

Il ciclo di vita dei processi

Ing. Giovanni Nardini - University of Pisa - All rights reserved

Cosa succede nel sistema quando si esegue un programma?

La funzione main

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

numero di argomenti

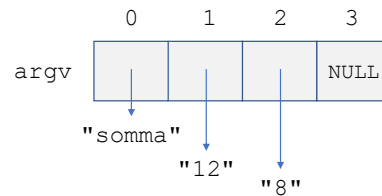
lista di argomenti
(array di stringhe)

- La funzione `main` è generalmente il punto di partenza del programma
- Gli argomenti sono passati dalla linea di comando

```
$ gcc somma.c -o somma
```

```
$ ./somma 12 8
```

argc = 3



Il `main` è una funzione. Come tutte le funzioni, può avere degli argomenti, in particolare:

- un array di stringhe (`argv`)
- il numero di elementi dell'array di stringhe (`argc`).

Gli elementi dell'array `argv` sono riempiti usando la stringa digitata nella linea di comando quando si lancia il programma, identificando le singole stringhe separate da spazi.

Il primo elemento dell'array è sempre il nome del programma che è stato eseguito, ed esiste sempre un elemento aggiuntivo nell'array, impostato a `NULL`, e indica la fine della lista di argomenti: `argv[argc] = NULL`.

Notare che tutti gli elementi di `argv` sono stringhe, per cui dovremo effettuare una conversione se vogliamo considerarli diversamente. Nell'esempio della slide, "12" e "8" sono stringhe, e prima di effettuarne la somma devo convertirli a un numero intero (vedere slide successiva).

Gli argomenti della funzione `main` sono opzionali – infatti, molto spesso scriviamo solamente `int main()`.

Se vogliamo avere la possibilità di specificare dei parametri dalla linea di comando, dobbiamo invece specificarli esplicitamente nel nostro programma.

Parametri dalla linea di comando

somma.c

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    int a, b;
    int i;
    for (i = 0; i < argc; i++)          // stampa tutti gli argomenti
        printf("argv[%d]: %s\n", i, argv[i]);

    a = atoi(argv[1]);                  // conversione a intero del primo argomento
    b = atoi(argv[2]);                  // conversione a intero del secondo argomento
    printf("La somma e' %d \n", (a+b));

    return 0;
}
```

All'interno del programma, gli argomenti del main possono essere usati come qualsiasi altro argomento di una normale funzione.

Variabili di ambiente

- All'interno della shell di Unix è possibile definire delle variabili
- Stringhe con formato `nome=valore`

```
$ env
```

```
$ echo $HOME
```

- Si chiamano variabili di ambiente (*environment variables*) perché vengono usate per determinare il comportamento del sistema durante la sessione corrente della shell
- Possibile definire variabili personalizzate, e.g.:

```
$ MYVAR=3
$ echo $MYVAR
3
```
- Possono essere utilizzate dai programmi eseguiti dalla shell

Il comando **env** mostra tutte le variabili d'ambiente definite nella sessione corrente della shell

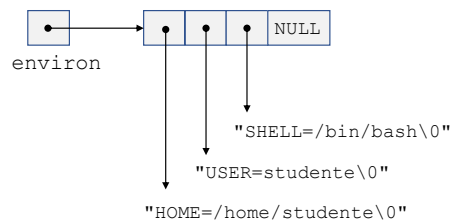
Usando **echo \$HOME** otteniamo il valore della variabile d'ambiente HOME, se definita, ovvero il percorso della directory HOME dell'utente corrente, per esempio `"/home/studente"`

Ci sono un certo numero di variabili predefinite che determinano il comportamento del sistema. Esempio: eseguire il comando `"cd"` nella shell è equivalente ad eseguire il comando `"cd $HOME"`, ovvero `cd /home/studente`. Possiamo però cambiare il valore della variabile HOME, per esempio `HOME=/home/studente/Desktop`. In questo caso se eseguiamo `"cd"`, ci spostiamo nella directory `/home/studente/Desktop`

Perché definire nuove variabili? Tutti i programmi eseguiti nella sessione corrente della shell possono usare quei valori (possiamo vederlo anche come un modo di passare dei parametri al programma, alternativo a usare `argc/argv` del main).

Ottenere variabili di ambiente

- Puntatore globale `char* environ[]`
- Accesso tramite funzioni di libreria definite da ISO C



```
#include <stdlib.h>
char* getenv(const char *name);
```

Nome della variabile da ottenere

Stringa contenente il valore della variabile
NULL se variabile non esiste

Dal punto di vista del programma, le variabili d'ambiente sono immagazzinate dentro un array chiamato **environ**, il cui funzionamento è del tutto simile all'array `argv`. Ogni elemento di `environ` è una stringa che contiene una coppia del tipo **nome=valore** che specifica una variabile d'ambiente.

Non si usa `environ` direttamente nel programma, ma vi si accede tramite delle funzioni di libreria.

La funzione `getenv()` permette di ottenere il valore della variabile di ambiente (ovvero ciò che sta a destra del simbolo '='). Per esempio, considerando l'array `environ` in alto a destra e invocando `getenv("HOME")` otterrei `"/home/studente"`

Impostare variabili di ambiente

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

```
int putenv(const char* name);
```

0 se ok
!=0 se errore

coppia *nome=valore*

Quando name esiste:
- Se 1, aggiorna il valore
- Se 0, non aggiorna il valore

```
int setenv(const char* name, const char* value, int rewrite);
```

0 se ok
-1 se errore

Nome della variabile
da assegnare

valore da assegnare

```
int unsetenv(const char* name);
```

0 se ok
-1 se errore

Nome della variabile da eliminare

- Influenzano solo il processo attualmente in esecuzione, ed eventuali processi figli

putenv e setenv sono due modi alternativi di impostare il valore di una variabile d'ambiente.

Al termine del programma, comunque, le variabili di ambiente vengono reimpostate ai valori che avevano all'inizio del programma.

Lancio di un processo

Chi invoca la funzione `main`?

- Quando si vuole eseguire un programma, il kernel invoca una funzione `exec` che:
 - Prende come argomenti il file eseguibile, i parametri da linea di comando e le variabili di ambiente
 - Il processo esegue le istruzioni contenute nel file eseguibile a partire dall'**entry point**
- Entry point
 - Specificato nell'intestazione del file eseguibile creato dal compilatore/linker
 - Indica l'indirizzo di memoria della prima istruzione assembler da eseguire
 - Quale istruzione?

Quando diamo il comando di esecuzione dalla shell, il kernel crea un nuovo processo – tramite una funzione `fork()` – e successivamente invoca una funzione di nome `exec` (che vedremo nel dettaglio più avanti).

L'entry point è letteralmente il punto di ingresso al programma, ovvero la prima istruzione che verrà eseguita - non è (ancora) l'esecuzione della funzione `main`.

Lancio di un processo

```
$ gcc helloworld.c -o helloworld
$ readelf -h helloworld
...
```

```
ELF Header:
[...]
Entry point address: 0x1050
Start of program headers: 64 (bytes into file)
Start of section headers: 14688 (bytes into file)
Flags: 0x0
Size of this header: 64 (bytes)
Size of program headers: 56 (bytes)
[...]
```

```
$ objdump -d helloworld
...
```

```
[...]
Disassembly of section .text:

00000000000001050 <start>:
1050: 31 ed                xor    %ebp,%ebp
1052: 49 89 dl            mov    %rdx,%r9
[...]
106d: 48 8d 3d c1 00 00 00 lea     0xc1(%rip),%rdi
                        # 1135 <main>
1074: ff 15 66 2f 00 00    callq *0x2f66(%rip)
                        # 3fe0 <__libc_start_main@GLIBC_2.2.5>
[...]
```

ELF = Executable and Linking Format. Rappresenta il formato in cui vengono scritti i file eseguibili in Unix.

Il comando shell "readelf" ci permette di esaminare il contenuto (che non sarebbe altrimenti leggibile, essendo scritto in linguaggio macchina). La prima parte del file ELF contiene un **header** (intestazione) che dà informazioni su come interpretare il file eseguibile stesso, tra cui l'indirizzo dell'entry point, scritto in esadecimale (prefisso 0x).

"readelf -h" mostra solamente l'header del file.

Di fatto, la funzione exec invocata dal kernel legge l'header per identificare l'entry point ed esegue l'istruzione che si trova a quell'indirizzo.


Il comando "objdump" mostra le istruzioni (in linguaggio assembler) del programma, con il loro indirizzo. All'indirizzo specificato dall'entry point troviamo una etichetta **_start**, che equivale all'inizio del corpo di una funzione che si chiama **_start** e aggiunta dal linker gcc. La funzione **_start** invoca una funzione chiamata **__libc_start_main**.

Anche **__libc_start_main** viene aggiunta dal linker e, come dice il nome, sarà la funzione che invocherà finalmente la funzione main del nostro programma.

Lancio di un processo

```
int __libc_start_main(  
    int (*main) (int, char **, char **),  
    int argc,  
    char **argv,  
    void (*init) (void),  
    void (*fini) (void),  
    void (*rtld_fini) (void),  
    void *stack_end  
) {  
    ...  
    int ret = main(argc, argv);  
    ...  
}
```

indirizzo della funzione `main`
argomenti da linea di comando
lista argomenti da linea di comando
Indirizzo della funzione `init`
Indirizzo della funzione `fini`
Indirizzo della funzione `rtld_fini`
Indirizzo di fine dello stack

 `___libc_start_main` invoca la funzione `main` del programma

Il corpo della funzione `__libc_start_main` esegue l'istruzione `main(argc,argv)`. Al termine dell'esecuzione della funzione `main`, alla variabile `ret` viene assegnato il valore di ritorno della funzione `main`, ovvero ciò che viene specificato come argomento dell'istruzione **return**.

Terminazione di un processo

Otto modalità di terminazione

- Return dal `main`
- Chiamata alla funzione `exit()`
- Chiamata alla system call `_exit()` o `_Exit()`
- Return dalla funzione di partenza dell'ultimo thread
- Chiamata alla funzione `pthread_exit()` dall'ultimo thread
- Chiamata alla funzione `abort()`
- Ricezione di un segnale
- Risposta dall'ultimo thread a una richiesta di cancellazione

Terminazione normale

Terminazione anormale

Attenzione: terminazione normale non significa che se il programma termina con uno di questi 5 modi non ci siano stati errori logici nel programma!

Per esempio, nei nostri programmi quando abbiamo un errore (ad esempio non riesco ad aprire un file), invochiamo la funzione `exit`.

Semplicemente terminare il processo con uno di questi modi permette al kernel di effettuare una serie di operazioni che rientrano nel normale ciclo di vita del processo. Questo non accade con gli altri 3 modi di terminazione anormale.

return

- Il processo termina quando si incontra l'istruzione `return` nel `main`
- Il controllo ritorna alla funzione `__libc_start_main`
- `__libc_start_main` invoca la funzione `exit`

```
int __libc_start_main( ... ) {  
    ...  
    int ret = main(argc, argv);  
    exit(ret);  
    ...  
}  
  
#include <stdio.h>  
int main(int argc, char *argv[]) {  
    printf("Hello, world!\n");  
    return 0;  
}
```

A dashed blue line illustrates the control flow. It starts at the `return 0;` statement in the `main` function, moves left and up to the `int ret = main(argc, argv);` line in the `__libc_start_main` function, then continues left and down to the `exit(ret);` line in the same function. This visualizes how the return value from `main` is passed to `exit`.

`return` restituisce il controllo alla funzione chiamante, in questo caso la funzione `__libc_start_main`, la quale non fa altro che invocare la funzione `exit` passandole come argomento il valore restituito dal `main`.

exit()

```
#include <stdlib.h>

void exit(int status);
```

- `$ man 3 exit`
- Funzione di libreria che:
 - effettua operazioni di cleanup, e.g. pulizia dei buffer di I/O
 - invoca `fclose()` sugli stream aperti
 - invoca le funzioni *exit handler*, se definite
 - restituisce il controllo al kernel
 - invocando la system call `_exit()`
- Ha come argomento un codice di stato che viene passato al kernel

Exit status

- L'intero passato come argomento alla `exit()` definisce l'**exit status** del processo
- La shell può esaminare l'exit status di un processo
 - `echo $?`
- Quando il `main` termina con `return` esplicito, l'intero restituito è l'argomento della `exit()`
 - Nel `main`, `return 0` è equivalente a `exit(0)`
- Exit status non è definito se:
 - Non viene passato alcun argomento alla funzione `exit()`
 - Il `main` termina senza l'istruzione `return`
 - Il `main` non è dichiarato per restituire un intero
 - `void main(int argc, char* argv[]) { ... }`

L'exit status è un codice utile a far capire a chi esegue il programma se questo è terminato con successo oppure no. Quando il programma viene lanciato dalla shell, dopo la sua terminazione possiamo eseguire il comando "`echo $?`" che stampa su standard output l'exit status.

Se exit status non viene definito, il risultato del comando "`echo $?`" sarà indefinito.

Exit status: esempio

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

```
$ gcc helloworld.c -o helloworld
$ ./helloworld
Hello, world!
$ echo $?
0
```

Exit status = 0

```
#include <stdio.h>

void main(int argc, char *argv[]) {
    printf("Hello, world!\n");
}
```

```
$ gcc helloworld.c -o helloworld
$ ./helloworld
Hello, world!
$ echo $?
183
```

Exit status casuale

Exit handler

```
#include <stdlib.h>

int atexit(void (*func) (void));
```

- `$ man 3 atexit`
- Funzione che registra un nuovo **exit handler**, prendendo come argomento l'indirizzo di una funzione
- Quella funzione verrà invocata automaticamente dalla `exit()`
- La funzione registrata non prende alcun parametro e non restituisce alcun valore
- Possibile registrare fino a 32 exit handler
 - Invocati in ordine inverso rispetto alla registrazione
- `atexit` restituisce un numero diverso da 0 in caso di errore

Una delle azioni intraprese da `exit()` è invocare gli exit handler, ovvero funzioni definite dal programmatore usate per compiere delle azioni personalizzate quando il programma termina, ad esempio scrivere a video un messaggio o deallocare della memoria.

Se definite, queste funzioni vengono invocate **automaticamente** dalla `exit`, senza dover invocarle manualmente nel `main`. L'unica cosa da fare è **registrare** gli exit handler, ovvero far sapere al programma che la funzione `exit` dovrà invocare quelle funzioni. La registrazione si effettua con la funzione `atexit()`.

Nell'exit handler è possibile eseguire il codice che vogliamo, con l'unico vincolo che tale funzione non abbia alcun parametro e non abbia alcun ritorno, esempio:

```
void my_exit_handler()
{
    ...
}
```

Exit handler: esempio

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

int i;

void my_exit1(void) {
    printf("first exit handler: %d\n", i);
    i++;
}

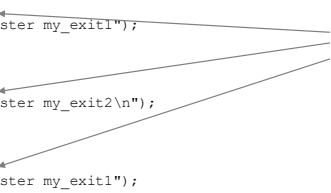
void my_exit2(void) {
    printf("second exit handler: %d\n", i);
}

int main(int argc, char *argv[]) {
    i = 0;
    if (atexit(my_exit1) != 0) {
        fprintf(stderr, "can't register my_exit1\n");
        exit(1);
    }

    if (atexit(my_exit2) != 0) {
        fprintf(stderr, "can't register my_exit2\n");
        exit(1);
    }

    if (atexit(my_exit1) != 0) {
        fprintf(stderr, "can't register my_exit1\n");
        exit(1);
    }
    printf("main has done\n");
    return(0);
}
```

Registrazione exit handler



L'output di questo programma sarà:

main has done
first exit handler
second exit handler
first exit handler

`_Exit()` / `_exit()`

```
#include <stdlib.h>

void _Exit(int status);
```

```
#include <unistd.h>

void _exit(int status);
```

- Sinonimi
 - `_Exit()` → ISO C99
 - `_exit()` → POSIX.1
- `$ man 2 exit`
- System calls che restituiscono immediatamente il controllo al kernel
 - Non invocano alcun exit handler
 - Non effettuano alcuna operazione di pulizia

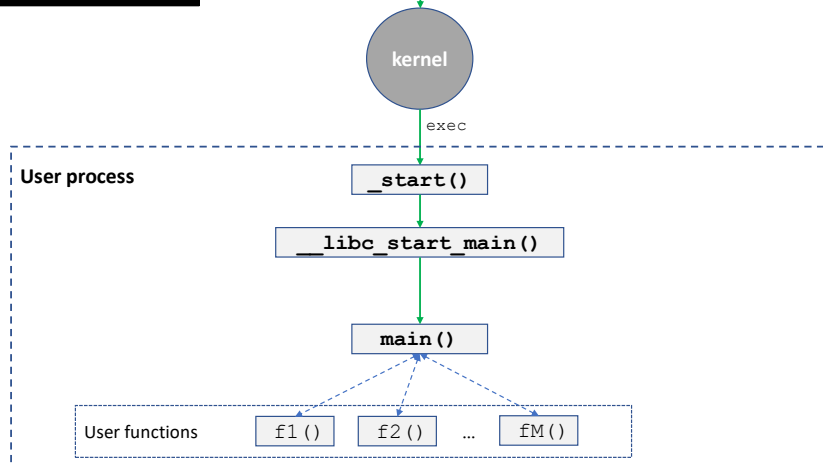
Mentre `exit()` è una funzione di libreria, queste due sono **system call**. Infatti sono le funzioni che interagiscono con il kernel per restituirgli il controllo.

Queste due system call sono equivalenti, definite da due standard diversi (includono da due header diversi)

In linea con le differenze tra system call e funzioni di libreria fanno un'operazione molto più "grezza" della `exit`, ovvero restituiscono il controllo al kernel senza effettuare alcuna operazione di pulizia o invocare exit handler.

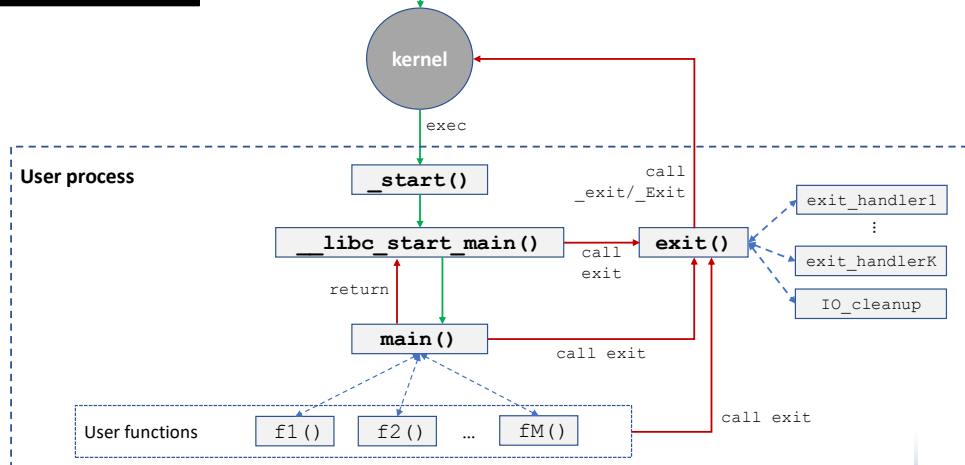
Ciclo di vita di un processo (1/3)

```
./my_program arg1 ... argN
```

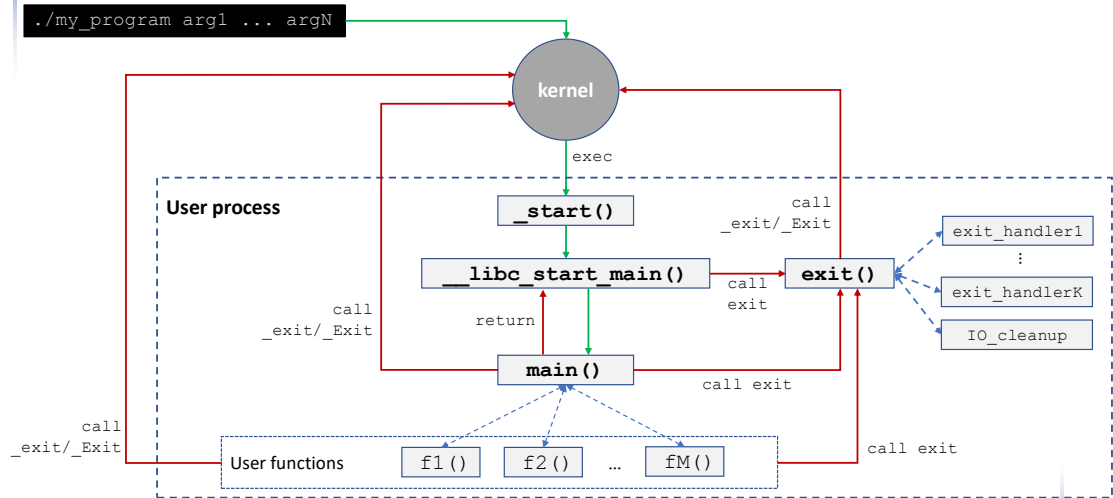


Ciclo di vita di un processo (2/3)

```
./my_program arg1 ... argN
```

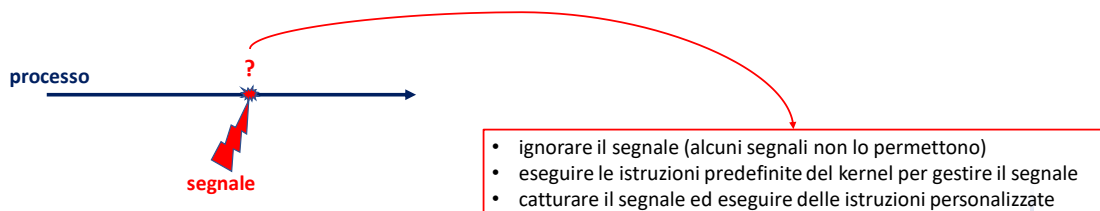


Ciclo di vita di un processo (3/3)



Segnali

- All'interno del sistema possono verificarsi eventi asincroni
 - Scatta un timer impostato in precedenza
 - Un processo ha terminato la sua esecuzione
 - Un processo si è interrotto a causa di un errore
 - Accesso a indirizzi di memoria non validi
- I segnali notificano gli eventi asincroni ai processi in esecuzione
 - Interruzioni software



Il ciclo di vita dei processi

Ing. Giovanni Nardini - University of Pisa - All rights reserved

153

Mentre studiate questa slide possono capitare **eventi asincroni**, cioè eventi che non vi aspettavate e che capitano in un momento imprevedibile.

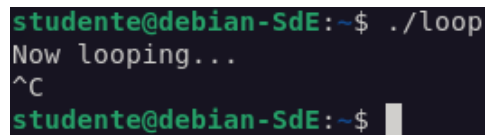
- Vi arriva un messaggio sul cellulare
- Entra qualcuno nella stanza
- Va via la corrente

Alcuni di questi eventi potete **ignorarli** (ad esempio il messaggio potete non leggerlo) mentre altri non potete ignorarli, anzi dovete **interrompere** quello che stavate facendo e darvi da fare per gestire questi eventi in qualche maniera (ad esempio se va via la corrente).

Anche un processo che sta eseguendo si può trovare ad affrontare eventi asincroni e a ignorarli/gestirli in qualche maniera

Esempio

```
#include <stdio.h>
int main()
{
    printf("Now looping...\n");
    while(1);
    printf("You cannot be here!\n");
    return 0;
}
```



```
studente@debian-SdE:~$ ./loop
Now looping...
^C
studente@debian-SdE:~$
```

Premere CTRL+C sul terminale provoca l'invio del segnale `SIGINT`, che fa terminare il programma

I segnali erano uno dei modi anormali di terminazione del programma, perché molti di essi hanno come comportamento predefinito quello di causare la terminazione del processo.

Alcuni segnali

- Ogni segnale è identificato da un nome e un numero intero
- Costanti definite da `<signal.h>`
- Sottoinsieme dei segnali definiti da Unix:

| Segnale | Causa | Azione di default |
|---------|--|--|
| SIGABRT | Processo ha invocato la funzione <code>abort()</code> | Il processo termina |
| SIGALRM | Un timer impostato con la funzione <code>alarm()</code> è scattato | Il processo termina |
| SIGCHLD | Un processo figlio è terminato | Il segnale viene ignorato |
| SIGINT | Il terminale ha inviato il comando CTRL+C | Il processo termina |
| SIGKILL | Il kernel vuole terminare il processo | Il processo termina (non può essere catturato né ignorato) |
| SIGSEGV | Accesso a memoria non valida (segmentation fault) | Il processo termina |
| SIGTERM | È stato eseguito il comando <code>kill</code> dalla shell | Il processo termina |
| ... | ... | ... |

Tutti i segnali sono elencati nella pagina del manuale **man signal**.

SIGCHLD è il segnale inviato al processo genitore quando il figlio termine, è quello che sveglia il processo genitore se in attesa sulla `wait`

SIGTERM = simile a SIGKILL, ma da' la possibilità di catturare il segnale e di terminare in maniera più delicata

Inviare segnali

- È possibile inviare un segnale a un processo direttamente dalla shell

```
$ kill -s <signal> <pid>
```

- `pid` è l'identificatore del processo a cui inviare il segnale
- L'opzione `-s` specifica quale segnale inviare, tra quelli definiti da `<signal.h>`
- Esempio:

```
$ kill -s SIGINT 9415    oppure    $ kill -INT 9415
```

- Usato spesso per terminare processi che non rispondono:

```
$ kill -9 9415
```

- `-9` è il numero corrispondente al segnale `SIGKILL`



man kill

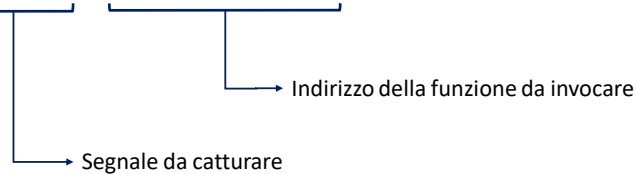
I segnali possono essere inviati esplicitamente a un processo dalla shell, tramite il comando "kill"

Devo conoscere il nome o il numero del segnale, e il PID del processo a cui voglio inviare il segnale (lo posso recuperare con il comando "ps" o "top")

Catturare i segnali

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

```
void (*signal(int signo, void (*func)(int))) (int);
```



- La system call `signal` permette di registrare un handler per il segnale specificato
 - Quando arriva il segnale, viene invocata la funzione specificata come secondo argomento
 - Concetto simile agli exit handler
- Restituisce
 - l'indirizzo della funzione da invocare se ok
 - `SIG_ERR` in caso di errore (setta `errno`)

Possibile specificare dei **signal handler**. È un concetto simile agli exit handler. Il programmatore definisce una funzione personalizzata che compie delle azioni, e come negli exit handler tale funzione deve essere **registrata**, ovvero si deve far sapere al programma che, nel caso dovesse ricevere un certo segnale, dovrà eseguire quella specifica funzione.

Catturare i segnali

```
#include <signal.h>

void (*signal(int signo, void (*func)(int))) (int);
```

- L'handler `func` deve essere una funzione che:
 - Prende come argomento un intero (il numero del segnale)
 - Restituisce void
- Può anche essere:
 - `SIG_IGN` → permette di ignorare il segnale
 - Alcuni segnali non possono essere ignorati
 - `SIG_DFL` → permette di accettare il comportamento predefinito del kernel per il segnale

Il signal handler può essere implementato a seconda delle necessità, ma deve avere un argomento di tipo intero e non restituire alcun valore. L'argomento intero conterrà il numero del segnale che ha generato l'invocazione della funzione.

```
void my_signal_handler(int signo)
{
    ...
}
```

Catturare i segnali: esempio

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

void sigint_handler(int signo) {
    printf("\nCatturato SIGINT! Arrivederci!\n");
}

int main()
{
    if (signal(SIGINT, sigint_handler) == SIG_ERR) {
        fprintf(stderr, "registrazione handler SIGINT non riuscita: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
    printf("Now looping...\n");
    while (1);
    printf("You cannot be here!\n");
    return 0;
}
```

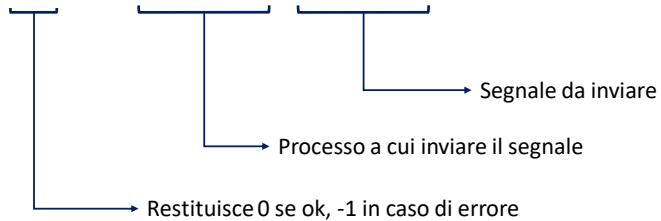
Definizione dell'handler

Registrazione di un handler per il segnale SIGINT

Inviare segnali

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

```
int kill(pid_t pid, int signo);
```



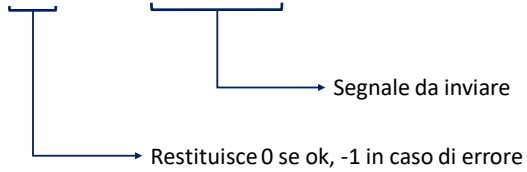
- Permette al processo di inviare un segnale a un altro processo
 - e.g. un processo può inviare il segnale di terminazione a un processo figlio

I segnali possono anche essere inviati da un processo a un altro processo tramite la system call **kill**

Inviare segnali

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

```
int raise(int signo);
```

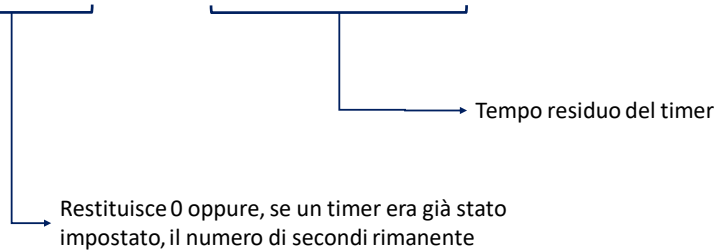


- Permette al processo di inviare un segnale a sé stesso
- Equivalente a:
 - `kill(getpid(), signo);`

Impostare un timer

```
#include <unistd.h>
```

```
unsigned int alarm(unsigned int second);
```



- Genera il segnale `SIGALRM` dopo il tempo specificato
- Se non viene catturato, il processo termina

Abortire il processo

```
#include <stdlib.h>

void abort();
```

- Il processo invia il segnale `SIGABRT` a sé stesso
 - Equivalente a `raise(SIGABRT);`
- Fa terminare il processo in modo anormale
- Può essere catturato per effettuare delle operazioni finali prima della terminazione
- Differenze con `exit()` ?

Era uno dei modi anormali di terminazione

Di default, la `abort` non fa operazioni di cleanup e non invoca gli exit handler → simile a `_exit`, con la differenza che il segnale `SIGABRT` potrebbe essere catturato e gestito con un signal handler.