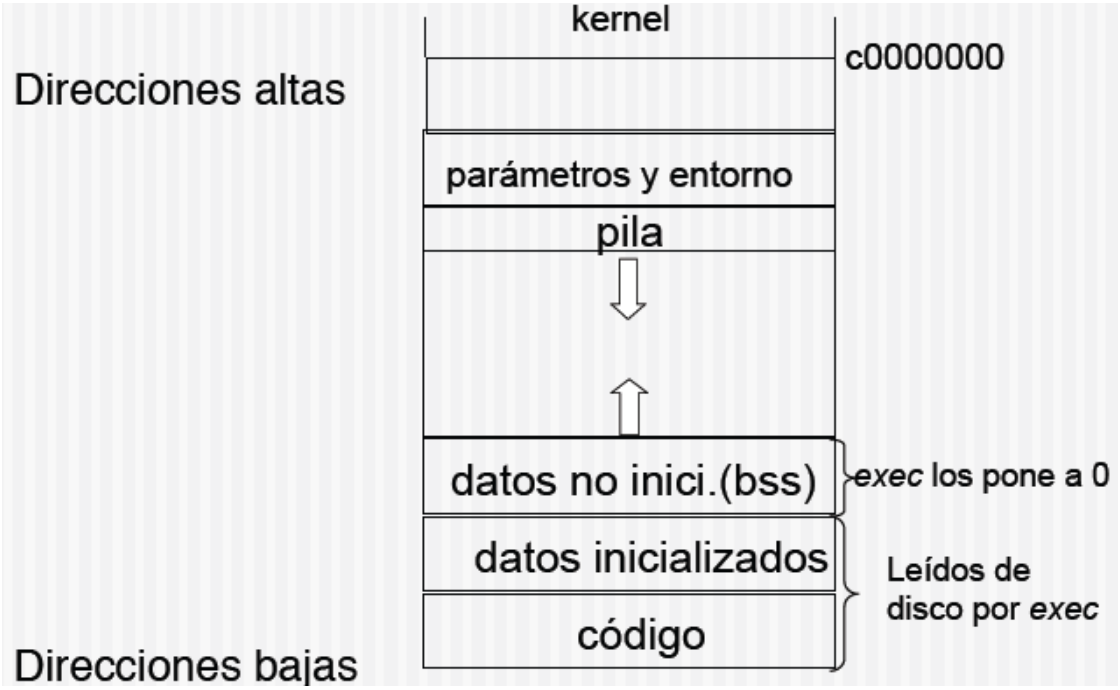
LECCIÓN 13: MMAP PROYECCIÓN DE UN FICHERO EN MEMORIA

LECCIÓN 13: MMAP PROYECCIÓN DE UN FICHERO EN MEMORIA

13.1 Introducción

A cada proceso se le asocia un espacio de direccionamiento que representa las zonas de memoria que tiene asignadas y a las que, por tanto, puede acceder. Este espacio de direccionamiento incluye:

- ✓ El código del proceso.
- ✓ Los datos del proceso:
  - Variables inicializadas
  - Variables no inicializadas
- ✓ El código y los datos de bibliotecas compartidas utilizadas por el proceso.
- ✓ La pila usada por el proceso.



Para conocer las direcciones de memoria donde se encuentran los segmentos de código y de datos de un proceso hay disponibles los siguientes simbolos:

- ✓ extern int _end
- ✓ extern int _etext
- ✓ extern int _edata

A este espacio de memoria asignado al proceso se le denomina Espacio de Direccionamiento (ED), en la tabla de procesos viene representado por la estructura **mm_struct:**

struct task_struct

```
{ /*...*/
        struct mm_struct*      mm;    /*Memoria asociada*/
/*...*/ }
```

Regiones de memoria (RM)

Cada uno de los recursos citados anteriormente (código del proceso, datos…) se aloja en una región/zona dentro del ED del proceso, las cuales poseen los siguientes atributos:

- ✓ Direcciones de inicio y fin.
- ✓ "Derechos de acceso" asociados (no accesible, lectura, escritura, ejecución)
- ✓ Objeto asociado, p.e. un archivo.

El núcleo mantiene en memoria (en la tabla de procesos) una descripción de las regiones utilizadas por cada proceso, a modo de lista.

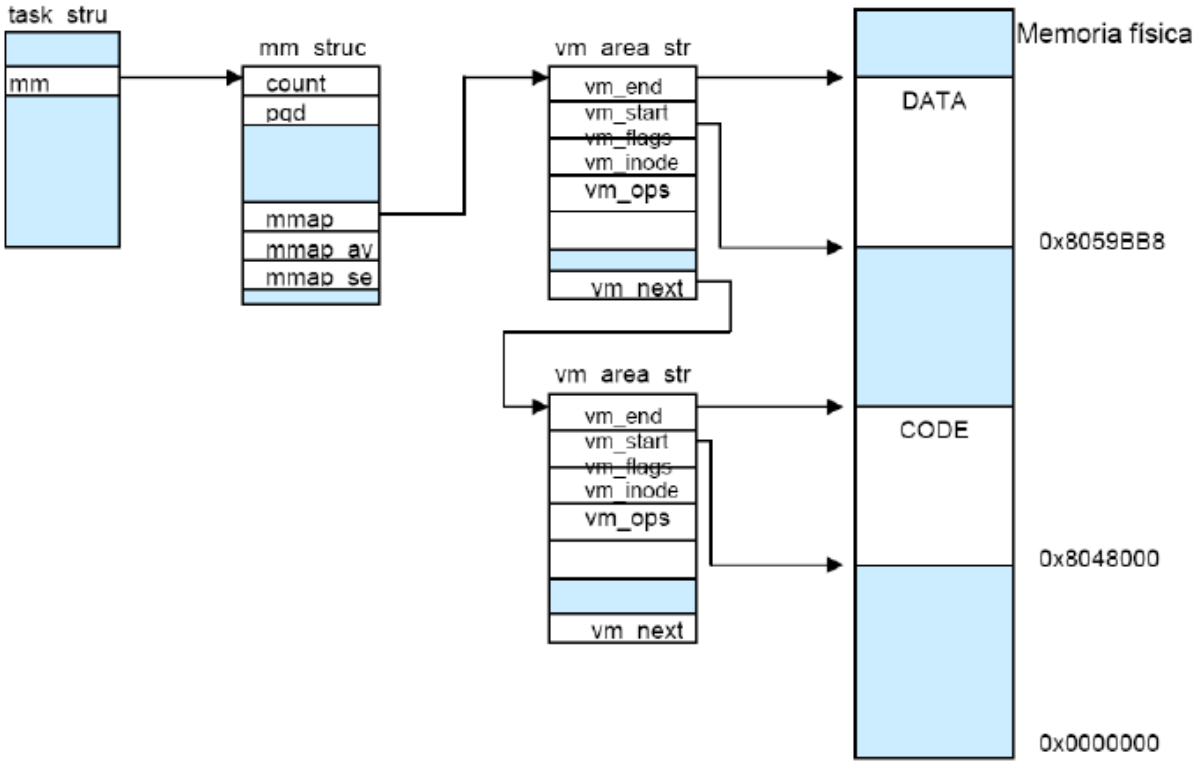
Existen una serie de descriptores para cada región. Entre ellos:

vm_start: Dirección de inicio.
vm_end: Dirección de fin.
vm_page_prot: Protección asociada.
vm_flags: Estado:
- VM_READ
- VM_WRITE
- VM_EXEC.
- VM_SHARED.
- VM_MAYREAD.
- VM_MAYWRITE.
- VM_MAYEXEC.
- VM_MAYSHARED.
- VM_DENYWRITE.
- VM_LOCKED.

vm_ops: Operaciones que tiene asociada la región.
vm_offset: Dirección de inicio de la zona de memoria respecto al inicio del objeto proyectado en memoria.

Los derechos de acceso a memoria son gestionados directamente por el procesador. Linux permite a un proceso modificar las protecciones de memorias asociadas a algunas de las regiones de su ED. Con la llamada **mprotect** podemos modificar las protecciones de acceso asociadas a la región especificada.

MEMORIA VIRTUAL DE UN PROCESO:



Como ya es sabido, el núcleo mantiene una estructura task_struct por cada proceso, la cual guarda cierta información referente al mismo. En este caso el campo que nos interesa tratar es el “mm” el cual es un puntero a una estructura **mm_struct**, que es la encargada de mantener la información asociada con la memoria virtual que maneja el proceso en cuestión. Esta estructura posee entre otros campos, un puntero a un área de memoria virtual (a un **vm_area_struct**) que controlará una determinada área de la memoria virtual referente a un fragmento del contenido del objeto que se desee volcar en memoria (para un proceso código, datos sin inicializar, datos inicializados, etc). Así, al tener dividida la memoria en zonas para cada una de las partes distintas del objeto, podemos asociar permisos y operaciones específicas para cada una de ellas, ofreciendo una mayor seguridad y flexibilidad en el manejo del objeto en memoria.

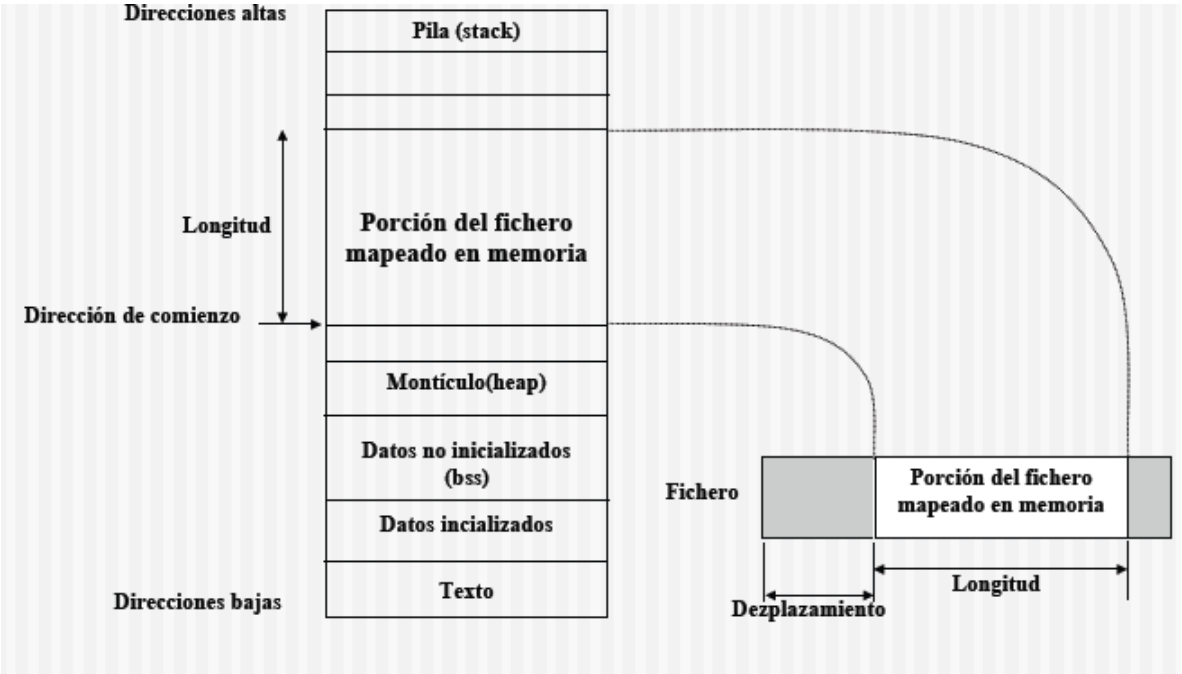
13.2 Proyección de archivos en memoria: File mapping

La manera de modificar fichero hasta ahora había sido comúnmente mediante la secuencia siguiente: 1. Open, 2. Read/write, 3. Close.

Pero esta no es la única manera:

Mediante la técnica de proyección de ficheros se pueden realizar modificaciones de ficheros de manera rápida y efectiva a través de una proyección de su contenido en memoria.

Es una operación que se encarga de crear una nueva región de memoria RM en el ED del proceso y el contenido del archivo proyectado será accesible en esta RM.



Su utilidad principal es la de facilitar la manipulación y acceso de archivos.

Los archivos proyectados en memoria (memory-mapped files) son un tipo de archivo especial que se basan en la capacidad de la memoria virtual para utilizar espacio físico en disco como si fueran páginas de memoria RAM.

File mapping es una asociación del contenido de un archivo con un trozo del espacio de direcciones virtual de un proceso. Para ello, el sistema operativo crea un **objeto file-mapping** con el fin de mantener esta asociación. Una vista de un archivo (file view) es una parte del espacio de direcciones virtual que el proceso puede usar para acceder al contenido de dicho archivo. Los procesos leen y escriben de esta vista utilizando punteros, tal y como lo harían con memoria creada dinámicamente.

El objetivo es poder manejar ficheros de forma mucho más eficiente. El proceso utiliza la dirección de la zona correspondiente como dirección base y hace uso de un índice para acceder al elemento deseado, de la misma forma que para una variable de tipo matriz. Los archivos pueden ser compartidos por varios procesos.

Tipos de proyecciones:

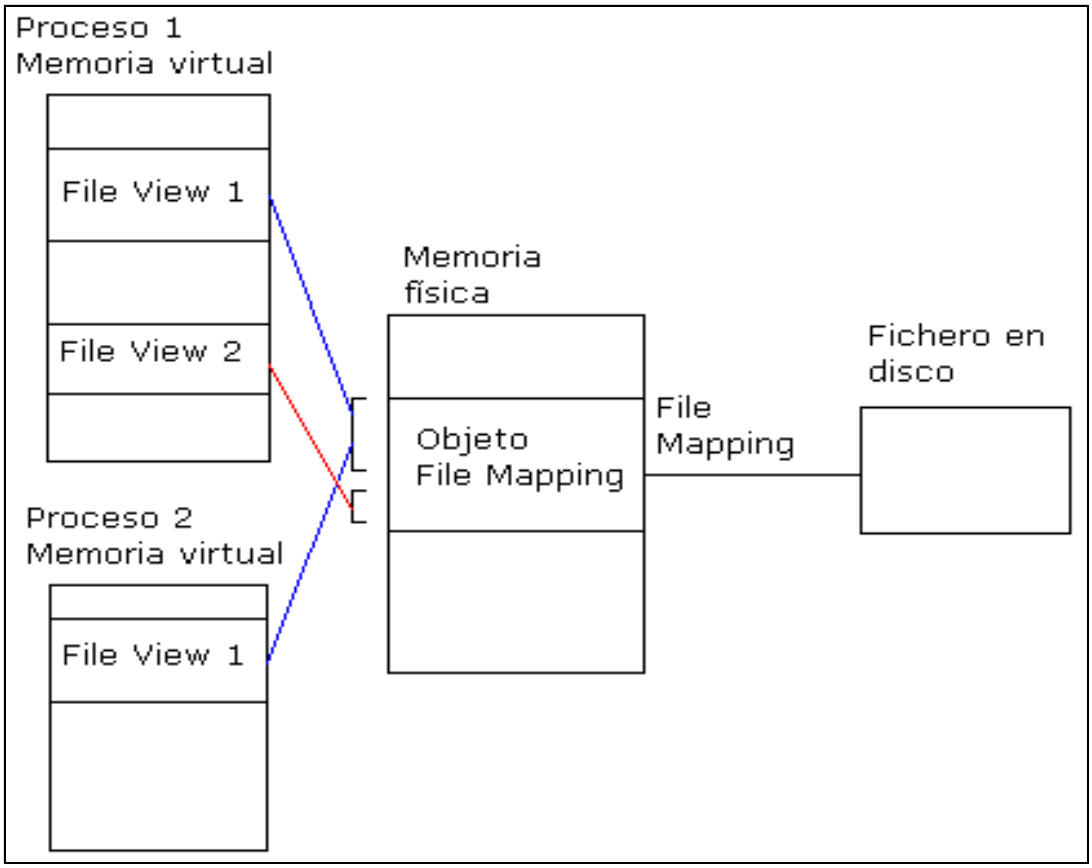
- **Proyecciones compartidas**: Varios procesos proyectan el contenido de un mismo archivo en sus espacios de direccionamiento, y toda modificación efectuada por un proceso es inmediatamente visible para todos ellos. (MAP_SHARED)

- **Proyecciones privadas**: Las modificaciones que efectúa cada proceso sobre el contenido del archivo son privadas, es decir, no son visibles para el resto de los procesos que han proyectado el mismo archivo en su espacio de direccionamiento. (MAP_PRIVATE)

- **Proyecciones anónimas**: La proyección en memoria no afecta a ningún archivo. Se crea una nueva región de memoria cuyo contenido se inicializará a cero. Normalmente utilizada para implementar un malloc propio sin necesidad de invocar directamente a la llamada del sistema sbrk(). (MAP_ANON)

Esta herramienta nos permite el poder manejar ficheros de manera mucho más eficiente. Las ventajas de mmap son:

- ✓ La e/s es considerablemente más rápida que empleando las llamadas read/write incluso cuando se dispone de cache de disco.
- ✓ El acceso es más sencillo, podemos usar punteros.
- ✓ Puede usarse para compartir datos permitiendo el acceso de múltiples procesos a los mismos.



Como se ha visto, una región de memoria puede ser asociada a una cierta porción de un archivo de un sistema de ficheros. A esta técnica se le llama memory mapping.

Un proceso puede crear un nuevo memory maping realizando la llamada al sistema mmap(). Los programadores deben especificar la bandera de MAP_SHARED o la bandera de MAP_PRIVATE como parámetro de la llamada del sistema.

Una vez que se cree el mapeo, el proceso puede leer los datos almacenados en el archivo simplemente leyendo en las posiciones de memoria de la nueva región de memoria. Si es en modo compartido, el proceso puede también modificar el archivo correspondiente simplemente escribiendo en las mismas posiciones de memoria. Como regla general, si se comparte el memory mapping, la región de memoria correspondiente tiene la bandera de VM_SHARED activada.

Para destruir un mapeo, el proceso puede utilizar la llamada del sistema del munmap().

/*Ejemplo de programa de usuario*/

```
void main ()
{
   /*...*/
```

```
  /*Obtener nombre y tamaño del archivo y abrirlo*/

  /*Proyección del archivo en memoria*/
  addr= (char *) mmap(NULL, Size, PROT_READ, MAP_SHARED, fd, (off_t) 0);

  /*Cierre del archivo*/
  /*Visualización del contenido del archivo*/
  for (int i = 0; i<Size; i++)
    putchar(addr[i]);
  /*Liberación del archivo*/
  flag=munmap (addr, Size);
  /*...*/
}
```

Llamada al sistema mmap

Mmap se corresponde con la llamada al sistema 90 y la nueva versión 192. Esta primitiva proyecta el contenido de un archivo en memoria, más concretamente, en el espacio de direcciones del proceso que la genera. Presenta la siguiente sintaxis:

```
#include <unistd.h>
#include <sys/mman.h>
void *mmap (void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

start →Dirección para alojar la proyección. Si NULL el sistema elige.
length →Cantidad del fichero a mapear.
prot →Describe la protección de memoria deseada:
PROT_EXEC = se permite ejecución
PROT_READ = se permite leer.
PROT_WRITE = se permite escribir.
PROT_NONE = las páginas no pueden ser accedidas.
Estas opciones se pueden combinar separándolas por "|".

flags →modificadores de acceso:
- MAP_FIXED No seleccionar una dirección diferente a la especificada. Si la dirección especificada no puede ser utilizada, *mmap* fallará. Utilizar esta opción es desaconsejable.
- MAP_SHARED Comparte este área con todos los otros procesos que señalan a este objeto.
- MAP_PRIVATE Crear un área privada "copy-on-write".
MAP_ANON, MAPDENY_WRITE, MAP_LOCKED.
fd →descriptor del fichero a usar. Previamente debe haber sido abierto.
Offset→desplazamiento del comienzo del mapeo en el archivo.

Hay que observar que ha de existir una coherencia entre el modo de apertura del fichero que se desea modificar y el modo de mapeo del mismo. Es decir, que si se desean realizar lecturas y escrituras sobre el archivo, esto habrá que especificarlo tanto en la llamada open como en la mmap:
OPEN → O_RDWR como segundo parámetro.

MMAP → PROT_WRITE | PROT_READ como tercer parámetro.
Si no se produce ningún error, la función devuelve un puntero al área creada.

13.3 Estructuras de datos en proyección de ficheros

Existen dos estructuras relacionadas con la memoria y los procesos que son esenciales para la implementación del mmap():

vm_area_struct

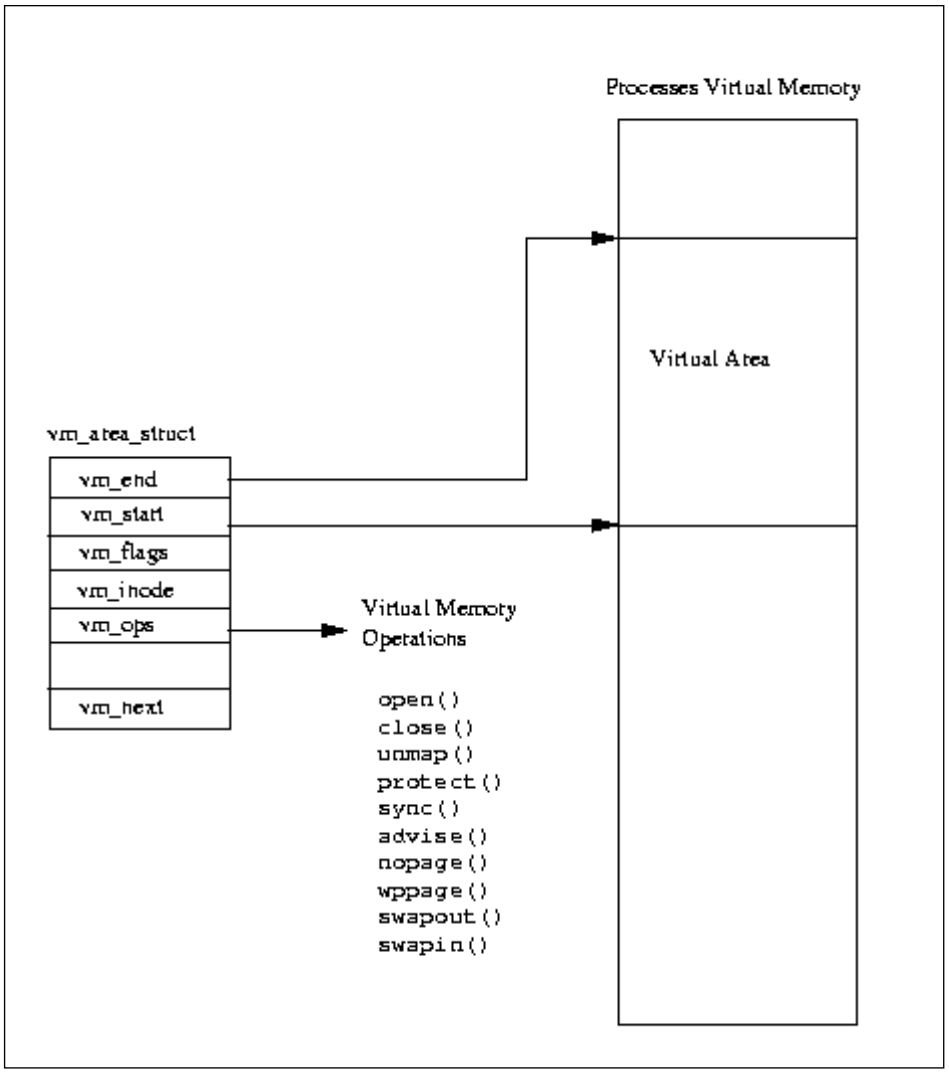
Cuyos campos más destacables son:

vm_start: Dirección de inicio.
vm_end: Dirección de fin.
vm_page_prot: Protección asociada.
vm_flags: Estado:
- VM_READ
- VM_WRITE
- VM_EXEC
- VM_SHARED
- VM_MAYREAD
- VM_MAYWRITE
- VM_MAYEXEC
- VM_MAYSHARED
- VM_DENYWRITE
- VM_LOCKED

vm_ops: Operaciones que tiene asociada.
vm_offset: Dirección de inicio de la zona de memoria respecto al inicio del objeto proyectado en memoria.

```
 99struct vm_area_struct {
100     struct mm_struct * vm_mm;    /* The address space we belong to. */
101     unsigned long vm_start;      /* Our start address within vm_mm. */
102     unsigned long vm_end;       /* The first byte after our end address
103                                        within vm_mm. */
104
105     /* linked list of VM areas per task, sorted by address */
106     struct vm_area_struct *vm_next;
107
108     pgprot_t vm_page_prot;      /* Access permissions of this VMA.
*/
109     unsigned long vm_flags;     /* Flags, listed below. */
110
111     struct rb_node vm_rb;
112
```

```
113     /*
114          * For areas with an address space and backing store,
115          * linkage into the address_space->i_mmap prio tree, or
116          * linkage to the list of like vmas hanging off its node, or
117          * linkage of vma in the address_space->i_mmap_nonlinear list.
118          */
119     union {
120          struct {
121               struct list_head list;
122               void *parent;  /* aligns with prio_tree_node parent */
123               struct vm_area_struct *head;
124          } vm_set;
125
126          struct raw_prio_tree_node prio_tree_node;
127     } shared;
128
129     /*
130          * A file's MAP_PRIVATE vma can be in both i_mmap tree and
anon_vma
131          * list, after a COW of one of the file pages.  A MAP_SHARED
vma
132          * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE,
stack
133          * or brk vma (with NULL file) can only be in an anon_vma
list.
134          */
135     struct list_head anon_vma_node; /* Serialized by anon_vma->lock
*/
136     struct anon_vma *anon_vma;   /* Serialized by page_table_lock */
137
138     /* Function pointers to deal with this struct. */
139     struct vm_operations_struct * vm_ops;
140
141     /* Information about our backing store: */
142     unsigned long vm_pgoff;     /* Offset (within vm_file) in
PAGE_SIZE
143                                    units, *not*
PAGE_CACHE_SIZE */
144     struct file * vm_file;     /* File we map to (can be NULL). */
145     void * vm_private_data;     /* was vm_pte (shared mem) */
146     unsigned long vm_truncate_count;/* truncate_count or
restart_addr */
147
148#ifndef CONFIG_MMU
149     atomic_t vm_usage;       /* refcount (VMAs shared if !MMU) */
150#endif
151#ifdef CONFIG_NUMA
152     struct mempolicy *vm_policy;  /* NUMA policy for the VMA */
153#endif
154};
```



mm-struct

Es la estructura utilizada para gestionar el mapeo de archivos, junto con el vma_struct. Veamos sus caraterísticas:

```
156struct mm_struct {
157     struct vm_area_struct * mmap;       /* list of VMAs */
158     struct rb_root mm_rb;
159     struct vm_area_struct * mmap_cache;   /* last find_vma result */
```

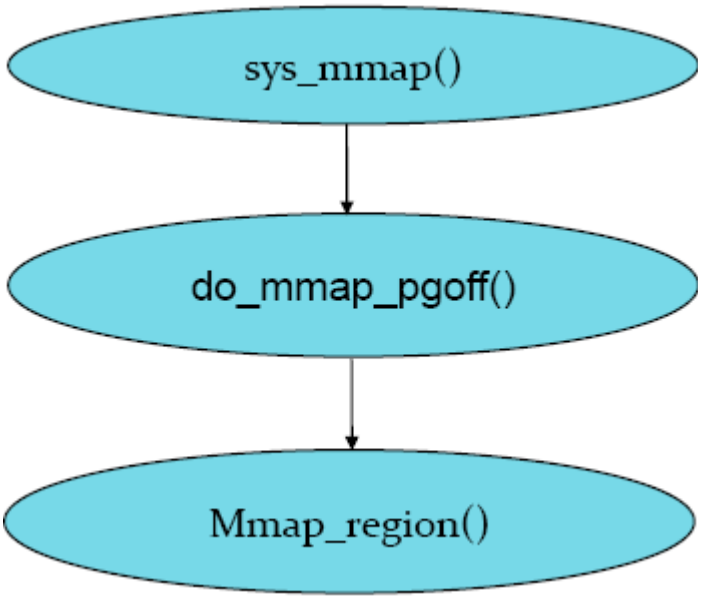
```
160     unsigned long (*get_unmapped_area) (struct file *filp,
161                     unsigned long addr, unsigned long len,
162                     unsigned long pgoff, unsigned long flags);
163     void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
164     unsigned long mmap_base;         /* base of mmap area */
165     unsigned long task_size;         /* size of task vm space */
166     unsigned long cached_hole_size;     /* if non-zero, the largest
hole below free_area_cache */
167     unsigned long free_area_cache;      /* first hole of size
cached_hole_size or larger */
168     pgd_t * pgd;
169     atomic_t mm_users;            /* How many users with user
space? */
170     atomic_t mm_count;            /* How many references to "struct
mm_struct" (users count as 1) */
171     int map_count;              /* number of VMAs */
172     struct rw_semaphore mmap_sem;
173     spinlock_t page_table_lock;       /* Protects page tables and
some counters */
174
175     struct list_head mmlist;         /* List of maybe swapped mm's.
These are globally strung
176                                         * together off
init_mm.mmlist, and are protected
177                                         * by mmlist_lock
178                                         */
179
180     /* Special counters, in some configurations protected by the
181          * page_table_lock, in other configurations by being atomic.
182          */
183     mm_counter_t _file_rss;
184     mm_counter_t _anon_rss;
185
186     unsigned long hiwater_rss;   /* High-watermark of RSS usage */
187     unsigned long hiwater_vm;    /* High-water virtual memory usage
*/
188
189     unsigned long total_vm, locked_vm, shared_vm, exec_vm;
190     unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
191     unsigned long start_code, end_code, start_data, end_data;
192     unsigned long start_brk, brk, start_stack;
193     unsigned long arg_start, arg_end, env_start, env_end;
194
195     unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv
*/
196
197     cpumask_t cpu_vm_mask;
198
199     /* Architecture-specific MM context */
200     mm_context_t context;
```

```
 201
 202     /* Swap token stuff */
 203     /*
 204          * Last value of global fault stamp as seen by this process.
 205          * In other words, this value gives an indication of how long
 206          * it has been since this task got the token.
 207          * Look at mm/thrash.c
 208          */
 209     unsigned int faultstamp;
 210     unsigned int token_priority;
 211     unsigned int last_interval;
 212
 213     unsigned long flags; /* Must use atomic bitops to access the bits
*/
 214
 215     /* coredumping support */
 216     int core_waiters;
 217     struct completion *core_startup_done, core_done;
 218
 219     /* aio bits */
 220     rwlock_t        ioctx_list_lock;
 221     struct kioctx       *ioctx_list;
 222};
```

13.4 mmap

El flujo de ejecución que sigue el núcleo para atender a la llamada ***mmap()*** es el siguiente:

Sys_mmap, llama a **do_mmap_pgoff**
do_mmap_pgoff llama a **mmap_region**



sys_mmap

```
  36asmlinkage long sys_mmap(unsigned long addr, unsigned long len,
unsigned long prot, unsigned long flags,
  37      unsigned long fd, unsigned long off)
  38{
  39      long error;
  40      struct file * file;
  41
  42      error = -EINVAL;
  43      if (off & ~PAGE_MASK)
  44          goto out;
  45
  46      error = -EBADF;
  47      file = NULL;
  48      flags &= ~(MAP_EXECUTABLE | MAP_DENYWRITE);
  49      if (!(flags & MAP_ANONYMOUS)) {
  50          file = fget(fd);
  51          if (!file)
  52              goto out;
  53      }
  54      down_write(&current->mm->mmap_sem);
  55      error = do_mmap_pgoff(file, addr, len, prot, flags, off >>
PAGE_SHIFT);
  56      up_write(&current->mm->mmap_sem);
```

```
57
58      if (file)
59          fput(file);
60out:
61      return error;
62}
```

do_mmap_pgoff

Su ejecución sigue a rasgos generales los siguientes pasos:

1. Comprueba la validez de los parámetros.
2. Busca un rango de memoria no mapeado.
3. Verifica compatibilidades archivo – proyección.
4. Llamada a la función mmap_region

```
891unsigned long do_mmap_pgoff(struct file * file, unsigned long addr,
892 unsigned long len, unsigned long prot,
893 unsigned long flags, unsigned long pgoff)
894{
895 struct mm_struct * mm = current->mm;
896 struct inode *inode;
897 unsigned int vm_flags;
898 int error;
899 int accountable = 1;
 900 unsigned long reqprot = prot;
```

1. Comprueba la validez de los parámetros:

```
908 if ((prot & PROT_READ) && (current->personality & READ_IMPLIES_EXEC))
909 if (!(file && (file->f_path.mnt->mnt_flags & MNT_NOEXEC)))
910 prot |= PROT_EXEC;
912 if (!len)
913 return -EINVAL;
915 if (!(flags & MAP_FIXED))
916     addr = round_hint_to_min(addr);
918 error = arch_mmap_check(addr, len, flags);
919 if (error)
920      return error;
```

Comprueba que no haya overflow.

```
923 len = PAGE_ALIGN(len);
924 if (!len || len > TASK_SIZE)
925 return -ENOMEM;
```

Revisa que no sobrepase el máximo de mappings en memoria

```
932 if (mm->map_count > sysctl_max_map_count)
933 return -ENOMEM;
```

2. Busca un rango de memoria no mapeado:

Obtiene una dirección de memoria de un área libre a la que tiene que mapear.

y se asegura de que representa un sector correcto del espacio de direcciones.

```
938 addr = get_unmapped_area(file, addr, len, pgoff, flags);
 939 if (addr & ~PAGE_MASK)
940 return addr;
```

3. Verifica compatibilidades archivo – proyección

```
965 inode = file ? file->f_path.dentry->d_inode : NULL;
967 if (file) {
968 switch (flags & MAP_TYPE) {
 969 case MAP_SHARED:
970 if ((prot&PROT_WRITE) && !(file->f_mode&FMODE_WRITE))
971 return -EACCES;
```

Comprobamos que no permitimos escritura para un fichero abierto en modo append.

```
977 if (IS_APPEND(inode) && (file->f_mode & FMODE_WRITE))
978 return -EACCES;
```

Comprueba que el fichero no tiene locks de mando, que es cuando el bit setgid esta activado pero no tiene el bit de ejecucion de grupo.

```
983 if (locks_verify_locked(inode))
984 return -EAGAIN;
```

Si está mapeado en modo compartido.

```
986 vm_flags |= VM_SHARED | VM_MAYSHARE;
```

Comprueba que el fichero permita la escritura así como el mapeo

```
987 if (!(file->f_mode & FMODE_WRITE))
988 vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
```

Para el caso del mapeo privado

Si está mapeado en modo privado, en caso que no esté en modo lectura o el mapeo permita ejecución devuelve un error.

```
991 case MAP_PRIVATE:
```

```
992 if (!(file->f_mode & FMODE_READ))
993 return -EACCES;
994 if (file->f_path.mnt->mnt_flags & MNT_NOEXEC) {
995 if (vm_flags & VM_EXEC)
996 return -EPERM;
997 vm_flags &= ~VM_MAYEXEC;
```

4. LLAMADA A MMAP_REGION

```
1029 return mmap_region(file, addr, len, flags, vm_flags, pgoff,
1030 accountable);
1031});
```

mmap_region

Realiza las siguientes funciones:

1. Comprueba si el rango de direcciones addr+len está dentro de una VMA y eliminar mapeos anteriores.
2. Creamos espacio para una VMA.
3. Inicializamos los campos de la estructura vma
4. Comprobar si se puede escribir en la región y enlazamos la vma
5. Finalizar la proyección actualizando la estructura mm del proceso y devolver la dirección de comienzo del VMA.

```
1067unsigned long mmap_region(struct file *file, unsigned long addr,
1068                          unsigned long len, unsigned long flags,
1069                          unsigned int vm_flags, unsigned long pgoff,
1070                          int accountable)
1071{
1072     struct mm_struct *mm = current->mm;
1073     struct vm_area_struct *vma, *prev;
1074     int correct_wcount = 0;
1075     int error;
1076     struct rb_node **rb_link, *rb_parent;
1077     unsigned long charged = 0;
1078     struct inode *inode =  file ? file->f_path.dentry->d_inode : NULL;
1079
1080     /* Clear old maps */
1081     error = -ENOMEM;
```

1.Comprueba si el rango de direcciones addr+len esta dentro de una VMA y eliminar mapeos anteriores.

```
1082munmap_back:
1083        vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
1084        if (vma && vma->vm_start < addr + len) {
1085                if (do_munmap(mm, addr, len))
1086                        return -ENOMEM;
1087                goto munmap_back;
1088        }
1089
1090        /* Check against address space limit. */
```

Verificamos el límite del espacio de direccionamiento

```
1091        if (!may_expand_vm(mm, len >> PAGE_SHIFT))
1092                return -ENOMEM;
1093
1094        if (accountable && (!(flags & MAP_NORESERVE) ||
1095                        sysctl_overcommit_memory == OVERCOMMIT_NEVER)) {
1096                if (vm_flags & VM_SHARED) {
1097                        /* Check memory availability in shmem_file_setup? */
1098                        vm_flags |= VM_ACCOUNT;
1099                } else if (vm_flags & VM_WRITE) {
1100                        /*
1101                         * Private writable mapping: check memory availability
1102                         */
1103                        charged = len >> PAGE_SHIFT;
1104                        if (security_vm_enough_memory(charged))
1105                                return -ENOMEM;
1106                        vm_flags |= VM_ACCOUNT;
1107                }
1108        }
1109
1110        /*
1111         * Can we just expand an old private anonymous mapping?
1112         * The VM_SHARED test is necessary because shmem_zero_setup
1113         * will create the file object for a shared anonymous map below.
1114         */
1115        if (!file && !(vm_flags & VM_SHARED) &&
1116            vma_merge(mm, prev, addr, addr + len, vm_flags,
1117                                        NULL, NULL, pgoff, NULL))
1118                goto out;
1119
```

2. Creamos espacio para una vma

Determinamos el tipo de objeto que vamos a mapear y llamamos a la función específica para mapear ese tipo de objeto.

```
1120        /*
1121         * Determine the object being mapped and call the appropriate
1122         * specific mapper. the address has already been validated, but
1123         * not unmapped, but the maps are removed from the list.
```

```
1124        */
1125       vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);
1126       if (!vma) {
1127             error = -ENOMEM;
1128             goto unacct_error;
1129       }
1130
```

3. icializamos los campos de la estructura vma

```
1131       vma->vm_mm = mm;
1132       vma->vm_start = addr;
1133       vma->vm_end = addr + len;
1134       vma->vm_flags = vm_flags;
1135       vma->vm_page_prot = vm_get_page_prot(vm_flags);
1136       vma->vm_pgoff = pgoff;
1137
```

4-.Comprobamos si se puede escribir en la región:

```
1138       if (file) {
1139             error = -EINVAL;
1140             if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
1141                   goto free_vma;
1142             if (vm_flags & VM_DENYWRITE) {No se puede escribir sobre la
región
1143                   error = deny_write_access(file);
1144                   if (error)
1145                         goto free_vma; buscar una nueva región libre
1146                   correct_wcount = 1; la región actual posee permisos de escritura
1147             }
1148             vma->vm_file = file; //asociamos el archivo a proyectar  en la VMA
actual
1149             get_file(file);
// tratamos de realizar un mmap con “file”
1150             error = file->f_op->mmap(file, vma);
1151             if (error)
1152                   goto unmap_and_free_vma; /desmaperar y liberar la VMA
1153       } else if (vm_flags & VM_SHARED) {
1154             error = shmem_zero_setup(vma);
1155             if (error)
1156                   goto free_vma;
1157       }
1158
1159       /* We set VM_ACCOUNT in a shared mapping's vm_flags, to inform
1160        * shmem_zero_setup (perhaps called through /dev/zero's ->mmap)
1161        * that memory reservation must be checked; but that reservation
1162        * belongs to shared memory object, not to vma: so now clear it.
1163        */
1164       if ((vm_flags & (VM_SHARED|VM_ACCOUNT)) ==
(VM_SHARED|VM_ACCOUNT))
1165             vma->vm_flags &= ~VM_ACCOUNT;
```

```
1166
1167        /* Can addr have changed??
1168         *
1169         * Answer: Yes, several device drivers can do it in their
1170         *         f_op->mmap method. -DaveM
1171         */
1172        addr = vma->vm_start;
1173        pgoff = vma->vm_pgoff;
1174        vm_flags = vma->vm_flags;
1175
1176        if (vma_wants_writenotify(vma))
1177                vma->vm_page_prot = vm_get_page_prot(vm_flags &
~VM_SHARED);
1178
1179        if (!file || !vma_merge(mm, prev, addr, vma->vm_end,
1180                        vma->vm_flags, NULL, file, pgoff, vma_policy(vma))) {
1181                file = vma->vm_file;
```

Enlaza el VMA en la lista y en el árbol rb de VMAs del proceso

```
1182                vma_link(mm, vma, prev, rb_link, rb_parent);
1183                if (correct_wcount)
1184                        atomic_inc(&inode->i_writecount);
1185        } else {
1186                if (file) {
1187                        if (correct_wcount)
1188                                atomic_inc(&inode->i_writecount);
1189                        fput(file);
1190                }
1191                mpol_free(vma_policy(vma));
1192                kmem_cache_free(vm_area_cachep, vma);
1193        }
```

5-.Finalizamos la proyección actualizando la estructura mm del proceso y devolvemos la dirección de comienzo del VMA.

```
1194out:
1195        mm->total_vm += len >> PAGE_SHIFT;
1196        vm_stat_account(mm, vm_flags, file, len >> PAGE_SHIFT);
1197        if (vm_flags & VM_LOCKED) {
1198                mm->locked_vm += len >> PAGE_SHIFT;
1199                make_pages_present(addr, addr + len);
1200        }
1201        if ((flags & MAP_POPULATE) && !(flags & MAP_NONBLOCK))
1202                make_pages_present(addr, addr + len);
1203        return addr; /devuelve la dirección de mapeo /
1204
```

Entrada de errores: Intentos de mapeos erróneos serán eliminados

```
1205unmap_and_free_vma:
1206        if (correct_wcount)
1207                atomic_inc(&inode->i_writecount);
1208        vma->vm_file = NULL;
```

```
1209        fput(file); /termina con el uso de la estructura file/
1210/* Deshacer mapeos parciales realizados */
1211        /* Undo any partial mapping done by a device driver. */
1212        unmap_region(mm, vma, prev, vma->vm_start, vma->vm_end);
1213        charged = 0;
1214free_vma:
1215        kmem_cache_free(vm_area_cachep, vma);
1216unacct_error:
1217        if (charged)
1218                vm_unacct_memory(charged);
1219        return error;
1220}
```

13.5 munmap

int ***munmap **(void *start**, size_t** length**)**

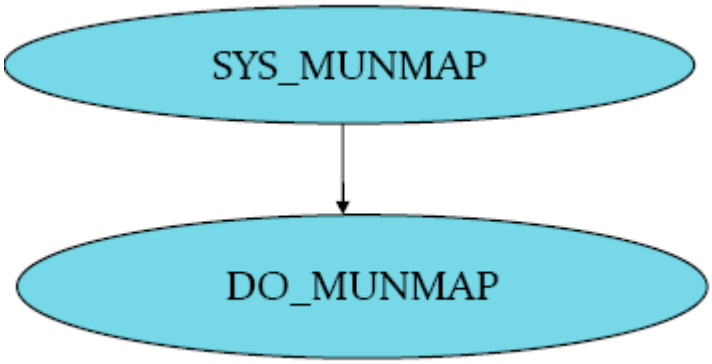
start → Especifica la dirección de inicio de la región/zona de memoria correspondiente a la proyección que se desea suprimir.

length → Nº de bytes de la región.

Si no se produce ningún error, la función devuelve 0.

Según podemos leer en Understanding the Linux kernel 3rd Edition “Hay que fijarse que aquí no es necesario volcar el contendido de las páginas de una proyección de memoria compartida para ser destruidas. En realidad, estas páginas siguen actuando como caché de disco porque ellas todavía están incluidas en la caché de páginas”.

El orden de ejecución es el siguiente, munmap() hace una llamada al sistema, que aterriza a la función sys_munmap:



La función sys_munmap está implementada como sigue:

```
1846 asmlinkage long sys_munmap(unsigned long addr, size_t len)
1847 {
1848         int ret;
1849         struct mm_struct *mm = current->mm;
1851         profile_munmap(addr);
```

Deshabilitamos la escritura en memoria, llamamos a do_munmap y luego volvemos a habilitar la escritura en la memoria:

```
1853         down_write(&mm->mmap_sem);
1854         ret = do_munmap(mm, addr, len);
1855         up_write(&mm->mmap_sem);
1856         return ret;
1857 }
```

do_munmap

```
1787 int do_munmap(struct mm_struct *mm, unsigned long start, size_t len)
1788 {
1789     unsigned long end;
1790     struct vm_area_struct *vma, *prev, *last;
```

Si el tamaño o la dirección de memoria no son validas devuelve un error.

```
1792     if ((start & ~PAGE_MASK) || start > TASK_SIZE || len > TASK_SIZE-start)
1793         return -EINVAL;
```

Si hay paginas no liberadas devuelve un error.

```
1795     if ((len = PAGE_ALIGN(len)) == 0)
1796         return -EINVAL;
```

Busca la vma que contenga la dirección de memoria de comienzo.

```
1798        /* Find the first overlapping VMA */
1799        vma = find_vma_prev(mm, start, &prev);
1800        if (!vma)
1801                return 0;
1802        /* we have  start < vma->vm_end  */
```

Comprueba que la dirección de inicio del mapeo sea menor que el final que hemos especificado a través de la suma del comienzo mas la longitud.

```
1804        /* if it doesn't overlap, we have nothing.. */
1805        end = start + len;
1806        if (vma->vm_start >= end)
1807                return 0;
1808
1809        /*
1810         * If we need to split any vma, do it now to save pain later.
1811         *
1812         * Note: mremap's move_vma VM_ACCOUNT handling assumes a
partially
1813         * unmapped vm_area_struct will remain in use: so lower split_vma
1814         * places tmp vma above, and higher split_vma places tmp vma below.
1815         */
```

Comprueba si tiene que dividir una vma, ya que la dirección de inicio del mapeo puede estar a mitad de una vma, si es así aquí la divide para liberar solo la parte correspondiente al mapeo.

```
1816        if (start > vma->vm_start) {
1817                int error = split_vma(mm, vma, start, 0);
1818                if (error)
1819                        return error;
1820                prev = vma;
1821        }
```

Comprueba si tiene que dividir la ultima vma, ya que el mapeo puede acabar a mitad de una vma, si es así la divide para liberar solo la parte correspondiente al mapeo.

```
1823        /* Does it split the last one? */
1824        last = find_vma(mm, end);
1825        if (last && end > last->vm_start) {
1826                int error = split_vma(mm, last, end, 1);
1827                if (error)
1828                        return error;
1829        }
1830        vma = prev? prev->vm_next: mm->mmap;
```

Remueve las vma’s y hace un unmap de las paginas actuales.

```
1832        /*
1833         * Remove the vma's, and unmap the actual pages
1834         */
1835        detach_vmas_to_be_unmapped(mm, vma, prev, end);
1836        unmap_region(mm, vma, prev, start, end);
```

Libera la lista de vma's.

```
1838        /* Fix up all other VM information */
1839        remove_vma_list(mm, vma);
1840
1841        return 0;
1842 }
```

13.6 Funciones auxiliares

unmap_region

Desmapea una región a través de una llamada a unmap_vmas y se ocupa de liberar otras estructuras.

Parámetros
- mm: descriptor de espacio de direccionamiento.
- vma, prev: las dos vma's donde comienza el mapeo.
- start: dirección de comienzo de la región.
- end: dirección de fin de la región.

Elimina la información de la tabla de páginas de la región indicada.
La función es llamada con el cerrojo de la tabla de páginas cogido.

```
1678 /*
1679  * Get rid of page table information in the indicated region.
1680  *
1681  * Called with the mm semaphore held.
1682  */
1683 static void unmap_region(struct mm_struct *mm,
1684              struct vm_area_struct *vma, struct vm_area_struct *prev,
1685                unsigned long start, unsigned long end)
1686 {
1687        struct vm_area_struct *next = prev? prev->vm_next: mm->mmap;
1688        struct mmu_gather *tlb;
1689        unsigned long nr_accounted = 0;
1690
1691        lru_add_drain();
1692        tlb = tlb_gather_mmu(mm, 0);
1693        update_hiwater_rss(mm);
```

//En esta funcion se escribe en memoria.

```
1694        unmap_vmas(&tlb, vma, start, end, &nr_accounted, NULL);
1695        vm_unacct_memory(nr_accounted);
1696        free_pgtables(&tlb, vma, prev? prev->vm_end: FIRST_USER_ADDRESS,
1697                                 next? next->vm_start: 0);
1698        tlb_finish_mmu(tlb, start, end);
1699 }
```

unmap_vmas

Desmapea toda una region de vma's.

Parámetros:
- tlbp: dirección de la estructura mmu_gather del llamador de la función.
- mm: la estructura mm_struct de control.
- vma: la vma de comienzo.
- start_addr: dirección virtual en la que comenzar a desmapear.
- end_addr: dirección virtual en la que comenzar a desmapear.
- nr_accounted: número de páginas desmapeadas en vm-accountable.
- details: detalles de truncar no lineal o invalidación de cache compartida.

Las acciones que realiza son las siguientes:

Devuelve el numero de vma's que fueron cubiertas por el desmapeado. Desmapea todas las páginas en la lista vma.

La función es llamada con el cerrojo de la tabla de páginas cogido. Solo las direcciones entre start y end serán desmapeadas.

La lista vma debe ser recorrida en orden ascendente de las direcciones virtuales.

La función asume que el que llama a la función hará un flush de todo el rango de direcciones desmapeadas después de que la función retorne.

```
785 /**
786  * unmap_vmas - unmap a range of memory covered by a list of vma's
787  * @tlbp: address of the caller's struct mmu_gather
788  * @vma: the starting vma
789  * @start_addr: virtual address at which to start unmapping
790  * @end_addr: virtual address at which to end unmapping
791  * @nr_accounted: Place number of unmapped pages in vm-accountable vma's
here
792  * @details: details of nonlinear truncation or shared cache invalidation
793  *
794  * Returns the end address of the unmapping (restart addr if interrupted).
795  *
796  * Unmap all pages in the vma list.
797  *
798  * We aim to not hold locks for too long (for scheduling latency reasons).
799  * So zap pages in ZAP_BLOCK_SIZE bytecounts.  This means we need to
800  * return the ending mmu_gather to the caller.
```

```
801  *
802  * Only addresses between `start' and `end' will be unmapped.
803  *
804  * The VMA list must be sorted in ascending virtual address order.
805  *
806  * unmap_vmas() assumes that the caller will flush the whole unmapped address
807  * range after unmap_vmas() returns.  So the only responsibility here is to
808  * ensure that any thus-far unmapped pages are flushed before unmap_vmas()
809  * drops the lock and schedules.
810  */
811 unsigned long unmap_vmas(struct mmu_gather **tlbp,
812             struct vm_area_struct *vma, unsigned long start_addr,
813             unsigned long end_addr, unsigned long *nr_accounted,
814             struct zap_details *details)
815 {
816         long zap_work = ZAP_BLOCK_SIZE;
817         unsigned long tlb_start = 0;    /* For tlb_finish_mmu */
818         int tlb_start_valid = 0;
819         unsigned long start = start_addr;
820         spinlock_t *i_mmap_lock = details? details->i_mmap_lock: NULL;
821         int fullmm = (*tlbp)->fullmm;
822
823         for ( ; vma && vma->vm_start < end_addr; vma = vma->vm_next) {
824             unsigned long end;
825
826             start = max(vma->vm_start, start_addr);
827             if (start >= vma->vm_end)
828                 continue;
829             end = min(vma->vm_end, end_addr);
830             if (end <= vma->vm_start)
831                 continue;
832
833             if (vma->vm_flags & VM_ACCOUNT)
834                 *nr_accounted += (end - start) >> PAGE_SHIFT;
835
836             while (start != end) {
837                 if (!tlb_start_valid) {
838                     tlb_start = start;
839                     tlb_start_valid = 1;
840                 }
841
842                 if (unlikely(is_vm_hugetlb_page(vma))) {
843                     unmap_hugepage_range(vma, start, end);
844                     zap_work -= (end - start) /
845                             (HPAGE_SIZE / PAGE_SIZE);
846                     start = end;
847                 } else
848                     start = unmap_page_range(*tlbp, vma,
849                             start, end, &zap_work, details);
850
```

```
851                     if (zap_work > 0) {
852                             BUG_ON(start != end);
853                             break;
854                     }
855
856                     tlb_finish_mmu(*tlbp, tlb_start, start);
857
858                     if (need_resched() ||
859                             (i_mmap_lock && need_lockbreak(i_mmap_lock))) {
860                             if (i_mmap_lock) {
861                                     *tlbp = NULL;
862                                     goto out;
863                             }
864                             cond_resched();
865                     }
866
867                     *tlbp = tlb_gather_mmu(vma->vm_mm, fullmm);
868                     tlb_start_valid = 0;
869                     zap_work = ZAP_BLOCK_SIZE;
870             }
871     }
872 out:
873     return start;   /* which is now the end (or restart) address */
874 }
```

Función find_vma_prev y Función find_vma

Sus objetivos son similares, buscan el vma indicada por los parámetros de entrada. Pero find_vma devuelve el puntero al VMA anterior.

Parámetros

- mm: decriptor de espacio de direccionamiento.
- addr: dirección especificada.
- pprev: predecesor del VMA

Esta parte es nueva antes usaba un árbol AVL y ahora usa un árbol rojo negro.

```
1436 /* Same as find_vma, but also return a pointer to the previous VMA in *pprev. */
1437 struct vm_area_struct *
1438 find_vma_prev(struct mm_struct *mm, unsigned long addr,
1439                     struct vm_area_struct **pprev)
1440 {
1441     struct vm_area_struct *vma = NULL, *prev = NULL;
1442     struct rb_node * rb_node;
1443     if (!mm)
1444             goto out;
1445
1446     /* Guard against addr being lower than the first VMA */
1447     vma = mm->mmap;
```

Recorre el árbol rojo negro en busca del nodo que corresponde a la direccion de memoria pasada.

```
1449        /* Go through the RB tree quickly. */
1450        rb_node = mm->mm_rb.rb_node;
1451
1452        while (rb_node) {
1453                struct vm_area_struct *vma_tmp;
1454                vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
1455
1456                if (addr < vma_tmp->vm_end) {
1457                        rb_node = rb_node->rb_left;
1458                } else {
1459                        prev = vma_tmp;
1460                        if (!prev->vm_next || (addr < prev->vm_next->vm_end))
1461                                break;
1462                        rb_node = rb_node->rb_right;
1463                }
1464        }
1465
1466 out:
1467        *pprev = prev;
1468        return prev ? prev->vm_next : vma;
1469 }
```

detach_vmas_to_be_unmapped

Desenlaza las vma's, con su árbol rojo negro, para que puedan ser desmapeadas por otras funciones.

Parámetros:
- mm: descriptor de espacio de direccionamiento.
- vma: la vma que se desea desmanear.

Crea una lista de vma's tocadas por el unmap, removiéndolas de la lista de vma's de la mm.

```
1701 /*
1702  * Create a list of vma's touched by the unmap, removing them from the mm's
1703  * vma list as we go..
1704  */
1705 static void
1706 detach_vmas_to_be_unmapped(struct mm_struct *mm, struct vm_area_struct
*vma,
1707        struct vm_area_struct *prev, unsigned long end)
1708 {
1709        struct vm_area_struct **insertion_point;
1710        struct vm_area_struct *tail_vma = NULL;
1711        unsigned long addr;
1712
1713        insertion_point = (prev ? &prev->vm_next : &mm->mmap);
```

Recorre la lista eliminando cada nodo de la misma.

```
1714        do {
1715                rb_erase(&vma->vm_rb, &mm->mm_rb);
1716                mm->map_count--;
1717                tail_vma = vma;
1718                vma = vma->vm_next;
1719        } while (vma && vma->vm_start < end);
1720        *insertion_point = vma;
1721        tail_vma->vm_next = NULL;
```

Destruye la cache

```
1722        if (mm->unmap_area == arch_unmap_area)
1723                addr = prev ? prev->vm_end : mm->mmap_base;
1724        else
1725                addr = vma ?  vma->vm_start : mm->mmap_base;
1726        mm->unmap_area(mm, addr);
1727        mm->mmap_cache = NULL;          /* Kill the cache. */
1728 }
```

remove_vma_list

Desmapea la lista que contiene las vma's.

Parámetros:
- mm: descriptor de espacio de direccionamiento.
- *mpnt: puntero al comienzo de la lista.

Elimina las vma's de la lista y actualiza la lista.

```
1656 /*
1657  * Ok - we have the memory areas we should free on the vma list,
1658  * so release them, and do the vma updates.
1659  *
1660  * Called with the mm semaphore held.
1661  */
1662 static void remove_vma_list(struct mm_struct *mm, struct vm_area_struct
*vma)
1663 {
1664        /* Update high watermark before we lower total_vm */
1665        update_hiwater_vm(mm);
1666        do {
1667                long nrpages = vma_pages(vma);
1668
1669                mm->total_vm -= nrpages;
1670                if (vma->vm_flags & VM_LOCKED)
1671                        mm->locked_vm -= nrpages;
1672                vm_stat_account(mm, vma->vm_flags, vma->vm_file, -nrpages);
1673                vma = remove_vma(vma);
```

```
1674        } while (vma);
1675        validate_mm(mm);
1676 }
```

13.7 msync

La llamada al sistema msync() puede ser usada por un proceso para volcar a disco las páginas de una proyección de memoria compartida que hayan sido modificadas. Recibe como parámetros la dirección de comienzo del intervalo, la longitud del mismo, y una banderas que pueden ser:

- MS_SYNC

Le pide a la llamada al sistema que suspenda el proceso hasta que la operación de E/S se haya completado. De esta forma, el proceso que llamó puede asumir que cuando la llamada al sistema termina, todas las páginas de esta proyección de memoria han sido volcadas a disco.

- MS_ASYNC (contraria a MS_SYNC)

Le pide a la llamada al sistema que retorne inmediatamente sin suspender al proceso llamador.

-MS_INVALIDATE

Le pide a la llamada al sistema que invalide otras proyecciones de memoria de este mismo fichero.

```
16 /*
17  * MS_SYNC syncs the entire file - including mappings.
18  *
19  * MS_ASYNC does not start I/O (it used to, up to 2.5.67).
20  * Nor does it marks the relevant pages dirty (it used to up to 2.6.17).
21  * Now it doesn't do anything, since dirty pages are properly tracked.
22  *
23  * The application may now run fsync() to
24  * write out the dirty pages and wait on the writeout and check the result.
25  * Or the application may run fadvise(FADV_DONTNEED) against the fd to start
26  * async writeout immediately.
27  * So by _not_ starting I/O in MS_ASYNC we provide complete flexibility to
28  * applications.
29  */
30 asmlinkage long sys_msync(unsigned long start, size_t len, int flags)
31 {
32         unsigned long end;
33         struct mm_struct *mm = current->mm;
34         struct vm_area_struct *vma;
35         int unmapped_error = 0;
36         int error = -EINVAL;
37
38         if (flags & ~(MS_ASYNC | MS_INVALIDATE | MS_SYNC))
39                 goto out;
40         if (start & ~PAGE_MASK)
```

```
41              goto out;
42      if ((flags & MS_ASYNC) && (flags & MS_SYNC))
43              goto out;
44      error = -ENOMEM;
45      len = (len + ~PAGE_MASK) & PAGE_MASK;
46      end = start + len;
47      if (end < start)
48              goto out;
49      error = 0;
50      if (end == start)
51              goto out;
52      /*
53       * If the interval [start,end) covers some unmapped address ranges,
54       * just ignore them, but return -ENOMEM at the end.
55       */
56      down_read(&mm->mmap_sem);
57      vma = find_vma(mm, start);
58      for (;;) {
59              struct file *file;
60
61              /* Still start < end. */
62              error = -ENOMEM;
63              if (!vma)
64                      goto out_unlock;
65              /* Here start < vma->vm_end. */
66              if (start < vma->vm_start) {
67                      start = vma->vm_start;
68                      if (start >= end)
69                              goto out_unlock;
70                      unmapped_error = -ENOMEM;
71              }
72              /* Here vma->vm_start <= start < vma->vm_end. */
73              if ((flags & MS_INVALIDATE) &&
74                              (vma->vm_flags & VM_LOCKED)) {
75                      error = -EBUSY;
76                      goto out_unlock;
77              }
```

En este fragmento de código es donde se escribe en disco, podemos ver como se hace un fput.

```
78              file = vma->vm_file;
79              start = vma->vm_end;
80              if ((flags & MS_SYNC) && file &&
81                              (vma->vm_flags & VM_SHARED)) {
82                      get_file(file);
83                      up_read(&mm->mmap_sem);
84                      error = do_fsync(file, 0);
85                      fput(file); //Escribimos las modificaciones en disco
86                      if (error || start >= end)
87                              goto out;
```

```
88                      down_read(&mm->mmap_sem);
89                      vma = find_vma(mm, start);
90              } else {
91                      if (start >= end) {
92                              error = 0;
93                              goto out_unlock;
94                      }
95                      vma = vma->vm_next;
96              }
97      }
98 out_unlock:
99      up_read(&mm->mmap_sem);
100 out:
101     return error ? : unmapped_error;
102 }
```