SISTEMAS OPERATIVOS

LABORATORIO 3

Grupo 16
Integrantes

PERES FERRE, MATIAS FABRIZIO	bdi_mail@hotmail.com	**CI:** 4474045
MUJICA CAZENAVE, FEDERICO	fedes15@hotmail.com	**CI:** 4786543
CRUZ TERMEIRO, DARIO DANIEL	dariodcruz@gmail.com	**CI:** 4768599
RUIZ RAVIALES, GERMAN ALEJANDRO	germanruizravi@gmail.com	**CI:** 4317743

Indice

1 Introducción

En el presente trabajo nos encargamos de realizar la resolución del Laboratorio número tres de la asignatura Sistemas Operativos.

Al igual que en el segundo Obligatorio, la elaboración del trabajo se ha enfocado de manera tal que sirva como guía para el lector, no solo como solución inmediata de los ejercicios, sino necesario, relevante, mediante el cual se pueda lograr una comprensión total y detallada de las cláusulas establecidas y de las soluciones.

Nuestros objetivos son: cumplir con las cláusulas, incorporar estrategias distintas a las utilizadas en el segundo obligatorio, profundizar en la utilización de los conceptos aprendidos, realizar una reseña introductoria a la faceta teórica e inducir a la comprensión de las decisiones de implementación. Al mismo tiempo, es de nuestro interés no redundar en información ajena a los conocimientos requeridos para la resolución del obligatorio.

En la primera sección realizamos un acercamiento al marco teórico de la creación de procesos y de los métodos IPC. Su función y manipulación. Intentamos focalizar la mirada en los conceptos tratados más adelante.

En la segunda sección del obligatorio se describe el diseño optado para el desarrollo de la agenda telefónica SoAgenda y el protocolo de comunicación entre el servidor y los clientes. Realizamos las observaciones necesarias en cuánto a detalles técnicos, siguiendo las bases establecidas en la teoría e introduciendo cambios y decisiones. Es importante señalar que en ningún momento nos apartamos de las cláusulas establecidas por el obligatorio.

Procedemos con el testeo no exhaustivo de la implementación concentrándonos en casos puntuales hasta poder afirmar con un alto de grado de confianza el correcto funcionamiento de la agenda.

El trabajo se encuentra respaldado por material bibliográfico confiable que ha sido seleccionado con atención, a efectos de que la información en la cual nos basemos para la realización del mismo sea certera, lo que brinda constantemente una seguridad en la lectura.

El proceso de elaboración del trabajo consistió en la división temática del mismo, asignando a cada uno de los integrantes del equipo una o varias secciones determinadas, teniendo en cuenta que el conocimiento sea general y llegando a una puesta en común antes de la división.

2 Creación de procesos - Fork

2.1 Procesos, hilos

Antes de hablar de la llamada al sistema fork propiamente dicha, conviene hablar sobre procesos e hilos. Dos conceptos muy parecidos y relacionados, pero con un conjunto de sutiles diferencias.

Uno de los principales motivos de la existencia de la informática es imitar el comportamiento de la mente humana. En un comienzo surgieron los algoritmos, que no son más que una secuencia de pasos para conseguir un objetivo, a partir de los cuales surgió el pensamiento de "por qué no hacer varias cosas a la vez" y es precisamente de esta inquietud de donde surgen los hilos o threads.

Si queremos que nuestro programa empiece a ejecutar varias cosas "a la vez", tenemos dos opciones. Por una parte podemos crear un nuevo proceso y por otra, podemos crear un nuevo hilo de ejecución (un thread). En realidad nuestro ordenador, salvo que tenga varias CPU's, no ejecutará varias tareas a la vez. Esto se refiere a que el sistema operativo, en este caso Linux, irá ejecutando los threads según la política del mismo, siendo lo más usual mediante rodajas de tiempo muy rápidas que dan la sensación de simultaneidad.

En un sistema Linux, que como ya sabemos es multitarea (sistema operativo multihilo), se pueden estar ejecutando distintas acciones a la par, y cada acción es un proceso que consta de uno o más hilos, memoria de trabajo compartida por todos los hilos e información de planificación. Cada hilo consta de instrucciones y estado de ejecución.

Cuando ejecutamos un comando en el shell, sus instrucciones se copian en algún sitio de la memoria RAM del sistema para ser ejecutadas. Cuando las instrucciones ya cumplieron su cometido, el programa es borrado de la memoria del sistema, dejándola libre para que más programas se puedan ejecutar a la vez. Cada uno de estos programas en ejecución son los procesos.

Los procesos son creados y destruidos por el sistema operativo, pero lo hace a petición de otros procesos. El mecanismo por el cual un proceso crea otro proceso se denomina bifurcación (fork). Los nuevos procesos son independientes y no comparten memoria (es decir, información) con el proceso que los ha creado.

En definitiva, es posible crear tanto hilos como procesos. La diferencia estriba en que un proceso solamente puede crear hilos para sí mismo y en que

dichos hilos comparten toda la memoria reservada para el proceso.

Los hilos son similares a los procesos ya que ambos representan una secuencia simple de instrucciones ejecutada en paralelo con otras secuencias. Los hilos son una forma de dividir un programa en dos o más tareas que corren simultáneamente, compitiendo, en algunos casos, por la CPU.

La diferencia más significativa entre los procesos y los hilos, es que los primeros son típicamente independientes, llevan bastante información de estados, e interactúan sólo a través de mecanismos de comunicación dados por el sistema. Por otra parte, los hilos generalmente comparten la memoria, es decir, acceden a las mismas variables globales o dinámicas, por lo que no necesitan costosos mecanismos de comunicación para sincronizarse.

2.2 Creación de procesos

A la hora de crear procesos linux provee de dos funciones para dicho cometido, la función *clone()* y la función *fork()*. Ambas crean un nuevo proceso a partir del proceso padre pero de una manera distinta.

Cuando utilizamos la llamada al sistema *fork*, el proceso hijo creado es una copia exacta del padre (salvo por el PID y la memoria que ocupa). Al proceso hijo se le facilita una copia de las variables del proceso padre y de los descriptores de fichero. Es importante destacar que las variables del proceso hijo son una copia de las del padre (no se refieren físicamente a la misma variable), por lo que modificar una variable en uno de los procesos no se refleja en el otro.

La llamada al sistema *clone* es mucho más genérica y flexible que el *fork*, ya que nos permite definir qué van a compartir los procesos padre e hijo.

Los procesos en Linux tienen una estructura jerárquica, es decir, un proceso padre puede crear un nuevo proceso hijo y así sucesivamente. Cuando se hace un *fork,* se crea un nuevo *task_struct* a partir del *task_struct* del proceso padre. Al hijo se le asigna un PID propio y se le copian las variables del proceso padre. Sin embargo, en la llamada al sistema *clone*, el *task_struct* del proceso padre se copia y se deja tal cual, por lo que el hijo tendrá el mismo PID que el proceso padre y obtendrá (físicamente) las mismas variables que el proceso padre. El proceso hijo creado es una copia del padre (mismas instrucciones, misma memoria). En cuanto al valor devuelto por el *fork*, se trata de un valor numérico que depende tanto de si el *fork* se ha ejecutado correctamente como de si nos encontramos en el proceso padre o en el hijo.

- Si se produce algún error en la ejecución del *fork*, el valor devuelto es -1.
- Si no se produce ningún error y nos encontramos en el proceso hijo, el *fork* devuelve un 0.
- Si no se produce ningún error y nos encontramos en el proceso padre, el *fork* devuelve el PID asignado al proceso hijo.

Veamos un fragmento del código:

```
pid_t pid;
pid = fork();

if(pid == -1)
{
        cout<<"Error al crear el proceso hijo"<<endl;
        exit(1);
}
else if(pid == 0)
{
        //proceso hijo
        ...
}
else
{
        //proceso padre
        idHijo = pid;
        ...
}
```

Como podemos observar, fácilmente asignamos porciones de código distintas a ejecutar, luego de la creación del proceso hijo. Un paso que se debe resaltar es la asignación en el código del proceso padre. Se almacena en *idHijo* lo que devuelve la función *fork()*, de manera de recordar el id del proceso hijo creado. El id será utilizado para enviar una señal al proceso hijo al salir del sistema, como veremos más adelante.

2.3 Handlers

Tras ver cómo se pueden generar varios procesos empleando *fork* ahora veremos cómo podemos controlar sus tiempos de ejecución con *wait*, *waitpid* y *waitid*.

Esta es la firma de las dos primeras funciones:

*pid_t wait(int *status)*

*pid_t waitpid(pid_t pid, int *status, int options)*

Estas funciones se utilizan para esperar hasta que un proceso cambie de estado. Se emplean para esperar por un cambio de estado en un proceso hijo del proceso que realiza la llamada a estas funciones. Un cambio de estado para el proceso hijo puede ser uno de estos 3: el hijo ha muerto, el hijo ha sido detenido por una señal o el hijo continúa su ejecución tras recibir una señal apropiada.

Si el hijo ha cambiado de estado antes de la ejecución del *wait* entonces la llamada a esta función retornará inmediatamente. En caso contrario la llamada bloqueará el proceso hasta que se produzca este cambio de estado.

wait() espera hasta que uno cualquiera de los hijos del proceso que realiza la llamada a *wait* cambie de estado. *waitpid* en su lugar suspende la ejecución del proceso actual hasta que el hijo, especificado por el argumento *pid* cambie su estado. Por defecto, *waitpid* espera sólo por la muerte de los hijos, pero este comportamiento es modificable mediante el parámetro *options*.

El valor de *pid* puede tomar estos valores:

- Menor que -1 hace que se espere por cualquier hijo cuyo id de proceso de grupo sea igual al valor absoluto de este argumento.
- Igual a -1 esperará por cualquier hijo, el primero en cambiar de estado (igual comportamiento que *wait*).
- 0 indica que se esperará por cualquier proceso hijo que tenga el mismo id de grupo de procesos y
- mayor que cero hará que se espere por el hijo cuyo *pid* coincida con este valor.

El parámetro optiones es una máscara OR de diversas constantes. Existe además otra función llamada *waitid* que realiza una función análoga a las anteriores pero proporciona un mayor control sobre qué cambios de estado en el hijo deben ser esperados.

En cuanto a los los parámetros que retornan estas tres funciones, *wait()* en caso de éxito devuelve el pid del proceso hijo que ha cambiado de estado. En caso de error devuelve -1. *waitpid()* en caso de éxito devuelve lo mismo que *wait()* y en caso de error -1 pero si se ha especificado la constante *WHOHANG* la función no bloquea y devuelve cero si no hay ningún hijo que haya cambiado de estado. Por último *waitid()* devuelve cero en caso de éxito o si *WHOHANG* ha sido activado y no hay hijo que haya cambiado de estado y -1 en caso de error.

La llamada a *wait(int*)* es equivalente a *waitpid(-1, int*, 0)*. En la resolución del obligatorio se utilizó la función *wait()*, para forzar a un proceso padre para que espere a que su proceso hijo se detenga o termine. La llamada a esta función se realiza con *NULL* en el parámetro *status*. De otro modo, la función guardaría en la dirección a la que apunta *status* información del estado.

Una señal (signal) es una forma limitada de comunicación entre procesos empleada en Unix y otros sistemas operativos compatibles con POSIX. En esencia es una notificación asíncrona enviada a un proceso para informarle de un evento. Cuando se le manda una señal a un proceso, el sistema operativo modifica su ejecución normal. Si se había establecido anteriormente un procedimiento (handler) para tratar esa señal se ejecuta éste, si no se estableció nada previamente se ejecuta la acción por defecto para esa señal.

Al pulsar Ctrl-C en el shell donde se ejecuta un proceso el sistema le envía una señal *SIGINT*, que por defecto causa la terminación del proceso. Ctrl-Z hace que el sistema envíe una señal *SIGTSTP* que suspende la ejecución del proceso. Excepciones como la división por cero o la violación de segmento generan señales. Los procesos pueden enviar señales tanto a otros procesos como a sí mismos usando *kill* (por supuesto con los permisos necesarios), por ejemplo *kill(pid,SIGUSR1)* siendo pid el identificador del proceso al cual deseamos enviar la señal *SIGUSR1*. El núcleo puede generar una señal para informar de un evento a un proceso. Por ejemplo, *SIGPIPE* se genera cuando un proceso escribe en una tubería que había sido cerrada por el proceso que leía de ella.

Los manipuladores de señales se establecen mediante la llamada al sistema *signal()*. Si hay un manipulador de señal para una señal dada se invoca y, si no lo hay, se usa el manipulador por defecto. El proceso puede especificar también dos comportamientos por defecto sin necesidad de crear un manipulador: ignorar la señal *(SIG_IGN)* y usar el manipulador por defecto *(SIG_DFL)*. Hay dos señales que no pueden ser interceptadas ni manipuladas: *SIGKILL y SIGSTOP*.

La manipulación de señales es vulnerable a que se produzca una condición de carrera, pues las señales son asíncronas y puede ocurrir que llegue otra señal (incluso del mismo tipo) al proceso mientras transcurre la ejecución de la función que manipula la señal. Puede usarse la función *sigprocmask* para desbloquear la entrega de señales.

Veamos algunas señales tratadas en el obligatorio:

- *SIGCHLD* - Proceso hijo terminado, detenido (*o que continúa). Tratamiento por defecto: ignorar. Reprogramable.
- *SIGTERM* - Terminación. Tratamiento por defecto: exit. Reprogramable.

- *SIGINT* - Interrupción, se genera al pulsar Ctrl-C durante la ejecución. Tratamiento por defecto: exit. Reprogramable.

En sistemas Unix, al proceso que tiene hijos creados mediante *fork*, cuando el hijo termina, se le envía la señal *SIGCHLD* al padre. Por defecto, la señal se ignora y se crea un proceso zombie. El padre debe instalar un manipulador de señales para actuar sobre la señal. En algunas plataformas Unix, se pueden evitar los zombies explícitamente ignorando la señal *SIGCHLD*. Sin embargo, instalar un manipulador de señales para *SIGCHLD* y llamar a *wait* es la mejor manera de evitar zombies conservando la portabilidad.

De la siguiente forma definimos un handler para para la señal *SIGCHLD*. *signal()* acepta un número de señal y un puntero al handler de la señal y setea a ese handler para que acepte la señal correspondiente:

```
...

int catchChild(int sig_num)
{
        wait(NULL);
        cout<<"Hijo terminado."<<endl;
}

...

/* Se define el handler de la señal SIGCHLD */

signal(SIGCHLD, catchChild);

...
```

Como ya mencionamos, una forma de enviar señales es usando *kill*. Esta es la forma normal de enviar señales de un proceso a otro. *kill* es usada por el comando '*kill*' o el comando '*fg*'. Enviamos la señal *SIGTERM* a un proceso hijo:

```
kill(idHijo,SIGTERM);
```

3 IPC

3.1 Introducción

En la computación, IPC (*Inter-process communication*) es un conjunto de métodos para el intercambio de datos entre múltiples threads en uno o más procesos. Los procesos podrían estar corriendo en una o más computadoras conectadas a una red. Los métodos IPC están divididos en: pasaje de mensajes, sincronización, memoria compartida y llamadas remotas a procedimientos. El método IPC usado varía según la banda ancha y latencia de comunicación entre los threads, y el tipo de datos en transferencia.

Para comunicar procesos en Unix System V se dispone, entre otros, de tres mecanismos:

- los semáforos, que van a permitir sincronizar procesos;
- la memoria compartida, para que los procesos puedan compartir su espacio de direcciones virtuales;
- las colas de mensajes, que permiten el intercambio de datos con un formato determinado.

Estos mecanismos se han implementado de manera uniforme y tienen una serie de características comunes, entre las que están: Cada mecanismo tiene una tabla cuyas entradas contienen información acerca del uso del mismo. Cada entrada de la tabla tiene una llave numérica elegida por el usuario. Cada mecanismo cuenta con una llamada get() para crear una entrada nueva o recuperar una existente. La llamada contendrá, el menos, dos argumentos: una llave, y una máscara de indicadores.

3.2 Llaves

Una llave es una variable o constante del tipo *key_t* que se utiliza para identificar colas de mensajes, memoria compartida, y grupos de semáforos.

Un programa puede obtener una llave de tres formas distintas:

- usando la constante *IPC_PRIVATE*, con lo que hacemos que el sistema genere una llave;
- escogiendo al azar un número llave;
- utilizando *ftok()* como veremos a continuación.

Para crear una llave System V se apoya en el concepto de proyecto, con la idea de que todos los mecanismos de comunicación entre procesos que

pertenezcan a un mismo proyecto compartan el mismo identificador. Esto impide que procesos que no estén relacionados, por no formar parte de un mismo proyecto, puedan interferirse involuntariamente debido a que usan la misma llave.

La librería estándar de C de System V proporciona la función *ftok()* para convertir una ruta y un identificador de proyecto a una llave de IPC. Esta función tiene la siguiente declaración:

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

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

En el fichero de cabecera <*sys/types.h*> se define el tipo *key_t,* que suele ser un entero de 32 bits. La función *ftok* devuelve una llave basada en *pathname* e *id*. *pathname* es un nombre de archivo que debe existir en el sistema, y al cual debe poder acceder el proceso que llama a *ftok*. *id* es un char que identifica al proyecto. *ftok* devolverá la misma llave para rutas enlazadas a un mismo fichero, siempre que se utilice el mismo valor para id. Para el mismo fichero, *ftok* devolverá diferentes llaves para distintos valores de id.

Si el fichero no es accesible para el proceso, bien porque no existe, bien porque sus permisos lo impiden, *ftok* devolverá el valor -1, que indica que se ha producido un error en la creación de la llave.

```
key_t llave;
...
llave = ftok("prueba", 'a');
if (llave == (key_t) -1)
{
        /* Error al crear la llave. Tratamiento del error */
}
```

3.3 Facilidades de IPC desde la línea de comandos

Los programas estándar *ipcs* e *ipcrm* nos 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. *ipcrm* se utiliza para liberar un mecanismo asignado. A continuación se explican las opciones más comunes de estos comandos:

ipcs: si no se muestra ninguna opción, *ipcs* muestra un resumen de la

información de control que se almacena para los semáforos, memoria compartida y mensajes que hay asignados. Las principales opciones son:

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

ipcrm: Algunas de las opciones más comunes son:

- *-q msqid* borra la cola de mensajes cuyo identicador coincide con *msqid*.
- *-m shmid* borra la zona de memoria compartida cuyo identicador coincide con *shmid*.
- *-s semid* borra el semáforo cuyo identicador coincide con *semid*.

3.4 Semáforos en System V

El mecanismo de semáforos implementado en UNIX System V es una generalización del concepto de semáforo visto en teoría, ya que permite manejar un conjunto o grupo de semáforos con un identicador asociado. Cuando realicemos una operación P o V, éstas actuarán de forma atómica sobre los semáforos del grupo.

Con la llamada *semget* podemos crear o acceder a un grupo de semáforos que tienen un identicador común. Además inicializa todos los semáforos del grupo a 0:

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

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

donde:

- *key* es la llave que indica a qué grupo de semáforos queremos acceder. Se podrá obtener de una de las tres formas vistas en la introducción.
- *nsems* es el número total de semáforos que forman el grupo devuelto por *semget*. Cada uno de los elementos dentro del grupo de semáforos puede ser referenciado por los números enteros desde 0 hasta *nsems*-1.

- *semflg* Es una máscara de bits que indica en qué modo se crean los semáforos, teniendo:
    - *IPC_CREAT,* si este flag está activo, se creará el conjunto de semáforos, en caso de que no hayan sido ya creados.
    - *IPC_EXCL,* se utiliza en conjunción con *IPC_CREAT*, para lograr que *semget* de un error en el caso de que se intente crear un grupo de semáforos que ya exista. En este caso, semget devolverá -1 e inicializará la variable *errno* con un valor *EEXIST*.
    - Permisos del semáforo. Los 9 bits menos signicativos de *semflg* indican los permisos del semáforo.

Si la llamada a *semget* funciona correctamente, devolverá un identicador del grupo de semáforos con el que podremos acceder a los semáforos en sucesivas llamadas. Si hay algún tipo de error, *semget* devuelve el valor -1 e introducirá en la variable global *errno* el código del error que se ha producido.

```
int sem;
key_t llave;

llave = ftok (".", 'a');

if ( llave != (key_t) -1)
{
        sem = semget (llave, NUM_SEM, PERMISOS | IPC_CREAT);
        if (sem == -1)
                /* Error al crear un nuevo grupo de semáforos con llave */
}
```

En este ejemplo, para asegurarnos de que creamos un nuevo grupo de semáforos, levantamos la bandera *IPC_CREAT*. Lo mismo ocurrirá en el caso de utilizar una llave ya hecha.

La llamada *semctl* sirve para controlar un grupo de semáforos. Su declaración es la siguiente:

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

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

```
};

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

Donde:

- *semid* es el identicador del grupo de semáforos sobre el que actuará *semctl*. Previamente se ha tenido que crear con *semget*.
- *semnum* indica cuál es el semáforo del grupo de semáforos al que se quiere acceder. Su valor estará en el rango 0..N-1, siendo N el número de semáforos que hay en el grupo.
- *cmd* es la operación de control a realizar sobre el semáforo. Veamos algunos valores definidos por las siguientes constantes:
    - *SETVALL* inicializa un semáforo a un valor determinado. Este valor se indica en el argumento *arg*.
    - *IPC_RMID* le indica al kernel que tiene que borrar el conjunto de semáforos asociados al identificador *semid.* Esta operación no tendrá efecto mientras haya un proceso que esté usando los semáforos.

semctl devuelve un número cuyo signicado dependerá del valor de *cmd*.

Para la resolución del obligatorio se desarrollaron varias funciones para actuar sobre los semáforos ubicados en un array de semáforos. Veamos la función de inicialización:

```
#define PERMISOS S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH

int inicializarSemaforos(char clave, int cantSemaforos, int valor)
{
        key_t llave;
        llave = ftok(".",clave);
        if(llave == (key_t)-1)
                return -1;
        int sid = semget(llave, cantSemaforos, PERMISOS | IPC_EXCL | IPC_CREAT);
        if(sid == -1)
                return -1; // si ya fueron o fue inicializado se retorna error
        for(int i = 0; i < cantSemaforos; i++)
                if(semctl(sid, i, SETVAL, valor) == -1)
                        return -1;
        return sid;
}
```

Como se puede observar, la inicialización retornará -1 si los semaforos ya estan inicializados o si ocurre algun error al crearlos y modificarlos. Los permisos utilizados siempre seran *0666*, que implica permiso de lectura y escritura a cualquier usuario (owner, group, other).

Para realizar las operaciones P y V tenemos que utilizar la llamada *semop*. Su declaración es la siguiente:

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

Donde:

- *semid* es el identicador del grupo de semáforos sobre el que se van a realizar las operaciones atómicas.
- *sops* es un puntero a un array de estructuras que indican las operaciones que se van a realizar sobre los semáforos.
- *nsops* es el número total de elementos que tiene el array de operaciones.

Cada elemento del array *sops* es una estructura de tipo *sembuf* que se define de la siguiente manera:

```
struct sembuf
{
        ushort_t sem_num;   /* semaphore index in array */
        short sem_op;       /* semaphore operation */
        short sem_flg;      /* operation flags */
};
```

Donde:

- *sem_num* es el número del semáforo. Su valor está en el rango 0.. N-1, siendo N el número total de semáforos que hay agrupados en el identicador. Este campo se utiliza como índice para acceder a un semáforo concreto del grupo.
- *sem_op* es la operación a realizar sobre el semáforo *sem_num*. El resultado es sumar el valor de *sem_op* al contador del semáforo, de forma que: si *sem_op* es positivo equivale a una operación V. Si *sem_op* es negativo equivale a una operación P.
- *sem_flg* son una serie de banderas para indicar cómo responderá la llamada *sem_op* en el caso de encontrarse con alguna dificultad.

semop devuelve -1 en el caso de que la llamada no se haya realizado con éxito, conteniendo la variable *errno* el código correspondiente. Esto puede ocurrir, con una operación P. Por ejemplo, si el semáforo tiene valor 2, no se le puede restar

un número mayor que 2, porque el semáforo entonces pasará a tener un valor negativo. Cuando no se puede ejecutar una operación, la respuesta de *semop* dependerá del valor del campo *sem_flg*. Estos pueden ser: *IPC_NOWAIT*, en este caso, *semop* devuelve el control si no se puede satisfacer la operación que se intenta hacer. Por defecto se trabaja con *IPC_WAIT*. Con *SEM_UNDO*, la operación se deshace cuando el proceso termina.

Si semop es interrumpido por una señal, entonces devuelve -1 y le da a la variable *errno* el valor *EINTR*. Una buena implementación de los semáforos deberá reiniciar la operación solicitada en el caso de que fuera interrumpida por una señal. Este aspecto se tiene muy en cuenta en el obligatorio, y se aplica en casos puntuales sin generalizar, teniendo en cuenta que una señal capturada implica en la mayoría de los casos terminar la ejecución del proceso.

Veamos las funciones desarrolladas para realizar P y V:

```
int obtenerIdArraySemaforos(char clave,int cantSemaforos)
{
        if(!cantSemaforos)
                return -1;
        key_t llave;
        llave = ftok(".",clave);
        if(llave == (key_t)-1)
                return -1;
        return semget(llave, cantSemaforos, PERMISOS);
}

int P(char clave, int cantSemaforos, int sem_num)
{
        int id = obtenerIdArraySemaforos(clave,cantSemaforos);
        if(id == -1)
                return -1;
        struct sembuf sem_lock = {sem_num, -1, 0};
        return  semop(id, &sem_lock, 1);
}

int V(char clave, int cantSemaforos, int sem_num)
{
        int id = obtenerIdArraySemaforos(clave,cantSemaforos);
        if(id == -1)
                return -1;
        struct sembuf sem_unlock = {sem_num, 1, 0};
        return semop(id, &sem_unlock, 1);
}
```

SISTEMAS OPERATIVOS – LABORATORIO 3 **2011**

el sistema (entero).
2. Localizar dicha zona a través de un puntero a un determinado tipo de datos.
3. Liberar el puntero obtenido.
4. Eliminación de la memoria compartida.

Los pasos 2 y 3 se repetirán todas las veces que sea necesario utilizar la memoria compartida durante la vida del proceso. Los pasos 1 y 4 serán realizados en este caso al inicializar el servicio y al finalizarlo.

Para crear zonas de memoria compartida utilizaremos la función shmget de una manera similar a los semáforos.

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

int shmget (key_t clave, int tam, int flag);
```

Se devuelve -1 en caso de error donde *errno* indica el tipo de error, o el identificador de la zona de memoria compartida en caso de éxito. Ese identificador lo utilizaremos para trabajar con la zona de memoria compartida.

- *key_t clave* – clave de la zona de memoria compartida.
- *int tam* – tamaño en bytes de la zona de memoria compartida.
- *int flag* – permisos. Se indican de la misma manera que con los semáforos.

Al igual que con semáforos, desarrollamos funciones para trabajar con memoria compartida. La siguiente función devuelve el identificador de la zona de memoria compartida para la clave *clave*. Si el segmento no existe lo crea.

```
int abrir_segmento(char clave, int segsize)
{
        key_t llave;
        llave = ftok(".",clave);
        if(llave == (key_t)-1)
                return -1;
        return shmget(llave, segsize, PERMISOS | IPC_CREAT);
}
```

Notar que no se devuelve error si el segmento ya fue creado. Otra función que será de utilidad es:

```
bool existe_segmento(char clave, int segsize)
{
        int shmid = shmget(ftok(".",clave), segsize, PERMISOS);
```

```
    return !(shmid == -1 && errno == ENOENT);
}
```

Una vez que tenemos creada la memoria compartida, necesitamos saber su dirección (observe que *shmget* no nos la indica). Para utilizar la memoria compartida debemos antes vincularla con alguna variable de nuestro código. De esta manera, siempre que usemos la variable vinculada estaremos utilizando la variable compartida.

Para establecer un vínculo utilizamos la función *shmat*.

*void *shmat(int shmid, const void *shmaddr, int shmflag)*

Se devuelve la dirección de memoria de comienzo de la memoria compartida o *NULL* si hubo algún error.

- *int shmid* – identificador de la memoria compartida obtenido con *shmget.*
- *const void *shmaddr* – Dirección concreta donde reside la memoria compartida. No lo vamos a utilizar, por lo que su valor siempre será *NULL*.
- *int shmflag* – permisos. Por ejemplo. Aunque hayamos obtenido una zona de memoria con permiso para escritura o lectura, podemos vincularla a una variable para solo lectura. Si no queremos cambiar los permisos usamos 0.

    - *SHM_RND*: Indica que la dirección de la memoria debe redondearse a la dirección de la página de memoria más cercana. No la utilizaremos.
    - *SHM_RDONLY*: Indica que la memoria compartida será de solo lectura.

Cuando ya no vamos a utilizar más la variable vinculada, hemos de desvincularla antes de poder eliminar la memoria compartida. Para ello utilizamos la función *shdmt*.

*int shmdt(const void *shmaddr)*

Devuelve -1 si hubo algún error, otro valor en caso contrario.

- *const void *shmaddr* – Variable vinculada a la memoria compartida.

Veamos las función implementadas:

```
void * mapear_segmento(char clave, int segsize)
{
```

```
        key_t llave;
        llave = ftok(".",clave);
        if(llave == (key_t)-1)
                return NULL;
         int shmid = shmget(llave, segsize,PERMISOS);
        if(shmid != -1)
                return shmat(shmid, 0, 0);
         else
                 return NULL;
}

int liberar_segmento(void* addr)
{
        return shmdt(addr);
}
```

La función liberar_segmento sólo libera el vínculo, pero no elimina la zona de memoria compartida. Recién destruimos la memoria compartida en el paso número 4.

Como los demás recursos IPC, la memoria compartida reservada seguirá estando presente en el sistema hasta que no la borremos explícitamente. Para ello utilizamos la función *shmctl* de una forma muy similar a como borramos grupos de semáforos o colas de procesos.

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

Devuelve -1 si error, u otro valor en caso contrario.

- *int shmid* – identificador de la memoria compartida obtenido con *shmget*.
- *int cmd* - alguna de las siguientes constantes:
  - *IPC_STAT*: Nos rellena la estructura shmid_ds() con los datos que maneja el núcleo en lo referente a la zona de memoria compartida.
  - *IPC_SET*: Lo contrario de lo anterior establece la estructura que le pasamos como parámetro en el kernel.
  - *IPC_RMID*: Borra el recurso.
- *struct sumid_ds *buf* - para el comando de borrado no es necesario el tercer argumento, por lo que utilizaremos *NULL*.

La siguiente función borra la zona de memoria compartida identificada por *clave*.

int destruir_segmento(char clave, int segsize)

```
{
        key_t llave;
        llave = ftok(".",clave);
        if(llave == (key_t)-1)
                return -1;
        int shmid = shmget(llave, segsize, PERMISOS);
        if(shmid != -1)
                return(shmctl(shmid, IPC_RMID, NULL));
        else
                return -1;
}
```

4 SoAgenda

4.1 Introducción

Una vez introducidas las herramientas utilizadas para la resolución del obligatorio, podemos comenzar a describir la implementación realizada.

La agenda induce a trabajar con procesos concurrentes que se comunican y sincronizan entre sí mediante herramientas del sistema UNIX. Se trata de construir un intérprete de órdenes al estilo de los shells de UNIX, con un proceso central (un servidor) que ejecutará órdenes solicitadas por otros procesos, que actuarán como clientes suyos.

Un proceso puede proporcionar unos servicios a los restantes procesos del sistema. Estos servicios serán operaciones de diverso tipo, por ejmplo imprimir un documento, leer o escribir una información, etc. En el modelo cliente/servidor, cuando un proceso desea un servicio que proporciona cierto proceso, le envía un mensaje solicitando ese servicio: una **petición**. El proceso que cumple el servicio se llama **servidor** y el solicitante se llama **cliente**.

Los procesos clientes y servidores han de seguir un **protocolo** de comunicaciones que defina:

- cómo se codifican las peticiones; y
- cómo se sincronizan entre sí los procesos.

Los clientes y servidores han de estar de acuerdo en cómo se escriben los mensajes: en qué orden van los posibles parámetros de la petición, cuántos *bytes* ocupan, etc.

La forma de sincronización nos dice si el cliente puede seguir adelante justo después de enviar la petición (no bloqueante), o por el contrario tiene que esperar a

que el servidor le envíe una respuesta (bloqueante). Si la comunicación es no bloqueante, habrá que definir un mecanismo para que el cliente pueda saber si la respuesta del cliente está disponible. En esta práctica se adoptará una comunicación bloqueante: el cliente siempre esperará hasta recibir una respuesta del servidor.

El diálogo cliente/servidor es casi siempre bidireccional. Por un lado, el cliente envía información al servidor (el tipo de servicio solicitado más los parámetros); por otro, el servidor devuelve información al cliente (los resultados del servicio, códigos de error en caso de producirse, etc.)

El programa servidor se ejecutará del siguiente modo:

$ ./serviagenda

Al invocar al servidor, 'una parte de el' quedará automáticamente en segundo plano (background) en espera de atender las futuras peticiones de los procesos clientes hasta que reciba una orden especial de finalización.

El servidor ejecutará las órdenes secuencialmente, una detrás de otra. Esto es, el servidor no puede lanzar varias órdenes concurrentes, y siempre tendrá que esperar a que una orden finalice para tramitar la siguiente.

Los clientes que envíen peticiones mientras el servidor está ejecutando una orden tendrán que esperar. Cuando una orden termine de ejecutarse, el servidor deberá notificar el resultado de la ejecución al proceso cliente que la demandó (el cual estará esperando por la notificación).

Mientras no haya órdenes que atender, el servidor permanecerá bloqueado. Sin embargo, en el servidor podrán ejecutarse comandos. Estos son los requeridos por la letra: *listar* y *salir.*

El programa cliente se ejecutará del siguiente modo:

$ ./cliagenda <*nombre*>

El cliente solo admitirá un argumento que constituye el nombre del cliente en el sistema. Este nombre es único y por tanto no se aceptará el ingreso de un cliente cuyo nombre ya pertenece a algún cliente que se encuentra en ese momento en el sistema.

Las órdenes recibidas en el cliente, se convertirán en peticiones al servidor, cuyo trabajo será responder de modo tal que el cliente pueda desplegar la respuesta en su pantalla.

Los clientes y el servidor han de transferirse información (las peticiones y

las respuestas). Además, los procesos del sistema tendrán que sincronizarse en ciertos momentos (ej. cuando un cliente espera a que se complete una petición).

4.2 Comandos

Repasemos los comandos que será posible ejecutar en el servidor:

- *listar:* este comando listara todas entradas de la agenda indicando para cada entrada su nombre, tipo, el nombre de la categoría en la que se encuentra y el nombre del usuario que creo dicha entrada.
- *salir:* si se ejecuta este comando cuando no hay usuarios en el sistema, el servidor finalizará su ejecución.

Del lado del cliente será posible ejecutar: *dir*, *ver*, *creacat*, *creacon*, *cd*, *cd ..*, *borrar* y *salir*. Para cada uno de estos comandos el cliente generará una petición al servidor. Este se encargará de procesar la petición y dar una respuesta. El servidor imprimirá en pantalla el comando recibido por el cliente y además será el encargado de enviar a todos los clientes que se encuentren en la misma categoría del cliente que realizó la petición, un *echo.* El *echo* es un mensaje que indica el comando recibido por un cliente, siempre y cuando sea un comando válido, el nombre del cliente que ejecutó el comando, y los datos correspondientes al comando. Si se trata de un *creacat* o un *creacon* el *echo* poseerá el nombre de la *categoría* o *contacto* y en el caso del *creacon* además poseerá el teléfono ingresado. El *echo* será enviado por el servidor aunque la ejecución no tenga éxito.

Por su lado, el cliente poseerá un administrador de *echos*, que se encargará de recibir los *echos* por parte del servidor e imprimir en pantalla los datos correspondientes.

El hecho de que el cliente deba procesar una petición y leer la respuesta recae sobre la restricción de que las estructuras de datos deben mantenerse únicamente en serviagenda. O mejor dicho, serviagenda será el único que acceda a las estructuras que albergan los datos de la agenda y que podrá modificar dichas estructuras.

4.3 Interfaz de comunicaciones

La comunicación entre cliente/servidor se establecerá mediante semáforos y memoria compartida. Básicamente, se tienen sendas estructuras para que el cliente o el servidor envíen mensajes.

Las estructuras de datos empleadas serán las siguientes:

```
enum tipoEntrada {categoria, contacto};

enum tipoPeticion {ingreso, dir, ver, creacat, creacon, cd, cdd, borrar, salir};

struct echo
{
        char nomUsuario[MAX_NOMBRE];
        char comando[8];
        char nombre[MAX_NOMBRE];
        char telefono[MAX_NOMBRE];
};


struct entrada
{
        int id; // -1 si la entrada no ha sido creada
        tipoEntrada tipo;
        int subcategoria;
        char nombre[MAX_NOMBRE];
        char telefono[MAX_NOMBRE];
        char creador[MAX_NOMBRE];
} ;

struct peticionEntrada
{
        tipoPeticion tipo;
        int idUsuario;
        char nombre[MAX_NOMBRE];
        char telefono[MAX_NOMBRE];
} ;

struct peticionSalida
{
        bool exito;
        int error;
        int cantEntradas;
        entrada entradasCC[MAX_DIR_ENTRADAS];
} ;
```

Estas estructuras se encuentran declaradas en el módulo *constcom.h,* módulo en el cuál se definen las constantes utilizadas y los elementos de comunicación entre el servidor y los clientes así como funciones auxiliares que actúan sobre los elementos.

4.4 Comunicación con memoria compartida y semáforos

4.4.1 Problemas de sincronización

La primera cuestión es cómo indica el cliente al servidor que hay una orden pendiente. Se dispondrá de un semáforo, cuya clave conocen tanto los clientes como el servidor. El semáforo estará inicializado a cero. Cada vez que un cliente solicita un servicio, realiza una operación V sobre el semáforo. Por su parte, en cada iteración el servidor haría una operación P sobre el mismo semáforo.

Un proceso cliente ha de bloquearse hasta que el servidor le envíe una respuesta. Esto se puede conseguir con semáforos, igual que en el caso anterior, pero en sentido inverso: el cliente, tras haber enviado su mensaje, se bloqueará con una operación P sobre un semáforo de *petición servida* correspondiente solo a él. Cuando el servidor complete su petición, ejecutará una V sobre ese mismo semáforo. Lo anterior implica que el cliente se identifique en el sistema y que el servidor conozca que cliente está solicitando la petición.

Como veremos más adelante, si a un cliente se le permite ingresar al sistema, este recibirá un número que lo identifica en el sistema. Este número será el id del cliente y le permitirá recibir la respuesta a sus peticiones así como el denominado *echo.* El servidor por su parte, almacenará el nombre del cliente y la categoría en la que se encuentra en un lugar identificado por el id del cliente. En cada petición, como se puede observar en la estructura *peticionEntrada* el cliente le indica al servidor quien es. Si por alguna razón esto no se cumple, podrían generarse serios conflictos. Es por ello que se debe seguir el protocolo de comunicación establecido.

4.4.2 Transferencia

Para realizar las transferencias de información, se utilizan varias zonas de memoria compartida. Si varios clientes, desean enviar una petición, estos deberán acceder a enviar la petición luego de haber obtenido la zona multiexcluida por un semáforo.

4.4.3 Algoritmo de comunicación

Estos son fragmentos algorítmicos que desarrollan las soluciones que acabamos de plantear, y que estarán en el interior de los módulos serviagenda y cliagenda. Se tendrá una zona de memoria compartida con una estructura de

peticionEntrada y otra zona para las respuestas. Para estas se dispondrá de un array de estructuras *peticionSalida*. El largo del array está determinado por la constante *MAX_USUARIOS*.

El semáforo *SEM_CERROJO_MEMORIA* inicialmente vale uno, mientras que el de *SEM_PETICION_PENDENTE* y los de *SEMS_PETICION_SERVIDA* están inicializados a cero.

Proceso cliente (al realizar una petición):

/ Bloqueo de la memoria compartida */*

P (CLAVE_SEM_CERROJO_MEMORIA,1,0)

... escribe petición en la memoria ...

/ Aviso al servidor */*

V (CLAVE_SEM_PETICION_PENDENTE,1,0);

/ Espera por que se complete */*

P (CLAVE_SEMS_PETICION_SERVIDA,MAX_USUARIOS,idCliente);

... recoge la respuesta de la memoria ...

El cliente no será el encargado de liberar el semáforo *SEM_CERROJO_MEMORIA* dado que desconoce el momento en el que el servidor lee la petición. Una vez leída la petición, el servidor podrá liberar el semáforo para que otro cliente pueda escribir su petición mientras el procesa la petición recien leída. Veamos,

Proceso servidor (al esperar por una petición):

/ Espera por una petición */*

P(CLAVE_SEM_PETICION_PENDENTE,1,0);

... recoge la petición de la memoria ...

V(CLAVE_SEM_CERROJO_MEMORIA,1,0);

... atiende la petición ...

... escribe la respuesta en la memoria ...

/ Para dar por servida una petición */*

V(CLAVE_SEMS_PETICION_SERVIDA,MAX_USUARIOS,idCliente);

4.5 idCliente

Cuando se ejecuta el serviagenda, este inicializa, entre otras cosas, todos los semáforos y la memoria compartida que será utilizada para la comunicación. Entre estos elementos, se encuentra algo fundamental para el posible ingreso de los clientes, y para que estos puedan detectar la presencia del servidor. Estamos hablando del semáforo *SEM_ID_CLIENTE* y del entero en memoria compartida *ID_CLIENTE.*

El entero *ID_CLIENTE* se inicializa en 0. Este será el id del primer cliente que ingrese al sistema. Cuando un cliente ingresa al sistema, el servidor modifica *ID_CLIENTE* y lo setea, si es posible, en un id libre. Sin embargo, si el sistema ha alcanzado la cantidad máxima de clientes, el servidor seteara *ID_CLIENTE* con el valor *MAX_USUARIOS*. Cuando un cliente se retira del sistema, el servidor simplemente setea el entero en memoria con el id del cliente que se retira.

Cuando se ejecuta cliagenda, su primera acción será verificar la existencia del semáforo *SEM_ID_CLIENTE*, dado que, para conocer el valor de *ID_CLIENTE*, deberá hacer P del semáforo. Si el semáforo no existe es porque el servidor aun no está corriendo y por lo tanto se termina la ejecución.

4.6 cliAgenda

Una vez chequeada la existencia del semáforo *SEM_ID_CLIENTE*, se realiza P sobre el semáforo y se lee *ID_CLIENTE*. Si el valor es *MAX_USUARIOS* entonces se ha alcanzado la máxima cantidad de clientes y por lo tanto liberamos el semáforo y terminamos la ejecución. De lo contrario, tenemos un id disponible. Sin liberar el semáforo aun, realizamos una petición de ingreso al servidor quien se encargará de chequear la unicidad de nuestro alias y demás. Una vez obtenida la respuesta, liberamos el semáforo.

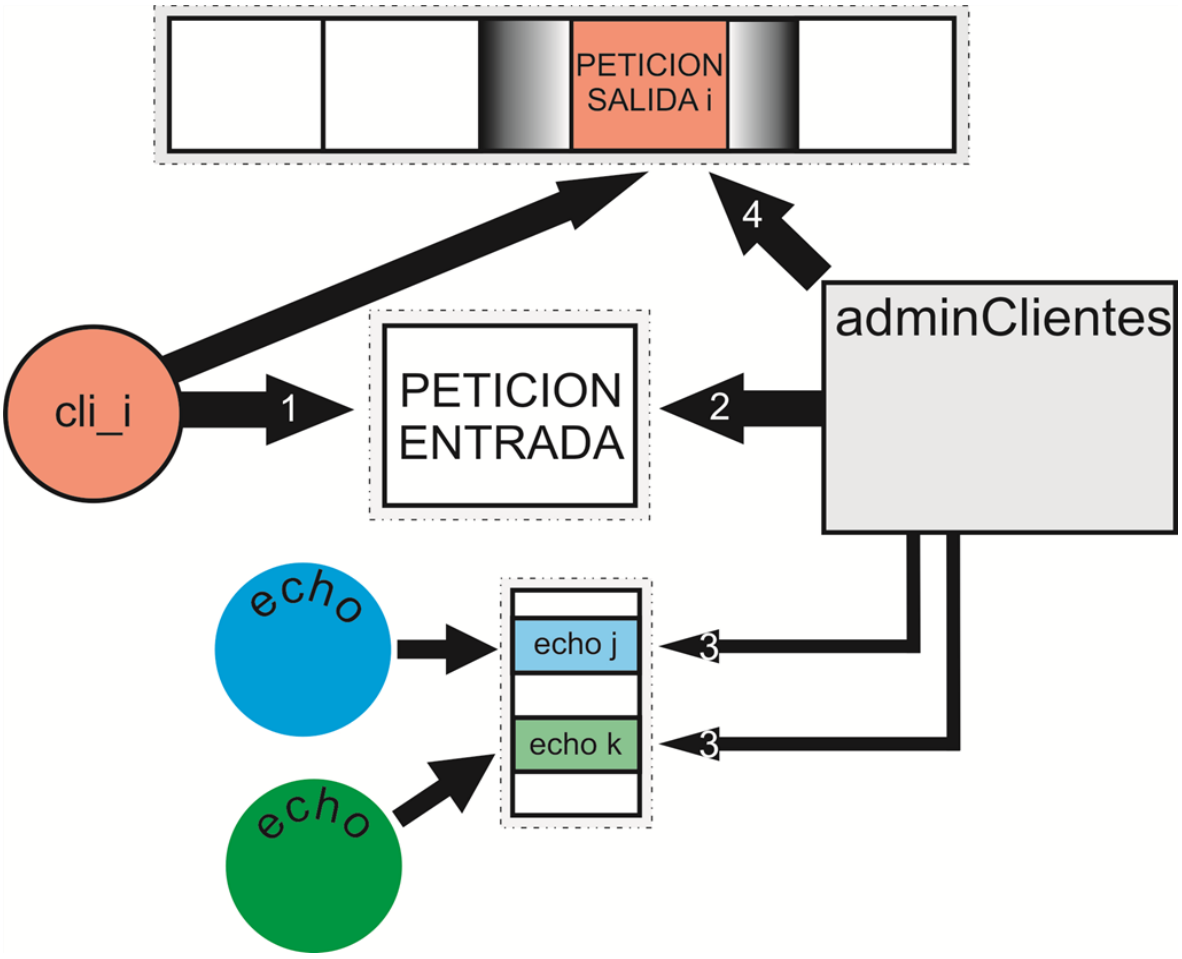

Ilustración 4.6.1

Como hemos mencionado, el módulo cliagenda dispone de dos funcionalidades bastante distintas. Por un lado, el cliente puede ejecutar comandos que son transformados en peticiones al servidor, pero además, el cliente recibe el *echo* de los comandos ejecutados por los clientes que se encuentran en la misma categoría que él. Evidentemente, estamos hablando de un proceso 'demonio' que se ejecuta en segundo plano y que no es controlado por el usuario. La única función de este proceso es imprimir en pantalla los *echos* enviados por el servidor. La comunicación entre el servidor y los procesos demonios es bastante similar a lo que sucede con las peticiones. Tendremos en memoria compartida un array de estructuras *echo* de largo *MAX_USUARIOS*. Según el id del cliente, es donde éste leerá su *echo* cuando el proceso demonio sea despertado por el servidor y es según el id de los clientes en la misma categoría que el servidor copiara el *echo* del cliente que ejecuta el comando.

En la ilustración 4.6.1 podemos observar como el administrador de clientes

se encarga de divulgar el comando recibido a los clientes j y k que se encuentran en la misma categoría del cliente i.

El proceso cliagenda creará a un nuevo proceso hijo mediante la función *fork*. Veamos:

```
int main (int argc, char * argv[])
{
	if(!existeArraySemaforos(CLAVE_SEM_ID_USUARIO ,1))
	{
		cout<<"<El servidor no esta corriendo>"<<endl;
		exit();
	}
	else
	{
		/* esto es para que ningun Usuario que ingrese tome el mismo valor como
		un posible id */
		P(CLAVE_SEM_ID_USUARIO ,1,0);
		int* idc =  (int*)mapear_segmento(	CLAVE_MEM_ID_USUARIO
							,SIZE_MEM_ID_USUARIO   );
		idCliente = *idc;
		liberar_segmento(idc);

		if(idCliente == MAX_USUARIOS)
		{
			V(CLAVE_SEM_ID_USUARIO ,1,0);
			cout <<"<Cantidad maxima de usuarios alcanzada>"<<endl;
			exit();
		}
		else
			categoriaActual = inicializarCliente(argv[1],idCliente);

		V(CLAVE_SEM_ID_USUARIO,1,0); /* luego de  haber ejecutado la
							inicialización */

		if (idCliente == ERROR_ALIAS_EXISTS)
		{
			cout <<"<El nombre utilizado pertenece a otro usuario>"<< endl;
			exit();
		}
		else
		{
			cout<<"<Bienvenido al sistema>"<<endl;

			/* Define el handler para la señal SIGCHLD */
			signal(SIGCHLD, catchChild);

			int childPid = fork();
```

```
                        if (childPid == 0)
                        {
                                /* CODIGO PROCESO HIJO */
                                adminEcho();
                        }
                        else
                        {
                                /* CODIGO PROCESO PADRE */
                                adminEchoId = childPid;
                                adminConsolaCliente();
                        }
                }
        }
}
```

Más adelante veremos porque es que el cliente necesita saber el id de la categoría en la que se encuentra. Ahora solo adelantaremos que en el sistema una categoría (y un contacto) se identifica por su id y el id de la subcategoria en la que se encuentra. Esto sucede dado que pueden existir categorías y contactos del mismo nombre pero ubicadas en categorías distintas. Cuando un cliente ingresa al sistema, este comienza en la categoría 'Principal' cuyo id es *MAX_ENTRADAS* y cuya subcategoria es 'Principal'.

```
void adminEcho()
{
        signal(SIGTERM, killHandler); /* se establece el handler */
        while(true)
        {
                /* se bloquea en el semaforo correspondiente a su id */
                P(CLAVE_SEMS_ECHO_PENDIENTE,MAX_USUARIOS,idCliente);

                ... se copia el echo ...

                V(CLAVE_SEMS_ECHO,MAX_USUARIOS,idCliente); /* permite que
                se vuleva a usar el echo mientras se imprime */

                ... impresion del echo ...
        }
}
```

En *adminEcho* se realiza un V del semáforo *(SEMS_ECHO,idCliente)*. Cuando veamos la implementación del lado del servidor podremos observar una multiexclusión cruzada. Es el servidor el que hace P del mismo semáforo para no sobrescribir un *echo* mientras el cliente aun lo está copiando. Y es el servidor el que hace V del semáforo *(CLAVE_SEMS_ECHO_PENDIENTE,idCliente)* para despertar al administrador de *echos*. Veamos la inicialización del cliente:

```
int inicializarCliente(char* alias,int &idCliente):
{
        peticionEntrada petEntrada;
        peticionSalida petSalida;
        petEntrada.tipo = ingreso;
        petEntrada.idUsuario = idCliente;
        petEntrada.nombre = alias;

        procesarPeticion(petEntrada,petSalida);

        if(petSalida.exito)
                return petSalida.entradasCC[0].id;
        else
        {
                idCliente = petSalida.error;
                return -1;
        }

}
```

El procesamiento de una petición *(procesarPeticion(...))* corresponde a realizar lo indicado en el capítulo 4.4.3 para el proceso cliente.

El proceso padre será el encargado de la administración de la consola. La función *adminConsolaCliente()* básicamente repite los siguientes pasos:

- Leer el comando y verificar que es válido. Si el comando es inválido se muestra el mensaje correspondiente y se continúa con la lectura.
- Procesar el comando del mismo modo que con el ingreso del cliente.
- Dormir durante un tiempo aleatorio entre 1 y 5 segundos.
- Imprimir el resultado del comando.

Veamos:

```
void adminConsolaCliente()
{
        peticionSalida petSalida;
        peticionEntrada petEntrada;
        petEntrada.idUsuario = idCliente;
        while(true)
        {
                comando = leerComando();

                 if(comando == dir)
                 {
                        petEntrada.tipo = dir;
                        procesarPeticion(petEntrada,petSalida);
```

```
                    sleep(rand()%(5) + 1); /* se duerme entre 1 y 5 segundos para dar
                                              por terminado el pedido */

                    if(petSalida.exito)
                    {
                        ... impresión del comando ...
                    }
                    else
                    {
                        ... impresión del error ...
                    }
                    else if(comando == ver) ...
                    else if( comando == creacat) ...
                    else if( comando == creacon) ...
                    else if(comando == cd) ...          ... categoriaActual = petSalida...
                    else if(comando == cd ..) ...       ... categoriaActual = petSalida...
                    else if(comando == borrar) ...
                    else if(comando == salir)
                    {
                        P(CLAVE_SEM_ID_USUARIO ,1,0);
                        petEntrada.tipo = salir;
                        procesarPeticion(petEntrada,petSalida);
                        sleep(rand()%(5) + 1);
                        if(petSalida.exito)
                        {
                            kill(adminEchoId,SIGTERM); /* se mata al
                                                          proceso hijo */
                            cout<<"<Ha finalizado la ejecucion de
                                cliagenda>"<<endl;
                            V(CLAVE_SEM_ID_USUARIO ,1,0);

                            exit(); // termina la ejecucion del cliagenda
                        }
                        else
                        {
                            cout<<"<ERROR ...>"<<endl;
                            V(CLAVE_SEM_ID_USUARIO ,1,0);
                            exit();
                        }
                    }
                    else
                        cout<<"<Comando no reconocido>"<<endl;
                }
            }

        }
```

Varios puntos a tener en cuenta. Al comenzar la función, asignamos a la *peticionEntrada* el id del cliente, ya que en cada petición el cliente debe indicarle al servidor quién es, como se estableció en el protocolo de comunicación. Se ejecuta un ciclo infinito procesando los comandos recibidos hasta recibir el comando *salir*. El comando *salir* implica comunicarle al servidor que un id quedará libre, y por lo tanto, si se había alcanzado el máximo números de usuarios, ahora si se podrá ingresar al sistema. El servidor, como mencionamos anteriormente, modificará el entero *ID_CLIENTE* y seteará su valor en el valor del id del cliente que se retira. Por lo tanto, el cliente, de igual modo que lo hace en la inicialización, realiza P sobre el semáforo que multiexcluye ese lugar de memoria y al finalizar la petición, libera el semáforo. Notar que antes de terminar la ejecución, se mata al proceso hijo enviándole la señal *SIGTERM* a la cuál se le establece un handler en el proceos hijo mediante la sentencia *signal(SIGTERM, killHandler)*. Es importante remarcar que, en el momento de ser enviada la señal al proceso hijo para terminar su ejecución, este se encuentra bloqueado en el semáforo de espera del *echo* ya que luego de procesada la petición, el cliente ha sido dado de baja en el servidor y no será notificado de los comandos procesados por otros clientes. El cliente se libera del semáforo y la función *semop* devuelve -1 indicando en *errno* el valor *EINTR*. Como el handler de la señal realiza un *exit()* y no cabe la posibilidad de que se acceda a la zona de memoria donde se escribe el *echo*.

4.7 ServiAgenda

Una vez ejecutado el serivagenda, ningún otro servidor podrá estar corriendo. Por lo tanto, al ejecutarse, se chequea la existencia del semáforo *SEM_ID_CLIENTE*. Si este existe, entonces un servidor ya está corriendo y finaliza. De lo contrario, el semáforo se crea, se inicializa en 1 y se realiza P sobre el mismo para realizar la inicialización de las estructuras y demás. Si un cliente intenta ingresar al sistema, permanecerá bloqueado hasta que el servidor este listo para atenderlo.

El servidor será el encargado de administrar las estructuras de datos que albergan la agenda. Es el que, en definitiva, crea categorías y contactos, borra contactos, devuelve todo lo que se debe desplegar en al ejecutar *dir* del lado de un cliente, es el que da de alta un cliente y el que da de baja un cliente. Para todo esto, el servidor dispondrá de un arreglo de tamaño *MAX_ENTRADAS+1* de estructuras *entrada*, un arreglo de tamaño *MAX_USUARIOS* de estructuras *usuario* y dos contadores: cantidad de usuarios y cantidad de entradas.

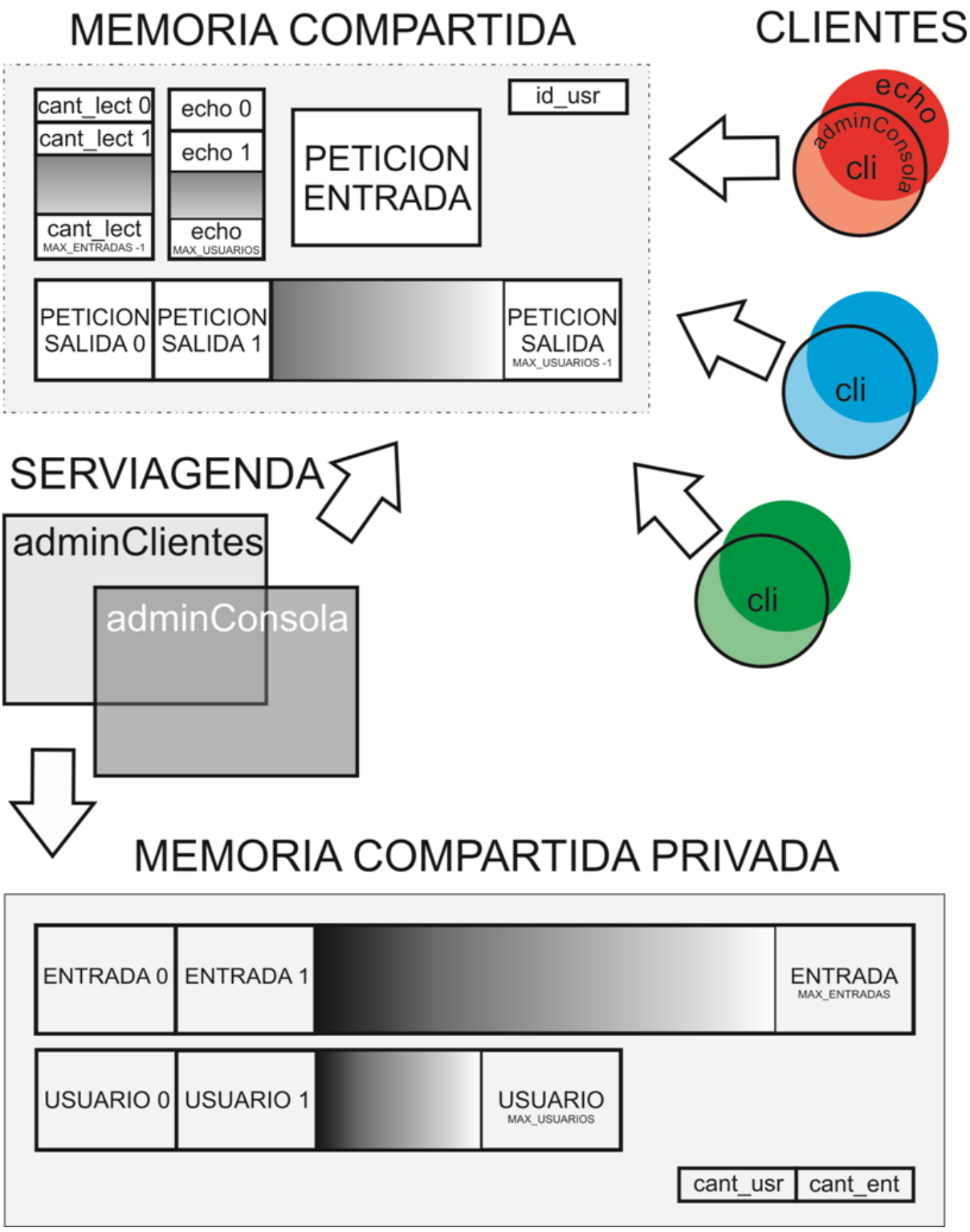

Ilustración 4.7.1

El servidor, al igual que el cliente, posee dos facetas. Por un lado, atiende peticiones, imprime la petición recibida y envía los *echos* correspondientes. Por otro, atiende los comandos *listar* y *salir*. Nuevamente tenemos un proceso demonio, el administrador de clientes. Ahora bien, si recordamos lo mencionado en la introducción de la herramienta *fork*, el declarar las estructuras como variables globales no nos ayudará, dado que estas se copiaran al proceso hijo, y ambos, padre e hijo tendrán copias distintas de las estructuras. Es por ello que tendremos las estructuras en memoria compartida de forma tal que el proceso padre y el proceso hijo accedan a los mismo datos. Sin embargo, a diferencia de las estructuras para la comunicación entre el servidor y los clientes, generaremos una llave única y desconocida mediante la bandera *IPC_PRIVATE.*

El servidor y los clientes tienen una relación distante, y corren de forma separada sin relacionarse y es deseable que los clientes no estén habilitados a acceder a la memoria donde se encuentran las estructuras de la agenda. Al setear en variables globales lo devuelto por *shmget* antes del fork, podremos acceder desde el proceso padre e hijo a la memoria compartida de forma privada.

```
int shmContID;          /* shmid de array de contadores [0] cantUsuarios, [1]
                           cantEntradas */
int shmEntradasID;      /* shmid de array de entradas */
int shmUsuariosID;      /* shmid de array de usuarios */

struct usuario
{
        int id; // -1 si el usuario no ha sido creado
        char nombre[MAX_NOMBRE];
        int idCategoria;
};

int main (int argc, char * argv[])
{
        if(inicializarSemaforos(CLAVE_SEM_ID_USUARIO,1,1) == -1)
        {
                cout<<"<El servidor ya esta corriendo>"<<endl;
                exit();
        }
        else
        {
                P(CLAVE_SEM_ID_USUARIO,1,0);

                inicializarSemaforosSoAgenda();
                inicializarMemoriaCompartidaSoAgenda();

                cout<<"<Se ha iniciado el servicio>"<<endl;
```

```
                /* Define el handler para la señal SIGCHLD */
                signal(SIGCHLD, catchChild);

                int childPid = fork();

                if (childPid == -1)
                {
                        cout <<"<ERROR: 'fork'>" << endl;
                        exit();
                }
                else if (childPid == 0)
                {
                        /* CODIGO PROCESO HIJO */
                        adminClientes();
                }
                else
                {
                        /* CODIGO PROCESO PADRE */
                        adminClientesId = childPid;
                        adminConsola();
                }
        }
}
```

En la función de inicialización asignamos los valores correspondientes a los enteros *shmContID*, *shmEntradasID*, *shmUsuariosID*.

```
void adminClientes()
{
        /* Define el handler para la señal SIGTERM */
        signal(SIGTERM, killHandler);
        peticionSalida* arraySalidas;   peticionEntrada petEntrada;
        peticionSalida petSalida;       echo echoAEnviar;           echo* arrayEcho;
        usuario* usuarios;              entrada* entradas;          int* contadores;
        while(true)
        {
                P(CLAVE_SEM_PETICION_PENDENTE,1,0);

                ... Se recoge la peticion de la memoria ...

                V(CLAVE_SEM_CERROJO_MEMORIA,1,0);

                switch(petEntrada.tipo)
                {
                        case ingreso:
                        {
                                if(ingresoUsuario() !=  ERROR_ALIAS_EXISTS)
                                {
```

```
                        petSalida.true
                else
                {
                        petSalida.exito = false;
                        petSalida.error = ERROR_ALIAS_EXISTS;
                }
                break;
        }
        case dir:
        {
                usuarios =
                (usuario*)mapear_segmento_id(shmUsuariosID);
                entradas =
                (entrada*)mapear_segmento_id(shmEntradasID);

                ...completar echoAEnviar...

                ...impresion del comando recibido...

                ...procesar el comando...

                liberar_segmento(usuarios);
                liberar_segmento(entradas);
                break;
        }
        case ver: ...
        case creacat: ...
        case creacon: ...
        case cdd: ...
        case cd: ...
        case borrar: ...
        case salir: ...
} /* switch */

/* se envia a los clientes que estan en la misma categoria el comando
recibidio del cliente */
if(petEntrada.tipo != ingreso)
{
        usuarios = (usuario*)mapear_segmento_id(shmUsuariosID);

        for(i = 0; i < MAX_USUARIOS; i++)
        {
                if      (usuarios[i].id != petEntrada.idUsuario &&
                        usuarios[i].idCategoria == categoria del usuario)
                {
                        P(CLAVE_SEMS_ECHO,MAX_USUARIOS,i);

                        ...se copia el echoAEnviar...
```

```
                                        /* despierto al echo de los clientes en la misma
                                        categoria */

                                        V(CLAVE_SEMS_ECHO_PENDIENTE,
                                        MAX_USUARIOS,usuarios[i].id);
                                }
                        }
                        liberar_segmento(usuarios);
                }

                ...Se escribe la respuesta en la memoria correspondiente al lugar del id
                cliente...

                /* Para dar por servida una petición */
                V(CLAVE_SEMS_PETICION_SERVIDA,
                MAX_USUARIOS,petEntrada.idUsuario);

        }

}
```

Observamos que la función del administrador de clientes es bastante mecánica. Una vez resueltos los problemas de sincronización solo debemos procesar el comando, y devolver una respuesta. El comando *ingreso* es un comando implícito y no tiene *echo*, aunque se procese como cualquier otro. Si un cliente solicita un ingreso al servidor, es porque 'hay lugar', es decir, porque el *ID_CLIENTE* es distinto de *MAX_USUARIOS* y por lo tanto el cliente y el servidor podrán usar el id disponible **momentáneamente** para comunicarse si el ingreso tiene éxito o si el alias utilizado ya pertenece a un cliente. Si el ingreso no tiene éxito, el *ID_CLIENTE* permanecerá incambiado, el cliente liberará la zona multiexcluída y otro cliente podrá intentar un ingreso.

El administrador de la consola es el proceso padre y será el que envié la señal *SIGTERM* al proceso hijo cuando se desee salir del serviagenda y no hay clientes conectados. Veamos:

```
void adminConsola()
{
        entrada* entradas;
        int* contadores;
        V(CLAVE_SEM_ID_USUARIO,1,0);
        while(true)
        {
                comando = leerComando();
                if(comando == salir)
```

```
                {
                        P(CLAVE_SEM_ID_USUARIO ,1,0);
                        if(cant_usr == 0)
                        {
                                destruirSemaforos();
                                destruirMemoriaCompartida();

                                kill(adminClientesId, SIGTERM);

                                cout<<"<Ha finalizado la ejecucion de
                                        serviagenda>"<<endl;
                                exit(0);
                        }
                        else
                        {
                                V(CLAVE_SEM_ID_USUARIO ,1,0);

                                cout<<"<No puedes salir del sistema en este
                                        momento>"<<endl;
                        }
                }
                else if(comando == listar)
                {
                        P(CLAVE_SEM_ENTRADAS,1,0);

                        impresionEntradas();

                        V(CLAVE_SEM_ENTRADAS,1,0);
                }
                else
                        cout<<"<Comando no reconocido>"<<endl;
        }
}
```

Cuando se ejecuta *salir* en el servidor, este debe chequear que la cantidad de usuarios en el sistema sea cero. Sin embargo, mientras realiza este chequeo puede suceder que un usuario intente ingresar y se produzca un conflicto. El hecho de que el servidor realice un P sobre el semáforo *SEM_ID_USUARIO* recae en que, el entero *ID_USUARIO* y la cantidad de usuarios en el sistema están muy relacionados y varían conjuntamente. El servidor bloquea la entrada de usuarios y cheque si no hay usuarios en el sistema. Si no los hay entonces se destruyen todos los elementos creados al ejecutarse. Si en el momento en que el servidor esta leyendo la cantidad de usuarios, un cliente intenta ingresar entonces se quedará bloqueado. Al destruirse la memoria compartida y los semáforos, el cliente bloqueado se despertará por un error y terminará su ejecución.

El semáforo *CLAVE_SEM_ENTRADAS* es utilizado para evitar conflictos con la creación de entradas y contactos y con el borrado de contactos.

4.8 Escritores y lectores

Una de las restricciones a resolver en la implementación del obligatorio es la serialización de los comandos *creacon, creacat* y todos los demás. Mientras un cliente ejecuta un *creacat* o un *creacon* ningún otro cliente podrá ejecutar otro comando (cualquiera) sobre la misma categoria en la que se encuentra el primer cliente. Si lo traducimos a escritores y lectores, dos escritores no pueden escribir a la vez, ningún lector puede leer mientras se está escribiendo y varios lectores pueden leer a la vez. Donde los escritores son: *creacat* y cracon, los lectores: todos los demás comandos. El algoritmo utilizado para resolver este problema es el siguiente: (semáforos *wrt* y *E* inicializados en 1)

```
escritor()
{
        P(wrt);
        escribir;
        V(wrt);
}

lector()
{
        P(E);
        cant_lect++;
        if(cant_lect == 1)
                P(wrt)
        V(E);
        leer;
        P(E);
        cant_lect--;
        if(cant_lect == 0)
                V(wrt)
        V(E);
}
```

Si aplicamos el algoritmo al obligatorio tendremos este problema pero por cada categoría. Los semáforos *wrt* y *E* seran arrays de semáforos de tamaño *MAX_ENTRADAS*, ya que sería posible tener tantas categorías como *MAX_ENTRADAS*. Cuando un cliente ejecute un comando, si se trata de un *creacat* o un *creacon* entonces realizará P sobre un semáforo del array *SEMS_WRITERS*, de forma análoga al algoritmo, pero sobre el semáforo correspondiente a la categoría

en la que se encuentra. Y es por este simple hecho que el cliente debe conocer el id de la categoría en la que se encuentra. Si el comando es otro, entonces se seguirá el algoritmo para lectores pero generalizándolo a arrays de semáforos.

En lugar de existir un solo contador de lectores, existirá un contador por cada categoría, *MEM_CANT_READERS*.

4.9 Impresión

Un detalle no menor, y que creemos es importante resaltar es la impresión en pantalla de los clientes y los usuarios.

Cuando se ejecuta un comando como *listar* o *dir*, es probable que suceda que entre la lista de entradas, se imprima un *echo*. Es por ello que se disponen de semáforos para multiexcluir la impresión en el servidor y para cada cliente.

4.10 Ctrl-C

La señal Ctrl-C es controlada en SoAgenda del lado del servidor y del lado de los clientes.

El handler de esta señal pretende ser rápido y dejar el sistema estable. Del lado del cliente, no hay restricciones para su salida, pero si este no realiza una petición de salida al servidor, el servidor tarde o temprano se bloqueará intentando enviarle algún *echo*. Si no se desea dañar al sistema, se debe si o si procesar una petición de salida, que es precisamente lo que se hace al presionar Ctrl-C del lado del cliente.

El handler de esta señal del lado del servidor simplemente imprime en pantalla "<*salir*>", indicando al usuario de que forma se debe salir del servidor. Ahora bien, como el handler de esta señal no realiza un *exit* entonces podria suceder que se precione Ctrl-C cuando el servidor esta bloqueado en un semáforo. Esto generaria que luego de la impresión, el cliente accediera a la zona multiexcluida sin haber obtenido el recurso. Este simple hecho, nos lleva a reconsiderar la función P implementada y realizar modificaciones.

Si el valor obtenido de la función *semop* al realizar un P es -1 y errno se encuentra en *EINTR* entonces, se obtuvo la zona multiexcluida por error y se debe volver a realizar P sobre el semáforo. Nuestra función entonces será:

int P(char clave, int cantSemaforos, int sem_num)

```
{
        int id = obtenerIdArraySemaforos(clave,cantSemaforos);
        if(id == -1)
                return -1;
        int e;
        struct sembuf sem_lock = {sem_num, -1, 0};
        do
        {
                semop(id, &sem_lock, 1);
        }while(e == -1 && errno == EINTR);
}
```

5 Conclusión

Para finalizar el informe sobre la realización del obligatorio número tres, es Importante mencionar los objetivos cumplidos.

El proceso hacia la formalización, ha tenido como propósito, según lo mencionado en ocasiones anteriores, ser coherentes en el desarrollo del informe, a diferencia de las ocasionales contradicciones que pueden encontrarse en piezas de información varias. Pretendimos fijar nuestra atención en aspectos importantes y esenciales para la comprensión del desarrollo.

Durante la preparación del trabajo los cambios y variantes, tanto de detalles como de perspectiva se hicieron numerosos y de múltiples estratos, sin embargo, en busca de una lectura confiable y coherente, basamos las notas, observaciones y conclusiones en los textos bibliográficos.

Antes de la implementación de la agenda, se intentó introducir al lector al aspecto teórico, y definir en forma precisa el vocabulario utilizado en etapas posteriores. Durante la descripción del desarrollo de serviagenda, cliagenda, y demás, se expusieron imágenes y tablas a modo de esclarecer la situación y agilizar la lectura.

Al probar el funcionamiento de la agenda, en lugar de llevar a cabo solo lo explicitado por la letra, se perseveró en obtener los resultados correspondientes a entradas claves. Aunque la prueba exhaustiva es imposible, estamos seguros de habernos enfocado en los puntos más importantes.

Debemos mencionar que las decisiones tomadas a lo largo de la implementación surgieron como respuesta a los resultados obtenidos hasta el momento.

No resulta fácil exponer una síntesis final que valga como resumen de los tres obligatorios de tan largo proceso y aspectos tan diversos. Reconocemos los riesgos que se derivan de toda investigación cuyo objeto suponga abarcar más de las pautas planteadas. Sin embargo, debemos admitir que han sido, los tres, actividades integradoras, enriquecedoras desde el punto de vista técnico y didáctico. El ejercicio o adiestramiento en la utilización de las herramientas estudiadas es fundamental para comprender mejor los postulados teóricos y creemos son el objetivo de las cláusulas en los obligatorios.

6 Bibliografía

Linux cross reference, http://lxr.linux.no/

Javier J. Gutiérrez, Prácticas de Sistemas Operativos.

Toñi Reina, David Ruiz y Juan Antonio Álvarez, Prácticas de Sistemas Operativos.Curso 2004/05.Boletín #6: Semáforos

Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel (3ª Edición) , Ed. O'Reilly, 2005, Maxvell, Remy Card.

http://linux.die.net

Sistema cliente-servidor, Universidad de Las Palmas de Gran Canaria. Sistemas Operativos. Práctica de comunicación entre procesos curso 1996/97.

Kernighan/Pike. El entorno de programación Unix.