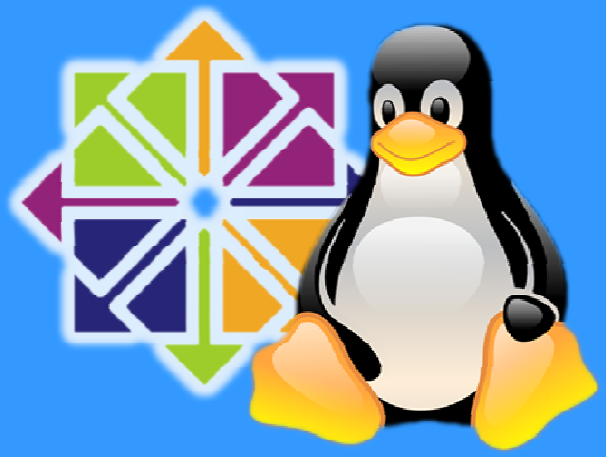
Memoria de un proceso
VMA's

Enrique Ismael Mendoza Robaina
Javier Antonio Monzón Santana



Índice de Contenidos

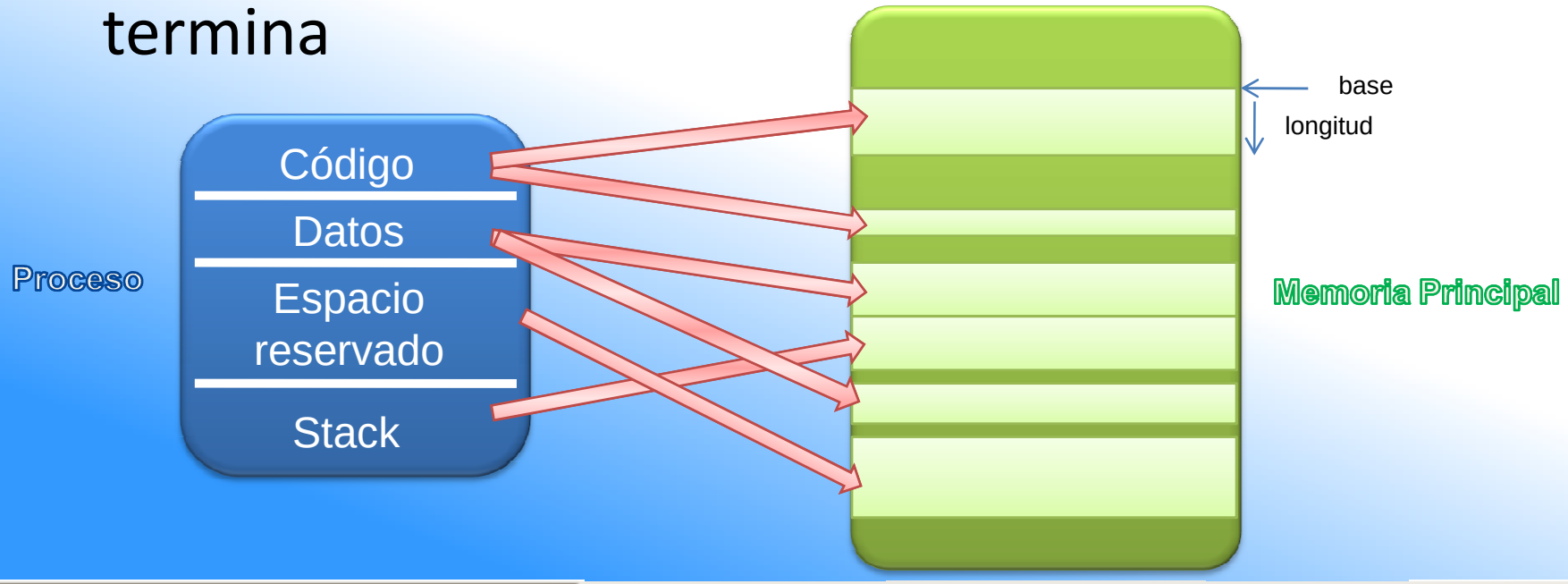
- Introducción
- ¿Cómo lo gestiona Linux?
- Estructuras de datos
  - mm_Struct
  - vma_area_struct
  - vma_operations_struct
- Funciones
  - vm_enough_memory
  - remove_vm_struct
  - find_vma
  - insert_vma_struct

Introducción

- Cuando se crea un proceso, este necesitará estar alojado en memoria para su ejecución
- Los procesos se estructuran en 4 grandes bloques para su inserción en memoria
  - Stack
  - Código
  - Datos
  - Espacio reservado memoria dinámica

Introducción

- Estos bloques a su vez se subdividen para ser alojados en memoria. Estos subdivisiones no necesariamente tienen que ser contiguos es decir están dispersos.
- Cada sección se caracteriza por una base y una longitud que indica donde empieza y donde termina



Introducción

```
[root@dhcppc0 6416]# cat maps
0026b000-0027e000 r-xp 00000000 08:02 1273009    /lib/libpthread-2.5.so
0027e000-0027f000 r-xp 00012000 08:02 1273009    /lib/libpthread-2.5.so
0027f000-00280000 rwxp 00013000 08:02 1273009    /lib/libpthread-2.5.so
00280000-00282000 rwxp 00280000 00:00 0
00504000-0051e000 r-xp 00000000 08:02 1272978    /lib/ld-2.5.so
0051e000-0051f000 r-xp 00019000 08:02 1272978    /lib/ld-2.5.so
0051f000-00520000 rwxp 0001a000 08:02 1272978    /lib/ld-2.5.so
005ec000-005ed000 r-xp 005ec000 00:00 0          [vdso]
007d8000-00915000 r-xp 00000000 08:02 1272985    /lib/libc-2.5.so
00915000-00917000 r-xp 0013c000 08:02 1272985    /lib/libc-2.5.so
00917000-00918000 rwxp 0013e000 08:02 1272985    /lib/libc-2.5.so
00918000-0091b000 rwxp 00918000 00:00 0
08048000-08049000 r-xp 00000000 08:02 457183     /root/Practica2/sem
08049000-0804a000 rw-p 00000000 08:02 457183     /root/Practica2/sem
09862000-09883000 rw-p 09862000 00:00 0
b7fd0000-b7fd2000 rw-p b7fd0000 00:00 0
b7fe7000-b7fe8000 rw-p b7fe7000 00:00 0
b7fe8000-b7fe9000 rw-s 00000000 00:13 12363      /dev/shm/sem.sem29
bfe58000-bfe6e000 rw-p bfe58000 00:00 0          [stack]
```

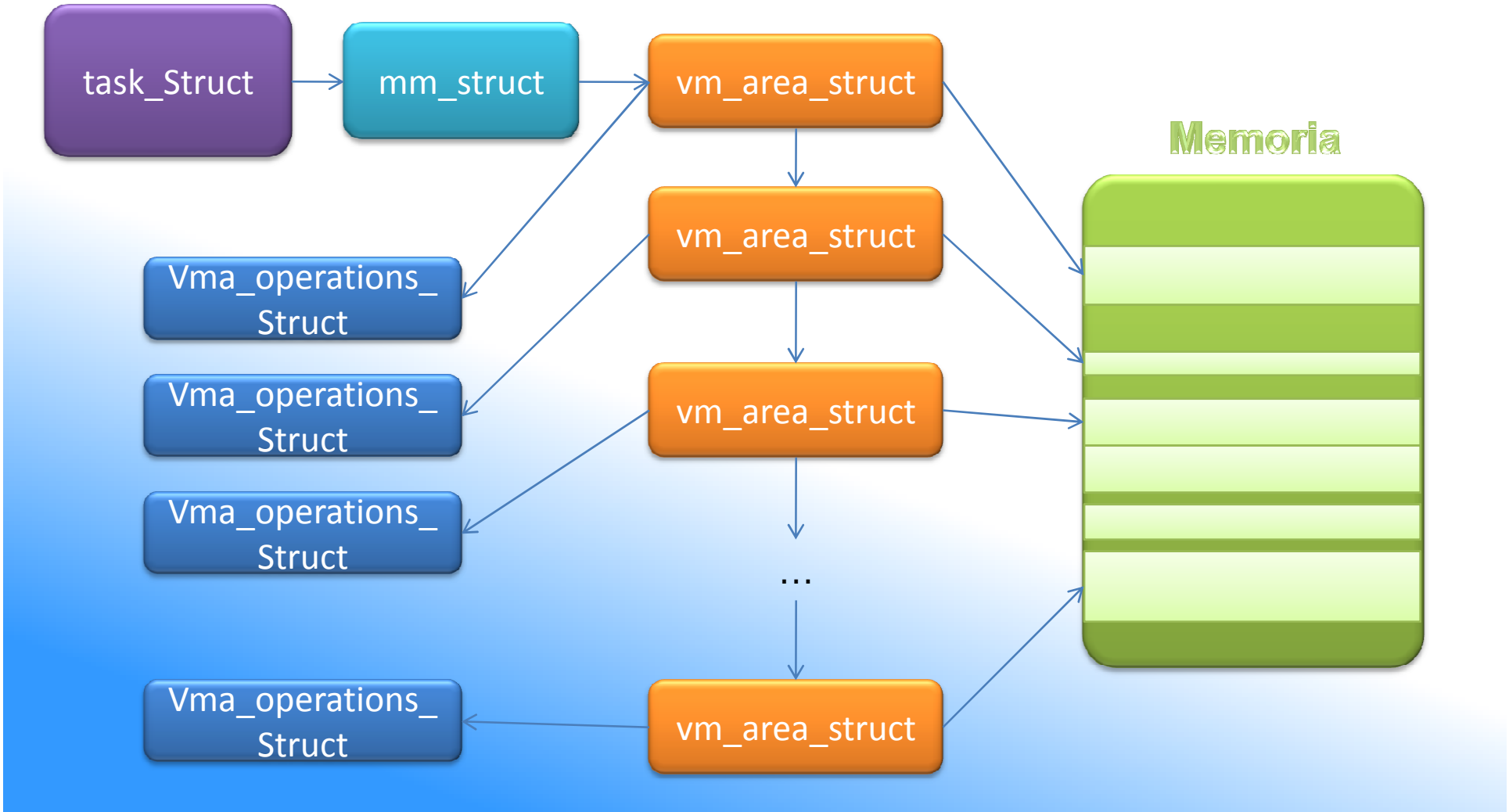
¿Cómo lo gestiona linux?

- Para llevar a cabo lo explicado anteriormente linux utiliza varias estructuras de datos y diversas funciones:

Estructuras
- mm_struct
- vm_area_struct
- vm_operations_struct

Funciones
- vm_enough_memory
- remove_vm_struct
- find_vma
- insert_vm_struct

¿Cómo lo gestiona linux?



Estructuras de datos

Mm_struct

- Características
  - *Fichero: Include/Linux/mm_types.h*
  - Representa el área de la memoria del proceso
  - Contiene un puntero a la estructura vm_area_struct

Mm_struct

Código(I)

```
173struct mm_struct {
174        struct vm_area_struct * mmap;           /* list of VMAs */
175        struct rb_root mm_rb;
176        struct vm_area_struct * mmap_cache;     /* last find_vma result */
177        unsigned long (*get_unmapped_area) (struct file *filp,
178                                unsigned long addr, unsigned long len,
179                                unsigned long pgoff, unsigned long flags);
180        void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
181        unsigned long mmap_base;                /* base of mmap area */
182        unsigned long task_size;                /* size of task vm space */
183        unsigned long cached_hole_size;         /* if non-zero, the largest hole below free_area_cache */
184        unsigned long free_area_cache;          /* first hole of size cached_hole_size or larger */
185        pgd_t * pgd;
186        atomic_t mm_users;                      /* How many users with user space? */
187        atomic_t mm_count;                      /* How many references to "struct mm_struct" (users count as 1) */
188        int map_count;                          /* number of VMAs */
189        struct rw_semaphore mmap_sem;
190        spinlock_t page_table_lock;             /* Protects page tables and some counters */
205
206        unsigned long total_vm, locked_vm, shared_vm, exec_vm;
207        unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
208        unsigned long start_code, end_code, start_data, end_data;
209        unsigned long start_brk, brk, start_stack;
210        unsigned long arg_start, arg_end, env_start, env_end;
211
```

Mm_struct

Código (II)

```
212         unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv */
213
214         cpumask_t cpu_vm_mask; 215 216        /* Architecture-specific MM context */
217         mm_context_t context; 218 219        /* Swap token stuff */
229
230         unsigned long flags; /* Must use atomic bitops to access the bits */
231
232         struct core_state *core_state; /* coredumping support */
233
234         /* aio bits */
235         rwlock_t                ioctx_list_lock;        /* aio lock */
236         struct kioctx           *ioctx_list;
237#ifdef CONFIG_MM_OWNER 238          /*
248         struct task_struct *owner;
249#endif
250
251#ifdef CONFIG_PROC_FS
252         /* store ref to file /proc/<pid>/exe symlink points to */
253         struct file *exe_file;
254         unsigned long num_exe_file_vmas;
255#endif
```

Estructuras de datos

Vm_area_struct

- Características
  - *Fichero: Include/Linux/mm_types.h*
  - Representa las secciones alojadas en memoria principal
  - Contiene un puntero a la estructura vm_area_struct que genera una lista de secciones de memoria
  - Si la lista es muy grande se utiliza una estructura de árbol rojo negro

Vm_area_struct

Código

```
105struct vm_area_struct {
106 struct mm_struct * vm_mm; /* The address space we belong to. */
107 unsigned long vm_start; /* Our start address within vm_mm. */
108 unsigned long vm_end; /* The first byte after our end address
109 within vm_mm. */
111 /* linked list of VM areas per task, sorted by address */
112 struct vm_area_struct *vm_next;
113
114 pgprot_t vm_page_prot; /* Access permissions of this VMA. */
115 unsigned long vm_flags; /* Flags, see mm.h. */
116
117 struct rb_node vm_rb;
125 union {
126 struct {
127 struct list_head list;
128 void *parent; /* aligns with prio_tree_node parent */
129 struct vm_area_struct *head;
130 } vm_set;
133 } shared;
144 /* Function pointers to deal with this struct. */
145 struct vm_operations_struct * vm_ops;
146
147 /* Information about our backing store: */
148 unsigned long vm_pgoff; /* Offset (within vm_file) in PAGE_SIZE
149 units, *not* PAGE_CACHE_SIZE */
150 struct file * vm_file; /* File we map to (can be NULL). */
151 void * vm_private_data; /* was vm_pte (shared mem) */
152 unsigned long vm_truncate_count;/* truncate_count or restart_addr */
```

Algunos Flags

```
79#define VM_READ          0x00000001      /* currently active flags */
80#define VM_WRITE         0x00000002
81#define VM_EXEC          0x00000004
82#define VM_SHARED        0x00000008

85#define VM_MAYREAD       0x00000010      /* r/w/x bits; limits for mprotect() etc */
86#define VM_MAYWRITE      0x00000020
87#define VM_MAYEXEC       0x00000040
88#define VM_MAYSHARE      0x00000080

103#define VM_DONTCOPY      0x00020000      /* Do not copy this vma on fork */
```

Estructuras de datos

Vm_operations_struct

- Características
  - *Fichero: Include/Linux/mm.h*
  - Abstrae de las operaciones que realiza la memoria para su gestión

Vm_operations_struct

Código

```
175struct vm_operations_struct {
176        void (*open)(struct vm_area_struct * area);
           /* Se invoca al crear una región de memoria */

177        void (*close)(struct vm_area_struct * area);
           /* Se invoca al eliminar una región de memoria */

178        int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
           /* Cuando se produce un fallo de página */

182        int (*page_mkwrite)(struct vm_area_struct *vma, struct page *page);
           /* Una region de solo lectura es accedida para escritura */

187        int (*access)(struct vm_area_struct *vma, unsigned long addr,
188        void *buf, int len, int write);
189#ifdef CONFIG_NUMA

197        int (*set_policy)(struct vm_area_struct *vma, struct mempolicy *new);

209        struct mempolicy *(*get_policy)(struct vm_area_struct *vma,
210        unsigned long addr);
211        int (*migrate)(struct vm_area_struct *vma, const nodemask_t *from,
212        const nodemask_t *to, unsigned long flags);
213#endif
214};
```

Funciones

_VM_ENOUGH_MEMORY

- Características
  - Comprueba si un proceso tiene bastante memoria para asignar nuevas direcciones virtuales
    - Hay memoria suficiente: 0
    - No hay memoria: - ENOMEM
  - Tres parámetros de entrada:
    - **mm_struct:** Estructura mm con todas la vma's del proceso y la dirección
    - **pages**: Número de páginas que se desea albergar
    - **cap_sys_admin**: indica privilegios de administrador,
      - » 1 es que tiene privilegios
      - » 0 es que no los tiene

_vm_enough_memory

Código

```
105int __vm_enough_memory(struct mm_struct *mm, long pages, int cap_sys_admin)

114          if (sysctl_overcommit_memory == OVERCOMMIT_ALWAYS)
115                       return 0;
117          if (sysctl_overcommit_memory == OVERCOMMIT_GUESS) {

120                       free = global_page_state(NR_FILE_PAGES);
121                       free += nr_swap_pages;

129                       free += global_page_state(NR_SLAB_RECLAIMABLE);

137                       if (free > pages)
138                                 return 0;

144                       n = nr_free_pages();

149                       if (n <= totalreserve_pages)
150                                 goto error;
151                       else
152                                 n -= totalreserve_pages;

...

187 error:
188          vm_unacct_memory(pages);
190          return -ENOMEM;
191 }
```

Funciones

REMOVE_VM_STRUCT

- Se encarga de borrar una zona de memoria virtual, para ello duerme al proceso y examina si posee memoria compartida, para posteriormente liberar la memoria virtual ocupada.

- Se pasa como parámetro de entrada el área de memoria que se desea eliminar (**vma**)

```
231static struct vm_area_struct *remove_vma(struct vm_area_struct *vma)
232{
233 struct vm_area_struct *next = vma->vm_next;
234
235 might_sleep();
236 if (vma->vm_ops && vma->vm_ops->close)
237 vma->vm_ops->close(vma);
238 if (vma->vm_file) {
239 fput(vma->vm_file);
240 if (vma->vm_flags & VM_EXECUTABLE)
241 removed_exe_file_vma(vma->vm_mm);
242 }
243 mpol_put(vma_policy(vma));
244 kmem_cache_free(vm_area_cachep, vma);
245 return next;
246}
```

Funciones

FIND_VMA

- Características
  - Busca la primera VMA cuya dirección final vm_end es mayor que la dirección de memoria pasada por parámetro. Devuelve NULL en caso de no encontrarla
  - Tres parámetros de entrada:
    - ***mm_struct* mm:** con las vma que posee el proceso
    - ***unsigned long* addr**: dirección para saber si se encuentra alojada en una vma de la estructura anterior

find_vma

Código

```
1472 struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr)
1473{
1474         struct vm_area_struct *vma = NULL;
1475
1476         if (mm) {
1477                         /* Check the cache first. */
1478                         /* (Cache hit rate is typically around 35%.) */
1479                         vma = mm->mmap_cache;
1480                         if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
1481                                         struct rb_node * rb_node;
1482
1483                                         rb_node = mm->mm_rb.rb_node;
1484                                         vma = NULL;
1485
1486                                         while (rb_node) {
1487                                                         struct vm_area_struct * vma_tmp;
1488
1489                                                         vma_tmp = rb_entry(rb_node,
1490                                                         struct vm_area_struct, vm_rb);
1491
1492                                                         if (vma_tmp->vm_end > addr) {
1493                                                                         vma = vma_tmp; 1494 if (vma_tmp->vm_start <= addr)
1495                                                                         break;
1496                                                                         rb_node = rb_node->rb_left;
1497                                                         } else
1498                                                                         rb_node = rb_node->rb_right;
1499                                         }
1500                                         if (vma)
1501                                                         mm->mmap_cache = vma;
1502                         }
1503         }
1504         return vma;
1505}
```

Funciones

INSERT_VM_STRUCT

- Características
  - Inserta un nuevo VMA en la lista encadenada y en el árbol Rojo Negro
  - Dos parámetros de entrada:
    - *mm_struct* **mm:** con las vma que posee el proceso
    - *vm_area_struct* **vma**: vma que queremos insertar

insert_vma_struct

Código

```
int insert_vm_struct(struct mm_struct * mm, struct vm_area_struct * vma)
2138{
2139        struct vm_area_struct * vma, * prev;
2140        struct rb_node ** rb_link, * rb_parent;
2141
2142
2154        if (!vma->vm_file) {
2155                BUG_ON(vma->anon_vma);
2156                vma->vm_pgoff = vma->vm_start >> PAGE_SHIFT;
2157        }
2158        vma = find_vma_prepare(mm,vma->vm_start,&prev,&rb_link,&rb_parent);
2159        if (vma && vma->vm_start < vma->vm_end)
2160                return -ENOMEM;
2161        if ((vma->vm_flags & VM_ACCOUNT) &&
2162        security_vm_enough_memory_mm(mm, vma_pages(vma)))
2163                return -ENOMEM;
2164        vma_link(mm, vma, prev, rb_link, rb_parent);
2165        return 0;
2166}
```

Funciones

- Las funciones se encuentran en el fichero
  - mm/mmap.c
- Existen más operaciones que sirven de apoyo a las ya comentadas:
  - **FIND_VMA_PREPARE**
  - **VMA_LINK**