# I N D I C E

# LLAMADAS AL SISTEMA “*BRK*”

## INTRODUCCIÓN A LA MEMORIA DINÁMICA

Es la memoria que solicitan los procesos en tiempo de ejecución a medida que la van necesitando. Cuando un proceso necesita de un espacio de memoria para poder guardar el valor de sus variables o simplemente requiere la construcción de una estructura en memoria, solicita al sistema, mediante determinadas llamadas, un espacio de memoria. Una vez este proceso finalice su ejecución, deberá liberar la memoria solicitada anteriormente.

Para poder llevar acabo la gestión o la asignación de memoria dinámica, se utilizan los punteros, aunque es necesario destacar que un puntero no implica directamente el uso de memoria dinámica, pero si están estrechamente relacionadas. El motivo del uso de punteros es sencillo, es la única forma de poder “*apuntar*” a las zonas de memoria disponibles.

Todo proceso en ejecución tiene un espacio de direccionamiento, que si no es idéntico uno de otro es bastantes similar, pudiendo sacar como patrones de su arquitectura las siguientes zonas del espacio de direccionamiento:

- Código del proceso.
- Datos del proceso (*heap*).
- Variables inicializadas.
- Variables no inicializadas.
- Código y datos de las bibliotecas compartidas.
- Pila usada por el proceso (*stack*).

Todos los procesos se componen de un código de ejecución, el cuál, evidentemente reside en una determinada zona de memoria. Los datos del proceso, como puedan ser las variables, ya sean inicializadas o no, se guardan el la “*zona heap*”. Por último, encontramos la “*zona de pila*”, en la que se almacenan los resultados de las diferentes operaciones llevadas a cabo por el proceso en cuestión.

## ESTRUCTURA DE LA MEMORIA

A continuación se explicarán las diferentes zonas de la que se compone la memoria, para posteriormente mostrar un dibujo ilustrativo, en donde poder observar la situación de las diferentes ranuras de la memoria.

- ✓ **Zona de código**: es la parte de la memoria donde residen las instrucciones de nuestro programa; suele ocupar las direcciones más bajas de la memoria.

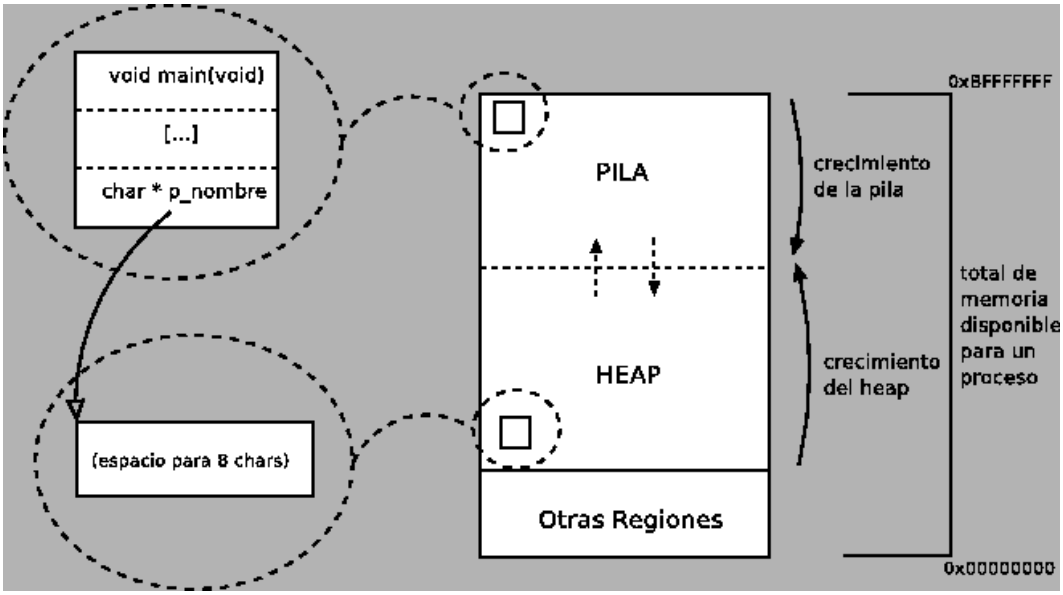
- ✓ **Zona de datos**: se alojan los datos estáticos de nuestro programa, es decir, las variables globales del mismo. Suele ser un espacio de memoria limitado, por lo que el número y tamaño de estas variables también está limitado. Forma junto con la zona de código la parte estática de la memoria.

- ✓ **La pila o stack**: que suele encontrarse en las posiciones más altas de memoria, creciendo dinámicamente hacia las direcciones más bajas de la misma. En el *stack* se alojarán las variables locales de los diferentes módulos conforme estos se van activando.

  Cada vez que se llama a una función se introduce en el Stack la información necesaria para poder retomar la ejecución de esta, como son los parámetros pasados y las variables declaradas en ella.

  Cuando un proceso empieza su ejecución, sus segmentos poseen un tamaño fijo. Existen funciones que permiten modificar el tamaño del segmento de datos del proceso.

- ✓ **El heap**: zona de datos dinámicos, se encuentra entre la zona de datos estáticos y la pila. Su tamaño varía dependiendo de cómo varíe el tamaño de la pila, aunque por lo general los compiladores asignan como *heap* todo el espacio que no es zona de código y de datos estáticos. El programador no debe preocuparse de cómo se gestiona esta memoria, sino solo de su utilización ya que es el propio compilador el que se carga de ello.



## RESERVA DE LA MEMORIA DINÁMICA

Para poder reservar memoria dinámica, se utilizan una serie de funciones que posteriormente se transformarán en llamadas al sistema. De la misma forma se deberá de invocar una función para poder liberar la memoria utilizada durante la ejecución de un determinado proceso.

Las funciones para solicitar la asignación de memoria dinámica son:

```
malloc (cantidad_de_memoria);

calloc (número_de_elementos, tamaño_de_cada_elemento);
```

Estas dos funciones reservan la memoria especificada y nos devuelven un puntero a la zona en cuestión. Si no se ha podido reservar el tamaño de la memoria especificado devuelve un puntero con el valor 0 o NULL. El tipo del puntero es, en principio *void*, es decir, un puntero a cualquier cosa. Por tanto, a la hora de ejecutar estás funciones es aconsejable realizar una operación *cast* (de conversión de tipo) de cara a la utilización de la aritmética de punteros a la que aludíamos anteriormente. Los compiladores modernos suelen realizar esta conversión automáticamente.

El operador *sizeof(tipo_de_dato)*, es indispensable para poder reservar un tamaño de espacio en concreto. Nos devuelve el tamaño que ocupa en memoria un cierto tipo de dato, de esta manera, podemos escribir programas independientes del tamaño de los datos y de la longitud de palabra de la máquina.

Una vez que se ha asignado una zona de memoria mediante la función "*malloc*", sólo podemos rehacer esta asignación mediante la función

```
realloc(void *ptr, size_t size);
```

La cual cambia el tamaño de una zona de memoria que ya ha sido asignada con malloc.

Por otro lado, se debe liberar la memoria asignada durante el proceso de ejecución. La función que nos permite liberar la memoria asignada con malloc y calloc es

```
free(puntero)
```

Se debe liberar todo el bloque que se ha asignado, con lo cual siempre debemos tener almacenados los punteros al principio de la zona que reservamos. Si mientras actuamos sobre los datos modificamos el valor del puntero al inicio de la zona reservada, la función free probablemente no podrá liberar el bloque de memoria.

## FUNCIÓN “BRK”

Un proceso puede modificar el tamaño de su segmento de datos. Para ellos, Linux proporciona la llamada al sistema “*brk*”:

```
#include <unistd.h>

int brk(void *end_data_segment);
```

El parámetro *end_data_segment*, especifica la dirección de fin del segmento de datos. Debe ser superior a la dirección de fin del segmento de código e inferior en 16 Kilobytes a la dirección de fin del segmento de pila. En caso de éxito, *brk* devuelve el valor 0. en caso de fallo, se devuelve el valor -1 y la variable de *errno* toma el valor ENOMEM.

## FUNCIÓN “SBRK”

Existes una llamada al sistema que permite al proceso actual modificar el tamaño de su segmento de datos:

```
#include <unistd.h>

void *sbrk(ptrdiff_t increment);
```

El parámetro *increment* especifica el número de Bytes a añadir al segmento de datos, o a sustraer si *increment* es negativo. La función *sbrk* devuelve la nueva dirección de fin de segmento de datos, o el valor -1 en caso de fallo. En éste último caso, la variable *errno* toma el valor ENOMEM.

## FUNCIONES DE ASIGNACIÓN Y LIBERACIÓN

BRK y SBRK no están incluidas en el estándar de C y es muy laborioso gestionar la memoria con ellas. Sin embargo son usadas por otras funciones explicadas en apartados anteriores tales como:

- **Malloc**: Asigna zona de memoria
- **Calloc**: Asigna zona de memoria, prevista para matriz.
- **Realloc**: Modifica el tamaño de zona de memoria.
- **Free**: Libera zona de memoria.

Estas funciones comentadas anteriormente, trabajan sobre las funciones brk y sbrk. Tanto una como la otra, generan una llamada al sistema, la cual es la verdadera encargada de realizar el procesos de asignación de memoria, así como controlar todos los mecanismos necesarios para garantizar un correcto funcionamiento de la memoria. La llamada comentada anteriormente es "sys_brk" y se encuentra implementada en mm/mmap.c

Esta llamada al sistema modifica el valor de brk para aumentar/disminuir el tamaño del segmento de datos.


Esta llamada al sistema realiza básicamente tres pasos:

1) Si la nueva dirección de *brk* que se le pasa es inferior a la actual, se desasigna la zona de memoria mediante *do_munmap.*

2) Si es superior, se asigna memoria con *do_brk..*

3) Si la nueva dirección es incorrecta (zona de pila o de código), simplemente devuelve el valor actual de *brk*.

Como estructura principal, trabaja con la *"mm_struct"*, la cual se aloja en el fichero include/linux/sched.h y básicamente define como es la estructura de la memoria.

## CÓDIGO DE “*SYS_BRK*”

Cuando un proceso en modo usuario invoca la llamada brk, el kernel ejecuta la función sys_brk(addr).

```
206 asmlinkage unsigned long sys_brk(unsigned long brk)
207 {
```

Declaración de variables

```
208 unsigned long rlim, retval;
209 unsigned long newbrk, oldbrk;
210 struct mm_struct *mm = current->mm;
```

Se bloquea la escritura en la memoria por parte del proceso actual.

```
212 down_write(&mm->mmap_sem);
```

Comprobamos que el brk no se coloca en la zona dedicada al código. Si esto sucede se saltará al final de la función y retorna ya que el heap no se puede solapar con la región del código.

```
214 if (brk < mm->end_code)
215     goto out;
```

La llamada sistema brk() actúa en la región de la memoria y se encarga de alojar y desalojar todas las páginas. Además la función alinea la dirección en múltiplos indicado por PAGE_SIZE y compara el nuevo brk con el actual. Si el nuevo brk se encuentra en la misma página que el antiguo, vamos a set_brk donde se le asigna al brk antiguo el nuevo brk

```
216 newbrk = PAGE_ALIGN(brk);
217 oldbrk = PAGE_ALIGN(mm->brk);
218 if (oldbrk == newbrk)
219     goto set_brk;
```

Si el nuevo brk esta por debajo del antiguo, libera el trozo de memoria que hay entre ambos utilizando la función do_munmap() y va a set_brk. Si no se ha podido liberar va a out.

```
222 if (brk <= mm->brk) {
223     if (!do_munmap(mm, newbrk, oldbrk-newbrk))
224             goto set_brk;
225     goto out;
226 }
```

Si el proceso quiere aumente el heap, sys_brk() primero chequea si el proceso tiene permiso para hacerlo. Si el proceso esta intentando alojar memoria fuera de sus limites, la función simplemente retorna el valor original de mm->brk sin asignar más memoria.

```
229 rlim = current->signal->rlim[RLIMIT_DATA].rlim_cur;
230 if (rlim < RLIM_INFINITY && brk - mm->start_data > rlim)
231     goto out;
```

Comprueba que la zona de memoria se puede utilizar, examinando que no se solapa con ninguna otra región de memoria en los márgenes establecidos.

```
234 if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
235     goto out;
```

```
Si todo ha salido bien, la función do_brk()  es invocada. Si esta retorna oldbrk, la
asignación fue satisfactoria y sys_brk() devuelve el valor addr, en caso contrario,
este retorna el viejo mm->brk.


238 if (do_brk(oldbrk, newbrk-oldbrk) != oldbrk)
239     goto out;
240 set_brk:
241 mm->brk = brk;
242 out:
243 retval = mm->brk;

volvemos a habilitar la escritura y devolvemos retval

244 up_write(&mm->mmap_sem);
245 return retval;
246 }
```

## ASIGNACIÓN DE ZONAS DE MEMORIA

Linux ofrece varios tipos de funciones que permiten asignar zonas de memoria para la utilización propia del núcleo:

- ***kmalloc*** **y** ***kfree*** permiten asignar y liberar zonas de memoria formadas por páginas contiguas en memoria central; suelen ser muy útiles para la gestión de zonas de pequeño tamaño, aunque son notoriamente ineficientes para zonas de grandes tamaños, ya que utilizan bloques de tamaño fijo y devuelven un bloque cuyo tamaño puede ser muy superior al tamaño solicitado.

- ***vmalloc*** **y** ***vfree*** permiten asignar y liberar zonas de memoria formadas por páginas que no son forzosamente contiguas en memoria central; no presentan el problema de las funciones descritas anteriormente. Estas páginas se insertan seguidamente en el segmento reservado al núcleo modificando las tablas de páginas. De este modo no se desperdicia tanto el tamaño de la memoria asignado.

Para la implementación de *kmalloc* y *kfree*, Linux utiliza listas de zonas disponibles. Para cada tamaño de zona, se gestiona una lista de páginas descompuestas en bloques. Cuando en *kmalloc* se llama con un tamaño especificado, se usa la lista correspondiente al tamaño inmediatamente superior. A cada una de las listas se les asocia un descriptor que apunta a una lista de grupo de páginas de memoria. *Kmalloc* y *kfree* hacen uso de **get_free_pages** y **free_pages**, que son otras funciones de bajo nivel para la manipulación de páginas.

- ✓ ***Vmalloc*** y ***vfree*** se implementan basándose en kmalloc y kfree
- ✓ ***vmalloc*** y ***vfree*** se encuentran en **mm/vmalloc.c**
- ✓ ***kmalloc*** y ***kfree*** están en **mm/slab.c**

***Kmalloc y vmalloc*** son los mecanismos de alto nivel que el kernel proporciona al programador para la reserva de memoria en el kernel, existen otros mecanismos de bajo nivel que ya estudiaremos en su momento. La diferencia principal, en cuanto a la interfaz de estos dos grupos de funciones, radica en la reserva de memoria en términos de páginas para las funciones de bajo nivel, y en términos de bytes paras las funciones de alto nivel.

Kmalloc es muy similar a la función de librería de espacio usuario, malloc, la cual estamos habituados a usar. Con kmalloc obtenemos reserva de memoria del kernel en bloques de bytes. La función devuelve un puntero a una región de memoria que es al menos size bytes de longitud (puede ser mayor, pues la memoria se asigna en unidades de páginas). Lo importante a destacar, es que la región de memoria es físicamente contigua. No entraremos en detalle con respecto los flags que se usan en esta función, sólo indicar que se usan para modificar el comportamiento a la hora de adquirir memoria, o indicar el tipo de memoria deseada.

La función vmalloc trabaja de forma similar a kmalloc, excepto que asigna memoria que solamente es contigua virtualmente, no físicamente. Esta es la forma de trabajar de la función de librería malloc de espacio de usuario, se devuelve un trozo de memoria que es contigua dentro del espacio virtual de direcciones del procesador, pero no se garantiza que sea contigua en la RAM física.

Generalmente los dispositivos hardware necesitan trozos de memoria contigua físicamente, es decir, necesitan usan kmalloc, también por rendimiento el kernel suele usar kmalloc, se reserva el uso de vmalloc a situaciones donde se quiere asignar gran cantidad de memoria (el coste de reservar memoria física contigua en grandes cantidades es mayor) y generalmente para aspectos puramente software, como buffers relacionados con procesos, estructuras de comunicaciones, etc.

Como estructura principal, trabaja con la *"vm_struct"*, la cual se aloja en el fichero **include/linux/vmalloc.h** y básicamente define como es la estructura de la memoria.

```
13 struct vm_struct {
14         void                    *addr;
15         unsigned long           size;
16         unsigned long           flags;
17         struct page             **pages;
18         unsigned int            nr_pages;
19         unsigned long           phys_addr;
20         struct vm_struct        *next;
21 };
```

## LLAMADA VMALLOC

```
Le pasamos el tamaño de memoria que solicitamos en size

void *__vmalloc(unsigned long size, int gfp_mask, pgprot_t prot)
471 {
472         struct vm_struct *area;
```

La dirección donde se encuentran los descriptores de páginas la podemos encontrar en pages.

```
473         struct page **pages;
474         unsigned int nr_pages, array_size, i;
```

Alineamos el tamaño que solicitamos y comprobamos si el tamaño es 0 o mayor que el permitido

```
476         size = PAGE_ALIGN(size);
477         if (!size || (size >> PAGE_SHIFT) > num_physpages)
478                 return NULL;
```

Mediante el get_vm_area, que comentaremos a continuación, creamos un nuevo descriptor y retornamos la dirección asignada para el área de memoria. Si no se ha podido alojar la memoria, retornamos NULL.

```
480         area = get_vm_area(size, VM_ALLOC);
481         if (!area)
482                 return NULL;
```

Calculamos el número el número de páginas y el total de bytes que solicitamos

```
484         nr_pages = size >> PAGE_SHIFT;
485         array_size = (nr_pages * sizeof(struct page *));
486
487         area->nr_pages = nr_pages;
```

Reservamos espacio en función de nuestro array_size. Si este es mayor que el tamaño de una página entonces llamamos a la función vmalloc que nos permite reservar páginas no contiguas en caso contrario llamamos a kmalloc.

```
489         if (array_size > PAGE_SIZE)
490                 pages = __vmalloc(array_size, gfp_mask, PAGE_KERNEL);
491         else
492                 pages = kmalloc(array_size, (gfp_mask & ~__GFP_HIGHMEM));
493         area->pages = pages;
```

Si no encontramos espacio suficiente eliminamos todo el espacio reservado y finalizamos devolviendo NULL.

```
494         if (!area->pages) {
495                 remove_vm_area(area->addr);
496                 kfree(area);
497                 return NULL;
498         }
```

Utilizando la función memset ponemos a nulo todo el espacio de memoria que indica la variable array_size en la dirección que indica area->pages.

```
499         memset(area->pages, 0, array_size);
```

Con la función alloc_page enmarca cada página y almacenamos el correspondiente descriptor de cada una de las páginas en el vm_struct. Por último, actualizamos el número de páginas totales.

```
501         for (i = 0; i < area->nr_pages; i++) {
502                 area->pages[i] = alloc_page(gfp_mask);
503                 if (unlikely(!area->pages[i])) {
504                         /* Successfully allocated i pages, free them in __vunmap() */
505                         area->nr_pages = i;
506                         goto fail;
507                 }
508         }
```

```
Mediante el map_vm_area utilizamos direcciones no contiguas de memoria física mapeadas por el
espacio virtual de direccionamiento del núcleo (contiguo). Si no podemos ubicar las páginas en
la zona de memoria devolvemos un fallo. En caso contrario se devolverá la dirección donde
reside nuestra zona de memoria


510         if (map_vm_area(area, prot, &pages))
511                 goto fail;
512         return area->addr;
513
514 fail:
515         vfree(area->addr);
516         return NULL;
517 }
```

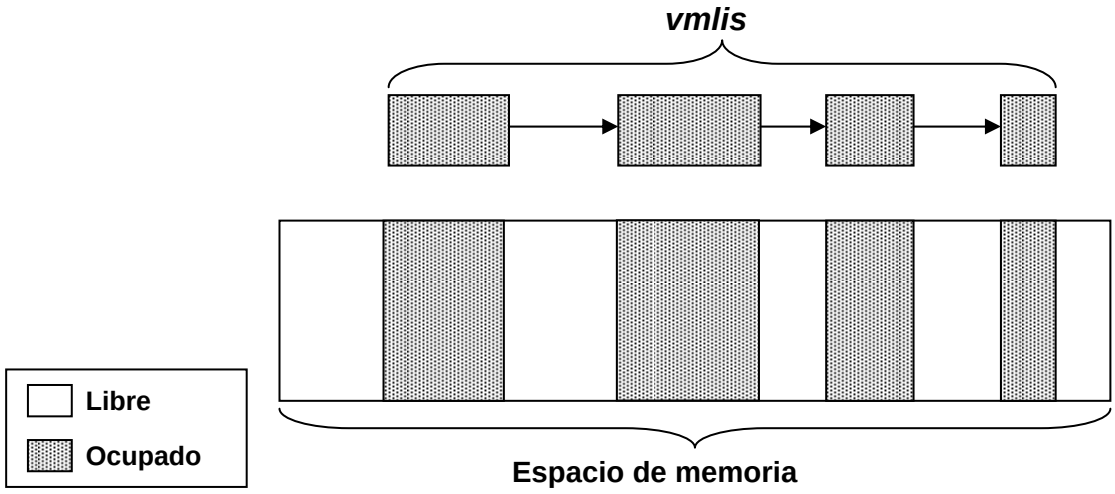
## LLAMADA GET_VM_AREA

```
vm_struct * get_vm_area (unsigned long size)
```

Esta función se encarga de buscar un hueco libre en la zona de memoria del kernel, del tamaño definido en el parámetro de la función. Cuando la función retorna correctamente, devolverá un puntero a dicha región.

El llamador es el responsable de que el tamaño solicitado sea distinto de cero y múltiplo del tamaño de página. Habitualmente utiliza la política  first-fit algoritmo para localizar la zona de memoria, aunque podría modificarse la política de selección del espacio de memoria siguiendo otras políticas de búsqueda.

Utiliza la variable global *vmlis*t que contiene una lista ordenada de trozos de memoria ocupada.



### *PASOS DEL ALGORITMO*

Esta función actúa con dos parámetros: el tamaño en bytes de la región de memoria creada y un flag que especifica el tipo de región.

```
237 struct vm_struct *__get_vm_area(unsigned long size, unsigned long flags,
238                                 unsigned long start, unsigned long end)
239 {
240         struct vm_struct **p, *tmp, *area;
241         unsigned long align = 1;
242         unsigned long addr;
243
244         if (flags & VM_IOREMAP) {
245                 int bit = fls(size);
246
247                 if (bit > IOREMAP_MAX_ORDER)
248                         bit = IOREMAP_MAX_ORDER;
249                 else if (bit < PAGE_SHIFT)
250                         bit = PAGE_SHIFT;
251
252                 align = 1ul << bit;
253         }
```

Los pasos realizados son los siguientes:

1. Kmalloc obtiene un area de memoria para el nuevo descriptor vm_struct.malloc( ).

```
254         addr = ALIGN(start, align);
255
256         area = kmalloc(sizeof(*area), GFP_KERNEL);
```

```
Si no encontramos el hueco libre, liberamos la memoria reservada para el nuevo
elemento de vmlist
```

```
257         if (unlikely(!area))
258                 return NULL;
263         size += PAGE_SIZE;
264         if (unlikely(!size)) {
265                 kfree (area);
266                 return NULL;
267         }
```

2. Utiliza la vmlist_lock para bloquear la escritura y a continuación escanea la lista de descriptores a fin de buscar un hueco libre.

```
269         write_lock(&vmlist_lock);
270         for (p = &vmlist; (tmp = *p) != NULL ;p = &tmp->next) {
271                 if ((unsigned long)tmp->addr < addr) {
272                         if((unsigned long)tmp->addr + tmp->size >= addr)
273                                 addr = ALIGN(tmp->size +
274                                              (unsigned long)tmp->addr, align);
275                         continue;
276                 }
277                 if ((size + addr) < addr)
278                         goto out;
279                 if (size + addr <= (unsigned long)tmp->addr)
280                         goto found;
281                 addr = ALIGN(tmp->size + (unsigned long)tmp->addr, align);
282                 if (addr > end - size)
283                         goto out;
284         }
```

3 Si tal intervalo existe, la función inicializa los campos del descriptor, libera el bloqueo de escritura y termina retornando la dirección inicial del area de memoria no continua. De otro modo, get_vm_area( ) libera el descriptor obtenido previamente, libera el bloqueo de escritura y retorna NULL.

```
286 found:
287         area->next = *p;
288         *p = area;
289
290         area->flags = flags;
291         area->addr = (void *)addr;
292         area->size = size;
293         area->pages = NULL;
294         area->nr_pages = 0;
295         area->phys_addr = 0;
296         write_unlock(&vmlist_lock);
297
298         return area;
299
300 out:
301         write_unlock(&vmlist_lock);
302         kfree(area);

305         return NULL;
306 }
```

## FUNCIÓN VFREE

La función vfree( ) libera áreas de memoria no contiguas creadas por vmalloc( ), mientras que la función vunmap( ) libera memoria de áreas creadas por vmap( ). Ambas funciones tiene de parámetro la dirección inicial del área que va a ser liberado.

```
401 void vfree(void *addr)
402 {
403         BUG_ON(in_interrupt());
404         __vunmap(addr, 1);
405 }
```

La función vunmap( ) recibe dos parámetros: la dirección inicial del área que va a ser liberado y el flag deallocate_pages, que se activa si la página mapeada en el área debería ser liberada.

```
351 void __vunmap(void *addr, int deallocate_pages)
352 {
353         struct vm_struct *area;
354
355         if (!addr)
356                 return;

Dirección incorrecta. No es múltiplo del tamaño de página (no está alineada)

358         if ((PAGE_SIZE-1) & (unsigned long)addr) {
359                 printk(KERN_ERR "Trying to vfree() bad address (%p)\n", addr);
360                 WARN_ON(1);
361                 return;
362         }
```

### *PASOS PRINCIPALES*

1. Invoca a remove_vm_area( ) para conseguir la dirección del área apuntada por el descriptor vm_struct y limpiar las entradas de la tabla de páginas del núcleo correspondientes a las direcciones en el área de memoria no contigua.

```
364         area = remove_vm_area(addr);
365         if (unlikely(!area)) {
366                 printk(KERN_ERR "Trying to vfree() nonexistent vm area (%p)\n",
367                                 addr);
368                 WARN_ON(1);
369                 return;
370         }
```

2. Si el flag deallocate_pages esta activado, este escanea el array area->pages y para cada elemento invoca free_page( ) que se encarga de liberar cada página. Además ejecuta kfree para liberar dicho array.

```
372         if (deallocate_pages) {
373                 int i;
375                 for (i = 0; i < area->nr_pages; i++) {
376                         if (unlikely(!area->pages[i]))
377                                 BUG();
378                         __free_page(area->pages[i]);
379                 }
381                 if (area->nr_pages > PAGE_SIZE/sizeof(struct page *))
382                         vfree(area->pages);
383                 else
384                         kfree(area->pages);
385         }
```

3. Invoca kfree(area) para liberar el descriptor de vm_struct.

```
387         kfree(area);
388         return;
389 }
```

## FUNCIÓN MPROTECT

Según el tipo de informaciones contenidas en memoria, se asocia una protección diferente a cada región de memoria perteneciente al espacio de direccionamiento de un proceso. Los derechos de acceso a la memoria son gestionados directamente por el procesador.

Linux permite a un proceso modificar las protecciones de memoria asociadas a alguna de las regiones comprendidas en su espacio de direccionamiento. Para ello dispone de una función encargada de modificar los derechos de accesos asociados a una determinada parte de la memoria. Esta función es:

```
#include <sys/mman.h>

int mprotect(const void *addr, size_t len, int prot);
```

El prototipado de la función es:

```
sys_mprotect(unsigned long start, size_t len, unsigned long prot)
```

- **Start**: es la dirección de la memoria sobre la que queremos cambiar el acceso.
- **Len:** longitud de la zona de memoria en bytes.
- **Prot:** es el tipo de protección que queremos establecer, que puede tomar 6 valores.

Estos valores corresponden a macros definidas en **mman.h**

- ***PROT_NONE***: La memoria no se puede acceder.
- ***PROT_READ:*** Solo hay permiso de lectura.
- ***PROT_WRITE:*** Solo hay permiso de escritura.
- ***PROT_EXEC***: Puede contener código ejecutable.
- ***PROT_SEM***: Suele ser usada para operaciones atómicas.
- ***PROT_GROWSDOWN***: marca para inicio de área de memoria virtual que está decreciendo.
- ***PROT_GROWSUP***: marca para fin de área de memoria virtual que está creciendo.

En caso de éxito, "***mprotect*"** devuelve cero. En caso de error, se devuelve -1, y se asigna a *errno* un valor apropiado:

- ***EINVAL:*** parámetros no válidos.
- ***EFAULT:*** la memoria no puede ser accedida
- ***ENOMEM:*** no hay memoria libre.
- ***EACCES***: la memoria no permite el acceso especificado, por ejemplo, si se asocia a una zona de memoria mediante **mmap**, un fichero que solo tiene acceso de lectura y se le pide a mprotect que lo marque con *PROT_WRITE*.

## CÓDIGO DE "*MPROTECT* "

El código de esta función se encuentra en el fichero *mm/mprotect.c*. Como estructura principal utiliza el vm_area_struct que se localiza en ***include/Linux/mm.h***

## *PASOS PRINCIPALES*

**1**

Comprobamos si las protecciones de la zona de memoria especificada por los parámetros de entrada (comienzo, final y permisos) son correctas y además en que situación nos encontramos para a continuación modificar los permisos que indica el parámetro *prot*.

```
222 asmlinkage long
223 sys_mprotect(unsigned long start, size_t len, unsigned long prot)
224 {

Declaración de variables locales

225   unsigned long vm_flags, nstart, end, tmp;
226     struct vm_area_struct *vma, *prev;
227   int error = -EINVAL;

Grows sirve para detectar si prot es o PROT_GRWSDOWN o PROT_GRSUP

228   const int grows = prot & (PROT_GROWSDOWN|PROT_GROWSUP);
229   prot &= ~(PROT_GROWSDOWN|PROT_GROWSUP);
230           if (grows == (PROT_GROWSDOWN|PROT_GROWSUP))
231           return -EINVAL;
233   if (start & ~PAGE_MASK)
234           return -EINVAL;
235   len = PAGE_ALIGN(len);
236   end = start + len;

Comprueba que la dirección de comienzo sea más pequeña que la de fin

237   if (end < start)
238           return -ENOMEM;

Comprueba que los permisos de prot son correctos.

239   if (prot & ~(PROT_READ | PROT_WRITE | PROT_EXEC | PROT_SEM))
240           return -EINVAL;



Si end=start, no devuelve error, pero no hace nada

241   if (end == start)
242           return 0;


Si la lectura implica ejecución, añade el bit de ejecución a prot

246   if (unlikely((prot & PROT_READ) &&
247                 (current->personality & READ_IMPLIES_EXEC)))
248           prot |= PROT_EXEC;
249
250   vm_flags = calc_vm_prot_bits(prot);


Deshabilitamos escritura

252   down_write(&current->mm->mmap_sem);

Encuentra la primera área de memoria virtual que empieza en start y devuelve en prev
la región anterior


254   vma = find_vma_prev(current->mm, start, &prev);
255   error = -ENOMEM;
256   if (!vma)
257           goto out;

Si estamos en el caso de PROT_GROWSDOWN, entrara en esta zona del código.
```

```
258   if (unlikely(grows & PROT_GROWSDOWN)) {

Si la memoria de vma es mayor que end o el flag VM_GROWSDOWN no está activo, saltamos
a out

259           if (vma->vm_start >= end)
260                   goto out;
261           start = vma->vm_start;
262           error = -EINVAL;
263           if (!(vma->vm_flags & VM_GROWSDOWN))
264                   goto out;
265   }

Si no estamos en PROT_GRWSDOW, comprobamos que la memoria vma es menor que start sino
va a out

266   else {
267           if (vma->vm_start > start)
268                   goto out;

Si estamos en el caso de PROT_GROWSUP, entra en el if y comprueba que el flag
VM_GROWSUP está activo

269           if (unlikely(grows & PROT_GROWSUP)) {
270                   end = vma->vm_end;
271                   error = -EINVAL;
272                   if (!(vma->vm_flags & VM_GROWSUP))
273                           goto out;
274           }
275   }
276   if (start > vma->vm_start)
277           prev = vma;
```

2

Si hasta este punto todo el chequeo ha sido correcto entonces ya podemos modificar las protecciones para dicha zona de memoria pero antes hay que comprobar que dichas modificaciones se pueden aplicar.

```
En cada región de memoria se chequea que se pueden aplicar las nuevas modificaciones
en las protecciones


279   for (nstart = start ; ; ) {
280           unsigned long newflags;

284           if (is_vm_hugetlb_page(vma)) {
285                   error = -EACCES;
286                   goto out;
287           }
288
289           newflags = vm_flags | (vma->vm_flags & ~(VM_READ | VM_WRITE | VM_EXEC));


291           if ((newflags & ~(newflags >> 4)) & 0xf) {
292                   error = -EACCES;
293                   goto out;
294           }
296           error = security_file_mprotect(vma, prot);
297           if (error)
298                   goto out;
299
300           tmp = vma->vm_end;
301           if (tmp > end)
302                 tmp = end;
```

**3**

Por último las nuevas protecciones se establecen mediante la función ***mprotect_fixup*** la cuál llama a una función  según la ubicación de la zona a modificar en la región de memoria. Seguidamente llama a ***change_protection*** para modificar las protecciones asociadas a las páginas de memoria correspondientes

```
303          error = mprotect_fixup(vma, &prev, nstart, tmp, newflags);
304          if (error)
305                  goto out;
306          nstart = tmp;
307
308          if (nstart < prev->vm_end)
309                  nstart = prev->vm_end;
310          if (nstart >= end)
311                  goto out;
312
313          vma = prev->vm_next;
314          if (!vma || vma->vm_start != nstart) {
315                  error = -ENOMEM;
316                  goto out;
317          }
318   }
```

Cuando finalizamos volvemos a habilitar la escritura

```
319 out:
320   up_write(&current->mm->mmap_sem);
321   return error;
322 }
```