LECCIÓN 24 MEMORIA COMPARTIDA

Índice de contenido

26.1. Notas previas

Para el estudio del código del núcleo implicado en la gestión de memoria compartida se ha utilizado la versión 2.6.11 de Linux.

Los fragmentos de código mostrados posiblemente contengan enlaces a Internet, concretamente hacia "Cross-Referencing Linux". Esto permitirá al lector visualizar rápidamente el código mencionado.

Sobre la terminología

A lo largo de la explicación usted se encontrará con los siguientes sinónimos: sección, región, segmento, zona y área, todos ellos referentes a espacios de memoria compartida. Son sólo un recurso para dinamizar la lectura.

26.2. Introducción: Comunicación entre procesos (IPC)

Se conoce por ***I**nter**P**rocess **C**ommunication* (IPC) al conjunto de mecanismos clave en los sistemas Unix para llevar a cabo la compartición de datos (intercomunicación) y sincronización entre distintos procesos de forma bastante sencilla.

Actualmente Linux proporciona tres mecanismos IPC: colas de mensajes, memoria compartida y semáforos.

Las **colas de mensajes** son un mecanismo muy similar a las FIFO. Una cola se puede ver como una lista enlazada de mensajes dentro del espacio de direccionamiento del núcleo. Una aplicación, siempre que tenga los derechos necesarios, puede depositar un mensaje (de cualquier tipo) en ella, y otras aplicaciones podrán leerlo. Es posibleasignar atributos a los mensajes, de forma que se puedan mantener ordenados por prioridad en lugar de por orden de llegada.

La **memoria compartida** es un medio que permite establecer una zona común de memoria entre varias aplicaciones.

Y los **semáforos**, que son una herramienta puramente de sincronización. Permiten controlar el acceso de varios procesos a recursos comunes.

Todos los mecanismos IPC basan su funcionamiento en un sistema de claves. Para crear un IPC o, simplemente, acceder a él, hay que contar con un identificador llamado clave, que identifica al IPC de manera única en todo el sistema. Para generar una clave se utiliza la función de biblioteca *ftok*, que debe recibir como parámetros el nombre de un fichero accesible al proceso *(pathname)* y un identificador (*proj*). Para el mismo fichero y diferentes valores del segundo parámetro se crearán

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok (char *pathname, char proj);
```

llaves diferentes.

26.3. El concepto de memoria compartida

La memoria convencional que puede direccionar un proceso a través de su espacio de direcciones virtuales es local a ese proceso, cualquier intento de direccionar esa memoria desde otro proceso va a provocar una violación de segmento.

El mecanismo de memoria compartida permite a dos o más procesos compartir un segmento de memoria, y por consiguiente, los datos que hay en él. Es por ello el método más rápido de comunicación entre procesos.

Al ser el objetivo de este tipo de comunicación la transferencia de datos entre varios procesos, los programas que utilizan memoria compartida deben normalmente establecer algún tipo de protocolo para el bloqueo. Este protocolo puede ser la utilización de semáforos, que es a su vez otro tipo de comunicación (sincronización) entre procesos.

La memoria que maneja un proceso, y también la compartida, va a ser virtual, por lo que su

dirección física puede variar con el tiempo. Esto no va a plantear ningún problema, ya que los procesos no generan direcciones físicas, sino virtuales, y es el núcleo (con su gestor de memoria) el encargado de traducir unas a otras.

26.4. Utilización

Para disfrutar de las ventajas que aporta el uso de memoria compartida se dispone de 4 llamadas al sistema: *shmget(), shmctl(), shmat()* y *shmdt()*. En este primer apartado se explicará de una forma general la utilidad de estas llamadas, para ser luego vistas con todo detalle en el siguiente apartado.

shmget()

Permite acceder a una zona de memoria compartida y, opcionalmente, crearla en caso de no existir. A esta llamada se le pasan tres argumentos: una clave, el tamaño del segmento a crear y el flag inicial de operación, y devuelve un entero, denominado *shmid*, que se utiliza para hacer referencia a

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

dicho segmento.

Nótese que *shmget()* únicamente reserva el espacio necesario para alojar el segmento. El segmento no se crea hasta que algún proceso se asigne a él. La posición de memoria final la decide, en principio, el sistema.

Si se utiliza para obtener una referencia a una zona de memoria compartida ya existente, el tamaño especificado debe ser inferior o igual al de la memoria existente. En caso contrario, el tamaño asignado debe ser múltiplo de PAGE_SIZE, que corresponde al tamaño de una página de memoria (4 KB en la arquitectura x86).

El valor de *shmflg* se compone básicamente con:

- IPC_CREAT: Para crear un nuevo segmento. Si este flag no se precisa, *shmget()* únicamente buscará el segmento asociado con la clave y comprobará si el usuario tiene permisos para acceder a él.
- IPC_EXCL: Se utiliza junto a IPC_CREAT para asegurar el fallo si el segmento existe.
- mode_flags: Los 9 bits menos significativos del número están reservados para el establecimiento de permisos de acceso. Actualmente los permisos de ejecución no son utilizados por el sistema.

Owner			Group			World		
R	W	X	R	W	X	R	W	X

shmctl()

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

Proporciona una variedad de operaciones para el control de la memoria compartida a través de *cmd*.

Algunas de las operaciones disponibles son:

- IPC_STAT: Lee el estado de la estructura de control de la memoria asociada a *shmid* y lo devuelve a través de la estructura de datos apuntada por *buf*.
- IPC_SET: Cambia el valor de los siguientes miembros de la estructura de datos asociada a shmid con el correspondiente valor encontrado en la estructura apuntada por *buf*: *shm_perm.uid shm_perm.gid shm_perm.mode*
- IPC_RMID: Marca una región de memoria compartida como destruida, aunque el borrado sólo se hará efectivo cuando el último proceso que la adjuntaba deje de hacerlo. Si en el momento en que se realiza la invocación, el segmento aún está asignado a otro proceso, la clave será puesta a IPC_PRIVATE.
- SHM_LOCK: Bloquea la zona de memoria compartida especificada por *shmid*. Esto quiere decir que el segmento es grabado en memoria, no siendo posible borrarlo a partir de esta operación. Tampoco se podrá hacer swapping sobre él. Esta *cmd* solo puede ser ejecutada por un proceso que tenga un ID de usuario efectivo igual al del superusuario. De esta forma se garantiza la permanencia de los datos en la memoria.
- SHM_UNLOCK: Desbloquea la región de memoria compartida especificada por *shmid*. Esta operación, al igual que la anterior, sólo la podrán ejecutar aquellos procesos con privilegios de acceso apropiados.

shmat()

Es la llamada que debe invocar un proceso para adjuntar una zona de memoria compartida dentro de su espacio de direcciones. Recibe tres parámetros: el identificador del segmento (*shmid*), una dirección de memoria (*shmaddr*) y las banderas descritas más adelante. La dirección puede ser NULL; en tal caso el sistema operativo tomará la responsabilidad de buscar una posición de memoria libre.

```
#include <sys/types.h>
#include <sys/shm.h>
void* shmat (int shmid, const void *shmaddr, int option);
```

En *option* se puede especificar:

- SHM_RND: En cuyo caso, el sistema intentará vincular la zona de memoria a una dirección múltiplo de SHMLBA (shmparam.h) lo más próxima posible a la especificada.
- SHM_RDONLY: Hará que el proceso sólo pueda acceder al segmento de memoria en

lectura. Si no se precisa, el segmento se vinculará en modo lectura/escritura.

Como resultado de la invocación, devuelve la dirección de comienzo del segmento de memoria asignado y se actualizan los siguientes campos de la estructura *shmid_kernel:*
shm_atim, que recibe la fecha actual; *shm_lprid*, que recibe el pid del proceso que llama, y *shm_nattch*, que se incrementa en una unidad.

shmdt()

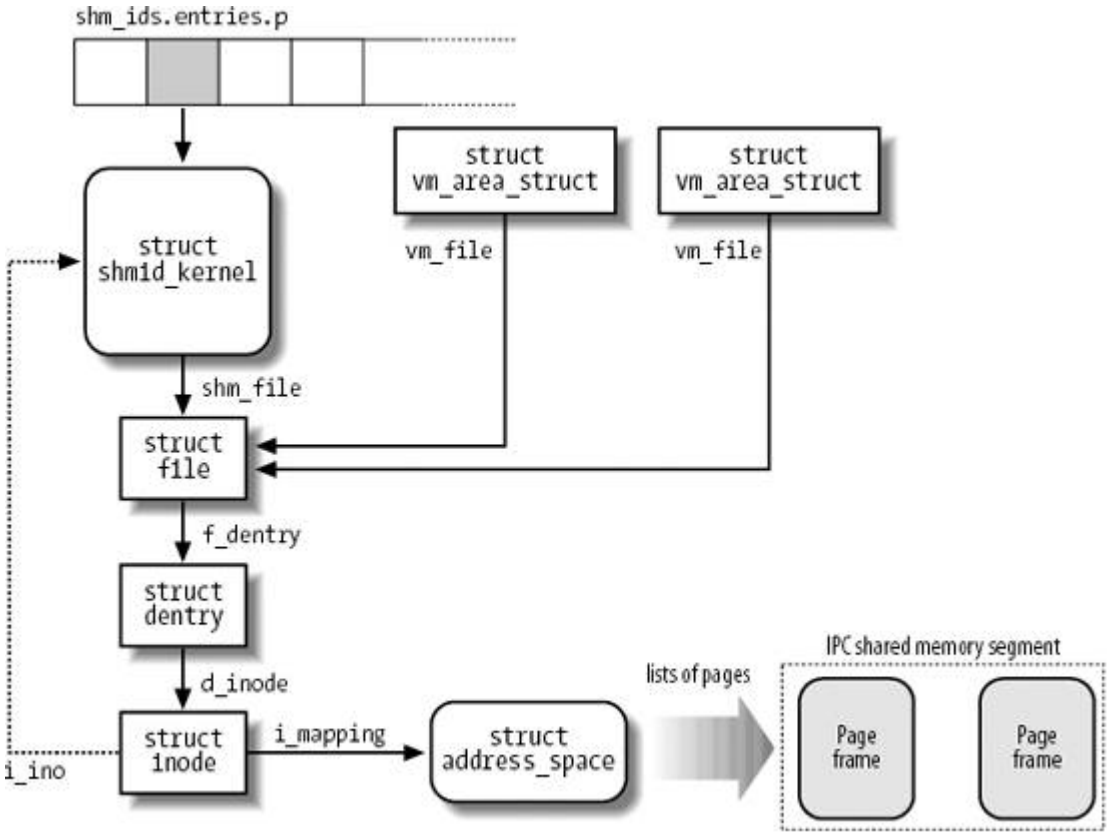
```
#include <sys/shm.h>
#include <sys/types.h>
int shmdt(void *shmaddr);
```

Es la operación que realizará el proceso anterior para desvincularse de una zona de memoria compartida. Para ello, deberá precisarse como primer y único parámetro la dirección *shmaddr* de dicha zona.

Esta llamada al sistema actualiza los campos siguientes de las estructura *shmid_kernel*: *shm_dtim*, que recibe la fecha actual; *shm_lprid*, que recibe el pid del proceso que llama y *shm_nattch*, que queda decrementado en una unidad.

26.5. Implementación

La siguiente figura nos proporciona una visión rápida de las estructuras asociadas a las regiones de memoria compartida de IPC.



La variable *shm_ids*, declarada en "ipc/shm.c" almacena una estructura de tipo *ipc_ids* y que se utiliza para almacenar un array de estructuras *shmid_kernel*, una por cada segmento de memoria compartido.

```
39 static struct ipc_ids shm_ids;
```

Como observación, y dado que no pretendemos salirnos de la temática que se está abordando, decir únicamente que una región de memoria compartida viene descrita por su *vm_area_struct* y que el campo *vm_file* de esta estructura apunta hacia un objeto fichero del sistema de ficheros *shm*. Esto permite realizar *swapping* sobre la memoria compartida.

Estructuras de datos básicas

struct shmid_ds

Actualmente se mantiene sólo por compatibilidad hacia detrás, básicamente con "libc". Recordemos, por ejemplo, que la llamada al sistema "shmctl" que hemos vimos con anterioridad espera como tercer parámetro un puntero a una estructura de este tipo. En la versión 2.6 de Linux se ha sido reemplazada en las gestiones del núcleo por *shmid_kernel.*

Almacena información sobre un segmento de memoria: permisos, tamaño, etc. Tiene la forma siguiente:

Dirección en fuentes: /include/linux/shm.h

```
22 struct shmid_ds {
23         struct ipc_perm         shm_perm;
24         int                     shm_segsz;
25         __kernel_time_t         shm_atime;
26         __kernel_time_t         shm_dtime;
27         __kernel_time_t         shm_ctime;
28         __kernel_ipc_pid_t      shm_cpid;
29         __kernel_ipc_pid_t      shm_lpid;
30         unsigned short          shm_nattch;
31         unsigned short          shm_unused;
32         void                    *shm_unused2;
33         void                    *shm_unused3;
34 };
```

El nombre y significado de los campos se mantiene en *shmid_kernel*. Para más detalle consulte la explicación de dicha estructura.

struct shmid_kernel

Mantiene información importante sobre el segmento de memoria. Está definida en su totalidad en el

Dirección en fuentes: /include/linux/shm.h

```
76 struct shmid_kernel /* private to the kernel */
77 {
78         struct kern_ipc_perm    shm_perm;
79         struct file *           shm_file;
80         int                     id;
81         unsigned long           shm_nattch;
82         unsigned long           shm_segsz;
83         time_t                  shm_atim;
84         time_t                  shm_dtim;
85         time_t                  shm_ctim;
86         pid_t                   shm_cprid;
87         pid_t                   shm_lprid;
88         struct user_struct      *mlock_user;
89 };
```

fichero “include/linux/shm.h”:

A continuación se describen los campos que componen esta estructura:

Campo	Descripción
kern_ipc_perm	Estructura donde se almacenan los permisos
shm_file	Almacena la dirección de un objeto fichero.
id	Índice del slot del segmento de memoria
shm_nattch	Nümero de procesos que actualmente se encuentran vinculados
shm_segsz	Tamaño del segmento (bytes
shm_atim	Fecha de la última vinculación
shm_dtim	Fecha de la última desvinculación
shm_ctim	Fecha de la última modificación
shm_cprid	PID del creador
shm_lprid	PID del último proceso que accedió
mlock_user	Puntero al descriptor *user_struct* del usuario usuario que bloqueó en RAM el recurso de memroia compartida.

El campo más importante de esta estructura es *shm_file*. Cada región de memoria está asociada con un fichero perteneciente al sistema de ficheros especial *shm*. Esto refleja la estrecha relación de *IPC shared memory* con la capa VFS en el núcleo 2.6.

struct shm_info

Esta estructura se utiliza para obtener información sobre los recursos del sistema consumidos por el

Dirección en fuentes: /include/linux/shm.h

```
66 struct shm_info {
67         int used_ids;
68         unsigned long shm_tot;
69         unsigned long shm_rss;
70         unsigned long shm_swp;
71         unsigned long swap_attempts;
72         unsigned long swap_successes;
73 };
```

uso de memoria compartida. Esto es posible invocando a *shmctl* con cmd igual a SHM_INFO.

El significado de los campos es el siguiente:

Campo	Descripción
used_ids	Segmentos que se están utilizando actualmente
shm_tot	Número total de páginas compartidas
shm_rss	Número total de páginas residentes
shm_swp	Número total de páginas intercambiadas (swapping)
swap_attempts	Número de intentos de swapping sobre una página de la región
swap_successes	Número de intentos de swapping que resultaron exitosos

struct shminfo

Se utiliza en la llamada *shmctl* cuando se precisa IPC_INFO como argumento. Se usa raramente, principalmente en programas de sistema de estadísticas o de observación de la máquina (como *ipcs*).

Dirección en fuentes: /include/linux/shm.h

```
58 struct  shminfo {
59         int shmmax;
60         int shmmin;
61         int shmmni;
62         int shmseg;
63         int shmall;
64 };
```

Los campos detallados son los siguientes:

Campo	*Descripción*
shmmax	Tamaño máximo del segmento(bytes)
shmmin	Tamaño mínimo del segmento(bytes)
shmmni	Número máximo de segmentos
shmseg	Número máximo de segmentos por proceso
shmall	Número máximo de segmentos en número de páginas de memoria.

Otras estructuras de datos relacionadas

Excepcionalmente fijaremos nuestra atención en dos estructuras más: *kern_ipc_perm* e *ipc_ids*. No es nuestro objetivo profundizar en ellas, ya que se utilizan de forma general en la gestión de cualquier tipo de IPC. Se encuentran en las cabeceras "/include/linux/ipc.h" y "/include/linux/util.h" respectivamente.

struct kern_ipc_perm

Cada descriptor IPC tiene un objeto de datos de este tipo como primer elemento. Esto hace posible acceder a cualquier descriptor desde cualquier función genérica IPC usando un puntero de este tipo

Dirección en fuentes: /include/linux/ipc.h

```
57 struct kern_ipc_perm
58 {
59         spinlock_t      lock;
60         int             deleted;
61         key_t           key;      // La clave que proporcionó shmget()
62         uid_t           uid;      // Usuario propietario
63         gid_t           gid;      // Grupo del propietario
64         uid_t           cuid;     // Usuario Creador
65         gid_t           cgid;     // Grupo del creador
66         mode_t          mode;     // Modo de acceso (9 bits)
67         unsigned long   seq;      // Número de secuencia
68         void            *security;
69 };
```

de datos.

Los campos referentes al creador son fijos y no es posible modificarlos. Por otro lado, los campos referentes al propietario indican que usuarios pueden realizar operaciones de control sobre el segmento, y sólo pueden modificarlos los usuarios creador o propietario. Los permisos de operación de un proceso sobre un segmento de memoria compartida afectan a la lectura y escritura sobre el mismo. El *lock* protege el acceso a esta estructura.

struct ipc_ids

La estructura *ipc ids* describe los datos comunes para los semáforos, colas de mensajes, y memoria compartida. Hay tres instancias globales de esta estructura de datos –*semid_ids, msgid_ids* y *shmid_ids*– para los semáforos, mensajes y memoria compartida respectivamente. En cada instancia, el semáforo *sem* es usado para proteger el acceso a la estructura. El campo *entries* apunta a una matriz de descriptores de IPC, y el campo *seq* es una secuencia de números global la cual será incrementada cuando un nuevo recurso IPC sea creado.

Dirección en fuentes: /include/linux/util.h

```
23 struct ipc_ids {
24         int in_use;
25         int max_id;
26         unsigned short seq;
27         unsigned short seq_max;
28         struct semaphore sem;
29         struct ipc_id_ary nullentry;
30         struct ipc_id_ary* entries;
31 };
32
```

Funciones principales

La definición de las funciones asociadas a las cuatro llamadas al sistema que hemos visto se realiza en el fichero fuente “ipc/shm.c”. Estas son *sys_shmget*, *sys_shmctl*, *sys_shmdt* y *do_shmat*, para shmget,shmctl, shmdt y shmat, respectivamente.

sys_shmget()

Esta llamada al sistema se encuentra protegida por el semáforo global de memoria compartida. Cuando se suministra (caso general) un valor de clave, éste se busca en la matriz de descriptores de memoria compartida. Si se encuentra, se devuelve el ID de la región de memoria correspondiente. Eso sí, no sin antes comprobar si los parámetros indicados sin válidos y si se tiene permisos para acceder a ella. Las operaciones de búsqueda y verificación son realizadas mientras es mantenido el spinlock global de memoria

```
248 asmlinkage long sys_shmget (key_t key, size_t size, int shmflg)
249 {
250         struct shmid_kernel *shp;
251         int err, id = 0;
252
253         down(&shm_ids.sem);
254         if (key == IPC_PRIVATE) {
255                 err = newseg(key, shmflg, size);
256         } else if ((id = ipc_findkey(&shm_ids, key)) == -1) {
257                 if (!(shmflg & IPC_CREAT))
258                         err = -ENOENT;
259                 else
260                         err = newseg(key, shmflg, size);
261         } else if ((shmflg & IPC_CREAT) && (shmflg & IPC_EXCL)) {
262                 err = -EEXIST;
263         } else {
264                 shp = shm_lock(id);
265                 if(shp==NULL)
266                         BUG();
267                 if (shp->shm_segsz < size)
268                         err = -EINVAL;
269                 else if (ipcperms(&shp->shm_perm, shmflg))
270                         err = -EACCES;
```

```
271                 else {
272                         int shmid = shm_buildid(id, shp->shm_perm.seq);
273                         err = security_shm_associate(shp, shmflg);
274                         if (!err)
275                                 err = shmid;
276                 }
277                 shm_unlock(shp);
278         }
279         up(&shm_ids.sem);
280
281         return err;
282 }
```

sys_shmctl

La llamada “sys_shmctl”, utilizada para realizar operaciones de control sobre la estructura shmid_kernel asociada a cada región de memoria compartida, recibe los siguientes parámetros: el identificador del segmento, el comando deseado y adicionalmente una estructura “shmid_ds” para pasar intercambiar información entre el proceso llamador y el sistema (para rellenarla o copiar en ella valores del *shm_info):*

```
405 asmlinkage long sys_shmctl (int shmid, int cmd, struct shmid_ds __user *buf)
406 {
407         struct shm_setbuf setbuf;
408         struct *shp;
409         int err, version;
410
```

El primer paso es comprobar que tanto el comando como el identificador *shmid* son válidos:

```
411         if (cmd < 0 || shmid < 0) {
412                 err = -EINVAL;
413                 goto out;
414         }
415
```

A continuación se actúa de acuerdo al comando que se explicitó. Por ello entramos en el siguiente switch:

```
418         switch (cmd) {
```

Tomemos, por ejemplo, que el comando pasado es IPC_INFO.

```
419         case IPC_INFO:
420         {
421                 struct shminfo64 shminfo;  // La estructura shminfo64 es
422
```

Se comprueba si la llamada cumple los requerimientos de seguridad

```
423                 err = security_shm_shmctl(NULL, cmd);
424                 if (err)
425                         return err;
426
```

Y se carga una antememoria temporal shminfo64con los parámetros del sistema de memoria compartida

```
427                 memset(&shminfo,0,sizeof(shminfo));
428                 shminfo.shmmni = shminfo.shmseg = shm_ctlmni;
429                 shminfo.shmmax = shm_ctlmax;
430                 shminfo.shmall = shm_ctlall;
431
432                 shminfo.shmmin = SHMMIN;
```

para ser luego copiada fuera del espacio de usuario para el acceso de la aplicación llamadora.

```
433                 if(copy_shminfo_to_user (buf, &shminfo, version))
434                         return -EFAULT;
435                 /* reading a integer is always atomic */
436                 err= shm_ids.max_id;
437                 if(err<0)
438                         err = 0;
439                 goto out;
440         }
```

El caso de SHM_INFO es análogo al de IPC_INFO, sólo varía la estructura de devolución implicada, que ahora será de tipo shm_info:

```
441         case SHM_INFO:
442         {
443                 struct shm_info shm_info;
444
445                 err = security_shm_shmctl(NULL, cmd);
446                 if (err)
447                         return err;
448
449                 memset(&shm_info,0,sizeof(shm_info));
450                 down(&shm_ids.sem);
451                 shm_info.used_ids = shm_ids.in_use;
452                 shm_get_stat (&shm_info.shm_rss, &shm_info.shm_swp);
453                 shm_info.shm_tot = shm_tot;
454                 shm_info.swap_attempts = 0;
455                 shm_info.swap_successes = 0;
456                 err = shm_ids.max_id;
457                 up(&shm_ids.sem);
458                 if(copy_to_user (buf, &shm_info, sizeof(shm_info))) {
459                         err = -EFAULT;
460                         goto out;
461                 }
462
463                 err = err < 0 ? 0 : err;
464                 goto out;
465         }
```

En los casos de SHM_STAT e IPC_STAT se procede prácticamente de la misma forma. Es por ello que se utiliza el mismo trozo de código para ambos:

```
466         case SHM_STAT:
```

```
467         case IPC_STAT:
468         {
469                 struct shmid64_ds tbuf;
470                 int result;
```

Se crea una antememoria temporal de tipo *shmid64_ds* (similar shmid_ds) y se toma el cerrojo de de acceso a la estructura de control:

```
471                 memset(&tbuf, 0, sizeof(tbuf));
472                 shp = shm_lock(shmid);
473                 if(shp==NULL) {
474                         err = -EINVAL;
475                         goto out;
476                 } else
```

Para el caso SHM STAT, el parámetro ID del segmento de memoria compartida se espera que sea un índice válido:

```
                    if(cmd==SHM_STAT) {
477                     err = -EINVAL;
478                     if (shmid > shm_ids.max_id)
479                             goto out_unlock;
```

Después de validar el índice se llama a ipc buildid() para convertir el índice en una ID de memoria compartida. En el caso de SHM STAT, la ID de la memoria compartida será el valor de retorno.

```
480                     result = shm_buildid(shmid, shp->shm_perm.seq);
481                 } else {
```

Para el caso IPC STAT, el parámetro ID del segmento de memoria compartida se espera que sea una ID válido (que haya sido generado por la llamada shmget()). Por ello, se valida el ID antes de procedecer. En el caso de IPC STAT, el valor de retorno será 0

```
482                     err = shm_checkid(shp,shmid);
483                     if(err)
484                             goto out_unlock;
485                     result = 0;
486                 }
487                 err=-EACCES;
```

A continuación se verifican los permisos de acceso:

```
488                 if (ipcperms (&shp->shm_perm, S_IRUGO))
489                         goto out_unlock;
490                 err = security_shm_shmctl(shp, cmd);
491                 if (err)
492                         goto out_unlock;
```

Y finalmente las estadísticas deseadas son cargadas en la ante memoria temporal y copiadas fuera de la aplicación llamadora.

```
493                 kernel_to_ipc64_perm(&shp->shm_perm, &tbuf.shm_perm);
494                 tbuf.shm_segsz  = shp->shm_segsz;
495                 tbuf.shm_atime  = shp->shm_atim;
496                 tbuf.shm_dtime  = shp->shm_dtim;
497                 tbuf.shm_ctime  = shp->shm_ctim;
```

```
498                     tbuf.shm_cpid   = shp->shm_cprid;
499                     tbuf.shm_lpid   = shp->shm_lprid;
500                     if (!is_file_hugepages(shp->shm_file))
501                             tbuf.shm_nattch = shp->shm_nattch;
502                     else
503                             tbuf.shm_nattch = file_count(shp->shm_file) - 1;
504                     shm_unlock(shp);
505                     if(copy_shmid_to_user (buf, &tbuf, version))
506                             err = -EFAULT;
507                     else
508                             err = result;
509                     goto out;
510             }
```

Tratemos ahora los casos de SHMLOCK y UNLOCK.

```
511             case SHM_LOCK:
512             case SHM_UNLOCK:
513             {
```

Después de validar los permisos de acceso se toma, como siempre, el spinlock global de memoria compartida:

```
514                     shp = shm_lock(shmid);
515                     if(shp==NULL) {
516                             err = -EINVAL;
517                             goto out;
518                     }
```

Luego se comprueba el si el ID es válido

```
519                     err = shm_checkid(shp,shmid);
520                     if(err)
521                             goto out_unlock;
522
```

Se comprueba si el proceso actual tiene capacidad para bloquear el segmento

```
523                     if (!capable(CAP_IPC_LOCK)) {
524                             err = -EPERM;
525                             if (current->euid != shp->shm_perm.uid &&
526                                 current->euid != shp->shm_perm.cuid)
527                                     goto out_unlock;
528                             if (cmd == SHM_LOCK &&
529                                 !current->signal->rlim[RLIMIT_MEMLOCK].rlim_cur)
530                                     goto out_unlock;
531                     }
```

Si la tiene, se comprueba se comprueban más permisos:

```
532
533                     err = security_shm_shmctl(shp, cmd);
534                     if (err)
535                             goto out_unlock;
536
```

Y finalmente se bloquea la sección de memoria compartida:

```
537                 if(cmd==SHM_LOCK) {
538                         struct user_struct * user = current->user;
539                         if (!is_file_hugepages(shp->shm_file)) {
540                                 err = shmem_lock(shp->shm_file, 1, user);
541                                 if (!err) {
542                                         shp->shm_flags |= SHM_LOCKED;
543                                         shp->mlock_user = user;
544                                 }
545                         }
546                 } else if (!is_file_hugepages(shp->shm_file)) {
547                         shmem_lock(shp->shm_file, 0, shp->mlock_user);
548                         shp->shm_flags &= ~SHM_LOCKED;
549                         shp->mlock_user = NULL;
550                 }
551                 shm_unlock(shp);
552                 goto out;
553         }
```

Durante el IPC RMID el semáforo global de memoria compartida y el spinlock global son mantenidos. Se valida la ID de la Memoria Compartida y entonces si no hay vínculos de la región de memoria con procesos, se destruye a través de shm destroy().

En otro caso, se establece la bandera SHM DEST para marcarlo para destrucción, y se fija la bandera IPC PRIVATE para prevenir que otro proceso sea capaz de referenciar la ID de la memoria compartida.

```
554         case IPC_RMID:
555         {
566                 down(&shm_ids.sem);
567                 shp = shm_lock(shmid);
568                 err = -EINVAL;
569                 if (shp == NULL)
570                         goto out_up;
571                 err = shm_checkid(shp, shmid);
572                 if(err)
573                         goto out_unlock_up;
574
575                 if (current->euid != shp->shm_perm.uid &&
576                     current->euid != shp->shm_perm.cuid &&
577                     !capable(CAP_SYS_ADMIN)) {
578                         err=-EPERM;
579                         goto out_unlock_up;
580                 }
581
582                 err = security_shm_shmctl(shp, cmd);
583                 if (err)
584                         goto out_unlock_up;
585
586                 if (shp->shm_nattch){
587                         shp->shm_flags |= SHM_DEST;
588                         /* Do not find it any more */
589                         shp->shm_perm.key = IPC_PRIVATE;
590                         shm_unlock(shp);
591                 } else
592                         shm_destroy (shp);
593                 up(&shm_ids.sem);
594                 goto out;
```

```
595         }
596
```

Y finalmente, en el caso de IPC_SET tras validar que es posible realizar la operación, las banderas uid, gid, y mode del segmento de la memoria compartida son actualizadas con los datos del usuario. El campo shm_ctime también es actualizado. Estos cambios son realizados mientras se mantiene el semáforo global de memoria compartida global y el spinlock global de memoria compartida.

```
597         case IPC_SET:
598         {
599                 if (copy_shmid_from_user (&setbuf, buf, version)) {
600                         err = -EFAULT;
601                         goto out;
602                 }
603                 down(&shm_ids.sem);
604                 shp = shm_lock(shmid);
605                 err=-EINVAL;
606                 if(shp==NULL)
607                         goto out_up;
608                 err = shm_checkid(shp,shmid);
609                 if(err)
610                         goto out_unlock_up;
611                 err=-EPERM;
612                 if (current->euid != shp->shm_perm.uid &&
613                     current->euid != shp->shm_perm.cuid &&
614                     !capable(CAP_SYS_ADMIN)) {
615                         goto out_unlock_up;
616                 }
617
618                 err = security_shm_shmctl(shp, cmd);
619                 if (err)
620                         goto out_unlock_up;
621
622                 shp->shm_perm.uid = setbuf.uid;
623                 shp->shm_perm.gid = setbuf.gid;
624                 shp->shm_flags = (shp->shm_flags & ~S_IRWXUGO)
625                         | (setbuf.mode & S_IRWXUGO);
626                 shp->shm_ctim = get_seconds();
627                 break;
628         }
629
630         default:
631                 err = -EINVAL;
632                 goto out;
633         }
634
635         err = 0;
636 out_unlock_up:
637         shm_unlock(shp);
638 out_up:
639         up(&shm_ids.sem);
640         goto out;
641 out_unlock:
642         shm_unlock(shp);
643 out:
644         return err;
645 }
```

do_shmat

Esta función es también una llamada al sistema, pese a que no siga la nomenclatura habitual de utilizar el prefijo “sys_”. Esto es debido a que en el código encargado de realizar las llamadas a las rutinas manejadoras (“arch/i386/kernel/sys_i386.c” en el caso de arquitecturas 386) se realiza alguna acción adicional.

La función *do_shmat* toma como parámetros una ID de segmento de memoria compartida, una dirección en la cual el segmento de memoria compartida debería de ser conectada (*shmaddr*), y las banderas

```
654 long do_shmat(int shmid, char __user *shmaddr, int shmflg, ulong *raddr)
655 {
656         struct shmid_kernel *shp;
657         unsigned long addr;
658         unsigned long size;
659         struct file * file;
660         int    err;
661         unsigned long flags;
662         unsigned long prot;
663         unsigned long o_flags;
664         int acc_mode;
665         void *user_addr;
667         if (shmid < 0) {
668                 err = -EINVAL;
669                 goto out;
670         } else if ((addr = (ulong)shmaddr)) {
671                 if (addr & (SHMLBA-1)) {
```

Si *shmaddr* no es cero, y la bandera SHM_RND es especificada, entonces se redondea inferiormente *shmaddr* a un múltiplo de SHMLBA. Si shmaddr no es un múltiplo de SHMLBA y no se especifica SHM_RND, entonces se devuelve EINVAL.

```
672                         if (shmflg & SHM_RND)
673                                 addr &= ~(SHMLBA-1);
672                         if (shmflg & SHM_RND)
673                                 addr &= ~(SHMLBA-1);
675 #ifndef __ARCH_FORCE_SHMLBA
676                                 if (addr & ~PAGE_MASK)
677 #endif
678                                         return -EINVAL;
679                 }
680                 flags = MAP_SHARED | MAP_FIXED;

681         } else {
682                 if ((shmflg & SHM_REMAP))
683                         return -EINVAL;
684
685                 flags = MAP_SHARED;
686         }
687
```

Se comprueban los permisos y se incrementa el campo shm nattch del segmento de memoria compartida. Nótese que este incremento garantiza que la cuenta de enlaces no es cero y previene que el segmento de memoria compartida sea destruido durante el proceso de enlazamiento al segmento. Todas estas operaciones son realizadas mientras se mantiene el spinlock global de

memoria compartida.

```
688         if (shmflg & SHM_RDONLY) {
689                 prot = PROT_READ;
690                 o_flags = O_RDONLY;
691                 acc_mode = S_IRUGO;
692         } else {
693                 prot = PROT_READ | PROT_WRITE;
694                 o_flags = O_RDWR;
695                 acc_mode = S_IRUGO | S_IWUGO;
696         }
697         if (shmflg & SHM_EXEC) {
698                 prot |= PROT_EXEC;
699                 acc_mode |= S_IXUGO;
700         }
701
702         /*
703          * We cannot rely on the fs check since SYSV IPC does have an
704          * additional creator id...
705          */
706         shp = shm_lock(shmid);
707         if(shp == NULL) {
708                 err = -EINVAL;
709                 goto out;
710         }
711         err = shm_checkid(shp,shmid);
712         if (err) {
713                 shm_unlock(shp);
714                 goto out;
715         }
716         if (ipcperms(&shp->shm_perm, acc_mode)) {
717                 shm_unlock(shp);
718                 err = -EACCES;
719                 goto out;
720         }
721
722         err = security_shm_shmat(shp, shmaddr, shmflg);
723         if (err) {
724                 shm_unlock(shp);
725                 return err;
726         }
727
728         file = shp->shm_file;
729         size = i_size_read(file->f_dentry->d_inode);
730         shp->shm_nattch++;
731         shm_unlock(shp);
732
```

Se llama a la función *do_mmap()* para crear un mapeo de memoria virtual de las páginas del segmento de memoria compartida, para lo cual se ha tomado el semáforo *mmap_sem* de la tarea actual.

NÓTESE que el incremento con la llamada a la función *do_mmap()* a través de la estructura también se establecerán el PID y la fecha actual y se incrementará el número de enlaces a este segmento de memoria compartida.

```
733         down_write(&current->mm->mmap_sem);
734         if (addr && !(shmflg & SHM_REMAP)) {
```

```
735                 user_addr = ERR_PTR(-EINVAL);
736                 if (find_vma_intersection(current->mm, addr, addr + size))
737                         goto invalid;
738                 /*
739                  * If shm segment goes below stack, make sure there is some
740                  * space left for the stack to grow (at least 4 pages).
741                  */
742                 if (addr < current->mm->start_stack &&
743                     addr > current->mm->start_stack - size - PAGE_SIZE * 5)
744                         goto invalid;
745         }
746
747         user_addr = (void*) do_mmap (file, addr, size, prot, flags, 0);
748
749 invalid:
```

Después de la llamada a *do_mmap()*, se obtiene el semáforo global de memoria compartida y el spinlock global de la memoria compartida. Se decrementa la cuenta de enlaces y es entonces decrementada.

```
750         up_write(&current->mm->mmap_sem);
751
752         down (&shm_ids.sem);
753         if(!(shp = shm_lock(shmid)))
754                 BUG();
755         shp->shm_nattch--;
```

Si después de decrementar la cuenta de enlaces, la cuenta resultante que se encuentra es cero, el segmento se marca para la destrucción y se llama a *shm_destroy()* para liberar los recursos del segmento de memoria compartida.

```
756         if(shp->shm_nattch == 0 &&
757            shp->shm_flags & SHM_DEST)
758                 shm_destroy (shp);
759         else
760                 shm_unlock(shp);
761         up (&shm_ids.sem);
762
763         *raddr = (unsigned long) user_addr;
764         err = 0;
765         if (IS_ERR(user_addr))
766                 err = PTR_ERR(user_addr);
```

Finalmente se devuelve la dirección virtual donde se encuentra la zona de memoria recién añadida:

```
767 out:
768         return err;
769 }
```

sys_shmdt

Esta llamada únicamente recibe como parámetro la dirección de la región de memoria de la que el proceso desea desvincularse:

```
672 asmlinkage long sys_shmdt (char *shmaddr)
673 {
```

Se toma el *mm_struct* del actual proceso para buscar la vm_area_struct asociada con la dirección de memoria pasada.

```
674         struct mm_struct *mm = current->mm;
675         struct vm_area_struct *shmd, *shmdnext;
676         int retval = -EINVAL;
```

Primeramente bloquea la escritura a través del semáforo *mmap_sem:*

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

Luego entra en un bucle de búsqueda sobre el *mm_struct*. Nótese también que *do_munmap()* realiza una llamada a *shm_close*() para libera los recursos del segmento de memoria compartida si no hay más enlaces.

```
679         for (shmd = mm->mmap; shmd; shmd = shmdnext) {
680                 shmdnext = shmd->vm_next;
681                 if (shmd->vm_ops == &shm_vm_ops
682                     && shmd->vm_start - (shmd->vm_pgoff << PAGE_SHIFT) ==
(ulong) shmaddr) {
683                         do_munmap(mm, shmd->vm_start, shmd->vm_end - shmd-
>vm_start);
684                         retval = 0;
685                 }
686         }
```

Finalmente se habilita la escritura en la la *mm_struct* y se retorna 0 si se liberó con éxito y -1 en caso contrario (no existía la zona direccionada)

```
687         up_write(&mm->mmap_sem);
688         return retval;
689 }
```

26.6. Curiosidades

Existen dos comandos útiles relacionados con el tema que estamos tratando, y que pueden ser utilizados por cualquier usuario desde una terminal.

Terminal

```
[gandalf@clark ~]$ ipcs
---- Segmentos memoria compartida ----
key        shmid      propietario perms      bytes      nattch     estado
0x00005d8b 32769      root        777        316        1
0x00000000 655362     gandalf     600        393216     2          dest

------ Matrices semáforo -------
key        semid      propietario perms      nsems

------ Colas de mensajes -----
key        msqid      propietario perms      bytes utilizados mensajes
0x42031ca6 0          gandalf    700        0            0
0x42031ca8 32769      gandalf    700        0            0
```

- **“ipcs”** permite visualizar información sobre recursos IPC en el sistema:

- **“ipcrm”** permite destruir un IPC. Es necesario ser el creador del IPC o el superusuario para poder efecturas esta operación.

26.7. Referencias bibliográficas

[***Oreilly***] Daniel P. Bovet, Marco Cesati, *"Understanding The Linux Kernel"*. 3rd Edition. O'Reilly Media, Inc., 2005. ISBN: 0-596-00565-2

[***INSIDE***] Tigran Aivazian, *"Dentro del n´ucleo Linux 2.4"*. 3rd Edition. Proyeto Lucas, 2001. URL:http://es.tldp.org/Manuales-LuCAS/DENTRO-NUCLEO-LINUX

[***RCard***] Rémy Card, Éric Dumas. *"Programación Linux 2.0. API y funcionamiento del cúcleo"*. Ediciones Gestión, 2000.