Esercizi

PROCESSI E THREADS

- Chiamate di sistema LINUX per la gestione dei processi
- Librearia pthread (threads Posix)

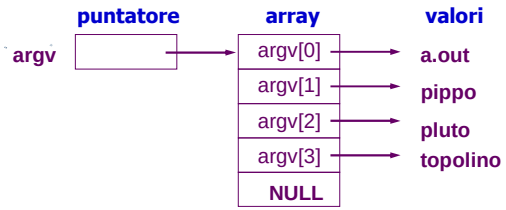
Per incominciare....

```
#include <stdio.h>

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

   for (i = 0; i < argc; i++)
                  printf("argv[%d]: %s\n", i, argv[i] );

   exit(0);
}
```


Esempio (cont.)

```
[lapegna%] cc program.c
[lapegna%] a.out  pippo  pluto  topolino
argv[0]: a.out
argv[1]: pippo
argv[2]: pluto
argv[3]: topolino
[lapegna%]
```


argc e argv



argc e' un intero (numero di argomenti sulla linea di comando)

argv e' un array di puntatori a carattere (gli argomenti sulla linea di comando)

Identificativi di processo

Ogni processo e' identificato univocamente da un intero

```
[lapegna%] ps
  PID  TTY          TIME  CMD
11949  pts/0    00:00:00  bash
12031  pts/0    00:00:00  ps
[lapegna%]
```


Primitive per la gestione dei processi

Unix fornisce le primitive per la creazione e terminazione di processi, e per l'esecuzione di programmi:

- **fork** crea nuovi processi;
- **exec** attiva nuovi programmi;
- **exit** termina un processo.
- **wait** attende la terminazione di un processo.

Funzione fork

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

L'unico modo per istruire il kernel a creare un nuovo processo è di chiamare la funzione **fork** da un processo esistente.

Il processo creato viene detto figlio. Il processo che chiama **fork** viene detto genitore

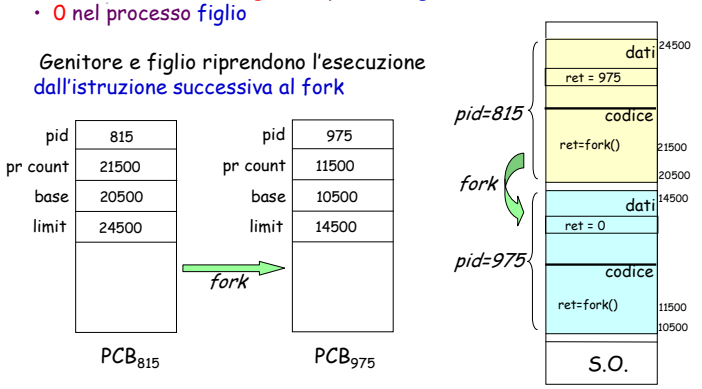
Il figlio è una copia del genitore (data space, heap e stack), cioè essi non condividono parti di memoria.

Funzione fork

La funzione ritorna :

- l'identificativo del figlio nel processo genitore,
- 0 nel processo figlio

Genitore e figlio riprendono l'esecuzione dall'istruzione successiva al fork



Come differenziare il padre dal figlio?

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

int main ()  {
  int pid;

  if( (pid=fork( ))==0)     /* sono il figlio */

          printf("Ciao, sono il figlio\n");

  else                      /* sono il padre */

         printf("Ciao, sono il padre\n");

  exit(0);
}
```


Esempio (esecuzione)

```
[lapegna%] cc –o eseg program.c
[lapegna%] eseg
Ciao sono il padre
Ciao sono il figlio
[lapegna%]
```


Identificativi dei processi

```
# include <unistd.h>
# include <sys/types.h>
pid_t  getpid(void);        identificativo di processo
pid_t  getppid(void);       identificativo del padre
```

Queste funzioni ritornano
gli identificativi di processo.

esempio

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


int main ( ) {
  pid_t pid;

  if( (pid=fork( )) ==0)     /* sono il figlio */

    printf("F: miopid = %d , pidpadre = %d \n", getpid( ), getppid( ) );

  else                       /* sono il padre */

    printf("P: miopid = %d , pidfiglio = %d \n", getpid( ), pid );

  exit(0);
}
```

Esempio (esecuzione)

```
[lapegna%] cc –o eseg program.c
[lapegna%] eseg
F: miopid = 428 , pidpadre = 426
P: miopid = 426 , pidfiglio = 428
[lapegna%]
```


Esercizio:

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

int main ( ) {
pid_t  pid;
int  i;

   for ( i=0 ; i<3 ; i++) {

         pid=fork( );

   }
   printf(“ processo %d terminato\n”, getpid() );

  exit(0);
}
```

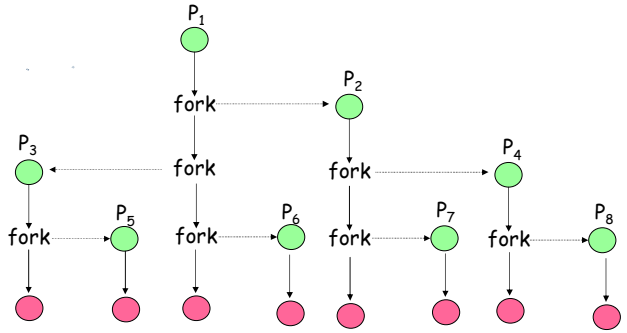
Quanti processi vengono generati?

Esecuzione

```
[lapegna%] cc -o eseg program.c
[lapegna%] eseg
processo 2167 terminato
processo 2170 terminato
processo 2163 terminato
processo 2164 terminato
processo 2169 terminato
processo 2166 terminato
processo 2168 terminato
processo 2165 terminato
[lapegna%]
```

8 processi !!!

infatti



In generale 2^n processi !!!

Funzione exit

exit termina un programma normalmente e libera le risorse

```
# include <stdio.h>

void exit(int status);
```

La funzione ha per argomento un intero che rappresenta il valore di ritorno del programma

Tale valore non è definito quando:
1. la funzione viene chiamata senza argomento,
2. se il programma termina prima del dovuto.

Comunicazione padre - figlio

Un figlio che termina normalmente comunica al genitore il suo valore di uscita mediante l'argomento della funzione exit

Nel caso di terminazione non normale, il kernel genera uno stato di terminazione per indicare la ragione.

Se il genitore termina prima del figlio, il processo **init** eredita il figlio e il parent process ID di questo diventa 1.

Se il figlio termina prima che il genitore sia in grado di controllare la sua terminazione, il kernel conserva almeno il process ID e lo stato di terminazione.
Tali processi vengono detti zombie.

esempio

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


int main ( ) {
  pid_t pid;

  if( (pid=fork( )) ==0) {    /* sono il figlio */

   sleep(1);
   printf("F: miopid = %d , pidpadre = %d \n", getpid( ), getppid( ) );

  }else {                     /* sono il padre */

    printf("P: miopid = %d , pidfiglio = %d \n", getpid( ), pid );

 }
  exit(0);
}
```


esempio

```
[lapegna%] cc –o eseg program.c
[lapegna%] eseg
P: miopid = 723 , pidfiglio = 724
F: miopid = 724 , pidpadre = 1
[lapegna%]
```

Funzione wait

Il genitore è in grado di ottenere lo stato di terminazione di un figlio mediante la funzione **wait**

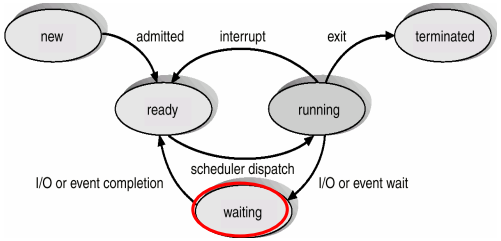
```
# include <sys/types.h>
# include <sys/wait.h>
pid_t wait(int *statloc)
```

l'argomento *statloc* é un puntatore ad un intero.

In tale intero sono codificati
- lo stato di terminazione
- il valore di ritorno del processo figlio (comunicato da exit)

Funzione wait

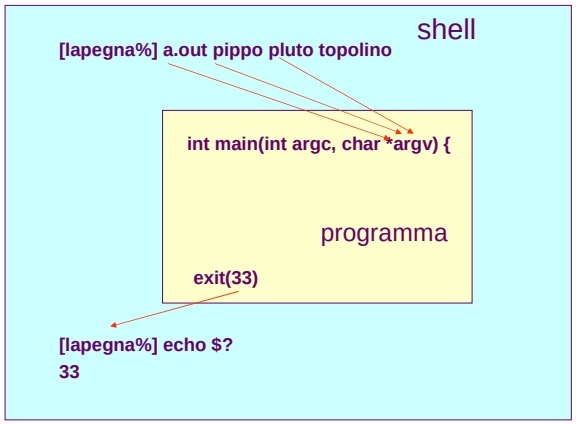
La funzione **wait** sospende il processo genitore finché:
- il figlio ha terminato la propria esecuzione, oppure
- riceve un segnale



Esempio: Shell e main

- un programma C e' un insieme di funzioni che si richiamano l'un l'altra
- una (e solo una) deve chiamarsi main, ed e' la funzione dalla quale inizia l'esecuzione
- la shell crea un nuovo processo per ogni comando che riceve
- la shell passa gli argomenti alla funzione main e da essa riceve i valori di ritorno

Shell e processi



Stato di terminazione

Le macro definite in <sys/wait.h> interrogano la variabile status

Macro	Descrizione
WIFEXITED(*status*)	vero se il figlio è terminato normalmente WEXITSTATUS(*status*) ritorna lo stato
WIFSIGNALED(*status*)	vero se il figlio è uscito per un segnale WTERMSIG(*status*) ritorna il segnale
WIFSTOPPED(*status*)	vero se il figlio è fermato WSTOPSIG(*status*) ritorna il segnale

esempio

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

 int main(void) {
      pid_t   pid;
      int     status, a, b, c;

      if ( (pid = fork()) == 0)    {                    /* figlio */

                  printf("dammi a e b --> ");
                  scanf("%d %d", &a, &b);
                  c = a+b;
                  exit(0);

            }else{                                      /* padre */

                  wait(&status) ;      /* aspetta il figlio */
                  if ( WIFEXITED(status))
                       printf(" ritorno dal figlio = %d \n", WEXITSTATUS(status));
                  else
                       printf("figlio non terminato correttamente\n");

            }
}
```


esempio

```
[lapegna%] cc –o eseg program.c
[lapegna%] eseg
dammi a e b -->8 9
ritorno dal figlio = 0
[lapegna%]
```


Funzione execl

La funzione **execl** sostituisce il codice e i dati del programma in esecuzione con il codice e i dati di un nuovo programma all'interno dello stesso processo

```
# include <unistd.h>

int execl   ( const char *path, const char *arg0, ... , NULL );
```

nuovo programma

argomenti del programma

L'identificativo del nuovo processo è lo stesso di quello sostituito.

Esempio

Programma “hello”

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

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

    printf(“hello! Il mio pid = %d, getpid() );
    printf(“argomenti = %s %s \n“, argv[1], argv[2]);

  exit(0);
}
```


Esempio (cont)

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

int main ()  {
  int pid;

  if( (pid=fork( ))==0)     /* sono il figlio */
    printf("Ciao, sono il figlio, pid = %d \n“, getpid());
    execl(“/home/lapegna/hello”, “hello”, “pippo”, “pluto”, NULL);

  else                      /* sono il padre */
    printf("Ciao, sono il padre\n");

  exit(0);
}
```


Esempio (esecuzione)

```
[lapegna%] a.out
Ciao sono il figlio, pid = 11543
Ciao sono il padre
hello, il mio pid = 11543
argomenti = pippo pluto
[lapegna%]
```


Threads POSIX (Pthreads)

- Uno standard POSIX (IEEE 1003.1c) per la creazione e sincronizzazione dei threads
- Definizione delle API
- Threads conformi allo standard POSIX sono chiamati Pthreads
- Lo standard POSIX stabilisce che registri dei processori, stack e signal mask sono individuali per ogni thread
- Lo standard specifica come il sistema operativo dovrebbe gestire i segnali ai Pthtrads i specifica differenti metodi di cancellazione (asincrona, ritardata, …)
- Permette di definire politiche dischedulingе priorita’
- Alla base di numerose librerie di supporto per vari sistemi

Creazione di un thread

```
#include <pthread.h>
int pthread_create( pthread_t  *tid,
                    const pthread_attr_t  *attr,
                    void *(*func)(void*),
                    void *arg );
```

tid: puntatore all'identificativo del thread ritornato dalla funzione

attr: attributi del thread (priorita', dimensione stack, …) . A meno di esigenze particolari si usa NULL

func: funzione di tipo **void *** che costituisce il corpo del thread

arg : unico argomento di tipo **void *** passato a **func**

La funzione ritorna subito dopo creato il thread

Attesa per la fine di un thread

```
#include <pthread.h>

int pthread_join( pthread_t  tid,
                  void ** status );
```

tid: identificativo del thread di cui attendere la fine

status: puntatore al valore di ritorno del thread. Di solito NULL

Processi vs Pthread

	Processi	Threads
creazione	**fork**	**pthread_create**
attesa	**wait**	**pthread_join**

Altre funzioni

```
int    pthread_detach(pthread_t   thread_id);
int    pthread_exit(void   *value_ptr);
pthread_t    pthread_self();
```


Esempio

Realizzare un programma con
2 threads tale che

- thread scrivi incrementa una variabile condivisa *shareddata* da 1 a 5, aspettando 1 secondo tra un incremento e l'altro
- thread leggi legge e stampa il contenuto di *shareddata* solo quando il thread scrivi ha modificato la variabile condivisa
- Far terminare il thread leggi quando termina il thread scrivi

Multithreading: main

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

int    shareddata =0  ,  finethread=0;      //   dati condivisi
void *scrivi(void *), *leggi(void *);

int main( ){

        pthread_t    tid_scrivi,   tid_leggi;
        pthread_create(&tid_scrivi, NULL, scrivi, NULL);
        pthread_create(&tid_leggi, NULL, leggi, NULL);
        pthread_join(tid_scrivi, NULL);
        pthread_join(tid_leggi, NULL);

}  //fine main
```


Thread scrivi

```
void *scrivi (void *arg){
int   i;

        printf(" partito  thread  scrivi\n");
        for (i=0;  i<5  ; i++){

                sleep(1);
                shareddata = shareddata+1;
                printf("thread leggi: shareddata = %d \n",shareddata);

        }

        finethread = 1;
}
```


thread leggi

```
void *leggi (void *arg){
        int oldshareddata=0;

        while  (finethread==0){
                while( shareddata == oldshareddata);
                printf("thread leggi: shareddata = %d \n",shareddata);
                oldshareddata = shareddata;
        }

}
```

N.B. compilare il tutto linkando la libreria libpthread.a

Esempio (esecuzione)

```
[lapegna%] a.out
 partito  thread  scrivi
 partito  thread  leggi
thread scrivi: shareddata = 1
thread leggi: shareddata = 1
thread scrivi: shareddata = 2
thread leggi: shareddata = 2
thread scrivi: shareddata = 3
thread leggi: shareddata = 3
thread scrivi: shareddata = 4
thread leggi: shareddata = 4
thread scrivi: shareddata = 5
thread leggi: shareddata = 5
[lapegna%]
```