TP-Entrega-A. Sistemas Operativos 2025.

Probar los códigos presentados en el trabajo de Semáforos que se encuentra a continuación, probarlos, corregirlo y establecer aquello ejemplos que crea necesarios dentro de una instalación Linux, con el fin de demostrar su buen funcionamiento. La fecha de presentación será previa al primer parcial.

DOCUMENTACIÓN

1. **Contenido**

   El cuerpo del documento deberá expresar con lenguaje claro los conceptos que se presentan, además deberá destacar aquellos archivos y/o registros de hardware que son modificados o consultados para el funcionamiento o descripción del proyecto.

   La memoria deberá en todos los casos tener un apartado destinado a las conclusiones del trabajo al igual que la bibliografía o curso que se hayan consultado.

   Los trabajos que no cumpliera con estos requisitos serán devueltos sin ser calificados, quedando como pendientes a ser presentados antes del fin del cuatrimestre.

2. **Formato de presentación**

   La memoria se realizara en Arial 11 espaciado simple, con tamaño de hoja A4, y márgenes Izq. y Sup.= 3 cm, Der. e Inf. = 2 cm. Los títulos principales estarán en negrita Arial 12. Deberá contener el nombre y apellido del alumno, además de su número de legajo. El archivo será del tipo Microsoft Word, el soporte en CD, e impreso.

INTRODUCCIÓN

Conceptos Básicos

Los semáforos son una herramienta básica de los sistemas operativos multitarea, permiten gestionar el acceso sincronizado (exclusión mutua) a la región crítica por parte de múltiples procesos. Desde la época en que fueron desarrollados por Dijkstra han evolucionado desde ser una variable con dos funciones asociadas hasta lo que son hoy; un mecanismo de conjuntos de semáforos conformado por estructuras de datos, ***system calls*** y código de usuario.

Cuando por la lógica de trabajo se utilizan dos o más recursos compartidos, pueden presentarse problemas de bloqueos cruzados, también llamados ***deadlock*** (abrazo mortal). Para evitar este conflicto los semáforos se implementan mediante conjuntos, en lugar de entidades independientes. Donde a cada recurso compartido se le asigna un semáforo. Es obvio que puede implementarse si es suficiente un conjunto con un único semáforo. También es imprescindible que las funciones asociadas (system calls) sean "***atómicas***". Esta característica les permite llevar a cabo TODA la tarea en el conjunto sin que en medio haya detenciones por medio del planificador de la CPU. Si no fuera así, por más que las operaciones se realicen por conjuntos la gestión sería inconsistente.

Los semáforos pueden ser inicializados con cualquier valor entero positivo (este valor indicará la cantidad de recursos disponibles). A veces es necesario utilizar un semáforo que como valor máximo admita un uno (1), a estos se los denomina "semáforos binarios".

El tipo de semáforos que se utilizará en este trabajo es la implementación **IPC UNIX System V** desarrollada por la empresa de comunicaciones AT&T. Las herramientas **IPC** (***I**nter **P**rocess **C**ommunication*) además de los semáforos son las colas de mensajes y las memorias compartidas. Estas herramientas son muy útiles a la hora de desarrollar aplicaciones que respondan al modelo **CLIENTE-SERVIDOR**.

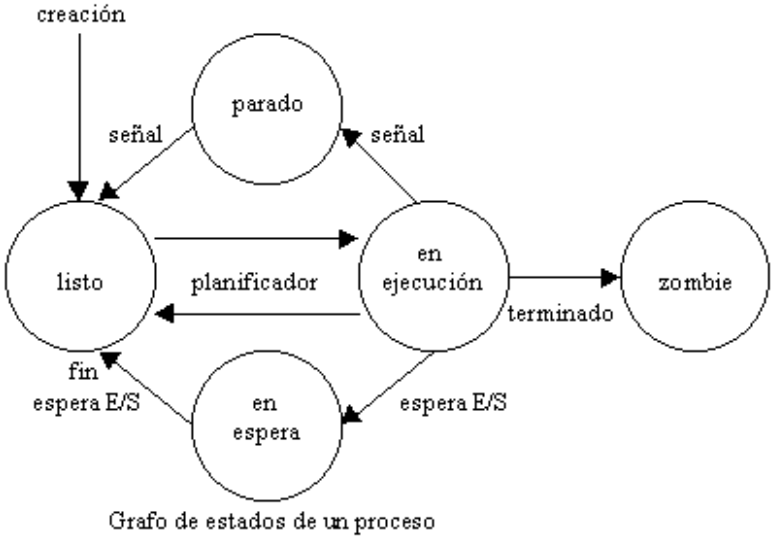
Lógica De Procesos

Para dar una idea de la mecanica de trabajo de los semáforos se explicará básicamente como es el funcionamiento. Cuando un proceso que está "***en ejecución***" necesita acceder a un recurso compartido ejecuta la ***system call*** correspondiente, esta verifica el semáforo asociado y según el valor que tenga el mismo puede pasar lo siguiente :

- Si el recurso **SI** está disponible (el contador del semáforo es >= al valor solicitado). El proceso toma el recurso y sigue en estado "***en ejecución***" hasta que el planificador de CPU resuelve sacarlo de ese estado.
- Si el recurso **NO** está disponible (el contador del semáforo es < al valor solicitado). El proceso se bloquea, pasa al estado "***en espera***" de que el (los) proceso(s) que previamente había(n) tomado recursos los devuelvan. Cuando se produce esto el kernel se encarga de pasar al proceso que estaba "***en espera***" al estado "***listo***". Tambien hay una posible instancia particular donde al no encontrar disponibilidad de recursos, el proceso puede seguir trabajando como si nunca hubiera verificado el semáforo (sólo en los casos en que no es imprescindible la utilizacion del recurso compartido). La salvedad en este caso es que cuando se devuelvan los recursos el kernel no realizará ninguna tarea con el proceso solicitante.

Introducción a las Estructuras de Datos

Primero tenemos que hacer una serie de aclaraciones con respecto a la lógica de trabajo del sistema operativo LiNUX. Cuando un proceso ejecuta código corriente trabaja en "modo usuario"; en el momento en que necesita ejecutar una llamada del sistema pasa a trabajar en "modo núcleo". Como las funciones que operan con semáforos son llamadas del sistema (por una cuestión de eficiencia y seguridad) las estructuras de datos "permanentes" referentes a los semáforos se encuentran en el área de memoria asignada al kernel del sistema. También se necesita contar con estructuras de datos en el área del usuario para poder ***enviar/recibir*** los datos resultantes de las operaciones realizadas.



Grafo de estados de un proceso

Estructuras de Datos del Kernel

La estructura de datos base es **SEMARY** (array de punteros). El objetivo de este array es almacenar la dirección de cada una de las estructuras **SEMID_DS** que definen a los conjuntos de semáforos. Los límites establecidos por el sistema para los conjuntos de semáforos están definidos en linux/sem.h, las siguientes son algunas de las más interesantes :

```
#define  SEMMNI     128 /* máximo número de identificadores de conjuntos          */
#define  SEMMSL      32 /* máximo número de semáforos por identificador           */
#define  SEMOPM      32 /* máximo número de operaciones por llamada semop         */
#define  SEMVMX   32767 /* máximo valor por semáforo                              */
```

La cantidad máxima de elementos que puede almacenar el array **SEMARY** está definida por SEMMNI. Y los valores para el índice van desde : (0 hasta (SEMMNI – 1)).

Estructura de datos **SEMID_DS** :

```
/* Hay una estructura semid_ds por cada juego de semáforos                       */
struct  semid_ds   {
   struct ipc_perm    sem_perm;          /* permisos ipc                         */
   time_t             sem_otime;         /* último instante semop (operación)    */
   time_t             sem_ctime;         /* último instante de cambio            */
   struct sem        *sem_base;          /* puntero al primer semáforo del array */
   struct sem_queue  *sem_pending;       /* primera operación pendiente          */
   struct sem_queue **sem_pending_last;  /* última operación pendiente           */
   struct sem_undo   *undo;              /* reestablecer por crash en proceso    */
   ushort             sem_nsems;         /* nro. de semáforos del array          */
                  };
```

Las operaciones con esta estructura son ejecutadas por llamadas especiales al sistema, y no se deben usar directamente. Aquí tenemos las descripciones de los campos más interesantes:

sem_perm	Este es un caso de la estructura **ipc_perm**, que se define en linux/ipc.h. Guarda la información de los permisos para el conjunto de semáforos, incluyendo permisos de acceso, información sobre el creador (y su grupo) y el identificador **IPC** del conjunto.
sem_otime	Instante de la última operación **semop**().
sem_ctime	Instante del último cambio de modo.
sem_base	Puntero al primer semáforo del array (*ver siguiente estructura*).
sem_pending	Puntero a la primera operación pendiente.
sem_pending_last	Puntero a la última operación pendiente.
sem_undo	Puntero a la lista de estructuras “deshacer” en el conjunto.
sem_nsems	Número de semáforos en el conjunto (cantidad de elementos en el array apuntado por sem_base).

Estructura de datos **SEM** :

En la estructura **SEMID_DS**, hay un puntero (sem_base) a la base del array de semáforos. Cada miembro del array es del tipo estructura **SEM**. También se define en linux/sem.h:

```
/* Una estructura por cada semáforo en el conjunto                              */
struct  sem    {
   short   sempid;   /* pid de última operación                                 */
   ushort  semval;   /* valor actual                                            */
              };
```

sem_pid	El **PID** (identificador del proceso) que realizó la última operación
sem_semval	Valor actual del semáforo

Estructura de datos **SEM_UNDO** :

Cada tarea puede tener una lista de operaciones anulables “undo”. Estas son ejecutadas por el kernel si el proceso por algún motivo termina de forma anormal.

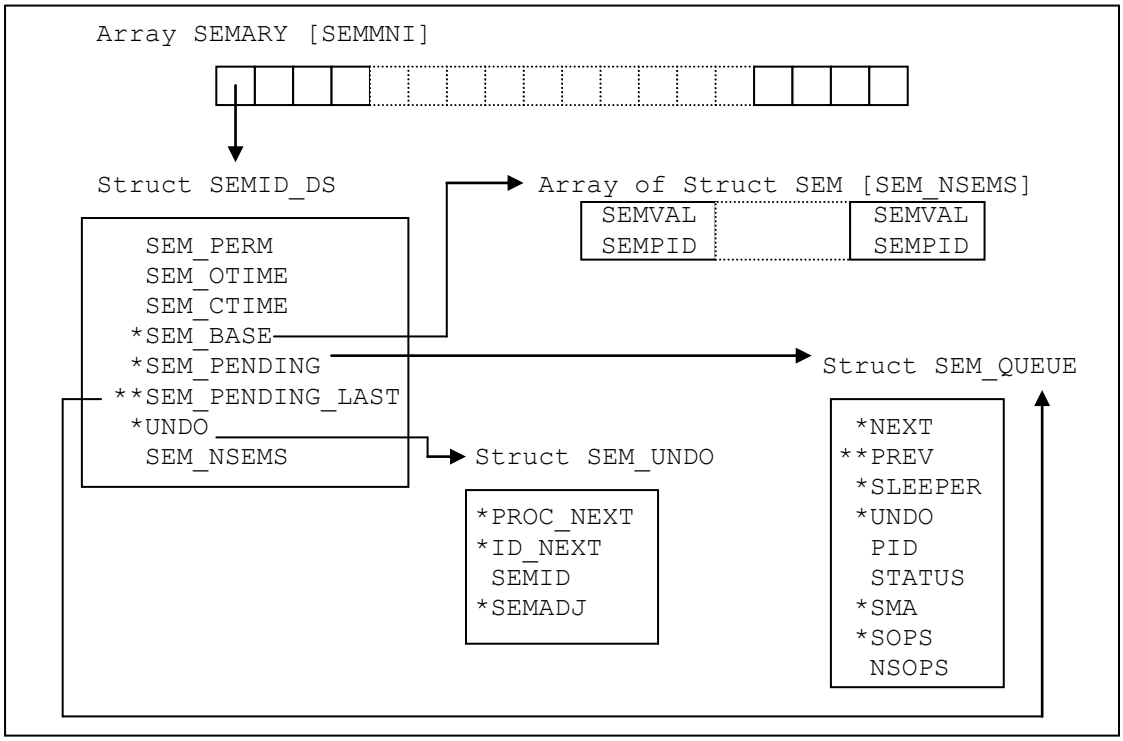
```
struct sem_undo {
    struct sem_undo  *proc_next; /* próxima entrada en este proceso                    */
    struct sem_undo  *id_next;   /* próxima entrada en este conjunto                   */
    int               semid;     /* identificador de conjunto                          */
    short            *semadj;    /* puntero al elemento de de rescate, uno por semáforo */
                };
```

Estructura de datos **SEM_QUEUE** :

Estructura para la cola de operaciones pendientes a realizar en el conjunto de semáforos. Esta cola se administra como una lista doblemente enlazada.

```
/* Una cola por cada conjunto de semáforos en el sistema                              */
struct sem_queue {
   struct sem_queue   *next;     /* primera entrada en la cola                        */
   struct sem_queue  **prev;     /* última entrada en la cola                         */
   struct wait_queue  *sleeper;  /* proceso en espera                                 */
   struct sem_undo    *undo;     /* estructura “undo”                                 */
   int                 pid;      /* id. de proceso solicitante                        */
   int                 status;   /* status de operación                               */
   struct semid_ds    *sma;      /* array de semáforos para operaciones               */
   struct sembuf      *sops;     /* array de operaciones pendientes                   */
   int                 nsops;    /* número de operaciones                             */
               };
```

Esquema de relación de las estructuras de datos del kernel



Estructuras de Datos del Usuario

Estructura de datos **SEMBUF** :

```
/* La llamada del sistema semop utiliza un array de este tipo                */
struct sembuf {
   ushort  sem_num;        /*  índice del semáforo en el array               */
   short   sem_op;         /*  operación a realizar en el semáforo           */
   short   sem_flg;        /*  flag de la operación                          */
              };
```

sem_num Número de semáforo sobre el que se desea actuar.

sem_op Operación a realizar (positiva, negativa o cero).

sem_flg Flags (parámetros) de la operación. Los valores permitidos son: **IPC_NOWAIT** y **SEM_UNDO**.

Estructura de datos **SEMUN** :

```
/* arg para la llamada del sistema semctl                                    */
union semun {
  int               val;     /* valor para SETVAL                           */
  struct semid_ds  *buf;     /* buffer para IPC_STAT e IPC_SET              */
  ushort           *array;   /* array para GETALL y SETALL                  */
  struct seminfo   *__buf;   /* buffer para IPC_INFO                        */
  void             *__pad;
           };
```

val Se usa con el comando SETVAL, para indicar el valor a poner en el semáforo.

Buf Se usa con los comandos IPC_STAT/IPC_SET. Es como una copia de la estructura de datos interna que tiene el kernel para los semáforos.

Array Puntero que se usa en los comandos GETALL/SETALL. Debe apuntar a un array de números enteros donde se ponen o recuperan valores de los semáforos.

Los demás argumentos, **_buf** y **_pad**, se usan internamente en el kernel y no son de excesiva utilidad para el programador. Además son específicos para el sistema operativo **L*i*NUX** y no se encuentran en otras versiones de **UNIX**.

Objetos IPC

Cada objeto **IPC** tiene un único identificador asociado a él. Se usa este identificador, dentro del kernel, para identificar de forma única al objeto.

La unicidad de un identificador es importante según el tipo de objeto en cuestión. El identificador puede repetirse para distintos tipos de **IPC** (colas de mensajes, semáforos, memorias compartidas). Mientras que no puede haber nunca dos objetos del mismo tipo **IPC** con el mismo identificador.

Claves IPC

Para obtener un identificador único, debe utilizarse una clave. Esta debe ser conocida por ambos procesos, cliente y servidor. Este es el primer paso para construir el entorno cliente/servidor de una aplicación.

La clave puede ser el mismo valor cada vez, incluyéndolo en el código de la propia aplicación. Esta es una desventaja pues la clave requerida puede estar ya en uso. Por eso, es necesario asegurar que el programa pueda generar claves no utilizadas ya en el sistema.

El algoritmo de generación de la clave usado está completamente a elección del programador de la aplicación. Mientras que tome medidas para prevenir las condiciones críticas, bloqueos, etc., cualquier método es correcto. En general se utiliza la función **ftok**() para esta tarea.

FUNCIÓN DE LIBRERÍA: ftok();

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

PROTOTIPO: `key_t` **`ftok`** `(char *nombre, char proj);`

REGRESA: si éxito, nueva clave **IPC**
si hubo error, -1; dejando **errno** con el valor de la llamada **stat()**

La clave del valor devuelto de **ftok**() se genera por la combinación del número del inodo y del número de dispositivo del archivo argumento, con el carácter identificador del proyecto del segundo argumento. Esto no garantiza la unicidad, pero la aplicación puede comprobar las colisiones y reintentar la generación de la clave.

```
key_t    miclave;
miclave  =  ftok("/tmp/aplicaciones",  'a');
```

En el caso anterior el directorio `/tmp/aplicaciones` se combina con la letra `'a'`. Otro ejemplo común es usar el directorio actual:

```
key_t    miclave;
mykey  =  ftok(".",  'a');
```

LLAMADA AL SISTEMA: semget()

Descripción: Se usa para ***crear*** un nuevo conjunto de semáforos o para ***acceder*** a uno existente.

PROTOTIPO:

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

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

REGRESA:
SI éxito: Identificador **IPC** del conjunto.
SI error: -1 y **errno** contiene uno de los siguientes códigos :

EACCESS	(permiso denegado)
EEXIST	(no puede crearse pues existe (IPC_EXCL))
EIDRM	(conjunto marcado para borrarse)
ENOENT	(no existe el conjunto ni se indico IPC_CREAT)
ENOMEM	(No hay memoria suficiente para crear)
ENOSPC	(Límite de conjuntos excedido)

NOTAS:

El primer argumento de **semget**() es el valor clave (en nuestro caso devuelto por la llamada a la función **ftok**()). Este valor clave se compara con los valores claves existentes en el kernel para otros conjuntos de semáforos. Ahora, las operaciones de apertura o acceso dependen del contenido del argumento **semflg**.

IPC_CREAT Crea el juego de semáforos si no existe ya en el kernel.

IPC_EXCL Al usarlo con IPC_CREAT, falla si el conjunto de semáforos existe ya.

Si se usa **IPC_CREAT** sólo, **semget**(), devuelve el identificador del semáforo para un conjunto nuevo creado, o devuelve el identificador para un conjunto que existe con el mismo valor clave. Si se usa **IPC_EXCL** junto con **IPC_CREAT**, entonces se crea un conjunto nuevo, o si el conjunto existe, la llamada falla con -1. **IPC_EXCL** es inútil por si mismo, pero cuando se combina con **IPC_CREAT**, se puede usar para garantizar que ningún semáforo existente se abra accidentalmente para accederlo.

Como sucede en otros puntos del **IPC UNIX System V**, puede aplicarse a los parámetros anteriores, un número octal para dar la máscara de permisos de acceso de los semáforos. Debe hacerse con una operación **OR** binaria.

El argumento **nsems** especifica el número de semáforos que se deben crear en un conjunto nuevo. Observe que el argumento **nsems** se ignora si abre un conjunto ya existente.

El siguiente es un ejemplo del uso de **semget()** para abrir un conjunto de semáforos en forma **NO** exclusiva.

```
int abrir_conjunto_de_semaforos (key_t clave, int cantsem)
{
     int sid;
     if (! cantsem)
         return(-1);
     if ((sid = semget(clave, cantsem, IPC_CREAT | 0660)) == -1)
         return(-1);
     return (sid);
}
```

Vea que se usan explícitamente los permisos 0660. Esta pequeña función retornará, bien un identificador entero del conjunto de semáforos, o bien un -1 si hubo un error.

LLAMADA AL SISTEMA: semop()

Descripción: Se utiliza para realizar las ***operaciones*** solicitadas sobre el conjunto de semáforos.

PROTOTIPO :

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

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

REGRESA :
si éxito : 0 (todas las operaciones realizadas)
si error : -1 y **errno** contiene uno de los siguientes códigos :

E2BIG	(nsops mayor que max. número de opers. permitidas atómicamente)
EACCESS	(permiso denegado)
EAGAIN	(IPC_NOWAIT incluido, la operación no termino)
EFAULT	(dirección no válida en el parámetro sops)
EIDRM	(el conj. de semáforos fue borrado)
EINTR	(Recibida señal durante la espera)
EINVAL	(el conj. no existe, o semid inválido)
ERANGE	(valor del semáforo fuera de rango)
ENOMEM	(SEM_UNDO incluido, sin memoria suficiente para crear la estructura de retroceso necesaria)

NOTAS :

El primer argumento de **semop**() es el valor clave (en nuestro caso devuelto por una llamada a **semget**()). El segundo argumento (sops) es un puntero a un array de operaciones para que se ejecute en el conjunto de semáforos, mientras el tercer argumento (nsops) es el número de operaciones en ese array. El argumento **sops** apunta a un array del tipo **sembuf**.

```
struct sembuf {
   ushort  sem_num;        /* índice del semáforo en el array del conjunto                    */
   short   sem_op;         /* operación a realizar en el semáforo (positiva, negativa o cero) */
   short   sem_flg;        /* flag de la operación  (IPC_NOWAIT o SEM_UNDO)                   */
             };
```

Si **sem_op** es negativo (*operación **DOWN***), entonces su valor se resta del valor del semáforo. Si el contenido de **sem_flg** es **SEM_UNDO**, y el valor del semáforo es menor al solicitado, entonces el proceso que efectúa la llamada duerme hasta que los recursos solicitados estén disponibles. Además, en caso de que el proceso tenga un final forzado; el kernel puede restablecer el valor del semáforo al estado previo a la llamada, gracias a la información previamente

guardada. Si el contenido de **sem_flg** es **IPC_NOWAIT** y el valor del semáforo es menor al solicitado por el proceso, este sigue su ejecución sin realizar ninguna espera sobre el semáforo.

Si **sem_op** es positivo (*operación **UP***), entonces su valor se suma al contador del semáforo. ***¡Siempre se deben devolver los recursos al conjunto de semáforos cuando ya no se necesitan más!***

Finalmente, si **sem_op** vale cero (0), entonces el proceso que efectúa la llamada esperará hasta que el valor del semáforo sea 0.

EJEMPLOS DE UTILIZACIÓN

Creamos una estructura de nombre "**bloqueo_sem**" de tipo **sembuf** con los siguientes datos :

```
struct sembuf bloqueo_sem = {0, -1, IPC_NOWAIT};
```

La traducción de la inicialización de la anterior estructura indica que un valor de "-1" se sumará al semáforo número 0 del conjunto. En otras palabras, se obtendrá una unidad de recursos del primer semáforo de nuestro conjunto (índice 0). Se especifica **IPC_NOWAIT**, así la llamada o se produce inmediatamente, o falla si otro trabajo está activo en ese momento. Aquí hay un ejemplo de como usar esta inicialización de la estructura **sembuf** con la llamada al sistema **semop**():

```
if ((semop (sid, &bloqueo_sem, 1) == -1)
     perror ("semop");
```

El tercer argumento **"nsops"** (en este caso : 1) dice que estamos sólo ejecutando una (1) operación (hay sólo una estructura **sembuf** en nuestro array de operaciones). El argumento **"sid"** es el identificador **IPC** para nuestro conjunto de semáforos.

Cuando la tarea del proceso ha terminado, debemos devolver el recurso al conjunto de semáforos, de manera que otros procesos (si existen) puedan utilizar el recurso compartido.

```
struct sembuf desbloqueo_sem = {0, 1, IPC_NOWAIT};
```

La traducción de la estructura anteriormente inicializada indica que un valor de "1" se sumará al semáforo número 0 en el conjunto. En otras palabras, una unidad del recurso se devuelve al semáforo.

LLAMADA AL SISTEMA: semctl()

Descripción: Realiza operaciones de control sobre conjuntos de semáforos.

PROTOTIPO

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

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

REGRESA:

si éxito : entero positivo

si error : -1 y **errno** contiene uno de los siguientes códigos :

EACCESS	(permiso denegado)
EFAULT	(dirección invalida en el argumento arg)
EIDRM	(el juego de semáforos fue borrado)
EINVAL	(el conj. no existe, o **semid** no es valido)
EPERM	(El **EUID** no tiene privilegios para el comando incluido en arg)
ERANGE	(Valor para semáforo fuera de rango)

La llamada al sistema **semctl** se usa para realizar operaciones de control sobre un conjunto de semáforos. Por esta causa, no sólo se necesita pasar la clave **IPC**, sino también el semáforo destino dentro del conjunto.

Esta llamada utiliza un argumento **cmd**, para la especificación del comando a ejecutar sobre el objeto **IPC**. Los valores permitidos son :

IPC_STAT	Obtiene la estructura **semid_ds** de un conjunto y la guarda en la dirección del argumento **buf** en la union **semun**.
IPC_SET	Establece el valor del miembro **ipc_perm** de la estructura **semid_ds** de un conjunto. Obtiene los valores del argumento **buf** de la union **semun**.
IPC_RMID	Elimina el conjunto de semáforos.
GETALL	Se usa para obtener los valores de todos los semáforos del conjunto. Los valores enteros se almacenan en un array de enteros cortos sin signo, apuntado por el miembro array de la unión.
GETNCNT	Devuelve el número de procesos que esperan recursos.
GETPID	Retorna el **PID** del proceso que realizó la última llamada **semop**.
GETVAL	Devuelve el valor de uno de los semáforos del conjunto.
GETZCNT	Devuelve el número de procesos que esperan la utilización del 100% de los recursos.
SETALL	Coloca todos los valores de semáforos con una serie de valores contenidos en el miembro array de la unión.
SETVAL	Coloca el valor de un semáforo individual con el miembro **val** de la unión.

El argumento **arg** representa un ejemplo de tipo **semun**. Esta unión particular se declara en linux/sem.h.

```
/*  argumento  para  llamadas  a  semctl                              */
  union  semun {
                int val;              /*  valor  para  SETVAL          */
                struct semid_ds *buf; /* buffer para IPC_STAT e IPC_SET */
                ushort *array;        /* array para GETALL y SETALL     */
                struct seminfo *_buf; /* buffer para IPC_INFO           */
                void *_pad;
               };
```

La siguiente función devuelve el valor del semáforo indicado. El último argumento de la llamada (la unión), es ignorada con el comando **GETVAL** por lo que no la incluimos:

```
int obtener_valor_sem(int sid, int idx_sem)
{
    return( semctl (sid, idx_sem, GETVAL, 0));
}
```

Utilicemos esta función para mostrar los valores de un ficticio conjunto de 5 semáforos.

```
 #define MAX 5
 uso_funcion()
 /* resolvamos que sid es el identificador del conjunto en cuestión */
    {
     int  idx;
     for(idx=0; idx<MAX; idx++)
        printf ("Impresora %d: %d\n\r", idx, obtener_valor_sem(sid, idx));
    }
```

Considérese la siguiente función, que se debe usar para iniciar un nuevo semáforo:

```
void iniciar_sem (int sid, int idx_sem, int initval)
         {
           union semun sem_opciones;

           sem_opciones.val = initval;
           semctl(sid, idx_sem, SETVAL, sem_opciones);
         }
```

Observe que el argumento final de **semctl** es una copia de la unión, más que un puntero a él.

Recuerde que los comandos **IPC_SET/IPC_STAT** usan el miembro **buf** de la unión, que es un puntero al tipo **semid_ds**. El miembro **buf** debe indicar una ubicación válida de almacenamiento para que nuestra función trabaje adecuadamente.

DESCRIPCIÓN DEL CÓDIGO SUMINISTRADO

La mejor forma de asimilar conceptos teóricos de esta naturaleza es desarrollar código y hacer las pruebas que se consideren necesarias. En el apéndice "**Código**" se suministran cuatro (4) programas de acuerdo al siguiente detalle :

`USEM`	Programa en lenguaje **C** que implementa todas las funciones comentadas en este trabajo.
`CREA-SEM.SH`	Programa **shell script** de LiNUX. Crea el conjunto IPC.
`TOMA-SEM.SH`	Programa **shell script** de LiNUX. Bloquea el proceso y permite al usuario definir el momento en que quiere desbloquear al proceso.
`TRABAJO.SH`	Programa **shell script** de LiNUX. Se recomienda analizar este programa como ejemplo de utilización de **`USEM`** con la opción **`A`**.

El programa **USEM** implementa las funciones correspondientes a la administración de SEMÁFOROS UNIX IPC System V. Para hacer más accesible el entendimiento se han preparado también los shell scripts mencionados anteriormente con dos simples objetivos :

1) permitir una utilización básica correcta del programa **USEM**.
2) introducir brevemente a la programación de shell scripts vinculados con programas en lenguaje **C**.

El programa **USEM** tiene los siguientes comandos disponibles :

- **`C`** Crea un conjunto IPC con un único semáforo de valor máximo uno (1). El objetivo de esta limitación es presentar un caso claro y simple que permita con un mínimo de tiempo entender cabalmente la metodología de trabajo en estudio.
- **`T`** Bloquea el semáforo (si existe la posibilidad) y queda esperando a que el usuario pulse ENTER para después desbloquearlo. La intención de esto es bridarle al usuario la posibilidad de que elija él el momento en que quiere que el bloqueo cese, mientras el proceso que ejecuta esta acción se bloquea; desde otra consola (lo más práctico sería una consola virtual) puede comprobar que cualquier acción sobre el semáforo que realice otro proceso quedará en espera hasta que el proceso mencionado en primer término finalice.
- **`E`** Esta opción utiliza una alternativa prácticamente no documentada de **IPC**. Es la espera a que el valor del semáforo seleccionado sea cero (0). La lógica de trabajo es crear primero el conjunto del unico semáforo, ejecutar esta opción (en la consola origen podrá comprobar que el proceso se bloquea) y desde otra consola ejecutar el programa con los comandos **`T`** o **`A`** (que ya se analizará). Hasta que no haya terminado el proceso en la segunda consola el primer proceso (utilizado con la opción **`E`**) no terminará.
- **`B`** Permite borrar (eliminar) el conjunto **IPC**.
- **`A`** Esta es la opción más útil, porque puede utilizarse con dos motivos muy distintos de acuerdo a los parámetros que se le pasen. Sin parámetros adicionales se bloquea (si existe la posibilidad) y se desbloquea inmediatamente. Esta acción sólo es útil complementándose con otro proceso que actúe en forma relacionada, como en el caso del comando **`E`**. También brinda la posibilidad de grabar en un archivo (a elección del usuario) una línea de texto. Esta grabación se realiza en forma exclusiva gracias a que el acceso al archivo está protegido por la sección crítica del programa. La línea de ejecución en este caso es : **`USEM A ARCHIVO_DESTINO LÍNEA_A_GRABAR`**. Es imprescindible contar con los permisos necesarios de acuerdo al archivo (y a su path) elegido.

```
/*************************************************************************/
/* - usem.c - Utilitario básico para semáforos IPC System V -            */
/*************************************************************************/

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEM_MAX_RECURSO    1   /* Valor inicial de todos los semáforos */
#define INI                0   /* Índice del primer semáforo           */

void abrir_sem(int *sid, key_t clave);
void crear_sem(int *sid, key_t clave, int idx);
void bloquear_sem(int clave, int idx);
void zero_espera_sem(int clave, int idx);
void desbloquear_sem(int clave, int idx);
void remover_sem(int sid);
void uso(void);

int main(int argc, char *argv[])
{
             key_t clave;
             int   semset_id;
             FILE *fptr;

             if(argc == 1)
                 uso();

             /* Crea una clave única mediante ftok() */
             clave = ftok(".", 'u');

             switch(tolower(argv[1][0]))
                  {
                   case 'c': if(argc != 2)
                                uso();
                             crear_sem(&semset_id, clave, 1);
                             break;
                   case 't': abrir_sem(&semset_id, clave);
                             bloquear_sem(semset_id, INI);
                             getchar();
                             desbloquear_sem(semset_id, INI);
                             break;
                   case 'e': zero_espera_sem(semset_id, INI);
                             break;
                   case 'b': abrir_sem(&semset_id, clave);
                             remover_sem(semset_id);
                             break;
                   case 'a': abrir_sem(&semset_id, clave);
                             bloquear_sem (semset_id, INI);
                             if ((fptr = fopen(argv[2], "a")) == NULL)
                                 exit (-1);
                             else
                                {
                                 fprintf(fptr, "%s\n", argv[3]);
                                 fclose(fptr);
                                }
                             desbloquear_sem (semset_id, INI);
                             break;
                    default: uso();
                  }

             return(0);
}
```

```
void abrir_sem(int *sid, key_t clave)
{
   /* Abre el conjunto de semáforos             */

   if((*sid = semget(clave, 0, 0666)) == -1)
     {
      printf("El conjunto de semáforos NO existe!\n");
      exit(1);
     }
}

void crear_sem(int *sid, key_t clave, int cantidad)
{
   int cntr;
   union semun semopciones;

   if(cantidad > SEMMSL)
     {
      printf("ERROR : cant. máx. de sem. en el conjunto es %d\n", SEMMSL);
      exit(1);
     }

   printf("Creando nuevo conjunto IPC con %d semáforo(s)...\n",cantidad);

   if((*sid = semget(clave, cantidad, IPC_CREAT|IPC_EXCL|0666)) == -1)
     {
      fprintf(stderr, "Ya existe un conjunto con esta clave!\n");
      exit(1);
     }

   printf("Nuevo conjunto IPC de sem. creado con éxito\n");

   semopciones.val = SEM_MAX_RECURSO;

   /* Inicializa todos los semáforos del conjunto */
   for(cntr=0; cntr<cantidad; cntr++)
       semctl(*sid, cntr, SETVAL, semopciones);
}

void bloquear_sem(int sid, int idx)
{
   struct sembuf sembloqueo={ 0, -1, SEM_UNDO};

   sembloqueo.sem_num = idx;

   if((semop(sid, &sembloqueo, 1)) == -1)
     {
      fprintf(stderr, "El Bloqueo falló\n");
      exit(1);
     }
}

void zero_espera_sem(int sid, int idx)
{
   struct sembuf esperazero={ 0,  0, SEM_UNDO};
   esperazero.sem_num = idx;

   printf("Proceso a la ESPERA de valor CERO en semáforo IPC...\n");

   if((semop(sid, &esperazero, 1)) == -1)
     {
      fprintf(stderr, "La Espera NO pudo establecerse \n");
      fprintf(stderr, "Valor de ERRNO : %d \n", errno);
      exit(1);
     }
   printf("ESPERA concluída.  Terminación del proceso.\n");
}
```

```
void desbloquear_sem(int sid, int idx)
{
   struct sembuf semdesbloqueo={ 0, 1, SEM_UNDO};

   semdesbloqueo.sem_num = idx;

   /* Intento de desbloquear el semáforo */
   if((semop(sid, &semdesbloqueo, 1)) == -1)
     {
      fprintf(stderr, "El Desbloqueo falló\n");
      exit(1);
     }
}

void remover_sem(int sid)
{
   semctl(sid, 0, IPC_RMID, 0);
   printf("Conjunto de semáforos eliminado\n");
}

void uso(void)
{
   fprintf(stderr, " - usem - Utilitario básico para semáforos IPC   \n");
   fprintf(stderr, "    USO : usem    (c)rear                        \n");
   fprintf(stderr, "                  (t)oma recurso compartido      \n");
   fprintf(stderr, "                  (e)spera IPC de valor cero     \n");
   fprintf(stderr, "                  (b)orrar                       \n");
   fprintf(stderr, "                  (a)gregar <PATH-DB> <LINE>     \n");
   exit(1);
}
/***************************** fín de usem.c *********************************/
```

shell script crea-sem.sh

```
#!/bin/sh
clear
echo "-------------------------------------------------------"
echo " Crea un Conjunto de un único Semáforo IPC System V  "
echo "-------------------------------------------------------"
./usem c
echo "------------------< FIN de crea-sem.sh >-------------"
```

shell script toma-sem.sh

```
#!/bin/sh
clear
echo "-------------------------------------------------------"
echo " Pulse <ENTER> cuando desee desbloquear el SEMÁFORO  "
echo "-------------------------------------------------------"
./usem t
echo "------------------< FIN de toma-sem.sh >-------------"
```

shell script trabajo.sh

```
#!/bin/sh
clear
archivo="p-usem.dat"
Narchivo="./$archivo"
echo "-------------------------------------------------------"
echo " 1) Ingrese una línea de texto para probar el código "
echo " 2) Compruebe que el texto que Ud. ingresó figure en "
echo "    el archivo : $archivo                             "
echo "-------------------------------------------------------"
read Param
./usem a $Narchivo $Param
echo "------------------< Trabajo Realizado >-------------"
```

CONCLUSIÓN

El objetivo de este trabajo fue hacer una introducción amplia pero simple de los SEMÁFOROS UNIX IPC System V, mecanismo de sincronización de procesos en sistemas operativos multiusuarios. La intención fue aclarar cual es la relación entre el kernel, las estructuras de datos intervinientes y los procedimientos asociados. Obviamente era imprescindible incluír código que permitiera asimilar los conceptos teóricos brindados. Y en este aspecto, se priorizó la idea de permitir con un mínimo de código brindar la posibilidad de hacer una gran serie de pruebas. No hay mejor forma de entender las particularidades de un mecanismo de este tipo que la de probar y modificar los programas brindados exhaustivamente.

El presente trabajo está basado en el estudio realizado con motivo de resolver problemas de sincronización en aplicaciones CLIENTE-SERVIDOR locales. El trabajo consistía en realizar con código propio (en lenguaje C) el programa SERVIDOR y los programas CLIENTES en C y en Shell Script BASH de LiNUX. Específicamente la tarea era brindar el servicio de decodificación, procesamiento, almacenamiento y respuesta dinámica de información recibida a través de un servidor WEB. La decisión de realizar los programas con código propio se debió a que una de las restricciones del trabajo era lograr que los programas fueran lo más compactos y livianos posibles.

Para el objetivo planteado los semáforos resultaron ser una herramienta consistente y eficiente. Ampliamente recomendable en administración de transacciones de ámbito local.

Como sugerencia para el lector interesado se deja la idea de implementar en un entorno de aplicaciones para Internet (servidor WEB) o para cualquier otra aplicación CLIENTE-SERVIDOR local el mecanismo de sincronización de SEMÁFOROS UNIX IPC System V.