LECCIÓN 23: SEMÁFOROS

23.1 Conceptos

El concepto de semáforo nace de la necesidad de crear un sistema operativo en el que puedan trabajar procesos cooperantes. No es un mecanismo de comunicación sino de sincronización y son utilizados para controlar el acceso a los recursos.

Un semáforo básico es una variable entera y dos operaciones atómicas (sin interrupciones) que la manejan:

- **Espera (P)**: Se usa cuando un proceso quiere acceder a un recurso compartido y puede ocurrir:
  - Si la variable entera es positiva, coge el recurso y decrementa dicho valor.
  - En caso de que el valor sea nulo el proceso se duerme y espera a ser despertado.
- **Señal (V)**: Se utiliza para indicar que el recurso compartido esta libre y despertar a los procesos que estén esperando por el recurso.

Problemas que resuelven principalmente los semáforos:
- La exclusión mutua.
- Sincronización de Procesos

Un proceso puede necesitar varios recursos, pero si aplicamos un único semárofo a todos los recursos se puede producir un interbloqueo. Para solucionar esto podemos utilizar una **tablas de semáforos** y aplicar las operaciones de espera y señal sobre cada semáforo de la tabla.

Tabla de semáforos: En algunos casos, un proceso necesita poseer varios recursos para proseguir su acción, por ej. deberá acceder tal vez a memoria intermedia de datos y a un segmento de memoria compartida si desea desplazar los datos de un lugar a otro. Será necesario que utilice dos semáforos y ejecute dos operaciones P (una para la memoria intermedia y otra para la memoria) a fin de poder disponer de los dos recursos. Esta situación puede provocar un bloqueo cruzado en el caso siguiente: un proceso posee el acceso exclusivo a la memoria intermedia y desea acceder a la memoria mientras que otro posee el acceso exclusivo a la memoria y desea utilizar la memoria intermedia.

Solución: P y V no se efectúan “atómicamente” con un solo semáforo sino con una tabla de semáforos.

Los ficheros que contienen el código sobre semáforos son:
- ***“include/linux/sem.h”***: Se encuentra las estructuras necesarias para el manejo de semaforos.
- ***“ipc/sem.c”***: Implementación de las funciones y procedimientos así como las llamadas al sistema asociadas a estos.
- ***“ipc/util.c”***: Implementación de las funciones ipcget, ipcget_new e ipcget_public utilizadas para la llamada semget.

23.2 Estructuras

Estructura sem_array: Se trata de una estructura de control asociada a cada uno de los distintos conjuntos de semáforos existentes en el sistema. Contiene información del sistema, punteros a las operaciones a realizar sobre el grupo, y un puntero hacia las estructuras sem que se almacenan en el núcleo y contienen información de cada semáforo.

TIPO	CAMPO	DESCRIPCIÓN
struct kern_ipc_perm	sem_perm	Permisos
time_t	sem_otime	Fecha de la última operación
time_t	sem_ctime	Fecha del último cambio por semctl
struct sem*	sem_base	Puntero al primer semáforo del grupo
struct sem_queue*	sem_pending	Operaciones en espera de realización
struct sem_queue**	sem_pending_last	Última operación en espera
struct sem_undo*	undo	Operaciones anuladas en caso de terminación
unsigned long	sem_nsems	Número de semáforos del grupo

Estructura semid_ds: Se trata de una estructura de control asociada a cada uno de los distintos conjuntos de semáforos existentes en el sistema. Contiene información del sistema, punteros a las operaciones a realizar sobre el grupo, y un puntero hacia las estructuras sem que se almacenan en el núcleo y contienen información de cada semáforo. Esta estructura está en desuso actualmente y ha sido sustituida por **sem_array** solo se utiliza para compatibilidades.

TIPO	CAMPO	DESCRIPCIÓN
struct ipc_perm	sem_perm	Permisos
__kernel_time_t	sem_otime	Fecha de la última operación
__kernal_time_t	Sem_ctime	Fecha del último cambio por semctl
struct sem*	sem_base	Puntero al primer semáforo del grupo
struct sem_queue*	sem_pending	Operaciones en espera de realización
struct sem_queue**	sem_pending_last	Última operación en espera
struct sem_undo*	undo	Operaciones anuladas en caso de terminación
unsigned short	sem_nsems	Número de semáforos del grupo

Estructura sembuf: Se trata de una estructura que se utiliza en semop, y cada dato de este tipo especifica una operación a realizar sobre un semáforo particular dentro del conjunto. Incrementar, decrementar o esperar un valor nulo y por lo tanto se usa en la llamada semop.

TIPO	CAMPO	DESCRIPCIÓN
unsigned short	sem_num	Número de semáforos en el grupo
short	sem_op	Operaciones sobre el semáforo
short	sem_flg	Opciones

***Estructura semun*:** Es una unión, se utiliza en la llamada semctl para almacenar o recuperar informaciones sobre los semáforos.

TIPO	CAMPO	DESCRIPCIÓN
int	val	Valor para **SETVAL**
struct semid_ds __user *	buf	Memoria de datos para **IPC_STAT** e **IPC_SET**
unsigned short __user *	array	Tabla para **GETALL** y **SETALL**
struct seminfo __user *	__buf	Memoria de datos para **IPC_INFO**
void __user*	__pad	Puntero de alineación de la estructura

Estructura seminfo: Estructura que permite conocer los valores límite o actuales del sistema mediante una llamada a semctl. Estas llamadas no se realizan generalmente en modo directo, sino que están reservadas a las utilidades del sistema como el mandato ipcs.

TIPO	CAMPO	DESCRIPCIÓN
int	semmap	No se usa
int	semmni	Número máximo de grupos de semáforos
int	semmns	Número máximo de semáforos
int	semmnu	Número de estructuras undo en el sistema
int	semmsl	Número máximo de semáforos por grupo
int	semmopm	Número máximo de operaciones por cada llamada semop
int	semume	Número máximo de estuctura undo por proceso
int	semusz	Número de grupos de semáforos actualmente definidos
int	semvmx	Valor máximo del contador de semáforos
int	semaem	Número de semáforos actualmente definidos

Estructura sem: Se usa una estructura sem para cada semáforo en el sistema. Esta indica el valor actual del semáforo y el pid de la última operación.

TIPO	CAMPO	DESCRIPCIÓN
int	semval	valor actual del semáforo
int	sempid	pid de la última operación

Estructura sem_queue: Estructura que permite conocer los procesos que están durmiendo (bloqueados).

TIPO	CAMPO	DESCRIPCIÓN
struct sem_queue *	next	próxima entrada en la cola
struct sem_queue **	prev	entrada previa en la cola
struct task_struct *	sleeper	proceso actual
struct sem_undo	undo	operaciones anuladas en caso de terminación
int	pid	identificador del proceso solicitado
int	status	estado de terminación de la operación
struct sem_array *	sma	conjunto de semáforos
int	id	identificador de semáforo interno
struc sembuf *	sops	esturctura de operaciones pendientes
int	nsops	número de operaciones pendientes
int	alter	si se ha modificado el vector

Estructura sem_undo: Estructura utilizada para poder deshacer las acciones en caso de fallo o terminación del proceso.

TIPO	CAMPO	DESCRIPCIÓN
struct sem_undo *	proc_next	próxima entrada de este proceso
struct sem_undo *	id_next	próxima entrada de este conjunto de semáforos
int	semid	identificador del conjunto de semáforos
short *	semadj	vector de ajustes, uno por semáforo

23.3 Llamadas al sistema

- **Semget**: Creación y búsqueda de grupos de semáforos.
- **Semctl:** Control de los semáforos.
- **Semop:** Operaciónes sobre los semáforos.

Semget (creación y búsqueda de grupos *de semáforos)*

Permite la creación de un grupo de semáforos, o bien la recuperación del identificador de un grupo ya existente. El grupo de semáforos esta unificado bajo un identificador común.

PROTOTIPO DE LA FUNCIÓN:

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>

int semget (key_t key, int nsems,int semflg);
```

- key: identificador del conjunto.
- nsems: número de semáforos del conjunto.
- Semflg: Máscara de operación.

CAMPO I:
IPC_CREAT
IPC_EXCL

CAMPO II:
00X00 usuario
000X0 grupo
0000x otro

El núcleo busca dentro de la tabla alguna entrada que se ajuste a la llave suministrada. Si la llave suministrada toma el valor IPC_PRIVATE, el núcleo ocupa la primera entrada que se encuentra libre y ninguna otra llamada “semget” podrá devolver esta entrada hasta que la liberemos.

Si dentro de la máscara de indicadores está activo el bit IPC_CREAT, el núcleo crea una nueva entrada en caso de que no haya ninguna que responda a la llave suministrada. Si la llave suministrada tiene activados IPC_CREAT y IPC_EXL nos dará error si ya existe la clave suministrada.

Retorna un descriptor de semáforo. Si no existe el descriptor y se especifica IPC_CREAT creará uno.

En caso de error retornará un -1 y errno tendrá el código de error. En este caso, la variable errno toma uno de los valores siguientes:

- *EACCES:* Existe un grupo de semáforos para la clave, pero el proceso no tiene todos los derechos necesarios para acceder al grupo.

- *EEXIST:* Existe un grupo de semáforos para la clave, pero están activadas las operaciones IPC_CREAT e IPC_EXCL.

- *EIDRM:* El grupo de semáforos ha sido borrado.

- *ENOENT:* El grupo de semáforos no existe y no está activada la opción IPC_CREAT.

- *ENOMEN*: El semáforo podría crearse pero el sistema no tiene más memoria para almacenar la estructura.

- *ENOSPC*: El semáforo podría crearse pero se sobrepasarían los límites para el número máximo de grupos (SEMMNI) de semáforos o el número de semáforos (SEMMNS).

Semop (operaciones sobre los semáforos)

Para realizar las operaciones sobre los semáforos que hay asociados bajo un identificador (incremento, decremento o espera de nulidad).

PROTOTIPO DE LA FUNCIÓN:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include<sys/sem.h>

int semop(int semid, struct sembuf __user  * sops, unsigned nsops);
```

- semid: Identificador del semáforo (grupo sobre el cual las operaciones tendrán lugar).
- sops: Puntero a un vector de operaciones (tabla de estructuras sembuf que contiene la lista de operaciones).
- nsops: Número de operaciones a realizar en esta llamada (en realidad, esto da el tamaño de la tabla apuntada por el 2º argumento).

Valor estrictamente positivo: Se suma sem_op al valor del semáforo invocado, y tiene como efecto el despertar de todos los procesos que esperan a que este valor aumente a dicho valor.

Valor nulo: Comprobación del valor del semáforo, si es nulo se bloquea el proceso.

Valor estrictamente negativo: Disminuye sem-op el valor del semáforo, si no es posible entonces bloquea el proceso. Si el valor alcanzado es cero entonces despierta a los procesos correspondientes.

Todas las operaciones deben efectuarse de manera atómica. En los casos en que esto no sea posible, o bien el proceso se suspende hasta que sea posible, o bien, si el identificador IPC_NOWAIT está activado (campo sem_flg de la estructura sembuf) por una de las operaciones, la llamada al sistema se interrumpe sin que se realice ninguna operación.

Por cada operación efectuada, el sistema controla si el indicador SEM_UNDO está posicionado. Si es así, crea una estructura para conservar el rastro de esa operación y poder anularla y finalizar la ejecución del proceso.

Si la llamada al sistema se desarrolla correctamente el valor devuelto es 0, si el proceso debe bloquearse devuelve 1, si no es –1 y **errno** toma uno de los valores siguientes:

- *E2BIG:* El argumento nsops es mayor que SEMOPM, y se sobrepasa el número máximo de operaciones autorizadas para una llamada.
- *EACCES:* El proceso que llama no tiene los derechos de acceso a uno de los semáforos especificados en una de las operaciones.
- *EAGAIN:* El indicador IPC_NOWAIT está activado y las operaciones no han podido realizarse inmediatamente.
- *EFAULT:* La dirección especificada por el campo sops no es válida.
- *EFBIG:* El número del semáforo (campo sem_num) es incorrecto para una de las operaciones (negativo o superior al número de semáforos en el grupo).
- *EIDRM:* El semáforo no existe.
- *EINTR:* El proceso ha recibido una señal cuando esperaba el acceso a un semáforo.
- *EINVAL:* O el grupo del semáforo solicitado no existe(argumeno semid), o bien el número de operaciones a realizar es negativo o nulo(argumento nsops).
- *ENOMEN:* El indicador SEM_UNDO está activado y el sistema no puede asignar memoria para almacenar la estructura de anulación.
- ERANGE: El valor añadido al contador del semáforo sobrepasa el valor máximo autorizado para el contador SEMVMX.

Semctl (el control de los semáforos)

Esta llamada al sistema permite la consulta, modificación o supresión de un grupo de semáforos. También permite inicializar los semáforos y obtener información sobre el número de semáforos en espera de aumento

PROTOTIPO DE LA FUNCIÓN:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include<sys/sem.h>

int semctl(int semid, int semnum, int cmd,union semun arg);
```

- semid: Identificador de semáforo válido.
- Semnum: Representa el ordinal o bien el número de semáforos.
- cmd: indica pues la operación a realizar

Según los valores de **cmd** se realizarán pasos diferentes. A continuación se describen los diferentes casos:

- *IPC_INFO/SEM_INFO:* causan una antememoria temporal *seminfo* para que sea inicializada y cargada con los datos estadísticos sin cambiar del semáforo, los elementos semusz y semaem de la estructura *seminfo* son actualizados de acuerdo con el comando dado (IPC_INFO o SEM_INFO). El valor de retorno de las llamadas al sistema es establecido al conjunto máximo de IDs del conjunto de semáforos.

- *SEM_STAT:* causa la inicialización de la antememoria temporal. El spinlock global del semáforo es mantenido mientras se copian los valores sem_otime, sem_ctime, y sem_nsems en la antememoria. Estos datos son entonces copiados al espacio de usuario.

- *IPC_STAT:* Permite obtener las informaciones respecto a un grupo de semáforo. Semmun se ignora, arg es un puntero a la zona que contiene las informaciones. El proceso debe tener derechos de lectura para realizar la operación.

- *IPC_SET:* Permite modificar ciertos valores de la estructura del grupo de semáforos. Los campos modificables son sem_perm.uid, sem_perm.gid y sem_perm.mode. El campo sem_ctime se actualiza automáticamente. El proceso debe ser el creador o bien el propietario del grupo, o bien el superusuario.

- IPC_RMID: Permite destruir el grupo de semáforos. El proceso debe ser el creador o el propietario del grupo, o bien el superusuario. Todos los procesos en espera sobre uno de los semáforos del grupo se despiertan y reciben el error EIDRM.

- *GETPID:* Permite devolver el valor de sempid del semáforo semnum. Se trata del identificador del último proceso que haya realizado la llamada al sistema semop sobre este semáforo. El proceso debe tener derechos de acceso en lectura sobre el grupo.

- *GETNCNT:* Permite devolver el valor de semzcnt del semáforo semnum. Corresponde al número de procesos que esperan que el contador del semáforo llegue a nulo. El proceso debe tener el derecho de acceso en lectura sobre el grupo.

- *GETZCNT:* Para GETZCNT en la caso de no error, el valor de retorno para la llamada al sistema es establecido al número de procesos esperando en el semáforo estando establecido a cero. Este número es calculado por la función count_semzcnt().

- *SETVAL:* Para SETVAL después de validar el nuevo valor del semáforo, las siguientes funciones son realizadas:

    - La cola de deshacer es buscada para cualquier ajuste en este semáforo. Cualquier ajuste que sea encontrado es reinicializado a cero.

    - El valor del semáforo es establecido al valor suministrado.

    - El valor del semáforo sem_ctime para el conjunto del semáforo es actualizado.

    - La función update_queue() es llamada para recorrer la cola de semops (operaciones del semáforo) pendientes y buscar a cualquier tarea que pueda ser completada como resultado de la operación SETALL. Cualquier tarea que no vaya a ser más bloqueada es despertada.

- La operación SETALL copia los valores del semáforo desde el espacio de usuario en una antememoria temporal, y entonces en el conjunto del semáforo. El spinlock es quitado mientras se copian los valores desde el espacio de usuario a la antememoria temporal, y mientras se verifican valores razonables. Si el conjunto del semáforo es pequeño, entonces una pila de antememoria es usada, en otro caso una antememoria más grande es asignado. El spinlock es recuperado y mantenido mientras las siguientes operaciones son realizadas en el conjunto del semáforo:

    - Los valores del semáforo son copiados en el conjunto del semáforo.

    - Los ajustes del semáforon de la cola de deshacer para el conjunto del semáforo son limpiados.

    - El valor sem_ctime para el conjunto de semáforos es establecido.

    - La función update_queue() es llamada para recorrer la cola de semops (operaciones del semáforo) pendientes y mirar por alguna tarea que pueda ser completada como un resultado de la operación SETALL. Cualquier tarea pendiente que no sea más bloqueada es despertada.

- *GETALL/SETALL:* Permite leer/posicionar los valores de los semáforos del grupo semid, arg contiene el resultado/ el valor de la operación: el parámetro semnum se ignora. El proceso que ejecuta la llamada debe tener acceso en lectura/escritura sobre el grupo.

Durante todas estas operaciones, es mantenido el semáforo global del núcleo sem_ids.sem. Dependiendo de la operación realizada su retorno será 0 o un valor positivo. En caso de error retornará un -1 y errno tendrá el código de error.

- *EACCES:* El proceso que llama no tiene derechos de aceso necesarios para realizar la operación.

- *EFAULT*: La dirección especificada por arg.bur o arg.array no es válida.

- *EIDRM:* El grupo de semáforos ha sido borrado.

- *EINVAL:*El valor de semid o de cmd es incorrecto.

- *EPERM:* El proceso no tiene los derechos necesarios para la operación (mandato IPC_SET o IPC_RMID).

- *ERANGE:* cmd tiene el valor SETALL o SETVAL y el valor del contador de uno de los semáforos a modificar es inferior o superior a SEMVMX.

23.4 Ejemplo de uso de semáforos

Explicaremos un ejemplo del uso de semáforos para realizar la sincronización con un recurso (en este caso la pantalla), en C++.
Cada uno de los procesos puede escribir en pantalla pero ninguno puede imprimir mientras el otro está usando el recurso.

int semaforo; **//Valor del semáforo**
struct sembuf P,V; **//Estructura para las operaciones de incremento y decremento**

/*Creamos un semáforo con semget después de obtener una clave mediante ftok. Solo se creará un semáforo para este conjunto*/

```
void crear_semaforo()
{
        key_t key=ftok("/bin/ls",1); //Crea la clave

        //Se crea un semáforo para controlar el acceso exclusivo al recurso compartido
         semaforo = semget(key, 1, IPC_CREAT | 0666);

        //Se inicializa el semáforo a 1.
        semctl(semaforo,0,SETVAL,1);

        //P decrementa el semaforo en 1 para cerrarlo y V lo incrementa para abrirlo.
        //El flag SEM_UNDO hace que si un proceso termina inesperadamente
        //deshace las operaciones que ha realizado sobre el semaforo.
        P.sem_num = 0;
         P.sem_op = -1;
        P.sem_flg = SEM_UNDO;
        V.sem_num = 0;
        V.sem_op = 1;
        V.sem_flg = SEM_UNDO;
}


void Imprimir_por_pantalla ()
{
        pid_t pid;

        /*Creamos un proceso hijo que imprime en pantalla*/
        pid = fork();

        // Comprobamos que ha sido posible crear el proceso hijo
        if (pid != -1)
        {
                if (pid == 0)
```

```
                {
                        sleep(1);
                        // El proceso hijo adquiere el semáforo para acceder al
recurso
                        // compartido (pantalla) antes de imprimir el texto por
pantalla.
                        semop(semaforo,&P,1);
                        cout << “Soy el proceso hijo” <<endl;
                        //Libera el semáforo
                        semop(semaforo,&V,1);
                }
                else
                {
                        sleep(2);
                        // El proceso padre adquiere el semáforo para acceder al
recurso
                        // compartido (pantalla)antes de imprimir el texto por
pantalla.
                        semop(semaforo,&P,1);
                        cout << “Soy el proceso padre” <<endl;
                        //Libera el semáforo
                        semop(semaforo,&V,1);
                }
        }
        else
        {
                cout<<"No se ha podido crear el proceso hijo"<<endl;
                exit(-1);
        }
}

int main ()
{
        crear_semaforo();
        while (true)
        {
                Imprimir_por_pantalla();
        }
}
```

23.5 Código fuente de semáforos

Para la llamada semget()

sys_semget

```
 310asmlinkage long sys_semget(key_t key, int nsems, int semflg)
 311{
 312        struct ipc_namespace *ns;
 313        struct ipc_ops sem_ops;
 314        struct ipc_params sem_params;
 315
 316        ns = current->nsproxy->ipc_ns;
 317
```

/*Comprueba que el numero de semáforos sea válido, es decir, que sea un valor entre 0 y el máximo(SEMMSL)*/

```
 318        if (nsems < 0 || nsems > ns->sc_semmsl)
 319                return -EINVAL;
 320
```

/*Indica las operaciones asociadas con el semáforo
newary – función que crea un nuevo conjunto de semáforos
sem_security – función que comprueba los permisos
sem_more_checks – función que realiza otras comprobaciones extras*/

```
 321        sem_ops.getnew = newary;
 322        sem_ops.associate = sem_security;
 323        sem_ops.more_checks = sem_more_checks;
 324
```

/*Guarda los parámetros del conjunto de semáforos*/

```
 325        sem_params.key = key;
 326        sem_params.flg = semflg;
 327        sem_params.u.nsems = nsems;
 328
```

/*La llamada ipcget es la que realmente crea o busca el conjunto de semáforos.*/

```
  329        return ipcget(ns, &sem_ids(ns), &sem_ops, &sem_params);
 330}
 331
```

ipcget

/*Si KEY = IPC_PRIVATE llama a ipcget_new, en caso contrario llama a ipcget_public. La primera crea un nuevo conjunto de semáforos, mientras que la segunda busca el conjunto de semáforos y si no existe los crea.*/

```
 755int ipcget(struct ipc_namespace *ns, struct ipc_ids *ids,
 756                struct ipc_ops *ops, struct ipc_params *params)
 757{
 758      if (params->key == IPC_PRIVATE)
 759            return ipcget_new(ns, ids, ops, params);
 760      else
 761            return ipcget_public(ns, ids, ops, params);
```

```
 762}
 763
```

ipcget_new

/*Crea un nuevo objeto IPC, llamado por sys_semget() cuando la key es IPC_PRIVATE*/

```
 251static int ipcget_new(struct ipc_namespace *ns, struct ipc_ids *ids,
 252                struct ipc_ops *ops, struct ipc_params *params)
 253{
 254        int err;
 255retry:
```

//reserve de recursos para la asignación mediante la función idr_pre_get

```
 256        err = idr_pre_get(&ids->ipcs_idr, GFP_KERNEL);
 257
 258        if (!err)
 259                return -ENOMEM;
 260
```

//bloqueo para escribir

```
 261        down_write(&ids->rw_mutex);
```

/*Crea un Nuevo conjunto de semáforos mediante getnew, que es un enlace a la función newary*/

```
 262        err = ops->getnew(ns, params);
```

//liberar un bloqueo para escribir

```
 263        up_write(&ids->rw_mutex);
 264
 265        if (err == -EAGAIN)
 266                goto retry;
 267
 268        return err;
 269}
```

ipcget_public

//Obtiene un objeto ipc o crea uno nuevo

```
 315static int ipcget_public(struct ipc_namespace *ns, struct ipc_ids *ids,
 316                struct ipc_ops *ops, struct ipc_params *params)
 317{
 318        struct kern_ipc_perm *ipcp;
 319        int flg = params->flg;
 320        int err;
 321retry:
```

//reserve de recursos para la asignación

```
 322        err = idr_pre_get(&ids->ipcs_idr, GFP_KERNEL);
 323
 324        /*
 325         * Take the lock as a writer since we are potentially going to add
 326         * a new entry + read locks are not "upgradable"
 327         */
```

```
//bloqueo para escritura
 328        down_write(&ids->rw_mutex);


//ipc_findkey encuentra una clave en un conjunto identificador IPC
 329        ipcp = ipc_findkey(ids, params->key);

/*Si la clave no está siendo utilizada y la máscara IPC_CREAT está activa, se crea una nueva entrada
y se llama a getnew(newary)*/
 330        if (ipcp == NULL) {
 331                /* key not used */
 332                if (!(flg & IPC_CREAT))
 333                        err = -ENOENT;
 334                else if (!err)
 335                        err = -ENOMEM;
 336                else

//crea un nuevo conjunto de semaforos mediante newary
 337                        err = ops->getnew(ns, params);

/*Si existe la clave suministrada:
                -    Si está activada IPC_CREAT y IPC_EXCL nos dará un error
                -    Realiza ciertas comprobaciones con more_checks e ipc_check_perms
                -    Elimina el bloqueo de escritura y devuelve el identificador*/
 338        } else {
 339                /* ipc object has been locked by ipc_findkey() */
 340
 341                if (flg & IPC_CREAT && flg & IPC_EXCL)
 342                        err = -EEXIST;
 343                else {
 344                        err = 0;
 345                        if (ops->more_checks)
 346                                err = ops->more_checks(ipcp, params);
 347                        if (!err)
 348                                /*
 349                                 * ipc_check_perms returns the IPC id on
 350                                 * success
 351                                 */
 352                                err = ipc_check_perms(ipcp, ops, params);
 353                }
 354                ipc_unlock(ipcp);
 355        }

//libera el bloqueo para escribir
 356        up_write(&ids->rw_mutex);
 357
 358        if (err == -EAGAIN)
 359                goto retry;
 360
 361        return err;
 362}
 363
 364
```

newary

/*Crea un nuevo identificadro sujeto a la disponibilidad de entradas libres en la tabla que gestiona el núcleo para los semáforos. Crea e inicializa un nuevo conjunto de semáforos*/

```
231static int newary(struct ipc_namespace *ns, struct ipc_params *params)
232{
233       int id;
234       int retval;
235       struct sem_array *sma;
236       int size;

//Obtiene los parametros del conjunto de semaforos
237       key_t key = params->key;
238       int nsems = params->u.nsems;
239       int semflg = params->flg;
240

//comprueba que el numero de semáforos que pide es correcto y si hay disponibilidad
241       if (!nsems)
242             return -EINVAL;
243       if (ns->used_sems + nsems > ns->sc_semmns)
244             return -ENOSPC;
245

//calculamos el tamaño que ocupará y asigna recursos (ipc + espacio rcu)
246       size = sizeof (*sma) + nsems * sizeof (struct sem);
247       sma = ipc_rcu_alloc(size);
248       if (!sma) {
249             return -ENOMEM;
250       }
251       memset (sma, 0, size);
252

//inicializa los parámetros del Nuevo conjunto de semáforos
253       sma->sem_perm.mode = (semflg & S_IRWXUGO);
254       sma->sem_perm.key = key;
255
256       sma->sem_perm.security = NULL;
257       retval = security_sem_alloc(sma);
258       if (retval) {
259             ipc_rcu_putref(sma);
260             return retval;
261       }
262

//obtine el identificador y añade la nueva entrada en el sistema
263       id = ipc_addid(&sem_ids(ns), &sma->sem_perm, ns->sc_semmni);
264       if (id < 0) {
265             security_sem_free(sma);
266             ipc_rcu_putref(sma);
267             return id;
268       }

/*se actualize el valor de la variable que lleva el número de semáforos utilizados*/
269       ns->used_sems += nsems;
270

/*inicializa el resto de los campos de la  estructura sem_array para el nuevo conjunto de semáforos*/
271       sma->sem_perm.id = sem_buildid(id, sma->sem_perm.seq);
272       sma->sem_base = (struct sem *) &sma[1];
273       /* sma->sem_pending = NULL; */
274       sma->sem_pending_last = &sma->sem_pending;
275       /* sma->undo = NULL; */
276       sma->sem_nsems = nsems; //numero de semáforos del grupo
```

```
 277        sma->sem_ctime = get_seconds(); //fecha del ultimo cambio
 278        sem_unlock(sma); //desbloqueo
 279

//retorna el identificador del nuevo conjunto de semáforos
 280        return sma->sem_perm.id;
 281}
 282
 283
```

Para la llamada semctl()

sys_semctl

```
/*Permite la modificación, consulta o suppression de un grupo de semaforos. Estructura de casos
para la llamada semctl, dirige la llamada según el parámetro cmd (operación a realizar)*/
 936asmlinkage long sys_semctl (int semid, int semnum, int cmd, union semun arg)
 937{
 938        int err = -EINVAL;
 939        int version;
 940        struct ipc_namespace *ns;
 941

//comprobamos el valor de semid
 942        if (semid < 0)
 943                return -EINVAL;
 944

//borra la bandera IPC_64
 945        version = ipc_parse_version(&cmd);
 946        ns = current->nsproxy->ipc_ns;
 947

//Según el valor de cmd seleccionaremos diferentes caminos
 948        switch(cmd) {

/*Para el caso de IPC_INFO, SEM_INFO, IPC_STAT o SEM_STAT realizaremos una llamada a la
función semctl_nolock que realizara la tarea correspondiente al cmd seleccionado*/
 949        case IPC_INFO:
 950        case SEM_INFO:
 951        case IPC_STAT:
 952        case SEM_STAT:
 953                err = semctl_nolock(ns, semid, cmd, version, arg);
 954                return err;

/*Para el caso de GETALL, GETVAL, GETPID, GETNCNT, GETZCNT, SETVAL o SETALL
realizaremos una llamada a la función semctl_main que realizara la tarea correspondiente al cmd
seleccionado*/
 955        case GETALL:
 956        case GETVAL:
 957        case GETPID:
 958        case GETNCNT:
 959        case GETZCNT:
 960        case SETVAL:
 961        case SETALL:
 962                err = semctl_main(ns,semid,semnum,cmd,version,arg);
```

```
963             return err;
```

/*Para el caso de IPC_RMID o IPC_SET realizaremos una llamada a la función semctl_down que realizara la tarea correspondiente al cmd seleccionado. Esta llamada se realiza dentro de un cerrojo de escritura que es obtenido por down_write y liberado por up_write*/

```
964     case IPC_RMID:
965     case IPC_SET
966             down_write(&sem_ids(ns).rw_mutex);
967             err = semctl_down(ns,semid,semnum,cmd,version,arg);
968             up_write(&sem_ids(ns).rw_mutex);
969             return err;
970     default:
971             return -EINVAL;
972     }
973}
974
```

semctl_nolock

/*llamada por sys_semctl() para realizar las operaciones:

- IPC_INFO y SEM_INFO: causan una antememoria temporal seminfo para que sea inicializada y cargada con los datos estadísticos sin cambiar el semáforo
- IPC_STAT y SEM_STAT: permite obtener las informaciones respect a un gurpo de semáforos. Semunm se ignora, ag es un puntero a la zona que contiene las informaciones. El proceso debe tener derechos de lectura para realizar la operación*/

```
585static int semctl_nolock(struct ipc_namespace *ns, int semid,
586                       int cmd, int version, union semun arg)
587{
588     int err = -EINVAL;
589     struct sem_array *sma;
590
591     switch(cmd) {
```

//para el caso de IPC_INFO o SEM_INFO

```
592     case IPC_INFO:
593     case SEM_INFO:
594     {
595             struct seminfo seminfo;
596             int max_id;
597
598             err = security_sem_semctl(NULL, cmd);
599             if (err)
600                     return err;
601
```

//para ambos casos IPC_INFO y SEM_INFO inicializa los campos de la estructura seminfo

```
602             memset(&seminfo,0,sizeof(seminfo));
603             seminfo.semmni = ns->sc_semmni;
604             seminfo.semmns = ns->sc_semmns;
605             seminfo.semmsl = ns->sc_semmsl;
606             seminfo.semopm = ns->sc_semopm;
607             seminfo.semvmx = SEMVMX;
608             seminfo.semmnu = SEMMNU;
609             seminfo.semmap = SEMMAP;
610             seminfo.semume = SEMUME;
```

//Obtenemos un cerrojo de lectura

```
611             down_read(&sem_ids(ns).rw_mutex);
```

```
/*Actualizamos el numero de grupos de semaforos defidos y de semaforos actualmente. Si estamos
en el caso de SEM_INFO estos valores son actualizados mediante la información de los semáforos,
en caso contrario son actualizados mediante las macros SEMUSZ y SEMAEM*/
 612            if (cmd == SEM_INFO) {
 613                    seminfo.semusz = sem_ids(ns).in_use;
 614                    seminfo.semaem = ns->used_sems;
 615            } else {
 616                    seminfo.semusz = SEMUSZ;
 617                    seminfo.semaem = SEMAEM;
 618            }
 619            max_id = ipc_get_maxid(&sem_ids(ns));

//Libera el cerrojo de lectura
 620            up_read(&sem_ids(ns).rw_mutex);

//copia los datos al espacio del usuario
 621            if (copy_to_user (arg.__buf, &seminfo, sizeof(struct seminfo)))
 622                    return -EFAULT; //direccion de arg no valida
 623            return (max_id < 0) ? 0: max_id;
 624    }

//En el caso de IPC_STAT o SEM_STAT
 625    case IPC_STAT:
 626    case SEM_STAT:
 627    {
 628            struct semid64_ds tbuf;
 629            int id;
 630

/*La única diferencia sustancial entre SEM_STAT e IPC_STAT es el valor de retorno que tienen.
SEM_STAT devuelve el identificador del conjunto de semáforos, e IPC_STAT devuelve 0*/
 631            if (cmd == SEM_STAT) {
 632                    sma = sem_lock(ns, semid);
 633                    if (IS_ERR(sma))
 634                            return PTR_ERR(sma);
 635                    id = sma->sem_perm.id;
 636            } else {
 637                    sma = sem_lock_check(ns, semid);
 638                    if (IS_ERR(sma))
 639                            return PTR_ERR(sma);
 640                    id = 0;
 641            }
 642
 643            err = -EACCES;

//Realiza el chequeo de permisos
 644            if (ipcperms (&sma->sem_perm, S_IRUGO))
 645                    goto out_unlock;
 646
 647            err = security_sem_semctl(sma, cmd);
 648            if (err)
 649                    goto out_unlock;
 650
 651            memset(&tbuf, 0, sizeof(tbuf));
 652

/* los valores sem_otime, sem_ctime, y sem_nems son copiados en una pila de antememoria*/
 653            kernel_to_ipc64_perm(&sma->sem_perm, &tbuf.sem_perm);
 654            tbuf.sem_otime  = sma->sem_otime;
 655            tbuf.sem_ctime  = sma->sem_ctime;
```

```
 656            tbuf.sem_nsems  = sma->sem_nsems;
 657            sem_unlock(sma); //libera cerrojo

/*Los datos son copiados al espacio de usuario después de liberar el cerrojo (spinlock)*/
 658            if (copy_semid_to_user (arg.buf, &tbuf, version))
 659                    return -EFAULT;

//Retorna el identificador obtenido anteriormente
 660            return id;
 661    }
 662    default:
 663            return -EINVAL;
 664    }
 665    return err;
 666out_unlock:
 667    sem_unlock(sma); //liberamos el cerrojo
 668    return err;
 669}
 670
```

semctl_main

/*Llamado pro sys_semctl() para relaizar muchas de las funciones soportadas. Anteriormente a realizar alguna de las siguientes operaciones, semctl_main() cierra el spinlock global del semáforo y valida la ID del conjunto de semáforos y los permisos. El spinlock es liberado antes de retornar*/

```
 671static int semctl_main(struct ipc_namespace *ns, int semid, int semnum,
 672            int cmd, int version, union semun arg)
 673{
 674    struct sem_array *sma;
 675    struct sem* curr;
 676    int err;
 677    ushort fast_sem_io[SEMMSL_FAST];
 678    ushort* sem_io = fast_sem_io;
 679    int nsems;
 680

//Obtiene el cerrojo
 681    sma = sem_lock_check(ns, semid);
 682    if (IS_ERR(sma))
 683            return PTR_ERR(sma);
 684

//obtiene el número de semáforos del grupo
 685    nsems = sma->sem_nsems;
 686
 687    err = -EACCES;
//comprueba los permisos del usuario
 688    if (ipcperms (&sma->sem_perm, (cmd==SETVAL||cmd==SETALL)?S_IWUGO:S_IRUGO))
 689            goto out_unlock;
 690
 691    err = security_sem_semctl(sma, cmd);
 692    if (err)
 693            goto out_unlock;
 694
 695    err = -EACCES;
```

/*Según el valor de cmd se realizará una operación u otra. Explicaremos cada una de ellas más adelante*/

```
696     switch (cmd) {
```

/*En el caso de getall carga los valores del semáforos en una antermemoria temporal del núcleo y entonces los copia al espacio de usuario. Es decir devuelve el valor de todos los semáforos del grupo (semval)*/

```
697     case GETALL:
698     {
699             ushort __user *array = arg.array;
700             int i;
701
```

/*Si el número de semáforos el mayor que la pila utilizada, entonces obtendremos más memoria para poder guardar la información*/

```
702             if(nsems > SEMMSL_FAST) {
703                     ipc_rcu_getref(sma);
704                     sem_unlock(sma);
705
706                     sem_io = ipc_alloc(sizeof(ushort)*nsems);
707                     if(sem_io == NULL) {
708                             ipc_lock_by_ptr(&sma->sem_perm);
709                             ipc_rcu_putref(sma);
710                             sem_unlock(sma);
711                             return -ENOMEM;
712                     }
713
714                     ipc_lock_by_ptr(&sma->sem_perm);
715                     ipc_rcu_putref(sma);
716                     if (sma->sem_perm.deleted) {
717                             sem_unlock(sma);
718                             err = -EIDRM;
719                             goto out_free;
720                     }
721             }
722
```

/*Guarda en el vector sem_io todos los valores de cada semáforo del conjunto de semáforos (solo guarda el valor de semval)*/

```
723             for (i = 0; i < sma->sem_nsems; i++)
724                     sem_io[i] = sma->sem_base[i].semval;
725             sem_unlock(sma);
726             err = 0;
```

/*Copia los valores obtenidos en el espacio de usuario*/

```
727             if(copy_to_user(array, sem_io, nsems*sizeof(ushort)))
728                     err = -EFAULT;
729             goto out_free;
730     }
```

/*En el caso de SETALL copia los valores del semáforo desde el espacio de usuario en una antememoria temporal. El spinlock es quitado mientras se copian los datos. Es decir, modifica todos los valores actuales de los diferentes semáforos del conjunto de semáforos*/

```
731     case SETALL:
732     {
733             int i;
734             struct sem_undo *un;
735
736             ipc_rcu_getref(sma);
737             sem_unlock(sma);
738
```

/*Si el número de semáforos el mayor que la pila utilizada, entonces obtendremos más memoria para poder guardar la información*/

```
739             if(nsems > SEMMSL_FAST) {
740                     sem_io = ipc_alloc(sizeof(ushort)*nsems);
741                     if(sem_io == NULL) {
742                             ipc_lock_by_ptr(&sma->sem_perm);
743                             ipc_rcu_putref(sma);
744                             sem_unlock(sma);
745                             return -ENOMEM;
746                     }
747             }
748
```

/*Copia los datos desde el espacio de usuario ha la antememoria temporal*/

```
749             if (copy_from_user (sem_io, arg.array, nsems*sizeof(ushort))) {
750                     ipc_lock_by_ptr(&sma->sem_perm);
751                     ipc_rcu_putref(sma);
752                     sem_unlock(sma);
753                     err = -EFAULT;
754                     goto out_free;
755             }
756
```

/*Comprueba que el nuevo valor de los semáforos no excede del valor máximo válido para los semáforos*/

```
757             for (i = 0; i < nsems; i++) {
758                     if (sem_io[i] > SEMVMX) {
759                             ipc_lock_by_ptr(&sma->sem_perm);
760                             ipc_rcu_putref(sma);
761                             sem_unlock(sma);
762                             err = -ERANGE;
763                             goto out_free;
764                     }
765             }
766             ipc_lock_by_ptr(&sma->sem_perm);
767             ipc_rcu_putref(sma);
768             if (sma->sem_perm.deleted) {
769                     sem_unlock(sma);
770                     err = -EIDRM;
771                     goto out_free;
772             }
773
```

/*Los nuevos valores son copiados en el conjunto de semáforos*/

```
774             for (i = 0; i < nsems; i++)
775                     sma->sem_base[i].semval = sem_io[i];
```

/*Los ajustes del semáforo de la cola deshacer para el conjunto del semáforos son limpiados*/

```
776             for (un = sma->undo; un; un = un->id_next)
777                     for (i = 0; i < nsems; i++)
778                             un->semadj[i] = 0;
```

/*Actualiza el valor de sem_ctime (fecha del último cambio por semctl)*/

```
779             sma->sem_ctime = get_seconds();
780             /* maybe some queued-up processes were waiting for this */
```

/*Llama a update_queue para recorrer la cola de semops (operaciones del semáforo) pendientes y mirar si alguna tarea puede ser completaa. Cualquier tarea pendiente que no sea más bloqueada es despertada*/

```
781             update_queue(sma);
```

```
 782                err = 0;
 783                goto out_unlock;
 784        }

//Para el resto de casos
 785        /* GETVAL, GETPID, GETNCTN, GETZCNT, SETVAL: fall-through */
 786        }
 787        err = -EINVAL;

/*Comprueba que el valor de sumnum (posición en el gurpo de semáforos) no se vaya fuera de los
rangos permitidos*/
 788        if(semnum < 0 || semnum >= nsems)
 789                goto out_unlock;
 790

//obtiene el semáforo indicado por la variable semnum
 791        curr = &sma->sem_base[semnum];
 792
 793        switch (cmd) {

/*Para GETVAL devuelve el valor del semáforo*/
 794        case GETVAL:
 795                err = curr->semval;
 796                goto out_unlock;

/*Para GETPID devuelve el valor del pid asociado con la última operación del semáforo*/
 797        case GETPID:
 798                err = curr->sempid;
 799                goto out_unlock;

/*Para GETNCNT devuelve el número de procesos esperando en el semáforo siendo menor que
cero*/
 800        case GETNCNT:
 801                err = count_semncnt(sma,semnum);
 802                goto out_unlock;

/*Para GETZCNT devuelve el número de procesos esperando en el semáforo estando establecido a
cero*/
 803        case GETZCNT:
 804                err = count_semzcnt(sma,semnum);
 805                goto out_unlock;

/*Para SETVAL establece un nuevo valor para el semáforo*/
 806        case SETVAL:
 807        {
//val es el nuevo valor que deberá tener el semáforo
 808                int val = arg.val;
 809                struct sem_undo *un;
 810                err = -ERANGE;

//Comprueba que el Nuevo valor no esté fuera de los rangos válidos
 811                if (val > SEMVMX || val < 0)
 812                        goto out_unlock;
 813

/*La cola de deshacer es buscada par cualquier ajuste en este semáforo. Cualqueir ajuste que sea
encontrado es reinicializado a cero*/
 814                for (un = sma->undo; un; un = un->id_next)
 815                        un->semadj[semnum] = 0;
```

```
//Se guarda el nuevo valor
 816            curr->semval = val;

/*Se obtiene el pid de la última operación y se actualiza sem_ctime (fecha del último cambio)*/
 817            curr->sempid = task_tgid_vnr(current);
 818            sma->sem_ctime = get_seconds();
 819            /* maybe some queued-up processes were waiting for this */

/*Llama a update_queue para recorrer la cola de semops (operaciones del semáforo) pendientes y
mirar si alguna tarea puede ser completaa. Cualquier tarea pendiente que no sea más bloqueada es
despertada*/

 820            update_queue(sma);
 821            err = 0;
 822            goto out_unlock;
 823    }
 824    }
 825out_unlock:
 826    sem_unlock(sma); //Libera el cerrojo
 827out_free:

//Libera la memoria utilizada
 828    if(sem_io != fast_sem_io)
 829            ipc_free(sem_io, sizeof(ushort)*nsems);
 830    return err;
 831}
 832
```

semctl_down

/*Suministra las operaciones IPC_RMID y IPC_SET de la llamada al sistema sys_semctl. La ID del conjunto de semáforos y los permisos de acceso son verificadas en ambas operaciones, y en ambos casos, el spinlock global del semáforo es mantenido a lo largo de la operación

- IPC_RMID: llama a freeary para borrar el conjunto de semáforos
- IPC_SET: actualiza los elementos uid, gid, mode y ctime del conjunto de semáforos*/

```
 873static int semctl_down(struct ipc_namespace *ns, int semid, int semnum,
 874            int cmd, int version, union semun arg)
 875{
 876    struct sem_array *sma;
 877    int err;
 878    struct sem_setbuf uninitialized_var(setbuf);
 879    struct kern_ipc_perm *ipcp;
 880
 881    if(cmd == IPC_SET) {
 882            if(copy_semid_from_user (&setbuf, arg.buf, version))
 883                    return -EFAULT;
 884    }
 885    sma = sem_lock_check_down(ns, semid);
 886    if (IS_ERR(sma))
 887            return PTR_ERR(sma);
 888
 889    ipcp = &sma->sem_perm;
 890
 891    err = audit_ipc_obj(ipcp);
 892    if (err)
 893            goto out_unlock;
 894
 895    if (cmd == IPC_SET) {
 896            err = audit_ipc_set_perm(0, setbuf.uid, setbuf.gid, setbuf.mode);
```

```
897             if (err)
898                     goto out_unlock;
899     }

/*Comrpobamos si el proceso tienen o no derechos para realizar la operación*/
900     if (current->euid != ipcp->cuid &&
901         current->euid != ipcp->uid && !capable(CAP_SYS_ADMIN)) {
902             err=-EPERM;
903             goto out_unlock;
904     }
905
906     err = security_sem_semctl(sma, cmd);
907     if (err)
908             goto out_unlock;
909
910     switch(cmd){

/*Para el caso de IPC_RMID llama a la función freeary que eliminará el conjunto de semáforos*/
911     case IPC_RMID:
912             freeary(ns, ipcp);
913             err = 0;
914             break;

/*Para el caso de IPC_SET actualizamos los diferentes elementos: uid, gid, mode y sem_ctime*/
915     case IPC_SET:
916             ipcp->uid = setbuf.uid;
917             ipcp->gid = setbuf.gid;
918             ipcp->mode = (ipcp->mode & ~S_IRWXUGO)
919                             | (setbuf.mode & S_IRWXUGO);
920             sma->sem_ctime = get_seconds();
921             sem_unlock(sma); //Libera cerrojo
922             err = 0;
923             break;
924     default:
925             sem_unlock(sma);
926             err = -EINVAL;
927             break;
928     }
929     return err;
930
931out_unlock:
932     sem_unlock(sma);
933     return err;
934}
935
```

freeary

```
/*freeary se encarga de liberar un conjunto de semáforos*/
523/* Free a semaphore set. freeary() is called with sem_ids.rw_mutex locked
524 * as a writer and the spinlock for this semaphore set hold. sem_ids.rw_mutex
525 * remains locked on exit.
526 */
527static void freeary(struct ipc_namespace *ns, struct kern_ipc_perm *ipcp)
528{
529     struct sem_undo *un;
530     struct sem_queue *q;
531     struct sem_array *sma = container_of(ipcp, struct sem_array, sem_perm);
532
```

```
533     /* Invalidate the existing undo structures for this semaphore set.
534      * (They will be freed without any further action in exit_sem()
535      * or during the next semop.)
536      */
```

/*Invalida las estructuras undo (operaciones anuladas en caso de terminación) que existen para este conjunto de semáforos. Estas serán liberadas en exit_sem() o durante la siguiente llamada a semop*/

```
537     for (un = sma->undo; un; un = un->id_next)
538             un->semid = -1;
539
```

/*Despierta todos los procesos pendientes y los deja como que han fallado con EIDRM (el grupo de semáforos ha sido borrado)*/

```
540     /* Wake up all pending processes and let them fail with EIDRM. */
541     q = sma->sem_pending;
542     while(q) {
543             struct sem_queue *n;
544             /* lazy remove_from_queue: we are killing the whole queue */
545             q->prev = NULL;
546             n = q->next;
547             q->status = IN_WAKEUP;
548             wake_up_process(q->sleeper); //despierta al proceso
549             smp_wmb();
550             q->status = -EIDRM;     /* hands-off q */
551             q = n;
552     }
553
```

/*Borra el conjunto de semáforos del vector de ID*/

```
554     /* Remove the semaphore set from the IDR */
555     sem_rmid(ns, sma);
556     sem_unlock(sma); //Libera el cerrojo
557
```

//Actualiza el número de semáforos que están siendo usados en el sistema

```
558     ns->used_sems -= sma->sem_nsems;
559     security_sem_free(sma);
560     ipc_rcu_putref(sma);
561}
562
```

update_queue

/*Recorre la cola de semops pendientes para un conjunto de semáforos y llama a try_atomic_semop() para determinar que secuencias de las operaciones de los semáforos serán realizadas*/

```
431/* Go through the pending queue for the indicated semaphore
432 * looking for tasks that can be completed.
433 */
434static void update_queue (struct sem_array * sma)
435{
436     int error;
437     struct sem_queue * q;
438
```

//obtiene el puntera a la cola de tareas pendientes

```
439     q = sma->sem_pending;
```

//mientras existan procesos pendientes

```
 440        while(q) {

//llama a try_atomic_semop que intentará relizar la tarea
 441                error = try_atomic_semop(sma, q->sops, q->nsops,
 442                                         q->undo, q->pid);
 443
 444                /* Does q->sleeper still need to sleep? */

//si no es necesario bloquear el proceso
 445                if (error <= 0) {
 446                        struct sem_queue *n;

//quitamos el proceso de la cola de dormidos y los despertamos
 447                        remove_from_queue(sma,q);
 448                        q->status = IN_WAKEUP;
 449                        /*
 450                         * Continue scanning. The next operation
 451                         * that must be checked depends on the type of the
 452                         * completed operation:
 453                         * - if the operation modified the array, then
 454                         *   restart from the head of the queue and
 455                         *   check for threads that might be waiting
 456                         *   for semaphore values to become 0.
 457                         * - if the operation didn't modify the array,
 458                         *   then just continue.
 459                         */

/*si se modifoco el array comenzamos desde la cabecera, sino continuamos con el siguiente elemento
de la cola*/
 460                        if (q->alter)
 461                                n = sma->sem_pending;
 462                        else
 463
                                n = q->next;
//despierta el proceso
 464                        wake_up_process(q->sleeper);
 465                        /* hands-off: q will disappear immediately after
 466                         * writing q->status.
 467                         */
 468                        smp_wmb();
 469                        q->status = error;
 470                        q = n;
 471                } else {

/*continua con el siguiente de la cola en caso de que el proceso actual no pueda ser despertado*/
 472                        q = q->next;
 473                }
 474        }
 475}
 476
```

remove_from_queue

```
/*Función que extrae de la cola de procesos dormidos , donde la lista es FIFO*/
 353static inline void remove_from_queue (struct sem_array * sma,
 354                                       struct sem_queue * q)
 355{
 356        *(q->prev) = q->next;
 357        if (q->next)
```

```
 358            q->next->prev = q->prev;
 359        else /* sma->sem_pending_last == &q->next */
 360            sma->sem_pending_last = q->prev;
 361        q->prev = NULL; //lo marca como eliminado
 362}
 363
```

count_semncnt

/*count_semncnt cuenta el numero de procesos esperando por el valor del semaforo para que sea menor que cero (esperando por semval=0).*/

```
 486static int count_semncnt (struct sem_array * sma, ushort semnum)
 487{
 488        int semncnt;
 489        struct sem_queue * q;
 490
```

//Inicializa semncnt (contador) a cero

```
 491        semncnt = 0;
```

//Recorremos la cola de procesos dormidos en el sistema

```
 492        for (q = sma->sem_pending; q; q = q->next) {
 493            struct sembuf * sops = q->sops;
 494            int nsops = q->nsops;
 495            int i;
```

/*bucle de las operaciones a realizar. Aumenta semncnt si el número de semáforo coincide que el parámetro pasado, si la operación es de decremento y no está activada el flag IPC_NOWAIT*/

```
 496            for (i = 0; i < nsops; i++)
 497                if (sops[i].sem_num == semnum
 498                    && (sops[i].sem_op < 0)
 499                    && !(sops[i].sem_flg & IPC_NOWAIT))
 500                    semncnt++;
 501        }
 502        return semncnt;
 503}
```

count_semzcnt

/*Cuenta el numero e procesos esperando por el valor del semaforo para que sea cero (esperando con semval /=0)*/

```
 504static int count_semzcnt (struct sem_array * sma, ushort semnum)
 505{
 506        int semzcnt;
 507        struct sem_queue * q;
 508
```

//Inicializa semzcnt (contador) a cero

```
 509        semzcnt = 0;
```

//Recorremos la cola de procesos dormidos en el sistema

```
 510        for (q = sma->sem_pending; q; q = q->next) {
 511            struct sembuf * sops = q->sops;
 512            int nsops = q->nsops;
 513            int i;
```

/*bucle de las operaciones a realizar. Aumenta semncnt si el número de semáforo coincide que el parámetro pasado, si la operación igual a cero y no está activada el flag IPC_NOWAIT*/

```
514             for (i = 0; i < nsops; i++)
515                     if (sops[i].sem_num == semnum
516                        && (sops[i].sem_op == 0)
517                        && !(sops[i].sem_flg & IPC_NOWAIT))
518                           semzcnt++;
519     }
520     return semzcnt;
521}
522
```

Para la llamada semop()

sys_semop

/*Se utiliza esta llamada para realizar las operaciones sobre los semáforos que hay asociados bajo un identificador (incremento, decremento o espera de nulidad.
Lo único que hace esta llamda es llamar a sys_semtimedop*/

```
1254asmlinkage long sys_semop (int semid, struct sembuf __user *tsops, unsigned nsops)
1255{
1256     return sys_semtimedop(semid, tsops, nsops, NULL);
1257}
1258
```

sys_semtimedop

```
1090asmlinkage long sys_semtimedop(int semid, struct sembuf __user *tsops,
1091                     unsigned nsops, const struct timespec __user *timeout)
1092{
1093     int error = -EINVAL;
1094     struct sem_array *sma;
1095     struct sembuf fast_sops[SEMOPM_FAST];
1096     struct sembuf* sops = fast_sops, *sop;
1097     struct sem_undo *un;
1098     int undos = 0, alter = 0, max;
1099     struct sem_queue queue;
1100     unsigned long jiffies_left = 0;
1101     struct ipc_namespace *ns;
1102
1103     ns = current->nsproxy->ipc_ns;
1104
```

/*Valida los parámetros de entrada de la llamada*/

```
1105     if (nsops < 1 || semid < 0)
1106          return -EINVAL;
1107     if (nsops > ns->sc_semopm)
1108          return -E2BIG;
1109     if(nsops > SEMOPM_FAST) {
1110          sops = kmalloc(sizeof(*sops)*nsops,GFP_KERNEL);
1111          if(sops==NULL)
1112               return -ENOMEM;
1113     }
```

/*Copia los datos desde el espacio de usuario a una antememoria temporal*/

```
1114        if (copy_from_user (sops, tsops, nsops * sizeof(*tsops))) {
1115                error=-EFAULT;
1116                goto out_free;
1117        }
1118        if (timeout) {
1119                struct timespec _timeout;
1120                if (copy_from_user(&_timeout, timeout, sizeof(*timeout))) {
1121                        error = -EFAULT;
1122                        goto out_free;
1123                }
1124                if (_timeout.tv_sec < 0 || _timeout.tv_nsec < 0 ||
1125                        _timeout.tv_nsec >= 1000000000L) {
1126                        error = -EINVAL;
1127                        goto out_free;
1128                }
1129                jiffies_left = timespec_to_jiffies(&_timeout);
1130        }
1131        max = 0;
```

/*En este bucle se optiene: el númro máximo de semáforos utilizados en los grupos, establece undos a 1 si la etiqueta SEM_UNDO (permite deshacer) está activada y pone alter a 1 si alguna de las operaciones puede modificar el valor de los semáforos*/

```
1132        for (sop = sops; sop < sops + nsops; sop++) {
1133                if (sop->sem_num >= max)
1134                        max = sop->sem_num;
1135                if (sop->sem_flg & SEM_UNDO)
1136                        undos = 1;
1137                if (sop->sem_op != 0)
1138                        alter = 1;
1139        }
1140
1141retry_undos:
```

/*Si la etiqueta SEM_UNDO está activada llama a find_undo para obtener una estructura sem_undo para el conjunto de semáforos*/

```
1142        if (undos) {
1143                un = find_undo(ns, semid);
1144                if (IS_ERR(un)) {
1145                        error = PTR_ERR(un);
1146                        goto out_free;
1147                }
1148        } else
1149                un = NULL;
1150
```

//obtiene el cerrojo

```
1151        sma = sem_lock_check(ns, semid);
1152        if (IS_ERR(sma)) {
1153                error = PTR_ERR(sma);
1154                goto out_free;
1155        }
1156
1157        /*
1158         * semid identifiers are not unique - find_undo may have
1159         * allocated an undo structure, it was invalidated by an RMID
1160         * and now a new array with received the same id. Check and retry.
1161         */
1162        if (un && un->semid == -1) {
1163                sem_unlock(sma);
1164                goto retry_undos;
```

```
1165        }
1166        error = -EFBIG;

/*Chequea que el número máximo de semáforos del grupo esté dentro de un intervalo válido*/
1167        if (max >= sma->sem_nsems)
1168                goto out_unlock_free;
1169
1170        error = -EACCES;

/*Crequa los permiso*/
1171        if (ipcperms(&sma->sem_perm, alter ? S_IWUGO : S_IRUGO))
1172                goto out_unlock_free;
1173
1174        error = security_sem_semop(sma, sops, nsops, alter);
1175        if (error)
1176                goto out_unlock_free;
1177

/*Llama a la función try_atomic_semop para realizar la tarea. En caso de éxito esta devolverá 0*/
1178        error = try_atomic_semop (sma, sops, nsops, un, task_tgid_vnr(current));
1179        if (error <= 0) { //si el proceso no se ha bloqueado
/*Si el proceso ha finalizado con éxito y la variable alter está activada,  realizamos una llamada a
update_queue para despertar a todos los procesos que puedan ser desbloqueados según la situación
actual del conjunto de semáforos*/
1180                if (alter && error == 0)
1181                        update_queue (sma);
1182                goto out_unlock_free;
1183        }
1184
1185        /* We need to sleep on this operation, so we put the current
1186         * task into the pending queue and go to sleep.
1187         */
1188

/*En caso de que el proceso deba bloquearse (try_atmic_semop retorna un 1)se rellena la estructura
sem_queue con la información actual del conjunto de semáforos y se introduce en la cola*/
1189        queue.sma = sma;
1190        queue.sops = sops;
1191        queue.nsops = nsops;
1192        queue.undo = un;
1193        queue.pid = task_tgid_vnr(current);
1194        queue.id = semid;
1195        queue.alter = alter;

/*Si el proceso puede modificar el conjunto de semáforos se introduce por el final, en caso contrario
se introduce al principio de la cola de procesos dormidos*/
1196        if (alter)
1197                append_to_queue(sma ,&queue);
1198        else
1199                prepend_to_queue(sma ,&queue);
1200
1201        queue.status = -EINTR;
1202        queue.sleeper = current;
1203        current->state = TASK_INTERRUPTIBLE;
1204        sem_unlock(sma); //Libera el cerrojo
1205
1206        if (timeout)
1207                jiffies_left = schedule_timeout(jiffies_left);
1208        else
```

```
/*Llama al planificador (schedule) para que entre otro proceso, y el actual pase a dormido*/
1209            schedule();
1210
1211    error = queue.status;
1212    while(unlikely(error == IN_WAKEUP)) {
1213            cpu_relax();
1214            error = queue.status;
1215    }
1216
1217    if (error != -EINTR) {
1218            /* fast path: update_queue already obtained all requested
1219             * resources */
1220            goto out_free;
1221    }
1222

//Una vez a sido despertado el proceso obtine el cerrojo nuevamente
1223    sma = sem_lock(ns, semid);
1224    if (IS_ERR(sma)) {
1225            BUG_ON(queue.prev != NULL);
1226            error = -EIDRM;
1227            goto out_free;
1228    }
1229
1230    /*
1231     * If queue.status != -EINTR we are woken up by another process
1232     */
1233    error = queue.status;
1234    if (error != -EINTR) {
1235            goto out_unlock_free;
1236    }
1237
1238    /*
1239     * If an interrupt occurred we have to clean up the queue
1240     */
1241    if (timeout && jiffies_left == 0)
1242            error = -EAGAIN;

/*Elimina el proceso de la cola de procesos dormidos*/
1243    remove_from_queue(sma,&queue);
1244    goto out_unlock_free;
1245
1246out_unlock_free:

/*Libera el cerrojo y la memoria utilizada*/
1247    sem_unlock(sma);
1248out_free:
1249    if(sops != fast_sops)
1250            kfree(sops);
1251    return error;
1252}
1253
```

try_atomic_semop

/*Funcion que determina si una secuencia de operaciones del semáforo tendrán éxito o no, intentado realizar cada una de las operaciones.
Retorna 0 si tiene éxito, 1 si necesita dormir sino retorna un código de error*/

```
369static int try_atomic_semop (struct sem_array * sma, struct sembuf * sops,
370                              int nsops, struct sem_undo *un, int pid)
371{
372       int result, sem_op;
373       struct sembuf *sop;
374       struct sem * curr;
375
```

/*Para cada una de las operaciones obtine el semáoro en el que hay que realizar la tarea, la operación a realizar y el valor del semáforo actualmente*/

```
376       for (sop = sops; sop < sops + nsops; sop++) {
377             curr = sma->sem_base + sop->sem_num;
378             sem_op = sop->sem_op;
379             result = curr->semval;
380
381             if (!sem_op && result) //no hay operacion pero hay semaforo
382                   goto would_block;
383
```

/*Realizamos la operación en la variable local result. Si tiene éxito será actualizada*/

```
384             result += sem_op;
```

/*Si el Nuevo resultado del semáforo sería menor que cero, significa que el semáforo debe bloquearse*/

```
385             if (result < 0)
386                   goto would_block;
```

/*se comprueba que el valor máximo del semáforo no ha sido superado*/

```
387             if (result > SEMVMX)
388                   goto out_of_range;
```

/*Comprobamos si SEM_UNDO está posicionado para cada operación efectuada, para crear estructura y conservar el rastro de operaciones hechas*/

```
389             if (sop->sem_flg & SEM_UNDO) {
390                   int undo = un->semadj[sop->sem_num] - sem_op;
391                   /*
392                    *      Exceeding the undo range is an error.
393                    */
394                   if (undo < (-SEMAEM - 1) || undo > SEMAEM)
395                         goto out_of_range;
396             }
```

//Actualizamos el valor del semáforo

```
397             curr->semval = result;
398       }
399
400       sop--;
```

//asignar el identificador de cada proceso

```
401       while (sop >= sops) {
402             sma->sem_base[sop->sem_num].sempid = pid;
403             if (sop->sem_flg & SEM_UNDO)
404                   un->semadj[sop->sem_num] -= sop->sem_op;
405             sop--;
406       }
407
408       sma->sem_otime = get_seconds();
409       return 0;
410
```

```
//entramos en este punto cuando el contador de semáforos no es valido
 411out_of_range:
 412        result = -ERANGE;
 413        goto undo;
 414

/*si IPC_NOWAIT – falla la operación, en caso contrario devuelve 1 (debe bloquearse)*/
 415would_block:
 416        if (sop->sem_flg & IPC_NOWAIT)
 417                result = -EAGAIN;
 418        else
 419                result = 1;
 420

//deshacemos las operaciones hechas
 421undo:
 422        sop--;
 423        while (sop >= sops) {
 424                sma->sem_base[sop->sem_num].semval -= sop->sem_op;
 425                sop--;
 426        }
 427
 428        return result;
 429}
 430
```