Avvio

Quando un programma viene eseguito, prima vengono caricate le librerie condivise che servono al programma, poi viene effettua il link dinamico del codice e alla fine avviene l'esecuzione vera e propria.

Infatti, a meno di non aver specificato il flag -static durante la compilazione, tutti i programmi in Linux sono incompleti e necessitano di essere linkati alle librerie condivise quando vengono avviati.

Il sistema fa partire qualunque programma chiamando la funzione main; sta al programmatore chiamare cosi' la funzione principale del programma da cui si suppone iniziare l'esecuzione; in ogni caso senza questa funzione lo stesso linker darebbe luogo ad errori.

La funzione main puo' avere due argomenti argc ed argv, ma in realta' oltre a argc ed argv c'e' qualche cosa di piu'.

char** env

Di fatto al nostro main vengono passate anche tutte le variabili d'enviroment:

```
#include <stdio.h>

int main(int argc, char** argv, char** env)
{
  int i = 0;
  while(env[i] != 0)
  {
    printf("%s\n", env[i++]);
  }

  return 0;
}
```

char** env

```
$ gcc -o main1 main1.c
$ ./main1
MANPATH=/home/redo/build/qt-x11-commercial-
3.3.7/doc/man:/home/redo/build/qt-x11-commercial-
3.3.7/doc/man:
SSH_AGENT_PID=25898
HOSTNAME=banquo.thch.unipg.it
DESKTOP_STARTUP_ID=
SHELL=/bin/bash
TERM=xterm
CATALINA_HOME=/usr/local/tomcat
HISTSIZE=1000
KDE_NO_IPV6=1
GTK_RC_FILES=/etc/gtk/gtkrc:/home/redo/.gtkrc-1.2-
gnome2

cut...
```

Terminazione

Come avviene la terminazione di un processo:

- return alla fine del main. A tal proposito lo stato di uscita e' sempre 1 intero a 8 bit (usare EXIT_SUCCESS e EXIT_FAILURE). Viene comunque richiamata la exit.

- Chiamate della exit. (atexit ed on_exit per registrare funzioni)

- Chiamate della _exit. E' una syscall che restituisce il controllo direttamente al kernel.

- Segnali.

exit

Uso di exit, atexit ed on_exit.

```
$ gcc -o main2 main2.c
$ ./main2
chiamata func
$ gcc -o main3 main3.c
$ ./main3
chiamata func
```

_exit

Usando la syscall _exit invece il controllo viene passato direttamente al kernel:

```
$ gcc -o main4 main4.c
$ ./main4
```

IPC

I processi possono comunicare l'uno con l'altro e comunicare con il kernel mediante diversi meccanismi:

- Segnali - mandati da kernel o da qualche altro processo.
- Pipes - “unamed pipes” preparati normalmente dalla shell con il simbolo “|” per guidare l'output di un programma verso l'altro. (“named pipes” sono delle FIFO).
- Sockets – meccanismi di comunicazione bidirezionale
- System V IPC – E' una classe ampia di meccanismi di comunicazione che prende il nome dalla prima relase di UNIX che li ha implementati e sono:
  - Message queues: Meccanismo che puo' essere usato per consentire a diversi processi di scrivere messaggi in coda diretti allo stesso processo
  - Semafori: contatoti usati per accedere a risorse condivise
  - Shared Memory: aree di memoria condivisa

Segnali

kill puo' inviare diversi segnali ad un processo oltre il semplice SIGKILL:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Segnali

```
$ gcc -o signal signal.c
$ ./signal &
[2] 16299
[redo@banquo:~/Lezioni/Inform/lab_gen_inf/2008/slides/3]$ ps
  PID TTY          TIME CMD
 5446 pts/15   00:00:00 bash
15935 pts/15   00:00:00 soffice
15945 pts/15   00:00:16 soffice.bin
16299 pts/15   00:00:01 signal
16302 pts/15   00:00:00 ps
$ kill -s SIGSEGV 16299
[2]+  Segmentation fault      ./signal
[redo@banquo:~/Lezioni/Inform/lab_gen_inf/2008/slides/3]$ ./signal &
[2] 16307
$ ps
  PID TTY          TIME CMD
 5446 pts/15   00:00:00 bash
15935 pts/15   00:00:00 soffice
15945 pts/15   00:00:16 soffice.bin
16307 pts/15   00:00:02 signal
16308 pts/15   00:00:00 ps
$ kill -s SIGFPE 16307
[2]+  Floating point exception./signal
```

Segnali – coredump

```
$ ulimit -a
core file size          (blocks, -c) 0
cut...
```

Settiamo come unlimited la dimensione dei file di core:

```
$ ulimit -c unlimited
$ ./signal &
[2] 17995
$ kill -11 17995
$ file core.17995
core.17995: ELF 64-bit LSB core file x86-64, version 1 (SYSV), SVR4-styl
```

i file di core sono in formato ELF. Posso ispezionarne il contenuto usando ad esempio readelf (readelf -a core.17995).

Posso anche usare il debugger per capire cosa e' successo e dove:

```
$ gdb ./signal core.17995
```

...cut

```
Program terminated with signal 11, Segmentation fault.
#0  0x0000000000400653 in main ()
```

Segnali

Un processo puo' intrappolare o ignorare alcuni segnali ed agire di conseguenza. Vediamo come:

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

SIGKILL e SIGSTOP fanno eccezione non possono quindi essere intrappolati o ignorati.

```
$ gcc -W -Wall -o signal signal.c
signal.c:12: warning: unused parameter ‘argc’
signal.c:12: warning: unused parameter ‘argv’
signal.c:12: warning: unused parameter ‘env’
$ ./signal &
[1] 19264
$ kill -s SIGQUIT 19264
$ kill -s SIGPWR 19264
Here I am          30

$ kill -9 19264
[1]+  Killed                    ./signal
```

Esercizio

Scrivete un programma (demone) che dati due numeri reali ricevuti come parametri da linea di comando, calcoli e stampi la loro somma quando riceve un SIGPWR, e calcoli e stampi la differenza quando riceve un SIGSYS. (soluzione1.c)$ gcc -W -Wall -o soluzione1 soluzione1.c

```
$ gcc -W -Wall -o soluzione1 soluzione1.c
soluzione1.c:9: warning: unused parameter ‘unu’
soluzione1.c:16: warning: unused parameter ‘unu’
soluzione1.c:23: warning: unused parameter ‘env’
$ ./soluzione1
usage: ./soluzione1 num1 num2
$ ./soluzione1 10.2 5.3
I am the parent
$ ps
  PID TTY          TIME CMD
19787 pts/17   00:00:10 soluzione1
19792 pts/17   00:00:00 ps
30978 pts/17   00:00:00 bash
$ kill -s SIGPWR 19787
   10.20000 +    5.30000 =   15.50000
$ kill -s SIGSYS 19787
   10.20000 -    5.30000 =    4.90000
$ kill -9 19787
```

Esercizio

A questo punto invece di usare kill per mandare il segnale scrivere 1 applicazione che, ricevendo come argomento da linea di comendo il PID del processo, mandi ad esso i due segnali di cui sopra (man 2 kill) (soluzione2.c)

```
#include <sys/types.h>
#include <signal.h>

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

```
$ gcc -W -Wall -o soluzione2 soluzione2.c
soluzione2.c:7: warning: unused parameter ‘env’
$ ./soluzione1 10.2 5.8
I am the parent
$ ps
  PID TTY          TIME CMD
20438 pts/17   00:00:01 soluzione1
20439 pts/17   00:00:00 ps
30978 pts/17   00:00:00 bash
$ ./soluzione2 20438

   10.20000 -    5.80000 =    4.40000

   10.20000 +    5.80000 =   16.00000
```