3' Esercitazione

https://politecnicomilano.webex.com/meet/gianenrico.conti

Gian Enrico Conti

Processes

Esercizi:

1) Shell demo: ps
2) fork
3) fork e variabili
4) wait e wait_pid
5) exec + cmd line
6) info nel del descrittore di processo
4) proc() del file system x MAPPA di memoria
5) signal
6) zombie process
7) orphan process

Recap

- **Processi**
  - Monitoraggio dei processi
  - *fork*
  - *wait*
  - *exec*
  - Segnali

Processi

- **Un processo**
  - Rappresenta una istanza di esecuzione di un programma
  - Accede alle risorse (che chiede al S.O.)
  - Ha un suo Address Space, un suo IP/PC, stack ed Heap.

  - Def:*In Linux, an executable stored on disk is called a **program**, and a program loaded into memory and running is called a **process**.*

- **Il Process Descriptor contiene**
  - PID del processo padre (PPID)
  - Informazioni sul''utente (uid, gid, permessi)
  - Informazioni sulla memoria
  - Riferimenti alle tabelle di sistema
  - Informazioni sui tempi di esecuzione
  - etc..

Monitoraggio dei processi

Monitoraggio dei processi

- Nei sistemi Unix e Linux si usa **`ps .`** Opzion: digitare **`ps --help`**
- Per monitorare l'evoluzione in tempo reale dei processi, utilizzare **`top`** e **`htop`**. Digitare il comando **`t`**, una volta lanciato **`htop`**, per visualizzare l'albero di gerarchia dei processi.
- È possibile terminare i processi attraverso il comando kill, passando come parametro il flag –KILL e il PID del processo da terminare

- Live Demo

Ps : Live Demo

Da shell proveremo:

ps -Ax

htop

Stop + albero processi

NOTA: molte opzioni valgono SOLO x Linux)

Ps : Live Demo

```
ps -A
```

```
ingconti@MBP-32GB-BigSur ~ % ps -A
  PID TTY           TIME CMD
    1 ??        10:24.10 /sbin/launchd
   78 ??         0:21.29 /usr/sbin/syslogd
   79 ??         0:50.72 /usr/libexec/UserEventAgent (System)
   82 ??         0:04.80 /System/Library/PrivateFrameworks/Uninstall.framework/Resources/uninstalld
   83 ??         6:16.21 /System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/FS
   84 ??         1:08.73 /System/Library/PrivateFrameworks/MediaRemote.framework/Support/mediaremot
   87 ??         7:34.53 /usr/sbin/systemstats --daemon
   89 ??         0:30.52 /usr/libexec/configd
   90 ??         0:00.02 endpointsecurityd
   91 ??         2:53.57 /System/Library/CoreServices/powerd.bundle/powerd
   94 ??         0:25.20 /usr/libexec/remoted
   96 ??         3:41.80 /usr/libexec/logd
......
```

```
ps –x
```

Solo I miei....

Ps : Live Demo

-fA formattato .. PID e PPID

OSX

```
ingconti@MBP-32GB-BigSur ~ % ps -Af
  UID   PID  PPID   C STIME   TTY           TIME CMD
    0     1     0   0 Thu11PM ??         10:28.38 /sbin/launchd
    0    78     1   0 Thu11PM ??          0:21.38 /usr/sbin/syslogd
    0    79     1   0 Thu11PM ??          0:50.83 /usr/libexec/UserEventAgent (System)
```

Linux server:

```
[ingconti@srv01: ~]$ ps -fA
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 Feb26 ?        00:00:22 /sbin/init
root           2     0  0 Feb26 ?        00:00:00 [kthreadd]
..
ingconti 55153     1  0 20:05 ?        00:00:00 /lib/systemd/systemd --user
ingconti 55154 55153  0 20:05 ?        00:00:00 (sd-pam)
ingconti 55190 55149  0 20:05 ?        00:00:00 sshd: ingconti@pts/0
```

Ma cosa sono PID e PPID?

Ps : Live Demo

PID e PPID:

PID: Process ID

PPID: parent process ID

```
[ingconti@srv01: ~]$ ps -fA
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 Feb26 ?        00:00:22 /sbin/init
root           2     0  0 Feb26 ?        00:00:00 [kthreadd]
..
ingconti 55153     1  0 20:05 ?        00:00:00 /lib/systemd/systemd --user
ingconti 55154 55153  0 20:05 ?        00:00:00 (sd-pam)
ingconti 55190 55149  0 20:05 ?        00:00:00 sshd: ingconti@pts/0
```

Si notino i processi..

Ps : Live Demo

Per monitorare in tempo reale lo stato dei processi:

htop

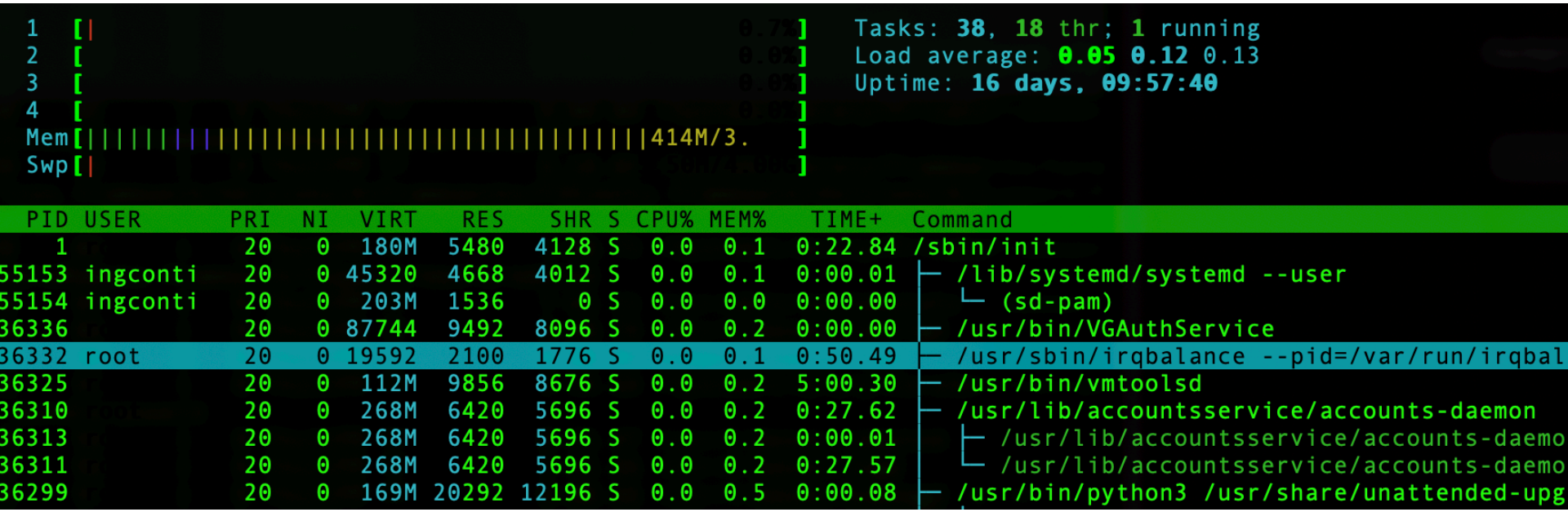
Una volta lanciato htop, digitare:

t

per conoscere l'albero dei processi..

Ps : Live Demo

htop:



Fork

Creazione dei processi

- Un processo viene creato tramite la funzione *fork*
  - *Codice proc2.c*
- Il processo figlio eredita tutte le variabili del padre, ma in forma di “copie”: non è concesso infatti al figlio di modificare i dati del padre, ma solo la propria copia locale di tali dati
  - proc2_bis.c

Fork: *proc2.c*

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int main(int argc, char** argv){
    pid_t pid;

    pid=fork();

    if(pid==0){
        //child
        printf("\nChild pid %d\n",(int)getpid());

    }else{
        //parent
        printf("\nParent pid %d\n",(int)getpid());
    }
}
```

(Int) x compatibility con altre piattaforme dove puo' non essere a 32 bit

Fork: *proc2.c*

Demo e commenti..

```
./proc2
```

```
Parent pid 51522
```

```
Child pid 51523
```

Fork: *proc2_bis.c*

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int main(int argc, char** argv){
     int  a=1;
     int* b=&a;
     pid_t pid;

     pid=fork();

     if(pid==0){
          //child
          printf("\nChild pid %d\n",(int)getpid());
          printf("\nChild b deref %d\n",(*b));

     }else{
          //parent
          a=4;
          printf("\nParent pid %d\n",(int)getpid());
          printf("\nParent b deref %d\n",(*b));
     }
}
```

Fork: *proc2_bis.c*

```
Parent pid 51542

Parent b deref 4

Child pid 51543

Child b deref 1
```

Logica:

A) entrambi ripartono
Da main

B) parent modifica la cella puntata da b

C) il child e' comunque UN ALTRO processo!

(Anche se passiamo x puntatore NON si puo' puntare alla aree di altri processi!)

```c
int main(int argc, char** argv){
	int  a=1;
	int* b=&a;
	pid_t pid;

	pid=fork();

	if(pid==0){
		//child
		printf("\nChild pid %d\n",(int)getpid());
		printf("\nChild b deref %d\n",(*b));

	}else{
		//parent
		a=4;
		printf("\nParent pid %d\n",(int)getpid());
		printf("\nParent b deref %d\n",(*b));
	}
}
```

Wait

- **Utilizzo della *wait***
- La famiglia *wait* è una famiglia di system call che permettono ad un processo padre di "attendere" la terminazione di un processo figlio e determinarne l'*exit status*
- Vi sono due varianti principali:
  - `pid_t wait(int* status);`
  - `pid_t waitpid(pid_t pid, int *status, int options);`

wait(int* status)

- La ***wait()*** è bloccante, attende la terminazione di un processo figlio qualunque e ritorna il pid del processo figlio che è terminato. Tale valore può essere:

pid	Il pid del processo figlio che è terminato e che la wait ha raccolto
-1	Segnala un errore nella terminazione e viene settata di conseguenza la variabile errnp

Il parametro int* status, viene invece settato al valore di ritorno restituito dal figlio all'atto della sua terminazione

waitpid(pid_t pid, int* status, int options)

- La ***waitpid()*** può essere bloccante o non bloccante a seconda del terzo argomento, `int options`, specificando `WNOHANG` == non bloccante; `WUNTRACED` che impone di ritornare anche nel caso in cui i figli siano rimasti bloccati, ignorandone di conseguenza l'*exit status*

Wait

< -1	Attende qualunque processo figlio avente process **group** id uguale al valore assoluto del pid
-1	Attende qualunque processo figlio, come la wait()
0	Attende qualunque processo figlio avente process **group** id uguale a quello corrente
> 0	Attende il processo avente il pid specificato

Esercizio	proc	waitpid

Wait

Controllare l'exit status

- Attraverso l'utilizzo di opportune macro, è possibile testare l'exit status registrato dalle wait:

`WIFEXITED(status)`	Vera se il figlio è terminato "naturalmente"
`WIFSIGNALED(status)`	Vera se il figlio è terminato a causa di un segnale, che non ha gestito
`WTERMSIG(status)`	Riporta il segnale che ha causato la terminazione del figlio
`WEXITSTATUS(status)`	Riporta i less-significant 8 bit dell'exit status ed ha senso solo se WIFEXITED(status) risulta verificata
`WIFSTOPPED(status)`	Ha valore solo se `WUNTRACED` è impostata nelle opzioni e risulta vera se il figlio risulta stopped
`WSTOPSIG(status)`	Ritorna il numero di segnale che ha causato lo stop del figlio

- Secondo quanto visto a lezione, l'utilizzo delle wait e la sequenza di terminazione di padre e figli influisce sullo stato degli stessi, che possono venire dunque a trovarsi in uno stato di Zombie.
- Tale stato si ha nel momento in cui il figlio termina ed il padre non ha ancora effettuato la wait o è terminato a sua volta senza chiamare tale funzione. A seconda dei casi il processo rimarrà zombie fino alla chiamata della wait, oppure, se il padre è terminato, rimarrà zombie finché non verrà adottato e terminato dal processo init.

Wait live demo

- Code:

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char** argv)
{
    pid_t pid_to_wait[10], pid, pid_ritornato;
    int status;
    int i;

    printf("Parent ID: %d\n", (int)getpid());
    for(i = 0; i < 10; i++)
    {
        pid = fork();
        if( pid == 0 ){ // FIGLIO
            printf("Child %d ID: %d\n",i, (int)getpid());
            exit(i);
        }
        else{           // PADRE
            pid_to_wait[i] = pid;
        }
    }

    printf("Parent - Aspetto il figlio 4\n");
    pid_ritornato = waitpid( pid_to_wait[3], &status, 0 );
    printf("Parent - Il pid del figlio 4 è %d, l'exit status è %d\n", pid_ritornato, WEXITSTATUS( status ) );

    // Attendere tutti gli altri figli
    for(i = 0; i < 9; i ++)
    {
        pid_ritornato = wait( &status );
        printf("Parent - Il pid è %d, l'exit status è %d\n", pid_ritornato, WEXITSTATUS( status ) );
    }

    return 0;
}
```

Wait live demo

- Si noti a exit.. provate a commentarla...

```
Parent ID: 51909
Child 0 ID: 51910
Child 1 ID: 51911
Child 2 ID: 51912
Child 3 ID: 51913
Child 4 ID: 51914
Child 5 ID: 51915
Child 6 ID: 51916
Child 7 ID: 51917
Parent - Aspetto il figlio 4
Child 8 ID: 51918
Parent - Il pid del figlio 4 è 51913, l'exit status è 3
Parent - Il pid è 51917, l'exit status è 7
Parent - Il pid è 51916, l'exit status è 6
Parent - Il pid è 51915, l'exit status è 5
Parent - Il pid è 51914, l'exit status è 4
Parent - Il pid è 51912, l'exit status è 2
Parent - Il pid è 51911, l'exit status è 1
Parent - Il pid è 51910, l'exit status è 0
Child 9 ID: 51919
Parent - Il pid è 51918, l'exit status è 8
Parent - Il pid è 51919, l'exit status è 9
```

WaitPID live demo

- Code:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char** argv){
        pid_t pid, pid_to_wait[10],pid_ritornato;
        int status;
        int i;

        printf("Parent pid %d\n",(int)getpid());
        for(i=0;i<10;i++){
        pid=fork();

                if(pid==0){
                        //child
                        printf("Child pid %d\n",(int)getpid());
                        exit(i);
                }else{
                        //parent
                        pid_to_wait[i]=pid;
                }
        }

        //esegue nel padre in quanto i figli hanno effettuato la exit
        printf("\nParent - Ho dieci figli, attenderò prima il figlio n°4 con waitpid, poi tutti gli altri con wait\n");
        pid_ritornato= waitpid( pid_to_wait[4], &status, 0 );
        printf( "\nParent - Effettuata waitpid sul quarto figlio\nIl suo pid è %d,l'exit status %d\n", pid_ritornato,
WEXITSTATUS( status ) );

        //attendo tutti gli altri
        // Waits for all other children
        printf( "Parent - Attendo gli altri\n" );
        for( i= 0; i< 9; i++ ) {
        pid_ritornato = wait( &status );
        printf( "Parent - Tornato il figlio con pid %d, exit status: %d\n",pid_ritornato, WEXITSTATUS( status ));
        }
}
```

WaitPID live demo

```
Parent pid 51954
Child pid 51955
Child pid 51956
Child pid 51957
Child pid 51958
Child pid 51959
Child pid 51960
Child pid 51961
Child pid 51962
Child pid 51963

Parent - Ho dieci figli, attenderò prima il figlio n°4 con waitpid, poi tutti gli altri
con wait

Parent - Effettuata waitpid sul quarto figlio
Il suo pid è 51959,l'exit status 4
Parent - Attendo gli altri
Parent - Tornato il figlio con pid 51961, exit status: 6
Parent - Tornato il figlio con pid 51960, exit status: 5
Parent - Tornato il figlio con pid 51958, exit status: 3
Parent - Tornato il figlio con pid 51957, exit status: 2
Parent - Tornato il figlio con pid 51956, exit status: 1
Parent - Tornato il figlio con pid 51955, exit status: 0
Parent - Tornato il figlio con pid 51962, exit status: 7
Parent - Tornato il figlio con pid 51963, exit status: 8
Child pid 51964
Parent - Tornato il figlio con pid 51964, exit status: 9
```

Exec

Utilizzo della *exec*

- Spesso all'utilizzo della *fork* è associato l'utilizzo della funzione *exec*, nelle sue varie versioni
- La famiglia di funzioni *exec* non fa altro che interrompere l'esecuzione del programma corrente e passare all'esecuzione del nuovo programma, il tutto all'interno del processo dalla quale è invocata
- Vi sono tre grandi famiglie di exec:
  - Quelle che contengono la lettera *p*: queste versioni di exec prendono come parametro il nome del programma da eseguire e lo ricercano all'interno della directory corrente. Alle versioni di exec senza la *p* occorre passare il path completo del programma da eseguire
  - Quelle che contengono la lettera *v*: queste versioni accettano i parametri da passare al programma in forma di vettori di puntatori a stringhe, terminate dal carattere nullo
  - Quelle che contengono la lettera *l*: a differenza delle versioni con la *v*, accettano il passaggio di parametri secondo il meccanismo *varargs* del C
  - Quelle che contengono la lettera *e*: a tali versioni vengono passate anche le variabili ambientali, sotto forma di vettore di puntatori a stringhe, terminato dal carattere nullo, in cui le stringhe sono nella forma (VARABILE=valore)
- Le exec non ritornano, a meno di errori, in quanto il programma chiamante viene completamente rimpiazzato
- Il primo argomento passato all'exec deve essere il nome del programma stesso, come avviene normalmente per il main.

Exec Live demo

2 codici

A) programma secondario semplice (programma.c)

B) programma con exec (execv.c)

Exec Live demo

A) programma secondario semplice (programma.c)

```c
#include <stdio.h>

int main(int argc, char** argv){
     int i;

     for(i=0;i<argc;i++)
          printf("\nProgramma - print argument %d: %s\n",i,argv[i]);
}
```

Exec Live demo 1

B) programma con exec (exec_simple.c)

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int main(int argc, char** argv){
    char*  arg[]={"./programma", "ciao", "mamma",NULL};
    int result = execvp("./programma",arg);
    printf("\n res:  %d\n", result);
}
```

Exec Live demo

B) programma con exec (execv.c)

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char** argv){
     pid_t pid;
     char*  arg[]={"./programma", "ciao", "mamma",NULL};

     pid=fork();

     if(pid==0){
          //child
          printf("\nChild pid %d\n",(int)getpid());
          int result = execvp("./programma",arg);

     }else{
          //parent
          printf("\nParent pid %d\n",(int)getpid());
     }
}
```

Utilizzo di kill per inviare segnali da shell

Attraverso il comando `kill` è possibile inviare segnali ai processi da shell

- La sintassi del comando kill segue il seguente schema:

```
kill -SEGNALE pid_processo
```

Dove `SEGNALE` è il codice del segnale che vogliamo inviare, ad esempio:

- KILL
- SIGUSR1
- SIGUSR2
- etc.

Il ' argomento, `pid_processo`, è il pid del processo a cui vogliamo inviare il segnale. Per recuperare il pid di un processo da shell, digitare il comando:

```
ps ax | grep nome_processo
```

e considerare il primo campo stampato, che corrisponde al pid

Terminazione dei processi

- un processo termina naturalmente a fronte di un'istruzione `exit()` o della return del `main`
- Un processo può anche terminare in maniera anomala a seguito della ricezione dei segnali:
  - SIGBUS, SIGSEGV, SIGFPE – illustrati in precedenza
  - SIGINT, segnale inviato quando l'utente digita il comando Ctrl+C
  - SIGTERM, tale segnale è inviato dal comando kill e come azione di default terminerà un processo.
  - inviando a se stesso un segnale SIGABRT, attraverso la funzione abort, un processo causa la terminazione di se stesso e la generazione di un *core file*
  - SIGKILL, in assoluto il più potente tra i segnali di terminazione, il quale impone la terminazione immediata di un processo che non potrà in alcun modo gestire il segnale
- Un segnale di tipo SIGKILL può essere inviato sia da shell, digitando il comando

```
% KILL –KILL pid
```

o anche all'interno di un programma attraverso la funzione `kill(pid_t child, int signal)`, utilizzando SIGTERM per simulare il comportamento del comando KILL:

```
kill(child_pid, SIGTERM);
```

Segnali

Utilizzo dei segnali

- I segnali, sono uno strumento estremamente semplice, ma al tempo stesso efficace per la comunicazione tra processi
- Ve ne sono vari tipi, definiti in `/usr/include/bits/signum.h`.
- Ubuntu: */usr/include/x86_64-linux-gnu/bits/signum.h*
- Un processo che riceve un segnale può gestirlo essenzialmente in tre modi:
  - **Default:** ogni segnale ha un handler di default che viene eseguito se nessun handler specifico viene definito e se il segnale non viene ignorato. Tale modalità può essere attivata tramite l'opzione `SIG_DFL` passata alla struttura `sigaction`, che vedremo tra breve.
  - **Ignorandolo**: in tal caso il segnale non viene gestito in alcun modo. Tale modalità può essere attivata tramite l'opzione `SIG_IGN` passata alla struttura `sigaction`.
  - **Gestito**: un segnale può essere gestito attraverso un opportuno handler, ovvero una funzione che specifica le azioni da intraprendere a fronte della ricezione del segnale. Non per tutti i segnali è possibile definire un handler.
- Il sistema operativo (nello specifico Linux) utilizza alcuni particolari segnali per comunicare con i processi, esempi importanti sono:
  - `SIGBUS`: per segnalare un bus error
  - `SIGSEGV`: per segnalare una segment violation
  - `SIGFPE` per le floating point exception

  In tutti e tre i casi il processo viene terminato e viene generato un opportuno codice di errore.

Segnali

- Vi è altresì la possibilità che dei processi si trasmettano dei segnali tra loro. Esempi di primo piano di tali segnali sono:
  - `SIGKILL` o `SIGTERM` per terminare il processo
  - `SIGUSR1` e `SIGUSR2` per innescare specifiche azioni
  - `SIGHUP` per risvegliare un processo in idle o costringerlo a leggere nuovamente il file di configurazione
  - `SIGALARM`, lanciato dalla funzione `unsigned int alarm(unsigned int seconds)` per generare un evento di allarme dopo tot secondi
  - `SIGCHL` segnale inviato al processo padre quando uno dei sui figli termina

Programmare con i segnali

- Per gestire i segnali vengono utilizzate la funzioni `signal()` e `sigaction()`
  - **signal:**

```
void (*signal(int sig, void (*func)(int)))(int)
```

    Il primo parametro è il numero del segnale che dovrà essere gestito, il secondo invece è un puntatore a funzione, in particolare al signal handler che specifica le azione da intraprendere per gestire il segnale
  - **sigaction:**

```
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact)
```

    Il primo parametro specifica il numero del segnale da gestire, il secondo e il terzo sono puntatori a `struct sigaction` che specificano, rispettivamente, le azioni che andranno intraprese al ricevimento del segnale e le azioni disposte in precedenza.

    Il campo più imporate delle struct sigaction è sa_handler, che può essere un *puntatore a funzione* allo handler definito dal programmatore (che a sua volta prende come parametro di ingresso il numero di segnale),o uno dei due flag `SIG_DFL` o `SIG_IGN`

Segnali

Programmare con i segnali [Continua]

- Per gestire i segnali il programmatore può definire dei signal handler, ovvero delle funzioni che prendano in ingresso il numero del segnale e svolgano il numero di operazioni **minino e indispensabile** utili a gestirlo.
- Esempio di signal handler:

```
typedef void (*sighandler_t)(int);

void signal_handler( int sig )
{
  if( sig == SIGUSR1 ) {
    printf( "Handling signal SIGUSR1.\n" );
    exit( 0 );
  } else {
    printf( "Handling signal SIGUSR2.\n" );
  }
}
```

- È possibile che un signal handler possa essere interrotto a sua volta dall'arrivo di un altro segnale (situazione molto difficile da "debuggare") e per tale motivo è necessario che compia il minor numero di operazioni possibile

Segnali

Programmare con i segnali [Continua]

- Persino l'operazione di assegnamento è rischiosa in quanto comporta in alcuni casi più di una operazione assembly, il flusso di esecuzione tra le quali verrebbe in ogni caso interrotto all'arrivo del segnale
- A tale scopo è definito il tipo di variabile `sig_atomic_t` il cui assegnamento è atomico: variabili di questo tipo possono essere utilizzate come variabili globali per tenere traccia, ad esempio, del fatto che il segnale sia stato ricevuto.
  - `sig_atomic_t` è di tipo int, ma possono essere utili al caso anche altri tipi di dimensione minore o uguale all'int, in quanto per tali variabili l'istruzione di assegnamento è atomica.

Segnali 3 live demo

Istruzioni:

sign-1)

- **compilare con: gcc sign-1.c -o sign-1**
- **lanciare il programma**
- **aprire un'altra shell**
- **digitare: ps ax | grep sign-1**
- **ricordare il pid di ./sign-1**
- **sempre nella shell, inviare il comando: kill -SIGUSR1 pid_trovato**
- **vedere cosa succede nell'altra shell,quella in cui ho lanciato il programma**

sign-2)

- **compilare con: gcc sign-2.c -o sign-2**
- **attendere 10 secondi**

sign-3)

- **compilare con: gcc sign-3.c -o sign-3**
- **attendere 2-3 secondi**

Segnali sig-1

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

typedef void(*sighandler_t) (int);

void signal_handler(int sig){
        if(sig==SIGUSR1){
                printf("\nHandling SIGUSR1\n");
                exit(0);
        }
}

void signal_handler2(int sig){
        if(sig==SIGUSR2){
                printf("\nHandling SIGUSR2\n");
                exit(0);
        }
}

int main(int argc, char** argv){
        sighandler_t previous;

        printf("\nAttendo un tuo segnale, scegli tu se inviare SIGUSR1 o SIGUSR2\n");

        //Registro l'handler per SIGUSR1
        previous= signal(SIGUSR1, signal_handler);
        if(previous==SIG_ERR)
                printf("\nErrore nel registrare il signal handler per SIGUSR1\n");

        //Registro l'handler per SIGUSR2
        previous= signal(SIGUSR2, signal_handler2);
        if(previous==SIG_ERR)
                printf("\nError Errore nel registrare il signal handler per SIGUSR2\n");


        while(1);
        return 0;

}
```

Segnali sig-2

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


void alarm_handler( int sig )
{
  printf( "Allarme ricevuto, sono passati 10 secondi\n" );
  exit( 1 );
}


int main( int argc, char** argv )
{
  // Registro l'handler
  if( signal( SIGALRM, alarm_handler ) == SIG_ERR ) {
    printf( "Errore nel registrare l'handler per SIGALARM\n" );
    exit( 1 );
  }

  printf( "Ho impostato l'allarme, scatterà tra 10 secondi\n" );
  alarm( 10 );

  while(1);

  return 0;
}
```

Segnali sig-3

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

typedef void (*sighandler_t)(int);

void signal_handler( int sig ) {
  int status;
  pid_t pid;

  // Waitpid non bloccante, che attende qualunque processo (primo parametro è zero)

  pid=waitpid( 0, &status, WNOHANG );
  printf( "Sono l'handler, è arrivato il figlio pid: %d, exit status: %d. Esco\n",pid, WEXITSTATUS( status ) );
  exit(0);
}

int main( int argc, char** argv ){
  pid_t pid;

  printf("Questo programma aspetta i figli con la waitpid non bloccante sfruttando i segnali\n");

  printf("Registro l'handler epr SIGCHLD\n");

  if( signal( SIGCHLD, signal_handler ) == SIG_ERR ) {
    printf( "Errore nel registrare l'handler per SIGCHLD\n" );
    exit( 1 );
  }


  pid = fork();

  if( pid == 0 ) {
    printf( "Child pid: %d\n", getpid());
    sleep(2);
    printf( "Child - exit con codice 33\n" );
    exit(33);

  } else {
    printf( "Parent - pid: %d, entro in while(1)\n", getpid() );
    while(1);
  }
}
```

Proc e memoria

`cat /proc/meminfo`

- Ci da il layout di memoria:

```
MemTotal:        4028024 kB
MemFree:          217560 kB
MemAvailable:    3323524 kB
Buffers:          339744 kB
Cached:          2871100 kB
SwapCached:          664 kB
Active:          2990604 kB
Inactive:         490484 kB
Active(anon):     152492 kB
Inactive(anon):   161120 kB
Active(file):    2838112 kB
Inactive(file):   329364 kB
Unevictable:           0
```

...

Proc e memoria di un processo

- 3 passi:

  - `Individuare PID`
  - `/proc/[pid]/mem`
  - `/proc/[pid]/maps`

Creeremo un file temporaneo "test":

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
int main(void)
{
    char *s;
    s = strdup("EX3_ACSO_2021");
    if (s == NULL)
    {
        fprintf(stderr, "malloc failed\n");
        return (EXIT_FAILURE);
    }
     int i = 0;
     while (s)
     {
          printf("[%d] %s (%p)\n", i, s, (void *)s);
          sleep(1);
          i++;
     }
     return (EXIT_SUCCESS);
}
```

Proc e memoria di un processo

```
gcc -Wall test.c -o test
```

E lanciamolo, da altra shell:

```
[ingconti@srv01: ~]$ ps -aux | grep test
```

```
ingconti 61012  0.0  0.0   4352   624 pts/0    S+   22:19   0:00 ./test
ingconti 6   1156  0.0  0.0  12948  1024 pts/1    S+   22:20   0:00 grep --color=auto
test
```

Ora:

```
cd /proc/61012
```

```
sudo cat maps
```

```
...
```

Proc e memoria di un processo

```
00dd1000-00df2000 rw-p 00000000 00:00 0                                  [heap]
7fef90d2c000-7fef90eec000 r-xp 00000000 08:01 1046675                    /lib/x86_64-linux-gnu/libc-2.23.so
7fef90eec000-7fef910ec000 ---p 001c0000 08:01 1046675                    /lib/x86_64-linux-gnu/libc-2.23.so
7fef910ec000-7fef910f0000 r--p 001c0000 08:01 1046675                    /lib/x86_64-linux-gnu/libc-2.23.so
7fef910f0000-7fef910f2000 rw-p 001c4000 08:01 1046675                    /lib/x86_64-linux-gnu/libc-2.23.so
7fef910f2000-7fef910f6000 rw-p 00000000 00:00 0
7fef910f6000-7fef9111c000 r-xp 00000000 08:01 1047041                    /lib/x86_64-linux-gnu/ld-2.23.so
7fef9130d000-7fef91310000 rw-p 00000000 00:00 0
7fef9131b000-7fef9131c000 r--p 00025000 08:01 1047041                    /lib/x86_64-linux-gnu/ld-2.23.so
7fef9131c000-7fef9131d000 rw-p 00026000 08:01 1047041                    /lib/x86_64-linux-gnu/ld-2.23.so
7fef9131d000-7fef9131e000 rw-p 00000000 00:00 0
7ffcd913c000-7ffcd915d000 rw-p 00000000 00:00 0                          [stack]
7ffcd9172000-7ffcd9175000 r--p 00000000 00:00 0                          [vvar]
7ffcd9175000-7ffcd9177000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]HEASP
```

Colonne:

```
address           perms offset  dev   inode       pathname
```

HEAP:

```
00dd1000-00df2000 rw-p 00000000 00:00 0                                  [heap]
```

Inizio fine RW

Dal run della altra shell la stringa allocato con strdup e' a:

```
[563] EX3_ACSO_2021 (0xdd1010)
```

Giusto all' inizio dello HEAP.

Zombie e Orphan process

Zombie:

is a process that has finished its execution and is waiting for its Parent Process to read its exit status. Because the child process has finished, it is technically a “dead” process however since it is waiting for its parent there is still an entry in the process table.

An Orphaned process is a child process that is still an active process whose parent has died.

Unlike zombies the orphaned process will be reclaimed or adopted by the init process.

Come trovare uno zombie:

```
ps aux | grep 'Z'
```

Z e' una colonna di stats...

Orphaned:

PPID == 1 (1 e' il processo **init**).