Sistemas Operativos – Práctica 2

FECHA DE ENTREGA: SEMANA 5-9 ABRIL (HORA LÍMITE DE ENTREGA: UNA HORA ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS).

La segunda práctica se va a desarrollar en dos semanas con el siguiente cronograma
Semana 1: Señales
Semana 2 y 3: Semáforos

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

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

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

SEMANA 1

Señales

Las señales son una forma limitada de comunicación entre procesos. Para comparar se puede decir que las señales son a los procesos lo que las interrupciones son al procesador. Cuando un proceso recibe una señal detiene su ejecución, bifurca a la rutina del tratamiento de la señal que está en el mismo proceso y luego una vez finalizado sigue la ejecución en el punto que había bifurcado anteriormente.

Las señales son usadas por el núcleo para notificar a los procesos sucesos asíncronos. Por ejemplo, (1) si se pulsa *Ctrl+C*, el núcleo envía la señal de interrupción *SIGINT,* (2) excepciones de ejecución. Por otro lado, los procesos pueden enviarse señales entre sí mediante la función *kill()* siempre y cuando los procesos tengan el mismo UID.

Cuando un proceso recibe una señal puede reaccionar de tres formas diferentes:

- Ignorar la señal, con lo cual es inmune a la misma.
- Invocar a la rutina de tratamiento de la señal por defecto. Esta rutina no la codifica el programador, sino que la aporta el núcleo. Por lo general suele provocar la terminación del proceso mediante una llamada a *exit()*. Algunas señales no sólo provocan la terminación del proceso, sino que además hacen que el núcleo genere, en el directorio de trabajo actual del proceso, un fichero llamado core que contiene un volcado de memoria del contexto del proceso.
- Invocar a una rutina propia que se encarga de tratar la señal. Esta rutina es invocada por el núcleo en el supuesto de que esté montada y será responsabilidad del programador codificarla para que tome las acciones pertinentes como tratamiento de la señal.

Cada señal tiene asociado un número entero y positivo y, cuando un proceso le envía una señal a otro, realmente le está enviando ese número. En el fichero de cabecera <signal.h> están definidas las señales que puede manejar el sistema.

Tabla 1 Señales UNIX System V

Nombre	Explicación	Número	Acción por defecto			No se puede ignorar	Restaura rutina por defecto
			Genera core	Termina	Ignora		
SIGHUP	Desconexión	1		✓			✓
SIGINT	Interrupción	2		✓			✓
SIGQUIT	Salir	3	✓	✓			✓
SIGILL	Instrucción Ilegal	4	✓	✓			
SIGTRAP	Trace trap	5	✓	✓			
SIGIOT	I/O trap instruction	6	✓	✓			✓
SIGEMT	Emulator trap instruction	7	✓	✓			✓
SIGFPE	Error en coma flotante	8	✓	✓			✓
SIGKILL	Terminación abrupta	9		✓		✓	✓
SIGBUS	Error de bus	10	✓	✓			✓
SIGSEGV	Violación de segmento	11	✓	✓			✓
SIGSYS	Argumento erróneo en la llamada a sistema	12	✓	✓			✓
SIGPIPE	Intento de escritura en una tubería de la que no hay nadie leyendo	13		✓			✓
SIGALRM	Despertador	14		✓			✓
SIGTERM	Finalización Controlada	15		✓			✓
SIGUSR1	Señal número 1 de usuario	16		✓			✓
SIGUSR2	Señal número 2 de usuario	17		✓			✓
SIGCLD	Terminación del proceso hijo	18			✓		✓
SIGPWR	Fallo de alimentación	19			✓		

Envío de señales

#include <signal.h>

int kill (pid_t pid, int sig);

- Manda señales entre procesos.
- Argumentos: PID del proceso al que se enviará la señal, entero con la señal.
  - pid > 0, es el PID al que le enviamos la señal.
  - pid=0, la señal es enviada a todos los procesos que pertenecen al mismo grupo que el proceso que la envía.
  - pid=-1, la señal es enviada a todos aquellos procesos cuyo identificador real es igual al identificador efectivo del proceso que la envía. Si el proceso que la envía tiene identificador efectivo de superusuario, la señal es enviada a todos los procesos, excepto al proceso 0 –swapper- y al proceso 1 –init-.
  - pid<-1, la señal es enviada a todos los procesos cuyo identificador de grupo coincide con el valor absoluto de pid.

En todos los casos, si el identificador efectivo del proceso no es el del superusuario o si el proceso que envía la señal no tiene privilegios sobre el proceso que la va a recibir, la llamada a *kill* falla.

El parámetro sig es el número de la señal que queremos enviar. Si sig vale 0 –señal nula- se efectúa una comprobación de errores, pero no se envía ninguna señal.

- Retorno: 0 si el envío es correcto, -1 en caso de error.

Ejercicio 1. (APRENDIZAJE) Ejecuta en línea de comando la orden $ kill –l

¿Qué obtienes?

Ejercicio 2. (ENTREGABLE) 1 ptos. Escribe un programa en C, ***ejercicio2.c,*** que creará 4 procesos hijos en un bucle de forma paralela. Cada hijo, imprimirá el mensaje "Soy el proceso hijo <PID>", después dormirá 30 segundos, imprimirá el mensaje "Soy el proceso hijo <PID> y ya me toca terminar.". Tras imprimir este mensaje, el hijo finalizará su ejecución. El padre, tras crear un hijo, dormirá 5 segundos y después enviará la señal SIGTERM al hijo que acaba de crear. A continuación, creará el siguiente hijo.

¿Qué mensajes imprime cada hijo? ¿Por qué?

Tratamiento de señales

sighandler_t signal(int signum, sighandler_t handler).

- Captura de señales. Cambia el manejador al definido (puede ser una función, ignorar o comportamiento por defecto).
- Argumentos: entero con la señal que se va a capturar, manejador nuevo de la señal.
- Retorno: valor previo del manejador, *SIG_ERR* en caso de error.

#include <signal.h>

*void (*signal (int sig, void (*action) ( ))) ( );*

- sig es el número de la señal que se va a capturar
- action es la acción que se tomará al recibir la señal y puede tomar tres clases de valores:
  - SIG_DFL, indica que la acción a realizar cuando se recibe la señal es la acción por defecto asociada a la señal –manejador por defecto-. Por lo general, esta acción consiste en terminar el proceso y en algunos casos también incluye generar un fichero core.
  - SIG_IGN, indica que la señal se debe ignorar
  - dirección de la rutina de tratamiento de la señal (manejador suministrado por el usuario); la declaración de esta función debe ajustarse al modelo:

    #include<signal.h>

    void handler (int sig [, int code, struct sigcontext *scp]);

    Cuando se recibe la señal sig, el núcleo es quien se encarga de llamar a la rutina handler pasándole los parámetros sig, code y scp.

    La llamada a la rutina handler es asíncrona, lo cual quiere decir que puede darse en cualquier instante de la ejecución del programa.

Los valores SIG_DFL, SIG_IGN y SIG_ERR son direcciones de funciones ya que los debe poder devolver signal:

#define SIG_DFL ((void (*) ( )) 0)

#define SIG_IGN ((void (*) ( )) 1)

#define SIG_ERR ((void (*) ( )) -1)

La conversión explícita de tipo que aparece delante de las constantes -1, 1 y 0 fuerza a que estas constantes sean tratadas como direcciones de inicio de funciones. Estas direcciones no contienen ninguna función, ya que en todas las arquitecturas UNIX son zonas reservadas para el núcleo. Además la dirección -1 no tiene existencia física.

Ejemplo de uso:

```
#include <stdio.h>
#include <signal.h>

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

     void manejador_SIGINT( );

if(signal (SIGINT, manejador_SIGINT)==SIG_ERR){
     perror("signal");
     exit(EXIT_FAILURE);
}

while(1){
       printf("En espera de Ctrl+C \n");
       sleep(9999);
 }/*End while*/

} /*End main*/

/****
     Manejador_SIGINT rutina de tratamiento de la señal SIGINT
*****/

void manejador_SIGINT (int sig){
     printf("Señal número %d recibida \n", sig);
}
```

Al ejecutar este programa, cada vez que se pulsa Ctrl+C aparece el mensaje por pantalla. Para volver a restaurar el comportamiento por defecto asociado a Ctrl+C, hemos de armar el manejador por defecto al final de nuestra rutina de tratamiento:

```
void manejador_SIGINT (int sig)
{
     printf("Señal número %d recibida \n", sig);
     if(signal (SIGINT, SIG_DFL)==SIG_ERR){
        perror("signal");
        exit(EXIT_FAILURE);
     }
}
```

Ejercicio 3. **(APRENDIZAJE)** Sea el siguiente código

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void captura (int sennal)
{
     printf ("Capturada la señal %d  \n", sennal);
     fflush (NULL);
     return;
}

int main (int argc, char *argv [], char *env [])
{
     if (signal (SIGINT, captura) == SIG_ERR)
     {
          puts ("Error en la captura");
          exit (1);
     }
     while (1);
     exit (0);
}
```

Contesta a las siguientes cuestiones:

a. ¿La llamada a *signal* supone que se ejecute la función captura?

b. ¿Cuándo aparece el *printf* en pantalla?

c. ¿Qué ocurre por defecto cuando un programa recibe una señal y no la tiene capturada?

d. El código de un programa captura la señal *SIGKILL* y la función manejadora escribe *"He conseguido capturar SIGKILL"*. ¿Por qué nunca sale por pantalla *"He conseguido capturar SIGKILL"*?

Espera de señales

#include <unistd.h>

int pause(void);

- Bloquea al proceso que lo invoca hasta que llegue una señal. No permite especificar el tipo de señal por la que se espera. Sólo la llegada de cualquier señal no ignorada ni enmascarada sacará al proceso del estado de bloqueo.
- Retorno: -1. En otras llamadas al sistema, esta terminación es una condición de error, en este caso es su forma correcta de operar.

Ejemplo de uso:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

/***
 main: arma los manejadores de SIGTERM y SIGUSR1 y se pone a esperar
señales
***/

int main(int argc, char *argv[ ])
{
     void manejador_SIGUSR1 ( );
     void manejador_SIGTERM ( );

     signal(SIGTERM, manejador_SIGTERM); /* Armar la señal */
     signal(SIGUSR1, manejador_SIGUSR1); /*Armar la señal */

     while(1)
       pause(); /* Bloquea  al proceso  hasta que llegue una señal*/
}

/***
  manejador_SIGTERM saca un mensaje por pantalla y termina el proceso.
***/

void manejador_SIGTERM (int sig)
{
      printf("Terminación del proceso %d a petición del usuario \n",
getpid( ));
      exit(-1);
}
/****
   Manejador_SIGUSR1: presenta un número aleatorio por pantalla.
***/

void manejador_SIGUSR1 (int sig)
```

```
{
        signal(sig, SIG_IGN);
        printf("%d \n", rand( ));
        signal(sig, manejador_SIGUSR1); /* Restaura rutina por defecto */
}
```

Para ejecutar este programa y que muestre los números aleatorios, debemos enviarle la señal SIGUSR1 con la orden kill,

$ kill –s USR1 pid

Podemos terminar la ejecución enviándole la señal SIGTERM desde otra terminal:

$ kill –15 pid

Ejercicio 4. (ENTREGABLE) 2 ptos Escribe un programa en C llamado ***Ejercicio4.c***. De forma general, en este ejercicio un padre creará un proceso hijo que realizará un trabajo. Pasado un tiempo, solicitará al padre que le releve un nuevo proceso hijo. El padre creará un nuevo proceso hijo y será, este nuevo proceso hijo, el encargado de avisar al hijo anterior de que ya puede terminar. Una vez avisado, el nuevo hijo empezará a trabajar.

Tras crear N hijos, el padre cambiará el comportamiento al recibir la solicitud de un hijo para ser relevado. Este nuevo comportamiento consistirá en no crear un nuevo relevo, enviar una señal al proceso que solicita ser relevado para que termine y, a continuación, terminará también el padre, asegurándose de que no queda ningún proceso hijo vivo.

De forma más específica, el trabajo que realizará cada hijo consistirá en:

- Imprimir el mensaje "Soy <PID> y estoy trabajando" y dormir un segundo repetidamente.
- Pasado un tiempo de 10 segundos (tras imprimir 10 veces el mensaje), este proceso hijo enviará una señal al proceso padre solicitando que otro hijo le tome el relevo. Mientras tanto, seguirá imprimiendo el mensaje cada 1 segundo.

Protección de zonas críticas

Máscaras

En ocasiones puede interesarnos proteger determinadas zonas de código contra la llegada de alguna señal. La máscara de señales de un proceso define un conjunto de señales cuya recepción serán bloqueadas. Bloquear una señal es distinto de ignorarla. Cuando un proceso bloquea una señal, ésta no será enviada al proceso hasta que se desbloquee o ignore. Si el proceso ignora la señal, ésta simplemente se deshecha.

La máscara que indica las señales bloqueadas en un proceso o hilo determinado es un objeto de tipo sigset_t al que llamamos conjunto de señales y está definido en <signal.h>. Aunque sigset_t suele ser un tipo entero donde cada bit está asociado a una señal, no necesitamos conocer su estructura y estos objetos pueden ser manipulados con funciones específicas para activar y desactivar los bits correspondientes.

#include <signal.h>

*int sigfillset (sigset_t *set);*

*int sigemptyset (sigset_t *set);*

La función *sigfillset()* incluye en el conjunto referenciado por set todas las señales definidas. Si utilizamos este conjunto como máscara todas las señales serán bloqueadas.

La función *sigemptyset()* excluye del conjunto referenciado por set todas las señales, desbloqueando así su recepción si utilizamos este conjunto como máscara.

Estas funciones devuelven 0 si se ejecutan satisfactoriamente o -1 en caso de error.

Una vez que se ha iniciado un conjunto podemos manipularlo para incluir una señal, excluirla o preguntar si está incluida.

#include <signal.h>

*int sigaddset (sigset_t *set, int signo);*

*int sigdelset(sigset_t *set, int signo);*

*int sigismember(const sigset_t *set, int signo);*

En las tres funciones set es la referencia al conjunto y signo es el número de la señal. Las dos primeras devuelven 0 si se ejecutan satisfactoriamente y -1 en caso de error y la tercera devuelve 1 si la señal signo pertenece al conjunto, 0 si no pertenece y -1 en caso de error.

A partir de un conjunto podemos modificar la máscara de señales bloqueadas en un proceso

#include <signal.h>

*int sigprocmask (int how, const sigset_t *set, sigset_t *oset);*

El parámetro set es una referencia al conjunto que se utilizará para modificar la máscara de señales de acuerdo con las condiciones de how: SIG_BLOCK, SIG_SETMASK y SIG_UNBLOCK

SIG_BLOCK: La máscara resultante es el resultado de la unión de la máscara actual y el conjunto.

SIG_SETMASK: La máscara resultante es la indicada en el conjunto.

SIG_UNBLOCK: La máscara resultante es la intersección de la máscara actual y el complementario del conjunto. Es decir, las señales incluidas en el conjunto quedarán desbloqueadas en la nueva máscara de señales.

En la zona de memoria referenciada por oset se almacena la máscara de señales anterior y podremos usar este puntero para restaurarla con posterioridad.

Si el puntero set vale NULL es equivalente a consultar el valor de la máscara actual sin modificarla.

Retorno: la función devuelve 0 si se ejecuta satisfactoriamente y -1 en caso de error.

Ejemplo de uso:

```
sigset_t set, oset;
int error;
…
```

```
sigemptyset(&set);

sigaddset(&set, SIGFPE); /* Máscara que bloqueará la señal por error en coma
flotante*/
sigaddset(&set, SIGSEGV); /* Añade a la máscara el bloqueo por violación de
segmento */

error = sigprocmask(SIG_BLOCK, &set,&oset); /*Bloquea la recepción de las
señales SIGFPE y SIGSEV en el proceso */

if(error)
  // Tratamiento del error
```

#include <signal.h>

*int sigpending (sigset_t *mask)*.

Devuelve el conjunto de señales bloqueadas que se encuentran pendientes de entrega al proceso. El servicio almacena en mask el conjunto de señales bloqueadas pendientes de entrega.

Si la función se ejecuta satisfactoriamente devuelve 0, en caso contrario devuelve -1

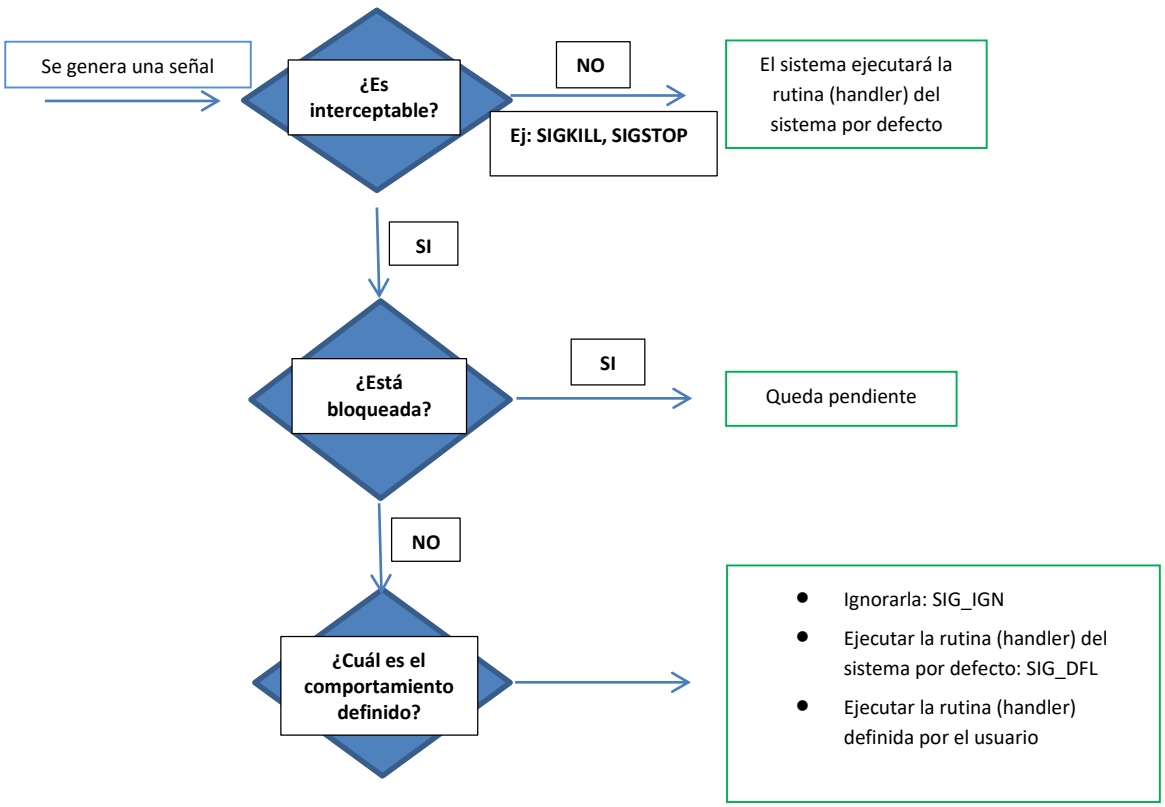

Ilustración 1Recepción y Manejo de Señales

En espera de señales (sigsuspend)

#include <signal.h>

*int sigsuspend (const sigset_t *mask)*;

La llamada sigsuspend permite bloquear un proceso hasta que se reciba alguna de las señales deseadas. De forma atómica, sustituye la máscara actual de señales (es decir, las señales bloqueadas) por la que recibe y queda en espera similar a la función pause. Si con pause podíamos parar un proceso en espera de la primera señal que se reciba, con sigsuspend podemos seleccionar la señal por la que se espera.

#include <signal.h>

*long sigsuspend (const sigset_t *mask);*

la función *sigsuspend*() bloquea la recepción de señales de acuerdo con el valor de mask. La espera se realiza sobre las señales no bloqueadas.

Cuando *sigsuspend()* termina su ejecución, restaura la máscara de señales que había antes de llamarla. La ejecución de *sigsuspend()* termina cuando es interrumpida por una señal.

Servicios de temporización

En determinadas ocasiones puede interesarnos que el código se ejecute de acuerdo con una temporización determinada. Esto puede venir impuesto por las especificaciones del programa, donde una respuesta antes de tiempo puede ser tan perjudicial como una respuesta con retraso.

#include <unistd.h>

unsigned int alarm(unsigned int seconds).

- Establece que se enviará una señal *SIGALRM* al cabo de *seconds* segundos.
- Retorno: número de segundos hasta la siguiente alarma (que se sobrescribe), 0 si no había ninguna otra alarma.

Esta llamada activa un temporizador que inicialmente toma el valor seconds segundos y que se decrementará en tiempo real. Cuando hayan transcurrido los seconds segundos, el proceso recibirá la señal SIGALRM. El valor de seconds está limitado por la constante MAX_ALARM, definida en <sys/param.h>. En todas las implementaciones se garantiza que MAX_ALARM debe permitir una temporización de al menos 31 días. Si seconds toma un valor superior al de MAX_ALARM, se trunca al valor de esta constante.

Para cancelar un temporizador previamente declarado, haremos la llamada alarm(0).

Una alarma no es heredada por un proceso hijo, después de la llamada a fork.

Ejercicio 5. (APRENDIZAJE) Estudia qué hace el siguiente programa.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define SECS 20

void captura(int sennal)
{
    printf("\nEstos son los numeros que me ha dado tiempo a contar
en %d segundos\n", SECS);

    exit(0);
}

int main(int argc, char *argv [], char *env [])
{
    long int i;

    if (signal(SIGALRM, captura) == SIG_ERR)
    {
        puts("Error en la captura");

        exit (EXIT_FAILURE);
    }
    if (alarm(SECS))
        fprintf(stderr, "Existe una alarma previa establecida\n");

    for (i=0;;i++)
        fprintf(stdout, "%10ld\r", i);

    fprintf(stdout, "Fin del programa\n");

    exit(0);
}
```

Procesos Padre-Hijo Herencia

Después de la llamada a la función *fork()*, el proceso hijo:

- Hereda la máscara de señales bloqueadas
- Tiene vacía la lista de señales pendientes
- Hereda las rutinas de manejo (handler)
- No hereda las alarmas

Tras una llamada a una función *exec()* el proceso lanzado

- Hereda la máscara de señales bloqueadas
- Mantiene la lista de señales pendientes
- No hereda las rutinas de manejo

Ejercicio 6. **(ENTREGABLE) 1ptos** Dado el siguiente programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
```

```
#include <time.h>

#define NUM_PROC 5

int main (void)
{
    int pid, counter;
    pid = fork();
    if (pid == 0){
        while(1){
            for (counter = 0; counter < NUM_PROC; counter++){
                printf("%d\n", counter);
                sleep(1);
            }
            sleep(3);
        }
    }
    while(wait(NULL)>0);
}
```

Ejercicio6a.c Modifica el código del enunciado para que:

- El hijo, justo después de ser creado, establecerá una alarma para ser recibida dentro de 40 segundos.
- Justo antes de comenzar cada bucle de imprimir números en un proceso hijo, las señales SIGUSR1, SIGUSR2 y SIGALRM deben quedar bloqueadas.
- Al finalizar el bucle de impresión de números, y antes de la espera de 3 segundos, se desbloqueará la señal SIGALRM y SIGUSR1.
- **¿Qué sucede cuando el hijo recibe la señal de alarma?**

Ejercicio6b.c Modifica el código del enunciado para que:

- El padre enviará la señal SIGTERM al proceso hijo cuando hayan pasado 40 segundos de la creación del hijo.
- El proceso hijo, al recibir la señal SIGTERM, imprimirá el mensaje “Soy <PID> y he recibido la señal SIGTERM” y finalizará su ejecución.
- Cuando el hijo haya finalizado su ejecución, el padre finalizará.

SEMANA 2

Semáforos

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

Existen dos tipos básicos de semáforos:

- *Semáforo binario*: sólo puede tomar dos valores, 0 y 1. Cuando está a 0 bloquea el acceso del proceso a la sección crítica, mientras que cuando está a 1 permite el paso (poniéndose además a 0 para bloquear el acceso a otros procesos posteriores). Este tipo de semáforos es especialmente útil para garantizar la exclusión mutua a la hora de realizar una tarea crítica. Por ejemplo, para controlar la escritura de variables en memoria compartida, de manera que

sólo se permita que un proceso esté en la sección crítica mientras que se están modificando los datos.

- *Semáforo N-ario*: puede tomar valores desde 0 hasta N. El funcionamiento es similar al de los semáforos binarios. Cuando el semáforo está a 0, está cerrado y no permite el acceso a la sección crítica. La diferencia está en que puede tomar cualquier otro valor positivo además de 1. Este tipo de semáforos es muy útil para permitir que un determinado número de procesos trabajen concurrentemente en alguna tarea no crítica. Por ejemplo, varios procesos pueden estar leyendo simultáneamente de la memoria compartida, mientras que ningún otro proceso intente modificar datos.

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

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

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

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

La función *semget* define un *array* completo de semáforos del tamaño especificado. Es decir, por defecto hay que trabajar con conjuntos de semáforos, de forma que para crear un único semáforo es necesario crear un *array* de un solo elemento. Como primer parámetro se le va a pasar un identificador de IPC (Inter Process Communication). Con la llamada *ftok()*, es posible convertir una ruta del sistema en un identificador IPC. Para realizar esta llamada se deben declarar una ruta del sistema y una clave numérica:

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

#define KEY 1300
```

Después sólo hay que invocar la llamada *ftok()* con estos dos argumentos:

```
int key = ftok(FILEKEY, KEY);
```

El segundo parámetro es el número de semáforos que queremos que tenga el conjunto que queremos usar y el último parámetro es un campo de bits que nos permite especificar opciones. De estas opciones, interesa IPC_CREAT. Si varios procesos hacen un semget para acceder a un semáforo, uno de ellos ha de especificar la macro IPC_CREAT con un OR (|) de los permisos con los que quiere crear el semáforo. El resto especificará 0 como último parámetro.

La llamada al sistema, si no hay error, devolverá un identificador para el conjunto de semáforos declarado (semid). Este identificador es similar al descriptor de fichero de los ficheros, es decir, se

usará en todas las llamadas al sistema con las que queramos operar sobre el semáforo recién creado. En caso de error devuelve el valor -1.

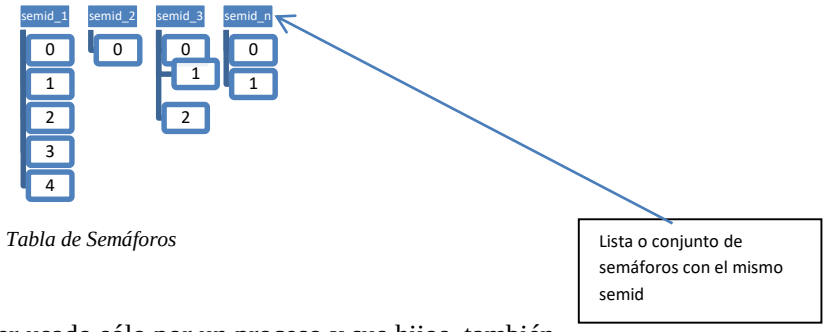

Tabla de Semáforos

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

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

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

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

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

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

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

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

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

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

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

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

- GETALL: permite leer el valor de todos los semáforos asociados al identificador *semid*. Estos valores se guardan en *arg*.
- SETALL: sirve para iniciar el valor de todos los semáforos asociados al identificador *semid*. Los valores de inicialización deben estar en *arg*.
- IPC_STAT e IPC_SET: nos permite conocer o establecer ciertas propiedades del array de semáforos. Las propiedades se especifican o se leen de una estructura de tipo struct semid_ds pasada por referencia. Ejemplo:

- GETPID: devuelve el PID del último proceso que actuó sobre el semáforo.
- GETNCNT: devuelve el número de procesos bloqueados en la cola del semáforo.

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

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

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

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

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

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

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

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

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

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

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

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

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

encarga de actualizar el valor del semáforo, deshaciendo las operaciones realizadas sobre él, cuando el proceso termina.

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

Ejercicio 7. (APRENDIZAJE) Estudia el siguiente código. Se ha ilustrado en él la creación de un array de semáforos, se ha operado sobre ellos y finalmente se liberan. Comprueba al finalizar el programa si los semáforos han sido liberados: $ ipcs -s

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>
#include <sys/shm.h>
#include <stdlib.h>
#define SEMKEY 75798
#define N_SEMAFOROS 2
int main ( )
{
    /*
    * Declaración de variables
    */
    int sem_id; /* ID de la lista de semáforos */
    struct sembuf sem_oper; /* Para operaciones up y down
    sobre semáforos */
    union semun {
        int val;
        struct semid_ds *semstat;
        unsigned short *array;
    } arg;
    /*
    * Creamos una lista o conjunto con dos semáforos
    */
    sem_id = semget(SEMKEY, N_SEMAFOROS, IPC_CREAT | IPC_EXCL | SHM_R | SHM_W);
    if((sem_id == -1) && (errno == EEXIST)){
        sem_id=semget(SEMKEY,N_SEMAFOROS,SHM_R|SHM_W);
    }
    if(sem_id==-1){
        perror("semget");
        exit(errno);
    }
    /*
    * Inicializamos los semáforos
    */
    arg.array = (unsigned short *)malloc(sizeof(short)*N_SEMAFOROS);
    arg.array [0] = arg.array [1] = 1;
    semctl (sem_id, N_SEMAFOROS, SETALL, arg);
    /*
    * Operamos sobre los semáforos
    */
    sem_oper.sem_num = 0; /* Actuamos sobre el semáforo 0 de la lista */
    sem_oper.sem_op =-1; /* Decrementar en 1 el valor del semáforo */
    sem_oper.sem_flg = SEM_UNDO; /* Para evitar interbloqueos si un
    proceso acaba inesperadamente */
    semop (sem_id, &sem_oper, 1);
    sem_oper.sem_num = 1; /* Actuamos sobre el semáforo 1 de la lista */
    sem_oper.sem_op = 1; /* Incrementar en 1 el valor del semáforo */
    sem_oper.sem_flg = SEM_UNDO; /* No es necesario porque ya se ha
    hecho anteriormente */
```

```
    semop (sem_id, &sem_oper, 1);
    /*
    * Veamos los valores de los semáforos
    */
    semctl (sem_id, N_SEMAFOROS, GETALL, arg);
    printf ("Los valores de los semáforos son %d y %d\n", arg.array [0], arg.array
[1]);
    /* Eliminar la lista de semáforos */
    semctl (sem_id, N_SEMAFOROS, IPC_RMID, 0);
    free(arg.array);
}/* fin de la función main */
```

Ejercicio 8. (ENTREGABLE) (2.0 ptos) Construcción de la biblioteca de semáforos. Se pide construir la biblioteca de semáforos semaforos.h. La biblioteca contendrá las siguientes funciones:

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

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

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

      int Borrar_Semaforo(int semid);

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

```
                0 si ha creado el semaforo,
               1 si ya estaba creado.
*************************************************************/

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


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

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

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

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


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

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



/*************************************************************
      Nombre: UpMultiple_Semaforo
```

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

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

Ejercicio 9. **(ENTREGABLE) (4 ptos)** En este ejercicio se implementará un sistema de gestión de transacciones bancarias en un hipermercado. Así, tendremos una cuenta global (guardada por el proceso padre) que contabiliza el total de dinero recaudado por todas las cajas. Habrá N cajas que serán las encargadas de realizar las transacciones con los clientes. Para simular las operaciones de los clientes, el proceso padre preparará un fichero para cada caja (*clientesCaja1.txt*, *clientesCaja2.txt* …) donde rellenará un listado de 50 operaciones aleatorias de compra, consistentes en escribir en cada línea del fichero una cantidad aleatoria entera entre 0 y 300€.

El proceso padre generará tantos procesos hijo (cajeros) como cajas, y cada hijo guardará el total de dinero existente en su caja mediante un fichero, incrementando la cantidad en cada una de las operaciones realizadas con clientes. Para ello, realizará lecturas consecutivas del fichero de operaciones de clientes asociado a esa caja e irá incrementando su balance en función de la cantidad pagada por el cliente. Para simular el tiempo de gestión de cada cliente, cuando un cajero haga una lectura del fichero realizará también una espera aleatoria de entre 1 y 5 segundos.

Las cajas, al superar los 1000€, solicitarán que se les retire dinero por seguridad. El cajero seguirá realizando operaciones y el proceso padre, cuando tenga la primera oportunidad (la cuenta de la caja no esté siendo modificada), retirará 900€ de la caja y se los llevará a la cuenta global. También avisarán al proceso padre de que han procesado a todos los clientes y que, el remanente en la caja debe ser retirado. Las operaciones de retirada de dinero de las diferentes cajas se realizarán por parte del proceso padre.

Siempre que se realice una operación de lectura de un fichero, el fichero se abrirá y al realizar la operación de lectura o escritura correspondiente, el fichero se cerrará.

Se deberán utilizar señales para avisar de los distintos eventos entre proceso padre y procesos hijos (por ejemplo, un proceso hijo avisa al padre de que tiene más de 1000€ en caja) y semáforos para controlar el acceso a los ficheros, de forma que nunca un fichero se modifique por dos procesos simultáneamente.

El uso de semáforos se realizará mediante la biblioteca implementada en el ejercicio anterior.