradicional" (sin hilos) ofrece una solución que satisface adecuadamente dos de los objetivos de los sistemas multiprogramados planteados al principio del tema: permite obtener un buen rendimiento en el uso del procesador y consigue proporcionar un soporte seguro y eficiente a múltiples usuarios. Sin embargo, presenta algunas deficiencias a la hora de cumplir el tercero de los objetivos planteados: dar un soporte adecuado y eficiente para la ejecución de aplicaciones concurrentes.

El problema reside en la propia esencia del concepto de proceso: una abstracción que crea un procesador virtual que independiza la ejecución de distintas actividades. Esa independencia es perfectamente válida para aislar la ejecución de actividades que pertenecen a diferentes usuarios o a distintas aplicaciones del mismo usuario. Sin embargo, cuando se trata de actividades concurrentes de la misma aplicación, esa independencia puede ser innecesaria y provocar una ejecución ineficiente. Expresado de manera informal, se puede decir que los muros que hemos levantado para proteger entre sí la ejecución concurrente de múltiples actividades, resultan inadecuados y molestos cuando esas actividades confían entre sí al tratarse de partes de la misma aplicación. A continuación, se enfatiza en los dos aspectos que pueden causar problemas de rendimiento en una aplicación paralela ejecutando en un sistema con un modelo de procesos tradicional:

- Las actividades concurrentes de la misma aplicación deberán utilizar los mismos mecanismos de interacción que usan actividades de distintas aplicaciones. Estos mecanismos están cuidadosamente diseñados para asegurar la protección entre los procesos, lo que conlleva una cierta sobrecarga ineludible, y con la suposición implícita de que se usarán para un patrón de interacción relativamente infrecuente. Las actividades concurrentes de una misma aplicación tienen en algunos casos un alto grado de interacción y necesitan un mecanismo eficiente para comunicarse.
- El cambio de contexto entre actividades concurrentes de la misma aplicación es el mismo que se lleva a cabo entre actividades de distintas aplicaciones. Esta operación es bastante costosa, como se ha estudiado a lo largo de este capítulo, puesto que hay desinstalar el contexto del proceso que deja el procesador e instalar el contexto del que lo recibe. Siguiendo con el símil de los muros de protección, hay que derribar los del primer proceso y reedificar los del segundo. Sin embargo, si se trata de dos actividades concurrentes de la misma aplicación, el cambio debería ser mucho más rápido porque comparten una parte importante de los recursos.

Para resolver este problema, se planteó en su momento la necesidad de introducir una nueva abstracción, el hilo (*thread*), y redefinir el concepto de proceso:

- Un hilo es un flujo que ejecuta en el contexto de un proceso.
- Un proceso consiste en un conjunto de recursos que proporcionan un contexto para la ejecución de un programa. Entre los recursos asociados a un proceso están:
    - El conjunto de ficheros abiertos.
    - El mapa de memoria del proceso.
    - El conjunto de elementos de sincronización usados por el mismo.
    - El conjunto de hilos activos en el mismo.

Con esta redefinición, el hilo se convierte en la única entidad activa del sistema. El proceso pasa a ser un ente pasivo, un contenedor de recursos. Antes de introducir los hilos, la abstracción del proceso proporcionaba un ámbito de protección y un soporte para expresar la concurrencia. Con la incorporación de los hilos, se separan estos dos aspectos: el proceso proporciona el ámbito de protección, mientras que el hilo ofrece el soporte de la concurrencia.

Desdoblando la funcionalidad en dos abstracciones, se resuelve el problema identificado: las múltiples actividades concurrentes de una misma aplicación se corresponderán con múltiples hilos del mismo proceso. Dado que todos los hilos de la misma aplicación ejecutan en el ámbito del mismo proceso, tendrán acceso a todos sus recursos, incluido el mapa de memoria, consiguiendo una interacción muy eficiente. Además, como veremos en la siguiente sección, el cambio entre hilos de la misma aplicación va a ser mucho más eficiente, puesto que el contexto que hay que instalar se reduce apreciablemente, al compartir los recursos del proceso.

Las actividades pertenecientes a diferentes aplicaciones, se corresponderán con hilos de distintos procesos. El cambio de contexto y la interacción entre hilos de distintos procesos tendrán las mismas características, en cuanto a eficiencia, que en un sistema sin hilos.

En la mayoría de los sistemas operativos, la ejecución del hilo está asociada a una función del programa. El servicio que permite crear un hilo, normalmente, requiere que se especifique la función que ejecutará ese hilo y un argumento que se le pasará a la misma (dado que todos los hilos del mismo proceso comparten el mapa de memoria, no resulta una restricción el sólo poderle pasar un único parámetro). La ejecución del hilo se restringirá a esa función y éste terminará cuando acabe dicha función. Observe que para el programador, un hilo es una forma de invocación asíncrona de una función.

Para dar cobertura a los procesos tradicionales, que no usan explícitamente el concepto de hilo, se considera que todo proceso tiene asociado un hilo implícito, que se dedica a ejecutar la función principal del programa (en C, la función `main`).

Cuando se diseña un nuevo sistema operativo que incorpora la abstracción del hilo, como ocurre en Windows, ésta encaja de forma natural en el modelo de operación del sistema operativo. Sin embargo, su incorporación a posteriori en un sistema operativo que no se diseñó teniendo en mente esta nueva abstracción resulta problemática. No se trata sólo de problemas en lo que se refiere a aspectos de implementación, que se verán en las próximas secciones, sino en la definición de la propia funcionalidad del sistema operativo, que debe ser replanteada en

algunos aspectos. Eso ha ocurrido en los sistemas operativos UNIX, cuyo diseño original precede al nacimiento del concepto del hilo. A continuación, se muestran tres ejemplos de aspectos que hay que reconsiderar al incorporar hilos en un sistema UNIX:

- Cuando un hilo de un proceso con múltiples hilos realiza una llamada `fork`, ¿el proceso hijo debe tener un único hilo o tantos como tenía el padre?
- Cuando un hilo de un proceso con múltiples hilos realiza una llamada `exec`, ¿qué ocurre con los otros hilos del proceso?
- ¿Cómo se adapta el modelo de señales a un sistema con hilos? Por ejemplo, si un hilo ejecuta `alarm`, ¿a quién se le manda la señal `SIGALRM`?, ¿al hilo que hizo la llamada (quizás ya haya terminado)?, ¿a uno cualquiera?, ¿a todos?

Estas preguntas tienen, evidentemente, respuesta (el lector puede consultar el estándar actual de UNIX para obtenerla), pero eso no es lo que nos interesa en esta exposición, donde lo que sí queremos reflejar es un problema general: incorporar una nueva abstracción a un sistema que se concibió sin tenerla en cuenta siempre resulta un proceso traumático.

Antes de entrar en analizar cómo se implementa esta nueva abstracción, y dado que en el resto del capítulo se ha obviado su existencia, es necesario reinterpretar todas las secciones previas bajo este nuevo prisma. Todas las referencias previas al concepto de proceso, siempre que se hiciera desde el punto de vista de una entidad activa, ahora se convierten en referencias a un hilo. A continuación, se comentan algunas de estas nuevas interpretaciones que surgen al incorporar el concepto de hilo.

- Los cambios de contexto se producen entre hilos, ya sean del mismo proceso o de distintos.
- El diagrama de estados estudiado corresponde a la evolución de un hilo, y las colas asociadas, son listas de hilos. El sistema operativo almacenará en una variable cuál es el hilo que está en ejecución actualmente.
- El hilo va transitando entre fases en modo usuario (utilizando su pila de usuario) y fases en modo sistema (usando su pila de sistema, puesto que, como se verá en la próxima sección, cada hilo tiene su propia pila de sistema).
- La sincronización se realiza entre hilos.
- La unidad de planificación del procesador, aunque este aspecto corresponda a otro tema de esta materia, es el hilo.

6.1 Aspectos de implementación

Como cada tipo de objeto que maneja el sistema operativo, un hilo tendrá vinculado un descriptor que almacenará la información asociada al mismo. Por similitud con el del proceso, le denominaremos "bloque de control de hilo" (BCH). Dado que el hilo toma prestada parte de la funcionalidad que antes estaba asignada al proceso, será precisamente esta información la que deje de formar parte del BCP y se incluya en el BCH. Corresponderá a toda aquella información que esté vinculada con aspectos asociados directamente a la ejecución:

- El estado del hilo.
- Un espacio para almacenar una copia de los registros del procesador.
- La dirección de las pilas de usuario y de sistema del hilo.
- Información de planificación (por ejemplo, la prioridad del hilo).
- Un puntero al BCP del proceso al que pertenece el hilo. Esta referencia permite acceder a los recursos del proceso desde el BCH.

En el BCP, quedan los recursos que conforman el ámbito pasivo de ejecución del proceso:

- El mapa de memoria del proceso. Además de las regiones habituales del proceso, en el mapa habrá una pila de usuario para cada hilo.
- Información de ficheros, de entrada/salida, de elementos de sincronización y de otros recursos asociados al proceso.
- Información sobre el usuario al que está vinculado este proceso.
- Una lista de los hilos del proceso.

La figura 32 muestra un ejemplo de un proceso con dos hilos, mostrando las relaciones entre ambos tipos de estructuras. Se ha supuesto un sistema UNIX con hilos (como, por ejemplo,

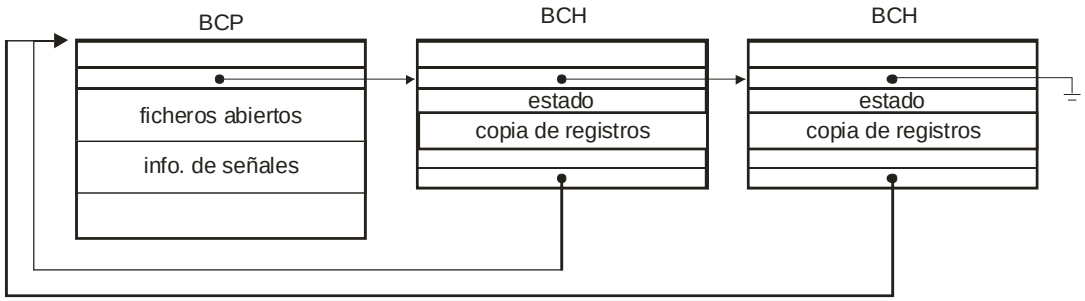


Figura 32 *Estructuras de datos para la implementación de hilos.*

Solaris), mostrando en la figura que la información sobre el estado de las señales (por cada señal, si está ignorada, capturada o tiene asociada la acción por defecto) se almacena en el BCP, por lo que la comparten todos los hilos del mismo proceso.

Cambios de contexto entre hilos

Como se comentó previamente, uno de los objetivos de los hilos es agilizar el cambio de contexto entre actividades de la misma aplicación. Este objetivo se logra gracias a que dos hilos del mismo proceso comparten gran parte de su contexto de ejecución, y sólo es necesario cambiar la parte en que difieren. A continuación, se analiza cómo se simplifica el cambio de contexto cuando se trata de hilos del mismo proceso.

Tomemos como punto de partida la operación del cambio de contexto entre procesos tal como se estudió en la sección dedicada al mismo. Todas las operaciones que tienen que ver con la salvaguarda y restauración de los registros del procesador, así como las que manipulan el estado de los procesos y de las colas de procesos, se mantienen en un cambio entre hilos, con la única diferencia que en este caso se actualiza el estado y las colas de hilos.

A primera vista, no se aprecia ninguna diferencia, en cuanto a eficiencia, entre un cambio entre hilos del mismo proceso y uno que involucre hilos de distintos procesos. La clave está en el mapa de memoria del proceso. Cuando el cambio de contexto involucra a hilos de distintos procesos, es necesario instalar el mapa de memoria del nuevo proceso como parte de la operación de cambio de contexto, al igual que lo que ocurría con un cambio de contexto entre procesos. Sin embargo, en un cambio entre hilos del mismo proceso, se mantiene el mapa de memoria en el cambio de contexto. Como se estudia en el tema de gestión de memoria, esta operación de instalar un nuevo mapa de memoria consume un tiempo apreciable, requiriendo, entre otras acciones, invalidar la TLB del procesador.

Dentro del código del cambio de contexto, se comprobará si los hilos afectados por el mismo pertenecen al mismo proceso. Sólo será necesario instalar el mapa del nuevo proceso en caso de que estén asociados a distintos procesos:

```
if (previo->proceso != siguiente->proceso)
      instalar_mapa(siguiente->proceso->mapa);
```

Creación y terminación de un hilo

Las operaciones requeridas para crear un hilo son similares a las usadas al crear un proceso en el modelo de tradicional para establecer el contexto inicial de ejecución del mismo. Como ocurría en ese caso, se deberá usar una función de lanzadera dentro del sistema operativo y habrá que asegurarse de que si el hilo termina sin invocar la llamada de terminación de un hilo, ésta se invoca de forma automática. En cuanto a este último aspecto, hay múltiples soluciones para asegurarlo. En esta sección, se plantea una estrategia en la que la pila de usuario del hilo se dispone de manera que si la función que ejecuta el hilo termina sin invocar la llamada de terminación de hilo, al retornar, se ejecutará de forma automática.

La figura 33 muestra el contexto inicial de ejecución de un nuevo hilo. A continuación, se enumeran las operaciones que conllevaría la creación de un hilo:

- Reserva un BCH para el nuevo hilo y lo incluye en la lista de hilos del proceso.

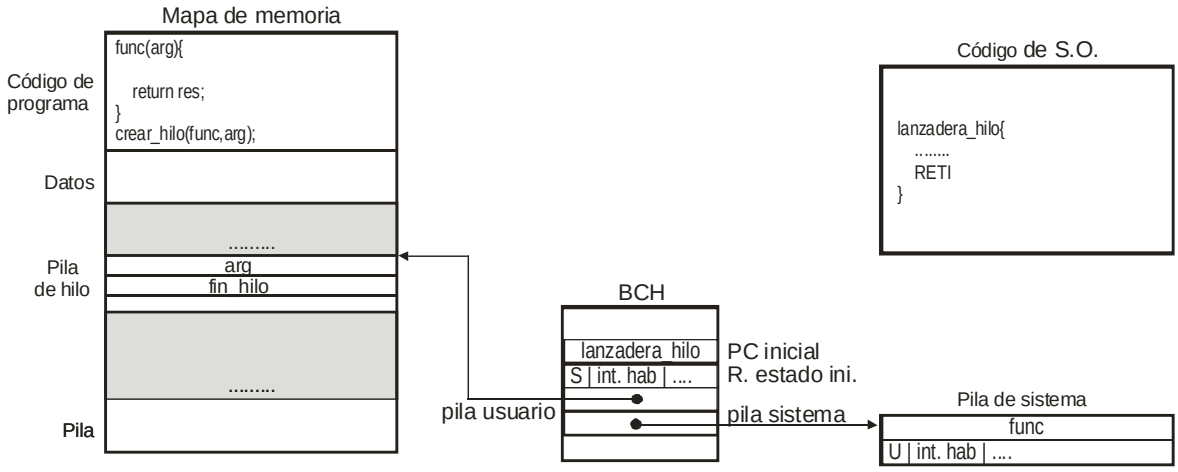


Figura 33 *Disposición inicial en la creación de un hilo.*

- Reserva espacio para la pila de sistema del hilo. Inicia esa pila de manera que contenga la dirección de la función que ejecutará el hilo, pasada como argumento al crearlo, y un valor que se corresponderá con el registro de estado inicial del hilo.
- Crea la pila de usuario incluyendo en ella el argumento del hilo y, para asegurar que siempre se invoca la función de terminación, se incluye como dirección de retorno la de la función de terminación de un hilo.
- Inicia los diversos campos del BCH con los valores correspondientes, entre ellos un puntero al BCP del padre. Asimismo, se fijará un valor inicial de los registros tal que cuando se active el hilo por primera vez comience en la rutina de lanzadera, en modo sistema y usando la pila de sistema del hilo.
- Pone el hilo en estado de listo para ejecutar y lo incluye en la cola de hilos en ese estado.

Por otro lado, habrá que modificar la función de creación de proceso para que lleve a cabo la creación del hilo inicial implícito, que ejecutará la función principal del programa.

En cuanto a la terminación del hilo, conlleva las siguientes operaciones:

- Libera la pila de usuario del hilo.
- Elimina el BCH de la cola de hilos listos para ejecutar y de la lista de hilos del proceso.
- Libera el BCH.
- Libera la pila de sistema del hilo.
- Activa el planificador y realiza un cambio de contexto voluntario al hilo seleccionado por el mismo.

Nuevamente, surge el problema de liberar la pila de sistema mientras se está utilizando. Para resolverlo, se puede usar la misma táctica que en el modelo tradicional: que sea el hilo al que se le cede el control el que se encargue de liberar la pila de sistema y el BCH del hilo terminado.

Por último, en cuanto a la terminación del proceso, se presentan dos alternativas sobre cuándo se lleva a cabo:

- Que el proceso termine cuando acabe su último hilo.
- Que el proceso, y todos sus hilos, terminen cuando acabe el hilo inicial implícito.

Con respecto a las operaciones asociadas a la terminación del proceso, serán similares a las del modelo tradicional, pero adaptándose a las nuevas estructuras de datos.

6.2 Hilos de usuario

La invención del concepto de hilo respondía a una necesidad real y, por tanto, tuvo muy buena acogida entre los desarrolladores de aplicaciones concurrentes. Sin embargo, su incorporación a un sistema operativo ya existente presentaba un reto exigente, tanto en lo que atañe a aspectos de implementación como en cuanto a la necesidad de rediseñar la funcionalidad de ciertas partes del propio sistema operativo. En algunos sistemas operativos, mientras sus creadores se enfrentaban a este nuevo reto, para intentar contentar a sus usuarios, se desarrollaron bibliotecas de usuario que implementaban esta nueva abstracción. A este tipo de hilos, creados sin la

intervención del sistema operativo, que desconoce totalmente su existencia, se les suele denominar hilos de usuario, frente a los implementados por el sistema operativo, a los que se conoce como hilos de sistema (o de núcleo).

La biblioteca de hilos se encargaba de crear esta abstracción gestionando múltiples flujos de ejecución concurrentes dentro de la ejecución del proceso. A este esquema se le suele denominar **modelo N:1**, puesto que todos los hilos de un programa se proyectan sobre una única entidad ejecutable del sistema operativo (frente al **modelo 1:1** que corresponde al uso directo de los hilos del sistema).

Todos los aspectos de gestión los llevaba a cabo la biblioteca: los cambios de contexto de hilos, su creación y terminación, así como la planificación y la sincronización de hilos. Se trataría de una implementación similar a la presente en algunos lenguajes de programación que ofrecen concurrencia (por ejemplo, las tareas del lenguaje Ada).

Evidentemente, esta solución presenta importantes deficiencias, lo que no puede sorprender ya que, en caso contrario, nadie se habría planteado incorporarlo dentro del sistema operativo:

- Si un hilo de usuario realiza una llamada al sistema bloqueante, dado que el sistema operativo no sabe nada de los hilos, bloqueará al proceso y, en consecuencia, a todos sus hilos. Algunas implementaciones de bibliotecas de hilos de usuario intentan capturar las llamadas bloqueantes y usar, en su lugar, versiones no bloqueantes, pero esta técnica no proporciona una solución general.
- Si se usa en un sistema multiprocesador, todos los hilos de usuario de un proceso ejecutarán en el mismo procesador, puesto que el sistema operativo asigna procesos a procesadores.

Esta solución, sin embargo, presentaba algunos beneficios debidos precisamente a no involucrar al sistema operativo en su gestión:

- Los hilos de usuario son más ligeros que los de sistema. Tanto su creación, planificación, sincronización, así como sus cambios de contexto, son más eficientes ya que se realizan en modo usuario, sin la intervención del sistema operativo.
- Los hilos de usuario consumen menos recursos del sistema operativo. No requieren una pila de sistema. Además, el espacio usado para su bloque de control y su pila de usuario corresponde a espacio de usuario que, gracias a la técnica de memoria virtual, no necesita estar todo el tiempo residente en memoria. Esta economía en el uso de recursos permite plantear aplicaciones con un gran número de hilos.
- La planificación de los hilos no la gestiona el sistema operativo sino la biblioteca de usuario. Esta característica proporciona una mayor flexibilidad porque permite adaptarse a las características específicas de cada aplicación.

Estos beneficios, sin embargo, no compensan las importantes deficiencias identificadas previamente, lo que hace que la solución razonable sea la implementación de hilos de sistema.

En principio, este tipo de estrategia puede parecer como una solución transitoria, ya obsoleta, sin más interés que reflejar la evolución histórica. Sin embargo, como se analiza en la siguiente sección, hay modelos de implementación de hilos que, aunque involucran al sistema operativo, intentan mantener algunos de los beneficios de los hilos de usuario.

6.3 Esquema híbrido de gestión de hilos

En el mundo de la informática se dan con cierta frecuencia alternativas de diseño contradictorias en el sentido de que la mejora de un cierto aspecto del sistema perjudica a otro factor de igual importancia. Eso es lo que ocurre en la alternativa entre hilos de usuario y de sistema: si el sistema operativo interviene en la implementación de los hilos, se obtiene una solución con una mejor funcionalidad; si no interviene, se consigue una implementación más eficiente y con un menor consumo de recursos. ¿Se puede lograr una solución que consiga lo mejor de cada alternativa de diseño?

Algunos sistemas operativos, como Solaris o Mach, optan por una solución híbrida, en la que el modelo de hilos se construye con el trabajo combinado de una biblioteca de usuario y del sistema operativo. Obsérvese que ya hemos estudiado una estrategia de esta índole cuando se analizó la sincronización entre procesos. Este esquema se denomina **modelo M:N**, puesto que

se proyectan los *M* hilos de usuario de un programa sobre *N* entidades ejecutables del sistema operativo.

A continuación, se exponen las ideas principales de esta estrategia híbrida:

- Los programas crean hilos de usuario, mientras que el sistema operativo proporciona hilos de sistema.
- La biblioteca de usuario se encarga de la correspondencia entre ambos niveles de hilos, multiplexando la ejecución de los hilos de usuario del proceso entre uno o más hilos de sistema (*M* hilos de usuario entre *N* hilos de sistema, siendo $M \geq N$).
- La biblioteca de usuario va creando hilos de sistema según vaya necesitándolos. Inicialmente, cuando arranca el proceso, crea al menos un hilo de sistema.
- Para que un hilo de usuario pueda ejecutar en un momento dado, tiene que tener asignado uno de los hilos de sistema creados por la biblioteca.
- Si durante la ejecución de un hilo de usuario éste realiza una llamada bloqueante, el sistema operativo bloquea el hilo de sistema, que es la única entidad que conoce. La biblioteca puede decidir en ese momento crear otro hilo de sistema para dar servicio a los hilos de usuario del proceso.
- Si la biblioteca aprecia que el número de hilos de sistema del proceso es excesivo, puesto que detecta que muchos de ellos apenas tienen trabajo, puede terminar la ejecución de algunos. De esta manera, el número de hilos de sistema creados por una biblioteca se va adaptando a las necesidades del programa.
- Bajo este modelo híbrido, el proceso podrá usar tantos procesadores como hilos del sistema tenga asociados.
- La mayor parte de las operaciones de los hilos se solventarán en la biblioteca, sin necesidad de que intervenga el sistema operativo, con la consiguiente mejora en la eficiencia.
- El gasto de recursos del sistema operativo por parte del proceso va a venir condicionado por el número de hilos de sistema, y no por el de hilos de usuario. Por tanto, se puede dar soporte a aplicaciones con un número muy elevado de hilos.

Una solución híbrida alternativa es la que representa el esquema de las "activaciones del planificador" (*Scheduler Activations*), que se comentará de forma muy breve a continuación.

En el esquema híbrido explicado previamente se ha perdido uno de los beneficios del uso de hilos de usuario: no es la biblioteca de usuario la que planifica los hilos de una aplicación, sino que de ello se encarga el sistema operativo que, al planificar los hilos de sistema, está indirectamente haciendo la planificación de los hilos de usuario.

Para resolver este problema, el modelo de "activaciones del planificador" propone el siguiente esquema:

- Una activación del planificador puede considerarse, a los efectos de esta exposición, similar a un hilo de sistema.
- Cuando se inicia una aplicación, el sistema operativo crea una activación del planificador para dar servicio a la misma. Posteriormente, el sistema operativo irá asignando a la aplicación un número de activaciones del planificador que dependerá de factores como el paralelismo intrínseco del programa, el número de procesadores disponibles en el sistema y la carga de procesamiento total existente en el mismo.
- Siempre que se crea una nueva activación del planificador, ésta realiza una especie de llamada que va de modo sistema a modo usuario (denominada "llamada ascendente", *upcall*) pasando a ejecutar directamente una rutina específica de la biblioteca de hilos de usuario. Esa rutina selecciona (planifica) un hilo de usuario listo para ejecutar y lo vincula con esa misma activación del planificador.
- Cuando una activación del planificador se bloquea como consecuencia de que el hilo de usuario que está vinculado a la misma realiza una llamada al sistema bloqueante, el sistema operativo crea una nueva activación del planificador que, mediante una llamada ascendente, se lo notifica al planificador de hilos de usuario. Éste cambia el estado de ese hilo a bloqueado y asigna un hilo de usuario listo para ejecutar a la propia activación del planificador que ha realizado la notificación.

- Cuando un hilo de usuario se desbloquea, el sistema operativo usa una activación del planificador para notificarlo al planificador de hilos de usuario. Dependiendo del número de activaciones del planificador asignadas al proceso, el sistema operativo puede crear una nueva, como ocurría en el punto anterior, o usar una de las actividades de planificación ya asignadas al programa desvinculándola del hilo de usuario que tuviera asociado. El planificador pone en estado de listo para ejecutar al hilo desbloqueado (y al hilo que ha perdido su activación del planificador en caso de que así haya sido) y selecciona a qué hilo de usuario asignar la activación del planificador que realizó la notificación.

6.4 Procesos de peso variable

Como se ha podido apreciar a lo largo de esta sección, la incorporación del modelo de hilos en un sistema operativo que no contemplaba esta abstracción resulta bastante traumática. Las repercusiones se extienden prácticamente por todos los componentes del sistema operativo, provocando un esfuerzo de reprogramación muy apreciable. Es, por tanto, bastante lógico que se hayan buscado alternativas que causen un menor impacto, como la que se analiza en esta sección. La idea es plantear una evolución en vez de una revolución.

Si nos retrotraemos al problema original, lo que se pretende es conseguir que las diversas actividades de una misma aplicación puedan compartir libremente sus recursos, facilitando su interacción y acelerando los cambios de contexto entre las mismas. Usando el concepto de hilo, se puede resolver el problema acogiendo todas estas actividades como hilos de un mismo proceso.

Sin embargo, es posible satisfacer estos requisitos sin necesidad de introducir una nueva abstracción. Basta con permitir a la hora de crear un proceso que se pueda especificar el grado de compartimiento entre los procesos involucrados: si los procesos pertenecen a la misma aplicación, se puede especificar un grado de compartimiento máximo, con lo que se consigue una interacción eficiente, puesto que los procesos pueden compartir el mapa de memoria, y unos cambios de contexto ligeros, ya que se limita considerablemente la información que se modifica en el cambio de proceso. Esta es la opción elegida en algunos sistemas operativos, como, por ejemplo, Linux.

En Linux se ha extendido la llamada al sistema de creación de procesos de manera que se pueda especificar si el proceso hijo comparte recursos con el padre o recibe un duplicado, que es el modo de operación convencional del `fork`. Los recursos se han clasificado en distintos grupos para facilitar la especificación de qué recursos se comparten y cuáles se duplican a la hora de usar la llamada de creación de procesos extendida (`clone`): el mapa de memoria, los ficheros abiertos, información adicional de ficheros (como el directorio actual o la máscara de creación de ficheros) e información sobre el manejo de las señales.

Con este esquema, se puede especificar el grado de compartimiento entre el padre y el hijo. De ahí surge el nombre de proceso de peso variable:

- Si se especifica que no hay compartimiento, el hijo recibirá un duplicado de los recursos, como en el caso de una creación de procesos convencional. Se trata de procesos que podemos catalogar como “pesados”, el uno con respecto al otro, en el sentido de que tanto su interacción como los cambios de contexto entre ellos tendrán un coste apreciable.
- Si se especifica que compartan todos los recursos, se pueden considerar que los procesos son “ligeros”, el uno en relación con el otro, pudiendo interaccionar e intercambiarse de manera eficiente.

Obsérvese que se podrían especificar grados intermedios de compartimiento de acuerdo con las necesidades de cada aplicación.

Para implementar este grado de compartimiento configurable es necesario que los recursos involucrados dejen de formar parte del BCP y pasen a ser objetos externos referenciados desde el BCP, como puede apreciarse en la figura 34. Cuando se crea un proceso, por cada uno de los tipos de recursos, se presentan dos alternativas:

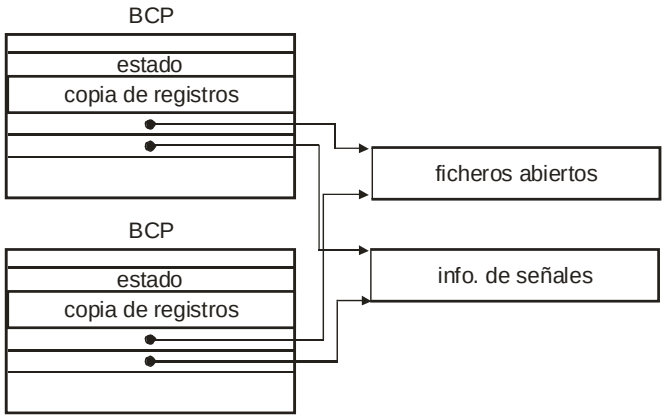


Figura 34 *Creación de un proceso ligero.*

- Si se comparte el recurso, el BCP del nuevo proceso apuntará al mismo objeto que el del padre.
- Si no se comparte, se reservará espacio para el objeto y se hará una copia del objeto del padre.

En la figura 35, se puede apreciar un ejemplo donde un proceso (BCP1) ha creado un proceso ligero (BCP2) y un proceso pesado (BCP3).

Aunque las aplicaciones pueden usar la nueva llamada al sistema de creación de procesos (`clone`), ésa no es la opción habitual. Linux ofrece una biblioteca de hilos estándar de UNIX (denominada *pthreads*). Cuando un programa solicita crear un hilo, la biblioteca hace uso de la función `clone` especificando que haya un compartimiento total de recursos. Es interesante resaltar que en las primeras versiones de la biblioteca de hilos era necesario crear un proceso adicional, que actuaba como gestor, para poder implementar la funcionalidad del estándar de hilos de UNIX, no logrando, además, en algunos aspectos respetar estrictamente dicho estándar. Versiones posteriores han ido incorporando en el sistema operativo nuevas funciones, lo que ha eliminado la necesidad del proceso gestor y ha logrado que se cumpla escrupulosamente con las especificaciones del estándar. Ha sido necesario que el sistema operativo sea consciente de alguna forma del concepto de hilo. Sin entrar en detalles, pero para dar una idea de la problemática, a continuación, se comenta brevemente esta cuestión.

El estándar de hilos de UNIX especifica que hay varias situaciones en las que hay que terminar todos los hilos de un proceso. Eso ocurre, por ejemplo, cuando se acaba el hilo

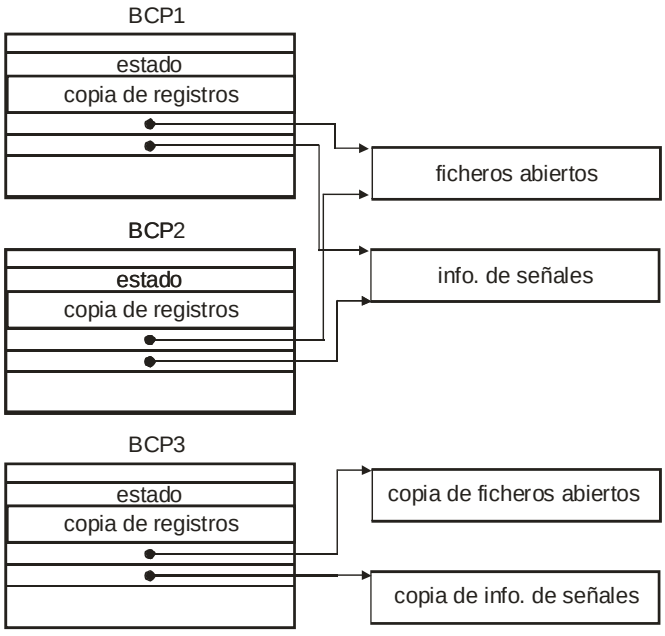


Figura 35 *Ejemplo con procesos pesados y ligeros.*

implícito del proceso o cuando se recibe una señal que no está siendo manejada ni ignorada, y cuyo comportamiento por defecto es matar al proceso. En versiones previas de Linux, el sistema operativo no sabía si dos procesos que compartían los recursos se correspondían con dos hilos del mismo programa o no (recuerde que un programa puede hacer uso directo de la llamada `clone`), lo que dificultaba implementar este requisito. Hubo que incorporar al sistema operativo el concepto de grupo de hilos, que incluye a todos los procesos ligeros que implementan los hilos de un determinado programa, y que permite tratarlos a todos de manera integral (por ejemplo, para abortarlos). Asimismo, se ha incorporado una nueva opción a la llamada `clone` para que permita especificar si el nuevo proceso forma parte del mismo grupo de hilos del padre o no.

7 Bibliografía

- M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.
- M. K. McKusick, K. Bostic, M. J. Karels y J. S. Quaterman, *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley, 1996.
- D. P. Bovet y M. Cesati, *Understanding the Linux Kernel*, 3ª edición, O'Reilly, 2005.
- M. E. Russinovich y D. A. Solomon, *Microsoft Windows Internals*, 4ª edición, Microsoft Press, 2005.
- R. P. Draves, et al., "Using Continuations to Implement Thread Management and Communication in Operating Systems", *Proceedings of the 13th ACM Symposium on Operating Principles*, Octubre de 1991.
- J. Liedtke, "Improving IPC by Kernel Design", *Proceedings of the 15th ACM Symposium on Operating Principles*, Noviembre de 1993.
- T. E. Anderson, et al., "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", *ACM Transactions on Computer Systems*, 10(1):53–79, Febrero de 1992.