Sistemas Operativos – Práctica 3

FECHA DE ENTREGA:

- **GRUPOS DE LOS LUNES Y MIÉRCOLES**
  - **SEMANA 04 - 08 ABRIL (HORA LÍMITE DE ENTREGA: UNA HORA ANTES DEL** COMIENZO DE LA CLASE DE PRÁCTICAS).
- **GRUPOS DE LOS JUEVES Y VIERNES**
  - **SEMANA 11- 15 ABRIL (HORA LÍMITE DE ENTREGA: UNA HORA ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS).**

La tercera práctica se va a desarrollar en tres semanas con el siguiente cronograma:
Semana 1: Memoria Compartida
Semana 2: Semáforos
Semana 3: Ejercicio de asimilación de los conceptos

Los ejercicios correspondientes a esta segunda práctica se van a clasificar en:

APRENDIZAJE, se refiere a ejercicios altamente recomendable realizar para el correcto seguimiento de los conocimientos teóricos que se presentan en esta unidad didáctica.

ENTREGABLE, estos ejercicios son obligatorios y deben entregarse correctamente implementados y documentados.

SEMANA 1

Memoria Compartida

La memoria convencional que puede direccionar un proceso a través de su espacio de direcciones virtual es local al proceso y cualquier intento de direccionar esa memoria desde otro proceso provocará una violación de segmento.

La memoria compartida es una zona de memoria común gestionada a través del sistema operativo, a la que varios procesos pueden conseguir acceso de forma que lo que un proceso escriba en la memoria sea accesible al resto de procesos.

Los procesos pueden comunicarse directamente entre sí compartiendo partes de su espacio de direccionamiento virtual, por lo que podrán leer y/o escribir datos en la memoria compartida. Para conseguirlo, se crea una región o segmento fuera del espacio de direccionamiento de un proceso y cada proceso que necesite acceder a dicha región, la incluirá como parte de su espacio de direccionamiento virtual.

Para trabajar con memoria compartida en C para Linux se utilizan básicamente las siguientes llamadas al sistema:
· *shmget*, crea una nueva región de memoria compartida o devuelve una existente.
· *shmat*, une lógicamente una región al espacio de direccionamiento virtual de un proceso.
· *shmdt*, separa una región del espacio de direccionamiento virtual de un proceso.
· *shmctl*, manipula varios parámetros asociados con la memoria compartida y realiza diversas operaciones de control, como por ejemplo borrar la memoria.

Estas funciones están incluidas en los ficheros de cabecera <sys/ipc.h>, <sys/shm.h> y <sys/types.h>.

La sintaxis de las funciones anteriores es la siguiente:

- *int shmget (key_t key, int size,int shmflg);*
  donde *size* es el número de bytes de la región. El núcleo del sistema busca en la tabla de memoria compartida por la clave (*key*) indicada. La clave será un valor entero único para todos los procesos que compartan la misma región de memoria. Si existe una entrada y los permisos lo permiten, devuelve un identificador a la región (*shmid*). Si no lo encuentra y se ha indicado la opción IPC_CREAT en *shmflg*, el núcleo creará una nueva región. Otras opciones que se pueden indicar en el parámetro *shmflg* son los permisos de la región.

  En caso de que la función genere un error, devolverá -1.

  La región de memoria quedará reservada y sólo se asignará al espacio de direccionamiento de los procesos (tablas de páginas) cuando éstos se unan a la región mediante la ejecución de la llamada al sistema *shmat()*.

  En algunos sistemas Unix, a la hora de utilizar la función *shmget* para crear un segmento de memoria compartida, es necesario usar los *flags* SHM_W y SHM_R además del *flag* IPC_CREAT. Estos *flags* dan permisos a la memoria compartida de escritura y lectura, respectivamente. Sin estos *flags*, el segmento de memoria compartida se crea, pero los procesos no pueden escribir o leer de ella.

- *char *shmat (int shmid, char *addr, int shmflg)*
  donde *shmid* (valor que devuelve la función *shmget()*) identifica a la región de memoria compartida, *addr* es la dirección virtual donde el usuario quiere unir la memoria compartida y *shmflg* especifica los permisos de la región.
  El valor que devuelve es la dirección virtual donde el núcleo ha unido la región. En caso de error, devuelve –1.
  Cuando se ejecuta *shmat*(), el núcleo verifica que el proceso tiene los permisos necesarios para acceder a la región. Si la dirección especificada por el proceso es 0, el núcleo elige una dirección virtual conveniente.
- *int shmdt (char *addr);*
  Para separar una región de memoria de su espacio de direccionamiento virtual, los procesos utilizan la llamada al sistema *shmdt()*, donde *addr* es la dirección que devuelve la llamada *shmat()*.

- *shmctl (int shmid, int cmd, struct shmid_ds shmstatbuf);*
  La llamada al sistema *shmctl()* permite a un proceso buscar el estado y establecer los parámetros de la región de memoria compartida.
  El parámetro *shmid* identifica a la entrada correspondiente en la tabla de memoria compartida, *cmd* indica el tipo de operación a realizar y *shmstatbuf* indica la dirección de una estructura de datos a nivel de usuario que contiene la información de estado de la entrada de la tabla de memoria compartida. La sintaxis de esta estructura es:

```
struct shmid_ds {
        struct ipc_perm shm_perm; /* permisos */
        int shm_segsz; /* Tamaño del segmento */
        ushort shm_cpid; /* PID del creador */
        ...
};
```

  Algunas operaciones que se pueden indicar en el parámetro *cmd* de la llamada *shmctl()* son:

- IPC_STAT sitúa el valor actual de cada elemento de la estructura de datos asociada con shmid en la estructura apuntada por *shmstatbuf*.
- IPC_SET establece el valor de los elementos de la estructura de datos asociada con shmid, obtenidos a partir de la estructura de datos apuntada por shmstatbuf.
- IPC_RMID elimina del sistema el identificador de memoria compartida especificado en *shmid*.

Resumiendo, para usar memoria compartida en Linux es necesario seguir una serie de pasos que luego se traducen a llamadas al sistema.

1. Necesitamos obtener un identificador de IPC (Inter Process Communication). Para ello, una posibilidad es convertir una ruta (path) del sistema en un identificador IPC. Este identificador es necesario para crear la crear la zona de memoria virtual. Esto es muy sencillo de hacer con la llamada al sistema *ftok()*.
2. Crear el segmento de memoria compartida con la llamada al sistema *shmget()*.
3. Operar con la memoria compartida. Indicamos lo que queremos compartir con la llamada al sistema *shmat()*.
4. Destruimos el segmento de memoria compartida con la llamada al sistema *shmdt()* y *shmctl()*. Cuando un segmento de memoria compartida es borrado (mediante *shmctl())* en realidad sólo se está marcando para ser borrado. La eliminación definitiva del segmento la realizará el sistema operativo cuando todos los procesos enganchados a él se desenganchen.

Ejemplo de uso:

Proceso 1

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

#define FILEKEY "/bin/cat" /*Util  para ftok */

#define KEY 1300
#define MAXBUF 10

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

int *buffer; /* shared buffer */
int key, id_zone;
int i;

char c;

/* Key to shared memory */
   int key = ftok(FILEKEY, KEY);
   if (key == -1) {
      fprintf (stderr, "Error with key \n");
      return -1;
   }

   /* We create the shared memory */
   id_zone = shmget (key, sizeof(int)*MAXBUF, IPC_CREAT | IPC_EXCL
                     |SHM_R | SHM_W);
   if (id_zone == -1) {
      fprintf (stderr, "Error with id_zone \n");
      return -1;
   }
```

```
   printf ("ID zone shared memory: %i\n", id_zone);


   /* we declared to zone to share */
   buffer = shmat (id_zone, (char *)0, 0);
   if (buffer == NULL) {
      fprintf (stderr, "Error reserve shared memory \n");
      return -1;
   }

   printf ("Pointer buffer shared memory: %p\n", buffer);

   for (i = 0; i < MAXBUF; i++)
      buffer[i] = i;


   /* The daemon executes until press some character */

   c = getchar();

   /* Free the shared memory */
   shmdt ((char *)buffer);
   shmctl (id_zone, IPC_RMID, (struct shmid_ds *)NULL);
   return 0;
}
```

Proceso 2

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <string.h>
#include <errno.h>
#include <sys/shm.h> /* shm*  */

#define FILEKEY "/bin/cat"
#define KEY 1300
#define MAXBUF 10

int main () {

int *buffer; /* shared buffer */
int key, id_zone;
int i;

char c;

   /* Key to shared memory */
   key = ftok(FILEKEY, KEY);
   if (key == -1) {
      fprintf (stderr, "Error with key \n");
      return -1;
   }

   /* we create the shared memory */
   id_zone = shmget (key, sizeof(int)*MAXBUF, IPC_CREAT | IPC_EXCL
                     |SHM_R | SHM_W);
   if (id_zone == -1) {
      fprintf (stderr, "Error with id_zone \n");
      return -1;
   }
```

```
   printf ("ID zone shared memory: %i\n", id_zone);

   /* we declared to zone to share */
   buffer = shmat (id_zone, (char *)0, 0);
   if (buffer == NULL) {
      fprintf (stderr, "Error reserve shared memory \n");
      return -1;
   }

   printf ("Pointer buffer shared memory: %p\n", buffer);

   /* Write the values of shared memory */

   for (i = 0; i < MAXBUF; i++)
      printf ("%i\n", buffer[i]);
   return 0;
}
```

Ejercicio 1. **(APRENDIZAJE)** Estudia qué hace el siguiente fragmento de código

```
if((shmid=shmget(clave,TAM_SEG,IPC_CREAT|IPC_EXCL|0660))==-1){

     printf("El segmento de memoria compartida ya existe\n");

     printf("    Abriendo como cliente\n");

     if((shmid=shmget(clave,TAM_SEG,0))==-1)

         printf("Error al abrir el segmento\n");

   }

  else

   printf("Nuevo segmento creado\n");
```

Facilidades de IPC desde la línea de comandos

Los programas estándar ipcs e ipcrm permiten controlar los recursos IPC que gestiona el sistema, y nos pueden ser de gran ayuda a la hora de depurar programas que utilizan estos mecanismos.

- ipcs, se utiliza para ver qué mecanismos están asignados y a quién. Si no se indica ninguna opción ipcs muestra un resumen de la información de control que se almacena para la memoria compartida, los semáforos y mensajes que hay asignados. Las principales opciones son:

  -m muestra información de los segmentos de memoria compartida que hay activos.
  -s muestra información de los semáforos que hay activos.
  -q muestra información de las colas de mensajes que hay activas.
  -b muestra información completa sobre los tipos de mecanismos IPC que hay activos.

- ipcrm, se utiliza para liberar un mecanismo asignado. A continuación se explican las opciones más comunes:
  -m shmid borra la zona de memoria compartida cuyo identificador coincide con shmid.
  -s semid borra el semáforo cuyo identificador coincide con semid.
  -q msqid borra la cola de mensajes cuyo identificador coincide con msqid.

Si interrumpimos un proceso con la señal de interrupción Ctrl-C o simplemente el proceso termina de forma anormal, el recurso (la memoria compartida, en este caso) no se libera y queda en el sistema. La forma de borrarla sería con este comando. Es bastante normal mientras desarrollamos y depuramos nuestro programa que no liberemos, abortemos el proceso, etc. El número de memorias compartidas que podemos crear está limitado, por tanto en las pruebas podríamos empezar a obtener errores debido a que no se pueden crear las memorias, ipcrm nos permite eliminar las memorias que se han quedado "pendientes" en nuestras pruebas.

Ejercicio 2. **(ENTREGABLE) (2.0 ptos) Condición de carrera** En este ejercicio se pide un programa escrito en lenguaje C, *ejercicio2.c*. El programa generará n procesos hijos (n es un argumento de entrada al programa). El proceso padre reservará un bloque de memoria, que compartirá con los procesos hijo, suficiente para una estructura del tipo

```
struct info{
    char nombre[80];
    int id;
}
```

Cuando el proceso padre reciba la señal SIG_USR1 leerá de la zona de memoria compartida e imprimirá su contenido, nombre del usuario e identificador.

Cada proceso hijo realizará los siguientes pasos:

- dormirá un tiempo aleatorio y
- solicitará al usuario que dé de alta un cliente (recoger el nombre del cliente)
- verificará de la zona de memoria compartida cuál fue el último id y lo incrementará en una unidad.
- enviará la señal SIG_USR1 al proceso padre
- terminará correctamente.

El proceso padre terminará cuando todos los proceso hijo hayan terminado.

Explica en qué falla el planteamiento de este ejercicio.

SEMANA 2

Semáforos

Los semáforos son un mecanismo de sincronización provisto por el sistema operativo. Permiten paliar los riesgos del acceso concurrente a recursos compartidos, y básicamente se comportan como variables enteras que tienen asociadas una serie de operaciones atómicas.

Existen dos tipos básicos de semáforos:

- *Semáforo binario*: sólo puede tomar dos valores, 0 y 1. Cuando está a 0 bloquea el acceso del proceso a la sección crítica, mientras que cuando está a 1 permite el paso (poniéndose además a 0 para bloquear el acceso a otros procesos posteriores). Este tipo de semáforos es especialmente útil para garantizar la exclusión mutua a la hora de realizar una tarea crítica. Por ejemplo, para controlar la escritura de variables en memoria compartida, de manera que sólo se permita que un proceso esté en la sección crítica mientras que se están modificando los datos.
- *Semáforo N-ario*: puede tomar valores desde 0 hasta N. El funcionamiento es similar al de los semáforos binarios. Cuando el semáforo está a 0, está cerrado y no permite el acceso a la sección crítica. La diferencia está en que puede tomar cualquier otro valor positivo además de 1. Este tipo de semáforos es muy útil para permitir que un determinado número de procesos trabajen concurrentemente en alguna tarea no crítica. Por ejemplo, varios procesos pueden estar leyendo simultáneamente de la memoria compartida, mientras que ningún otro proceso intente modificar datos.

Los semáforos tienen asociadas a dos operaciones fundamentales, caracterizadas por ser atómicas (es decir, se completan o no como una unidad, no permitiéndose que otros procesos las interrumpan a la mitad). Éstas son:

- *Down*: consiste en la petición del semáforo por parte de un proceso que quiere entrar en la sección crítica. Internamente el sistema operativo comprueba el valor del semáforo, de forma que si está a 1 se le concede el acceso a la sección crítica (por ejemplo escribir un dato en la memoria compartida) y decrementa de forma atómica el valor del semáforo. Por otro lado, si el semáforo está a 0, el proceso queda bloqueado (sin consumir tiempo de CPU, entra en una espera *no activa*) hasta que el valor del semáforo vuelva a ser 1 y obtenga el acceso a la sección crítica.
- *Up*: Consiste en la liberación del semáforo por parte del proceso que ya ha terminado de trabajar en la sección crítica, incrementando de forma atómica el valor del semáforo en una unidad. Por ejemplo, si el semáforo fuera binario y estuviera a 0 pasaría a valer 1, y se permitiría el acceso a cualquier otro proceso que estuviera bloqueado en espera de conseguir acceso a la sección crítica.

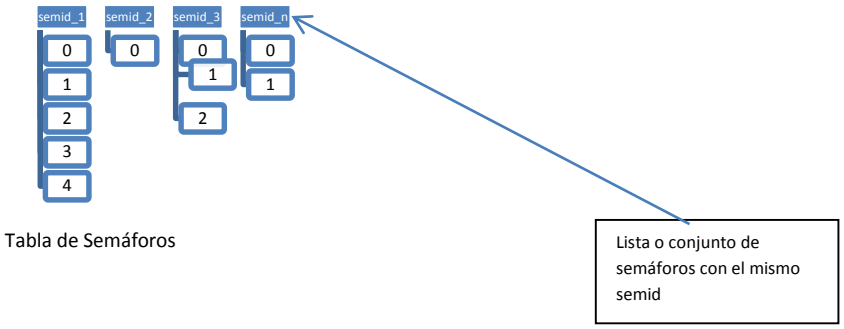
Las funciones para gestionar los semáforos en C para Unix están incluidas en los ficheros de cabecera *<sys/ipc.h>*, *<sys/sem.h>* y *<sys/types.h>*. En particular, las principales funciones de estos módulos son: *semget* (crear los semáforos o localizarlos), *semop* (realizar operaciones con los semáforos) y *semctl* (realizar diversas operaciones de control sobre los semáforos, como borrarlos). Estas funciones son muy potentes, pero habitualmente complicadas de usar, por lo que se recomienda utilizarlas en su versión más básica siempre que sea posible. Entre las peculiaridades de estas funciones es importante prestar especial atención a las siguientes:

- **Creación de semáforos: *int semget(key_t clave, int nsems, int semflg);***

La función *semget* define un *array* completo de semáforos del tamaño especificado. Es decir, por defecto hay que trabajar con conjuntos de semáforos, de forma que para crear un único semáforo es necesario crear un *array* de un solo elemento.

Como primer parámetro se le va a pasar una clave como las que hemos visto en la Semana 1 de esta práctica con el tema de Memoria Compartida. El segundo es el número de semáforos que queremos que tenga el conjunto que queremos usar y el último parámetro es un campo de bits que nos permite especificar opciones. De estas opciones, interesa IPC_CREAT. Si varios procesos hacen un semget para acceder a un semáforo, uno de ellos ha de especificar la macro IPC_CREAT con un OR (|) de los permisos con los que quiere crear el semáforo. El resto especificará 0 como último parámetro.

La llamada al sistema, si no hay error, devolverá un identificador para el conjunto de semáforos declarado (semid). Este identificador es similar al descriptor de fichero de los ficheros, es decir, se usará en todas las llamadas al sistema con las que queramos operar sobre el semáforo recién creado. En caso de error devuelve el valor -1.



Si el semáforo va a ser usado sólo por un proceso y sus hijos, también se puede dejar al sistema operativo que genere una clave única para el proceso. Esto se hace usando como primer parámetro de semget la macro IPC_PRIVATE. Todos los hijos usarán entonces el mismo valor devuelto por semget para acceder al semáforo. Por ejemplo:

```
if ((semid=semget(IPC_PRIVATE, 3, 0600)) == -1){
        perror("semget: IPC_PRIVATE");
}
```

La creación de un semáforo es independiente de su inicialización, lo cual es especialmente peligroso ya que no se puede crear un semáforo inicializado de forma atómica. Es el programador el que debe tener cuidado de inicializar los semáforos que cree.

- **Control de las estructuras de semáforos*: int semctl (int semid, int semnum, int cmd, union semun arg);***

Todas las operaciones que podemos hacer con un semáforo salvo las más importantes (incrementarlo y decrementarlo) se hacen con la llamada al sistema *semctl*.

Argumento 1. Es el identificador del conjunto de semáforos sobre el que queremos trabajar, que lo habremos obtenido con *semget()*.

Argumento 2. Es el índice del semáforo sobre el que queremos trabajar (al primer semáforo le corresponde el cero).

Argumento 3. Es la operación que queremos realizar. Puede ser una de estas:

- IPC_RMID: elimina el array de semáforos, le indica al núcleo que debe borrar el conjunto de semáforos agrupados bajo el identificados semid. Es necesario eliminar el array de semáforos si ya no se va a usar, pues el número de arrays que existe en el sistema es limitado. Normalmente lo eliminará el proceso que lo haya creado. Ejemplo:

  semctl(semaforo,0,IPC_RMID); /* Aunque pone 0, elimina todo el array. */

- GETVAL: devuelve el valor actual del semáforo. No se suele usar más que para depurar.
- SETVAL: da un valor al semáforo. Ejemplo:

  semctl(semaforo,3,SETVAL,2); /* Asigna el valor 2 al cuarto semáforo del array semaforo*/

- GETALL: permite leer el valor de todos los semáforos asociados al identificador *semid*. Estos valores se guardan en *arg*.
- SETALL: sirve para iniciar el valor de todos los semáforos asociados al identificador *semid*. Los valores de inicialización deben estar en *arg*.
- IPC_STAT e IPC_SET: nos permite conocer o establecer ciertas propiedades del array de semáforos. Las propiedades se especifican o se leen de una estructura de tipo struct semid_ds pasada por referencia. Ejemplo:
- GETPID: devuelve el PID del último proceso que actuó sobre el semáforo.
- GETNCNT: devuelve el número de procesos bloqueados en la cola del semáforo.

Argumento 4. Este último argumento en caso de ser requerido por la función de *semctl* ha de ser del tipo *union semun* definido en el archivo de cabecera de <*sys/sem.h*>

```
union semun {
        int  val;
        struct semid_ds *buf;
        ushort_t   *array;
    };
```

La definición de esta unión tiene que incluirse explícitamente en nuestro programa.

- **Operaciones con los semáforos: *int semop(int semid, struct sembuf *sops, unsigned int nsops);***

La llamada a *semop()* permite realizar sobre un semáforo las operaciones de incremento (up-signal) o decremento (down-wait) de forma atómica. Si la llamada a la función no se realiza con éxito devuelve -1.

Argumento 1. *semid* corresponde al identificador del grupo de semáforos asociados.

Argumento 2. *sops* es un puntero a un array de estructuras que indican las acciones que se van a realizar sobre los semáforos.

La estructura *sembuf* está definida en el archivo de cabecera de <*sys/sem.h*> de la siguiente forma:

```
struct sembuf  {
        ushort sem_num;  /*Número del semáforo dentro del grupo*/
        short sem_op;  /*Operación: incrementar o decrementar*/
        short sem_flg; /*Máscara de bits*/
    };
```

El campo *sem_num* es el número del semáforo y se utiliza como índice para acceder a él, su valor va de 0 a N-1, siendo N el número total de semáforos agrupados bajo el identificador *semid*.

El campo *sem_op* es la operación a realizar sobre el semáforo especificado en *sem_num*. Si *sem_op* es positivo, el semáforo *sem_num* se incrementa en ese valor. Si es negativo, se decrementa. Si es cero su valor no se modifica. Cuando se intenta decrementar un semáforo de valor 0, el proceso se bloquea, en la cola del semáforo, a la espera de que otro proceso incremente el valor de dicho semáforo.

Las operaciones de incremento siempre se ejecutan satisfactoriamente, ya que un semáforo puede tomar valores positivos; sin embargo, las operaciones de decremento no siempre se pueden realizar. Si por ejemplo, el semáforo tiene valor 2, no podemos restarle un número mayor 2 porque el semáforo pasaría a tener valor negativo. Ante esta situación, la llamada a la función *semop()* responderá de diferentes formas, según el valor del campo *sem_flg*; sus bits significativos pueden ser:

- IPC_WAIT, valor por defecto. Se bloquea el proceso.
- IPC_NOWAIT, en este caso la llamada a *semop()* devuelve el control en caso de que no se pueda satisfacer la operación especificada en *sem_op*.
- SEM_UNDO, este bit previene contra el bloqueo accidental de semáforos. Si un proceso decrementa el valor de un semáforo y termina de forma anormal (por ejemplo, porque recibe una señal), este semáforo quedará bloqueado para otros procesos. Este problema se previene con el bit SEM_UNDO del campo *sem_flg*, ya que el núcleo se encarga de actualizar el valor del semáforo, deshaciendo las operaciones realizadas sobre él, cuando el proceso termina.

Argumento 3. *nsops* es el total de elementos que tiene el array de operaciones.

Ejercicio 3. **(APRENDIZAJE)** Estudia el siguiente código. Se ha ilustrado en él la creación de un array de semáforos, se ha operado sobre ellos y finalmente se liberan. Comprueba al finalizar el programa si los semáforos han sido liberados: $ ipcs -s

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

#define SEMKEY 75798
#define N_SEMAFOROS 2

main ( )
{

/*
 * Declaración de variables
 */

int sem_id;  /* ID de la lista de semáforos */
struct sembuf sem_oper;  /* Para operaciones up y down sobre semáforos */

union semun {
   int val;
   struct semid_ds *semstat;
   unsigned short *array;
} arg;

/*
 * Creamos una lista o conjunto con dos semáforos
 */

semid = semget(SEMKEY, N_SEMAFOROS,
               IPC_CREAT | IPC_EXCL | SHM_R | SHM_W);
if((semid == -1) && errno == EEXIST))
     semid=semget(SEMKEY,N_SEMAFOROS,SHM_R|SHM_W);
 if(semid==-1){
     perror("semget");
     exit(errno);
 }


/*
 * Inicializamos los semáforos
 */

arg.array = (unsigned short *)malloc(sizeof(short)*N_SEMAFOROS);

arg.array [0] = arg.array [1] = 1;

semctl (semid, N_SEMAFOROS, SETALL, arg);
```

```
   /*
    * Operamos sobre los semáforos
    */

   sem_oper.sem_num = 0; /* Actuamos sobre el semáforo 0 de la lista */
   sem_oper.sem_op =-1; /* Decrementar en 1 el valor del semáforo */
   sem_oper.sem_flg = SEM_UNDO; /* Para evitar interbloqueos si un
   proceso acaba inesperadamente */

   semop (semid, &sem_oper, 1);


   sem_oper.sem_num = 1; /* Actuamos sobre el semáforo 1 de la lista */
   sem_oper.sem_op = 1; /* Incrementar en 1 el valor del semáforo */
   sem_oper.sem_flg = SEM_UNDO; /* No es necesario porque ya se ha
hecho anteriormente */

   semop (semid, &sem_oper, 1);

  /*
   * Veamos los valores de los semáforos
   */

   semctl (semid, N_SEMAFOROS, GETALL, arg);

   printf ("Los valores de los semáforos son %d y %d",
           arg.array [0], arg.array [1]);

   /* Eliminar la lista de semáforos */

   semctl (semid, N_SEMAFOROS, IPC_RMID, 0);

   free(arg.array);

   }/* fin de la función main *
```

Ejercicio 4. **(ENTREGABLE) (2.0 ptos) Construcción de la biblioteca de semáforos.** Se pide construir la biblioteca de semáforos semaforos.h. La biblioteca contendrá las siguientes funciones:

```
/*****************************************************************
      Nombre:
            Inicializar_Semaforo.
      Descripcion:
            Inicializa los semaforos indicados.
      Entrada:
         int semid: Identificador del semaforo.
         unsigned short *array: Valores iniciales.
      Salida:
            int: OK si todo fue correcto, ERROR en caso de error.
      *****************************************************************/

      int Inicializar_Semaforo(int semid, unsigned short *array);


/*****************************************************************
      Nombre: Borrar_Semaforo.
```

```
      Descripcion: Borra un semaforo.
      Entrada:
         int semid: Identificador del semÃ¡foro.
      Salida:
           int: OK si todo fue correcto, ERROR en caso de error.
*************************************************************/

      int Borrar_Semaforo(int semid);

/*************************************************************
      Nombre: Crear_Semaforo.
      Descripcion: Crea un semaforo con la clave y el tamaño
      especificado. Lo inicializa a 0.
      Entrada:
         key_t key: Clave precompartida del semaforo.
         int size: Tamaño del semaforo.
      Salida:
         int *semid: Identificador del semaforo.
         int: ERROR en caso de error,
               0 si ha creado el semaforo,
               1 si ya estaba creado.
*************************************************************/

      int Crear_Semaforo(key_t key, int size, int *semid);


/*************************************************************
      Nombre:Down_Semaforo
      Descripcion:Baja el semaforo indicado
      Entrada:
         int semid: Identificador del semaforo.
         int num_sem: Semaforo dentro del array.
         int undo: Flag de modo persistente pese a finalización
                  abrupta.
      Salida:
        int: OK si todo fue correcto, ERROR en caso de error.
*************************************************************/

      int Down_Semaforo(int id, int num_sem, int undo);

/*************************************************************
      Nombre: DownMultiple_Semaforo
      Descripcion: Baja todos los semaforos del array indicado
                   por active.
      Entrada:
         int semid: Identificador del semaforo.
         int size: Numero de semaforos del array.
         int undo: Flag de modo persistente pese a finalización
                  abrupta.
         int *active: Semaforos involucrados.
      Salida:
           int: OK si todo fue correcto, ERROR en caso de error.
*************************************************************/

int DownMultiple_Semaforo(int id,int size,int undo,int *active);
```

```
/*****************************************************************
      Nombre:Up_Semaforo
      Descripcion: Sube el semaforo indicado
      Entrada:
         int semid: Identificador del semaforo.
         int num_sem: Semaforo dentro del array.
         int undo: Flag de modo persistente pese a finalizacion
                   abupta.
      Salida:
        int: OK si todo fue correcto, ERROR en caso de error.
*****************************************************************/

      int Up_Semaforo(int id, int num_sem, int undo);



/*****************************************************************
      Nombre: UpMultiple_Semaforo
      Descripcion: Sube todos los semaforos del array indicado
por active.
      Entrada:
         int semid: Identificador del semaforo.
         int size: Numero de semaforos del array.
         int undo: Flag de modo persistente pese a finalización
                 abrupta.
         int *active: Semaforos involucrados.

      Salida:
           int: OK si todo fue correcto, ERROR en caso de error.
*****************************************************************i
int UpMultiple_Semaforo(int id,int size, int undo, int *active);
```

Ejercicio 5. **(ENTREGABLE) (2.0 ptos)** Diseñar e implementar los clientes de pruebas para validar la biblioteca de semáforos que has confeccionado.

SEMANA 3

Es recomendable, a la hora de utilizar los semáforos, intentar seguir algún algoritmo ya diseñado para resolver casos tipo, como por ejemplo lectores/escritores, productor/consumidor, el problema de la barbería, la cena de los filósofos... Además hay que prestar especial atención a evitar el interbloqueo entre los procesos.

Ejercicio 6. **(ENTREGABLE)** (4 ptos) Un puente es estrecho y sólo permite pasar vehículos en un único sentido al mismo tiempo. Si pasa un coche en un sentido y hay coches en el mismo sentido que quieren pasar, entonces estos tienen prioridad frente a los del otro sentido (si hubiera alguno esperando para entrar en el puente). No hay límite al número de vehículos que pueden haber en el puente al mismo tiempo.
Simula el sistema suponiendo que los coches son procesos hijo y el puente el recurso compartido. Utiliza semáforos para garantizar que se cumplen las condiciones de acceso al

puente. Cada proceso hijo debe mostrar por pantalla cuándo entra en el puente y cuándo lo abandona. Se generarán un total de 100 vehículos, 50 en un sentido y 50 en el otro. Tras un tiempo de espera al azar (utilizar sleep(random() %20) o algo similar) los vehículos intentan entrar en el puente y, si lo consiguen, permanecerán en él durante un segundo (sleep(1)) antes de abandonarlo.

Nota: Se apreciará más el comportamiento del sistema si se alterna la creación de los procesos hijos en un sentido u otro.

Ejercicio 7. **(ENTREGABLE/OPCIONAL)** (1 pto) Repite el ejercicio anterior considerando que los vehículos son hilos en vez de procesos hijos. ¿Aprecias diferencias con el ejercicio anterior? ¿Cuál es más sencillo de implementar? ¿Cuál de las dos implementaciones consume más recursos?