Prácticas de Sistemas Operativos

Toñi Reina y David Ruiz

Boletín ♯5: Señales

Curso 2003/04

Índice

---

1. Introducción

Una señal es una interrupción *software* que permite la comunicación entre procesos, de hecho es muy frecuente que el núcleo envíe señales a los procesos durante su ejecución. Un proceso puede enviarle una señal a otro y éste, al recibirla, puede comportarse de tres formas diferentes:

- *Ignorar la señal*.
- *Invocar a una rutina de tratamiento por defecto*. Esta rutina la aporta el núcleo y, según el tipo de señal, esta rutina por defecto realizará una acción u otra. Generalmente, provoca la finalización del proceso. En algunos casos, provoca la creación de un fichero *core*, que contiene el contexto del proceso antes de recibir la señal. Estos ficheros se pueden examinar con la ayuda de un depurador.
- *Invocar una rutina específica realizada por el programador*. No todas las señales permiten este tipo de atención.

Cada señal tiene asociado un número entero positivo, que es el que intercambian los procesos cuando uno le envía una señal a otro. En UNIX System V hay definidas 19 señales, y en 4.3BSD, 30. Las 19 señales del UNIX System V las tienen prácticamente todas las versiones de UNIX, y a éstas los fabricantes les añaden las que creen convenientes. En general, las señales se pueden clasificar en los siguiente grupos:

- Señales relacionadas con terminación de procesos.
- Señales relacionadas con las excepciones inducidas por los procesos. Por ejemplo, los errores producidos al manejar números en coma flotante.
- Señales relacionadas con los errores irrecuperables originados en el transcurso de una llamada al sistema.
- Señales originadas desde un proceso que se está ejecutando en modo usuario.
- Señales relacionadas con la interacción con el terminal.
- Señales para ejecutar un proceso paso a paso. Éstas son usadas por los depuradores.

En el fichero de cabecera <`signal.h`> están definidas las señales que puede manejar el sistema y sus nombres. En la tabla 1 se pueden ver el conjunto de señales definidas en System V, junto con su significado y su acción por defecto.

2. El comando kill

Para generar una señal desde el intérprete de comandos se utiliza el comando *kill*. Su sintaxis es la siguiente:

```
kill -s señal pid...
kill -l [exit_status]
kill [-señal] pid...
```

-s señal Indica la señal a enviar. Esta señal se puede especificar con un número o con un nombre de señal.

pid Especifica la lista de procesos a los que se les enviará la señal. Si:

`pid>0` La señal se le envía al proceso cuyo identificador es el dado por `pid`.

`pid=0` La señal se manda a todos los procesos del mismo grupo que el actual.

`pid=-1` La señal es enviada a todos los procesos cuyo `pid` sea mayor que uno.

`pid<-1` La señal se envía a todos los procesos del grupo indicado por `pid`.

-l Imprime una lista de los nombres de señales que hay en el sistema.

-señal Esta opción es equivalente a la opción `-s`.

Núm.	Señal	Descripción	Acción por defecto
1	SIGHUP	Colgar. Generada al desconectar el terminal.	Terminar.
2	SIGINT	Interrupción. Generada por teclado cuando se pulsa la tecla de interrupción.	Terminar.
3	SIGQUIT	Salir. Generada por teclado cuando se pulsa la tecla de salida (Control + \).	Generar core y terminar.
4	SIGILL	Instrucción ilegal. No se puede recapturar.	Generar core y terminar.
5	SIGTRAP	*Trace trap*. Trazado. No se puede recapturar.	Generar core y terminar.
6	SIGIOT	*I/O trap instruction*. Generada cuando se da un fallo de hardware.	Generar core y terminar.
7	SIGEMT	*Emulator trap instruction*. Indica un fallo de hardware.	Generar core y terminar.
8	SIGFPE	Excepción aritmética, de coma flotante o división por cero.	Generar core y terminar.
9	SIGKILL	Terminación abrupta. No puede capturarse, ni ignorarse.	Generar core y terminar.
10	SIGBUS	Error de bus. Se produce cuando se da un error de acceso a memoria.	Generar core y terminar
11	SIGSEGV	Violación de segmentación. Es enviada a un proceso cuando intenta acceder a datos que se encuentran fuera de su segmento de datos.	Generar core y terminar
12	SIGSYS	Argumento erróneo en una llamada al sistema. No se usa.	Generar core y terminar.
13	SIGPIPE	Escritura en una tubería que otro proceso no lee.	Terminar.
14	SIGALRM	Alarma de reloj. Enviada a un proceso cuando alguno de sus temporizadores descendientes llega a cero.	Terminar.
15	SIGTERM	Finalización controlada. Se utiliza para indicar a un proceso que debe terminar su ejecución. Puede ser ignorada.	Terminar.
16	SIGUSR1	Señal número 1 de usuario. Se reserva para uso del programador.	Terminar
17	SIGUSR2	Señal número 2 de usuario.	Terminar.
18	SIGCHLD	Terminación del proceso hijo. Es enviada al proceso padre cuando alguno de los hijos termina.	Es ignorada.
19	SIGPWR	Fallo de alimentación.	Es ignorada.

Figura 1: Señales del UNIX System V

Ejemplo 1 *El siguiente comando envía la señal* `SIGUSR1` *al proceso cuyo identificador es* `3423`.

```
kill -USR1 3423
```

Ejemplo 2 *El siguiente comando da una lista de los nombres de señal simbólicos del sistema*

```
$ kill -l
EXIT HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV
SYS PIPE ALRM TERM USR1 USR2 CLD PWR WINCH URG POLL STOP TSTP CONT
TTIN TTOU VTALRM PROF XCPU XFSZ WAITING LWP FREEZE THAW CANCEL
LOST RTMIN RTMIN+1 RTMIN+2 RTMIN+3 RTMAX-3 RTMAX-2 RTMAX-1 RTMAX
```

3. La llamada kill

La llamada *kill* sirve para enviar una señal a un proceso:

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

int kill(int pid, int sig);
```

sig es la señal que se quiere enviar. Si `sig` vale 0 (señal nula) se realiza una comprobación de errores, pero no se envía ninguna señal. Esta opción se suele utilizar para comprobar la validez del identificador `pid`.

pid es el ID del conjunto de procesos a los que se quiere enviar la señal. Los distintos valores que puede tomar son los siguientes:

`pid>0` Le estamos indicando el pid del proceso al que le queremos enviar 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 los 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`.

devuelve En caso de error devuelve -1 y en errno estará el código del error producido. En caso contrario, devuelve el valor 0.

Ejemplo 3 *El siguiente programa envía la señal SIGSTOP a otro proceso cuyo ID recibe como parámetro.*

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>

int main(int argc, char * argv[]) {
  long pid = 0;

  if (argc != 2)
  {
    printf("usage: ejemplo_kill pid\n");
    exit(1);
  }
  sscanf(argv[1],"%ld",&pid);
  if (kill(pid, SIGSTOP) == -1)
  {
    printf("Error al enviar la señal SIGSTOP al proceso %ld\n",pid);
    exit(1);
  }
}
```

4. La llamada raise

La llamada `raise` sirve para que un proceso pueda enviarse una señal a si mismo, esta función sólo necesita como parámetro el número de la señal:

```
#include <signal.h>

int raise(int sig);
```

sig Es el código de la señal a enviar

devuelve En caso de error devuelve -1 y en `errno` estará el código del error producido. En caso contrario, devuelve el valor 0.

Ejemplo 4 *La llamada* `raise` *se puede implementar a partir de* `kill` *de la siguiente forma:*

```
int raise (int sig)
{
    return kill(getpid(), sig);
}
```

5. La llamada alarm

La llamada `alarm` sirve para enviar una señal `SIGALRM` al proceso que la llama después de que pasen un número determinado de segundos:

```
#include <stdio.h>

unsigned int alarm (unsigned int seconds);
```

Las llamadas a `alarm` no se apilan, de modo que si el programa llama a `alarm` antes de que el tiempo expire, la alarma será inicializada con un nuevo valor.

seconds Número de segundos a esperar hasta que se envíe la señal `SIGALARM` al proceso invocante. Si `seconds` vale cero, se cancela la solicitud previa de alarma.

devuelve Devuelve el número de segundos que quedan en la *alarm* antes de la llamada que reinicializará el valor.

Ejemplo 5 *Ya que la acción por defecto de la señal* `SIGALARM` *es terminar el proceso, el siguiente trozo de código se ejecuta aproximadamente durante 10 segundos.*

```
#include <unistd.h>

void main(void)
{
  alarm(10);
  for ( ; ; ){ }
}
```

6. Tratamiento de señales. signal

Para indicar el tratamiento que debe realizar un proceso al recibir una señal se utiliza la llamada signal. Su declaración es la siguiente:

```
#include <signal.h>

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

`signal` es una llamada al sistema del tipo *"función que devuelve un puntero a una función* `void` *y recibe dos parámetros"*.

sig Es el número de la señal sobre la que queremos especificar la forma de tratamiento.

action Es la acción que queremos que se inicie cuando reciba la señal. `action` puede tomar tres tipos 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). Como se ha visto antes, esta acción suele ser terminar el proceso, y, en algunos casos, generar un fichero *core*.

SIG_IGN Indica que la señal se debe ignorar.

dirección Es la dirección de la función que tratará la señal (manejador creado por el usuario). La declaración de este manejador debe ajustarse al siguiente prototipo:

```
#include <signal.h>

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

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

sig Es el número de la señal.

code Contiene información sobre el estado del hardware en el momento de invocar al manejador. Es opcional.

scp Contiene información de contexto. Es opcional. Tanto `scp` como `code` dependen del *hardware* de la máquina.

devuelve Devuelve el valor que tenía `action`, que puede servir para restaurarlo en un momento posterior. Si se produce algún error, `signal` devuelve `SIG_ERR` y almacena en `errno` el código del error producido.

Ejemplo 6 *El siguiente programa trata la señal* `SIGINT` *generada al pulsar las teclas* `Control + C`.

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

void manejador_sigint(int);

int main(void) {

  if (signal (SIGINT, manejador_sigint) == SIG_ERR)
     {
     perror("Error en signal");
     exit (-1);
     }
  while (1)
     {
     printf ("Esperando un Ctrl-C\n");
     sleep(999);
     }
}

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

Al ejecutar este programa, la primera vez que se pulsa `Ctrl-C` *se puede ver un mensaje impreso por pantalla, pero la segunda vez que se ejecuta, el proceso termina. Esto se debe a que cuando el núcleo llama a la rutina de tratamiento de la señal, se restaura la nueva rutina de tratamiento por defecto como rutina manejadora. ¿Cómo podríamos solventar esto? La solución propuesta es volver a llamar a* `signal` *dentro del manejador de la señal del siguiente modo:*

```
void manejador_sigint(int sig) {
static cnt= 0;

  printf ("Señal número %d recibida.\n", sig);
  if (cnt <20)
    printf ("Contador = %d\n", cnt++);
  else
    exit(0);

  if (signal (SIGINT, manejador_sigint) == SIG_ERR)
     {
     perror("Error en signal");
     exit (-1);
     }
}
```

7. La espera de señales. pause

La llamada a `pause` se utiliza para suspender la ejecución de un proceso hasta que ocurra algún evento exterior a él. Su declaración es la siguiente:

```
#include <unistd.h>

int pause(void);
```

devuelve `pause` hace que el proceso se quede esperando la llegada de alguna señal. Cuando esto ocurre, y tras ejecutar la rutina de tratamiento de la señal, devuelve -1 y en `errno` coloca el valor `EINTR`, para indicar que se ha producido una interrupción de la llamada. El programa continúa con la sentencia siguiente a `pause`.

Ejemplo 7 *El siguiente programa es un ejemplo de la utilización de* `pause`.

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

void manejador_sigusr1(int);
void manejador_sigterm(int);
int main(void)
{
   if (signal(SIGTERM, manejador_sigusr1)== SIG_ERR ||
       signal(SIGUSR1, manejador_sigterm)== SIG_ERR){
      perror("Manejador no asignado");
      exit(1);
      }

   while (1)
      {
      pause();
      }
}

void manejador_sigterm(int sig)
{
   printf("El usuario pide terminar %ld\n", (long)getpid());
   exit (-1);
}

void manejador_sigusr1(int sig)
{

   signal(sig, SIG_IGN);
   printf ("%d\n", rand());
   signal (sig, manejador_sigusr1);
}
```

8. Ejercicios

1. Realizar un programa semejante al comando `kill` para enviarle señales a un proceso desde la línea de comandos. La forma de invocar a este programa será la siguiente:

```
$enviar señal pid
```

2. Realizar un programa llamado `manejadores` que cree un manejador por cada una de las señales de la tabla 2.

Señal	Manejador
SIGINT	manejador_sigint
SIGILL	manejador_sigill
SIGTERM	manejador_sigterm
SIGUSR1	manejador_sigusr
SIGUSR2	manejador_sigusr

Figura 2: Señales y manejadores a implementar (Ejercicio 2)

El programa deberá quedarse esperando la recepción de señales. Cuando reciba una señal de las anteriores, debe imprimir un mensaje por pantalla con el siguiente formato:

```
NOMBRE_SEÑAL        Proceso pid: Señal NOMBRE_SEÑAL recibida.
```

Por ejemplo,

```
SIGINT        Proceso 2323: Señal SIGINT recibida.
```

Sugerencia Para probar el programa puede utilizar órdenes como las siguientes:

```
$ manejadores &
[1] 325
$ kill -16 325
Proceso 325: Señal SIGUSR1 recibida.
$ kill -15 325
Proceso 325: Señal SIGTERM recibida.
```

3. Las señales se pueden considerar como una forma de sincronización de procesos. Realizar un programa que comunique dos procesos a través de un archivo. El problema a resolver es conseguir que un proceso escriba en un archivo y el otro lea de él. Realizar el ejercicio creando el proceso emisor y el receptor con la llamada a `fork()`. Para que el proceso receptor pueda leer algo del archivo, previamente el proceso emisor ha tenido que escribir algo en él antes. El proceso emisor le indicará al proceso receptor que hay algo disponible mediante el envío de una señal. El receptor le devolverá otra señal al emisor, una vez que ha ejecutado la operación de lectura, para indicarle que está listo para recibir más datos. Para entender mejor este proceso, puede ayudarse del organigrama presentado en la Figura 3.

4. ***(Ejercicio de Exámen 2ª CONV ITI 2002-03)***

   Realice un programa `mondir` que cada `<n>` segundos liste los archivos que se encuentran en el directorio `<dir>` junto con su tamaño. La invocación al programa es:

   ```
   $ mondir <n> <dir>
   ```

   Un ejemplo de ejecución del programa es el que aparece en la Figura 4.

   Para hacerlo, se deberán crear dos procesos. El proceso padre deberá esperar `<n>` segundos, transcurridos los cuales, enviará una señal (`SIGUSR1`) al proceso hijo, el cual, se dará por aludido y mostrará por pantalla el listado. Cuando el hijo termine de realizar su tarea, le enviará al padre una señal (`SIGUSR1`) para indicarle que puede comenzar de nuevo el ciclo.

   NOTAS:

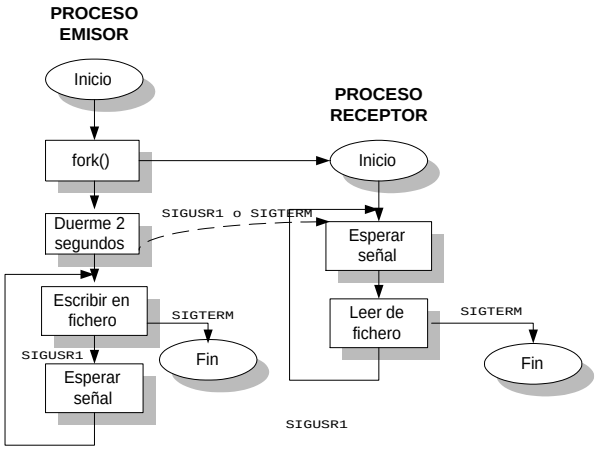

Figura 3: Sincronización del proceso emisor y el receptor mediante señales (Ejercicio 3)

```
$ mondir 5 $HOME/src
Listando directorio: /export/home/cursos/so/src
Entrada: ficheros.tar   Tam: 174592
Entrada: last.txt       Tam: 1482
Entrada: eje.c          Tam: 2082
Entrada: eje.o          Tam: 2828
Entrada: eje            Tam: 7612
```

Figura 4: Ejemplo de ejecución del programa a implementar(Ejercicio 4).

- No se puede utilizar la llamada al sistema `system()`.
- Cuando se produzca algún error, deberán terminar su ejecución los dos procesos.

5. ***(Ejercicio de Exámen 1ª CONV II 2002-03)***

   Escriba un programa en C, que cree un proceso hijo. Cada `n` segundos el padre le manda una señal `SIGUSR1` al proceso hijo, el cual, cuando reciba la señal, deberá ejecutar el programa que se pasa por la línea de comandos

   ```
   $ ejecutar <segundos> <comando>
   ```

   Una invocación posible sería:

   ```
   murillo:/tmp> ejecutar 5 ls -la
   ```

   NOTAS:

   - No se permite la utilización de la llamada al sistema `system()` para resolver el ejercicio.
   - Dispone de una variable global, `char *argumentos[]` que contendrá una copia del `argv` que se recoge en la función `main()`, y que podrá utilizar en la función `manejador_SIGUSR1`.

6. ***(Ejercicio de Exámen 2ª CONV II 2002-03)***

   Se desean sincronizar dos procesos en un sistema que solamente dispone de tuberías como mecanismo de comunicación/sincronización. Se pide realizar un programa `catsincro <seg> <archivo>` que cree dos procesos. El proceso padre, esperará `<seg>` segundos, transcurridos los cuales avisará al proceso hijo. El hijo, al recibir el aviso, imprimirá por la salida estándar byte a byte el contenido de `<archivo>`. Cuando

acabe de realizar esta tarea, avisará al padre, el cual tendrá que volver a esperar `<seg>` segundos para volver a avisar a su hijo.

NOTAS:

- El archivo debe tratarse a bajo nivel.
- Este ejercicio también podía haber aparecido en el boletín anterior, pero como se trata de hacer una sincronización sin el mecanismo de señales, se ha publicado en este.