

Trabajos práctico 5 - Sincronización

Práctica con Linux

Tuberías

A continuación veremos un ejemplo de Carretero Pérez (2001) de ejecución de comandos con tuberías. Podemos colocar el comando:

```
ubuntu@ubuntu:~$ ls | wc
136 143 2245
ubuntu@ubuntu:~$
```

que genera un listado de archivos (comando `ls`) y lo comunica a través de una tubería o pipe con `wc` que es el comando «word count», para contar palabras, para mayor información coloque el comando:

```
ubuntu@ubuntu:~$ man wc
```

El carácter pleca «|» le indica al intérprete `bash` que debe crear una tubería entre los dos procesos. El programa `wc` por defecto devuelve tres valores: el número de líneas(136), el de palabras(143) y el de bytes (2245).

A continuación se presenta el programa de Carretero Pérez que permite la ejecución de este comando visto, pero mediante llamadas a sistema.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
void main(void){
    int fd[2];
    pid_t pid;
    if (pipe(fd) < 0){
        perror("Error al crear la tuberia");
        exit(-1);
    }
    pid = fork();
    switch(pid){
        case -1: /* error */
            perror("Error en el fork");
            exit(-1);
        case 0: /* proceso hijo ejecuta "ls" */
            close(fd[0]); /* cierra el pipe de lectura */
            close(STDOUT_FILENO); /* cierra la salida estandar */
            dup(fd[1]);
            close(fd[1]);
            execlp("ls", "ls", NULL);
            perror("execlp");
            exit(-1);
        default: /* proceso padre ejecuta "wc" */
            close(fd[1]); /* cierra el pipe de escritura */
            close(STDIN_FILENO); /* cierra la entrada estandar */
            dup(fd[0]);
            close(fd[0]);
            execlp("wc", "wc", NULL);
            perror("execlp");
    }
}
```

Se compila con:

```
ubuntu@ubuntu:~$ gcc -o tuberia tuberia.c
```

Y se ejecuta:

```
ubuntu@ubuntu:~$ ./tuberia
```

Tuberías en POSIX En POSIX existen tuberías sin nombre, o simplemente pipes, y tuberías con nombre, o FIFOs.

Un pipe no tiene nombre y, por tanto, sólo puede ser utilizado entre los procesos que lo hereden a través de la llamada fork().

Para leer y escribir de una tubería en POSIX se utilizan descriptores de archivo.

Descriptores de archivo: Un descriptor de archivo en POSIX es un número indicador utilizado para acceder a un archivo. Funciona como una clave a una estructura de datos residente en el núcleo, que contiene detalles del archivo abierto. Esta estructura es un bloque de control que contiene información que el sistema necesita para administrar un archivo.

Hay 3 descriptores de archivo estándar de POSIX que tiene cada proceso, como vemos en la siguiente tabla:

Valor	Nombre	Constante #define en C
0	Entrada estándar (stdin)	STDIN_FILENO
1	Salida estándar (stdout)	STDOUT_FILENO
2	Error estándar (stderr)	STDERR_FILENO

Valores descriptores de archivo en POSIX

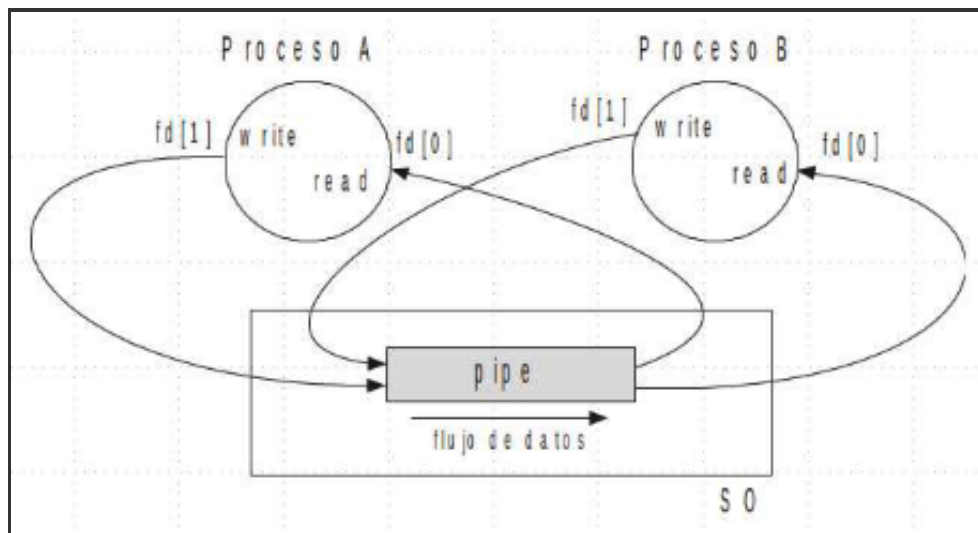
Tuberías sin nombre: Las tuberías sin nombre tienen asociados dos descriptores de archivos. Uno de ellos se emplea para leer y el otro para escribir.

La llamada `pipe(fd)` permite crear la tubería sin nombre. Su prototipo es el siguiente:

```
int pipe(int fildes[2]);
```

Esta llamada devuelve dos descriptores de archivos (`fildes`) que se utilizan como identificadores, como vemos en la siguiente figura:

- `fildes[0]`: descriptor de archivo que se emplea para leer de la tubería o pipe.
- `fildes[1]`: descriptor de archivo que se utiliza para escribir en la tubería.



La llamada `pipe()` devuelve 0 si ejecutó bien y -1 en caso de error.

El proceso hijo redirige su salida estándar a la tubería. Por su parte, el proceso padre redirecciona su entrada estándar a la tubería. Con esto se consigue que el proceso que ejecuta el programa «ls» escriba sus datos de salida en la tubería y el proceso que ejecuta el programa «wc» lea sus datos de la tubería.

Los pasos que realiza el proceso hijo para redirigir su salida estándar a la tubería son los siguientes:

- Con la instrucción `close(fd[0])` cierra el descriptor de lectura de la tubería, `fd[0]`, ya que no lo utiliza.
- Con `close(STDOUT_FILENO)` cierra la salida estándar, que inicialmente en un proceso referencia la terminal. Esta operación libera el descriptor de archivo 1, es decir, el descriptor `STDOUT_FILENO`.
- Duplica el descriptor de escritura de la tubería mediante la sentencia `dup(fd[1])`. Esta llamada devuelve y consume un descriptor, que será el de número más bajo disponible, en este caso el descriptor 1 que coincide en todos los procesos como el descriptor de salida estándar. Con esta operación se consigue que el descriptor de archivo 1 y el descriptor almacenado en `fd[1]` sirvan para escribir datos en la tubería. De esta forma se ha conseguido redirigir el descriptor de salida estándar en el proceso hijo a la tubería.
- Se cierra el descriptor `fd[1]`, ya que el proceso hijo no lo va a utilizar en adelante. Recuérdese que el descriptor 1 sigue siendo válido para escribir datos en la tubería.
- Cuando el proceso hijo invoca el servicio `exec()` para ejecutar un nuevo programa, se conserva en el PCB la tabla de descriptors de archivos abiertos y, en este caso, el descriptor de salida estándar 1 está referenciando a la tubería. Cuando el proceso comienza a ejecutar el código del programa «ls», todas las escrituras que se hagan sobre el descriptor de salida estándar se harán realmente sobre la tubería.

Candados o locks

Se trata simplemente de procesos en concurrencia sobre los 100 primeros caracteres de un archivo (se ejecutarán varios ejemplares del mismo programa). Para resolver el problema de la exclusión mutua sobre el acceso, se coloca un candado (lock) bloqueante, utilizamos la función `lockf()`, cuyo primer parámetro es un descriptor de archivo abierto, ya sea para lectura o para lectura/escritura. El siguiente parámetro de operación es alguno de los siguientes:

- `F_ULOCK`: eliminación de candado
- `F_LOCK`: candado exclusivo en modo bloqueante
- `F_TLOCK`: candado exclusivo en modo no bloqueante
- `F_TEST`: comprobación de existencia de un candado

El tercer parámetro es el tamaño, que permite especificar la extensión del bloqueo. La cual se expresa con respecto a la posición actual (el valor puede ser negativo para bloquear una zona anterior a la posición actual). Un tamaño nulo permite bloquear hasta el final del archivo (cualquiera que pueda ser en el futuro).

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <stdio.h>
#include <stdlib.h>
void main(void){
    int d;
    char buffer[100];
    d=open("archivo", O_RDWR | O_CREAT,S_IRWXU, 0);
    write(d,buffer,100);
    if(lockf(d, F_LOCK, 100) == -1){
        perror("lockf");
    }else{
        printf("Proceso %d: candado colocado\n", getpid());
        sleep(5);
        printf("Proceso %d: candado eliminado\n", getpid());
        lockf(d, F_ULOCK, 100);
    }
}
```

La llamada a `open()` abre el archivo en modo lectura/escritura, si «archivo» no existe lo crea. Luego la llamada a `write()` le escribe 100 caracteres, pero como «buffer» no ha sido inicializado le escribe basura, lo cual no nos importa en este caso. Se compila con:

```
ubuntu@ubuntu:~$ gcc -o candado candado.c
```

Primero, ejecútelo sólo para ver cómo funciona:

```
ubuntu@ubuntu:~$ ./candado
Proceso 2659: candado colocado
Proceso 2659: candado eliminado
ubuntu@ubuntu:~$
```

Y luego se ejecutan varias copias:

```
ubuntu@ubuntu:~$ ./candado & ./candado & ./candado &
```

enviándolas al background, veremos que el primer proceso coloca el candado sobre el archivo, espera 5 segundos (mediante la llamada `sleep`) y luego lo saca, sólo entonces el segundo proceso puede colocar el candado, esperar 5 segundos y luego sacarlo, para que el tercer proceso pueda hacer lo mismo. Pero observe, mediante los números PID, si lo hace en el orden esperado.

Uso Este método de colocar candados o locks sobre archivos para lograr exclusión mutua se usa de

muchas maneras en los sistemas operativos modernos. Por ejemplo, en Linux existe el directorio «/var/lock» o «/run/lock» en el que los procesos pueden crear un archivo testigo para indicar que se está accediendo a un determinado recurso en exclusión mutua. El contenido del archivo es irrelevante; es más, generalmente está vacío y, a veces, contiene el PID del proceso que ha solicitado el acceso exclusivo al recurso.

Por ejemplo, mientras se está actualizando el software del sistema -*mediante update manager, aptitude, apt, synaptic, yum*-, se coloca un candado para indicar que se está accediendo a la base de datos que contiene los datos del software instalado. En ese caso, en vez de la llamada «sleep(5)» de nuestro ejemplo, tendríamos la función o procedimiento encargada de hacer toda la actualización.

El archivo «/proc/locks» contiene información acerca de los candados que el núcleo ha puesto sobre archivos abiertos. Es generado por la función `get_locks_status` que está en el archivo fuente «`../linux/fs/locks.c`» del núcleo. Cada línea representa la información del lock para un archivo específico y documenta el tipo de candado aplicado al archivo. Las funciones `fcntl` y `flock` se utilizan para aplicar candados a los archivos. El núcleo también puede aplicar candados obligatorios (mandatory locks) a los archivos cuando sea necesario. Esta información aparece en el archivo «/proc/locks», como vemos en la siguiente figura.

```
ubuntu@ubuntu:~$ cat /proc/locks
1: POSIX ADVISORY WRITE 1441 08:04:131274 0 EOF
2: POSIX ADVISORY READ 1062 08:04:152062 128 128
3: POSIX ADVISORY READ 1062 08:04:135657 1073741826 1073742335
4: OFDLCK ADVISORY WRITE -1 08:04:3014816 0 0
5: OFDLCK ADVISORY WRITE -1 08:04:3014811 0 0
6: OFDLCK ADVISORY WRITE -1 08:04:3014802 0 0
7: OFDLCK ADVISORY WRITE -1 08:04:3014798 0 0
8: POSIX ADVISORY WRITE 1096 08:04:3014793 0 EOF
9: POSIX ADVISORY WRITE 1096 08:04:3014718 0 EOF
10: POSIX ADVISORY WRITE 1096 08:04:3014716 0 EOF
11: POSIX ADVISORY WRITE 1096 08:04:3014713 0 EOF
12: POSIX ADVISORY READ 589 00:16:658 0 0
13: FLOCK ADVISORY WRITE 552 00:16:609 0 EOF
14: POSIX ADVISORY READ 1441 08:04:164664 128 128
15: POSIX ADVISORY READ 1441 08:04:150973 1073741826 1073742335
16: POSIX ADVISORY READ 1441 08:04:164570 128 128
17: POSIX ADVISORY READ 1441 08:04:131293 1073741826 1073742335
18: POSIX ADVISORY READ 1441 08:04:160927 128 128
19: POSIX ADVISORY READ 1441 08:04:131299 1073741826 1073742335
20: POSIX ADVISORY READ 1441 08:04:157382 128 128
21: POSIX ADVISORY READ 1441 08:04:131304 1073741826 1073742335
22: OFDLCK ADVISORY WRITE -1 08:04:3014826 0 0
23: OFDLCK ADVISORY WRITE -1 08:04:3014821 0 0
24: POSIX ADVISORY READ 959 08:04:3276833 128 128
25: POSIX ADVISORY READ 959 08:04:3276825 1073741826 1073742335
26: POSIX ADVISORY WRITE 1096 08:04:3014709 0 EOF
27: POSIX ADVISORY WRITE 1096 08:04:3014711 0 EOF
28: POSIX ADVISORY WRITE 1096 08:04:3014791 0 EOF
29: POSIX ADVISORY WRITE 1096 08:04:3014789 0 EOF
30: POSIX ADVISORY WRITE 1096 08:04:3014787 0 EOF
31: POSIX ADVISORY WRITE 1096 08:04:3014785 0 EOF
32: POSIX ADVISORY WRITE 1096 08:04:3014783 0 EOF
33: POSIX ADVISORY WRITE 1096 08:04:3014775 0 EOF
34: POSIX ADVISORY WRITE 1096 08:04:3014766 0 EOF
35: POSIX ADVISORY WRITE 1096 08:04:3014764 0 EOF
36: POSIX ADVISORY WRITE 1096 08:04:3014762 0 EOF
37: POSIX ADVISORY WRITE 1096 08:04:3014760 0 EOF
38: POSIX ADVISORY WRITE 1096 08:04:3014755 0 EOF
39: POSIX ADVISORY WRITE 1096 08:04:3014752 0 EOF
40: POSIX ADVISORY WRITE 1096 08:04:3014722 0 EOF
41: POSIX ADVISORY WRITE 1096 08:04:3014940 0 EOF
42: POSIX ADVISORY WRITE 1096 08:04:3014696 0 EOF
43: POSIX ADVISORY WRITE 1096 08:04:3014700 0 EOF
44: POSIX ADVISORY WRITE 1096 08:04:3014699 0 EOF
45: POSIX ADVISORY READ 917 08:04:3276833 128 128
46: POSIX ADVISORY READ 917 08:04:3276825 1073741826 1073742335
47: FLOCK ADVISORY WRITE 726 00:1a:6 0 EOF
48: FLOCK ADVISORY WRITE 562 00:16:635 0 EOF
ubuntu@ubuntu:~$
```

Cada candado tiene su propia línea que comienza con un número único. La segunda columna se refiere a la clase de bloqueo utilizada, con FLOCK significando los bloqueos de archivos UNIX de estilo antiguo de una llamada al sistema flock y POSIX representando los bloqueos POSIX más nuevos de la llamada al sistema lockf.

La tercera columna puede tener dos valores: ADVISORY o MANDATORY (asesoramiento u obligatorio, respectivamente). ADVISORY significa que el bloqueo no impide que otras personas accedan a los datos, sólo impide que se produzcan otros intentos de bloqueo. MANDATORY significa que no se permite ningún otro acceso a los datos mientras se mantiene el bloqueo. La

cuarta columna revela si el bloqueo permite al propietario acceso para leer o escribir (READ or WRITE) al archivo. La quinta columna muestra el ID del proceso del candado. La sexta columna muestra el ID del archivo que se está bloqueando, en el formato

MAJOR-DEVICE:MINOR-DEVICE:INODE-NUMBER.

La séptima y octava columna muestran el inicio y el final de la región bloqueada del archivo.

Semáforos

Los semáforos en Ciencias de la Computación controlan el acceso a los recursos compartidos, al igual que los semáforos en las calles controlan el flujo de tráfico a través de una intersección. Sin embargo, son muy diferentes de los otros medios de la IPC que se han visto hasta ahora, porque no hacen que la información esté disponible entre los procesos, sino que sincronizan el acceso a recursos compartidos a los que no se debe acceder simultáneamente. En este sentido, el uso de semáforos se parece más al bloqueo de archivos o registros, excepto que los semáforos se pueden aplicar a más recursos que sólo a los archivos. Veremos solamente el tipo más simple de semáforo, un semáforo binario. Un semáforo binario toma uno de dos valores: 0 cuando un recurso está bloqueado y no debería ser accedido por otros procesos y 1 cuando el recurso está desbloqueado.

¿Cómo se utilizan los semáforos? Cuando un proceso necesita acceso a un recurso controlado, como un archivo, primero comprueba el valor del semáforo, del mismo modo que un conductor comprueba si un semáforo está en verde. Si el semáforo tiene un valor 0, que corresponde a una luz roja, el recurso está en uso, por lo que el proceso se bloquea hasta que el recurso está disponible (es decir, el valor del semáforo pasa a ser distinto de cero). En la terminología de la IPC del System V, este bloqueo se denomina espera o wait. Si el semáforo tiene un valor positivo, que corresponde a una luz verde en una intersección, el recurso asociado está disponible, por lo que el proceso decrementa el semáforo, realiza sus operaciones sobre el recurso, y luego incrementa el semáforo para liberar el bloqueo.

Creación de un semáforo Naturalmente, antes de que se pueda incrementar o decrementar un semáforo, éste debe existir. La llamada de función para crear un nuevo semáforo o acceder a uno existente es «semget», prototipada de la siguiente manera:

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

La función «semget» devuelve el identificador de semáforo asociado a un conjunto de semáforos nsems. Se crea un nuevo conjunto de semáforos si «key» es IPC_PRIVATE o si «key» no está ya en uso y el bit IPC_CREAT se fija en «flags». Al igual que con los segmentos de memoria compartidos y las colas de mensajes, «flags» también pueden ser modificadas (mediante operación lógica OR) en bits de permiso (en octal) para establecer los modos de acceso para el semáforo. Tenga en cuenta, sin embargo, que los semáforos tienen permisos de lectura y modificación, en lugar de permisos de lectura y escritura. Los semáforos usan la noción de alterar en lugar de escribir porque nunca se escriben datos en un semáforo, simplemente se cambia(o altera) su estado incrementando o disminuyendo su valor. «semget» devuelve -1 y establece «errno» si ocurre un error. En caso contrario, devuelve el identificador del semáforo asociado al valor de «key».

La función «semop» es el «caballo de batalla» de las rutinas de semáforos. Realiza operaciones en uno o más de los semáforos creados o a los que se accede mediante la llamada «semget». Su prototipo es el siguiente:

```
int semop(int semid, struct sembuf *semops, unsigned nops);
```

«semid» es un identificador de semáforo devuelto previamente por «semget» y apunta al semáforo establecido para manipular. «nops» es el número de elementos del conjunto de estructuras «sembuf» a las que apunta «semops».

El siguiente programa «creasem.c» crea un semáforo y lo incrementa, marcando así el recurso asociado como desbloqueado o disponible.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int semid;
    int nsems = 1;
    int flags = 0666;
    struct sembuf buf;
    semid = semget(IPC_PRIVATE, nsems, flags);
    if(semid<0){
        perror("semget");
        exit(EXIT_FAILURE);
    }
    printf("semáforo creado: %d\n", semid);
    buf.sem_num = 0;
    buf.sem_op = 1;
    buf.sem_flg = IPC_NOWAIT;
    if(semop(semid, &buf, nsems) < 0) {
        perror("semop");
        exit(EXIT_FAILURE);
    }
    system("ipcs -s");
    exit(EXIT_SUCCESS);
}
```

Note al final la llamada a «system("ipcs -s")». Esta llamada ejecuta un comando en el intérprete de comandos. El comando «ipcs» muestra información relacionada con la infraestructura IPC.

Entonces, luego de compilar el programa «creasem.c» vamos a mostrar los semáforos que hubiera en nuestro sistema, luego ejecutaremos nuestro programa que creará uno nuevo y además nos mostrará nuevamente a todos los que existen.

```
ubuntu@ubuntu:~$ gcc -o creasem creasem.c
ubuntu@ubuntu:~$ ipcs -s
----- Semaphore Arrays -----
key semid owner perms nsems
0x00105b6b 0 root 600 1
0x00105b6c 32769 root 666 1
ubuntu@ubuntu:~$ ./creasem
semaforo creado: 98307
----- Semaphore Arrays -----
key semid owner perms nsems
0x00105b6b 0 root 600 1
0x00105b6c 32769 root 666 1
0x00000000 98307 ubuntu 666 1
ubuntu@ubuntu:~$
```

El ejemplo utiliza «IPC_PRIVATE» para asegurar que el semáforo se crea como se solicita y luego muestra el valor de retorno de «semget» para mostrar el identificador del semáforo. La llamada «semop» inicializa el semáforo apropiadamente: puesto que sólo se crea un semáforo, «sem_num» es cero. Debido a que el recurso imaginario no está en uso (más precisamente, el semáforo no está asociado programáticamente con ningún recurso en particular), «creasem» inicializa su valor a 1, el equivalente a desbloqueado. Sin requerir un comportamiento de bloqueo en este caso, el «sem_flg» del semáforo está configurado en «IPC_NOWAIT», por lo que la llamada vuelve inmediatamente. Finalmente, el ejemplo utiliza la llamada de sistema para invocar el comando «ipcs» a nivel de usuario para confirmar que la estructura IPC solicitada existe de hecho.

Control y eliminación de semáforos La función de control y eliminación de semáforos es «semctl», prototipada de la siguiente manera:

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

«semid» identifica el conjunto de semáforos que desea manipular, y «semnum» especifica el semáforo en particular en el que está interesado. Este Práctico ignora la situación en la que hay múltiples semáforos en un conjunto, por lo que «semnum» (en realidad, un índice de una matriz de semáforos) siempre será cero.

Los valores posibles para el argumento «cmd» se enumeran en la siguiente tabla:

Comando	Descripción
GETVAL	Devuelve el estado actual del semáforo (bloqueado o desbloqueado). SETVAL Establece el estado actual del semáforo en «arg»
GETPID	Devuelve el «PID» del último proceso que llamó a «semop».
GETNCNT	Hace que el valor de retorno de «semctl» sea el número de procesos que esperan que el semáforo aumente, es decir, el número de procesos que esperan en el semáforo.
GETZCNT	Hace que el valor de retorno de «semctl» sea el número de procesos que esperan a que el semáforo sea cero.
GETALL	Devuelve el valor de todos los semáforos del conjunto asociados a «semid».
SETALL	Establece el valor de todos los semáforos del conjunto asociado a «semid» a los valores almacenados en «arg.array»
IPC_RMID	Elimina el semáforo con «semid»
IPC_SET	Establece el modo (bits de permiso) en el semáforo.
IPC_STAT	Cada semáforo tiene una estructura de datos, semid_ds, que describe completamente su configuración y comportamiento.
IPC_STAT	copia esta información de configuración en el miembro "arg.buf" de la estructura semun.

Si la llamada a «semctl» falla, devuelve -1 y establece «errno» apropiadamente. En caso contrario, devuelve un valor entero de GETPID, GETVAL o GETZCNT, dependiendo del valor de «cmd». Como habrá podido suponer, el argumento de «semun» juega un papel vital en la rutina de «semctl».

El siguiente programa utiliza la llamada «semctl» para eliminar un semáforo del sistema. Para hacerlo, deberá ejecutar «creasem» para crear un semáforo y luego usar ese identificador de semáforo como argumento para «cntrlsem».

Se compila de la manera usual:

```
ubuntu@ubuntu:~$ gcc -o cntrlsem cntrlsem.c
```

Y, si no habíamos ejecutado «creasem», hay que ejecutarlo porque debemos pasarle como parámetro el identificador «semid» para poder eliminar el semáforo.

```
ubuntu@ubuntu:~$ ./creasem
semáforo creado: 98306
----- Semaphore Arrays -----
key semid owner perms nsems
0x00105b6b 0 root 600 1
0x00105b6c 32769 root 666 1
0x00000000 98306 ubuntu 666 1
```

Programa «cntrlsem.c»

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    int semid;
    if(argc != 2) {
        puts("Uso: cntrlsem <id-semaforo>");
        exit(EXIT_FAILURE);
    }
    semid = atoi(argv[1]);
    if((semctl(semid, 0, IPC_RMID)) < 0) {
        perror("semctl IPC_RMID");
        exit(EXIT_FAILURE);
    } else {
        puts("semáforo eliminado\n");
        system("ipcs -s");
    }
    exit(EXIT_SUCCESS);
}
```

```
ubuntu@ubuntu:~$ ./cntrlsem 98306
semáforo eliminado
----- Semaphore Arrays -----
key semid owner perms nsems
0x00105b6b 0 root 600 1
0x00105b6c 32769 root 666 1
ubuntu@ubuntu:~$
```