# LECCIÓN 10:
# VMA’s  MEMORIA DE UN PROCESO

# 10.1 Introducción. Conceptos básicos

En el sistema operativo existe un administrador de memoria que se encarga de gestionar la misma, esta gestión se lleva a cabo a través de diferentes operaciones como, llevar un registro de las secciones de la memoria que estén siendo utilizadas, así como también de aquellas que estén libres, para poder asignar o liberar memoria para los procesos.

Además también puede administrar intercambios entre la memoria principal y el disco, cuando la memoria principal no tenga la capacidad suficiente para albergar todas las peticiones que le puedan hacer los procesos.

A continuación se muestran una serie de conceptos básicos para poder entender mejor la gestión de la memoria por parte de los sistemas operativos.

## Memoria real

La memoria real o principal es en donde son ejecutados los programas y procesos de una computadora y es el espacio real que existe en memoria para que se ejecuten los procesos.

## Memoria virtual

El término memoria virtual se asocia a dos conceptos que normalmente a parecen unidos:

- El uso de almacenamiento secundario para ofrecer al conjunto de las aplicaciones la ilusión de tener más memoria RAM de la que realmente hay en el sistema.

- Ofrecer a las aplicaciones la ilusión de que están solas en el sistema, y que por lo tanto, pueden usar el espacio de direcciones completo.

## Direcciones del Microprocesador

Estas son las que vienen determinadas por los bits del bus de direcciones, en el caso de las arquitecturas x86 el espacio máximo de direccionamiento va desde 0 hasta 4 Gbytes, ya que poseen un bus de direcciones de 32 bits.

## Direcciones Físicas

Son aquellas que referencia alguna posición en la memoria física. La memoria RAM del computador define las Direcciones Físicas, en donde los procesos son ubicados.

## Direcciones lógicas o lineales

Son las direcciones utilizadas por los procesos (comienzan en cero). Sufren una serie de transformaciones, realizadas por el procesador (la MMU), antes de convertirse en direcciones físicas.

## Direcciones virtuales

Se obtienen a partir de direcciones lógicas tras haber aplicado una transformación dependiente de la arquitectura.

Los programas de usuario siempre tratan con direcciones virtuales; nunca ven las direcciones físicas reales.

Linux trabaja con un espacio de direcciones virtuales sobre una memoria virtual de 4GB (máximo espacio de direcciones del micro), para ubicar las tareas y los procesos cuando se ejecutan. Este espacio de direccionamiento virtual único para todo el sistema se divide entre:

**Espacio del núcleo:** En el que se ejecutan los procesos propios del núcleo (drivers de dispositivos, etc.) último Gbyte.

**Espacio de usuario:** Donde se ejecutan los procesos de usuario, de 0 a 3 G bytes, 0xc0000000 en hexadecimal. Cuando una llamada al sistema requiere unos datos que están en el espacio del núcleo, es este último el encargado de copiarlos al espacio de usuario  por ejemplo: datos hacia o desde un dispositivo por bloque (disco).

# Unidad De Manejo De Memoria MMU

Este es un hardware que ubica a los procesos en diferentes direcciones físicas separadas. La unidad de manejo de memoria (MMU) es parte del procesador. Sus funciones son:

- Convertir las direcciones lógicas emitidas por los procesos en direcciones físicas.
- Comprobar que la conversión se puede realizar.
- La dirección lógica podría no tener una dirección física asociada. Por ejemplo, la página correspondiente a una dirección se puede haber trasladado a una zona de almacenamiento secundario temporalmente.
- Comprobar que el proceso que intenta acceder a una cierta dirección de memoria tiene permisos para ello.

La MMU se Inicializa para cada proceso del sistema. Esto permite que cada proceso pueda usar el rango completo de direcciones lógicas (memoria virtual), ya que las conversiones de estas direcciones serán distintas para cada proceso.

En todos los procesos se configura la MMU para que la zona del núcleo solo se pueda acceder en modo privilegiado del procesador.

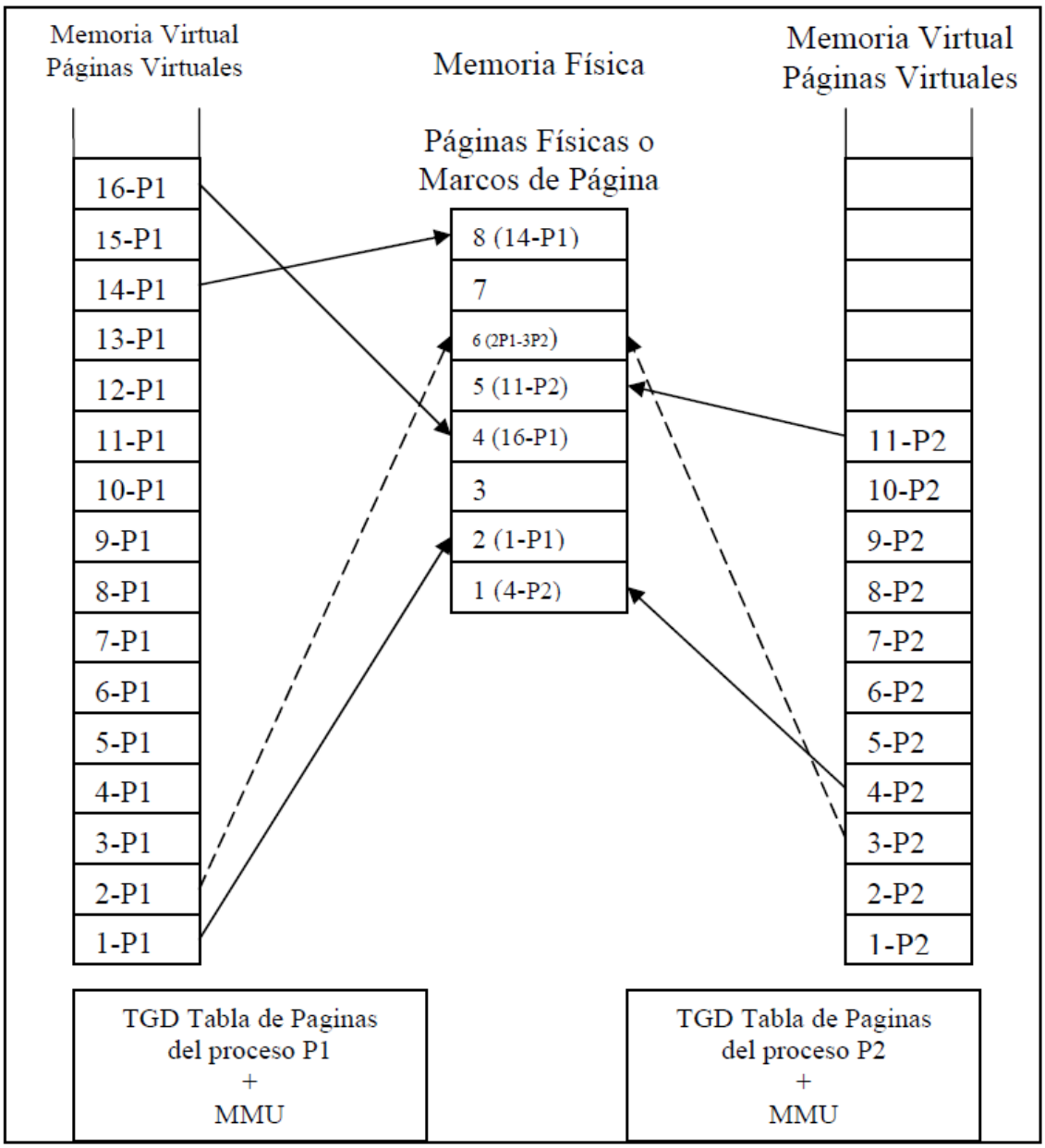
El hardware unidad de manejo de memoria MMU, integrado en la CPU, divide la memoria física en páginas, (**páginas físicas o marcos de página**), en x86 con un tamaño de 4Kbytes.

*Memoria Virtual de un Proceso*

El sistema tiene que dividir las direcciones virtuales en trozos (4k en Linux), llamados páginas, (**páginas lógicas**), para poder ser manejadas por el hardware MMU.

El sistema operativo define la política de que páginas de un proceso van a colocarse en memoria por la MMU.

La MMU también es responsable de enviar una señal al núcleo cuando no se puede traducir una dirección o una página no esta en memoria física (page fault) y de comprobar la protección de memoria.



## 10.2 Representación de la memoria de un proceso

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:

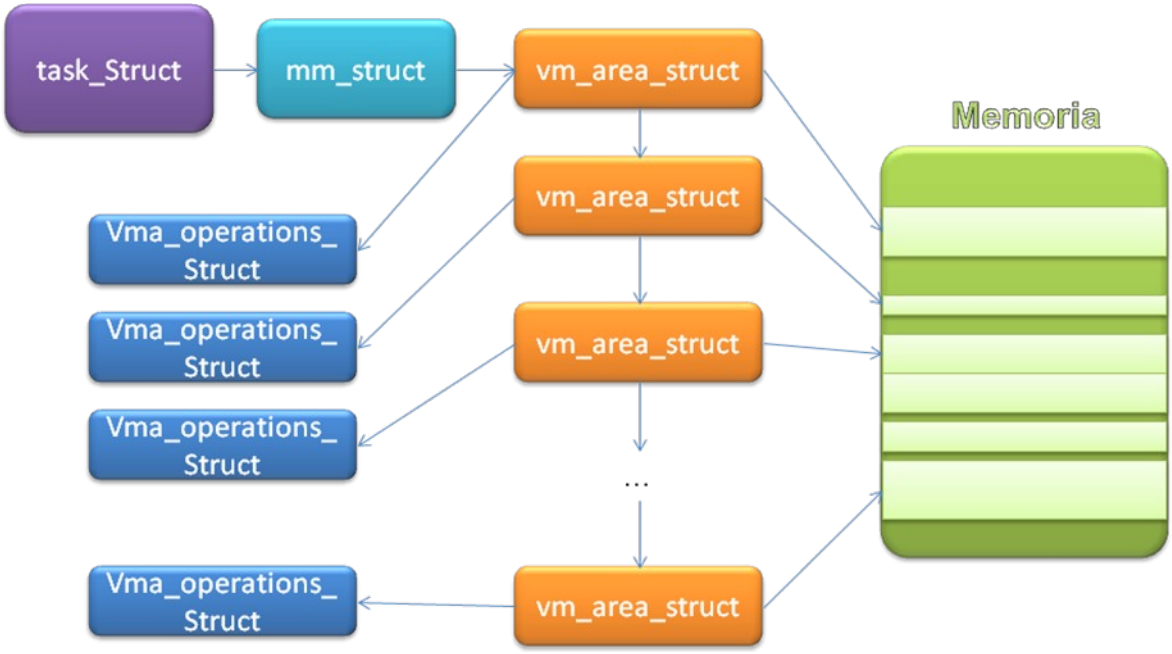
- Stack
- Código
- Datos
- Espacio reservado memoria dinámica

Estos bloques a su vez se subdividen para ser alojados en memoria virtual. Estas 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.

En nuestro caso Linux descompone el espacio de direcciones virtuales en áreas (Virtual Memory Area) utilizables por los procesos. El sistema operativo ubica las direcciones lógicas de un proceso en las direcciones virtuales, de forma que ya no comienzan en cero, a cada proceso se le asigna un trozo distinto o áreas de direcciones virtuales VMAs. A continuación se detallan algunas de las características que poseen las VMA’s:

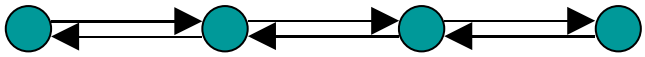
- Una VMA representa un conjunto contiguo de direcciones.
- Dos VMA’s de un mismo proceso nunca pueden solaparse.
- Pueden ser discontinuas (el final de una VMA no tiene por qué ser el principio de la siguiente).
- Dos VMAs de un proceso pueden tener diferentes privilegios de acceso (datos en modo lectura, datos en modo lectura / escritura).
- La definición de una VMA no implica que su homóloga esté cargada en memoria física (fundamento de la paginación y swapping).
- Estas áreas de memoria vma pueden crecer o disminuir en tiempo de ejecución.

*Memoria Virtual de un Proceso*



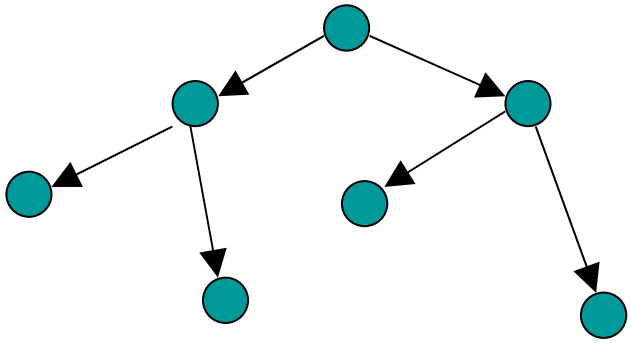

Los descriptores o estructuras VMA's de un proceso se agrupan de dos formas:

- Una lista doblemente encadenada de descriptores ordenados según su dirección final. Mmap es un puntero al primer elemento de la lista. Las búsquedas son de orden(n).



- Un árbol rojo negro llamado RB (Red Black) (Definido en: /include/linux/rbtree.h y /lib/rbtree.c):

Se crea el árbol binario rojo negro (RB) para reducir el tiempo de búsqueda de orden(log2 n). root es un puntero al primer elemento del árbol. La lista encadenada se sigue manteniendo y actualizando.



Linux intenta que su representación de la memoria sea lo más independiente de la arquitectura. Por ello hablamos de áreas y no de segmentos o páginas, que son dependientes de la arquitectura en la que se compila y ejecuta.

Cada área de memoria posee ciertos atributos:

- Dirección de inicio y final
- Privilegios de acceso (rwx, lectura, escritura, ejecución; p indica que el área puede compartirse)
- Objeto asociado (binario, librería, fichero, etc.)
- Desplazamiento del inicio del área en el objeto
- Número de dispositivo que contiene el objeto
- Número del i-nodo del objeto

Las áreas de memoria que definen el espacio de direccionamiento de un proceso pueden visualizarse en el sistema de fichero virtual /proc, en el archivo maps de ese proceso.

El espacio de direccionamiento de un proceso se divide en áreas (Virtual Memory Area's VMA)

- área de código (code)
- área de datos inicializados (data)
- área de datos sin inicializar (bss, heap)
- área de código de las bibliotecas compartidas (libcode)
- área de datos de la biblioteca compartida (libdata, libbss)
- área de pila para ese proceso (stack)
- variables argumento (argsv)
- variables de entorno (env)

A continuación se muestra un ejemplo de las diferentes áreas de memoria.

```
[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]
```

Los dos primeros campos de cada línea representan las direcciones de principio y de fin de la región. El campo siguiente expresa los derechos de acceso asociados. Los campos siguientes indican las informaciones referidas al objeto asociado a la región de memoria: el desplazamiento del inicio de la región en el objeto, el número de dispositivo que contiene el objeto y el número de i-nodo, y por último el objeto almacenado.

En el ejemplo anterior, las tres primeras regiones corresponden al cargador de programas (segmento de código, segmento de datos inicializados y segmento de datos no inicializados). Las tres regiones siguientes corresponden a la biblioteca C compartida (segmento de código, segmento de datos inicializados). La siguiente que comienza en 08048000 corresponde al programa ejecutado (segmento de código, segmento de datos inicializado y segmento de datos no inicializados). Las tres regiones siguientes son del archivo "/usr/lib/locale/locale-archive" que se carga automáticamente al arrancar el programa. Finalmente, la última región corresponde al segmento de pila usado por el proceso.

# 10.3 Estructuras de datos para Memoria Virtual

Tres estructuras de datos son importantes en la representación del uso de la memoria del proceso: struct vm_area_struct, struct vm_operation_struct y struct mm_struct.

## VM_AREA_STRUCT

Esta estructura define las áreas de memoria virtual utilizadas por un proceso.

Dos de los campos más importantes de la vm_area struct son: la vm_start y la vm_end. Las cuales definen el principio y final de las direcciones virtuales que la vma cubre.

Vma_start es la menor dirección dentro de la vma. Por su parte vm_end es un byte mayor que la dirección más alta.

Se debe notar que vm_start y vm_end son de tipo unsigned long, El núcleo utiliza unsigned long cuando representa direcciones.

### Código

```
105struct vm_area_struct {
106         struct mm_struct * vm_mm;       /*Puntero a la estructura vm_mm, contiene todas las vma's asignadas a un proceso*/
107       unsigned long vm_start;     /* nuestra direccion inicial en vm_mm. */
108       unsigned long vm_end;         /* direccion final dentro de vm_mm. */
110
111           /* Lista enlazada de áreas de memoria virtual por tareas, puntero al siguiente nodo de la lista. */
112       struct vm_area_struct *vm_next;
113
114       pgprot_t vm_page_prot;        /* permisos para VMA. */
115       unsigned long vm_flags;       /* flags enumeradas más adelante*/
116
117       struct rb_node vm_rb; /*nodo del árbol rojo negro.*/
118
119       /*
120         * For areas with an address space and backing store,
121         * linkage into the address_space->i_mmap prio tree, or
122         * linkage to the list of like vmas hanging off its node, or
123         * linkage of vma in the address_space->i_mmap_nonlinear list.
```

```
124         */
125        union {
126             struct {
127                  struct list_head list;
128                  void *parent;   /* los une a prio_tree_node*/
129                  struct vm_area_struct *head;
130             } vm_set;
131
132             struct raw_prio_tree_node prio_tree_node;
133        } shared;
134
135        /*
136         * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
137         * list, after a COW of one of the file pages.  A MAP_SHARED vma
138         * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
139         * or brk vma (with NULL file) can only be in an anon_vma list.
140         */
141        struct list_head anon_vma_node; /* Serialized by anon_vma->lock */
142        struct anon_vma *anon_vma;      /* Serialized by page_table_lock */
143
144         /* Puntero a una estructura que almacena las operaciones que se pueden realizar
sobre la VMA */
145        struct vm_operations_struct * vm_ops;
146
147        /* Information about our backing store: */
/* Información sobre el almacenamiento por detrás */
/* Dirección de inicio de la zona de memoria respecto al inicio del objeto
proyectado */
148         unsigned long vm_pgoff;         /* en el tamaño de paginas no en el tamaño de
pagina caché.*/
149
150        struct file * vm_file;          /* archivo a mapear puede ser nulo*/
151        void * vm_private_data;         /* was vm_pte (shared mem) */
152        unsigned long vm_truncate_count;/* truncar contador o reiniciarlo*/
153
154#ifndef CONFIG_MMU /*configuracion MMU*/
155          atomic_t vm_usage;                    /* Al utilizar MMU se ejecutaba de forma
atómica*/
156#endif
157#ifdef CONFIG_NUMA
158        struct mempolicy *vm_policy;    /*politica NUMA con VMA*/
159#endif};
```

## Flags

```
 79#define VM_READ              0x00000001        /*región de memoria es accesible en
Lectura.*/
 80#define VM_WRITE       0x00000002/*la región es accesible en escritura.*/
 81#define VM_EXEC        0x00000004 /*la región es accesible en ejecución.*/
 82#define VM_SHARED      0x00000008          /* la región es compartida entre
varios procesos*/
 83
 84/* mprotect() hardcodes VM_MAYREAD >> 4 == VM_READ, and so for r/w/x bits.
*/
 85#define VM_MAYREAD      0x00000010      /* limits for mprotect() etc */
 86#define VM_MAYWRITE     0x00000020
 87#define VM_MAYEXEC      0x00000040
 88#define VM_MAYSHARE     0x00000080
 89
 90#define VM_GROWSDOWN    0x00000100      /* información general del segmento */
 91#define VM_GROWSUP      0x00000200
 92#define VM_PFNMAP       0x00000400      /* Page-ranges managed without "struct
page", just pure PFN */
 93#define VM_DENYWRITE    0x00000800      /* ETXTBSY on write attempts.. */
 94
 95#define VM_EXECUTABLE   0x00001000
 96#define VM_LOCKED       0x00002000
```

```
 97#define VM_IO           0x00004000      /* Memoria mapeada */
 98
 99                                      /* usado por madvise() */
100#define VM_SEQ_READ     0x00008000      /* App will access data sequentially */
101#define VM_RAND_READ    0x00010000      /* App will not benefit from clustered
reads */
102
103#define VM_DONTCOPY     0x00020000      /* no se copia esta vma en fork*/
104#define VM_DONTEXPAND   0x00040000      /* Cannot expand with mremap() */
105#define VM_RESERVED     0x00080000      /* Count as reserved_vm like IO */
106#define VM_ACCOUNT      0x00100000      /* un objeto VMA*/
107#define VM_NORESERVE    0x00200000      /* should the VM suppress accounting */
108#define VM_HUGETLB      0x00400000      /* Huge TLB Page VM */
109#define VM_NONLINEAR    0x00800000      /* Is non-linear (remap_file_pages) */
110#define VM_MAPPED_COPY  0x01000000      /* T if mapped copy of data (nommu
mmap) */
111#define VM_INSERTPAGE   0x02000000      /* se utiliza en la VMA */
112#define VM_ALWAYSDUMP   0x04000000      /* siempre se incluye en el nucleo*/
113
114#define VM_CAN_NONLINEAR 0x08000000     /* Has ->fault & does nonlinear pages
*/
115#define VM_MIXEDMAP     0x10000000      /* Can contain "struct page" and pure
PFN pages */
116#define VM_SAO          0x20000000      /* Strong Access Ordering (powerpc) */
```

## VM_OPERATIONS_STRUCT

Una vma puede contener una región con las direcciones del segmento de código o del segmento de datos, pero también puede contener otro tipo de objeto como un fichero, una memoria compartida para intercambiar datos, o algún objeto especial. Esto se conoce como mapear un objeto en memoria y se realiza con la llamada al sistema mmap.

Esta variedad de objetos que se pueden mapear en memoria obliga a conocer por parte del sistema todas la funciones que se pueden realizar sobre ellos (abrir o cerrar un fichero, operaciones obre la memoria compartida) es muy costoso para el sistema conocer todas la funciones especiales de cada uno de los objetos. Para quitar esta dificultad, se define una estructura del tipo struct vm_operations_struct que abstrae las operaciones disponibles como open, close, etc., ya que solo mantiene punteros a funciones que serán rellenados con las funciones de cada objeto particular.

Una estructura vm_operations_struct es un sistema de punteros de las funciones, algunos de los cuales podrían ser nulos para indicar que la operación no esta habilitada para el tipo de objeto mapeado. Por ejemplo, porque no tiene sentido sincronizar paginas de objetos de memoria compartida con el disco cuando la memoria compartida no esta mapeada. Los miembros sincronizados podrían ser nulos para la estructura vm_operations struct que representan las operaciones de memoria compartida.

Todo esto significa que si una Vma es mapeada a un objeto , vm_op contiene punteros a las operaciones que el objeto proporciona. Para cada tipo de objeto que puede ser mapeado por VMA, hay una estructura vm_operations_struc estática.

En resumen:

Una VMA puede representar tanto un área de memoria física devuelta por la función malloc, como a un fichero, una memoria compartida, una partición de swapping o cualquier objeto utilizado en la llamada al sistema “mmap”.

Al igual que los manejadores de dispositivos por bloque, se definen un conjunto de operaciones abstractas (punteros a funciones, open, close) para cada una de las vmas que representan a un objeto y que luego se asignarán a esta estructura. Si para un objeto la operación no existe la función apuntará a NULL.

## Código

```
175struct vm_operations_struct {
/* La operación open se llama cuando se crea una nueva región de memoria, referenciada por
area. */
176     void (*open)(struct vm_area_struct * area);
/* La función close se llama cuando se suprime un área de memoria referenciada por area. */
177     void (*close)(struct vm_area_struct * area);
/* La función nopage se llama por el manejador de fallos de pagina para cargar una página
virtual en el área de memoria referenciada por area, cuando una página que no está presente
en memoria física es accedida. El parámetro address indica la dirección en la que la página
debe cargarse, y type el tipo de vma. */
178     int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
179
180     /* notification that a previously read-only page is about to become
181      * writable, if an error is returned it will cause a SIGBUS */
            /* Es invocada por la llamada al sistema remap_pages() */
182     int (*page_mkwrite)(struct vm_area_struct *vma, struct page *page);
183
184     /* called by access_process_vm when get_user_pages() fails, typically
185      * for use by special VMAs that can switch between memory and hardware
186      */
187     int (*access)(struct vm_area_struct *vma, unsigned long addr,
188                   void *buf, int len, int write);
189#ifdef CONFIG_NUMA
190     /*
191      * set_policy() op must add a reference to any non-NULL @new mempolicy
192      * to hold the policy upon return.  Caller should pass NULL @new to
193      * remove a policy and fall back to surrounding context--i.e. do not
194      * install a MPOL_DEFAULT policy, nor the task or system default
195      * mempolicy.
196      */
197     int (*set_policy)(struct vm_area_struct *vma, struct mempolicy *new);
198
199     /*
200      * get_policy() op must add reference [mpol_get()] to any policy at
201      * (vma,addr) marked as MPOL_SHARED.  The shared policy infrastructure
202      * in mm/mempolicy.c will do this automatically.
203      * get_policy() must NOT add a ref if the policy at (vma,addr) is not
 204          * marked as MPOL_SHARED. vma policies are protected by the mmap_sem.
205      * If no [shared/vma] mempolicy exists at the addr, get_policy() op
206      * must return NULL--i.e., do not "fallback" to task or system default
207      * policy.
208      */
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};
```

## MM_STRUCT

Todas las vma asignadas a un proceso son accedidas desde "struct mm_struct". Un puntero mm dentro de task_struck apunta a esta estructura. Dos tareas hilo que vienen del mismo proceso comparten esta estructura y por lo tanto están manejando las mismas áreas de la memoria virtual, siendo esta una característica de procesos hilos creados con la llamada clone.

Cada proceso tiene su propio "descriptor de espacio de direccionamiento" "mm_struct", (excepto los creados con CLONE_VM) y su tabla global de páginas PGD.

### Código

```
173struct mm_struct {
174       struct vm_area_struct * mmap;         /* lista gr VMAs */
175       struct rb_root mm_rb; /*raíz del árbol rojo-negro de vma's*/
176       struct vm_area_struct * mmap_cache;     /* última región VMA visitada */
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
*/
191
192        struct list_head mmlist;              /* List of maybe swapped mm's.  These are
globally strung
193                                            * together off init_mm.mmlist, and are protected
194                                            * by mmlist_lock
195                                            */
196
197       /* Special counters, in some configurations protected by the
198        * page_table_lock, in other configurations by being atomic.
199        */
200       mm_counter_t _file_rss;
201       mm_counter_t _anon_rss;
202
203       unsigned long hiwater_rss;      /* High-watermark of RSS usage */
204       unsigned long hiwater_vm;       /* High-water virtual memory usage */
205
/* La dirección inicial y final del código y de los datos del proceso. */
206       unsigned long total_vm, locked_vm, shared_vm, exec_vm;
/* La dirección inicial y final de los bloques de memoria asignados al heap y la
dirección de
comienzo de la pila.*/
207       unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
208       unsigned long start_code, end_code, start_data, end_data;
/* La direcciones inicial y final del bloque de memoria que contienelos argumentos
del programa ejecutado y del bloque de memoria que contiene las variables de
entorno. */
209       unsigned long start_brk, brk, start_stack;
```

```
/* El número de páginas residentes en memoria para el proceso, el número total de
páginas contenidas en el espacio de direccionamiento, el número de páginas
bloqueadas en memoria y el número de páginas compartidas. */
 210        unsigned long arg_start, arg_end, env_start, env_end;
 211
 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 */
 220        /*
 221         * Last value of global fault stamp as seen by this process.
 222         * In other words, this value gives an indication of how long
 223         * it has been since this task got the token.
 224         * Look at mm/thrash.c
 225         */
 226        unsigned int faultstamp;
 227        unsigned int token_priority;
 228        unsigned int last_interval;
 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        /*
 239         * "owner" points to a task that is regarded as the canonical
 240         * user/owner of this mm. All of the following must be true in
 241         * order for it to be changed:
 242         *
 243         * current == mm->owner
 244         * current->mm != mm
 245         * new_owner->mm == mm
 246         * new_owner->alloc_lock is held
 247         */
 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
 256#ifdef CONFIG_MMU_NOTIFIER
 257        struct mmu_notifier_mm *mmu_notifier_mm;
 258#endif
 259};
 260
 261#endif /* _LINUX_MM_TYPES_H */
```

# 10.4 Funciones que utilizan Vmas

A continuación se describen algunas de las funciones que manejan las áreas de memoria virtual, estas funciones se encuentran en /mm/mmap.c:

## __VM_ENOUGH_MEMORY

Comprueba si hay bastante memoria virtual libre para asignar nuevas direcciones virtuales a un proceso. Cero significa que existe memoria suficiente para su asignación, y ENOMEM implican que no la hay.

Posee tres parámetros de entrada, en el primero se le pasa como parámetro la estructura mm con todas la vma's del proceso y la dirección, en el segundo se le pasa el número de páginas que se desea albergar (pages), con el tercero (cap_sys_admin), se puede observar si el proceso tiene privilegios de administrador, si se le pasa un 1 es que tiene privilegios y si se le pasa un 0, es que no los tiene.

### Código

```
105int __vm_enough_memory(struct mm_struct *mm, long pages, int cap_sys_admin)
106{
107     unsigned long free, allowed;
// Se va a reservar espacio para el número de páginas que se ha introducido
108
109     vm_acct_memory(pages);
110
111     /*
112      * Sometimes we want to use more memory than we have
113      */
114     if (sysctl_overcommit_memory == OVERCOMMIT_ALWAYS)
//Si no se desea guardar más memoria de la que tenemos, devolvemos un cero.
115          return 0;
116
//En caso contrario se busca una alternativa
117     if (sysctl_overcommit_memory == OVERCOMMIT_GUESS) {
118          unsigned long n;
119
//Se calcula el número de páginas libres
120          free = global_page_state(NR_FILE_PAGES);
121          free += nr_swap_pages;
122
123          /*
124           * Any slabs which are created with the
125           * SLAB_RECLAIM_ACCOUNT flag claim to have contents
126           * which are reclaimable, under pressure.  The dentry
127           * cache and most inode caches should fall into this
128           */
129          free += global_page_state(NR_SLAB_RECLAIMABLE);
130
131          /*
 132       //Al* número de páginas libres,  se le  resta  un 3% para el  root
133           */
134          if (!cap_sys_admin)
135               free -= free / 32;
136
//Devolvemos un cero si hay más paginas libres de las que se pide.
137          if (free > pages)
138               return 0;
139
140          /*
141           * nr_free_pages() is very expensive on large systems,
142           * only call if we're about to fail.
143           */
144          n = nr_free_pages();
145
146          /*
147           * Leave reserved pages. The pages are not for anonymous pages.
148           */
```

```
//Si el número de páginas libres es menor que el número de páginas que queremos
reservar,  se  salta  a  error.
 149            if (n <= totalreserve_pages)
 150                    goto error;
 151            else
 152                    n -= totalreserve_pages;
 153
 154            /*
 155          * //Al  número de páginas libres,  se le  resta  un 3% para el  root
 156              */
 157            if (!cap_sys_admin)
 158                    n -= n / 32;
 159            free += n;
 160
//Devolvemos un cero si hay más paginas libres de las que se pide.
 161            if (free > pages)
 162                    return 0;
 163
//si no, se va a error
 164            goto error;
 165      }
 166
//Se vuelve a hacer lo mismo que con la variable free
//Se calcula aproximadamente el número de paginas disponibles.
 167      allowed = (totalram_pages - hugetlb_total_pages())
 168            * sysctl_overcommit_ratio / 100;
 169      /*
 170      * //Al  número de páginas disponibles,  se le  resta  un 3% para el  root
 171        */
 172      if (!cap_sys_admin)
 173            allowed -= allowed / 32;
 174      allowed += total_swap_pages;
 175
 176      /* Don't let a single process grow too big:
 177         leave 3% of the size of this process for other processes */
 178      if (mm)
 179            allowed -= mm->total_vm / 32;
 180
 181      /*
 182        * cast `allowed' as a signed long because vm_committed_space
 183        * sometimes has a negative value
 184        */
//Hay que hacer un cast a allowed ya que vm_committed_space a veces tiene un
valor negativo
 185      if (atomic_long_read(&vm_committed_space) < (long)allowed)
 186            return 0;
 187error:
//Al no tener suficiente memoria, hay que liberar el espacio que reservamos al
principio
 188      vm_unacct_memory(pages);
 189
 190      return -ENOMEM;
 191}
```

## 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.

Los parámetros de la función que se le pasan es la vma que se desea borrar que es del tipo,  vm_area_struct.

### Código

```
231static struct vm_area_struct *remove_vma(struct vm_area_struct *vma)
232{
233        struct vm_area_struct *next = vma->vm_next;
234
/* Se duerme el proceso para la eliminación de su memoria virtual */
235        might_sleep();
/* Si la memoria posee una instrucción específica para suprimir el área de memoria
se le
llamará, la cuál se encargará de borrar la memoria virtual */
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); /*  Se libera el fichero.*/
240                if (vma->vm_flags & VM_EXECUTABLE)
241                        removed_exe_file_vma(vma->vm_mm);
242        }
243        mpol_put(vma_policy(vma)); /*  Se quita el vma de la lista y del árbol*/
244        kmem_cache_free(vm_area_cachep, vma);
/* Se libera el espacio en caché ocupado por esta zona de memoria virtual.*/
245        return next;
246}
```

## FIND_VMA

Busca en la estructura árbol rojo negro 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. Los parámetros que se le pasan a la función es una de tipo mm_struct, con las vma que posee el proceso, y una dirección para saber si se encuentra  alojada en una vma de la estructura pasada.

### Código

```
1472struct 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%.) */
                        //Miramos en la cache
1479                vma = mm->mmap_cache;
1480                if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
1481                        struct rb_node * rb_node;//Se crea el árbol rojo negro
1482
1483                        rb_node = mm->mm_rb.rb_node;
1484                        vma = NULL;
1485//Se recorre el árbol rojo negro en busca de una vma apropiada
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)
        //Llegado a este punto, se ha encontrado el área que buscamos
```

```
1495                                    break;
1496                                rb_node = rb_node->rb_left;
                        //Si no, seguimos descendiendo por el árbol
1497                            } else
1498                                rb_node = rb_node->rb_right;
1499                        }
1500                        if (vma) //Si se ha encontrado un área la almacenamos en la cache
1501                            mm->mmap_cache = vma;
1502                    }
1503                }
1504                return vma;
1505}
```

## FIND_VMA_PREV

Esta función es exactamente igual que la anterior, excepto por el hecho de que ésta devuelve además el predecesor del VMA encontrado.

Además de los mismos parámetros que en la find_vma, uno de tipo mm_struct, con las vma que posee el proceso, una dirección para saber si se encuentra alojada en una vma de esta estructura, tambien se dispone aquí de un puntero en el que se enlazará el antecesor de la vma que se buscaba (** pprev).

### Código

```
1510struct vm_area_struct *
1511find_vma_prev(struct mm_struct *mm, unsigned long addr,
1512                    struct vm_area_struct **pprev)
1513{
1514        struct vm_area_struct *vma = NULL, *prev = NULL;
1515        struct rb_node * rb_node;
1516        if (!mm)
1517                goto out;
1518
1519        /* Guard against addr being lower than the first VMA */
1520        vma = mm->mmap;
1521
1522        /* Go through the RB tree quickly. */
1523        rb_node = mm->mm_rb.rb_node;
1524
                //Se recorre el árbol rojo negro en busca de una vma apropiada
1525        while (rb_node) {
1526                struct vm_area_struct *vma_tmp;
1527                vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
1528
1529                if (addr < vma_tmp->vm_end) {
1530                        rb_node = rb_node->rb_left;
1531                } else {
1532                        prev = vma_tmp;
1533                        if (!prev->vm_next || (addr < prev->vm_next->vm_end))
1534                                break; //Llegado a este punto, se ha encontrado el área que
buscamos
1535                        rb_node = rb_node->rb_right;
1536                }
1537        }
1538
1539out:
1540        *pprev = prev;
1541        return prev ? prev->vm_next : vma;
1542}
```

## INSERT_VM_STRUCT

Inserta un nuevo VMA en la lista encadenada y en el árbol Rojo Negro. Se le pasa como parámetros la estructura mm con todas las vmas del proceso y la vma que queremos insertar.

### Código

```
2137int 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
      /* find_vma_prepare prepara el vma para su inclusión en la lista y en el
      árbol,  nos devuelve
      los antecesores */
2142      /*
2143       * The vm_pgoff of a purely anonymous vma should be irrelevant
2144       * until its first write fault, when page's anon_vma and index
2145       * are set.  But now set the vm_pgoff it will almost certainly
2146       * end up with (unless mremap moves it elsewhere before that
2147       * first wfault), so /proc/pid/maps tells a consistent story.
2148       *
2149       * By setting it to reflect the virtual start address of the
2150       * vma, merges and splits can happen in a seamless way, just
2151       * using the existing file pgoff checks and manipulations.
2152       * Similarly in do_mmap_pgoff and in do_brk.
2153       */
2154      if (!vma->vm_file) {
2155            BUG_ON(vma->anon_vma);
      /* __vma_link enlaza el vma a la lista y el árbol */
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}
```

# 10.5 Funciones auxiliares

Existen una series de funciones auxiliares cuyo cometido es el de ser utilizadas por otras para realizar ciertas tareas. En los sistemas operativos en general, se suelen utilizar estas delegaciones de tareas a fin de que el código sea lo más modular posible.

Las funciones que por su interés se han considerado importantes con el sistema de memoria de Linux son las siguientes:

## FIND_VMA_PREPARE

Prepara el vma para su inclusión en la lista y en el árbol y devuelve los vmas anteriores (antecesores). Esta función es utilizada por insert_vm_struct.

### Código

```
353static struct vm_area_struct *
354find_vma_prepare(struct mm_struct *mm, unsigned long addr,
355             struct vm_area_struct **pprev, struct rb_node ***rb_link,
356             struct rb_node ** rb_parent)
357{
358     struct vm_area_struct * vma;
359     struct rb_node ** __rb_link, * __rb_parent, * rb_prev;
360
361     __rb_link = &mm->mm_rb.rb_node;
362     rb_prev = __rb_parent = NULL;
363     vma = NULL;
364
          //Se recorre el árbol rojo negro para prepararlo
365     while (*__rb_link) {
366          struct vm_area_struct *vma_tmp;
367
368          __rb_parent = *__rb_link;
369          vma_tmp = rb_entry(__rb_parent, struct vm_area_struct, vm_rb);
370
371          if (vma_tmp->vm_end > addr) {
372               vma = vma_tmp;
373               if (vma_tmp->vm_start <= addr)
374                    break;
375               __rb_link = &__rb_parent->rb_left;
376          } else {
377               rb_prev = __rb_parent;
378               __rb_link = &__rb_parent->rb_right;
379          }
380     }
381
          //Se recorre la lista y se prepara
382     *pprev = NULL;
383     if (rb_prev)
384          *pprev = rb_entry(rb_prev, struct vm_area_struct, vm_rb);
385     *rb_link = __rb_link;
386     *rb_parent = __rb_parent;
387     return vma;
388}
```

## VMA_LINK

Enlaza el vma a la lista y al árbol. Se usa por insert_vm_struct.

**Código**

```
446static void vma_link(struct mm_struct *mm, struct vm_area_struct *vma,
447                      struct vm_area_struct *prev, struct rb_node **rb_link,
448                      struct rb_node *rb_parent)
449{
450       struct address_space *mapping = NULL;
451
452       if (vma->vm_file)
453             mapping = vma->vm_file->f_mapping;
454
455       if (mapping) {
456             spin_lock(&mapping->i_mmap_lock);
457             vma->vm_truncate_count = mapping->truncate_count;
458       }
459       anon_vma_lock(vma);
460
461       __vma_link(mm, vma, prev, rb_link, rb_parent);
462       __vma_link_file(vma);
463
464       anon_vma_unlock(vma);
465       if (mapping)
466             spin_unlock(&mapping->i_mmap_lock);
467
468       mm->map_count++;
469       validate_mm(mm);
470}
```

## Otras

- **find_extend_vma:** Busca si hay tamaño para expandir la vma y si existe la expande con el expand_stack, que es otra función la cual se encarga de expandir la memoria vma jugando con los vm_start y vm_end.

- **copy_vma:** Copia una vma y la añade a la misma estructura mm.

- **split_vma:** Divide la vma en dos regiones en la dirección especificada, y se inserta la nueva vma en la cabeza o en la cola de la lista de vma's.

- **Detach_vmas_to_be_unmapped:** Cree una lista de los vma tocados por el unmap, quitándolos del mm1's lista del vma.

- **unmap_vma_list**: Podemos tener zonas de memoria que podían estar libre en la lista de libres entonces se informa de ello y se actualiza la lista de vma's.

- **unmap_vma:** Determina las vma's creadas que no tienen una función asignada.

- **vma_merge**: Combina vma's, con la ayuda de is_mergeable_vma que comprueba si la vma se puede combinar con otras, can_vma_merge_after que mira si se puede combinar detrás can_vma_merge_before observa si es posible combinar por delante.

- **vma_adjust:** No es posible modificar el tamaño del área de memoria si esta está en el árbol, sin modificar el árbol, con la ayuda de esta función esto se hace posible.