Fundamentos de Software de Comunicaciones
E.T.S.I. Telecomunicación

Práctica de Señales

Este guión pretende una primera toma de contacto con el mundo de las comunicaciones entre usuarios, procesos y S.O., estudiando las **señales** en los sistemas operativos de la familia Unix. A lo largo de las siguientes páginas se introducirán conceptos que serán reforzados mediante la realización de pequeños códigos en C.

Como material de consulta se recomienda un repaso a los punteros (en particular los punteros a funciones) y a los comandos del sistema operativo que se introducen a lo largo de este guión.

1. Señales en Unix

Son eventos, alertas software que el sistema operativo envía a un proceso (son similares a las interrupciones, aunque éstas se producen a nivel hardware). Uno de los ejemplos más claros lo constituye el hecho de finalizar el sistema (shutdown). El sistema operativo envía a todos los procesos que corren en ese momento una señal de terminación. Esta notificación permite a los procesos guardar los datos importantes, liberar memoria, notificar por red esta circunstancia o cerrar los descriptores abiertos. El conocimiento de que existen las señales nos preparará para programar unas aplicaciones de comunicaciones muy robustas.

Ante la llegada de una señal, un proceso puede elegir:

- atenderla: indicando una función de atención, denominada *manejador de señal*.
- ignorarla: comunicándolo directamente al S.O. Ojo, porque algunas **NO** se pueden ignorar.

Lo más importante es que la señal es un evento de tipo **ASÍNCRONO**, que hace saltar la secuencia normal de ejecución. No tener esto en cuenta puede llevar a situaciones erróneas, ya que ciertas funciones de librería y llamadas al sistema reaccionan ante las señales, comportándose de manera distinta a la prevista inicialmente (este hecho lo veremos más adelante, cuando avancemos en las comunicaciones mediante la librería de sockets).

2. El modelo de Señal

a) Cada señal tiene asociado un identificador (numérico o simbólico). Ejemplo: la señal KILL, que indica la terminación inmediata y abrupta del proceso que la recibe, es la número 9, y también se emplea como SIGKILL.

b) Se indica un patrón de acción, que corresponde a la respuesta que dará el proceso ante la llegada de la señal. El S.O. da un patrón de acción por defecto para cada señal existente. A su vez, los programas pueden registrar nuevos patrones de acción. Estos patrones son los *manejadores de señal*.

Nombre	Id. Numérico	Acción Por defecto	Descripción
SIGHUP	1	Exit	Hangup (ref termio(7I))
SIGINT	2	Exit	Interrupt (ref termio(7I))
SIGQUIT	3	Core	Quit (ref termio(7I))
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace or breakpoint trap
SIGABRT	6	Core	Abort
SIGKILL	9	Exit	Kill
SIGSEGV	11	Core	Segmentation fault, an address reference boundary error
SIGSYS	12	Core	Bad system call
SIGPIPE	13	Exit	Broken pipe
SIGALRM	14	Exit	Alarm clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User defined signal 1
SIGUSR2	17	Exit	User defined signal 2
SIGCHLD	18	Ignore	Child process status changed
SIGPWR	19	Ignore	Power fail or restart
SIGURG	21	Ignore	Urgent socket condition
SIGPOLL	22	Exit	Pollable event (ref streamio(7I))

SIGSTOP	23	Stop	Stop (cannot be caught or ignored)
SIGTSTP	24	Stop	Stop (job control, e.g., CTRL-z))
SIGVTALRM	28	Exit	Virtual timer expired
SIGPROF	29	Exit	Profiling timer expired

Tabla 1.- Señales, sus valores y significado

Ejercicio 1:

a) Todavía son escasos los conocimientos que tenemos sobre el mundo de las señales. Sin embargo, para ir abriendo boca, la tabla 1 muestra los datos sobre algunas de las señales más importantes. Revisa estas páginas del manual y, utilizando Internet, encuentra significado al contenido de la tercera columna de la tabla:

```
man 1 kill
man 2 kill
man 3 raise
man 2 signal
man 7 signal
```

b) Identifica al menos dos situaciones en las que sería bueno cambiar el comportamiento por defecto de alguna señal en un programa.

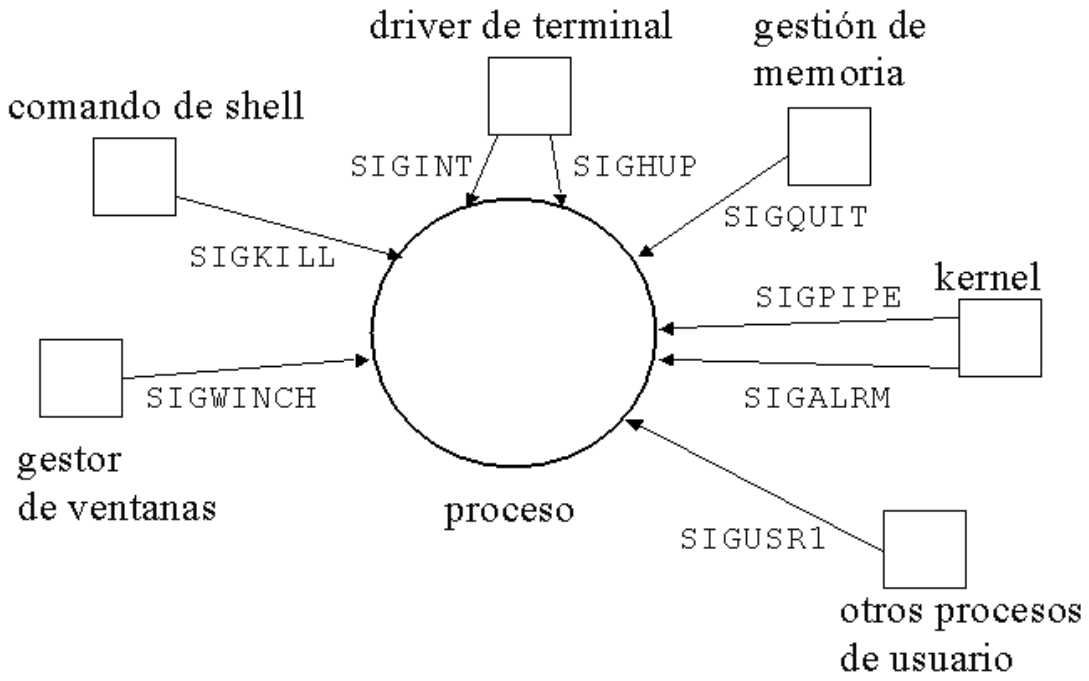

Figura 1.- Algunas posibilidades de recepción de señales

3. Manejando señales desde el terminal Unix

Recordaréis que Unix es un sistema multiusuario y multiproceso. Esto viene a decir que, en el sistema, a la vez, pueden estar trabajando distintas personas y que cada una de ellas puede lanzar distintos procesos al mismo tiempo. En este contexto, el S.O. asigna a cada proceso en ejecución un identificador de proceso (PID).

El comando **`ps`** nos permite visualizar los procesos que corren en el sistema. Mediante distintas opciones, podemos ver tanto los que hemos lanzado nosotros, como los que pertenecen a los demás usuarios y demonios[1].

Cuando llamamos a **`ps`**, la primera columna que aparece en pantalla es el PID del proceso (su nombre se indica en la columna CMD). En Linux, usar en consola:
%> `ps aux`.

Hagamos una prueba. Vamos a crear un nuevo terminal gráfico desde la consola que tenemos abierta. Para que no nos moleste, la lanzamos a un segundo plano (&) y dejará libre de nuevo el terminal mientras se ejecuta:

```
%> xterm &
```

Recordad el número que se os devuelve como respuesta a este comando, pues corresponde al PID asignado a este nuevo proceso.

Vamos a enviarle a este proceso una señal de terminación. Utilizaremos el comando **`kill`**:

```
%> kill –TERM pid
```

¿Qué ha sucedido? ¿Seríais capaces de enviar señales a otros procesos que están disponibles a través de **`ps`**? Comprobad el efecto de algunas señales, tal y como se presentaba en la tabla 1.

TRUCO: Si alguna vez se os olvidan los nombres de señales, podéis recordarlos con `kill –l`. También se pueden usar sus identificadores numéricos (`kill -9 pid` es equivalente a `kill –KILL pid`)

4. Manejando señales desde nuestros programas C

Mediante la inclusión de la librería `signal.h`, tendremos acceso a las funciones y macros que detallaremos a continuación.

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

`signal(sig_id, funcion_manejadora):`

[1] Un demonio es un proceso que se ejecuta permanentemente hasta que la máquina se apaga. Los servidores Web, de Correo, administradores de impresión, son ejemplos de demonios Unix.

Esta llamada al sistema asigna (registra) una función manejadora para una señal. Sus argumentos son un número de señal y un puntero a una función manejadora. Recordemos ante la llegada de la señal, el flujo de ejecución saltará a esta función y, tras su atención, regresará al punto original.

Ejemplo:

```
void manejador(int num_signal){
   /*tratamiento de la senal*/
   write(1,"No!\n",4);
   return;   /*vuelta al programa*/
}

int main(){

   signal(SIGINT,manejador);
   /*aqui poner codigo que se ejecute indefinidamente para que de
tiempo a mandar la señal*/
   while(1);

}
```

En este sencillo ejemplo, hemos creado una función manejadora para la señal INT (Ctrl-C), y la hemos registrado mediante una llamada a `signal()`. Cread un programa con esta plantilla que se quede en bucle infinito, y ejecutadlo. ¿Qué pasa si pulsamos Ctrl-C? (equivalente a kill –INT). ¿Qué pasa si ahora pulsamos otra vez el Ctrl-C? ¡Cuidado! Esta respuesta depende de la versión de Unix en la que se ejecute vuestro programa.
Ejemplo:

- compila en C y ejecuta: gcc –ansi –o exec ejemplosignal.c (comportamiento Unix System 5)
- compila en C y ejecuta: gcc –o exec ejemplosignal.c (comportamiento Unix BSD)

En la semántica de Unix System 5, el sistema operativo realiza una acción cuando atiende una señal: resetea el manejador de la misma a su valor por defecto para la siguiente vez. Sin embargo, la semántica de BSD mantiene el manejador definido por el usuario.

¿Cómo se puede implementar un código único que se comporte de la misma manera para distintos tipos de Unix? Pista: cuando el proceso se ejecute en System 5 hay que reasignar de nuevo el manejador definido por el usuario de forma manual, pero ¿dónde?

Manejadores predefinidos

- Si en lugar de indicar el nombre de una función manejadora en los argumentos a `signal`, utilizamos SIG_IGN, estaremos indicando al S.O. que vamos a ignorar dicha señal.
  Aplica `signal(SIGINT, SIG_IGN)` al código anterior. ¿Qué sucede?

Recuerda: existen señales que NO se pueden ignorar, como SIGKILL o SIGSTOP. Pruébalo.

- Otro manejador predefinido es SIG_DFL. Esto indicará al sistema que asigne el patrón por defecto (manejador), que usa cuando comienza el programa. Se puede interpretar como un "reset" de manejadores.

Reponer un manejador

- La llamada a `signal` retorna el puntero al manejador previo que tenía la señal, esto es **un puntero a función** Guardar este valor puede ser muy útil cuando queremos restaurar el comportamiento previo tras la atención, pero... ¿de qué tipo es este valor?

Aunque los punteros a funciones son siempre complicados de entender, veamos cómo es el tipo de función a la que apunta el retorno de signal en Unix:

```
void (*signal(int signo, void(*manejador)(int)))(int);
```

Parece difícil de interpretar. Es mejor utilizar un typedef en el código:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signo, sighandler_t manejador);
```

O lo que es lo mismo, `signal` devuelve un puntero a una función de tipo

void manejador(int)

Si se produjo un error de algún tipo, `signal` devolverá SIG_ERR.

Resumen

Ante la llegada de una señal:

1) Si el manejador es SIG_IGN, la señal se ignora
2) Si el manejador es SIG_DFL, se ejecuta la acción por defecto para la señal.
3) Si el manejador es una función registrada por el usuario, a) el manejador se resetea a SIG_DFL (comportamiento System 5), o b) el manejador se mantiene con el valor registrado por el usuario (comportamiento BSD). A continuación, ¡y sólo después!, la función manejadora de usuario es llamada con el argumento *signo*.

Ejercicio 2:

En el código de más abajo, hay con dos manejadores de señal para SIGINT (Ctrl-C). Su misión es escribir un texto en pantalla. En el código del programa, registra sólo el primer manejador. El registro del segundo se hará dentro del cuerpo del primer manejador, guardando como variable global el resultado que devuelva `signal` (un puntero a función). En el momento en que este segundo manejador se ejecute, restaura el valor del manejador previo que guardaste en la variable global. Provoca SIGINT una vez más. ¿Has conseguido que se ejecute de nuevo el primer manejador? Razónalo.

NOTA: En algunos Unix, si muchas señales del mismo tipo esperan ser atendidas, es posible que solo se reparta una y las demás se descarten.

```c
/*Compila con -ansi y sin este flag y observa el comportamiento de los procesos!
NOTA: cuando se compila en ansi, los comentarios // no los acepta,
hay que ponerlos con barra-asterisco-asterisco-barra
*/

#include <signal.h>
#include <sys/time.h>

/*Definicion de tipo puntero a funcion que devuelve void y recibe
como argumento un entero*/
typedef void (*sighandler_t)(int);
/*Defino una variable global de ese tipo*/
sighandler_t puntero_a_funcion;

void manejador2(int s); /*prototipo para que no se queje el compilador*/


void manejador1(int s){
  write(1,"manejador1\n",11);
  /*2. Registra el manejador2 para que atienda a SIGINT (Ctrl-C),
  guarda en la variable puntero_a_funcion lo que devuelve signal */
}

void manejador2(int s){
    write(1,"manejador2\n",11);
    /*3. Registra puntero_a_funcion para que atienda a SIGINT (Ctrl-C)*/
}

int main(){
    /*1. Registra el manejador1 para que atienda a SIGINT (Ctrl-C)*/
    while(1); /*quiza sea mas solidario poner un pause() dentro del cuerpo del while!*/
    return 0;
}
```

Seguimos con las funciones de la librería signal.h:

`kill(pid, signal_id):`

Al igual que el comando de consola, nos sirve para mandar señales a procesos desde un programa C. Usadlo con `kill(pid,signal)`

`raise(signal_id)`

Manda señales a nuestro programa, y es equivalente a `kill(getpid(),signal)`. Por ejemplo: `raise(signal)`

`pause()`

Muy útil para estas prácticas, pues detiene el programa hasta que se recibe una señal. Tras atender la señal, se retoma la ejecución normal del programa. Está en `<unistd.h>`. Ojo, ¡ahora podemos quitar los bucles infinitos en nuestras pruebas y sustituirlos por esta llamada!

5. Generación de Alarmas

Una señal que podemos enviarnos a nosotros mismos es SIGALRM. Lo interesante de esta señal es que existe una llamada al sistema `alarm()` que la activa en un periodo indicado en segundos. Podemos usarla, por ejemplo, para que salte a los 3 segundos:

```
alarm(3);
```

El argumento que acepta es de tipo `long int`, así que no admite decimales.

Ejercicio 3:

Implementa un programa inmune al Ctrl-C durante los segundos que se introduzcan como argumento. Transcurridos éstos, se volverá a habilitar la posibilidad de que se cierre el programa con Ctrl-C.

6. Durmiendo un proceso

Existen dos funciones de librería (que no llamadas al sistema), para dormir un proceso:

`sleep`[2]`(segundos)` trabaja a una precisión de segundos
`usleep(microsegundos)` trabaja a una precisión de microsegundos

Cuando un proceso duerme, el sistema operativo aprovecha para ejecutar otros procesos que estuviesen en espera de CPU. Tras la espera obligada, el programa proseguirá su ejecución normal.

7. Temporizadores

Un paso en complejidad lo constituye el uso de temporizadores básicos en nuestros programas. En Unix hay tres formas de contar el tiempo:

- ITIMER_REAL: cuenta el tiempo real (enviando SIGALRM al proceso)
- ITIMER_VIRTUAL: cuenta el tiempo sólo cuando el proceso está activo (envía SIGVTALRM al proceso)
- ITIMER_PROF: cuenta el tiempo en que la CPU trabaja para el proceso, esté activo o atendiendo el kernel a una llamada al sistema. (enviando SIGPROF al proceso).

Es decir, existen tres tipos de temporización. En el momento de declarar un temporizador, se inicia la cuenta atrás hasta llegar a cero, momento en el que el S.O. lanza una de las señales indicadas arriba. Para ajustar el tiempo hasta que salte la señal correspondiente, utilizamos:

[2] `sleep()` y `usleep()` se llevan mal con `alarm()` ya que, habitualmente, las dos primeras se construyen usando SIGALRM, por lo que podría interferir en lo que esperáis al utilizar la llamada a `alarm()`. Evitad usarlas a la vez en vuestros programas.

```
#include <sys/time.h>

void manejador(int s){
     /*hacer lo que sea*/
}

int main(){
  struct itimerval timer;
  struct timeval valor;

  valor.tv_sec = 2; /*dos segundos*/
  valor.tv_usec = 2000; /*dos milisegundos*/

  timer.it_value = valor;
  timer.it_interval = valor;  /* Consultar en   el manual sobre cómo
declarar y asignar valores al campo it_interval */

  signal(SIGALRM,manejador);
  setitimer(ITIMER_REAL, &timer, NULL);

  while(1); /*bucle infinito de espera de senales*/
  return 0;
}
```

Ejercicio 4:

a) Escribe un programa que, periódicamente, (3,3 segundos de CPU) imprima un "." en la pantalla. Protégelo contra Ctrl-C.
b) Haz lo mismo, pero sin escribir el punto en el manejador del programa, sino en el main.
(NOTA: para pararlo, usa Ctrl-Z y luego mátalo por línea de comandos)

8. Problemas con las señales

Desafortunadamente, la aparición de señales en los programas junto a su tratamiento, tiene ciertas consecuencias que hay que tratar si queremos que nuestros programas sean muy robustos. A continuación se enumeran tres problemas fundamentales, que se solucionaron más adelante con la introducción de unas normas de obligado cumplimiento para todos los Unix (las normas POSIX, fuera del alcance de esta asignatura).

8.1 El problema del RESET del manejador

Si os fijasteis bien, antes de llamar a la función manejadora, la mayoría de sistemas Unix familia System 5 suelen resetear a SIG_DFL inmediatamente después de que la señal ha sido recibida. Esto puede presentar un serio problema si queremos atender a la señal permanentemente, por lo que se debe volver a llamar a signal() de nuevo para reinstalar el manejador. Ejemplo:

```
void manejador(int num_signal){
   /*tratamiento de la senal*/
   write(1,"No!\n",4);
   signal(SIGINT, manejador);  /*REINSTALADO!!*/
}
```

Pero, ¿qué sucede si la señal aparece de nuevo justo antes de que nos haya dado tiempo a reinstalar el manejador? ¡Esto es una condición de carrera (*race condition)* típica, que provoca resultados inesperados! En este caso, el programa terminaría. Afortunadamente, en las nuevas implementaciones de Unix, si una señal aparece mientras se está ejecutando su manejador previo, la señal se bloquea (no se le da al proceso) hasta salir del manejador.

8.2 El problema de la atomicidad en la ejecución de llamadas al sistema

Supongamos que estamos en medio de una operación que está gestionándonos una llamada al sistema. Por ejemplo, estamos leyendo con `read()` unos datos de un fichero o de la red (¡eso lo veremos más tarde!). Imaginemos que, en ese momento, una señal llega y es atendida. ¿Qué sucede cuando se retorna del manejador?

Ejercicio 5:

Implementa un programa en C (y compílalo con gcc –ansi), donde se utilice la función read(), que es bloqueante, para recoger desde teclado (descriptor número 0) un conjunto de caracteres hacia un cadena y luego imprimirlos por pantalla. Observa el efecto de la llegada de una señal durante esta operación de lectura de teclado, por ejemplo una alarma (la función read devuelve el número de caracteres leídos o un valor negativo en caso de error. ¿Es este error de tipo EINTR?).

Vuelve ahora a repetir la compilación con gcc sin la opción –ansi (o con g++). ¿Qué sucede ahora?

En principio, si una llamada al sistema ha sido interrumpida por una señal, el programador podría decidir volver a realizar dicha llamada. En el ejemplo anterior, si se detecta que la llamada a read() devuelve -1 y el tipo de error es EINTR se puede volver a llamar a read(). Vuelve a implementar el programa del ejercicio para que realice este nuevo comportamiento.

¿Qué esperaríamos para quedarnos más tranquilos?:
a) Que las llamadas al sistema "críticas" se sigan ejecutando después de atender a la señal
b) Que se posponga la atención de la señal hasta que la llamada sea atendida.

Malas noticias. De nuevo caemos en problemas de compatibilidad entre familias de Unix.

8.3 El problema de qué puede hacerse en un manejador (operaciones reentrantes)

Este es un problema mucho más sutil. En primer lugar, se deberían realizar pocas operaciones en un manejador, de manera que su ejecución terminase rápidamente. En segundo lugar, ciertas llamadas al sistema usadas dentro de un manejador tendrán resultados inesperados: las llamadas al sistema **no reentrantes**.

Una función es reentrante cuando un proceso puede contener llamadas a ella al mismo tiempo sin que se corrompa su funcionamiento, por ejemplo: `read(), write(), fork()`…

Una función es no reentrante en el momento en que hace uso de variables globales, o manipula estructuras como el heap ¡como `malloc(), free(), etc`! Tampoco son reentrantes las funciones contenidas en la librería estándar de E/S: `scanf(), printf()…` Si se utiliza una función no entrante en un manejador, es posible que corrompa información de la misma función si esta se estaba ejecutando y fue interrumpida por la llegada de una señal. ¡Hay que evitarlo!

Por último, la variable global `errno` es muy sensible a cambios por culpa de llamadas al sistema que se hagan dentro del manejador. Antes de hacer la llamada en el manejador se tiene que guardar el valor que tuviera `errno`, y luego restaurar a `errno` dicho valor antes de salir del manejador.