redundantes, ya que ambas se utilizan para separar los espacios de direcciones físicas de los procesos: la segmentación puede asignar un espacio de direcciones lineales a cada proceso mientras la paginación puede hacer corresponder el mismo espacio de direcciones lineales en diferentes espacios de direcciones físicas. Linux prefiere la paginación a la segmentación por las siguientes razones:

a) La gestión de memoria es más simple y eficiente cuando todos los procesos utilizan los mismos valores de registro de segmento, esto es, cuando comparten el mismo conjunto de direcciones lineales.
b) Uno de los objetivos de diseño de Linux es su transportabilidad a las arquitecturas más populares, algunas de las cuales, como los procesadores RISC, soportan segmentación de una forma limitada.

Linux utiliza segmentación sólo cuando lo requiere la arquitectura Intel 80x86. En concreto, todos los procesos utilizan las mismas direcciones lógicas, así el número total de segmentos a definir es muy limitado y es posible almacenar todos los descriptores de segmentos en la Tabla Global de Descriptores (GDT). Esta tabla se implementa por la matriz `gdt_table()` referenciada por la variable `gdt`. El kernel no utiliza las tablas de descriptores locales (LDT), si bien existe una llamada al sistema que permite a los procesos crear sus propias LDTs. Esto permite a aplicaciones como *Wine* ejecutar aplicaciones de Microsoft Windows orientadas a segmentos. Los segmentos utilizados por Linux son:

a) Un segmento de código kernel y otro segmento para datos del kernel.
b) Un segmento de código de usuario y otro segmento de datos de usuario, compartidos por todos los procesos en modo usuario.
c) El Segmento Estado de Tarea (TSS) para cada procesador. Si bien el procesador Intel suministra un segmento TSS para realizar el cambio de contexto a nivel hardware, Linux no lo realiza así por razones de transportabilidad, pero utiliza este segmento para: recuperar la dirección de la pila kernel en el cambio de contexto, y permitir a las aplicaciones acceder directamente a los puertos de E/S.
d) Un segmento con la LDT (Tabla de Descriptores Locales) por defecto, que es normalmente compartido por todos los procesos.
e) Tres segmentos TLS (*Thread-Local Storage*).
f) Tres segmentos relacionados con el soporte para la *Gestión Avanzada de Potencia* (APM: rutinas BIOS dedicadas a gestionar los estados de potencia del sistema).
g) Cinco segmentos relacionados con los servicios BIOS del "*Plug and Play*" (PnP).
h) Un segmento especial TSS utilizado por el kernel para manejar la "Doble Falta".

2.7.1 Paginación en Linux

Linux adopta un modelo de paginación a cuatro niveles de forma que sea adecuado para arquitecturas de 32 y 64-bits. Los cuatro tipos de tablas de páginas:

a) Directorio global de páginas (tabla de 1º nivel)
b) Directorio superior de páginas (tablas de 2º nivel)
c) Directorio intermedio de páginas (tablas de 3º nivel)
d) Tabla de páginas (tablas de 4º nivel)

Cuando este modelo se aplica al Pentium, que utiliza paginación a dos niveles, se eliminan los directorios superior e intermedio de páginas indicando que contiene 0 bits. Sin embargo, la posición de estas tablas se mantiene en la cadena de punteros de forma que el mismo código funcione en arquitecturas de 32-bits y 64-bits. El kernel mantiene una posición para cada uno de estos directorios ajustando a 1 el número de entradas en ella y haciendo corresponder esta única entrada dentro de la posición adecuada en el Directorio global de páginas.

Las entradas de los Directorios de páginas y de las Tablas de páginas tienen la misma estructura. Cada una de ellas contiene los siguientes campos:

- Indicador `Presente` que si esta activo indica que la página correspondiente está en memoria.
- Campo con los 20 bits más significativos de la dirección física del marco de página.
- Indicador de `Accedida`, activado cada vez que el unidad de paginación direcciona el correspondiente marco.
- Indicador `Sucio`, aplicado sólo a PTEs, se activa cada vez que se realiza una operación de escritura sobre el marco de página.
- Indicador `Lectura/Escritura`, contiene los derechos de acceso de la página.
- Indicador `Usuario/Supervisor`, contiene el nivel de privilegio necesario para acceder a la página o tabla.
- Indicadores `PCD` y `PWT`, que controlar la forma en que la página o tabla de páginas es gestionada por la caché hardware.
- Indicador `Tamaño de Página`, aplicado solo a Directorios de Páginas.
- Indicador `Global`, aplicado solo a Directorios de Páginas, introducido en el Pentium Pro para evitar la limpieza de páginas de la caché frecuentemente accedidas.

Paginación extendida

Los microprocesadores 80x86 a partir del Pentium introducen la *paginación extendida* que permite marcos de páginas de 4MB en lugar de 4KB. La paginación extendida se utiliza para traducir grandes rangos de direcciones lineales continuas, de forma que no sea necesario utilizar directorios intermedios. Así, ahorra memoria y entradas de TLB.

La paginación extendida se activa ajustando el indicador `Tamaño de página` en una entrada de Directorio de página. En este caso, la unidad de paginación divide la dirección lineal de 32 bits en dos campos: 10 bits para el directorio, y 22 bits para el desplazamiento. La paginación extendida coexiste con la paginación normal. Se activa ajustando el indicador `PSE` del registro cr4.

Protección hardware

La unidad de paginación utiliza un esquema de protección diferente a la unidad de segmentación. Mientras que el procesador 80x86 permite cuatro niveles de privilegio para los segmentos, las páginas y tablas de páginas tienen asociados dos niveles de privilegio controlados por el indicador `User/Kernel`.

Además, en lugar de tres tipos de derechos de acceso (lectura, escritura, y ejecución) asociados con los segmentos, las páginas tiene asociados dos tipos de derechos (lectura y escritura) determinados por el indicador `Read/Write`.

El mecanismo de paginación PAE (*Physical Address Extension*)

La cantidad de RAM que soporta un procesador esta limitada por el número de patillas del bus de direcciones. Los procesadores del Intel hasta el Pentium utilizan direcciones físicas de 32 bits. En teoría se podrían direccionar hasta 4GB de RAM; en la práctica, debido a los requisitos del espacio de direcciones lineales de los procesos en modo usuario, el kernel no puede direccionar más de 1 G de RAM.

Si embargo, los grandes servidores, que pueden ejecutar cientos de procesos simultáneamente, necesitan más de 4 GB de RAM. Para satisfacer estas necesidades, Intel ha incrementado el número de patillas de dirección de 32 a 36. A partir del Pentium Pro, todos los procesadores pueden direccionar hasta 64 GB de RAM. Para ello se introdujo un mecanismo denominado *Extensión de Direcciones Físicas* (PAE). Este se activa, ajustando el bit `PAE` del registro de control `cr4`. El indicador de tamaño de página (`PS`) en la entrada de

directorio de páginas permite tamaños de páginas de 2MB. Para soportar PAE, Intel ha tenido que cambiar el mecanismo de paginación.

El principal problema con PAE es que las direcciones lineales siguen siendo de 32 bits. Claramente, el mecanismo no amplia el espacio de direcciones de un proceso. La ventaja es que el kernel si puede utilizar hasta 64 GB de RAM y por tanto incrementar notablemente el número de procesos en el sistema.

Manejo de las tablas de páginas

Los tipos `pte_t`, `pmd_t`, `pud_t` y `pgd_t` describen respectivamente el formato de la Tabla de páginas, del Directorio Intermedio de Páginas, del Directorio intermedio y del Directorio Global de Páginas. El kernel dispone de varias macros y funciones para:

- Leer o modificar las entradas de las tablas de páginas.
- Consultar/ajustar los indicadores de la PTEs: lectura, escritura, ejecución, sucia, accedida.
- Construir una PTE a partir de una dirección y unos indicadores, y a la inversa.
- Asignar/liberar pte's, pmd's, y pgd's.

Marcos reservados de páginas

El código y datos del kernel se almacena en un grupo de marcos de páginas reservados. Las páginas contenidas en estos marcos no se asignan dinámicamente, ni se intercambian a disco.

Como regla general, el kernel de Linux se instala en RAM desde la dirección física `0x00100000`, es decir, desde el segundo megabyte. El número total de marcos de páginas que necesita depende de cómo se configura el kernel. Una configuración típica produce un kernel que puede cargarse en menos de 2 MB de RAM.

¿Por qué no se carga el kernel en el primer mega disponible de RAM? Esto se debe a las peculiaridades de la arquitectura PC. Por ejemplo:

1. El marco 0 se utiliza por la BIOS para almacenar al configuración hardware detectada durante la prueba de arranque (POST – *Power-On Self-Test*); sin embargo, la BIOS de muchos equipos escriben datos en el incluso después de la inicialización del equipo.
2. Las direcciones físicas `0x000a0000` a `0x000fffff` normalmente están reservadas a las rutina BIOS y proyecta la memoria interna de las tarjetas gráficas ISA. Esta área corresponde con el agujero bien conocido que hay desde los 640KB a 1 MB: las direcciones físicas existen, pero están reservadas, y los correspondientes marcos de páginas no pueden ser utilizados.
3. Otros marcos dentro del primer mega se reservan según el modelo específico del computador. Por ejemplo, el IBM ThinkPad proyecta el marco `0xa0` en el `0x9f`.

Así, para evitar cargar el kernel de Linux en grupos de marcos no contiguos se prefiere saltar el primer mega. Es obvio, que los marcos del primer mega no utilizados se utilizarán por Linux para almacenar páginas asignadas dinámicamente. La Figura 2.16 muestra los primeros megas de RAM. Hemos asumido que el kernel ocupa memos de 1MB de RAM.

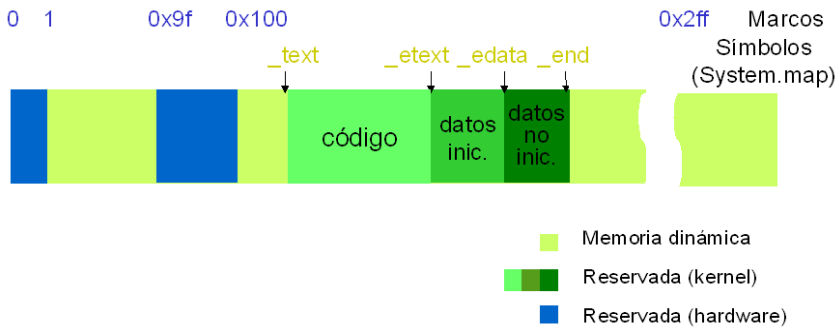


Figura 2.16.- Los primeros 768 marcos (3 MB) de páginas en Linux 2.6.

Tablas de páginas de los procesos

El espacio de direcciones lineales de un proceso se divide en dos partes (parte izquierda de la Figura 2.17):

- Direcciones lineales desde `0x00000000` a `0xbfffffff` pueden ser direccionadas por el proceso en modo kernel o usuario.
- Las direcciones desde `0xc0000000` a `0xffffffff` pueden sólo direccionarse en modo kernel.

Tablas de páginas del kernel

El kernel mantiene un conjunto de tablas de páginas para su uso propio, cuya raíz es el *Directorio Global de Páginas maestro del kernel*. Estas tablas, una vez inicializadas, no son utilizadas nunca ni por los procesos ni por las hebras kernel, si no que las entradas más altas del Directorio Global de Páginas maestro del kernel son el modelo de referencia para las entradas correspondientes de los Directorios Globales de Páginas de cualquier proceso.

Las tablas de páginas del kernel se crean durante la inicialización del sistema. Una vez creadas, tienen la misión de transformar las direcciones lineales que empiezan en `0xc0000000` en la dirección física que empieza en 0 (a excepción de los *direcciones lineales con mapeo fijo*). Esto si la RAM esta por debajo de los 896 MB. Si la RAM esta por encima de los 896 MB, es necesario remapear ciertos intervalos de direcciones ajustando las entradas de las tablas de páginas.

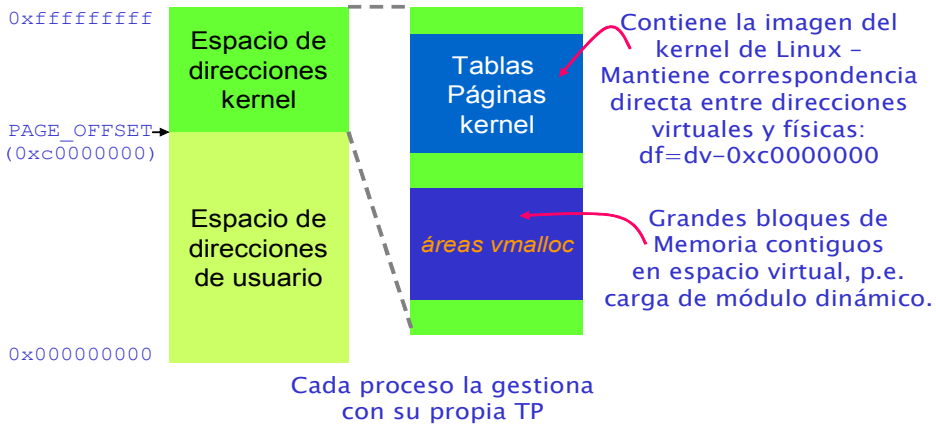


Figura 2.17.- Estructura del espacio de direcciones de un proceso.

2.7.2 Gestión de memoria

En el apartado anterior, vimos como Linux utiliza la circuitería de segmentación y paginación de Intel para traducir direcciones lógicas en direcciones físicas. También, vimos como una porción de RAM es asignada permanentemente al kernel y se utiliza para almacenar el código kernel y las estructuras de datos estáticas del mismo.

El resto de la RAM se denomina memoria dinámica. El rendimiento del sistema completo depende la eficiencia en la gestión de esta memoria. Por ello, todos los sistemas operativos multitarea intentan optimizar el uso de la memoria dinámica, asignándola sólo cuando se necesita y liberándola tan pronto como sea posible.

En este apartado, describiremos cómo el kernel asigna memoria dinámica para su propio uso. Esta gestión va a depender del tipo de uso que se va a hacer de la memoria. Para la asignación de memoria de uso frecuente se va a asignar memoria contigua, por ejemplo, descriptores de procesos. Para asignaciones poco frecuentes, como puede ser la asignación de memoria de un módulo cargado dinámicamente, se utilizan marcos no contiguos. En este

caso, es necesario el uso de las tablas de páginas del kernel. En el apartado siguiente, veremos la asignación de memoria para procesos de usuario.

Gestión de marcos de páginas

Linux sobre Intel opta por el tamaño de página más pequeño, 4 KB, como unidad de asignación de memoria. Esto hace las cosas más simples por dos razones:

a) La circuitería de paginación gestiona de forma más fácil una falta de página si esta tiene un tamaño de 4 KB.
b) La transferencia de datos entre memoria y disco es más eficiente cuando se utiliza un tamaño de 4 KB,en lugar de 4 MB.

El kernel debe mantener la pista del estado actual de cada marco de página. Por ejemplo, debe ser capaz de distinguir marcos de página utilizados para contener páginas de un proceso de los que contienen código y datos del kernel; de forma similar, debe determinar si un marco en memoria dinámica esta libre o no. Esta clase de información de estado se mantiene en una matriz de descriptores, uno por cada marco de página. Los descriptores son de tipo `struct page` y sus campos aparecen en la Tabla 2.16. Como cada descriptor tiene 32 bytes de tamaño, `mem_map` es ligeramente menor del 1% de la RAM.

Tabla 2.16. Los campos del descriptor de página.

Tipo	Nombre	Descripción
`unsigned long`	`flags`	Matriz de indicadores (Tabla 2.17).
`atomic_t`	`_count`	Contador de referencias del marco de página.
`atomic_t`	`_mapcount`	Número de entradas de tabla de página que referencian esta marco (−1 si no hay ninguna).
`unsigned long`	`private`	Disponible para el componente kernel que utiliza esta página. Si esta libre, se utiliza por el sistema amigo.
`struct address_space *`	`mapping`	Utilizado cuando la página se inserta en la caché de páginas, o cuando pertenece a una región anónima.
`unsigned long`	`index`	Usado por varios componentes kernel con varios propósitos.
`struct list_head`	`lru`	Punteros a la lista doble enlazada LRU de páginas.

Sólo describiremos algunos campos (el resto los veremos en los capítulos siguientes):

`_count` Ajustado a -1 si el correspondiente marco de página esta libre; si tiene un valor mayor o igual a 0 si el marco esta asignado a uno o más procesos, o es utilizado por alguna estructura de datos del kernel.

`flags` Una matriz de hasta 32 indicadores (Tabla 2.17) que describen el estado del marco. Por cada indicador `PG_xyz`, se define una macro `PageZxyz` para leer/ajustar su valor. En lo que resta, veremos el uso de algunos de estos indicadores.

Zonas de memoria

En una arquitectura ideal, un marco de página puede almacenar cualquier cosa. Sin embargo, el un computador real, la arquitectura impone restricciones que limitan la forma en que podemos utilizar los marcos. En concreto, Linux debe tratar con dos restricciones hardware de la arquitectura 80x86:

- Los procesadores DMA para los viejos buses ISA tienen una gran limitación: solo pueden direccionar los primero 16 MB de RAM.

- En los procesadores actuales de 32 bits con una gran cantidad de RAM, la CPU no puede acceder a toda la memoria física dado que su espacio de direcciones lineales es muy pequeño.

Table 2.17. Indicadores que describen el estado de un marco de página.

Nombre	Descripción
PG_locked	Página bloqueada (p. ej., involucrada en operación de E/S a disco).
PG_error	Se ha producido un error de E/S durante la transferencia la página.
PG_referenced	Páginas accedida recientemente.
PG_uptodate	Activado al completar una operación de lectura de disco con éxito.
PG_dirty	La página ha sido modificada
PG_lru	Página en la lista de páginas activa o en la de inactivas.
PG_active	Página en la lista activa.
PG_slab	El marco de la página esta en una “tableta”.
PG_highmem	El marco pertenece a la zona ZONE_HIGHMEM.
PG_checked	Utilizada por Ext2 o Ext3.
PG_arch_1	No se utiliza en la arquitectura 80x86.
PG_reserved	Marco reservado para código kernel o es inutilizable.
PG_private	El campo private del descriptor contiene datos útiles.
PG_writeback	El método writepage está escribiendo la página en disco.
PG_nosave	Utilizada para suspender/reanudar el sistema.
PG_compound	Marco manejado por el mecanismo PAE.
PG_swapcache	La página pertenece a la cache de intercambio.
PG_mappedtodisk	Todos los datos del marco corresponden a bloques asignados en disco.
PG_reclaim	Página marcada para escribirse en disco y recuperarla.
PG_nosave_free	Utilizada para suspender/reanudar el sistema.

Para solventar estas limitaciones, el kernel 2.6 particiona la memoria física de cada nodo de memoria en tres *zonas* (Cada zona tiene su propio descriptor de zona). En una arquitectura UMA (Acceso a Memoria Uniforme) 80x86 estas zonas son:

ZONE_DMA - Contiene marcos por debajo de los 16 MB. Es decir, que pueden ser utilizados por dispositivos basados en ISA.

ZONE_NORMAL - Contiene marcos a partir de los 16MB y debajo de los 896 MB. Es decir, contiene los marcos a los que el kernel no puede acceder directamente a través del mapeo lineal de los 4 GB.

ZONE_HIGHMEM - Contiene marcos a partir de los 896 MB. Esta zona esta vacía en arquitecturas de 64 bits.

Cuando el kernel invoca a la función para asignar memoria, debe especificar la zona que contiene los marcos solicitados.

La bolsa de marcos de páginas reservadas

Las solicitudes de memoria pueden satisfacerse de dos forma. Si hay suficiente memoria libre, una solicitud se satisface inmediatamente. En otro caso, se debe recuperar memoria, y el camino de código kernel que realiza la solicitud debe bloquearse hasta que se libere memoria.

Sin embargo, algunos caminos de control kernel no pueden bloquearse mientras se asigna memoria, por ejemplo, cuando se maneja una interrupción o se ejecuta código en una sección crítica. En estos casos, el camino de control kernel invoca una *solicitud de asignación de memoria atómica* (nunca se bloquea, y si no hay memoria, la asignación falla).

Aunque no hay forma de asegurar que una solicitud atómica de memoria falle, el kernel trata de minimizar la probabilidad de este evento. Para ello, el kernel reserva una bolsa de marcos de página para asignaciones atómicas que se utiliza en condiciones de escasa memoria. La cantidad de memoria reservada se almacena en la variable `min_free_kbytes`, inicializada en el arranque del sistema y que tiene un valor dependiente de la cantidad de memoria mapeada en los 4 GB de direcciones lineales del kernel, esto es, depende de marcos incluidos en las zonas `ZONE_DMA` y `ZONE_NORMAL`:

$$\text{Tamaño del depósito reservado} = \lfloor \sqrt{16 * \textit{memoria mapeada directamente}} \rfloor \text{(Kilobyte)}$$

Sin embargo, inicialmente esta variable no puede ser menor de 128 ni mayor de 65,536. El administrador del sistema puede cambiar la cantidad de memoria reservada escribiendo en el archivo */proc/sys/vm/min_free_kbytes* o invocando la correspondiente llamada `sysctl()`.

Solicitando y liberando marcos de páginas

Los marcos pueden solicitarse haciendo uso de varias funciones y macros ligeramente diferentes. Algunas de ellas son:

`alloc_pages(gfp_mask, order)` y `__get_free_pages(gfp_mask, order)`
: Funciones usadas para solicitar 2^{order} marcos contiguos. El parámetro `gfp_mask` especifica cómo buscar los marcos de páginas.

`__get_dma_pages(gfp_mask, order)`
: Macro utilizada para obtener marcos adecuados para DMA; esta se expande a: `__get_free_pages(gfp_mask | GFP_DAM, order)`

`get_zeroes_page(gfp_mask)`
: Función que invoca a `alloc_pages(gfp_mask |GFP_ZERO, 0)` y entonces rellena el marco obtenido con ceros.

`__free_pages(page, order)` Comprueba el descriptor de marco de página apuntado por `page`; si el marco no esta reservado (es decir, el indicador `PG_reserved` esta a 0), decrementa el campo `count` del descriptor. Si `count` se hace 0, asume que los 2^{order} marcos contiguos que empiezan en la dirección `addr` no se van a utilizar más y los libera.

El algoritmo del Sistema Amigo

El kernel debe establecer una estrategia robusta y eficiente para asignar grupos de marcos contiguos. Para hacerlo, debe tratar con el bien conocido problema de la fragmentación externa. Existen dos formas básicas para evitarla:

1. Hacer uso de la circuitería de paginación para asignar grupos de marcos libres no contiguos en intervalos de direcciones lineales contiguos.
2. Desarrollar una técnica adecuada para mantener la pista de los bloques de marcos contiguos existentes, evitando cuando sea posible la necesidad de dividir un bloque grande libre para satisfacer una solicitud de memoria pequeña.

El kernel prefiere esta segunda estrategia por tres buenas razones:

- En algunos casos, los marcos contiguos son realmente necesarios, ya que las direcciones lineales contiguas no son suficientes para satisfacer la petición. Un ejemplo típico es una solicitud de memoria para búferes que serán asignados a un procesador DMA. Ya que el DMA ignora la circuitería de paginación y accede al bus de direcciones directamente

mientras transfiere varios sectores de disco en una única operación de E/S; los búferes solicitados deben estar localizados en marcos contiguos.

- Incluso si la asignación de marcos contiguos no es estrictamente necesaria, tiene la gran ventaja de dejar las tablas de paginación de kernel sin cambios. Si bien no hay problema en modificar las tablas de páginas, una modificación frecuente de estas produce mayor tiempo de acceso a memoria.
- Trozos grandes de memoria física pueden accederse a través de páginas de 4 MB. Esto reduce las faltas de TLB, aumentando significativamente el TAE.

La técnica adoptada por Linux para solventar la fragmentación externa se basada en el algoritmo *sistema amigo*. En éste todos los marcos libres se agrupan en 11 listas de bloques que contienen grupos de 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, y 1024 marcos contiguos, respectivamente. La dirección física del primer marco de un bloque es múltiplo del tamaño del grupo (p. ej. la dirección inicial de un bloque de 16 marcos es un múltiplo de 16.2^{12}).

Veremos como funciona el algoritmo a través de un ejemplo. Supongamos una solicitud de 128 marcos. Primero se comprueba la lista de 128. Si no existe un bloque de ese tamaño, el algoritmo mira en el siguiente tamaño de bloque, es decir, en la lista de 256. Si existe un bloque de este tamaño, el kernel asigna 128 marcos de los 256 para satisfacer la petición e inserta los 128 restantes en la lista de bloques de 128 marcos libres. Si no hay bloques libres de 256, se mira en el siguiente tamaño, es decir, un bloque de 512 marcos. Si existe este bloque, se asignan 128 marcos de los 512, y se insertan 256 de los 384 restantes en la lista de 256, y el resto de 128 en la lista de este tamaño. Si no hay se mira en la de 1024, y se repite la asignación y reparto del resto libre. Si la lista de bloques de 1024 marcos esta vacía, el algoritmo abandona e indica la condición de error.

La operación inversa, liberar bloques de marcos, da lugar al nombre del algoritmo. El kernel intenta mezclar juntos pares de bloques amigos libres de tamaño *b* en un único bloque de tamaño *2b*. Se considera que dos bloques son amigos si:

- Ambos bloques tiene el mismo tamaño, por ejemplo *b*.
- Están localizados en direcciones físicas contiguas.
- La dirección física del primer marco del primer bloque es un múltiplo de $2xbx2^{12}$.

Este algoritmo es interactivo; si tiene éxito en mezclar bloques liberados, dobla *b* y trata de formar un bloque de tamaño mayor.

Gestión de áreas de memoria

El sistema amigo adopta la página como unidad de medida. Esto esta bien para peticiones relativamente grandes de memoria, pero ¿qué ocurre con peticiones pequeñas de algunos cientos de bytes? Esta claro, que en este caso la asignación de una página completa es un desperdicio de memoria. Para ello, Linux utiliza el *distribuidor tableta* ("slab") que descansa sobre el sistema amigo para obtener marcos de páginas que luego divide.

El distribuidor de memoria tableta agrupa los objetos en cachés (Figura 2.18). Cada caché almacena los objetos del mismo tipo. Por ejemplo, cuando abrimos un archivo, el área de memoria necesitada para almacenar el objeto archivo abierto se obtiene de una caché del distribuidor tableta denominada `filp` (de *file pointer*).

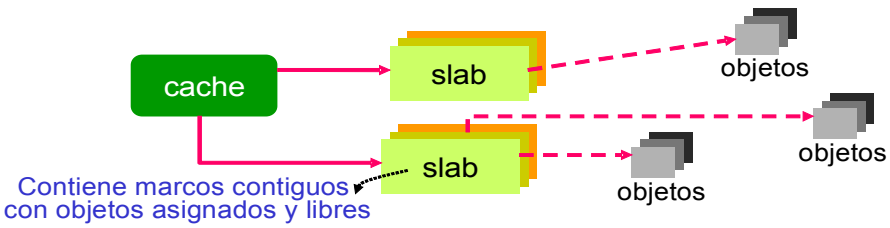


Figura 2.18.- Componentes del distribuidor tableta.

Las cachés utilizadas por este distribuidor pueden verse en Linux en tiempo de ejecución leyendo la información de */proc/slabinfo*. Veamos un fragmento de esta información:

```
% cat /proc/slabinfo
                    Objetos activos
                  |      Objetos asignados
                  |      |         Tamaño del objeto
                  |      |         |   asignaciones tableta activas
                  |      |         |   |     asignaciones tableta totales
                  |      |         |   |     |        Tamaño asignación
                  |      |         |   |     |        |
inode_cache       423370 423556    512 60482 60508    1 :  248   62
dentry_cache      435756 436260    128 14526 14542    1 :  504  126
. . .                                                      |     |
                                                         limite    |
                                                      contador batch
```

2.7.3 Espacio de direcciones de los procesos

La asignación de memoria para los procesos de usuario es muy diferente de la asignación de memoria para las funciones kernel debido a que:

- Las solicitudes de memoria dinámica realizadas por los procesos no se consideran urgentes. Cuando se carga un ejecutable, es improbable que el proceso acceda a todas sus páginas en un futuro inmediato. Por ejemplo, cuando un proceso invoca a `malloc()`, esto no significa que el proceso acceda pronto a la memoria así obtenida. Por tanto, como regla general, el kernel intenta diferir la asignación de memoria dinámica a los procesos en modo usuario.
- Dado que los procesos de usuario no son confiables, el kernel debe estar preparado para atrapar errores de direccionamiento provocados por los procesos en modo usuario.

En este apartado, comenzaremos estudiando como un proceso ve su memoria dinámica. A continuación, describiremos los principales componentes del espacio de direcciones de un proceso. Pasaremos a estudiar, el papel jugado por el manejador de la excepción de falta de página en la asignación diferida de marcos de páginas a los procesos, y, para finalizar, veremos como el kernel crea y borra espacios de direcciones completos.

El espacio de direcciones de un proceso

El espacio de direcciones de un proceso consta del conjunto de direcciones lineales a las que el proceso puede acceder. Cada proceso ve un conjunto diferentes de direcciones lineales. Como veremos en breve, el kernel puede modificar el espacio de direcciones lineales añadiendo, o eliminando, intervalos de direcciones lineales.

El kernel representa intervalos de direcciones lineales mediante unos recursos denominados *regiones de memoria*, que están caracterizadas por una dirección lineal de inicio, una longitud, y unos derechos de acceso. Por razones de eficiencia, tanto la dirección inicial como la longitud de una región de memoria son múltiplos de 4096.

La Tabla 2.18 ilustra algunas llamadas al sistema relacionadas con la manipulación de regiones de memoria. Para entenderlas, veamos algunos escenarios típicos en los que un proceso obtiene nuevas regiones de memoria:

- Cuando tecleamos una orden en la consola, el shell crea un nuevo proceso para ejecutarla. Como resultado, se crea un nuevo espacio de direcciones, es decir, nuevas regiones, para el nuevo proceso.
- Un proceso ejecutándose puede decidir ejecutar un nuevo programa, llamada al sistema *exec()*. En este caso, se mantiene el descriptor de proceso, pero se liberan las antiguas regiones de memoria y se crean una nuevas.

- Un proceso ejecutándose puede realizar una proyección de memoria de un archivo. En este caso, el kernel asigna una nueva región de memoria al proceso para proyectar el archivo.
- Un proceso puede continuar añadiendo datos en su pila de usuario hasta que se haya utilizados todas las direcciones de la región de memoria que mantiene la pila. En estos casos, el kernel debe expandir el tamaño de la región de memoria de pila.
- Un proceso puede crear una región de memoria compartida IPC para compartir datos con otros procesos. En este caso, el kernel asigna una nueva región de memoria al proceso para implementar esta construcción.
- Un proceso puede expandir su área dinámica (*heap*) a través de la función `malloc`(). Como resultado, el kernel expande el tamaño de ésta región.

Tabla 2.18.- Llamadas al sistema relacionadas con la creación/destrucción de regiones.

Llamada al sistema	Descripción
`brk()`	Cambia el tamaño del *heap* de un proceso
`exec()`	Carga un nuevo programa ejecutable
`_exit()`	Termina el proceso actual
`fork()`	Crea un nuevo proceso
`mmap()`	Crea una proyección de memoria para un archivo
`munmap()`	Destruye una proyección de memoria ara un archivo
`shmat()`	Crea una región de memoria compartida
`shmdt()`	Destruye una región de memoria compartida

Como veremos en la sección del "Manejador de la falta de página", es esencial para el kernel identificar las regiones de memoria asignadas a un proceso (es decir, su espacio de direcciones) dado que el Manejador de la falta de página debe distinguir entre los dos tipos de direcciones lineales inválidas que causan su invocación: las causadas por errores de programación, y las causadas por una falta de página. Las últimas direcciones no son inválidas desde el punto de vista del proceso.

El descriptor de memoria

Toda la información relacionada con el espacio de direcciones de un proceso esta incluida en un objeto denominado *descriptor de memoria*. Este objeto esta referenciado por el campo `mm` del descriptor de proceso y es una estructura de tipo `mm_struct`, cuyos campos se muestran en la Tabla 2.19.

Tabla 2.19. Campos del descriptor de memoria

Tipo	Campo	Descripción
`struct vm_area_struct *`	`mmap`	Puntero a la cabeza de la lista de objetos regiones de memoria
`struct rb_root`	`mm_rb`	Puntero a raíz del árbol rojo-negro de objetos regiones de memoria
`struct vm_area_struct *`	`mmap_cache`	Puntero al último objeto región de memoria referenciado
`unsigned long (*) ( )`	`get_unmapped_area`	Método que busca un intervalo disponible de direcciones lineales en el espacio de direcciones del proceso
`void (*)( )`	`unmap_area`	Método invocado cuando se libera un intervalo de direcciones lineales
`unsigned long`	`mmap_base`	Identifica la dirección lineal de la primera región anónima, o proyección de archivo en memoria, asignada

unsigned long	free_area_cache	Dirección a partir de la cual el kernel busca un intervalo libre de direcciones lineales en el espacio de direcciones del proceso
pgd_t *	pgd	Puntero al Directorio Global de Páginas
atomic_t	mm_users	Contador de uso secundario
atomic_t	mm_count	Contador de uso principal
int	map_count	Número de regiones de memoria
struct rw_semaphore	mmap_sem	Semáforo lectura/escritura de regiones de memoria
spinlock_t	page_table_lock	Cerrojo de regiones y Tablas de Páginas
struct list_head	mmlist	Puntero a elementos adyacentes en la lista de descriptores de memoria
unsigned long	start_code	Dirección inicial del código ejecutable
unsigned long	end_code	Dirección final del código ejecutable
unsigned long	start_data	Dirección inicial de datos inicializados
unsigned long	end_data	Dirección final de datos inicializados
unsigned long	start_brk	Dirección inicial de heap
unsigned long	brk	Dirección actual final del heap
unsigned long	start_stack	Dirección inicial de la pila de usuario
unsigned long	arg_start	Dirección inicial de los argumentos de la línea de órdenes
unsigned long	arg_end	Dirección final de los argumentos línea de órdenes
unsigned long	env_start	Dirección inicial de las variables de entorno
unsigned long	env_end	Dirección final de las variables de entorno
unsigned long	rss	Nº de marcos asignados al procesos
unsigned long	anon_rss	Nº marcos asignados a mapeos anónimos
unsigned long	total_vm	Tamaño del espacio de direcciones (nº de páginas)
unsigned long	locked_vm	Nº de páginas “bloqueadas” en memoria
unsigned long	shared_vm	Nº de páginas en mapeos de archivos compartidos
unsigned long	exec_vm	Nº páginas en mapeos de memoria ejecutables
unsigned long	stack_vm	Nº de páginas en la pila de usuario
unsigned long	reserved_vm	Nº de páginas en reserva o regiones especiales de memoria
unsigned long	def_flags	Indicadores de acceso por defecto a regiones
unsigned long	nr_ptes	Nº de tablas de páginas de este proceso
unsigned long []	saved_auxv	Usado al iniciar la ejecución de un programa ELF
unsigned int	dumpable	Indica si un proceso ha producido un “core dump”
cpumask_t	cpu_vm_mask	Máscara bits para conmutación perezosa de TLB
mm_context_t	context	Puntero a tabla para dependiente de la plataforma (en 80x86, dirección de la LDT)
unsigned long	swap_token_time	Cuando es elegible el proceso para tener un “token de intercambio”
char	recent_pagein	Indica si se ha producido recientemente una falta de página principal
int	core_waiters	Nº de procesos ligeros que están volcando el contenido del espacio de direcciones en un core dump
struct completion *	core_startup_done	Puntero a la finalización usada al crear archivo core
struct completion	core_done	Finalización usada cuando se crea un archivo core

rwlock_t	ioctx_list_lock	Cerrojo usado para proteger la lista de contextos de E/S asíncronas
struct kioctx *	ioctx_list	Lista de contextos E/S asíncronos
struct kioctx	default_kioctx	Contextos E/S asíncronos por defecto
unsigned long	hiwater_rss	Nº máximo de marcos propiedad del proceso en todo momento
unsigned long	hiwater_vm	Nº máximo de páginas incluidas en todo momento en las regiones de memoria del proceso

Todos los descriptores de memoria se almacenan en una lista doble enlazada. Cada elemento mantiene la dirección de su adyacente en `mmlist`. El primer elemento de la lista es el `mmlist` del `init_mm`, el descriptor de memoria utilizado por el proceso 0 en la inicialización. La lista esta protegida contra accesos concurrentes en multiprocesadores por el cerrojo `mmlist_lock`.

El campo `mm_users` almacena el número de procesos ligeros que comparten la estructura `mm`. El campo `mm_count` es el contador principal de uso que indica cuando proceso comparten la estructura; cuando se decrementa se comprueba si es 0, en cuyo caso se puede desasignar el descriptor. Este doble contador es necesario pues las hebras kernel puede acceder al espacio descrito por `mm`.

Regiones de memoria

En teoría, todo lo que el kernel necesita para implementar memoria virtual son los tablas de páginas. Sin embargo, las tablas de páginas no son el mecanismo más efectivo para representar grandes espacios de direcciones, especialmente cuanto son dispersos (tiene muchas regiones con sus respectivos huecos). Pongamos un ejemplo, sea un proceso que utiliza 1GB de su espacio de direcciones para una tabla hash e introduce 128 KB de datos en ella. Si suponemos que las páginas son de 4KB y cada entrada es de 4 bytes, entonces la tabla de páginas será de 1 GB / 4 KB * 4 bytes = 1 MB de espacio, que es un orden de magnitud mayor que los datos almacenados en la tabla.

Para evitar esta clase de ineficiencias, Linux representa los espacios de direcciones con *vm-áreas*, en lugar de con tablas de páginas. La idea es dividir el espacio en rangos contiguos de páginas que pueden manejarse de la misma forma. Cada rango es representado por una estructura vm-área. Si un proceso accede a una página para la que no hay traducción en la tabla de páginas, la vm-área que cubre esa página tiene toda la información necesaria para cargar la página. En el ejemplo anterior, un vm-área es suficiente para mapear la tabla hash. Linux implementa las regiones de memoria mediante descriptores de tipo `vm_area_struct` (ver Tabla 2.19).

Table 2.19. Campos del objeto región de memoria

Tipo	Campo	Descripción
struct mm_struct *	vm_mm	Puntero al descriptor de memoria que posee la región
unsigned long	vm_start	Primera dirección lineal dentro de la región.
unsigned long	vm_end	Primera dirección lineal después de la región.
struct vm_area_struct *	vm_next	Siguiente región del proceso en la lista
pgprot_t	vm_page_prot	Permisos de acceso de los marcos de la región
unsigned long	vm_flags	Indicadores de la región

`struct rb_node`	`vm_rb`	Datos para el árbol rojo-negro
`union`	`shared`	Enlaces a las estructuras de datos usadas por el mapeo inverso
`struct list_head`	`anon_vma_node`	Punteros a la lista de regiones de memoria anónimas
`struct anon_vma *`	`anon_vma`	Puntero a la estructura `anon_vma`
`struct vm_operations_struct*`	`vm_ops`	Puntero a los métodos de la región de memoria
`unsigned long`	`vm_pgoff`	Desplazamiento en el archivo proyectado. Para páginas anónimas es 0 ó =`vm_start/PAGE_SIZE`
`struct file *`	`vm_file`	Puntero al objeto archivo del archivo mapeado, si existe
`void *`	`vm_private_data`	Puntero a datos privados de la región de memoria
`unsigned long`	`vm_truncate_count`	Usado cuando liberamos un intervalo lineal de direcciones un mapeo de memoria no lineal

Cada descriptor de región de memoria identifica un intervalo lineal. El campo `vm_start` contiene la primera dirección lineal del intervalo, mientras que `vm_end` contiene la primera dirección lineal fuera del intervalo. Por tanto, `vm_end - vm_start` denota la longitud de la región de memoria. El campo `vm_mm` apunta al descriptor de memoria `mm_struct` del proceso que posee la región. Describiremos más tarde los otros campos de la estructura.

Las regiones de memoria de un proceso nunca se solapan, y el kernel intenta unir regiones cuando se asigna una nueva región a continuación de una existente. Dos regiones adyacentes se pueden fundir si sus derechos se acceso son iguales.

Para entender mejor cómo el kernel utiliza las vm-áreas, tomemos el ejemplo de la Figura 2.19. Este muestra un proceso que mapea los 32 KB primeros (8 páginas) del archivo */etc/termcap* en la dirección virtual `0x2000`. En ella, vemos como el descriptor del proceso apunta a la estructura `mm` que apunta al vm-área (suponemos solo una por simplicidad) y a la tabla de páginas que esta inicialmente vacía. La Figura también representa la zona de memoria virtual del proceso que mapea la vm-área, y el archivo en disco que estamos mapeando en memoria.

Supongamos el proceso intenta leer una palabra en la dirección `0x6008`, como muestra la flecha marcada con la etiqueta (1). Como la tabla de páginas esta vacía, se produce una falta de página. En respuesta a esta falta, el kernel busca la vm-area del proceso actual que cubre la dirección que ha producido la falta. En nuestro caso, encuentra que la única región existente, cuyo rango es `0x2000-0xa000`, cubre la dirección. Calculando la distancia desde el inicio del área mapeada, el kernel encuentra que el proceso intenta acceder a la página 4 (⌊`0x6008-0x2000/4096`⌋=4). Como el vm-área mapea un archivo, Linux inicia la lectura de disco, que muestra la flecha etiquetada con (2). Asumimos que el vm-área mapea los primeros 32KB del archivo, de forma que los datos para la página 4 se pueden encontrar en el desplazamiento `0x4000` hasta el `0x5fff`. Cuando se lee esta información, se copia en el marco de pagina mostrado en la etiqueta (3). Para finalizar, el kernel actualiza la tabla de páginas con una entrada que mapea la página virtual `0x6000` en el marco que contiene los datos. En este punto, el proceso reanuda su ejecución.

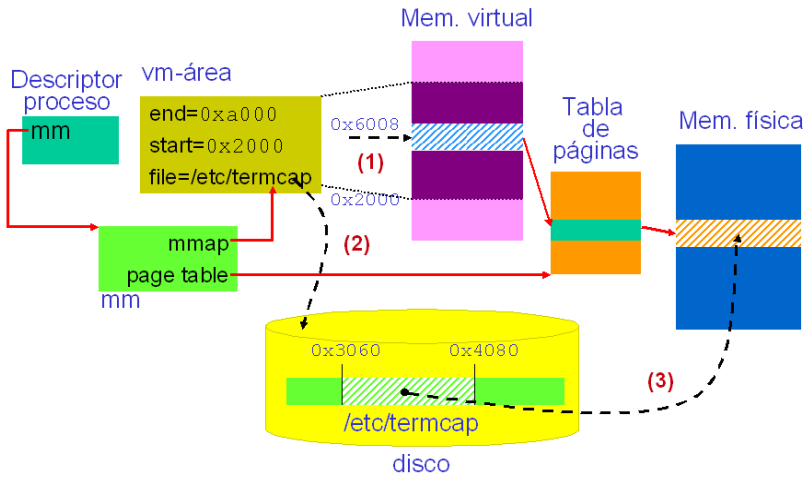


Figura 2.19.- Ejemplo de mapeo de un archivo en una vm-área.

Como muestra el ejemplo, las vm-áreas permiten a Linux re-crear las entradas de las tablas de páginas para una dirección que esta mapeada en el espacio de direcciones del proceso. Esto significa que la tabla de páginas puede tratarse casi como una caché: si esta presenta la traducción para una página concreta, el kernel la utiliza; si esta no está, el kernel la crea a partir del vm-área que la contiene. Tratar las tablas de páginas de esta forma da bastante flexibilidad ya que la traducción de páginas limpias puede eliminarse cuando se desee. La traducción para páginas sucias puede eliminarse si están respaldadas por el archivo (no por el espacio de intercambio). Antes de la eliminación, deben limpiarse escribiendo su contenido en el archivo. Como veremos en breve, la conducta tipo cache de las tablas de páginas suministra los fundamentos para el mecanismo "copy-on-write" que utiliza Linux.

Los principales campos de un vm-área, agrupados por uso, son:

- *Rango de direcciones* – describe en rango de direcciones que cubre en la forma dirección de inicio y de fin.
- *Indicadores VM* – una palabra que contiene varios bits que indica los derechos de acceso (`VM_READ`, `VM_WRITE`, `VM_EXEC`, ... indican si el proceso puede leer, escribir, ejecutar, ... la memoria virtual mapeada por el vm-área). También, los bits `VM_GROWSDOWN` y `VM_GROWSUP`, que indica si la región crecer hacia abajo o hacia arriba, respectivamente (como veremos en breve, esto permite crecer la pila de usuario dinámicamente).
- *Información de enlazado* – punteros para mantener la lista de vm-áreas y al descriptor de memoria que contiene la región.
- *Operaciones VM y datos privados* – Contiene el puntero `vm_ops` a operaciones VM que es un puntero a un conjunto de funciones que definen cómo son manejados determinados eventos, tal como las faltas de página. También, contiene un puntero a datos privados que son utilizados por esta funciones como un gancho para mantener información específica del vm-área. Los métodos aplicables a una región pueden ser:
  - `open()` - se invoca al añadir una región al conjunto de regiones del proceso.
  - `close()` - se invoca cuando se elimina una región de la lista.
  - `nopage()` - invocada por el manejador de la excepción falta de página cuando un proceso intenta acceder a una página no presente en RAM cuya dirección lineal pertenece a la región.
  - `populate()` - invocado para ajustar las entradas de tabla de páginas para la direcciones lineales de la región. Usado principalmente en mapeos no lineales.
- *Información del archivo mapeado* – si el vm-área mapea una porción de un archivo, este componente almacena el archivo y el desplazamiento para localizar los datos.

Se puede observar que esta estructura no contiene contador de referencias. Esto es debido a que esta estructura pertenece a una única estructura `mm`, que ya tiene un contador de referencias. Es decir, cuando el contador de referencias de la estructura mm llega a 0, la estructuras vm-áreas que cuelgan de ella ya no son necesarias.

Las operaciones VM dan a las vm-áreas características de orientación a objetos ya que diferentes tipos de vm-áreas pueden tener diferentes manejadores para los eventos de memoria virtual. Así, Linux permite que cada sistema de archivos, dispositivo de caracteres, y, más generalmente, cada objeto que puede ser proyectado en el espacio de direcciones de usuario con `mmap()`, suministre su propio conjunto de operaciones VM: `open()`, close(), `nopage()`, o `populate()`. Linux implementa funciones por defecto para estas operaciones. Estas versiones por defecto se utilizan si el puntero esta a NULL. Por ejemplo, si la función `nopage()` apunta a NULL, Linux maneja las falta de página creando una página anónima, que es una página privada del proceso cuyo contenido se inicializa a cero.

Estructuras de datos de regiones de memoria

Todas las regiones de memoria de un proceso se enlazan en una lista simple. Las regiones aparecen en la lista en orden ascendente por dirección de memoria. Si embargo, entre dos regiones consecutivas puede existir un área de direcciones de memoria no utilizadas. El campo `vm_next` de cada elemento `vm_area_struct` apunta al siguiente elemento de la lista. El kernel encuentra las regiones de memoria mediante el campo `mmap` del descriptor de memoria del proceso, que apunta al campo `vm_next` del primer descriptor de región de memoria de la lista. El campo `map_count` contiene el número de descriptores de memoria que mantiene el proceso. Por defecto un proceso puede tener hasta 65,536 regiones, pero el administrador puede cambiar este valor escribiendo el nuevo límite en el archivo */proc/sys/vm/max_map_count*. La Figura 2.20 ilustra la relación entre el espacio de direcciones de un proceso, sus descriptores de memoria, y la lista de regiones de memoria.

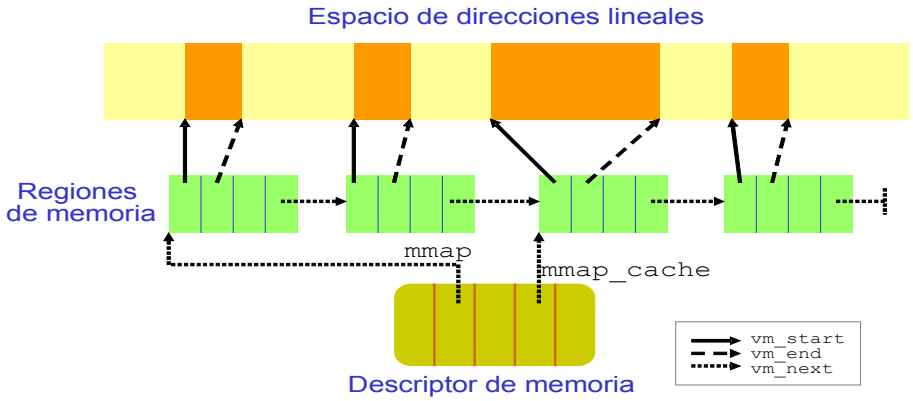


Figura 2.20.- Descriptores relacionados con el espacio de direcciones de un proceso.

Una operación frecuente realizada por el kernel es la búsqueda de una región de memoria que contiene una dirección lineal específica. Como la lista esta ordenada, la búsqueda termina tan pronto se encuentra la región de memoria que finaliza después de la dirección buscada.

Sin embargo, el uso de la lista es conveniente sólo si el proceso tiene pocas regiones de memoria, por ejemplo, menos de unas pocas decenas. Las búsquedas, inserciones, y borrados de elementos, en la lista involucra un número de operaciones cuyo tiempo es proporcional linealmente a la longitud de la lista.

Aunque los procesos de Linux utilizan pocas regiones de memoria, existen grandes aplicaciones, por ejemplo, bases de datos orientas a objetos, que pueden utilizar cientos o

miles de regiones. En estos casos, la gestión de regiones es ineficiente, de forma que el rendimiento de las llamadas al sistema relacionadas con memoria se degrada hasta un punto intolerable. Cuando un proceso tiene un gran número de regiones, Linux almacena sus descriptores en una estructura denominada *árbol rojo-negro* (*Red-Black tree*). Un árbol rojo-negro es un árbol binario de búsqueda en que cada nodo tiene un atributo de color, rojo o negro, y se adhiere al ciertas reglas (más detalles en http://es.wikipedia.org/wiki/%C3%81rbol_rojo-negro) que aseguran que para *n* nodos el árbol tiene una altura máxima de *2.log(n+1)*. La principal ventaja de este tipo de árboles es que ofrecen un tiempo del peor caso garantizado para la inserción, el borrado y la búsqueda.

La principal desventaja de los árboles rojo-negro, es su complejidad de gestión comparada con las listas. Por ello, Linux utiliza las listas siempre y cuando el número de elementos sea pequeño. Cuando el número de regiones de memoria sobrepasa el límite `AVL_MIN_MAP_COUNT` (normalmente 32 elementos), Linux comienza a utilizar los árboles. Por ello, el descriptor de memoria de un proceso incluye el campo `mm_rb` que apunta a un árbol rojo-negro.

Derechos de acceso a regiones de memoria

Antes de continuar, debemos clarificar la relación entre una página y una región de memoria. Como ya hemos indicado, una región de memoria consta de un conjunto de páginas que tiene números de páginas consecutivos.

Ya indicamos como asociados a una página existe dos clases de indicadores (los indicadores almacenados en la PTE y los almacenados en el descriptor de página). Ahora vamos a introducir una tercera clase de indicadores, los indicadores asociados con las páginas de una región de memoria. Estos se almacenan en el campo `vm_flags` del descriptor `vm_area_struct`. Algunos indicadores ofrecen información kernel sobre todas las páginas de la región, tales como qué contienen y qué derechos tiene el proceso al acceder a cada página (`VM_READ`, `VM_WRITE`, ...), otros describen la propia región e indican si ésta puede crecer y cómo (`VM_GROWSDOWN`, ...).

Los derechos de acceso incluidos en un descriptor de región pueden combinarse de forma arbitraria. Es posible, por ejemplo, permitir que las páginas de una región se puedan leer pero no ejecutar. Para implementa la protección de forma eficiente, los permisos de escritura, lectura, y ejecución de una página de una región deben duplicarse en la PTE correspondiente, de forma que la MMU pueda realizar las comprobaciones pertinentes.

Los valores iniciales de los indicadores de la Tabla de Páginas se almacenan en el campo `vm_page_prot` del vm-área. Cuando se añade una página, el kernel ajusta los indicadores de la PTE correspondiente de acuerdo con este campo.

Hay que hacer constar, que la traducción de los derechos de acceso a memoria de las regiones en los bits de protección de página no es directa, y hay que aplicar ciertas reglas dependiendo de la arquitectura.

2.7.4 El manejador de la excepción falta de página

El manejador debe distinguir entre las excepciones causadas por errores de programación y las provocadas por referencias a páginas que pertenecen a un proceso pero que aún no han sido asignadas.

Los descriptores de regiones de memoria permiten que el manejador realice su trabajo correctamente. La función `do_page_fault()`, que es la rutina de servicio de la excepción falta de página para la arquitectura x86, compara la dirección lineal que provoca la falta de página con las regiones de memoria del proceso actual. Así, puede determinar la forma adecuada de tratar la excepción de acuerdo al esquema ilustrado en la Figura 2.21. En la

práctica, es algo más complicado pues el manejador debe analizar algunos subcasos particulares y distinguir varias clases de accesos legales.

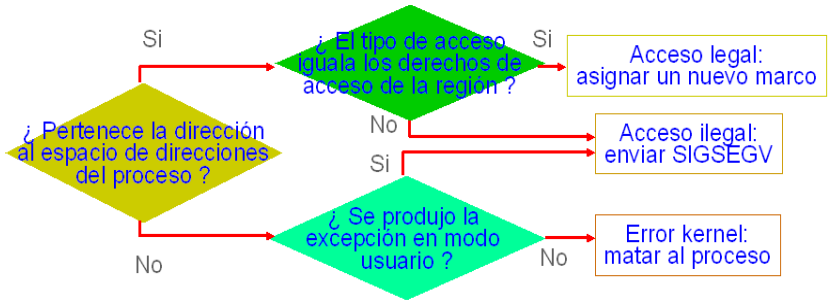


Figura 2.21.- Esquema general del manejador de falta de página.

La función `do_page_fault()` acepta como parámetros de entradas los siguientes:

- La dirección `regs` de la estructura `pt_regs` que contiene los valores de los registros del microprocesador cuando se produce la excepción.
- Un `error_code` de 3 bits almacenados por la unidad de control en la pila al producirse la excepción, y que tienen el siguiente significado:
  - Bit 0: si esta limpio, la excepción se produjo por el acceso a una página que no esta presente (el indicador `Present` de la PTE esta limpio); si esta activo, la excepción se produjo por un derecho de acceso inválido.
  - Bit 1: si esta limpio, se produjo por un acceso de lectura o ejecución; si esta activo, por un acceso de escritura.
  - Bit 2: si esta limpio, se produjo mientras el procesador estaba en modo kernel; si activo, estaba en modo usuario.

Lo primero que hace la función es recuperar la dirección lineal que produjo la falta del registro `cr2`, donde fue almacenado por la unidad de control. También se asegura que las interrupciones están habilitadas si lo estaban con anterioridad o si la CPU se estaba ejecutando en modo virtual 8086, y salva el puntero al descriptor de la tarea actual en `tsk`.

Como se muestra en la cima de la Figura 2.22, `do_page_fault()` comprueba si la dirección lineal que genera la falta pertenece al cuarto gigabyte. Si la excepción se produjo cuando el kernel intentaba acceder a un marco de página inexistente, se salta a la etiqueta `vmalloc_fault` que se encarga de manejar faltas provocadas en el acceso a áreas de memoria no continuas. En otro caso, salta a la etiqueta `bad_area`, que describiremos en el apartado “Manejando una falta fuera del espacio de direcciones”.

A continuación, comprueba si la excepción se produjo mientras el kernel estaba ejecutando alguna rutina crítica o una hebra kernel:

```
if (in_atomic( ) || !tsk->mm)
        goto bad_area_nosemaphore;
```

La macro `in_atomic()` devuelve 1 si la falta se produjo mientras se da una de las siguientes condiciones:

- el kernel estaba ejecutando un manejador de interrupciones o una función aplazable.
- El kernel estaba ejecutando un región crítica con la apropiación deshabilitada.

Si se produjo en algunos de estos casos, el manejador no compara la dirección lineal con las regiones del proceso actual. Ni las hebras kernel, ni los manejadores de interrupciones, ni las funciones aplazables, usan direcciones lineales por debajo de los tres gigas, dado que se podría bloquear al proceso actual (como veremos en breve).

Si la falta no se produjo en un manejador de interrupciones, función aplazable, región crítica, o hebra kernel, la función debe inspeccionar las regiones de memoria del proceso para

ver si la dirección esta incluida en el espacio de direcciones del proceso para lo cual adquiere el semáforo de lectura/escritura `mmap_sem`.

```
vma = find_vma(tsk->mm, address);
    if (!vma)
        goto bad_area;
    if (vma->vm_start <= address)
        goto good_area;
```

Si `vma` es `NULL`, no hay región que contenga a `address`, por lo que la dirección es mala. El caso contrario, si la primera región que acabe después de `address`, esta incluye la dirección, por lo que salta a la etiqueta `good_area`.

Si no se cumplen ninguna de las dos condiciones, se debe aún realizar una comprobación más ya que la dirección de la falta podría estar causada por una instrucción `push` o `pusha` sobre la pila de usuario.

Vamos a realizar un inciso para indicar como se mapea la pila en la región de memoria. Cada región que contiene una pila se expande hacia las direcciones inferiores: tiene activo su indicador `VM_GROWSDOWN`, por lo que `vm_end` se mantiene fijo y `vm_start` puede decrecer. Los límites de región incluyen, pero no de forma precisa, el tamaño actual de la pila de usuario. Los factores para esta indefinición son:

- el tamaño de la región es múltiplo de 4KB (incluye páginas completas), mientras que la pila tiene un tamaño arbitrario.
- Los marcos de página asignados a la región nunca se liberan hasta que la región se borra. En concreto, el valor del campo `vm_start` que incluye la pila solo puede decrecer, nunca crecer. Incluso si un proceso ejecuta una serie de instrucciones `pop`, el tamaño de la pila no cambia.

Queda claro cómo un proceso que ha rellenado el último marco de pila puede provocar una falta de página: `push` referencia una dirección fuera de la región (sobre un marco inexistente). Como esta excepción no se produce por un error de programación, debe manejarse de forma separada por el manejador de falta de página.

El caso que acabamos de describir se maneja por `do_page_fault()` de la forma

```
if (!(vma->vm_flags & VM_GROWSDOWN))
        goto bad_area;
    if (error_code & 4      /* User Mode */
        && address + 32 < regs->esp)
            goto bad_area;
    if (expand_stack(vma, address))
        goto bad_area;
    goto good_area;
```

si `VM_GROWSDOWN` esta activo y la excepción se produjo en modo usuario, se comprueba si `address` es menor que el puntero de pila `regs->esp` (debería ser solo un poco menor). Dado que solo un conjunto pequeño de instrucciones en ensamblador (tales como `pusha`) realizan un decremento del registro `esp` después de un acceso a memoria, se permite al proceso un intervalo de tolerancia de 32 bytes. Si la dirección es suficientemente alta (dentro del intervalo tolerado), se invoca al código de `expand_stack()` que comprueba si al proceso le esta permitido expandir la pila y el espacio de direcciones. Si es así, se ajusta `vm_start` de `vma` a `address` y retorna 0. Si no, se devuelve `-ENOMEM`.

Observar que el código anterior salta la comprobación de tolerancia si esta `VM_GROWSDOWN` activo y la excepción no se produce en modo usuario. Estas condiciones significan que el kernel esta direccionando la pila de usuario y este código siempre expande la pila.

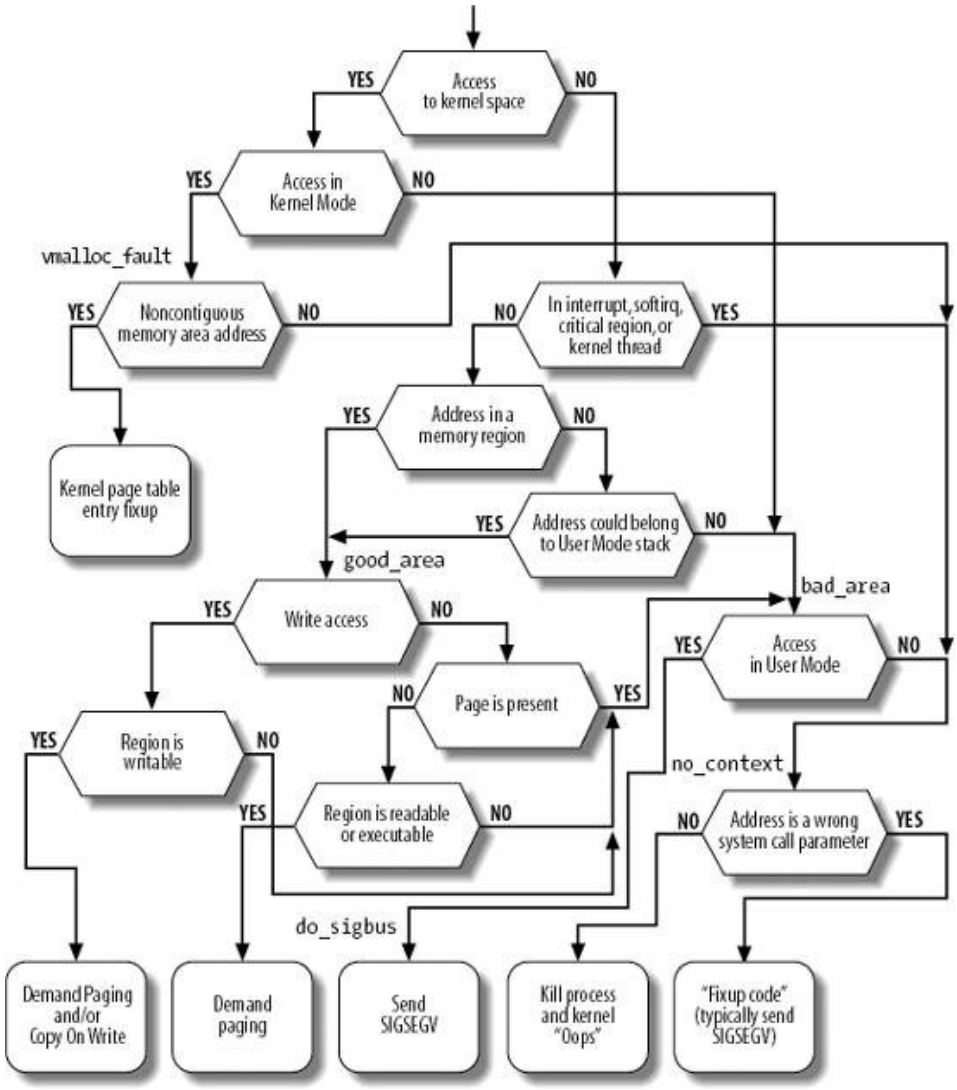


Figura 2.22.- Diagrama de flujo del manejador de falta de página.

Manejo de la falta fuera del espacio de direcciones

Si `address` no pertenece al espacio de usuario, `do_page_fault()` ejecuta el estamento `bad_area`. Si el error se produjo en modo usuario, envía la señal `SIGSEGV` al proceso actual y termina. Si la excepción ocurre en modo kernel, existen dos alternativas:

- La excepción se produjo mientras se usaba una dirección lineal pasadas al kernel como parámetro de una llamada al sistema – en este caso, se envía la señal `SIGSEGV` al proceso actual o se termina el manejador de la llamada con el error adecuado.
- La excepción de debe a un error kernel – la función imprime un volcado completo de los registros de la CPU y de la pila kernel en consola y sobre un búfer de mensajes del sistema, y mata al proceso actual. Este es el error "*Kernel oops*", denominado así por el mensaje que se muestra. Los valores volcados pueden utilizarse para reconstruir las condiciones que activaron el error y poder así corregirlo.

Manejando una falta dentro del espacio de direcciones

Si la dirección pertenece al espacio de direcciones del proceso, la función sigue por la etiqueta `good_area`. Si se produjo por un acceso de escritura, se comprueba si se puede escribir en la

región. So no lo es, se salta a la etiqueta `bad_area_code`; si lo es, se pone a 1 la variable local `write`.

Si la excepción se produjo por un acceso de lectura o ejecución, se comprueba si la página esta presente en RAM. En este caso, la excepción se produjo al intentar acceder a un marco privilegiado (con el indicador `User/Supervisor` activo) en modo usuario, por lo que se salta a la etiqueta `bad_area_code`. Si la página no esta presente, se comprueba si la región es legible o ejecutable.

Si los derechos de acceso de la región igualan el tipo de acceso que causó la excepción, se invoca a la función `handle_mm_fault()` para asignar un nuevo marco. Esta función, si tiene éxito en la asignación, devuelve:

- `VM_FAULT_MINOR` – indica que la falta de página ha sido manejada sin bloquear al proceso actual, es una *falta menor*.
- `VM_FAULT_MAYOR` – la falta de página ha forzado al proceso actual a dormir (lo más probable es que se empleó el tiempo en rellenar el marco de página con los datos del disco), es una *falta mayor*.

La función también retorna `VM_FAULT_OOM`, si no hay suficiente memoria, o `VM_FAULT_SIGBUS`, para cualquier otro error. Para este último error, se envía la señal `SIGBUS` al proceso. Si se produce cualquier otro error, normalmente se mata al proceso.

La función `handle_mm_fault()` actúa sobre cuatro parámetros:

- `mm` – puntero al descriptor de memoria del proceso que se estaba ejecutando cuando se produjo la excepción.
- `vma` – un puntero al descriptor de la región de memoria que contiene la dirección lineal que produjo la falta.
- `address` – la dirección lineal que produjo la falta.
- `write_access` – con valor 1 si `tsk` intentó escribir `address`, y, a 0, si intentó leer o ejecutar.

La función comienza por comprobar si existen el directorio intermedio de páginas y la tabla de páginas utilizados para mapear `address`. Incluso si `address` pertenece al proceso, las correspondientes tablas de páginas podrían no estar asignadas, y por tanto, habría que hacerlo. Si estas operaciones tienen éxito, la variable `pte` apunta a la entrada de la tabla de páginas que contiene a `address`. Tras lo cual, se invoca a `handle_pte_fault()` para determinar como asignar el nuevo marco:

- Si la página accedida no esta presente, el kernel asigna un marco y lo inicializa, es decir, implementamos la demanda de página.
- Si esta presente pero marcada como sólo-lectura, el kernel asigna un nuevo marco y inicializa su contenido copiando los datos del viejo marco, implementa la técnica copy-on-write.

Demanda de página

Como ya vimos en SOI, el término *demanda de página* denota la técnica de asignación dinámica de memoria que consiste en diferir la asignación de marcos de páginas hasta que éstos sean referenciados, momento en el que se produce una falta de página.

Esta técnica tiene la ventaja de hacer una mejor gestión de la memoria principal (tener más marcos libres) y poder así tener memoria para nuevos procesos o asignar más a los existentes. El precio a pagar por ello es la sobrecarga introducida por el manejador de faltas de páginas. Afortunadamente, el principio de localidad asegura que una vez que el proceso ha iniciado su ejecución con un grupo de páginas, permanece sin referencias adicionales durante cierto tiempo. Así, podemos considerar que el evento falta de página "no es muy frecuente".

Una página puede no estar "presente" en memoria principal bien por que nunca ha sido accedida por el proceso, bien habiendo sido usada por el proceso ha sido reclamada por el kernel (ver apartado "recuperación de páginas").

En ambos casos, el manejador de FP (falta de página) debe asignar un nuevo marco al proceso. Si embargo, la forma de inicializar la página depende de si la página fue o no previamente accedida. En particular:

1. Si la página nunca ha sido accedida (esté o no mapeada en un archivo de disco), el kernel reconoce esta situación porque la entrada de la tabla de páginas esta a cero.
2. La página pertenece a un mapeo de archivo no lineal (mapeo en el que las páginas de memoria no están proyectadas en páginas secuenciales del archivo). El kernel reconoce este caso pues el indicador `Present` esta a cero, y el `Dirty` a 1.
3. La página fue accedida por el proceso, pero su contenido esta temporalmente salvado en disco. En este caso, la PTE no esta rellena de ceros, pero los indicadores `Present` y `Dirty` esta a cero.

Así, el manejador de la falta de página puede distinguir los tres casos inspeccionando la TPE que referencia la dirección `address`.

En el caso 1, cuando la página nunca ha sido accedida o proyecta linalmente un archivo de disco, se invoca a la función `do_no_page()`. Hay dos formas de cargar la página ausente, dependiendo de si esta mapeada en disco. La función determina la forma comprobando en método `nopage` de la región de memoria correspondiente, que apunta a la función que carga la página no presente desde el disco a RAM. Por tanto, las posibilidades son:

1. El campo `vma->vm_ops->nopage` no es NULL. En este caso, la región de memoria mapea un archivo de disco y el campo apunta a la función para cargar la página. Este caso es cubierto en el Apartado "Demanda de página para mapeos de memoria".
2. Bien el campo `vma->vm_ops` o el campo `vma->vm_ops->nopage` son NULL. La región de memoria no proyecta un archivo de disco, es un mapeo anónimo. Así, la función invoca a `do_anonymous_page()` para obtener un nuevo marco. Esta maneja de forma separada las escrituras y las lecturas:
   a) En escrituras – se asigna una página y se incrementa el campo `rss` del descriptor de memoria. Se ajusta la PTE con la dirección del marco asignado, que se marca de escritura y sucio.
   b) En lecturas – por razones de seguridad debemos dar al proceso una página rellena de ceros de forma que no pueda leer información de otro proceso. Llevando la demanda de página a sus extremos, no se le da inmediatamente, si no que le asignamos una página ya rellena a ceros, la *página cero*, así diferimos la asignación del marco. La página cero de asigna estáticamente durante la inicialización del kernel. La PTE se ajusta para que apunte a la página cero. Dado que esta página esta marcada como de "no escritura", si un proceso intenta escribir en ella, se activa el mecanismo *copy-on-write*, que describiremos a continuación.

Copia-al-escribir (copy-on-write)

En los primeros sistemas Unix la creación de un proceso era costosa debido ha que había que duplicar el espacio de direcciones del padre en el hijo: asignar e inicializar marcos para las tablas de páginas del hijo, asignar los marcos para las páginas del hijo y copiar el contenido del padre. Esta forma de crear un espacio de direcciones involucra muchos accesos a memoria, consume muchos ciclos de CPU y descartar completamente los contenidos de caché. Además, a veces esto se hacía para que el hijo iniciara su ejecución cargando un nuevo programa, descartando por tanto el espacio de direcciones heredado.

Los sistemas Unix actuales, incluido Linux, utilizan un mecanismo más eficiente denominado *copia-al-escribir* (*COW*). La idea: en lugar de duplicar los marcos de páginas, estos son compartidos por el padre e hijo. Mientras que estos son compartidas, no pueden modificarse. En cuanto uno de los dos proceso intenta escribir en una de las páginas compartidas, se produce una excepción (intento de escritura en una zona de lectura). En este instante, el kernel duplica la página con permiso de escritura. La página original permanece protegida contra escritura, de forma que cuando el otro proceso intenta escribir se comprueba si el proceso es el propietario de la misma, en cuyo caso, se permite la escritura en la página.

El campo `_count` del descriptor de página se utiliza para llevar la cuenta del número de procesos que comparten el marco. Cuando un proceso libera un marco o se ejecuta un COW sobre el, su contador se decrementa. El marco se libera solo cuando el contador llega a -1.

Vamos a describir como Linux implementa el mecanismo COW. Cuando el manejador de la falta de página determina que la excepción se produjo por el acceso a una página presente en memoria, ejecuta las siguientes acciones:

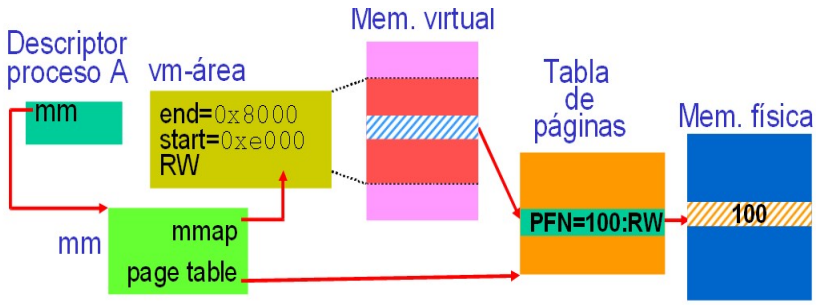
- Determina el descriptor de página del marco referenciado por la PTE involucrada en la excepción.
- Determina si debe duplicarse la página. Si solo un proceso posee la página, no se aplica COW. Realmente la comprobación es más complicada que mirar el contador, pues `_count` también se incrementa cuando la página se inserta en la cache de intercambio. Si no se aplica COW, la página se marca como de escritura, para que no provoque más excepciones.
- Si la pagina es compartida, se copia su contenido del viejo marco de página en el recién asignado. Para evitar condiciones de carrera se invoca a `get_page()` para que incremente el contador de uso del viejo marco. Si la vieja página es la página cero, el nuevo marco se rellena con ceros en la asignación para aumentar la eficiencia. En otro caso, se copia el contenido con la macro `copy_page()`.
- Para finalizar, se copia en la PTE la dirección física del nuevo marco y se invalida el correspondiente registro del TLB.

La mejor forma de ver como el manejador de falta de página coordina su trabajo con el resto del kernel es ver como se manejan las páginas COW. Supongamos un proceso A con una única región de memoria que se puede escribir y que ocupa el rango de direcciones 0x8000 a 0xe000. Supongamos además que solo una página reside en memoria en la dirección 0xa000. La Figura 2.23(a) ilustra este ejemplo. Para simplificar, hemos representado la tabla de páginas a un nivel, en lugar de los cuatro que debe tener. La única región contiene solo la información necesaria para el ejemplo, es decir, las direcciones de inicio y fin, y los derechos de acceso (RW – lectura / escritura). Dado que asumimos que la página es residente en la dirección 0xa000, hay una entrada en la tabla de páginas. La Figura supone que esta página esta cargada en el marco 100. Puesto que la pagina se puede leer y escribir, la PTE tiene los bit correspondientes activos (R y W).

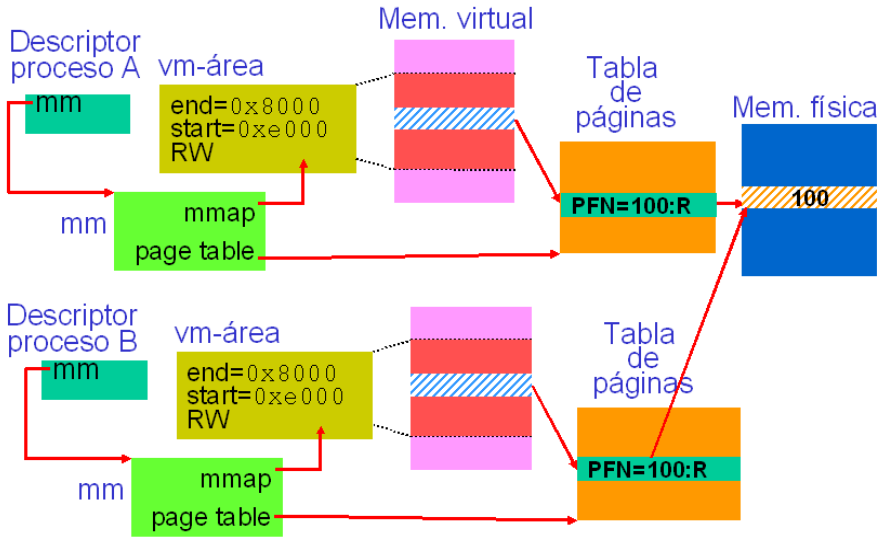
Supongamos que ahora el proceso A invoca a la llamada `clone()` sin especificar el indicador `CLONE_VM`, lo que es equivalente a invocar a `fork()` en un Unix tradicional. La Figura 2.23(b), muestra el estado tras la creación del hijo. Los dos procesos son idénticos, salvo que se ha desactivado el bit W de la PTE de los dos procesos. El hacerlo así permite retrasar lo más posible la copia de páginas con permiso de escritura. Este es el primer paso de COW, cualquier intento de escritura en la página provoca una falta de página. Observar, que el permiso en el vm-área permanece inalterado.

Supongamos que el proceso A intenta escribir en la página. Como la PTE no permite la escritura, se dispara una falta de página. El manejador de la falta de página localiza la vm-área donde se ha producido la falta y comprueba si permite las escrituras. A continuación

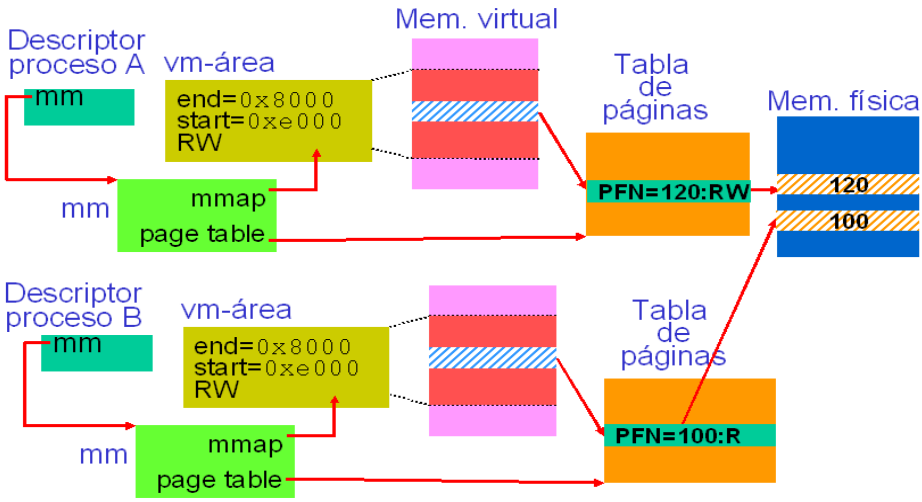
localiza la PTE para ver si existe y si esta presente. Como ambas comprobaciones son ciertas, determina que estamos en un caso de COW. Se comprueba entonces a través del descriptor de la página cuantos procesos la comparten: como en este caso el contador de referencias esta a 2, se decide copiar la página en un nuevo marco, el 131, tras lo cual actualizan las PTEs correspondientes. Ahora, como el proceso A tiene una copia privada de la página, se puede activar el permiso W. En B, se deja desactivada la escritura, y así permanece hasta que intente escribir en ella (Figura 2.23c). En este caso, se siguen los pasos anteriores y cuando se detecte que es el único usuario de ella, se activará el permiso de escritura.



(a) El proceso A tiene mapeada una página



(b) El proceso A crea un proceso hijo B.



(c) Situación final tras una escritura del proceso A.

Figura 2.23.- Ejemplo del mecanismo copia al escribir (COW).

2.7.5 Creando y borrando espacios de direcciones

Creando un espacio de direcciones

Como vimos, las llamadas al sistema `fork()`, `clone()` y `vfork()` invocan a la función `copy_mm()` durante la creación de un proceso para crear un nuevo espacio de direcciones ajustando todas las tablas de páginas y los descriptores de memoria del nuevo proceso. Si bien, como acabamos de ver, las páginas no se duplican, sino que se comparten activando el mecanismo COW.

La función `copy_mm()` copia un crea un espacio de direcciones aunque no se le asigne memoria hasta que el proceso solicite una dirección. Esta función asigna un nuevo descriptor de memoria, almacena su dirección en el campo `mm` del descriptor del nuevo proceso `tsk`, y copia el contenido de `current->mm` en `tsk->mm`. Finalmente, invoca a la función `dup_mmap()` para duplicar las regiones de memoria y las tablas de páginas del proceso padre. Esta función inserta el nuevo descriptor de memoria `tsk->mm` en la lista global de descriptores de memoria. Escanea la lista de regiones del proceso del padre, empezando desde la apuntada por `current->mm->mmap`. Duplica cada descriptor de región `vm_area_struct` encontrado e inserta la copia en la lista de regiones y árbol rojo-negro del hijo.

Después de insertar el nuevo descriptor de región, `dup_mmap()` invoca a `copy_page_range()` para crear, si es necesario, las tablas de páginas necesarias para mapear el grupo de páginas incluidas en la región de memoria y para inicializar las nuevas PTEs. En particular, cada marco de página correspondiente a una página privada de escritura (VM_SHARED desactivado y `VM_MAYWRITE` activo) se marca como de sólo-lectura tanto en el padre como en el hijo, de forma que pueda ser manejada con el mecanismo COW.

Cada proceso suele tener su propio espacio de direcciones, pero los procesos ligeros se crean con el identificador `CLONE_VM` de `clone()` activo. Estos comparten el espacio de direcciones de su creador, es decir, acceden al mismo conjunto de páginas. Como puede verse en el siguiente código:

```
if (clone_flags & CLONE_VM) {
        atomic_inc(&current->mm->mm_users);
        spin_unlock_wait(&current->mm->page_table_lock);
        tsk->mm = current->mm;
        tsk->active_mm = current->mm;
        return 0;
    }
```

Otra llamada que crea el espacio de direcciones de un proceso es `exec()`. Cuando creamos un proceso con `fork()`, el espacio de direcciones del hijo es una copia del espacio del padre. Cuando invocamos a `exec()`, la función realiza las siguientes actividades.

- Copia los argumentos de la llamada en la pila kernel.
- Destruye el espacio de direcciones a nivel de usuario.
- Crea un nuevo espacio de direcciones para el proceso a partir de la información del ejecutable ELF que se pasó como argumento 0 de la llamada al sistema.
- Retorna los argumentos almacenados en la pila al espacio de usuario.

Borrando espacio de direcciones

Cuando un proceso finaliza, el kernel invoca a la función `exit_mm()` que libera el espacio de direcciones del proceso. Si el proceso no es una hebra kernel, la función debe liberar todos los descriptores de memoria y estructuras de datos relacionadas. Primero, comprueba si el indicador `mm->core_waiters` esta activo. Si lo está, el proceso esta volcando el contenido

de memoria en un archivo core. Para evitar la corrupción del archivo core hace uso de un mecanismo de sincronización, denominado *finalización* (*completions* – es similar a los semáforos pero esta pesando para evitar algunas sutiles condiciones de carrera que se dan con los semáforos en multiprocesadores), para serializar el acceso de los procesos ligeros que puedan compartir el mismo descriptor de memoria.

A continuación, la función incrementa el contador de uso principal del descriptor de memoria, limpia el campo `mm` del descriptor del proceso, y pone al proceso en modo TLB perezoso. Finalmente, libera la tabla de descriptores locales, los descriptores de regiones de memoria, y las tablas de páginas. Sin embargo, el propio descriptor de memoria no se librera, dado que se incrementó el contador de uso principal. El descriptor se liberará por la función `finish_task_switch()` cuando el proceso que estamos terminando sea efectivamente desalojado desde la CPU local (recordar la función `schedule()`).

Gestión del heap

Cada proceso Unix posee su propia región de memoria denominada heap, que se utiliza para satisfacer las solicitudes de memoria dinámica. Los campos `start_brk` y `brk` del descriptor de memoria delimitan las direcciones de inicio y fin, respectivamente, de esta región.

Los procesos pueden usar las siguientes funciones de la API para solicitar y liberar memoria dinámica:

- `malloc(size)` – solicita `size` bytes de memoria dinámica; si la asignación tiene éxito, retorna la dirección lineal de la primera dirección.
- `calloc(n, size)` – solicita una matriz de `n` elementos de tamaño `size`. Si tiene éxito, inicializa los componente de la matriz a cero y retorna la dirección lineal del primer elemento.
- `realloc(ptr, size)` – cambia el tamaño de un área de memoria previamente asignado por `malloc` o `calloc`.
- `free(addr)` – libera la región de memoria asignada por `malloc()` o `calloc()` cuya dirección inicial es `addr`.
- `brk(addr)` – modifica directamente el tamaño del heap; `addr` especifica el nuevo valor de `current->mm_brk`, y devuelve la nueva dirección de finalización de la región (el proceso debe comprobar si ésta coincide con el valor de `addr` solicitado).
- `sbrk(incr)` – similar a `brk()`, excepto que el parámetro `incr` especifica un incremento o decremento del tamaño del heap en bytes.

La única de estas funciones que esta implementada como llamada al sistema es `brk()`. El resto están implementadas como funciones de la biblioteca C utilizando `brk()` y `mmap()`.

Cuando un proceso en modo usuario invoca a la llamada `brk()`, el kernel ejecuta la función `sys_brk(addr)`. Esta función, verifica primero si `addr` cae dentro de la región de memoria que contiene el código del proceso. Si es así, retorna inmediatamente ya que heap no se puede solapar con regiones que contengan el código del proceso.

Como esta llamada actúa sobre regiones de memoria, asigna y desasigna páginas completas. Por tanto la función alinea `addr` a un múltiplo de PAGE_SIZE y compara el resultado con el valor del campo `brk` del descriptor. Si el proceso desea recortar el heap, se desmapea con `do_mmap()`. Si lo que desea hacer es aumentarlo, tras comprobar que no excede los límites permitidos, ni invade otro región, se invoca a la función `do_brk()` que efectivamente hace la asignación (en la actualidad, esta función es una versión simplificada de `do_mmap()` que sólo maneja regiones anónimas).

Trabajos en Grupo 2.: Gestión de memoria de diferentes regiones de usuario.

Objetivo formativo: Utilizar las estructuras de datos del kernel para gestionar memoria.
Tareas del grupo: Supongamos el espacio de direcciones virtuales de un proceso ficticio compuesto por una única página de código, otra de datos y una tercera de pila. Indica con un esquema, que muestre las estructuras de datos involucradas, cómo trataría el kernel los siguientes casos:

a) Se produce una falta de página sobre la región de código.

b) Se produce una falta de página sobre la pila de usuario.

Criterio de éxito: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos..

Duración: 20 minutos.

Trabajos en Grupo 2.: Mecanismo COW.

Objetivo formativo: Entender el funcionamiento del mecanismo copia-al-escribir.
Tareas del grupo: Sea un proceso con solo tres regiones de memoria (código, datos y pila) y cada región solo tiene una página de tamaño. Este proceso tiene una variable definida `var` con valor 5. ¿Qué ocurre cuando este proceso crea a otro proceso mediante `fork()` y el nuevo proceso realiza la asignación `var=6`? Dibuje con detalle las estructuras de datos relevantes del kernel..

Criterio de éxito: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.

Duración: 20 minutos.

2.7.6 La cache de páginas

Como vimos en el Tema 1, una caché de disco es un mecanismo software que permite al sistema mantener en RAM algunos datos que están almacenados normalmente en disco, de forma que accesos posteriores pueden satisfacerse sin acceder al disco. Estas caches son fundamentales para el rendimiento del sistema. Este apartado aborda la cache de páginas, que trabaja con páginas completas, a diferencia de otras caches vistas en VFS.

La cache de páginas es la cache principal de disco utilizada por el kernel cuando escribe o lee de disco. Para satisfacer solicitudes de lectura de procesos de usuario, el kernel añade nuevas páginas a la cache. Si una página no esta a en la caché, se añade una nueva página a la cache y se rellena con los datos de disco. Si hay suficiente memoria libre, la página se mantiene de forma indefinida de forma que pueda ser reutilizada por otros procesos sin acceder a disco.

De forma similar, antes de escribir una página de datos en un dispositivo de bloques, el kernel verifica si la correspondiente página esta ya incluida en la cache. Si no es así, se añade una nueva entrada en la cache y se rellena con los datos que van a ser escritos en disco. La transferencias de datos de entrada/salida no se inicia inmediatamente: la escritura en disco se retrasa unos segundos, dando una oportunidad a los procesos para modificaciones posteriores de los datos (realiza una escritura diferida).

La página incluidas en la caché son de uno de los siguientes tipos (el código y datos del kernel no se leen/escriben en disco):

- Páginas con datos de archivos regulares.
- Páginas que contienen directorios. Estas son manejadas de forma similar a las anteriores.
- Páginas de datos leídos directamente de bloques de disco (saltándose la capa del sistema de archivos). Se maneja de forma similar a los archivos regulares.
- Páginas que contienen datos de procesos de usuario que han sido intercambiados a disco. El kernel puede verse forzado a mantener algunas de estas páginas en la caché de páginas incluso habiendo escrito su contenido en el área de swap.
- Páginas que pertenecen a archivos de sistemas de archivos especiales, tales como *shm* (*shared memory* - un mecanismo de comunicación entre procesos).

Como podemos ver, cada página de la caché contiene datos que pertenecen a algún archivo. Este archivo, más precisamente el inodo del archivo, se denomina *propietario de la página*.

Prácticamente, todas las operaciones `read()` o `write()` descansan en la caché de páginas. La única excepción se produce cuando un proceso abre un archivo con la opción `O_DIRECT` activa: en este caso, se sobrepasa la caché y los datos de entrada/salida transferidos hacen uso de los búferes del espacio de direcciones de usuario del proceso.

Los diseñadores del kernel han implementando la caché de páginas para satisfacer dos importantes requisitos:

- Localizar rápidamente una página específica que contiene datos relativos a un propietario específico.
- Mantener la pista de cómo se debe manejar cada página de la caché cuando el propietario la lee o escribe. Por ejemplo, es diferente leer una página de un archivo regular, un archivo de dispositivo de bloques, o un área de intercambio.

Como indicamos, la unidad de información mantenida por la cache es la página. Una página no tiene por que mantener bloques de disco físicamente adyacentes, por tanto, no puede identificarse por un número de dispositivo y un número de bloque. En su lugar, una página de la caché de páginas se identifica por un propietario y un índice dentro de los datos del propietarios, normalmente un inodo y un desplazamiento dentro del archivo correspondiente.

El objeto “espacio de direcciones”

El objeto `address_space` es la estructura de datos central de la cache de páginas. Esta estructura está embebida en el objeto inodo que posee la página (salvo para la páginas que han sido sacadas de memoria). Todas las páginas de la cache que tienen el mismo propietario están enlazadas al mismo objeto espacio de direcciones. El objeto también establece un enlace entre el propietario de las páginas y un conjunto de métodos que operan esas páginas.

Cada descriptor de página incluye los campos `mapping` e `index` que la enlazan en la caché. El primero apunta al objeto espacio de direcciones del inodo propietario de la página. El segundo especifica el desplazamiento en unidades de tamaño de página dentro del “espacio de direcciones” del propietario, es decir, la posición de la página dentro de la imagen de disco del propietario. Estos dos campos se utilizan para localizar la página en la cache.

Sorprendentemente, la cache de paginas pueden contener varias copias de los mismos datos de disco. Por ejemplo, el mismo bloque de datos de 4KB de un archivo regular puede accederse de las siguientes formas:

- Lectura del archivo; así, los datos están incluidos en una página propiedad del inodo del archivo regular.
- Lectura del bloque desde el archivo de dispositivo (partición de disco) que contiene al archivo. Aquí, los datos están incluidos en una página propiedad del inodo maestro del archivo de dispositivo de bloques.

Así, los mismos datos de disco que aparecen en dos páginas están referenciados por dos objetos espacio de direcciones.

Los campos del objeto espacio de direcciones se muestran en la Tabla 2.20.

Tabla 2.20. Los campos del objeto "espacio de direcciones"

Tipo	Campo	Descripción
`struct inode *`	`host`	Puntero al inodo que hospeda el objeto, si hay
`struct radix_tree_root`	`page_tree`	Raíz del árbol radix que identifica las páginas del propietario
`spinlock_t`	`tree_lock`	Cerrojo de protección del árbol radix
`unsigned int`	`i_mmap_writable`	Nº de mapeos de memoria compartida en el espacio de direcciones
`struct prio_tree_root`	`i_mmap`	Raíz del árbol de búsqueda de prioridad radix
`struct list_head`	`i_mmap_nonlinear`	Lista de regiones de memoria no lineales del espacio de direcciones
`spinlock_t`	`i_mmap_lock`	Cerrojo protección árbol de búsqueda de prioridad radix
`unsigned int`	`truncate_count`	Contador de secuencia utilizando cuando se trunca el archivo
`unsigned long`	`nrpages`	Nª total de páginas del propietario
`unsigned long`	`writeback_index`	Índice de página de la última operación de escritura sobre las páginas del propietario
`struct address_space_operations *`	`a_ops`	Métodos que operan sobre las páginas del propietario
`unsigned long`	`flags`	Bits de error e indicadores de asignación de memoria
`struct backing_dev_info *`	`backing_dev_info`	Puntero a `backing_dev_info` del dispositivo de bloques que mantiene los datos de éste propietario
`spinlock_t`	`private_lock`	Normalmente, cerrojo utilizado para manejar la lista `private_list`
`struct list head`	`private_list`	Normalmente, lista de buferes sucios de bloques indirectos asociados al inodo
`struct address_space *`	`assoc_mapping`	Normalmente, puntero al objeto `address_space` del dispositivo de bloques que incluyendo los bloques indirectos

Si un archivo es propietario de una página de la caché, el objeto espacio de direcciones esta embebido en el campo `i_data` del objeto inodo VFS. Si el campo `i_mapping` del inodo siempre apunta al objeto `address_space` del propietario de las páginas conteniendo los datos del inodo. El campo `host` del objeto espacio de direcciones apunta al objeto inodo en el cual esta embebido el descriptor.

Los campos `i_mmap`, `i_mmap_writable`, `i_mmap_nonlinar`, y `i_mmap_lock` se utilizan para mapeos de memoria y mapeos inversos. Ya hablamos de la interfaz de los mapeos de memoria en el Tema 1, pero no veremos su implementación por no extender demasiado este Tema. Respecto a los mapeos inversos, indicar que se utilizan en el algoritmo de recuperación de marcos de página (donde el kernel quita marcos de página los procesos

que no los utilizan) para poder conocer todas las entradas de tablas de páginas que apuntan a un marco dado.

Un campo crucial del objeto `address_space` es `a_ops` que apunta a una tabla de tipo `address_space_operations` que contiene los métodos que definen cómo se manejan las páginas. Estos métodos se muestran en la Tabla 2.21.

Tabla 2.21.- Los métodos del objeto espacio de direcciones.

Método	**Descripción**
`writepage`	Escritura (de la página a la imagen de disco del propietario)
`readpage`	Lectura (de la imagen de disco del propietario a la página)
`sync_page`	Inicia la transferencia E/S de las operaciones ya planificadas sobre las páginas del propietario
`writepages`	Escribe en disco cierto número de páginas sucias del propietario
`set_page_dirty`	Marca una pagina del propietario como sucia
`readpages`	Lee una lista de páginas del propietario de disco
`prepare_write`	Prepara una operación de escritura (usada por los sistemas de archivos basados en disco)
`commit_write`	Completa una operación de escritura (en sistemas archivos de disco)
`bmap`	Obtiene un nº de bloque lógico a partir de un índice de bloque de archivo
`invalidatepage`	Invalida páginas del propietario (cuando se trunca un archivo)
`releasepage`	En sistemas de archivos *journaling* para preparar liberación de página
`direct_IO`	Transferencia de E/S directa de páginas del propietario (saltándose la caché de páginas)

Los métodos más importantes son `readpage`, `writepage`, `prepare_write`, y `commit_write`. Por ejemplo, la función que implementa `readpage` para un inodo de un archivo regular sabe como localizar los posiciones en el dispositivo físico disco los bloques correspondientes para cada página del archivo.

El árbol radix

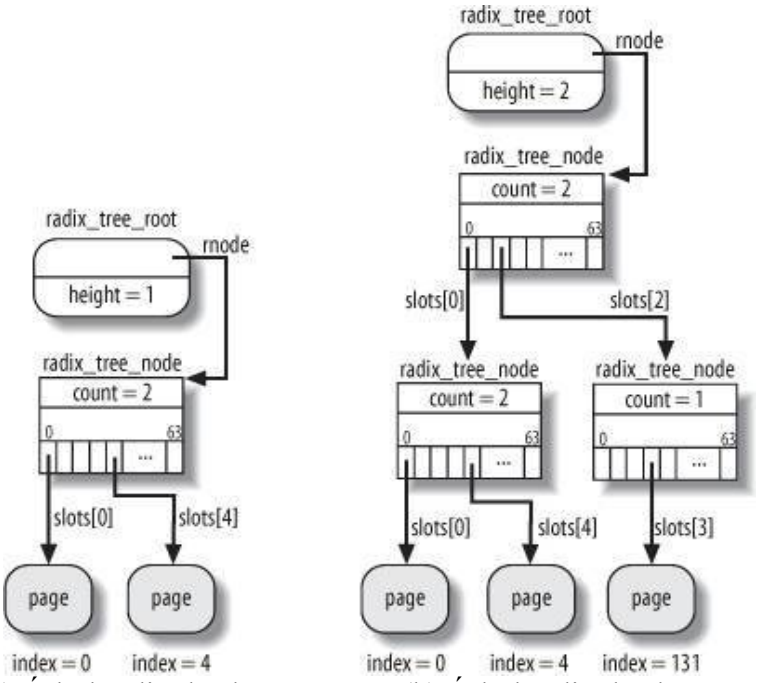
En Linux un archivo puede llegar a los terabytes. Cuando se accede a un archivo grande, la cache de páginas se llena con muchas páginas que escanean secuencialmente todo el archivo. Esto consume mucho tiempo. Para realizar una búsqueda eficiente en la cache de página, Linux 2.6 hace uso de una gran conjunto de árboles de búsqueda, uno para cada objeto `address_space`.

El campo `page_tree` del objeto `address_space` es la raíz de un *árbol radix*, que contiene punteros a los descriptores de las páginas del propietario. Dado un indice de página que denota la posición de ésta dentro de la imagen de disco del propietario, el kernel puede realizar un búsqueda muy rápida para determinar si una página dada esta ya incluida en la cache. Cuando se busca la pagina, el kernel interpreta el índice como un camino dentro del árbol radix y localiza donde la posición donde esta almacenado el descriptor de la página. Si lo encuentra, puede recuperar del árbol radix el descriptor de la página y también puede determinar rápidamente si la página esta sucia y si se esta realizando actualmente una transferencia a disco de sus datos.

Cada nodo del árbol radix puede tener hasta 64 punteros a otros nodos o a descriptores de páginas. Nodos de nivel más profundo almacenan punteros a descriptores de páginas (las

hojas) mientras que los nodos de niveles superiores almacenan punteros a otros nodos (los hijos). Cada nodo esta representado por una estructura `radix_tree_node`, que incluye tres campos: `slots` que es una matriz de 64 punteros, `count` que es un contador de cuantos punteros en el nodo son distintos de NULL, y `tags` que es una matriz de dos componentes de indicadores (el elemento cero, contiene los indicadores "sucios", es decir, que la página esta sucia; y el 1, los indicadores de que denotan que la página se esta escribiendo en disco). La raíz de este árbol esta representada por la estructura `radix_tree_root`, que posee tres campos: `heigth` que designa la altura actual del árbol (número de niveles excluyendo las hojas), `gfp_mask` que especifica los indicadores utilizados cuando se solicita memoria para un nuevo nodo, `rnode` apunta al `radix_tree_node` correspondiente a nodo de nivel 1 del árbol, si existe.

Veamos un ejemplo sencillo. Si ninguno de los índices almacenados en el árbol es mayor de 63, la altura del árbol es igual a 1, dado que las 64 hojas potenciales pueden almacenarse en el nivel 1 (Figura 2.24a). Sin embargo, si se almacena un nuevo descriptor en la cache correspondiente al índice 131, la altura del árbol se incrementa 2, de forma que el indice del árbol radix pueda apuntar índices hasta el 4095.



(a) Árbol radix de altura 1 (b) Árbol radix de altura 2

Figura 2.24.- Dos ejemplos de árbol radix.

La mejor forma para comprender cómo se realiza la búsqueda es recordar como el sistema de paginación utiliza las tablas de páginas para traducir una dirección lineal en una física. Ahora, el equivalente de la dirección lineal es el índice de página. Sin embargo, el número de campos a considerar en el índice de pagina depende de la altura del árbol radix. Si el árbol tiene altura 1, solo se representan los índices del 0 al 63, así los 6 bits menos significativos del índice de página se interpretan como el índice de la matriz `slots` para el único nodo a nivel 1. Si tenemos altura 2, indices 0 a 4095, los 12 bits menos significativos del indice se dividen en dos campos de 6 bits. Los 6 más significativos se usan para la altura 1, los otros para el nodo a nivel 2. Así para el resto de niveles. Si el indice mayor de un árbol radix es menor que el índice de una página que debe añadirse, el kernel incrementa la altura del árbol correspondiente. Los nodos intermedios del árbol dependen del valor del índice de la página.

Almacenando bloques en la cache de páginas

En versiones antiguas de Linux, existían dos caches diferentes de disco: la cache de páginas y la cache de búferes. Esta última se utilizaba para mantener en memoria los contenidos de los bloques accedidos vía VFS para manejar los sistemas de archivos basado en disco.

Actualmente, por razones de eficiencia, los búferes de bloques no se asignan individualmente, en su lugar se almacenan en páginas dedicadas, denominadas *páginas de búferes*, que se mantienen en la caché de páginas. Formalmente, una página de búferes es una página de datos asociados con descriptores adicionales denominados cabeceras de búferes, cuyo fin principal es localizar rápidamente la dirección de disco de cada bloque individual de la página. De hecho, los trozos de datos almacenados en una página de la cache de páginas no son necesariamente adyacentes en disco.

Cada búfer de bloque tiene un descriptor de *cabeza de búfer* de tipo `buffer_head` que contiene toda la información que necesita el kernel para saber cómo manejar el bloque. Los campos de la cabeza de búfer son los que aparecen en la Tabla 2.22.

Tabla 2.22. Los campos de una cabeza de búfer

Tipo	Campo	Descripción
`unsigned long`	`b_state`	Indicadores del estado del búfer
`struct buffer_head *`	`b_this_page`	Puntero al siguiente elemento en la lista de páginas de búferes
`struct page *`	`b_page`	Puntero al descriptor de la página de búferes que contiene a éste bloque
`atomic_t`	`b_count`	Contador de uso de bloque
`u32`	`b_size`	Tamaño del bloque
`sector_t`	`b_blocknr`	Número de bloque relativo al dispositivo de bloques (número de bloque lógico)
`char *`	`b_data`	Posición del bloque dentro de la página de búferes
`struct block_device *`	`b_bdev`	Puntero al descriptor del dispositivo de bloques
`bh_end_io_t *`	`b_end_io`	Método de finalización de E/S
`void *`	`b_private`	Puntero para datos del método de finalización E/S
`struct list_head`	`b_assoc_buffers`	Punteros a la lista de bloques indirectos asociados con un inodo

Dos campo codifican la dirección de disco del bloque: `b_bdev` identifica el dispositivo de bloques que contiene el bloque, y `b_blocknr` que contiene el *número de bloque lógico*, es decir, el índice del bloque dentro de su disco o partición. El campo `b_data` especifica la posición del búfer de bloque dentro de la página de búferes.

Páginas de búferes

Cuando el kernel debe direccionar un bloque individual, referencia a la página de búferes que contiene al búfer de bloques y comprueba su cabeza correspondiente. A continuación comentamos 2 casos comunes en los que el kernel crea una página de búferes:

- Cuando leer o escribe páginas de un archivo que no están almacenadas en bloques de disco continuos. Esto ocurre ya sea porque el sistema de archivos ha asignado bloques no contiguos al archivo, o porque el archivo contiene agujeros.

- Cuando se accede a un único bloque, por ejemplo, cuando se accede al superbloque o a un bloque de inodos.

En el primer caso, el descriptor de la página de búferes se inserta en el árbol radix de un archivo regular. Las cabezas de búferes se preservan puesto dado que almacena información privilegiada: el dispositivo de bloques y el número de bloque lógico. En el segundo caso, el descriptor de la página de búferes se inserta en el árbol radix enraizado en el objeto espacio de direcciones del inodo del sistema de archivos especial *bdev* asociado con el dispositivo de bloques. Este clase de páginas de búferes deben satisfacer una restricción muy fuerte: todos los búferes de bloques deben referenciar a bloques adyacentes del dispositivo de bloques subyacente. De este segundo caso nos ocuparemos ahora, las denominadas *páginas de búferes de dispositivo de bloque* o *páginas blockdev*.

Todos lo búferes de bloques dentro de una página de búferes deben tener el mismo tamaño. En la arquitectura 80x86, una pagina de búferes puede incluir de uno a ocho búferes, dependiendo del tamaño de bloque.

Cuando una pagina actual como página de búferes, todas las cabezas de búferes asociadas con sus búferes de bloques se completan en una única lista circular enlazada. El campo `private` del descriptor de la página de búferes apunta a la cabeza del primer bloque de la pagina; cada cabeza de búfer almacena en el campo `b_this_page` un puntero a la siguiente cabeza de búfer de la lista. Además, cada cabeza de búfer almacena la dirección del descriptor de la página en el campo `b_page`. La Figura 2.25 muestra una página conteniendo cuatro búferes de bloques y sus correspondientes cabezas.

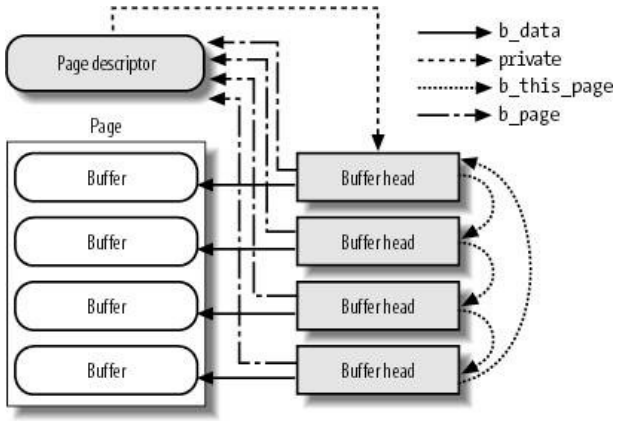


Figura 2.25.- Una página de búferes con 4 búferes y sus correspondientes cabezas.

Escritura de páginas sucias en disco

Como hemos visto, el kernel rellena la caché de páginas con páginas de datos de dispositivos de bloques. Cuando un proceso modifica algún dato, la página correspondiente se marca como sucia, se activa el indicador `PG_dirty`.

Los sistemas Unix permite las escrituras diferidas de páginas sucias en dispositivos de bloques, ya que esto mejora notablemente el rendimiento del sistema. Se pueden satisfacer varias operaciones de escritura sobre una página de la caché con sólo una modificación lenta real de los correspondientes sectores de disco. Además, las escrituras son menos críticas que las lecturas, dado que un proceso no se queda suspendido debido a las escrituras retardadas, mientras que generalmente si se suspende en las lecturas. Gracias a las escrituras diferidas, cada dispositivo de bloques físico servirá, en media, muchas más operaciones de lectura que de escritura.

Una página sucia puede permanecer en memoria hasta el último momento, esto es, el cierre del sistema. Si embargo, forzar la estrategia hasta estos límites tiene dos desventajas: si

falla la potencia, se perderán los contenidos de RAM y por tanto las modificaciones realizadas; por otro lado, el tamaño de la cache de páginas puede llegar a ser elevado.

Por tanto, las páginas sucias se escriben, *limpian*, en disco bajo las siguientes condiciones:

- La cache de páginas esta muy llena y son necesarias más páginas, o el número de páginas sucias es muy elevado.
- Ha pasado mucho tiempo desde una página esta sucia.
- Un proceso solicita la limpieza de todos los cambios pendientes de un dispositivo de bloques o de un archivo particular. Esto se realiza con las llamadas al sistema `sync()`- esta tiene su correspondiente orden-, `fsync()`, y `fdatasync()`.

Versiones anteriores de Linux utilizan una hebra kernel, denominada *bdflush*, para escanear sistemáticamente la cache de páginas buscando páginas sucias para limpiar, y utilizan una hebra kernel, denominada *kupdate*, para asegurar que ninguna página permanece sucia demasiado tiempo. El kernel 2.6 ha sustituido a ambas por un grupo de hebras kernel de propósito general denominado *pdflush*.

Estas hebras tienen una estructura flexible. Actúan sobre dos parámetros: un puntero a la función a ejecutar y un parámetro para la función. El número de estas hebras se ajusta dinámicamente: se crean nuevas hebras cuando hay pocas y se terminan cuando hay muchas. Dado que las funciones que ejecutan estas hebras pueden bloquearse, el crear varias hebras kernel *pdflush* tiene efectos positivos sobre el rendimiento. El nacimiento y muerte de estas esta gobernado por las reglas:

- Debe haber al menos 2 hebras *pdflush* y no más de 8.
- Si no hay una hebra ociosa durante el último segundo, se debe de crear otra.
- Si ha pasado más de un segundo desde que quedó ociosa la última hebra, se debe eliminar una hebra.

Los trabajos o funciones que realiza una hebra *pdflush* son de dos tipos:

- `background_writeout()` - recorre sistemáticamente la caché de páginas buscando paginas sucias que limpiar. Esta función se activa cuando: un proceso en modo usuario invoca la llamada `sync()`, la función `grow_buffers()` falla en asignar una nueva página de búferes, el algoritmo de recuperación de marcos de página invoca a `free_more_memory()` o `try_to_free_pages()`, o la función `mempool_alloc()` falla al asignar un nuevo elemento del depósito de memoria. Además, la función se invoca cada vez que un proceso modifica los contenidos de una página de la caché que provoca que la fracción de páginas sucias caiga por encima de un *valor umbral sucio* (que suele ser el 10% de las páginas del sistema y que podemos modificar escribiendo el archivo */proc/sys/vm/dirty_background_ratio*).
- `wb_kupdate()` - comprueba que las páginas de la caché no permanecen sucias durante mucho tiempo. Esta función se invoca mediante el temporizador dinámico ajustado en la inicialización del sistema que despierta a una hebra *pdflush* cada `dirty_writeback_centisecs` centésimas de segundo (normalmente 500, pero que puede modificarse escribiendo un nuevo valor en el archivo */proc/sys/vm/dirty_writeback_centisecs*).

Como curiosidad indicar que otro parámetro de configuración de *pdflsuh* es el `laptop_mode`. Esta estrategia para realizar las escrituras en disco esta pensada para optimizar la vida de la batería minimizando la actividad de disco y permitiendo que el disco duro permanezca inactivo la mayor parte posible del tiempo. Se configura mediante */proc/sys/vm/laptop_mode*. Por defecto, este fichero contiene un cero, lo que indica que esta deshabilitado. Para habilitarlo debemos escribir un uno.

Recuperación de marcos de página

Uno de los aspectos fascinantes de Linux es que las comprobaciones realizadas antes de asignar memoria dinámica a los procesos o al kernel son superficiales. No se realiza ninguna comprobación rigurosa sobre la cantidad de memoria asignada a un proceso al crearlo, ni se impone límite de tamaño a muchas cachés de disco y de memoria utilizadas por el kernel.

Esta falta de control es una elección de diseño que permite al kernel utilizar la RAM disponible de la mejor forma posible. Cuando la carga del sistema es baja, la RAM se rellena principalmente con las caches de disco y unos cuantos procesos pueden beneficiarse de la información contenida en ellas. Si embargo, cuando se incrementa la carga, la RAM se llena con páginas de procesos y se recortan las caches para hacer sitio a procesos adicionales.

Como ya vimos, las caches nunca liberan marcos de páginas. Esto tiene sentido pues las caches no saben si los procesos reutilizaran algunos datos de la cache, ni cuando, y por tanto son incapaces de identificar la porción de la cache que debe liberarse. Además, debido a la demanda de páginas, los procesos obtiene marcos conforme avanzan en su ejecución. Sin embargo, este mecanismo no tiene forma de forzar a los procesos a liberar los marcos cuando no los utilizan.

Así, tarde o temprano toda la memoria libre se asignará a los procesos y caches. El *algoritmo de recuperación de marcos de página* rellena la lista de bloques libres del sistema amigo *robando* marcos de páginas de los procesos de usuario y las caches kernel. Realmente, este algoritmo debe ejecutarse antes de que se utilice toda la memoria libre. Si no, el kernel podría verse atrapado en una cadena de solicitudes de memoria que podrían provocar su caída. Esencialmente, para liberar un marco el kernel debe escribir sus datos en disco, pero para conseguir esto necesita otro marco (por ejemplo, para asignar las cabezas de búferes para la transferencia de datos de E/S). ¡Si no existen marcos libres, no se pueden liberar marcos!

Así, uno de los objetivos del algoritmo de recuperación de marcos es conservar un depósito mínimo de marcos para que el kernel pueda recuperarse de condiciones de poca memoria.

Seleccionando un página objetivo

El objetivo del *algoritmo de recuperación de marcos de página* (PFRA) es tomar un marco y liberarlo. Este algoritmo maneja los marcos de forma diferente dependiendo de su contenido. Ver Tabla 2.23 con los diferentes tipos de marcos.

Tabla 2.23.- Tipos de marcos considerados por el PFRA.

Tipo de página	Descripción	Acción
No recuperable	Páginas libres (incluidas en listas del sistema buddy) Páginas reservadas Páginas dinámicamente asignadas por el kernel Páginas pilas kernels de los procesos Páginas bloqueadas temporalmente	No necesaria o no permitida la recuperación
Intercambiable	Páginas anónimas en espacio de usuario Páginas mapeadas en *tmpfs*	Salva contenido en un área de intercambio
Sincronizable	Páginas mapeadas en espacio de usuario Páginas de la caché con datos de archivos páginas de búferes de dispositivos de bloques Páginas de cachés de disco (ej. caché de inodos)	Sincroniza la página con si imagen en disco, si es necesario
Descartable	Páginas sin usar incluidas en las cachés de memoria Páginas sin usar en la caché dentry	No hacer nada

Diseño del PFRA

Si bien es fácil identificar las páginas candidatas para la recuperación (aproximadamente hablando, cualquier página de la caché de disco o memoria, o del espacio de usuario de los procesos), es complicado seleccionar las páginas objetivo desde el punto de vista del diseño del kernel. Es difícil encontrar un algoritmo que asegure un rendimiento aceptable tanto en "desktop" (donde las solicitudes de memoria son limitadas pero las responsividad del sistema es básica) a máquinas tales como servidores de bases de datos (donde las solicitudes de memoria son enormes).

Dado que el algoritmo usado es bastante empírico, podemos aseverar que las ideas generales del algoritmos que vamos a ver se mantendrán en los futuros kernel 2.6, pero los detalles de implementación pueden variar.

Las reglas generales que rigen al algoritmo son:

- *Libera primero las páginas "inofensivas"* - Las páginas incluidas en las cachés de disco y memoria no referenciadas por ningún proceso deben reclamarse antes que las que pertenecen al espacio de usuario de los procesos. De hecho, en las primeras, la recuperación se puede realizar sin modificar ninguna entrada de tablas de páginas. Como veremos en breve, en la descripción de LRU, esta regla es mitigada de alguna forma introduciendo un "factor de tendencia de intercambio".
- *Hacer recuperables todas las páginas de procesos en modo usuario* – Con la excepción de las páginas bloqueadas, PFRA debe ser capaz de robar cualquier página a los procesos en modo usuario, incluyendo páginas anónimas. De esta forma, procesos que duermen durante bastante tiempo van perdiendo progresivamente sus páginas.
- *Recuperar una página compartida desmapeando de una vez todas las entradas de tablas de páginas que la referencian* – Cuando FPRA desea liberar una página compartida por varios procesos, éste limpia todas las entradas de tablas de páginas que referencian al marco compartido, y recupera dicho marco.
- *Recuperar sólo páginas "sin usar"* - PFRA utiliza un esquema LRU simplificado para clasificar las páginas en dos tipos: "en uso" y "sin usar". PFRA solo recupera paginas "sin usar"

Además, PFRA es una mezcla de varias heurísticas:

- Selecciona de forma cuidadosa el orden en el que se examinan las cachés.
- Ordenas las páginas basándose en su edad (las páginas menos recientemente accedidas se liberan antes que las accedidas recientemente).
- Distinción de las páginas según su estado (por ejemplo, páginas no sucias son mejores candidatos que las sucias ya que no es necesario escribirlas en disco).

Activación del PFRA

El PFRA tiene varios puntos de entrada como puede apreciarse en la Figura (un flecha representa una llamada a la función). Realmente, la recuperación de páginas se realiza esencialmente en tres ocasiones:

- Recuperación con poca memoria – el kernel detecta la condición "escaso en memoria". Se activa cuando falla la asignación de una nueva página de búferes, la asignación de un cabeza de búferes, la asignación de un grupo de marcos contiguos en una lista dada de zonas de memoria.
- Recuperación por hibernación – el kernel debe liberar memoria dado que va a entrar en el estado "suspendido en disco".
- Recuperación periódica – una hebra kernel se activa periódicamente para realizar la recuperación de memoria, si es necesario. Las hebras encargadas de ella son:

- hebra *kswapd* – comprueba si el número de marcos libres en una zona de memoria ha caído por debajo de la marca `page_high`.
- Las hebras *events*, que son hebras trabajadoras de la cola de trabajos. PFRA planifica periódicamente una tarea en la cola de trabajo predefinida para recuperar todas las tabletas libres incluidas en la cache de memoria gestionada por el distribuidor tableta.

Las listas LRU (*Least Recently Used*)

Todas las páginas que pertenecen a los espacios de direcciones en modo usuario de los procesos o de la caché de páginas se agrupan en dos listas denominadas *listas LRU*. La *lista activa* tiende a incluir las páginas que han sido accedidas recientemente, y la *lista inactiva*, que tiende a incluir que no han sido accedidas durante algún tiempo. Claramente, las páginas se robaran de la lista de inactivas.

Una página que pertenezca a una de las listas LRU tiene activo el indicador `PG_lru` del descriptor de página. Es más, si la página pertenece a la lista activa tiene activo el indicador `PG_active`, mientras que `PG_active` se limpia si la página pertenece a la lista inactiva. El campo `lru` del descriptor de página almacena punteros al elemento previo y siguiente de la lista LRU.

PFRA acumula las páginas recientemente utilizadas en la lista activa, por tanto, no escanea esta lista cuando necesita recuperar páginas. Por contra, PFRA acumula páginas no accedidas durante largo tiempo en la lista de inactivas. Por supuesto las paginas se pueden mover entre las dos listas según los accesos que se realizan a ellas.

Por supuesto que dos estados no son suficientes para caracterizar todos los patrones de acceso. Podemos aumentar el número de estados utilizando el indicador `PG_referenced` del descriptor de página para duplicar el número de accesos necesarios para mover una página de la lista inactiva a la lista activa, pero también se utiliza para doblar el número de "ausencia de accesos" en el paso inverso. Lo que hacemos el implementar una variante del algoritmo de segunda oportunidad visto en SOI. Por ejemplo, supongamos que una página en la lista inactiva tiene `PG_referenced=1`, el primer acceso a una página pone el indicador a 1 pero la página sigue en esa lista. El segundo acceso encontrará el indicador activo y provoca que la página se mueva a la lista activa. Si por ejemplo, el segundo acceso no se produce en un intervalo de tiempo dado, el algoritmo de reclamación puede limpiar el indicador `PG_referenced`. La Figura 2.26 muestra los movimientos entre las listas LRU y las funciones de PFRA que los realizan.

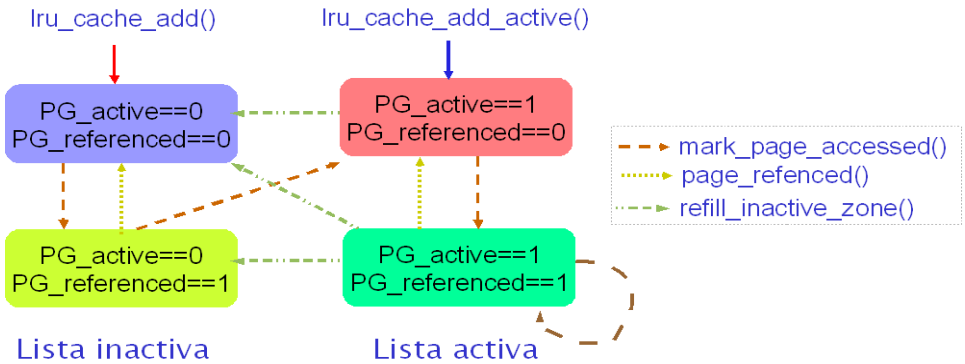


Figura 2.26.- Movimientos de páginas entre las listas LRU.

2.7.7. Intercambio (swapping)

El intercambio se introduce para salvar en disco las páginas no mapeadas. El subsistema de intercambio, de lo visto anteriormente, maneja tres clases de páginas:

- Páginas que pertenecen a regiones de memoria anónimas de los procesos (pilas en modo usuario y heap).
- Páginas sucias que pertenecen a mapeos privados de los procesos.
- Páginas que pertenecen a una región de memoria compartida IPC.

Lo mismo que la demanda de página, el intercambio es transparente a los programas. Por tanto, no hay instrucciones relacionadas con él. Las principales características del intercambio son:

- Activamos un "área de intercambio" en disco para almacenar que no tienen imagen de respaldo en disco.
- Gestiona el espacio de intercambio asignado y liberando "ranuras de páginas" con forme se necesiten.
- Suministra funciones tanto para sacar (*swap out*) páginas de la RAM en el área de intercambio e introducir (*swap in*) páginas desde el área de intercambio a RAM.
- Hace uso de los "identificadores de páginas sacadas de memoria" en las entradas de las tablas de páginas para mantener la ubicación de los datos en las áreas de intercambio.

Resumiendo, el intercambio es la característica que culmina la recuperación de páginas. Si queremos estar seguros de que todas las páginas obtenidas por un proceso, y no sólo las contienen páginas que tienen su imagen en disco, son recuperadas por PFRA, entonces debemos utilizar intercambio. Por supuesto, podemos desactivar el intercambio con la orden `swapoff`. En este caso, sin embargo, es probable que se produzca una degradación de disco tan pronto como se incremente la carga de disco.

Área de intercambio

Las páginas sacadas de memoria se almacenan en un *área de intercambio*, que puede ser tanto una partición de disco como un archivo en una partición. Se pueden definir varias áreas de intercambio, hasta un máximo especificado por `MAX_SWAPFILES` (normalmente 32). La existencia de múltiples áreas de intercambio permite al administrador del sistema repartir el espacio de intercambio entre varios discos de forma que estos puedan operar de forma concurrente. También permite que se incremente el espacio de intercambio en tiempo de ejecución sin re-arrancar el sistema.

Cada área de intercambio consta de un secuencia de *ranuras de páginas*: bloques de 4 KB utilizados para contener la página sacada. La primera ranura de página se utiliza para almacenar información sobre el área de intercambio. Su formato esta descrito por la unión `swap_store` compuesta de dos estructuras:

- `info` - indica la versión del algoritmo de intercambio, la última ranura utilizable, el número efectivo de ranuras, y las ranuras defectuosas.
- `magic` – suministra una cadena que marca parte del disco como área de intercambio de forma no ambigua.

Los datos almacenados en las áreas de intercambio tiene sentido mientras que el sistema esta funcionando. Cuando apagamos la máquina, y se matan los procesos, los datos de los procesos almacenado en éstas se descartan. Por esta razón, las áreas de intercambio contienen poca información de control: esencialmente el tipo de área de intercambio y la lista de ranuras defectuosas. Esta información encaja perfectamente en una única página de 4KB.

Normalmente, el administrador crea una partición de intercambio cuando crea el resto de particiones del sistema con la orden `mkswap`. Esta orden inicializa los campos descritos anteriormente dentro de la primera ranura. Como el disco puede contener algunos bloques malos, el programa examina todas las ranuras de páginas para localizar las defectuosas. La

ejecución de esta orden deja la partición en estado activo. Cada partición de intercambio puede activarse en un script ejecutado en el arranque o, con posterioridad, de forma dinámica.

Cuando se sacan páginas de memoria, el kernel intenta almacenarlas en ranuras contiguas para minimizar los tiempos de búsqueda de disco cuando accede al área. Sin embargo, cuando hay varias áreas de intercambio, la cosa se complica. Las áreas de intercambio almacenadas en discos más rápidos, tienen mayor prioridad. Cuando se busca una ranura libre, se empieza por el área de mayor prioridad. Si hay varias, las áreas se eligen de forma cíclica para evitar la sobrecarga de alguna. Si no se encuentran, se sigue con áreas menos prioritarias.

Cada área de intercambio tiene en memoria un descriptor denominado `swap_info_struct`, cuya estructura aparece detallada en la Tabla 2.24.

Table 2.24. Campos del descriptor de un área de intercambio.

Tipo	Campo	Descripción
`unsigned int`	`flags`	Indicadores del área de intercambio
`spinlock_t`	`sdev_lock`	Cerrojo de protección del área de intercambio
`struct file *`	`swap_file`	Puntero al objeto archivo del archivo regular o archivo dispositivo que almacena el área
`struct block_device *`	`bdev`	Descriptor del dispositivo de bloques que contiene el área de intercambio
`struct list head`	`extent_list`	Cabeza de la lista de extensiones que componen el área de intercambio
`int`	`nr_extents`	Números de extensiones que componen el área
`struct swap_extent *`	`curr_swap_extent`	Puntero al descriptor de la extensión más recientemente utilizada
`unsigned int`	`old_block_size`	Tamaño del bloque natural de la partición que contiene el área de intercambio
`unsigned short *`	`swap_map`	Puntero a vector de contadores, uno por cada ranura de página del área de intercambio
`unsigned int`	`lowest_bit`	Primera ranura a escanear cuando buscamos una libre
`unsigned int`	`highest_bit`	Última ranura a escanear cuando buscamos una libre
`unsigned int`	`cluster_next`	Siguiente ranura a escanear al buscar una libre
`unsigned int`	`cluster_nr`	Nº de asignaciones de ranuras libres antes de empezar de nuevo por el principio
`int`	`prio`	Prioridad del área de intercambio
`int`	`pages`	Número de ranuras utilizables
`unsigned long`	`max`	Tamaño del área en páginas
unsigned long	inuse_pages	Número de ranuras utilizadas en el área
`int`	`next`	Puntero al siguiente descriptor de área

El campo `swap_map` apunta a un vector de contadores, uno por cada ranura. Si el contador es 0, la ranura esta libre. Si es positivo, la ranura esta ocupada con una página sacada de memoria. Esencialmente, el contador de ranura denota el número de procesos que comparten la página almacenada. Si el contador tiene el valor `SWAP_MAP_MAX` (igual a 32767), la página almacenada es "permanente" y no puede eliminarse de la ranura. Si el contador vale `SWAP_MAP_BAD` (igual a 32768), se considera que la ranura es defectuosa, y por tanto no utilizable.

La Figura 2.27 muestra la matriz `swap_info`, un área de intercambio, y la correspondiente matriz de contadores.

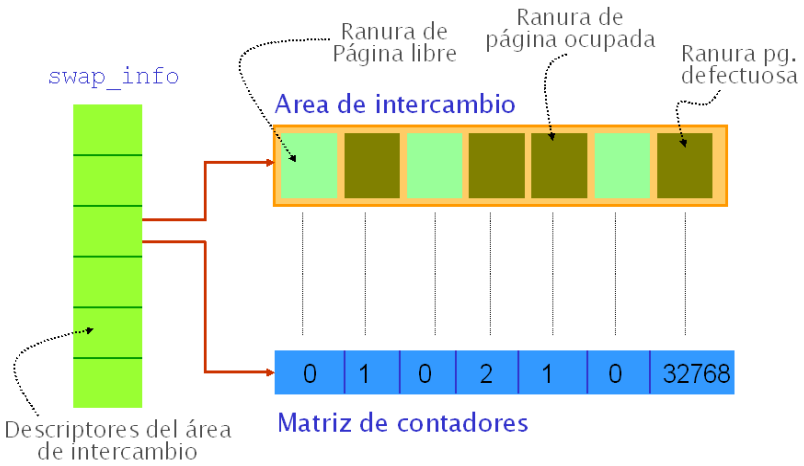


Figura 2.27.- Estructura de datos de área de intercambio.

Identificadores de páginas sacadas de memoria

Una página sacada de memoria se identifica de forma unívoca y simple especificando el índice del área de intercambio en la matriz `swap_info` y el índice de ranura dentro de área. Puesto que la primera ranura esta reservada, la primera ranura libre es la 1. El formato de una identificador de páginas sacadas de memoria se muestra en la Figura 2.28

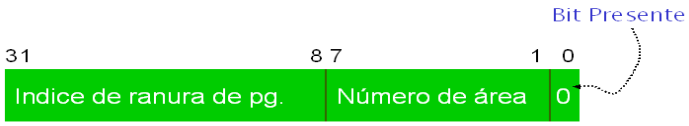


Figura 2.28.- Identificador de una página sacada de memoria.

Cuando se saca una página de memoria, su identificador se inserta en la entrada de la tabla de páginas correspondiente, de forma que la página se pueda localizar cuando se necesite. Observar que el bit menos significativo del identificador, que corresponde con el identificador `Presente`, siempre esta a cero para indicar que la página no esta en RAM. Además, al menos uno de los 31 bits restantes debe ser distinto de cero pues ninguna página se almacena en la ranura 0 del área de intercambio 0. Por tanto, es posible establecer tres casos para el valor de la PTE:

- Entrada nula – La página no pertenece al espacio de direcciones del proceso, o el correspondiente marco no ha sido aún asignado al proceso.
- Los primeros 31 bits más significativos no iguales a 0, y el último es 0 – la página esta actualmente fuera de memoria.
- El último bit menos significativo es 1- la página esta en memoria.

El tamaño máximo de un área de intercambio viene dado por el número de bits disponible para identificar una ranura. En la arquitectura 80x86, los 24 bits disponible limitan el tamaño de un área de intercambio a 64 GB.

Activando y desactivando áreas de intercambio

Una vez que se ha inicializado un área de intercambio, el superusuario puede utilizar los programas `swapon` y `swapoff` para activar y desactivar el área de intercambio, respectivamente. Estos programas utilizan las llamadas al sistema `swapon()` y `swapoff()`.

2.8 Arquitectura de E/S y Manejadores de dispositivos

En este apartado, veremos cómo el kernel invoca realmente a los dispositivos. Primero, describiremos el modelo de dispositivo. A continuación veremos como VFS asocia un archivo de dispositivo con cada dispositivo hardware. Para finalizar, detallaremos algunas características generales de los manejadores de dispositivos, para pasar a centrarnos en los dispositivos de bloques.

El modelo de manejador de dispositivo

El kernel 2.6 suministra algunas estructuras de datos y funciones que ofrecen una visión unificada de todos los buses, dispositivos y controladores del sistema. Este marco se denomina *modelo de manejador de dispositivo*.

El seudosistema de archivos *sysfs*, que se suele montar en */sys*, tiene básicamente los mismos objetivos que /proc, es decir, suministrar a los procesos de usuario acceso a las estructuras de datos del kernel. Además, este sistema ofrece la información de una manera más estructurada. Probablemente ambos sistemas coexistan durante algún tiempo más.

El objetivo de *sysfs* es exponer al usuario las relaciones de jerarquía entre los componentes del modelo de manejador de dispositivo. Los principales directorios de este sistema de archivos son:

- *block* – dispositivos de bloques, independientemente del bus en que están conectados.
- *devices* – todos los dispositivos hardware reconocidos por el kernel, organizados de acuerdo al bus en que están conectados.
- *bus* – los buses del sistema, que soportan los dispositivos
- *drivers* – los manejadores de dispositivos registrados en el kernel.
- *class* – los tipos de dispositivos del sistema (tarjetas de audio, red, etc.); la misma clase puede incluir dispositivos soportados por diferentes buses y controlados por diferentes manejadores.
- *power* – archivos para manejar los estados de potencia de algunos dispositivos.
- *firmware* – archivos para gestionar el firmware de algunos dispositivos.

Las relaciones entre componentes del modelo se expresan en éste sistema de archivos como enlaces simbólicos entre directorios y archivos. Por ejemplo, */sys/block/sda/device* puede ser un enlace simbólico a un subdirectorio anidado en */sys/devices/pci000:00*, que representa el controlador SCSI conectado al bus PCI.

El papel principal de los archivos regulares del sistema de archivos sysfs es representar los atributos de los manejadores y dispositivos. Por ejemplo, el archivo *dev* en el directorio */sys/block/sda* contiene los números principales y secundarios del disco maestro en la primera cadena IDE.

La estructura de datos básica del modelo de manejador de dispositivos es una estructura genérica denominada `kobject`, que esta inherentemente ligada a *sysfs*: cada `kobject` se corresponde con un directorio en éste sistema de archivos.

Los kobjetos esta embebidos dentro de objetos mayores denominados *contenedores* (*containers*) que describen los componentes del modelo. Ejemplos de contenedores son los descriptores de buses, dispositivos, y controladores. Por ejemplo, el descriptor de la primera partición del primer disco IDE se corresponde con el directorio */sys/block/hda/hda1*.

Embeber los kobjetos dentro de los contenedores tiene para el kernel las ventajas siguiente:

- Mantiene un contador de referencias para el contenedor.
- Mantiene listas jerárquicas o conjuntos de contenedores (por ejemplo, un directorio sysfs asociado con un dispositivo de bloques incluye un subdirectorio diferente para cada partición de disco). Ver Figura 2.29.

- Suministra al usuario una vista para los atributos de los contenedores.

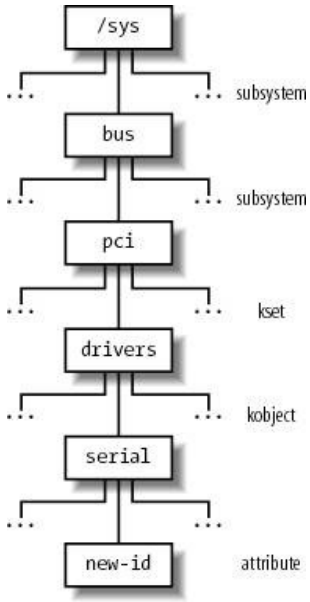


Figura 2.29.- Un ejemplo de la jerarquía del modelo de manejador de dispositivo.

Archivos de dispositivos

Como ya hemos visto, el enfoque de Unix hace que los dispositivos de E/S sean tratados como archivos especiales denominados *archivos de dispositivos*. De esta forma, la misma llamada al sistema que utilizamos para leer un archivo, nos sirve la leer de un dispositivo.

Los archivos de dispositivos pueden clasificarse en dos tipos: de caracteres y de bloques. Las diferencias entre ambas clases no tiene un corte claro, pero podemos asumir que:

- Los datos de un dispositivo de bloques pueden direccionarse aleatoriamente, y el tiempo para transferir bloques de datos es pequeño y a groso modo el mismo, al menos desde el punto de vista humano. Ejemplos típicos son los discos, los manejadores de CDs, reproductores de DVDs, etc.
- Los datos en los manejadores de caracteres pueden no direccionable aleatoriamente (por ejemplo, una tarjeta de sonido) o si, pero el tiempo de acceso aleatorio a un dato depende en gran medida de su posición en el dispositivo (manejador de una cinta magnética).

Las tarjetas de red son una excepción notable a este esquema, ya que son dispositivos hardware que no están directamente asociados con los archivos de dispositivos.

Un archivo de dispositivo es un archivo real almacenado en un sistema de archivos cuyo inodo no incluye punteros a bloques de datos de disco ya que no los hay. En su lugar, el inodo incluye un identificador del dispositivo hardware que describe. Tradicionalmente este identificador es el tipo de dispositivo (caracteres o bloques) y un par de números: el *número principal* (que identifica el tipo de dispositivo) y el *número secundario* (que identifica el dispositivo específico de entre los dispositivos que comparten el mismo número principal). Por ejemplo, un grupo de disco gestionados por el mismo controlador tienen el mismo número principal, y cada disco tiene un número secundario diferente.

La llamada al sistema `mknod()` permite crear archivos de dispositivos, y tiene como parámetros el nombre del archivo, el tipo de dispositivo y los números principal y secundario. La Tabla 2.25 ilustra algunos atributos de unos cuantos archivos de dispositivos.

Tabla 2.25. Ejemplos de archivos de dispositivos.

Nombre	Tipo	Principal	Secundario	Descripción
/dev/fd0	bloque	2	0	Disquete
/dev/hda	bloque	3	0	Primer disco IDE
/dev/hda2	bloque	3	2	Segunda partición primaria del primer disco IDE
/dev/hdb	bloque	3	64	Segundo disco IDE
/dev/hdb3	bloque	3	67	Tercera partición primaria del 2º disco IDE
/dev/ttyp0	carácter	3	0	Terminal
/dev/console	carácter	5	1	Consola
/dev/lp1	carácter	6	1	Impresora paralela
/dev/ttyS0	carácter	4	64	Primer puerto serie
/dev/rtc	carácter	10	135	Reloj de tiempo-rea
/dev/null	carácter	1	3	Dispositivo nulo (“agujero negro” de datos)

En cuanto al kernel, el nombre de un archivo de dispositivo es irrelevante (estos pueden ser relevantes para algunas aplicaciones). El kernel los identifica por los números principal y secundario. Si creamos un archivo con nombre */dev/disco* de tipo “bloque” con número principal 3 y número secundario 0, éste es equivalente al archivo de dispositivo */dev/hda*. Podemos ver los números principal y secundario asignados a los dispositivos con la orden `ls`:

```
% ls -l /dev/*
 . . .
lrwxrwxrwx 1 root root           3 may  8 16:53 /dev/cdrom -> sr0
crw------- 1 root root      5,   1 may  8 16:53 /dev/console
brw-r----- 1 root floppy    2,   0 may  8 16:53 /dev/fd0
brw-r----- 1 root disk      8,   0 may  8 16:53 /dev/sda
 . . .
```

b - dispositivo de bloques
c - dispositivo de caracteres
número principal
número secundario

El registro oficial de números de dispositivos asignados y nodos del directorio /dev se encuentra en el archivos *Documentation/devices.txt*.

Si bien la asignación de números puede hacerse de forma estática, en la actualidad, se prefiere una asignación dinámica tanto en los números de dispositivos como en la creación de archivos de dispositivos (a través del juego de herramientas *udev*).

Los archivos de dispositivos viven en el árbol de directorios pero son intrínsecamente diferentes de los archivos regulares y directorios. Para su funcionamiento VFS cambia las operaciones de archivo por defecto y las sustituye por una invocación al manejador correspondiente.

Supongamos que un proceso realiza un `open()` sobre un archivo de dispositivo. Los pasos de esta llamada ya los describimos en el apartado de VFS. Esencialmente, la rutina de servicio de la llamada resuelve el nombre del archivo de dispositivo y asigna los correspondiente objetos *inodo*, *dentry* y *file*. El objeto inodo se inicializa leyendo el correspondiente inodo en disco. Cuando esta función detecta que es un inodo de un archivo de dispositivo, inicializa el campo `i_rdev` con objeto inodo con los números principal y secundario, y el campo `i_fop` con la dirección de la tabla de operaciones de dispositivo `def_blk_fops` o `def_chr_fops`, según el tipo de dispositivo. La rutina de servicio de open invoca a la operación `dentry_open()`, que asigna un nuevo objeto archivo y ajusta su campo

`f_op` a las direcciones apuntadas por `i_fop`. Gracias a estas dos tablas, cualquier llamada al sistema realizada sobre el archivo de dispositivo, activa las funciones del manejador de dispositivo, en lugar de una de las operaciones del sistema de archivos subyacente.

Manejadores de dispositivos de bloques

Este apartado trata con los manejadores de E/S para los dispositivos de bloques. El aspecto clave en el tratamiento de estos es la disparidad entre el tiempo empleado para acceder a CPU y los buses para acceder a la lectura/escritura de datos y la velocidad de los discos hardware. Nuestro objetivo es bosquejar la estructura general del tratamiento de estos.

La arquitectura general de los componentes kernel que intervienen en la gestión de dispositivos de bloques aparece reflejada en la Figura 2.30.

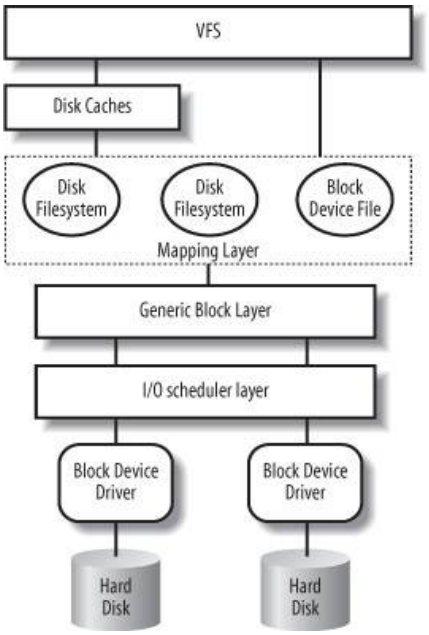


Figura 2.30.- Componentes kernel relacionados con las E/S de dispositivos de bloques.

Vamos a describir brevemente los elementos de ésta estructura, para lo cual vamos a seguir el recorrido de una operación de servicio, por ejemplo, una llamada a `read()`:

1. La llamada `read()` activa la correspondiente función VFS, como vimos en el apartado correspondiente.
2. La función VFS determina si la solicitud esta disponible en las caches de disco. Si no lo esta, realiza el accedo a disco.
3. Para acceder al disco necesita la localización física de los datos. Para hacerlo descansa en la *capa de mapeo*, que ejecuta normalmente dos pasos:
   a) Determina el tamaño del bloque del sistema de archivos que contiene al archivo y calcula el número de bloques de archivo que contiene la búsqueda solicitada.
   b) Invoca a la función específica para acceder al inodo en disco del archivo y determinar la ubicación de los bloques de datos lógicos datos.
4. El kernel esta en disposición de invocar la operación de lectura sobre el dispositivo de bloques, para la cual hace uso de la *capa genérica de bloque*, que inicia la operación de E/S que transfiere los datos. Como los bloques no tienen por que esta adyacentes de disco, esta capa puede iniciar varias operaciones de E/S, cada una de ellas esta representada por una estructura "bloque de E/S", abreviadamente `bio`, que recoge toda la información necesitada por los componentes de bajo nivel para satisfacer la

solicitud. Esta capa oculta las peculiaridades de cada dispositivo de bloques, ofreciendo una visión abstracta de estos dispositivos.

5. Bajo de la capa genérica de bloque, esta el planificador de disco (*planificador de E/S*), que ordena las solicitudes de transferencia de datos de E/S con una política predefinida (como ya vimos en el Tema 1).
6. Finalmente, el manejador de dispositivo de bloques controla la transferencia real dado las órdenes adecuadas al controlador de disco.

Cada componente de los citados maneja datos almacenados en el dispositivo de bloques utilizando trozos de diferente longitud:

- El controlador de disco transfiere datos en trozos de longitud física denominados sectores. Por tanto, el planificador de E/S y el manejador de disco manejan *sectores*.
- VFS, la capa de mapeo y el sistema de archivos agrupan los datos de disco en unidades lógicas denominadas *bloques*. Un bloque se corresponde con la unidad mínima de almacenamiento de disco dentro del sistema de archivos.
- Como veremos en breve, los manejadores de dispositivos de bloques debe tratar con *segmentos* de datos que son porciones de páginas de memoria que contienen trozos de datos que esta físicamente adyacentes en disco.
- Las caches de disco trabajan con *páginas* de datos de disco, cada una de las cuales encaja en un marco de memoria.
- La capa genérica de bloque aglutina todos los componentes superiores e inferiores, por lo que conoce sectores, bloques, segmentos y páginas.

La Figura 2.31 muestra una página de 4 KB. Los componente superiores ven la página compuesta de cuatro búferes de 1,024 bytes cada uno. Los últimos tres bloques de la página están siendo transferidos por el manejador, así que se insertan en un segmento que cubre 3,072 bytes. El controlador de disco considera que el segmento esta compuesto de seis sectores de 512 bytes.

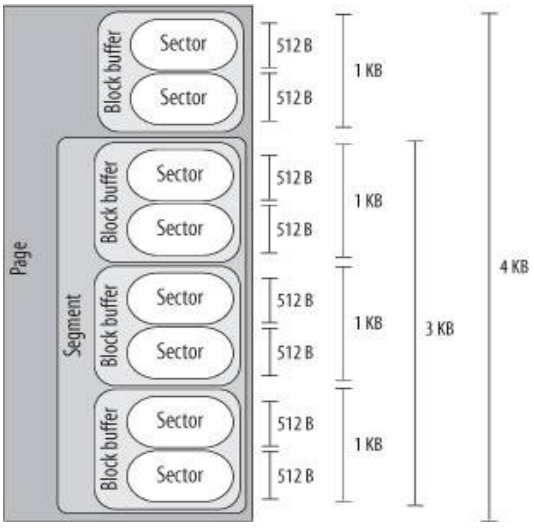


Figura 2.31.- Estructura básica de una página que contiene datos de disco.

Búferes y cabezas de búferes

Como ya vimos en el apartado "Almacenando bloques en la caché de páginas", cuando almacenamos un bloque en memoria, éste se almacena en un *búfer*. Cada búfer esta asociado exactamente con un bloque, es decir, sirve como un objeto que representa el bloque de disco en memoria (recordar que un bloque puede contener uno o más sectores pero no más de una página). Dado que se necesita cierta información de control para describir los datos (tal como de que dispositivo de bloques viene y de que bloque especifico), cada búfer tiene asociado un

descriptor denominado *cabeza de búfer* (de tipo `struct buffer_head`). Los principales campos de esta estructura se describieron en la Tabla 2.22.

El campo `b_state` especifica el estado de un búfer particular. Algunos de los estados posibles son: el búfer contiene datos válidos, el búfer esta sucio, esta bloqueado por intervenir en una E/S de disco, no esta asociado aún con un bloque en disco, etc.

El marco de memoria donde reside un búfer esta apuntado por `b_page`, y más específicamente `b_data` apunta al bloque dentro del marco, que tiene un tamaño `b_size`.

En los kernels anteriores al 2.6, esta estructura era mucho más importante: era la unidad de E/S. No solo constituía un mapeo entre bloques de disco y páginas física, sino que actuaba como contenedor utilizado por todos los bloques de E/S. Esto presentaba dos problemas importantes. Primero, la cabeza de búfer era una estructura grande e inmanejable y no era limpia ni sencilla la manipulación de datos en términos de cabezas de búferes. Segundo, una cabeza solo describe un único búfer. Como contenedora de todas las operaciones de E/S, forzaba al kernel a trocear las operaciones de grandes bloques de E/S. Esto significaba una sobrecarga y consumo de espacio.

Los diseñadores del kernel 2.5 introdujeron una nueva estructura, más flexible y ligera, para contener las operaciones de bloques de E/S, la estructura *bio*.

La estructura bio

El contenedor básico del kernel para bloques de E/S es la estructura *bio*. Esta estructura representa operaciones de E/S de bloques que están "en vuelo" (activas) como una lista de segmentos. De esta forma los búferes individuales no necesitan estar contiguos en memoria. Permitiendo que los búferes se describan a trozos, la estructura *bio* suministra al kernel la capacidad para realizar operaciones de E/S de incluso un único búfer desde múltiples ubicaciones de memoria. Un vector de E/S de este tipo se denomina "E/S reunir-dispersar" (*scatter-gather I/O*, ver llamadas `readv()` y `writev()`). Esta estructura se describe en la Tabla 2.27.

Como hemos indicado el principal objetivo de la estructura bio es representar las operaciones de E/S de bloques en vuelo. Para ello, la mayoría de los campos de la estructura son administrativos. Los campos más importante son `bi_io_vecs`, `bi_vcnt`, y `bi_idx`. La Figura 2.32 muestra la relación entre la estructura y sus amigos.

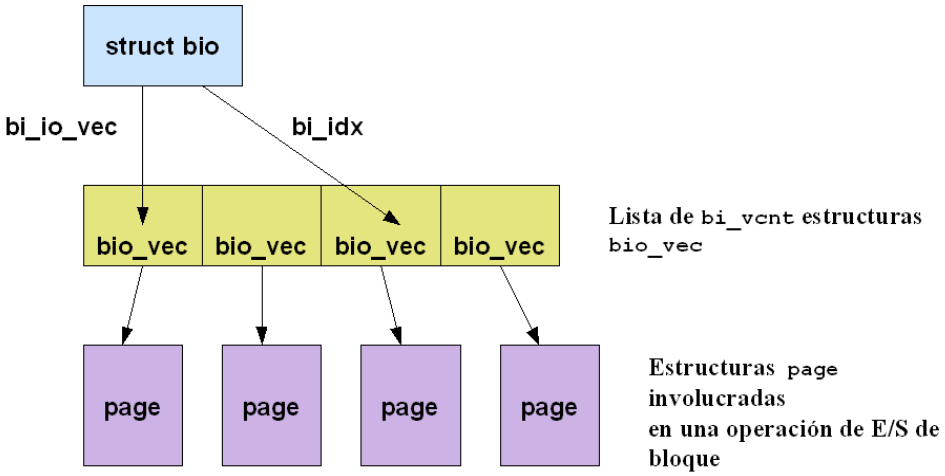


Figura 2.32.- Relaciones entre las estructuras `bio`, `bio_vect`, y `page`.

El campo `bi_io_vecs` apunta a la matriz de estructuras `bio_vec`. Estas estructuras se utilizan como listas de segmentos individuales en las operaciones de E/S de bloques específicas. Cada `bio_vec` se trata como un vector de la forma `<page, offset, len>`, que describe un segmento específico: el marco en el que reside, la ubicación del bloque como un

desplazamiento en la página, y la longitud del bloque comenzando desde el desplazamiento especificado.

Tabla 2.27. Los campos de la estructura bio.

Tipo	Campo	Descripción
`sector_t`	`bi_sector`	Primer sector en disco de la operación de bloque de E/S
`struct bio *`	`bi_next`	Enlace a la siguiente bio en la cola de peticiones
`struct block_device *`	`bi_bdev`	Puntero al descriptor del dispositivo de bloques
`unsigned long`	`bi_flags`	Indicadores de estado de la bio
`unsigned long`	`bi_rw`	Indicadores de la operación de E/S
`unsigned short`	`bi_vcnt`	Nº de segmentos en la matriz `bio_vec` de la bio
`unsigned short`	`bi_idx`	Indice actual en la matriz `bio_vec` de segmentos
`unsigned short`	`bi_phys_segments`	Número de segmentos físicos de la bio después de la mezcla
`unsigned short`	`bi_hw_segments`	Nº de segmentos hardware después de la mezcla
`unsigned int`	`bi_size`	Bytes (aún) por transferir
`unsigned int`	`bi_hw_front_size`	Usado por el algoritmo de mezcla de segmentos hardware
`unsigned int`	`bi_hw_back_size`	Como el campo anterior
`unsigned int`	`bi_max_vecs`	Nº máximo de segmentos permitidos en la matriz `bio_vec`
`struct bio_vec *`	`bi_io_vec`	Puntero a la matriz `bio_vec`
`bio_end_io_t *`	`bi_end_io`	Método invocado al final de la operación E/S
`atomic_t`	`bi_cnt`	Contador de referencias de la `bio`
`void *`	`bi_private`	Puntero usado por la capa genérica de bloque y el método de conclusión del manejador de dispositivo de bloques
`bio_destructor_t *`	`bi_destructor`	Método destructor que se invoca al liberar la bio (normalmente `bio_destructor()`)

El campo `bi_idx` se utiliza para apuntar al elemento actual de la lista `bio_vec`, lo que permite a la capa de E/S bloques seguir la pista de las operaciones parcialmente completadas. Una característica importante es poder partir las estructuras bio. Con ella, manejadores tales como RAID, pueden tomar una estructura bio, inicialmente pensada para un único dispositivo, y partirla entre los múltiples manejadores de disco de la matriz RAID. Los manejadores RAID lo único que deben de hacer es copiar la estructura bio y actualizar el índice `bi_idx` para que apunte donde se debe iniciar la operación en el manejador individual.

La diferencia entre las cabezas de búferes y la nueva estructura bio es importante. Esta estructura representa una operación de E/S de bloque, que puede incluir varias páginas, mientras que la estructura `buffer_head` representa un único búfer que describe un único sector de disco. Así una cabeza de búfer divide innecesariamente una solicitud en trozos de tamaño del bloque, sólo para reunirlos después (cosa que no es necesario en la estructura bio).

El cambio de la estructura cabeza de búfer a la estructura bio tiene otros beneficios:

- La estructura bio puede representar de forma fácil memoria alta ya que trata con páginas física y no punteros directos.
- La estructura bio puede representar tanto páginas normales de E/S como E/S directas (es decir, operaciones que no se realizan a través de la caché).

- También realiza de una forma cómoda operaciones de E/S reunidas-dispersas, con los datos involucrados en la operación provenientes de múltiples marcos.
- Esta estructura es más ligera que la cabeza de búferes dado que contiene sólo la información mínima para representar la operación de E/S y no información innecesaria relacionada con el búfer.

Sin embargo, es necesario seguir manteniendo las cabezas de búferes. Estas funcionan como descriptores, mapean bloques de disco en páginas. Las estructuras bio no contienen información a cerca del estado del búfer, solo describen operaciones E/S en vuelo.

Colas de peticiones

Los dispositivos de bloques mantienen *colas de peticiones* para almacenar sus peticiones pendientes de bloques de E/S. La cola de peticiones se representa por la estructura `request_queue`. Esta cola contiene una lista doble enlazada de solicitudes y su información del control asociada. Las solicitudes se insertan en la cola por el código de alto nivel, como el sistema de archivos. Tan pronto como una cola de peticiones no esta vacía, el manejador del dispositivo de bloques asociado con la cola agarra la solicitud de la cabeza de la cola y la remite al dispositivo asociado. Si hay más peticiones, como vimos en el Tema 1, el *planificador de E/S* se encarga de ordenarlas para mejorar el rendimiento.

Cada item de la cola de peticiones representa una sola petición, representada por una estructura `request`. Cada solicitud puede componerse de varias estructuras bio dado que solicitudes individuales pueden operar sobre múltiples bloques de disco consecutivos.