3

Señales

CONTENTS

3.1 Definición

El mecanismo de señalización es considerado el mecanismo básico de comunicación entre el sistema operativo y los procesos. Su concepción es similar en funcionamiento y estructura al mecanismo de comunicación a nivel de hardware basado en interrupciones. Las señales al igual de las interrupciones son mecanismos de comunicación orientados a la notificación de eventos más que al paso de datos o estructuras basadas en datos.

Aunque la fuente de las señales puede ser el mismo proceso o otro proceso, la principal fuente de generación de señales es el *kernel* del sistema operativo y lo usa para notificar la ocurrencia de un conjunto definido de eventos. En los sistemas *Linux* modernos este conjunto representa 31 eventos diferentes, cada uno representado con una señal. La mayoría de estas señales fueron incluidas en el estándar original *POSIX.1-1990* y otras fueron incluidas después en el estándar *POSIX.1-2001*.

Como mecanismo de comunicación entre procesos, las señales son limitadas debido a que no permiten el envío de datos, el mecanismo de generación/recepción/atención es lento y puede generar ambigüedades debido a

que existe diferentes implementaciones con comportamientos no estándares antes la llegada de una señal.

3.1.1 Taxonomía de señales

La evolución moderna del mecanismo de señalización considera dos grupos de categorización de las señales. Un primer grupo denominado señales *estándar* que corresponden al modelo básico y primitivo de comunicación del sistema operativo con los procesos y un segundo grupo de señales denominadas señales *tiempo real* que se incorporó al modelo base como una de las extensiones que el estándar definió para dar soporte a los requerimientos de tiempo real de los sistemas operativos. Las diferencias en la representación y funcionamiento de estos grupos se describe a continuación.

3.1.1.1 Señales *estándar*

Las señales se identifican mediante un valor numérico que determina el tipo de la señal. Este valor puede variar dependiendo de la arquitectura en donde este implementado el mecanismo. Así, y con el objeto de que los códigos escritos en una determinada plataforma no pierda su semántica al migrar el código, es decir por portabilidad, el estándar recomienda el uso de un conjunto de macros que representan nombres simbólicos para cada señal en lugar de utilizar la representación numérica.

Estos nombre simbólicos inician con el prefijo *SIG* y una abreviación nemotécnica del evento que representa la señal.

Table 3.1: Listado de las señales *estandar* de Linux.

Nombre	Id	Descipción
SIGHUP	1	Hangup (POSIX)
SIGINT	2	Interrupción de terminal (ANSI)
SIGQUIT	3	Terminal cerrado (POSIX)
SIGILL	4	Instrucción ilegal (ANSI)
SIGTRAP	5	Traza trampa (POSIX)
SIGIOT	6	Trampa IOT (4.2 BSD)
SIGBUS	7	Error de BUS (4.2 BSD)
SIGFPE	8	Excepción de punto flotante (ANSI)
SIGKILL	9	Matar (no puede ser atrapado o ignorado) (POSIX)
SIGUSR1	10	Señal definida por el usuario 1 (POSIX)
SIGSEGV	11	Acceso de segmento de memoria no válido (ANSI)
SIGUSR2	12	Señal definida por el usuario 2 (POSIX)
SIGPIPE	13	Escribir en una tubería sin lector, tubería rota (POSIX)
SIGALRM	14	Despertador (POSIX)
SIGTERM	15	Terminación (ANSI)
SIGSTKFLT	16	Falla de pila
SIGCHLD	17	El proceso hijo se detuvo o salió, cambió (POSIX)
SIGCONT	18	Continúe ejecutando, si está detenido (POSIX)
SIGSTOP	19	Dejar de ejecutar (no se puede capturar ni ignorar) (POSIX)
SIGTSTP	20	Señal de parada terminal (POSIX)
SIGTTIN	21	Proceso en segundo plano que intenta leer, desde TTY (POSIX)
SIGTTOU	22	Proceso en segundo plano que intenta escribir, en TTY (POSIX)
SIGURG	23	Condición urgente en el zócalo (4.2 BSD)
SIGXCPU	24	Límite de CPU excedido (4.2 BSD)
SIGXFSZ	25	Límite de tamaño de archivo excedido (4.2 BSD)
SIGVTALRM	26	Despertador virtual (4.2 BSD)
SIGPROF	27	Despertador de perfil (4.2 BSD)
SIGWINCH	28	Cambio de tamaño de ventana (4.3 BSD, Sun)
SIGIO	29	E / S ahora posible (4.2 BSD)
SIGPWR	30	Reinicio de falla de energía (Sistema V)

3.1.1.2 Señales *en tiempo real*

La introducción del soporte del tiempo real para los sistemas operativos introdujo un conjunto de modificaciones para soportar las aplicaciones con requerimientos especiales. Los requerimientos descritos en *POSIX.4* denominados *Real-time Extensions*, incluyen entre otros, planificación en procesos,

sincronización, administración de memoria y señales de tiempo real. Así, se incluyen un conjunto de 33 señales diferentes numeradas del 32 al 64 y se identifican con el prefijo *SIGRT* seguido de un acrónimo del nombre del evento.

La principal diferencia de la extensión de señales en tiempo real es que en la señales básicas, las señales del mismo tipo no se encolan, no existe un esquema de prioridad entre ellas (salvo *SIGKILL* y *SIGSTOP* que tienen condiciones de manejo diferentes), es decir, no hay forma de priorizar la ocurrencia de eventos importantes o con atención prioritaria dentro del programa. En general, La extensión de señales en tiempo real tiene las siguientes diferencias:

- Señales del mismo tipo se pueden encolar. En contraste, si múltiples instancias de una misma señal estándar son entregado mientras ese tipo de señal está bloqueada, solo una instancia se encola.
- El conjunto de señales pendientes son atendidas en orden de prioridad basadas en el valor numérico de la señal. Esto permite diseñar aplicaciones en las cuales algunos eventos sean más *importantes* que otros y sean atendidos con mayor eficiencia. Si múltiples señales en tiempo real del mismo tipo se generan, éstas se entregan en el orden en el cual fueron enviadas. Si múltiples señales de tiempo real pero de diferente tipo son generadas, estás son entregadas en orden basado en el esquema de prioridad de menor valor numérico a mayor valor numérico (menor valor mayor prioridad). En contraste, las señales estándar de diferente tipo cuando son generadas el orden de entrega no es especificado.
- Las señales en tiempo real no tienen significados predefinidas, el conjunto de señales en tiempo real se puede utilizar para fines definidos por cada aplicación.

como su prioridad esta definida por el valor numérico, el estándar incluye dos macros *SIGRTMIN* y *SIGRTMAX* que indica el rango de las señales de tiempo real de acuerdo a cada implementación de sistema operativo. La reocmendacion es no utilizar el valor numérico en la codificación de los programas, en su lugar utilizar el esquema de referencia *SIGRTMIN+n*.

Cada programa debe asignarle el significado del evento que representa cada señal. Sin embargo, la acción por defecto para señales de este tipo recibidas y que no se haya suministrado un manejador es terminar el proceso de forma *term: anormal I*.

Un listado de las señales soportadas en cada distribución puede observarse con el comando *kill -l*:.

```
$> kill -l
1) SIGHUP       2) SIGINT       3) SIGQUIT
4) SIGILL       5) SIGTRAP      6) SIGABRT
7) SIGBUS       8) SIGFPE       9) SIGKILL
10) SIGUSR1    11) SIGSEGV     12) SIGUSR2
```

```
13) SIGPIPE      14) SIGALRM      15) SIGTERM
16) SIGSTKFLT    17) SIGCHLD      18) SIGCONT
19) SIGSTOP      20) SIGTSTP      21) SIGTTIN
22) SIGTTOU      23) SIGURG       24) SIGXCPU
25) SIGXFSZ      26) SIGVTALRM    27) SIGPROF
28) SIGWINCH     29) SIGIO        30) SIGPWR
31) SIGSYS       34) SIGRTMIN     35) SIGRTMIN+1
36) SIGRTMIN+2   37) SIGRTMIN+3   38) SIGRTMIN+4
39) SIGRTMIN+5   40) SIGRTMIN+6   41) SIGRTMIN+7
42) SIGRTMIN+8   43) SIGRTMIN+9   44) SIGRTMIN+10
45) SIGRTMIN+11  46) SIGRTMIN+12  47) SIGRTMIN+13
48) SIGRTMIN+14  49) SIGRTMIN+15  50) SIGRTMAX-14
51) SIGRTMAX-13  52) SIGRTMAX-12  53) SIGRTMAX-11
54) SIGRTMAX-10  55) SIGRTMAX-9   56) SIGRTMAX-8
57) SIGRTMAX-7   58) SIGRTMAX-6   59) SIGRTMAX-5
60) SIGRTMAX-4   61) SIGRTMAX-3   62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX
```

3.1.2 Recepción de señales

Un proceso recibe una señal cuando ocurre algunos eventos, la fuente de estos eventos puede ser el mismo proceso, un proceso diferente y el Sistema Operativo, por ejemplo:

- Desde el mismo proceso utilizando la función *raise()* de forma directa o algún llamado de forma indirecta, por ejemplo, mediante *abort()*.
- Cuando se envía desde otro proceso como resultado de la ejecución de la instrucción *kill(...)*
- Desde el sistema operativo cuando termina o cambia de estado un proceso hijo se envía *SIGCHLD*.
- Cuando el proceso padre muere o se detecta un bloqueo en el terminal de control, se envía SIGHUP.
- Cuando el usuario interrumpe el programa desde el teclado con *ctrl+c*, se envía SIGINT.
- Cuando el programa se comporta incorrectamente, se entrega uno de SIGILL (ejecución de instrucción ilegal), SIGFPE (excepción de punto flotante), SIGSEGV (violación de segmento).
- Cuando se ejecuta *write* o funciones de envío de datos similares en los que medie una tubería como mecanismo de comunicación y el sistema operativo detecte que no existe proceso para recibir los datos se notifica una *SIGPIPE*.

Cada señal tiene una acción predeterminada asociada a ella. La acción predeterminada para una señal es la acción que realiza cuando recibe una señal. Algunas de las posibles acciones predeterminadas son:

- Ign: Ignorar la señal.
- Term: Terminar el proceso de forma *anormal I*.
- Core: Terminar el proceso de forma *anormal II*. Se genera un archivo *Core dump* que contiene la imagen de memoria del proceso cuando recibió la señal.
- Stop: Detener el proceso.
- Cont: Continuar la ejecución del proceso detenido.

Sin embargo, además de las acciones por defecto, tras la recepción de una señal es posible ejecutar un conjunto de instrucciones definidas por el programador, esto es, un manejador de señal.

3.1.3 Registro de manejadores de señales

Desde la versiones iniciales de Unix se han incluido interfaces para el manejo de señales, sin embargo, las primeras versiones eran pocos confiables, en el sentido que algunas señales se perdían y no eran entregadas a los procesos, presentaban poca flexibilidad en el manejo de éstas como el bloqueo de un conjunto de señales para ejecutar instrucciones en secciones criticas. Sin embargo, hoy existe un modelo confiable y versátil para la manipulación de señales. Estás funcionalidades están estandarizadas desde *POSIX.1* y describe dos interfaces para este objetivo.

La ejecución de un manejador es similar en comportamiento a lo descrito en el manejo de interrupciones del procesador. La llegada de una señal es un evento asíncrono, por lo que durante la ejecución del flujo normal del proceso podría generarse el evento. En el momento en el que la señal es recibida por el proceso, esto es, es entregada, el flujo normal del proceso se detiene, y el *kernel* ejecuta la función asignada como manejador. Al finalizar las instrucciones del manejador de la señal el flujo del proceso es reiniciado en la instrucción siguiente a la ultima ejecutada.

La función más sencilla, debido a que constituyó la primera funcionalidad disponible en los sistemas *UNIX*, para definir manejador de funciones en los sistemas es la función *signal(...)*. Ésta fue introducida en la versión de *Unix System V*, y proporciona una forma obsoleta de interfaz de señales. Las nuevas implementaciones no deberían utilizar dicha semántica sino la definidas bajo la función *sigaction* que es más confiable y robusta. el aspecto más importante a considerar en el uso de *signal* es que esta ha sufrido modificaciones en su comportamiento a través de las diferentes implementaciones de los sistemas

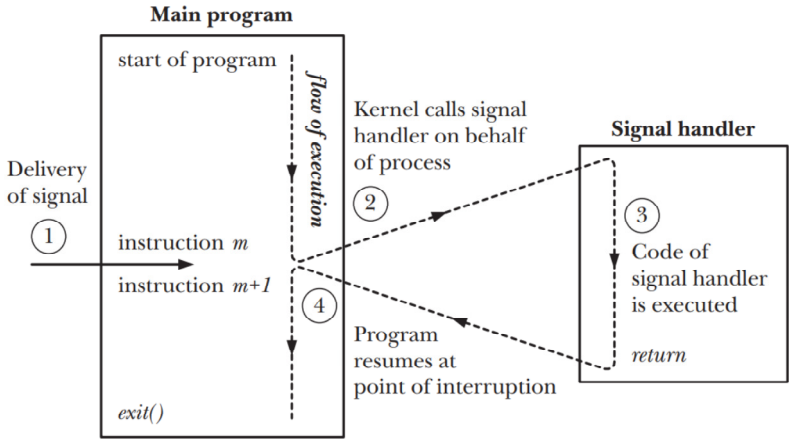

Figure 3.1: Diagrama de la secuencia de ejecución del manejador de la señal.
Modificado de: The Linux Programming Interface

operativos (Unix, Linux), por lo que no se puede considerar con una alternativa para programas que desean ser portables.

Por completitud se describen los dos modelos.

3.1.3.1 la interfaz *signal(...)*

Constituye la función básica para el registro de manejadores de señales, está definida en el archivo de cabecera *signal.h* y permite notificarle al sistema operativo un cambio en la forma en que debe responder el proceso tras la recepción de una señal en particular. El prototipo de la función es:

```
1
2 #include <signal.h>
3 typedef void (*handler_t)(int);
4 sighandler_t signal(int signo, handler_t handler);
```

La función recibe dos parámetros, el primero un entero **signo** que representa el número de la señal sobre la cual se debe modificar el comportamiento de respuesta, por portabilidad en este parámetro se debe hacer uso de las macros definidas en la tabla 3.1 o las macros *SIG_IGN*, *SIG_DFL* para establecer el comportamiento de ignorar o establecer el comportamiento por defecto, respectivamente. El segundo parámetro es un puntero a una función, la que se desea ejecutar tras la recepción de la señal definida en *signo*. Esta función debe tener un prototipo exactamente declarado como *void sighandler(int sig)* , es decir no debe retornar ningún valor y recibe un entero. Este ultimo es cargado por el sistema operativo y representa la señal que disparo la función *sighandler* en tiempo de ejecución. El valor de retorno de la función es una

dirección al manejador anterior en el sistema operativo para la señal *signo*, o el valor representado por la macro *SIG_ERR* en caso de error, estableciendo la variable *errno* en la causa por la que se generó el error.

El programa 3.1 que muestra una forma correcta de modificar el comportamiento de la respuesta a la recepción de la señal *SIGUSR1*. La estructura del programa incluye una sección de modificación y almacenamiento del manejador anterior, una sección durante la cual, en tiempo de ejecución, el programa responderá con la ejecución de la función *sighandler* tras la recepción de una señal *SIGUSR1*. finalmente, una sección para restablecer el manejador anterior que fue modificado.

Programa 3.1: Estableciendo una manejador para SIGUSR1.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4
5  void * oldhandler;
6  void sighandler( int sig ){
7      /*
8
9      */
10 }
11
12 int main(){
13     oldhandler = signal( SIGUSR1, sighandler);
14     if(oldhandler == SIG_ERR){
15         perror("signal:");
16         exit(EXIT_FAILURE);
17     }
18     /*
19     .
20     .
21     .
22     .
23     .
24     */
25     if(signal(SIGUSR1, oldhandler) == SIG_ERR){
26         perror("signal:");
27         exit(EXIT_FAILURE);
28     }
29     return EXIT_SUCCESS;
30 }
```

El bloque de las lineas 19 hasta la linea 23 indica que durante la ejecución de las instrucciones que se encuentren allí el sistema al recibir la señal de

tipo *SIGUSR1* responderá ejecutando el manejar *sighandler*. Como buena practica de programación al finalizar se debe restaurar el manejador con el valor previamente capturado y almacenado en *oldhandler* en la linea 13

En el ejemplo siguiente complementaremos el código anterior con una funcionalidad que permita capturar la señal *SIGINT*. Esta señal se genera cuando se pulsa desde el teclado *ctrl+C* y se envía al proceso que tenga la terminal *foreground*. La acción por defecto es terminar el programa que la recibe, sin embargo, modificaremos este comportamiento de tal forma que el proceso termine al recibir tres señales *SIGINT*.

Programa 3.2: Capturando tres SIGINT *ctrl+c*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4
5  int count = 0;
6  void * oldhandler;
7  void sighandler( int sig ){
8      count ++;
9  }
10
11 int main(){
12     oldhandler = signal( SIGINT, sighandler);
13     if(oldhandler == SIG_ERR){
14         perror("signal:");
15         exit(EXIT_FAILURE);
16     }
17
18     do{
19     }while(count < 3);
20     printf("\n %d veces SIGINT recibida\n", count);
21
22     if(signal(SIGUSR1, oldhandler) == SIG_ERR){
23         perror("signal:");
24         exit(EXIT_FAILURE);
25     }
26     return EXIT_SUCCESS;
27 }
```

para generar este comportamiento se realizaron varios cambios. Inicialmente se declara en la linea 5 una variable de tipo entera que almacenara el numero de veces que de genera la señal. El manejador de la señal ejecuta la instrucción *count ++;* de la linea 8. La linea 12 ahora le indica a la función *signal* que la señal a manejar es *SIGINT*. Luego en el bloque de la linea 18 el programa entra en una espera activa. Durante este periodo de tiempo que el

proceso está en el procesador haciendo saltos, cada vez que se entregue una señal el *kernel* activa el manejador *sighandler* el cual incrementa la variable *count*.

Al ejecutar el programa la salida generada es similar a :

```
$>a.out
^C^C^C
3 veces SIGINT recibida
```

Programa 3.3: Enviando señales a un proceso hijo.

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


void * oldhandler;
void sighandler( int sig ){
    printf("sig %d capturada\n", sig);
}

int main(){
    pid_t pidhijo;
    oldhandler = signal( SIGUSR1, sighandler);
    if(oldhandler == SIG_ERR){
        perror("signal:");
        exit(EXIT_FAILURE);
    }

        pidhijo = fork();
    switch(pidhijo){
    case -1:
          perror("fork");
          exit(EXIT_FAILURE);
    case  0:
          pause();
          printf("[%d]Terminando\n", getpid());
          break;
    default:
          usleep(100);
          kill(pidhijo, SIGUSR1);
          printf("[%d]Senal enviada\n", getpid());
          wait(NULL);
    }
```

```
36
37
38      if(signal(SIGUSR1, oldhandler) == SIG_ERR){
39          perror("signal:");
40          exit(EXIT_FAILURE);
41      }
42      return EXIT_SUCCESS;
43  }
```

3.2 Ejercicios propuestos:

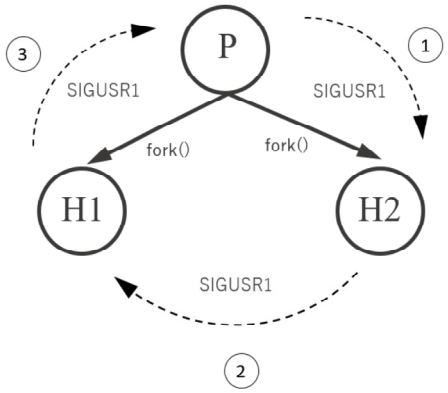

Figure 3.2: Diagrama de la secuencia de comunicación padre-hijos.