GNU/Linux System Calls

dr. Andrea E. Naimoli

Università degli Studi di Trento

Dipartimento di Ingegneria e Scienza dell'Informazione

via Sommarive 14

I - 38050 Trento - Povo, Italy

L'architettura del sistema (1/2)

- L'elemento di base di un sistema Unix-like è il nucleo del sistema operativo ovvero il *kernel*
- Al kernel si demanda la gestione delle risorse essenziali
  (CPU, memoria, periferiche, ecc...)
- Al boot il kernel verifica lo stato delle periferiche, monta la prima partizione in *read-only* e lancia il primo programma (**/sbin/init**)
- Tutto il resto, come l'interazione con l'utente, viene realizzato con programmi eseguiti dal kernel tramite **init**

L'architettura del sistema (2/2)

- I programmi utilizzati dall'utente accedono alle periferiche chiedendo al kernel di farlo per loro.
- I kernel Unix-like sfruttano alcune caratteristiche intrinseche ai processori come la gestione hardware della memoria virtuale e la modalità protetta
- Solo il kernel è eseguito in modalità privilegiata, con il completo accesso all'hardware
- Tutti gli altri programmi sono eseguiti in modalità protetta

User e kernel space

- Due sono i concetti fondamentali su cui si basa l'architettura dei sistemi Unix-like:
  - **User space**: Ambiente in cui sono eseguiti i programmi
  - **Kernel space**: Ambiente in cui viene eseguito il kernel
- Ogni programma vede se stesso come unico possessore della CPU
- Per questo fatto, non è possibile ad un singolo programma disturbare l'azione degli altri
- Da qui deriva la stabilità dei sistemi Unix-like

Chiamate di sistema (1/2)

- Le interfacce con cui i programmi accedono all'hardware si chiamano **system call**

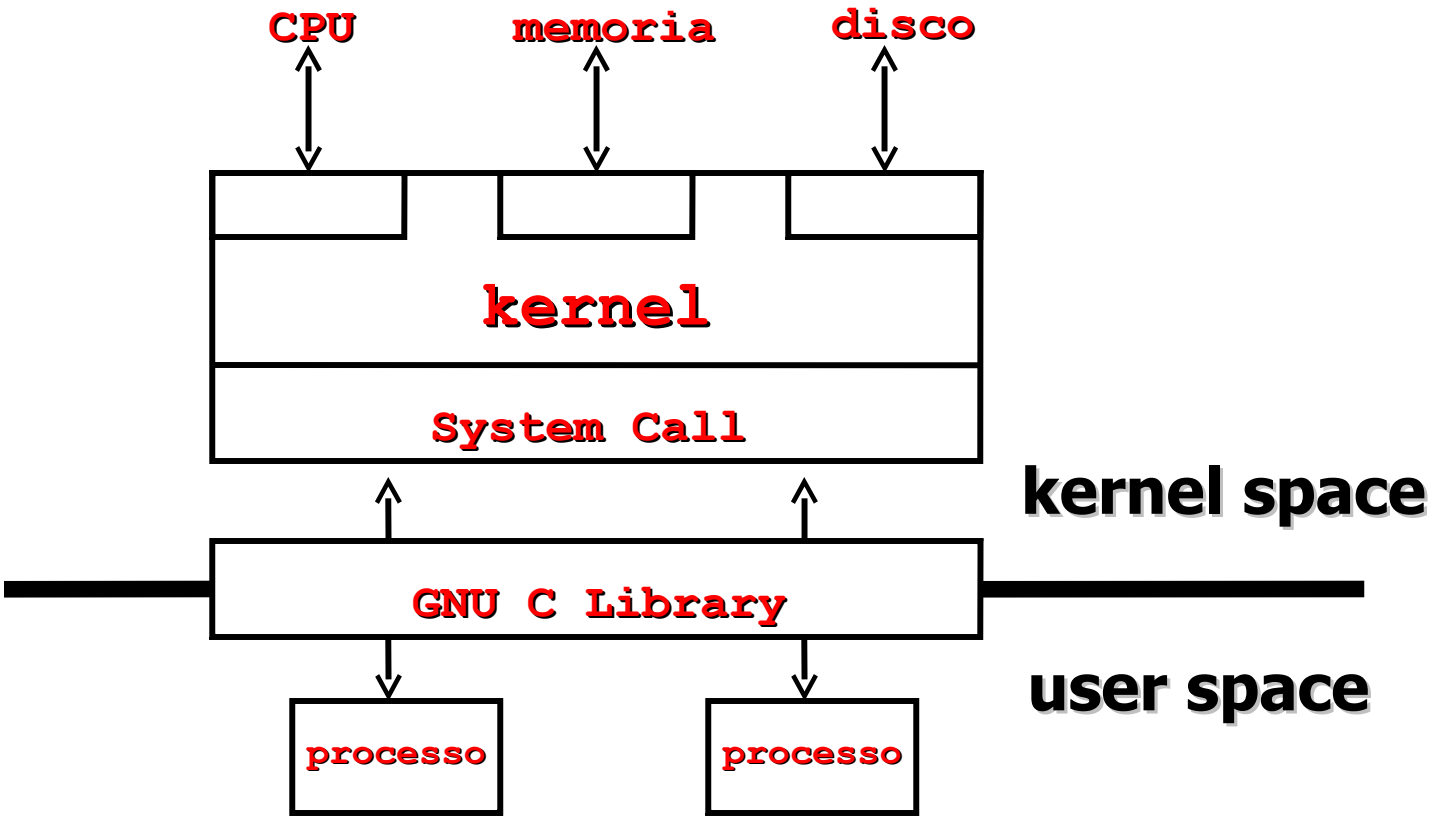
- Il kernel le esegue nel *kernel space* e ritorna i risultati al programma chiamante che invece opera in *user space*

- Infatti, utilizzando il comando di shell **ldd** su di un eseguibile si possono visualizzare le librerie condivise caricate e, fra queste, vi sono sicuramente anche **ld-linux.so**, e **libc.so.**

- **ld-linux.so:** *when a program is loaded, the OS passes control to ld-linux.so instead of normal entry point of the application. ld-linux.so searches for and loads the unresolved libraries, and then it passes control to the application starting point.*

- ***libc.so*** *: The GNU C Library, commonly known as glibc, is the C standard library released by the GNU Project.*

Chiamate di sistema – Linux (2/2)



System Calls

- Esempi di system calls:
- Processi : **`getpid()`, `wait()`, `fork()`, `execve()`...**
- Segnali : **`kill()`, `pause()`, `alarm()`...**
- File : **`open()`, `close()`, `read()`, `write()`, `dup()`...**
- File System : **`mount()`, `umount()`, `chdir()`...**
- Permessi : **`chmod()`, `chown()`...**
- Tempo : **`time()`...**

System call: **exit**

```
#include <stdlib.h>

void exit(int status)
```

- Funzione usata per effettuare un'uscita *"normale"* da un programma (**`man 3 exit`**)
- Tutti i file descriptor associati al processo vengono chiusi

System call: **exit**

- Il valore di uscita di **main** (i.e., **exit status**) viene passato al processo che aveva lanciato il programma (in genere la shell)
- Questo valore fornisce indicazioni sulla riuscita o fallimento del programma
- Il valore di ritorno è compreso fra 0 e 255: 0 in caso di successo, >0 in caso di fallimento
- Esempio, con la shell:
  - **ls ./ &**
  - **ls ./something &** (usare un nome inesistente)
  - utilizzo di **echo $?**

System call: errori

```
#include <stdio.h>
void perror(const char *s);
```

- In caso di errore, le system call ritornano **sempre** -1
- *perror* permette di interpretare gli errori dovuti alle system call
- Manda sullo standard output la stringa referenziata da ***s** seguita da **:** e *da una stringa* che descrive il tipo di errore

System call: errori

```
#include <stdio.h>
#include <errno.h>
//La fopen restituisce lo stream oppure NULL se fallisce
int main()
{
    FILE *fp;
    if ((fp=fopen("nome_file","r"))==NULL)
        perror("nome_file");
    else
        fclose(fp);
}
```

Output in caso di errore:

```
nome_file: No such file or directory
```

perror - errno

- Come funziona **perror**?
- Le system call scrivono il tipo di errore avvenuto in una variabile intera globale: **errno**
- C'e' un array di stringhe, globale, che associa a ogni valore di errno un messaggio: **sys_errlist[]**
- **perror** stampa **sys_errlist[errno]**
- Note:
  - ✓ TUTTI possono assegnare un valore a **errno**!
  - ✓ Per ottenere il messaggio di errore corrispondente c'e' una funzione apposita: **strerror(errno)**

Gestione dei processi

- L'architettura della gestione dei processi
- Ottenere informazioni sui processi
- Creare nuovi processi
- Mandare in esecuzione un programma
- Sincronizzazione fra processi
- Una semplice shell

L’architettura della gestione dei processi

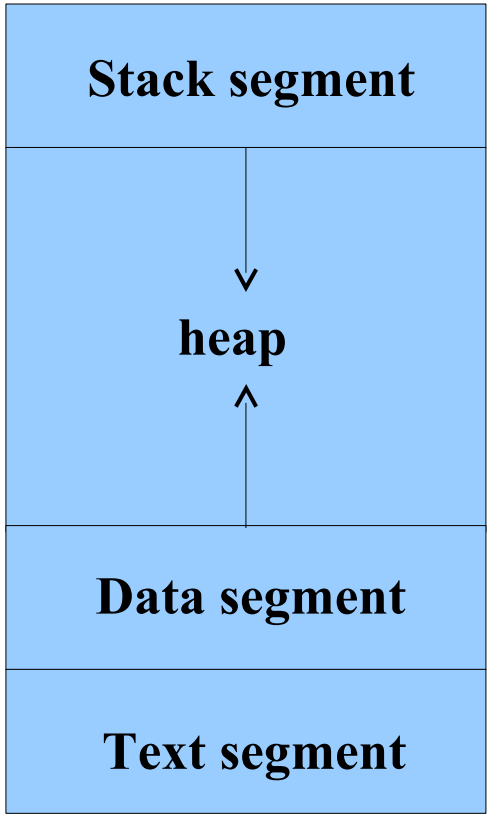
- In Unix qualunque processo può generare altri processi detti *child process*
- Ciascuno ha un proprio identificatore, il *PID* (Process IDentifier), assegnato in maniera (generalmente) progressiva
- E' importante ricordare che ogni processo è sempre stato generato da un altro processo (*parent process*)
- **/sbin/init** è l’unico processo che sfugge a questa regola; infatti ha sempre pid 1 e non ha padre (provare a lanciare **ps ufax* | grep init**)

  * ax = all in the system (not only curren shell), f = hierarchy, u = owner user

Struttura processi in Linux

- I processi in Linux hanno la memoria divisa in quattro segmenti:

- **Text segment** (program code)
- **Data segment** (variabili)
- Lo **heap**: estensione del segmento dati per l'allocazione dinamica memoria
- **Stack segment**: variabili automatiche



Ottenere informazioni sui processi

- Ogni processo ha un unico *process ID*; è un tipo di dato standard, il **pid_t**, che in genere è un intero

int getpid();

- Tutti i processi memorizzano anche il *pid* del genitore da cui sono stati creati, il *parent process ID* (PPID)

int getppid();

- L'utente per cui un processo esegue è identificato dallo user ID

int getuid();

Creare nuovi processi

Eseguire un programma (1/2)

```
#include <stdlib.h>
int system (char *cmd);
```

- Il processo corrente crea un processo figlio
- **system** esegue la *Bourne shell* **/bin/sh**
- La shell esegue **cmd**
- La shell termina
- Il processo padre continua da dove aveva interrotto

Eseguire un programma (2/2)

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

int main()
{
      printf("Adesso faccio un ls -l\n\n");
      system("ls -l");
      printf("\nEcco fatto!\n\n");
}
```

Esercizio per casa

- Realizzare una prima semplicissima shell utilizzando la **system**
- La shell deve:
  - ✓ Mostrare un prompt all'utente
  - ✓ Eseguire i comandi che l'utente inserisce (ad esempio potete usare la funzione **getline**)
  - ✓ Ripetere i passi 1 e 2 fintantoché il comando inserito non è **quit**

```
#include <stdio.h>
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

Esempio:

```
getline(&line, (size_t*)&len, stdin);
```

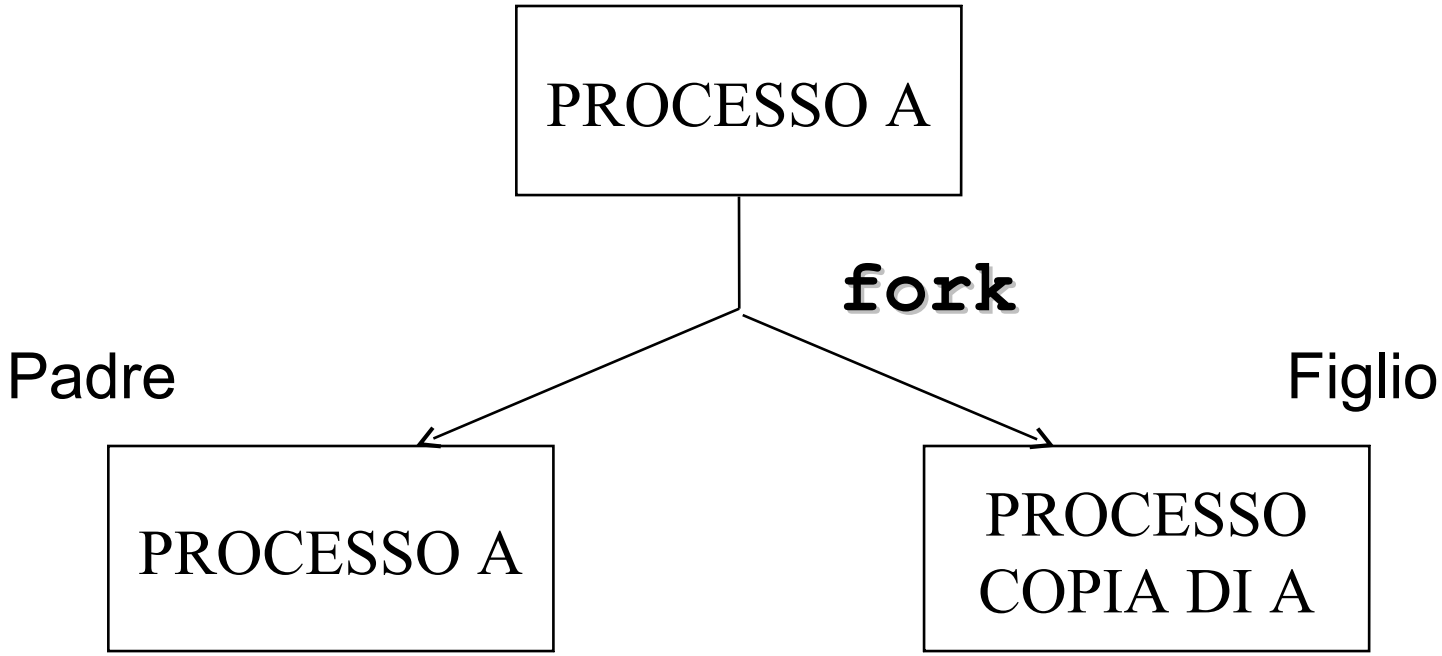
[size_t = unsigned integer type]

fork() (1/2)

```
#include <unistd.h>
int fork();
```

- Dopo la sua esecuzione il processo padre e figlio sono una copia identica, a meno del PID
- Il processo figlio riceve una copia del segmento di testo, dati e stack (la memoria è copiata e **non condivisa**)
- Al padre viene ritornato il pid del figlio, al figlio ritorna sempre 0. La funziona ritorna DUE volte.
  - ✓ Il figlio puo' sempre recuperare il pid del padre (getppid) mentre il padre lo recupera dalla fork

fork() (2/2)



Esercizio (10 min.)

- Scrivere un programma che crei

  f figli (distinti dal processo padre)

  per **g** generazioni (**f** per figlio – distinti)

- Ogni processo così creato deve stampare a video

  “Io sono x della generazione y”

  (considerando il processo padre di generazione 0, i figli 1, ed i nipoti 2, etc...)

Soluzione esercizio

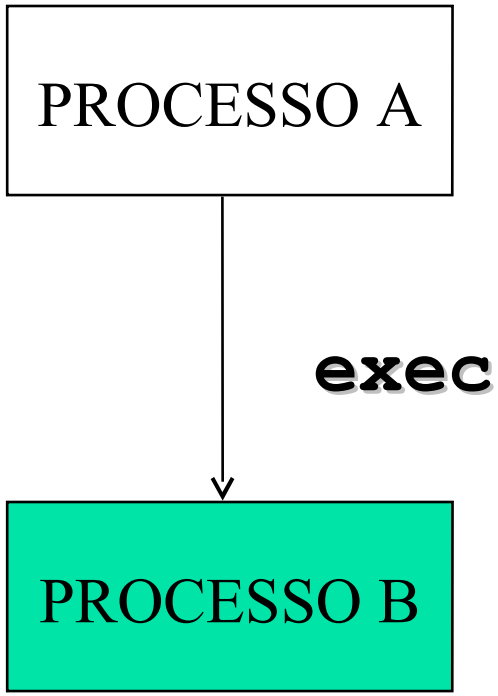
```
// alb_gen.c

#include <stdio.h>

int main(void)
{
  int F = 2;
  int G = 2;

  int f = 0;
  int g = 0;

  while(f < F && g < G)
  {
    //printf("%i, %i\n", f, g);

    if ( fork() == 0 )
    {
      // Il figlio resetta f, e incrementa la generazione

      f = 0;
      g++;
    }
    else
    {
      // Il padre deve solo contare i figli che genera

      f++;
    }
  }

  printf("Io sono %d della generazione %d \n", getpid(), g);
  return 0;
}
```

Limitare il fork: ulimit (bash)

- Spesso vogliamo limitare le risorse (CPU, memoria, numero di processi ecc.) degli utenti
- Talvolta commettiamo errori e l'uso delle risorse sfugge di mano (es. fork infinito)
- Come regolare le risorse? **ulimit** (bash)
- Esempi:
  - **ulimit -a** (mostra le risorse disponibili e le limitazioni)
  - **ulimit -u 100** (permetti al piu' 100 processi all'utente)
- **ulimit** si basa su funzioni C di basso livello:
  - **setrlimit** , **getrlimit** (vedi man page)

exec(): mutazione

- Il process ID rimane invariato
- exec sostituisce text, data, heap, e stack con il codice letto dal disco



fork() + exec()

- L'utilizzo della chiamata *system()* si dimostra utile per piccole e limitate applicazioni
- Una alternativa a **system()** è rappresentato dall'uso combinato di **fork()** e **exec()**
- **fork()** consente di creare copie di un processo; **exec()** di sostituire l'immagine di un processo con un "nostro" eseguibile

exec - execlp()

int execlp(char *file, char* arg0, char* arg1, ..., char* argn, NULL);

- **file** è il nome del programma da eseguire. Viene cercato nel PATH
- **argX** sono gli argomenti con cui viene chiamato il programma (**arg0** è per convenzione il nome del programma stesso)
- Ritorna SOLO in caso di errore (-1)

execlp() – Esempio

```
// specchio-main.c
#include <stdio.h>
#include <unistd.h>

int main()
{
        printf("Specchio delle mie brame\nChi è la più bella del reame?\n");
        execlp("./specchio-magic“, NULL, NULL);
        printf("Voi, o Regina!\n");
}
```

```
// specchio-magic.c
#include <stdio.h>

int main()
{
      printf("Biancaneve!!\n");
}
```

Altre versioni di **exec()**

- Mettono gli argomenti in un array e non in una lista
- Cercano o meno il programma nel PATH
- Passano in modo esplicito un puntatore a environment

wait e waitpid (1/3)

- Uno degli usi più comuni nei server è avere un programma daemon il quale accetta più connessioni e attende la loro terminazione

- A questo punto si aprono tre casi:
  - ✓ Il padre attende la terminazione regolare del figlio e cattura lo stato di uscita mediante le funzioni **wait** e **waitpid**
  - ✓ Il padre termina prima del figlio. L' "orfano" viene adottato da **init** il quale si occupa della sua terminazione
  - ✓ Il figlio termina ma il padre non è in grado di raccogliere lo stato di uscita (il kernel mantiene alcune info. in memoria)

- Questo ultimo è il caso in cui si generano processi *zombie*

wait e waitpid (2/3)

int wait(int *statusp);

- **wait** non ritorna fintantoché un qualsiasi figlio non termina (non posso specificare quale)
- Ritorna il pid del figlio che ha terminato, ritorna -1 senza aspettare se non ci sono figli da aspettare
- Nell'intero puntato da **statusp** viene messo lo stato di exit del figlio
- La variante **waitpid** accetta anche il parametro del child pid da attendere
- Se si specifica -1 attende il PRIMO figlio in uscita, qualunque esso sia

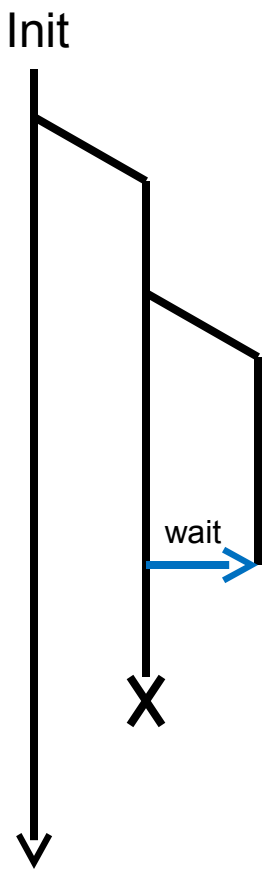
wait e waitpid (3/3)

int waitpid(int pid, int *statusp, int options);

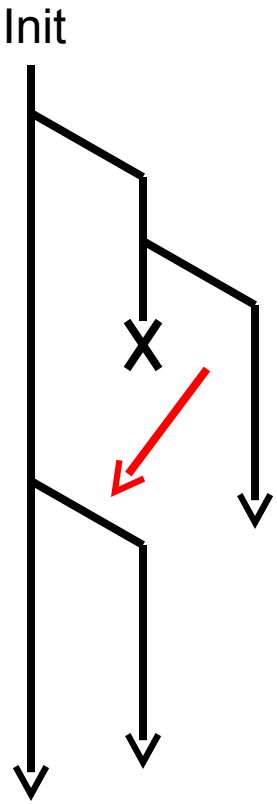
- La variante waitpid accetta come primo parametro il pid del *child* da attendere
- Se si specifica -1 come primo parametro la waitpid attende il PRIMO figlio in uscita, qualunque esso sia

- waitpid(-1, &status, 0) e' equivalente a wait(&status)

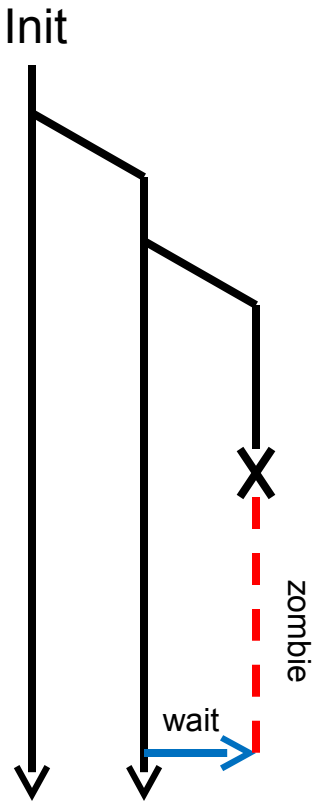
La vita dei processi



Normal execution



Parent terminates before the child



Child terminates before the parent

Scheduling and nice

- I processi sono schedulati in modo indipendente: il figlio puo' essere schedulato prima del padre o vicerversa.
- In Linux c'e' la possibilita` di specificare la priorita` dei processi: possiamo ridurre la priorita` ad un processo oneroso per evitare rallentamenti di sistema.

  nice -n 8 find . -name “*.c”
- In questo modo riduciamo la priorita' dell'utility eseguita. Il valore di default di nice e' 0.
  - ✓ Valori negativi (solo root) danno maggiore priorita'
  - ✓ Valori positivi rappresentano minore prioritia'

Segnali

- Comunicazione con i processi e gestione
- Messaggio speciale spedito al processo con diversi significati
- Il segnale ricevuto viene processato subito dalla corrispondente funzione (vedi dopo)
- Definiti in: (/usr/include/bits/signum.h) <signal.h>

```
✓ #define   SIGHUP     1      /* Hangup (POSIX) */
✓ #define   SIGINT     2      /* Interrupt (ANSI)  */
✓ #define   SIGKILL    9      /* Kill, unblockable (POSIX).  */
```

Gestione dei segnali

Ogni segnale ha la sua *action* che determina il comportamento da seguire per quel segnale

- Ignorare il segnale (tranne SIGKILL, SIGSTOP)
- Override della funzione definendo un comportamento differente: signal-handler
- Default action (most of the cases TERMINATE)

Spedizione dei segnali

- Un processo puo` spedire segnali ad un altro processo
  - ✓ Terminare un determinato processo: SIGTERM
  - ✓ Spedire un comando ad un programma: SIGUSR1

- Il processo ricevente potrà gestire i segnali attraverso opportuni handlers tramite i quali potrà reagire all'evento

Bibliografia

- *GaPiL. Guida alla Programmazione in Linux*. Simone Piccardi (liberamente disponibile su internet)
- *Modern Operating Systems*. Andrew. S. Tanenbaum, Prentice-Hall 1992, Edizione italiana *Moderni sistemi operativi*, Jackson Libri.