TEMA 3. GESTIÓN DE MEMORIA

3.1. INTRODUCCIÓN

La memoria es uno de los recursos más valiosos que gestiona el sistema operativo. Uno de los elementos principales que caracterizan un proceso es la memoria que utiliza. Ésta está lógicamente separada de la de cualquier otro proceso del sistema (excepto los threads de un mismo proceso que comparten normalmente la mayor parte de la memoria que tienen asignada). Un proceso no puede acceder, al espacio de memoria asignado a otro proceso, lo cual es imprescindible para la seguridad y estabilidad del sistema. El direccionamiento es una parte importante de la gestión de memoria, puesto que influye mucho en la visión del mismo por parte de un proceso, como en el aprovechamiento del hardware y el rendimiento del sistema. En Linux, además, un proceso tiene dos espacios de memoria: el *espacio de memoria del usuario*, único para ese proceso, y el *espacio de memoria del kernel*, idéntico en todos los procesos.

Objetivos del sistema de gestión de memoria:

- Ofrecer a cada proceso un espacio lógico propio.
- Proporcionar protección entre procesos.
- Permitir que los procesos compartan memoria.
- Dar soporte a las distintas regiones del proceso.
- Maximizar el rendimiento del sistema.
- Proporcionar a los procesos mapas de memoria muy grandes.

Espacio de direcciones de un proceso $\Rightarrow$ Conjunto de direcciones a las que hace referencia. Los espacios de direcciones involucrados en la gestión de la memoria son de tres tipos:

- *Espacio de direcciones físicas*. Las direcciones físicas son aquellas que referencian alguna posición de la memoria física. Se obtienen después de aplicar una transformación por parte de la MMU (Unidad de Manejo de Memoria).
- *Espacio de direcciones lógicas o virtuales*. Las direcciones lógicas son las direcciones utilizadas por los procesos. Sufren una serie de transformaciones, realizadas por el procesador (la MMU), antes de convertirse en direcciones físicas.
- *Espacio de direcciones lineales*. Las direcciones lineales se obtienen a partir de las direcciones lógicas tras haber aplicado una transformación dependiente de la arquitectura. En Linux las direcciones lógicas y lineales son idénticas. En el i386, es el nombre que reciben las direcciones tras haber aplicado la técnica de segmentación. En la segmentación, tras haber realizado las correspondientes comprobaciones de seguridad, se le suma a la dirección lógica una cierta dirección base, obteniendo así la dirección lineal. A partir del kernel de Linux 2.2.x, las direcciones base de casi todos los segmentos es 0, y por lo tanto, las direcciones lineales y las lógicas son las mismas.

La unidad de manejo de memoria (MMU) es parte del procesador. Sus funciones son:

- Convertir las direcciones lógicas emitidas por los procesos en direcciones físicas.
- Comprobar que la conversión se puede realizar. La dirección lógica podría no tener una dirección física asociada. Por ejemplo, la página correspondiente a una dirección se puede haber intercambiada a una zona de almacenamiento secundario temporalmente.
- Comprobar que el proceso que intenta acceder a una cierta dirección de memoria tiene permisos para ello.

En caso de fallo se lanzará una excepción que deberá ser resuelta por el *kernel* del sistema operativo. El *kernel* del sistema operativo está siempre en memoria principal, puesto que si se intercambia a una zona de almacenamiento secundario, ¿quién sería el encargado de llevarlo a memoria principal cuándo se produjera un fallo de acceso a memoria?

La MMU se inicializa para cada proceso del sistema. Esto permite que cada proceso pueda usar el rango completo de direcciones lógicas (memoria virtual), ya que las conversiones de estas direcciones serán distintas para cada proceso.

En todos los procesos se configura la MMU para que la zona del *kernel* (el cuarto gigabyte) sólo se pueda acceder en modo *kernel* (modo privilegiado).

La configuración correspondiente al espacio de memoria del kernel es idéntica en todos los procesos. Todos los threads de un mismo proceso también compartirán la configuración del espacio de memoria del usuario.

Para ejecutar un proceso $\Rightarrow$ debe estar, al menos en parte, en memoria principal. *Subsistema de Gestión de Memoria* (parte del *kernel* del sistema operativo):

- Decide qué procesos residen en memoria principal (al menos una parte).
- Maneja parte del espacio de direcciones virtuales que ha quedado fuera.
- Controla la cantidad de memoria principal.
- Gestiona el intercambio de procesos entre memoria principal y memoria secundaria o dispositivo de *swap*.

Históricamente UNIX:

- Política de gestión de memoria llamada intercambio (swapping).
- Transferían procesos enteros entre memoria principal y swap.
- Problema: Límite al tamaño de los procesos.
- Ventaja: Facilidad de implementación y el menor overhead del sistema.

UNIX BSD $\Rightarrow$ Paginación por demanda o demanda de páginas. Transferencia páginas de memoria, no procesos, entre memoria principal y swap. *Kernel* carga páginas de un proceso cuando las referencia. Ventajas (paginación por demanda):

- Flexibilidad al mapear el espacio de direcciones virtuales de un proceso con la memoria física de la máquina
- Permite tamaño proceso > cantidad de memoria física disponible.
- Más procesos residiendo en memoria simultáneamente.

3.2. MEMORIA VIRTUAL.

El tamaño combinado del programa, datos y pila puede exceder la cantidad de memoria física disponible. El sistema operativo guarda aquellas partes del programa concurrentemente en uso en memoria central y el resto en disco. Cuando un programa espera que se le cargue en memoria central de disco otra parte del mismo, la CPU se puede asignar a otro proceso.

Memoria virtual, el sistema operativo gestiona niveles de memoria principal y memoria secundaria:

- Transferencia de bloques entre ambos niveles (normalmente basada en paginación).
- De memoria secundaria a principal: por demanda.

- De memoria principal a secundaria: por expulsión.

Beneficios: (1) Aumenta el grado de multiprogramación; (2) Permite ejecución de programas que no quepan en memoria principal.

3.2.1. Paginación.

El espacio virtual de direcciones se divide en unidades llamadas páginas, todas del mismo tamaño. La memoria principal se divide en marcos de páginas (page frames) del mismo tamaño que las páginas virtuales y son compartidas por los distintos procesos del sistema (en cada marco de página se carga una página de un proceso).

No todo el espacio virtual de direcciones está cargado en memoria central. Una copia completa se encuentra en disco y las páginas se traen a memoria central cuando se necesitan.

- Tabla de páginas (TP) ⇒ Relaciona cada página con el marco que la contiene.
- MMU usa TP para traducir direcciones lógicas a físicas.
- Típicamente usa 2 TPs: TP usuario y TP sistema (sólo se permite usar estas direcciones en modo sistema).

Transformación de la dirección virtual en dirección física: Los bits de mayor peso de la dirección se interpretan como el número de la página en la TP y los de menor peso como el número de palabra dentro de la página (desplazamiento).

Contenido de cada entrada de la TP: (1) Número de marco asociado; (2) Información de protección (RWX), si operación no permitida ⇒ Excepción; (3) Bit de página válida/ inválida (utilizado en memoria virtual para indicar si página presente), si se accede ⇒ Excepción; (4) Bit de página accedida (referenciada) ⇒ MMU lo activa cuando se accede a esta página; (5) Bit de página modificada ⇒ MMU lo activa cuando se escribe en esta página; (6) Bit de desactivación de caché ⇒ utilizado cuando la entrada corresponde con direcciones de E/S.

Tamaño de página (condicionado por diversos factores contrapuestos) ⇒ Potencia de 2 y múltiplo del tamaño del bloque de disco (compromiso entre 2K y 16K).

Gestión de la TP por parte del sistema operativo: (1) El sistema operativo mantiene una TP por cada proceso ⇒ en el cambio de contexto notifica a MMU qué TP debe usar; (2) El sistema operativo mantiene una única TP para el propio sistema operativo ⇒ proceso en modo *kernel* accede directamente a su mapa y al del sistema operativo; (3) El sistema operativo mantiene tabla de marcos de páginas ⇒ estado de cada marco (libre o ocupado, ...); (4) El sistema operativo mantiene tabla de regiones de memoria por cada proceso.

Implementación de las TPs que se mantiene normalmente en memoria principal ⇒ problemas: eficiencia y gasto de almacenamiento. (1) Eficiencia ⇒ Cada acceso lógico requiere dos accesos a memoria principal (a la tabla de páginas + al propio dato o instrucción) → solución: caché de traducciones (TLB, Translation Lookaside Buffer); (2) Gasto de almacenamiento ⇒ Tablas muy grandes → solución: tablas multinivel.

TLB. Buffer caché con información sobre últimas páginas accedidas ⇒ caché de entradas de TP correspondientes a estos accesos. 2 alternativas: (1) Entradas en TLB no incluyen información sobre proceso ⇒ invalidar TLB en cambios de contexto; (2) Entradas en TLB incluyen información sobre proceso ⇒ registro de CPU debe mantener un identificador de proceso actual. TLB gestionada por hardware: (1) MMU consulta TLB, si fallo, entonces utiliza la TP en memoria. TLB gestionada por software (ceder al sistema operativo parte del trabajo de traducción): (1) MMU no usa las TPs, sólo consulta TLB; (2) el sistema operativo mantiene TPs que son independientes del hardware, si fallo en TLB ⇒ activa el sistema operativo que se encarga de: buscar en TP la traducción y rellenar la TLB con la traducción; proporcionando flexibilidad en el diseño del sistema operativo pero menor eficiencia.

TP multinivel. Tablas de páginas organizadas en M niveles: entrada de TP del nivel K apunta a TP de nivel K+ 1 y entrada del último nivel apunta a marco de página. Dirección lógica especifica la entrada a usar en cada nivel: un campo por nivel + desplazamiento. Un acceso lógico $\Rightarrow$ M + 1 accesos a memoria (uso de TLB). Si todas las entradas de una TP son inválidas: no se almacena esa TP y se pone inválida la entrada correspondiente de la TP superior.

Ventajas de las tablas de páginas multinivel: (1) Si proceso usa una parte pequeña de su espacio lógico $\Rightarrow$ ahorro en espacio para almacenar TPs; (2) Ahorro en la memoria utilizada para su implementación $\Rightarrow$ Por ejemplo: proceso que usa 12MB superiores y 4MB inferiores $\rightarrow$ 2 niveles, páginas de 4K, dirección lógica 32 bits (10 bits por nivel) y 4 bytes por entrada $\rightarrow$ Tamaño: 1 TP N_1 + 4 TP N_2 = 5 * 4KB = 20KB (frente a 4MB); (3) Permite compartir TPs intermedias; (4) Sólo se requiere que esté en memoria la TP de nivel superior $\Rightarrow$ TPs restantes pueden estar en disco y traerse por demanda, cuando se necesiten.

Con la paginación, la MMU no sabe nada sobre las distintas regiones de los procesos, sólo entiende de páginas. El sistema operativo debe guardar para cada proceso una tabla de regiones que especifique qué paginas pertenecen a cada región. Esto tiene dos desventajas: (1) Para crear una región hay que rellenar las entradas de las páginas pertenecientes a la región con la mismas características (por ejemplo, que no se puedan modificar si se trata de una región de código); y (2) Para compartir una región, hay que hacer que las entradas correspondientes de dos procesos apunten a los mismos marcos. En resumen, lo que se está echando en falta es que la MMU sea consciente de la existencia de regiones y que permita tratar a una región como una entidad.

3.2.2. Segmentación.

El espacio de direcciones se divide en segmentos, cada uno de los cuales corresponderá a una rutina (procedimiento, función), un programa o un conjunto de datos (una entidad lógica). Todo aquello que se corresponda con sub-espacio de direcciones independientes.

Cada programa contiene una cierta cantidad de segmentos. Los primeros segmentos se reservan para procedimientos, datos y pila, pertenecientes al programa en ejecución. Los segmentos restantes contienen un archivo por segmento, así que los procesos pueden direccionar todos sus archivos directamente sin tener que abrirlos ni usar primitivas especiales de entrada/salida. Cada archivo puede crecer de forma completamente independiente de los otros, con cada byte direccionado por un par (segmento, desplazamiento).Por otro lado, colocando objetos diferentes en diferentes segmentos, se facilita la compartición de estos objetos entre procesos múltiples.

Segmentación: (1) Esquema hardware que intenta dar soporte directo a las regiones; (2) Generalización de los registros base y límite (una pareja por cada segmento); (3) Dirección lógica: número de segmento + desplazamiento en el segmento; (4) MMU usa una tabla de segmentos (TS); (5) El sistema operativo mantiene una TS por proceso $\Rightarrow$ en un cambio contexto notifica a MMU qué TS debe usar; (6) Entrada de TS contiene (entre otros): registro base y límite del segmento, protección “RWX”; (7) Dado que cada segmento se almacena en memoria contigua, este esquema presenta “fragmentación externa”, ya que el segmento es la unidad de asignación; (8) el sistema operativo mantiene información sobre el estado de la memoria $\Rightarrow$ estructuras de datos que identifiquen huecos y zonas asignadas.

3.2.3. Segmentación Paginada.

Como su propio nombre indica, la segmentación paginada intenta aunar lo mejor de los dos esquemas. La segmentación proporciona soporte directo a las regiones del proceso y la paginación permite un mejor aprovechamiento de la memoria y una base para construir un esquema de memoria virtual. Con esta técnica, un segmento está formado por un conjunto de páginas, y por tanto, no tiene que estar contiguo en memoria. La MMU utiliza una tabla de segmentos, tal que cada entrada de la tabla apunta a una tabla de páginas.

Entrada en una TS apunta a una TP para el segmento:

- Segmentación ⇒ (1) Soporte directo de segmentos; (2) Facilita operaciones sobre regiones: (a) Establecer protección → Modificar sólo una entrada de la TS; (b) Definir compartición de segmento → entradas de la TS apuntando a la misma TP de segmento.
- Paginación ⇒ (1) Facilitar la asignación no contigua de segmento en memoria; (2) Evitar fragmentación interna que supone la segmentación (cada segmento se almacena en memoria de forma contigua).

3.2.4. Paginación por Demanda.

Segmentación pura no es adecuada para memoria virtual ⇒ Tamaño de segmentos variable.

Paginación y segmentación paginada sí son para memoria virtual ⇒ (1) Bloque transferido → Página; (2) Memoria virtual + Paginación → Paginación por demanda.

Estrategia de implementación: Uso del bit de validez, donde la página no residente se marca como no válida. Acceso a una página (Excepción de falta de página de la MMU al *kernel* del sistema operativo si la página no está en la MMU): el *kernel* del sistema operativo trae la página correspondiente de memoria secundaria ⇒ el kernel del sistema operativo debe diferenciar entre página no residente y página inválida.

Prepaginación ⇒ Traer páginas por anticipado (no por demanda): En caso de falta de página se traen además otras páginas que se consideran que necesitará el proceso → Beneficiosa dependiendo de si hay acierto en la predicción.

Tratamiento de falta de página (peor de los casos, falta de página puede implicar dos operaciones de E/S al disco) ⇒ Tratamiento de la excepción provocado por la MMU (dirección de fallo disponible en registro): (1) Si dirección inválida → aborta proceso o le manda señal; (2) Si no hay ningún marco libre (consulta tabla de marcos de página): (a) Selección de víctima (algoritmo de reemplazo de página): página P marco M → Marca P como inválida; (b) Si P modificada (bit Modificado de P activo) → inicia escritura P en memoria secundaria; (2) Si hay marco libre (se ha liberado o lo había previamente): (a) Inicia lectura de página en marco M; (b) Marca entrada de página válida referenciando a M; (c) Pone M como ocupado en tabla de marcos de página (si no lo estaba).

Políticas de administración de la memoria virtual: (1) Política de reemplazo: ¿Qué página reemplazar si hay falta de página y no hay marco libre? ⇒ (a) Reemplazo local → sólo puede usarse para reemplazo un marco de página asignado al proceso que causa la falta de página; (b) Reemplazo global → puede usarse para reemplazo cualquier marco; (2) Política de asignación de espacio a los procesos: ¿Cómo se reparten los marcos entre los procesos? ⇒ Asignación fija o dinámica.

3.2.5. Algoritmos de Reemplazo de Página.

Objetivo de los algoritmos de reemplazo: Minimizar la tasa de fallos de página. Cada algoritmo tiene versión local y global: local ⇒ criterio se aplica a las páginas residentes del proceso, y global ⇒ criterio se aplica a todas las páginas residentes. Algoritmos conocidos: Óptimo, FIFO (First In First Out), Clock (Reloj o segunda oportunidad) = FIFO + uso del bit de referencia; y LRU (Least Recently Used). Además estos algoritmos se utilizan en técnicas de buffering de páginas.

LRU ⇒ Criterio: reemplaza la página residente menos usada recientemente. Está basado en el principio de proximidad temporal de referencias: si es probable que se vuelve a referenciar las páginas accedidas recientemente, entonces la página que se debe reemplazar es la que no se ha referenciado desde hace más tiempo. Sutileza en su versión global: menos recientemente usada en el tiempo lógico de cada proceso (no tiempo real).Difícil implementación estricta (hay aproximaciones) ⇒ Precisaría una MMU específica. Posible implementación con hardware específico: en entrada de TP hay un contador, en cada acceso a memoria

MMU copia contador del sistema a entrada referenciada; y el reemplazo sería para la página con contador más bajo.

Buffering de páginas. Mantiene una lista de marcos de página libres. (1) Si falta de página ⇒ siempre usa marco de página libre (no reemplazo); (2) Si número de marcos libres < umbral ⇒ "demonio de paginación" aplica repetidamente el algoritmo de reemplazo: páginas no modificadas pasan a lista de marcos de páginas libres y páginas modificadas pasan a lista de marcos de páginas modificados (cuando se escriban a disco pasan a lista de libres y pueden escribirse en tandas (mejor rendimiento)); (3) Si se referencia una página mientras está en estas listas ⇒ falta de página: dicha página la recupera directamente de la lista (no E/ S).

Puede darse el caso de que haya páginas marcadas como "no reemplazables" ⇒ se aplica a páginas del propio sistema operativo, también se aplica mientras se hace DMA sobre una página. Algunos sistemas operativos ofrecen a aplicaciones un servicio para fijar en memoria una o más páginas, pudiendo afectar positivamente al rendimiento del sistema.

3.2.6. Políticas de Asignación de Marcos de Página.

Estrategia de asignación fija ⇒ Número de marcos asignados al proceso (conjunto residente) es constante. Puede depender de características del proceso (tamaño, prioridad, etc.). No se adapta a las diferentes necesidades de memoria de un proceso a lo largo de su ejecución. Comportamiento del procesos es relativamente predecible. Sólo tiene sentido usar reemplazo local. La arquitectura de la máquina impone un número mínimo de marcos de página que debe asignarse a un proceso (por ejemplo, la instrucción "move" requiere un mínimo de 3 marcos de página).

Estrategia de asignación dinámica ⇒ Número de marcos varía dependiendo de comportamiento del proceso (y posiblemente de los otros procesos en el sistema). Asignación dinámica + reemplazo local → el proceso va aumentando o disminuyendo su conjunto residente dependiendo de su comportamiento (distintas fases de ejecución del programa) y tiene un comportamiento relativamente predecible. Asignación dinámica + reemplazo global → los procesos se quitan las páginas entre ellos, teniendo un comportamiento difícilmente predecible.

3.2.7. Operaciones sobre las Regiones de un Proceso.

Como ya sabemos, el espacio de direccionamiento de un proceso se compone de varias regiones de memoria y cada región de memoria se caracteriza por varios atributos: (1) sus direcciones de inicio y fin; (2) los derechos de acceso que tiene asociados; (3) el objeto asociado (por ejemplo, un archivo ejecutable que contiene el código ejecutado por el proceso). Las regiones de memoria contenidas en el espacio de direccionamiento de un proceso pueden determinarse mostrando el contenido del archivo *maps*, situado en el directorio de cada proceso en el sistema de archivos /proc.

Las operaciones que el *kernel* del sistema operativo puede realizar sobre las regiones de memoria de un proceso son las siguientes: (1) Creación de región ⇒ Al crear mapa inicial o por solicitud posterior; (2) Liberación de región ⇒ Al terminar el proceso o por solicitud posterior; (3) Cambio de tamaño de región ⇒ Del heap o de la pila (stack); (4) Duplicado de región ⇒ Operación requerida por el servicio *fork*.

3.3. GESTIÓN DE MEMORIA EN UNIX

Puesto que UNIX se ideó para ser independiente de la máquina, su esquema de gestión de memoria varia de un sistema a otro. Las primeras versiones de UNIX simplemente empleaban particiones variables sin ningún esquema de memoria virtual. Las implementaciones actuales, incluida SVR4, utilizan memoria virtual paginada. En SVR4 existen actualmente dos esquemas de gestión de memoria separados. El sistema de paginación ofrece una memoria virtual que asigna marcos de página en la memoria principal a los procesos y

también asigna marcos de página a los buffers (memorias intermedias) de los bloques de disco. Aunque esto es un esquema de gestión de memoria efectiva para procesos de usuario y de E/S de disco, un esquema de memoria virtual paginado se adapta peor a la gestión de asignación de memoria para el *kernel*, y para este último propósito se utiliza un gestor de memoria del *kernel*.

3.3.1. Intercambio (swap)

¿Qué hacer si no caben todos los programas en memoria principal? ⇒ Uso de intercambio (swapping)

- Swap ⇒ partición de disco que almacena imágenes de procesos
- Swap out ⇒ Cuando no caben en memoria procesos activos, se "expulsa" un proceso de memoria principal, copiando su imagen a swap (área de intercambio), aunque no es necesario copiar todo el mapa (ni código ni huecos). Existen diversos criterios de selección del proceso a intercambiar: (1) Dependiendo de la prioridad del proceso; (2) Preferencia a los procesos bloqueados; (3) No intercambiar si está activo DMA sobre mapa del proceso.
- Swap in ⇒ Cuando haya espacio en memoria principal, se intercambia el proceso a memoria copiando imagen desde swap.

Asignación de espacio en el dispositivo de swap: (1) Con preasignación ⇒ se asigna espacio al crear el proceso; (2) Sin preasignación ⇒ se asigna espacio al intercambiarlo (cuando se crea una región, no se hace ninguna reserva en el swap). La tendencia actual es utilizar la segunda estrategia, dado que la primera conlleva un peor aprovechamiento de la memoria secundaria, puesto que toda página debe tener reservado espacia en ella.

Dispositivo de *swap* ⇒ dispositivo de bloques en sección configurable del disco.

El *kernel* asigna espacio en el dispositivo de *swap* en grupos de bloques contiguos ⇒ Diferente al sistema de archivos (rapidez). Uno o varios dispositivos *swap*.

El *kernel* mantiene espacio libre en tabla en memoria llamada *map* o *mapa*. Mapa ⇒ array

- Entrada = (dirección recurso disponible, número unidades del recurso)
- Unidad en mapa del dispositivo de *swap* ⇒ grupo de bloques de disco
- Dirección ⇒ desplazamiento (offset) de bloques desde el principio del área de intercambio.
- Inicialmente ⇒ Figura 3.1.

Dirección	Unidades
Inicial	Todas

Figura 3.1. Situación inicial del mapa

3.3.1.1. Asignación del Espacio para Intercambio por parte del *Kernel*

Dirección map que indica qué map usar + número unidades pedidas:

- Éxito ⇒ la dirección
- Fracaso ⇒ 0

Asignar espacio para intercambio:

- El *kernel* busca en mapa 1ª entrada con suficiente espacio para la petición.
- Si petición consume todos los recursos de la entrada del mapa:
  - El *kernel* quita la entrada del array.
  - Mapa ⇒ Una entrada menos.
- En caso contrario
  - Ajusta dirección y unidades acorde a cantidad recursos asignados

Liberar espacio de intercambio:

- El *kernel* busca posición adecuada en el mapa por la dirección
- Casos posibles:
  - Recursos liberados llenan un hueco en el mapa ⇒ Combinar todos los recursos en una entrada en el mapa.
  - Recursos liberados llenan parcialmente un hueco en el mapa y son contiguos a alguna entrada adyacente ⇒ El *kernel* ajusta la dirección y unidades de la entrada apropiada de acuerdo a recursos liberados.
  - Recursos liberados llenan parcialmente un hueco en el mapa y no son contiguos a ninguna entrada ⇒ El *kernel* crea nueva entrada en el mapa para los recursos liberados y la inserta en la posición apropiada.

Ejemplo. Figura 3.2

- Dispositivo de *swap* con 10000 bloques empezando en la dirección 1 (a).
- Secuencia de peticiones y liberaciones:
  - El *kernel* asigna 100 unidades del recurso (b).
  - El *kernel* recibe una petición de 50 (c).
  - Recibe una última petición de 100 unidades (d).
  - Libera 50 unidades empezando en la dirección 101 (e).
  - Libera 100 unidades empezando en la dirección 1 (f).
  - Demanda 200 unidades (g).
  - Libera 350 empezando en dirección 151. Se asignaron por separado 0 (h).

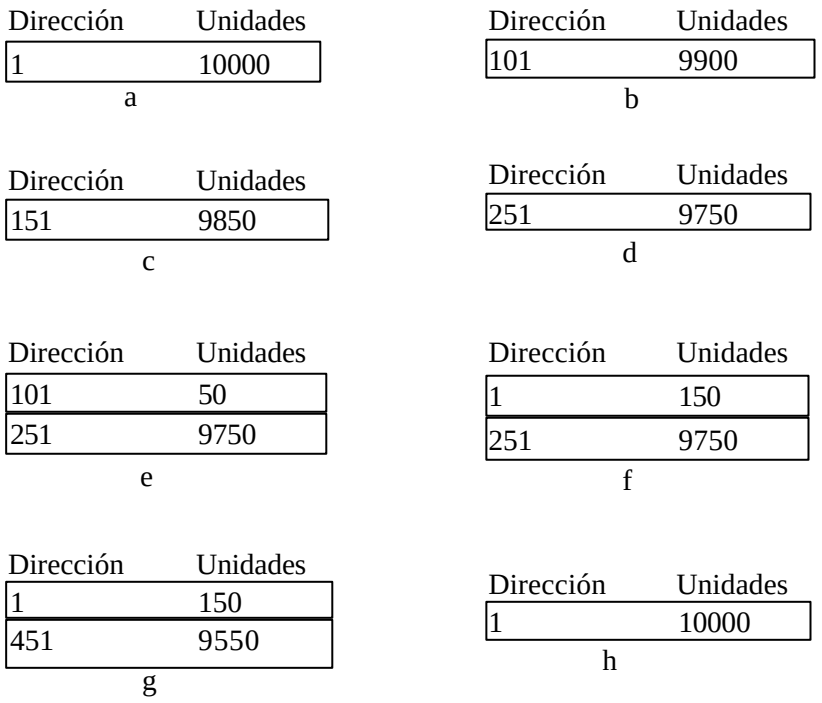

Figura 3.2. Asignación de espacio en el dispositivo de *swap*.

3.3.1.2. Intercambiar Procesos fuera de Memoria.

El *kernel* intercambia procesos al *swap* si necesita espacio en memoria:

- Fork: Asignar espacio para el proceso hijo (crea un nuevo proceso).
- Brk: Incremento tamaño del segmento de datos de un proceso.
- Crecimiento pila proceso.
- Para intercambiar procesos que había intercambiado al dispositivo de *swap*.

Todos los casos son iguales: (1) Se decrementa contador de referencia de cada región del proceso; (2) Lleva la región al *swap* si el contador de referencia el igual a 0; (3) Salva dirección de *swap* de la región en la tabla de regiones.
Diferencia ⇒ Primer caso no libera la copia en memoria ocupada por el proceso.
No pasa por buffer caché, ni intercambia direcciones del proceso que no tengan memoria asignada.
Ejemplo. Figura 3.3. Correspondencia de un proceso en el swap e intercambio de un proceso a memoria.

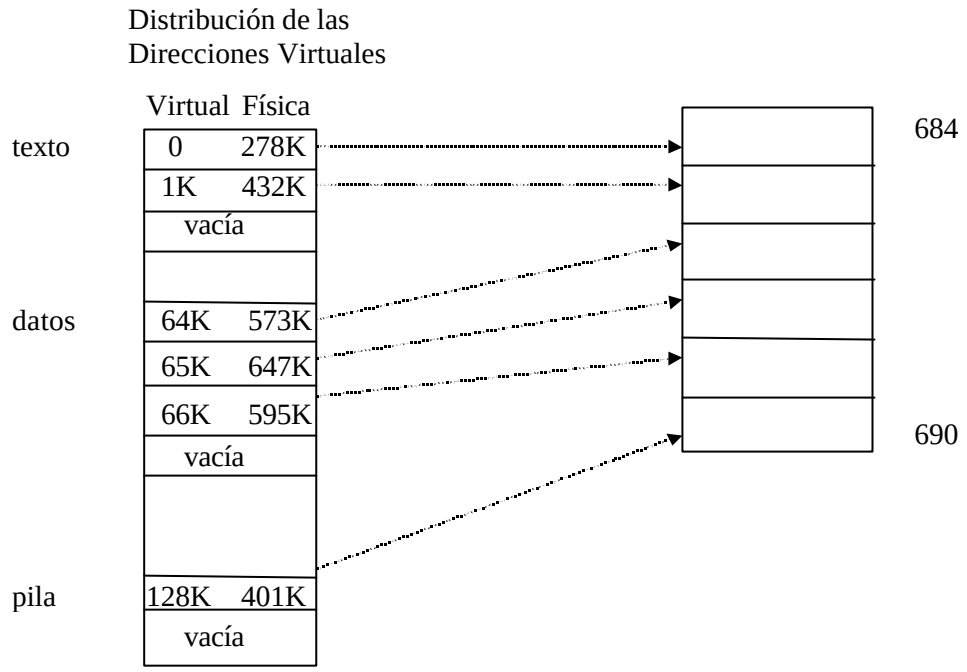

Figura 3.3.a. Correspondencia de un proceso en el dispositivo de *swap*.

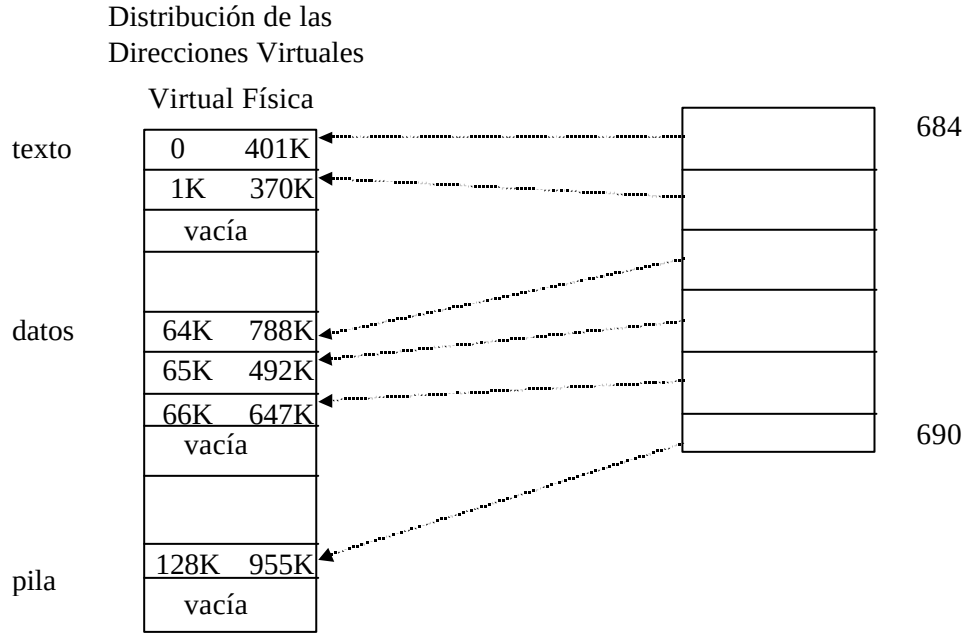

Figura 3.3.b. Intercambio de un proceso a memoria.

Fork:

- Padre no encuentra suficiente memoria para crear el contexto del proceso hijo ⇒ El *kernel* intercambia el proceso hijo al swap, no libera la memoria ocupada por padre.

- Fin swap: (1) El proceso hijo existirá en el dispositivo de swap; (2) El proceso padre lo pondrá en el estado "listo para ejecutarse" y volverá a modo usuario; (3) El swapper lo lleva a memoria cuando el *kernel* lo planifique (scheduler).

Swap de Expansión: Proceso requiere más memoria de la que tiene asignada en el momento (crecimiento pila o Brk) y no hay suficiente memoria disponible para satisfacer la petición ⇒ El *kernel* hace un *swap de expansión* del proceso en el que realiza los siguientes pasos: (1) Reserva suficiente espacio en *swap* para el proceso, incluyendo petición actual; (2) Ajusta el mapa de traducción de direcciones del proceso para la nueva configuración de la memoria virtual ⇒ no asigna memoria física porque no está disponible; (3) Intercambia el proceso fuera de memoria con nueva memoria igual a cero en el *swap*; (4) Al volver a memoria, el *kernel* le asignará suficiente memoria según el nuevo mapa de traducción de direcciones.

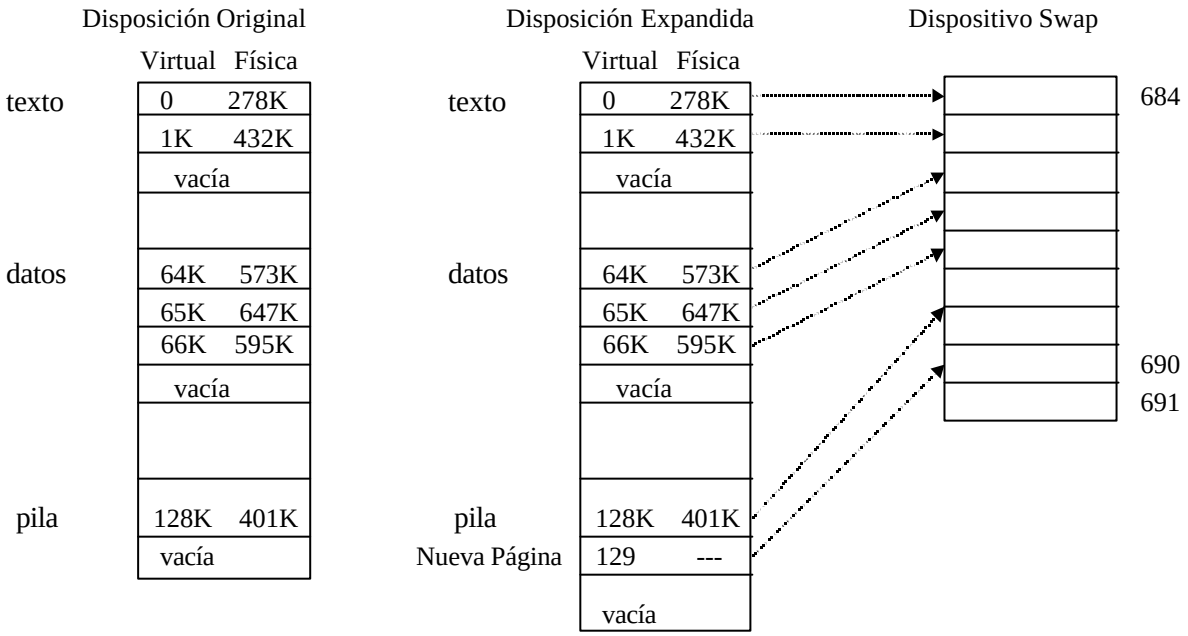

Figura 3.4. Ajuste del mapa de memoria en un *swap* de expansión.

3.3.1.3. Intercambiar Procesos entre Memoria Principal y swap

Proceso 0, swapper (proceso del *kernel*) ⇒ único capaz de mover procesos entre memoria y swap.
Comportamiento:

- Fin inicialización del sistema ⇒ swapper empieza un *bucle infinito*.
- Única tarea que tiene que hacer el swapper es intercambiar procesos desde swap: (1) Examina procesos "listos para ejecutarse en área de swap" y selecciona el que haya estado más tiempo intercambiado; (2) Si tiene suficiente memoria libre disponible asigna memoria física, lee el proceso desde el dispositivo de swap y libera la memoria en el mismo; (3) Si no tiene suficiente espacio en memoria principal, intenta intercambiar procesos al dispositivo de *swap*.
- Si no hay procesos para intercambiar a memoria, o ninguno de los procesos que hay en memoria pueden ser intercambiados al dispositivo de *swap* ⇒ swapper se *duerme*, y se despierta: (1) El reloj, lo despertará una vez cada segundo en este estado (dormido); (2) el *kernel* si otro proceso va a dormir; y (3) vuelve a empezar desde el principio buscando más procesos.

Criterios del swapper para elegir un proceso para echarlo (intercambiarlo) de memoria: (1) Examina cada uno de los procesos que estén en memoria; (2) No examina ni Zombies, ni procesos bloqueados en memoria; (3) Prefiere "Dormidos" a "listos para ejecutarse"; (4) Elegir de los "Dormidos" los que menos f(prioridad, tiempo que proceso lleva en memoria) tengan; (5) Elegir de los "listos para ejecutarse" los que menos f(nice, tiempo que el proceso haya estado en memoria) tengan.

Ejemplo. Figura 3.5, operaciones de intercambio. (1) Procesos A, B, C, D y E; (2) Todos son intensivos en CPU (misma prioridad); (3) Swapper tiene la mayor prioridad ⇒ se ejecutará cada segundo si hay carga de trabajo; (3) Todos los procesos tienen igual tamaño; (4) Puede haber en memoria sólo dos procesos simultáneamente; (5) Situación inicial: A y B en memoria y los demás intercambiados (en *swap*).

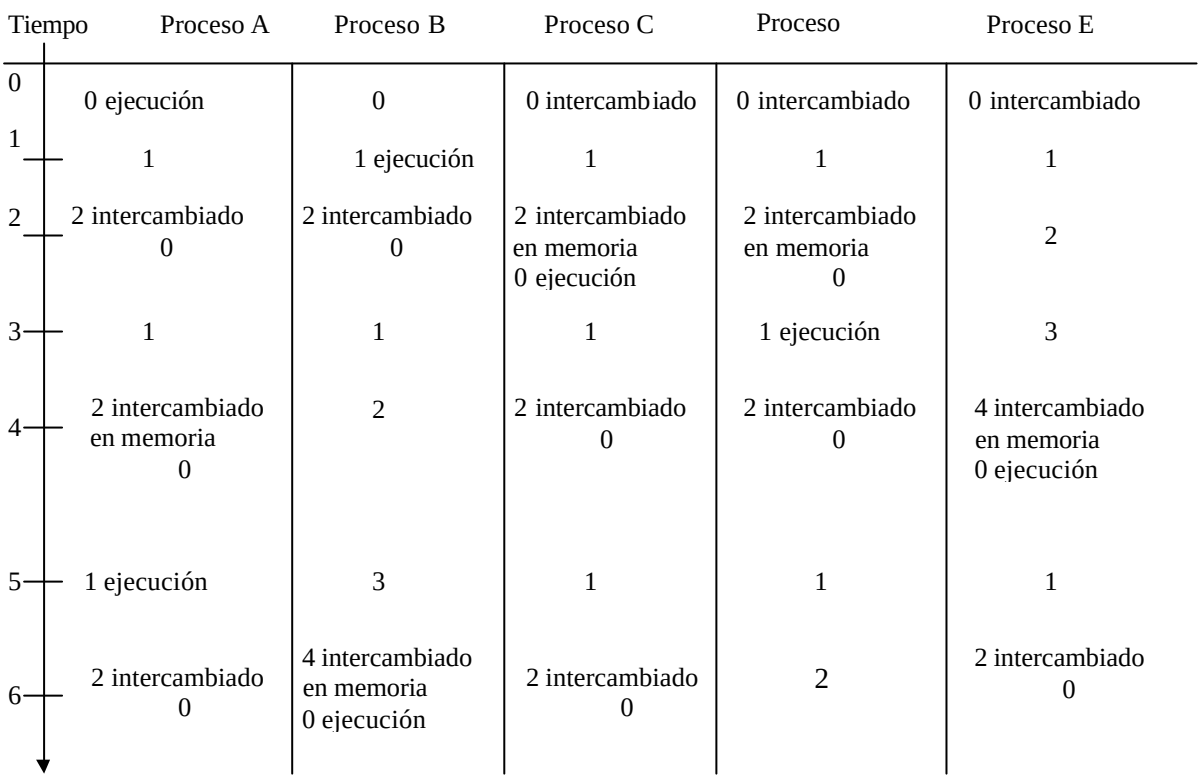

Figura 3.5. Ejemplo de operaciones de intercambio.

Problemas:

1. El swapper intercambia fuera en función de prioridad, tiempo de permanencia en memoria y valor nice ⇒ Proceso elegido puede no suministrar suficiente memoria. *Alternativa*: intercambiar fuera grupos sólo si dan suficiente memoria.
2. Si el swapper duerme porque no pudo encontrar suficiente memoria para traer un proceso, cuando despierta, busca otra vez un proceso para traerse al memoria aunque ya había encontrado uno antes porque pudo despertar proceso más elegible ⇒ Proceso sigue en el dispositivo de swap. *Alternativa*: el swapper intenta intercambiar fuera muchos procesos más pequeños para hacer sitio al proceso más grande, que estamos intentando intercambiar a memoria, antes de buscar otro proceso para intercambiar a memoria.
3. El swapper puede elegir un proceso que esté en estado "listo para ejecutarse", para intercambiarlo fuera de memoria (*swap*) en caso de que no se hubiera ejecutado nunca.
4. El swapper intenta intercambiar fuera un proceso y no encuentra espacio en *swap* ⇒ deadlock. Condiciones: (1) Todos los procesos en memoria principal están dormidos; (2) Todos los procesos "listos para ejecutarse" están intercambiados fuera de memoria principal; (3) No hay sitio en el dispositivo de *swap* para nuevos procesos.

3.3.2. Paginación por Demanda

3.3.2.1. Visión General

Hardware: arquitectura basada en páginas y CPU instrucciones rearrancables ⇒ Soporta que el *kernel* implemente un algoritmo de demanda de páginas. Un proceso no tiene que caber entero en memoria física

para ejecutarse. La carga en memoria principal de una parte de un proceso relevante se hace de forma dinámica.

Transparente a programas de usuario excepto por el tamaño virtual permisible a proceso (controlado por el kernel).

Working Set (conjunto de trabajo) ⇒ El conjunto de trabajo de un proceso en un instante virtual t y con un parámetro Δ, denominado $W(t, \Delta)$, es el conjunto de páginas a las que el proceso ha hecho referencia en las últimas Δ unidades de tiempo virtual, siendo el tiempo virtual el tiempo que transcurre mientras el proceso está realmente en ejecución.

Si un proceso direcciona una página que no pertenece al Working Set ⇒ provoca una falta de página, entonces el *kernel* actualiza el Working Set, leyendo páginas desde memoria secundaria si es necesario.

Este concepto de Working Set puede motivar la siguiente estrategia para el tamaño del conjunto residente de un proceso (parte del proceso que está realmente en memoria principal): (1) Supervisar el conjunto de trabajo de cada proceso; (2) Eliminar periódicamente del conjunto residente de un proceso aquellas páginas que no pertenezcan a su conjunto de trabajo. (3) Un proceso puede ejecutarse sólo si su conjunto de trabajo está en la memoria principal, es decir, si su conjunto residente incluye a su conjunto de trabajo.

Problemas de la estrategia del conjunto de trabajo: (1) Tanto el tamaño como el conjunto de trabajo variarán con el tiempo, siendo desconocido el valor óptimo de Δ; (2) El casi impracticable una medida real del conjunto de trabajo, ya que sería necesario marcar cada referencia de página de cada proceso con su tiempo virtual y mantener una cola ordenada en el tiempo de cada proceso.

3.3.2.2. Estructuras de Datos para Paginación por Demanda (UNIX)

Estructuras de datos para la gestión de memoria a bajo nivel y demanda de páginas (sistema de paginación):

- Tabla de páginas ⇒ Normalmente, hay una tabla por proceso, con una entrada para cada página de la memoria virtual del proceso.
- Descriptores de bloques de disco ⇒ Asociado a cada página de un proceso hay una entrada en la tabla que describe la copia en el disco de la página virtual.
- Tabla de marcos de página (pfdata) ⇒ Describe cada marca de la memoria real y está indexada por el número de marco.
- Tabla de uso de *swap* ⇒ Existe una tabla de uso de *swap* por cada dispositivo de intercambio, con una entrada para cada página en dicho dispositivo.

El *kernel* asigna espacio para pfdata una vez durante la vida del sistema, aunque para las otras estructuras le asigna y desasigna espacio dinámicamente. Cada región contiene tablas de páginas para acceder a memoria física.

Tabla de Páginas.

- Entrada de la tabla de páginas: (1) *Número de marco de página*, es la dirección física de la página (marco en la memoria real); (2) *Protección*, que indican si procesos pueden leer, escribir o ejecutar página (indica si está permitida la operación de escritura); (3) *Válido*, que indica si la página está en la memoria principal; (4) *Referencia*, que indica si un proceso referenció recientemente la página (se pone a cero cuando la página se carga por primera vez y puede ser restaurado periódicamente por el algoritmo de reemplazo de página); (5) *Modificado*, que indica si proceso modificó recientemente contenido de la página; (6) *Copia en escritura*, usado en fork, indica que el *kernel* debe crear nueva copia de la página cuando un proceso modifique su contenido (activo cuando más de un proceso comparte la página). (7) *Edad*, indica cuánto hace que la página pertenece al Working Set del proceso (indica cuánto tiempo ha estado la página en memoria sin ser referenciada).
- El kernel manipula: *válido*, *copia en escritura* y *edad*. El hardware activa: *referencia* y *modificado* de cada entrada de la tabla de páginas.

Descriptores de bloques de disco.

- Entrada del descriptor de bloques de disco (describe copia de disco de la página virtual): (1) *Número de dispositivo de swap*, indica el número de dispositivo lógico del dispositivo secundario que contiene la página correspondiente (se puede utilizar más de un dispositivo para el intercambio); (2) *Número de bloque del dispositivo*, indica la posición del bloque que contiene la página en el dispositivo *swap*; (3) *Tipo de almacenamiento*, el almacenamiento puede ser la unidad de *swap* o un archivo ejecutable, en este último caso, existe un indicador de si debe borrarse primero la memoria virtual a asignar.
- Los procesos que comparten una misma región ⇒ Entradas comunes para la tabla de páginas y los descriptores de bloques de disco.

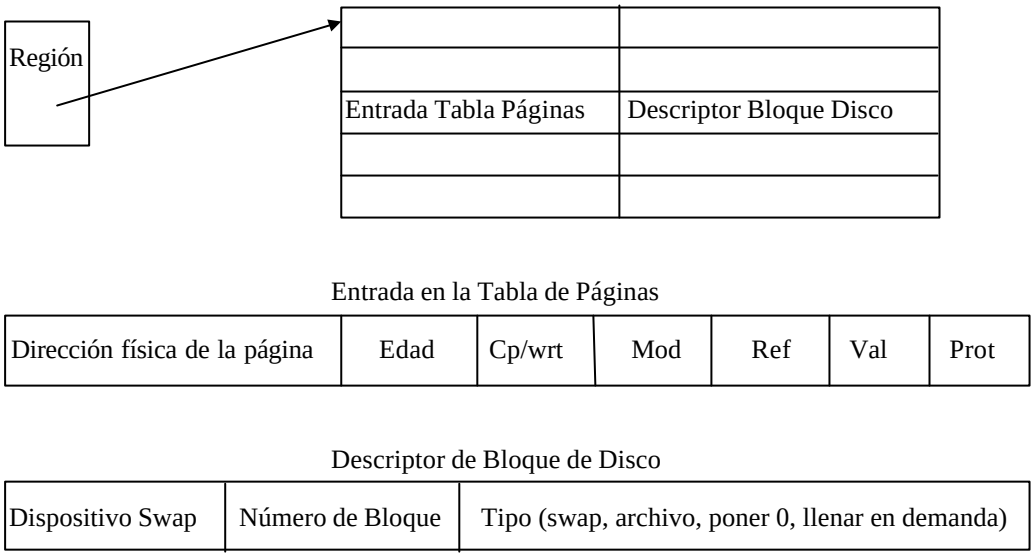

Figura 3.6. Entrada en la tabla de Páginas y del descriptor de bloques de disco.

Contenidos de una página virtual: (1) Bloque particular del dispositivo de swap ⇒ descriptor de bloques de disco contiene el número de dispositivo lógico y el número de bloque que contiene la página; (2) Archivo ejecutable ⇒ descriptor de bloques de disco contiene el número de bloque lógico que contiene la página virtual; (3) No está en swap (página sin contenido) ⇒ descriptor de bloques de disco indica condiciones especiales.

Tabla de Marcos de Páginas (pfdata).

- Describe páginas de memoria física y se indexa por número de página.
- Entrada de la tabla de marcos de página: (1) *Estado de la página*, indica si el marco de página está disponible (puede ser asignado) o tiene una página asociada, en este último caso, especifica el estado de la página: en el dispositivo de *swap*, en un archivo ejecutable o en una operación de DMA pendiente); (2) *Contador de referencias*, indica el número de procesos que hacen referencia a la página ⇒ es igual al número entradas válidas en la tabla de páginas referenciando a la página, aunque puede ser distinto del número de procesos que comparten regiones que contengan la página; (3) *Dispositivo lógico* (swap), indica el dispositivo lógico que contiene una copia de la página; (4) *Número de bloque*, que indica la posición del bloque de la copia de la página en el dispositivo; (5) *Punteros a marco de página*, es un puntero a otras entradas de la tabla de marcos (pfdata) en una lista de páginas libres y lista hash de páginas.
- Lista de páginas libres: (1) Caché de páginas disponibles para reasignar, (2) El proceso provoca falta de página y puede encontrarla intacta en lista de páginas libres; (3) Permite al *kernel* evitar operaciones de lectura innecesarias; (4) Orden asignación páginas libres es LRU (menos utilizada recientemente).
- Lista hash de páginas: (1) Función hash depende del número de dispositivo lógico y del número de bloque de la página ⇒ localización rápida de la página si está en memoria.

- Proceso de asignación de página física a una región (similar al buffer caché): (1) El *kernel* quita una entrada de una página libre de la cabeza de la lista de páginas libres; (2) Actualiza su dispositivo de *swap* y el número de bloque; (3) La pone en la cola hash específica según el dispositivo de swap (lógico) y el número de bloque.

Tabla de uso de swap.

- Contiene una entrada por cada página en el dispositivo de swap.
- Entrada de la tabla de uso del *swap*: (1) *Contador de referencias*, que representa el número de entradas en tabla de páginas que apuntan a páginas situadas en el dispositivo de *swap*. (2) *Página/número de unidad de almacenamiento*, identificador de la página en la unidad de almacenamiento.

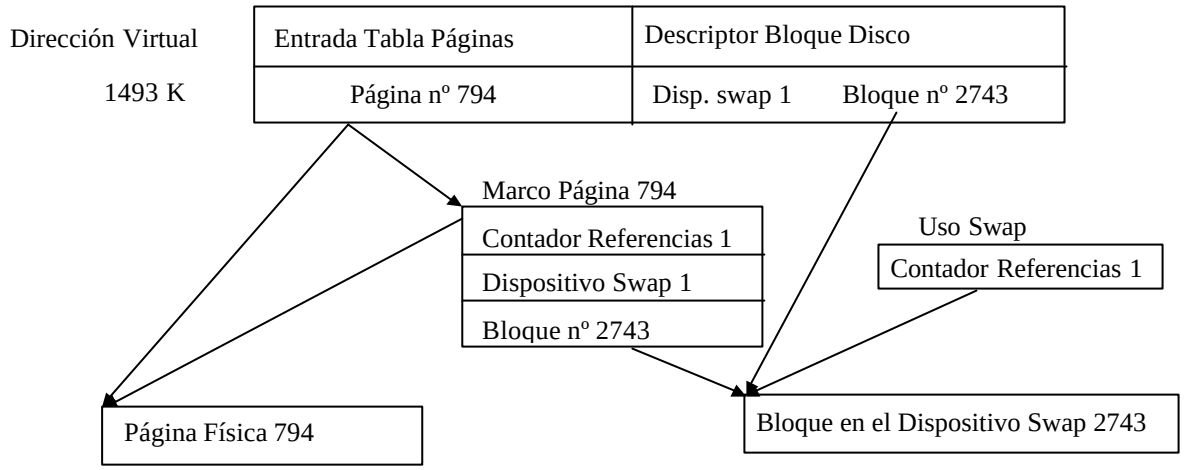

Figura 3.7. Relaciones entre estructuras de datos para demanda de páginas.

Sistema de paginación de UNIX System V. El *kernel* evita copiar la página, incrementa el contador de referencia regiones compartidas y para las regiones privadas asigna una nueva entrada en la tabla de regiones y tabla de páginas. Además, el *kernel* examina cada entrada en la tabla de regiones del proceso padre: Si la página es válida: (1) Incrementa el contador de referencia en la entrada de la tabla de marcos de páginas (pfdata), (2) Si la página existe en un dispositivo de swap, entonces contador de referencias de la tabla de uso del *swap* para página se incrementa en (Figura 3.8, fork en demanda de páginas), referencias a página a través de ambas regiones; (3) Si un proceso escribe en ella (página) $\Rightarrow$ *kernel* crea versión privada $\rightarrow$ el *kernel* pone bit *Copia en escritura* = 1 para cada entrada de la tabla de páginas en regiones privadas del proceso padre y del proceso hijo durante el fork.

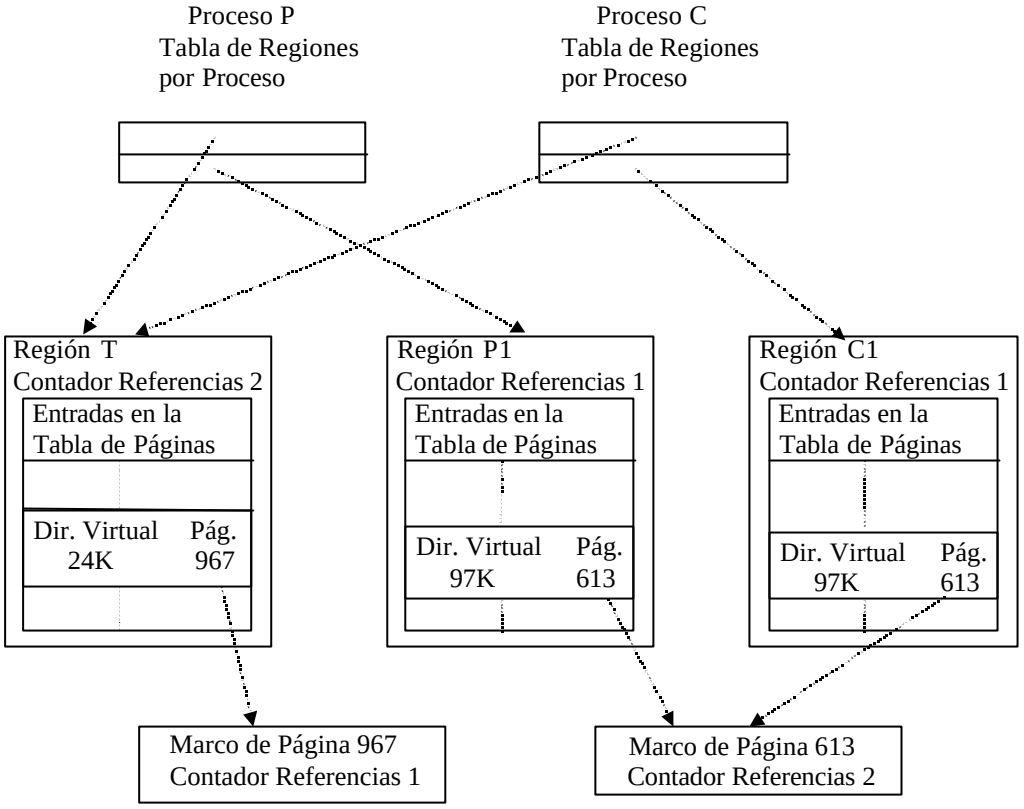

Figura 3.8. Fork en demanda de páginas.

Sistema de paginación en BSD. (1) Fork ⇒ Copia física de las páginas del proceso padre. (2) Vfork ⇒ Asume que el proceso hijo llama a exec a continuación ⇒ (2.1) No copia tablas de páginas → Rapidez; (2.2) Proceso hijo se está ejecutando en el mismo espacio de direcciones físicas que su padre hasta el exec o el exit → Podría sobreescribir en la pila y en los datos del proceso padre.

3.3.2.3. Reemplazo de Páginas

La tabla de marcos de página se utiliza en el reemplazo de páginas. Se emplean varios punteros para crear listas dentro de una tabla. Todos los marcos de páginas disponibles se encadenan en una lista de marcos de páginas vacíos disponibles para cargar páginas (lista de páginas libres). Cuando el número de páginas disponibles es inferior a un cierto umbral, el *kernel* "robará" varias páginas para compensarlo. El algoritmo de reemplazo de página en UNIX es un refinamiento del la política del reloj (clock), conocido como el "algoritmo del reloj de dos agujas". El algoritmo utilizará un bit de referencia en la entrada de la tabla de páginas para cada página de la memoria que no esté bloqueada para ser "expulsada". Este bit se pone a 0 cuando la página entra por primera vez y a 1 cuando se hace referencia a la página para una lectura o escritura. Una aguja del algoritmo del reloj, la aguja frontal, recorre la lista de páginas elegibles y pone el bit de referencia a 0 en cada página. Algún tiempo más tarde, la aguja trasera recorre la misma lista y comprueba el bit de referencia. Si el bit está a 1, entonces la página se ha referenciado desde que la aguja frontal hizo el recorrido y se ignora el marco. Si el bit está todavía a 0, entonces la página no se ha referenciado en el intervalo de tiempo entre la visita de la aguja frontal y la aguja trasera, y se pone a estas páginas en una lista para ser reemplazadas.

Dos parámetros determinan la operación del algoritmo: (1) Velocidad de recorrido, que indica la velocidad con la cual las dos agujas se mueven a través de la lista de páginas (páginas por segundo); (2) Alcance entre agujas, que indica el espacio entre la aguja frontal y la aguja trasera (páginas). Estos dos parámetros tienen valores por defecto asignados en el arranque y basados en la cantidad de memoria física disponible. El parámetro de velocidad de recorrido puede modificarse para cumplir condiciones cambiantes, y varía linealmente entre los valores de recorrido lento y recorrido rápido (asignados en la configuración) conforme la cantidad de memoria libre varía entre los valores "mucho_libre" y "poco_libre". Es decir, conforme la cantidad de memoria libre desminuye, las agujas del reloj se mueven más rápidamente para liberar más páginas. El parámetro de alcance entre agujas determina el espacio entre la aguja frontal y la trasera, y

además, junto con la velocidad de recorrido, determina la ventana de oportunidades de recuperar una página antes de "expulsarla" debido a su falta de uso.

3.3.3. Gestor de Memoria del *Kernel*

El *kernel* genera y destruye frecuentemente pequeñas tablas y buffers durante la ejecución, cada una de las cuales requiere asignación de memoria dinámica, por ejemplo, asignar objetos (estructuras de procesos, inodos, bloques de descriptores de archivo, etc.) dinámicamente cuando se necesitan. La mayoría de estos objetos son significativamente más pequeños que el tamaño de página en las máquinas normales, y por tanto el mecanismo de paginación sería ineficaz para la asignación de memoria dinámica del *kernel*. En SVR4, se utiliza una modificación del *Buddy System* (sistema de colegas). En dicho sistema, el coste de asignar y liberar un bloque de memoria es bajo comparado con otras políticas (mejor ajuste o primer ajuste). No obstante, en el caso de la gestión de memoria del kernel, las operaciones de asignación y liberación tienen que hacerse tan rápido como sea posible. El inconveniente que tiene el *Buddy System* es el tiempo requerido para fragmentar y unir bloques.

3.3.3.1. *Buddy System* (sistema de colegas)

El *Buddy System* es un esquema para la gestión de la partición de memoria que trata de encontrar un equilibrio entre los esquemas de partición estáticos (limitan el número de procesos activos y pueden utilizar el espacio ineficientemente si hay poca concordancia entre los tamaños de las particiones disponibles y los tamaños de los procesos) y dinámicos (más complejo de mantener y incluye la sobrecarga de compartir).

En un *Buddy System*, los bloques de memoria disponibles son de tamaño 2^K, para valores de K tal que $L \leq K \leq U$ y donde: 2^L = tamaño de bloque más pequeño asignable, 2^U = tamaño de bloque más grande asignable (generalmente, 2^U es el tamaño de memoria completa disponible para asignación o gestión).

Para empezar, el espacio entero disponible para la asignación se trata como un solo bloque de tamaño 2^U. Si se hace una solicitud de tamaño *s* tal que $2^{U-1} < s \leq 2^U$, entonces el bloque entero se asigna. En otro caso, el bloque se divide en dos colegas (buddies) de igual tamaño 2^{U-1}. Si $2^{U-2} < s \leq 2^{U-1}$, entonces la solicitud se asigna a uno de los dos colegas. Si no, uno de los colegas se divide por la mitad nuevamente. Este proceso continua hasta que el bloque más pequeño sea mayor o igual que *s*, generándose y asignándose a la solicitud. En cualquier instante, el *Buddy System* mantiene una lista de huecos (bloques no asignados) para cada tamaño 2^i. Un hueco puede eliminarse de la lista (i + 1) dividiéndolo en dos mitades para crear dos colegas (buddies) de tamaño 2^i en la lista i. Cuando una pareja de colegas de la lista i pasa a estar libre, se elimina de la lista y se unen en un solo bloque de la lisa (i + 1). Dada una solicitud para una asignación de tamaño *k*, tal que $2^{i-1} < k \leq 2^i$, para encontrar un hueco de tamaño 2^i se utiliza el siguiente algoritmo recursivo:

```
Conseguir_hueco(i)
{
        if (i = (U + 1))
                Fallo;
        if (lista I vacía)
        {
                Conseguir_hueco(i + 1);
                Dividir el hueco en colegas (buddies);
                Poner colegas (buddies) en lista i;
        }
        Coger el primer hueco en la lista i;
}
```

La mejor estructura de datos para implementar este esquema de partición es un árbol binario, donde los nodos hoja representan la partición actual de la memoria, si dos colegas (buddies) están en nodos hoja, entonces el

menos uno de ellos está signado; en otro caso, se liberaría para que quedara sólo el nodo padre de ambos (unirían en un bloque mayor).

3.3.3.2. *Buddy System* retardado

Esta técnica es la que adopta SVR4. Los autores observan que UNIX exhibe, a menudo, un comportamiento estable en la demanda de memoria del *kernel*, es decir, la cantidad de demandas para bloques de un tamaño determinado varían lentamente en el tiempo. De esta forma, si se libera un bloque de tamaño 2^i e inmediatamente se une con su colega (buddy) en un bloque de tamaño 2^{i+1}, el *kernel* puede, a continuación, hacer una solicitud de un bloque de tamaño 2^i que necesariamente divide de nuevo el bloque mayor. Para evitar estas uniones y divisiones innecesarias, el *Buddy System* retardado difiere la unión hasta que parezca que sea necesaria, y entonces se une tantos bloques como sea posible.

El *Buddy System* retardado utiliza los siguientes parámetros: (1) N_i = número actual de bloques de tamaño 2^i; (2) A_i = número actual de bloques de tamaño 2^i que están asignados (ocupados); (3) G_i = número de bloques de tamaño 2^i que están libres globalmente; estos son bloques elegibles para la unión; si el colega (buddy) de un bloque de este tipo se convierte en un bloque libre globalmente, entonces los dos bloques se unirán en un bloque libre globalmente de tamaño 2^{i+1}. Todos los bloques libres (huecos) en el *Buddy System* estándar se podrían considerar libre globalmente; (4) L_i = número actual de bloques de tamaño 2^i que están libres localmente; estos bloques no son elegibles para la unión. Aunque, el colega de estos bloques se convierta en libre, los dos bloques no se unen. Más bien, los bloques libres localmente quedan retenidos como anticipación de futuras demandas de un bloque de ese tamaño. Se cumple la siguiente relación: $N_i = A_i + G_i + L_i$.

En general, el *Buddy System* retardado trata de mantener una reserva de bloques libres localmente y solamente invoca uniones si el número de bloques libres localmente excede un umbral. Si hay demasiados bloques libres en el siguiente nivel para satisfacer las demandas. La mayor parte del tiempo, mientras los bloques estén libres, no se produce la unión, para que la sobrecarga de operaciones y contabilidad se minimicen. Cuando un bloque va a ser asignado, no se distingue si se hace de entre los bloques libres local o globalmente, también esto minimiza la contabilidad. El criterio utilizado para la unión es que el número de bloques libres localmente de un tamaño dado no excedan del número de bloques asignados de este tamaño (es decir, debe ser $L_i \leq A_i$). Esta es una guía razonable para restringir el crecimiento de los bloques libres localmente, confirmándose que este esquema genera un notable ahorro. Para implementar este esquema, se define un retraso variable según la siguiente ecuación: $D_i = A_i - L_i = N_i - 2\ L_i - G_i$. A continuación se muestra el algoritmo,

```
Di = 0;        // Después de una operación el valor de Di se actualiza como sigue:
if (siguiente operación es una solicitud de asignación de bloque)
{
        if (existe algún bloque libre)
        {
                Seleccionar uno para asignarlo;
                if (bloque seleccionado está localmente libre)
                        Di = Di + 2;
                else
                        Di = Di + 1;
        }
        else
        {
                Conseguir dos bloques dividiendo en dos uno más grande (operación recursiva);
                Asignar uno y marcar el otro libre localmente;
                // Di no cambia, aunque puede cambiar a otros tamaños de bloque por la llamada
recursiva
        }
}
```

```
if (siguiente operación es una solicitud de bloque libre)
{
        if (Di ≥ 2)
        {
                Marcarlo libre localmente y liberarlo localmente;
                Di = Di – 2;
        {
        else
        {
                if (Di = 1)
                {
                        Marcarlo libre globalmente, liberarlo globalmente y unirlo si es posible;
                        Di = 0;
                }
                else
                {
                        if (Di = 0)
                        {
                                Marcarlo libre globalmente, liberarlo globalmente y unirlo si es posible;
                                Seleccionar un bloque de tamaño 2i y liberarlo globalmente y unirlo si es posible;
                                Di = 0;
                        }
                }
        }
}
```

3.4. GESTIÓN DE MEMORIA EN LINUX

3.4.1. Visión General

Linux comparte muchas de las características de los esquemas de gestión de memoria de otras implementaciones UNIX, pero tiene sus características propias y únicas, aunque hay que destacar que el esquema de gestión de memoria de Linux es bastante complejo.

En lo que respecta a memoria virtual, el direccionamiento de memoria virtual de Linux, hace uso de una estructura de tabla de páginas con tres niveles, formada por los siguientes tipos de tablas (cada tabla individual es del tamaño de una página): (1) Directorio de páginas ⇒ un proceso activo tiene un solo directorio de páginas que es del tamaño de una página. Cada entrada en el directorio de páginas apunta a una página del directorio intermedio de páginas. Para un proceso activo, el directorio de páginas tiene que estar en la memoria principal; (2) Directorio intermedio de páginas ⇒ este directorio puede ocupar varias páginas y cada entrada de este directorio apunta a una página de la tabla de páginas; (3) Tabla de páginas ⇒ esta tabla de páginas también puede ocupar varias páginas, y cada entrada de la tabla de página hace referencia a una tabla virtual del proceso.

Para utilizar esta estructura de la tabla de páginas a tres niveles, una dirección virtual en Linux se ve como un conjunto de cuatro campos. El campo más a la izquierda (más significativo) se utiliza como índice en el directorio de páginas. El siguiente campo sirve como índice en el directorio intermedio de páginas. El tercer campo sirve como índice en la tabla de páginas. Y el cuarto y último campo, indica el desplazamiento dentro de la página seleccionada de la memoria. La estructura de tabla de página en Linux es independiente de la plataforma y fue diseñada para ajustarse al procesador Alpha de 64 bits, el cual proporciona soporte de hardware para los tres niveles de paginación. Con direcciones de 64 bits, el uso de sólo dos niveles de páginas en el procesador Alpha generaría tablas de páginas y directorios muy grandes. Los 32 bits de la arquitectura

i386 tienen un mecanismo de paginación a dos niveles en su hardware. El software de Linux se ajusta al esquema de dos niveles definiendo el tamaño del directorio intermedio como uno.

Para la asignación de páginas y aumentar la eficiencia de cargar y descargar páginas a y desde la memoria principal, Linux define un mecanismo para tratar bloques de páginas contiguos correspondientes a bloques de marcos de páginas contiguos. Pare este propósito, se utiliza el *Buddy System*. El *kernel* mantiene una lista de grupos de marcos de página contiguos de tamaño fijo; un grupo puede estar formado por 1, 2, 3, 8, 16 o 32 marcos de páginas. Como las páginas se asignan y liberan en la memoria principal, los grupos se dividen y se intercalan utilizando el algoritmo de los colegas (buddy algorithm).

El algoritmo de reemplazo de páginas de Linux se basa en el algoritmo del reloj (clock). En el algoritmo de reloj estándar, se asocia un bit de uso y un bit de modificación a cada página de la memoria principal. En el esquema de Linux, una variable de edad de 8 bits sustituye al bit de uso. Cada vez que se accede a una página, se incrementa la variable edad, Linux recorre periódicamente la reserva de páginas globales y disminuye la variable de edad de cada página cuando rota por todas las páginas de la memoria principal. Una página con un envejecimiento 0 es una página "vieja" que no se ha referenciado en bastante tiempo y es la mejor candidata para el reemplazo. Cuando mayor valor de edad, más frecuentemente se ha usado la página recientemente y menos elegible es para el reemplazo. De esta forma, el algoritmo de reemplazo de páginas de Linux es una aproximación de la política LRU.

Los fundamentos de asignación de memoria al *kernel* de Linux es el mecanismo de asignación de páginas utilizado para la gestión de memoria virtual de usuario. Como en el esquema de memoria virtual se utiliza un algoritmo de colegas (buddy algorithm) par asignar y liberar la memoria del *kernel* en unidades de uno o más páginas. Puesto que la mínima cantidad de memoria que se puede asignar de esta forma es una página, el gestor de páginas solamente podría ser ineficiente si el *kernel* requiere pequeños trozos de memoria a corto plazo y en tamaños irregulares. Para gestionar a estas pequeñas asignaciones, Linux utiliza un esquema conocido como asignación por fragmentos (slab allocation) dentro de una página asignada. En un procesador i386, el tamaño de página es de 4Kbytes y los fragmentos que se pueden asignar dentro de una página pueden ser de 32, 64, 128, 252, 508, 2040 y 4080 bytes. La asignación de fragmentos es relativamente compleja, en general, Linux mantiene un conjunto de listas enlazadas, una por cada tamaño de fragmento. Los fragmentos pueden dividirse y agregarse de forma similar al algoritmo de los colegas (buddy algorithm) y se mueve entre las listas en función de ello.

3.4.2. Direccionamiento de Memoria en Linux

La memoria es uno de los recursos fundamentales para un proceso. El sistema operativo debe ofrecer la memoria a todos los procesos por igual de una forma sencilla y uniforme. Al mismo tiempo, el sistema operativo debe tratar con el hardware real para realizar dicha función, aprovechándolo al máximo. El direccionamiento es una parte importante de la gestión de memoria, puesto que influye mucho tanto en la visión de la misma por parte de un proceso, como en el aprovechamiento del hardware y el rendimiento del sistema.

Desde el punto de vista de la arquitectura, el sistema operativo suele tener asistencia del hardware para realizar la gestión memoria: (1) *Memory Management Unit* (MMU), unidad que realiza, en el i386, segmentación y paginación; (2) Bits reservados en selectores de segmento; (3) Bits reservados en descriptores de página; (4) Bits de protección. Sin embargo, el sistema operativo es libre de usar o no dichos recursos. Por ejemplo, la segmentación de los i386 no se emplea con toda su potencia en Linux. Por otro lado, el sistema operativo debe suplir carencias del hardware si es posible. Por ejemplo, usando bits disponibles o con técnicas más sofisticadas.

Desde el punto de vista del proceso, son deseables ciertas características relacionadas con el direccionamiento: (1) Protección. La memoria de un proceso debe estar separada de la memoria de los demás procesos. Salvo para threads del mismo proceso. En Linux un proceso realmente tiene un espacio de direcciones de usuario, que es propio, y un espacio de direcciones del kernel, que comparte con todos los

procesos. (2) Memoria virtual. El proceso debe tener la ilusión de estar solo en el sistema. Espacio de direcciones contiguo y que comienza en 0. Esto facilita el trabajo de los compiladores.

Otras características de la gestión de memoria también son deseables, si bien están menos relacionadas con el direccionamiento: (1) Ofrecer al proceso más memoria de la que hay físicamente disponible. Se emplean técnicas de *swapping* y paginación por demanda. (2) Aprovechar la memoria mediante técnicas *Copy-on-write* . (3) Mapeado de ficheros sobre memoria. (4) En general, mejorar el rendimiento del sistema mediante diversas técnicas a la hora de asignar o liberar memoria *(Buddy system, Slab allocator,* caches, etc).

3.4.2.1. Espacios de Direcciones

Tres tipos de direcciones:

- *Direcciones lógicas*. Generadas por el proceso, cada dirección lógica consiste en un selector de segmento y un desplazamiento (offset) que denota la distancia del principio del segmento a la dirección actual.
- *Direcciones lineales (direcciones virtuales)*. Obtenidas tras aplicar una transformación a la dirección lógica por parte de la MMU. 32 bits se pueden utilizar para direccionar 4Gb (es decir 4294967296 direcciones físicas de memoria). Las direcciones lineales se representan normalmente en hexadecimal, su rango de valores va desde 0x00000000 hasta 0xffffffff.
- *Direcciones físicas*. Referencian la memoria física. Se obtienen tras aplicar una transformación por parte de la MMU.

Las transformaciones y el formato de las direcciones dependen de la arquitectura. En Linux los espacios de direcciones lógico y lineal son idénticos.

En los procesadores de la arquitectura i386:

- El paso de dirección lógica a lineal se denomina segmentación.
- El paso de dirección lineal a física se denomina paginación. Si se deshabilita la paginación, dirección física = dirección lineal.

La memoria virtual se soporta en la paginación y en bits de presencia y referencia.
dirección_logica (48 bits = segmento + desplazamiento) ⇒ unidad de segmentación ⇒ dirección_lineal (32 bits) ⇒ unidad de paginación ⇒ dirección_física (32 bits).
Tablas separadas para la segmentación y paginación. Las direcciones lógicas son de 48 bits divididas en dos partes: descriptor de segmento (16 bits) y desplazamiento (32 bits). Las direcciones lineales y físicas son de 32 bits. En Linux, dado que no puede desactivarse la segmentación, la identidad entre los espacios lógico y lineal se consigue realizando la programación adecuada.

3.4.2.2. Segmentación en el i386

Características de i386: (1) microprocesador de 32 bytes, (2) hasta de 4 Gbytes de memoria física direccionable; (3) soporta segmentación y paginación (página de 4 Kbytes); (4) 4 niveles de privilegio (0 – 3); (5) Soporta mecanismos de multiproceso (multitarea) y de protección entre procesos, incluido protección de memoria; (6) modo de compatibilidad con 8086 (modo real); (7) soporta un coprocesador x87; (8) presentado en 1985, la arquitectura se mantiene hasta hoy (Pentium 4) con mejoras estructurales, 100% compatible hacia atrás.

Segmentación = traducción de dirección virtual a dirección lineal (o virtual). Segmento = Espacio contiguo de dirección lineales. Cada segmento está caracterizado por: dirección base (dirección lineal de inicio de segmento), límite (tamaño en bytes o páginas), atributos.

El i386 contiene registros de segmentación para hacer más rápida la selección de selectores de segmentos. Estos registros son: CS, SS. DS, ES, FS y GS. Aunque estos son sólo 6 de todos los que dispone el i386, un programa puede reutilizar los mismos registros de segmentación para diferentes propósitos para guardar su contenido en memoria y restaurarlo más tarde. Tres de estos seis registros de segmentación tienen propósitos

específicos (ES, FS y GS son registros de propósito general): (1) CS (registro de segmento de código, que apunta a una segmento que contienen instrucciones del programa, incluyendo dos campos de bits para el nivel de privilegio actual (CPL, current privilege level) de la CPU, valor 0 denota el nivel de privilegio más grande (modo kernel) y 3 denota la menor prioridad (modo usuario)); (2) SS (registro de segmento de pila, que apunta a un segmento que contiene la pila del programa actual); y (3) DS (registro de segmento de datos, que apunta a un segmento que contiene datos externos y estáticos).

El registro contiene un *selector de segmento* $\Rightarrow$ especifica el descriptor de segmento dentro de una tabla de descriptores de segmentos. Con este selector se obtiene un *descriptor de segmento*. Con el descriptor (en la tabla de descriptores) y el offset (desplazamiento) se obtiene la dirección lógica o lineal. Los descriptores de segmento residen en dos tablas, la GDT = Tabla de Descriptores Global (global a todos los procesos) y la LDT = Tabla de Descriptores Locales (local a cada proceso). La posibilidad de compartir segmentos de código entre tareas se puede conseguir mediante la GTD o mediante un segmento compartido en la LTDs. Cada tabla almacena hasta 2^{13} descriptores de 64 bits $\Rightarrow$ ocupa 64 Kbytes y el espacio de direcciones virtuales contiene hasta 2^{14} segmentos. El selector de segmento es un índice dentro de una u otra tabla. Otros registros relacionados con la gestión de segmentos (en lo que respecta a la gestión de GDT y LDT) e interrupciones son: GDTR (contiene la dirección de la GTD en memoria principal, dirección lineal de 32 bits y 16 bits de límite), IDTR (dirección de la tabla de descriptores de interrupción (IDT), dirección lineal de 32 bits y 16 bits de límite), LDTR (contiene la dirección donde se encuentra la LDT en memoria principal del proceso actual) y TR (selector de segmento de estado de tarea (TSS) del proceso actual).

Tipos de descriptores de segmento: datos, código y sistema, que tienen en común (base, límite y atributos). La estructura de un descriptor de segmento consiste en los siguientes campos: (1) un campo base (base) de 32 bits que contiene la dirección lineal del primer byte de un segmento; (2) un bit G de granuralidad en bytes (G = 0) o páginas (G = 1) (múltiplos de 4096 bytes); (3) un campo límite (limit) de 20 bits que denota la longitud del segmento en bytes (segmentos con un campo límite igual a cero son considerados nulos; además, si G = 0 el tamaño de un segmento no nulo puede variar entre 1 byte y 1 Mbyte; mientras que en el caso de que G = 1, la longitud del segmento puede variar entre 4 Kbytes y 4 Gbytes); (4) un bit S de sistema (si S = 0 el segmento es un segmento del sistema que almacena estructuras de datos del kernel, en caso contrario (S = 1) es un segmento normal de datos o código; (5) un campo de 4 bits para el tipo (type) donde se caracteriza al tipo de segmento y sus derechos de acceso. La siguiente lista muestra los tipos de descriptores de segmentos más ampliamente utilizados: (5.1) Descriptor de segmento de código (CSD, code segment descriptor) que indica que el descriptor de segmento se refiera a un segmento de código y puede ser incluido en la GDT o en la LDT (el descriptor tiene S = 1); (5.2) descriptor de segmento de datos (DSD, data segment descriptor) indica que el descriptor de segmento se refiere a un segmento de datos y puede ser incluido en la GDT o LDT (S = 1); (5.3) descriptor de segmento de estado de tarea (TSSD, task state segment descriptor) indica que el descriptor de segmento se refiere a un TSS (segmento de estado de tarea), es decir un segmento utilizado para guardar el contenido de los registros del procesador, sólo puede aparecer en la GDT, y el valor correspondiente para el campo tipo (type) puede ser 11 o 9, dependiendo si el proceso correspondiente se está ejecutando actualmente en CPU (S = 0); (5.4) Descriptor de la tabla de descriptores global (LDTD, local descriptor table descriptor) indica que el descriptor de segmento se refiere a un segmento que contiene una LDT, éste puede aparecer sólo en la GDT, el campo tipo correspondiente tiene el valor 2 y S = 0. (6) un campo de 2 bits DPL (descriptor privilege level) para el privilegio asociado al segmento o lo que es lo mismo restringir su acceso, éste representa el nivel de privilegio de CPU mínimo requerido para acceder al segmento, por tanto, un segmento con DPL = 0 es accesible sólo cuando CPL = 0 (modo kernel), mientras que un segmento con DPL = 3 es accesible para cualquier valor de CPL. (7) un bit P para segmento presente (segment-present) que es igual a 0 cuando el segmento no está actualmente almacenado en memoria principal, y Linux establece este campo a P = 1, ya que este nunca intercambia el segmento completo a disco. (8) un bit D/B que depende si el segmento contiene código o datos (es establecido a 1 si las direcciones utilizadas como desplazamiento del segmento son de 32 bits), es decir D/B = 1 para i386 y 0 para i286. (9) un bit reservado que está establecido siempre a 0. (10) un bit AVL (Available) disponible para el software y que puede ser utilizado por el sistema operativo aunque Linux lo ignora.

Para acelerar la traducción de dirección lógica a dirección lineal (unidad de segmentación), el los procesadores i386 proporcionan un registro adicional no programable por cada uno de los registros de segmentación programables. Cada registro no programable contiene el descriptor de segmento de 8 bytes especificado por el selector de segmento contenido en el correspondiente registro de segmentación. Cada vez que un selector de segmento es cargado en un registro de segmentación, el descriptor de segmento correspondiente es cargado de memoria en el registro de CPU no programable. Entonces, traducciones de direcciones lógicas referentes a ese segmento pueden realizarse sin acceder a la GDT o LDT almacenada en memoria (el procesador sólo tiene que referenciar directamente a los registros de la CPU que contienen el descriptor de segmento. Los accesos a la GDT o LDT son necesarios sólo cuando los contenidos del registro de segmentación cambia. Cada selector de segmento incluye los siguientes campos: (1) Un campo índice de 13 bits que identifica la entrada del descriptor de segmento correspondiente contenida en la GDT o LDT. (2) un bit TI (table indicator) que especifica si el descriptor de segmento se encuentra en la GDT (TI = 0) o ne la LDT (TI = 1). (3) un campo RPL (requestor privilege level) de 2 bits, que es precisamente el CPL de la CPU cuando el correspondiente selector de segmento se carga en CR.

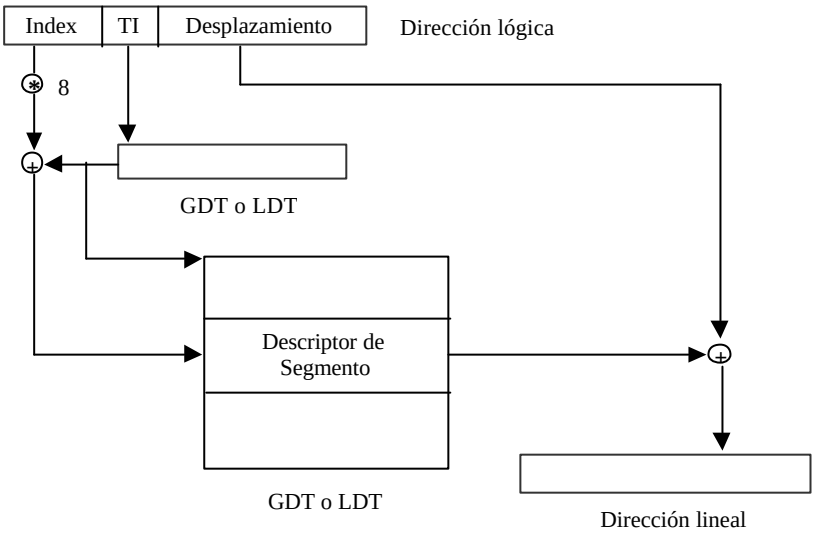

Figura 3.9. Traducción de una dirección lógica a una dirección lineal (virtual)

3.4.2.3. Segmentación en Linux

Linux no aprovecha la segmentación del i386. Sin embargo, no puede desactivarla, la utiliza de una forma muy limitada. Se ha preferido la paginación sobre la segmentación porque: (1) La gestión de memoria es más simple cuando todos los procesos usan los mismos valores de segmentos, es decir, tienen las mismas direcciones lineales. (2) Un objetivo de Linux es la portabilidad, y muchos procesadores soportan la segmentación de forma muy limitada.

Para hacer uso de la segmentación, se programa adecuadamente la GDT = Tabla de Descriptores Global. Esta tabla es implementada por el array gdt_table referenciada por la variable gdt, que se encuentra definido en el archivo arch/i386/kernel/head.S.

Los segmentos que se definen se superponen en el espacio de direcciones lineal. Como se emplean muy pocos segmentos, solo es necesaria la GDT. La LDT no se usa por el kernel salvo que lo requiera un proceso, aunque existe una llamada al sistema que permite crear sus propias LDTs.

Los segmentos empleados en Linux son los siguientes:

- Segmento de código del kernel.
- Base: 0x00000000, Límite (limit): 0xfffff,
- G = 1 (granularidad en páginas),
- S = 1 (Segmento normal de código o datos),
- Type = 0xa (Código, puede ser leído y ejecutado),

- DPL = 0 (Modo kernel para el nivel de privilegio del descriptor),
- D/B = 1 (Offset de 32 bits),
- Abarca desde 0 a $2^{32} - 1$, y el selector de segmentose define por la macro __KERNEL_CS. Para direccionar el segmento, el kernel se tiene que cargar el valor asociado a esta macro en el registro CS.

Segmento de datos del kernel:

- Base: 0x00000000, Límite (limit): 0xfffff,
- G = 1 (granularidad en páginas),
- S = 1 (Segmento normal de código o datos),
- Type = 0x2 (Datos, puede ser leído y escrito),
- DPL = 0 (Modo kernel),
- D/B = 1 (Offset de 32 bits),
- Idéntico al descriptor de segmento anterior salvo por el tipo, y se define por la macro __KERNEL_DS.

Segmento de código de usuario:

- Base: 0x00000000, Límite: 0xfffff
- G = 1 (granularidad en páginas)
- S = 1 (Segmento normal de código o datos)
- Type = 0xa (Código, puede ser leído y ejecutado)
- DPL = 3 (Modo usuario)
- D/B = 1 (Offset de 32 bits)
- El descriptor de segmento se define por la macro __USER_CS.

Segmento de datos de usuario:

- Base: 0x00000000, Límite: 0xfffff,
- G = 1 (granularidad en páginas),
- S = 1 (Segmento normal de código o datos),
- Type = 0x2 (Datos, puede ser leído y escrito),
- DPL = 3 (Modo usuario),
- D/B = 1 (Offset de 32 bits),
- El descriptor de segmento se define por la macro __USER_DS.

Un Segmento de Estado de Tarea (TSS) para cada proceso. El campo base del descriptor de TSS contiene la dirección del campo tss del descriptor de proceso correspondiente.

- G = 0 (Granularidad de bytes)
- Limite = 0xeb (el TSS es de 236 bytes)
- Type = 9 o 11 (TSS disponible)
- DPL = 0 (Los procesos no deben acceder a un TSS en modo usuario)

Un segmento de LDT por defecto para cada proceso, que apunta a una LDT compartida por todos los procesos con contenido nulo. El descriptor segmento se almacena en la variable default_ldt. Cada procesador tiene su propio descriptor de segmento LDT, su Base se inicializa al la dirección almacenada en default_ldt y el campo límite se establece a 7.

4 segmentos más para funciones de APM (Advanced Power Management, que consiste en un conjunto de rutinas BIOS dedicada a la gestión de los estados de la potencia del sistema) y 4 se dejan sin usar. Como la GDT tiene 8192 entradas, el máximo número de procesos es 4090.

3.4.2.4. Paginación en el i386

La unidad de paginación de la MMU traduce la dirección lineal a física: (1) Se chequea el tipo de acceso respecto a los derechos de acceso de la dirección lineal. (2) Si el acceso no es válido, se genera una excepción de falta de página.

Página: espacio contiguo de direcciones lineales de tamaño fijo 4Kb (se divide la memoria en 2^{20} páginas de 2^{12} bytes) y a los datos contenidos en este grupo de direcciones. Cada página está caracterizada por: dirección base (dirección lineal de inicio de página) y atributos. De los 32 bits de una dirección lineal, los 12 bits menos significativos de las direcciones lineal y física son iguales, mientras que los 20 bits más significativos se traducen de la dirección lineal a la física: (1) La página no está presente *(swapping)*. (2) La página está sin asignar. (3) El acceso es incorrecto *(copy-on-demand)*. Se agrupan las direcciones lineales contiguas en *páginas*, que se corresponden con direcciones físicas contiguas. Los chequeos se hacen a nivel de página. La memoria física se divide en *marcos de página* (página física), que contendrán páginas (no siempre las mismas) Es importante distinguir entre páginas y marcos de página. Una página es tan sólo un bloque de datos que ser almacenado en cualquier marco de página o en disco.

Para traducir la dirección lineal (32 bits) a física (32 bits) se emplean *tablas de páginas* (estructuras de datos). Se necesita una tabla con 2^{20} entradas de 20 bits más los atributos que son 12 bits (en total 4 bytes = 32 bits) ⇒ La tabla de páginas (traducción) ocuparía 4 Mbytes contiguos. Se emplean dos niveles de tablas por motivos de eficiencia (directorio de tablas de páginas (2^{10} entradas) y tablas de páginas (2^{10} entradas cada una)). Tanto el directorio como las tablas de páginas caben en una página (2^{10} * 4 = 4Kbytes).

Los 10 bits más significativos de la dirección lineal indexan en el *directorio de tablas de páginas*, de donde se obtiene la tabla de paginas. Los 10 bits siguientes se emplean para indexar en la *tabla de páginas* y obtener el *marco de página*. Los 12 últimos bits son de desplazamiento (offset) dentro de la página. Por tanto, las páginas son pues de 4Kb (32 bits).

Es posible compartir páginas entre varios procesos ⇒ tablas con entradas comunes o tablas de páginas globales. Además, existen buffers de traducción (TLB) ⇒ Buffer caché de entradas de la tabla de páginas con información sobre las últimas páginas accedidas. El directorio y las tablas de páginas contienen 1024 (2^{10}) entradas de 4 bytes (32 bits). La estructura de dichas entradas es (empezando por el bit menos significativo): P (bit presente), R/W (bit para lectura o lectura/escritura), U/S (bit para indicar el nivel de privilegio, user o supervisor), 0, 0, A (bit para indicar accedido o no accedido), D (dirty bit, modificado), 0, 0, AVL (3 bits para indicar la disponibilidad), los restantes 20 bits para indicar la dirección del marco de página. La dirección del directorio de tablas de páginas se almacena en el registro CR3 del procesador (almacena la dirección física que contiene el primer nivel de la tabla de páginas (directorio de tablas de páginas), según el formato anterior.

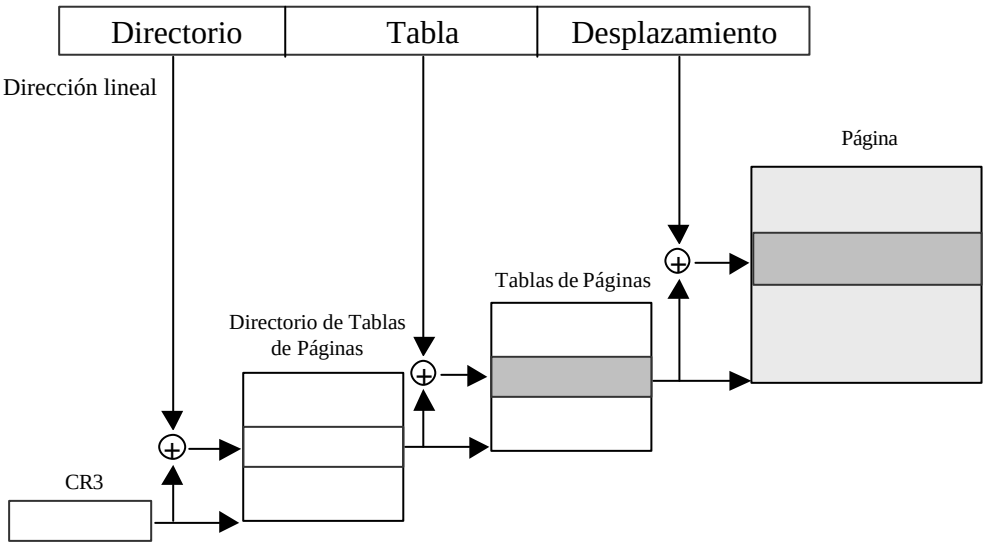

Figura 3.10. Paginación en los procesadores de la arquitectura i836

3.4.2.5. Paginación en Linux

Para tener en cuenta arquitecturas de 64 bits, la paginación en Linux tiene 3 niveles de tablas de páginas en lugar de 2. En los i386, la tabla de *directorio intermedio de tablas de páginas* se obvia.

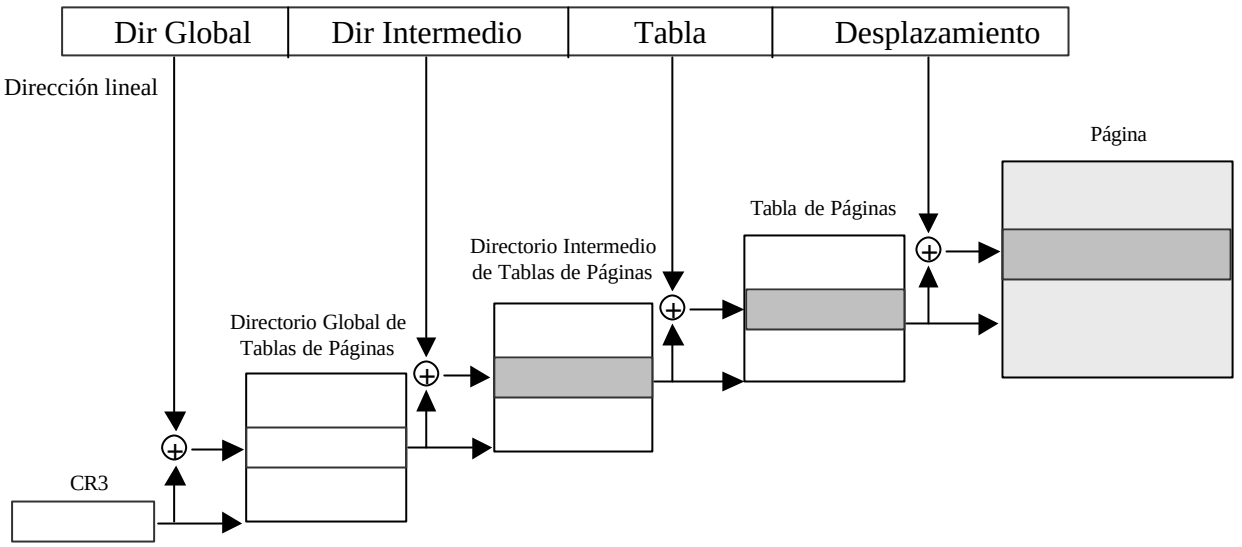

Figura 3.11. Modelo de Paginación de Linux.

Linux gestiona la memoria central y las tablas de páginas utilizadas para convertir las direcciones lineales (virtuales) en direcciones físicas. Implementa una gestión de la memoria que es ampliamente independiente del procesador sobre el que se ejecuta. En realidad, la gestión de la memoria implementada por Linux considera que dispone de una tabla de páginas a tres niveles: (1) directorio global de tablas de páginas *(page global direrory)* cuyas entradas contienen las direcciones de páginas que contienen tablas intermedias; (2) directorio intermedio de tablas de páginas *(page middle directory)* cuyas entradas contienen las direcciones de páginas que contienen tablas de páginas; (3) las tablas de páginas *(page table)* cuyas entradas contienen las direcciones de páginas de memoria que contienen el código o los datos utilizados por el kernel o los procesos de usuario. Evidentemente, este modelo no siempre corresponde al procesador sobre el cual Linux se ejecuta (los procesadores i386, por ejemplo, utilizan una tabla de páginas que sólo posee dos niveles). El kernel efectúa una correspondencia entre el modelo implementado por el procesador y el modelo de Linux. En los procesadores i386, por ejemplo, el kernel considera que el directorio intermedio de tablas de páginas sólo contiene una entrada.

La MMU de reinicializa con un nuevo directorio de tablas de páginas para cada proceso. Así se consiguen aislar espacios de direcciones. Se emplean los bits User/Supervisor y Read/Write para la protección. Se emplean los anteriores bits junto con los bits Dirty y Accessed para manejar las diferentes posibilidades de falta de página. Linux depende fuertemente de la paginación para la gestión de procesos. De hecho, la traducción automática de dirección lineal a dirección física hace posible los siguientes objetivos: (1) Asignar un espacio de direcciones físicas diferentes a cada proceso, asegurando una protección eficiente ante errores de direccionamiento. (2) Distinguir páginas (grupos de datos) de marcos de páginas (direcciones físicas en memoria principal). Esto permite a la misma página que se almacene en un marco de página, entonces guardar en disco y después recargarla en un marco de página diferente. Esto es muy importante para el mecanismo de memoria virtual.

En Linux el espacio de direcciones lineales se divide en dos partes: (1) Direcciones lineales desde 0x0000000 a PAGE_OFFSET – l, accesibles en cualquier modo (usuario o kernel). (2) Direcciones lineales desde PAGE_OFFSET hasta 0xffffffff, sólo en modo kernel. Generalmente PAGE_OFFSET se inicializa a 0xc0000000 (3Gb). Las primeras 768 entradas del directorio de tablas de páginas (que mapean los primeros 3Gb) dependen de cada proceso. El resto de entradas son las mismas para todos los procesos. El mapeado final proyecta las direcciones lineales empezando en PAGE_OFFSET en direcciones físicas empezando en 0.

Toda la memoria física debe ser accesible por el kernel. Éste mantiene un conjunto de tablas de páginas para su propio uso, cuya raíz es el master kernel Page Global Directory. Después de la inicialización del sistema este conjunto de tablas de páginas no son utilizadas nunca por ningún proceso o thread del kernel.

3.4.2.6. Gestión de las Tablas de Páginas (Directorios y Tablas de Páginas)

El archivo fuente <mm/memory.c> contiene la gestión de las tablas de páginas. Utiliza el modelo de Paginación de Linux. Las funciones dependientes de la arquitectura se definen en <asm/pgtable.h>:

- *pt_none*: devuelve el valor 1 si la entrada especificada de la tabla de páginas no está inicializada;
- *pte_present*: devuelve el valor l si la página especificada está presente en memoria;
- *pte_clear*: inicializa a 0 la entrada especificada de la tabla de páginas;
- *pmd_none*: devuelve el valor 1 si la entrada especificada del directorio intermedio de tablas de páginas no se inicializa;
- *pmd_bad*: devuelve el valor l si la entrada especificada del directorio intermedio de tablas de páginas es errónea;
- *pmd_present*: devuelve el valor l si la página que contiene el directorio intermedio de tablas de páginas está presente en memoria;
- *pmd_clear*: inicializa a 0 la entrada especificada del directorio intermedio de tablas de páginas;
- *pgd_none*: devuelve el valor l si la entrada especificada del directorio global de tablas de páginas no se inicializa;
- *pgd_bad*: devuelve el valor l si la entrada especificada del directorio global de tablas de páginas es errónea;
- *pgd_present*: devuelve el valor l si la página que contiene el directorio global de tablas de páginas presente en memoria;
- *pgd_clear*: inicializa a 0 la entrada especificada del directorio global de tablas de páginas;
- *pte_read*: devuelve l si la página especificada es accesible en lectura;
- *pte_write*: devuelve 1 si la página especificada es accesible en escritura;
- *pte_exec*: devuelve 1 si la página especificada es accesible en ejecución;
- *pte_dirty*: devuelve 1 si el contenido de la página especificada se ha modificado:
- *pte_young*: devuelve l si el contenido de la página especificada ha sido accedido;
- *pte_wrprotect*: hace la página especificada inaccesible en escritura;
- *pte_rdprotect*: hace la página especificada inaccesible en lectura;
- *pte_exprotect*: hace la página especificada inaccesible en ejecución;
- *pte_mkclean*: pone a 0 el indicador de modificación del contenido de la página especificada;
- *pte_mkold*: pone a 0 el indicador de acceso al contenido de la página especificada;
- *pte_mkwrite*: hace la página especificada accesible en escritura;
- *pte_mkread*: hace la página especificada accesible en lectura;
- *pte_mkexec*: hace la página especificada accesible en ejecución;
- *pte_mkdirty*: marca la página como modificada;
- *pte_mkyoung*: marca la página como accedida;
- *mk_ptte*: devuelve el contenido de una entrada de la tabla de páginas combinando la dirección de la página de memoria asociada y su protección;
- *pte_modify*: modifica el contenido de una entrada de la tabla de páginas;
- *pte_page*: devuelve la dirección de la página de memoria contenida en una entrada de la tabla de páginas;
- *pmd_page*: devuelve la dirección de la página de memoria que contiene una tabla intermedia;
- *pgd_offset*: devuelve la dirección de una entrada de la tabla global;
- *pmd_offset*: devuelve la dirección de una entrada de la tabla intermedia;
- *pte_offset*: devuelve la dirección de una entrada de la tabla de páginas;
- *pte_free_kernel*: libera una tabla de páginas utilizada por el kernel;
- *pte_alloc_kernel*: asigna una tabla de páginas utilizada por el kernel;
- *pmd_free_kernel*: libera una tabla intermedia utilizada por el kernel;
- *pmd_alloc_kernel*: asigna una tabla intermedia utilizada por el kernel;
- *pte_free*: libera una tabla de páginas utilizada por un proceso;
- *pte_alloc*: asigna una tabla de páginas utilizada por un proceso;
- *pmd_free*: libera un directorio intermedio de tablas de páginas utilizado por un proceso;

- *pmd_alloc*: asigna un del directorio intermedio de tablas de páginas utilizado por un proceso;
- *pgd_free*: libera el del directorio global de tablas de páginas utilizado por un proceso;
- *pgd_alloc*: asigna el del directorio global de tablas de páginas utilizado por un proceso.

Las funciones independientes de la arquitectura son las siguientes:

- *copy_page*: copia el contenido de una página de memoria en otra;
- *com*: muestra un mensaje de error, y envía la señal SIGKILL al proceso que haya sobrepasado la memoria disponible;
- *free_one_pmd*: libera la tabla de páginas apuntada por una entrada del directorio intermedio de tablas de páginas;
- *free_one_pgd*: libera el directorio intermedio de tablas de páginas apuntado por una entrada del directorio global de tablas de páginas, llamando a free_one_pmd;
- *clear_page_tables*: libera las tablas asociadas al espacio de direccionamiento de un proceso usuario, llamando a free_one_pmd para cada directorio intermedio de tablas de páginas;
- *free_page_tables*: libera las tablas asociadas al espacio de direccionamiento de un proceso usuario, llamando a free_one_pmd para cada directorio intermedio de tablas de páginas, y seguidamente a pgd_free;
- *new_page_tables*: asigna nuevas tablas para un proceso;
- *copy_one_pte*: copia el contenido de una entrada de la tabla de páginas en otra, borrando la autorización de escritura si la copia en escritura se especifica;
- *copy_pte_range*: copia el contenido de una serie de entradas de la tabla de páginas, llamando a copy_one_pte para cada entrada;
- *copy_pmd_range*: copia el contenido de una serie de entradas del directorio intermedio de tablas de páginas, llamando a copy_pte_range para cada entrada del directorio intermedio de tablas de páginas;
- *copy_page_range*: copia el contenido de una región de memoria, llamando a la función copy_pmd_range para cada directorio intermedio de tablas de páginas afectado;
- *free_pte*: libera una página apuntada por una entrada de la tabla de páginas;
- *forget_pte*: libera una página apuntada por una entrada de la tabla de páginas, si esta entrada no es nula;
- *zap_pte_range*: libera varias páginas llamando a free_pte;
- *zap_pmd_range*: libera una serie de entradas del directorio intermedio de tablas de páginas, llamando a la función zap_pte_range para cada entrada del directorio intermedio de tablas de páginas;
- *zap_page_range*: libera el contenido de una región de memoria, llamando a la función zap_pmd_range para cada directorio intermedio de tablas de páginas afectado;
- *zeromap_pte_range*: inicializa varias entradas de la tabla de páginas con el mismo descriptor;
- *zeromap_pmd_range*: asigna tablas de páginas y las inicializa, llamando a la función zeromap_pte_range;
- *zeromap_page_range*: asigna varios directorios intermedios de tablas de páginas, y llama a la función zeromap_pmd_range para cada directorio, para inicializar las entradas de las tablas de páginas;
- *remap_pte_range*: modifica las direcciones de páginas contenidas en varias entradas de la tabla de páginas;
- *remap_pmd_range*: asigna tablas de páginas y modifica las direcciones contenidas, llamando a relmap_pte_range;
- *remap_page_range*: asigna varios directorios intermedios de tablas de páginas y modifica las entradas de tablas, llamando a la función remap_pmd_range.

3.4.3. Gestión de Memoria en Linux.

Hemos visto que Linux hace uso de las ventajas de la segmentación y de los circuitos de paginación de los procesadores i386 para traducir direcciones lógicas en direcciones físicas. Podemos también decir que alguna porción de RAM está permanentemente asignada al kernel y utilizada para almacenar el código del kernel y estructuras de datos estáticas del mismo. La restante parte de la RAM se denomina memoria dinámica, y ésta es un recurso muy valioso y necesitado no sólo por los procesos sino también por el propio kernel. De

hecho el rendimiento global del sistema depende fuertemente de cómo de eficientemente se gestiona la memoria dinámica. Por tanto, todos los sistemas operativos multitarea actuales tratan de optimizar el uso de la memoria dinámica, asignándola sólo cuando es estrictamente necesario y liberándola tan pronto como sea posible. En esta sección describiremos: estructuras básicas del kernel para gestionar la memoria dinámica desde dos puntos de vista: asignación de memoria para el kernel y asignación de memoria para procesos, política de asignación de la memoria por parte del kernel, el gestor de faltas de página, etc.

3.4.3.1. Gestión de Marcos de Página.

El kernel debe mantener el estado actual de cada marco de página ⇒ debe mantener información adicional a la que puede guardarse en el descriptor de las tablas de página. Se almacena información de cada página en un array (mem_map) de descriptores de marco de página, definidos por la estructura struct page en <linux/mm.h>

```
typedef struct page {
        struct list_head list;
        struct address_space *mapping;
        unsigned long index;
        struct page *next;
        struct page *prev;
        struct inode *inode;
        unsigned long offset;
        struct page *next_hash;
        atomic_t count;
        unsigned long flags;
        struct wait_queue *wait;
        struct page **pprev_hash:
        struct buffer_head *buffers;
        strcut list_head lru;
} mem_map_t;
```

Algunos de los campos más representativos son:

- *count*: Es un contador de referencia de uso para la página y vale 0 si el marco de página está libre, o el número de procesos a los que ha sido asignada.
- *prev*, *next*: Para insertar el descriptor en una lista circular doblemente enlazada. El significado depende del uso de la página en cada momento
- *flags*: Array de hasta 32 fIags de la forma PG_xyz que indican el estado del marco de página

Algunos de los flags más importantes son:

- PG_DMA: Se puede usar la página para DMA ISA
- PG_locked: La página no acepta swap
- PG_referenced: La página ha sido accedida recientemente en la caché de páginas
- PG_reserved: Reservada para el kernel, o no usable
- PG_slab: La página está en un slab
- PG_swap_cache: La página está en la caché de swap

En la siguiente figura se ilustra la memoria dinámica y los valores utilizados para referirse a ella.

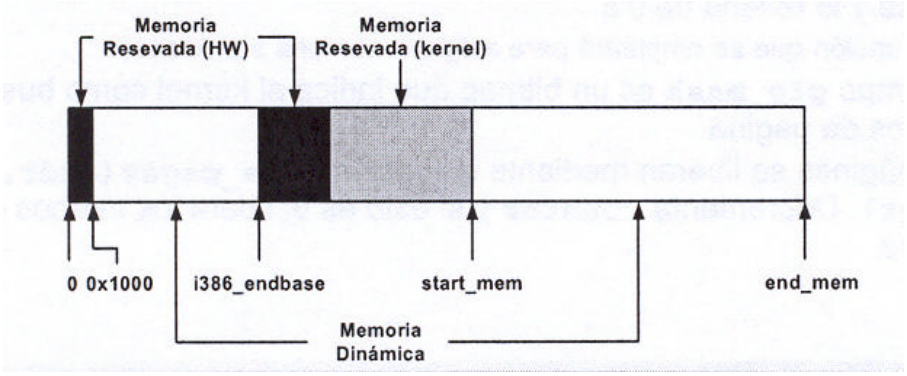

Figura 3.12. Perfil de memoria.

Los descriptores se inicializan en free_area_init(). En general, la función asigna un área de memoria para mem_map y modifica start_mem. Primero pone todos los campos a 0 y a continuación establece los flags PG_DMA y PG_reserved. A continuación la función mem_init() realiza la inicialización de los descriptores: (1) pone a 0 el flag PG_reserved de los marcos de página no reservados; (2) pone a 0 también el flag PG_DMA de las páginas con dirección física mayor que 0x1000000 (16 MBytes); y (3) pone a 1 el campo count de cada marco de página asociado a memoria dinámica y llama a la función free_page(). Esto añade la página al *Buddy system*, incrementando el número de páginas libres totales ⇒ al final del bucle, nr_free_pages contiene el número total de páginas libres en memoria dinámica.

3.4.3.2. Asignación y Liberación de Marcos de Página

Los marcos de página se asignan en grupos contiguos potencia de 2 ⇒ alloc_pages(gfp_mask, order). Es consecuencia del *Buddy system* subyacente. La función de asignación más general es __get_free_pages(gfp_mask, order). La función get_free_page(gfp_mask) obtiene un único marco de página y lo rellena de 0's ⇒ es la función que se empleará para asignar memoria a procesos. El campo gfp_mask es un bitmap que indica al kernel cómo buscar los marcos de página. Los marcos de página se liberan mediante la función free_pages(addr, order). Decrementa counter y si este es 0, libera los marcos de página.

Flags del campo gfp_mask utilizado para la búsqueda de marcos de página libres:

- __GFP_WAIT: El kernel puede descartar contenido de páginas para liberar memoria
- __GFP_IO: El kernel puede guardar páginas en disco
- __GFP_DMA: Las páginas deben soportar DMA
- __GFP_HIGH, __GFP_MED, __GFP_LOW: Prioridad

En realidad en el kernel se usan valores predeterminados de combinaciones de los anteriores flags:

	__GFP_WAIT	**__GFP_IO**	**Prioridad**
GFP_ATOMIC	0	0	__GFP_HIGH
GFP_BUFFER	1	0	__GFP_LOW
GFP_KERNEL	1	1	__GFP_MED
GFP_NFS	1	1	__GFP_HIGH
GFP_USER	1	1	__GFP_LOW

3.4.3.3. Políticas de Asignación de Memoria

Se puede observar la asignación de memoria desde dos puntos de vista: peticiones por parte del kernel y por parte de procesos de usuario.

El kernel: (1) Es un componente de alta prioridad, si solicita memoria no tiene sentido retardar su asignación; (2) Confía en si mismo y se asume que no tiene errores de programación. Un proceso de usuario: (1) No tiene por qué usar el espacio solicitado inmediatamente, y por el principio de localidad seguramente no lo hará,

además, se puede retardar la asignación de memoria real; (2) No es confiable, y el kernel debe estar listo para capturar todos los posibles errores de programación.

El kernel puede solicitar memoria de tres formas: (1) directamente al *Buddy system*, para asignaciones genéricas de grupos de marcos de página potencia de 2; (2) al *Slab allocator*, para objetos frecuentemente usados; y (2) utilizando vmalloc() para obtener un área de memoria empleando marcos de página no contiguos. Cuando los procesos solicitan memoria, no se les asigna realmente páginas, sino áreas de memoria. Se les da rangos de direcciones lineales válidos que se asignarán en el momento en que se vayan a usar.

3.4.3.4. El Buddy System (sistema de colegas)

El kernel debe establecer una estrategia robusta y eficiente para asignar grupos de marcos de páginas contiguos. Por ello, el objetivo principal del Buddy system es: *Evitar la fragmentación externa*. Éste es un fenómeno que se produce cuando existen frecuentes asignaciones y liberaciones de grupos de marcos de página contiguos de diferentes tamaños, pudiendo derivar en una situación en la que varios bloques pequeños de marcos de páginas libres están dispersos dentro de bloques de marcos de páginas asignados. Y como resultado puede convertirse en imposible el asignar un bloque grande de marcos de páginas contiguos, incluso si existen suficientes páginas libres para satisfacer la petición.

Para evitar la fragmentación externa existen dos posibilidades: (1) Utilizar la unidad de paginación para agrupar marcos de página dispersos en direcciones lineales contiguas. (2) Desarrollar un sistema que controle los marcos de página contiguos y evite en lo posible dividir un bloque libre grande para una asignación pequeña.

El kernel opta por la segunda opción por las siguientes razones: (1) En ocasiones es necesario que los marcos de página sean contiguos, ya que direcciones lineales contiguas no son suficientes para satisfacer la petición (por ejemplo, un buffer asignado a DMA, DMA ignora toda el sistema de paginación hardware y accede al bus de direcciones directamente mientras transfiere varios sectores de disco en una única operación de E/S y los buffers requeridos deben estar en marcos de página contiguos). (2) No se modifican las tablas de páginas del kernel, por lo que la CPU no tiene que limpiar el contenido de los TLBs. (3) Porciones grandes de memoria física contigua pueden ser accedidas por el kernel median páginas de 4 KBytes.

El kernel mantiene una lista de marcos de página disponibles en memoria utilizando el principio del *Buddy system* que es bastante simple: el kernel mantiene una lista de grupos de marcos de página, siendo estos grupos son de tamaño fijo (pueden contener 1, 2,4, 8, 16, 32, 64, 128, 256 o 512 marcos de página) y se refieren a marcos de página contiguas en memoria. El principio básico del *Buddy System* es el siguiente: a cada petición de asignación, se usa la lista no vacía que contiene los grupos de marcos de página de tamaño inmediatamente superior al tamaño especificado, y se selecciona un grupo de páginas de esta lista. Este grupo se descompone en dos partes: los marcos de página correspondientes al tamaño de memoria especificado, y el resto de marcos de página que siguen disponibles. Este resto puede insertarse en las otras listas. Al liberar un grupo de marcos de página, el kernel intenta fusionar este grupo con los grupos disponibles, con el objetivo de obtener un grupo disponible de tamaño máximo. Para simplificar la asignación y liberación de un grupo de marcos de página, el kernel sólo permite asignar grupos de páginas cuyo tamaño sea predeterminado y corresponda a los tamaños gestionados en las listas.

Si suponemos que se deben asignar 8 marcos de página y sólo está disponible un grupo de 32 marcos de página, el kernel utiliza dicho grupo, asigna los 8 marcos de páginas solicitados, y reparte los 24 marcos de página restantes en un grupo de 16 y otro de 8 marcos de página. Estos dos grupos se insertan en las listas de grupos correspondientes. Al liberar estos 8 marcos de página, el kernel comprueba si los 8 marcos de página adyacentes están disponibles, es decir, si un grupo de marcos de página disponibles contiene estas páginas. Si no es así, se crea un grupo disponible para los 8 marcos de página liberadas. Si existe un grupo adyacente, su tamaño se modifica para incluir los 8 marcos de página liberados, y el kernel comprueba si puede fusionado con otro grupo de 16 páginas, y así sucesivamente.

Funcionamiento básico (algoritmo del sistema de colegas, *Buddy system* algorithm): (1) Los marcos de página se agrupan en 10 listas de bloques que contienen grupos de 1, 2, 4, 8, 16, 32, 64, 128, 256 y 512 marcos de página contiguos. (2) La dirección física del primer marco de página de un grupo es un múltiplo del tamaño del grupo (por ejemplo, La dirección inicial de un grupo de 16 páginas es múltiplo de $16*2^{12}$, $2^{12} = 4096$, que es tamaño de pagina regular). (3) Si se solicita *n* marcos de página de memoria contigua, el algoritmo primero busca en la lista de bloques de tamaño de marcos de página *n*. Si hay algún bloque de tamaño *n* libre, se asigna. (4) Si la lista está vacía, se busca el siguiente bloque más grande (2*n) en la siguiente lista, y así sucesivamente hasta un bloque libre. (5) Si el bloque es más grande que el solicitado, se divide en partes y los bloques sobrantes se insertan en las listas de bloques adecuadas. Observar que SIEMPRE se puede hacer que los bloques resultantes cumplan el requisito de alineación. Por ejemplo, si existe una demanda en el sistema de un grupo de 128 marcos de página contiguos y no hay bloques de marcos de página libres de 128 ni de 256; sólo 512. Entonces ocupa los 128 de los 512 para satisfacer la petición y los restantes 384 los fragmenta el kernel y los añade en las listas de 256 y 128 respectivamente (384 = 256 + 128). (6) Si no se puede obtener el bloque del tamaño solicitado se devuelve error. (7) En la liberación (que da nombre al algoritmo), el kernel intenta fusionar pares de bloques 'colegas' (buddies) de tamaño *b* en un único bloque de tamaño 2*b de forma iterativa. Dos bloques se consideran 'colegas' si: (7.1) ambos bloques son del mismo tamaño (b); (7.2) están contiguos en memoria física; (7.3) La dirección física del primer marco de página del primer bloque es múltiplo de 2*<tamaño del bloque = b>*2^{12}.

Ahora vamos a ver las estructuras de datos necesarias para soportar el *Buddy system.* Linux emplea dos *Buddy systems* diferentes; para marcos de página válidos para DMA y para los que no (gestionar marcos de página normales). Trataremos el tema sin distinción. Estructuras de datos: (1) El array mem_map descrito anteriormente; (2) Un array free_area de 10 elementos (uno por cada tamaño de bloque)de tipo free_area_struct (o equivalentemente free_area_t); el k-ésimo elemento se relaciona con bloques de 2^k marcos de página. (3) 10 arrays binarios llamados bitmaps, uno por cada tamaño de bloque. Sirve para controlar los bloques que el *Buddy system* asigna al grupo; formados con arrays de enteros.

La estructura free_area_struct se emplea para gestionar cada grupo de bloques de un tamaño concreto

```
struct free_area_struct {
        struct page *next;
        struct page *prev;
        unsigned long map;
        unsigned long count;
};
```

prev y next se emplean para crear una lista circular doble de bloques del tamaño requerido, empleando los campos prev y next del descriptor de marco de página. count indica el número de elementos de la lista. map es un puntero al bitmap del grupo.

El bitmap tiene las siguientes características: (1) El tamaño de cada bitmap depende del número total de marcos de página y del tamaño del bloque. (2) Cada bit del bitmap de la k-ésima entrada describe el estado de dos bloques 'colegas' (buddies) de tamaño 2^k marcos de página: Si es 0 indica que ambos están ocupados o ambos libres, Si es 1 indica que solo uno de los bloques está libre. Cuando ambos 'colegas' están libres, el kernel les trata como un único bloque libre de 2k+1. Por ejemplo, si el sistema tiene 128 MBytes de RAM, se puede dividir en 32768 páginas, o 16384 bloques de 2 marcos de página, u 8192 de 4, y así sucesivamente hasta 64 grupos de 512 marcos de página. El bitmap de páginas simples (free_area[0]) tendrá 16384 bits, el de bloques de 2 páginas (free_area[1]) tendrá 8192 bits, y así sucesivamente hasta el último bitmap (free_area[9]) que consta de 32 bits uno por cada par de bloques de 512 marcos de página contiguos.

Cuando se extrae un bloque del *Buddy system*: (1) Si el tamaño era el solicitado, se saca de la lista y se cambia el bit correspondiente del bitmap (se pondrá a 0). (2) Si el tamaño no era el solicitado, se extrae el bloque obtenido cambiando el bit correspondiente del bitmap y a continuación los bloques sobrantes se insertan en las listas correspondientes marcando sus bits en el bitmap (se pondrán a 1).

Cuando se devuelve un bloque al *Buddy system*: (1) Se cambia el bit correspondiente en la lista de su tamaño; si el resultado es 1 es que su colega (buddy) no estaba en la lista así que insertamos el bloque en la lista porque no se puede fusionar. (2) Si el resultado es 0, su colega (buddy) estaba en la lista, así que lo extraemos de la misma y ya tenemos un bloque de mayor tamaño. Repetimos la operación en la siguiente lista, fusionando bloques hasta que al cambiar el bit, el resultado sea 1.

3.4.3.5. El Slab Allocator

El *Buddy system* asigna como mínimo una página. Sistema adecuado para asignaciones grandes de memoria Para asignaciones de pocos bytes, se desperdician grandes cantidades de memoria. *Fragmentación interna*, que es provocada por una mala combinación entre el tamaño de la memoria requerida y el tamaño del área de memoria asignada para satisfacer la solicitud. Es decir, éste es un fenómeno en el que se malgasta el área de memoria de una página cuando el tamaño de memoria es más pequeño que la página.

Es necesario un sistema superpuesto al *Buddy system* para el mejor aprovechamiento de la memoria. Linux 2.0, Se definen 13 listas de áreas de memoria (secuencias de posiciones de memoria teniendo direcciones físicas contiguas y una longitud arbitraria) donde guardar objetos de tamaños geométricamente distribuidos, de 32 a 131056 bytes, es decir, el tamaño depende más de ser potencia de 2 que del tamaño de los datos a almacenar. Se emplea el *Buddy system* tanto para asignar nuevas páginas para estas áreas como para liberar las que no son necesarias. Dentro de cada marco de página se emplea una lista dinámica para gestionar los elementos libres. Por fragmentación interna se pierde aproximadamente un 50% de la memoria.

En Linux 2.2 se revisa el sistema y se aplica un algoritmo derivado del *Slab allocator* de Solaris 2.4, simplificado. Los objetivos que persigue son: (1) Acelerar las asignaciones/liberaciones de objetos y evitar fragmentación interna. (2) Se basa en que el kernel pide y libera frecuentemente áreas de memoria pequeñas y de estructura consistente, como descriptores de proceso, o de memoria, inodos, etc. (3) Se ven las áreas de memoria que pide el sistema como objetos que consisten en estructuras de datos y algunas funciones, como un constructor y un destructor (que no se emplean en Linux). (4) Se mantienen una serie de cachés que gestionan slabs de objetos del mismo tipo. Habrá tantos cachés como tipos de objeto específico interese gestionar, y una caché mantiene varios slabs. (5) Un slab es una serie de marcos de página contiguos en los que se almacenan los objetos del mismo tipo. (6) Cuando se pide memoria para un objeto (por ejemplo un descriptor de proceso), se obtiene de la caché correspondiente, ahorrando tiempo y evitando fragmentación interna. (7) Cuando se libera un objeto, no se desasigna realmente la memoria sino que se marca la entrada en el slab como disponible. (8) Si no hay espacio en ningún slab de la caché para un nuevo objeto, se crea un nuevo slab. (9) Existen funciones que permiten crear una caché para objetos de un tipo específico. (10) Los objetos que se usan poco como para justificar el uso de una caché, se guardan en objetos de tamaño geométricamente distribuido como en Linux 2.0, incluso corriendo el riesgo de fragmentación interna. (11) No usar tamaños geométricamente distribuidos mejora la eficiencia de la caché hardware.

El *Slab allocator* agrupo objetos en cachés. Cada caché es un "almacén" de objetos del mismo tipo. Los cachés creados del *Slab allocator* pueden verse mediante /proc/slabinfo.

[…]	**Activos**	**Total**	**Tamaño**	**Paginas**		**pps**
Filp	664	690	128	23	23	1
name_cache	0	4	4096	0	4	1
buffer_head	54004	222440	96	1355	5561	1
mm_struct	45	72	160	3	3	1
vs_area_struct	1643	2840	96	42	71	1
fs_cache	44	118	64	1	2	1
files_cache	44	63	416	6	7	1
signal_act	49	66	1312	17	22	1
size-131072 (DMA)	0	0	131072	0	0	32
size-131072	0	0	131072	0	0	32
size-65536 (DMA)	0	0	65536	0	0	16

size-65536	0	0	65536	0	0	16
[...]						
size-64 (DMA)	0	0	64	0	0	1
size-64	183	236	64	4	4	1
size-32 (DMA)	0	0	32	0	0	1
size-32	3104	3164	32	28	28	1

El área de memoria principal que contiene un caché se divide en slabs; cada slab consiste en uno o más marcos de página contiguos que contienen objetos libres y asignados como se muestra en la siguiente figura.

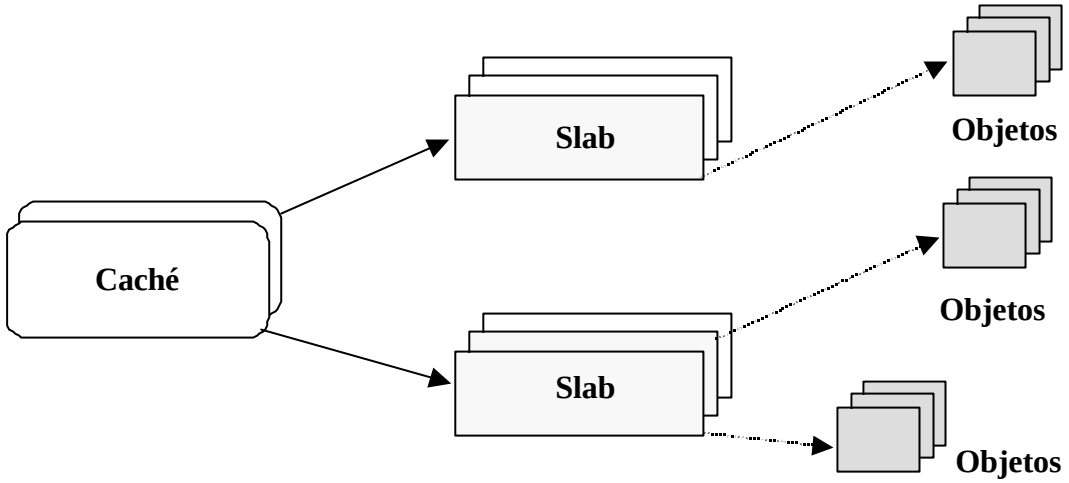

Figura 3.13. Componentes del *Slab allocator*

El *Slab allocator* no libera slabs vacíos por si mismo por motivos de eficiencia. El *Slab allocator* se sitúa también por encima del *Buddy system*, existiendo una interfaz entre ambos elementos del kernel (*Buddy system* y *Slab allocator*) para facilitar su interacción.

Estructura del descriptor de caché ⇒ cada caché se describe por una tabla de tipo struct kmem_cache_s en <mm/slab.c>. Además, en dicho archivo podemos encontrar las funciones que gestionan al relación entre el caché y los slabs, como por ejemplo, asignar un slab a un caché (kmem_cache_grow()), liberar un slab de un caché (kmem_cache_destroy(), kmem_free_pages()), etc. Algunos de los campos representativos del descriptor de caché son: name (array de caracteres que almacena el nombre del caché), slabs_full (lista circular doblemente enlazada de descriptores de slabs con objetos ocupados (no libres)), slabs_partial (lista circular doblemente enlazada de descriptores de slabs con objetos ocupados y libres), slabs_free (lista circular doblemente enlazada de descriptores de slabs con sólo objetos libres), num (número de objetos empaquetados en un slab), next (puntero a la lista doblemente enlazada de descriptores de caché), flags (conjunto de flags que describen las propiedades permanentes del caché), dflags (conjunto de flags que describen las propiedades dinámicas del caché), gfpflags (conjunto de flags pasados a la función del *Buddy system* cuando se asignan marcos de página), etc.

Por lo que respecta a la estructura de un descriptor de slab ⇒ cada slab de un caché tiene su propio descriptor de tipo struct slab_s en <mm/slab.c>. Los descriptores de slab se pueden clasificar en una de las siguientes categorías dependiendo en donde se almacenen: (1) descriptor de slab externo, almacenado fuera del slab, en uno de los cachés generados y no apropiado para DMA (apuntado por cache_sizes; (2) descriptor de slab interno, almacenado dentro del slab al principio del primer marco de página asignado al slab. El Slab allocator escoge la segunda alternativa cuando el tamaño de los objetos es menor de 512 o cuando la fragmentación interna deja suficiente espacio para el slab. Los campos mas representativos del descriptor de slab son: inuse (número de objetos en el slab que están actualmente asignados), s_mem (apunta al primer objeto dentro del slab), free (apunta al primer objeto libre del slab, list (apunta a una de las tres listas doblemente enlazadas de descriptores de slab (slabs_full, slabs_partial o slabs_free)).

El sistema agrupa los cachés formando una lista enlazada. Los slabs dentro de una caché se agrupan formando una lista circular doblemente enlazada. Se mantiene la lista parcialmente ordenada; primero los slabs llenos, luego los parcialmente llenos y luego los vacíos. La caché mantiene un puntero al primer slab con al menos un objeto libre. Los descriptores de slab se pueden almacenar dentro o fuera del propio slab. Para ilustrar este comportamiento podemos estudiar la siguiente figura,

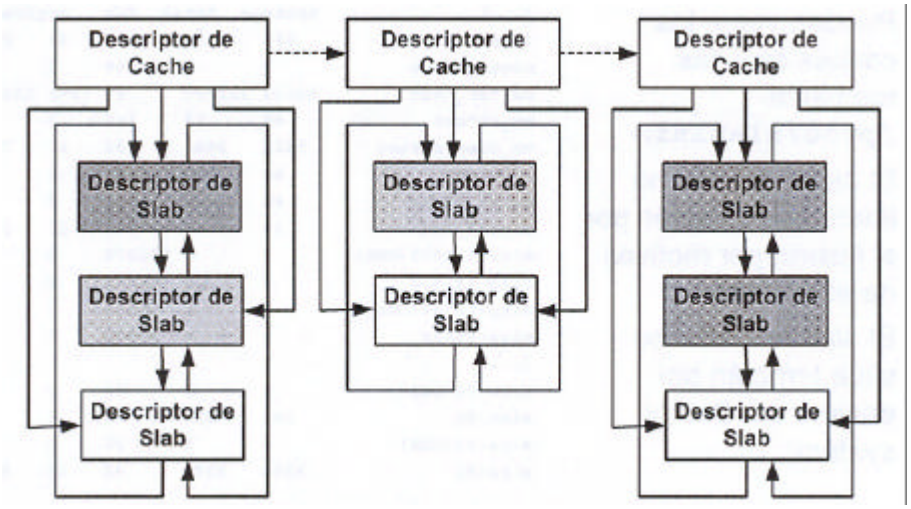

Figura 3.14. Relación entre los descriptores de caché y los descriptores de slabs.

Los cachés se dividen en dos categorías: (1) cachés generales, utilizados por el *Slab allocator* para propósitos propios); (2) cachés específicos, utilizados por lss restantes partes del kernel. Los cachés generales son: (1) un primer caché que contiene los descriptores de caché utilizados por el kernel (la variable cache_cache contiene su descriptor). (2) Trece cachés adicionales para almacenar áreas de tamaño geométricamente distribuido, de tamaños 32, 64, 128, 256, 512 hasta 131072 bytes. Por cada tamaño hay dos cachés, uno apropiado para asignaciones DMA y el otro para asignaciones normales (la tabla cache_sizes se utiliza eficientemente para derivar la dirección de caché correspondiente a un tamaño dado. Los cachés generales se inicializan con kmem_cache_init() y kmem_cache_sizes_init() durante la inicialización del sistema. Por otro lado, los cachés específicos se crean mediante kmem_cache_create(). Para destruir un caché se llama a la función kmem_cache_destroy() que se encuentra en el archivo fuente <mm/slab.c>.

Cada objeto tiene un descriptor de tipo kmem_bufctl_t en el archivo fuente <mm/slab.c>. Los descriptores de objetos se almacenan en un array ubicado después del correspondiente descriptor de slab. Al igual que los descriptores de slabs, los descriptores de objetos de un slab pueden almacenarse de dos formas: descriptores de objetos externos e internos. El primer descriptor de objeto en el array describe el primer objeto en el array y así sucesivamente. Un descriptor de objetos es simplemente en entero (unsigned int), que tiene significado cuando el objeto está libre. Éste contiene el índice al siguiente objeto libre en el slab, implementando así una liste de objetos libres en el slab. El descriptor del objeto del último elemento de la lista de objetos libres se marca como BUFCLT_END (0xffffffff). Los objetos se asignan y se liberan mediante las funciones kmem_cache_alloc() y kmem_cache_free(). Los objetos de propósito general se obtienen mediante las funciones kmalloc() y se liberan con kfree(), implementadas en <mm/slab.c>, permitiendo asignar y liberar áreas de memoria formadas por páginas contiguas en memoria central.

3.4.3.6. Gestión de Área de Memoria no Contigua

En ocasiones no es posible o no es ventajoso emplear marcos de página contiguos. En ese caso, se mapea un rango de direcciones lineal contiguo sobre marcos de página no contiguos. La principal ventaja de este esquema es que evita la fragmentación externa, mientras que el inconveniente es que es necesario modificar las tablas de páginas del kernel. Es práctico para asignaciones infrecuentes. El tamaño de la asignación lógicamente debe ser múltiplo de 4 KBytes.

Un pregunta que en este momento puede hacerse es la siguiente: ¿cuál es el rango de direcciones lineales para asignar en calidad de áreas de memoria no contigua?. Desde 0 a PAGE_OFFSET – 1 está reservado para procesos. Recordemos que el kernel inicialmente mapea la memoria física existente a partir de PAGE_OFFSET (comienzo del cuarto GigaByte). El final de la memoria física está en la variable high_memory. Desde high_memory hasta el final, está disponible. Se dejan 8 MBytes de margen a partir de high_memory, VMALLOC_START, y 4 KBytes entre áreas no contiguas asignadas. VMALLOC_END define la dirección final del espacio lineal reservado para áreas de memoria no contiguas.

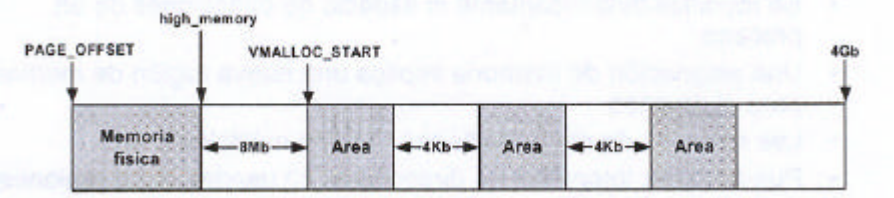

Figura 3.15. El intervalo de dirección lineal empezando desde PAGE_OFFSET

Descriptores de áreas de memoria no contiguas ⇒ cada área de memoria no contigua va asociada a un descriptor de tipo struct vm_struct que tiene la siguiente estructura (flags, dirección de comienzo, tamaño, puntero al siguiente descriptor):

```
struct vm_struct {
        unsigned long flags;
        void *addr;
        unsigned long size;
        struct vm_struct *next;
};
```

Se mantienen los descriptores de áreas de memoria no contiguas en una lista enlazada con direcciones en orden creciente para facilitar las búsquedas, y la búsqueda de regiones libres de memoria lineal. La función get_vm_area() crea nuevos descriptores del tipo vm_struct, llamando en su interior a la función kmalloc. Asignación y liberación ⇒ una vez se obtiene el rango de direcciones lineal, se asignan las páginas, y para cada una de ellas: (1) se han de modificar las tablas de páginas (incluidos PMD y PGD); (2) se piden las páginas al *Buddy system* con __get_free_paqe().

Para asignar área de memoria no contigua el kernel utiliza la función vmalloc que se encuentra declarada en el archivo cabecera <include/linux/vmalloc.h>. Esta función llama a get_vm_area() para crear un nuevo descriptor y devolver la dirección lineal asignada al área de memoria. También llama a vmalloc_area_pages() marcos de página no contiguos. Y termina devolviendo la dirección lineal inicial del área de memoria no contigua. Para liberar área de memoria no contigua el kernel utiliza la función vfree(). En primer lugar se llama a vmlist() para encontrar la dirección lineal del descriptor del área asociada al área a liberar. El área en sí es liberada llamando a vmfree_area_pages(), mientras que el descriptor es liberado llamando a kfree().

3.4.3.7. Memoria para Procesos

Como acabamos de ver, una de las funciones del kernel es obtener memoria dinámica de forma sencilla llamando a una variedad de funciones: __get_free_pages() o pages_alloc() para obtener páginas del algoritmo del Buddy system; kmem_cache_alloc() o kmalloc() para utilizar el Slab allocator para objetos de propósito general o específico, y vmalloc() para obtener un área de memoria no contigua. En estos casos, si la petición puede realizarse satisfactoriamente, cada una de estas funciones devuelve una dirección del descriptor de página o una dirección lineal identificando el principio del área de memoria dinámica asignada. Cuando se asigna memoria a procesos en modo usuario, la situación es diferente: (1) Los procesos que demandan memoria dinámica se consideran no urgentes. Cuando se carga el archivo ejecutable de un proceso, es poco probable que el proceso direccionará todas las páginas de código en un futuro próximo. Equivalentemente, cuando un proceso llama a la función malloc() para obtener memoria dinámica adicional, esto no quiere decir que el proceso accederá pronto a toda la memoria adicional obtenida. Por tanto, como

regla general, el kernel trata de aplazar la asignación de memoria dinámica a procesos en modo usuario. (2) Debido a que programas de usuario no pueden ser fiables, el kernel debe estar preparado para capturar todos los errores de direccionamiento provocados por un proceso en modo usuario.

El espacio de direcciones de un proceso son las direcciones lineales que el proceso puede utilizar. Cuando un proceso pide memoria dinámica, no se le dan marcos de página; se le da el derecho a usar un nuevo rango de direcciones lineales. La asignación de marcos de página se retarda todo lo posible. El kernel representa los intervalos de direcciones lineales mediante un recurso llamado regiones de memoria (caracterizadas por una dirección lineal inicial, una longitud y unos atributos): (1) Se modifica dinámicamente el espacio de direcciones de un proceso; (2) Una asignación de memoria implica una nueva región de memoria para el proceso. (3) Las regiones de memoria tienen tamaño múltiplo de 4 KBytes. (4) Puede haber intervalos de direcciones no usadas entre regiones.

Algunas situaciones en que un proceso obtiene nuevas regiones de memoria son las siguientes: (1) Cuando se crea el proceso (fork()), o se sustituye (execve()). (2) Cuando mapea un archivo en memoria (mmap(), munmap()). (3) Cuando la pila de usuario se desborda, el kernel puede expandirla. (4) Cuando el proceso crea un área de memoria compartida mediante IPC (shmat(), shmdt()). (5) Cuando el proceso desea expandir el heap mediante malloc(). Es esencial para el kernel identificar las regiones de memoria para que el gestor de falta de página sepa cómo actuar. Las excepciones de falta de página pueden deberse a errores de programación o a una página correspondiente a una dirección correcta que no está presente por diversos motivos.

Descriptor de memoria (toda la información relacionada con el espacio de direcciones de un proceso se incluye en una estructura de datos denominada descriptor de memoria) ⇒ Cada proceso tiene asignado un descriptor de memoria de tipo struct mm_struct

```
struct mm_struct{
        struct vm_area_struct *mmap, *mmap_avl, *mmap_cache;
        pgd_t *pgd;
        atomic_t mm_count, mm_users;
        int map_count;
        struct semaphore mmap_sem;
        struct list_head mmlist;
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
        unsigned long rss, total_vm, locked_vm;
        unsigned long def_flags;
        unsigned long cpu_vm_mask;
        unsigned long swap_cnt;
        unsigned long swap_address;
        unsigned int dumpable;
        mm_context_t context;
};
```

NOTA: Cabe destacar que versiones del kernel de Linux posteriores a 2.4.9 una estructura de datos denominada *árboles roji-negros* (red-black trees), siendo uno de los elementos de la estructura rb_root_t mm_rb;

Varios threads pueden compartir el mismo descriptor de memoria. Todos los descriptores de memoria se almacenan en una lista doblemente enlazada. Cada descriptor almacena la dirección de los elementos de la lista adyacentes en el campo mmlist. El primer elemento de la lista es el campo mmlist de init_mm, el descriptor de memoria utilizado por el proceso 0 en la inicialización del sistema.

Otros campos interesantes de mm_struct son: (1) rss: marcos de página asignados al proceso. (2) total_vm: tamaño del espacio de direcciones del proceso en páginas. (3) locked_vm: Número de páginas bloqueadas. (4) mm_count: número de procesos que comparten el mismo descriptor. (5) mmap, mmap_avl, mmap_cache: punteros a regiones de memoria. Los descriptores se almacenan en un caché del *Slab allocator*.

3.4.3.8. Regiones de Memoria

Linux organiza los intervalos de memoria utilizados por un proceso en lo que se denominan regiones de memoria. Estas regiones están caracterizadas por una dirección lógica inicial, una longitud y unos ciertos permisos de acceso. Linux implementa una región de memoria por medio de un objeto del tipo vm_area_struct (descriptor del área de memoria virtual). Cada descriptor identifica un rango o intervalo de direcciones lineales.

```
struct vm_area_struct {
        struct mm_struct *vm_mm;
        unsigned long vm_start, vm_end;
        pgprot_t vm_page_prot;
        unsigned short vm_flags;
        short vm_avl_height;
        struct vm_area_struct *vm_next;
        struct vm_area_struct *vm_avl_left, *vm_avl_right;
        struct vm_area_struct *vm_next_share;
        struct vm_uea_struct **vm_pprev_share;
        struct vm_cperations_struct *vm_ops;
        unsigned long vm_offset;
        struct file *vm_file;
        unsigned long vm_pte;
};
```

Algunos campos interesantes de esta estructura de datos son los siguientes: (1) vm_start: primera dirección válida del intervalo. (2) vm_end: última dirección válida más 1 (primera dirección fuera del intervalo, vm_end – vm_start denota la longitud de la región de memoria). (3) vm_mm: descriptor de memoria dueño de la región (apunta al descriptor de memoria virtual vm_struct del proceso que es dueño de la región). (4) vm_flags: derechos de acceso. (5) vm_next, vm_avl_left, vm_avl_right: punteros a otras regiones del espacio de direcciones.

Las regiones nunca se solapan, y el kernel intenta fusionarlas siempre que los derechos de acceso coincidan: (1) El kernel puede crear una región nueva o agrandar una existente. (2) Igualmente, al eliminar una región el kernel redimensiona las regiones afectadas, incluso dividiéndolas en dos.

Organización de las regiones de memoria. Primera aproximación: lista enlazada mediante vm_next. Una operación que se realiza muy frecuentemente es la búsqueda de la región correspondiente a una dirección lineal $\Rightarrow$ Mejora: ordenar ascendentemente la lista por direcciones lineales. Se considera encontrada la región cuando por primera vez vm_end es mayor a la dirección. Como es muy probable que las siguientes búsquedas que se realicen tengan como resultado la misma región, se mantiene un puntero a la última región encontrada en el descriptor de memoria, mmap_cache.

Las regiones se utilizan para representar intervalos contiguos de direcciones lógicas con los mismos permisos o el mismo comportamiento. Todas las acciones que se deben tomar en relación a una región vienen dadas por las operaciones establecidas en el campo vm_ops (struct vm_operations_struct). Esta estructura contiene tres campos bastante importantes: la función open, la función close y la función nopage. En particular, la función nopage indica que acción realizar cuando una dirección lógica perteneciente a esa región no tiene una página física asociada o da un fallo de protección.

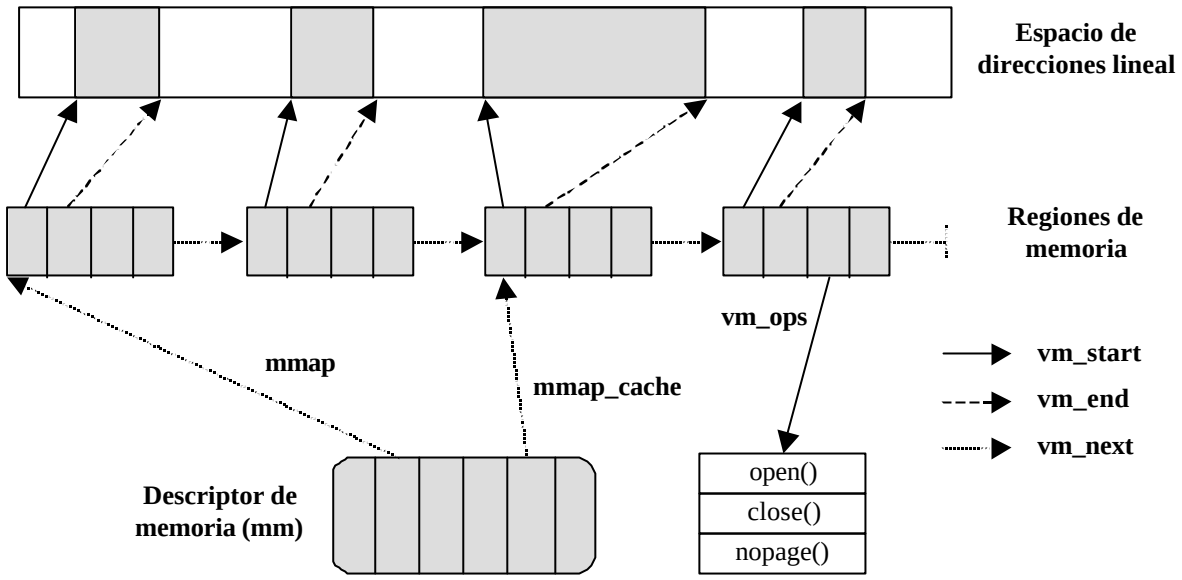

Figura 3.16. Descriptores relacionados con el espacio de direcciones de un proceso en Linux

Pero esta primera alternativa puede conllevar el siguiente problema: La lista puede ser muy grande y las búsquedas requerir mucho tiempo ⇒ Solución: a partir de un cierto número de regiones (AVL_MIN_MAP_COUNT), emplear un árbol AVL además de dicha lista, mediante vm_avl_left y vm_avl_right. Es decir, los descriptores de región de un proceso son referenciados por el descriptor de memoria de un proceso de dos formas diferentes: (1) Manteniendo una lista enlazada de descriptores (el campo mmap contiene la dirección del primer descriptor de región asociado al proceso, y cada descriptor contiene la dirección del siguiente campo vm_next y además la lista se ordena por dirección de fin de regiones). (2) Esta lista no se utiliza para buscar un descriptor de región particular. Para acelerar las búsquedas, los descriptores también se colocan también en un árbol AVL, lo que permite reducir la complejidad de la búsqueda de O(n) a O($\log_2$ n). Recordar que a partir de la versión 2.4.9 del kernel de Linux se utilizan árboles roji-negros para acelerar las búsquedas de un descriptor de región particular.

Un árbol *AVL* (Adelson-Velskii and Landis) es aquel árbol binario que: (1) Siempre está balanceado. (2) Cada elemento (nodo) tiene dos nodos hijos, izquierdo y derecho. (3) Está ordenado, es decir, para cada nodo N, todos los elementos del subárbol izquierdo preceden a N, y los del derecho le siguen. (4) Tiene un factor de balanceo (diferencia de profundidad de los subárboles izquierdo y derecho) que siempre es -1, 0 o +1. Si este factor se modifica más allá de esos límites, al añadir o suprimir elementos en el árbol, se efectúan operaciones de rotación para reequilibrar el árbol AVL. El tiempo de búsqueda pasa de ser proporcional a n a ser proporcional a $\log_2$ n. Es necesario mantener el balance del árbol tras cada inserción o eliminación de una región de memoria.

En la siguiente figura representa la organización del AVL para un proceso cuyo espacio de direcciones contiene las siguientes regiones:

08048000-0804a000	r-xp	00000000	03:02	7914
0804a000-0804b000	rw-p	00001000	03:02	7914
0804b000-08053000	rwxp	00000000	00:00	0
40000000-40005000	rwxp	00000000	03:02	18336
40005000-40006000	rw-p	00004000	03:02	18336
40006000-40007000	rw-p	00000000	00:00	0
40007000-40009000	r--p	00000000	03:02	18255
40009000-40082000	r-xp	00000000	03:02	18060
40082000-40087000	rw-p	00078000	03:02	18060
40087000-400b9000	rw-p	00000000	00:00	0
bfffe000-c0000000	rwxp	fffff000	00:00	0

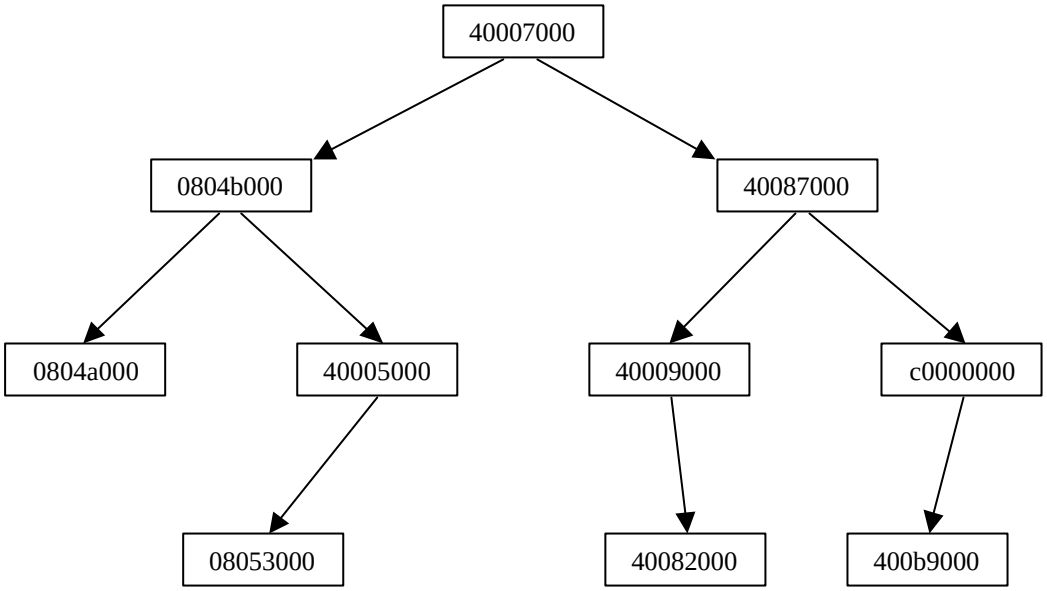

Figura 3.17. Organización de las regiones de memoria de un proceso.

Derechos de acceso. Hasta ahora hemos visto varios flags de página: (1) Los incluidos en cada entrada de la tabla de página (Read/Write, Present y User/Supervisor), utilizados por el hardware para comprobar si el tipo de direccionamiento puede ser ejecutado. (2) Los incluidos en el campo flags del descriptor de página (page), utilizados por Linux para diferentes propósitos. (3) Hay un tercer grupo de flags; los relacionados con todas las páginas de una región de memoria o con la región en sí, en vm_flags. Estos flags son fundamentales para que el gestor de excepción de falta de página determine qué tipo de falta ha ocurrido y qué hacer al respecto.

Flags de región de memoria:

- VM_DENYWRITE: La región mapea un archivo que no puede abrirse para escribir.
- VM_EXEC: Las páginas pueden ejecutarse.
- VM_EXECUTABLE: Las páginas contienen código ejecutable (la región mapea un archivo ejecutable).
- VM_GROWSDOWN: La región puede expandirse hacia abajo (direcciones más bajas).
- VM_GROWSUP: La región puede expandirse hacia arriba(direcciones más altas).
- VM_IO: La región mapea el espacio de direcciones de E/S de un dispositivo.
- VM_LOCKED: La región está bloqueada (las páginas de la región) y no permite swap.
- VM_MAYEXEC: Puede establecerse VM_EXEC.
- VM_MAYREAD: Puede establecerse VM_READ.
- VM_MAYSHARE: Puede establecerse VM_SHARE.
- VM_MAYWRITE: Puede establecerse VM_WRITE.
- VM_READ: Las páginas pueden leerse.
- VM_SHARED: Las páginas pueden ser compartidas.
- VM_SHM: Las páginas de la región se utilizan para memoria compartida IPC.
- VM_WRITE: Las páginas pueden escribirse.
- VM_SEQ_READ: La aplicación accede a páginas secuencialmente.
- VM_RAND_READ: La aplicación accede a páginas de forma aleatoria.
- VM_DONTCOPY: No copiar la región cuando se crea un nuevo proceso con fork().
- VM_DONTEXPAND: Prohíbe la expansión de la región a través de la llamada al sistema mremap().
- VM_RESERVED: No intercambiar (swap) la región.

Existe una correspondencia con los flags de protección $\Rightarrow$ los flags de protección de la región no siempre pueden traducirse directamente a los de la página a nivel hardware: (1) A veces puede que queramos generar

una excepción cuando se realiza cierto tipo de acceso, aunque el acceso esté permitido (por ejemplo, *Copy-on-Write*). (2) Los procesadores i386 sólo tienen 2 bits de protección, y el bit User/Supervisor siempre debe estar a 1 para procesos. Debido a las limitaciones hardware de los procesadores i836, Linux adopta las siguientes reglas: (1) Si una página puede leerse (derechos de lectura), entonces puede ejecutare. (2) Si una página puede escribirse (derecho de ejecución), entonces puede leerse.

Se reducen las 16 posibles combinaciones de los derechos de acceso read, write, execute y share a tan sólo tres: (1) Si la página tiene activado VM_WRITE y VM_SHARE, entonces el bit Read/Write se pone a 1. (2) Si la página tiene activado VM_READ o VM_EXEC, pero no tiene VM_SHARE o VM_WRITE, entonces el bit Read/Write se pone a 0 (para que funcione *Copy-on-Write*). (3) Si la página no tiene ningún derecho, se pone el bit Present a 1 y Linux pone Page size a 1, para distinguirlo del caso en que realmente la página no está presente. Cuando se accede a una página con derechos VM_WRITE y se genera excepción de falta de página, se chequea cuántos procesos usan dicha página. Si sólo la usa un proceso, se pone a 1 el bit Read/Write.

Para realizar la correcta gestión de las regiones de memoria. Las regiones de memoria asociadas a los procesos se gestionan por las funciones contenidas en el archivo fuente <mm/mmap.c> Se definen dos funciones de apoyo en el archivo de cabecera <linux/mm.h>: (1) find_vma, esta función explora el AVL que contiene los descriptores de regiones de memoria asociadas a un proceso para buscar una región de memoria con la dirección especificada, o la primera región situada tras la dirección indicada. (2) find_vma_intersection, esta función se llama para buscar el descriptor de una región de memoria con una intersección con una región especificada. Llama a find_vma para obtener el descriptor de región con la dirección de fin especificada. Si se encuentra un descriptor, comprueba si las dos regiones poseen una intersección no vacía comparando sus direcciones de inicio y de fin. Además, la función get_unmapped_area busca la región de memoria no asignada en el espacio de direccionamiento del proceso actual, a partir de una dirección especificada. Llama a find_vma para obtener el descriptor de la región situada justo tras la dirección especificada. Este tratamiento se efectúa haciendo variar la dirección mientras no se encuentre un área de memoria no utilizarla de tamaño suficiente. Se definen varias funciones de gestión del AVL: (1) avl_neighbours, esta función explora el AVL y devuelve los vecinos de un nodo, es decir, el nodo mayor de su subárbol izquierdo y el más pequeño de su subárbol derecho. (2) avl_rebalance, esta función efectúa las rotaciones necesarias para mantener el AVL equilibrado. (3) avl_insert, esta función inserta un nuevo nodo en el árbol. (4) avl_insert_neighbours, esta función inserta un nuevo nodo en e] árbol y devuelve sus vecinos. (5) avl_remove, esta función suprime un nodo del árbol. (6) La función build_mmap_avl construye el AVL que contiene los descriptores de las regiones de memoria de un proceso llamando a avl_insert para cada región.

También, la función insert_vma_struct añade una región de memoria en el espacio de direccionamiento de un proceso. Llama a avl_insert_neighbours para insertar e] descriptor de la región en el AVL, e inserta este descriptor en la lista de regiones encadenándolo a los descriptores de las regiones vecinas. La función remove_shared_vm_struct suprime un descriptor de región de memoria de un i-nodo. La función merge_segments se llama para fusionar las regiones de memoria de un proceso entre dos direcciones especificadas. Obtiene la primera región de memoria por una llamada a find_vma, y explora todas las regiones situadas delante de la dirección de fin. Para cada región, comprueba si la región es adyacente y si las características de las dos regiones son idénticas. Si es así, las dos regiones se fusionan en una sola.

En lo que respecta a la creación y supresión de regiones de memoria debemos destacar que en el archivo fuente <mm/mmap.c> contiene la implementación de las llamadas al sistema que permiten crear y suprimir regiones de memoria (primitivas mmap y munmap). Contiene también la implementación de la llamada al sistema brk. La función do_mmap procede a la creación de una región de memoria. Primero verifica la validez de sus parámetros. En el caso en que el contenido de un archivo deba proyectarse en memoria en la nueva región, verifica también que el modo de apertura del archivo sea compatible con las modalidades de la proyección. Seguidamente se asigna e inicializa un descriptor de región. Si la región de memoria debe corresponder a un archivo proyectado en memoria, se llama a la operación mmap asociada al i-nodo del archivo. Finalmente, el descriptor de la región se inserta en el espacio de direccionamiento del proceso por una llamada a insert_vm_struct, y se llama a merge_segments para fusionar regiones de memoria si ello es

posible. La función unmap_fixup procede a la liberación de una parte de una región de memoria, cuyo descriptor ha sido suprimido ya del espacio de direccionamiento del proceso actual. Pueden producirse cuatro casos: (1) la región completa debe liberarse: se llama a la operación de memoria clase asociada; (2) una sección situada al principio de la región debe liberarse: la dirección de inicio de la región se actualiza en el descriptor de región; (3) debe liberarse una sección situada al final de la región: la dirección de fin de la región se actualiza en el descriptor de región; (4) debe liberarse una sección situada en medio de la región: la región se descompone entonces en dos. Se asigna un nuevo descriptor de región y se inicializa. El descriptor de la región original se actualiza. Finalmente, el nuevo descriptor se inserta en el espacio de direccionamiento llamando a insert_vm_struct. En los tres últimos casos, unmap_fixup crea un nuevo descriptor para la región modificada, y la inserta en el espacio de direccionamiento del proceso actual.

La función do_munmap libera las regiones de memoria situadas en un intervalo especificado. Primero explora la lista de regiones de memoria contenidas en el intervalo, memoriza sus descriptores en una lista, y los suprime del espacio de direccionamiento del proceso llamando a avl_remove. Tras este bucle, explora la lista de descriptores guardados anteriormente. Por cada región se llama a la operación de memoria unmap, las páginas correspondientes se suprimen de la tabla de páginas, se llama a la función unmap_fixup, y el descriptor de la región se libera. La función sys_munmap implementa la llamada al sistema munmap llamando simplemente a do_munmap. La implementación de la llamada al sistema mmap se encuentra en el archivo fuente <arch/i386/kernel/sys_i386.c> para la arquitectura x86. La función old_mmap controla la validez de sus parámetros y llama a la función do_mmap. La función sys_brk implementa la primitiva brk. Ésta verifica que la dirección pasada está situada en eL segmento de datos, y que no sobrepasa los límites impuestos al proceso. En el caso en que la dirección de fin del segmento de datos sea disminuida, se llama a la función do_mumnap para liberar el área de memoria especificada. En el caso en que el tamaño del segmento de datos deba incrcmentarse, se llama a la función find_vma_intersection para verificar que el área a asignar no entra en conflicto con una región existente, y seguidamente se extiende la región de memoria del segmento llamando a la función do_mmap.

3.4.3.9. Gestor de Faltas de Página

El gestor de faltas de página debe distinguir la causa de la falta: (1) Error de programación. (2) Referencias a páginas legítimas del espacio de direcciones del proceso pero no presentes: (2.1) Por *Copy-on-Write*. (2.2) Por swapping o intercambio (cuando el sistema necesita marcos de página libres, Linux puede volcar a disco parte de las páginas de datos de un proceso, luego cuando el proceso desea continuar su ejecución necesita volver a traer a memoria principal estas páginas). Para este caso existe la función, do_swap_page, que se llama para volver a cargar en memoria el contenido de una página situada en el espacio de swap. Si una operación swapin está asociada a la región de memoria que contiene la página, se llama. En caso contrario, se llama a la función swap_in. En ambos casos, la página asignada se inserta en el espacio de direccionamiento del proceso actual. (2.3) Porque no han sido asignadas todavía. (2.4) Expansión de pila. El gestor de falta de página es do_page_fault() $\Rightarrow$ el gestor compara la dirección lineal que causó el falta (cr2) con las regiones de memoria del proceso actual (current process) para determinar qué hacer, de acuerdo con el esquema que se muestra en la siguiente figura.

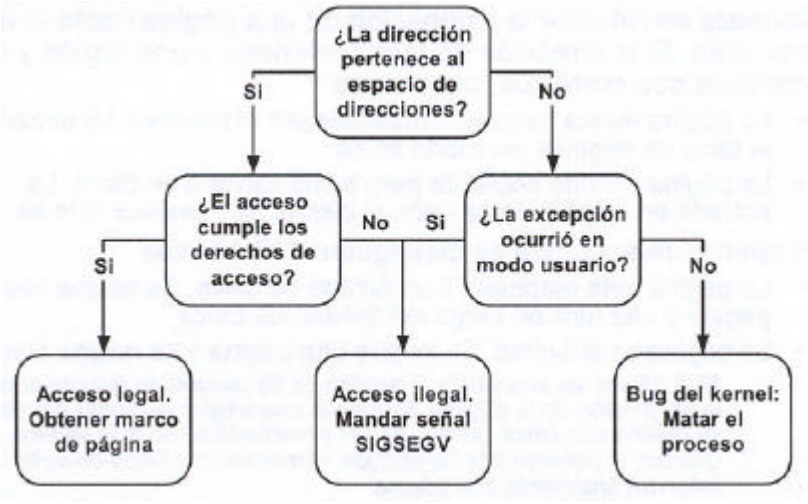

Figura 3.18. Esquema general del gestor de faltas de páginas.

En general, el gestor de faltas de páginas es mucho más complejo ya que debe reconocer varios casos particulares, según se muestra en la siguiente figura:

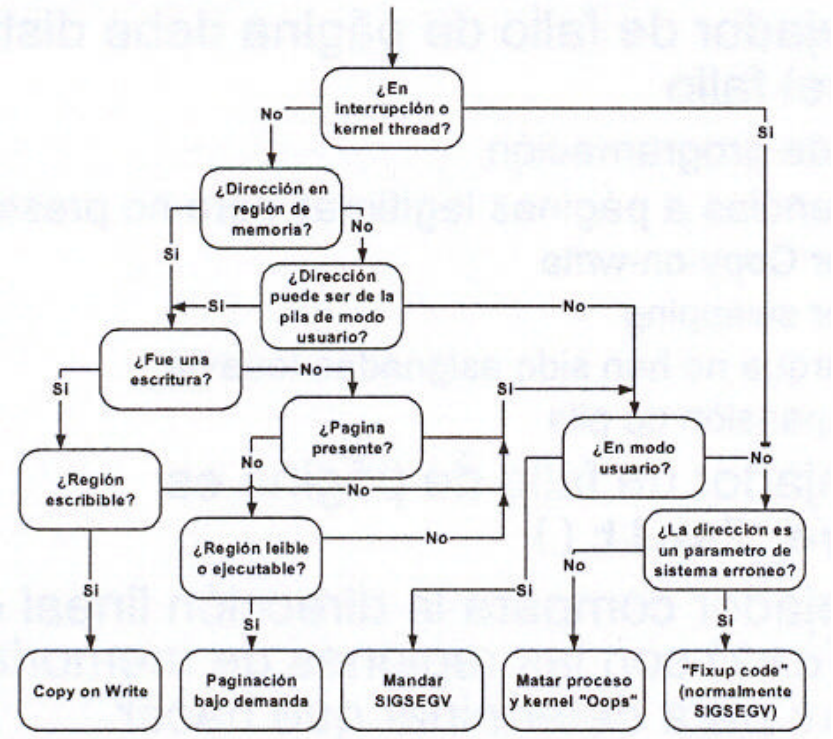

Figura 3.19. Diagrama de flujo del gestor de faltas de páginas.

En general, la función do_page_fault, declarada en el archivo fuente <arch/i386/mm/fault.c> para la arquitectura i836, se llama para tratar las excepciones de faltas de páginas. Primero obtiene el descriptor de la región de memoria afectada llamando a find_vma, y luego comprueba, entre otros, lis siguientes tipos de error: (1) Si el error lo ha causado un acceso a una página no presente en el segmento de pila, se llama a la función expand_stack para aumentar el tamaño de la pila. (2) Si el error se debe a un acceso en escritura en una región de memoria protegida en lectura, el error se señala al proceso actual enviándole la señal SIGSEGV. (3) Si el error es debido a un acceso en escritura a una página de memoria protegida en lectura, a fin de implementar la copia en escritura, se llama a la función do_wp_page (función para gestionar *Copy-on-Write*, cuando un proceso accede en escritura a una página compartida y protegida en lectura exclusiva) para efectuar la copia. (4) Si el error se debe a un acceso a una página no presente en memoria, se llama a la función do_no_page (función que se llama en el acceso a una página no presente en memoria) a fin de cargar la página en memoria.

3.4.3.9.1. Paginación por Demanda

Un acceso a un fragmento de programa que no está en memoria principal fuerza una lectura del bloque correspondiente del archivo ejecutable en disco. En general, consiste en retardar la asignación de una página hasta el último momento. Si la dirección de falta pertenece a una región y los permisos son correctos, puede que: (1) La página nunca haya sido accedida por el proceso. La entrada en la tabla de páginas será todo ceros. (2) La página ha sido accedida pero se ha salvado en disco. La entrada en la tabla no es cero, si bien el bit Present si lo es.

Si nunca fue accedida se distinguen dos casos: (1) La página está mapeada a un archivo de disco. Se asigna una página y una función carga del archivo los datos. (2) La página es anónima. Se asigna una página y se rellena con ceros. Si la página es anónima y el acceso es de lectura, se retarda aun más la asignación de la página. Se asigna una página estándar del sistema ya rellena con ceros (ZERO_PAGE) y marcada como sólo lectura. Cuando el proceso intente escribir, el mecanismo *Copy-on-Write* asignará finalmente una página.

3.4.3.9.2. Copy-on-Write

Cuando se crea un proceso nuevo con la llamada fork(), el proceso hijo se ejecuta exactamente sobre la misma memoria que le proceso padre hasta que uno de los dos intenta modificar alguna variable, momento en el que efectivamente se duplica la página modificada para que cada uno tenga su propia copia.

En sistemas UNIX de primera generación, ante una llamada fork(), el kernel duplica por completo el espacio de direcciones del padre en el estricto sentido de la palabra y asignan la copia al proceso hijo. Para el proceso hijo habría que: (1) asignar marcos de página para la tabla de páginas; (2) asignar marcos de página para las páginas; (3) inicializar las tablas de páginas; (4) copiar las páginas del proceso padre. Este proceso es largo, ensucia los cachés y además muchas veces es inútil puesto que el proceso hijo inmediatamente ejecuta execve().

Linux ofrece una aproximación más eficiente denominada *Copy-on-Write* (copia en escritura) que se caracteriza por los siguientes aspectos: (1) En lugar de duplicar los marcos de página, se comparten entre padre e hijo. (2) Las páginas no se pueden modificar. (3) Cuando un proceso padre o hijo intenta modificar una página (intenta escribir un marco de página compartido) ocurre una excepción de falta de página y el kernel duplica la página en un nuevo marco de página y lo marca como modificable. (4) El marco de página original sigue como solo lectura (write-protected). Si el proceso intenta modificarlo, el kernel comprueba que es el único propietario y la marca como modificable.

Se emplea el campo count del descriptor de página para saber cuántos procesos comparten el marco de página. Cuando un proceso libera un marco de página o ejecuta sobre él un *Copy-on-Write*, su campo count se decrementa. El marco de página se libera sólo cuando count se queda a NULL.

La función do_wp_page se llama para gestionar la operación *Copy-on-Write* (copia en escritura), cuando un proceso accede en escritura a una página compartida y protegida en lectura exclusiva (write-protected). Se asigna una nueva página llamando a __get_free_page, y se comprueba si la página afectada por la excepción es compartida entre varios procesos. Si es así, su contenido se copia en la nueva página, que se inserta en la tabla de páginas del proceso actual utilizando set_pte, y el número de referencias a la anterior página se decrementa por una llamada a free_page. En el caso en que la página afectada no sea compartida, su protección simplemente se modifica para hacer posible la escritura.

3.4.3.10. Gestión del *heap*

Cada proceso en UNIX posee una región de memoria específica denominada *heap*, que se utiliza para satisfacer las demandas de memoria dinámica del proceso. Los campos start_brk y brk del descriptor de memoria delimitan las direcciones iniciales y finales, respectivamente, de una región. Las siguientes funciones

de la librería C pueden utilizarse por el proceso para asignar y liberar memoria dinámica: mallos(size), calloc(n, size), free(addr), brk(addr) y sbrk(incr).

La función malloc permite asignar un nuevo bloque de memoria dinámica. El parámetro size especifica el tamaño en bytes del bloque a asignar, y devuelve la dirección del área asignada, o bien el valor NULL en caso de error. La función calloc permite también asignar un bloque de memoria dinámica, pero está previsto par asignar un array. El parámetro n especifica el número de elementos del array, y size indica el tamaño de cada uno de los elementos del array, expresado en bytes. Por otra parte, la función free libera un bloque de memoria dinámica previamente asignado por malloc o calloc y que tiene como dirección inicial addr. Tras la ejecución de free el bloque de memoria se libera y addr ya no contiene una dirección válida.

Linux puede modificar el tamaño del heap directamente, utilizando la función brk (un proceso modifica el tamaño del segmento de datos). El parámetro addr especifica el nuevo valor de current->mm->brk y el valor devuelto es la nueva dirección final de la región de memoria (el proceso debe comprobar si ésta coincide con el valor addr requerido). Por último, sbrk es similar a brk(), excepto que el parámetro incr especifica el incremento o decremento del tamaño del *heap* en bytes.

3.4.4. Intercambio (swapping) en Linux

3.4.4.1. Visión General del Intercambio (swapping) en Linux

Si un proceso necesita cargar una página de memoria virtual a memoria física y no hay ninguna página de memoria física libre, el sistema operativo tiene que crear espacio para la nueva página eliminando alguna otra página de memoria física. Si la página que se va a eliminar de memoria física provenía de un archivo imagen o de un archivo de datos sobre el que no se ha realizado ninguna escritura, entonces la página no necesita ser guardada. Tan sólo se tiene que desechar y si el proceso que la estaba utilizando la vuelve a necesitar simplemente se carga nuevamente desde el archivo imagen o de datos.

Por otra parte, si la página había sido modificada, el sistema operativo debe conservar su contenido para que pueda volver a ser accedido. Este tipo de página se conoce como página modificada (dirty page) y para poderla eliminar de memoria se ha de guardar en un dispositivo de intercambio (dispositivo de swap). El tiempo de acceso al dispositivo de intercambio es muy grande en relación a la velocidad del procesador y la memoria física, y el sistema operativo tiene que conjugar la necesidad de escribir páginas al disco con la necesidad de retenerlas en memoria para ser usadas posteriormente.

Si el algoritmo utilizado para decidir qué páginas se descartan o se envían a disco (algoritmo de intercambio) no es eficiente, entonces se produce una situación llamada hiperpaginación (thrashing). En este estado, las páginas son continuamente copiadas a disco y luego leídas, con lo que el sistema operativo está demasiado ocupado para hacer trabajo útil. Si, por ejemplo, supongamos que a un marco de página se accede constantemente, entonces no es un buen candidato para intercambiarlo a disco. El conjunto de páginas que en el instante actual está siendo utilizado por un proceso se llama páginas activas *(working set)*. Un algoritmo de intercambio eficiente ha de asegurarse de tener en memoria física las páginas activas de todos los procesos. Linux utiliza la técnica de reemplazo de marco de página por antigüedad (Last Recently Used, LRU) para escoger de forma equitativa y justa las páginas a ser intercambiadas o descartadas del sistema. Este esquema implica que cada página del sistema ha de tener una antigüedad que ha de actualizarse conforme la página es accedida. Cuanto más se accede a una página más joven es; por el contrario cuanto menos se utiliza más antigua e inútil. Las páginas antiguas son las mejores candidatas para ser intercambiadas.

3.4.4.2. Dispositivos de *swap*

Cuando el kernel necesita memoria, puede eliminar páginas en memoria principal. Si el contenido de estas páginas ha sido modificado, es necesario guardarlas en disco: una página correspondiente a un archivo proyectado en memoria se rescribe en el archivo, y una página correspondiente a los datos se guarda en un

dispositivo de swap. Un dispositivo de swap puede ser un dispositivo en modo bloque, por ejemplo una partición en disco, o un disco normal. Previamente debe inicializarse mediante la orden *mkswap*.

Linux es capaz de utilizar varios dispositivos de swap. Cuando una página debe guardarse, los dispositivos de swap activos se exploran para encontrar un lugar donde escribir la página. La activación de un dispositivo de swap se efectúa llamando a la función de sistema *swapon*.

Contrariamente a la mayor parte de sistemas operativos, Linux no se limita a la activación de dispositivos de swap. Un dispositivo activo puede desactivarse, sin tener que reiniciar el sistema. Al desactivarlo; todas las páginas guardadas en el dispositivo se vuelven a cargar en memoria. La llamada al sistema *swapoff* efectúa esta desactivación.

3.4.4.3. Gestión de los Dispositivos de *swap*

La llamada al sistema *swapon* permite activar un dispositivo de swap. Su prototipo es el siguiente:

```
#include <unistd.h>
#include <linux/swap.h>
int swapon(const char *pathname, int swapflags);
```

La primitiva *swapon* activa el dispositivo cuyo nombre se especifica en el parámetro *pathname*. Éste puede ser un dispositivo en modo bloque, o un archivo regular. En los dos casos, debe haber sido inicializado por la orden *mkswap*. El parámetro *swapflags* permite indicar la prioridad del dispositivo. Cuando debe guardarse una página, Linux explora la lista de dispositivos de swap y utiliza el que tiene mayor prioridad y que posee páginas sin asignar. El valor de *swapflags* se expresa añadiendo la prioridad deseada (un entero entre 1 y 32767) a la constante SWAP_FLAG_PREFER. En caso de éxito se devuelve el valor 0. En caso de error, *swapon* devuelve el valor-1 y la variable errno puede tomar los valores de error siguientes: (1) EFAULT (pathname contiene una dirección no válida); (2) EINVAL (pathname no se refiere a un dispositivo en modo bloque ni a un archivo regular); (3) ELOOP (se ha encontrado un ciclo de enlaces simbólicos); (4) ENAMETOOLOOO (pathnarne especifica un nombre de archivo demasiado largo); (5) ENOENT (pathnarne se refiere a un nombre de archivo inexistente); (6) ENOMEM (el kernel no ha podido asignar memoria para sus descriptores internos); (7) ENOTDIR (uno de los componentes de pathnarne, utilizado como nombre de directorio, no lo es); (8) EPERM (el proceso que llama no posee los privilegios necesarios, o el número máximo de dispositivos activos se ha alcanzado ya)

La desactivación de un dispositivo de *swap* se efectúa llamando a la primitiva *swapoff*. Su prototipo es el siguiente:

```
#include <unistd.h>
int swapoff(const char *pathname);
```

El parámetro pathname especifica el nombre del dispositivo de swap a desactivar. En caso de éxito, *swapoff* devuelve el valor 0; si no devuelve el valor -1 y la variable errno puede tomar los valores de error siguientes: (1) EFAULT (pathnarne contiene una dirección no válida); (2) EINVAL (pathnarne no se refiere a un dispositivo de swap activo); (3) ELOOP (se ha encontrado un ciclo de enlaces simbólicos); (4) ENAMETOOLOOO (pathnarne especifica un nombre de archivo demasiado largo); (5) ENOENT (pathnarne se refiere a un nombre de archivo inexistente); (6) ENOMEM (la memoria disponible es demasiado restringida para recargar todas las páginas contenidas en el dispositivo de swap); (7) ENOTDIR (uno de los componentes de pathname, utilizado como nombre de directorio, no lo es); (8) EPERM (el proceso que llama no posee los privilegios necesarios).

3.4.4.4. Gestión del *swap*, Perspectiva General de la Implementación

3.4.4.4.1. Formato de los Dispositivos de swap

Bajo Linux, todo dispositivo en modo bloque o archivo regular puede usarse como dispositivo de swap. Sin embargo, un dispositivo de swap debe haber sido inicializado por la orden *mkswap*. Esta orden crea un directorio al arrancar el dispositivo. Este directorio tiene el tamaño de una página de memoria y está constituido por una tabla de bits. Cada uno de los bits indica si la página correspondiente en el dispositivo es utilizable. Si suponemos, por ejemplo, que la orden *mkswap* se ejecuta sobre un archivo de cuatro Megabytes (4 Mb), en una máquina basada en un procesador i386 que utiliza un tamaño de página de cuatro kilobytes (4 Kb), el contenido del directorio es el siguiente: (1) el primer bit está a 1, e indica que la primera página, que contiene el directorio, no es utilizable; (2) . los 1023 bits siguientes están a 0, e indican que las páginas correspondientes son utilizables; (3) los bits siguientes están a 1, para indicar que las páginas siguientes no existen en el archivo.

La primera página contiene también una firma: los 10 últimos bytes contienen la cadena de caracteres "SWAP-SPACE", Esta firma permite que el kernel verifique la validez de un dispositivo en su activación. El formato de un dispositivo de swap define su tamaño límite. En la medida en que un bit debe existir en el directorio (que tiene el tamaño de una página) para cada página, el número máximo de páginas puede expresarse por la fórmula: (tamaño_página – 10) * 8. En la arquitectura i386, el tamaño máximo de un dispositivo de swap es pues de unos 128 megabytes.

3.4.4.4.2. Descriptores de Dispositivos de swap

El kernel guarda en memoria una lista de dispositivos de swap activos. Se utiliza una tabla de descriptores: en la que cada uno describe un dispositivo de swap. La estructura swap_info_struct, declarada en el archivo de cabecera <linux/swap.h>, define el formato de estos descriptores. Contiene los campos siguientes:

```
struct swap_info_struct {
        unsigned int flags;             // Estado del dispositivo
        kdev_t swap_device;             // Identificador del dispositivo en modo bloque
        struct inode * swap_file;       // Puntero al descriptor del inodo correspondiente, en el caso de
un
                                        // archivo regular
        struct vfsmount *swap_vfsmnt; // Descriptor del sistema de archivos montado del dispositivo
                                        // de swap
        unsigned short * swap_map;      // Puntero a una tabla de bytes que representan el estado de cada
                                        // página
        unsigned char * swap_lockmap // Puntero a una tabla de bits que indican por cada página si está
                                        // bloqueada o es utilizable
        int lowest_bit;                 // Número de la menor página utilizable
        int highest_bit;                // Número de la mayor página utilizable
        int cluster_next;               // Siguiente página para ser scaneada cuando se esté buscando
uno
                                        // libre
        int cluster_nr;                 // Número de página libre localizadas
        int prio;                       // Prioridad asociada al dispositivo
        int pages;                      // Número de páginas disponibles (asignadas o no)
        unsigned long max;          // Número de páginas en el dispositivo
        int next;                       // Puntero al descriptor de dispositivo de swap siguiente en la
lista
};
```

Los campos swap_map y swap_lockmap se utilizan para mantener actualizadas las páginas utilizadas. swap_map apunta a una tabla de bytes, cada uno de los cuales contiene el número de referencias a la página. Si este número es nulo, significa que la página está disponible, si no está asignada. swap_lockmap apunta a una tabla de bits, en la que cada miembro indica si la página correspondiente está bloqueada (si está al) o está

libre (si está a 0). Esta tabla se inicializa a partir de la que se encuentra en el directorio del dispositivo de swap. En las entradas/salidas sobre el dispositivo, el bit correspondiente se posiciona a 1 para impedir cualquier otra lectura o escritura durante la entrada/salida.

3.4.4.4.3. Direcciones de Entradas del swap

Cuando una página se escribe en un dispositivo de swap, se le atribuye una dirección. Esta dirección combina el número del dispositivo de swap y el índice de la página utilizada en el dispositivo. Varias macros, declaradas en el archivo de cabecera <asm/pgtable.h>, manipulan estas direcciones: (1) SWP_ENTRY (Combina un número de dispositivo y un índice de página para formar una dirección de entrada de swap); (2) SWP_TYPE (Devuelve el número del dispositivo correspondiente a una dirección de entrada de swap); (3) ISWP_OFFSET (Devuelve el índice de la página correspondiente a una dirección de entrada de swap).

Cuando una página debe descartarse de la memoria, se le asigna una página en un dispositivo de swap, y la dirección de dicha página se memoriza para que Linux pueda recargar la página posteriormente. En lugar de utilizar una tabla que establezca una correspondencia entre las direcciones de páginas de memoria y las direcciones de entradas de swap, Linux utiliza un método original: (1) Cuando una página de memoria se descarta, la dirección de la entrada de swap asignada se guarda en la tabla de páginas, en lugar de la dirección física de la página. Esta dirección está concebida para indicar al procesador que la página no está presente en memoria. (2) Cuando se efectúa un acceso de memoria sobre una página que se ha guardado en un dispositivo de swap, el procesador detecta que la página no está presente en memoria y desencadena una interrupción. Al tratar esta interrupción, Linux extrae la dirección de entrada de swap de la entrada correspondiente de la tabla de páginas, y utiliza esta dirección para localizar la página en el swap.

3.4.4.4.4. Selección de páginas a descartar

Con el objetivo de descartar páginas de la memoria, se ejecuta el proceso *kswapd*. Este proceso se lanza al arrancar el sistema y se ejecuta en modo kernel. Es más o menos equivalente al proceso bdflush, descrito al hablar del buffer caché. La función del proceso *kswapd* es descartar las páginas inútiles de la memoria. La mayor parte del tiempo, kswapd se duerme y se despierta cuando el kernel se queda sin memoria. Entonces explora la lista de proceso e intenta descartar páginas no utilizadas.

Con el objetivo de determinar las páginas de memoria no utilizadas, el kernel emplea el campo age del descriptor de página de memoria. Este contador se incrementa cuando se utiliza la página, y se decrementa cuando deja de utilizarse. Sólo las páginas que poseen un valor del campo age nulo pueden descartarse de la memoria.

3.4.4.5. Gestión del *swap*, Perspectiva Detallada de la Implementación

3.4.4.5.1. Gestión de los Dispositivos de swap

El archivo fuente mm/swapfile.c contiene las funciones que gestionan los archivos o dispositivos utilizados para guardar las páginas descargadas de la memoria.

La tabla swap_info contiene las características de los dispositivos de swap activos. La variable swap_list contiene el índice del dispositivo al que está asociada la prioridad más importante. La función scan_swap_map busca una página disponible en el dispositivo especificado. Explora la tabla de bytes que indican el estado de las páginas y devuelve el número de una página disponible.

La función get_swap_page se llama para asignar una página en un dispositivo de swap. Explora la lista de dispositivos disponibles y llama a scan_swap_map para cada dispositivo. Cuando se encuentra una página, se devuelve su dirección de swap. La función swap_free se llama para liberar una página en un dispositivo de swap. Decrementa el byte que contiene el número de referencias a la página.

Las funciones unuse_pte, unuse_pmd y unuse_pgd se llaman al desactivarse un dispositivo de swap. Exploran las tablas de páginas y recargan en memoria todas las páginas presentes en el dispositivo especificado. La función unuse_vma utiliza unuse_pgd para recargar todas las páginas correspondientes a una región de memoria.

La función unuse_process explora la lista de regiones de memoria contenidas en el espacio de direccionamiento de un proceso. Llama a unuse_vma por cada región. La función try _to_unuse se llama para recargar en memoria todas las páginas presentes en el dispositivo especificado. Explora la tabla de procesos y llama a la función unuse_process para recargar el espacio de direccionamiento de cada proceso.

La función sys_swapoff implementa la primitiva *swapoff* que desactiva un dispositivo de swap. Primero verifica que el proceso que llama posee los privilegios necesarios, y busca el descriptor del dispositivo en la tabla swap_info, y lo suprime de la lista. Seguidamente llama a try_to_unuse para recargar en memoria todas las páginas guardadas en ése dispositivo. Si esto falla, el descriptor del dispositivo se inserta de nuevo en la lista y se devuelve un error. En caso de éxito, se llama a la operación de archivo release asociada al dispositivo, y el descriptor se libera.

La función sys_swapon implementa la primitiva *swapon* que activa un dispositivo de swap. Primero verifica que el proceso que llama posee los privilegios necesarios, luego busca un descriptor libre en la tabla swap_info, e inicializa dicho descriptor. Seguidamente, se asigna una página de memoria, el directorio del dispositivo de swap se lee en esta página, y se verifica la firma del dispositivo de swap. Tras esta verificación, se efectúa un bucle con el objetivo de contar las páginas disponibles en el dispositivo, y la tabla de bytes que indica el estado de cada página se asigna e inicializa. Para terminar, el descriptor del dispositivo se inserta en la lista.

3.4.4.5.2. Entrada/Salida de páginas de swap

El archivo fuente mm/page_io.c contiene las funciones de entrada/salida de páginas sobre dispositivos de swap. La función rw_swap_page lee o escribe una página en un dispositivo de swap. Verifica que el dispositivo especificado está activo, y que el número de página es válido, y a continuación bloquea la página en cuestión en el descriptor del dispositivo de swap. Seguidamente, comprueba el tipo del dispositivo de swap: (1) Si se trata de un dispositivo en modo bloque, se llama a la función ll_rw_page para leer o escribir el contenido de la página. (2) Si se trata de un archivo de swap, se llama a la operación brnap asociada al inodo del archivo para determinar la dirección de los bloques que componen la página, y se llama a ll_rw_swap_file para leer o escribir los bloques. Esta función, definida en el archivo fuente drivers/block/ll_rw_blk.c, genera peticiones de entradas/salidas para los bloques especificados.

La función swap_after_unlock_page se llama cuando la lectura o la escritura de una página en un dispositivo de swap ha terminado, y desbloquea la página en cuestión en el descriptor del dispositivo de swap. La función ll_rw_page se utiliza rara leer o escribir una página en un dispositivo en modo bloque. Bloquea la página en memoria activando el indicador PG_locked, y llama a la función brw_page para efectuar la entrada/salida.

Las funciones read_swap_page y write_swap_page, declaradas en el archivo de cabecera <linux/swap.h>, llaman a rw_swap_page para efectuar una lectura o una escritura de página.

La función swap_in, definida en el archivo fuente mm/page_alloc.c, se llama para cargar una página en memoria para un proceso. Asigna una página de memoria, y llama a read_swap_page para cargar su contenido. Seguidamente, la dirección de la página se registra en la entrada de la tabla de páginas afectada.

3.4.4.5.3. Eliminación de Páginas de Memoria

El archivo fuente mm/vmscan.c contiene las funciones que deciden la eliminación de páginas de memoria y su escritura en un dispositivo de swap. La función try_to_swap_out se llama para intentar eliminar una página específica. Si la página está reservada o bloqueada, o si ha sido accedida recientemente, no se elimina de la

memoria. En caso contrario, se comprueba su estado. Si la página ha sido modificada, debe guardarse en disco: si hay asociada una operación de memoria swapout a la región que contiene la página, se llama; si no, se asigna una entrada de swap llamando a get_swap_page, y se llama a la función write_swap_page para escribir la página. En el caso en que la página no haya sido modificada, se llama a la función page_unuse para suprimirla del caché de páginas, y la página se libera por free_page.

Las funciones swap_out_prnd y swap_out_pgd exploran la tabla de páginas de un proceso intentando eliminar cada página. Estas funciones paran su ejecución en cuanto una página ha sido eliminada. La función swap_out_vma llama a swap_out_pgd para intentar eliminar una página en una región de memoria, parando su ejecución cuando una página ha sido eliminada. La función swap_out_process se llama para eliminar una página del espacio de direccionamiento de un proceso. Explora la lista de regiones de memoria contenidas en el espacio de direccionamiento del proceso, llama a swap_out_vma para cada región y para su ejecución en cuanto se elimina una página.

La función swap_out se llama para eliminar una página que forme parte del espacio de direccionamiento de uno de los procesos existentes. Explora la tabla de procesos y llama a la función swap_out_process para cada proceso que posee páginas residentes en memoria. Cuando una página ha podido ser eliminada o cuando ninguna página puede ser eliminada de la memoria, para su ejecución. La función try_to_free_page se llama para intentar liberar una página de memoria cualquiera. Se llama sucesivamente a shrink_mmap, shm_swap y swap_out para intentar liberar páginas de memoria.

La función kswapd implementa el proceso kswap que elimina páginas de la memoria en segundo plano. Este proceso efectúa un bucle infinito durante el cual se suspende por una llamada a interruptible_sleep_on, y seguidamente intenta liberar páginas de memoria llamando a try_to_free_page cuando se despierta. La función swap_tick se llama en cada ciclo de reloj, y despierta el proceso kswapd para liberar memoria, si el número de páginas disponibles es inferior al umbral tolerado.

3.4.5. Cachés en Linux para la Gestión de la Memoria

Linux emplea varios cachés para la gestión de la memoria que pasamos a enunciar a continuación:

- *Buffer Caché* <fs/buffer.c>. Contiene buffers de datos que son utilizados por los gestores de dispositivos de bloques. Estos buffers son de tamaño fijo (por ejemplo 512 bytes) y contienen bloques de información que ha sido bien leída de un dispositivo de bloques o que ha de ser escrita. Un dispositivo de bloques es un dispositivo sobre el que sólo se pueden realizar operaciones de lectura o escritura de bloques de tamaño fijo. Todos los discos duros son dispositivos de bloque. El buffer caché (caché de disco formado por buffers, que almacena un único bloque de disco) está indexado vía el identificador de dispositivo y el número de bloque deseado, índice que es utilizado para una rápida localización del bloque. Los dispositivos de bloque son exclusivamente accedidos a través del buffer caché. Si un dato (bloque) se puede encontrar en el buffer caché, entonces no es necesario leerlo del dispositivo de bloques físico, por el disco duro, y por tanto el acceso es mucho más rápido, puesto que se reduce el número de accesos a disco.
- *Caché de Páginas* <mm/filemap.c>. Éste caché es un caché de disco formado por páginas, y cada página en el caché corresponde a varios bloques de un archivo regular (normal) o de un archivo de dispositivo de bloque (el número de bloques contenidos en una página depende del tamaño del bloque). Todos los bloques son contiguos desde un punto de vista lógico (es decir, ellos representan una parte íntegra de un archivo regular o de un archivo de dispositivo de bloque). Éste se utiliza para acelerar el acceso a imágenes y datos en disco. Se usa para guardar el contenido lógico de un archivo de páginas y se accede vía el archivo y el desplazamiento dentro del archivo. Conforme las páginas se leen en memoria, se almacenan en el caché de páginas. Para reducir el número de accesos a disco, antes de activar la operación de E/S de página, el kernel debería comprobar si los datos solicitados están ya en el caché de páginas.
- No debemos también olvidar el *Dentry Caché* </file/dcache.c> (también denominado, Dcache o caché de nombres) que lo utiliza el sistema de archivos virtual (VFS) para acelerar la traducción de un nombre completo (pathname) de archivo al correspondiente inodo.

- *Cache de intercambio o swap* <swap.h, mm/swap_state.c, mm/swapfile.c>. Sólo las páginas que han sido modificadas (dirty) se guardan en el dispositivo de intercambio. Mientras no vuelvan a ser modificadas después de haber sido guardadas en el dispositivo de intercambio, la próxima vez que necesiten ser descartadas (swap out) no será necesario copiarlas al dispositivo de intercambio pues ya están allí. Simplemente se las elimina. En un sistema con mucho trasiego de páginas, esto evita muchas operaciones de disco innecesarias y costosas.
- *Caches Hardware*. Es un caché normalmente implementada en el propio procesador; la caché de entradas de tabla de página. En este caso, el procesador no necesita siempre leer la tabla de páginas directamente, sino que guarda en este caché las traducciones de las páginas conforme las va necesitando. Estos son los Translation Lookaside Buffers (TLB) que contienen copias de las entradas de la tabla de páginas de uno o más procesos del sistema. Cuando se hace la referencia a una dirección virtual, el procesador intenta encontrar en el TLB la entrada para hacer la traducción a memoria física. Si la encuentra, directamente realiza la traducción y lleva a cabo la operación. Si el procesador no puede encontrar la página buscada, entonces tiene que pedir ayuda al sistema operativo. Esto lo hace enviando una señal al sistema operativo indicando que se ha producido un fallo de TLB. Un mecanismo específico al sistema se utiliza para enviar esta excepción al código del sistema operativo que puede arreglar la situación. El sistema operativo genera una nueva entrada de TLB para la dirección que se estaba traduciendo. Cuando la excepción termina, el procesador hace un nuevo intento de traducir la dirección virtual. Esta vez tendrá éxito puesto que ahora ya hay una entrada en la TLB para esa dirección.

El inconveniente de utilizar memorias caché, tanto hardware como de otro tipo, es que para evitar esfuerzos Linux tiene que utilizar más tiempo y espacio para mantenerlas y, si se corrompe su contenido, el sistema dejará de funcionar.