CAPITULO 12: SWAP

CAPITULO 12: SWAP....1
12.1 Introducción....2
12.2 Swap en Linux....3
12.3 Estructuras de datos....7
swap_info_struct....7
12.4 Funciones....10
swp_entry....10
swap_duplicate()....10
sys_swapon() 12
sys_swapoff()....19
try_to_unuse() 23
scan_swap_map()....26
get_swap_page() 28
swap_free() 30
La cache swap 31
12.5 Creación de un archivo de intercambio....39
12.6 Bibliográfia....40

12.1 Introducción

Históricamente se entiende por swapping la técnica mediante la cual se intercambia un proceso que está en memoria por otro que no lo está. Para esto se hace uso de un área de memoria conocida como de intercambio (swap). Actualmente no se intercambian procesos sino páginas de memoria.

Un sistema operativo puede disponer a nivel lógico de más memoria RAM que la que existe físicamente en el hardware que éste controla. Esto se consigue mediante lo que se conoce como memoria virtual.
Cuando se necesita memoria, y no hay más físicamente, el núcleo puede eliminar páginas de memoria para obtener espacio libre. Se tienen en cuenta varios aspectos:

Si la página a descartar es de sólo lectura, se elimina sin más.

Si es de escritura y no ha sido modificada, también se elimina.

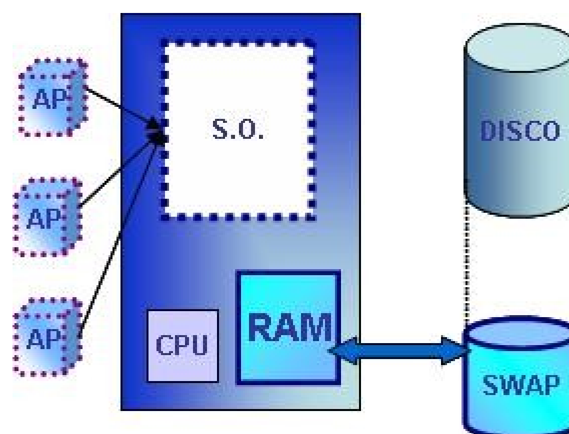
Si en cambio es de escritura y ha sido modificada, se debe almacenar su contenido antes de eliminarla, y así el proceso al que pertenece puede recuperarlo cuando lo requiera.

Si la página es la proyección de un fichero en memoria, se reescribe en el archivo.

Si no lo es, se guarda en la memoria de intercambio (dispositivo de swap).

El espacio de intercambio (swap) ofrece una reserva de espacio en el disco para aquellas páginas no mapeadas en memoria. Hay tres clases de páginas que se deben ser manejadas por el subsistema de intercambio:

- Las páginas que pertenecen a una región de memoria anónima de un proceso (User Mode stack or heap)
- Las páginas "Dirty" de un proceso que están mapeadas en memoria privada.
- Las páginas que pertenecen a la región de memoria compartida de un IPC.



Como la paginación por demanda, el swapping debe ser transparente a los programas. Es decir, no se necesita introducir ninguna instrucción especial relacionada con el intercambio en el código. Cada entrada de tabla de página

incluye una bandera que identifica donde se sitúa realmente la página (RAM, Swap o disco). Por tanto el núcleo utiliza esta bandera para señalar que una página que pertenecía a un espacio de dirección de proceso se ha intercambiado. Además de esa bandera, Linux también se aprovecha de los bits restantes de la entrada de tabla de página para almacenar en ellos un swapped-out page identifier (un identificador) que codifique la localización de la página descargada al almacenamiento swap del disco. Cuando ocurre una excepción de fallo de página, el manejador de la excepción puede detectar accediendo a la tabla de páginas que la página no está presente en la RAM e invocar la función que intercambia la página que falta del disco.

Las características principales y funciones de las que se encarga el sistema de intercambio pueden ser resumidas en:

- Crear las "zonas swap" en el disco para almacenar las páginas que no tienen una imagen en el disco.
- Manejar el espacio de las zonas de swap para asignar y liberar "marcos de página" según las necesidades.
- Proporcionar una función para realizar el intercambio de las páginas de la RAM a una zona swap “swap out” y otra función para intercambiar las páginas de una zona swap a la RAM “swap in”.
- Hacer uso los "swapped-out page identifiers" en las entradas de tabla de página, para saber en todo momento las páginas que se intercambian y así no perder de vista las posiciones de los datos en zona de memoria swap en un momento determinado.

La realización de un swapping puede estar deshabilitada (manteniendo inactivas las zonas swap que existan) o activa manteniendo alguna zona activada (funciones swapoff y swapon respectivamente).

Para tener una visión del uso de swap cabe destacar que su uso es básicamente para obtener un espacio adicional para la memoria principal, ampliando así el espacio de direcciones de memoria útiles para los procesos en modo usuario. De hecho, las zonas de swap grandes permiten que el núcleo lance varias peticiones de memoria que excedan la cantidad de RAM física instalada en el sistema. Sin embargo, la simulación de la RAM no es como una RAM física en términos del funcionamiento. Cada acceso para un proceso a una página que está siendo intercambiada de la RAM al espacio swap, es mayor que el acceso a una página de la RAM. Por tanto siempre será mucho mejor aumentar la RAM como solución al aumento de necesidades de un sistema, que utilizar el swapping y sólo debemos utilizar éste método como último recurso. Sin embargo hoy por hoy aún es recomendable el uso de swap, ya que nos permitirá en cualquier momento desalojar de RAM procesos de poca actividad dando paso a otros que requieran espacio en memoria principal.

12.2 Swap en Linux

Usa como memoria de intercambio dispositivos de bloque: ficheros, particiones de disco. Puede gestionar varios ficheros y/o particiones de swap. Permite

activar/desactivar espacios de swap mientras el sistema está en marcha (no hay que reiniciar). Al contrario que otros sistemas Unix, Linux puede funcionar sin memoria de swap.

Un dispositivo de swap se inicializa con el comando **mkswap**, se activa con **swapon** y se desactiva con **swapoff**.

Cada dispositivo de swap tiene asociada una prioridad. Cuando el sistema ha decidido guardar una página, se recorre la lista de dispositivos swap activos y la almacena en el de mayor prioridad que disponga de páginas sin asignar. Para dispositivos con igual prioridad, linux distribuye las páginas entre ellos mediante un esquema round-robin:

Mediante varios dispositivos swap se puede mejorar el rendimiento, ya que mientras se espera a que un dispositivo responda a una petición de página se pueden hacer peticiones a otros. Se pueden definir distintas zonas swap hasta un número máximo especificado por la macro MAX_SWAPFILES (fijada generalmente a 32).

Tener múltiples zonas swap permite a un administrador de sistema separar el espacio swap entre varios discos de modo que el hardware pueda actuar sobre ellos concurrentemente; además permite que la zona swap se aumente sin necesidad de reiniciar el sistema.

Se puede empeorar el rendimiento, ya que si dos dispositivos swap del mismo disco tienen la misma prioridad, las cabezas del disco tienen que moverse constantemente de un lado a otro, y consiguiendo el efecto contrario al pretendido.

El demonio **kswapd** se ejecuta desde el inicio del sistema y es el encargado de mantener el sistema de memoria funcionando eficientemente.

La mayor parte del tiempo, kswapd se duerme y se despierta cuando el núcleo se queda sin memoria. Entonces explora la lista de procesos e intenta descartar páginas no utilizadas. A fin de determinar las páginas en memoria no utilizadas, el núcleo utiliza el campo age del descriptor de página de memoria. Este contador se incrementa cuando se utiliza la página, y se decrementa cuando deja de utilizarse. Sólo las páginas que poseen un campo age nulo pueden descartarse de memoria.

Formato de los dispositivos swap:

- La primera página de memoria (4 kb en x86) del dispositivo contiene lo que se conoce como catálogo, que es una tabla de bits que indican si la página correspondiente es utilizable (0) o no (1).

- Los 10 últimos bytes del catálogo contienen la ristra SWAP_SPACE .

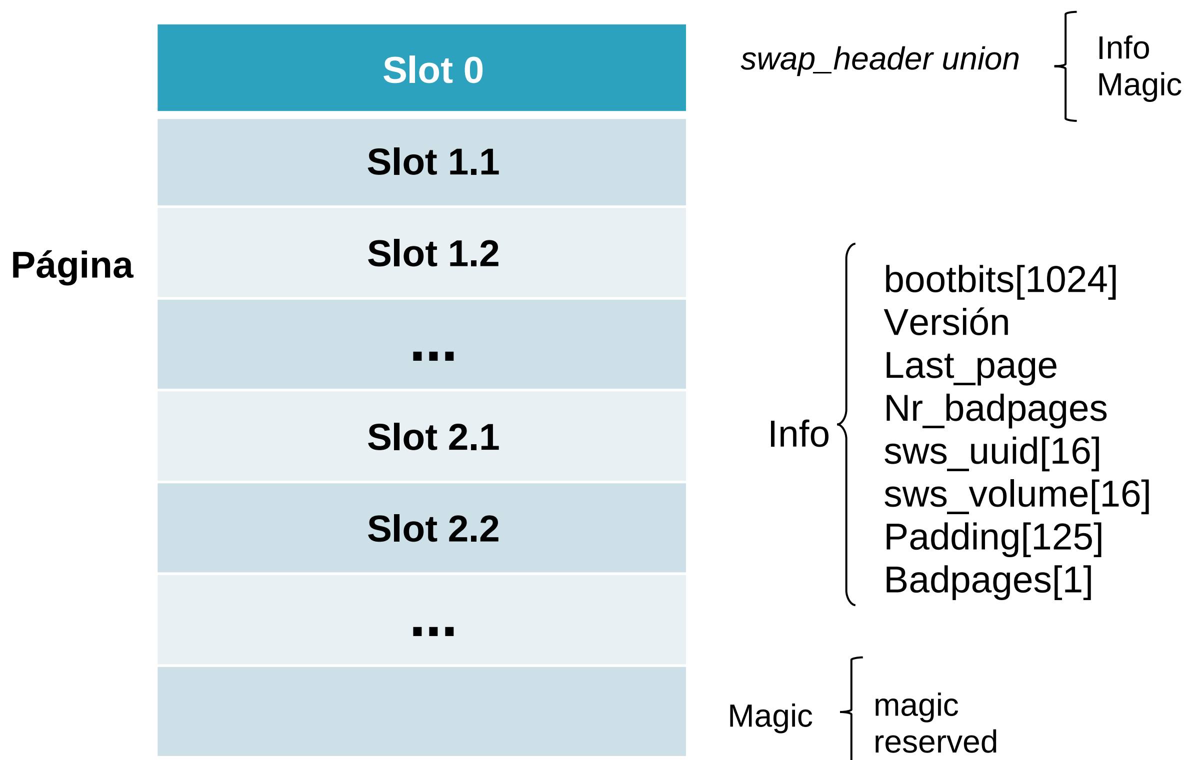
- El formato de un dispositivo swap determina su tamaño máximo:

4 Kb * 1024 bytes/Kb = 4096 bytes (tamaño del catálogo)

4096 bytes - 10 bytes = 4086 bytes (zona del catálogo que referencia páginas)

4086 bytes * 8 bits/byte = 32688 bits (número máximo de páginas)

32688 páginas * 4 kb/página = 130752 kb = 127.69 Mb (tamaño máximo de un dispositivo swap)



Cada zona swap está compuesta por una secuencia de slots (ranuras) de página: necesitamos un bloque de 4096 byte para contener una página descargada del almacenamiento de swap. El primer slot de la página de una zona swap se utiliza para almacenar información sobre la zona swap; su formato está descrito por swap_header union compuesta por dos estructuras, info y magic. La estructura Magic proporciona una marca que indica que parte del disco se usa como zona swap; consiste en apenas un campo, magic.magic, que contiene una secuencia "mágica" de 10 caracteres. La estructura magic esencialmente permite que el núcleo identifique inequívocamente un archivo o una partición como zona swap; el campo es "SWAP-SPACE" y está situado siempre al final del primer slot de la página.

La estructura del *info* incluye los campos siguientes:

- **bootbits:** No se usan en el algoritmo de intercambio; este campo corresponde a los primeros 1.024 octetos de la zona swap, que puede almacenar los datos de la partición, etiquetas del disco, etcétera.
- **Versión**: Versión del algoritmo de intercambio
- **Last_page:** El último slot de la página que se puede usar.
- **Nr_badpages:** Número de slots por página defectuosos.
- **sws_uuid[16].**
- **sws_volume[16].**
- **Padding[125]:** Padding bytes (bytes de acolchado no es una muy buena traducción).
- **Badpages[1]:** Hasta 637 números que especifican la localización de las ranuras de página defectuosas.

Crear y activar una zona swap

Los datos almacenados en una zona swap son importantes mientras el sistema está encendido. Cuando se apaga el sistema, se matan todos los procesos, así que los datos almacenados por procesos en zonas de swap se desechan. Por esta razón, las zonas de swap contienen muy poca información de control: esencialmente, el tipo de la zona swap y la lista de los slots de página defectuosos. Esta información de control se puede almacenar en una sola página de 4 KB.

Generalmente, el administrador del sistema crea una partición del intercambio al crear las otras particiones en el sistema de Linux, y después utiliza el comando *mkswap* para crear un área de disco como una nueva zona swap. El comando inicializa los campos con la información obtenida del primer slot de página. El disco puede incluir algunos slots erróneos, por lo que el programa también examina el resto de los slots de página para localizar los defectuosos. Al ejecutar el comando ***mkswap*** se deja la zona swap en un estado inactivo. Cada zona swap se puede activar con un script en el fichero cargador del sistema o dinámicamente después de que el sistema esté funcionando.

Cada zona swap consiste en unos o más bloques de intercambio "*swap extents*", cada uno de los cuales es representado por un descriptor *swap_extent*. Cada bloque corresponde a un grupo de páginas o más exactamente, a slots de página que son físicamente adyacentes en disco. Por lo tanto, el descriptor *swap_extent* indica el índice del bloque (que contiene la primera página) en la zona swap, la longitud de las páginas en el bloque, y el número del sector del disco en el que comienza el bloque. Al activarse una zona swap se crea una lista con los bloques que componen la zona. Una zona swap almacenada en una partición del disco se compone de apenas un bloque; inversamente, una zona swap almacenada en un archivo se puede componer de varios bloques, porque pueden que no estén en bloques contiguos en el disco.

Cómo distribuir las páginas en las zonas de swap

Al intercambiar de RAM a espacio swap, se intenta almacenar las páginas en slots contiguos para reducir al mínimo el tiempo de la búsqueda en el disco al acceder a la zona swap; éste es un elemento importante en el algoritmo de intercambio.

Sin embargo, si se utiliza más de una zona swap, las cosas son más complicadas. A las áreas de swap más rápidas se les suele asignar una prioridad más alta. Al buscar un slot libre, la búsqueda comienza en la zona swap que tiene la prioridad más alta. Si hay varias de ellas con la misma prioridad, se seleccionan de forma rotatoria para evitar la sobrecarga de una zona determinada, es decir, las zonas se van turnando (Round Robin). Si no se encuentra ningún slot libre en las zonas de swap que tienen la prioridad más alta, se busca en las zonas swap que tienen la prioridad más alta excluyendo las anteriores zonas.

12.3 Estructuras de datos

swap_info_struct

Cada zona swap activa tiene su propio descriptor del *swap_info_struct* en memoria. Los campos del descriptor se ilustran en la siguiente tabla.

Type	Field	Description
unsigned int	Flags	Swap area flags
struct file *	swap_file	Pointer to the file object of the regular file or device file that stores the swap area
struct block_device *	Bdev	Descriptor of the block device containing the swap area
struct list head	extent_list	Head of the list of extents that compose the swap area
struct swap_extent *	curr_swap_extent	Pointer to the most recently used extent descriptor
unsigned int	old_block_size	Natural block size of the partition containing the swap area
unsigned short *	swap_map	Pointer to an array of counters, one for each swap area page slot
unsigned int	lowest_bit	First page slot to be scanned when searching for a free one
unsigned int	highest_bit	Last page slot to be scanned when searching for a free one
unsigned int	cluster_next	Next page slot to be scanned when searching for a free one
unsigned int	cluster_nr	Number of free page slot allocations before restarting from the beginning
int	Prio	Swap area priority
int	Pages	Number of usable page slots

Type	Field	Description
unsigned long	Max	Size of swap area in pages
unsigned long	inuse_pages	Number of used page slots in the swap area
int	Next	Pointer to next swap area descriptor

El campo *flags* incluye tres subcampos:

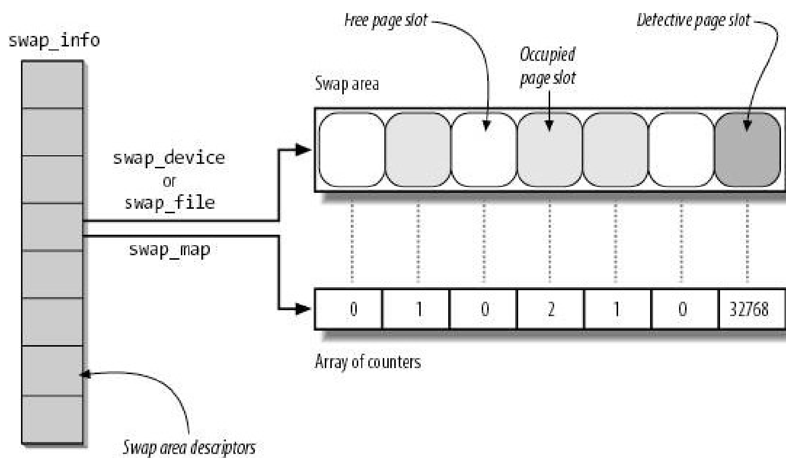
- ***SWP_USED*:** 1 si la zona swap está activa; 0 si está inactivo.
- ***SWP_WRITEOK*:** 1 si es posible escribir en la zona swap; o 0 si sólo es de lectura.
- ***SWP_ACTIVE*:** Este campo de 2 bits es la combinación de *SWP_USED* y de *SWP_WRITEOK*; se fija una bandera cuando ambas están *previamente activadas.*
- ***SWP_SCANNING:*** contador de referencias en el rastreo del mapa de swap (scan_swap_map)

El campo *swap_map* señala a un vector de contadores, uno para cada slot de página de la zona swap. Si el contador es igual a 0, el slot de página está libre; si es positivo, el slot de página está lleno con una página descargada al almacenamiento swap. Esencialmente, el contador denota el número de los procesos que comparten la página descargada al almacenamiento swap. Si el contador tiene el valor *SWAP_MAP_MAX* (igual a 32, 767), la página almacenada en el slot de página es "permanente" y no se puede quitar del correspondiente slot. Si el contador tiene el valor *SWAP_MAP_BAD* (igual a 32,768), el slot de página se considera defectuoso, y así inutilizable.

Los slots de página "permanentes" protegen al sistema contra desbordamientos de los contadores del *swap_map*.

El campo *prio* es un entero con signo que denota el orden en el cual el sistema del intercambio debe considerar cada zona swap, la prioridad. .

El vector *swap_info* incluye descriptores *MAX_SWAPFILES* de la zona swap. Solamente se utilizan las áreas que *SWP_USED* señala por medio de una bandera, porque son las áreas activadas. El cuadro 17-6 ilustra el vector *swap_info*, una zona swap, y el vector correspondiente de contadores.



Los descriptores de las zonas de swap activas también se insertan en una lista clasificada por la prioridad de la zona swap. La lista es implementada a través del campo *next* del descriptor de la zona swap, que almacena el índice del siguiente descriptor en el *array swap_info*. Este uso del campo como índice es diferente de la mayoría de los campos con el nombre *next*, que son generalmente indicadores.

La variable *swap_list*, del tipo *swap_list_t*, incluye los campos siguientes:

- *Head* : primer elemento de la lista en el swap_info

- *Next*: índice del vector *swap_info* de la zona swap siguiente que se seleccionará para el intercambio de las páginas. Este campo se utiliza para implementar el algoritmo Round Robin para elegir la zonas de swap de la máxima prioridad con slots libres.

El campo *max* del descriptor de la zona swap almacena el número de páginas de la zona swap, mientras que el campo *pages* almacena el número de slots usables por página. Estos números se diferencian en que el campo *page*s no cuenta el primer slot de página ni los slots de página defectuosas.

Identificador de la página descargada al almacenamiento swap

Esta página se identifica especificando la entrada del índice de la zona swap *swap_info* y el slot de página dentro de la zona swap. Porque la primera página (con el índice 0) de la zona swap se reserva para la *swap_header union*, luego el primer slot útil de la página tiene índice 1. El formato de un identificador descargado al almacenamiento swap de una página es:

12.4 Funciones

swp_entry

La función ***swp_entry****(type, offset)* construye un identificador para una página descargada al almacenamiento swap, utilizando el índice de la zona swap *type* y del índice de slot de página *offset*.

Inversamente, las funciones *swp_type* y *swp_offset* extraen del identificador de la página descargada al almacenamiento auxiliar, el índice de la zona swap y el índice del slot de página, respectivamente.

Cuando una página se intercambia de RAM a espacio swap, se inserta su identificador en la tabla de página así que la página puede ser encontrada cuando sea necesario. El bit menos significativo del identificador corresponde a la actual bandera *Present*, y denota el hecho de que la página no está actualmente en RAM. Sin embargo, por lo menos uno de los 31 bits restantes tiene que estar a 1 ya que no se almacena ninguna página en el slot 0 de la zona swap 0. Por lo tanto podemos identificar tres casos en el valor de una entrada en la tabla de página:

- La entrada nula: la página no pertenece al espacio de dirección del proceso, o el marco de página subyacente todavía no se ha asignado al proceso (paginación por demanda).
- Los primeros 31 bits más significativos son distintos a 0, el último bit es igual a 0: la página se está intercambiando de RAM al espacio swap.
- El bit menos significativo igual a 1: la página está contenida en RAM.

El tamaño máximo de una zona swap es determinado por el número de bits disponibles para identificar un slot. En la arquitectura 80x86, los 24 bits disponibles el tamaño de una zona swap a 2^{24} slots (es decir, a 64 GB).

Una página puede pertenecer a los espacios de dirección de varios procesos, puede ser intercambiada hacia el espacio swap del espacio de dirección de un proceso y todavía permanecer en memoria central; por lo tanto, es posible intercambiar la misma página varias veces. Una página se intercambia físicamente y se almacena una vez, y por cada intercambio hacia el espacio swap aumenta el contador del *swap_map*.

swap_duplicate()

La función ***swap_duplicate()*** se invoca generalmente cuando se intenta intercambiar una página ya descargada al almacenamiento swap. Verifica simplemente que el identificador de página descargado al almacenamiento swap pasado como parámetro sea válido y aumenta el contador *swap_map* correspondiente. Más exactamente, realiza las acciones siguientes:

1. Utiliza las funciones *swp_type* y *swp_offset* para extraer el número de zona swap y el índice de slots de página para el parámetro.

2.Comprueba si el número de identificación de zona swap está activo; si no, devuelve 0 (identificador inválido).
3.Comprueba si el slot de página es válido y no está libre (su contador *swap_map* es mayor de 0 y menor que *SWAP_MAP_BAD*); si no, devuelve 0 (identificador inválido).
4.Si no, el identificador de página descargada al almacenamiento swap establece una página válida. Aumenta el contador *swap_map* del slot de página si no ha alcanzado ya el valor *SWAP_MAP_MAX*.
5.Devuelve 1 (identificador válido).
6.

Codigo:

```
1778int swap_duplicate(swp_entry_t entry)
1779{
1780        struct swap_info_struct * p;
1781        unsigned long offset, type;
1782        int result = 0;
1783
1784        if (is_migration_entry(entry))
1785                return 1;
1786
1787        type = swp_type(entry);
1788        if (type >= nr_swapfiles)
1789                goto bad_file;
1790        p = type + swap_info;
1791        offset = swp_offset(entry);
1792
1793        spin_lock(&swap_lock);
1794        if (offset < p->max && p->swap_map[offset]) {
1795                if (p->swap_map[offset] < SWAP_MAP_MAX - 1) {
1796                        p->swap_map[offset]++;
1797                        result = 1;
1798                } else if (p->swap_map[offset] <= SWAP_MAP_MAX) {
1799                        if (swap_overflow++ < 5)
1800                                printk(KERN_WARNING "swap_dup:
swap entry overflow\n");
1801                        p->swap_map[offset] = SWAP_MAP_MAX;
1802                        result = 1;
1803                }
1804        }
1805        spin_unlock(&swap_lock);
1806out:
1807        return result;
1808
1809bad_file:
1810        printk(KERN_ERR "swap_dup: %s%08lx\n", Bad_file, entry.-
val);
1811        goto out;
1812}
1813
1814struct swap_info_struct *
1815get_swap_info_struct(unsigned type)
1816{
1817        return &swap_info[type];
1818}
```

Activar y desactivar una zona swap

Una vez que se inicializa una zona swap, el súper usuario (o, más exacto, cada usuario que tiene la capacidad de *CAP_SYS_ADMIN*) pueden utilizar los programas del *swapon* y del *swapoff* para activar y desactivar la zona swap, respectivamente. Estos programas utilizan *swapon()* y llamadas del sistema *swapoff();* comentaremos brevemente las rutinas del servicio correspondientes.

Las funciones y rutinas de servicio explicadas a continuación se localizan en /mm/swapfile.c

sys_swapon()

La rutina de servicio *sys_swapon()* recibe los siguientes parámetros:

- **Specialfile:** este parámetro señala al pathname (en el espacio de dirección del modo usuario) del archivo del dispositivo (partición) o del archivo usado para poner la zona swap en ejecución.
- **swap_flags**: este parámetro consiste en un solo bit *SWAP_FLAG_PREFER* más 31 bist de prioridad de la zona swap (estos bits son significativos solamente si el bit *SWAP_FLAG_PREFER* está a 1).

La función chequea los campos de *swap_header union* que fue puesta en el primer slot cuando la zona swap fue creada. La función realiza estos pasos:

1. Comprueba que el proceso actual tenga la capacidad *CAP_SYS_ADMIN*.
2. Mira en los primeros componentes *nr_swapfiles* del vector *swap_info* de descriptores de zona swap para el primer descriptor que tiene la bandera *SWP_USED* desactivada, esto significa que la zona swap correspondiente está inactiva. Si se encuentra una zona swap inactiva, ir al paso 4.
3. El nuevo índice del vector de zona swap es igual a los *nr_swapfiles*: comprueba que el número de bits reservados para el índice de la zona swap sea suficientemente grande para codificar el nuevo índice; si no, devuelve un código de error; si es suficientemente grande, aumenta en uno el valor de *nr_swapfiles*.
4. Se ha encontrado un índice de zona swap que no está siendo utilizado: se inicializa los campos del descriptor; más específicamente, se fija flags a *SWP_USED*.
5. Copia la secuencia señalada por al parámetro *specialfile* del espacio de dirección de modo usuario.
6. Invoca el *filp_open()* para abrir el archivo especificado por el parámetro *specialfile* .
7. Almacena las direcciones del objeto archivo devuelto por el *filp_open()* en el campo *swap_file* del descriptor de la zona swap.
8. Comprueba que la zona swap no esté activada mirando las otras zonas de swap activas en *swap_info*. Esto se hace comprobando las direcciones de los objetos del *address_space* almacenados en el campo *swap_file->f_mapping* de los descriptores de la zona swap.
9. Si la zona swap está ya activa, devuelve un código de error. Si el parámetro *specialfile* identifica un archivo del dispositivo en modo bloque, realiza los siguientes pasos:

a.Invoca *bd_claim()* para fijar el subsistema de intercambio como el sostenedor del dispositivo en modo bloque . Si el dispositivo en modo bloque tiene ya un sostenedor, devuelve un código de error.

b.Almacena la dirección del descriptor del dispositivo en modo bloque en el campo del *bdev* del descriptor de la zona swap.

c.Almacena el tamaño del bloque actual del dispositivo en el campo *old_block_size* del descriptor de la zona swap, después fija el tamaño de bloque del dispositivo a 4.096 octetos (el tamaño de la página).

10.Si el parámetro *specialfile* identifica un archivo regular, realiza los pasos siguientes:

a.Comprueba el campo *S_SWAPFILE* de los campos *i_flags* del inode del archivo. Si está fijada esta bandera, devuelve un código de error porque el archivo se está utilizando ya como zona swap.

b.Almacena la dirección del descriptor del dispositivo en modo bloque del fichero en el campo *bdev* del descriptor de la zona swap.

11.Lee el descriptor *swap_header* almacenado en el slot 0 de la zona swap. A tal efecto, invoca *read_cache_page()* que se le pasa como parámetros el objeto *address_space* señalado por *swap_file->f_mapping*, el índice 0 de la página, la dirección del método *readpage* del archivo (almacenado en *swap_file->f_mapping->a_ops->readpage*), y el objeto que apunta al archivo *swap_file*. Espera hasta que la página se ha leído en memoria.

12.Comprueba que los últimos 10 caracteres de la cadena “mágica” pasadas de la primera página sean igual a "*SWAPSPACE2*." Si no, devuelve un código de error.

13.Inicializa los campos *lowest_b*it y *highest_bit* del descriptor de la zona swap según el tamaño de la zona swap almacenada en el campo de *info.last_pa*ge de la estructura *swap_header*.

14.Invoca *vmalloc*() para crear el vector de contadores asociado a la nueva zona swap y almacena su dirección en el campo *swap_map* del descriptor del intercambio. Inicializa los elementos del vector a 0 o a *SWAP_MAP_BAD*, según la lista de los sltos de página defectuosos almacenados en el campo *info.bad_pages* de la unión *swap_h*eader.

15.Computa el número de slots de página útiles accediendo a los campos *info.last_page* y *info.nr_badpages* en el primer slot de página, y lo almacena en el campo *pages* del descriptor de la zona swap. También fija el campo *max* con el número total de páginas en la zona swap.

16.Construye la lista *extent_list* de los grados del intercambio para la nueva zona swap (solamente una si la zona swap es una partición del disco), y fija correctamente los *nr_extents* y los campos *curr_swap_extent* en el descriptor de la zona swap.

17.Si el parámetro *swap_flags* especifica una prioridad para la nueva zona swap, la función fija el campo del *prio* del descriptor. Si no, inicializa el campo a la prioridad más baja entre todas las zonas de swap activas menos 1. Si no hay otras zonas de swap activas, la función asigna el valor -1.

18.Fija el campo *flags* del descriptor de la zona swap a *SWP_ACTIVE*.

19.Actualiza los *nr_good_pages*, los *nr_swap_pages*, y las variables globales de los *total_swap_pages*.

20.Inserta el descriptor de la zona swap en la lista que es apuntado por la variable *swap_list*.

21.Devuelve 0 (éxito).

Codigo:

```
1470SYSCALL_DEFINE2(swapon, const char __user *, specialfile, int,
swap_flags)
1471{
1472         struct swap_info_struct * p;
1473         char *name = NULL;
1474         struct block_device *bdev = NULL;
1475         struct file *swap_file = NULL;
1476         struct address_space *mapping;
1477         unsigned int type;
1478         int i, prev;
1479         int error;
1480         union swap_header *swap_header = NULL;
1481         int swap_header_version;
1482         unsigned int nr_good_pages = 0;
1483         int nr_extents = 0;
1484         sector_t span;
1485         unsigned long maxpages = 1;
1486         int swapfilesize;
1487         unsigned short *swap_map = NULL;
1488         struct page *page = NULL;
1489         struct inode *inode = NULL;
1490         int did_down = 0;
1491
1492         if (!capable(CAP_SYS_ADMIN))
1493                 return -EPERM;
1494         spin_lock(&swap_lock);
1495         p = swap_info;
1496         for (type = 0 ; type < nr_swapfiles ; type++,p++)
1497                 if (!(p->flags & SWP_USED))
1498                         break;
1499         error = -EPERM;
1500         if (type >= MAX_SWAPFILES) {
1501                 spin_unlock(&swap_lock);
1502                 goto out;
1503         }
1504         if (type >= nr_swapfiles)
1505                 nr_swapfiles = type+1;
1506         memset(p, 0, sizeof(*p));
1507         INIT_LIST_HEAD(&p->extent_list);
1508         p->flags = SWP_USED;
1509         p->next = -1;
1510         spin_unlock(&swap_lock);
1511         name = getname(specialfile);
1512         error = PTR_ERR(name);
1513         if (IS_ERR(name)) {
1514                 name = NULL;
1515                 goto bad_swap_2;
1516         }
1517         swap_file = filp_open(name, O_RDWR|O_LARGEFILE, 0);
1518         error = PTR_ERR(swap_file);
1519         if (IS_ERR(swap_file)) {
1520                 swap_file = NULL;
1521                 goto bad_swap_2;
1522         }
1523
```

```
1524         p->swap_file = swap_file;
1525         mapping = swap_file->f_mapping;
1526         inode = mapping->host;
1527
1528         error = -EBUSY;
1529         for (i = 0; i < nr_swapfiles; i++) {
1530                 struct swap_info_struct *q = &swap_info[i];
1531
1532                 if (i == type || !q->swap_file)
1533                         continue;
1534                 if (mapping == q->swap_file->f_mapping)
1535                         goto bad_swap;
1536         }
1537
1538         error = -EINVAL;
1539         if (S_ISBLK(inode->i_mode)) {
1540                 bdev = I_BDEV(inode);
1541                 error = bd_claim(bdev, sys_swapon);
1542                 if (error < 0) {
1543                         bdev = NULL;
1544                         error = -EINVAL;
1545                         goto bad_swap;
1546                 }
1547                 p->old_block_size = block_size(bdev);
1548                 error = set_blocksize(bdev, PAGE_SIZE);
1549                 if (error < 0)
1550                         goto bad_swap;
1551                 p->bdev = bdev;
1552         } else if (S_ISREG(inode->i_mode)) {
1553                 p->bdev = inode->i_sb->s_bdev;
1554                 mutex_lock(&inode->i_mutex);
1555                 did_down = 1;
1556                 if (IS_SWAPFILE(inode)) {
1557                         error = -EBUSY;
1558                         goto bad_swap;
1559                 }
1560         } else {
1561                 goto bad_swap;
1562         }
1563
1564         swapfilesize = i_size_read(inode) >> PAGE_SHIFT;
1565
1566         /*
1567          * Read the swap header.
1568          */
1569         if (!mapping->a_ops->readpage) {
1570                 error = -EINVAL;
1571                 goto bad_swap;
1572         }
1573         page = read_mapping_page(mapping, 0, swap_file);
1574         if (IS_ERR(page)) {
1575                 error = PTR_ERR(page);
1576                 goto bad_swap;
1577         }
1578         kmap(page);
1579         swap_header = page_address(page);
1580
1581         if (!memcmp("SWAP-SPACE",swap_header->magic.magic,10))
1582                 swap_header_version = 1;
1583         else if (!memcmp("SWAPSPACE2",swap_header->magic.magic,
10))
```

```
1584                    swap_header_version = 2;
1585            else {
1586                    printk(KERN_ERR "Unable to find swap-space signa-
ture\n");
1587                    error = -EINVAL;
1588                    goto bad_swap;
1589            }
1590
1591            switch (swap_header_version) {
1592            case 1:
1593                    printk(KERN_ERR "version 0 swap is no longer sup-
ported. "
1594                            "Use mkswap -v1 %s\n", name);
1595                    error = -EINVAL;
1596                    goto bad_swap;
1597            case 2:
1598                    /* swap partition endianess hack... */
1599                    if (swab32(swap_header->info.version) == 1) {
1600                            swab32s(&swap_header->info.version);
1601                            swab32s(&swap_header->info.last_page);
1602                            swab32s(&swap_header->info.nr_badpages);
1603                            for (i = 0; i < swap_header->info.nr_bad-
pages; i++)
1604                                    swab32s(&swap_header->info.bad-
pages[i]);
1605                    }
1606                    /* Check the swap header's sub-version and the
size of
                       the swap file and bad block lists */
1607
1608                    if (swap_header->info.version != 1) {
1609                            printk(KERN_WARNING
1610                                   "Unable to handle swap header ver-
sion %d\n",
1611                                   swap_header->info.version);
1612                            error = -EINVAL;
1613                            goto bad_swap;
1614                    }
1615
1616                    p->lowest_bit  = 1;
1617                    p->cluster_next = 1;
1618
1619                    /*
1620                     * Find out how many pages are allowed for a sin-
gle swap
1621                     * device. There are two limiting factors: 1) the
number of
1622                     * bits for the swap offset in the swp_entry_t
type and
1623                     * 2) the number of bits in the a swap pte as de-
fined by
1624                     * the different architectures. In order to find
the
1625                     * largest possible bit mask a swap entry with
swap type 0
1626                     * and swap offset ~0UL is created, encoded to a
swap pte,
1627                     * decoded to a swp_entry_t again and finally the
swap
1628                     * offset is extracted. This will mask all the
bits from
```

```
1629                     * the initial ~0UL mask that can't be encoded in
either
1630                     * the swp_entry_t or the architecture definition
of a
1631                     * swap pte.
1632                     */
1633                    maxpages = swp_offset(pte_to_swp_en-
try(swp_entry_to_pte(swp_entry(0,~0UL)))) - 1;
1634                    if (maxpages > swap_header->info.last_page)
1635                            maxpages = swap_header->info.last_page;
1636                    p->highest_bit = maxpages - 1;
1637
1638                    error = -EINVAL;
1639                    if (!maxpages)
1640                            goto bad_swap;
1641                    if (swapfilesize && maxpages > swapfilesize) {
1642                            printk(KERN_WARNING
1643                                   "Swap area shorter than signature
indicates\n");
1644                            goto bad_swap;
1645                    }
1646                    if (swap_header->info.nr_badpages && S_ISREG(in-
ode->i_mode))
1647                            goto bad_swap;
1648                    if (swap_header->info.nr_badpages > MAX_SWAP_BAD-
PAGES)
1649                            goto bad_swap;
1650
1651                    /* OK, set up the swap map and apply the bad block
list */
1652                    swap_map = vmalloc(maxpages * sizeof(short));
1653                    if (!swap_map) {
1654                            error = -ENOMEM;
1655                            goto bad_swap;
1656                    }
1657
1658                    error = 0;
1659                    memset(swap_map, 0, maxpages * sizeof(short));
1660                    for (i = 0; i < swap_header->info.nr_badpages; i+
+) {
1661                            int page_nr = swap_header->info.bad-
pages[i];
1662                            if (page_nr <= 0 || page_nr >= swap_head-
er->info.last_page)
1663                                    error = -EINVAL;
1664                            else
1665                                    swap_map[page_nr] = SWAP_MAP_BAD;
1666                    }
1667                    nr_good_pages = swap_header->info.last_page -
1668                                    swap_header->info.nr_badpages -
1669                                    1 /* header page */;
1670                    if (error)
1671                            goto bad_swap;
1672            }
1673
1674            if (nr_good_pages) {
1675                    swap_map[0] = SWAP_MAP_BAD;
1676                    p->max = maxpages;
1677                    p->pages = nr_good_pages;
1678                    nr_extents = setup_swap_extents(p, &span);
1679                    if (nr_extents < 0) {
```

```
1680                        error = nr_extents;
1681                        goto bad_swap;
1682                }
1683                nr_good_pages = p->pages;
1684        }
1685        if (!nr_good_pages) {
1686                printk(KERN_WARNING "Empty swap-file\n");
1687                error = -EINVAL;
1688                goto bad_swap;
1689        }
1690
1691        mutex_lock(&swapon_mutex);
1692        spin_lock(&swap_lock);
1693        if (swap_flags & SWAP_FLAG_PREFER)
1694                p->prio =
1695                  (swap_flags & SWAP_FLAG_PRIO_MASK) >>
SWAP_FLAG_PRIO_SHIFT;
1696        else
1697                p->prio = --least_priority;
1698        p->swap_map = swap_map;
1699        p->flags = SWP_ACTIVE;
1700        nr_swap_pages += nr_good_pages;
1701        total_swap_pages += nr_good_pages;
1702
1703        printk(KERN_INFO "Adding %uk swap on %s.  "
1704                        "Priority:%d extents:%d across:%lluk\n",
1705                nr_good_pages<<(PAGE_SHIFT-10), name, p->prio,
1706                nr_extents, (unsigned long
long)span<<(PAGE_SHIFT-10));
1707
1708        /* insert swap space into swap_list: */
1709        prev = -1;
1710        for (i = swap_list.head; i >= 0; i = swap_info[i].next) {
1711                if (p->prio >= swap_info[i].prio) {
1712                        break;
1713                }
1714                prev = i;
1715        }
1716        p->next = i;
1717        if (prev < 0) {
1718                swap_list.head = swap_list.next = p - swap_info;
1719        } else {
1720                swap_info[prev].next = p - swap_info;
1721        }
1722        spin_unlock(&swap_lock);
1723        mutex_unlock(&swapon_mutex);
1724        error = 0;
1725        goto out;
1726bad_swap:
1727        if (bdev) {
1728                set_blocksize(bdev, p->old_block_size);
1729                bd_release(bdev);
1730        }
1731        destroy_swap_extents(p);
1732bad_swap_2:
1733        spin_lock(&swap_lock);
1734        p->swap_file = NULL;
1735        p->flags = 0;
1736        spin_unlock(&swap_lock);
1737        vfree(swap_map);
1738        if (swap_file)
```

```
1739                filp_close(swap_file, NULL);
1740out:
1741        if (page && !IS_ERR(page)) {
1742                kunmap(page);
1743                page_cache_release(page);
1744        }
1745        if (name)
1746                putname(name);
1747        if (did_down) {
1748                if (!error)
1749                        inode->i_flags |= S_SWAPFILE;
1750                mutex_unlock(&inode->i_mutex);
1751        }
1752        return error;
1753}
```

sys_swapoff()

La rutina de servicio *sys_swapoff()* desactiva una zona swap identificada por el parámetro *specialfile*. Es mucho más compleja y consume más tiempo que *sys_swapon()*, ya que la partición que se desactivará puede contener páginas pertenecientes a varios procesos. La función explora forzosamente la zona swap e intercambia todas las páginas existentes. Cada intercambio requiere un nuevo marco de página, en el caso en el que falle puede que no haya marcos de página libres contiguos a la izquierda. En este caso, la función devuelve un código de error. Para realizar esto se realizan los siguientes pasos principales:

1. Comprueba que el proceso actual tenga la capacidad de *CAP_SYS_ADMIN*.
2. Copia la secuencia señalada por al parámetro *specialfile* en espacio del kernel.
3. Invoca *filp_open()* para abrir el archivo referenciado por el parámetro *specialfile*; esta función devuelve la dirección de un objeto del archivo.
4. Explora la lista *swap_list* del descriptor de la zona swap, y compara la dirección del objeto del archivo devuelto por *filp_open()* con las direcciones almacenadas en los campos *swap_file* de los descriptores activos de la zona swap. Si no se encuentra ninguno, el parámetro pasado a la función es inválido, así que devuelve un código de error.
5. Invoca *cap_vm_enough_memory()* para comprobar si hay bastantes marcos de página libres para intercambiar todas las páginas almacenadas en la zona swap. Si no, la zona swap no puede ser desactivada; lanza el objeto del archivo y devuelve un código de error. Esto es solamente un chequeo, pero podría ahorrar al kernel mucho tiempo de actividad inútil en el disco. Mientras se está chequeando, *cap_vm_enough_memory()* busca los marcos de página asignados a través de la información obtenida de las banderas fijadas por *SLAB_RECLAIM_ACCOUNT* . El número de tales páginas, que se consideran como recuperables, se almacenan en las variables *slab_reclaim_pages*.
6. Quitar el descriptor de la zona swap de la lista del *swap_list*.
7. Actualizar los *nr_swap_pages* y las variables *total_swap_pages* quitando el valor del campo de *pages* del descriptor de la zona swap.

8.Limpia el valor de la bandera de *SWP_WRITEOK* en el campo *flags* del descriptor de la zona swap; esto prohíbe el PFRA para el intercambio de más páginas en la zona swap.
9.Invocar *try_to_unuse()* sucesivamente para forzar todas las páginas izquierdas en la zona swap de la RAM y para actualizar las tablas de página de los procesos que utilizan estas páginas. Mientras, el proceso actual, que está ejecutando el comando *swapoff*, tiene la bandera de *PF_SWAPOFF* fijada. Fijar esta bandera tiene apenas una consecuencia: en caso de que haya escasez de marcos de página, la función *select_bad_process()* forzará la muerte de este proceso.
10.Esperar hasta que se desactive el dispositivo que contiene la zona swap . De esta manera, las peticiones de lectura enviadas por *TRy_to_unuse()* serán dirigidas por el manejador antes de que se desactive la zona swap.
11.Si *TRy_to_unuse()* falla en el asignación de todos los marcos de página solicitados, la zona swap no puede ser desactivada. Por lo tanto, la función ejecuta los siguientes pasos:

- oReinserta el descriptor de la zona swap en la lista del *swap_list* y fija su campo *flag* a *SWP_WRITEOK*.
- oRestaura el contenido original de los *nr_swap_pages* y de las variables *total_swap_pages* agregando el valor en el campo pages del descriptor de la zona swap.
- oInvoca *filp_close()* para cerrar el archivo abierto en el paso y devuelve un código de error.

12.Si no, todos los slots de página se han transferido con éxito a RAM. Y se ejecutan los siguientes pasos:

- oLanzar las áreas de memoria usadas para almacenar el vector *swap_map* y los descriptores del grado.
- oSi la zona swap se almacena en una partición del disco, se restaura el tamaño de bloque a su valor original, que se almacena en el campo *old_block_size* del descriptor de la zona swap; por otra parte, se invoca la función *bd_release()* de modo que el subsistema de intercambio no alargue los bloques del dispositivo.
- oSi la zona swap se almacena en un archivo regular, se limpia la bandera de *S_SWAPFILE* del inode del archivo.
- oSe invoca *filp_close()* dos veces, la primera vez para el objeto *swap_file* del archivo, la segunda vez con el objeto devuelto por *filp_open()* en el paso 3.
- oDevolver 0.

Codigo:

```
1226SYSCALL_DEFINE1(swapoff, const char __user *, specialfile)
```

```
1227{
1228         struct swap_info_struct * p = NULL;
1229         unsigned short *swap_map;
1230         struct file *swap_file, *victim;
1231         struct address_space *mapping;
1232         struct inode *inode;
1233         char * pathname;
1234         int i, type, prev;
1235         int err;
1236
1237         if (!capable(CAP_SYS_ADMIN))
1238                 return -EPERM;
1239
1240         pathname = getname(specialfile);
1241         err = PTR_ERR(pathname);
1242         if (IS_ERR(pathname))
1243                 goto out;
1244
1245         victim = filp_open(pathname, O_RDWR|O_LARGEFILE, 0);
1246         putname(pathname);
1247         err = PTR_ERR(victim);
1248         if (IS_ERR(victim))
1249                 goto out;
1250
1251         mapping = victim->f_mapping;
1252         prev = -1;
1253         spin_lock(&swap_lock);
1254         for (type = swap_list.head; type >= 0; type =
swap_info[type].next) {
1255                 p = swap_info + type;
1256                 if ((p->flags & SWP_ACTIVE) == SWP_ACTIVE) {
1257                         if (p->swap_file->f_mapping == mapping)
1258                                 break;
1259                 }
1260                 prev = type;
1261         }
1262         if (type < 0) {
1263                 err = -EINVAL;
1264                 spin_unlock(&swap_lock);
1265                 goto out_dput;
1266         }
1267         if (!security_vm_enough_memory(p->pages))
1268                 vm_unacct_memory(p->pages);
1269         else {
1270                 err = -ENOMEM;
1271                 spin_unlock(&swap_lock);
1272                 goto out_dput;
1273         }
1274         if (prev < 0) {
1275                 swap_list.head = p->next;
1276         } else {
1277                 swap_info[prev].next = p->next;
1278         }
1279         if (type == swap_list.next) {
1280                 /* just pick something that's safe... */
1281                 swap_list.next = swap_list.head;
1282         }
1283         if (p->prio < 0) {
1284                 for (i = p->next; i >= 0; i = swap_info[i].next)
1285                         swap_info[i].prio = p->prio--;
1286                 least_priority++;
```

```
1287            }
1288            nr_swap_pages -= p->pages;
1289            total_swap_pages -= p->pages;
1290            p->flags &= ~SWP_WRITEOK;
1291            spin_unlock(&swap_lock);
1292
1293            current->flags |= PF_SWAPOFF;
1294            err = try_to_unuse(type);
1295            current->flags &= ~PF_SWAPOFF;
1296
1297            if (err) {
1298                    /* re-insert swap space back into swap_list */
1299                    spin_lock(&swap_lock);
1300                    if (p->prio < 0)
1301                            p->prio = --least_priority;
1302                    prev = -1;
1303                    for (i = swap_list.head; i >= 0; i =
swap_info[i].next) {
1304                            if (p->prio >= swap_info[i].prio)
1305                                    break;
1306                            prev = i;
1307                    }
1308                    p->next = i;
1309                    if (prev < 0)
1310                            swap_list.head = swap_list.next = p -
swap_info;
1311                    else
1312                            swap_info[prev].next = p - swap_info;
1313                    nr_swap_pages += p->pages;
1314                    total_swap_pages += p->pages;
1315                    p->flags |= SWP_WRITEOK;
1316                    spin_unlock(&swap_lock);
1317                    goto out_dput;
1318            }
1319
1320            /* wait for any unplug function to finish */
1321            down_write(&swap_unplug_sem);
1322            up_write(&swap_unplug_sem);
1323
1324            destroy_swap_extents(p);
1325            mutex_lock(&swapon_mutex);
1326            spin_lock(&swap_lock);
1327            drain_mmlist();
1328
1329            /* wait for anyone still in scan_swap_map */
1330            p->highest_bit = 0;             /* cuts scans short */
1331            while (p->flags >= SWP_SCANNING) {
1332                    spin_unlock(&swap_lock);
1333                    schedule_timeout_uninterruptible(1);
1334                    spin_lock(&swap_lock);
1335            }
1336
1337            swap_file = p->swap_file;
1338            p->swap_file = NULL;
1339            p->max = 0;
1340            swap_map = p->swap_map;
1341            p->swap_map = NULL;
1342            p->flags = 0;
1343            spin_unlock(&swap_lock);
1344            mutex_unlock(&swapon_mutex);
1345            vfree(swap_map);
```

```
1346        inode = mapping->host;
1347        if (S_ISBLK(inode->i_mode)) {
1348                struct block_device *bdev = I_BDEV(inode);
1349                set_blocksize(bdev, p->old_block_size);
1350                bd_release(bdev);
1351        } else {
1352                mutex_lock(&inode->i_mutex);
1353                inode->i_flags &= ~S_SWAPFILE;
1354                mutex_unlock(&inode->i_mutex);
1355        }
1356        filp_close(swap_file, NULL);
1357        err = 0;
1358
1359out_dput:
1360        filp_close(victim, NULL);
1361out:
1362        return err;
1363}
```

try_to_unuse()

La función *TRy_to_unuse()* actúa con el índice que identifica la zona swap que se vaciará; intercambia las páginas (de swap a RAM) y actualiza todas las tablas de página de los procesos que han intercambiado las páginas (de RAM a swap) en esta zona swap. La función visita los espacios de dirección de todos los hilos y procesos del kernel, comenzando con el descriptor de la memoria *init_mm* que se utiliza como marcador. Es una función que desperdicia mucho tiempo y funciona sobre todo con las interrupciones habilitadas. La sincronización con otros procesos es por lo tanto crítica.

La función *TRy_to_unuse()* explora el vector *swap_map* de la zona swap. Cuando la función encuentra un slot de página en uso, intercambia (de swap a RAM) la página, y después comienza a buscar los procesos que se refieren a la página. El ordenar estas dos operaciones es crucial para evitar condiciones de tipo. Mientras que la transferencia de datos I/O está en curso, la página es bloqueada, así que ningún proceso puede tener acceso a ella. Una vez que la transferencia de datos de I/O termine, la página es bloqueada otra vez por *try_to_unuse(),* así que no puede ser intercambiada hacia fuera otra vez por otra proceso. Las condiciones de tipo también se evitan porque cada proceso mira la cache la página antes de comenzar un intercambio, bien de RAM a zona swap o viceversa. Finalmente, la zona swap considerada por *try_to_unuse()* está marcada como no-escritura (la bandera *SWP_WRITEOK* no se fija), así que ningún proceso puede realizar un intercambio de RAM a swap en un slot de página de esta área.

Sin embargo, *try_to_unuse()* puede forzar la exploración de vector *swap_map* de los contadores usados de la zona swap varias veces. Esto es porque las regiones de memoria que contienen referencias a las páginas descargadas al almacenamiento swap pudieron desaparecer durante una exploración y reaparecer más adelante en las listas de procesos.
Por ejemplo, recuerde la descripción de la función del *do_munmap():* siempre que un proceso lance un intervalo de direcciones lineales, el *do_munmap()*

quita de la lista de procesos todas las regiones de memoria que incluyan las direcciones lineales afectadas; más adelante, la función reinserta las regiones de memoria que no han sido mapeadas parcialmente en la lista de procesos. El *do_munmap()* libera las páginas descargadas al almacenamiento swap que pertenecen al intervalo de direcciones lineales lanzadas. Es recomendable no liberar las páginas descargadas al almacenamiento swap que pertenecen a las regiones de memoria que tienen que ser reinsertadas en la lista de procesos.

Por lo tanto, *TRy_to_unuse()* puede fallar en encontrar un proceso que se refiera a un slot de página dada porque la región de memoria correspondiente no se incluye temporalmente en la lista de procesos. Para hacer frente a este hecho, *try_to_unuse()* guarda la información explorada del vector *swap_map* hasta que todos los contadores de la referencia son nulos. A veces, las regiones de memoria fantasmas que referencian las páginas descargadas al almacenamiento swap reaparecerán en las listas de proceso, así que *TRy_to_unuse()* tendrá éxito en liberar todos los slots de página.

Vamos ahora describir las operaciones principales ejecutadas por *TRy_to_unuse().* Ejecuta un bucle continuo sobre contadores de referencia del vector *swap_map* de la zona swap pasada como su parámetro. Se interrumpe este bucle y la función retorna un código de error si el proceso actual recibe una señal. Para cada contador de referencia, la función realiza los pasos siguientes:

1. Si el contador es igual a 0 (no se almacena ninguna página) o *SWAP_MAP_BAD*, continúa con el slot siguiente de la página.
2. Si no, se invoca la función *read_swap_cache_async()* para intercambiar de swap a RAM la página. Esto consiste en el asignar, en caso de necesidad, un nuevo marco de página, llenándolo con los datos almacenados en el slot de página, y poniendo la página en la cache swap.
3. Esperar hasta que la página nueva se haya actualizado y bloquearla.
4. Mientras la función del paso anterior se estaba ejecutando, el proceso habría podido ser suspendido. Por lo tanto, se comprueba otra vez si el contador de la referencia del slot de página es nulo; si es así esta página de intercambio ha sido liberada por otro proceso de control del kernel, así que la función continúa con el slot siguiente de la página.
5. Invocar *unuse_process()* en cada descriptor de la memoria en la lista doble de links cuya cabecera es *init_mm* . Esta función desperdicia mucho tiempo en explorar todas las entradas de tabla de página del proceso que posee el descriptor de la memoria, y sustituye cada ocurrencia del identificador descargado al almacenamiento swap de la página por la dirección física del marco de página. Para reflejar este movimiento, la función también disminuye el contador de slot de la página en el vector *swap_map* (a menos que sea igual a *SWAP_MAP_MAX*) y aumenta el contador de uso del marco de página.
6. Invocar *shmem_unuse()* para comprobar si la página descargada al almacenamiento swap es utilizada como recurso de memoria compartido IPC .
7. Comprobar el valor del contador de la referencia de la página. Si es igual a *SWAP_MAP_MAX*, el slot de página es "permanente." Para liberarlo, se pone el valor 1 en el contador de la referencia.

8. La cache swap puede poseer la página también (esto contribuye al valor del contador de la referencia). Si la página pertenece a la cache swap, se invoca la función *swap_writepage()* para limpiar su contenido (si la página es dirty) y se invoca el *delete_from_swap_cache()* para borrar la página de la cache swap y para disminuir su contador de la referencia.
9. Se fija la bandera *PG_dirty* del descriptor de página, se abre el marco de página, y se disminuye su contador de referencia (deshacer el incremento hecho en el paso 5).
10. Se comprueba el campo *need_resched* del proceso actual; si está fijado, se invoca el *schedule()* para abandonar a la CPU. Desactivar una zona swap es un trabajo largo, y el núcleo debe asegurarse de que los otros procesos en el sistema todavía continúen ejecutándose. La función *try_to_unuse()* continúa desde este paso siempre que el proceso sea seleccionado otra vez por el planificador.
11. Continuar con el siguiente slot de página, comenzando en el paso 1.

La función continúa hasta que cada contador de la referencia en el vector *swap_map* es nulo. Recuerde que incluso si la función comienza a examinar el slot de página siguiente, el contador de referencia del slot de página anterior podría todavía ser positivo. De hecho, un proceso "fantasma" podría aún referenciar la página, normalmente porque algunas regiones de memoria se han quitado temporalmente de la lista de proceso explorada en el paso 5. Eventualmente, *try_to_unuse()* coge cada referencia. Sin embargo, la página no es más larga en la cache swap, es desbloqueada, y una copia se incluye en el slot de página de la zona swap que es desactivada.

Uno pudo contar con que esta situación pueda conducir a la pérdida de los datos. Por ejemplo, suponga que algunos procesos "fantasmas" acceden al slot de página y comienza el intercambio de la página. Debido a que la página no es más larga en la cache swap, el proceso llena un nuevo marco de página con los datos leídos de disco. Sin embargo, este marco de página sería diferente de los marcos de página poseídos por los procesos que se suponen que comparten la página con el proceso "fantasma".

Este problema no se presenta cuando se desactiva la zona swap, porque un proceso fantasma podría interferir sólo cuando una página descargada al almacenamiento swap pertenece a una memoria anónima, mapeada y privada. En este caso, el marco de página se maneja por medio de Copy On Write, así que es perfectamente legal asignar diversos marcos de página a los procesos que se refieren a la página. Sin embargo, la función *try_to_unuse()* marca la página como "dirty" ; si no, la función *shrink_list()* puede quitar más adelante la página de la tabla de página de un cierto proceso sin el ahorro en otra zona swap.

Asignar y lanzar un slot de página

Como veremos más adelante, cuando se libera memoria, el kernel intercambia (de RAM a swap) muchas páginas en un período del tiempo corto. Es por lo tanto importante intentar almacenar estas páginas en slots contiguos para reducir al mínimo el tiempo de búsqueda en el disco cuando tenemos acceso a la zona swap.

Un primer acercamiento a un algoritmo de búsqueda de un slot libre podría ser elegir uno de dos casos:

1. Comenzar siempre desde el principio de la zona swap. Este planteamiento puede aumentar el tiempo medio de la búsqueda durante las operaciones de descarga al almacenamiento swap, porque los slots de página libres se pueden estar lejos unos de otros.
2. Compenzar siempre desde el slot de página anterior. Este algoritmo aumenta el tiempo medio de búsqueda durante las operaciones de descarga al almacenamiento swap, porque el grupo de slots de página ocupados pueden estar dispersos.

Linux adopta una mezcla de los dos, una búsqueda híbrida. Empieza siempre con el slot de página que fue asignado en la vez pasada a menos que ocurra una de estas condiciones:

- Se alacanza el final de la zona swap.
- Los sltos de página libres *SWAPFILE_CLUSTER* (generalmente 256) fueron asignados después del recomenzar desde el principio de la zona swap.

El campo *cluster_nr* en el descriptor *swap_info_struct* almacena el número de los slots de página libres asignados. Este campo se reajusta a 0 cuando la función recomienza la asignación desde el principio de la zona swap. El campo *cluster_next* almacena el índice del primer slots de página que se examinará en la siguiente asignación.

Para acelerar la búsqueda de los slots de página libres, el núcleo guarda los campos del *lowest_bit* y del *highest_bit* de cada descriptor de la zona swap actualizado. Estos campos especifican el primer y último slots de página que podrían estar libre; es decir cada slots de página debajo del *lowest_bit* y sobre *highest_bit* puede ser ocupado.

scan_swap_map()

La función del *scan_swap_map()* se utiliza para encontrar un slot de página libre en una zona swap dada. Actúa con un solo parámetro, que señala a un descriptor de la zona swap y devuelve el índice de un slot de página libre. Devuelve 0 si la zona swap no contiene ningún slot libre. La función realiza los pasos siguientes:

1. Intenta primero utilizar el cluster actual. Si el campo *cluster_nr* del descriptor de la zona swap es positivo, se explora el vector *swap_map* de contadores que empiezan con el elemento del índice *cluster_next* y busca una entrada nula. Si se encuentra una entrada nula, disminuye el campo del *cluster_nr* y va al paso 4.
2. Si se alcanza este punto, es porque el campo *cluster_nr* le falta información o la búsqueda no encontró una entrada nula en el vector *swap_map*. Es el momento de intentar la segunda etapa de la búsqueda híbrida. La función reinicializa el *cluster_nr* a *SWAPFILE_CLUSTER* y comienza de nuevo la exploración del vector del índice *lowest_bit* que

intenta encontrar un grupo de slots de página libres *SWAPFILE_CLUSTER*. Si encuentran a tal grupo, va al paso 4.
3.Si ningún grupo de slots de página libres de *SWAPFILE_CLUSTER* existe. La función recomienza la exploración del vector del índice del *lowest_bit* que intenta encontrar un solo slot de página libre. Si no se encuentra ninguna entrada nula, fija el campo *lowest_bit* al índice máximo en el vector, el campo del highest_bit a 0, y devulve 0 (la zona swap está llena).
4.Se encuentra una entrada nula. Pone el valor 1 en la entrada, decrementa nr_swap_pages, actualiza los campos del *lowest_bit* y del *highest_bit* si es necesario, aumenta el campo de los *inuse_pages* en uno, y fija el campo del *cluster_next* al índice del slot de página asignado más 1.
5.Devuelve el índice del slot de página asignado.

Código:

```
  89static inline unsigned long scan_swap_map(struct swap_info_struct
*si)
  90{
  91        unsigned long offset, last_in_cluster;
  92        int latency_ration = LATENCY_LIMIT;
  93
  94        /*
  95         * We try to cluster swap pages by allocating them sequen-
tially
  96         * in swap.  Once we've allocated SWAPFILE_CLUSTER pages
this
  97         * way, however, we resort to first-free allocation,
starting
  98         * a new cluster.  This prevents us from scattering swap
pages
  99         * all over the entire swap partition, so that we reduce
 100         * overall disk seek times between swap pages.  -- sct
 101         * But we do now try to find an empty cluster.  -Andrea
 102         */
 103
 104        si->flags += SWP_SCANNING;
 105        if (unlikely(!si->cluster_nr)) {
 106                si->cluster_nr = SWAPFILE_CLUSTER - 1;
 107                if (si->pages - si->inuse_pages < SWAPFILE_CLUS-
TER)
 108                        goto lowest;
 109                spin_unlock(&swap_lock);
 110
 111                offset = si->lowest_bit;
 112                last_in_cluster = offset + SWAPFILE_CLUSTER - 1;
 113
 114                /* Locate the first empty (unaligned) cluster */
 115                for (; last_in_cluster <= si->highest_bit; offset+
+) {
 116                        if (si->swap_map[offset])
 117                                last_in_cluster = offset + SWAP-
FILE_CLUSTER;
 118                        else if (offset == last_in_cluster) {
 119                                spin_lock(&swap_lock);
```

```
 120                            si->cluster_next = offset-SWAP-
FILE_CLUSTER+1;
 121                            goto cluster;
 122                    }
 123                    if (unlikely(--latency_ration < 0)) {
 124                            cond_resched();
 125                            latency_ration = LATENCY_LIMIT;
 126                    }
 127            }
 128            spin_lock(&swap_lock);
 129            goto lowest;
 130    }
 131
 132    si->cluster_nr--;
 133cluster:
 134    offset = si->cluster_next;
 135    if (offset > si->highest_bit)
 136lowest:        offset = si->lowest_bit;
 137checks: if (!(si->flags & SWP_WRITEOK))
 138            goto no_page;
 139    if (!si->highest_bit)
 140            goto no_page;
 141    if (!si->swap_map[offset]) {
 142            if (offset == si->lowest_bit)
 143                    si->lowest_bit++;
 144            if (offset == si->highest_bit)
 145                    si->highest_bit--;
 146            si->inuse_pages++;
 147            if (si->inuse_pages == si->pages) {
 148                    si->lowest_bit = si->max;
 149                    si->highest_bit = 0;
 150            }
 151            si->swap_map[offset] = 1;
 152            si->cluster_next = offset + 1;
 153            si->flags -= SWP_SCANNING;
 154            return offset;
 155    }
 156
 157    spin_unlock(&swap_lock);
 158    while (++offset <= si->highest_bit) {
 159            if (!si->swap_map[offset]) {
 160                    spin_lock(&swap_lock);
 161                    goto checks;
 162            }
 163            if (unlikely(--latency_ration < 0)) {
 164                    cond_resched();
 165                    latency_ration = LATENCY_LIMIT;
 166            }
 167    }
 168    spin_lock(&swap_lock);
 169    goto lowest;
 170
 171no_page:
 172    si->flags -= SWP_SCANNING;
 173    return 0;
 174}
```

get_swap_page()

La función *get_swap_page()* se utiliza para encontrar un slot de página libre buscando en todas las zonas de swap activas. La función, devuelve el identificador de la página intercambiada de un slot de página asignado o un 0 si todas las areas de swap están llenas, para lo que se toma en consideración las diversas prioridades de las zonas de swap activas.

El primer paso es parcial y se aplica solamente a las áreas que tienen una sola prioridad; la función busca tales áreas usando Round Robin. Si no se encuentra ningún slot de página libre, se hace un segundo paso, se comienza desde el principio de la lista de la zona swap; durante este segundo paso, se examinan todas las zonas de swap. Más exactamente, la función realiza los pasos siguientes:

1. Si los *nr_swap_pages* son nulos o si no hay zonas de swap activas, devuelve 0.
2. Se comienza teniendo en cuenta el puntero a la zona swap señalada por *swap_list.next*.
3. Si la zona swap está activa, se invoca *scan_swap_map()* para asignar un slot de página libre. Si el *scan_swap_map()* devuelve un índice de el slot de página, se prepara para su siguiente invocación. Es decir, se actualiza *swap_list.next* para señalar a la zona swap siguiente en la lista de la zona swap, si el último tiene la misma prioridad se sigue utilizando el método Round Robin. Si la zona swap siguiente no tiene la misma prioridad que la actual, la función fija *swap_list.next* a la primera zona swap en la lista (de modo que la búsqueda siguiente comience con las zonas de swap que tienen la prioridad más alta). La función acaba devolviendo el identificador descargado al almacenamiento swap de la página que corresponde al slot de página asignada.
4. O la zona swap no tiene posibilidad de escritura, o no tiene slots de página libres. Si la zona swap siguiente en la lista de la zona swap tiene la misma prioridad que la actual, la función la hace actual y va al paso 3.
5. En este punto, la zona swap siguiente en la lista de la zona swap tiene una prioridad más baja que la anterior. El paso siguiente depende de cuáles de los dos pasos está realizando la función.
    a. Si éste es el primer paso (parcial), considera la primera zona swap en la lista y va al paso 3, comenzando el segundo paso.
    b. Si no, comprueba si hay un elemento siguiente en la lista; si es así lo considera y va al paso 3.
6. En este punto la lista ha sido explorada totalmente por el segundo paso y no se ha encontrado ninguna slot de página libre; se devuelve 0.

Código:

```
 176swp_entry_t get_swap_page(void)
 177{
 178         struct swap_info_struct *si;
 179         pgoff_t offset;
 180         int type, next;
 181         int wrapped = 0;
 182
 183         spin_lock(&swap_lock);
```

```
 184         if (nr_swap_pages <= 0)
 185                 goto noswap;
 186         nr_swap_pages--;
 187 
 188         for (type = swap_list.next; type >= 0 && wrapped < 2; type_
= next) {
 189                 si = swap_info + type;
 190                 next = si->next;
 191                 if (next < 0 ||
 192                     (!wrapped && si->prio != 
swap_info[next].prio)) {
 193                         next = swap_list.head;
 194                         wrapped++;
 195                 }
 196 
 197                 if (!si->highest_bit)
 198                         continue;
 199                 if (!(si->flags & SWP_WRITEOK))
 200                         continue;
 201 
 202                 swap_list.next = next;
 203                 offset = scan_swap_map(si);
 204                 if (offset) {
 205                         spin_unlock(&swap_lock);
 206                         return swp_entry(type, offset);
 207                 }
 208                 next = swap_list.next;
 209         }
 210 
 211         nr_swap_pages++;
 212noswap:
 213         spin_unlock(&swap_lock);
 214         return (swp_entry_t) {0};
 215}
```

swap_free()

La función *swap_free()* es invocada cuando se intercambia una página para disminuir el *swap_map* correspondiente al contador. Cuando el contador alcanza 0, el slot de página será libre puesto que su identificador no está incluido en ninguna entrada de la tabla de página.

La función tiene un parámetro *entry* que especifica el identificador de la página intercambiada y realiza los pasos siguientes:

1. Deriva el índice de la zona swap y el índice de desplazamiento *offset* del slot de página desde el parámetro *entry* y toma la dirección del descriptor de area swap.
2. Comprueba si la zona swap está activa y devuelve si lo está o no.
3. Si el contador *swap_map* correspondiente con el slot de página que ha sido liberada es más pequeño que *SWAP_MAP_MAX*, la función lo disminuye. Recuerde que las entradas que tienen el valor de *SWAP_MAP_MAX* están consideradas como persistentes (undeletable).
4. Si el contador *swap_map* es 0, la función aumenta el valor de *nr_swap_pages*, disminuye el campo de los *inuse_pages*, y actualiza, en

caso de necesidad, el *lowest_bit* y los campos del *highest_bit* del descriptor de la zona swap.

Código:

```
 298void swap_free(swp_entry_t entry)
 299{
 300        struct swap_info_struct * p;
 301
 302        p = swap_info_get(entry);
 303        if (p) {
 304                swap_entry_free(p, swp_offset(entry));
 305                spin_unlock(&swap_lock);
 306        }
 307}
```

La cache swap

La transferencia de páginas desde o hacia la cache swap son una actividad que puede inducir muchas condiciones de tipo. En particular, el subsistema de intercambio debe manejar cuidadosamente los casos siguientes:

- **Múltiples intercambios desde swap a RAM. (swap-ins):** dos procesos pueden intentar concurrentemente intercambiar la misma página anónima compartida.
- **Procesos concurrentes de swap-ins (intercambios entre swap y RAM) y de swap-outs (intercambios de RAM a swap):** un proceso puede descargar una página desde la zona swap hasta RAM y esta página a su vez puede que esté siendo intercambiada hacia la zona swap por el PFRA.

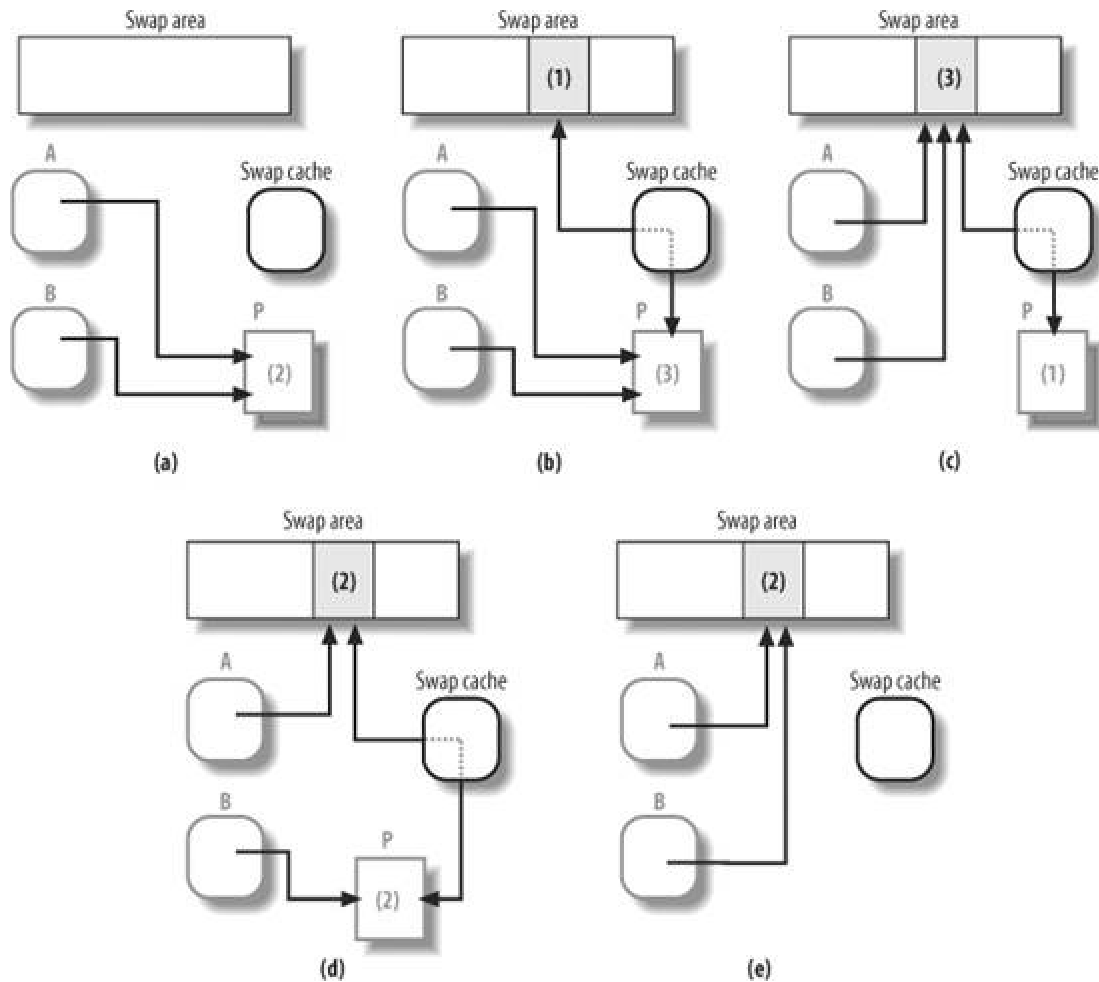
La cache swap se ha introducido para solucionar estas clases de problemas de sincronización. La regla dominante es que nadie puede comenzar un intercambio sin la comprobación de si la cache incluye ya la página afectada. Gracias a la cache swap, las operaciones concurrentes del intercambio que afectan a la misma página actúan siempre en el mismo marco de página; por lo tanto, el núcleo puede confiar con seguridad en la bandera *PG_locked* del descriptor de la página para evitar cualquier condición de tipo.

Por ejemplo, considere dos procesos que compartan la misma página descargada al almacenamiento swap. Cuando el primer proceso intenta tener acceso a la página, el núcleo comienza la operación de la migración. El primer paso consiste en comprobar si el marco de página está incluido ya en el la cache swap. Vamos a suponer que no está: entonces, el núcleo asigna un nuevo marco de página y lo inserta en la cache swap; después, comienza la operación de I/O para leer el contenido de la página de la zona swap. Mientras tanto, el segundo proceso tiene acceso a la página anónima compartida. Como arriba, el núcleo comienza una operación de migración y comprueba si el marco de página afectado está incluido ya en la cache swap. Ahora, está incluido, así que el núcleo tiene acceso simplemente al descriptor del marco de

página y pone el proceso actual a dormir hasta que la bandera *PG_locked* ya no esté activada, es decir, hasta que la transferencia de datos de I/O termina.

La cache swap desempeña un papel crucial también cuando concurrentemente se descarga una página a la memoria o al almacenamiento swap. La función *shrink_list()* comienza el intercambio de una página anónima (de RAM a espacio swap) solamente si *TRy_to_unmap()* tiene éxito en quitar el marco de página de las tablas de página en modo usuario de todos los procesos que posean la página. Sin embargo, un proceso puede tener acceso a la página y causar una migración (de swap a RAM) mientras que el intercambio de RAM al almacenamiento swap está todavía está en marcha.

Antes de ser escrito en el disco, cada página que se intercambia hacia el espacio swap es almacenada en la cache swap por el *shrink_list().* Considere una página P que se comparta entre dos procesos, A y B. Inicialmente, las entradas de tabla de página de ambos procesos contienen una referencia al marco de página, y la página tiene dos dueños; este caso se ilustra en el cuadro (a). Cuando el PFRA selecciona la página reclamada, *shrink_list()* inserta el marco de página en la cache swap. Según lo ilustrado en el cuadro (b), ahora el marco de página tiene tres dueños, mientras que el slot de página en la zona swap es referenciada solamente por la cache swap. Después, el PFRA invoca el *try_to_unmap()* para quitar las referencias al marco de página de la tabla de página de los procesos; una vez que esta función termine, el marco de página es referido solamente por la cache swap, mientras que el slot de página es referida por los dos procesos y la cache swap, según lo ilustrado a la figura(c). Vamos a suponer que, mientras que el contenido de la página se está escribiendo al disco, es el proceso B el que tiene acceso a la página, he intenta tener acceso a una celda de memoria usando una dirección lineal dentro de la página. Entonces, el manejador de fallos de página encuentra el marco de página en la cache swap y pone detrás su dirección física en la entrada de tabla de página del proceso B, como se ve en la figura(d). Inversamente, si la operación de descargar al almacenamiento swap termina sin operaciones concurrentes de migración, la función del *shrink_list()* borra el marco de página de la cache swap y lanza el marco de página al Buddy system, ver(e).



Usted puede considerar la cache swap como área de tránsito que contiene los descriptores de página de las páginas anónimas que se están descargando desde o hacia la memoria. Cuando la migración o descarga al almacenamiento swap termina, el descriptor de página de la página anónima se quita de la swap cache

Implementación de la cache swap

La cache swap está implementada con las estructuras y los procedimientos de la cache de páginas. Recuerde que la base de la cache de páginas es un sistema de árboles que permite que el algoritmo derive rápidamente a la dirección de un descriptor de la página de la dirección de un objeto del *address_space* que identifica al dueño de la página así como de un valor compensado.

Las páginas en la cache swap se almacenan como una página en la cache de páginas, con el tratamiento especial siguiente:

- El campo *mapping* del descriptor de la página se fija a *NULL*.
- Se fija la bandera *PG_swapcache* del descriptor de la página.

- El campo *private* almacena el identificador de la página descargada al almacenamiento swap.

Por otra parte, cuando la página se pone en la cache swap, el campo *count* del descriptor de página y los slots de página usados como contadores se incrementan, porque la cache swap utiliza tanto el marco de página y como el slot de página.

Finalmente, un solo espacio de dirección del *swapper_space* se utiliza para todas las páginas en la cache swap, así que solo la raíz del árbol es señalado por un puntero *swapper_space.page_tree* que direcciona las páginas en la cache swap. El campo *nrpages* del espacio de dirección *swapper_space* almacena el número de las páginas contenidas en la cache swap.

Las funciones de ayuda de la cache swap

El núcleo utiliza varias funciones para manejar la cache swap. Demostramos más adelante cómo estas funciones relativamente bajas son invocadas por funciones de alto nivel para intercambiar las páginas según lo necesitado.
Las funciones principales que manejan la cache swap son:

- ***lookup_swap_cache()*** : encuentra una página en la cache swap a través de su identificador de la página pasado como parámetro y devuelve la dirección del descriptor de la página. Devuelve 0 si la página no está en la cache. Para encontrar la página requerida, invocamos *radix_tree_lookup(),* pasando como parámetros un indicador a la raíz del árbol *swapper_space.page_treethe* y el identificador de página descargada al almacenamiento auxiliar.
- ***add_to_swap_cache()*** : inserta una página en la cache swap. Esencialmente invoca el *swap_duplicate()* para comprobar si el slot de página pasado como parámetro es válido y aumenta el contador del uso del slot de página; entonces, invoca el *radix_tree_insert()* para insertar la página en la cache; finalmente, aumenta el contador de la referencia de la página y fija las banderas de *PG_swapcache* y de *PG_locked.*
- ***_ _add_to_swap_cache():*** similar al *add_to_swap_cache(),* con la diferencia de que la función no invoca al *swap_duplicate()* antes de insertar el marco de página en la cache swap.
- ***delete_from_swap_cache()*** : quita una página la cache swap invocando el *radix_tree_delete(*), disminuye el contador correspondiente del uso en swap_map, y disminuye el contador de la referencia de la página.
- ***free_page_and_swap_cache():*** quita una página de la cache swap si ningún proceso en modo del usuario además del actual se está refiriendo al slot de página correspondiente, y disminuye el contador del uso de la página.
- ***free_pages_and_swap_cache():*** análogo al *free_page_and_swap_cache(),* pero funciona en un sistema de páginas dado.
- ***free_swap_and_cache():*** libera una entrada de intercambio, y comprueba si la página referida por la entrada está en la cache swap. Si ningún proceso está en modo usuario, además del actual, se está refiriendo a la

página o más del 50% de las entradas están ocupadas, la función quita la página de la cache swap.

Intercambio de RAM a espacio swap de páginas

Insertar el marco de página en la cache swap
El primer paso de una operación de descarga al almacenamiento swap consiste en preparar la cache swap. Si la función *shrink_list()* determina que una página es anónima (el *PageAnon()* retorna un 1) y que la cache swap no incluye el marco de página correspondiente (la bandera *PG_swapcache* en el descriptor de la página está desactivada), el núcleo invoca la función *add_to_swap().*

La función *add_to_swap()* asigna un slot de página nuevo en una zona swap e inserta una dirección del descriptor de la página del marco de página que se pasa como parámetro a la cache swap. Esencialmente, la función realiza los pasos siguientes:

1. Invocar *get_swap_page()* para asignar un slot de página nueva. Devuelve 0 en caso de que falle (por ejemplo, no se encuentra ningún slot de página libre).
2. Invocar _ _ *add_to_page_cache(),* pasándole el índice de el slot de página, la dirección del descriptor de la página, y algunas banderas de asignación.
3. Fijar las banderas de *PG_uptodate* y de *PG_dirty* en el descriptor de la página, de modo que la función *shrink_list()* sea forzada a escribir la página en el disco.
4. Devuelve 1 (éxito).

Actualizar las entradas de la tabla de páginas.
Una vez que el *add_to_swap()* termine, el *shrink_list()* invoca el *try_to_unmap(),* que determina la dirección de cada entrada de tabla de página en modo usuario que refiere a la página anónima y escribe en él un identificador de página.

Escribir la página el area de intercambio (swap area)

La siguiente acción que se realizará para terminar la descarga al almacenamiento swap consiste en escribir el contenido de la página en la zona swap. Esta transferencia de I/O es activada por la función *shrink_list(),* que comprueba si la bandera de *PG_dirty* del marco de página está activa y por lo tanto ejecuta la función *pageout().*

La función *pageout()* instala un descriptor *writeback_control* e invoca el método *writepage* del objeto de página *address_space*. El método *writepage* se pone en ejecución por la función *swap_writepage().*

La función *swap_writepage(),* alternadamente, realiza esencialmente los pasos siguientes:

1. Comprueba si por lo menos un proceso en modo usuario hace referencia a la página. Si no, quita la página de la cache swap y devuelve 0. Este

chequeo es necesario porque un proceso pudo competir con el PRFA y lanzar una página después del chequeo realizado por el *shrink_list().*

2. Invoca *get_swap_bio()* para asignar y para inicializar un bio descriptor. La función deriva la dirección del descriptor de la zona swap del identificador de la página intercambiada; entonces, se recorre las listas del grado del intercambio para determinar el sector inicial del disco del slot de página. El bio descriptor incluirá un pedido a una sola página de datos (el slot de página); el método finaliza con la función *end_swap_bio_write().*
3. Fija la bandera de *PG_writeback* en el descriptor de la página. Por otra parte, la función reajusta la bandera de *PG_locked*.
4. Invoca *submit_bio(),* pasándole el comando de *WRITE* y la dirección del descriptor *bio*.
5. Devuelve 0.

Una vez que la transferencia de datos de I/O termine, se ejecuta la función *end_swap_bio_write().* Esta función despierta cualquier proceso que estuviera en espera hasta que es la bandera de *PG_writeback* de la página es desactivada, desactiva la bandera de *PG_writeback* y las etiquetas correspondientes en el árbol de la raíz, y lanza el *bio* descriptor usado para la transferencia de I/O.

Borrado del marco de página de la cache swap

El último paso para la operación migrar una página de RAM a almacenamiento swap lo realiza una vez más el *shrink_list():* si verifica que ningún proceso haya intentado tener acceso al marco de página mientras que se hacía la transferencia de datos de I/O, se invoca el *delete_from_swap_cache()* para quitar el marco de página de la cache swap. Debido a que la cache swap era el único dueño de la página, el marco de página se lanza al sistema buddy.

El intercambio de páginas de zona swap a RAM

El proceso de migración de las páginas del espacio swap a RAM ocurre cuando un proceso hace referencia a una página que ha sido intercambiada fuera del disco. El manejador de la excepción de fallo de página comienza la migración (de swap a RAM) cuando ocurren las condiciones siguientes:

- La página incluyendo la dirección que causó la excepción es válida, pertenece a una región de memoria del proceso actual.
- La página no está presente en memoria, es decir es, la actual bandera *Present* en la entrada de tabla de página está desactivada.
- La entrada de tabla de página asociada a la página no es nula, pero el bit *dirty* está a 0; esto significa que la entrada contiene un identificador de página descargada al almacenamiento swap.

Si todas las condiciones anteriores se cumplen, el *handle_pte_fault()* invoca la función *do_swap_page()* para intercambiar la página requerida.

La función do_swap_page()

La función *do_swap_page()* actúa con los parámetros siguientes:

- **Mm : l**a dirección del descriptor de la memoria del proceso que causó la excepción fallo de página.
- **Vma**: la dirección del descriptor de la región de memoria que incluye la dirección *address.*
- **Address**: dirección lineal que causa a excepción
- **Page_table:** dirección de la entrada en la tabla de página indicada en *address*
- **Pmd:** dirección del Page Middle Directory que indica *address*
- **Orig_pte:** contenido de la entrada de la tabla de paginas indicado por *address*
- **Write_access**: Bandera que denota si el acceso es de escritura o lectura

Diferente al resto de funciones expuestas, *do_swap_page()* nunca devuelve 0. Devuelve 1 si la página está ya en la cache swap (fallo sin importancia), 2 si la página fue leída en la zona swap (fallo importante), y -1 si ocurrió un error mientras se realizaba la migración. Se ejecutan los pasos siguientes:

1. Consigue el identificador de la página descargada al almacenamiento swap de *orig_pte*.
2. Invoca el *pte_unmap()* para lanzar cualquier núcleo temporal indicado por la tabla de página creada por la función *handle_mm_fault().*
3. Lanza la cerradura del *page_table_lock* del descriptor de la memoria (fue adquirido por el *handle_pte_fault()*).
4. Invoca *lookup_swap_cache()* para comprobar si la cache swap contiene ya una página que corresponde al identificador de página descargado al almacenamiento swap; si la página está ya en la cache, salta al paso 6.
5. Invoca la función *swapin_readahead()* para leer en la zona swap un grupo de al menos 2n páginas, incluyendo solicitada. El valor n se almacena en la variable *page_cluster*, y es generalmente igual a 3.
6. Invoca *read_swap_cache_async()* una vez más para intercambiar la página del proceso que causó el fallo de página. Este paso puedo parecer redundante, pero no lo es realmente. La función *swapin_readahead()* pudo fallar en la lectura de la página solicitada por ejemplo, porque el *page_cluster* estaba a 0 o porque la función intenta leer un grupo de páginas incluyendo un slot de página libre o un slot de página defectuoso (*SWAP_MAP_BAD*). Por otra parte, si el *swapin_readahead()* ha tenido éxito, esta invocación del *read_swap_cache_async()* termina rápidamente porque encuentra la página en la cache swap.
7. Si, a pesar de todos los esfuerzos, la página solicitada no fue agregada a la cache swap, otro proceso del control del kernel pudo haber intercambiado ya dentro la página solicitada. Este caso es comprobado temporalmente adquiriendo la cerradura del *page_table_lock* y comparando la entrada *page_table* con el *orig_pte*. Si son diferentes, la página ha sido intercambiada ya dentro por un proceso del control del núcleo, así que la función devuelve 1 (avería de menor importancia); si no, vuelve -1 (falta).
8. En este punto, sabemos que la página está en la cache swap. Si la página se ha intercambiado con éxito del espacio de swap a la RAM, la

función invoca *grab_swap_token()* para intentar coger el símbolo del intercambio .

9. Invoca el *mark_page_accessed()* y bloquea la página.
10. Adquiere la cerradura de *page_table_lock*.
11. Comprueba si otro proceso del control del núcleo ha intercambiado adentro la página solicitada en nombre de una copia de este proceso. En este caso, lanza la cerradura del *page_table_lock*, abre la página, y devuelve 1 (avería de menor importancia).
12. Invoca el *swap_free()* para disminuir el contador del uso del slot de página que corresponde a la entrada.
13. Comprueba si la cache swap está llena por lo menos el 50 por ciento (los *nr_swap_pages* son más pequeños que la mitad de *total_swap_pages*). Si es así, comprueba si la página está poseída solamente por el proceso que causó el fallo (o uno de sus clones); si éste es el caso, quita la página de la cache swap.
14. Aumenta el campo de los *rss* del descriptor de la memoria del proceso.
15. Actualiza las entradas de la tabla de página para que el proceso puede encontrar la página. La función logra esto escribiendo la dirección física de la página solicitada y los bits de protección que se encuentran en el campo *vm_page_prot* de la región de memoria en la entrada de tabla de página tratada por *page_table*. Por otra parte, si el acceso que causó el fallo fue una escritura y el proceso que causó el fallo es el dueño único de la página, la función también fija la bandera *dirty* y la bandera de Read/Write para prevenir una Copy On Write inútil.
16. Desbloquea la página.
17. Invoca el *page_add_anon_rmap()* para insertar la página anónima en object-based indicado en la estructura de datos.
18. Si el parámetro *write_access* es igual a 1, la función invoca el *do_wp_page()* para hacer una copia del marco de página .
19. Lanza la cerradura *mm->page_table_lock* y devuelve el código *ret*: 1 (avería de menor importancia) o 2 (avería importante).

La función read_swap_cache_async()

La función *read_swap_cache_async()* se invoca siempre que el núcleo deba intercambiar una página del espacio swap a RAM. Necesita de tres parámetros:

- **Entry:** un identificador de página obtenido del proceso de intercambio *swapped-out* (intercambio de RAM a espacio swap)
- **Vma:** un punter a una región de memoria que conteniene la página.
- **Addr:** una dirección lineal de la página.

Como sabemos, antes de tener acceso a la partición del intercambio, la función debe comprobar si la cache swap incluye ya el marco de página deseado. Por lo tanto, la función ejecuta las operaciones siguientes:

1. Invoca el *radix_tree_lookup()* para localizar en el árbol que contiene los objetos *swapper_space* un marco de página en la posición dada por la variable *entry*. Si se encuentra la página, aumenta su contador de referencia y devuelve la dirección de su descriptor.
2. Si la página no se incluye en la cache swap (no se encuentra). Se invoca el *alloc_pages()* para asignar un nuevo marco de página. Si no hay un

marco de página libre disponible, devuelve 0 (indica que el sistema está fuera de memoria).
3.Invoca el *add_to_swap_cache()* para insertar el descriptor de la página del nuevo marco de página en la cache swap. Según lo mencionado en la sección anterior, esta función también bloquea la página.
4.El paso anterior pudo fallar si el *add_to_swap_cache()* encuentra un duplicado de la página en la cache swap. Por ejemplo, el proceso en el paso 2 podía bloquear permitiendo que otro proceso comience una operación de migración en el mismo slot de página. En este caso, se lanza el marco de página asignado en el paso 2 y se vuelve al paso 1.
5.Invoca el *lru_cache_add_active()* para insertar la página en la lista activa de LRU .
6.El descriptor del nuevo marco de página ahora está en la cache swap. Se invoca el *swap_readpage()* para leer el contenido de la página de la zona swap. Esta función es absolutamente similar al *swap_writepage()* descrito en la sección anterior : desactiva la bandera de *PG_uptodate* del descriptor de la página, invoca el *get_swap_bio()* para asignar y para inicializar un bio descriptor para la transferencia de I/O, e invoca el *submit_bio()* para enviar la petición de I/O a la capa del subsistema de bloque.
7.Devuelve la dirección del descriptor de la página.

12.5 Creación de un archivo de intercambio

Creación del fichero de 64 MB=1024x65536

dd if=/dev/zero of=swapfile bs=1024 count=65536

dd commando que copia sector a sector
if = fichero de entrada
of = fichero de salida
/dev/zero dispositivo para inicializar todo a cero
bs = tamaño bloque
count = nº de veces a multiplicar al valor de "bs"

Dar formato al fichero swap

mkswap -v1 swapfile

-v1= crea en estilo nuevo el área swap

Activación del fichero-swap

swapon swapfile

Comprobación de funcionamiento
swapon –s

Filename	*Type*	*Size*	*Used*	*Priority*
/dev/sda1	*partition*	*40156*	*0*	*-1*
/root/Desktop/swapfile	*file*	*65532*	*0*	*-2*

Desactivación del fichero-swap

swapoff swapfile

12.6 Bibliográfia

Maxvell(cap 8)
Card (cap 8)
Understanding Linux Kernel 3ª edición
http://lxr.linux.no/