LECCIÓN 11: PAGINACIÓN EN LINUX

PÁGINAS

11.1 Introducción 2

11.2 Memoria Virtual 3

11.3 Modelo Abstracto de Memoria Virtual 6

11.3.1 Paginación por Demanda 11

11.3.2 Intercambio (swapping).......... 12

11.3.3 Memoria virtual compartida.......... 13

11.3.4 Modos de direccionamiento físico y virtual.......... 13

11.3.5 Control de acceso.......... 13

11.4 Cachés.......... 16

11.5 Tablas de Páginas en Linux.......... 18

11.6 Asignación y liberación de páginas.......... 22

11.6.1 Buddy System.......... 22

11.6.2 Slab Allocator.......... 27

11.6.3 Gestión del área de memoria contigua.......... 28

11.6.4 Gestión del área de memoria no contigua.......... 31

11.7 Proyección de Memoria (Memory Mapping).......... 33

11.8 Paginación por Demanda.......... 34

11.9 La Cache de Páginas de Linux.......... 35

11.10 Copy-on-Write.......... 36

11.11 Tratamiento de las excepciones.......... 37

11.1 Introducción

Para entender cómo funciona el proceso de ***paginación*** en un sistema Linux, primero vamos a dar una introducción al mismo dando una ligera idea de todos los aspectos que pueden influir de alguna manera en dicho proceso.

Gestión de memoria

El subsistema de gestión de memoria es una de las partes más importantes del sistema operativo. Ya desde los tiempos de los primeros ordenadores, existió la necesidad de disponer de más memoria de la que físicamente existía en el sistema. Entre las diversas estrategias desarrolladas para resolver este problema, la de mayor éxito ha sido la memoria virtual. La memoria virtual hace que el sistema parezca disponer de más memoria de la que realmente tiene compartiéndola entre los distintos procesos conforme la necesitan.

La memoria virtual hace más cosas aparte de ampliar la memoria del ordenador. El subsistema de gestión de memoria ofrece:

Espacio de direcciones grande

El sistema operativo hace que el sistema parezca tener una gran cantidad de memoria. La memoria virtual puede ser muchas veces mayor que la memoria física del sistema.

Protección

Cada proceso del sistema tiene su propio espacio de direcciones virtuales. Este espacio de direcciones está completamente aislado de otros procesos, de forma que un proceso no puede interferir con otro. También, el mecanismo de memoria virtual ofrecido por el hardware permite proteger determinadas áreas de memoria contra operaciones de escritura. Esto protege el código y los datos de ser sobre-escritos por aplicaciones perversas.

Proyección de Memoria (Memory Mapping)

La proyección de memoria se utiliza para asignar un fichero sobre el espacio de direcciones de un proceso. En la proyección de memoria, el contenido del fichero se “engancha” directamente sobre el espacio de direcciones virtual del proceso.

Asignación Equitativa de Memoria Física

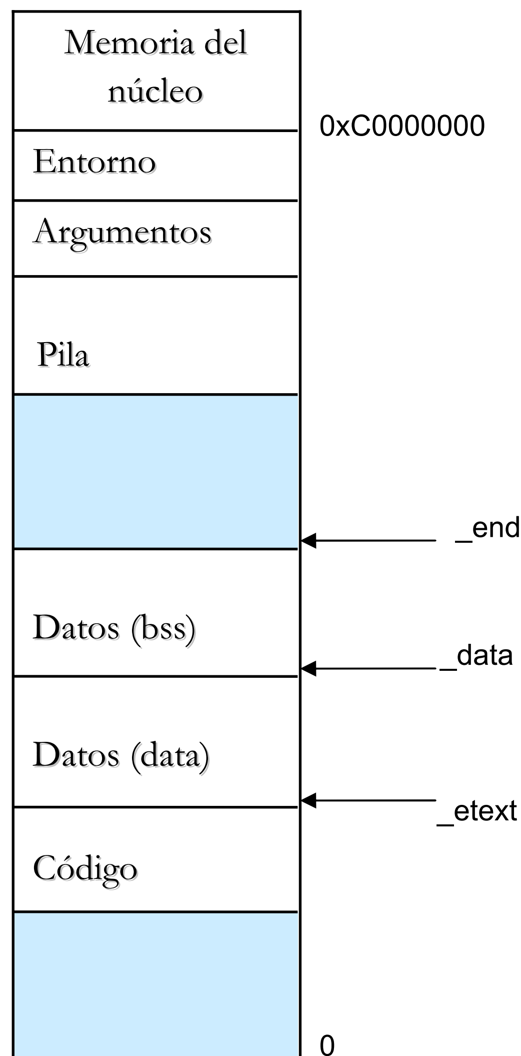
El subsistema de gestión de memoria permite que cada proceso del sistema se ejecute con una cantidad de memoria justa de toda la memoria física disponible, de forma que todos los procesos dispongan de los recursos que necesitan.

Memoria virtual compartida

Aunque la memoria virtual permite que cada proceso tenga un espacio de memoria separado (virtual), hay veces que es necesario que varios procesos compartan memoria. Por ejemplo puede haber varios procesos del sistema ejecutando el intérprete de órdenes “bash”. En lugar de tener varias copias del “bash”, una en cada memoria virtual de cada proceso, es mejor sólo tener una sola copia en memoria física y que todos los procesos que ejecuten bash la compartan. Las bibliotecas dinámicas son otro ejemplo típico de código ejecutable compartido por varios procesos.

11.2 Memoria Virtual

La memoria virtual de un proceso contiene el código ejecutable y datos de diversas fuentes.



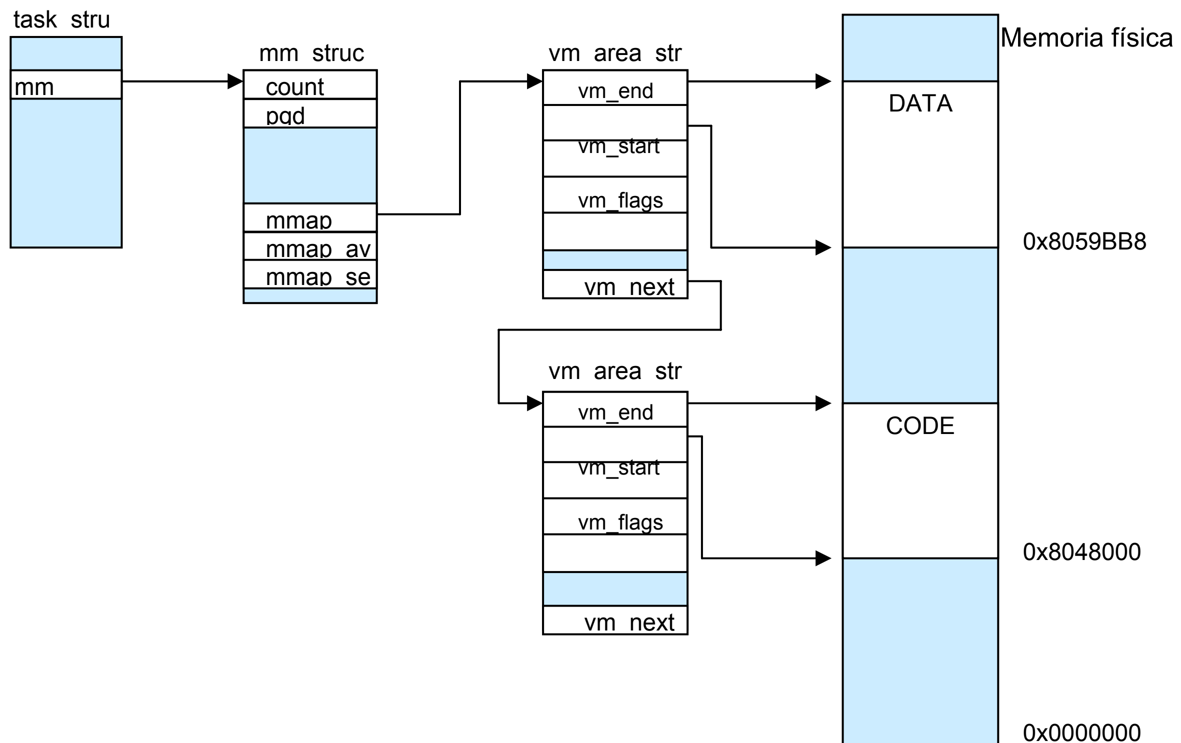
Estructura de un proceso en memoria

Primero se carga la imagen del programa; por ejemplo, una orden como *"ls"*. Este comando, como toda imagen ejecutable, se compone de código ejecutable y de datos. El fichero de imagen contiene toda la información necesaria para cargar el código ejecutable y los datos asociados con el programa en la memoria virtual del proceso.

Segundo, los procesos pueden reservar memoria (virtual) para usarla durante su procesamiento, por ejemplo para guardar los contenidos de los ficheros que esté leyendo. La nueva memoria virtual reservada tiene que asociarse con la memoria virtual que el proceso ya posee para poder usarla.

En tercer lugar, los procesos de Linux usan bibliotecas de código común, como por ejemplo rutinas de manejo de ficheros. No tendría sentido que cada proceso tenga su propia copia de la biblioteca, así pues Linux usa bibliotecas compartidas que varios procesos pueden usar al mismo tiempo. El código y los datos de estas bibliotecas compartidas tienen que estar unidos al espacio virtual de direccionamiento de un proceso y también al espacio virtual de direccionamiento de los otros procesos que comparten la biblioteca.

Un proceso no utiliza todo el código y datos contenidos en su memoria virtual dentro de un período de tiempo determinado. La memoria virtual del proceso puede que tenga código que sólo se usa en ciertas ocasiones, como en la inicialización o para procesar un evento particular. Puede que sólo haya usado unas pocas rutinas de sus bibliotecas compartidas. Sería superfluo cargar todo su código y datos en la memoria física donde podría terminar sin usarse. El sistema no funcionaría eficientemente si multiplicamos ese gasto de memoria por el número de procesos en el sistema. Para solventar el problema, Linux usa una técnica llamada *Paginación por Demanda* (*demand paging*) que sólo copia una página de memoria virtual de un proceso en la memoria física del sistema cuando el proceso trata de usarla. De esta manera, en vez de cargar el código y los datos en la memoria física de inmediato, el núcleo de Linux altera la tabla de páginas del proceso, designando las áreas virtuales como existentes, pero no en memoria. Cuando el proceso trata de acceder el código o los datos, el hardware del sistema generará una falta de página (*page fault*) y le pasará el control al núcleo para que arregle las cosas. Por lo tanto, por cada área de memoria virtual en el espacio de direccionamiento de un proceso, Linux necesita saber de dónde viene esa memoria virtual y cómo ponerla en memoria para arreglar los fallos de página.



La Memoria Virtual de un Proceso

El núcleo de Linux necesita gestionar todas estas áreas de memoria virtual, y el contenido de la memoria virtual de cada proceso se describe mediante una estructura mm_struct a la cual se apunta desde la estructura task_struct del proceso. La estructura mm_struct del proceso también contiene información sobre la imagen ejecutable cargada y un puntero a las tablas de páginas del proceso. Contiene punteros a una lista de estructuras vm_area_struct, cada una de las cuales representa un área de memoria virtual dentro del proceso.

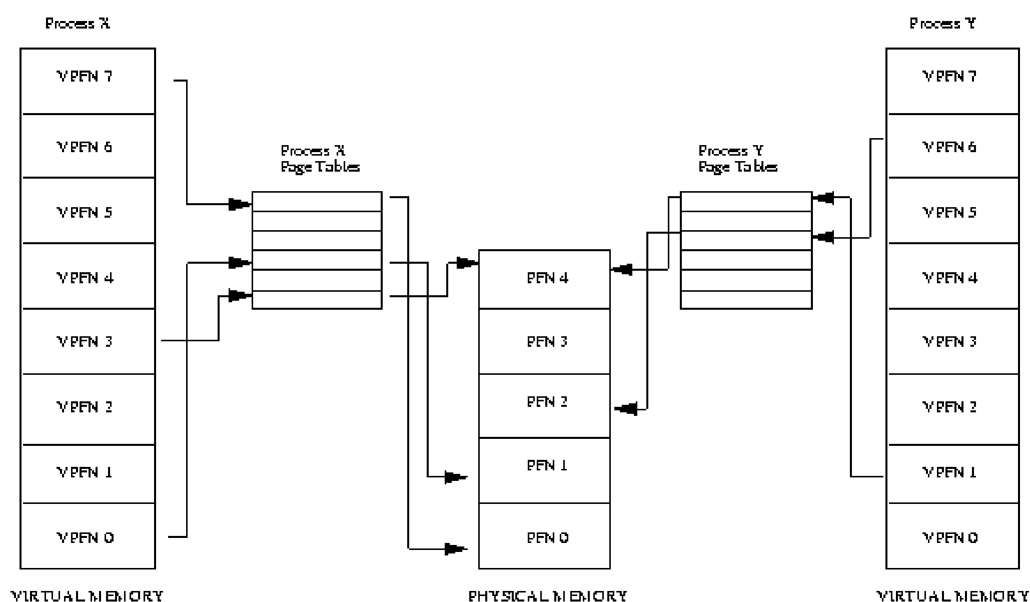
Esta lista enlazada está organizada en orden ascendente, la *figura 1* muestra la disposición en memoria virtual de un simple proceso junto con la estructura de datos del núcleo que lo gestiona. Como estas áreas de memoria virtual vienen de varias fuentes, Linux introduce un nivel de abstracción en la interfaz haciendo que la estructura vm_area_struct apunte a un grupo de rutinas de manejo de memoria virtual (via vm_ops).

De esta manera, toda la memoria virtual de un proceso se puede gestionar de una manera consistente sin que importe las diferentes maneras de gestionar esa memoria por parte de distintos servicios de gestión. Por ejemplo, hay una rutina que se utiliza cuando el proceso trata de acceder a la memoria y esta no existe, así es como se resuelven los fallos de página.

El núcleo de Linux accede repetidamente al grupo de estructuras vm_area_struct del proceso según crea nuevas áreas de memoria virtual para el proceso y según corrige las referencias a la memoria virtual que no está en la memoria física del sistema. Por esta razón, el tiempo que se tarda en encontrar la estructura vm_area_struct correcta es un punto crítico para el rendimiento del sistema. Para acelerar este acceso, Linux también organiza las estructuras vm_area_struct en un árbol Rojo-Negro, anteriormente en un árbol AVL (Adelson-Velskii and Landis).

Cuando un proceso reserva memoria virtual, en realidad Linux no reserva memoria física para el proceso. Lo que hace es describir la memoria virtual creando una nueva estructura vm_area_struct. Esta se une a la lista de memoria virtual del proceso. Cuando el proceso intenta escribir en una dirección virtual dentro de la nueva región de memoria virtual, el sistema creará un fallo de página. El procesador tratará de decodificar la dirección virtual, pero dado que no existe ninguna entrada de tabla de páginas para esta memoria, no lo intentará más, y creará una excepción de fallo de página, dejando al núcleo de Linux la tarea de reparar el fallo. Linux mira a ver si la dirección virtual que se trató de usar está en el espacio de direccionamiento virtual del proceso en curso. Si así es, Linux crea los PTEs (entrada en la tabla de páginas) apropiados y reserva una página de memoria física para este proceso. Puede que sea necesario cargar el código o los datos desde el sistema de ficheros o desde el disco de intercambio dentro de ese intervalo de memoria física. El proceso se puede reiniciar entonces a partir de la instrucción que causó el fallo de página y esta vez puede continuar dado que memoria física existe en esta ocasión.

11.3 Modelo Abstracto de Memoria Virtual



Modelo abstracto de traducción de memoria virtual a física.

Antes de considerar los métodos que Linux utiliza para implementar la memoria virtual, es interesante estudiar un modelo abstracto que no esté plagado de pequeños detalles de implementación.

Conforme el procesador va ejecutando un programa lee instrucciones de la memoria y las decodifica. Durante la decodificación de la instrucción puede necesitar cargar o guardar el contenido de una posición de memoria. El procesador ejecuta la instrucción y pasa a la siguiente instrucción del

programa. De esta forma el procesador está siempre accediendo a memoria tanto para leer instrucciones como para cargar o guardar datos.

En un sistema con memoria virtual, todas estas direcciones son direcciones virtuales y no direcciones físicas. Estas direcciones virtuales son convertidas a direcciones físicas por el procesador utilizando para ello información guardada en un conjunto de tablas mantenidas por el sistema operativo.

Para hacer la traducción más fácil, tanto la memoria virtual como la física están divididas en trozos de un tamaño manejable llamados *páginas*. Estas páginas son todas del mismo tamaño, en principio no necesitarían serlo pero de no serlo la administración del sistema se complicaría muchísimo. Linux en un sistema Alpha AXP utiliza páginas de 8 Kbytes, y en un sistema Intel x86 utiliza páginas de 4 Kbytes. La página física en Linux 2.6 puede tener un tamaño de 4MB o 2 MB.

Cada una de estas páginas tiene asociado un único número; el número de marco de página (PFN). En este modelo de paginación, una dirección virtual está compuesta de dos partes: un desplazamiento y un número de página virtual. Cada vez que el procesador encuentra una dirección virtual ha de extraer el desplazamiento y el número de marco de página. El procesador tiene que traducir el número de marco de la página virtual a la física y luego acceder a la posición correcta dentro de la página física. Para hacer todo esto, el procesador utiliza la *tabla de páginas*.

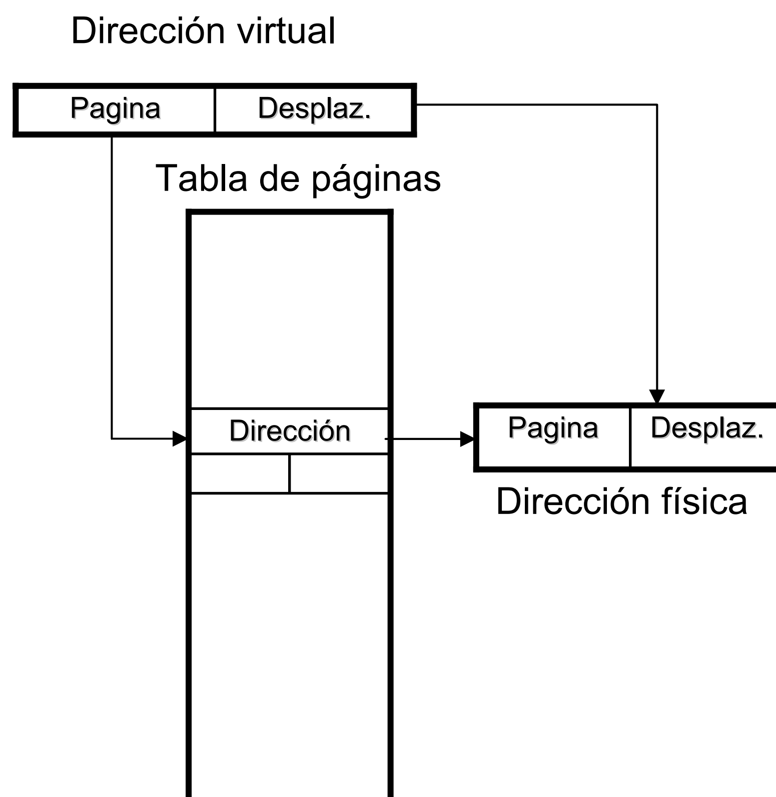


Tabla de páginas

El número de página se utiliza como índice de una **tabla de páginas** y el desplazamiento hace referencia a la posición que ocupa el dato dentro de la página en cuestión.

Todos los accesos a memoria se realizan a través de la tabla de páginas y cada proceso tiene una propia.

En la Figura 2 se muestra el espacio de direcciones virtuales de dos procesos, proceso *X* y proceso *Y*, cada uno con su propia tabla de páginas. Estas tablas de páginas asocian las páginas de memoria virtual de cada proceso sobre páginas físicas. Se puede ver que el número 0 de marco de página virtual del proceso *X* se asocia al número de marco de página físico 1 y que el número de marco de página virtual 1 de proceso *Y* está asociado en el número de marco de página físico 4. La tabla de páginas teórica contiene la siguiente información:

- Flag de Válido. Éste indica si la estrada de la tabla de páginas es válida o no,
- El número de marco de página físico que describe esta entrada,
- Información de control de acceso. Describe cómo se puede utilizar la página ¿Se puede leer? ¿Contiene código ejecutable?

A la tabla de páginas se accede utilizando el número de marco de página virtual como desplazamiento. El marco de página virtual 5 será el sexto elemento de la tabla (el primer elemento es el 0).

El procesador utiliza el número de marco de página virtual como índice a la tabla de páginas del proceso para obtener la entrada de la tabla de páginas correspondiente. Si la entrada en la tabla de páginas de ese índice es válida, el procesador coge el número de marco de página físico de esta entrada. Si por el contrario la entrada no es válida, entonces el proceso ha accedido a un área de memoria virtual que no existe. En este caso, el procesador no puede resolver la dirección y deberá pasar el control al sistema operativo para que éste solucione el problema.

La forma en la que el procesador informa al sistema operativo que un proceso concreto ha intentado realizar un acceso a una dirección virtual que no se ha podido traducir, es específico del procesador. Independientemente del mecanismo empleado por el procesador, a esto se le conoce como ***fallo de página*** y el sistema operativo es informado de la dirección virtual que lo ha producido y la razón por la que se produjo.

Suponiendo que ésta sea una entrada válida en la tabla de páginas, el procesador toma el número de marco de página físico y lo multiplica por el tamaño de página para obtener la dirección base de la página física en memoria. Finalmente, el procesador le suma el desplazamiento a la instrucción o dato que necesita, y ésta es la dirección física con la que accede a memoria.

Asignando direcciones virtuales a direcciones físicas de esta forma, la memoria virtual se puede asociar sobre las páginas físicas en cualquier orden. Por ejemplo, en la Figura 2 el número de marco de página 0 del proceso *X* está asociado sobre el número de marco de página física 1, mientras que el número de marco 7 está asociado con el marco físico número 0, a pesar de ser una dirección de memoria virtual mayor que la del marco de página 0. Esto demuestra un interesante efecto lateral de la memoria virtual: las páginas de memoria virtual no tienen que estar necesariamente presentes en memoria física en un orden determinado.

Esta es la estructura de una página:

```
  39struct page {
  40        unsigned long flags; /*Estado de la página
  42        atomic_t _count; /*Números de referencia a la página
  43        union {
  44                atomic_t _mapcount; /*Números de entrada a tablas
  48                struct {
  49                        u16 inuse; /*Número de objetos
  50                        u16 objects;
  51                };
  52        };
  53        union {
  54            struct {
  55                   /*Cuando la página está libre indica el orden en el 
buddy system
  56                unsigned long private;
  61                   /*Si no es NULL apunta a inode, si no apunta a 
anon_vma
  62                struct address_space *mapping;

  69            };
  70#if USE_SPLIT_PTLOCKS
  71            spinlock_t ptl;
  72#endif
  73            struct kmem_cache *slab;
  74            struct page *first_page;
  75        };
  76        union {
  77                pgoff_t index;/*Desplazamiento dentro del  mapeado
  78                void *freelist;
  79        };
  80        struct list_head lru;/*Lista de páginas a desalojar mediante 
algoritmo LRU

  93#if defined(WANT_PAGE_VIRTUAL)
  94        void *virtual;
  96#endif /* WANT_PAGE_VIRTUAL */
  97};
```

Y a continuación se muestran los diferentes campos de la estructura y su significado:

TIPO	**CAMPO**	*DESCRIPCIÓN*
unsigned_long	flags	Estado de la página
atomic_t;	_count	Números de referencia a la página
atomic_t	_mapcount	Contador de las entradas a la tabla de páginas. Limita la búsqueda reverse mapping
Unsigned int	Inuse	Número de objetos

unsigned long	private	Cuando la pagina esta libre indica el orden en el buddy system.
struct address_space	***mapping**	Si bit menos significativo esta a 0 o NULL es un puntero a inode, sino es un puntero a anon_vma (memoria anónima) object. Esta estructura contiene métodos como set_page_dirty

El campo flags indica el estado que puede tener una página, estos estados se describen a continuación:

CONSTANTE (*FLAGS*)	**VALOR**	**SIGNIFICADO**
PG_locked	0	La página está bloqueada en memoria
PG_error	1	Se ha producido un error en la carga de la página en memoria
PG_referenced	2	La página ha sido accedida
PG_uptodate	3	El contenido de la página está actualizado
PG_dirty	4	Indica si el contenido de la página se ha modificado
PG_lru	5	La página está en la lista de páginas activas e inactivas
PG_active	6	La página está en la lista de páginas activas
PG_slab	7	El marco de página está incluido en un slab
PG_owner_priv_1	8	
PG_arch_1	9	
PG_reserved	10	La página está reservada para un uso futuro, no es posible acceder a ella.
PG_private	11	Contiene datos privados (something 10rívate)
PG_writeback	12	Se indica que la pagina usa el método writeback
PG_compound	14	
PG_swapcache	15	Indica si la pagina esta intercambiada
PG_mappedtodisk	16	Indica si la página está intercambiada
PG_reclaim	17	Página marcada para escribirse a disco
PG_buddy	19	Página libre en listas buddy

11.3.1 Paginación por Demanda

Puesto que hay mucha menos memoria física que memoria virtual, el sistema operativo ha de tener especial cuidado de no hacer un mal uso de la memoria física. Una forma de conservar memoria física es cargar sólo las páginas que están siendo utilizadas por un programa. Por ejemplo, un programa de bases de datos puede ser ejecutado para realizar una consulta a una base de datos. En este caso no es necesario cargar en memoria toda la base de datos, sino sólo aquellos registros que son examinados. Si la consulta consiste en realizar una búsqueda, entonces no tiene sentido cargar el fragmento de programa que se ocupa de añadir nuevos registros. Esta técnica de cargar sólo páginas virtuales en memoria conforme son accedidas es conocida como *Paginación por Demanda*.

Memoria

página

El espacio de cada región de memoria se organiza en páginas para un mejor manejo de la información.

Cada página es de tamaño fijo.

Cuando un proceso intenta acceder a una dirección virtual que no está en esos momentos en memoria, el procesador no puede encontrar la entrada en la tabla de páginas de la página virtual referenciada. Por ejemplo, en la Figura 2 no existe una entrada en la tabla de páginas del proceso *X* para el marco número 2, por lo que si el proceso *X* intenta leer de una dirección perteneciente al marco de página virtual 2 no podrá traducirla a una dirección física. Es en este momento cuando el procesador informa al sistema operativo que se ha producido un fallo de página.

Si dirección virtual que ha fallado es inválida, significa que el proceso ha intentado acceder a una dirección que no debería. Puede ser que la aplicación haya hecho algo erróneo, por ejemplo escribir en una posición aleatoria de memoria. En este caso, el sistema operativo ha de terminarlo, protegiendo así a otros procesos de este “perverso” proceso.

Si la dirección virtual que ha producido el fallo era válida pero la página que referencia no está en memoria en ese momento, el sistema operativo tiene que traer la página apropiada a memoria desde el disco. Los accesos a disco requieren mucho tiempo, en términos relativos, y por tanto el proceso tiene que esperar cierto tiempo hasta que la página se haya leído. Si hay otros procesos que pueden ejecutarse entonces el sistema operativo elegirá alguno de éstos para ejecutar. La página pedida se escribe en una página física libre y se añade una entrada a la tabla de páginas del proceso para esta página. El proceso en entonces se pone otra vez en ejecución justo en la instrucción donde se produjo el fallo de página. Esta vez sí que se realizará con éxito el acceso a la dirección de memoria virtual, el procesador puede hacer la traducción de dirección virtual a física y el proceso continua normalmente.

Linux utiliza la paginación por demanda para cargar imágenes ejecutables en la memoria virtual de un proceso. Siempre que se ejecuta un proceso, se abre el fichero que la contiene y su contenido se asocia en la memoria virtual del proceso. Esto se hace modificando las estructuras de datos que describen el mapa de memoria del proceso y se conoce como *asociación de memoria*. Sin embargo, sólo la primera parte de la imagen se copia realmente en memoria física. El resto de la imagen se deja en disco. Conforme se va ejecutando, se generan fallos de página y Linux utiliza el mapa de memoria del proceso para determinar qué partes de la imagen ha de traer a memoria para ser ejecutadas.

11.3.2 Intercambio (swapping)

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 fichero imagen o de un fichero 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 fichero imagen o de datos.

Por otra parte, si la página había sido modificada, el sistema operativo debe preservar su contenido para que pueda volver a ser accedido. Este tipo de página se conoce como página *sucia* (dirty page) y para poderla eliminar de memoria se ha de guardar en un fichero especial llamado fichero de intercambio (swap file). El tiempo de acceso al fichero de intercambio es muy grande en relación a la velocidad del procesador y la memoria física, por lo que 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 (el algoritmo de intercambio) no es eficiente, entonces se produce una situación llamada *hiper-paginación* (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, el número de marco de página 1 de la Figura 2 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 paginación por antigüedad (LRU Last Redently Used) 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 vieja e inútil. Las páginas viejas son las mejoras candidatas para ser intercambiadas (Principio de Localidad Temporal).

11.3.3 Memoria virtual compartida

Gracias a los mecanismos de memoria virtual se puede conseguir fácilmente que varios procesos compartan memoria. Todos los accesos a memoria se realizan a través de las tablas de páginas y cada proceso tiene su propia tabla de páginas. Para que dos procesos compartan una misma página de memoria física, el número de marco de esta página ha de aparecer en las dos tablas de páginas.

La Figura 2 muestra dos procesos que comparten el marco de página física número 4. Para el proceso *X* esta página representa el su marco de página virtual número cuatro, mientras que para el proceso *Y* es el número 6. Esto ilustra una interesante cuestión que aparece al compartir páginas: la memoria física compartida no tiene porqué estar en las mismas direcciones sobre memoria virtual en todos los procesos que la comparten.

11.3.4 Modos de direccionamiento físico y virtual

No tiene mucho sentido que el propio sistema operativo se ejecute sobre memoria virtual. Sería una verdadera pesadilla que el sistema operativo tuviera que mantener tablas de páginas para él mismo. La mayoría de los procesadores de propósito general ofrecen la posibilidad del modo de direccionamiento físico junto con direccionamiento virtual. El modo de direccionamiento físico no necesita las tablas de páginas y el procesador no intenta realizar ningún tipo de traduccines en este modo. El núcleo de Linux está preparado para funcionar sobre un espacio de direccionamiento físico.

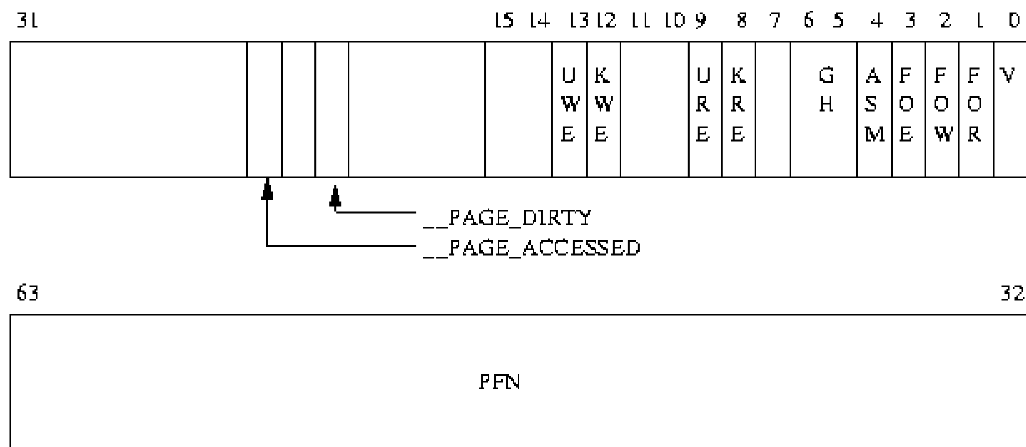
El procesador Alpha AXP no tiene un modo especial de direccionamiento físico. Lo que hace es dividir el espacio de memoria en varias áreas y designa a dos de ellas como direcciones físicas. A este espacio de direcciones del núcleo se le conoce como espacio de direcciones KSEG y contiene todas las direcciones a partir de la *0xfffffc0000000000*. Para poder ejecutar código enlazado en KSEG (por definición, el código del núcleo) o acceder a datos de esta zona, el código debe ejecutarse en modo núcleo. El núcleo de Linux en Alpha está enlazado para ejecutarse a partir de la dirección *0xfffffc0000310000*.

11.3.5 Control de acceso

Las entradas de la tabla de páginas también contienen información relativa al control de acceso. Puesto que el procesador tiene que utilizar la información de la tabla de páginas para traducir las direcciones virtuales a direcciones físicas, puede fácilmente utilizar la información de control de acceso para comprobar que el proceso no está accediendo a memoria de forma apropiada.

Hay muchas razones por las que se puede querer restringir el acceso a determinadas áreas de memoria. Hay memoria, como la que contiene el ejecutable, que es claramente memoria de sólo lectura; el sistema operativo no ha de dejar que el proceso escriba sobre su propio código de programa. Por el contrario, páginas de memoria que contengan datos han de poder ser leídas y escritas, pero un intento de ejecutar algo de estas páginas ha de fallar. La mayoría de los procesadores tienen al menos dos modos de operación: *modo núcleo* y *modo usuario*. No es

deseable que un proceso ejecute código del núcleo, o que acceda a datos del núcleo excepto cuando el procesador está funcionando en modo núcleo.



Entrada de tabla de páginas del Alpha AXP

La información de control de acceso se guarda en la PTE y es dependiente el procesador; la figura muestra el PTE del **Alpha AXP**. Los bits que aparecen tienen el siguiente significado:

V

Valido, si está activado, la PTE es válida;

FOE

"Fault on Execute", cuando se intente ejecutar una instrucción de esta página, el procesador informará de un fallo de página y dará el control al sistema operativo;

FOW

"Fault on Write" como el anterior pero el fallo de página se produce cuando se intenta escribir sobre alguna posición de memoria de la página.

FOS

"Fault on Read" como el anterior pero el fallo de página se produce cuando se intenta leer de esta página.

ASM

"Address Space Match". Este se utiliza cuando el sistema operativo quiere eliminar sólo algunas de las entradas de la Tabla de Traducción (Translation Table);

KER

Solo el código ejecutado en modo núcleo puede leer esta página;

URE

Código ejecutado en modo usuario puede leer esta página;

GH

"Granularity Hint" utilizado para asociar un bloque completo con una sola entrada del buffer de traducción en lugar de con varias;

KWE

Código ejecutado en modo núcleo puede escribir en esta página;

UWE

Código ejecutado en modo usuario puede escribir en esta página;

PFN

Para páginas con el bit **V** activado, este campo contiene la dirección física del número de marco de página para esta PTE. Para PTE inválidas, si el campo es distinto de cero, contiene información acerca de dónde está la página en memoria secundaria.

Los siguientes dos bits los define y utiliza Linux:

_PAGE_DIRTY

Si está activado, la página necesita ser copiada a disco.

_PAGE_ACCESSED

Utilizada por Linux para marcar una página como accedida.

En un procesador x86 una entrada de la tabla de páginas contiene además de la base de la dirección de una página algunos flags que indican entre otras cosas que operaciones se pueden aplicar sobre la página. En una arquitectura de 32 bits como x86 sólo son necesarios los 20 bits de mayor peso de los 32 bits que componen una dirección. Esto es debido a que las direcciones de páginas están alineadas al tamaño de página (2^{12} = 4Kbs). Por tanto los 12 bits restantes se utilizan para almacenar información importante acerca de cada página. Los flags que se muestran a continuación representan estos 12 bits.

_PAGE_PRESENT

Activo si la página está físicamente en memoria.

_PAGE_RW

0 si la página es de sólo lectura y 1 si es de lectura-escritura (no existen páginas sólo de escritura).

_PAGE_USER

Activo si la página pertenece al espacio de usuario e inactivo si pertenece al núcleo.

_PAGE_WT

Indica la política de caché de la página. Este bit no es utilizado por el núcleo.

- *writethought (0)*: Actualiza inmediatamente los datos escritos en la caché en memoria principal.
- *writeback (1)*: Sólo se actualizan los cambios escritos en la memoria caché en la memoria principal cuando la línea de caché va a ser sustituida.

_PAGE_PCD

Activa o desactiva el uso de la caché para esa página.

_PAGE_ACCESSED

Activo si la página ha sido accedida recientemente.

_PAGE_DIRTY

Activo si la página ha sido escrita.

_PAGE_PROTNONE

Este bit está actualmente en desuso.

_PAGE_4M y _PAGE_GLOBAL

Estos bits no son usados en el nivel de protección de página, por tanto, son ignorados.

11.4 Cachés

Si implementáramos un sistema utilizando el modelo teórico que acabamos de describir, obviamente funcionaria, pero no sería particularmente eficiente. Tanto los diseñadores de sistemas operativos como los de procesadores, se esfuerzan al máximo para obtener el mayor rendimiento posible del sistema. Además de hacer los procesadores, la memoria y otros dispositivos más rápidos la mejor forma de obtener un buen rendimiento consiste en mantener en memoria caches de la información que se utiliza muy a menudo. Linux emplea unas cuantas caches para la gestión de la memoria:

Buffer Cache

Contiene buffers de datos que son utilizados por los manejadores 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 cache buffer 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 cache. Si un dato (bloque) se puede encontrar en el buffer cache, entonces no es necesario leerlo del dispositivo de bloques físico, por el disco duro, y por tanto el acceso es mucho más rápido.

Cache de Páginas

Este se utiliza para acelerar el acceso a imágenes y datos en disco. Se utiliza para guardar el contenido lógico de un fichero de página en página y se accede vía el fichero y el desplazamiento dentro del fichero. Conforme las páginas se leen en memoria, se almacenan en la page cache.

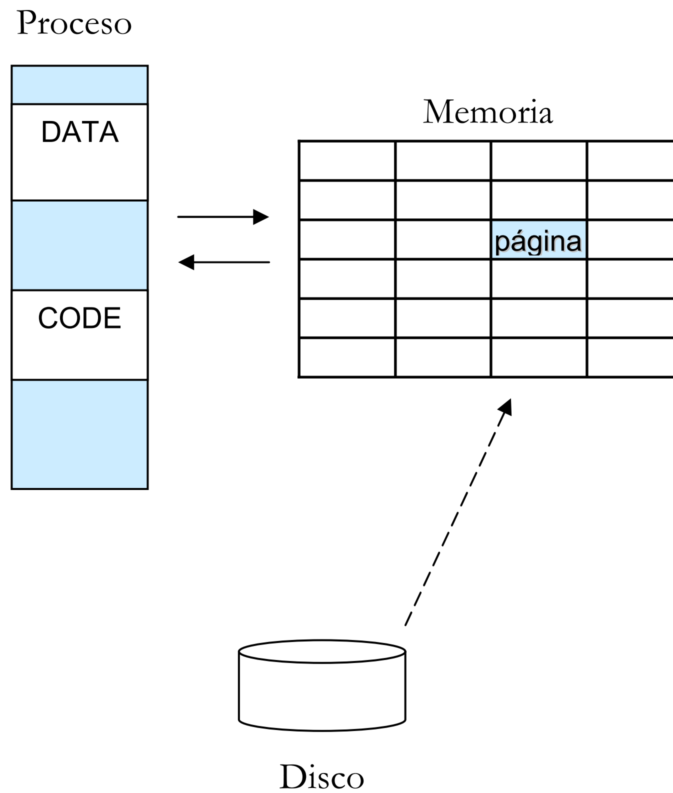


Figura 6: Cache de páginas

Cache de Intercambio (swap)

Solo las páginas que han sido modificadas (dirty) son guardadas en el fichero de intercambio. Mientras no vuelvan a ser modificadas después de haber sido guardadas en el fichero de swap, la próxima vez que necesiten ser descartadas (swap out) no será necesario copiarlas al fichero 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 una cache normalmente implementada en el propio procesador; la cache de entradas de tabla de página. En este caso, el procesador no necesita siempre leer la tabla de páginas directamente, sino que guarda en esta cache las traducciones de las páginas conforme las va

necesitando. Estos son los Translation Look-aside 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 TPE 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 (en el x86 un fallo de una TLB no produce una invocación al S.O., sino que el x86 accede a memoria principal para buscar la entrada en la tabla correspondiente. Si al acceder a memoria principal no encuentra una entrada válida, entonces sí se produce un Fallo de Página que es enviado al S.O. (mediante una interrupción)). 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 cache, 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.

11.5 Tablas de Páginas en Linux

Es necesario mantener la **tabla de páginas** en **memoria**, sin embargo, dado el espacio de direccionamiento de cada proceso sería imposible.

Otro problema que surge con este tipo de tablas tiene que ver con la velocidad. Si la tabla de un proceso es muy grande, acceder a ella ralentizaría la ejecución del mismo.

Como solución, Linux descompone **la tabla de páginas** original en distintos niveles. Este conjunto de tablas se denomina directorio de páginas.

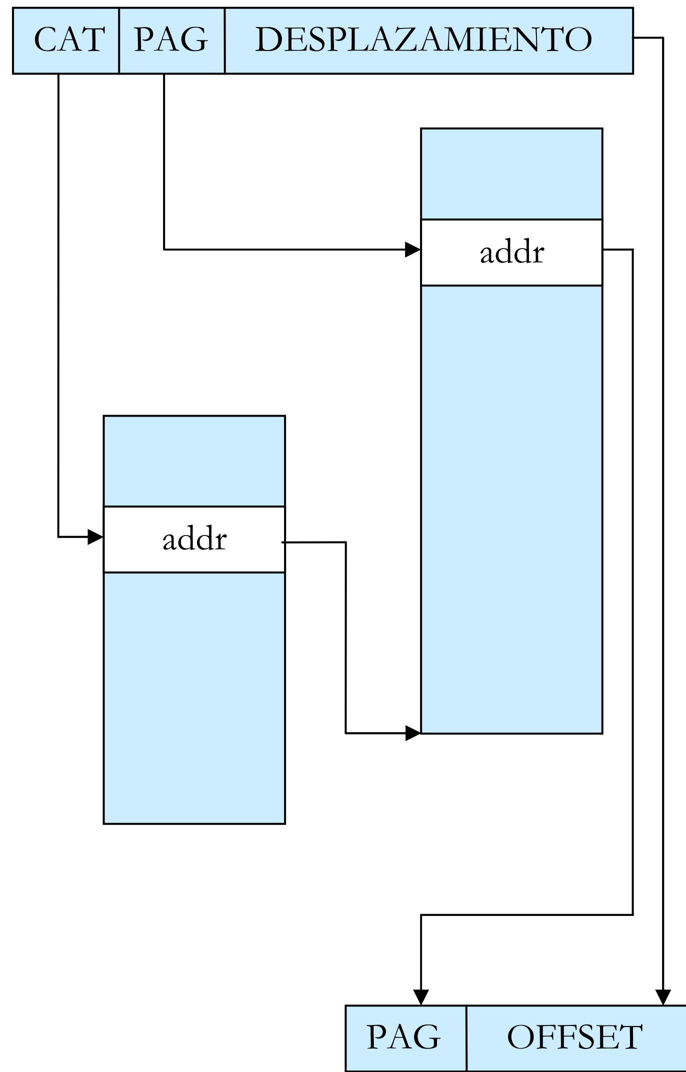
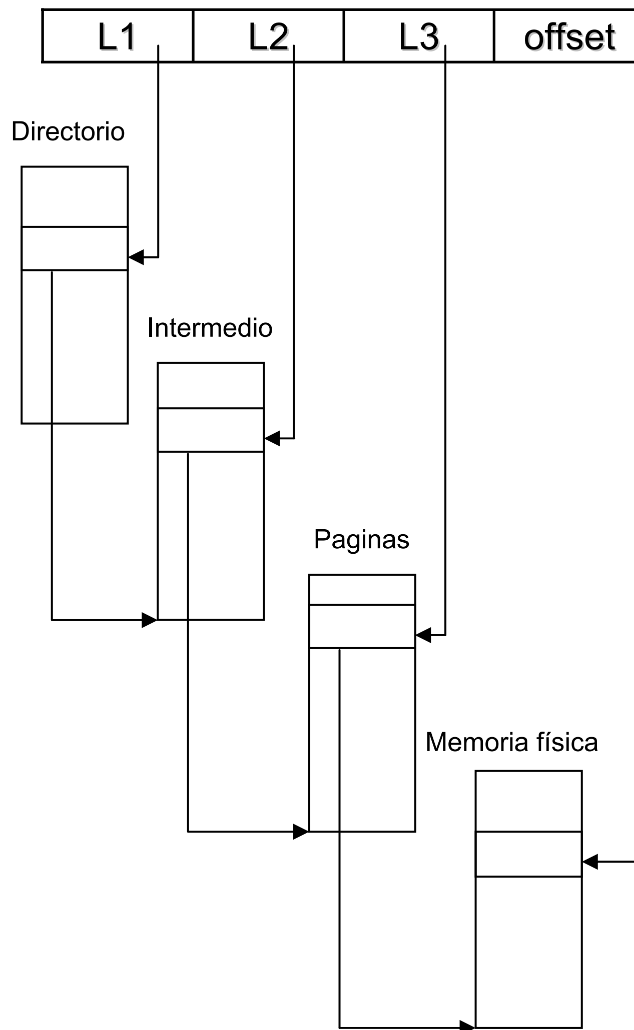


Tabla de páginas

Linux supone que hay tres niveles de tablas de páginas:

- El directorio de tablas de páginas (punteros a tablas intermedias).
- La tabla de páginas intermedia (punteros a tablas de páginas).
- Las tablas de páginas (punteros a páginas).



Tablas de páginas de tres niveles

Cada nivel de tablas contiene el número de marco de página del siguiente nivel en la tabla de páginas. La figura 4 muestra cómo una dirección virtual se divide en un número de campos, donde cada uno de ellos representa un desplazamiento dentro de una tabla de páginas. Para traducir una dirección virtual a una física, el procesador tiene que tomar el contenido de cada uno de estos campos, convertirlos en desplazamientos de la página física que contiene la tabla de páginas y leer el número de marco de página del siguiente nivel de la tabla de páginas. Esta operación se repite tres veces hasta que se encuentra el número de la página física que contiene la dirección virtual. Ahora el último campo de la dirección virtual se utiliza para encontrar el dato dentro de la página.

Las siguientes estructuras de datos muestran los 3 niveles de tablas de páginas que existen, además en esta última versión se ha añadido una nueva característica, y es el poder marcar una página como protegida:

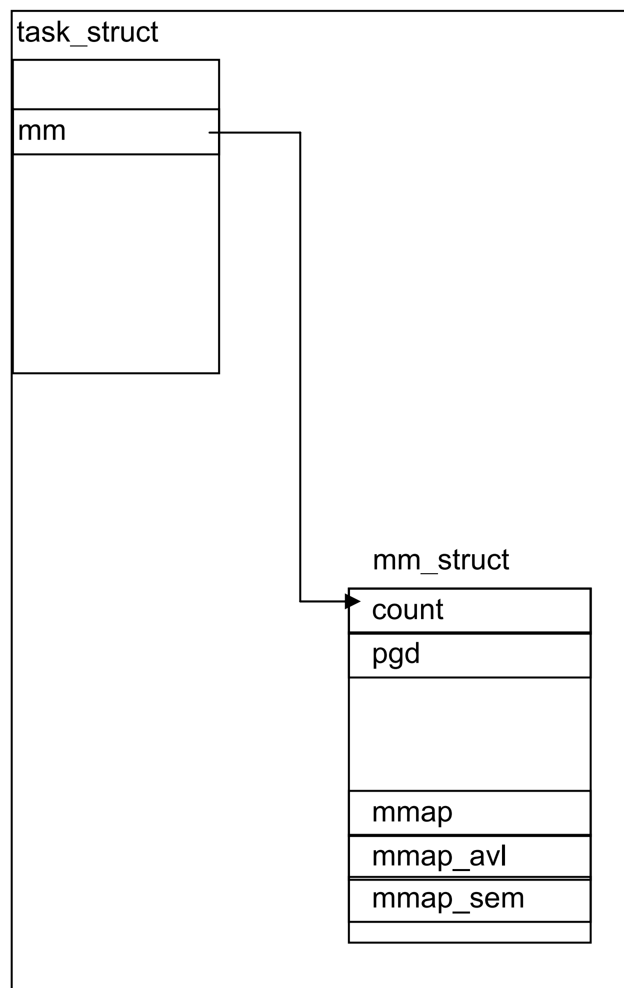
```
typedef struct { unsigned long pte_low, pte_high; } pte_t;

typedef struct { unsigned long long pmd; } pmd_t;

typedef struct { unsigned long long pgd; } pgd_t;

typedef struct { unsigned long long pgprot; } pgprot_t;
```

Cada proceso (*task_struct*) tiene un campo ***mm*** de tipo *mm_struct* que almacena la información de su espacio de direcciones. El campo pgd mantiene el directorio de páginas utilizado para resolver la dirección física dada una dirección virtual.



Estructura de mm_struct

11.6 Asignación y liberación de páginas

Se producen muchas peticiones de páginas físicas. Por ejemplo, cuando una imagen se carga a memoria, el sistema operativo necesita asignar páginas. Éstas serán liberadas cuando la imagen concluya su ejecución y se descargue. Otro uso de páginas físicas es para contener estructuras de datos específicas tales como las propias tablas de páginas. Los programas y las estructuras de datos relacionados con la asignación y liberación de páginas son quizás los más críticos para obtener un subsistema de memoria virtual eficiente.

1. El kernel puede solicitar memoria de tres formas:
   - Directamente al Buddy System, para asignaciones genéricas de grupos de marcos de página contiguos y potencia de 2.
   - Al Slab Allocator (asignación de memoria por fragmentos dentro de una página, para evitar así la fragmentación interna), para objetos frecuentemente usados (kmalloc())
   - Usando vmalloc() para obtener áreas de memoria virtual contigua sin garantía de que también lo sea físicamente
2. A los procesos de usuario no se les asigna realmente páginas, sino áreas de memoria (memory descriptor (mm_struct) + memory areas (vm_area_struct)), en otras palabras, se les da rangos de direcciones lineales válidos que se asignarán en el momento en que se vayan a usar

11.6.1 Buddy System

El objetivo en cualquier sistema de memoria es evitar la fragmentación externa. La fragmentación externa es un fenómeno que se produce cuando los procesos asignados han ocupado posiciones no contiguas de memoria dejando demasiados bloques libres de pequeño tamaño, en los que no "caben" nuevos procesos.

Para evitar esto existen dos posibilidades:

- Utilizar la unidad de paginación para agrupar marcos de página dispersos en direcciones lineales contiguas.
- Desarrollar un sistema que controle los marcos de página contiguos y evite en lo posible romper un bloque grande para una asignación pequeña.

El kernel de Linux opta por la segunda opción a través del Buddy por las siguientes razones:

- En ocasiones es necesario que los marcos de páginas sean contiguos, ya que direcciones lineales contiguas no son suficientes para satisfacer la petición (por ejemplo, un buffer asignado a DMA).

- No se tocan las tablas de páginas del kernel, por lo que la CPU no tiene que limpiar el contenido de los TLBs (Translation Lookaside Buffers).

- Fragmentos grandes de memoria física contigua pueden ser accedidas por el kernel mediante páginas de 4 Kbytes.

Linux utiliza el algoritmo Buddy para asignar y liberar eficientemente bloques de páginas. El código de asignación intenta asignar un bloque de una o más páginas físicas. Las páginas se asignan en bloques de tamaño potencia de 2. Esto quiere decir que pueden asignar bloques de 1, 2, 4, etc páginas.

El **tamaño** de los **bloques** es **fijo**, siendo asignados un número de páginas igual a potencias de 2.

- 0 Grupos de 1 página
- 1 Grupos de 2 páginas

...

- 5 Grupos de 32 páginas

Cada bloque hace referencia a páginas contiguas en memoria.

Mientras haya suficientes páginas libres en el sistema para satisfacer esta petición (nr_free_pages, min_free_pages) el código de asignación buscará en la zona bloques de páginas del tamaño pedido.

El bloque se divide en 2 partes:

- tantas páginas como tamaño de memoria especificado.
- resto de páginas que continúan disponibles

Así, cada elemento de la zona tiene un mapa de bloques de páginas asignados y libres para ese tamaño de bloque. Por ejemplo, el elemento 3 del vector tiene un mapa de memoria que describe los bloques de 4 páginas libres y asignados.

Las páginas que no son utilizadas se insertan en las otras listas. Dentro de cada lista, se comprobaría si el grupo de páginas adyacentes se encuentra disponible, en ese caso se fusionan los grupos y pasan a la lista de grupos de tamaño inmediatamente superior. Así sucesivamente hasta llegar al tope.

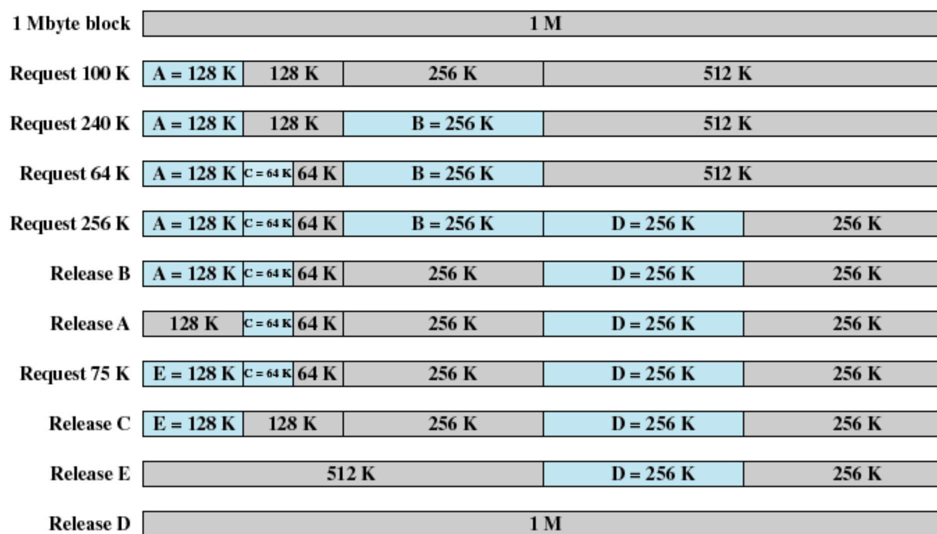
Este proceso se repite si en vez de ser páginas que sobran de una asignación, son páginas que han sido liberadas.

El algoritmo de asignación busca primero entre los bloques de páginas de igual tamaño que el pedido. Luego sigue la lista de páginas libres que está encolada en el elemento *free_list* de la estructura de datos free_area. Si no encuentra ningún bloque de páginas del tamaño pedido libre, entonces busca en los siguientes (los cuales son del doble del tamaño pedido). Este proceso continua hasta que bien todos los elementos de la zona han sido examinados o bien se he encontrado un bloque de páginas libres. Si el bloque de páginas encontrado es mayor que el pedido, entonces se trocea hasta conseguir un bloque del tamaño deseado. Puesto que el número de páginas de cada bloque es potencia de 2, simplemente dividiendo el bloque por la mitad se obtienen dos bloques con un tamaño de bloque inmediatamente inferior. Los bloques libres se insertan en la cola apropiada y el bloque de páginas asignado se devuelve al que realizó la llamada.

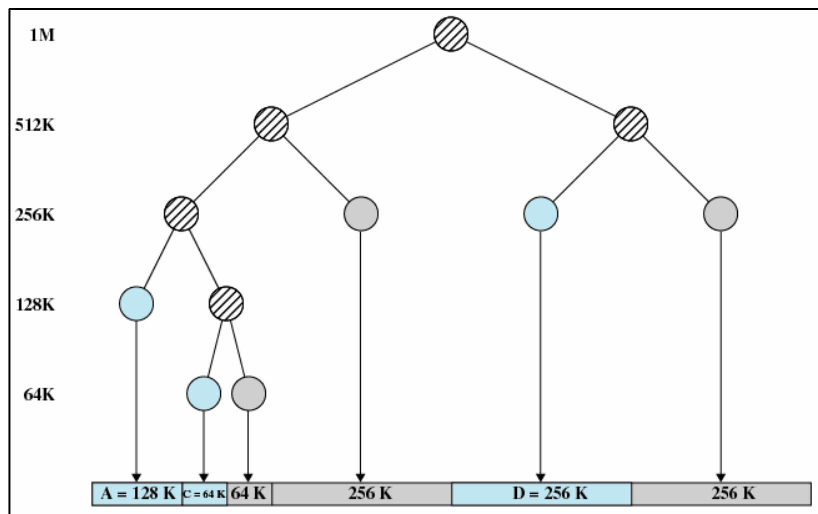
Asignar bloques de páginas tiende a fragmentar la memoria al dividir los bloques grandes para conseguir bloques más pequeños. El código de liberación de páginas recombina páginas en bloques de mayor tamaño siempre que es posible. De hecho, el tamaño de bloque de página es importante pues facilita la recombinación en bloques grandes.

Siempre que se libera un bloque de páginas, se comprueba si está libre el bloque adyacente de igual tamaño. Si es así, se combina con el bloque de páginas recién liberado para formar un bloque nuevo de tamaño doble. Cada vez que dos bloques de páginas se recombinan en uno mayor, el algoritmo de liberación intenta volver a recombinarlo en otro aún mayor. De esta forma, los bloques de páginas libres son tan grandes como la utilización de la memoria permitida.

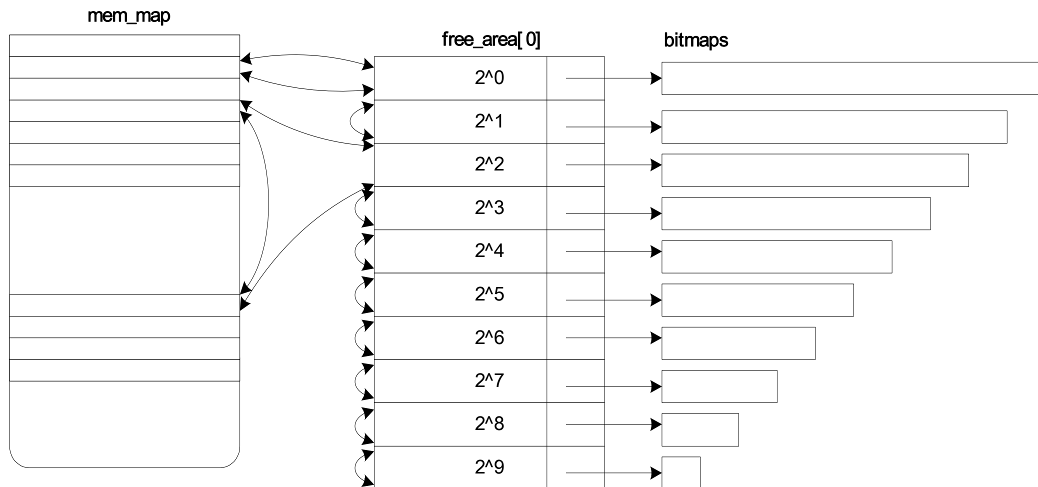
Ejemplo de funcionamiento del Buddy System



Representación en árbol del Buddy System



Estructuras de datos del Buddy System



El vector free_area es utilizado por el código de asignación de páginas para contrar páginas libres. Cada elemento de free_area contiene información sobre bloques de páginas. El primer elemento del vector describe páginas simples, el siguiente bloques de 2 páginas, el siguiente bloques de 4 páginas y así creciendo en potencias de 2. El elemento free_list se utiliza como cabeza de una cola que contiene punteros a la estructura page.

```
58struct free_area {
59         struct list_head free_list[MIGRATE_TYPES];
60         unsigned long nr_free;
61};
```

Como podemos observar en la estructura free_area contiene un campo que es una lista que contiene a su vez punteros al siguiente y al anterior.

```
567struct zonelist {
568        struct zonelist_cache *zlcache_ptr;
569        struct zoneref _zonerefs[MAX_ZONES_PER_ZONELIST + 1];
570#ifdef CONFIG_NUMA
571        struct zonelist_cache zlcache;
572#endif
573};
```

Existe una estructura llamada zonelist, la cual contiene un vector de zonas, y cada zona contiene un vector de free_area de tamaño definido por la macro MAX_ORDER.

```
266struct zone {
267 /*Páginas reservadas para la zona (pages_min) y umbrales(pages_low,
pages_high)*/
268        unsigned long           pages_min, pages_low, pages_high;
277        unsigned long           lowmem_reserve[MAX_NR_ZONES];
279#ifdef CONFIG_NUMA
280        int node;
284        unsigned long           min_unmapped_pages;
285        unsigned long           min_slab_pages;
286        struct per_cpu_pageset  *pageset[NR_CPUS]; /*Caches especiales
para marcos de página
287#else
288        struct per_cpu_pageset  pageset[NR_CPUS];
289#endif
293        spinlock_t              lock;
294#ifdef CONFIG_MEMORY_HOTPLUG
296        seqlock_t               span_seqlock;
297#endif
298        struct free_area        free_area[MAX_ORDER]; /*Bloques de marcos
de páginas libres en la zona
299
300#ifndef CONFIG_SPARSEMEM
305        unsigned long           *pageblock_flags;
306#endif /* CONFIG_SPARSEMEM */
309        ZONE_PADDING(_pad1_)
312        spinlock_t              lru_lock;
313        struct {
314                struct list_head list;
315                unsigned long nr_scan;
316        } lru[NR_LRU_LISTS];
326        unsigned long           recent_rotated[2];
327        unsigned long           recent_scanned[2];
329        unsigned long           pages_scanned;
330        unsigned long           flags;
333        atomic_long_t           vm_stat[NR_VM_ZONE_STAT_ITEMS];
344        int prev_priority;
354        unsigned int inactive_ratio;
355
357        ZONE_PADDING(_pad2_)
384        wait_queue_head_t       * wait_table;
385        unsigned long           wait_table_hash_nr_entries;
386        unsigned long           wait_table_bits;
391        struct pglist_data      *zone_pgdat;
392        /* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
393        unsigned long           zone_start_pfn;
405        unsigned long           spanned_pages; /* total size, including
holes */
406        unsigned long           present_pages; /* amount of memory
(excluding holes) */
411        const char              *name;
412} ____cacheline_internodealigned_in_smp;
```

11.6.2 Slab Allocator

Es un sistema superpuesto al Buddy System para mejor aprovechamiento de la memoria ya que el Buddy System es ineficiente para asignaciones de pocos bytes, se desperdician grandes cantidades de memoria esto se debe

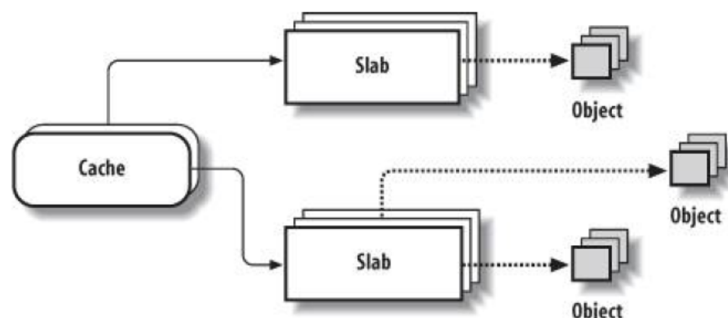
En el Linux 2.0:

- Se definen 13 listas de áreas de memoria donde guardar objetos de tamaños geométricamente distribuidos, de 32 a 131056 bytes.
- Se emplea el Buddy System tanto para asignar nuevas páginas para estas áreas como para liberar las que no son necesarias. Las áreas siempre ocupan páginas enteras.
- Dentro de cada marco de página se emplea una lista dinámica para gestionar los elementos libres.
- Por fragmentación interna se pierde aprox. un 50% de la memoria.

A partir de Linux 2.2:

- Aplica algoritmo derivado del Slab Allocator de Solaris 2.4 simplificado.
- Objetivos: Acelerar las asignaciones/liberaciones de objetos y evitar fragmentación interna.
- Se basa en que el kernel solicita y libera frecuentemente áreas de memoria pequeñas y de estructura consistente como descriptores de proceso, de memoria, inodes, etc.
- Se ven las áreas de memoria que pide el sistema como objetos que consisten en estructuras de datos y algunas funciones.
- Se mantienen una serie de **caches** que gestionan slabs de objetos del mismo tipo. Habrá tantas caches como tipos de objetos específicos interese gestionar, y una cache mantiene varios slabs.
- Un slab es una serie de marcos de página contiguos en los que se almacenan los objetos del mismo tipo.

Componentes y estructura del Slab Allocator



11.6.3 Gestión del área de memoria contigua

- Útiles para la gestión de zonas pequeñas.
- Es más apropiada para el manejo de memoria utilizada por dispositivos o tareas en tiempo real.
- Ineficaces con zonas de grandes tamaños.
- Bloques de tamaño fijo.

Reserva memoria para un array.

- n: número de elementos.
- size: tamaño de elementos.
- flags: tipo de la memoria reservada.

En la implementación de kmalloc y kfree, Linux utiliza listas de zonas disponibles. Existe una para cada tamaño de zona.

Aunque kmalloc pida un tamaño específico, Linux busca dentro de la lista de tamaño inmediatamente superior.

El número de páginas varía según el tamaño de los bloques almacenados en la lista. Como mínimo se establece tantas como sean necesarias para formar un bloque.

```
  31static inline void *kmalloc(size_t size, gfp_t flags)
  32{
//si la variable en tiempo de compilación es conocida como una constante
  33        if (__builtin_constant_p(size)) {
  34                int i = 0;
  36                if (!size)
  37                        return ZERO_SIZE_PTR;
  39#define CACHE(x) \
  40                if (size <= x) \
  41                        goto found; \
  42                else \
  43                        i++;
  44#include "kmalloc_sizes.h"
  45#undef CACHE
  50found:
  51#ifdef CONFIG_ZONE_DMA
  52                if (flags & GFP_DMA)
  53                        return
kmem_cache_alloc(malloc_sizes[i].cs_dmacachep,
  54                                                flags);
  55#endif
//se coge memoria con kmem_cache_alloc
  56                return kmem_cache_alloc(malloc_sizes[i].cs_cachep,
flags);
  57        }
  58        return __kmalloc(size, flags);
  59}
```

```
3780void kfree(const void *objp)
3781{
3782        struct kmem_cache *c;
3783        unsigned long flags;
3784
3785        if (unlikely(ZERO_OR_NULL_PTR(objp)))
3786                return;
//salva el estado de la CPU actual y deshabilita interrupciones
3787        local_irq_save(flags);
3788        kfree_debugcheck(objp);
3789        c = virt_to_cache(objp);
3790        debug_check_no_locks_freed(objp, obj_size(c));
3791        debug_check_no_obj_freed(objp, obj_size(c));
3792        __cache_free(c, (void *)objp);
3792        local_irq_restore(flags);
3793}
```

Estas funciones del núcleo hacen uso de __get_free_pages y free_pages, que son funciones de bajo nivel para la manipulación de páginas:

```
1657unsigned long get_zeroed_page(gfp_t gfp_mask)/*obtiene un único marco
de página y lo rellena con 0's
1658{
1659        struct page * page;
1660
1661        /*
1662         * get_zeroed_page() returns a 32-bit address, which cannot
1663         * represent a highmem page
1664         */
1665        VM_BUG_ON((gfp_mask & __GFP_HIGHMEM) != 0);
/*Get_zeroed_page devuelve una dirección de 32 bits
1667        page = alloc_pages(gfp_mask | __GFP_ZERO, 0);
1668        if (page)
1669                return (unsigned long) page_address(page);
1670        return 0;
1671}
```

```
1647 /* Intenta conseguir páginas libres
1646unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
1647{
1648        struct page * page;
1649        page = alloc_pages(gfp_mask, order);
1650        if (!page)
1651                return 0;
1652        return (unsigned long) page_address(page);
1653}
```

Las páginas se liberan mediante la función free_pages (addr, order). Decrementa el counter y si es 0, libera los marcos de páginas. La función free_pages llama a la función __free_pages dependiendo del parámetro order.

```
1683void __free_pages(struct page *page, unsigned int order)
1684{
1685        if (put_page_testzero(page)) {
1686                if (order == 0)
1687                        free_hot_page(page);
1688                else
1689                        __free_pages_ok(page, order);
1690        }
1691}
/* Libera la página
1695void free_pages(unsigned long addr, unsigned int order)
1696{
1697        if (addr != 0) {
1698                VM_BUG_ON(!virt_addr_valid((void *)addr));
1699                __free_pages(virt_to_page((void *)addr), order);
1700        }
1701}
```

Se distinguen dos tipos de páginas:

- Hot: La página está en la cache del procesador.
- Cold: La página está en la memoria principal.

Para ello, para liberar las páginas diferenciamos entre los 2 tipos de páginas

```
1018void free_hot_page(struct page page)
1019{
1020        free_hot_cold_page(page, 0); /*la página está en la cache del
procesador
1021}
1023void free_cold_page(struct page *page)
1024{
1025        free_hot_cold_page(page, 1); /*la página está en memoria
principal
1026}
```

11.6.4 Gestión del área de memoria no contigua

Para asignar área de memoria no contigua el kernel utiliza la función vmalloc que se encuentra declarada en el archivo cabecera.

- Llama a get_vm_area() para crear un nuevo descriptor y devolver la dirección lineal asignada al área de memoria.
- Llama a kmalloc() obtiene grupo de marcos de página contiguos.
- Devuelve la dirección lineal inicial del área de memoria no contigua.

Para liberar área de memoria no contigua el kernel utiliza la función vfree().

- Llama a remove_vm_area() para encontrar la dirección lineal del descriptor del área asociada al área a liberar.
- El área en sí se libera llamando a deallocate_pages().
- El descriptor se libera llamando a kfree().

```
1380void *vmalloc(unsigned long size)
1381{
1382       return __vmalloc(size, GFP_KERNEL | __GFP_HIGHMEM,
PAGE_KERNEL, -1, __builtin_return_address(0));
1383}
```

```
1390static void *__vmalloc_node(unsigned long size, gfp_t gfp_mask,
pgprot_t prot,int node, void *caller)
1391{
   //declaración de las estructuras que se van a usar.
1349        struct vm_struct *area;
1351        size = PAGE_ALIGN(size);
/* Comprueba si el tamaño es 0 o mayor que el permitido */
1352        if (!size || (size >> PAGE_SHIFT) > num_physpages)
1353                return NULL;
1354/* Intenta alojar un área de memoria del tamaño especificado */
1355        area = get_vm_area_node(size, VM_ALLOC, VMALLOC_START,
VMALLOC_END, node, gfp_mask, caller);
//Si no se ha podido alojar la memoria, retornamos NULL. Si no,
actualizamos el puntero a la zona de memoria.
1358        if (!area)
1359                return NULL;
1360
//Retornamos la dirección de la nueva zona de memoria que se ha asignado
1361        return __vmalloc_area_node(area, gfp_mask, prot, node,
caller);
1362}
1364void *__vmalloc(unsigned long size, gfp_t gfp_mask, pgprot_t prot)
1365{
1366        return __vmalloc_node(size, gfp_mask, prot, -1, __builtin
return address(0));
1367}
```

```
1166static void __vunmap(const void *addr, int deallocate_pages)
1167{
1168        struct vm_struct *area;
1170        if (!addr)
1171                return;/* Dirección incorrecta. No es múltiplo del
tamaño de página (no está alineada
1173        if ((PAGE_SIZE-1) & (unsigned long)addr) {
1174                WARN(1, KERN_ERR "Trying to vfree() bad address
(%p)\n", addr);
1175                return;
1176        }
       //se borra el área de memoria apuntada por addr
1178        area = remove_vm_area(addr);
1179        if (unlikely(!area)) {
1180                WARN(1, KERN_ERR "Trying to vfree() nonexistent vm
area (%p)\n", addr);
1182                return;
1183        }
1185        debug_check_no_locks_freed(addr, area->size);
1186        debug_check_no_obj_freed(addr, area->size);
1187
1188        if (deallocate_pages) {
1189                int i;
1191                for (i = 0; i < area->nr_pages; i++) {
1192                        struct page *page = area->pages[i];
1194                        BUG_ON(!page);
1195                        __free_page(page);
1196                }
1198                if (area->flags & VM_VPAGES)
1199                        vfree(area->pages);
1200                else
1201                        kfree(area->pages);
1202        }
1204        kfree(area);
1205        return;
1206}
```

```
1218void vfree(const void *addr)
1219{
1220        BUG_ON(in_interrupt());
1221        __vunmap(addr, 1);
1222}
```

11.7 Proyección de Memoria (Memory Mapping)

Cuando se ejecuta un programa, el contenido del fichero imagen ha de copiarse al espacio de memoria virtual del proceso. Lo mismo sucede con cualquier biblioteca compartida que el proceso necesite. El fichero ejecutable realmente no se lleva a memoria física, sino que solamente se enlaza en la memoria virtual del proceso. Luego, conforme partes del programa son referenciadas por la ejecución de la aplicación, la imagen es llevada a memoria física desde la imagen del ejecutable. Este tipo de enlazado de una imagen sobre el espacio de direcciones virtuales de un proceso se conoce como "proyección de memoria".

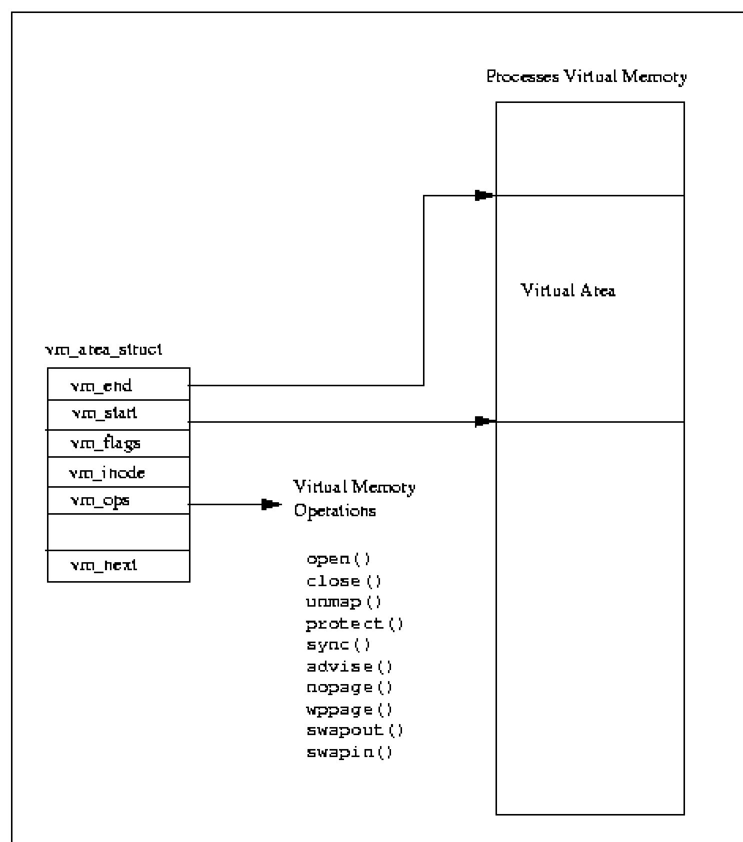


Figura 12: Áreas de Memoria Virtual

La memoria virtual de cada proceso está representado por la estructura de datos mm_struct. Ésta contiene información sobre la imagen que actualmente se está ejecutando (por ejemplo *"bash"*) así como punteros a unas cuantas estructuras vm_area_struct. Cada estructura de datos vm_area_struct describe el inicio y fin de un área de memoria virtual, los permisos del proceso sobre esta memoria y el conjunto de operaciones para gestionarla. Estas operaciones son el

conjunto de rutinas que Linux tiene que utilizar para manipular esta área de memoria virtual. Por ejemplo, una de las operaciones de memoria virtual lleva a cabo las acciones necesarias cuando el proceso ha intentado acceder a la memoria virtual pero encuentra (debido a un fallo de página) que la memoria no está realmente en memoria física. Esta es la operación *nopage*. La operación *nopage* se emplea cuando Linux pide páginas de un fichero ejecutable a memoria.

Cuando una imagen ejecutable se proyecta sobre las direcciones virtuales de un proceso se generan un conjunto de estructuras de datos vm_area_struct. Cada estructura de éstas representa una parte de la imagen del ejecutable; el código ejecutable, datos inicializados (variables), datos no inicializados y demás. Linux tiene unas cuantas operaciones de memoria virtual estándar, y cuando se crean las estructuras de datos vm_area_struct, el conjunto de operaciones correcto se asocia con ésta.

11.8 Paginación por Demanda

Una vez una imagen ejecutable ha sido proyectada sobre la memoria virtual de un proceso, éste puede comenzar su ejecución. Puesto que sólo el principio de la imagen ha sido realmente copiado en memoria física, rápidamente el programa accederá a una área de memoria virtual que todavía no ha sido llevada a memoria física. Cuando un proceso accede a una dirección virtual que no tiene una entrada válida en la tabla de páginas, el procesador informará sobre el fallo de página a Linux. El fallo de página indica la dirección virtual donde se produjo el fallo de página y el tipo de acceso que lo causó.

Linux debe encontrar la vm_area_struct que representa el área de memoria donde sucedió el fallo de página. Puesto que la búsqueda por las estructuras de datos vm_area_struct es crítica para la gestión eficiente de los fallos de páginas, éstas están organizadas juntas en un estructura de árbol AVL (Adelson-Velskii and Landis) Rojo-Negro. Si no hay ninguna estructura vm_area_struct para la dirección virtual que produjo el fallo de página entonces el proceso ha accedido a una dirección de memoria virtual ilegal. Linux enviará al proceso la señal SIGSEGV, y si el proceso no ha instalado un manejador de señales para esta señal entonces morirá.

Lo siguiente que hace Linux es comprobar el tipo de fallo de página producido y los tipos de acceso permitidos para el área de memoria virtual en cuestión. Si el proceso ha intentado acceder a la memoria de forma ilegal, por ejemplo intentando escribir sobre un área de la que sólo tenía permisos de lectura, también se señala un error de memoria.

Ahora que Linux ha determinado que el fallo de página es legal, tiene que tratarlo. Linux ha de diferenciar entre páginas que están en un fichero de intercambio y aquellas que son parte de una imagen ejecutable localizadas en algún lugar del disco. Esto lo hace utilizando la entrada en la tabla de páginas de la página que causó el fallo.

Si la entrada de la tabla de páginas es inválida pero no está vacía, el fallo de página se debe a que la página está en esos momentos en un fichero de intercambio. Estas entradas se identifican en el Alpha AXP porque no tienen el bit de válido activado pero tienen un valor distinto de cero en el campo PFN. En este caso, el campo PFN contiene información sobre dónde se encuentra la página: fichero de intercambio y posición dentro de éste.

No todas las estructuras vm_area_struct tienen un conjunto de operaciones de memoria virtual e incluso éstas pueden no tener la operación *nopage*. Esto es debido a que por defecto Linux

gestiona los accesos asignando una página de memoria física nueva y creando para ésta la correspondiente entrada en la tabla de páginas. Si no hay operación de *nopage* para esta área de memoria virtual, Linux hará esto último.

La operación genérica de *nopage* de Linux se utiliza para proyectar imágenes ejecutables y utiliza la cache de páginas para traer la página imagen requerida a memoria física.

Cuando la página necesitada es cargada en memoria física, las tablas de páginas del proceso son actualizadas. Puede ser necesario realizar determinadas acciones especificas del hardware para actualizar estas entradas, en particular si el procesador utilizar buffers cache TLB (Translation Look-aside Buffer). Ahora que el problema con la página que produjo el fallo ha sido resuelto, se puede olvidar el percance y el proceso puede continuar su ejecución en la instrucción que produjo el fallo de página.

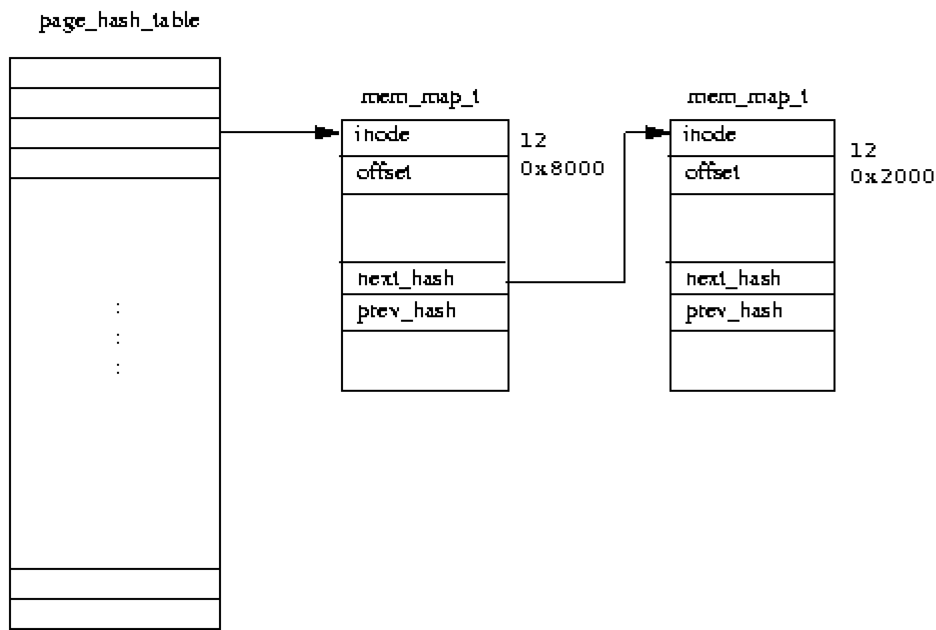
11.9 La Cache de Páginas de Linux

El cometido de la cache de páginas en Linux es el de acelerar el acceso a los fichero de disco. Las lecturas sobre los ficheros proyectados en memoria se realizan página a página y estas páginas se guardan en la cache de páginas. La Figura 7 muestra que la cache de páginas consta de la page_hash_table y un vector de punteros a estructuras mem_map_t. Cada fichero en Linux se identifica por una estructura inode. Cada inode es único y describe a un único fichero. El índice en la tabla de páginas se construye a partir del inode del fichero y el desplazamiento dentro de éste.

Siempre que se lee en una página de un fichero proyectado en memoria, por ejemplo cuando se necesita traer a memoria una página desde un fichero de intercambio, la página se lee a través de la cache de páginas. Si la página está presente en la cache, se devuelve un puntero a la estructura de datos mem_map_t a la rutina de tratamiento de fallos de página. En caso contrario, la página se ha de traer a memoria desde el sistema de ficheros que contiene la imagen. Linux asigna una página física y lee la página desde el fichero del disco.

Si es posible, Linux comenzará una lectura de la siguiente página del fichero. Con esta página de adelanto se consigue que si el proceso está accediendo las páginas de forma secuencial, la siguiente página esté lista y esperando en memoria la petición del proceso.

Con el tiempo, la cache de páginas va creciendo conforme las imágenes se leen y ejecutan. Las páginas han de ser eliminadas de la cache cuando dejan de utilizarse, por ejemplo cuando una imagen ya no es utilizada por ningún proceso. Conforme Linux utiliza memoria puede comenzar a escasear las páginas de memoria física. En esta situación Linux reducirá el tamaño de la cache de páginas.

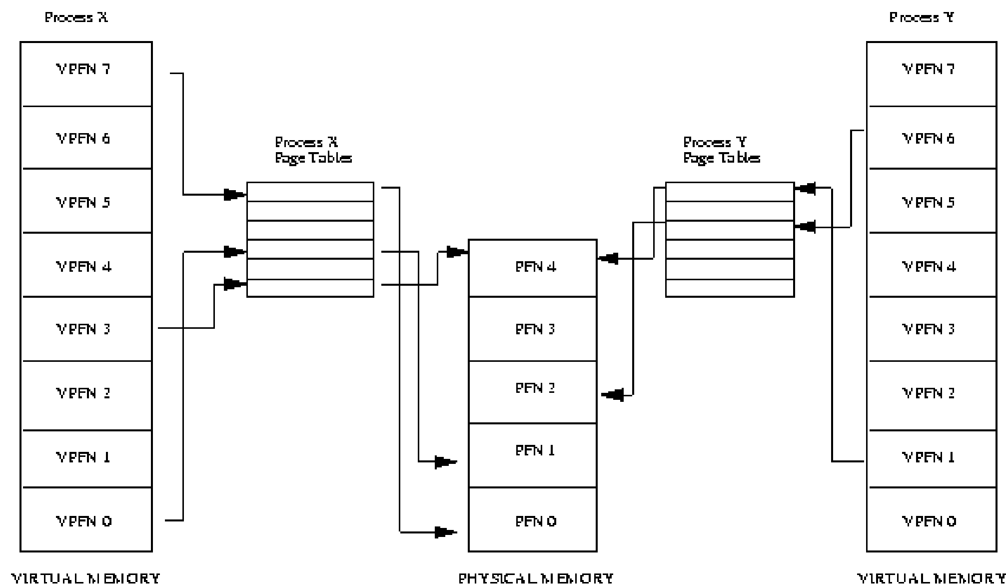


La Cache de Páginas de Linux

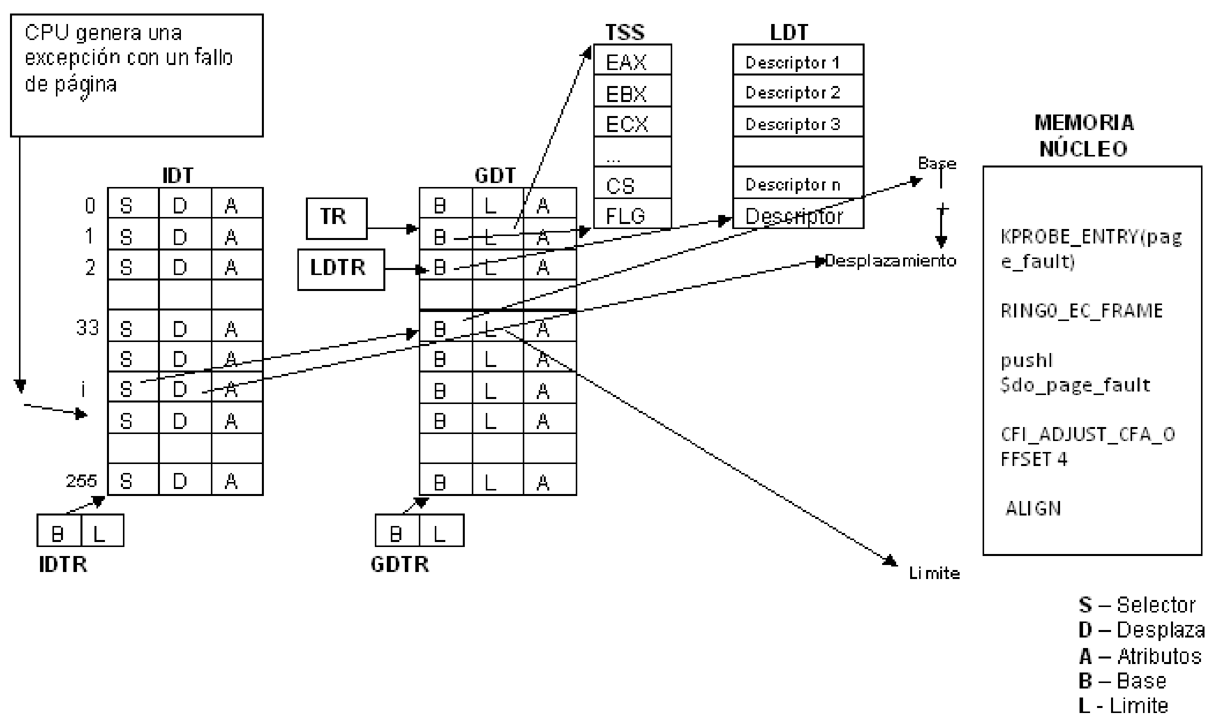
11.10 Copy-on-Write

Copy-on-write es una técnica para realizar eficientemente (tanto en tiempo como en memoria) la copia de páginas. Cuando se invoca a la primitiva *fork*, Linux no duplica las páginas de memoria que son necesarias para el nuevo proceso, sino que hace apuntar las entradas de la tabla de páginas del nuevo proceso a las páginas del proceso padre. Cuando alguna de las páginas es modificada por alguno de los procesos, entonces el núcleo pasa a realizar la duplicación de dicha página. Así, Linux se evita la copia de páginas de memoria que no van ha ser utilizadas (p.e. el código situado antes de la primitiva fork, en muy probable que no sea utilizado por el proceso hijo), ahorrando la memoria correspondiente y el tiempo necesario para copiarlas.

La forma de llevar a cabo este proceso consiste en establecer los permisos de estas páginas a sólo-lectura pero sabiendo que dichas páginas se pueden modificar (indicándolo en el *vma* correspondiente). Cuando ocurre una violación de acceso a estas páginas (uno de los procesos intenta escribir) es cuando se realiza la duplicación propiamente dicha.



11.11 Tratamiento de las excepciones.



Tratamiento de excepciones

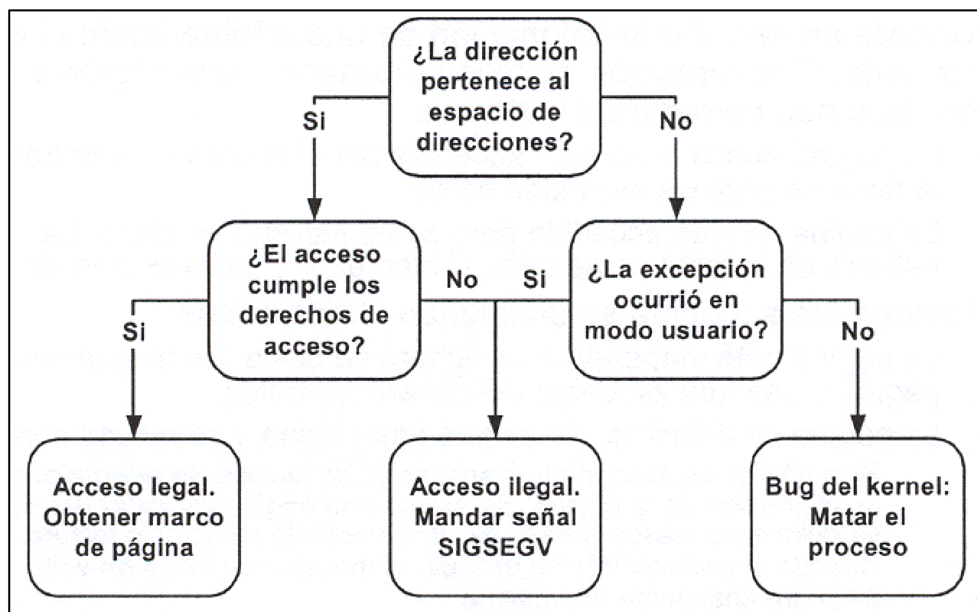
Cuando se genera una excepción se determinará mediante la tabla gdt y idt, la función manejadora de dicha excepción. Para ello se guarda el estado del proceso en la pila, se llama a la función manejadora y posteriormente se recupera el estado del proceso.

El gestor o manejador de fallos de página debe distinguir la causa del fallo:

- Error de programación
- Referencias a páginas legítimas pero no presentes
    1. Por Copy-On-Write
    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, y cuando el proceso desea continuar su ejecución necesita volver a traer a memoria principal estas páginas.
    3. Porque no han sido asignadas todavía.
    4. Por expansión de pila.

La función manejadora es do_page_fault. Esta función compara la dirección lineal que causó el fallo (cr2) con las regiones de memoria de current para determinar qué hacer, según el esquema de la figura siguiente

Esquema general del manejador de fallos de página



do_page_fault

Los valores devueltos, definidos por las macros, se harán usados por las distintas funciones:

```
#define VM_FAULT_ERROR   -> Nos indica que hay algún tipo de error.
#define VM_FAULT_OOM      -> out_of_memory().
#define VM_FAULT_SIGBUS  -> do_sigbus().
#define VM_FAULT_MAJOR  -> incrementa  tsk->maj_flt;
```

```
/*
* Esta rutina se encarga de manejar los fallos de pagina, determina la
* dirección y el problema, llamando a la rutina apropiada
* error_code:
*   bit 0 == 0 página no encontrada, 1 significa fallo de protección
*      bit 1 == 0 lectura, 1 escritura
*      bit 2 == 0 modo kernel, 1 modo usuario
*      bit 3 == 1 uso del bit reservado detectado
*      bit 4 == 1 el fallo se produjo en la búsqueda de una instrucción
*/
 584void __kprobes do_page_fault(struct pt_regs *regs, unsigned long
error_code)
 585{
 586        struct task_struct *tsk;
 587        struct mm_struct *mm;
 588        struct vm_area_struct *vma;
 589        unsigned long address;
 590        int write, si_code;
 591        int fault;
 592#ifdef CONFIG_X86_64
 593        unsigned long flags;
 594        int sig;
 595#endif
 596        /* Tomamos del proceso actual su información de gestión de la
memoria
 597        tsk = current;
 598        mm = tsk->mm;
 599        prefetchw(&mm->mmap_sem);
 600
 601        /* Cogemos la dirección que provocó la excepción */
            /* Los procesadores Intel almacenan en el registro cr2 la
dirección de la instrucción que provocó el fallo de la página*/
 602        address = read_cr2();
 603
 604        si_code = SEGV_MAPERR;
 605
 606        if (unlikely(kmmio_fault(regs, address)))
 607                return;
 608
 609        /*
 610         * Comprobamos si el error se produjo en el espacio kernel
 611         * (error_code & 4) == 0, y que el fallo no fue un
 612         * error de protección(error_code & 9) == 0.
 613        */
 622#ifdef CONFIG_X86_32
 623        if (unlikely(address >= TASK_SIZE)) {
 624#else
 625        if (unlikely(address >= TASK_SIZE64)) {
 626#endif
 627                if (!(error_code & (PF_RSVD|PF_USER|PF_PROT)) &&
 628                    vmalloc_fault(address) >= 0)
 629                        return;
 630
```

```
 631                /* Can handle a stale RO->RW TLB */
 632                if (spurious_fault(address, error_code))
 633                        return;
 634
 635                /* kprobes don't want to hook the spurious faults. */
 636                if (notify_page_fault(regs))
 637                        return;
 638                /*
 639                 * Don't take the mm semaphore here. If we fixup a
prefetch
 640                 * fault we could otherwise deadlock.
 641                 */
 642                goto bad_area_nosemaphore;
 643        }
 644
 645        /* kprobes don't want to hook the spurious faults. */
 646        if (notify_page_fault(regs))
 647                return;
 648
 649        /*
 650         * It's safe to allow irq's after cr2 has been saved and the
 651         * vmalloc fault has been handled.
 652         *
 653         * User-mode registers count as a user access even for any
 654         * potential system fault or CPU buglet.
 655         */
 656        if (user_mode_vm(regs)) {
 657                local_irq_enable();
 658                error_code |= PF_USER;
 659        } else if (regs->flags & X86_EFLAGS_IF)
 660                local_irq_enable();
 661
 662#ifdef CONFIG_X86_64
 663        if (unlikely(error_code & PF_RSVD))
 664                pgtable_bad(address, regs, error_code);
 665#endif
 666
 667        /*
 668         * If we're in an interrupt, have no user context or are
running in an
 669         * atomic region then we must not take the fault.
 670         */
 671        if (unlikely(in_atomic() || !mm))
 672                goto bad_area_nosemaphore;
 673
 690        if (!down_read_trylock(&mm->mmap_sem)) {
 691                if ((error_code & PF_USER) == 0 &&
 692                    !search_exception_tables(regs->ip))
 693                        goto bad_area_nosemaphore;
 694                down_read(&mm->mmap_sem);
 695        }
 696        // Se obtiene el descriptor de la región de memoria afectada
mediante la función find_vma, y luego comprueba el tipo de error */
```

```
 697        vma = find_vma(mm, address);
 698        if (!vma) /* si no existe una vma*/
 699                goto bad_area;
 700        if (vma->vm_start <= address) /* Cuando la dirección virtual
está dentro de rango de direcciones del programa saltamos a good_area */
 701                goto good_area;
 702        if (!(vma->vm_flags & VM_GROWSDOWN))
 703                goto bad_area;
/* Si el fallo de página es en modod usuario se comprueba que el proceso no
se ha salido de la pila (bit 2 error_code es 1 si esta en modo usuario y 0
si esta en modod núcleo */
 704        if (error_code & PF_USER) {
 705                /*
 706                 *No podemos acceder a la pila por debajo de %esp.
 707                 * porque siempre es un fallo.
 710                */
 711                if (address + 65536 + 32 * sizeof(unsigned long) <
regs->sp)
 712                        goto bad_area;
 713        }
/* Intenta ajustar el vma (el campo start) para contener la nueva
dirección*/
 714        if (expand_stack(vma, address))
 715                goto bad_area;
 716/*
 717 * Si llegamos aquí es porque tenmos un buen vm_area así que podremos
manejar el fallo siempre y cuando no tengamos problemas con los permisos de
lectura y escritura */
 720good_area:
 721        si_code = SEGV_ACCERR;
 722        write = 0;
/* Bit 0 de error_code:
        0: significa que la página no estaba en memoria
        1: significa que la página estaba en memoria pero que se violaron
sus protecciones de acceso
        Bit 1 de error_code:
          0: para una lectura y 1 para una escritura */
 723        switch (error_code & (PF_PROT|PF_WRITE)) {
/* "11" La página estaba en memoria, se intentó una escritura y se provocó
un fallo de protección. En estos casos, el Kernel comprueba si se debe
hacer un Copy-On-Write mirando si el VMA es modificable, en caso contrario,
es un fallo de protección */
 724        default:        /* 3: write, present */
 725                /* fall through */
 726        case PF_WRITE:          /* write, not present */
 727                if (!(vma->vm_flags & VM_WRITE))
 728                        goto bad_area;
 729                write++;
 730                break;
/* "01" La página estaba en memoria y se intentó leer una pagina que no se
puede leer, es un fallo de protección */
 731        case PF_PROT:           /* read, present */
 732                goto bad_area;
```

```
/* “0” La página no estaba en memoria y se intentó leer. Si la VMA no es
legible ni ejecutable, no se carga la pagina desde disco, ya que el proceso
no la podrá utilizar */
 733        case 0:                 /* read, not present */
 734                if (!(vma->vm_flags & (VM_READ | VM_EXEC | VM_WRITE)))
 735                        goto bad_area;
 736        }
*/*Si hemos llegado hasta aquí implicará que el acceso a memoria es
correcto pero que la página no está en memoria, por tanto llamaremos a
handle_mm_fault para que nos atienda la petición *
 743        fault = handle_mm_fault(mm, vma, address, write);
 744        if (unlikely(fault & VM_FAULT_ERROR)) {
 745                if (fault & VM_FAULT_OOM)
 746                        goto out_of_memory
/* En este punto, es donde se intenta cargar una pagina desde el
dispositivo de intercambio (swap). Si se no puede, se da un error */;
 747                else if (fault & VM_FAULT_SIGBUS)
 748                        goto do_sigbus;
 749                BUG();
 750        }
 751        if (fault & VM_FAULT_MAJOR)
 752                tsk->maj_flt++;
 753        else
 754                tsk->min_flt++;
 755
 756#ifdef CONFIG_X86_32
 760        if (v8086_mode(regs)) {
 761                unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
 762                if (bit < 32)
 763                        tsk->thread.screen_bitmap |= 1 << bit;
 764        }
 765#endif
 766        up_read(&mm->mmap_sem);
 767        return;
 768
 769/*
 770 * Se intentó un acceso a memoria fuera de nuestro mapa de memoria
 771 * arregalrlo, pero comprobando primero si fue modo user o kernel
 772 */
/* Si se llega aquí, hay un fallo de protección. Existen dos tipos de
error: se intenta escribir en una página que no corresponde al proceso, o
se intenta escribir en una página que si corresponde al proceso pero que
está protegida contra escrituras */
 773bad_area:
 774        up_read(&mm->mmap_sem);
 775
 776bad_area_nosemaphore:
 777        /* Si el fallo lo provocó un proceso de usuario, se le envía
una señal SIGSEGV (segmentation fault) */
 778        if (error_code & PF_USER) {
 782                local_irq_enable();
 783
```

```
 784                /*Si el fallo es en el espacio de usuario podemos
lanzar otro do_page_fault*/
 788                if (is_prefetch(regs, address, error_code))
 789                        return;
 790
 791                if (is_errata100(regs, address))
 792                        return;
 793
 794                if (show_unhandled_signals && unhandled_signal(tsk,
SIGSEGV) &&
 795                    printk_ratelimit()) {
 796                        printk(
 797                        "%s%s[%d]: segfault at %lx ip %p sp %p error
%lx",
 798                        task_pid_nr(tsk) > 1 ? KERN_INFO : KERN_EMERG,
 799                        tsk->comm, task_pid_nr(tsk), address,
 800                        (void *) regs->ip, (void *) regs->sp,
error_code);
 801                        print_vma_addr(" in ", regs->ip);
 802                        printk("\n");
 803                }
 804
 805                tsk->thread.cr2 = address;
 806                /* El espacio del kernel esta siempre protegido */
 807                tsk->thread.error_code = error_code | (address >=
TASK_SIZE);
 808                tsk->thread.trap_no = 14;
/*Segmentation fault, se envía señal al proceso: SISSEGV */
 809                force_sig_info_fault(SIGSEGV, si_code, address, tsk);
 810                return;
 811        }
 812
 813        if (is_f00f_bug(regs, address))
 814                return;
 815
/* Se llega a este punto si, es un problema del núcleo: Si el fallo lo
provocó el núcleo y no es la instrucción invalida Si el fallo ocurrió
durante una interrupción o no ha se estaba ejecutando una tarea de usuario
Fallo handle_mm_fault y el sistema estaba en modo núcleo. Aún no se ha dado
el caso*/
 816no_context:
*/ Se mira el núcleo esta preparado para controlar esta excepción*/
 818        if (fixup_exception(regs))
 819                return;
 830        if (is_prefetch(regs, address, error_code))
 831                return;
 832
 833        if (is_errata93(regs, address))
 834                return;
 837 /* el kernel ha intendado acceder a alguna pagina no valida, e intenta
recuperarse */
 840#ifdef CONFIG_X86_32
 841        bust_spinlocks(1);
```

```
 842#else
 843        flags = oops_begin();
 844#endif
 845
 846        show_fault_oops(regs, error_code, address);
 847
 848        tsk->thread.cr2 = address;
 849        tsk->thread.trap_no = 14;
 850        tsk->thread.error_code = error_code;
 851/* se imprime en pantalla los errores ocurridos con las respectivas
direcciones de memoria. */
 852#ifdef CONFIG_X86_32
 853        die("Oops", regs, error_code);
 854        bust_spinlocks(0);
/* Error del núcleo no ha podido ser tratado → Detención del sistema.*/
 855        do_exit(SIGKILL);
 856#else
 857        sig = SIGKILL;
 858        if (__die("Oops", regs, error_code))
 859                sig = 0;
 860        /* Executive summary in case the body of the oops scrolled away
*/
 861        printk(KERN_EMERG "CR2: %016lx\n", address);
 862        oops_end(flags, regs, sig);
 863#endif
 864 */ Estamos en un área fuera de la memoria del proceso
 865out_of_memory:
 870        up_read(&mm->mmap_sem);
 871        pagefault_out_of_memory();
 872        return;
/* Se llega a esta etiqueta si hubo un fallo de página y no se pudo cargar
la pagina correspondiente desde el dispositivo de intercambio. Se envía una
señal SIGBUS al proceso. Si la tarea era el núcleo, se vuelve a atrás para
intentar tratar la excepción o para detener la ejecución del sistema */
 874do_sigbus:
 875        up_read(&mm->mmap_sem);
 876
 877        /* Estamos en modo kernel, manejamos la excepción o morimos */
 878        if (!(error_code & PF_USER))
 879                goto no_context;
 880#ifdef CONFIG_X86_32
 881        /* User space => ok to do another page fault */
 882        if (is_prefetch(regs, address, error_code))
 883                return;
 884#endif
 885        tsk->thread.cr2 = address;
 886        tsk->thread.error_code = error_code;
 887        tsk->thread.trap_no = 14;
 888        force_sig_info_fault(SIGBUS, BUS_ADRERR, address, tsk);
 889}
```

handle_mm_fault

A continuación comentaremos la función handle_mm_fault. Esta función carga la página de memoria apuntada por *address* desde un dispositivo de intercambio. La página que se desea cargar es válida y pertenece al VMA indicado por *vma*. Esta función es invocada por do_page_fault, cuando una página a la que se intenta acceder no está en memoria principal.

Valores devueltos:

VM_MINOR_FAULT: El proceso no se duerme, la página está en caché.

VM_MAJOR_FAULT: El proceso debe dormirse en lo que se transfiere la página (presumiblemente de disco)

VM_FAULT_OOM: No hay memoria disponible

```
2844int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
2845                unsigned long address, int write_access)
2846{
2847        pgd_t *pgd;
2848        pud_t *pud;
2849        pmd_t *pmd;
2850        pte_t *pte;
2851
2852        __set_current_state(TASK_RUNNING);
2853
2854        count_vm_event(PGFAULT);
2855
2856        if (unlikely(is_vm_hugetlb_page(vma)))
2857                return hugetlb_fault(mm, vma, address, write_access);
/* Se localiza la entrada correspondiente en el directorio de páginas del
vma */
2858
2859        pgd = pgd_offset(mm, address);
2860        pud = pud_alloc(mm, pgd, address);
2861        if (!pud)
2862                return VM_FAULT_OOM;
/* Se localiza la entrada correspondiente de la tabla de páginas intermedia
a partir del directorio. En x86 coinciden. */
2863        pmd = pmd_alloc(mm, pud, address);
2864        if (!pmd)
2865                return VM_FAULT_OOM;
/* Se intenta localizar la tabla de páginas a partir de la tabla
intermedia*/
2866        pte = pte_alloc_map(mm, pmd, address);
2867        if (!pte)
2868                return VM_FAULT_OOM;
2869
2870        return handle_pte_fault(mm, vma, address, pte, pmd,
write_access);
2871}
```

handle_pte_fault

La siguiente función a comentar es el handle_pte_fault. Función que inspecciona la entrada de tabla de páginas para determinar qué ocasionó el fallo y llamar al manejador adecuado.

```
2788static inline int handle_pte_fault(struct mm_struct *mm,
2789                struct vm_area_struct *vma, unsigned long address,
2790                pte_t *pte, pmd_t *pmd, int write_access)
2791{
2792        pte_t entry;
2793        spinlock_t *ptl;
2794
2795        entry = *pte;
/* Si la página no está en memoria */
2796        if (!pte_present(entry)) {
/* Si además la página nunca ha sido asignada (no tiene asignada una
entrada en una tabla de páginas */
2797                if (pte_none(entry)) {
2798                        if (vma->vm_ops) {
2799                                if (likely(vma->vm_ops->fault))
2800                                        return do_linear_fault(mm, vma,
address,
2801                                                pte, pmd, write_access,
entry);
2802                        }
2803                        return do_anonymous_page(mm, vma, address,
2804                                                 pte, pmd,
write_access);
2805                }
2806                if (pte_file(entry))
2807                        return do_nonlinear_fault(mm, vma, address,
2808                                        pte, pmd, write_access, entry);
/* La página no está en memoria, pero ya había sido mapeada, por tanto debe
estar en el espacio de intercambio */
2809                return do_swap_page(mm, vma, address,
2810                                        pte, pmd, write_access, entry);
2811        }
2812
2813        ptl = pte_lockptr(mm, pmd);
2814        spin_lock(ptl);
2815        if (unlikely(!pte_same(*pte, entry)))
2816                goto unlock;
/* Si la página está en memoria, se concluye que hubo una violación de la
protección de la página. */
2817        if (write_access) {
2818                if (!pte_write(entry))
/* Se ha intentado escribir en una página no modificable. Por tanto se
trata el fallo de protección de escritura */
2819                        return do_wp_page(mm, vma, address,
2820                                        pte, pmd, ptl, entry);
/* Se marca la página como escrita (bit dirty=1) en el campo entry */
2821                entry = pte_mkdirty(entry);
2822        }
```

```
2823        entry = pte_mkyoung(entry);
2824        if (ptep_set_access_flags(vma, address, pte, entry,
write_access)) {
2825                update_mmu_cache(vma, address, entry);
2826        } else {
2827                /*
2828                 * This is needed only for protection faults but the
arch code
2829                 * is not yet telling us if this is a protection fault
or not.
2830                 * This still avoids useless tlb flushes for .text page
faults
2831                 * with threads.
2832                 */
2833                if (write_access)
2834                        flush_tlb_page(vma, address);
2835        }
2836unlock:
2837        pte_unmap_unlock(pte, ptl);
2838        return 0;
2839}
```

do_swap_page

La función do_swap_page carga 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 a esta función, 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.

Puede devolver los siguientes valores:

1: La página ya estaba en la swap cache.

2: La página tuvo que leerse del área de swap.

-1: Hubo algún problema en la transferencia de la página.

```
2394static int do_swap_page(struct mm_struct *mm, struct vm_area_struct
*vma,
2395                unsigned long address, pte_t *page_table, pmd_t *pmd,
2396                int write_access, pte_t orig_pte)
2397{
2398        spinlock_t *ptl;
2399        struct page *page;
2400        swp_entry_t entry;
2401        pte_t pte;
2402        struct mem_cgroup *ptr = NULL;
2403        int ret = 0;
```

```
2404
2405        if (!pte_unmap_same(mm, pmd, page_table, orig_pte))
2406                goto out;
2407
2408        entry = pte_to_swp_entry(orig_pte);
2409        if (is_migration_entry(entry)) {
2410                migration_entry_wait(mm, pmd, address);
2411                goto out;
2412        }
2413        delayacct_set_flag(DELAYACCT_PF_SWAPIN);
2414        page = lookup_swap_cache(entry);
*/ Se comprueba que se tiene la página especificada */
2415        if (!page) {
2416                grab_swap_token(); /* Contend for token _before_ read-
in
2417                page = swapin_readahead(entry,
2418                                        GFP_HIGHUSER_MOVABLE, vma,
address);
2419                if (!page) {
2424                        page_table = pte_offset_map_lock(mm, pmd,
address, &ptl);
2425                        if (likely(pte_same(*page_table, orig_pte)))
/* tiene que leer la página del área de swap Major fault */
2426                                ret = VM_FAULT_OOM;
2427                        delayacct_clear_flag(DELAYACCT_PF_SWAPIN);
2428                        goto unlock;
2429                }
2430
2431                /* Had to read the page from swap area: Major fault */
2432                ret = VM_FAULT_MAJOR;
2433                count_vm_event(PGMAJFAULT);
2434        }
2435/* Marca la página como accedida y se bloquea */
2436        mark_page_accessed(page);
2437
2438        lock_page(page);
2439        delayacct_clear_flag(DELAYACCT_PF_SWAPIN);
2440
2441        if (mem_cgroup_try_charge_swapin(mm, page, GFP_KERNEL, &ptr)) {
2442                ret = VM_FAULT_OOM;
2443                unlock_page(page);
2444                goto out;
2445        }
2446
2447        /*
2448         * Back out if somebody else already faulted in this pte.
2449         */
2450        page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
2451        if (unlikely(!pte_same(*page_table, orig_pte)))
2452                goto out_nomap;
2453
2454        if (unlikely(!PageUptodate(page))) {
2455                ret = VM_FAULT_SIGBUS;
```

```
2456                goto out_nomap;
2457        }
2458
2459        /*
2460         * The page isn't present yet, go ahead with the fault.
2461         *
2462         * Be careful about the sequence of operations here.
2463         * To get its accounting right, reuse_swap_page() must be
called
2464         * while the page is counted on swap but not yet in mapcount
i.e.
2465         * before page_add_anon_rmap() and swap_free();
try_to_free_swap()
2466         * must be called after the swap_free(), or it will never
succeed.
2467         * Because delete_from_swap_page() may be called by
reuse_swap_page(),
2468         * mem_cgroup_commit_charge_swapin() may not be able to find
swp_entry
2469         * in page->private. In this case, a record in swap_cgroup  is
silently
2470         * discarded at swap_free().
2471         */
2472
2473        inc_mm_counter(mm, anon_rss);
2474        pte = mk_pte(page, vma->vm_page_prot);
2475        if (write_access && reuse_swap_page(page)) {
2476                pte = maybe_mkwrite(pte_mkdirty(pte), vma);
2477                write_access = 0;
2478        }
2479        flush_icache_page(vma, page);
2480        set_pte_at(mm, address, page_table, pte);
2481        page_add_anon_rmap(page, vma, address);
2482        /* It's better to call commit-charge after rmap is established
*/
2483        mem_cgroup_commit_charge_swapin(page, ptr);
2484/* Libera la entrada pasada por parámetro */
2485        swap_free(entry);
2486        if (vm_swap_full() || (vma->vm_flags & VM_LOCKED) ||
PageMlocked(page))
2487                try_to_free_swap(page);
2488        unlock_page(page);
2489/* Se ha intentado escribir en una página compartida */
2490        if (write_access) {
/* Se ha intentado escribir en una página no modificable. Por tanto, se
trata el fallo de protección de escritura
2491                ret |= do_wp_page(mm, vma, address, page_table, pmd,
ptl, pte);
2492                if (ret & VM_FAULT_ERROR)
2493                        ret &= VM_FAULT_ERROR;
2494                goto out;
2495        }
2496
```

```
2497        /* No need to invalidate - it was non-present before */
2498        update_mmu_cache(vma, address, pte);
2499unlock:
2500        pte_unmap_unlock(page_table, ptl);
2501out:
2502        return ret;
2503out_nomap:
2504        mem_cgroup_cancel_charge_swapin(ptr);
2505        pte_unmap_unlock(page_table, ptl);
2506        unlock_page(page);
2507        page_cache_release(page);
2508        return ret;
2509}
```

do_wp_page

La función do_wp_page gestiona la copia en escritura.

Cuando un proceso accede en escritura a una página compartida y protegida en lectura exclusiva, se asigna una nueva página, y se comprueba si la página afectada es compartida por varios procesos.

En caso afirmativo se copia su contenido en la nueva página, y se inserta en la tabla de páginas del proceso. El número de referencias a la anterior página se decrementa por la llamada a liberar la pagina.

En el caso de que la página afectada no sea compartida, su protección simplemente se modifica para hacer posible la escritura.

```
1888static int do_wp_page(struct mm_struct *mm, struct vm_area_struct *vma,
1889                unsigned long address, pte_t *page_table, pmd_t *pmd,
1890                spinlock_t *ptl, pte_t orig_pte)
1891{
1892        struct page *old_page, *new_page;
1893        pte_t entry;
1894        int reuse = 0, ret = 0;
1895        int page_mkwrite = 0;
1896        struct page *dirty_page = NULL;
1897
1898        old_page = vm_normal_page(vma, address, orig_pte);
/* Se obtiene la página vieja */
1899        if (!old_page) {
1900                /*
1901                 * VM_MIXEDMAP !pfn_valid() case
1902                 *
1903                 * We should not cow pages in a shared writeable
mapping.
1904                 * Just mark the pages writable as we can't do any
dirty
1905                 * accounting on raw pfn maps.
1906                 */
1907                if ((vma->vm_flags & (VM_WRITE|VM_SHARED)) ==
1908                                     (VM_WRITE|VM_SHARED))
```

```
1909                        goto reuse;
1910                goto gotten;
1911        }
1912
1913        /*
1914         * Take out anonymous pages first, anonymous shared vmas are
1915         * not dirty accountable.
1916         */
1917        if (PageAnon(old_page)) {
/* Comprueba que la página no esta compartida */
1918                if (!trylock_page(old_page)) {
1919                        page_cache_get(old_page);
1920                        pte_unmap_unlock(page_table, ptl);
1921                        lock_page(old_page);
1922                        page_table = pte_offset_map_lock(mm, pmd,
address,
1923                                                         &ptl);
1924                        if (!pte_same(*page_table, orig_pte)) {
1925                                unlock_page(old_page);
1926                                page_cache_release(old_page);
1927                                goto unlock;
1928                        }
1929                        page_cache_release(old_page);
1930                }
1931                reuse = reuse_swap_page(old_page);
/* Se desbloquea la página ya que no esta compartida. */
1932                unlock_page(old_page);
1933        } else if (unlikely((vma->vm_flags & (VM_WRITE|VM_SHARED)) ==
1934                                        (VM_WRITE|VM_SHARED))) {
1935                /*
1936                 * Only catch write-faults on shared writable pages,
1937                 * read-only shared pages can get COWed by
1938                 * get_user_pages(.write=1, .force=1).
1939                 */
1940                if (vma->vm_ops && vma->vm_ops->page_mkwrite) {
1941                /* Se inserta en la tabla de páginas del proceso*/
1949                        page_cache_get(old_page);
1950                        pte_unmap_unlock(page_table, ptl);
1951
1952                        if (vma->vm_ops->page_mkwrite(vma, old_page) <
0)
1953                                goto unwritable_page;
1954
1955                        /*
1956                         * Since we dropped the lock we need to
revalidate
1957                         * the PTE as someone else may have changed it.
If
1958                         * they did, we just return, as we can count on
the
1959                         * MMU to tell us if they didn't also make it
writable.
1960                         */
```

```
1961                    page_table = pte_offset_map_lock(mm, pmd,
address,
1962                                                     &ptl);
1963                    page_cache_release(old_page);
1964                    if (!pte_same(*page_table, orig_pte))
1965                            goto unlock;
1966
1967                    page_mkwrite = 1;
1968            }
1969            dirty_page = old_page;
1970            get_page(dirty_page);
1971            reuse = 1;
1972    }
1973
1974    if (reuse) {
1975reuse:
1976            flush_cache_page(vma, address, pte_pfn(orig_pte));
1977            entry = pte_mkyoung(orig_pte);
1978            entry = maybe_mkwrite(pte_mkdirty(entry), vma);
1979            if (ptep_set_access_flags(vma, address, page_table,
entry,1))
1980                    update_mmu_cache(vma, address, entry);
1981            ret |= VM_FAULT_WRITE;
1982            goto unlock;
1983    }
1988    page_cache_get(old_page);
1989gotten:
1990    pte_unmap_unlock(page_table, ptl);
1991
1992    if (unlikely(anon_vma_prepare(vma)))
1993            goto oom;
1994    VM_BUG_ON(old_page == ZERO_PAGE(0));
1995    new_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma, address);
1996    if (!new_page)
1997            goto oom;
2002    if ((vma->vm_flags & VM_LOCKED) && old_page) {
2003            lock_page(old_page);    /* for LRU manipulation */
2004            clear_page_mlock(old_page);
2005            unlock_page(old_page);
2006    }
2007    cow_user_page(new_page, old_page, address, vma);
2008    __SetPageUptodate(new_page);
2009
2010    if (mem_cgroup_newpage_charge(new_page, mm, GFP_KERNEL))
2011            goto oom_free_new;
2012
2016    page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
2017    if (likely(pte_same(*page_table, orig_pte))) {
2018            if (old_page) {
2019                    if (!PageAnon(old_page)) {
2020                            dec_mm_counter(mm, file_rss);
2021                            inc_mm_counter(mm, anon_rss);
2022                    }
```

```
2023                } else
2024                        inc_mm_counter(mm, anon_rss);
2025                flush_cache_page(vma, address, pte_pfn(orig_pte));
2026                entry = mk_pte(new_page, vma->vm_page_prot);
2027                entry = maybe_mkwrite(pte_mkdirty(entry), vma);
2034                ptep_clear_flush_notify(vma, address, page_table);
2035                page_add_new_anon_rmap(new_page, vma, address);
2036                set_pte_at(mm, address, page_table, entry);
2037                update_mmu_cache(vma, address, entry);
2038                if (old_page) {
2061                        page_remove_rmap(old_page);
2062                }
2063
2064                /* Free the old page.. */
2065                new_page = old_page;
2066                ret |= VM_FAULT_WRITE;
2067        } else
2068                mem_cgroup_uncharge_page(new_page);
2069
2070        if (new_page)
2071                page_cache_release(new_page);
2072        if (old_page)
2073                page_cache_release(old_page);
2074unlock:
2075        pte_unmap_unlock(page_table, ptl);
2076        if (dirty_page) {
2077                if (vma->vm_file)
2078                        file_update_time(vma->vm_file);
2088                wait_on_page_locked(dirty_page);
2089                set_page_dirty_balance(dirty_page, page_mkwrite);
2090                put_page(dirty_page);
2091        }
2092        return ret;
2093oom_free_new:
2094        page_cache_release(new_page);
2095oom:
2096        if (old_page)
2097                page_cache_release(old_page);
2098        return VM_FAULT_OOM;
2099
2100unwritable_page:
2101        page_cache_release(old_page);
2102        return VM_FAULT_SIGBUS;
2103}
```

Diagrama de flujo del manejador de fallos de página: