4 – GESTIONE DEI PROCESSI IN LINUX

- Un processo in Unix/Linux è caratterizzato da quattro sezioni logiche:

  - Codice = Istruzioni del programma
  - Dati = Dati statici
  - Heap = Dati allocati dinamicamente
  - Stack = Dati per le chiamate alla funzioni

- Quando un processo viene creato si trova assegnati i segmenti ID (numeri positivi):

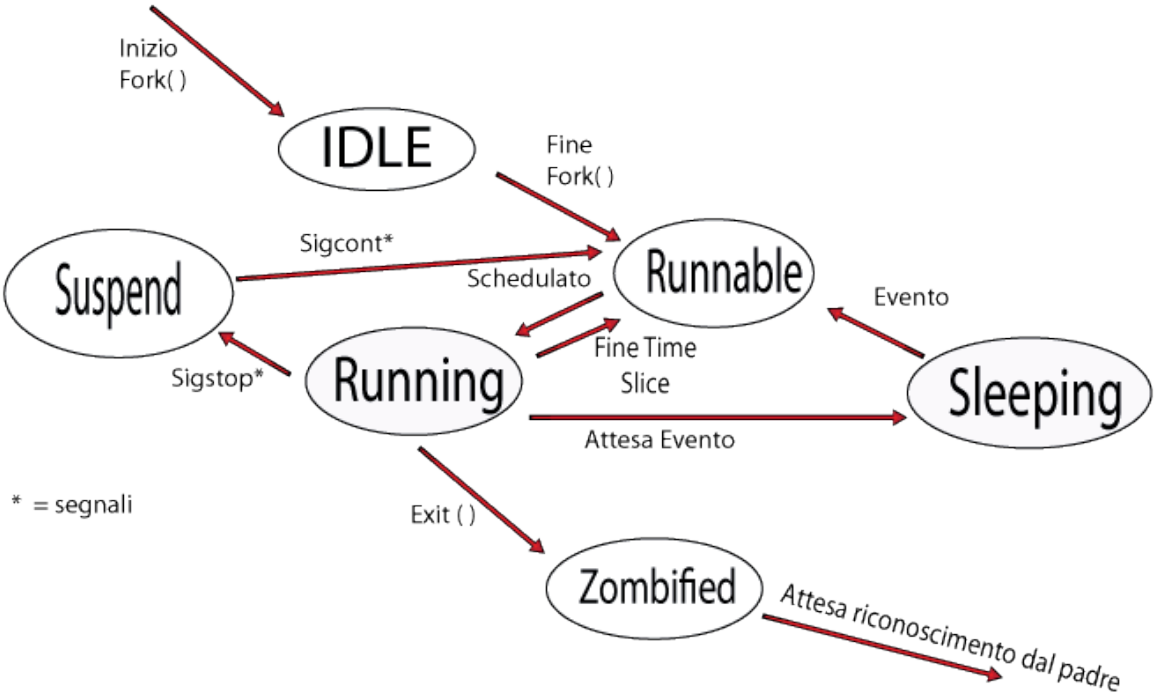
  - PID = Process-ID
  - PPID = Process-ID del Padre
  - UID = User-ID
  - UGID = User Group-ID

- All'avvio del sistema Linux, una copia del kernel viene portata in memoria centrale, inizializza le proprie strutture dati e effettua alcuni controlli; dopo ciò crea il processo **INIT** (PID = 1) che è il padre di tutti i processi successivi.
- Tutti i processi sono gestiti grazie alla **tabella dei processi** che contiene le relative informazioni; alla creazione di un processo, il kernel gli assegna una posizione in tabella, alla terminazione del processo, essa viene liberata. L'identificativo della posizione in tabella corrisponde al PID; le prime due posizioni sono riservate: 0 → kernel , 1 → INIT . La tabella ha ovviamente uno spazio finito che determina il numero massimi di processi simultaneamente (secondo il concetto di multitasking) in esecuzione.

STATI DI UN PROCESSO:

Un processo in Linux può trovarsi in uno di questi stati:

- IDLE: Processo appena creato
- RUNNABLE: Pronto per passare in stato di esecuzione
- RUNNING: In esecuzione
- SLEEPING: In attesa di un evento (Es. I/O)
- SUSPENDED: Processo "Frozen" (congelato) da un segnale
- ZOMBIFIED: Processo finito, ha rilasciato le risorse, ma è ancora presente nella tabella dei processi



- La creazione di un processo avviene con la *System call* FORK() (vedi più avanti).
- Un processo può cambiare il codice che sta eseguendo con una delle System call della famiglia *Exec()*.
- In conseguenza della fork() si ha una duplicazione in due processi detti padre e figlio. Quando un figlio termina (ad esempio con la System call exit()), il padre riceve la relativa segnalazione: se è in grado di receverla, il processo terminato viene eliminato dalla tabella processi, atrimenti rimane in *Zombie.*
  Se un padre termina prima del figlio, quest'ultimo rimane *Orfano* e viene "adottato" da INIT.

PROCESSI E AMBIENTE

- Un processo può interagire con l'ambiente (S.O.) in vari modi:
  1. con gli argomenti ARGC e *ARGV[] del main;
  2. con i file standard di I/O (stdin, stdout, stderr);
  3. con la ridirezione e la pipe;
  4. con gli exit code (stati di uscita di un processo);
  5. con l'uso delle variabili di ambiente

Esaminiamo i punti 1,4,5, considerando già noti gli altri.

1) Parametri al main.

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

- **argc** è un intero che contiene il numero di parametri forniti al programma (+1 che è il programma stesso);
- **argv** è un vettore di stringhe, ognuna delle quali è un parametro (la prima è il nome del programma).

Esempio: se lanciamo il programma "pippo" così:
./pippo A B 50
avremmo: argc = 4
argv[0] = ./pippo
argv[1] = A argv[2] = B argv[3] = 50

4) S.C void exit(int status): status = 0 → ok
   - Chiude tutti i descrittori dal processo chiamante
   - Dealloca il suo codice, dati, heap, stack.
   - Termina il processo e invia il suo codice di terminazione (fatto di 8 bit) al processo padre (o a INIT) in modo che venga accettato (altrimenti il processo rimane zombie):
   - Ovviamente exit() nn torna niente; in un main equivale a **return**.

5) Le variabili d'ambiente sono disponibili attraverso la variabile globale **environ**.

```
extern char **environ;
```

È un array di stringhe ognuna delle quali contiene una variabile.
Esempio di uso:

```
extern char **environ;

int main()
{
    char **var;
    for(var = environ; *var != NULL; var++)
        printf("%s\n", *var);
    return 0;
}
```

Esercizio: estrarre delle var. d'ambiente, nome e valore inserendoli in due vettori paralleli.

Funzioni per l'uso delle variabili di ambiente:

- *char* * getenv (*char* *): restituisce il valore della variabile passata come argomento;
  Esempio: printf("shell=%s",getenv("shell"));
- int putenv (char *s) : il parametro s è nella forma nome = valore, se la variabile non esiste la sovrascrive, altrimenti la aggiunge; Restituisce 0 se o.k.
- int setenv (const char *n, char *v, int overwrite) : imposta la variabile di nome n al valore a patto che overwrite sia non zero; restituisce 0 se o.k. ; se la chiave non c'è, la crea.
- void unsetenv (cost char *n) : rimuove la variabile n dall'ambiente.
- int clearenv () : cancella tutto; ritorna 0 se o.k. .

GESTIONE DI PROCESSI

- Individuazione:
  int getpid(); restituisce il pid del processo
  int getppid(); restituisce il pid del padre
  è più corretto usare il tuo pid_t definito in sys/types.h
- Creazione:
  La system call system():
  int system (char *comando);
  Richiede l'uso della libreria unistd.h; serve a eseguire la stringa comando come comando, generando un processo figlio.
  Restituisce il codice di uscita (exit code): 0 se tutto o.k.
  La chiamata a system() pone il chiamante in stato di attesa (del completamento del comando).
  Esempio:

```
int main()
{
    int i;
    printf("main \n");
    i=system("echo comando di shell\n");
    sleep(5);
    printf("Di nuovo main\n");
    printf("Valore di ritorno %d\n",i);
    return 0;
}
```

[Si usa sleep che è una s.c. Che mette in cattesa il programma dai secondi indicati come argomento]

La system call fork():
int fork();
richiede la libreria unistd.h. fork=biforcare.
Crea un processo figlio dal processo chiamante,del tutto identico a quest'ultimo.
Avviene quindi una duplicazione,almeno del codice del programma,mentre i dati e lo stack sono specifici per ognuno dei due processi.
La s.c. fork ritorna 0 nel processo figlio e il pid del figlio nel processo padre(o -1 in caso di insuccesso).
Oltre al codice vengono condivisi anche i file aperti; lo spazio di memoria(e quindi le variabili)è invece privato.
Dopo la fork() i due processi proseguono eseguendo lo stesso codice(presumibilmente differenziato in base al pid di ritorno)e entrano in concorrenza per l'uso della CPU.L'ordine temporale con cui vengono eseguite le istruzioni del padre o del figlio è impredicibile.
Esempio:

```
#include<stdio.h>
int main()
{
    int codrit;
    codrit=fork();
    if(codrit==0)
    {
        printf("Processo padre: id=%d,Suo padre=%d,Figlio=%d,getpid(),getppid(),codrit");
    }
    else
    {
        printf("Processo padre: id=%d,Ritorno=%d,getpid(),codrit");
    }
    printf("\nIl processo %d stampa\n",getpid());
    return 0;
}
```

Il risultato sarà:

Processo figlio: id=num,Ritorno=0
Il processo num stampa

oppure al contrario

Processo padre: id=num2,Suo padre=num3,Figlio=num
Il processo num2 stampa

– Terminazioni anomale e funzioni wait e waitpid():
  - Nel caso un processo padre termini prima del figlio,quest'ultimo,orfano,viene adottato dal processo init che riceverà la notifica di terminazione del figlio e la riconoscerà in ogni caso(evitando che diventi zombie).
  - Nel caso il figlio termini mentre il padre non può ricevere le informazioni sulla terminazione, il figlio rimane nella situazione di zombie (tutte le risorse sono state liberate eccetto la riga occupata dal processo nella tabella dei processi attivi).
    Per evitare situazioni anomale occorre fare in modo che un processo padre si ponga in attesa della terminazione di un figlio grazie alle funzioni wait() e waitpid() (vedere più avanti).

  Esempi di situazioni anomale:
  - Scrivere un programma che crei un figlio (che svolga una elaborazione qualsiasi, ad esempio un ciclo di printf) e vada in sleep per 30 secondi; In tal caso il figlio termina e diventa zombie fino al risveglio del padre. Dopo l'esecuzione verificare la situazione con il comando si shell " ps –lx | grep *nomeprogramma*" e vedere la presenza di uno "z" come stato del processo zombie.
  - Scrivere un programma che generi un figlio e lo termini; il figlio invece vada in sleep per 60 secondi (durante i quali, con ps dalla shell, si veda che è divenuto figlio di init) e poi si stampi il suo pid e il ppid (che sarà 1).
  - Scrivere un programma che riceva dalla linea di comando due argomenti interi; il programma crea un figlio che calcola il prodotto dei due numeri; il padre invece calcola la somma. Entrambi stampano il risultato. Questo esempio dimostra che i due processi condividono il codice (e i file e le variabili di input) ma hanno il ognuno il proprio spazio negli indirizzi. (oltre che i propri valori dei registri).

  Funzione Wait():

  int wait (int *status);

  pone il processo in attesa che uno qualunque dei figli termini (gestendone correttamente il termination status e quindi senza farlo diventare zombie). Status è lo stato di terminazione del figlio; il valore di ritorno è il pid del figlio (-1 se non ci sono figli).
  Se sono presenti più figli, la wait() deve essere usata più volte. In tal caso allora è nello waitpid().

  Funzione waitpid():

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

  pone in attesa il processo, in attesa che il figlio che ha PID=PID termini e fornisce lo stato di terminazione in status.
  Restituisce il PID del figlio che ha cambiato stato oppure -1 in caso di errore.
  Il campo options serve a decidere quale cambiamento di stato del figlio ci interessa attendere; di solito si mette 0 (terminazione).

Funzioni exec() e meccanismo di spawn

- execl: riceve una lista (l) variabile di argomenti, terminata con null
- execv: argomenti in un vettore di stringhe terminato da null
- execlp: come le altre due ma con il programma da eseguire presente nella directory corrente.
- execle: come execl ma con la possibilità di passare variabili di ambiente come vettori di strighe terminato da null.

Tutte necessitano di <unistd.h>

L'effetto è quello di rimpiazzare il codice del processo che chiama la funzione con quello del comando passato con argomento alla exec().

int execl (const char *path, const char *arg0, … , null);
int execv (const char *path, char *argcv[]);
int execlp (const char *file, const char *arg0, … , null);
int execvp (cont chat *file, char argv[]);
int execle (const char *path, const *arg0, …, null, char *envp[]);

n.b. exec() non crea nuovi processi; ritorna -1 in caso il comando non esista;
in caso di successo non torna niente in quanto il codice del chiamante viene completamente sostituito da quello del comando.
Gli attributi che invece vengono mantenuti dal processo dopo la exec() sono:

- pid, ppid e altri identificativi (es: vid, ...)
- descrittori dei file aperti
- dir. corrente e terminale di controllo
- maschera dei segnali e maschera dei diritti sui file (vmask)

Nelle varie versioni di exec() arg 0 o argv[0] dev' essere sempre il nome del file eseguibile corrispondente al comando.

Esempio:

```
#include<stdio.h>
int main ()
{
    printf("lista dei file\n");
    execl("/bin/ls","ls","-l",NUL
    return 0;
}
```

La fork() può servire in pratica in due scenari diversi:

1. all' interno di un programma si creano dei figli cui viene affidata l' esecuzione di certi compiti mentre il padre continua a fare altro (esempio: apache server quando è gestito a processi anziche a thread);
2. un processo vuole eseguire un altro programma e allora crea un figlio che farà subito una exec dell' altro programma.

Fork() + exec() in molti sistemi operativi viene realizzato con una chiamata denominata spawn().

In Unix-Linux i due passaggi sono stati tenuti separati perchè in molti casi è utile anche la fork() da sola (vedi 1).

Esempio di "spawn" in linux è un programma di simulazione di una shell.