

Sistemas
Operativos

Señales

Señales (signals)

- Características
- ¿Quién puede enviar señales?
- Comportamientos
- Tipos y llamadas al sistema asociadas
- Ejemplo sencillo señales
- Terminología e Implementación
- Fork y exec *versus* señales
- Ejemplos
- Problemas con las señales
- Señales seguras

[Ste05]: capítulo 10

Características

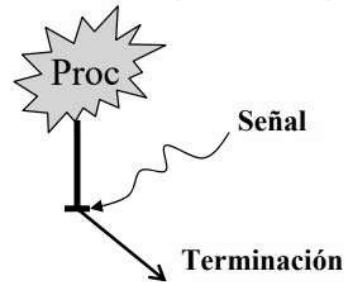
- Comunican eventos especiales a procesos en ejecución
 - Son interrupciones software
 - No contienen información. El proceso señalado no conoce la identidad del proceso señalador
 - No es la forma habitual de comunicar procesos. Pueden servir para sincronizar

¿Quién puede enviar señales?

- Un proceso recibe una señal originada por:
 - El kernel en respuesta a una condición hardware violenta:
 - SIGSEGV intento de acceso a @ fuera de su espacio
 - SIGFPE error en aritmética FP
 - El kernel en nombre del propio proceso:
 - el proceso se quiere programar una alarma
 - El kernel en nombre del usuario:
 - se generan desde el teclado
 - Ctrl \ señala a todos los procesos creados por el usuario
 - El kernel en nombre de otro proceso:
 - se generan con kill por parte del proceso señalador

Comportamientos (1)

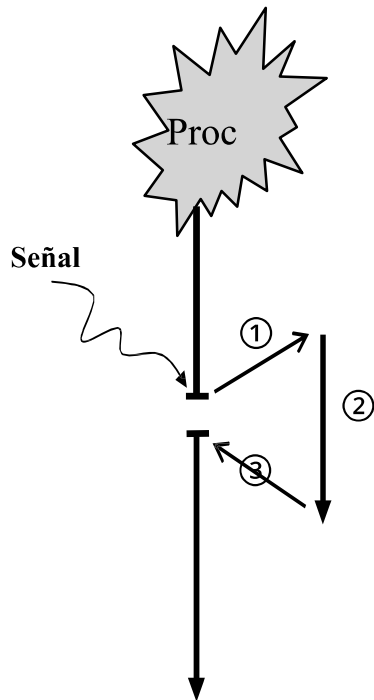
- El proceso señalado puede:
 - ① No querer manifestar su disposición frente a la señal
actúa el comportamiento por **defecto**
normalmente terminación del proceso (+/- core)



- ② Manifestar su disposición a **ignorar** la señal
el proceso se hace inmune a la señal

Comportamientos (2)

- ③ Querer **capturar** la señal
actúa el comportamiento
programado



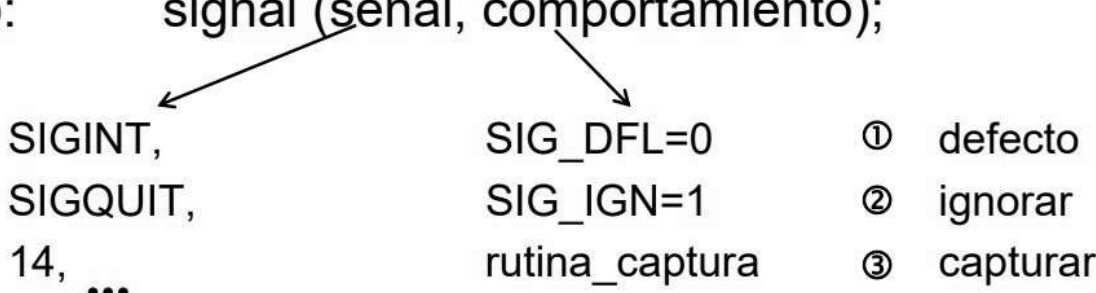
- ① Llamada a la rutina de tratamiento (signal handler)
- ② Ejecución y posible indicación de querer capturar de nuevo la señal
- ③ Retorno y restauración del contexto (pila de usuario)

Tipos

- Declaradas en <signal.h>
- Constantes que empiezan por SIG
- Todas asociadas a un entero positivo
- Lista completa en /usr/include/sys/iso/signal_iso.h

<u>Nombre</u>	<u>Núm.</u>	<u>Defecto</u>	<u>Observaciones</u>
SIGINT	2	terminación	Ctrl C
SIGQUIT	3	terminación + core	Ctrl \
SIGKILL	9	terminación	no ignorar ni capturar
SIGPIPE	13	terminación	escritura en pipe cerrada
SIGALRM	14	terminación	se fija con alarm
SIGTERM	15	terminación	como SIGKILL pero si ignorar
SIGUSR[1,2]	16,17	terminación	definidas por el programador
SIGCHLD	18	ignorada	muerte del proceso HIJO
SIGSTOP	23	detener	no ignorar ni capturar

Llamadas al sistema (1): **signal**

- Sintaxis: `# include <signal.h>`
`void (* signal (sig, func)) (int)`
`int sig; void (* func) (int);`
- Acción: Cambia comportamiento para la señal
- Uso: `signal (señal, comportamiento);`


SIGINT,	SIG_DFL=0	①	defecto
SIGQUIT,	SIG_IGN=1	②	ignorar
14, ...	rutina_captura	③	capturar
- Devolución: anterior comportamiento ó -1 si error
`#define SIG_ERR (void (*)()) -1`

Llamadas al sistema (1): **signal** (cont.)

- Si programamos capturar la señal:

En la rutina de captura: `void func (int)`

- ① reset del comportamiento al
- ② defecto ejecución del código de
- ③ la rutina retorno y restauración del estado

un solo parámetro para func = número de la señal

Si la señal llega durante la ejecución de una SC bloqueante: la interrumpe, devuelve <code>-1</code> y <code>errno = EINTR</code>
--

Llamadas al sistema (2): **kill**

- Sintaxis: `# include <signal.h>`
`int kill (pid_t pid, int sig)`
- Acción: envía una señal a un proceso
(ruid_señalado = euid_emisor)
- Uso: `kill (pid_señalado, señal);`
- Devolución: 0 si bien, -1 si error

Llamadas al sistema (2): **kill** (cont.)

- $pid > 0$ destino = pid
- $pid = 0$ destino = todos los procesos con *process group id* igual al del emisor
(todos los procesos de su grupo)
- $pid = -1$
 - a) si (emisor \neq superusuario)
destino = todos los procesos con *ruid* igual al *euid* del emisor
 - b) si (emisor $==$ superusuario)
destino = todos los procesos (! PID=0, PID=1)
- $pid < -1$ destino = procesos cuyo *process group id* igual al valor absoluto de pid

Llamadas al sistema (3): **alarm**

- Sintaxis: `# include <unistd.h>`
`unsigned int alarm (unsigned int sec)`
- Acción: programa la recepción de SIGALRM para dentro de `sec` segundos
- Uso: `alarm (5);` `/* alarm clock = 5 */`
`alarm (0);` `/* cancelación */`
Ojo !!: solo un alarm clock activo a la vez
Si se quiere capturar, `signal()` antes de `alarm()`
- Devolución: segs que quedan del anterior alarm
0 la primera vez

Llamadas al sistema (4): **pause**

Sintaxis: `# include <unistd.h>`

`int pause (void)`

Acción: suspende al proceso hasta la llegada de una señal cualquiera capturada (si defecto: terminar, si ignorar → sigue pause)

- Uso: `pause ();`

Devolución: `-1` si la señal se captura y se retorna del `signal_handler` (`errno = EINTR`)

Ejemplo sencillo señales

```
#include <signal.h>
void fcaptura();
main(){
    signal(SIGALRM, fcaptura);
    alarm(10);
    pause();
    printf("Trabajo terminado\n");
}
void fcaptura( ) { };
```

SIG_DFL
SIG_IGN
rutina de captura

devuelve?

reprogramación:
signal(SIGALRM,fcaptura);

Terminología

- Una señal es GENERADA para un proceso cuando ocurre el suceso que causa la señal
- Una señal es TRATADA en un proceso cuando la acción programada para la señal se lleva a cabo en el proceso receptor
- Una señal está PENDIENTE desde que se genera hasta que es tratada

Implementación

- Campos añadidos en la struct PCB del proceso

- Conjunto de señales que el proceso tiene pendientes:

Señales pendientes: 1 bit por señal
consultada cada vez que el proceso entra en
ejecución

- Indicación para cada señal del comportamiento
programado con signal

① `#define SIG_DFL (void (*)(void)) 0`

② `#define SIG_IGN (void (*)(void)) 1`

③ `@` rutina de captura

fork() *versus* señales

- El proceso HIJO:
 - No hereda las señales pendientes en el proceso

PADRE: señales pendientes en el HIJO = \emptyset

- Hereda todos los comportamientos programados por el proceso PADRE:

- ① defecto
- ② ignorar
- ③ capturar

exec() *versus* señales

- El proceso ejecutado:

- Mantiene todas las señales que están

pendientes: mismo struct PCB

- No mantiene todos los comportamientos programados:

① defecto

② ignorar

③ capturar pasa a defecto: Ya que ha desaparecido el código de la rutina de captura


`/* ej90.c */`

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

```
int main() {
    void newhandler(int);
    void (* syshandler)(int);

    syshandler = signal(SIGALRM, newhandler);
    alarm(5);
    printf("Me bloqueo esperando la alarma\n");
    pause();
    printf("Ya me he despertado\n");
    exit(0);
}
```

```
/* Rutina mia de servicio */
void newhandler(int n)
{
    printf("En la rutina de servicio %d\n", n);
};
```



**podemos imprimir
el parámetro...**

Ejercicio: /* autolesion.c */

```
#include <stdlib.h>
#include <signal.h>
#include "error.h"
#define MAL (void (*)(int)) -1
```

```
void main(){
    int i, *ip;
    void func(), captura();

    ip = (int *) func;
    for( i= 1; i<41; i++ )
        if ((i!=9)&&(i!=23)) /* no podemos capturar SIGKILL (9)
                               ni SIGSTOP(23) */
            if (signal( i, captura ) == MAL) syserr( "signal" );
    *ip = 1; /* PROVOCA EXCEPCION Y ENVIO DE SEÑAL SIGSEGV(11)*/
    printf( "Asignado ip\n" );
    func();
}
```



```
void func(){}

void captura(int n) {
    printf( "Capturada %d\n", n );
    exit( 1 );
}
```

SIGCHLD / SIGCLD

Señal enviada a un proceso cuando muere/para un hijo. Por defecto se **ignora**

SIGCHLD / SIGCLD

Señal enviada a un proceso cuando muere/para un hijo. Por defecto se **ignora**

ej1001.c: zombies en modo asíncrono

Solución?

- SIGCHLD (BSD, POSIX)
 - como el resto de señales (si se ignora, deja zombies)
- SIGCLD (System V, **Solaris**)
 - signal (SIGCLD, SIG_IGN)
 - NO deja zombies
 - Si se llama a wait => bloquea hasta muerte de todos los hijos y luego devuelve -1
 - signal (SIGCLD, captura)
 - si ya hay un zombie => llamar a captura() YA
 - sino se llamará a captura() cuando muera uno

En hendrix existen los 2 nombres, con el mismo valor = 18, con este comportamiento

Función de captura para SIGCLD

```
void captura () {  
    int pid, estado;  
    signal (SIGCLD, captura);  
    pid = wait (&estado);  
    printf ("Ha terminado PID=%d con estado %x \n",  
        pid, estado);  
}
```



Implementaciones antiguas de Sistem V puede entrar en bucle infinito de llamadas a captura sin ejecutar wait.

En Solaris 10 (hendrix) no pasa

Problemas con señales

- Las señales expuestas hasta ahora son las conocidas como señales no seguras (non reliable signals).
 - Se recomienda en programas modernos no utilizarlas
 - Es necesario conocerlas para analizar o cambiar programas viejos, por ello las seguiremos estudiando
- Problemas que dan las señales no seguras:
 1. Al capturar una señal el comportamiento de la misma cambia a SIG_DFL
 2. Posibles bloqueos por llegadas de señales en momentos no adecuados (por ej. antes de un pause)
 3. Interrupción de SCs bloqueantes

Problemas con las señales (1)

- Recordar:
con *signal* hay reset al SIG_DFL en la rutina de captura
- Solución (no siempre funciona):
reprogramar el comportamiento de captura en la rutina:

```
void rutina_captura() {  
    signal(SIGUSR1,  
           rutina_captura);  
    ...  
}
```

si llega señal => ¡el proceso acaba!

no siempre funciona

Problemas con las señales (1): /* sig_dfl.c */

Al capturar señal, el comportamiento pasa a SIG_DFL

```
#include <stdlib.h>, <unistd.h>, <signal.h>, "error.h"
#define MAL (void (*)(int))-1
#define ESPERA 1

int main() {
    if (signal(SIGINT, captura) == MAL) syserr("signal");
    while (1) pause();
}

void captura(int n) {
    void (*ff)(int);
    printf( "\nSeñal capturada %d\n", n );
    sleep(ESPERA);
    if (signal(SIGINT, captura) == MAL) syserr("signal");
    printf( "Nuevo comportamiento\n");
}
```

Problemas con las señales (2): pelosisig.c

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

main() {
    int pid;

    signal(SIGUSR1, f);
    if((pid=fork())==0) {
        pid=getppid();
        while(1) {
            fprintf(stderr, "h");
            kill(pid, SIGUSR1);
            pause();
        }
    }
    else{
        pause();
        while(1) {
            fprintf(stderr, "p");
            kill(pid, SIGUSR1);
            pause();
        }
    }
}
```

```
void f() {
    signal(SIGUSR1, f);
    fprintf(stderr, "-");
}
```

Dos posibles resultados:

- 1) h-p-h-p- ... -h-p- (acabado en p-)
- 2) h-p-h-p- ... -h- (acabado en h-)

en los dos casos, al final "se cuelga"

SOLUCIÓN?

Problemas con las señales (3): /* ej901.c */

Interrupción de llamadas al sistema bloqueantes

```
#include <stdio.h>, <stdlib.h>, <unistd.h>
#include <signal.h>, "error.h"

#define MAL (void (*)()) -1

void main(){
    if( signal(SIGALRM, trap) == MAL )
        syserr("signal");
    alarm(2);
    getchar();
    printf("Caracter leído\n");
    pause();          /* ¿? */
    puts("Aquí seguiría el programa...\n");
}
```

De señales no-seguras a señales seguras

Cosas que mejoran de *non-reliable signals* a *reliable signals*:

	<u>non-reliable</u>	<u>reliable</u>
• Posibilidad de dejar señales bloqueadas para qué? – ejercicio pelosig.c	NO	SI
• Desprotección del proceso en la rutina de captura por qué? – reset al DFT en captura	SI	opcional
• Posibilidad de <i>restart</i> SC bloqueantes interrumpidas	NO	opcional

Funciones no reentrantes => efectos laterales que siguen existiendo

mala programación!

Soluciones

- Necesitamos poder recordar la GENERACION de una señal para TRATARLA más tarde
 - Significado: poder retrasar la acción programada para la señal
- Necesitamos poder BLOQUEAR señales durante el tiempo que no queremos TRATARLAS
- Se puede hacer con señales seguras
 - signal_sets Conjunto de señales
 - sigprocmask SC para bloquear/desbloquear señales
 - sigpending SC para saber señales pendientes del proceso
- Entre sigprocmask y pause aun puede llegar una señal
 - Se resuelve ejecutando sigprocmask y pause de forma atómica, es decir, con **sigsuspend**

Signal sets

- Manipulación del conjunto de señales

recordar:	máscara de bits para señales pendientes, 1 bit por señal
<u>si</u> bit $i = 0 \Rightarrow$ señal número $i \notin$ set	<u>si</u> bit $i = 1 \Rightarrow$ señal número $i \in$ set

int sigemptyset (sigset_t *set)

int sigfillset (sigset_t *set)

int sigaddset (sigset_t *set, int signo)

int sigdelset (sigset_t *set, int signo)

int sigismember (sigset_t *set, int signo)

Llamar siempre a
sigemptyset o sigfillset
para asegurarse correcta
inicialización de sigset

Funciones sobre sets

(*tipo sigset_t definido
en signal.h*)

Signal mask y sigpending

- Signal mask: máscara indicando qué señales se deben BLOQUEAR para un proceso

Para modificar/consultar *signal mask*

int **sigprocmask** (int how, sigset_t *set, sigset_t *oset)

how: SIG_BLOCK añade *set a cjto. señales bloqueadas

SIG_UNBLOCK quita *set del cjto. señales bloq.

SIG_SETMASK pone nuevo cjto. señales bloq.

si oset != NULL devuelve en *oset el cjto. previo de señales bloqueadas

int **sigpending** (sigset_t *set)

Para consultar qué señales están pendientes

devuelve en *set el conjunto de señales pendientes

sigsuspend()

- Modifica signal mask (sigprocmask) y hace pause de forma atómica

int **sigsuspend** (sigset_t *sigmask)

- asigna sigmask a *signal mask*
- bloquea al proceso
hasta que llega una señal NO bloqueada NI ignorada
- restaura la máscara previa

sigaction()

- Versión *reliable* de signal (sustituye y amplía a signal)

int **sigaction** (int signo, struct sigaction *act, struct sigaction *oact)

- signo: número de la señal sobre la que fijar el comportamiento
- act: nuevo comportamiento (input)
- oact: anterior comportamiento (output)

struct sigaction

```
struct sigaction {  
    void (*sa_handler)();    /* SIG_IGN, SIG_DFL o captura */  
  
    sigset_t sa_mask;        /* señales a bloquear durante la captura */  
                             /* añadidas a signal mask (restaurado al  
                             acabar) */  
  
    int sa_flags;            /* para programar las opciones:  
}
```

defecto:

SA_RESTART (4)	reinicio automático de SC	no reinicio
SA_RESETHAND (2)	reset a DFL en captura	no reset
SA_NODEFER (16)	no bloqueo de la propia señal durante la función de captura	bloqueo

Solución de problemas con las señales

1. Reset al SIG_DFL en la rutina de captura modificado en señales seguras:
 - **Es opcional con sigaction**
 - NO hay reset a SIG_DFL en la rutina de captura (4.2 BSD y SVR3)
2. Bloqueo de señales imposible con no seguras
 - **Es posible con sigprocmask**
3. Un proceso captura señal mientras está bloqueado en una SC “lenta”, la SC es interrumpida, devuelve -1 y errno = EINTR. Si se quería reiniciar la SC había que hacerlo a mano.
 - **Es opcional con sigaction** (sa_flags de struct sigaction) para ciertas SCs (ioctl, read, readv, write, writev, wait, waitpid):
 - interrupción SIN reinicio (no poner flag SA_RESTART)
 - interrupción CON reinicio automático (SA_RESTART)

Problemas con las señales (seguras y no seguras)

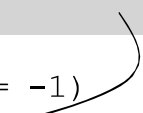
- Funciones no reentrantes (ej. *malloc*)
 - un proceso hace una llamada a *malloc*
 - durante el servicio de *malloc* llega una señal a **capturar**
 - en la rutina de captura se vuelve a llamar a *malloc*

Si la función no es reentrante (ej. usa variables *static*) la 2ª llamada puede provocar problemas a la 1ª

- En [Ste05] Fig 10.4 (pag 306) tabla de funciones reentrantes, es decir, que se pueden llamar con “menos” problemas desde rutina de captura.
- En general:
 - ¡Cuidado con los efectos laterales provocados por una rutina de captura!

señal que en captura modifica
errno

```
if (wait(NULL) == -1)
    if(errno ==
        ECHILD);
...
```



Ejercicio

Reescribir el siguiente código usando las llamadas de señales seguras. Deben programarse las señales de forma que su comportamiento sea lo más parecido posible al código original

```
void  captura() {  
    printf("Funcionde  captura...\n");  
}  
  
main() {  
    signal(SIGALRM, captura);  
    alarm(5);  
    pause();  
    printf("Fin programa\n");  
}
```