# Introducción - Sistema operativo

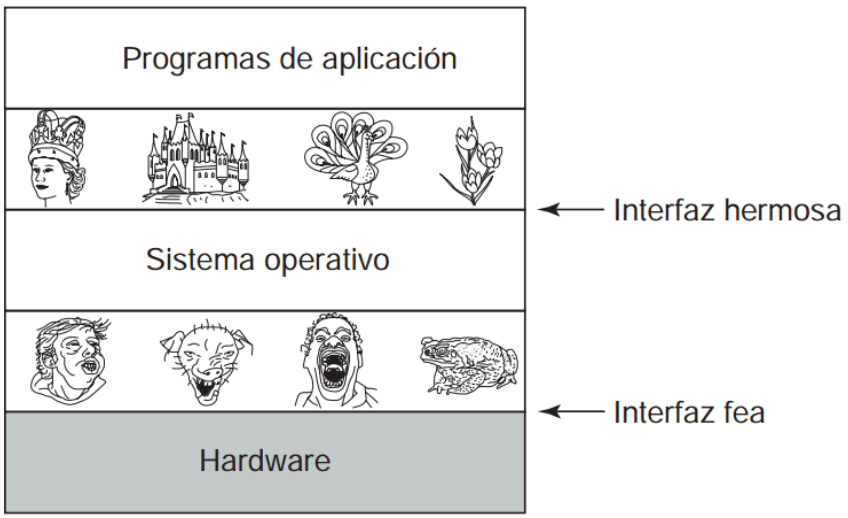
Un **sistema operativo** es un ***software*** que actúa como ***intermediario*** entre el ***usuario*** de una computadora y su ***hardware***. Como es un software, necesita procesador y memoria para ejecutarse.

El sistema operativo se encarga de gestionar el hardware y controlar la ejecución de procesos. Actúa como intermediario entre un usuario y el hardware, brindando una interfaz (un conjunto de directivas) que le permiten al usuario tener una visión más agradable y sencilla del sistema al darle una abstracción de alto nivel a los procesos sobre la administración del hardware.

## Perspectivas

**Existen dos perspectivas:**

- **Desde el usuario (de arriba hacia abajo):** el sistema operativo oculta el hardware y proporciona abstracciones más simples de manejar a los programas de aplicación.
- **Desde el sistema (de abajo hacia arriba):** el sistema operativo administra los recursos de hardware de uno o más procesos y provee un conjunto de servicios a los usuarios del sistema.




## Kernel (núcleo)

El kernel es, en un sentido estricto, el sistema operativo. Es una “porción de código” que se encuentra en la memoria principal y que se encarga de la administración de los recursos. Implementa **servicios** esenciales: como el manejo de la memoria y la CPU, la administración de procesos, la comunicación y concurrencia, y la gestión de E/S.

# Problemas que un sistema operativo debe evitar

El sistema operativo debe evitar que un proceso: se apropie de la CPU, intente ejecutar instrucciones de E/S, intente acceder a posiciones de memoria fuera de su espacio declarado.

Por ejemplo, un proceso no puede trabajar directamente con los dispositivos de E/S porque el sistema operativo tiene que encargarse de gestionarlos. Cuando un proceso hace un *read* para leer un disco, el proceso no puede ejecutar esta tarea por sí mismo y necesita ayuda del sistema operativo. Las instrucciones que realizan tareas de E/S son instrucciones *especiales* que hacen que el sistema operativo se encargue de manejarlas y acceden a un espacio de memoria que el proceso no puede acceder normalmente.

Pero hay un problema; el sistema operativo es un software que se ejecuta cuando obtiene la CPU, y si hay un proceso ejecutándose, entonces es el proceso el que estará usando a la CPU y no el sistema operativo. Entonces, ¿cómo hace el sistema operativo para evitar que un proceso que obtuvo la CPU se apropie de ella? La base del funcionamiento de estas cuestiones son las interrupciones.

# Apoyo del hardware

## Interrupciones

Cuando ocurre algún evento que requiera la atención del sistema operativo, el hardware encargado de procesarlo escribe directamente a una ubicación predeterminada de memoria la naturaleza de la solicitud (el **vector de interrupciones**) y, levantando una solicitud de interrupción, detiene el proceso que estaba siendo ejecutado. El sistema operativo entonces ejecuta su rutina de manejo de interrupciones (típicamente comienza grabando el estado de los registros del CPU y otra información relativa al estado del proceso desplazado) y posteriormente la atiende.

Se hace la distinción entre interrupciones y excepciones según su origen: una **interrupción** es generada por causas externas al sistema (un dispositivo requiere atención), mientras que una **excepción** es un evento generado por un proceso (una condición en el proceso que requiere la intervención del sistema operativo).

Las interrupciones pueden organizarse por prioridades, de modo que una interrupción de menor jerarquía no interrumpa a una más importante, dado que las interrupciones muchas veces indican que hay datos disponibles en algún buffer, el no atenderlas a tiempo podría llevar a la pérdida de datos. A su vez, el sistema operativo puede elegir ignorar (enmascarar) ciertas interrupciones, pero hay algunas que son no enmascarables.

*http://www.fdi.ucm.es/profesor/jjruz/web2/temas/Curso05_06/EC9.pdf*

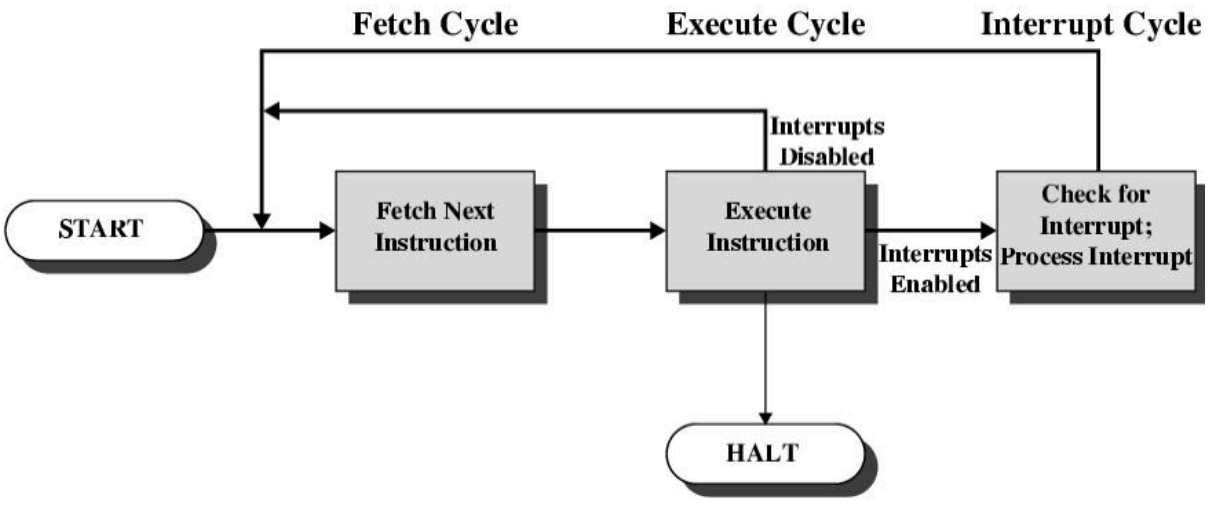
## Modos de ejecución

Los **modos de ejecución** definen limitaciones en el conjunto que se puede ejecutar en cada modo. Un bit en la CPU indica el modo actual. Existen dos modos:

- **Modo kernel:** puede ejecutar cualquier instrucción. Las instrucciones privilegiadas, que necesitan acceder a estructuras del kernel o ejecutar código que no es del proceso, deben ejecutarse en este modo.
- **Modo usuario:** puede ejecutar un conjunto reducido de instrucciones. En este modo, el proceso solo puede acceder a su espacio de direcciones, es decir, a las direcciones propias.

El kernel del sistema operativo se ejecuta en modo kernel, el resto del sistema operativo y los programas de usuario se ejecutan en modo usuario. El sistema arranca con el bit en modo kernel y cada vez que comienza a ejecutarse un proceso, este bit se debe poner en modo usuario.

Las **interrupciones** son la única manera de pasar a modo kernel, no es el proceso de usuario el que hace el cambio. Para obtener servicios del sistema operativo, un programa usuario debe lanzar una llamada al sistema (system call), la cual se atrapa en el kernel e invoca al sistema operativo. La instrucción *TRAP* cambia del modo usuario al modo kernel e inicia el sistema operativo. Cuando se ha completado el trabajo, el control se devuelve al programa de usuario en la instrucción que va después de la llamada al sistema.



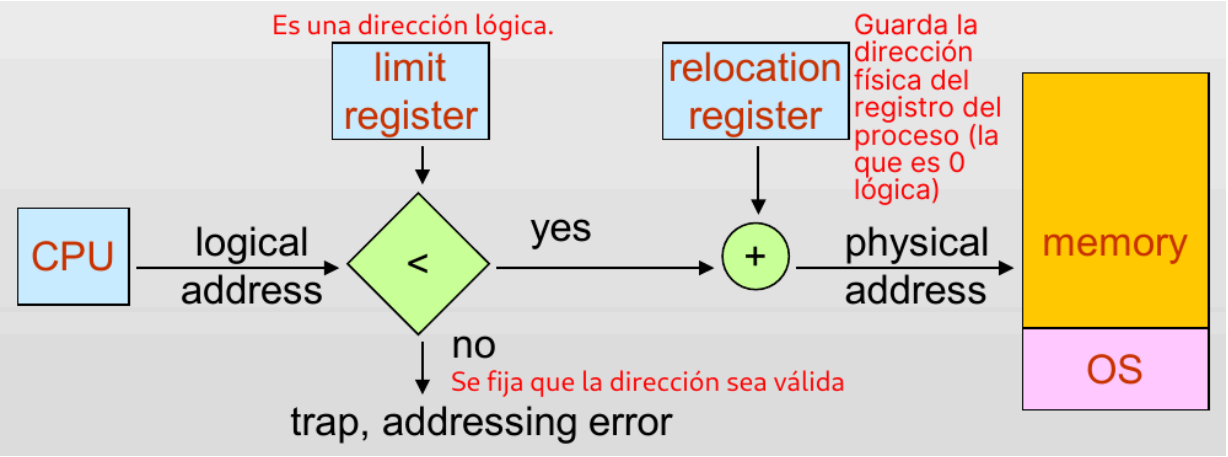

Al final del ciclo de ejecución de una instrucción se chequea
si hay alguna señal de interrupción para ser atendida

El bit de modo se encuentra en el registro PSW. El **registro PSW** (Program Status Word) es un registro especial de la CPU que almacena información sobre el estado actual de ejecución del proceso que se encuentra en la CPU.

## Protección de la memoria

Para proteger a la memoria se busca poner límites a las direcciones que puede utilizar un proceso. Se delimita el espacio de direcciones de un proceso usando, por ejemplo, un registro base y un registro límite. El kernel carga estos registros por medio de instrucciones privilegiadas (en modo kernel).

La memoria principal aloja al sistema operativo y a los procesos de usuario. El kernel debe proteger el espacio de direcciones de un proceso, evitando que un proceso acceda al espacio de otro.




Toda la información que está en los registros de la CPU está relacionada con el proceso que se está ejecutando en ese momento.

Cada vez que se intenta acceder a una posición de memoria, se verifica que este dentro de los límites a los que puede acceder el proceso (que este dentro del espacio de memoria del proceso). Esta verificación no es tarea del sistema operativo, sino que se apoya en la MMU. La MMU (unidad de manejo de memoria) convierte las direcciones de memoria lógicas en direcciones físicas y es la que genera un *trap* al sistema operativo en caso de que se intente acceder a una dirección inválida (addressing error).

## Protección de la E/S

Las instrucciones de E/S se definen como privilegiadas. Deben ejecutarse en modo Kernel y deberían ser gestionadas en el kernel del sistema operativo. Los procesos de usuario realizan tareas de E/S mediante llamadas a servicios del sistema operativo.

## Protección de la CPU

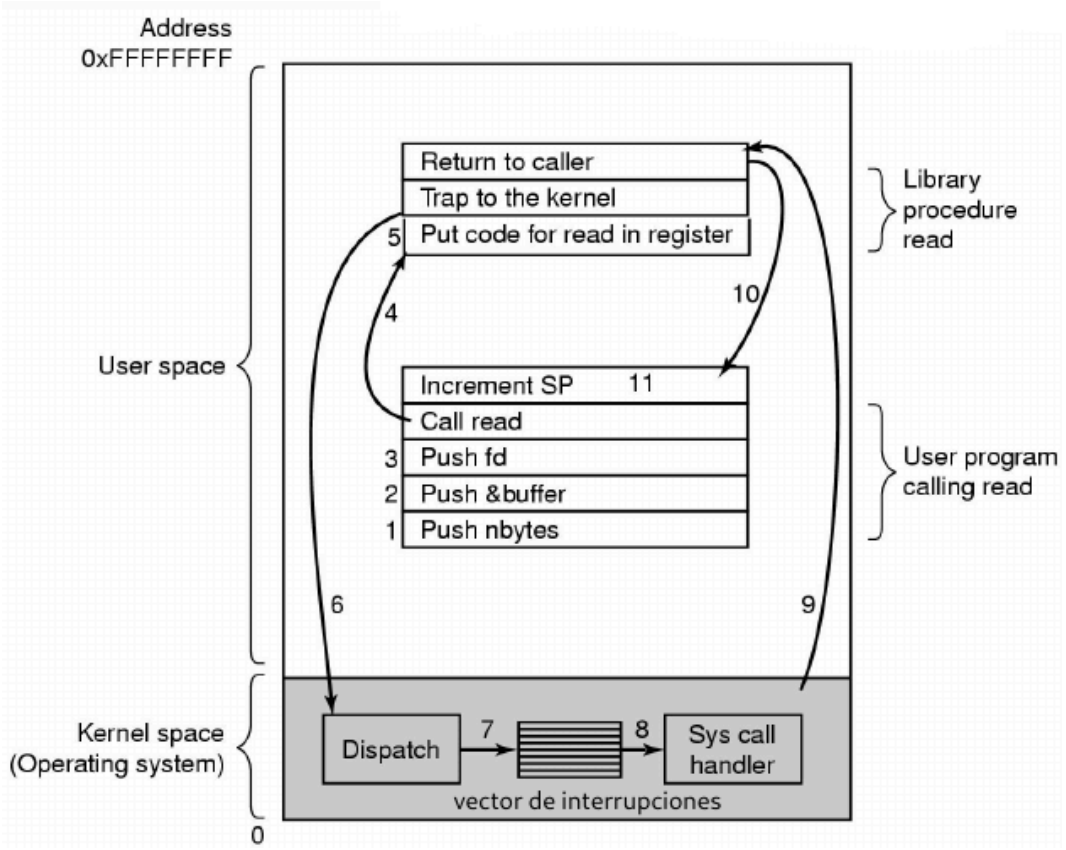
Se utiliza una **interrupción por clock** para evitar que un proceso se apropie de la CPU. Normalmente se implementa a través de un clock y un contador. El kernel le da un valor al contador, que se decrementa con cada tick del reloj, y que al llegar a cero genera un *trap* al sistema que puede expulsar al proceso.

# System calls

Es una interfaz que provee el sistema operativo para que los programas de usuario accedan a sus servicios. Los procesos solo pueden hacer operaciones dentro de su espacio de memoria, cuando el proceso necesita acceder a algo fuera de su espacio de memoria necesitan la intervención del sistema operativo.

## Llamada a una system call

```
count = read(file, buffer, nbytes);
```




**1 a 3:** Para llamar al procedimiento de biblioteca *read*, que es quien realmente hace la llamada al sistema, el programa llamador primero mete los parámetros en la pila.

**4:** llamada al procedimiento de biblioteca. Esta instrucción es la instrucción de llamada a procedimiento normal utilizada para llamar a todos los procedimientos.

**5:** El procedimiento de biblioteca (probablemente escrito en lenguaje ensamblador) coloca por lo general el número de la llamada al sistema en un lugar en el que el sistema operativo lo espera, como en un registro

**6:** Después ejecuta una instrucción *TRAP* para cambiar del modo usuario al modo kernel y empezar la ejecución en una dirección fija dentro del núcleo

**7:** El código de kernel que empieza después de la instrucción *TRAP* examina el número de llamada al sistema y después la pasa al manejador correspondiente de llamadas al sistema, por lo general a través de una tabla de apuntadores a manejadores de llamadas al sistema, indexados en base al número de llamada al sistema

**8:** En ese momento se ejecuta el manejador de llamadas al sistema

**9:** Una vez que el manejador ha terminado su trabajo, el control se puede regresar al procedimiento de biblioteca que está en espacio de usuario

**10:** Luego este procedimiento regresa al programa de usuario en la forma usual en que regresan las llamadas a procedimientos

**11:** Para terminar el trabajo, el programa de usuario tiene que limpiar la pila, como lo hace después de cualquier llamada a un procedimiento

# Procesos

Un **programa** es un archivo ejecutable que tiene una organización específica que le permite al SO crear una imagen de proceso. Un **proceso** es una abstracción de un programa en ejecución, es una entidad que reside en memoria y se le asignan ciclos de la CPU.

Un *programa* es estático, no tiene program counter y existe desde que se edita hasta que se borra. Por otro lado, un *proceso* es dinámico, tiene program counter y su ciclo de vida comprende desde que solicita ejecutarse hasta que termina.

## Componentes de un proceso

Un proceso para poder ejecutarse incluye como mínimo:

- Sección de código (texto)
- Sección de datos (variables globales)
- Stack(s) (datos temporarios: parámetros, variables temporales y direcciones de retorno).

### Stacks

Un proceso cuenta con uno o más stack, en general, modo usuario y modo kernel. Se crean automáticamente y su medida se ajusta en run-time.

Está formado por **stack frames** que son agregados (push) al llamar a una rutina y eliminados (pop) cuando se retorna de ella. Un *stack frame* tiene los parámetros de la rutina (variables locales), y datos necesarios para recuperar el stack frame anterior (el contador de programa y el valor del stack pointer en el momento del llamado).

### PCB (Process Control Block)

Es una estructura de datos asociada a un proceso y es lo primero que se genera cuando se crea un proceso y lo último que se borra cuando termina. Existe una por proceso y contiene información asociada a cada proceso.

- **Identificador del proceso (PID):** un identificador único asociado al proceso.
- **Identificador del padre del proceso (PPID):** identificación del proceso padre que lo creó.
- **Estado:** si el proceso está actualmente corriendo, está en el estado en ejecución.
- **Prioridad:** nivel de prioridad relativo al resto de procesos.
- **Contador de programa:** la dirección de la siguiente instrucción del programa que se ejecutará.
- **Punteros a memoria:** incluye los punteros al código de programa y los datos asociados a dicho proceso, además de cualquier bloque de memoria compartido con otros procesos.
- **Datos de contexto:** estos son datos que están presenten en los registros del procesador cuando el proceso está corriendo.

- **Información de estado de E/S:** incluye las peticiones de E/S pendientes, dispositivos de E/S asignados a dicho proceso, una lista de los ficheros en uso por el mismo, etc.
- **Información de auditoría:** puede incluir la cantidad de tiempo de procesador y de tiempo de reloj utilizados, así como los límites de tiempo, registros contables, etc.
- Identificación del usuario que lo disparó.
- Grupo que lo disparó, si hay estructura de grupos.
- Desde qué terminal y quién lo ejecutó, en ambientes multiusuario.
- Planificación (estado, prioridad, tiempo consumido, etc.)
- Ubicación (representación) en memoria.

La PCB contiene suficiente información para que sea posible interrumpir el proceso cuando está corriendo y posteriormente restaurar su estado de ejecución como si no hubiera habido interrupción alguna. La PCB es la herramienta clave que permite al sistema operativo dar soporte a múltiples procesos y proporcionar multiprogramación. Cuando un proceso se interrumpe, los valores actuales del contador de programa y los registros del procesador (datos de contexto) se guardan en los campos correspondientes de la PCB y el estado del proceso se cambia a cualquier otro valor, como *esperando* o *listo*. De esta forma, el sistema operativo puede realizar un cambio de contexto y ejecutar a otro proceso.

### Espacio de direcciones de un proceso

Es el conjunto de direcciones de memoria que ocupa el proceso (lo que puede manipular: stack, text y datos). No incluye su PCB o tablas asociadas, ya que son estructuras del sistema operativo.

## El contexto de un proceso

El **contexto de un proceso** incluye toda la información que el sistema operativo necesita para administrar el proceso y la CPU para ejecutarlo correctamente. Son partes del contexto: la PCB, los registros de la CPU (incluido el contador de programa), la prioridad del proceso, etc.

### Descripción de procesos en UNIX

Un proceso en UNIX es un conjunto de estructuras de datos que proporcionan al sistema operativo toda la información necesasria para manejar y activar los procesos. Todas estas estructuras forman la imagen (o contexto) de un proceso y está dividida en tres partes:

- **Contexto a nivel de usuario:** contiene los elementos básicos de un programa de usuario que se pueden generar directamente desde un archivo ejecutable. Se divide en tres áreas:
    - **Área de texto:** es de solo lectura y contiene las instrucciones del programa.
    - **Área de pila:** cuando el proceso está en ejecución, el procesador utiliza el área de pila de usuario para gestionar las llamadas a procedimientos, sus parámetros y sus retornos.

    - **Área de memoria compartida:** es un área de datos que se comparte con otros procesos. Solo existe una única copia física del área de memoria compartida, pero, gracias a la memoria virtual, se presenta a cada uno de los procesos como una región de memoria común dentro de su espacio de direcciones.

- **Contexto de registros:** contiene la información de estado del procesador de un proceso que se está ejecutando.
- **Contexto a nivel de sistema:** contiene el resto de información que necesita el sistema operativo para manejar al proceso.
    - El **área de usuario (área U)** contiene información adicional del proceso que necesita el kernel cuando está ejecutando el contexto del proceso y para el *swapping.*

**Tabla 3.10.** Imagen de un proceso UNIX.

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

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

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

**Tabla 3.12.** Área U de UNIX.

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

## Cambio de contexto (context switch)

El **cambio de contexto** es el proceso que ocurre cuando se cambia el proceso que se está ejecutando en la CPU por otro. Este proceso requiere tiempo de la CPU para:

- Guardar el contexto del proceso actual en la memoria. Esto incluye los registros de la CPU, la pila y el estado del sistema.
- Cargar el estado del nuevo proceso en la memoria. Esto incluye los registros de la CPU, la pila y el estado del sistema.
- Cambiar el puntero de instrucción a la dirección de inicio del nuevo proceso.

Durante el cambio de contexto, la CPU no se dedica a ejecutar el código del proceso, por lo que se considera tiempo improductivo. El tiempo de context switch puede ser significativo, especialmente en sistemas con muchos procesos.

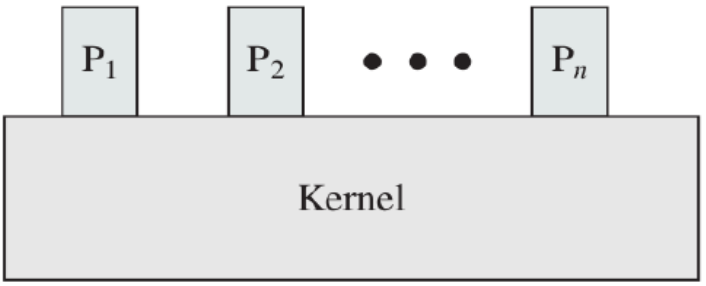
# Sobre el kernel del sistema operativo

Si bien el kernel necesita de memoria y CPU para poder ejecutarse, no se considera un proceso porque este concepto solo se asocia a programas de usuario y el kernel se ejecuta como una entidad independiente en modo privilegiado. Existen diferentes enfoques de diseño.

## Enfoque 1 - El kernel como entidad independiente

Fue una arquitectura utilizada por los primeros sistemas operativos. En este enfoque el kernel se ejecuta fuera de todo proceso. El kernel tiene su propia región de memoria, su propio stack y, cuando finaliza su actividad, le devuelve el control al proceso (o a otro diferente).

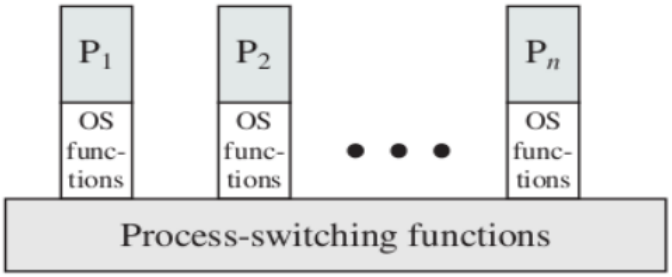
Cuando un proceso es “interrumpido” o realiza una system call, el contexto del proceso se salva y el control pasa al kernel del sistema operativo. Esto genera mucho tiempo improductivo en la CPU al tener que hacer cambios de contexto.




## Enfoque 2 - El kernel “dentro” del proceso

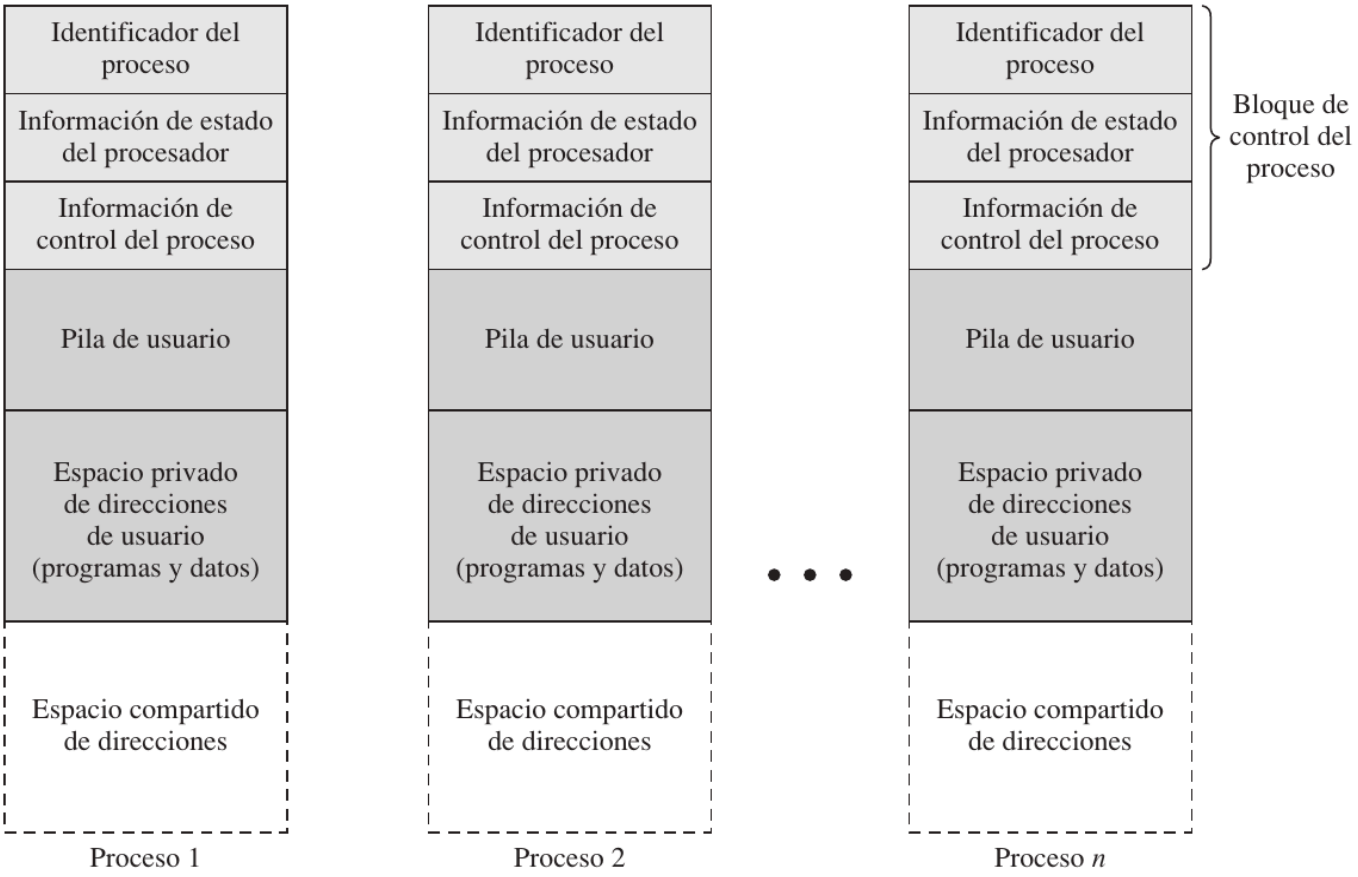
En este enfoque el kernel se encuentra dentro del espacio de direcciones de cada proceso, así, el kernel se ejecuta en el mismo contexto que el proceso de usuario que se estaba ejecutando.

El kernel se puede ver como una colección de rutinas que el proceso utiliza. Dentro de un proceso se encuentra el código del programa (user) y el código de los módulos de software del sistema operativo (kernel). Cada proceso tiene su propio stack (ino en modo usuario y otro en modo kernel), y el proceso se ejecuta en modo usuario y el kernel del sistema operativo se ejecuta en modo kernel.



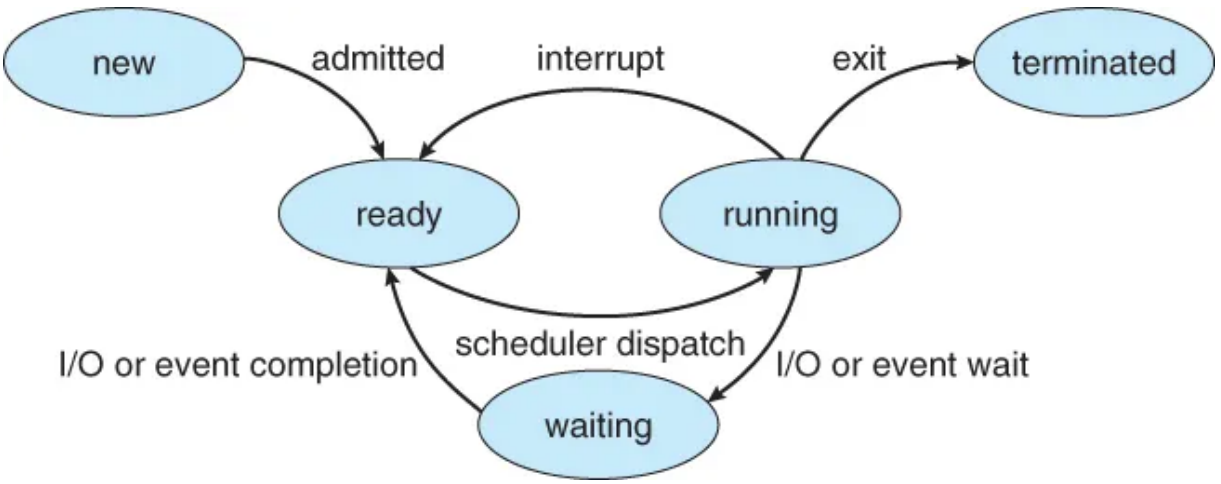

El código del kernel es compartido por todos los procesos.

Cada interrupción (incluyendo las system call) son atendidas en el contexto del proceso que se encontraba en ejecución pero en modo kernel. Si el sistema operativo determina que el proceso debe seguir ejecutándose, luego de atender la interrupción, cambia a modo usuario y devuelve el control. Es más económico y perforante.




## Estados de un proceso

Al ejecutar un programa, lo primero que hace el SO es crear una PCB con información para ese proceso (recién cuando se crea la PCB es que podemos hablar de proceso). Durante su ciclo de vida, un proceso pasa por diferentes estados:

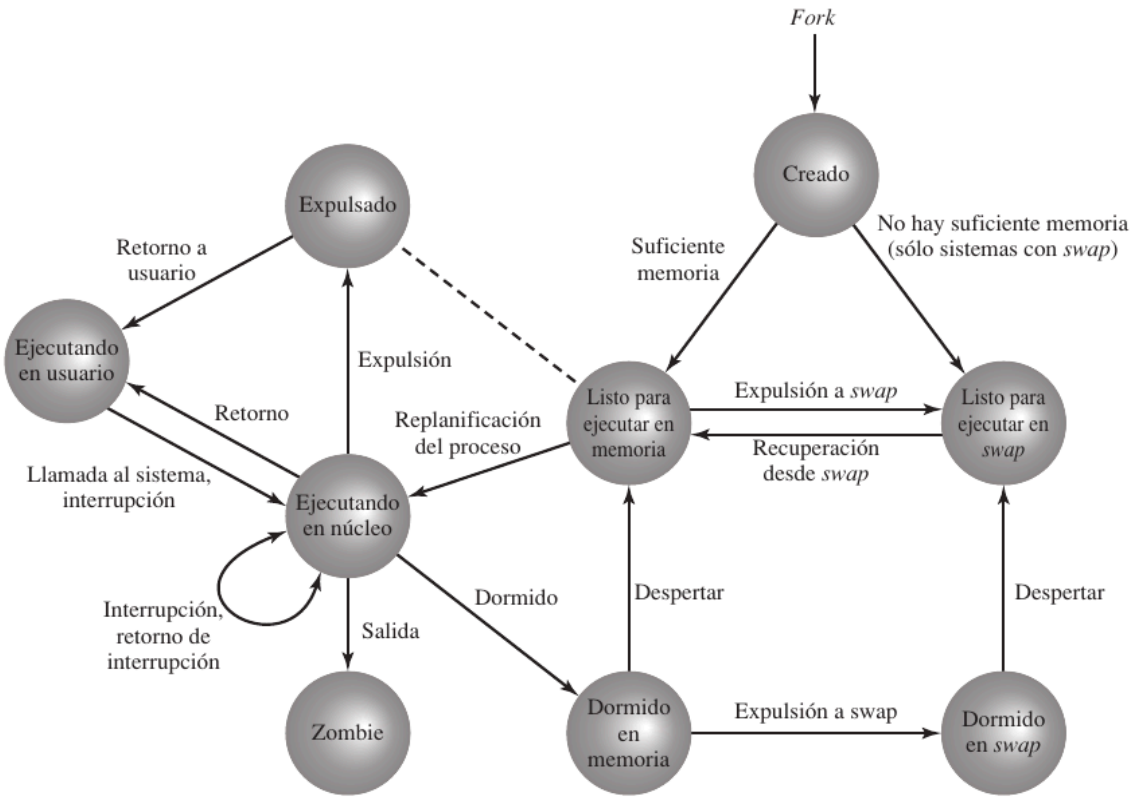


- **Nuevo (new):** un proceso que se acaba de crear y que aún no ha sido admitido en el grupo de procesos ejecutables por el sistema operativo. Típicamente, se trata de un nuevo proceso que no ha sido cargado en memoria principal, aunque su PCB si ha sido creada.
    - El proceso tiene una PCB y sus estructuras asociadas, pero todavía no está cargado en memoria
- **Listo (ready):** el proceso está a la espera a que se le asigne un procesador.
    - El proceso está cargado en memoria, pero no está usando CPU y está compitiendo por usarla.
- **Ejecutando (running):** el proceso está actualmente en ejecución
- **Esperando (waiting):** el proceso está esperando que se produzca un suceso (como la terminación de una operación de E/S o la recepción de una señal) y no se puede ejecutar hasta que el suceso se cumpla.
    - El proceso no está usando CPU ni está compitiendo por usarla
- **Terminado (terminated):** un proceso que ha sido liberado del grupo de procesos ejecutables por el sistema operativo, debido a que ha sido detenido o que ha sido abortado por alguna razón.
    - Este estado está desde que el proceso termina de ejecutarse (exit) hasta que el SO puede borrar la PCB, después de eso no hay más estados porque ya no hay más proceso.

## Estados de los procesos en UNIX

Los sistemas UNIX siguen el enfoque del kernel “dentro” de los procesos. Los procesos en Unix se dividen en dos categorías:

- **Procesos del sistema:** se ejecutan en ***modo kernel*** y ejecutan código del sistema operativo para realizar tareas administrativas o funciones internas.
- **Procesos de usuario:** se ejecutan en ***modo usuario*** para ejecutar programas y utilidades, y en ***modo kernel*** para ejecutar instrucciones que pertenecen al kernel. Un proceso de usuario entra en modo kernel por medio de una llamada al sistema (system call).




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

# Jerarquías de procesos

En **UNIX**, un proceso puede tener cero, uno o más hijos; formando un árbol de procesos. Un proceso y todos sus hijos, junto con sus posteriores descendientes, forman un grupo de procesos y los procesos no pueden desheredar a sus hijos.

En contraste, **Windows** no tiene un concepto de una jerarquía de procesos. Todos los procesos son iguales. La única sugerencia de una jerarquía de procesos es que, cuando se crea un proceso, el padre recibe un indicador especial, un *token* (llamado manejador) que puede utilizar para controlar al hijo. Sin embargo, tiene la libertad de pasar este indicador a otros procesos, con lo cual invalida la jerarquía.

# Creación de procesos

Hay cuatro eventos principales que provocan la creación de procesos:

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

**Procesos en primer plano:** interactúan con los usuarios (humanos) y realizan trabajos para ellos.

**Procesos en segundo plano:** no están asociados con usuarios específicos, sino con una función específica. Los procesos que permanecen en segundo plano para manejar ciertas actividades como correo electrónico, servidores web, impresiones, etc. se conocen como ***daemons*** (demonios).

## Cómo se crea un proceso

Para crear un proceso hay que hacer que un proceso *existente* (que puede ser un proceso de usuario en ejecución, del sistema invocado mediante teclado/ratón, o de administrador de lotes) ejecute una llamada al sistema para crear el *nuevo* proceso.  Esta llamada al sistema indica al sistema operativo que cree un proceso y le indica, directa o indirectamente, qué programa debe ejecutarlo.

### Creación de procesos en UNIX

En **UNIX** solo hay una llamada al sistema para crear un proceso: **`fork`**. Esta llamada crea un clon exacto del proceso que hizo la llamada. Después de `fork`, los dos procesos (padre e hijo) tienen la misma imagen de memoria, las mismas cadenas de entorno y los mismos archivos abiertos. Eso es todo. Por lo general, el proceso hijo después ejecuta a **`execve`**, o una llamada al sistema similar, para cambiar su imagen de memoria y ejecutar un nuevo programa. El proceso de creación se divide en dos pasos para permitir al hijo manipular sus descriptores de archivo después del `fork`, pero antes del `execve`, para así poder redirigir la entrada, la salida y el error estándar.

`fork` realiza las siguientes funciones en modo núcleo, dentro del proceso padre:

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

Cuando el kernel completa estas funciones, puede realizar cualquiera de las siguientes acciones:

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

### Creación de procesos en Windows

En **Windows,** la función de Win32, **`CreateProcess`**, maneja la creación de procesos y carga el programa correcto en el nuevo proceso. Esta llamada tiene 10 parámetros, 9 que sirven para especificar la forma en la que se debe crear el proceso y 1 puntero a una estructura donde se devuelve al proceso creador la información acerca del proceso recién creado.

Además de CreateProcess, Win32 tiene cerca de 100 funciones más para administrar y sincronizar procesos y temas relacionados.

### Comparativa de la creación de procesos en UNIX y Windows

Tanto en UNIX como en Windows, una vez que se crea un proceso, el padre y el hijo tienen sus propios espacios de direcciones distintos. Si cualquiera de los procesos modifica una palabra en su espacio de direcciones, esta modificación no es visible para el otro proceso.

En **UNIX**, el espacio de direcciones inicial del hijo es una copia del padre, pero en definitiva hay dos espacios de direcciones distintos involucrados; no se comparte memoria en la que se pueda escribir (algunas implementaciones de UNIX comparten el texto del programa entre los dos, debido a que no se puede modificar). Sin embargo, es posible para un proceso recién creado compartir algunos de los otros recursos de su creador, como los archivos abiertos.

En **Windows**, los espacios de direcciones del hijo y del padre son distintos desde el principio.

## Terminación de procesos

Todos los sistemas deben proveer mecanismos para que los procesos indicquen su finalización, o que ham completado su tarea.

Ante un ***exit***, se retorna el control al sistema operativo. El proceso padre puede esperar a recibir un código de retorno; esto se usa cuando el padre necesita esperar a los hijos.

Un proceso padre puede terminar la ejecución de sus hijos (***kill***). Esto puede ser por dos motivos:

- La tarea asignada al hijo se terminó.
- El padre terminó su ejecución.

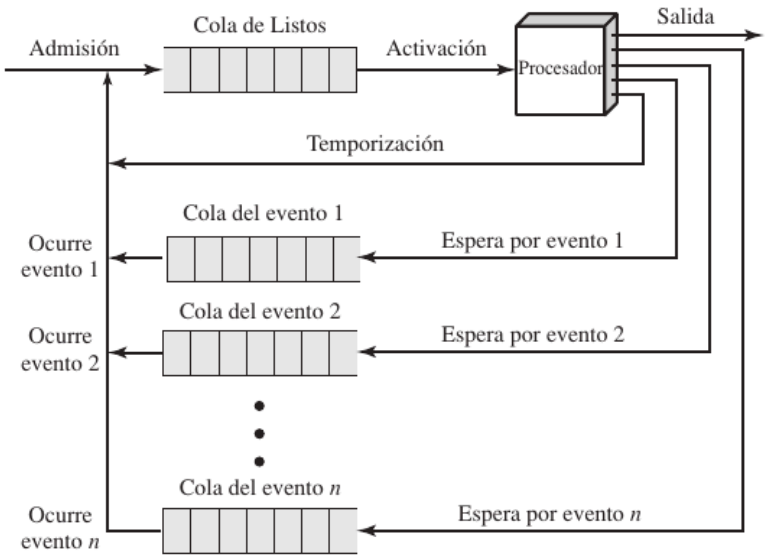
## Procesos cooperativos e independientes

- **Independiente:** el proceso no afecta ni puede ser afectado por la ejecución de otros procesos. No comparte ningún tipo de dato.
- **Cooperativo:** afecta o es afectado por la ejecución de toros procesos en el sistema. Estos procesos sirven para:
    - Compartir información
    - Acelerar el cómputo, separar una tarea en sub-taresas que cooperan ejecutándose paralelamente.
    - Planificar de manera tal que se puede ejecutar en paralelo.

## Colas de planificación de procesos

Para realizar la planificación, el sistema operativo enlaza las PCB de los procesos en colas siguiendo un orden determinado.

- **Cola de trabajos o procesos:** contiene todas las PCB de los procesos en el sistema.
- **Colas de procesos listos:** contiene todas las PCB de procesos residentes en memoria principal esperando a ejecutarse.
- **Cola de dispositivos:** PCB de procesos esperando por un dispositivo de E/S.




## Módulos de planificación

Cuando dos procesos compiten por la CPU al mismo tiempo, hay que decidir cuál proceso se va a ejecutar a continuación. El módulo del sistema operativo que realiza esa decisión se conoce como **planificador de procesos** y el algoritmo que se utiliza se conoce como **algoritmo de planificación.** Los módulos de planificación se ejecutan en la creación/terminación de procesos, eventos de sincronización o de E/S, finalización de lapso de tiempo, etc.

Existen distintos tipos de planificación:

- **Planificación a largo plazo (long term scheduler):** se realiza cuando se crea un nuevo proceso. Hay que decidir si se añade un nuevo proceso al conjunto de los que están activos actualmente; es decir, controla el grado de multiprogramación (la cantidad de procesos en memoria).
- **Planificación a medio plazo (medium term scheduler):** es parte de la función de intercambio (*swapping function*). Hay que decidir si se añade un proceso a los que están al menos parcialmente en memoria y que, por tanto, están disponibles para su ejecución. Esta planificación puede disminuir el grado de multiprogramación al realizar el *swapping* (intercambio) entre la memoria principal y el disco.
- **Planificación a corto plazo (short term scheduler):** determina cuál de los procesos listos para ejecutar es ejecutado.

En términos de frecuencia de ejecución, el ***long term scheduler*** se ejecuta con relativamente poca frecuencia y toma la decisión de grano grueso de admitir o no un nuevo proceso y qué proceso admitir. El ***medium term scheduler*** se ejecuta más frecuentemente para tomar decisiones de intercambio. El ***short term scheduler***, conocido también como ***dispatcher*** *(activador)*, ejecuta mucho más frecuentemente y toma las decisiones de grano fino sobre qué proceso ejecutar el siguiente.

Otros módulos son el ***dispatcher*** y el ***loader***. Estos pueden no existir como módulos separados de los schedulers, pero su función debe cumplirse.

- El **dispatcher** es el módulo del SO que se encarga de hacer los cambios de contexto, es decir, "despacha" el proceso elegido por el *short term scheduler* en el procesador.
- El **loader** carga en memoria el proceso elegido por el *long term scheduler.*

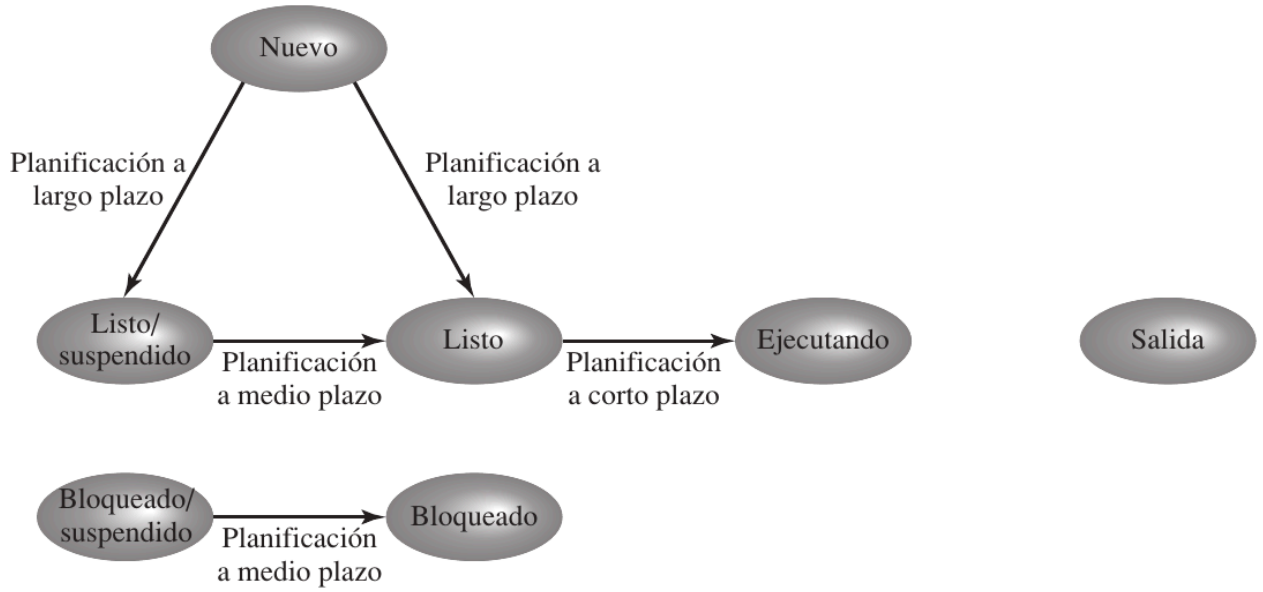



**Figura 9.1.** Planificación y transiciones de estado de los procesos.

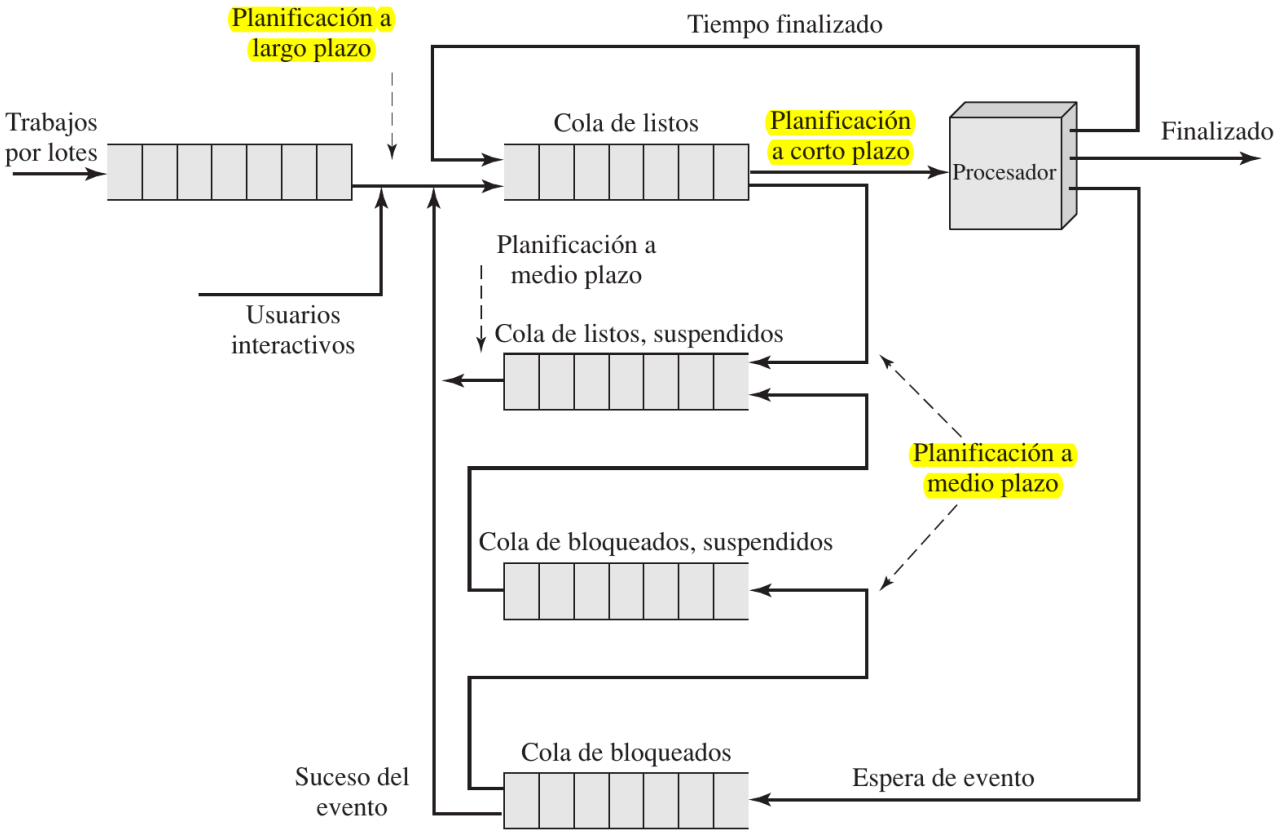



**Figura 9.3.** Diagrama de encolamiento para la planificación.

## Long term scheduler

El planificador a largo plazo determina qué programas se admiten en el sistema para su procesamiento. De esta forma, se controla el grado de multiprogramación. Una vez admitido, un trabajo o programa de usuario se convierte en un proceso y se añade a la cola del planificador a corto plazo. En algunos sistemas, un proceso de reciente creación comienza en la zona de intercambio, en cuyo caso se añaden a la cola del planificador a medio plazo.

La decisión de cuándo crear un nuevo proceso se toma dependiendo del grado de multiprogramación deseado. Cuanto mayor sea el número de procesos creados, menor será el porcentaje de tiempo en que cada proceso se pueda ejecutar (es decir, más procesos compiten por la misma cantidad de tiempo de procesador). De esta forma, el planificador a largo plazo puede limitar el grado de multiprogramación a fin de proporcionar un servicio satisfactorio al actual conjunto de procesos. Cada vez que termina un trabajo, el planificador puede decidir añadir uno o más nuevos trabajos. Además, si la fracción de tiempo que el procesador está ocioso excede un determinado valor, se puede invocar al planificador a largo plazo.

## Medium term scheduler

La planificación a medio plazo es parte de la función de intercambio. Con frecuencia, la decisión de intercambio se basa en la necesidad de gestionar el grado de multiprogramación. En un sistema que no utiliza la memoria virtual, la gestión de la memoria es también otro aspecto a tener en cuenta. De esta forma, la decisión de meter un proceso en la memoria, tendrá en cuenta las necesidades de memoria de los procesos que están fuera de la misma.

## Short term scheduler

El planificador a corto plazo se invoca siempre que ocurre un evento que puede conllevar el bloqueo del proceso actual y que puede proporcionar la oportunidad de expulsar al proceso actualmente en ejecución en favor de otro. Algunos ejemplos de estos eventos son:

- Interrupciones de reloj.
- Interrupciones de E/S.
- Llamadas al sistema.
- Señales (por ejemplo, semáforos).

## Transiciones

- **New-Ready:** Por elección del scheduler de largo plazo (carga en memoria).
- **Ready-Running:** Por elección del scheduler de corto plazo (asignación de CPU).
- **Running-Waiting:** el proceso “se pone a dormir”, esperando por un evento.
- **Waiting-Ready:** Terminó la espera y compite nuevamente por la CPU.

# Comportamiento de los procesos

Casi todos los procesos alternan ráfagas de E/S con cálculos de peticiones de E/S. La E/S es cuando un proceso entra al estado bloqueado en espera de que un dispositivo externo complete su trabajo. A partir de esto podemos distinguir dos tipos de procesos:

- **CPU bound:** el proceso se pasa la mayor parte del tiempo utilizando la CPU, *fig. 2-38 (a)*.
    - Ráfagas de CPU *largas* y esperas *infrecuentes* a la E/S.
- **I/O bound:** el proceso se pasa la mayor parte del tiempo esperando por E/S, *fig. 2-38 (b)*.
    - Ráfagas de CPU *cortas* y esperas *frecuentes* a la E/S.

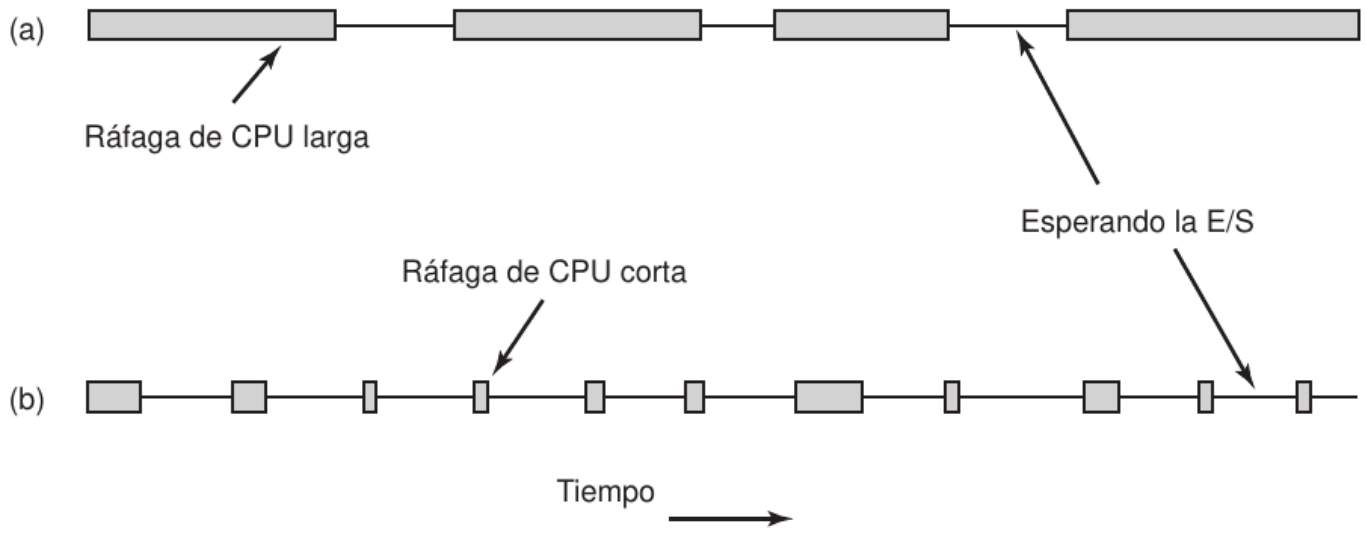



**Figura 2-38.** Las ráfagas de uso de la CPU se alternan con los periodos de espera por la E/S. (a) Un proceso ligado a la CPU. (b) Un proceso ligado a la E/S.

La velocidad de la CPU es mucho más rápida que la velocidad de los dispositivos de E/S, por lo que los procesos tienden a estar más ligados a la E/S.

# Planificación

Los algoritmos de planificación se pueden dividir en dos categorías:

- **Non-preemptive (no apropiativo):** una vez que un proceso está en estado de ejecución, continúa hasta que termina o se bloquea por algún evento. En este tipo de algoritmos, no hay decisiones de planificación durante las interrupciones del reloj.
- **Preemptive (apropiativo):** el proceso en ejecución puede ser interrumpido y llevado a la cola de listos. Esto da mayor overhead, pero mejor servicio, y evita que los procesos monopolicen el procesador.

## Categorías de los algoritmos de planificación

Distintos entornos tienen distintos objetivos, por lo que requieren distintos algoritmos de planificación. El planificador no debe optimizar lo mismo en todos los sistemas. Algunos de los entornos son:

1. **Sistemas de procesamiento por lotes (batch):** no hay usuarios que esperen una respuesta rápida a una petición corta, por lo que son aceptables los *algoritmos no apropiativos* (o apropiativos con largos periodos para cada proceso). Este método reduce la conmutación de procesos y por ende, mejora el rendimiento.

   **Metas:**
   - **Rendimiento:** maximizar el número de trabajos por hora.
   - **Tiempo de retorno:** minimizar los tiempos entre comienzo y finalización.
   - **Uso de la CPU:** mantener la CPU ocupada el mayor tiempo posible.
   - El tiempo en espera se puede ver afectado.

   **Algoritmos:**
   - **FCFS:** first come first served.
   - **SJF:** shortest job first.

2. **Sistemas interactivos:** la *apropiación* es esencial para evitar que un proceso acapare la CPU y niegue el servicio a los demás. Los servidores también se encuentran en esta categoría.

   **Metas**
   - **Tiempo de respuesta:** responder a peticiones con rapidez.
   - **Proporcionalidad:** cumplir las expectativas de los usuarios.

   **Algoritmos**
   - **RR:** round robin
   - Prioridades
   - Colas multinivel
   - **SRTF:** shortest remaining time first.

3. **Sistemas de tiempo real:** la apropiación a veces no es necesaria debido a que los procesos saben que no pueden ejecutarse durante periodos extensos.

    **Metas**

    - **Cumplir con los plazos:** evitar perder datos.
    - **Predictibilidad:** Evitar la degradación de la calidad en los sistemas multimedia.

**Metas comunes a todos los sistemas:**

- **Equidad:** otorgar a cada proceso una parte justa de la CPU.
- **Aplicación de políticas:** verificar que se lleven a cabo las políticas establecidas.
- **Balance:** mantener ocupadas todas las partes del sistema.

# Política versus mecanismo

Existen situaciones en las que es necesario que la planificación se comporte de manera diferente. El algoritmo de planificación debe estar parametrizado de forma tal que los procesos/usuarios puedan indicar los parámetros para indicar la planificación.

El Kernel implementa el mecanismo y el usuario/proceso utiliza los parámetros para determinar la política. Por ejemplo, un algoritmo de planificación por prioridades y una system call que permite modificar la prioridad de los procesos que él crea (comando *nice*).

# Algoritmos de planificación

## FCFS - First-Come, First-Served

Es un algoritmo *no apropiativo* donde la CPU se asigna a los procesos en el orden en el que se solicitaron.

Hay una sola cola de procesos listos. A medida que van entrando otros trabajos se colocan al final de la cola. Si el proceso en ejecución se bloquea, el primer proceso en la cola se ejecuta a continuación. Cuando un proceso bloqueado pasa al estado listo, al igual que un proceso recién llegado, se coloca al final de la cola.

En principio el algoritmo es equitativo, ya que no favorece a ningún tipo de proceso, pero podríamos decir que los procesos CPU bound terminarán en su primera ráfaga (ya que no son interrumpidos por E/S), mientras que los I/O bound no.

## SJF - Shortest Job First

Es un algoritmo *no apropiativo*, que supone que los tiempos de ejecución de los procesos se conocen de antemano. Cuando hay varios procesos de igual importancia esperando ser iniciados en la cola de entrada, el planificador selecciona el trabajo más corto primero. Este algoritmo solo

es óptimo cuando todos los trabajos están disponibles al mismo tiempo. Los procesos largos pueden sufrir *starvation* (inanición, sin uso de CPU).


**Figura 2-40.** Un ejemplo de planificación tipo el trabajo más corto primero. (a) Ejecución de cuatro trabajos en el orden original. (b) Ejecución de cuatro trabajos en el orden del tipo "el trabajo más corto primero".

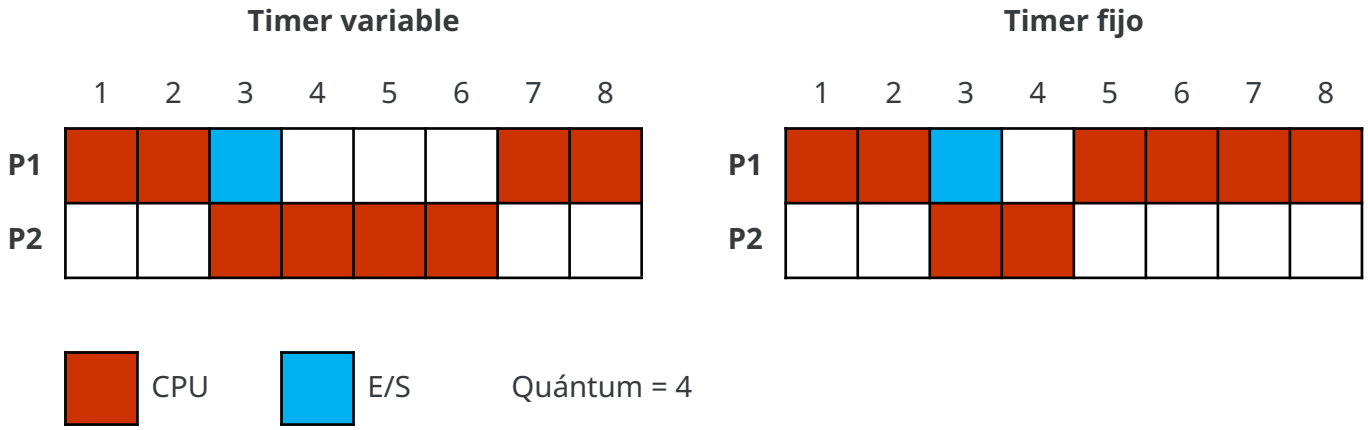
## RR - Round robin

Es un algoritmo *apropiativo* donde a cada proceso se le asigna un intervalo de tiempo (**quántum**) durante el cuál se le permite ejecutarse. Si el proceso se sigue ejecutando al finalizar este tiempo, la CPU es apropiada para dársela a otro proceso y el proceso original se coloca al final de la lista. Si el proceso se bloquea o termina antes de que haya transcurrido el quántum, también se realiza la conmutación de la CPU.

En cada conmutación se debe realizar un cambio de contexto, lo que genera tiempos improductivos de la CPU. Si utilizamos un **quántum chico** se desperdicia mucho tiempo de CPU en realizar los cambios de contexto (context switching overhead); pero si se establece un **quántum largo**, se puede producir una mala respuesta a las peticiones interactivas cortas.

Existen dos variaciones:

- **Timer variable:** el contador se restablece cada vez que un proceso es asignado a la CPU.
- **Timer fijo:** el contador se restablece solo cuando llega a 0. El quántum se comparte entre los procesos.



## Por prioridad

La planificación round robin asume que todos los procesos tienen la misma importancia. En la **planificación por prioridad**, a cada proceso se le asigna una prioridad y se ejecuta al proceso ejecutable con la prioridad más alta.

Para evitar que los procesos con alta prioridad se ejecuten indefinidamente, el scheduler puede reducir la prioridad del proceso en ejecución en cada interrupción del reloj, hasta que otro proceso tenga una prioridad más alta y se realice una conmutación. Otra alternativa es que a cada proceso se le puede asignar un quántum máximo de tiempo que tiene permitido ejecutarse y cuando lo alcanza, el siguiente proceso con la prioridad más alta recibe la oportunidad de ejecutarse.

A las prioridades se les pueden asignar procesos en forma estática o dinámica. El comando *nice* permite establecer la prioridad de un proceso (**estática**). Para asignar las prioridades de forma **dinámica** se podría establecer la prioridad a 1/*f*, donde *f* es la fracción del último quántum que se utilizó en un proceso. Así, un proceso que solo utilizó 1 ms de su quántum de 50 ms obtendría una prioridad de 50, mientras que un proceso que se ejecutó durante 25 ms recibirá la prioridad 2 y un proceso que utilizó todo su quántum recibirá la prioridad 1.

## Múltiples colas

Se pueden agrupar los procesos en clases de prioridad y utilizar la planificación por prioridad entre las clase, pero la planificación round robin dentro de cada clase.

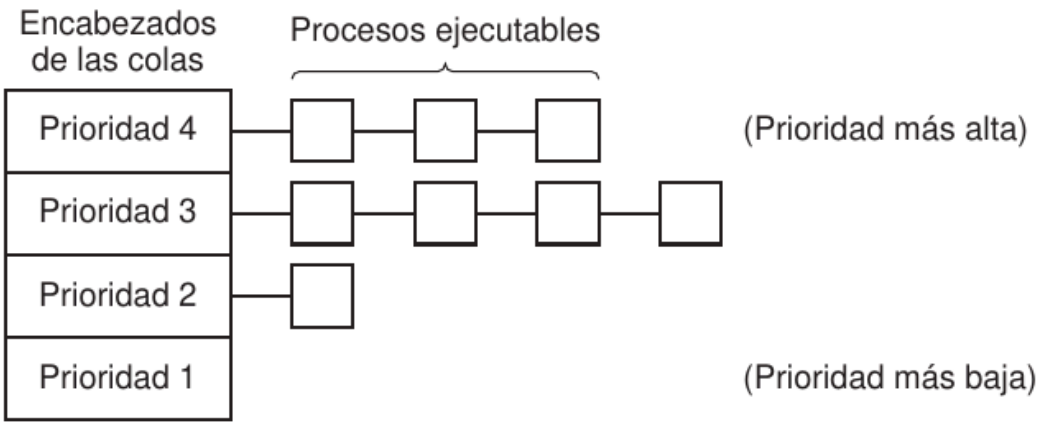



**Figura 2-42.** Un algoritmo de planificación con cuatro clases de prioridad.

El algoritmo funciona de la siguiente manera: se ejecutan todos los procesos en la cola con prioridad 4 usando round robin. Cuando la cola de prioridad 4 está vacía, se ejecutan todos los procesos en la cola con prioridad 3 usando round robin. Si las colas con prioridad 4 y 3 están vacías, se ejecutan todos los procesos con prioridad 2 por round robin y así sucesivamente. Si las prioridades no se ajustan de manera ocasional, todas las clases de menor prioridad pueden morir de inanición (*starvation*).

### SRTF - Shortest remaining time first

Es la versión apropiativa (*preemptive*) de SJF. Selecciona el proceso al cual le resta menos tiempo de ejecución en su siguiente ráfaga. Favorece a los procesos I/O bound.

Se estima el siguiente valor en una serie mediante la obtención del promedio ponderado del valor actual medio y la estimación anterior, este proceso se conoce como **envejecimiento (*aging*).**

## Planificación con múltiples procesadores

La planificación con múltiples procesadores es más compleja. La carga se divide entre distintos procesadores logrando capacidades de procesamiento mayores.

### Criterios

- **Planificación temporal:** que proceso y durante cuánto.
- **Planificación espacial:** en qué procesado ejecutar:
  - **Huella:** estado que el proceso va dejando en la cache de un procesador.
  - **Afinidad:** preferencia de un proceso para ejecutar en un procesador.
- La **asignación** puede ser:
  - **Estática:** existe una afinidad de un proceso a una CPU.
  - **Dinámica:** la carga se comparte.
- La **política** puede ser:
  - **Tiempo compartido:** se puede considerar una cola global o una cola local a cada procesador.
  - **Espacio compartido:** grupos (threads) o particiones.
- **Clasificaciones:**
  - **Procesadores homogéneos:** todos los procesadores son iguales, no existen ventajas físicas sobre el resto.
  - **Procesadores heterogéneos:** cada procesador tiene su propio clock y su propio algoritmo de planificación.
  - **Otras clasificaciones:**
    - **Procesadores débilmente acoplados:** cada procesador tiene su propia memoria principal y canales.
    - **Procesadores fuertemente acoplados:** comparten memoria y canales.
    - **Procesadores especializados:** uno o más procesadores principales de uso general y uno o más procesadores de uso específico.

# Memoria

La organización y administración de la memoria principal es uno de los factores más importantes en el diseño de los sistemas operativos. Los programas y datos deben estar en el almacenamiento principal para poder ejecutarlos y referenciarlos directamente.

La parte del sistema operativo que administra (parte de) la jerarquía de memoria se conoce como **administrador de memoria.** Su trabajo es administrar la memoria con eficiencia: llevar el registro de cuáles partes de la memoria están en uso y cuáles no, asignar memoria a los procesos cuando necesiten y designarla cuando terminen. Además, el sistema operativo debe abstraer al programador de la alocación de los programas. Brindar seguridad entre los procesos para que no accedan a secciones de memoria privadas de otros procesos, y a la vez brindar la posibilidad de acceso a zonas de memoria compartidas.

## Gestión de memoria

El sistema realiza una división lógica de la memoria física para alojar múltiples procesos. La gestión de memoria debe satisfacer cinco requisitos: reubicación, protección, compartición, organización lógica y organización física.

### Reubicación

Como la memoria principal se comparte entre varios procesos y, cuando se *swapea* un proceso, podría ser necesario **reubicarlo** en un área de memoria diferente; entonces, no se puede anticipar en dónde se va a colocar un programa y se debe permitir que pueda moverse en la memoria principal debido al *swap*.

El sistema operativo necesita conocer la ubicación de la información de control del proceso y de la pila, así como el punto de entrada que utilizará el proceso para iniciar la ejecución. Debido a que el sistema operativo se encarga de gestionar la memoria y es el responsable de traer el proceso a la memoria principal, estas direcciones son fáciles de adquirir. Sin embargo, el procesador también debe tratar con referencias de memoria dentro del propio programa.

De alguna forma, el hardware del procesador y el software del sistema operativo deben poder traducir las referencias de memoria encontradas en el código del programa en direcciones de memoria físicas, que reflejan la ubicación actual del programa en la memoria principal. **Las referencias a la memoria se deben "traducir" según su ubicación actual en el proceso.**

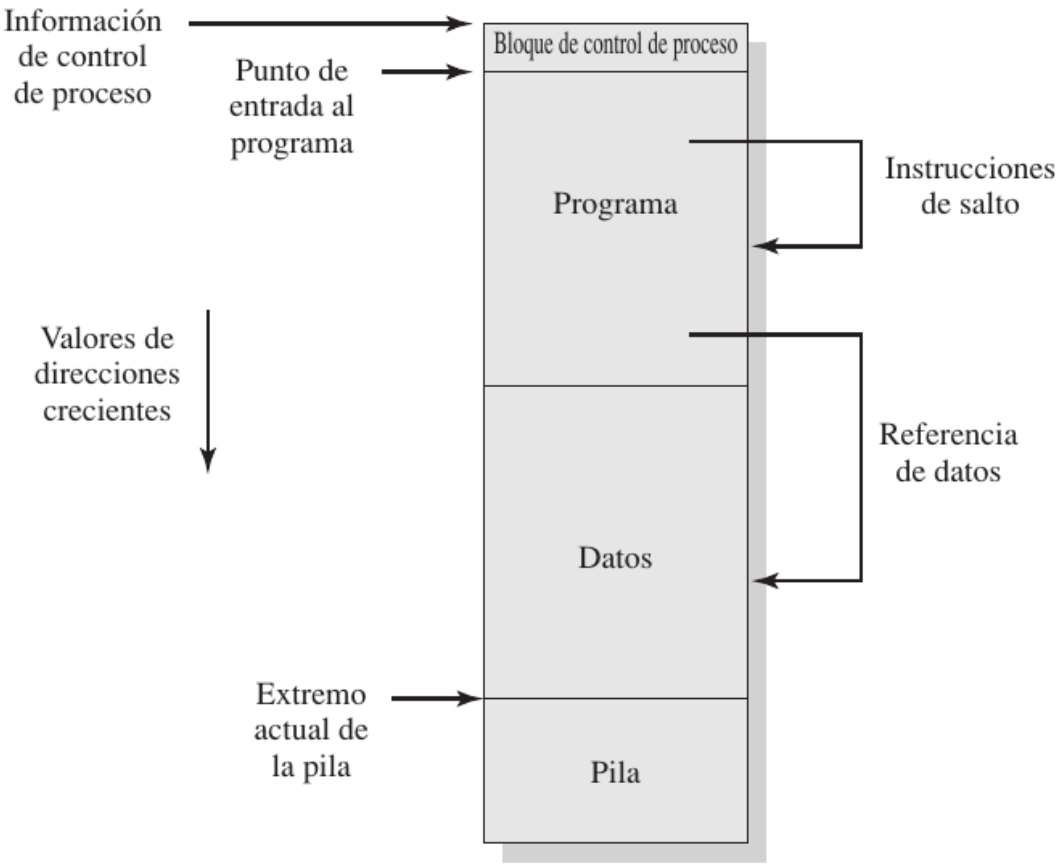



**Figura 7.1.** Requisitos de direccionamiento para un proceso.

## Protección

Los procesos no deben ser capaces de referenciar sin permiso posiciones de memoria de otros procesos, tanto en modo de lectura como de escritura. Por lo tanto, todas las referencias de memoria generadas por un proceso deben comprobarse en tiempo de ejecución para poder asegurar que se refieren solo al espacio de memoria asignado a dicho proceso.

Los requisitos de protecciṕon de memoria deben ser satisfechos por el procesador (hardware) en lugar del sistema operativo (software). Esto es porque el sistema operativo no puede anticipar todas las referencias de memoria que un programa hará (y sería muy costoso hacerlo).

## Compartición

El sistema de gestión de memoria debe permitir el acceso controlado a áreas de memoria compartidas (porción en común de memoria principal para distintos procesos) sin comprometer la protección esencial. Por ejemplo, varias instancias de un programa podrían acceder a la misma área de texto, en lugar de tener su propia copia separada. Esto permite realizar un mejor uso de la memoria principal, evitando copias repetidas de instrucciones.

### Organización lógica

La memoria principal de un computador se organiza como un espacio de almacenamiento lineal, pero esto no se corresponde a la forma en la cual los programas se construyen. Los programas se organizan en módulos que pueden o no ser modificados y el sistema operativo y el hardware deben tratar de forma efectiva.

### Organización física

La memoria de una computadora se organiza en al menos dos niveles: la *memoria principal* (pequeña y que contiene programas y datos en uso) y la *memoria secundaria* (grande y almacena programas y datos a largo plazo), y la organización del flujo de información entre ambas es responsabilidad del sistema.

## Espacio de direcciones

El espacio de direcciones es una abstracción para la memoria que crea un tipo de memoria abstracta para que los programas vivan ahí. Un **espacio de direcciones (address space)** es el conjunto de direcciones que puede utilizar un proceso. Cada proceso tiene su propio espacio de direcciones, independiente de los que pertenecen a otros procesos (excepto las áreas de memoria compartida).

## Registro base y registro límite

Cada CPU cuenta con dos registros conocidos como registro base y límite. Cuando se ejecuta un proceso, el **registro base** carga la dirección física donde empieza el programa en memoria y el **registro límite** se carga con la longitud del programa. En muchas implementaciones, estos registros están protegidos de forma tal que solo el sistema operativo puede modificarlos.

Cada vez que un proceso hace referencia a la memoria, el hardware de la CPU suma de manera automática el valor base a la dirección generada por el proceso antes de enviar la dirección al bus de memoria. Al mismo tiempo comprueba si la dirección ofrecida es igual o mayor que el valor resultante de sumar los valores de los registros límite y base, en cuyo caso se genera un fallo y se aborta el acceso.

Esta tarea es realizada por la **MMU (Memory Management Unit),** una unidad del procesador (hardware) que mapea direcciones virtuales a direcciones físicas. Re-programar la MMU es una operación privilegiada, ya que solo puede ser realizada en modo kernel.

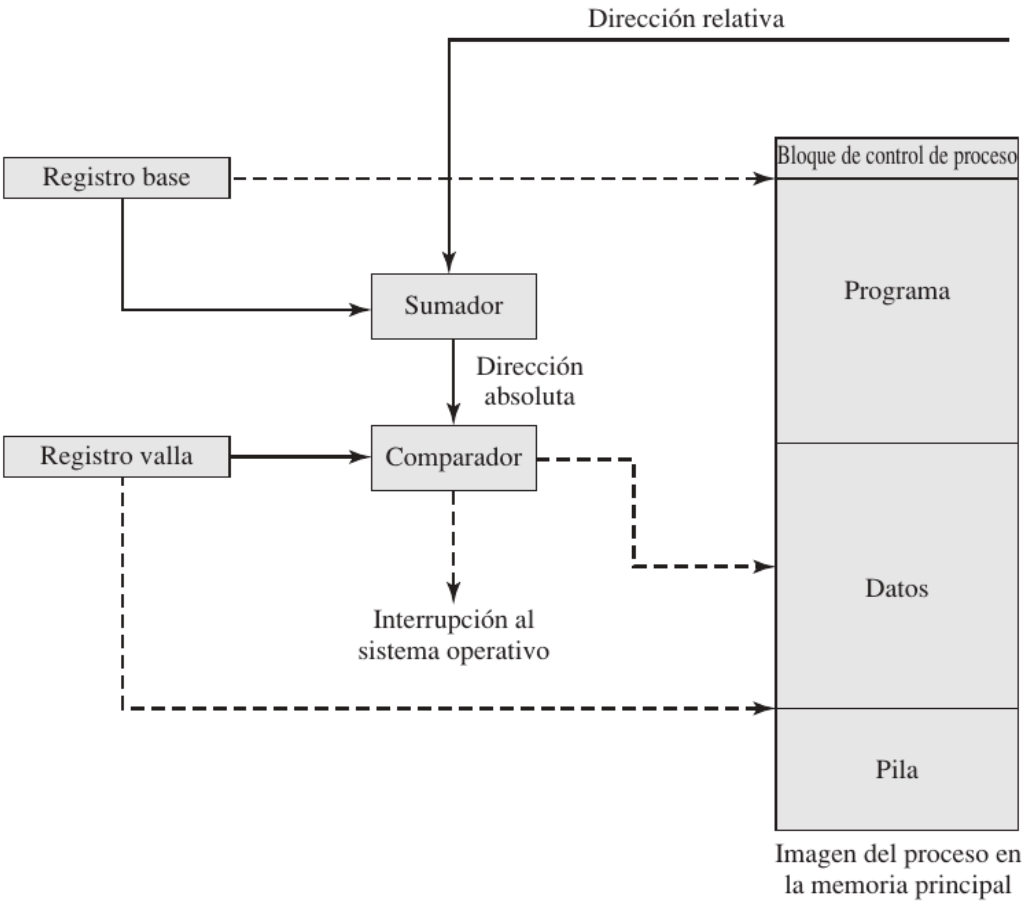



**Figura 7.8.** Soporte hardware para la reubicación.

## Mecanismos de asignación de memoria

**Particiones fijas:** la memoria se divide en particiones o regiones de tamaño fijo (pueden ser todas del mismo tamaño o no) y cada proceso se coloca en alguna de estas particiones siguiendo algún criterio (first fit, best fit, worst fit, next fit).

**Particiones dinámicas:** varían en tamaño y en número, alojan un proceso cada una y cada partición se genera en forma dinámica del tamaño justo que necesita el proceso.

## Problemas del esquema

El esquema de registro base y registro límite presenta varios problemas:

- Necesidad de almacenar el espacio de direcciones de forma continua en la memoria física.
- Los primeros sistemas operativos definían particiones fijas de memoria, luego evolucionaron a particiones dinámicas.
- Fragmentación.
- Mantener “partes” del proceso que no son necesarias.
- Los esquemas de particiones fijas y dinámicas no se usan hoy en día.

Para solucionar estos problemas se utilizan la paginación y la segmentación.

### Fragmentación

La fragmentación se produce cuando una localidad de memoria no puede ser utilizada por no encontrarse en forma contigua.

- **Fragmentación interna:** se produce en esquema de ***particiones fijas*** en la porción de la partición que queda sin utilizar.
- **Fragmentación externa:** se produce en el esquema de ***particiones dinámicas***. Son los huecos que van quedando en memoria a medida que los procesos finalizan, que, al no encontrarse en forma contigua, puede darse el caso de que tengamos memoria libre para alocar un proceso perno que no la podamos utilizar. Para solucionar este problema se puede utilizar la compactación de memoria (pero es muy costosa).

## Paginación

La ***memoria física*** es dividida lógicamente en trozos de igual tamaño llamados **marcos** y la ***memoria lógica*** (espacio de direcciones) es dividida en trozos de igual tamaño que los marcos llamados **páginas**. Con paginación, la *fragmentación interna* corresponde solo a una fracción de la última página de un proceso, y no existe *fragmentación externa*.

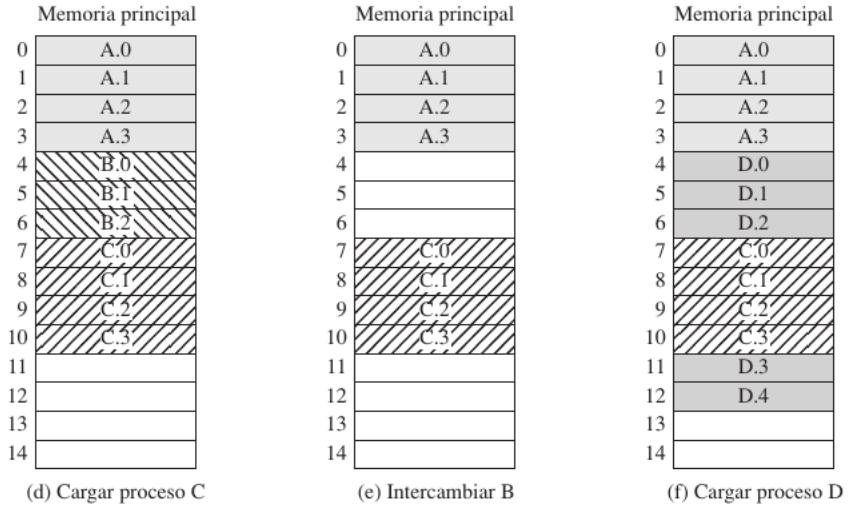



**Figura 7.9.** Asignación de páginas de proceso a marcos libres.

El sistema operativo debe mantener una tabla de páginas por cada proceso, donde cada entrada contiene (entre otras cosas) el *marco* en el que se coloca cada página. Así, la dirección lógica se interpreta como un número de página y un desplazamiento dentro de la misma.

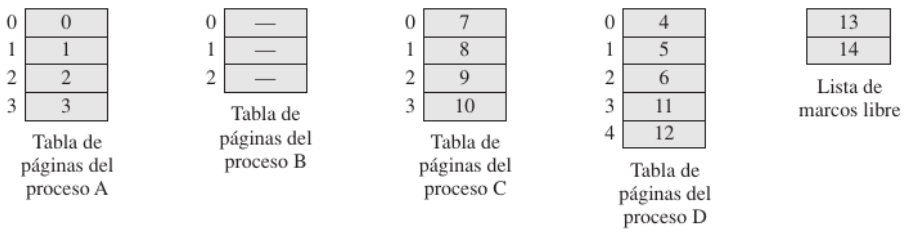
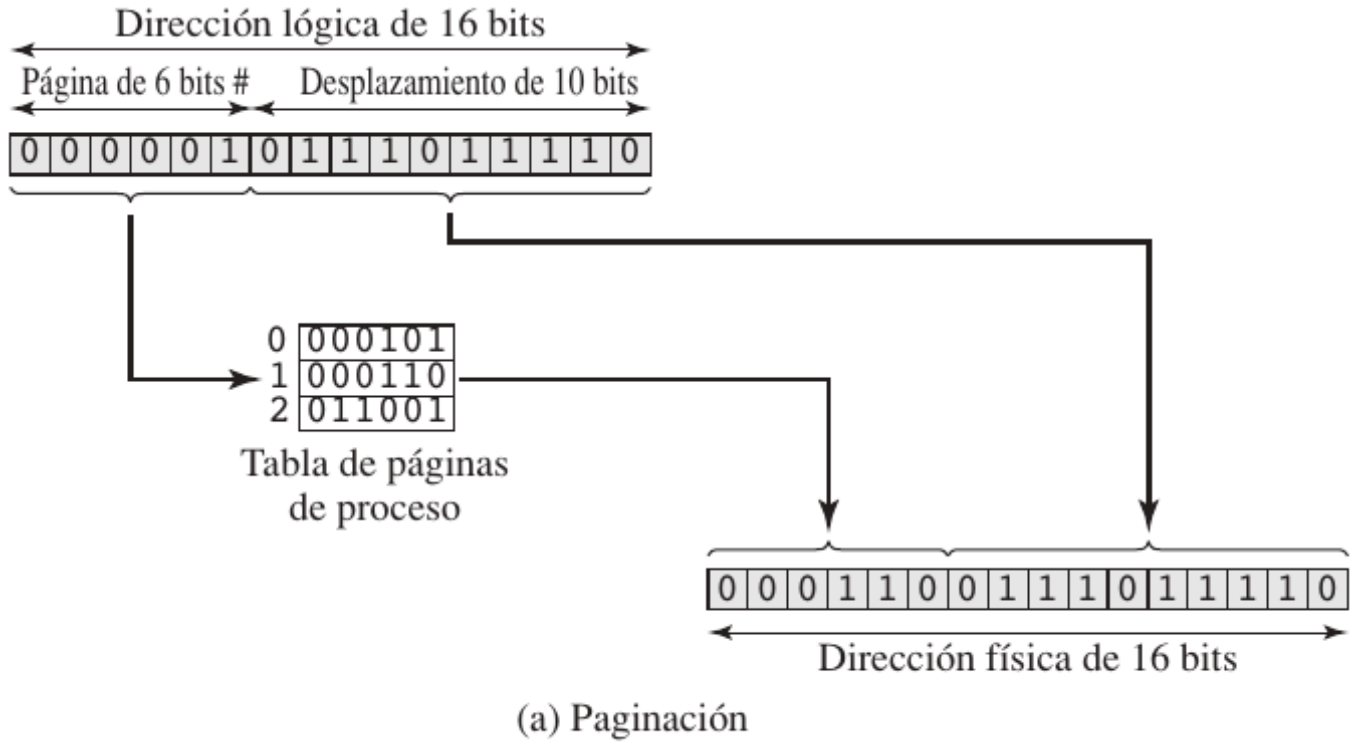



**Figura 7.10.** Estructuras de datos para el ejemplo de la Figura 7.9 en el instante (f).

Con paginación, la traducción de direcciones lógicas a físicas las continúa realizando el hardware del procesador, por lo que ahora debe conocer cómo acceder a la tabla de páginas del proceso.




(a) Paginación

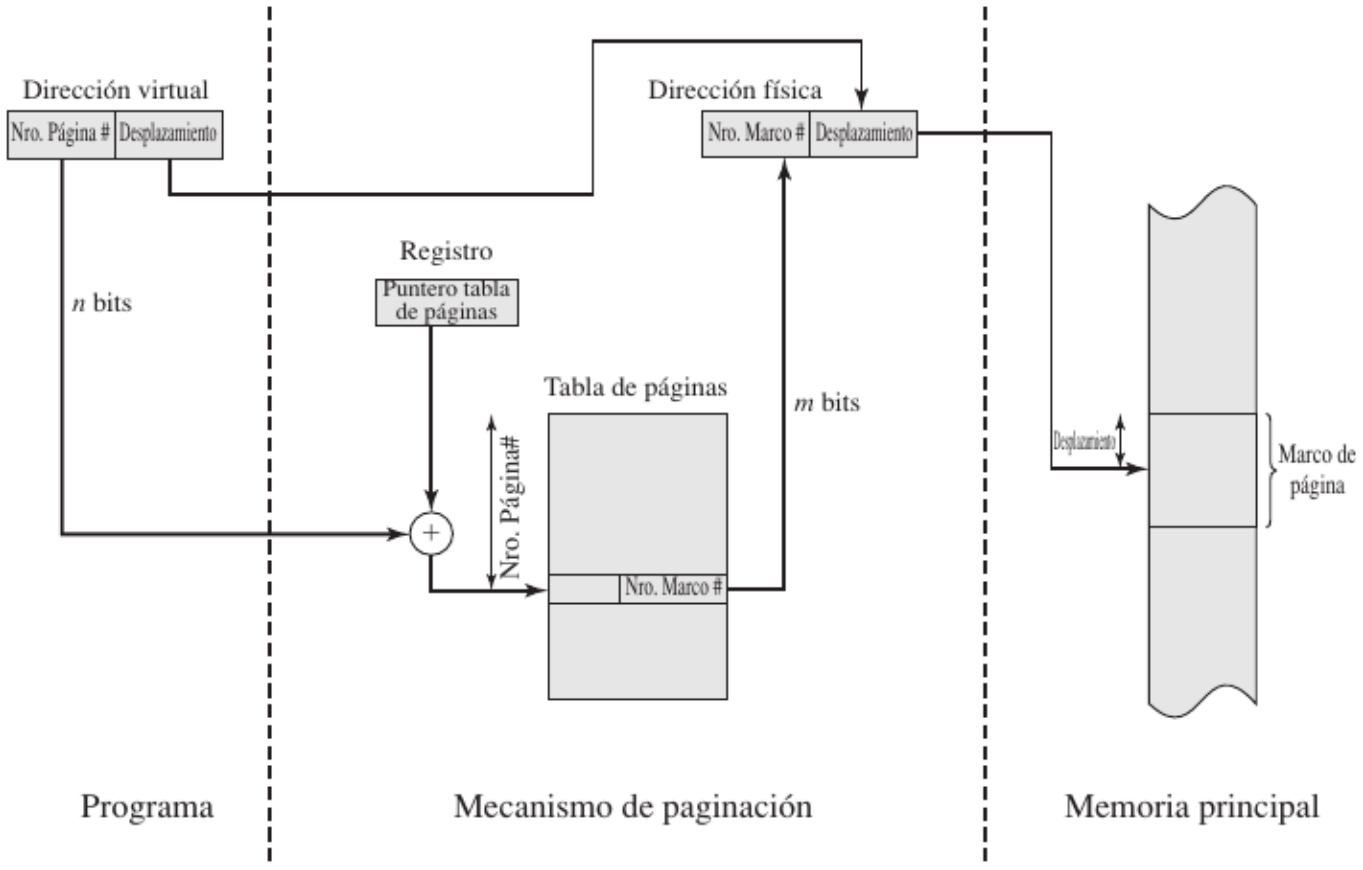


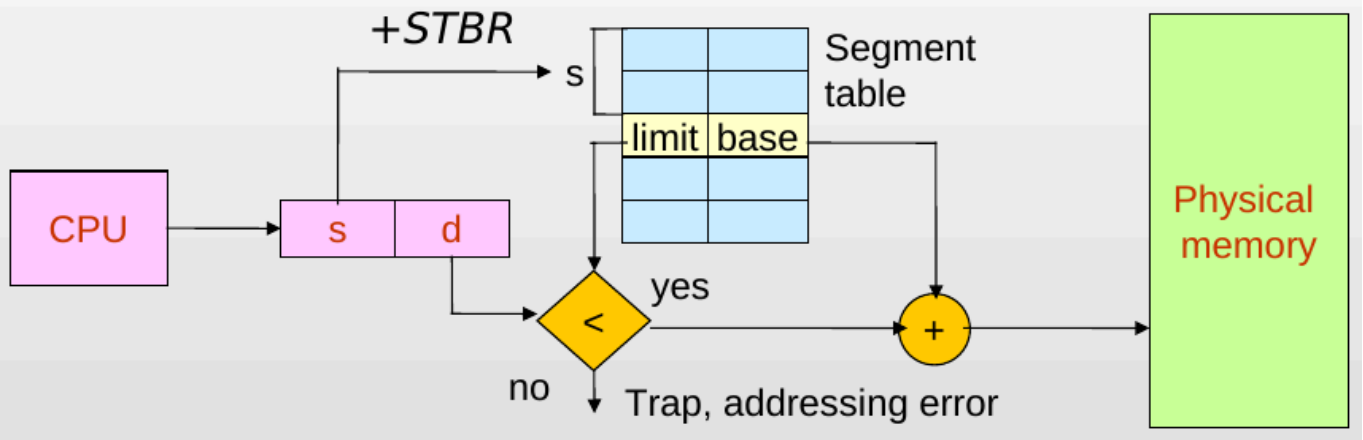

**Figura 8.3.** Traducción de direcciones en un sistema con paginación.

# Segmentación

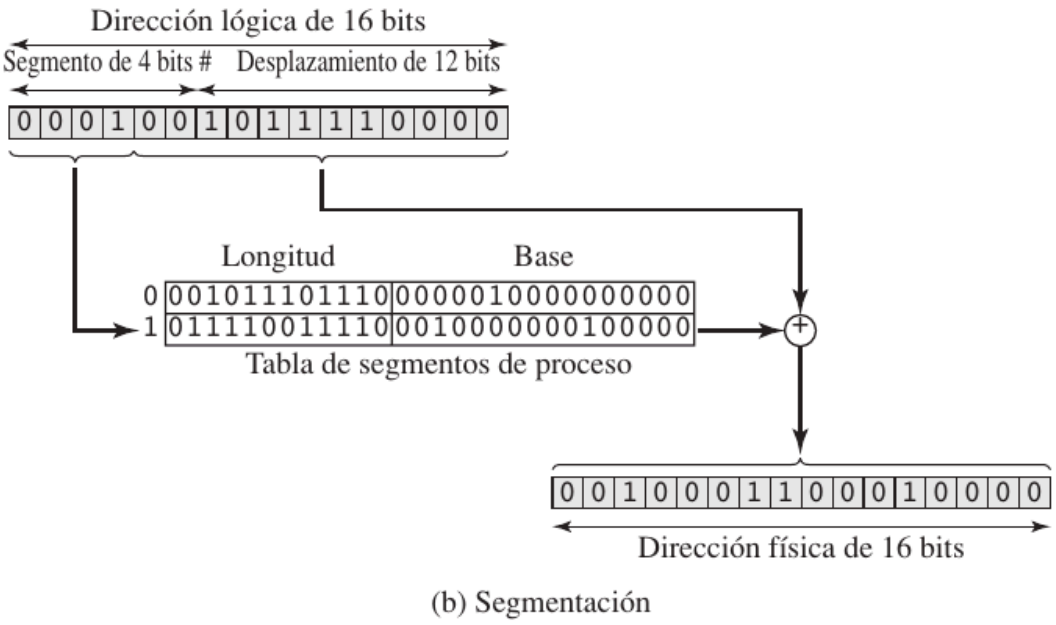
Con **segmentación** el programa y sus datos asociados se divide en un número de segmentos. Un **segmento** es una unidad lógica como: programa principal, funciones, variables locales, variables globales, stack, etc. Se necesita que todos los segmentos de un programa se carguen en memoria para su ejecución y, a diferencia de la paginación, la segmentación es normalmente visible y se proporciona como una utilidad para organizar programas y datos. La segmentación elimina la fragmentación interna, pero sufre de fragmentación externa.

Los segmentos pueden no tener el mismo tamaño y las direcciones lógicas consisten en dos partes: **selector de segmento** y **desplazamiento** dentro del segmento. Existe una **tabla de segmentos** que permite mapear la dirección lógica en física y cada entrada contiene:

- **base:** dirección física del comienzo del segmento.
- **limit:** longitud del segmento.




**Segment-table base register (STBR):** apunta a la ubicación de la tabla de segmentos.
**Segment-table length register (STLR):** cantidad de segmentos de un programa.




(b) Segmentación

# Memoria virtual

La necesidad de cargar en memoria todo nuestro programa limita al tamaño de nuestro programa al tamaño de la memoria física. Sin embargo, en muchos casos no se necesita todo el programa y, incluso cuando se necesita todo el programa, puede que no se necesite todo al mismo tiempo.

La habilidad de ejecutar un programa que está solo parcialmente cargado en memoria ofrece ventajas tanto al sistema como a los usuarios:

- **Los programas no se limitan a la cantidad de memoria física disponible:** un proceso puede ser mayor que toda la memoria principal, ya que el programador trabaja con una memoria enorme, con un tamaño asociado al almacenamiento en disco, y el sistema operativo automáticamente carga porciones de un proceso en la memoria principal cuando estas se necesitan.
- **Pueden mantenerse un mayor número de procesos en memoria principal:** como solo vamos a cargar porciones de los procesos a ejecutar, existe espacio para más procesos.
- **Se necesita menos E/S para cargar o intercambiar partes de los programas en la memoria:** por lo que cada programa podría ejecutarse más rápido.

La **memoria virtual** implica la separación de la memoria lógica, tal y como la perciben los programadores, de la física.

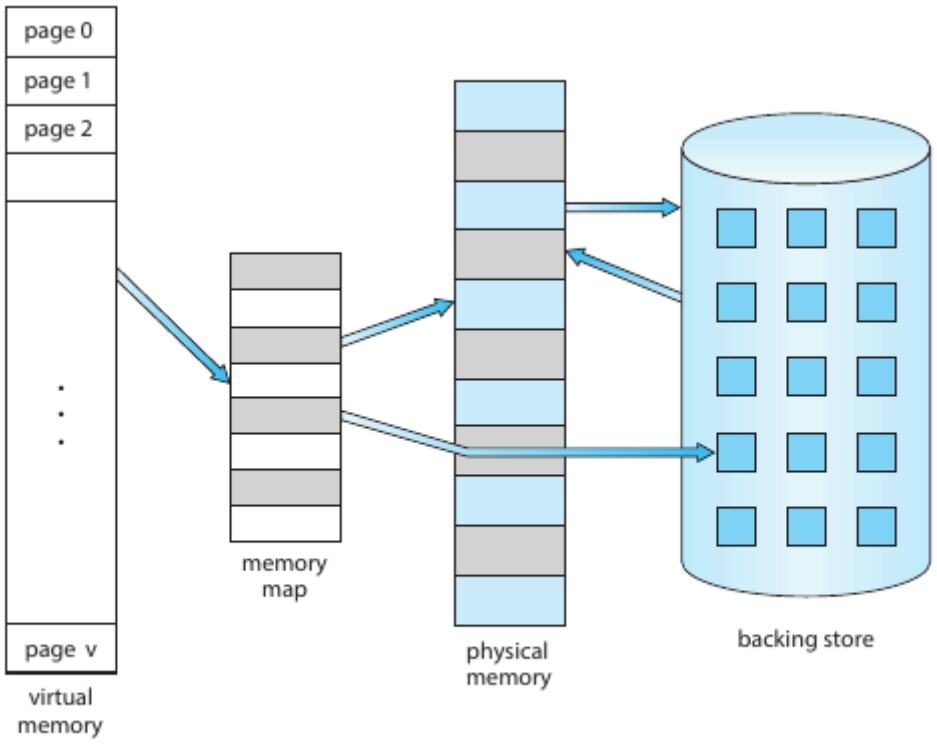



**Figure 10.1** Diagram showing virtual memory that is larger than physical memory.

El **espacio de direcciones virtual** de un proceso se refiere a la visión lógica (o virtual) de cómo se almacena un proceso en la memoria. Típicamente, esta visión es que un proceso comienza en una cierta dirección lógica (digamos, dirección 0) y existe en memoria contigua, sin embargo, la

memoria física está organizada en marcos de página y los marcos de página físicos asignados a un proceso pueden no ser contiguos. Depende de la unidad de gestión de memoria (MMU) asignar páginas lógicas a marcos de páginas físicas en memoria.

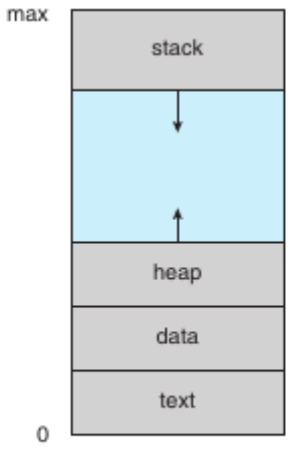



**Figure 10.2** Virtual address space of a process in memory.

Además, la memoria virtual permite que dos procesos compartan archivos y memoria mediante el uso compartido de páginas.

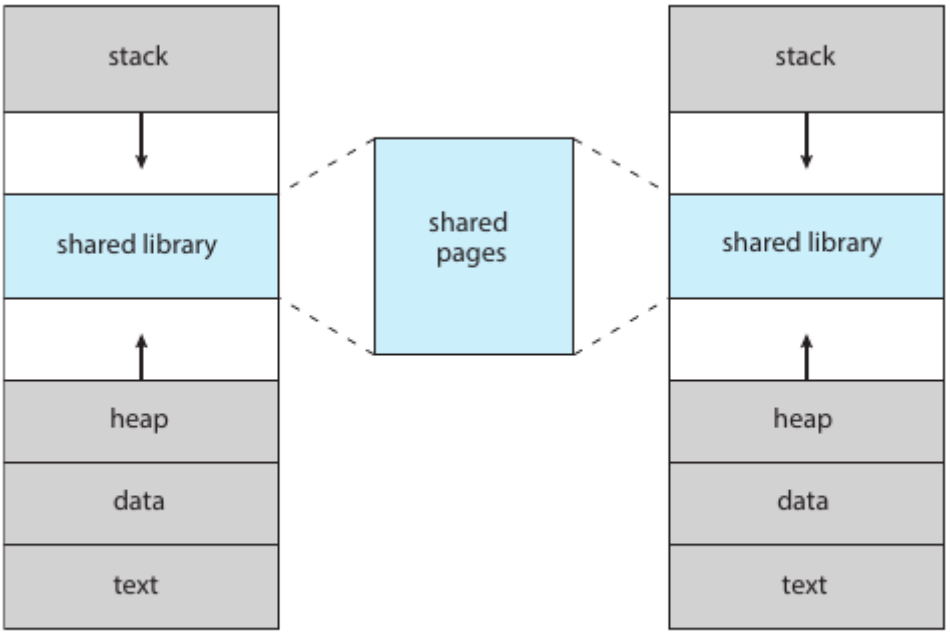



**Figure 10.3** Shared library using virtual memory.

## Hardware y estructuras de control

La paginación y la segmentación tienen dos características clave:

1. Todas las referencias a memoria dentro de un proceso se realizan a direcciones lógicas que se traducen dinámicamente a direcciones físicas durante la ejecución.
    - Esto permite que un proceso pueda ocupar distintas regiones de la memoria principal durante su ejecución.
2. Un proceso puede dividirse en varias *porciones* (páginas o segmentos) que pueden no estar localizadas en la memoria de forma contigua durante la ejecución.

Supongamos que se tiene que traer un nuevo proceso de memoria:

1. **Inicio del proceso:** el sistema operativo comienza trayendo únicamente una o dos porciones, que incluye la porción inicial del programa y la porción inicial de datos sobre la cual acceden las primeras instrucciones.
    - Esta parte del proceso que se encuentra realmente en la memoria principal se denomina **conjunto residente (working set)** del proceso.
2. **Ejecución del proceso:** mientras que todas las referencias a la memoria se encuentren dentro del conjunto residente el proceso se ejecuta sin problemas.
    - El procesador utiliza una tabla de segmentos o páginas para determinar esto.
3. **Fallo de acceso a la memoria:** si el procesador encuentra una dirección lógica que no se encuentra en la memoria principal, generará una interrupción indicando un fallo de acceso a la memoria. El sistema operativo coloca al proceso interrumpido en un estado de bloqueado y toma el control.
4. **Gestión del fallo de acceso:** para que la ejecución de este proceso pueda reanudarse más adelante, el sistema operativo necesita traer a la memoria principal la porción del proceso que contiene la dirección lógica que ha causado el fallo de acceso. Con este fin, el sistema operativo realiza una petición de E/S, una lectura a disco.
5. **Activación de otro proceso:** después de realizar la petición de E/S, el sistema operativo puede activar otro proceso que se ejecute mientras el disco realiza la operación de E/S.
6. **Finalización de la E/S y reanudación del proceso:** una vez que la porción solicitada se ha traído a la memoria principal, una nueva interrupción de E/S se lanza, dando control de nuevo al sistema operativo, que coloca al proceso afectado de nuevo en el estado Listo.

Debido a que un proceso se ejecuta solo en la memoria principal, a esta se la denomina **memoria real;** mientras que, a la memoria localizada en disco, se la denomina **memoria virtual.**

Esta estrategia trae consigo dos **ventajas**:

1. **Pueden mantenerse un mayor número de procesos en memoria principal:** como solo vamos a cargar porciones de los procesos a ejecutar, existe espacio para más procesos.
2. **Un proceso puede ser mayor que toda la memoria principal:** el programador trabaja con una memoria enorme, con un tamaño asociado al almacenamiento en disco y, el sistema operativo, automáticamente carga porciones de un proceso en la memoria principal cuando estas se necesitan.

## Proximidad y memoria virtual

El funcionamiento de la memoria virtual se basa en el **principio de proximidad:** las referencias al programa y a los datos dentro de un proceso tienden a agruparse; por lo tanto, solo unas pocas porciones del proceso se necesitarán a lo largo de un periodo de tiempo corto. Así, en cualquier momento, solo unas pocas porciones de cada proceso se encuentran en memoria.

En estado estable, prácticamente toda la memoria principal se encontrará ocupada por porciones de procesos, de forma que el procesador y el sistema operativo tengan acceso directo al mayor número posible de procesos. Así, cuando el sistema operativo traiga una porción a la memoria, debe expulsar otra. Si elimina una porción justo antes de que vaya a ser utilizada, deberá recuperar dicha porción de nuevo caso de forma inmediata. Un abuso de esto lleva a una condición denominada **hiperpaginación (trashing):** el sistema pasa más tiempo paginando (enviando y trayendo porciones de *swap*) que ejecutando procesos.

## Paginación con memoria virtual

Para la memoria virtual basada en el esquema de paginación también se asocia una tabla de páginas por cada proceso; pero, debido a que solo algunas páginas del proceso se encuentran en la memoria principal, se necesitan bits adicionales en las entradas de las tablas de páginas:

- **P (presente):** indica si la página está presente en memoria principal.
- **M (modificado):** indica si los contenidos de la página han sido alterados desde que se cargó por última vez en memoria principal.
    - Si no hubo ningún cambio, no es necesario escribir la página cuando llegue el momento de reemplazarla por otra página en el marco que actualmente ocupa.

Como en la mayoría de los sistemas existe una única tabla de página por proceso, la cantidad de memoria demandada por las tablas de página puede ser inaceptablemente grande. Para resolver esto, la mayoría de esquemas de memoria virtual almacenan las tablas de páginas en la memoria virtual. Así, las tablas de páginas están sujetas a paginación igual que cualquier otra página. Cuando un proceso está en ejecución, al menos parte de su tabla de páginas debe encontrarse en memoria, incluyendo la entrada de la tabla de páginas actualmente en ejecución.

### Paginación de dos niveles

Algunos procesadores utilizan un esquema de dos niveles para organizar las tablas de páginas de gran tamaño. En este esquema, existe un directorio de páginas, en el cual cada entrada apunta una tabla de páginas.

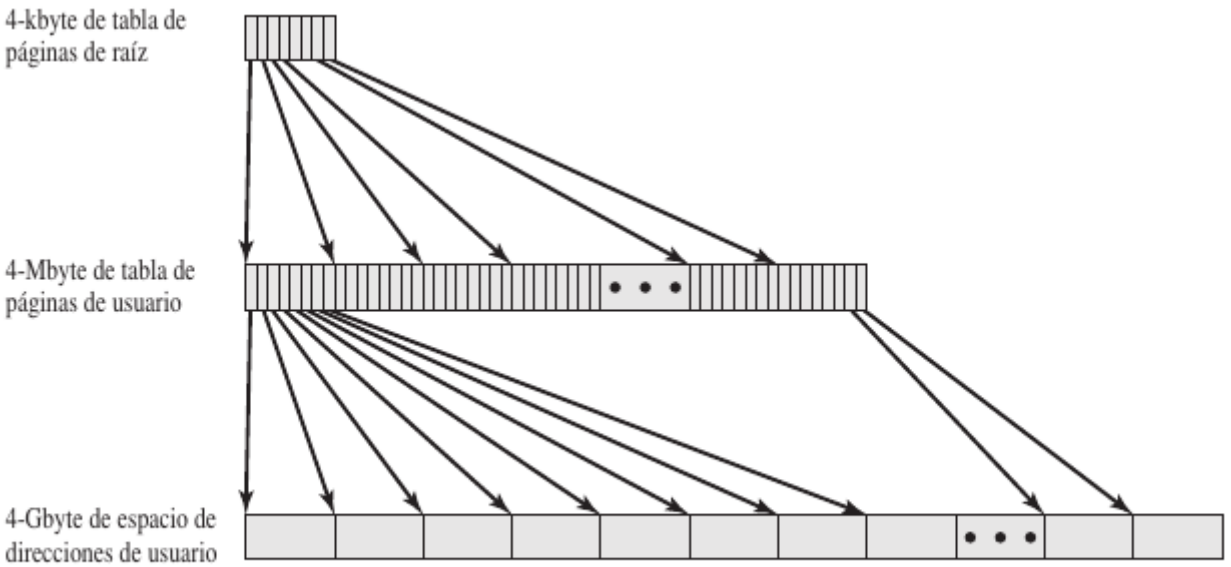



**Figura 8.4.** Una tabla de páginas jerárquica de dos niveles.

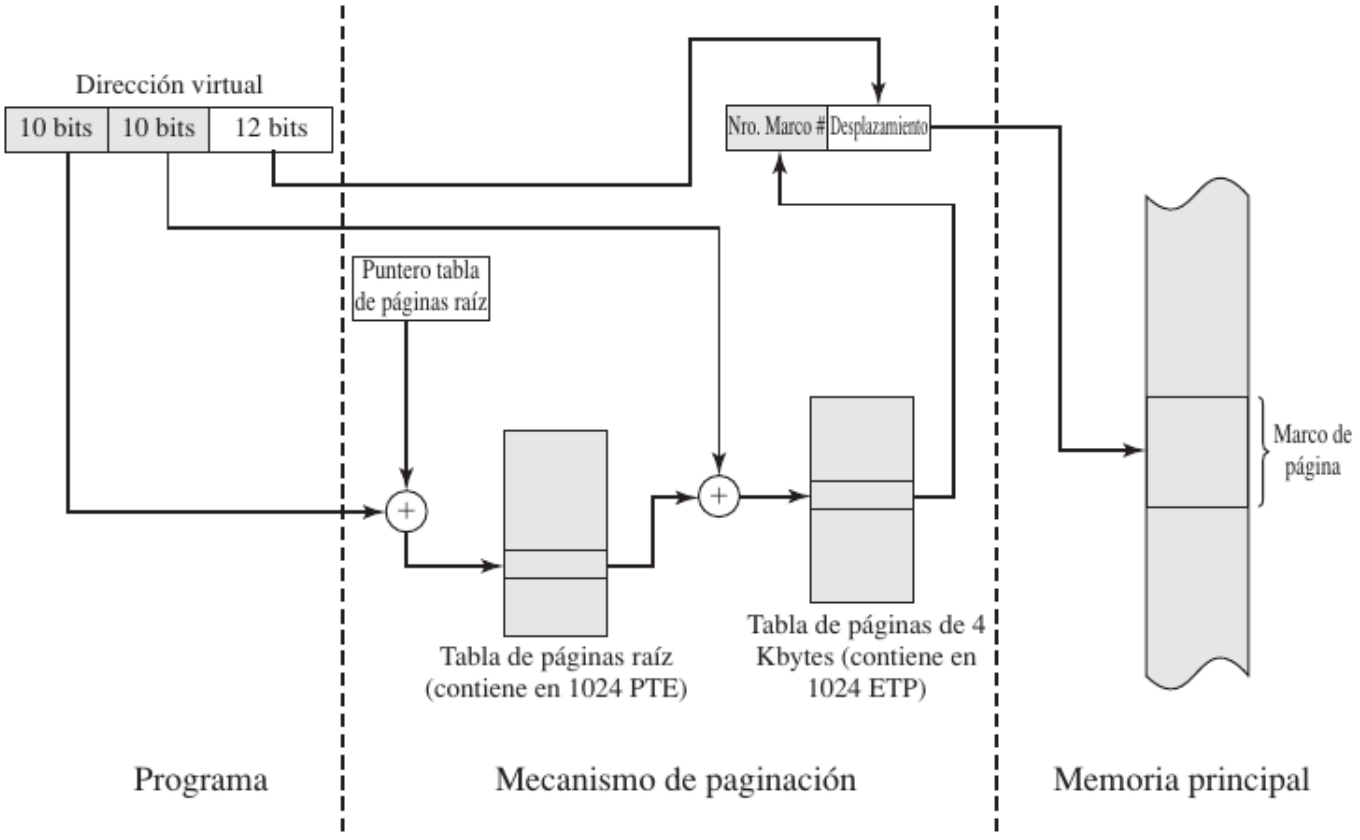



**Figura 8.5.** Traducción de direcciones en un sistema de paginación de dos niveles.

### Tabla de páginas invertida

Una desventaja del tipo de tablas de página es que su tamaño es proporcional al espacio de direcciones virtuales.

Una alternativa son las **tablas de páginas invertidas.** En esta estrategia, la parte correspondiente al número de página de la dirección virtual se referencia por medio de un valor *hash,* que es un puntero para la tabla de páginas invertidas, que contiene las entradas de tablas de página.

En la tabla de páginas invertida hay una entrada por cada marco de página real en lugar de uno por cada página virtual. De esta forma, lo único que se requiere es una proporción fija de la memoria real, independientemente del número de procesos. Así, para un tamaño de memoria física de $2^m$ marcos, la tabla de páginas invertida contiene $2^m$ entradas, de forma que la entrada *i-ésima* se refiere al marco *i.*

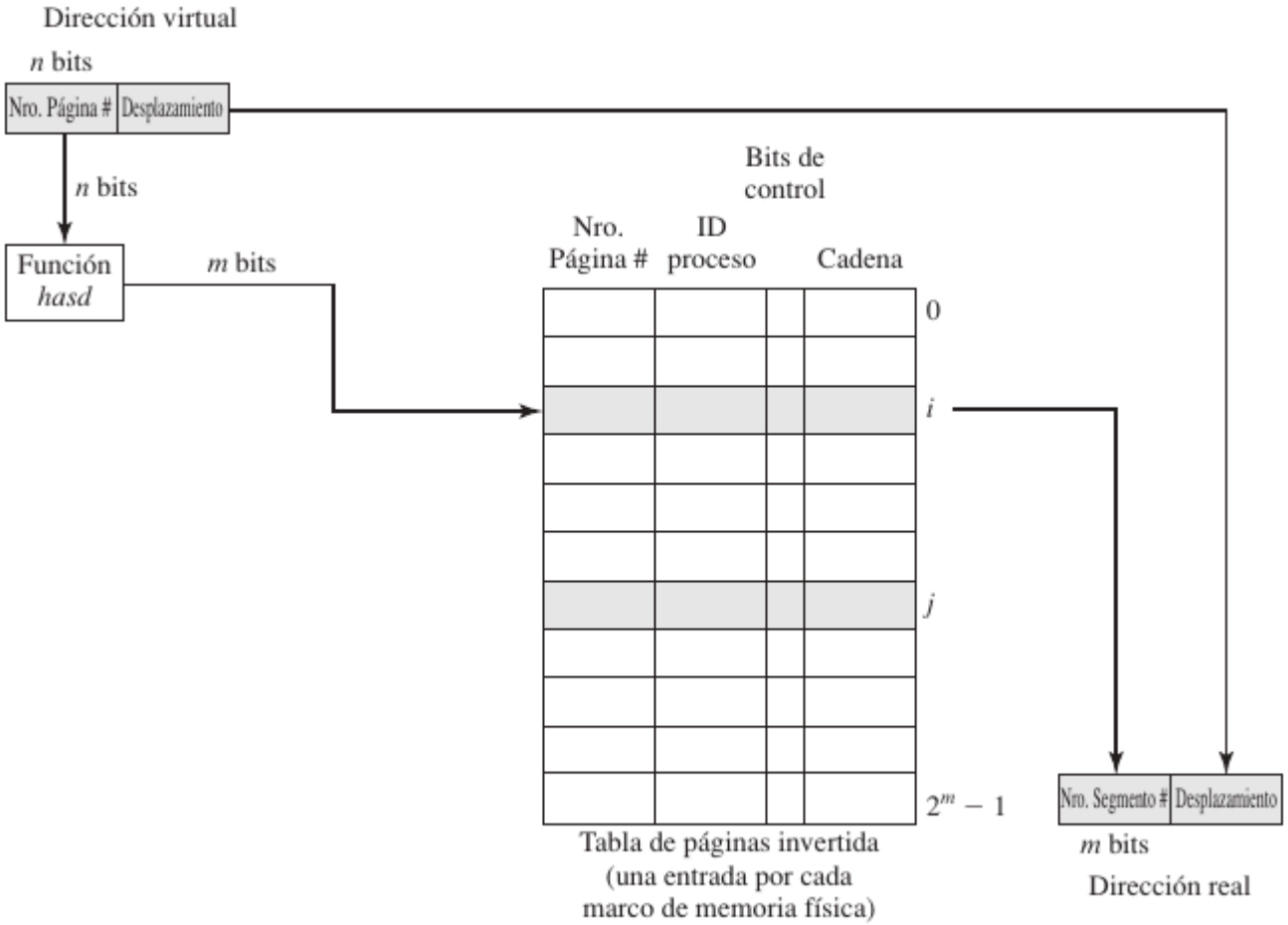



**Figura 8.6.** Estructura de tabla de páginas invertida.

La entrada en la tabla de páginas incluye la siguiente información: número de página, identificador del proceso, bits de control y puntero de la cadena (para manejar las colisiones).

### Buffer de traducción anticipada (TLB, translation lookaside buffer)

Toda referencia a la memoria virtual puede causar dos accesos a memoria física: uno para buscar la entrada en la tabla de páginas apropiada y otro para buscar los datos solicitados, lo que podría duplicar el tiempo de acceso a la memoria. Para solventar este problema, la mayoría de esquemas de memoria virtual utilizan una caché especial de alta velocidad para las entradas de la tabla de página, la TLB, que contiene las entradas de la tabla de páginas que han sido usadas de forma más reciente.

Dada una dirección virtual, el procesador examinará la TLB en busca de la entrada de la tabla de página solicitada:

- Si está en la TLB (acierto), entonces se recupera el número de marco y se construye la dirección real.
- Si no se encuentra en la TLB (fallo), el procesador utiliza el número de página para indexar la tabla de páginas del proceso y examinar la correspondiente entrada de la tabla de páginas.
    - Si el bit de presente está puesto en 1, entonces la página se encuentra en memoria principal y el procesador puede recuperar el número de marco desde la entrada de la tabla de páginas para construir la dirección real. El procesador también autorizará a la TLB para incluir esta nueva entrada de la tabla de páginas.
    - Si el bit de presente está puesto en 0, entonces la página solicitada no se encuentra en memoria principal y se produce un *fallo de página*. En este punto intervendrá el sistema operativo, que cargará la página necesaria y actualizará la tabla de páginas.

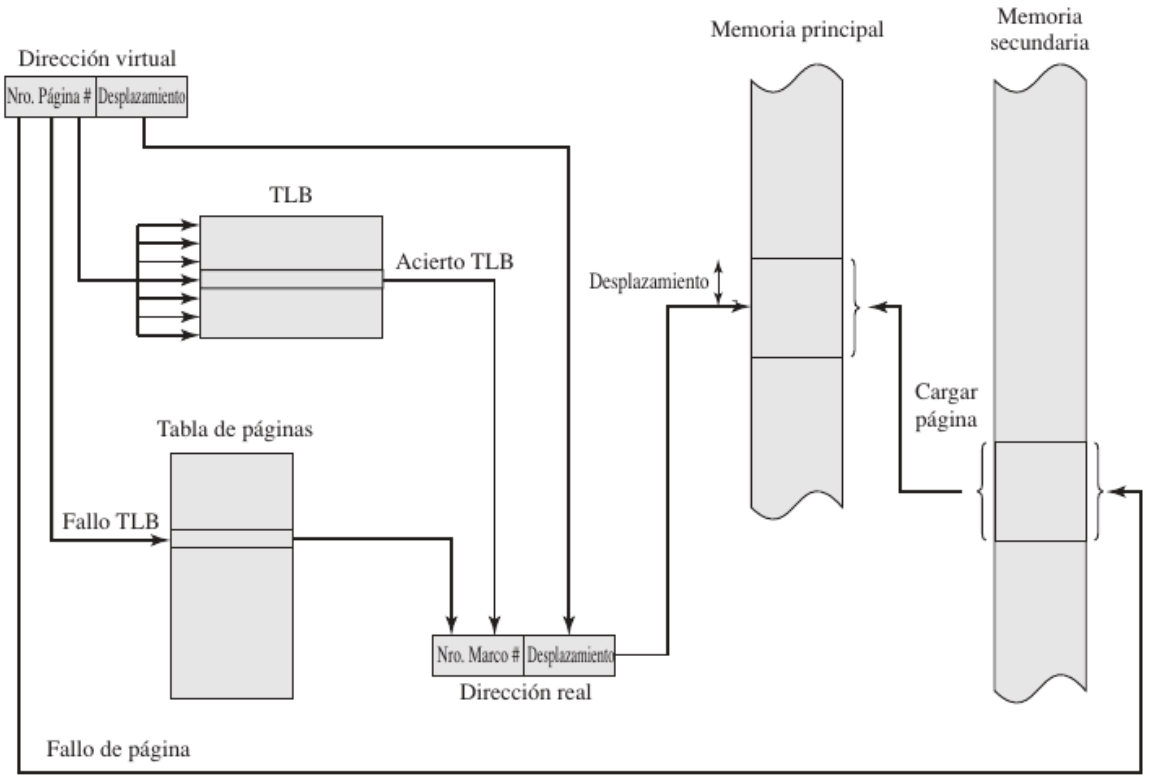
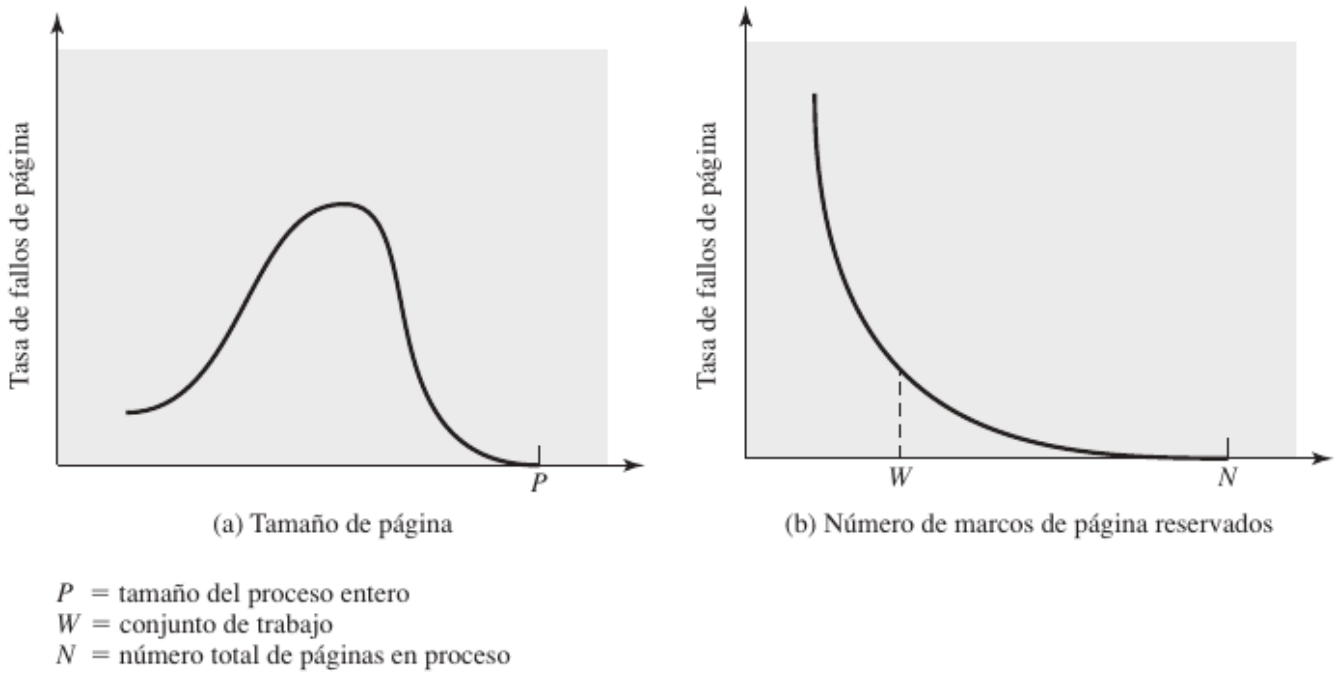



**Figura 8.7.** Uso de la TLB.

### Tamaño de página

- A menor tamaño de página, menor fragmentación interna.
- A menor tamaño de página, mayor cantidad de páginas son necesarias por cada proceso, por lo que se requieren mayores tablas de páginas, que pueden producir mayor cantidad de fallos de página doble para una referencia sencilla (falla el acceso a la entrada en la tabla de páginas y a la página propiamente dicha).
- Si el tamaño de página es pequeño, habrá un número relativamente alto de páginas disponibles en la memoria principal para cada proceso y, después de un tiempo, as páginas en memoria contendrán las partes de los procesos a las que se ha hecho referencia de forma reciente; reduciendo la tasa de fallos de página.
- A medida que el tamaño de páginas se incrementa, la página en particular contendrá información más lejos de la última referencia realizada, debilitando el principio de localidad y aumentando la tasa de fallos de página.
- Para un número de marcos fijos, la tasa de fallos cae a mediad que el número de páginas mantenidas en memoria principal crece.




(a) Tamaño de página (b) Número de marcos de página reservados

$P$ = tamaño del proceso entero
$W$ = conjunto de trabajo
$N$ = número total de páginas en proceso

**Figura 8.11.** Comportamiento típico de la paginación de un programa.

## Segmentación con memoria virtual

La segmentación con memoria virtual tiene las siguientes ventajas:

1. Simplifica el tratamiento de estructuras de datos que pueden crecer.
2. Permite programas que se modifican o recopilan de forma independiente.
3. Da soporte a la compartición entre procesos.
4. Soporta mecanismos de protección.

### Organización

- Cada proceso tiene su propia tabla de segmentos.
- Cada entrada en la tabla de segmentos contiene: la dirección de comienzo del segmento y su longitud.
- Debido a que solo algunos de los segmentos pueden encontrarse en memoria principal, se necesitan dos bits adicionales: bit P (presente) y bit M (modificado) (igual que en paginación).

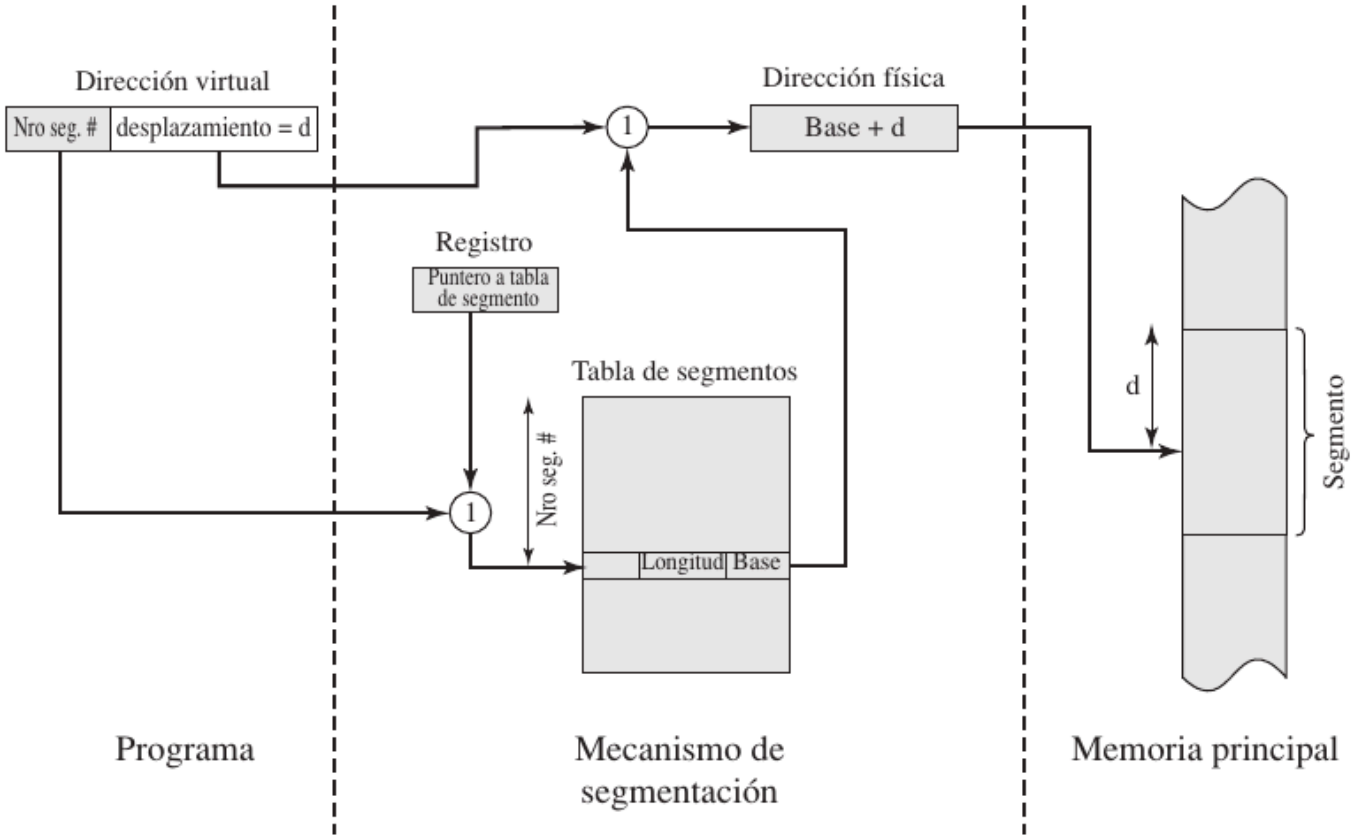



**Figura 8.12.** Traducción de direcciones en un sistema con segmentación.

## Segmentación paginada

La **paginación** es transparente al programador y elimina la fragmentación externa. La **segmentación** es visible al programador y facilita la modularidad, las estructuras de datos grandes y da mejor soporte a la compartición y protección.

En **segmentación paginada** cada segmento es dividido en páginas de tamaño fijo (igual a un marco de memoria). Si un segmento es inferior a una página, ocupará únicamente una página.

Cada proceso tiene una tabla de segmentos y cada uno de estos segmentos tiene su propia tabla de páginas, además, un registro mantiene la dirección de comienzo de la tabla de segmentos del proceso.

Las direcciones virtuales están formadas por el segmento de página, el número de página y un desplazamiento. El procesador utiliza el ***número de segmento*** para buscar, en la tabla de segmentos del proceso, la tabla de páginas asociada a ese segmento. Luego, utiliza el ***número de página*** para buscar el número de marco dentro de la tabla de páginas del segmento. Finalmente, utiliza el ***desplazamiento*** para generar la dirección real requerida.

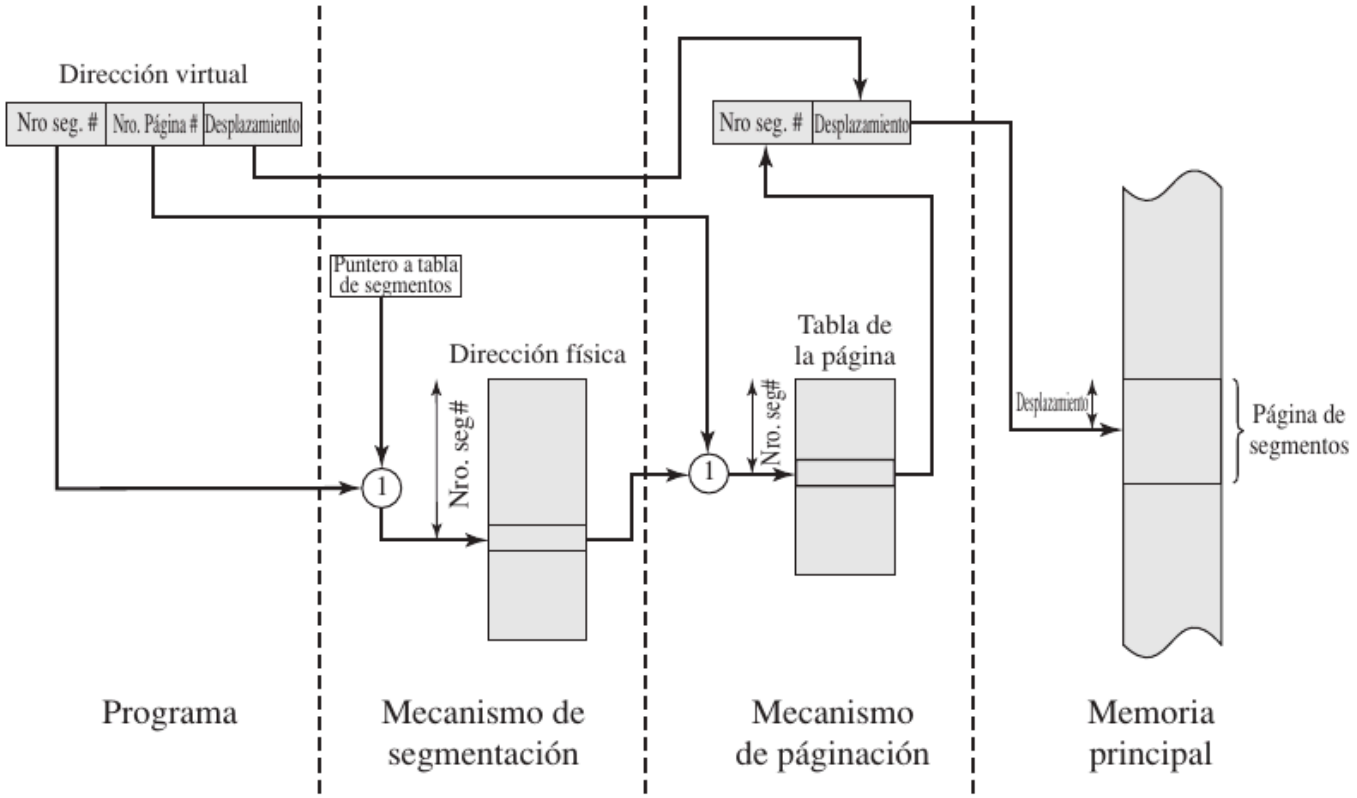



**Figura 8.13.** Traducción de direcciones en un sistema con segmentación/paginación.

Dirección virtual

| Segmento de página | Número de página | Desplazamiento |
|---|---|---|

Entrada de la tabla de segmentos

| Bits de control | Longitud | Comienzo de segmento |
|---|---|---|

Entrada de la tabla de páginas

| P | M | Otros bits de control | Número de marco |
|---|---|---|---|

P = bit de presente
M = bit de modificado

(c) Combinación de segmentación y paginación

## Protección y compartición

- **Protección:** como cada entrada en la tabla de segmentos incluye la longitud y la dirección base, un programa no puede acceder a posiciones fuera de los límites del segmento.
- **Compartición:** un segmento pude ser referenciado desde las tablas de segmentos de más de un proceso.