Memoria compartida, semáforos y colas de mensajes

Dr. Roberto Gómez Cárdenas
ITESM-CEM, Dpto. Ciencias Computacionales

15 de abril de 2002

El presente documento representa un borrador de lo que será el capítulo de un libro. Acaba de ser terminado y debido a la urgencia de tiempo no se presto mucha atención a la forma, pero el fondo es muy bueno. Cualquier aclaración, comentario o correción se agradecera que se le notifique al autor vía personal o a través del correo electrónico, (`rogomez@campus.cem.itesm.mx`*).*

1 Introducción

Un sistema computacional grande, en especial un sistema operativo, cuenta con una gran cantidad de procesos ejecutándose al mismo tiempo. El encargar todo el trabajo a un solo proceso provocaría un desempeño muy pobre por parte del sistema operativo.

Una de las características del sistema operativo Unix es la de multiprogramación (varios programas, propiedad de un mismo usuario, ejecutandose al mismo tiempo) y la de tiempo compartido (diferentes usuarios utilizando los recursos del sistema al mismo tiempo). Esta característica trae como consecuencia que existan una gran cantidad de procesos interactuando al mismo tiempo. Ahora bien, es necesario poder sincronizar todos estos procesos a través de herramientas de comunicación, para que se pongan de acuerdo en el caso de que varios de ellos deseen acceder un mismo recurso al mismo tiempo.

En este capítulo trataremos las llamadas de sistema relacionadas con lo que se conoce como IPC (Inter Process Communication), es decir memoria compartida, semáforos y colas de mensajes.

2 El sistema de comunicación IPC

El sistema Unix proporciona un conjunto de instrucciones para comunicación entre procesos. Este conjunto se encuentra definido dentro lo que se conoce como IPC (InterProcess Communication). Estas instrucciones proporcionan un "canal" de comunicación entre dos o más procesos. Existen tres tipos de canales IPC:

1. La memoria compartida
2. Los semáforos
3. Las colas de mensajes

Cada uno de ellos proporciona un medio de sincronía/comunicación para procesos. Para este fin, cada uno cuenta con un conjunto de llamadas propias y archivos de encabezado. En esta sección solo se tratarán aspectos generales y comunes de las dos primeras. Es posible combinar varios de ellos en una misma aplicación.

En la tabla de abajo, (figura 1), se encuentra una lista de los archivos de encabezado y de las prinicales llamadas de cada tipo de IPC.

	Colas Mensajes	**Semáforos**	**Memoria Compartida**
Archivo encabezado	$< sys/msg.h >$	$< sys/sem.h >$	$< sys/smh.h >$
Llamadas para crear o abrir	*msgget*	*semget*	*shmget*
Llamadas operaciones control	*msgctl*	*semctl*	*shmctl*
Llamadas operaciones IPC	*msgsnd, msgrcv*	*semop*	*shmat, shmdt*

Figura 1: *Tabla características IPC*

Las llamadas *...get* que aparecen en el recuadro de la figura 1 son usadas para crear y/o acceder un canal de comunicación IPC; las llamadas *...ctl* permiten modificar cierto tipo de características, éstas últimas son particulares a cada uno de los canales.

Unix mantiene una estructura de información para cada canal IPC, parecida a la que se mantiene cuando se crea un archivo. Esta información se encuentra definida dentro del archivo de encabezado `/usr/include/sys/ipch.h`.

Los principales campos son:

```
struct ipc_perm {
   ushort  uid;   /* identificador del usuario */
   ushort  gid;   /* identificador del grupo    */
   ushort  cuid;  /* identificador del creador del creador */
   ushort  cgid;  /* identificador del grupo de creadores */
   ushort  seq;   /* numero secuencia en la cola */
   key_t   key;   /* llave */
}
```

2.1 Creación canales IPC

Para que dos procesos se puedan comunicar o sincronizar es necesario crear un canal de comunicación entre ambos. Las llamadas *...get* son las responsables de crear o abrir (si este ya existe) un canal IPC (ver figura 2). Las tres llamadas necesitan un valor de llave de tipo *key_t*. Dicha llave puede ser definida por el usuario (a través de un macro `#define`) o puede crearse a partir de la llamada `ftok()`. La figura 2 muestra todos los pasos necesarios para la creación de un canal utilizando la llamada `ftok()`.

La sintaxis de `ftok()` es la siguiente:

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

key_t ftok(char *pathname, char proj);
```

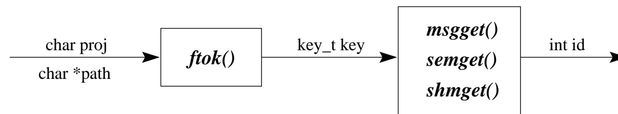


Figura 2: *Los diferentes pasos para la creación de un canal IPC*

Dentro del archivo `<sys/types.h>` se encuentra definido el tipo *key_t* como un entero de 32 bits. Esto es muy útil en aplicaciones de tipo cliente/servidor ya que ambos pueden estar de acuerdo en un *pathname* y, en combinación con algún valor de *proj*, crear una misma llave. Si se necesitan varios canales (i.e. varios clientes y un solo servidor) se puede usar el mismo *pathname* con diferentes valores de *proj* por canal. Si el *pathname* no existe, o no es accesible, la llamada regresa un -1.

A continuación se presenta un ejemplo de un programa que crea una llave:

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

main()
{
   key_t llave;
   char *archivo;
   char idkey;

   archivo = "/home/dic/rogomez/Apuntes/Sist-Oper/llave.cle"
   idkey =  'b';
   llave=ftok(archivo, idkey);
   if (llave < 0) {
     fprintf(stderr, "Error en la generacion de la llave \n");
     exit(1);
   }
   printf("Llave creada con exito \n");
}
```

2.1.1 Los parámetros de las llamadas get

La llave no es el único parámetro para la creación de un canal IPC. Las llamadas *...get* requieren un parámetro de tipo bandera. Esta bandera especifica los nueve bits menos significativos del canal IPC que se creó o al que se hace referencia.

Los diferentes valores que puede tomar la bandera y las consecuencias de estos son:

- `IPC_PRIVATE`: garantiza que se va a crear un canal IPC único, ninguna combinación en los parámetros de `ftok()` generan una llave para dicho canal.
- `IPC_CREAT` crea una nueva entrada para la llave especificada si ésta no existe. Si ya existe la entrada es regresada.
- `IPC_CREAT` e `IPC_EXCL`: crean una entrada para la llave especificada, solo si la llave no existía de antemano. Si existía se produce un error, (bit `IPC_EXCL` no garantiza al proceso acceso exlcusivo al canal).
- `0`: el canal ya fue creado previamente por algún otro proceso; el proceso que realiza la llamada solo utilizará dicho canal.

La tabla 3 presenta un condensado de todos los posibles valores de la bandera y sus consecuencias.

Argumento Bandera	**La llave no existe**	**Llave si existe**
ninguna bandera en especial	error, $errno = ENOENT$	OK
IPC_CREAT	OK, crea una nueva entrada	OK
IPC_CREAT | IPC_EXCL	OK, crea una nueva entrada	error, $errno = EEXIST$

Figura 3: *Tabla de argumentos para la creación de canales*

NOTA: ejemplos concretos del uso de las diferentes llamadas de creación de canales serán tratados en secciones posteriores.

2.2 Verificando estado de los canales, el comando *ipcs*

Muchas veces es necesario saber si los canales fueron cerrados después de haber sido utilizados o si estan activos dentro del sistema. Unix proporciona un comando que permite verificar el estado de los canales IPC. Este comando es *ipcs* y presenta la sintaxis siguiente:

```
ipcs [-abcmopqsr]  [ -C corefile]  [-N namelist]
```

Si no se proporciona ninguna opción el comando despliega información resumida acerca de las colas de mensajes, la memoria compartida y los semáforos que se encuentran activos en el sistema.

Un ejemplo de salida de dicho comando se presenta a continuación:

```
rogomez@armagnac:10>ipcs
IPC status from armagnac as of Wed Apr  9 18:11:47 1997
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
Semaphores:
rogomez@armagnac:10>prodcons
rogomez@armagnac:11>ipcs
IPC status from armagnac as of Wed Apr  9 18:12:42 1997
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m      0 0x000004d5 --rw-rw-rw-  rogomez       40
Semaphores:
s      0 0x000004d2 --ra-ra-ra-  rogomez       40
s      1 0x000004d3 --ra-ra-ra-  rogomez       40
s      2 0x000004d4 --ra-ra-ra-  rogomez       40
rogomez@armagnac:11>
```

En el primer caso ningún proceso esta utilizando los canales; en el segundo los procesos generados por el programa `prodcon` utilizan un canal de memoria compartida y tres de semáforos.

En ciertas ocasiones cuando se intenta crear un canal, la llamada puede regresar un error indicando que es no es posible dicha creación. Esto puede ser provocado por el hecho de que los canales IPC no se encuentran disponibles o activos dentro del sistema. El comando `ipc` puede ser usado para asegurarse si los canales IPC están disponibles o no. El ejemplo de abajo muestra como se puede usar el comando para llevar a cabo lo anterior, donde el programa ejecutable `usocanal` es un programa ejecutable que utiliza canales IPC.

```
rogomez@armagnac:5>usocanal
   Error en pline: FALLO el shmget
rogomez@armganac:6>ipcs
IPC status from armagnac as of Wed Apr 29 21:12:42 1997
Message Queues facility not in system
Shared Memory facility not in system
Semaphore facility not in system
       :
       :
rogomez@armganac:181>ipcs
IPC status from armagnac as of Wed Apr 10 1:48:35 1997
Message Queues facility not in system
Shared Memory
Semaphore facility not in system
rogomez@armganac:182>
```

En el primero de los ejemplos ninguno de los tres canales de comunicación esta activo; en el segundo solo el de la memoria compartida se encuentra disponible y lista para usarse.

2.3 Cerrando canales IPC: comando *ipcrm*

Algunas veces el canal queda abierto y es necesario borrarlo o cerrarlo desde la línea de comandos. Usando el comando *ipcrm* es posible borrar un canal IPC que quedó abierto después de la ejecución de un proceso. La sintaxis del comando es:

```
ipcrm [-m shmid]  [ -q msqid ] [ -s semid]  [-M shmkey] [-Q msqkey] [-S semkey]
```

según la opción proporcionada, se puede borrar un semáforo (`-s -S`), memoria compartida (`-m -M`) o una cola de mensajes (`-q -Q`), ya sea a través de su llave o su identificador, (los cuales se pueden obtener a partir del comando *ipcs*).

Un ejemplo del uso de este comando en combinación con el anterior, se muestra a continuación:

```
rogomez@armagnac:32>ipcs
IPC status from armagnac as of Wed Apr  9 18:12:42 1997
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m      0 0x000004d5 --rw-rw-rw-  rogomez       40
Semaphores:
s      0 0x000004d2 --ra-ra-ra-  rogomez       40
s      1 0x000004d3 --ra-ra-ra-  rogomez       40
s      2 0x000004d4 --ra-ra-ra-  rogomez       40
rogomez@armagnac:33>ipcrm -m1 -s1
rogomez@armagnac:34>ipcs
IPC status from armagnac as of Wed Apr  9 18:12:42 1997
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m      0 0x000004d5 --rw-rw-rw-  rogomez       40
Semaphores:
s      0 0x000004d2 --ra-ra-ra-  rogomez       40
s      2 0x000004d4 --ra-ra-ra-  rogomez       40
rogomez@armagnac:35>
```

En este ejemplo la salida del comando *icpcs* reporta la existencia de cuatro canales: uno de memoria y tres de semáforos. Después se borra el de semáforos, y como la segunda ejecución de ipcs lo reporta, solo quedan los canales relacionados con memoria compartida.

Este comando es útil a nivel línea de comandos pero a nivel código cada uno de los canales cuenta con llamadas para cerrar y/o borrar los canales. Estas llamadas serán tratadas posteriormente.

3 La memoria compartida

Cuando un proceso es creado, este cuenta con su propia área de variables dentro de su contexto. Si este proceso crea otro proceso (usando la llamada `fork()`) el contexto del padre es duplicado y heredado al hijo. Se presenta un problema si los dos procesos requieren compartir información a través de una variable o una estructura. Una forma en que los dos procesos podrían compartir información es vía un archivo, para lo cual uno tendrá que crearlo y otro que accederlo. Los canales IPC nos proporcionan una segunda alternativa para que dos procesos compartan información, ésta se conoce como *memoria compartida*. Al igual que en el caso de los archivos, un proceso tendrá que crear el canal y otro(s) que accederlo. En esta sección se verán todos los aspectos relacionados con el manejo de memoria compartida[1].

Toda memoria compartida cuenta con un identificador (llamémosle `shmid`) el cual es un entero único positivo creado a partir de la llamada de sistema `shmget()`. Este identificador tiene una estructura de datos asociada con él. La estructura de datos se conoce como `shmid_ds` y tiene la siguiente declaración:

```
struct shmid_ds {
    struct ipc_perm  shm_perm;    /* estructura de operacion permiso */
    int              shm_segsz;   /* longitud del segmento en bytes */
    struct region   *shm_reg;     /* ptr a la estructura de la region */
    char             pad [4] ;    /* para compatibilidad de intercambio (swap)*/
    pid_t            shm_lpid ;   /* pid proceso que hizo la ultima operacion */
    pid_t            shm_cpid ;   /* pid del creador del segmento*/
    ushort           shm_nattch ; /* numero de "ataduras" actuales */
    ushort           shm_cnattch  /* usado unicamente por shminfo */
    time_t           shm_atime    /* tiempo de la ultima "atadura" */
    time_t           shm_dtime ;  /* tiempo de la utilma "desatadura" */
    time_t           shm_ctime ;  /* tiempo del utltimo cambio  */
}
```

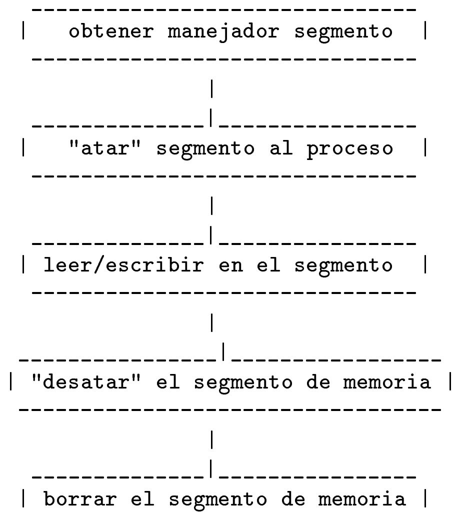
Dentro de la estructura `shmid_ds` se hace referencia a la estructura `ipc_perm` que es una estructura que especifica los permisos de operación de memoria compartida. Misma que cuenta con los siguientes campos:

```
struct ipc_perm {
    uid_t   cuid;   /* id del usuario creador del segmento */
    gid_t   cgid;   /* id del grupo del usario que creo el segmento */
    uid_t   uid;    /* id del usuario */
    gid_t   gid;    /* id del grupo */
    mode_t  mode;   /* permiso de escritura/lectura */
    ulong   seg;    /* secuencia de uso # */
    key_t   key;    /* llave */
}
```

[1]A lo largo del presente documento se utilizará el termino memoria compartida o segmento de memoria indistintamente

3.1 Llamadas para el manejo de memoria compartida

Los pasos para poder crear/acceder un segmento de memoria, se ilustran en el esquema de abajo:



En lo que sigue de la sección nos concretaremos a explicar las diferentes llamadas de sistema que son usadas para llevar a cabo los pasos anteriores.

Todas la llamadas de sistema van a necesitar de los siguientes archivos de encabezado:

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

3.1.1 Obtener un manejador de memoria compartida

Lo primero es obtener un manejador, lo cual se realiza a través de la llamada `shmget()`, cuya sintaxis es:

```
int shmget(key_t key, size_t size, int shmflg);
```

donde:

`key` es la llave que identifica a la memoria compartida, es posible definirla a través de una macro estilo `#define` o crearla a partir de la función `ftok()`.

`size` representa el tamaño de la memoria compartida en bytes, generalmente obtenido aplicando la función *sizeof*() a la información almacenada en el segmento.

`shmflg` bandera que indica si se va a crear la memoria (valor `ipc_create` y permisos) o si se se va a obtener un manejador de memoria ya existente (valor cero).

La llamada regresa un manejador de tipo entero, que va a servir de referencia para las operaciones sobre la memoria compartida.

3.1.2 Atar el segmento a una estructura del proceso

Para poder leer/escribir en el segmento de memoria es necesario *atar* el segmento a una estructura de datos. Esta "atadura" tiene por objeto el darle forma al segmento de memoria, ya que dependiendo de la estructura a la que se ate el segmento será la interpretación de la información ahí almacenada. La llamada de sistema para efectuar lo anterior es `shmat()` que presenta la siguiente sintáxis:

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

donde:

`shmid` es el manejador que identifica al segmento de memoria, obtenido a partir de la llamada `shmget()`.

`addr` especifica una dirección de mapeo, generalmente se pasa un valor de cero para que el sistema se ocupe de todo.

`shmflg` bandera para el manejo de la dirección de mapeo, al igual que ésta generalmente tiene un valor de cero.

A continuación se presenta un ejemplo de estas dos últimas llamadas.

```
main()
{
                    :
                    :

    struct info {
        char nombre[30];
        int  edad;
    };
    struct info *ptr;
    int shmid;

    shmid = shmget(key,sizeof(struct info), IPC_CREAT | 0666);
    ptr = (struct info *) shmat(shmid, 0,0);
                    :
                    :
}
```

Como puede constatarse el segmento de memoria es creado con permisos `0666` y después es atado. Un detalle a notar es el casting que se hace del regreso de la llamada de `shmat()`, el cual permite que la variable `ptr` referencie a la información dentro de la memoria compartida sin problemas.

Se necesita poner especial atención a los permisos. Si estos valores no son bien asignados, otros procesos no podran acceder al segmento de memoria. También podría darse el caso de que, procesos pertenecientes a otros usuarios no podrían acceder al segmento de memoria.

3.1.3 Leer y escribir en el segmento

Una vez que el segmento esta atado a la estructura de datos del proceso es posible leer/escribir en él. Para escribir se asigna el valor deseado a alguna de las variables de la estructura atada. Para poder leer y/o utilizar los valores del segmento basta con usar la variable del campo correspondiente.

Continuando con el código del ejemplo anterior, se asignará el valor de 33 al campo `edad` del segmento y después se desplegará en patalla dicho valor. El código es:

```
main()
{
                :
    struct info {
        char nombre[30];
        int  edad;
    };
                :

    ptr = (struct info) shmat(shmid, 0,0);
                :

    ptr->edad = 33;
    printf("El nombre es: %s ",ptr->nombre);
}
```

3.1.4 Desatar la memoria

Se recomienda que al final de la aplicación se desate la memoria para evitar incongruencias en la aplicación. Sin embargo esto es una recomendación no una condición.

La llamada de sistema que permite realizar lo anterior es *shmdt*() y su sintaxis es:

```
int shmdt(char *shmaddr);
```

donde `shmaddr` es la estructura de datos a la que se ató el segmento. Un ejemplo de uso se presenta a continuación:

```
main()
{
                          :

    shmid = shmget(key,sizeof(struct info), IPC_CREAT | 0666);
    ptr = (struct info) shmat(shmid, 0,0);

                          :

         <codigo uso memoria compartida>

                          :

    shmdt(ptr);
}
```

3.1.5 Borrar el segmento y otras operaciones

Una vez utilizado el segmento y desatado es conveniente (más no forzozo) borrarlo para evitar confusiones. El borrado se efectua a partir de la llamada *shmctl*(), a partir de la cual se pueden realizar otras operaciones. La sintaxis de esta llamada es:

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

donde

shmid es el manejador que identifica al segmento de memoria obtenido a partir de la llamada *shmget*.

cmd especifica la operación a realizar sobre el segmento de memoria

buf estructura que sirve de parámetro de entrada y/o salida para algunas operaciones

Los posibles valores que puede tomar `cmd`, y por ende las posibles operaciones, son:

- IPC_STAT asigna cada uno de los valores de los campos de la estructura de datos asociada con `shmid`, en la estructura apuntada por el parámetro `buf`.
- IPC_SET asigna el valor de los campos `shm_perm, shm_perm.gid` y `shm_perm.mode` (permisos) de la estructura `buf`.
- IPC_RMID borra el identificador del segmento de memoria del sistema especificado por `shmid` y destruye tanto el segmento como la estructura de datos asociada a él, (`buf` tiene un valor de cero). Este comando sólo puede ser ejecutado por el susperusuario o por un proceso cuyo id de usuario sea igual al valor de `shm_perm.guid` o `shm_perm.gid` de la estructura asociada con el segmento.
- SHM_LOCK bloquea el segmento de memoria especificado por el identificado `shmid` (el valor de `buf` es de cero); este comando sólo puede ser ejecutado por el superusuario.
- SHM_UNLOCK desbloquea el segmento de memoria especificado por el identificador `shmid` (el valor de `buf` es de cero); este comando solo puede ser ejecutado por el superusuario.

3.2 Ejemplo uso memoria compartida

En esta sección se presenta un ejemplo de memoria compartida tomado de [Bro92]. El cual consiste en dos procesos que utlizan el mismo segmento de memoria. El segmento de memoria consta de dos campos: `c` y `n`, donde el primero es un caracter y `n` es un número.

Uno de los dos procesos es el encargado de crear el segmento de memoria, de inicializarlo e imprimir su contenido. Este proceso debe desplegar `n` veces el caracter `c`, lo cual lo debe hacer mientras el valor de `n` sea diferente de cero. El otro proceso es utilizado para modificar el contenido del segmento de memoria.

Para evitar confusiones y posibles errores, tanto las llaves de acceso y la estructura que definen al archivo se encuentran definidas en el archivo de encabezado *line.h*

```
/* Estructura de datos utilisada en los programas pline.c
   y cline.c, utilisados como ejemplo de memoria compartida */

struct info{
    char c;
    int length;
};

#define KEY   ((key_t) (1243))
#define SEGSIZE  sizeof (struct info)
```

El programa que se encarga de crear la memoria compartida, de inicializar su contenido con los valores a y 10 y de imprimir su contenido se llama `pline.c`. El proceso continuará imprimiendo el contenido hasta que el valor de `n` sea 0.

```
/* Progama ejemplo de memoria c ompratida en UNIX */
/* Nombre: pline.c    */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "line.h"

main()
{

   int i, id;
   struct info *ctrl;
   struct shmid_ds shmbuf;

   /* creando el segmento de memoria */
   id = shmget(KEY, SEGSIZE, IPC_CREAT | 0666);
   if (id <0){
      printf(" Error en pline: FALLO EL shmget \n");
      exit(1);
   }

   /* atando el segmento de memoria */
   ctrl = (struct info*) shmat(id,0,0);
   if (ctrl <= (struct info *) (0)){
      printf(" Error en pline: fallo shmat \n");
      exit(2);
   }

   /* asignando valores iniciales al segmento de memoria */
   ctrl->c = 'a';
   ctrl->n = 10;

   /* imprimiendo contenido memoria cada 4 segundos, mientras que n > 0 */
   for (i=0; i < ctrl->n; i++)
       putchar(ctrl->c);
       putchar('\n');
       sleep(4);
   }
}
```

El programa encargado de realizar los cambios en la memoria compartida tiene el nombre de `cline.c`.

```
/* Programa para ejemplificar el uso de memoria compartida en UNIX */
/* Nombre: cline.c    */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "line.h"
```

```
main(argc,argv)
  int argc;
  char *argv[];
{

   int id;
   struct info *ctrl;
   struct shmid_ds shmbuf;

   /* verificando parametros de entrada */
   if (argc != 3){
      printf("Error en uso : cline <car> <tama~o> \n");
      exit(3);
   }

   /* obtendieno un manejador del segmento de memoria */
   id = shmget(KEY, SEGSIZE, 0);
   if (id <0){
      printf(" Error en cline: FALLO EL shmget \n");
      exit(1);
   }

   /* atando el segmento de memoria */
   ctrl = (struct info*) shmat(id,0,0);
   if (ctrl <= (struct info *) (0)){
      printf(" Error en cline: fallo shmat \n");
      exit(2);
   }

   /* copiado de la linea de comando a la memoria en comun */
   ctrl->c = argv[1][0];
   ctrl->length = atoi(argv[2]);
}
```

3.3 Un último detalle sobre memoria compartida

Se debe tener cuidado al desatar una memoria o al borrarla. ¿Qué puede ocurrir en el caso de que se borre o desate la memoria cuando se esta utilizando? La figura de abajo presenta la salida de un programa que realiza lo anterior:

```
rogomez@armagnac:32>imp
aaaaaaaaaa
aaaaaaaaaa
xx
xx
Se desato el segmento de memoria
Segmentation fault (core dumped)
rogomez@armagnac:33>\rm core
rogomez@armagnac:34>!cc
cc imp.c -o imp
rogomez@armagnac:35>imp2
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
```

```
zz
zz
Se borro el segmento de memoria
zz
zz
zz
rogomez@armagnac:36>
```

El programa `imp` realiza lo mismo que `pline` del ejemplo anterior, salvo que cuando le llega una señal específica desata el segmento de memoria (ejecuta un `shmdt`). El segundo programa, `imp2`, realiza lo mismo, solo que cuando la señal llega borra la memoria compartida (`shmctl(id,IPC_RMID,0)`).

Dejamos como ejercicio que el lector obtenga sus conclusiones de lo anterior.

3.4 Memoria compartida y memoria dinámica

Una de las caracteristicas de la

En algunas aplicaciones es necesario co

Para crear una lista con memoria compartida se tomo el mismo principio de las listas ligadas: una zona de memoria de datos y un apuntador a la siguiente zona.

En vez de usar un apuntador, usamos una llave, la cual abre la siguiente zona de memoria compartida. Si la llave es 0, significa que este es el ultimo segmento de memoria compartida. Para comenzar la cadena necesitamos una llave inicial, para que todos los procesos tengan acceso al principio de la lista. Este numero fue definido como 4000. Despues para cada segmento que se crea, se le suma uno a la llave anterior y se pone en el apuntador next del ultimo segmento. De esta forma las llaves nos quedan secuenciales (4000,4001,4002,etc).

Finalmente, para que el proceso fin pudiera saber los pid de los procesos a matar, se creo un arreglo de n casillas donde n es el numero de procesos. Como "fin" tampoco sabe cuantos procesos son, al principio del arreglo se puso byte con el numero de procesos, de esta forma "fin" ya sabe de que tamano tiene que pedir el segmento de memoria compartida y cuantas casillas ver, por lo que vuelve a pedir el mismo segmento de memoria, pero ahora con el tamano correcto.

A continuacion se muestra la ejecucion del programa. Se muestra la ejecucion de los programas, aunque la parte de ./crealista 15 se quedo sin sacar nada en la pantalla hasta que se ejecuto fin, sin embargo aqui se pone todo junto para llevar los bloques:

4 Los semáforos

Los semáforos fueron inventados por Dijkstra [Dij68] como un mecanismo para la solución de programas concurrentes que utilizan recursos en común (i.e. una solución al problema de la exclusión mutua). El semáforo es un tipo de variable que sirve de *bandera* para sincronizar el acceso a un recurso por parte de varios procesos.

Al igual que los semáforos utilizados en las calles, estos van a decidir qué proceso tiene derecho a usar un recurso compartido por varios de ellos. Antes de ver las diferentes llamadas de sistema relacionadas con los semáforos, se analizarán las características principales de los semáforos en un rango más teórico.

4.1 Los semáforos de Dijkstra

Una variable de tipo semáforo cuenta sólo con dos tipos de operaciones: $P(S)$ y $V(S)$. Las definiciones de esta funciones se encuentra en la figura 4

<table>
<tr>
<td>
P(S)
begin
    if (S> 0) then
        S := S-1;
    else
        <bloquear al proceso>
end
</td>
<td>
V(S)
begin
    if (alguien espera) then
        < desbloquear un proceso >
    else
        S := S+1;
end
</td>
</tr>
</table>

Figura 4: *Definición funciones P(S) y V(S)*

La operación $P(S)$ verifica si el valor del semáforo es mayor que cero, en cuyo caso lo disminuirá a una unidad; en caso contrario bloqueará el proceso hasta que este valor se vea modificado. Este valor puede ser modificado, o el proceso puede ser desbloqueado a partir de la operación $V(S)$. Esta operación verifica si no existe ningún proceso bloqueado debido al semáforo S, en caso positivo desbloquea al proceso, de no existir ningún proceso bloqueado aumenta en uno el valor del semáforo.

Los semáforos presentan dos características esenciales que lo convierten en una herramienta muy útil para los problemas de concurrencia. Éstas son:

1. *Desbloqueo de procesos:* es el semáforo mismo, no el proceso que efectúa una operación $V(S)$, el que se encarga de desbloquear a los procesos bloqueados. Es importante remarcar que si existe más de un proceso bloqueado, todos los procesos se desbloquean. Al desbloquearse los procesos pasan a un estado listo y es el administrador de procesos quien decide cual de los procesos puede ejecutarse, (dependiendo de la implementación LIFO, FIFO, etc). Los procesos no ejecutados regresan al estado bloqueado.
2. *Atomicidad:* muchos de los problemas de exclusión mutua están relacionados con el quantum de los procesos. Cuando un proceso estaba listo para usar el recurso su quantum expiraba y otro proceso podía usarlo a pesar de que el primero lo tenia asignado. Para evitar este tipo de problemas se garantiza que al iniciar una operación con un semáforo, ningún otro proceso puede tener acceso al semáforo hasta que la operación termine o que el proceso se bloquee. En este tiempo ningún otro proceso puede simultáneamente modificar el valor del semáforo.

Los semáforos tienen muchas aplicaciones, dependiendo de ésta pueden ser clasificados como semáforos contadores, semáforos binarios o semáforos sincronizadores.

4.1.1 Semáforos contadores

Los semáforos contadores son útiles cuando un recurso se toma de un conjunto de recursos idénticos. El semáforo es inicializado con el número de recursos existentes y cada vez que se toma un recurso, o se regresa uno al conjunto, se aplica una de las operaciones de semáforos. Las operaciones son:

- $P(S)$ decrementa en S el valor de 1, indicando que un recurso ha sido suprimido del conjunto. Si el valor de S es cero, entonces no hay más recursos y el proceso se bloquea, hasta que se cuente con recursos.

- $V(S)$ incrementa el valor de S en 1, indicando que un recurso ha sido regresado al conjunto. Si un proceso esperaba por un recurso, éste se desbloqueará.

El código de abajo presenta un ejemplo de dos procesos (`consume` y `produce`) que se ejecutan concurrentemente. Uno de ellos produce recursos mientras que el otro los consume.

```
S: semaforo

/* suprimiendo recursos */            /* regresando recursos */
procedure cosume()                    procedure produce;
begin                                 begin
   while(TRUE) do                         while(TRUE) do
      P(S);                                  construye_recurso;
      utiliza_recurso;                       V(S);
   enddo                                  enddo;
end                                   end

/* llamando a uno y a otro */
begin
   S = 10;
   parbegin
      consume;
      usando_el_recurso;
      produce;
   parend;
end
```

4.1.2 Los semáforos binarios

Los semáforos binarios se caracterizan porque sólo pueden tomar dos valores, cero o uno. Generalmente se incializan con un valor de 1. Son usados por dos o más procesos para garantizar que solamente uno puede tener acceso a un recurso o ejecutar un código determinado (sección crítica).

La sincronización se realiza de la siguiente forma, antes de entrar en sección crítica un proceso ejecuta un $P(S)$ y antes de salir de ella ejecuta un $V(S)$.

En el código siguiente se utiliza una variable de tipo semáforo, llamada `entrar` e inicializada en 1, para ejemplificar el uso de los semáforos binarios:

```
entrar = 1;
while (TRUE)  do
        P(entrar);
                Seccion Critica: acceso recurso
                ejecucion codigo especial
        V(entrar)
enddo
```

4.1.3 Semáforos sincronizadores

Los semáforos sincronizadores son una solución a diferentes problemas de sincronización, en particular cuando el código de un proceso necesita ser ejecutado antes o después del código de otro proceso.

Para ejemplificar el uso de estos semáforos se consideran dos procesos P_1 y P_2 que se encuentran corriendo concurrentemente. El proceso P_1 cuenta con un enunciado S1 y el proceso P_2 cuenta con un enunciado S2. El objetivo es que el enunciado S2 sea ejecutado después de que S1 haya terminado. Para lograr esto se utilizará un semáforo sincronizador, denominado `sincro` e inicializado en cero. La solución se presenta a continuación:

```
                          sincro = 0;
P1:  --------------              P2: ---------------
       --------------                    ---------------
             S1                             P(sincro)
        V(sincro)                              S2
       --------------                    ---------------
       --------------                    ---------------
```

Como puede constatarse los semáforos tienen muchas aplicaciones. Sin embargo este un punto de vista meramente teórico. A continuación se analizarán las características de los semáforos y las llamadas de sistema que Unix ofrece para el manejo de semáforos.

4.2 Los semáforos de Unix

Unix ofrece su propia versión de los semáforos de Dijkstra, los cuales presentan características propias. Los semáforos en Unix son multi-valuados, no binarios y es posible crear un arreglo de semáforos con una simple llamada. Esto último permite especificar una simple operación en un sólo semáforo, o definir un conjunto de operaciones sobre un conjunto de semáforos de forma atómica.

Unix libera todos los candados (lock) que un proceso retenga en sus semáforos, cuando éste aborte, (llamada `exit()`). Los semáforos, de forma parecida a los archivos, tienen dueños, modos de acceso y estampillas de tiempo.

¿Para qué puede servir toda estas características complejas en los semáforos? Consideremos el siguiente escenario: en un sistema un proceso hace un petición de m recursos de un conjunto de n recursos $R_1, R_2, ...R_n$, $(m < n)$. Si cada recurso, R_i, esta protegido por un semáforo S_i, la adquisión de m recursos del mismo tipo requiere de m operaciones $P()$ sucesivas del semáforo S: $P(S_1)$ $P(S_2)$ ... $P(S_m)$.

Bajo el mismo escenario anterior, asumamos que un proceso P_i requiere los recursos R_1, R_2 y R_3 y otro proceso, P_2 requiere los mismos recursos. P_i realiza las siguientes operaciones: $P(S_1), P(S_2), P(S_3)$ y P_2 realiza $P(S_3), P(S_2), P(S_1)$. Ahora bien, si después de completar la operación $P(S_1)$ por parte de P_1 el quantum se le termina y le corresponde a P_2 realizar sus peticiones. Cuando P_2 intenta ejecutar $P(S_1)$ quedara bloqueado, y en el momento que P_1 intente acceder a R_2 a través de la operación $P(S_2)$ también quedará bloqueada. EL resultado final es que P_1 estará bloqueado por un recurso que tiene asignado P_2 y viceversa. El sistema se bloqueará.

Sin embargo, en este esquema la atomicidad no esta garantizada. Esto puede provocar un problema de bloqueo mutuo (deadlock). Para evitar lo anterior, Unix proporciona la posibilidad de realizar un conjunto de operaciones sobre un conjunto de semáforos, si no es posible completar una operación ninguna de las operaciones se llevará a cabo.

4.3 Las estructuras de datos de los semáforos

Un semáforo, al igual que la memoria compartida, cuenta con un identificador único y con datos asociados a él. Los semáforos de Unix cuentan con tres estructuras de datos asociadas a ellos.

Recordemos que es posible contar con un conjunto de semáforos y manejarlos como una entidad única.

La estructura `semid_ds` contiene la información que define al conjunto de semáforos como una entidad. Los campos de la estructura `sem` definen a cada semáforo del conjunto. Por último la estructura `sembuf` permite definir un conjunto de operaciones a ser aplicadas sobre un conjunto de semáforos.

4.3.1 La estructura *semid_ds*

La estructura proporciona la información relacionada con el conjunto de semáforos como unidad. Esta compuesta por los siguientes campos:

```
struct ipc_perm   sem_perm;   /* permiso de operacion */
struct sem       *sem_base;   /* apuntador hacia el primer semaforo del conjunto*/
ushort            sem_nsems;  /* numero de semaforos en el conjunto * /
time_t            sem_otime;  /* la ultima operacion de tiempo */
time_t            sem_ctime;  /* el ultimo cambio de tiempo */
```

El campo `sem_perm` es una estructura `ipc_perm` que especifica el permiso de operación de un semáforo. La estructura `ipc_perm` respeta la siguiente declaración:

```
uid_t   uid;    /* id del usuario */
gid_t   gid;    /* id del grupo */
uid_t   cuid:   /* id del usuario creador */
gid_t   cgid:   /* id del grupo creador */
mode_t  mode;   /* permiso de r/a */
ulong   seq;    /* slot uso de la secuencia numerica */
key_t   key;    /* llave */
```

El campo `sem_nsems` es el número de semáforos que conforman el conjunto. Cada semáforo es referenciado por valores, no negativos, corridos secuencialmente desde 0 hasta el valor del campo `sem_nsems` menos uno. El campo `sem_otime` almacena la fecha de la última operación *semop*, mientras que `sem_ctime` guarda la fecha de la última operación *semctl* que cambió el valor de algún campo de la estructura anterior.

4.3.2 La estructura *sem*

Los principales atributos de un semáforo se encuentran almacenados dentro de una estructura de base de datos llamada `sem` que contiene los siguientes campos:

```
ushort   semval;  /* valor de semaforo */
pid_t    sempid;  /* pid de la ultima operacion */
ushort   semncnt; /* cantidad procesos que esperan que: semval > val */
ushort   semzcnt; /* cantidad de procesos que esperan que:  semval = 0 */
```

El campo `semval` es un entero no negativo que representa el valor actual del semáforo. El segundo campo, `sempid`, es el identificador del último proceso que realizó una operación de semáforo en este semáforo. La variable `semncnt` representa el número de procesos bloqueados esperando que el valor de `semval` aumente su valor, mientras que `semzcnt` son los procesos bloqueados esperando que el valor de `semval` sea igual a cero.

4.3.3 La estructura *sembuf*

Unix permite definir un conjunto de operaciones sobre uno o varios semáforos, La forma más simple de implementar lo anterior es reunir las operaciones a aplicar en una sola estructura de datos, en particular en un arreglo. Cualquier operación a realizar sobre un semáforo se define a través de los campos de la estructura `sembuf`.

Los campos de la estructura `sembuf`, que definen una operación, son los siguientes:

```
short sem_num; /* numero de semaforo */
short sem_op;  /* operacion semaforo */
shor sem_flg;  /* bandera operacion */
```

El primer campo, `sem_num` precisa sobre cual semáforo del conjunto se va a realizar la operación. La operación se almacena en el campo `sem_op`, mientras que `sem_flg` define la forma en que se va a realizar dicha operación.

La forma en que se utiliza dicha estructura se describe en la sección 4.4.2.

4.3.4 Relaciones entre las estructuras *semid_ds* y *sem*

Como puede apreciarse existe una relación entre los diferentes campos de las estructuras que se describieron en las secciones anteriores. El alterar el valor de un campo de una estructura, puede afectar el valor de unos de los campos de otra estructura.

La figura 5 presenta un diagrama que muestra los primeras estructuras mencionadas en esta sección. Se trata de un conjunto de tres semáforos.

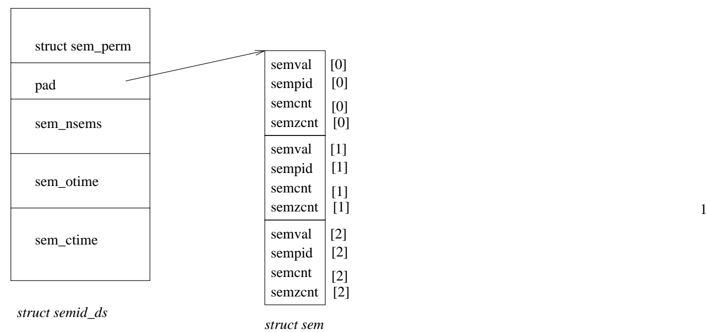


Figura 5: *Relación estructuras semid_ds y sem de los semáforos*

4.4 Las llamadas de sistema relacionadas con los semáforos

Existen diferentes llamadas de sistema para el uso de semáforos. El orden en el uso y los parámetros de cada una de ellas varían de acuerdo a la aplicación. Las principales llamadas se pueden clasificar como:

- creación de semáforos

- operaciones sobre semáforos
- modificación de los atributos de los semáforos

Todas las llamadas de sistema que se van a tratar en esta sección necesitan los siguientes archivos de encabezado:

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

4.4.1 Creación de semáforos

La llamada de sistema utilizada para la creación de un semáforo es `semget()` cuya sintaxis es la siguiente:

```
int semget(key, nsems, bandera);
```

donde:

`key` llave de identificación del semáforo

`nsems` número de semáforos con que contará el arreglo

`bandera` bandera de creación/acceso al semáforo, dependiendo del valor:

> `IPC_CREAT` crea un semáforo con permisos especiales en *perms* (`IPC_CREAT| perms`)
> `0` obtiene un manejador del semáforo existente[2].

La llamada regresa un identificador del semáforo a través del cual se hará referencia a todo el conjunto de semáforos.

4.4.2 Operaciones sobre semáforos

Dijkstra define dos tipos de operaciones relacionadas con semáforos, P(S) y V(S). Unix no proporciona este tipo de operaciones, pero si un medio de implementarlas. Es a través de la llamada de sistema `semop()` que se van a poder realizar operaciones sobre los semáforos. La sintaxis de la llamada es la siguiente:

```
int semop(id, ops, nops);
```

donde:

`int id` manejador de semáforos, obtenido de la llamada `semget()`

`struct sembuf **ops` definición de la operación a realizar:

[2]Para mayor información ver la sección 2.1.1.

```
struct sembuf {
  short sem_num; /* numero semaforo sobre el que se hara la operacion */
  short sem_op;  /* operacion a realizar en el semaforo */
  short sem_flg; /* parametro de la operacion */
}
```

`nops` número de operaciones a realizar

La llamada permite realizar atómicamente `nops` operaciones diferentes, definidas en `ops`, sobre un arreglo de semáforos direccionados por `id`. Es una primitiva bloqueante, lo que significa que puede bloquear al proceso en su ejecución dependiendo de las circunstancias.

El parámetro `ops` es un apuntador de apuntadores, ya que en realidad se trata de un apuntador a un arreglo de tipo `sembuf` de tamaño `nops`. Cada elemento de dicho arreglo define una operación a realizar. Dicha operación varía de acuerdo al valor del campo `sem_op` de la estructura `sembuf`, dependiendo si el valor es estrictamente negativo, positivo o cero.

Si el valor de `sem_op` es estrictamente negativo especifica una operación de tipo $P(S)$, la cual tiene por objeto el disminuir el valor del semáforo S. Si la operación no es posible el proceso se bloquea esperando que el valor del semáforo aumente. Recordemos que el valor del semáforo a verificar se encuentra dentro del campo `semval` de la estructura `sem`, por lo que esta operación puede resumirse como sigue:

si (`semval` $\geq$ abs(`sem_op`) **entonces**
    `semval` = `semval` - abs(`sem_op`)
sino
    proceso bloqueado hasta que (`semval` $\geq$ abs(`sem_op`))

Por otro lado si el valor del campo `sem_op` es estrictamente positivo se especifica una operación de tipo $V(S)$, la cual tiene por objeto aumentar el valor del semáforo S. Tiene el efecto de despertar a todos los procesos en espera de que el valor del semáforo S aumente. Como se dijo anteriormente dependerá de la política de administración del procesador, el elegir el proceso que se ejecutará. Una forma de expresar lo anterior es:

si (`sem_op` > 0) **entonces**
    `semval` = `semval` + `sem_op`; /* *desbloqueo procesos* */

El último caso a analizar es cuando `sem_op` tiene un valor de cero. Esto permite probar si el valor del semáforo es cero. Si no es cero causa que el proceso sea bloqueado. El siguiente pseudo-código resume lo anterior:

si (`semval` = `sem_op`) **entonces**
    regreso inmediato
sino
    proceso bloqueado hasta que (`semval` = `sem_op`)

El último campo de la estructura `sembuf` permite especificar la forma en que se realizará la operación definida en el resto de los campos. El campo `sem_flg` puede tomar dos valores: `IPC_NOWAIT` o `SEM_UNDO`.

La primera de las opciones permite definir operaciones no bloqueantes. Si durante la ejecución de `semop()` una operación no pudo ser ejecutada ninguna lo es, pero el proceso no se bloquea. El regreso es imediato, con un valor de -1 y con el estatus de error 11 (errno = 11 = EAGAIN). Este tipo de parámetro no aplica para un valor de `sem_op` positivo.

La segunda bandera, `SEM_UNDO`, previene del posible bloqueo accidental de recursos. Cuando esta activa y se asignan recursos (`sem_op > 0`) el núcleo del sistema operativo maneja un valor de ajuste para dejar los valores de los semáforos tal y como se encontraban antes de realizar la operación.

Si el proceso termina, voluntaria o involuntariamente el núcleo verifica si el proceso tiene *ajustes pendientes*, y en caso positivo aplica el ajuste correspondiente. Dichos ajustes se pueden resumir de la siguiente forma:

si (sem_op $>$ 0) **entonces**
    ajuste = ajuste - sem_op
si (sem_op $<$ 0) **entonces**
    ajuste = ajuste + abs(sem_op)
si (sem_op = 0) **entonces**
    nada (no aplica)

El siguiente código presenta un ejemplo de las dos llamadas vistas anteriormente:

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

#define KEY  45623
main()
{
    int id;
    struct sembuf oper[2];

    id = semget(KEY, 3, IPC_CREAT | 0666);

    oper[0].sem_num = 2;
    oper[0].sem_op = 0;
    oper[0].sem_flg = 0;
    oper[1].sem_num = 1;
    oper[1].sem_op = 1;
    oper[1].sem_flg = SEM_UNDO;

    semop(id, &oper, 2);
}
```

Este código crea un arreglo de semáforos con tres elementos. Después se realizan dos operaciones atómicas sobre el arreglo creado. La primera de ellas verifica si el valor del tercer elemento del arreglo (la numeración empieza en cero) tiene un valor de cero. Si este no es el caso el proceso se bloquea, en caso contrario el valor del segundo elemento es incrementado en uno, previniendo que si no se termina la operación este quede con su valor inicial.

La operación también puede definirse de la siguiente forma:

```
struct sembuf oper[2] = {
```

```
        2, 0, 0,
        1, 1, SEM_UNDO
};
```

La figura 6 es un complemento de la figura 5, en la que incluye la operación definida en el ejemplo anterior.

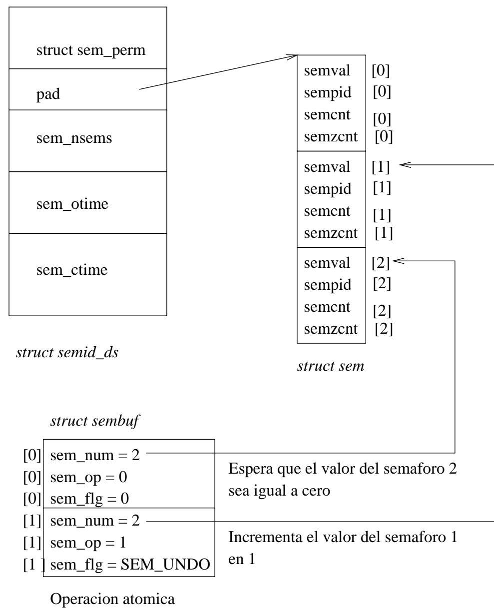


Figura 6: *Relación estructuras semáforos*

4.4.3 Modificación atributos semáforos

Se cuenta con la llamada `semctl()` para llevar a cabo modificaciones en varios atributos de los semáforos. La sintaxis de `semctl()` es:

```
int semtcl(id, num, cmd, arg}
```

donde

- `int id` manejador del semáforo, obtenido de la llamada `semget()`
- `int num` número de semáforo sobre el que se va a efectuar la operación
- `int cmd` comando a ejecutar
- `union semnum arg` argumento del comando a ejecutar, varía según el comando a ejecutar, es por eso que esta definido como `union`.

```
union sembuf {
   int               val;   /* usado con SETVAL o GETVAL */
   struct semid_ds  *buf;   /* usado con IPC_STAT o IPC_SET */
   ushort           *array; /* usado con GETALL o SETALL */
}
```

La tabla de abajo (figura 7) proporciona un resumen de los posibles argumentos de la llamada `semctl()`

Operación	semnum	interpret. arg	Regreso
GETNCNT	número semáforo	-	número procesos esperando que el semáforo aumente su valor
GETZCNT	número semáforo	-	número procesos esperando que el semáforo tome un valor de cero
GETVAL	número semáforo	-	válor del semáforo
GETALL	cuantos semáforos	array	0 si todo salió bien y -1 si no El arreglo contiene los valores de los *semnum* primeros semáforos
GETPID	número semáforo	-	Identificador del último proceso que realizó una operación sobre el semáforo
SETVAL	número semáforo	val	0 si todo salió bien y -1 si no Inicialización semáforo con un valor dado
SETALL	cuantos semáforos	array	0 si todo salió bien y -1 si no Inicialización de los *semnum* primeros semáforos
IPC_STAT	-	buf	0 si todo salió bien y -1 si no Extracción información almacenada en *semid_ds*
IPC_SET	-	buf	0 si todo salió bien y -1 si no Modificación campos estructura permisos
IPC_RMID	-	-	Se borra el arreglo de semáforos

Figura 7: *Tabla argumentos de la llamada semctl()*

4.5 Una librería de funciones de semáforos

Con el fin de ejemplificar las llamadas de sistema vistas anteriormente, se presenta un conjunto de funciones para la creación, acceso y uso de semáforos.

En varias de las funciones, siguiendo el mismo principio de las llamadas de sistema, se regresa un -1 en caso de que la operación no pueda llevarse a cabo.

4.5.1 Las operaciones P(S) y V(S) de Dijkstra

El código presentado a continuación permite realizar una operación P(S) y V(S) tal y como lo Dijkstra lo define (ver sección 4.1). La sintaxis de la función es:

```
P(sem, n)
V(sem, n)
```

Las funciones realizan una operación P(S) o V(S) sobre uno de los elementos (parámetro `n`) del conjunto de semáforos definido en `sem`. Estas funciones definen un arreglo de operaciones de un elemento. Las dos funciones utilizan la llamada de sistema `semop` con un valor de `sem_num` de `n` y 0 para `sem_flg`. La diferencia radica en que P(S) asigna un valor 1 a `sem_op` y V(S) un valor de -1 a `sem_op`.

```
/********* P(S[n]): lock semaforo n del conjunto sem **********/
```

```
P(sem, n)
  int sem, n;
{
  struct sembuf sop;

  sop.sem_num = n;
  sop.sem_op = -1;
  sop.sem_flg = 0;
  semop(sem, &sop, 1);
}


/******** V(S[n]): unlock semaforo n del conjunto sem **********/

V(sem, n)
   int sem, n;
{
   struct sembuf sop;

   sop.sem_num = n;
   sop.sem_op = 1;
   sop.sem_flg = 0;
   semop(sem, &sop, 1);
}
```

4.5.2 Creación de un arreglo de semáforos

La función `make_semas` crea un arreglo de semáforos de n elementos con una llave k. En caso de que exista un semáforo asociado con la llave pasada como parámetro este es borrado. Una vez que el semáforo es creado, este es inicializado con un valor de 1, para poder ser usado como semáforo binario para coordinar la entrada/salida a una sección crítica.

```
/* Funcion que crea un arreglo de n semaforos con llave k */
int make_semas(k, n)
   key_t k; int n;
{

   int semid, i;

  /* Si ya existe un conjunto de de semaforos lo destruye  */
  if ((semid=semget(k, n, 0)) != -1)
      semctl(semid, 0, IPC_RMID);                /* destruccion del semaforo */
  if ((semid=semget(k, n, IPC_CREAT | 0666)) != -1) {
     for (i=0;  i<n;  i++)
        P(S)(semid, i);
  }
  else
    return(-1);
  return semid;
}
```

4.5.3 Leyendo el valor de un semáforo

Unix permite crear un semáforo o un conjunto de semáforos. En los dos casos se utiliza un identificador para referenciarlos. Si se desea imprimir el valor del semáforo es necesario consultar el campo correspondiente en la estructra `sembuf`. Se proponen dos funciones para realizar lo anterior. La primera, `imp_semval()`, imprime el valor de un arreglo de semáforos que solo contiene un elemento. La función `valsem_n()` imprime el valor deli semáforo i del arreglo de semáforos.

```
/* regresa el valor del semaforo sem */
int imp_semval(sem)
  int sem;
{
  int semval;
  union{
    int             val;
    struct semid_ds  *buf;
    ushort           *array;
  } semctl_arg;

  semval = semctl(sem, 0,GETVAL, semctl_arg);
  if (semval < 0)
    return(-1);
  else
    return(semval);
}

/* regresa el valor del semaforo n de  sem */
int valsem_n(sem,n )
  int sem, n;
{
  int semval;
  union{
    int             val;
    struct semid_ds  *buf;
    ushort           *array;
  } semctl_arg;

  semval = semctl(sem, n,GETVAL, semctl_arg);
  if (semval < 0)
    return(-1);
  else
    return(semval);
}
```

4.5.4 Asignando un valor a un semáforo

Si se utiliza la función `make_semas()` para crear un semáforo, éste inicaliza el semáforo con un valor de 1, por otro lado si se utiliza la llamada `semget()` el semáforo tendrá un valor inicial de 0. Ahora bien, si se desea que el valor del semáforo sea diferente a uno o cero, se debe modificar uno de los campos de `sembuf`. La función de abajo `asig_val_sem()` asigna el valor `x` al semáforo `n` del conjunto de semáforos identificados por la variable `sem`.

```
/* asigna el valor x, al semaforo n de sem */
int asig_val_sem(sem,n,x)
  int sem, x, n;
```

```
{
  union{
    int             val;
    struct semid_ds  *buf;
    ushort           *array;
  } semctl_arg;

  semctl_arg.val = x;
  if (semctl(sem, n, SETVAL, semctl_arg) < 0)
    return(-1);
  else
    return(0);
}
```

4.5.5 Borrando un semáforo

Una vez que se termino de utilizar el semáforo es necesario borrarlo para evitar interferencias con futuras ejecuciones del sistema. La siguiente función, `borra_sem()`, borra el conjunto de semáforos especificado por el identificador `sem`.

```
/* Funcion que borra el semaforo especificado por sem */
int borra_sem(sem)
  int sem;
{
  int status;

  status = semctl(sem, 0, IPC_RMID);
  if (status < 0)
    return(-1);
  else
    return(status);
}
```

5 Un ejemplo de uso de memoria compartida y semáforos

En las dos últimas secciones se trataron todos los aspectos relacionados con memoria compartida y semáforos. Se asume que las funciones definidas anteriormente están agrupadas en un archivo de encabezado denominado `semop.h`

5.1 El problema del productor/consumidor

También conocido como *bounded buffer problem* o problema del buffer limitado, ([Dij65]). Dos procesos comparten un almacén (buffer) de tamaño fijo. Uno de ellos, el productor, coloca información en el almacén (buffer) mientras que el otro, el consumidor, la obtiene de él. Si el productor desea colocar un nuevo elemento, y el almacén se encuentra lleno, este debera irse a "dormir". El consumidor despertará al productor cuando elimine un elemento del almacén. De forma análoga, si el almacén esta vacío y el consumidor desea eliminar un elemento del almacén, este debe "dormirse" hasta que el productor coloque algo en el almacén.

Una solución del problema, tomada de [Tan93], en pseudo-código se presenta abajo.

```
semaforo mutex = 1;   /* semaforo que controla el acceso seccion critica */
semaforo vacio = N;   /* semaforo que cuenta las entradas vacias del almacen */
semaforo lleno = 0;   /* semaforo que cuenta los espacios llenos del almacen */

productor()a
{
  while (TRUE) {
    produce_item();     /* genera un item para colocarlo en el almacen */
    P(vacio);           /* decrementa contador entradas vacias */
    P(mutex);           /* entrada a la seccion critica */
     introduce_item();  /* colocando el item en el almacen */
    V(mutex);           /* salida de la seccion critica */
    V(lleno);           /* incrementa contador espacios llenos */
}

consumidor()
{
  while (TRUE) {
    P(lleno);          /* decrementa el contador de espacios llenos */
    P(mutex);          /* entrada a la seccion critica */
     retirar_item();   /* retirando un item del almacen */
    V(mutex);          /* salida seccion critica */
    V(vacio);          /* incrementando contador entradas vacias */
    consume_item();    /* utilizando el item retirado */
}
```

5.2 Codificando la solución

Lo primero a definir son los archivos de encabezado necesarios para implementar la solución presentada arriba. Los encabezados son los necesarios para el manejo de semáforos, memoria compartida y el que reune las funciones definidas en las sección 4.5.2.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "semop.h"       /* archivo encabezado de funciones relacionadas con semaforos */
```

A continuación se definen las llaves de la memoria compartida (`KEY_SHMEM`) y la de los tres semáforos a utilizar. También se define el tamaño máximo del almacén, así como el número de casillas llenas.

```
#define KEY_SHMEM         (key_t)447885   /* llave de la memoria compartida */
#define KEY_MUTEX         (key_t)478854   /* llave semaforo de entrada/salida seccion critica */
#define KEY_LLENO         (key_t)788544   /* llave semaforo del almacen lleno */
#define KEY_VACIO         (key_t)885447   /* llave semaforo del almacen vacio */
#define N            20                   /* numero maximo elementos del almacen */
#define LLENO        5                    /* numero casillas llenas del almacen */
```

Se definen las variables a usar para los identificadores de los semáforos y de la memoria compartida, así como la estructura que define los elementos del almacén.

```
int mutex,vacio,lleno;                    /* identificador semaforos */
int id;                                   /* identificador memoria compartida */

struct info {                             /* definicion de la memoria compartida */
  int index;
  int almacen[N];
};
```

La declaración de las estructuras de datos necesarias para el manejo de la memoria compartida se presenta a continuación:

```
#define SEGSIZE (sizeof(struct info))     /* longitud de la memoria compartida */

struct shmid_ds *shmbuf;
struct info *ptg;                         /* zona memoria local para atar memoria compartida */
```

La parte central del código se encuentra en la función `main()` del programa. Dentro de la función se declara la union necesaria para el manejo de semáforos. A continuación se crea la memoria compartida y se ata a la estructura. Después se crean los tres semáforos y se inicializa la memoria compartida, hay que notar que se asume que se encuentra llena la mitad del almacén. Los procesos del consumidor y del productor se crean. Cuando termina la ejecución de estos procesos se desata la memoria compartida y se borra la memoria y los semáforos.

```
main()

   union semun {
     int val;
     struct semid_ds *buf;
     ushort *array;
   } arg;

   /* creando la memoria compartida */
   id = shmget(KEY_SHMEM, SEGSIZE, IPC_CREAT|0666);

   /* atando la memoria compartida */
   ptg = (struct info *)shmat(id,0,0);

   /* creando los semaforos */
   mutex = make_semas(KEY_MUTEX,1);
   lleno = make_semas(KEY_LLENO,1);
   vacio = make_semas(KEY_VACIO,1);

   /* incializando los semaforos */
   asig_val_sem(mutex,0,1);
   asig_val_sem(vacio,0,N-LLENO);
   asig_val_sem(lleno,0,LLENO);

   /* inicializando la memoria compartida  */
   for (k=0; k<LLENO; k++)
      ptg->almacen[k]=1;
   ptg->index=LLENO;

   /* proceso padre es el productor y el hijo es el consumidor */
   if (fork() == 0)
      consumidor();
   else
```

```
      productor();

   /* borrando la memoria compartida y los semaforos */
   shmdt(0);
   shmctl(id, IPC_RMID, shmbuf);
   semctl(mutex,0,IPC_RMID);
   semctl(lleno,0,IPC_RMID);
   semctl(vacio,0,IPC_RMID);
}
```

El código a ejecutar por los procesos productor y consumidor, se encuentra definido en dos funciones del mismo nombre. El productor produce un elemento, verifica que el almacén no este lleno. Si el almacén esta lleno el proceso se bloquea hasta que se retire un elemento del almacén. En caso contrario se introduce el elemento en el almacén. La entrada al almacén esta supervisada por un semáforo tipo binario. Al final incrementa el contador de las casillas llenas del almacén, el cual podría despertar al consumidor en caso de que éste se encuentre dormido.

```
/* codigo del productor */
productor()
{
  while(1) {
     printf("Produciendo un item \n");
     P(vacio,0);
     P(mutex,0);
      printf("Introduciendo un elemento al almacen \n");
     V(mutex,0);
     V(lleno,0);
  }
}
```

El consumidor verifica si hay elementos a consumir dentro del almacén. Si no los hay se queda bloqueado hasta que se produzca algún elemento. En caso contrario, se retira el elemento del almacén (el acceso al almacén se realiza en exclusión mutua). Al final se incrementa el contador de las casillas vacías del almacén, con lo cual se podría despertar al productor en caso de que éste se encuentre dormido.

```
/* codigo del consumidor */
consumidor()
{
  while(1) {
     P(lleno,0);
     P(mutex,0);
      printf("\t Retirando un elemento del almacen \n");
     V(mutex,0);
     V(vacio,0);
     printf("\t Consumiendo un item \n");
  }
}
```

6 Las colas de mensajes

La sincronización y comunicación a través del envío y recepción de mensajes es una técnica utilizada por muchos sistemas hoy en día. Las colas de mensajes proporcionan un mecanismo para que dos

procesos puedan comunicarse entre ellos. Como su nombre lo indica la comuniación se efectua a través del envío/recepción de mensajes.

Los procesos que se comunican a través de este medio no necesitan tener ningún parentesco entre ellos; basta que tengan los medios necesarios para poder acceder a la cola de mensajes para depositar y retirar mensajes de ella. La cola de mensajes residen en el núcleo del sistema operativo.

Un mensaje es uno o más bloques de información con ciertas estructuras de control streams asociadas a ellos. Los mensajes pueden ser de diferentes tipos, el cual identifica el contenido del mensaje. El significado el tipo de mensaje es asunto del usuario.

Un identificador de cola de mensajes (`msqid`) es un entero positivo único. Cada identificador tiene una cola de mensajes y una estructura de datos asociado a él. La estructura de datos se conoce como `msqid_ds` y contiene los siguientes miembros:

```
struct     ipc_perm msg_perm;  /* permisos de operaciones */
struct     msg *msg_first;     /* primer mensaje en la cola */
struct     msg *msg_last;      /* ultimo mensaje en la cola */
ulong_t    msg_cbytes;         /* numero bytes en la cola */
ulong_t    msg_qnum;           /* numero de mensajes en la cola */
ulong_t    msg_qbytes;         /* numero maximo de bytes en la cola */
pid_t      msg_lspid;          /* ultimo proceso hizo msgsnd() */
pid_t      msg_lrpid;          /* ultimo proceso hizo msgrcv() */
time_t     msg_stime;          /* tiempo ultima operacion msgsnd() */
time_t     msg_rtime;          /* tiempo ultima operacion msgrcv() */
time_t     msg_ctime;          /* tiempo ultima operacion msgctl() */
```

El primer campo, `msg_perm` es una estructura de permiso que especifica los permismos de una operación sobre la cola de mensajes.

El campo `msg_first` apunta al primer mensaje de la lista, mientras que `msg_last` es un apuntador al último mensaje de la cola. El número de bytes en la cola de mensajes está almacenado en `msg_cbytes`, el número de mensajes esta en `msg_qnum` y `msg_qbytes` es el número máximo de bytes permitidos en la cola.

También se cuenta con información acerca de los procesos que usan la cola de mensajes. El campo `msg_lspid` almacena el identificador del último proceso que realizó una operación `msgsnd()`, en `msg_lrpid` se encuentra el identificador del último proceso que realizó una operación `msgrcv()`.

Al igual que en el manejo de archivos se cuentan con estampillas de tiempo donde se almacena la hora en que se hicieron ciertas operaciones. En el campo `msg_stime` se almacena la hora en la que se llevo a cabo la última operación `msgsnd()`, `msg_stime` es el tiempo de la última operación `msgsnd()` y en `msg_ctime` se encuentra la fecha de la última operación `msgctl()` que cambio un miembro de la lista de arriba.

6.1 Llamadas relacionadas con las colas de mensajes

Todas las llamadas de sistema utilizan el siguiente archivo de encabezado:

```
#include <sys/msg.h>
```

Las llamadas de sistema permiten realizar las siguientes operaciones:

- crear/acceder una cola de mensajes
- añadir mensajes a la cola
- retirar mensajes de la cola
- borrar la cola de mensajes

6.1.1 Creando una cola de mensajes

Lo primero a realizar es crear la cola de mensajes, lo cual se hace a través de la llamada de sistema `msgget`, que tiene la siguiente sintaxis:

```
int msgget(key_t key, int msgflg);
```

donde:

`key` es la llave de creación/acceso del canal.

`msgflg` es la bandera de creación del canal, se aplican los mismos principios que para los canales vistos anteriormente. Con un valor de 0 se accede a la cola de mensajes y con un valor `IPC_CREAT` se crea el canal.

La llamada regresa un valor de -1 si hubo algún error, en caso de éxito regresa un identificador para el manejo de la cola de mensajes.

6.1.2 Enviando un mensaje

Para enviar un mensaje se usa la llamada de sistema

```
int msgsnd(int msqid, const void *msgp,  size_t  msgsz,  int msgflg);
```

donde:

`msgid`: el identificador de la cola de mensajes, obtenido a partir de la llamada `msgget()`.

`msgp`: el mensaje a enviar

`msgsz`: la longitud del mensaje a enviar

`msgflg`: la bandera de envío de mensajes, puede tomar un valor de cero (bloqueante) o de `IPC_NOWAIT` (no bloqueante).

La llamada regresa 0 en caso de éxito y -1 en caso de error.

Es necesario considerar algunos detalles con respecto a esta llamada. La cola tiene una capacidad finita (generalmente algunos kbytes), si la cola esta llena y un proceso intenta enviar un mensaje este se bloqueará. Si se desea realizar una operación no bloqueante se debe pasar `IPC_NOWAIT` como valor del parámetro `msgflg`. Si la cola esta llena el proceso no se bloquea y el sistema regresa asigna `EAGAIN` a la variable `errno`. Por último la cola no solo se puede llenar por contener mensajes grandes, sino también por el número de mensajes almacenados.

El proceso se va a bloquear hasta que:

```
\item exista espacio para el mensaje
\item la cola de mensajes fue borrada del sistema (error \verb|EIDRM| es
      regresado)
\item el proceso, o thread, que realiz\'o la llamada es interrumpido por
      una se\~nal (en este caso se asigna \verb|EINTR| a la variable
      \verb|errno|.
```

Otro aspecto a considerar es el del mensaje en sí. El parámetro `msgp` es el que recibe el mensaje, el cual debe cumplir con el siguiente formato (definido en `<sys/msg.h>`):

```
struct msgbuf {
  long mtype;      /* tipo de mensaje */
  char mtext[1];   /* datos del mensaje */
}
```

El tipo de mensaje (`mtype`) debe ser mayor a cero (valores negativos y cero son usados para otros propósitos). El tipo de los datos de mensaje no esta restringido a solo texto, el usuario puede asignarle cualquier tipo de información (binario o texo). El núcleo no interpreta el contenido de los mensajes.

Dependiendo de la aplicación el usuario definirá la estructura del mensaje a intercambiar. Por ejemplo, si se desea intercambiar mensajes que consisten de un número de punto flotante, seguido de un caracter y de una cadena de diez caracteres, la estructura del mensaje tendrá la siguiente forma:

```
typedef struct my_msgbuf {
     long  tipo;        /* tipo de mensaje */
     float num;         /* principio mensaje, numero flotante */
     char  letra;       /* caracter */
     char  texto[10];   /* datos del mensaje */
}
```

6.1.3 Recibiendo un mensaje

Para recibir un mensaje (i.e. retirarlo de la cola de mensajes), se utiliza la siguiente llamada de sistema:

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

donde:

`msgid` identificador de la cola de mensajes

`msgp` dirección donde se va a almacenar el mensaje

`msgzs` tamaño de la parte de datos del mensaje

`msgtyp` tipo de mensaje

`msgflg` especifica qué hacer si un mensaje de un determinado tipo no se encuentra en la cola de mensajes

El parámetro `msgtyp` especifica cual mensaje dentro de la cola será retirado. Si tiene un valor de cero, el primer mensaje en la cola es retirado (debido a la política FIFO se extrae el más viejo de los mensajes). Si es más grande que cero el primer mensaje cuyo tipo sea igual al valor de `msgtyp`. Si el valor es menor que cero, el primer mensaje con el menor valor de tipo que sea menor o igual al valor absoluto del parámetro es regresado.

El último argumento, `msgflg` es utilizado para evitar bloqueos debido a la no existencia de un determinado mensaje. Si se asigna un valor de `IPC_NOWAIT` a este parámetro y no existe ningún mensaje disponible, la llamada regresa inmediatamente con un error `ENIMSG`. Por otro lado, el proceso que hizó la llamada se bloquea hasta que cualquiera de las siguientes acciones se lleva a cabo:

- un mensaje del tipo solicitado esta disponible
- la cola de mensajes es borrada del sistema (se genera el error `EIFRM`)
- el proceso o thread es interrumpido por una señal (la variable `errno` toma el valor de `EINTR`).

6.1.4 Accediendo los atributos de la cola de mensaje

Al igual que los otros canales existe una llamada para el manejo de los diferentes atributos de una cola de mensajes. La sintaxis de la llamada de sistema que permite realizar lo anterior es:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Las opciones son semejantes a las de las llamadas *...ctl* vistas anteriormente. Estas son:

- IPC_STAT: asigna cada uno de los valores de los campos de la estructura de control asociada a la cola de mensajes, a los campos del parámetro `buf`.
- IPC_SET: inicializa los campos `msg_permi.uid`, `msg_perm.gid` y `msg_perm.mode` (permisos) de la estructura de control de la cola apuntada por `buf`.
- IPC_RMID: borra el identificador de la cola de mensajes

6.2 Ejemplo uso colas de mensajes

Se presentan tres programas para ejemplificar el uso de las llamadas de sistema presentadas en las secciones anteriores. Los tres programas utilizan la misma cola de mensajes. Para evitar confusiones se utiliza un archivo de encabezado (`mensaje.h`) donde se almacenan las estructuras de datos comunes a todos los programas.

```
#define LLAVE  34251
#define TAMCOLA sizeof(struct mensaje)

struct mensaje {
    long  tipo;         /* tipo de mensaje */
    long  hora;         /* hora generacion mensaje */
    int   pid;          /* identificador del proceso */
    char  texto[30];    /* datos del mensaje */
};

typedef struct mensaje my_msgbuf;
```

El primero de ellos crea una cola de mensajes, después crea cinco procesos hijos, cada uno de los cuales escribe un mensaje en dicha cola. Hay que notar que el tipo de mensajes esta en función directa del identificador del proceso que genero el mensaje. Esto ejemplifica una posible aplicación del campo de tipo de mensajes, es decir diferenciar el origen de los diferentes mensajes.

```
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "mensaje.h"

int cont;
int msgid;
my_msgbuf buf;

main()
{
   msgid = msgget(LLAVE, IPC_CREAT | 0666);

   if ( msgid < 0  ) {
     printf("Error al crear la cola de mensajes \n");
     exit(1);
   }

   cont=1;
   do {
     if (fork() == 0) {
        /* generando el mensaje */

        buf.tipo = getpid()%10;
        buf.hora = time(NULL);
        strcpy(buf.texto,"Hola mundo");
        buf.pid = getpid();

        /* enviando el mensaje */

        msgsnd(msgid, &buf, TAMCOLA, 0);
        exit(0);
     }
     usleep(100);
     cont++;
   }
   while (cont <= 5);
   printf("Se enviaron %d mensajes \n",cont-1);
}
```

El segundo programa accede a la cola de mensajes creada por el primer programa, y despliega todos los mensajes contenidos en ella. A notar que al final se borra la cola de mensajes. Se asume que la cola de mensajes contiene cinco mensajes.

```
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>
#include "mensaje.h"

int cont;
int msgid;
my_msgbuf buf;

main()
{
   msgid = msgget(LLAVE, 0);

   if ( msgid < 0  ) {
     printf("Error al acceder la cola de mensajes \n");
     exit(1);
   }

   cont=1;
   do {
      /* retirando mensajes de la cola de mensajes */

      msgrcv(msgid, &buf, TAMCOLA, 0, 0);
      printf("Mensaje %d de %d bytes\n",cont,TAMCOLA);
      printf("Tipo mensaje: %d \n",buf.tipo);
      printf("Hora envio mensaje: %d \n",buf.hora);
      printf("Identificador proceso emisor: %d \n",buf.pid);
      printf("Texto asociado: %s \n\n",buf.texto);
      cont++;
   }
   while (cont <= 5);

   /* borrando la cola de mensajes */

   msgctl(msgid, IPC_RMID, 0);

   printf("\n\nSe recibieron %d mensajes \n",cont-1);
}
```

El tercer y último programa lee un número de la entrada estándar (en caso de que no se proporcione ninguno se genera al azar) y busca un mensaje del mismo tipo en la cola de mensajes. Si el mensaje existe despliega su contenido, si no existe despliega un error y si hay un error en la lectura despliega un mensaje de error diferente.

```
int nmsg;
int res;
int msgid;
my_msgbuf buf;

main(int argc, char *argv[])
{
   if (argc == 2)
     nmsg = atoi(argv[1]);
   else {
     srandom(getpid());
     nmsg = random() % 9;
   }

   msgid = msgget(LLAVE, 0);
```

```
   if ( msgid < 0  ) {
     printf("Error al acceder la cola de mensajes \n");
     exit(1);
   }

   res = msgrcv(msgid, &buf, TAMCOLA, nmsg, IPC_NOWAIT);

   if (res < 0) {
      if (errno == ENOMSG) {
         printf("No existe ningun mensaje del tipo %d \n", nmsg);
         exit(0);
      }
      else {
         printf("Error %d en la lectura de mensajes \n", errno);
         exit(0);
      }
   }
   else {
      printf("Mensaje de %d bytes\n",TAMCOLA);
      printf("Tipo mensaje: %d \n",buf.tipo);
      printf("Hora envio mensaje: %d \n",buf.hora);
      printf("Identificador proceso emisor: %d \n",buf.pid);
      printf("Texto asociado: %s \n\n",buf.texto);
   }
}
```

Algunas posibles salidas de los programas anteriores se presentan abajo. En el caso del segundo programa (`recibe`) solo se imprime una parte de la salida. Para el último de los programas (`busca`) se despliegan dos o más salidas.

```
rogomez@armagnac:388>envia
Se enviaron 5 mensajes
rogomez@armagnac:389>recibe
Mensaje 1 de 44 bytes
Tipo mensaje: 3
Hora envio mensaje: 960582420
Identificador proceso emisor: 4133
Texto asociado: Hola mundo

Mensaje 2 de 44 bytes
Tipo mensaje: 5
Hora envio mensaje: 960582421
     :
     :
     :
Texto asociado: Hola mundo

Se recibieron 5 mensajes
rogomez@armagnac:390>envia
Se enviaron 5 mensajes
rogomez@armagnac:391>busca
Mensaje de 44 bytes
Tipo mensaje: 6
Hora envio mensaje: 960586371
Identificador proceso emisor: 4216
Texto asociado: Hola mundo

rogomez@armagnac:392>busca
```

```
No existe ningun mensaje del tipo 3
rogomez@armagnac:393>busca 5
Mensaje de 44 bytes
Tipo mensaje: 5
Hora envio mensaje: 960586371
Identificador proceso emisor: 4215
Texto asociado: Hola mundo

rogomez@armagnac:394>
```

Referencias

[Bro92] Unix Distributed Programming, Chris Brown, Ed. Prentice Hall

[Dij65] Dijkstra E.W., *Cooperating Sequential Processes* Technological University, Eindhoven, Netherlands, 1965 (Reprinted in F. Genuys (ed.), *Programming Languages*, New York, NY: Academic Press, 1968).

[Dij68] Dijkstra, 1968

[Ga94] Unix Programación Avanzada, Fco. Manuel Garcíaz. Addison Wesley Iberoamericana

[Ri92] Comunicaciones en Unix, Jean Michel Rifflet; Ed. Mc. Graw Hill

[St90] Unix Networking Programming, Stevens, Ed. Prentice Hal

[Ste97] Advanced Programming in the Unix Environment, W. Richard Stevens, Ed. Addison Wesley

[Tan93] Sistemas Operativos Modernos, Andrew S. Tanenbaum, Ed. Prentice Hall, 1993