PROYECCIÓN DE ARCHIVOS EN MEMORIA

mmap

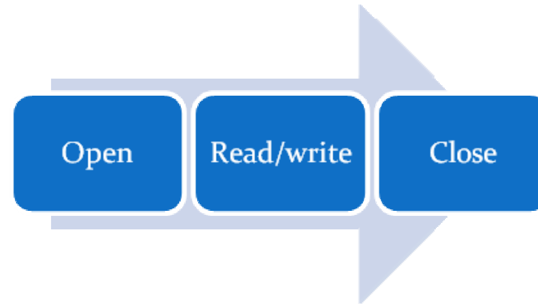
munmap

Francisco Ruano Suárez

Armide González Arencibia

Introducción

Para modificar ficheros hasta el momento hemos tenido que seguir los pasos que se muestran a continuación:

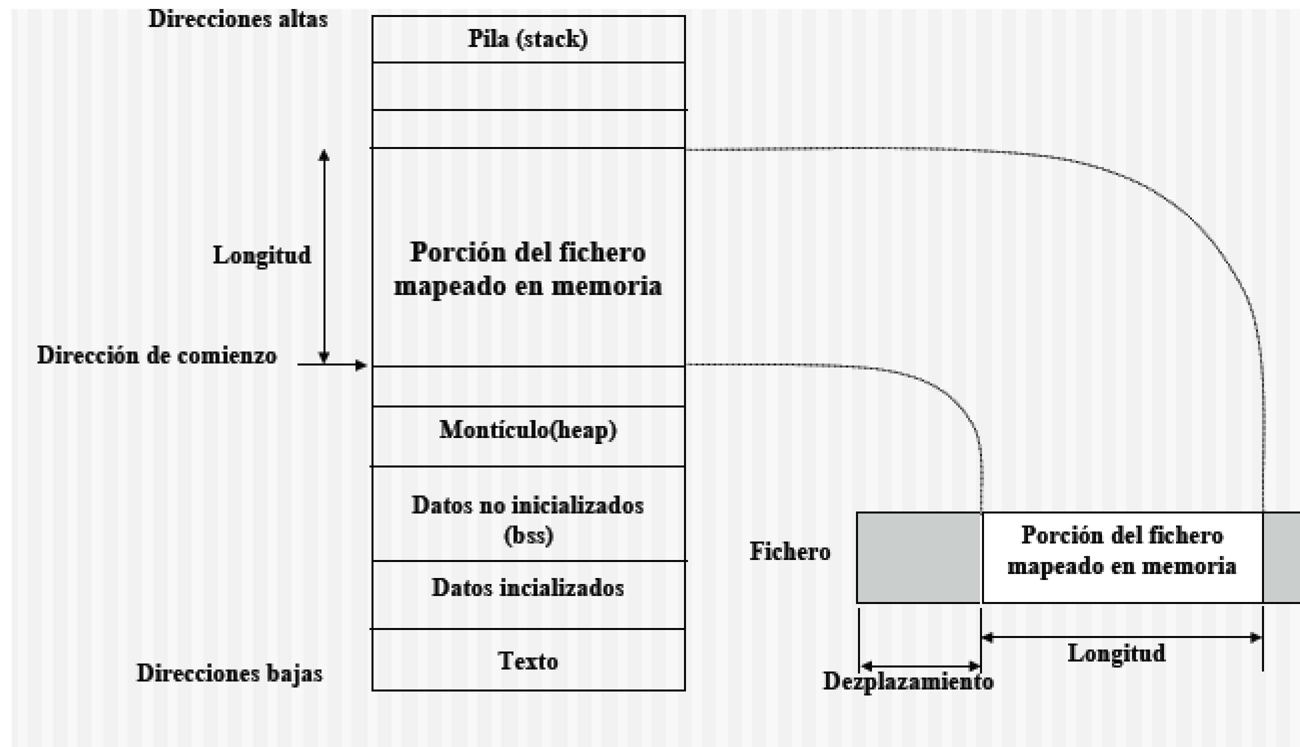


Se puede crear una correspondencia de un archivo de disco con la memoria, bit a bit.

Las proyecciones de archivos en memoria ofrecen las siguientes **ventajas**:

- La e/s es más rápida que usando read y write incluso con la existencia de cache de disco.
- El acceso es más sencillo, podemos usar punteros.
- Pueden usarse para compartir datos permitiendo el acceso de múltiples procesos a los mismos.

Recordatorio: **Espacio de direccionamiento (ED)**



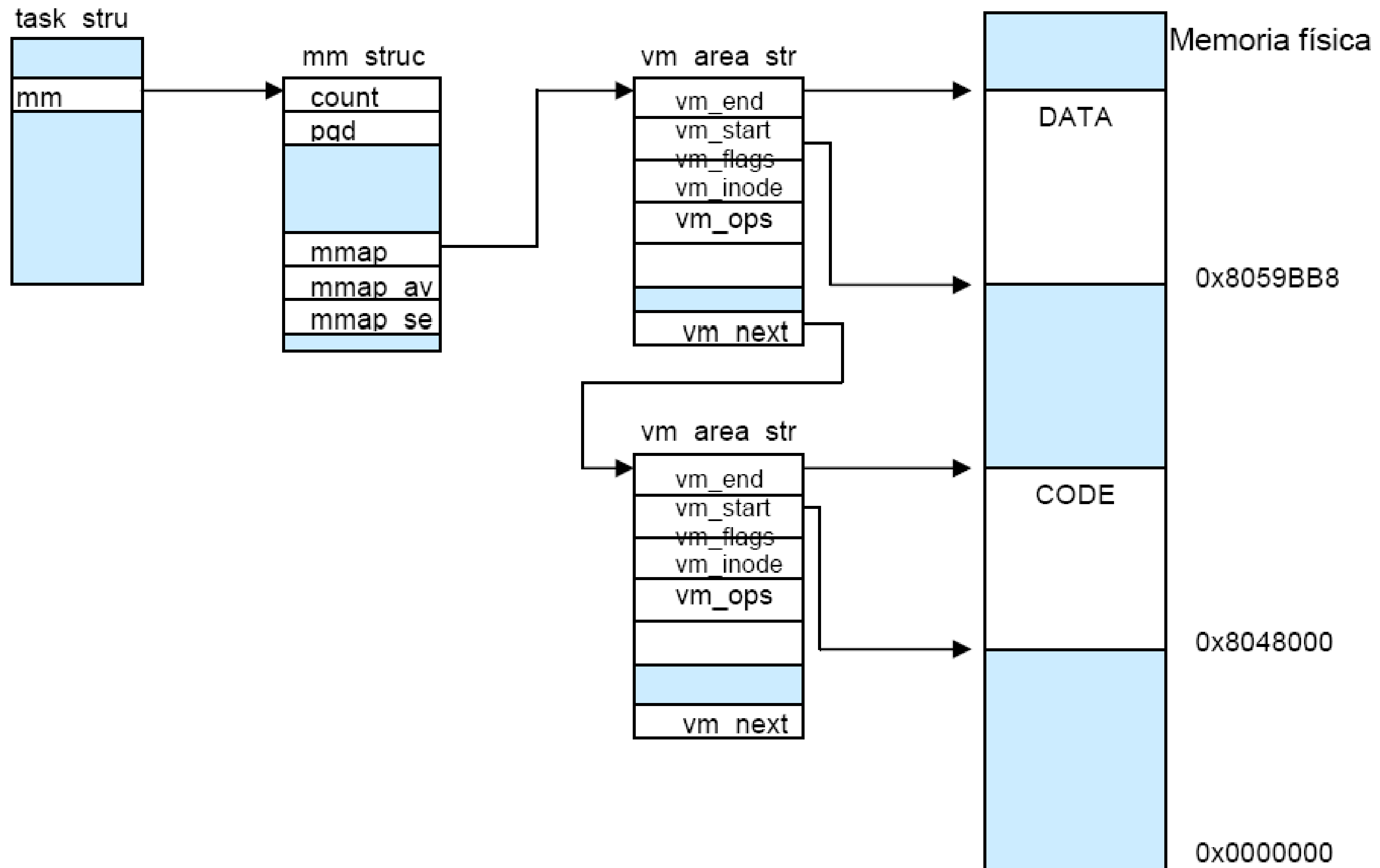
Recordatorio: Regiones de memoria

Cada uno de los recursos citados anteriormente (código del proceso, datos…) se alojan en zonas dentro del espacio de direccionamiento del proceso. Estas regiones poseen los siguientes atributos:

- Direcciones de inicio y fin.
- Derechos de acceso asociados.
- Objeto asociado, p.e. un archivo.

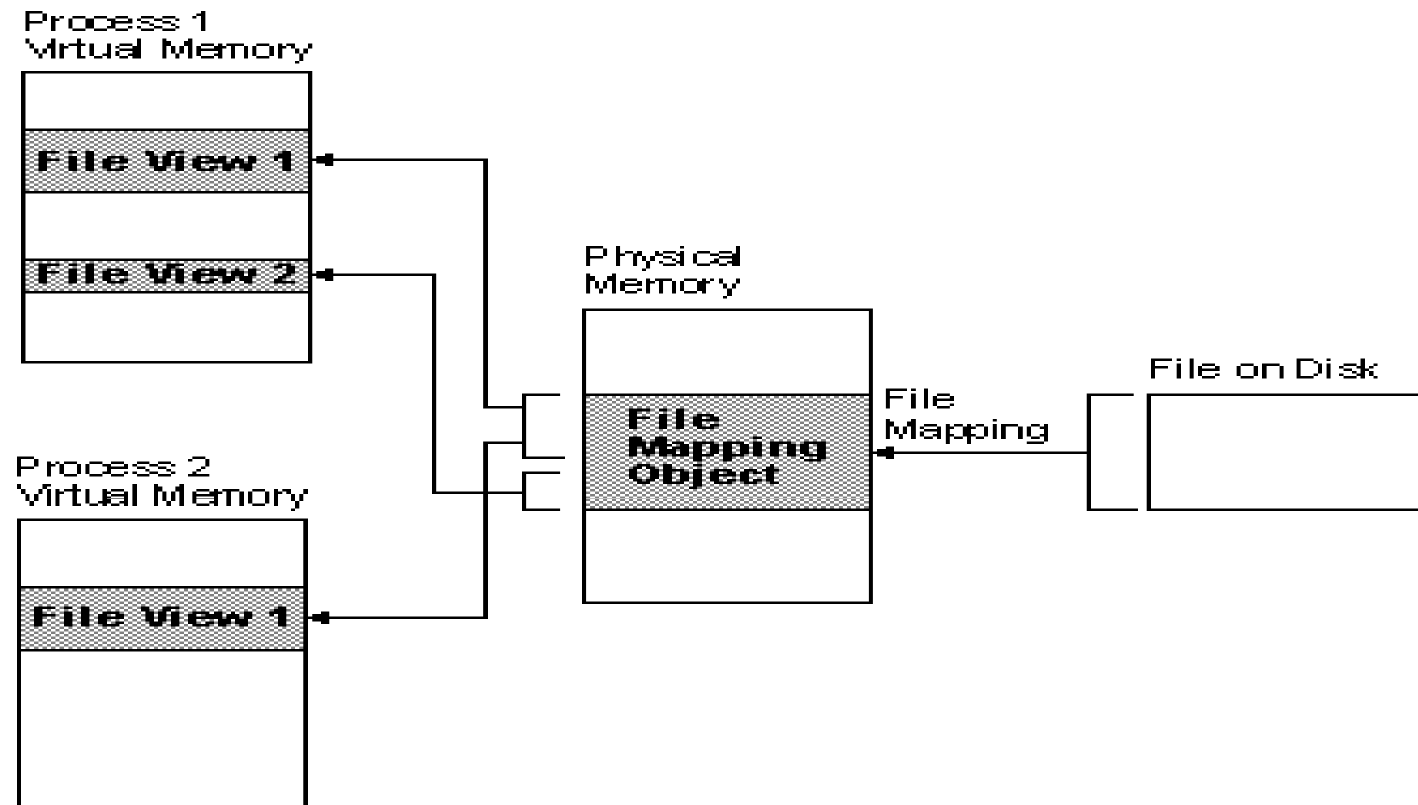
El núcleo mantiene en memoria (en la tabla de procesos) una descripción de las regiones utilizadas por cada proceso, a modo de lista.

Memoria Virtual de un proceso



Proyección de ficheros en memoria (file mapping)

La proyección de ficheros, también conocida como *file mapping*, es una herramienta con la cual se pueden realizar modificaciones de ficheros a través de una proyección de su contenido en memoria.



File mapping

File mapping es una asociación del contenido de un archivo con un trozo del espacio de direcciones virtual de un proceso.

Cuando un proceso proyecta el contenido de un archivo en memoria, se crea una **nueva *región de memoria*** en su *(ED)* y el contenido del archivo se hace accesible en esta región.

Para ello, el sistema operativo crea un objeto ***file-mapping*** con el fin de mantener esta asociación.

File mapping: Tipos de Proyecciones

- **Proyecciones compartidas** (MAP_SHARED):

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.

- **Proyecciones privadas**(MAP_PRIVATE):

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.

- **Proyecciones anónimas**(MAP_ANON):

La proyección en memoria no afecta a ningún archivo (se ignora fd). Se crea una nueva región de memoria cuyo contenido se inicializará a cero. Puede usarse para realizar una implementación particular del malloc.

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, el proceso puede utilizar la llamada del sistema del ***munmap()***.

mmap(I)

```
#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 "|".

mmap(II)

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.

Implementación de la llamada al sistema

Como se ha comentado existen dos estructuras relacionadas con la memoria y los procesos que son esenciales para la implementación del *mmap():*

vm_struct

mm_struct

vm_struct (I)

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

vm_start: Dirección de inicio.

vm_end: Dirección de fin.

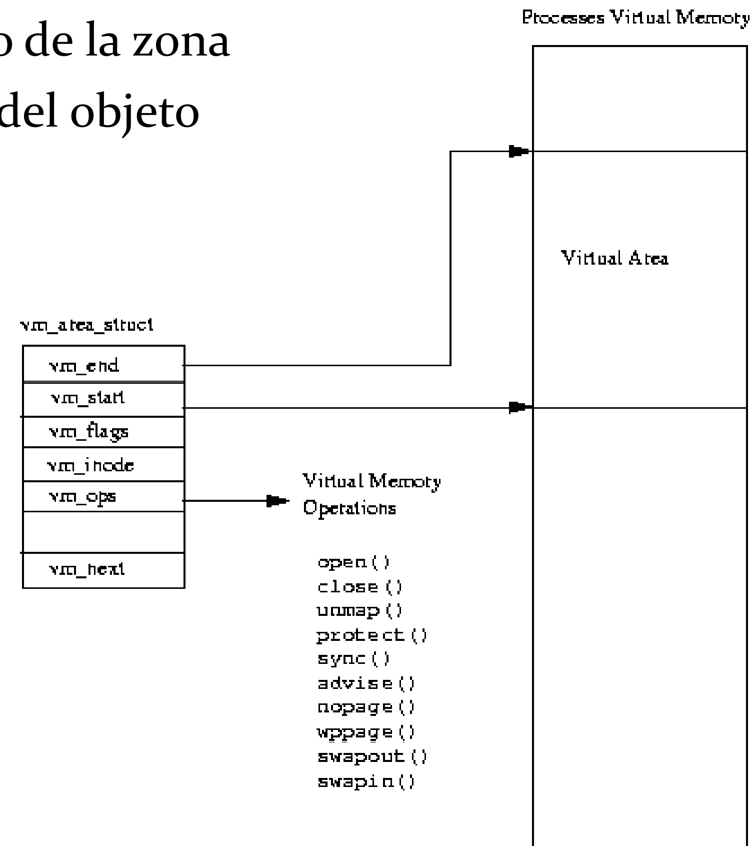
vm_page_prot: Protección asociada.

vm_flags: Estado:

- VM_READ
- VM_WRITE
- VM_EXEC
- VM_SHARED
- VM_MAYREAD
- VM_MAYWRITE
- VM_MAYEXEC
- VM_MAYSHARED
- VM_DENYWRITE
- VM_LOCKED

vm_struct (II)

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

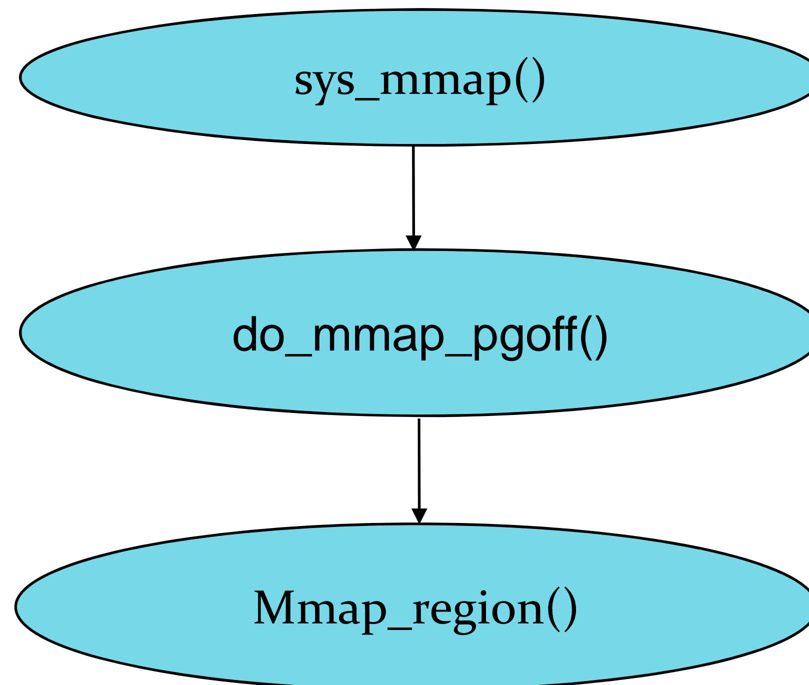


mmap()

mm_struct

```
struct mm_struct {
   struct vm_area_struct *mmap, *mmap_avl, *mmap_cache;
   pgd_t * pdg;
   atomic_t count;
   int map_count;
   struct semaphore mmap_sem;
   unsigned long context;
   unsigned long start_code, end_code, start_data,
   end_data;
   unsigned long start_brk, brk, start_stack;
   unsigned long arg_start, arg_end, env_start, env_end;
   unsigned long rss, total_vm, locked_vm;
   unsigned long def_flags;
   unsigned long cpu_vm_mask;
   unsigned long swap_cnt;
   unsigned long sawp_address;
   void * segments;
};
```

mmap()



1. Comprobar la validez de los parámetros.

2. Buscar un rango de memoria no mapeado.

3. Verificar compatibilidades archivo – proyección.

4. Llamar a la función mmap_region

do_mmap_pgoff (I)

Comenzamos…

```
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;
```

do_mmap_pgoff (II)

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;
```

do_mmap_pgoff (III)

Comprueba que el tamaño del fichero no sea mayor que la memoria disponible para evitar desbordamientos

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

Comprueba que no se hayan hecho demasiados mapeos sobre la memoria virtual

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

do_mmap_pgoff (IV)

2 -.Busca un rango de memoria no mapeado

Obtiene la dirección a la que tiene que mapear. La verifica 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;
```

do_mmap_pgoff (V)

3.-Verifica que la proyección es compatible con el fichero a proyectar. Es decir, si el fichero no tiene permisos de escritura no podrá estar el flag PROT_WRITE activo. Este es para el caso de mapeo compartido.

```
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;
```

do_mmap_pgoff (VI)

Comprobamos que no permitimos sobreescritura 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 tenga cerrojo de modificación (mandatory locks) que es cuando el bit setgid está activado.

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

do_mmap_pgoff (VII)

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);
```

do_mmap_pgoff (VIII)

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;
```

do_mmap_pgoff (IX)

4-. Llamada a **mmap_region** para obtener un VMA, inicializar su descriptor, comprobar si se puede escribir en la región, actualizar las estructuras y devolver la dirección de comienzo del VMA.

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

1. Obtener una VMA para el mapeo y eliminar mapeos anteriores.
2. Continuar inicializando el descriptor de VMA.
3. Llenamos los campos de la estructura vma y enlazamos el mismo en la lista de vma's.
4. Comprobar si se puede escribir en la región.
5. Finalizar la proyección actualizando las estructuras del sistema y del proceso y devolver la dirección de comienzo del VMA.

Mmap_region(I)

```
unsigned 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
```

Mmap_region (II)

1.-Obtenemos una VMA para el mapeo y eliminamos mapeos anteriores

```
munmap_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;
```

Mmap_Region (III)

Verificamos el limite del espacio de direccionamiento

```
1091 if (!may_expand_vm(mm, len >> PAGE_SHIFT))
1092 return -ENOMEM;
```

Mmap_Region (IV)

En caso de mapeos privados, si está activado el flag VM_WRITE comprobamos que exista disponibilidad de memoria

```
1046 } else if (vm_flags & VM_WRITE) {
1050 charged = len >> PAGE_SHIFT;
1051 if (security_vm_enough_memory(charged))
1052 return -ENOMEM;
1053 vm_flags |= VM_ACCOUNT;
1054 }
1055 }
```

Mmap_region (V)

2-.Inicializamos el descriptor de VMA.

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

```
1125 vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);

1126 if (!vma) {

 1127 error = -ENOMEM;

1128 goto unacct_error;}
```

Mmap_region (VI)

3-.Llenamos 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;
```

Enlaza el VMA en la lista de VMAs del proceso

```
1182 vma_link(mm, vma, prev, rb_link, rb_parent);
1183 if (correct_wcount)
1184 atomic_inc(&inode->i_writecount);
```

Mmap_region (VII)

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 }
```

Mmap_region (VIII)

1148 vma->vm_file = file; //asociamos el archivo a proyectar en la VMA actual

1149 get_file(file);

1150 error = file->f_op->mmap(file, vma); // tratamos de realizar un mmap con “file”

1151 if (error)

1152 goto unmap_and_free_vma; //debemos liberar la VMA actual y encontrar una nueva

1153 } else if (vm_flags & VM_SHARED) {

1154 error = shmem_zero_setup(vma);

1155 if (error)

1156 goto free_vma;

1157 }

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);
 1211 /* Deshacer mapeos parciales realizados */
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}
```

Mmap_region (X)

5-.Finalizamos la proyección actualizando las estructuras del sistema y 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;
```

int *munmap (void *start, size_t length)

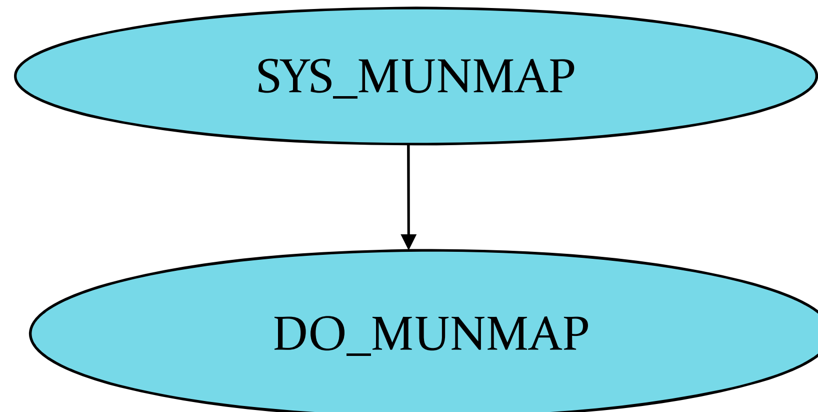
start → Dirección de inicio de la región de memoria correspondiente a la proyección que se desea suprimir.

length → Nº de bytes de la región.

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

Munmap (II)

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:

```
1897 asmlinkage long sys_munmap(unsigned long addr, size_t len) 98 {
1898 int ret;
1899 struct mm_struct *mm = current->mm;
1900 profile_munmap(addr);
```

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

```
1901 down_write(&mm->mmap_sem);
1902 ret = do_munmap(mm, addr, len);
1903 up_write(&mm->mmap_sem);
1904 return ret;
1905 }
```

do_munmap (I)

La función do_munmap empieza como sigue:

```
1838 int do_munmap(struct mm_struct *mm, unsigned long start,
    size_t len)
1839 {
1940 unsigned long end;
1841 struct vm_area_struct *vma, *prev, *last;
```

Donde los parámetros que recibe son:
mm-> puntero al mm_struct del proceso
Start-> posición de memoria desde donde empieza el desmapeo
len-> tamaño del fichero a desmapear

do_munmap(II)

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

```
1843 if ((start & ~PAGE_MASK) || start > TASK_SIZE || len > TASK_SIZE-
start)
1844 return -EINVAL;
```

Si hay paginas no liberadas devuelve un error.

```
1845 if ((len = PAGE_ALIGN(len)) == 0)
1846 return -EINVAL;
```

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

```
1850 vma = find_vma_prev(mm, start, &prev);
1851 if (!vma)
1852 return 0;
```

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.

```
1856 end = start + len;
1857 if (vma->vm_start >= end)
1858 return 0;
```

do_munmap(V)

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.

```
1867 if (start > vma->vm_start) {
1868 int error = split_vma(mm, vma, start, 0);
1869 if (error)
1870 return error;
1871 prev = vma;
1872 }
```

do_munmap (VI)

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.

```
1875 last = find_vma(mm, end);
1876 if (last && end > last->vm_start) {
1877 int error = split_vma(mm, last, end, 1);
1878 if (error)
1879 return error;
1880 }
1881 vma = prev? prev->vm_next: mm->mmap;
```

Remueve las VMA’s del árbol RojoNegro y hace un unmap de las paginas actuales.

```
1886 detach_vmas_to_be_unmapped(mm, vma, prev, end);
1887 unmap_region(mm, vma, prev, start, end);
```

Libera la lista de vma’s.

```
1890 remove_vma_list(mm, vma);
1891 return 0;
1892 }
```

find_vma

Devuelve el puntero al VMA que contiene el final del espacio que se quiere desmapear, o sea, la dirección start+len.

Parámetros:

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

Es muy similar a la función anterior pero devuelve el puntero al VMA anterior del VMA que contiene la dirección de comienzo de la zona a desmapear.

Parámetros:

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

find_vma_prev (II)

En versiones anteriores la búsqueda de VMA se implementaba mediante arboles AVL, en la actualidad son arboles RojoNegro.

```
1470 struct vm_area_struct *
1471 find_vma_prev(struct mm_struct *mm, unsigned long addr,
1472 struct vm_area_struct **pprev)
1473 {
1474 struct vm_area_struct *vma = NULL, *prev = NULL;
1475 struct rb_node * rb_node;
1476 if (!mm)
1477 goto out;
1480 vma = mm->mmap;
```

Recorre el árbol RojoNegro en busca del nodo que corresponde a la dirección de memoria pasada.

```
1483 rb_node = mm->mm_rb.rb_node;
1485 while (rb_node) {
1486 struct vm_area_struct *vma_tmp;
1487 vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
```

find_vma_prev (IV)

```
1489 if (addr < vma_tmp->vm_end) {
1490 rb_node = rb_node->rb_left;
1491 } else {
1492 prev = vma_tmp;
1493 if (!prev->vm_next || (addr < prev->vm_next->vm_end))
1494 break;
1495 rb_node = rb_node->rb_right;
1496 }
1497 }
1498
1499 out:
1500 *pprev = prev;
1501 return prev ? prev->vm_next : vma;
1502 }
```

detach_vmas_to_be_unmapped (I)

Desenlaza las VMA’s, con su árbol RojoNegro, para que puedan ser desmapeadas por otras funciones.

Parámetros:

- mm: descriptor de espacio de direccionamiento.
- vma: la vma que se desea desmapear.
- prev: puntero a la vma anterior.
- end: dirección de fin de la región que queremos liberar

detach_vmas_to_be_unmapped (II)

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

```
1757 static void
1758 detach_vmas_to_be_unmapped(struct mm_struct *mm, struct vm_area_struct *vma,
1759 struct vm_area_struct *prev, unsigned long end)
1760 {
1761 struct vm_area_struct **insertion_point;
1762 struct vm_area_struct *tail_vma = NULL;
1763 unsigned long addr;
1764 insertion_point = (prev ? &prev->vm_next : &mm->mmap);
```

Recorre la lista eliminando cada nodo del árbol RojoNegro.

```
1765 do {
1766 rb_erase(&vma->vm_rb, &mm->mm_rb);
1767 mm->map_count--;
1768 tail_vma = vma;
1769 vma = vma->vm_next;
1770 } while (vma && vma->vm_start < end);
1771 *insertion_point = vma;
1772 tail_vma->vm_next = NULL;
```

detach_vmas_to_be_unmapped (IV)

Destruye la cache

```
1773if (mm->unmap_area == arch_unmap_area)
1774 addr = prev ? prev->vm_end : mm->mmap_base;
1775 else
1776 addr = vma ? vma->vm_start : mm->mmap_base;
1777 mm->unmap_area(mm, addr);
1778 mm->mmap_cache = NULL; /* Kill the cache. */
1779 }
1780
```

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.

unmap_region (II)

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. Hace uso de unmap_vmas.

```
1734 static void unmap_region(struct mm_struct *mm,
1735 struct vm_area_struct *vma, struct vm_area_struct *prev,
1736 unsigned long start, unsigned long end)
1737 {
1738 struct vm_area_struct *next = prev? prev->vm_next: mm->mmap;
1739 struct mmu_gather *tlb;
.
1740 unmap_vmas(&tlb, vma, start, end, &nr_accounted, NULL);
.
1742 free_pgtables(&tlb, vma, prev? prev->vm_end: FIRST_USER_ADDRESS,
1743 next? next->vm_start: 0);
1744 tlb_finish_mmu(tlb, start, end);
1745 }
```

remove_vma_list (I)

Desmapea la lista que contiene las vma's.

Parámetros:

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

remove_vma_list (II)

```
1716 static void remove_vma_list(struct mm_struct *mm, struct vm_area_struct
    *vma)
1717 {
1718 /* Update high watermark before we lower total_vm */
1719 update_hiwater_vm(mm);
1720 do {
1721 long nrpages = vma_pages(vma);
1722
1723 mm->total_vm -= nrpages;
1724 if (vma->vm_flags & VM_LOCKED)
1725 mm->locked_vm -= nrpages;
1726 vm_stat_account(mm, vma->vm_flags, vma->vm_file, -nrpages);
1727 vma = remove_vma(vma);
1728 } while (vma);
1729 validate_mm(mm);
1739 }
```

unmap_vmas (I)

Desmapea toda una region de vma's:

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)

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.

unmap_vmas (II)

- 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 de 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.

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.

msync (II)

```
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;
```

msync (III)

```
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             }
```

msync (IV)

En este fragmento de código es donde se escribe en disco, podemos ver cómo 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);
```

msync (V)

```
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 }
```

Lo más importante de mmap()

1. Comprueba la validez de los parámetros.

2. Busca un rango de memoria no mapeado.

3. Obtenemos una VMA para el mapeo.

4. Enlaza el VMA en la lista de VMAs del proceso.

5. Finalizamos la proyección actualizando las estructuras del sistema y del proceso y devolvemos la dirección de comienzo del VMA.

Lo más importante de munmap()

La lógica del munmap está bien distribuida en funciones, así que lo más importante son las llamadas a otras funciones como:

```
1799 vma = find_vma_prev(mm, start, &prev);

1817 int error = split_vma(mm, vma, start, 0);

1824 last = find_vma(mm, end);

1835 detach_vmas_to_be_unmapped(mm, vma, prev, end);

1836 unmap_region(mm, vma, prev, start, end);

1839 remove_vma_list(mm, vma);
```

FIN

¿PREGUNTAS?