Señales

Señal

Una señal es una notificación entregada a un proceso debido a que ha ocurrido un evento asíncrono.

Las señales son como interrupciones de software, es decir interrumpe el flujo normal de ejecución de un programa.

Un proceso puede (si se tiene permisos adecuados) enviar una señal a otro proceso. En este uso, las señales se pueden emplear como:

- Una técnica de sincronización.

- Una forma primitiva de comunicación entre procesos (IPC).

Eventos que hacen que el kernel genere una señal

- **Una excepción de hardware**: detecta un error de software o de hardware, por ejemplo (división por 0).

- **El usuario desde la línea de comandos:** Ejemplo: carácter de interrupción (generalmente Control+C) y el suspender (por lo general Control+Z).

- **Un evento de software:** el proceso hijo termina, expira un timer, etc.

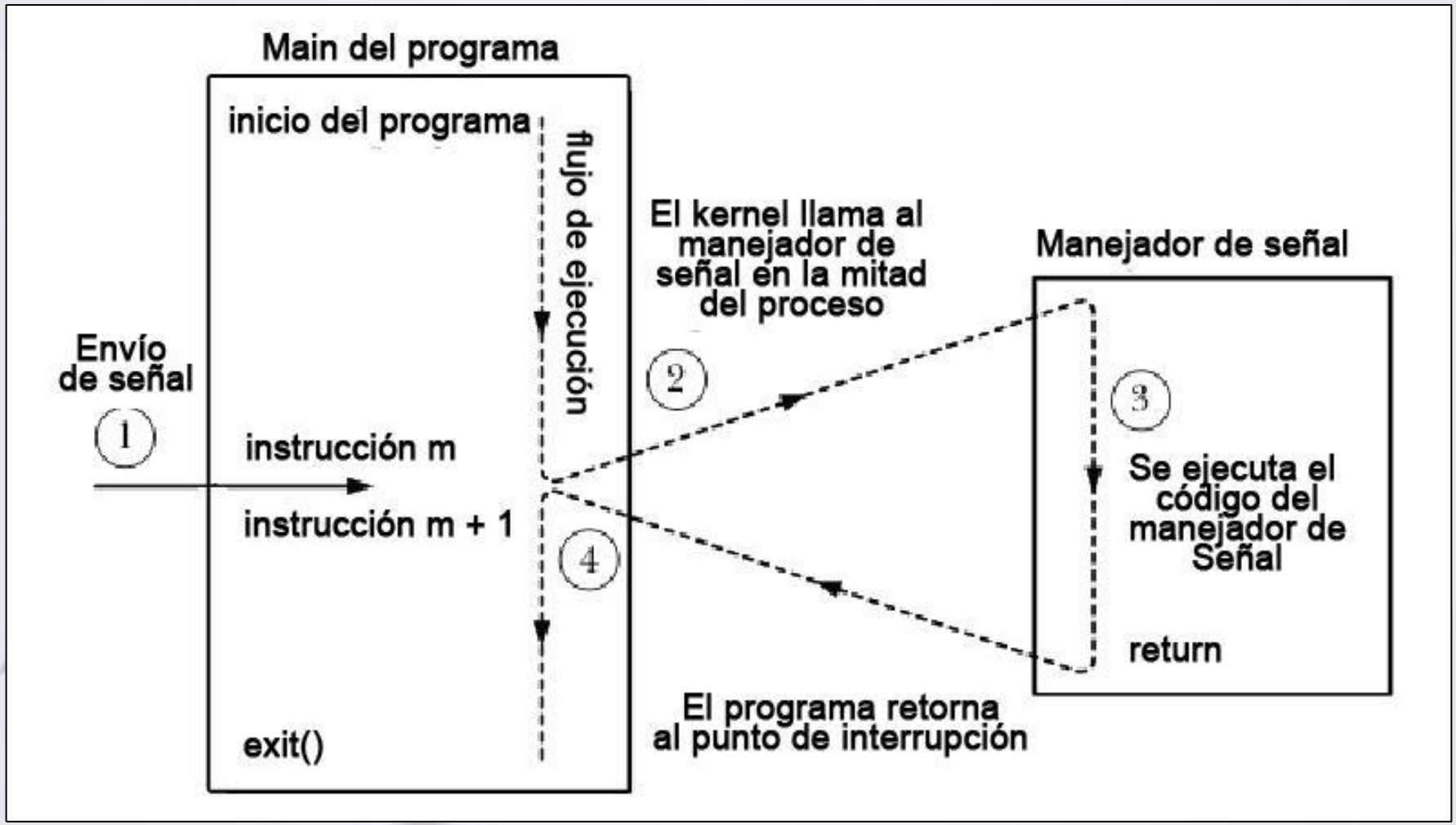
Eventos generados por procesos

- Un proceso puede mandar una señal a otro (si tiene permisos).

Proceso recibe una señal desde consola

Cuando un proceso recibe una señal desde consola, puede suceder que:

- El proceso se termine (`SIGINT`, Control+C).
- El proceso se detenga (`SIGTSTP`, Control+Z).
- El proceso se reanuda (`SIGCONT`, fg 'nombre proceso').
- Nada.


Main del programa
inicio del programa
flujo de ejecución
Envío de señal
1
instrucción m
instrucción m + 1
exit()
El kernel llama al manejador de señal en la mitad del proceso
2
Manejador de señal
3
Se ejecuta el código del manejador de Señal
return
4
El programa retorna al punto de interrupción

Denominación

- Cada señal tiene un nombre del tipo `SIGxxx` con un significado específico.
- Cada señal se define con un número único, empezando de 1.
- Estos números están definidos con nombres simbólicos `SIGxxx` en la biblioteca:

```
<signal.h>
```

- Linux cuenta con 62 señales.
- Para listar las señales en un terminal usamos el comando `kill -l`.

Señales

```
$ 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) SIGSTKF
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+1
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

$ kill -9 11428         # termina un proceso completamente
$ kill -SIGKILL 11428   # lo mismo que lo anterior
```

Comando kill

El comando `kill` sirve para enviar señales a los procesos desde consola.

```
$ kill –SIGXXX PID
```

Ejemplos:

`$ kill –KILL PID` # finalizar la ejecución del proceso PID.

`$ kill -SIGSTOP PID` # detener la ejecución del proceso PID.

`$ kill -SIGUSR1 PID` # enviar una señal tipo usuario al proceso PID.

La acción que ejecute el proceso PID al recibir `SIGUSR1` depende de cómo se haya programado el manejador de la señal `SIGUSR1`.

Formas de tratar una señal

Las señales pueden aparecer en cualquier instante, el proceso debe indicar al kernel qué es lo que tiene que hacer cuando recibe una señal determinada.

El kernel puede actuar de tres formas diferentes:

A. Señales tratadas por defecto (`SIGN_DFL`)

B. Señales ignoradas (`SIG_IGN`)

C. Usar un manejador propio.

Función signal()

La función `signal()` se utiliza para especificar qué se debe hacer cuando se recibe una señal determinada.

```
#include <signal.h>

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

`sig`: señal cuyo manejador se desea designar.
`*handler`: puntero al manejador (puntero a función) que debe ejecutarse al recibir la señal `sig`.

Ejemplo: `signal(SIGINT, manejador)`

En lugar de `(*handler)` se puede especificar:

- `SIG_DFL`: fijar la acción por defecto asociada la señal.
- `SIG_IGN`: ignorar la señal.

A. Señales tratadas por defecto (SIGN_DFL)

Para ver la acción por defecto (default) de cada señal se ejecuta en consola:

```
$ man 7 signal
```

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from **abort**(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from **alarm**(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

B. Señales ignoradas (SIG_IGN)

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

Dentro del `main()`, se define:

```
signal(SIGUSR1,SIG_IGN);   // ignoramos la señal SIGUSR1
```

C. Asociar un manejador a una señal

La función en C que nos permite redefinir la función de tratamiento de una señal es `signal()`.

```
#include <signal.h>        // biblioteca
void       manejador       (int)       {                 .......       };
```

En el main():

```
signal (SIGxxx, manejador);
```

- `SIGxxx`, es un entero definido en `<signal.h>` que identifica a la señal.
- `manejador`, puntero a la función que se ejecuta cuando se recibe la señal indicada.

`signal()` devuelve un puntero a la función de tratamiento que había antes de definir la nuestra (`manejador`). De esta forma podemos guardarla y restaurarla cuando nos interese. Si no se ha podido asociar `manejador` a la señal `SIGxxx`, devuelve un error (SIG_ERR).

Función para enviar una señal a un proceso

Un proceso puede enviar una señal a otro proceso mediante la función `kill()`.

```
include <signal.h>
int kill(pid_t pid, int sig);
```

pid, identificador del proceso.
sig, señal a enviar.

Devuelve 0 si tuvo éxito, o -1 cuando hay error.

Si:

- pid > 0 , señal enviada al proceso con ese pid.
- pid = 0 , se envía la señal a todos los procesos del mismo grupo que el proceso emisor.
- pid < -1 , la señal se envía a todos los procesos en el grupo de procesos cuyo ID es igual al valor absoluto de pid.
- pid = 1 , la señal se envía a todos los procesos a los cuales tiene permiso de enviar (todos menos init y él mismo).

SIGUSR1 y SIGUSR2

- Estas señales están disponibles para los propósitos definidos por el programador.
- El kernel nunca genera estas señales para un proceso.
- Los procesos pueden utilizar estas señales para notificarse mutuamente de los hechos o para sincronizar con otro proceso.

SIGKILL

Esta señal termina la ejecución de un proceso. No puede ser bloqueada, ignorada, y por lo tanto siempre termina un proceso.

```
$ kill -SIGKILL PID
```

SIGSTOP

Esta señal detiene la ejecución de un proceso. No puede ser bloqueada, ignorada, y por lo tanto siempre detiene un proceso.

```
$ kill -SIGSTOP PID
```

SIGCONT

Esta señal reanuda la ejecución de un proceso.

```
$ kill -SIGCONT PID
```

Bibliografía

Kerrisk, Michael. *The linux programming Interface.* 2011. **Capítulos 20.1 a 20.6**

Puntero a función

Veamos otra forma de definir el prototipo de la función `signal()`,

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

`signal()` devuelve un puntero a una función que a su vez devuelve un tipo de dato `void` y recibe un tipo de dato `int`.

Analice el siguiente ejemplo.

```
#include<stdio.h>

void fun(int a){ printf("Valor de a es %d\n", a); }

int main(void)
{
    void (*fun_ptr)(int) = &fun; // fun_ptr es un puntero a fun()

    (*fun_ptr)(10);   // se invoca fun() usando fun_ptr

    return 0; }
```