Sistemas Operativos – Práctica 2

FECHA DE ENTREGA: SEMANA 07-11 MARZO (HORA LÍMITE DE ENTREGA: UNA HORA ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS).

EL GRUPO DEL JUEVES (2211) Y LOS GRUPOS DEL VIERNES ENTREGA LA SEMANA DEL 14-18 MARZO

EN EL CASO DE LOS GRUPOS DE LOS VIERNES EL EXAMEN DE LA PRÁCTICA 3 SE REALIZARÁ EL VIERNES 18 DE MARZO

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

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
Hilos

Los hilos son unidades de trabajo que se pueden expedir para su ejecución. En general, un proceso puede tener asociados varios hilos, que compartirán entre ellos los recursos que el sistema asigne al proceso, permitiendo desarrollar programación concurrente de diversas acciones dentro del proceso.

A diferencia de los procesos, que son completamente independientes (aunque puedan ejecutar el mismo código fuente), los hilos mantienen una relación de dependencia entre ellos que les da ciertas ventajas frente a los procesos: duplican menos recursos, se crean y se destruyen más rápido y pueden comunicarse de forma más rápida e intuitiva ya que comparten el mismo área de memoria (espacio de direcciones).

Los hilos se crean para ejecutar una función concreta, con su propia pila local (*stack*), donde se crearán las variables automáticas. No obstante, compartirán los recursos asignados por el sistema operativo, como los ficheros abiertos, así como la misma área de memoria dinámica (*heap*) y las mismas variables globales.

Para escribir programas multihilo en C podemos hacer uso de la biblioteca de hilos pthread que implementa el standard POSIX (Portable Operating System Interface). Para ello en nuestro programa debemos incluir la cabecera correspondiente (#include <pthread.h>) y a la hora de compilar es necesario enlazar el programa con la biblioteca de hilos, es decir se debe añadir a la llamada al compilador gcc el parámetro *–lpthread:*

gcc –o programa programa.c –lpthread

Ejercicio 1. (APRENDIZAJE) Estudia las siguientes funciones de la biblioteca (#include <pthread.h>):

- ```
  int pthread_create(pthread_t *thread,
                     const pthread_attr_t *attr,
                     void *(*start_routine) (void *),
                     void *arg);
  ```

- ```
  void pthread_exit(void *retval);
  ```
- ```
  int pthread_join(pthread_t thread, void **retval);
  ```
- ```
  int pthread_detach(pthread_t thread);
  ```
- ```
  int pthread_cancel(pthread_t thread);
  ```
- ```
  pthread_t pthread_self(void);
  ```

Ejercicio 2. (APRENDIZAJE) Estudia qué hace el siguiente programa en C.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

void *slowprintf (void *arg) {
   char *msg;
   int i;
   msg = (char *)arg;

   for ( i = 0 ; i < strlen(msg) ; i++ ) {
      printf(" %c", msg[i]);
      fflush(stdout);
      usleep (1000000) ;
   }
   pthread_exit(NULL);
}

int main(int argc , char *argv[]) {
   pthread_t h1;
   pthread_t h2;
   char *hola = "Hola ";
   char *mundo = "Mundo";

   pthread_create(&h1, NULL , slowprintf , (void *)hola);
   pthread_create(&h2, NULL , slowprintf , (void *)mundo);

   pthread_join(h1,NULL);
   pthread_join(h2,NULL);

   printf("El programa %s termino correctamente \n", argv[0]);
   exit(EXIT_SUCCESS);
}
```

¿Qué hubiera pasado si el proceso no hubiera esperado por los hilos? Para probarlo elimina las dos líneas de la llamada a la función *int pthread_join(pthread_t thread, void **retval).*

Ejercicio 3. (ENTREGABLE) 2ptos Comparando tiempos. Se pide escribir dos programas en C, ***ejercicio3a.c*** y ***ejercicio3b.c***. En el primer programa se calculará el tiempo que se invierte en la creación de 100 procesos hijos, la ejecución de cada proceso hijo y la finalización correcta del programa. Cada proceso hijo debe escribir en pantalla un número aleatorio entre 0 y RAND_MAX. (No olvidar que el proceso padre debe esperar por todos los hijos creados y cada proceso hijo debe devolver al padre el estado de terminación.)

Del mismo modo, el programa ***ejercicio3b.c*** calculará el tiempo que tarda en crear 100 hilos y cada hilo escribirá en pantalla un número aleatorio entre 0 y RAND_MAX.

¿Con cuál de las dos opciones obtienes mejor rendimiento? Razona tu respuesta.

Ejercicio 4. (ENTREGABLE) 2ptos Paso de Parámetros en Hilos. Realizar un programa en C en el que se introducirán 2 números y dos matrices cuadradas desde línea de comandos. El programa generará un hilo de ejecución para multiplicar cada uno de los números por las respectivas matrices. Se deben introducir retardos en los bucles de multiplicación para que se perciba el paralelismo en la ejecución. Véase el siguiente ejemplo:

```
Introduzca multiplicador 1:
3
Introduzca multiplicador 2:
6
Introduzca matriz 1:
3 5 4 6 2 2 1 2 3
Introduzca matriz 2:
3 5 5 3 2 2 2 2 3
Realizando producto:
Hilo 1 multiplicando fila 0 resultado 9 15 12
Hilo 2 multiplicando fila 0 resultado 18 30 30
Hilo 2 multiplicando fila 1 resultado 18 12 12
…
```

¿Cómo se podría compartir información entre hilos? Modificar el programa anterior para que desde cada hilo se sepa por qué fila está el otro hilo.

```
Hilo 1 multiplicando fila 0 resultado 9 15 12 – el Hilo 2 va por la fila 1
```

SEMANA 2

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

¿Qué obtienes?

Ejercicio 6. (ENTREGABLE) 1.5 ptos. Escribe un programa en C, ***ejercicio6.c***, que cree un proceso hijo, el proceso hijo permanentemente realizará los dos siguientes pasos: (1) escribir en pantalla, "Soy el proceso hijo con PID: xxx" y (2) dormir 5 unidades de tiempo, *sleep(5).*

El proceso padre, una vez creado el proceso hijo, esperará 30 unidades de tiempo y a continuación enviará una señal de terminación al proceso hijo y después él mismo terminará.

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 7. (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 8. (ENTREGABLE) 2.5 ptos Escribe un programa en C, ***Ejercicio8.c***, en el que el proceso padre genere 4 procesos hijo en serie (al igual que el Ejercicio 5 de la Práctica1). Una vez tenemos el proceso padre raíz y los cuatro hijos, se establece el siguiente protocolo:

- El último hijo espera 5 segundos y envía SIGUSR1 a su padre
- Cada proceso padre espera a recibir SIGUSR1 de su hijo. Una vez recibida la señal, espera 2 segundos y envía a su padre SIGUSR1

Una vez que el proceso raíz haya recibido SIGUSR1, enviará SIGUSR2 a su hijo. En esta ocasión el paso de testigo se realizará mediante la señal SIGUSR2 según el siguiente criterio:

- Cada proceso hijo espera a que su padre le envíe SIGUSR2
- Cuando un proceso recibe SIGUSR2, espera 1 segundo y envía SIGUSR2 a su hijo
- En el caso del último hijo, éste enviará SIGUSR2 al proceso padre raíz

De esta forma tendremos un paso de testigo circular que se va a repetir tres veces. Tras ello cada proceso padre enviará SIGTERM a su hijo. El protocolo de terminación involucra que cada proceso realice las siguientes órdenes:

1. Cada proceso espera la recepción de SIGTERM
2. Espera un segundo y envía SIGTERM a su hijo
3. Termina llamando a exit

Al igual que en el caso anterior, el último proceso hijo envía SIGTERM al proceso raíz. Éste terminará con exit tras la recepción de la señal.

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

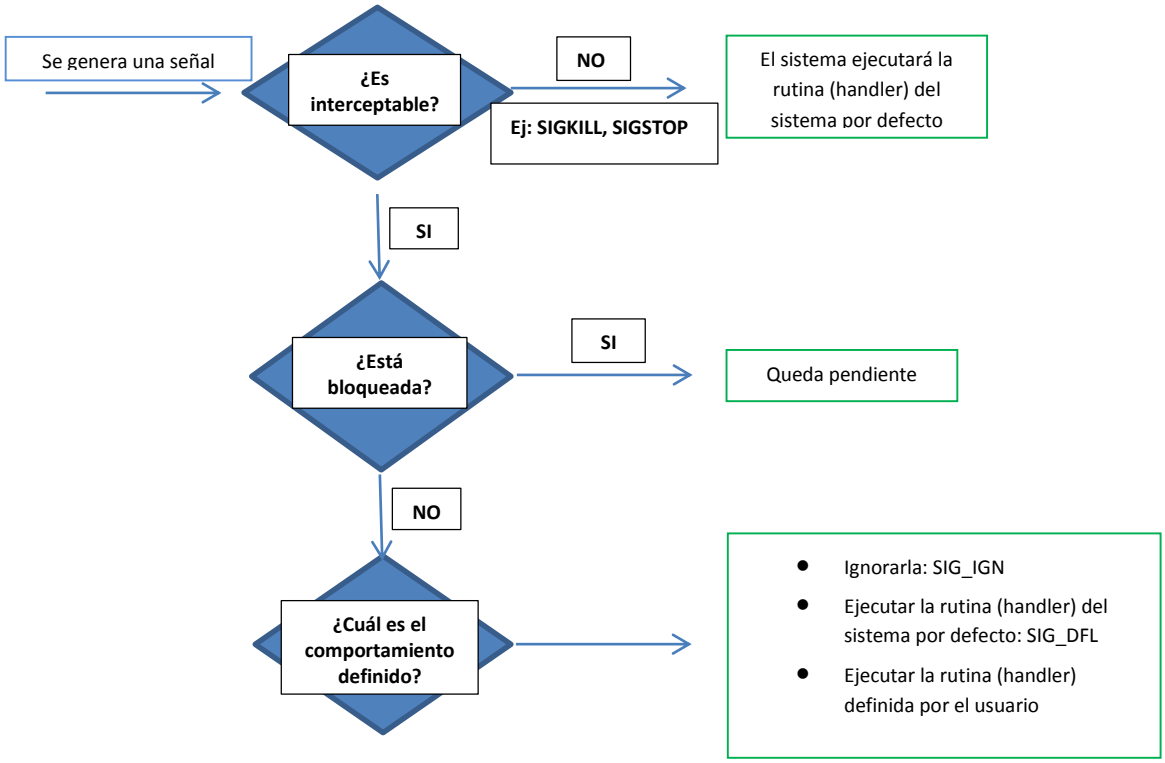

Ilustración 1Recepción y Manejo de Señales

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 9. (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 10. **(ENTREGABLE) 2ptos** Escribe un programa en C, ***ejercicio10.c***, que genere un par de procesos de acuerdo con los siguientes requerimientos. Sean dos procesos, proceso A y proceso B, tales que el proceso B crea al proceso A y verifica que está activo. El proceso A selecciona de modo aleatorio una palabra de la siguiente frase:

EL PROCESO A ESCRIBE EN UN FICHERO HASTA QUE LEE LA CADENA FIN

Tras seleccionar la palabra, el proceso A la escribe en un fichero. Si el proceso A selecciona "FIN", escribe dicha palabra en el fichero y termina.
Por su parte, cada 5 segundos el proceso B lee una palabra del fichero en el que escribe el proceso A. Si el valor que lee el proceso B es igual a "FIN", el proceso B interpreta que el proceso A ha muerto y lo vuelve a generar. Por último, tras la lectura de 50 cadenas el proceso B mata el proceso A y finaliza su ejecución.