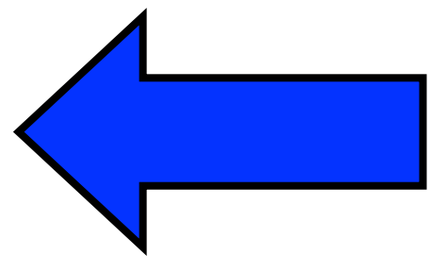
laboratorio di sistemi operativi

segnali

Marco Botta
Materiale preparato da D. Radicioni

argomenti del laboratorio UNIX

1. introduzione a UNIX;
2. integrazione C: operatori bitwise, precedenze, preprocessore, pacchettizzazione del codice, compilazione condizionale e utility make;
3. controllo dei processi;
4. segnali;
5. pipe e fifo;
6. code di messaggi;
7. semafori;
8. memoria condivisa;
9. introduzione alla programmazione bash.



- il materiale di questa lezione è tratto:
  - dai lucidi del Prof. Gunetti degli anni scorsi;
  - Michael Kerrisk, ***The Linux Programming interface - a Linux and UNIX® System Programming Handbook***, No Starch Press, San Francisco, CA, 2010;
  - W. Richard Stevens (Author), Stephen A. Rago, ***Advanced Programming in the UNIX® Environment*** (2nd Edition), Addison-Wesley, 2005;

cause che generano i segnali

- Un segnale è una notifica a un processo che è occorso un certo ***evento***. Fra i tipi di eventi che causano il fatto che il kernel generi un segnale per un processo ci sono i seguenti:

1. È occorsa una ***eccezione hardware***, l'HW ha verificato una condizione di errore che è stata notificata al kernel, il quale a propria volta ha inviato un segnale corrispondente al processo in questione.
   - per esempio, l'esecuzione di istruzioni di linguaggio macchina malformate, divisioni per ***0***, o riferimenti a parti di memoria inaccessibili.

cause che generano i segnali

2. L'utente ha digitato sul terminale dei ***caratteri speciali*** che generano i segnali.
   - questi caratteri includono il carattere ***interrupt*** (normalmente associato a ***Control-C***) e il ***suspend*** carattere (***Control-Z***).
3. È occorso un evento software.
   - per esempio, l'input è divenuto disponibile su un descrittore di file, un timer è arrivato a ***0***, il tempo di processore per il processo è stato superato, o un figlio del processo è terminato.

symbolic names and numbers

- I segnali sono definiti con ***interi*** unici, la cui sequenza inizia da 1. Tali interi sono definiti in ***<signal.h>*** (o in ***<sys/signal.h>***) con ***nomi simbolici*** della forma ***SIGxxxx***.
  - Poiché gli effettivi ***numeri*** utilizzati per ogni segnale variano a seconda delle implementazioni, all'interno dei programmi è meglio utilizzare questi nomi.
    - per esempio, quando l'utente digita il carattere dell'interrupt, ***SIGINT*** (il segnale numero 2) è inviato a un processo.

Listing signals with *strsignal(signum)*

```
// NSIG is a macro equal to the total number of signal
for (i=0; i<NSIG; i++) {
  printf("Signal #%2d: %s\n", i, strsignal(i));
}
```

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
24	SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
25	SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

	Function	*Character*
`intr`	Terminates the current job.	`^c (CTRL-c)`
`quit`	Terminates the current job; makes a core file.	`^\ (CTRL-\)`
`susp`	Stops the current job (so you can put it in the background).	`^z (CTRL-z)`

signals lifecycle

- Si dice che un segnale è ***generato da*** qualche evento.
- Dopo essere stato generato, un segnale è ***inviato*** (e ***delivered***: propriamente, ***consegnato***) ad un processo, che quindi ***esegue una qualche azione*** in risposta al segnale.
- Fra il momento in cui è generato e il momento in cui è inviato al processo, il segnale è ***pendente*** (***pending***).

pending signals, delivery and block

- Di norma, un segnale pendente è inviato a un processo appena il processo è scelto per l'esecuzione, oppure immediatamente se il processo è già in esecuzione (per esempio, nel caso in cui il processo invia un segnale a se stesso).
- A volte invece è necessario assicurare che un segmento di codice non sia interrotto dalla consegna di un segnale.
  - in questo caso, possiamo aggiungere un segnale alla ***maschera dei segnali*** del processo, cioè un insieme di segnali la cui ricezione è attualmente bloccata.
  - Se un segnale è generato mentre il processo è bloccato, il segnale rimane pendente fino a quando non viene successivamente sbloccato e rimosso dalla maschera dei segnali. Varie system call permettono ai processi di aggiungere e rimuovere segnali dalla propria maschera dei segnali.

delivery and process actions

- Al momento della ricezione di un segnale, un processo continua con una delle seguenti ***azioni di default***, a seconda del segnale:

1. Il segnale è ***ignorato***; in questo caso viene scartato dal kernel e non ha effetti sul processo (il processo non è neppure informato del fatto che quel segnale è occorso).
2. Il processo viene ***terminato*** (killed). Questa terminazione è detta anche ***abnormal process termination***, opposta alla terminazione normale del processo, che occorre quando un processo termina usando ***exit()***.

delivery and process actions (2)

3. È generato un file contenente un ***core dump file***, e il processo viene terminato.
   - Un file con ***core dump*** contiene un'immagine della memoria virtuale del processo; tale immagine può essere caricata in un debugger per ispezionare lo stato del processo al momento della terminazione.
4. Il processo viene bloccato (***stopped***): l'esecuzione è in questo caso sospesa.
5. L'esecuzione del processo è ripresa (***resumed***) dopo essere stata bloccata in precedenza.

filter_per_pipe_2015-11-20-093754_echo.crash

Hide Log List | Move to Trash | Clear Display | Insert Marker | Reload | Search | Filter

- es2_main_pipe_2015-11-20-101443_echo.crash
- es2_main_pipe_2015-11-20-101709_echo.crash
- es2_main_pipe_2015-11-20-101915_echo.crash
- f_pipe_per_filter_2015-11-20-093548_echo.cr...
- f_pipe_per_filter_2015-11-20-093754_echo.cr...
- filter_per_pipe_2015-11-20-093548_echo.cra...
- **filter_per_pipe_2015-11-20-093754_echo.crash**
- iconservicesagent_2015-11-11-184453_echo....
- java_2015-11-10-155335_echo.crash
- java_2015-11-10-155901_echo.crash
- Mail_2015-11-02-063801_echo.crash
- mdworker_2015-11-29-165557_echo.crash
- prova_kill_2015-11-29-191844_echo.crash
- QuickLookSatellite_2015-11-05-144332_echo...
- simple_pipe_modificato_2015-11-16-072555_...
- tm_dialog2_2015-11-13-135340_echo.crash
- Yummy FTP_2015-11-11-182847_echo.crash
- Yummy FTP_2015-11-11-182856_echo.crash
- Yummy FTP_2015-11-11-184037_echo.crash
- Yummy FTP_2015-11-11-184902_echo.crash
- Yummy FTP_2015-11-12-164257_echo.crash
- Yummy FTP_2015-11-12-164351_echo.crash
- Yummy FTP_2015-11-12-164356_echo.crash
- Yummy FTP_2015-11-27-122118_echo.crash
- Yummy FTP_2015-11-29-171252_echo.crash
- Yummy FTP_2015-11-29-171257_echo.crash
- Yummy FTP_2015-11-29-171301_echo.crash

▶ System Diagnostic Reports

FILES

- system.log
- ▶ ~/Library/Logs
- ▶ /Library/Logs
- ▶ /var/log

```
Process:               filter_per_pipe [33432]
Path:                  /Users/USER/Documents/*/filter_per_pipe
Identifier:            filter_per_pipe
Version:               0
Code Type:             X86-64 (Native)
Parent Process:        f_pipe_per_filter [33431]
Responsible:           Terminal [894]
User ID:               503

Date/Time:             2015-11-20 09:37:54.424 +0100
OS Version:            Mac OS X 10.11.1 (15B42)
Report Version:        11
Anonymous UUID:        A55D802B-ABF1-8E84-78EB-5AEF2C96A197

Sleep/Wake UUID:       4E6588EF-4DB5-4BA4-BE96-4ED66CB16117

Time Awake Since Boot: 150000 seconds
Time Since Wake:       900 seconds

System Integrity Protection: enabled

Crashed Thread:        0  Dispatch queue: com.apple.main-thread

Exception Type:        EXC_CRASH (SIGQUIT)
Exception Codes:       0x0000000000000000, 0x0000000000000000

Thread 0 Crashed:: Dispatch queue: com.apple.main-thread
0   libsystem_kernel.dylib                  0x00007fff92380132 __read_nocancel + 10
1   libsystem_c.dylib                       0x00007fff98867df1 _sread + 16
2   libsystem_c.dylib                       0x00007fff98867405 __srefill1 + 24
3   libsystem_c.dylib                       0x00007fff98867520 __srget + 14
4   libsystem_c.dylib                       0x00007fff98863239 getchar + 58
5   filter_per_pipe                         0x0000000102ecfe34 main + 20
6   libdyld.dylib                           0x00007fff98bec5ad start + 1

Thread 0 crashed with X86 Thread State (64-bit):
  rax: 0x0000000000000004  rbx: 0x00007fff78a492b0  rcx: 0x00007fff5cd30758  rdx: 0x0000000000001000
  rdi: 0x0000000000000000  rsi: 0x00007fd612802600  rbp: 0x00007fff5cd30770  rsp: 0x00007fff5cd30758
   r8: 0x000000000127d5a   r9: 0xffffffff00000000  r10: 0x0000000000000885  r11: 0x0000000000000206
  r12: 0x0000000000000000  r13: 0x0000000000000000  r14: 0x00007fff78a49bc0  r15: 0x0000000000000000
  rip: 0x00007fff92380132  rfl: 0x0000000000000207  cr2: 0x00007ff185015dfe

Logical CPU:     0
Error Code:      0x0200018c
Trap Number:     133


Binary Images:
       0x102ecf000 -        0x102ecffff +filter_per_pipe (0) <851CA87A-A97D-3878-9C93-9A736867BFE0> /Users/USER/Documents/*/filter_per_pipe
    0x7fff5ff1f000 -     0x7fff5ff55f5f  dyld (360.17) <03673B53-B8B7-34D1-ADCE-F449E78E39CC> /usr/lib/dyld
    0x7fff84da1000 -     0x7fff84da6ff7  libmacho.dylib (875.1) <A9EC23EC-11A0-3B4F-A8AC-B990C8267A6E> /usr/lib/system/libmacho.dylib
```

Size: 9 KB | Earlier | Later

Il sistema di segnali in Unix: le ***trap***

- Una classe di segnali sono le ***trap***: segnali generati da eventi prodotti da un processo e inviati al processo stesso.
- Alcune ***trap*** sono causate da comportamenti errati del processo stesso, e immediatamente inviate al processo che normalmente reagisce terminando.
  - per esempio tentativi di divisione per zero (*SIGFPE*), indirizzamento errato degli array (*SIGSEGV*), tentativo di eseguire istruzioni privilegiate (*SIGILL*), etc..

Il sistema di segnali in Unix: gli *interrupt*

- Gli interrupt sono segnali inviati ad un processo da un agente esterno: l'utente o un altro processo
- Utente (da terminale):
  - ***CTRL-C*** (invia ***SIGINT***)
  - ***CTRL-Z*** (invia ***SIGSTOP***)
  - Comando ***kill***: ***kill –s signame PID***
  - Comando ***kill***: ***kill –signumber PID***
- Altro processo:
  - System call kill: ***kill(PID, SIGNAL)***

setting the disposition

- Invece di accettare l'azione di default per un particolare segnale, un programma può modificare l'azione da intraprendere al momento della consegna (***delivery***) del segnale. Questo è noto come ***impostazione della disposizione del segnale***. Un programma può impostare una delle seguenti disposizioni del segnale:
  - L'***azione di default*** dovrebbe essere intrapresa. Utile per cancellare precedenti modifiche della disposizione del segnale che modificavano la disposizione di default.
  - Il segnale è ***ignorato***. Utile per un segnale la cui disposizione sarebbe quella di terminare il processo.
  - Viene eseguito un ***signal handler***.

signal handlers

- Un signal handler o ***gestore di segnali*** è una funzione, scritta dal programmatore, che esegue azioni appropriate in risposta alla ricezione di un segnale.
  - Per esempio, la shell ha un gestore per il segnale ***SIGINT*** (generato dal carattere interrupt, ***Control-C***) che causa il suo blocco (***stop***) e la restituzione del controllo al ciclo di input principale: in questo caso all'utente viene presentato il prompt della shell.
  - ***Installare*** o ***stabilire un signal handler*** significa notificare al kernel che deve essere invocata una funzione handler.
  - Quando un handler è invocato in risposta alla ricezione di un segnale, si dice che il segnale è stato gestito (***handled***) o intercettato (***caught***).

Signal Types and Default Actions

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
24	SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
25	SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

6 SIGABRT create core image abort program (formerly SIGIOT)
14 SIGALRM terminate process real-time timer expired

- *SIGABRT*. Un processo riceve questo segnale quando invoca la funzione ***abort()***. Di default questo segnale termina il processo con un core dump. Questo produce l'effetto della chiamata ***abort()***, che produce un core dump a fini di debug.

- *SIGALRM*. Il kernel genera questo segnale al momento del raggiungimento dello zero di un timer impostato da una chiamata ad ***alarm()*** o ***setitimer()***.

20	*SIGCHLD*	*discard signal*	*child status has changed*
19	*SIGCONT*	*discard signal*	*continue after stop*

- *SIGCHLD*. Segnale inviato dal kernel a un processo genitore quando uno dei figli termina (chiamando ***exit(),*** o ucciso da un qualche segnale). Può essere inviato a un processo quando uno dei suoi figli è bloccato o risvegliato da un segnale.

- *SIGCONT*. Quando viene inviato a un processo bloccato (***stopped***), questo segnale causa il risveglio del processo (***resume***), cioè che il processo venga schedulato per successivamente essere eseguito. Quando è ricevuto da un processo che non è bloccato, questo segnale è ignorato di default. Un processo può intercettare questo segnale, in modo da eseguire qualche azione particolare al momento della ripresa dell'esecuzione.

2	*SIGINT*	*terminate process*	*interrupt program*
9	*SIGKILL*	*terminate process*	*kill program*
13	*SIGPIPE*	*terminate process*	*write on a pipe with no reader*

- *SIGINT*. Quanto l'utente digita il carattere di interrupt (***Control-C***), il terminale invia questo segnale al gruppo del processo in foreground. L'azione di default per questo segnale è terminare il processo.
- *SIGKILL*. È il segnale ***sicuro*** di ***kill***. Non può essere bloccato, ignorato, o intercettato da un handler, e quindi termina sempre un processo.
- *SIGPIPE*. Segnale generato quando un processo tenta di scrivere su un ***pipe*** o un ***FIFO*** per il quale non c'è un corrispondente processo lettore. Questo normalmente occorre perché il processo lettore ha chiuso il proprio file descriptor per il canale ***IPC***.

11 SIGSEGV create core image segmentation violation

- ***SIGSEGV***. Segnale generato quando un programma tenta un riferimento in memoria non valido. Il riferimento può non essere valido perché la pagina riferita non esiste (per esempio, giace in un'area non mappata, fra lo heap e lo stack), oppure il processo ha tentato di modificare una locazione in read-only memory (il segmento di testo del programma o una regione di memoria marcati come disponibili in sola lettura), o il processo ha tentato di accedere a una parte della memoria del kernel durante l'esecuzione in ***user mode***.
  - In C, questi eventi spesso derivano dalla dereferenziazione di un puntatore che contiene un 'bad address' (come un puntatore non inizializzato) o dal passaggio di un argomento non valido in una chiamata a funzione.
  - Il nome del segnale deriva da ***segmentation violation***.

17 SIGSTOP stop process stop (cannot be caught or ignored)

15 SIGTERM terminate process software termination signal

- ***SIGSTOP***. Segnale per il blocco (***stop***) sicuro. Non può essere bloccato, ignorato, o intercettato da un handler; quindi questo segnale blocca ***sempre*** un processo.
- ***SIGTERM***. Segnale standard utilizzato per terminare un processo; inviato di default dai comandi ***kill*** e ***killall***. Gli utenti a volte inviano esplicitamente il segnale ***SIGKILL*** a un processo, usando kill –***KILL*** or ***kill –9***.
  - in generale, questo è un errore. Un'applicazione ben progettata deve avere un handler per ***SIGTERM*** che consenta una 'graceful' exit, che consenta di pulire i file temporanei e di rilasciare le altre risorse. L'uccisione di un processo con ***SIGKILL*** bypassa l'handler di ***SIGTERM***, e quindi si dovrebbe sempre prima cercare di terminare un processo con ***SIGTERM***, e tenere ***SIGKILL*** come ultima scelta per terminare i processi che non rispondono a ***SIGTERM***.

5 SIGTRAP create core image trace trap
18 SIGTSTP stop process stop signal generated from keyboard

- *SIGTRAP*. Segnale utilizzato per implementare i breakpoint in fase di debugging e per la tracciatura delle system call.
  - Cercare ***ptrace()*** sul manuale per ulteriori informazioni.
- *SIGTSTP*. Segnale per lo stop, inviato per bloccare il gruppo di processi in foreground quando l'utente digita il carattere di sospensione (***Control-Z***) sulla tastiera.
  - il nome di questo segnale deriva da "terminal stop".

30	*SIGUSR1*	*terminate process*	*User defined signal 1*
31	*SIGUSR2*	*terminate process*	*User defined signal 2*

- *SIGUSR1*. Questo segnale e *SIGUSR2* sono disponibili per fini specificati dal programmatore. Il kernel non genera mai questi segnali per un processo.
  - I processi possono utilizzare questi segnali per notificarsi a vicenda eventi, o per sincronizzarsi.

Changing Signal Dispositions: *signal()*

signal() and *sigaction()*

- I sistemi UNIX forniscono due modi per cambiare la disposizione di un segnale: ***signal()*** e ***sigaction()***.
  - La system call ***signal()*** è l'API originale per assegnare la disposizione di un signal, e fornisce un'interfaccia più semplice di ***sigaction()***.
- Inoltre, vi sono differenze nel comportamento di ***signal()*** fra le varie implementazioni di UNIX, il che la rende più problematica per lo sviluppo di programmi portabili.
  - A causa di questi problemi di portabilità, ***sigaction()*** è l'API migliore per gli handler di segnali.

```
#include <signal.h>

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

     Returns previous signal disposition on success, or
                                          SIG_ERR on error
```

- Il primo argomento, ***sig***, identifica il segnale di cui vogliamo modificare la disposizione.
- Il secondo argomento, ***handler***, è l'***indirizzo*** della funzione che dovrebbe essere chiamata quando questo segnale è inviato (e consegnato, ***delivered***).
  - Questa funzione non restituisce (è ***void***) e prende un intero. Quindi un handler di segnali ha la seguente forma generale:

```
void handler(int sig) {
  /* codice dell'handler */
}
```

- Il valore di ritorno di ***signal()*** è la precedente disposizione del segnale. Come l'argomento dell'handler, questo è un puntatore a funzione che non restituisce nulla e che prende un intero come argomento.
- Il seguente blocco di codice stabilisce un handler temporaneo per un segnale, e quindi setta nuovamente la disposizione del segnale come era impostata prima :

```
void (*oldHandler)(int);
oldHandler = signal(SIGINT, newHandler);
if (oldHandler == SIG_ERR)
  errExit("signal error!");

/* altre istruzioni ...
   se durante l'esecuzione arriva un SIGINT,
   la sua gestione sarà affidata a newHandler */

if (signal(SIGINT, oldHandler) == SIG_ERR)
  errExit("signal error!");
```

2 disposizioni particolari

- Se la disposizione è settata a ***SIG_IGN***, il segnale è ignorato.
- Se la disposizione è settata a ***SIG_DFL***, l'azione di default è associata con il segnale.

signal(SIGTERM, SIG_DFL);

puntatori a funzioni

pointers to functions

```
int func(int a, float b);
```

- dichiarazione dei puntatori a funzione: partiamo dalla dichiarazione della funzione.

pointers to functions

```
int func(int a, float b);
```

- dichiarazione dei puntatori a funzione: partiamo dalla dichiarazione della funzione.
- e mettiamo le parentesi attorno al nome e uno * prima del nome. a causa della precedenza, se mancano le parentesi, si dichiara una funzione che restituisce un puntatore:

```
/* function returning pointer to int */
int *func(int a, float b);
/* pointer to function returning int */
int (*func)(int a, float b);
```

usage

- una volta creato il puntatore, possiamo assegnare l'indirizzo al giusto tipo di funzione semplicemente usando il suo nome;
- come per gli array, il nome della funzione è convertito in un indirizzo quando è utilizzato in un'espressione. possiamo chiamare la funzione in una delle due forme:

```
(*func)(1,2);
func(1,2);
```

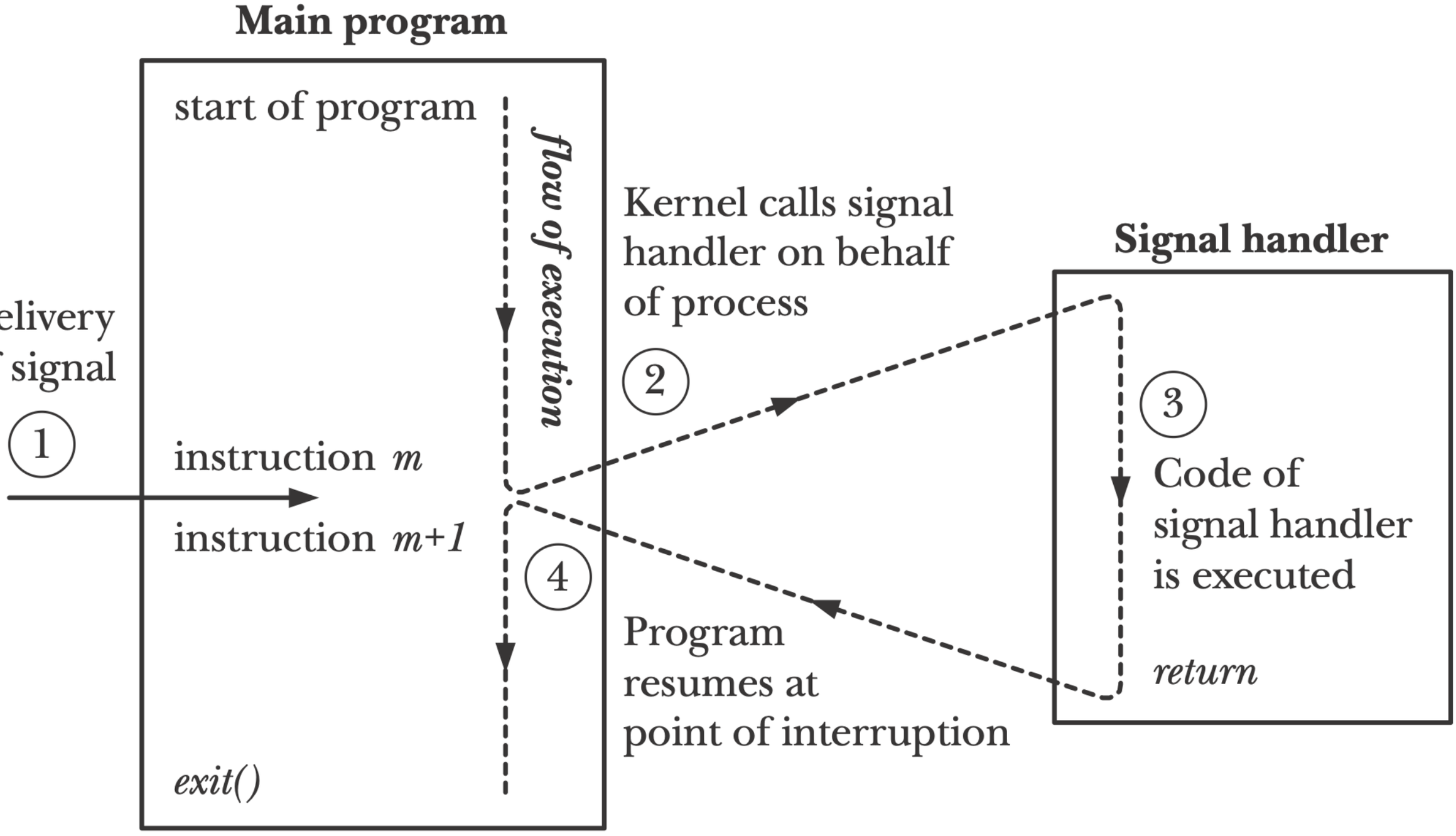
usage example

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

void func(int);

int main(void){
   void (*fp)(int);
   fp = func;
   (*fp)(1);
   fp(2);
   return 0;
}
void func(int arg){
   printf("%d\n", arg);
}
```

Signal Handlers

signal handlers

- Un s***ignal handler*** è una funzione chiamata quando un processo riceve uno specifico segnale.
- L'invocazione di un handler può interrompere il flusso principale del programma in qualsiasi momento;
  - il kernel chiama l'handler da parte del processo, e
  - quando l'handler restituisce, l'esecuzione del programma riprende dal punto in cui l'handler lo aveva interrotto.



- Cosa succede se un processo riceve un segnale mentre è sospeso su una ***semop*** o su una *msgrcv*?
  - Dipende, ma nella maggior parte degli Unix si prosegue con l'istruzione successiva alla ***semop / msgrcv*** ...

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

static void dd_signal_handler(int sig) {
  printf("ahiaaaaaa!\n");
}

int main(int argc, char *argv[]) {
  int j;

  if (signal(SIGINT, dd_signal_handler) == SIG_ERR)
    errExit("signal (SIG_ERR) error");

  for (j = 0; ; j++) {
    printf("%d\n", j);
    sleep(1);
  }
}
```

```
...
#inc signal.h>

static void dd_signal_h
  printf("ahiaaaaaaa!\n"
}

int main(int argc, char
  int j;

  if (signal(SIGINT, dd_
    errExit("signal (SI

  for (j = 0; ; j++) {
    printf("%d\n", j);
    sleep(1);
  }
}
```

Ctrl+C

Ctrl+\

```
$./prova_segnali
0
1
^Cahiaaaaaaa!
2
3
4
^Cahiaaaaaaa!
5
^Cahiaaaaaaa!
6
^Cahiaaaaaaa!
7
8
9
^\Quit
$
```

```c
static void dd_signal_handler(int sig) {
  static int count = 0;
  // NB: questa implementazione è UNSAFE: l'handler utilizza
  // funzioni non-async-signal-safe, come printf(), exit()
  if (sig == SIGINT) {
    count++;
    printf("intercettato SIGINT (%d)\n", count);
    return; // l'esecuzione riprenderà dall'istruzione successiva
            // a quella in esecuzione al momento dell'interruzione
  }
  // PRE: se non è SIGINT è SIGQUIT -
  //      stampo un msg prima di terminare
  printf("intercettato SIGQUIT - termino!!!\n");
  exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[]) {
    // associo lo stesso handler ai due segnali SIGINT e SIGQUIT
    if (signal(SIGINT, dd_signal_handler) == SIG_ERR)
        errExit("signal SIGINT error");
    if (signal(SIGQUIT, dd_signal_handler) == SIG_ERR)
        errExit("signal SIGQUIT error");
    for (;;) // ciclo infinito di attesa dei segnali
      pause(); // il processo bloccato finché non riceve un segnale
}
```

```c
void handler_int(int s) {
  printf("\n ricevuto segnale SIGTERM \n");
}

int main(int argc, char *argv[]) {
  int i;
  // associazione segnale-handler
  signal(SIGTERM, handler_int);

  while(1)  sleep(1);
  exit(EXIT_SUCCESS);
}
```

- Lanciate il programma in background e date più volte il comando "kill %1" Cosa succede? Perché? Che cosa contiene la variabile s?

```c
void handler_int(int s) {
  printf("\n ricevuto segnale SIGTERM \n");
  signal(SIGTERM, SIG_DFL);// reset dell'associazione
}

int main(int argc, char *argv[]) {
  int i;
  // associazione segnale-handler
  signal(SIGTERM, handler_int);

  while(1) sleep(1);
  exit(EXIT_SUCCESS);
}
```

- Notate la ***signal*** dentro ***handler_int***. Ora cambia qualche cosa rispetto all'esempio precedente?

persistenza dell'associazione

- Nei sistemi in cui l'associazione con il nuovo handler vale per un'unica ricezione del segnale, se si vuole rendere persistente l'associazione, bisogna ripeterla a ogni invocazione dell'handler.

```
void handler_int(int s) {
  signal(SIGTERM, handler_int);
  printf("\n ricevuto segnale SIGTERM \n");
}
```

Sending Signals: *kill()*

kill() system call

```
#include <signal.h>

int kill(pid_t pid, int sig);

                Returns 0 on success, or -1 on error
```

- L'argomento ***pid*** identifica uno o più processi a cui inviare il segnale; ***pid*** può essere interpretato in 4 modi:
  - *pid > 0*: il segnale è inviato al processo identificato da ***pid***.
  - *pid == 0*: il segnale è inviato a ogni processo nello stesso gruppo del chiamante, chiamante incluso.
  - *pid < –1*: il segnale è inviato a tutti i processi nel gruppo del processo il cui *ID* è uguale al valore assoluto di ***pid***. Inviare un segnale a tutti i processi nel gruppo di un processo è utile nel controllo dei job effettuato con la shell.
  - *pid == –1*: (broadcast signal) il segnale è inviato a tutti i processi per i quali il processo ha i permessi di inviare un segnale, eccetto ***init*** (che ha ***pid 1***) ed il chiamante. Se l'utente non è super user, il segnale è inviato a tutti i processi con stesso uid dell'utente, escluso il processo che invia il segnale.

```
#include <signal.h>

int kill(pid_t pid, int sig);

                                Returns 0 on success, or -1 on error
```

- Se nessun processo corrisponde al ***pid*** predefinito, ***kill()*** fallisce e setta errno a ***ESRCH*** ("No such process")
- Verifica dell'esistenza di un processo. Se l'argomento ***sig*** è settato a ***0*** (detto ***null signal***), non è inviato alcun segnale.
  - In questo caso ***kill()*** esegue unicamente un controllo degli errori per vedere se è possibile inviare segnali al processo: il ***null signal*** può essere utilizzato per testare se un processo con un certo ***pid*** esiste.
  - Se la chiamata fallisce con errore ***EPERM***, il processo esiste, ma non abbiamo i permessi per inviargli un segnale.
  - Se la chiamata va a buon fine, sappiamo che il processo esiste.

kill() system call

```
int main() {
  printf("process with pid %ld\n", (long)getpid());
  printf("going to sleep for 3 secs\n
            and then SIGKILLing...\n");
  sleep(3);

// (broadcast signal) il segnale è inviato a tutti i
// processi per i quali il processo ha i permessi di
// inviare un segnale, eccetto init (che ha pid 1) ed
// il chiamante. Se l'utente non è super user, il
// segnale è inviato a tutti i processi con stesso
// uid dell'utente, escluso il processo che invia il
// segnale.
  kill(-1,SIGKILL);
}
```

sigaction system call

sigaction() system call

```
#include <signal.h>

int sigaction(int signum,
        const struct sigaction *act,
        struct sigaction *oldact);

          Returns 0 if successful; otherwise the value -1 is
              returned and the global variable errno is set.
```

- syscall utilizzata per impostare un gestore di segnali
  - signum: numero del segnale da gestire
  - act, puntatore al nuovo gestore del segnale; se NULL il gestore resta invariato
  - oldact, puntatore al vecchio gestore; se impostato a NULL non viene restituito alcun handler.

- NB: sigaction è ***sia una sys call sia una struct***

sigaction() system call

```
#include <signal.h>

int sigaction(int signum,
        const struct sigaction *act,
        struct sigaction *oldact);

          Returns 0 if successful; otherwise the value -1 is
              returned and the global variable errno is set.
```

- syscall utilizzata per impostare un gestore di segnali
  - signum: numero del segnale da gestire
  - act, puntatore al nuovo gestore del segnale; se NULL il gestore resta invariato
  - oldact, puntatore al vecchio gestore; se impostato a NULL non viene restituito alcun handler.

- NB: sigaction è ***sia una sys call sia una struct***

sigaction() system call

- (dichiarazione e) tre possibili invocazioni

```
struct sigaction new, old;

sigaction(signum, new, NULL); //set new handler to new
sigaction(signum, NULL, old); //current handler in old
sigaction(signum, new, old);  //do both
```

the sigaction structure

```
#include <signal.h>

struct sigaction {
 void (*sa_handler)(int signum);
 sigset_t sa_mask;
 int sa_flags;
 // plus others (for advanced usage)
};
```

- *sa_handler*, puntatore alla funzione per la gestione del segnale;
- *sa_mask*, maschera dei segnali bloccati durante l'esecuzione dell'handler;
- *sa_flags*, insieme di flag messi in bitwise OR per modificare il comportamento del segnale.

mask & fork()

- The collection of signals that are currently blocked is called the signal mask.
  - Each process has its own signal mask.
- When a new process is created, it inherits its parent's mask.
  - You can block or unblock signals with total flexibility by modifying the signal mask.

sblocco del segnale durante l'handler

- quando un segnale è inviato a un processo durante l'esecuzione di un handler, il segnale è automaticamente bloccato fino a quando l'handler restituisce
  - a meno che non sia settato il flag *SA_NODEFER*

```
void handle_signal(int signal);

int main() {
 struct sigaction sa;
 sigset_t  my_mask;
 sa.sa_handler = &handle_signal; // funct ptr
 sa.sa_flags = SA_NODEFER;  // allow nested invocations
 sigemptyset(&my_mask);     // do not mask any signal
 sa.sa_mask = my_mask;
 sigaction(SIGUSR1, &sa, NULL); // set the handler
 // further code
}
```

signal mask

- la maschera è la collezione di segnali ***attualmente blocked***.
  - ogni processo ha la propria maschera di segnali: un nuovo processo eredita la maschera del genitore
- le maschere sono di tipo ***sigset_t***. le funzioni per la manipolazione dei set sono:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

- man ***sigsetops*** per ulteriori dettagli

signal mask

```
int sigemptyset(sigset_t *set);
                        initializes a signal set to be empty.

int sigfillset(sigset_t *set);
                initializes a signal set to contain all signals.

int sigaddset(sigset_t *set, int signum);
              adds the specified signal signo to the signal set.

int sigdelset(sigset_t *set, int signum);
          deletes the specified signal signo from the signal set.

int sigismember(const sigset_t *set, int signum);
     returns whether a specified signal signo is contained in
                                                  the signal set.
```

Signal sets

- The ***sigset_t*** data type is used to represent a signal set. Internally, it may be implemented as either an integer or structure type.
  - For portability, use only the listed functions to initialize, change, and retrieve information from ***sigset_t*** objects—don't try to manipulate them directly.
- There are two ways to initialize a signal set.
  - Either you can initially specify it to be empty with ***sigemptyset***, and then add specified signals individually;
  - Or you can specify it to be full with ***sigfillset*** and then delete specified signals individually.

```c
void handle_signal(int signal) {
  printf("Got signal #%d: %s\n", signal, strsignal(signal));
}

int main() {
  struct sigaction sa;
  sigset_t  my_mask;
  int i;

  sa.sa_handler = &handle_signal;
  sigemptyset(&my_mask);   // signal mask is now EMPTY
  sa.sa_mask = my_mask;
  sa.sa_flags = 0;              // no special behavior

  for (i=0; i<NSIG; i++) { // set the handler for all signals
    if (sigaction(i, &sa, NULL) == -1) {
      fprintf(stderr,
        "Cannot set a user-defined handler for Signal #%d: %s\n",
        i, strsignal(i));
    }
  }
  for (;;) {
    printf("Sleeping for 3 seconds\n");
    sleep(3);
  }
}
```

setting the signal mask for a process

```
#include <signal.h>

int sigprocmask(int how,
                const sigset_t *set,
                sigset_t *oldset);
  Returns 0 if successful; otherwise the value -1 is returned
                        and the global variable errno is set.
```

- per impostare la maschera di segnali bloccati durante l'esecuzione dei processi si usa la syscall ***sigprocmask()***;
- l'argomento ***how*** può assumere i seguenti valori:
  - SIG_BLOCK: i segnali nel set sono bloccati;
  - SIG_UNBLOCK: i segnali nel set sono rimossi dalla maschera esistente;
  - SIG_SETMASK: il set diventa la nuova maschera del segnale.
- ***oldset*** è la vecchia maschera;

```
static int segnale_ricevuto = 0;
int main (int argc, char *argv[]) {
  sigs
  sigs
  stru
      utilizzando la sigprocmask()
  prin
  mems
  act.
      1. mascheriamo (=blocchiamo) SIGTERM per 20
  if (secondi utilizzando sigprocmask().
  // g
  sige
  siga
      2. dopo 20 secs il segnale viene sbloccato e il
  if (segnale gestito.
    ;
  slee
  if (
    ;
  slee
  if (
  return 0;
}
```

```c
static int segnale_ricevuto = 0;

...

static void mio_handler (int sig) {
   segnale_ricevuto = 1;
}
```

```c
static int segnale_ricevuto = 0;
int main (int argc, char *argv[]) {
  sigset_t orig_mask;
  sigset_t mask;
  struct sigaction act;

  printf("process pid: %d\n", getpid());
  memset (&act, 0, sizeof(act));
  act.sa_handler = mio_handler;

  if (sigaction(SIGTERM, &act, 0))
    ; // gestione errore
  sigemptyset (&mask); // inizializzo maschera vuota
  sigaddset (&mask, SIGTERM); // aggiungo SIGTERM

  if (sigprocmask(SIG_BLOCK, &mask, &orig_mask) < 0)// set mask
    ; // gestione errore
  sleep (20);
  if (sigprocmask(SIG_SETMASK, &orig_mask, NULL) < 0)
    ; // gestione errore
  sleep (1);
  if (segnale_ricevuto) puts ("signal ricevuto!!!");
  return 0;
}
```

```c
static void mio_handler (int sig) {
   segnale_ricevuto = 1;
}
```

signal mask during a handler

- l'handler può comunque essere interrotto da altri segnali (o dallo stesso segnale, nel caso *SA_NODEFER* sia settato)
  - quando l'handler restituisce, l'insieme di segnali bloccati è reimpostato al suo valore precedente la sua esecuzione, indipendentemente dalle possibili manipolazioni dei segnali bloccati eventualmente presenti nell'handler.

signal mask during a handler

- l'handler può comunque essere interrotto da altri segnali (o dallo stesso segnale, nel caso ***SA_NODEFER*** sia settato)

```
struct sigaction sa;
sigset_t my_mask;

sa.sa_handler = &handle_signal; // handler
sa.sa_flags = 0; // No special behaviour

// Create an empty mask
sigemptyset(&my_mask); // Do not mask any signal
sigaddset(&my_mask, signal_to_mask_in_handler);
sa.sa_mask = my_mask;

sigaction(SIGINT, &sa, NULL); // Set the handler
```

delivery di segnali a processi sospesi

- all'arrivo (asincrono) di un segnale
  1. Lo stato del processo è salvato (registers, etc.)
  2. La funzione dell'handler è eseguita
  3. Lo stato del processo è restored
- per processi in attesa su ***wait()***, o sospesi con ***pause()*** o ***sleep()*** sono possibili 2 comportamenti:
  - il processo non è in esecuzione (è sospeso su qualche system call); oppure
  - la funzione dell'handler è eseguita normalmente.
- quando l'handler ritorna:
  A. la syscall restituisce un errore, con ***errno*** settato a ***EINTR***; o
  B. la syscall viene automaticamente ripresa.
- A o B? dipende dal sistema operativo, e dal flag ***SA_RESTART*** nella syscall ***sigaction()***.

writing handlers

- l'handler deve essere associato al corrispondente segnale ***prima*** che il segnale possa essere (lanciato e) ricevuto dal processo: immaginando di dovere gestire un segnale ***SIGCHLD***, è necessario associare il gestore al segnale prima della ***fork()***;
- spesso una funzione può servire a gestire vari tipi di segnale.

```
void handle_signal(int signum) {
   switch (signum) {
      case SIGINT:
      // gestione SIGINT
      break;
      case SIGALRM:
      // gestione SIGALRM
      break;
      // gestione altri segnali
   }
}
```