Corso di Fondamenti di Informatica 2
CdL Ingegneria Informatica
Ing. Franco Zambonelli

IL SISTEMA OPERATIVO UNIX: GESTIONE DEI PROCESSI

Lucidi Realizzati in Collaborazione con:

Prof. Letizia Leonardi
Università di Modena

Prof. Antonio Corradi
Università di Bologna

Prof. Cesare Stefanelli
Università di Ferrara

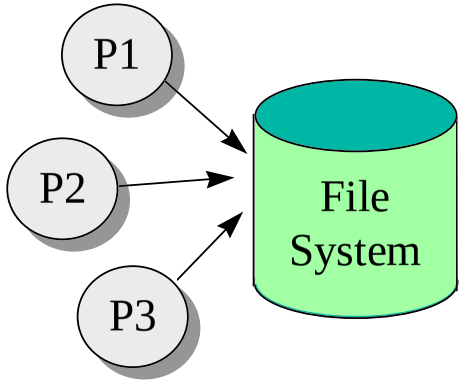
Modello di Processo in UNIX

Ogni processo ha un proprio spazio di indirizzamento **completamente** locale e **non condiviso**

➔ **Modello ad Ambiente Locale**

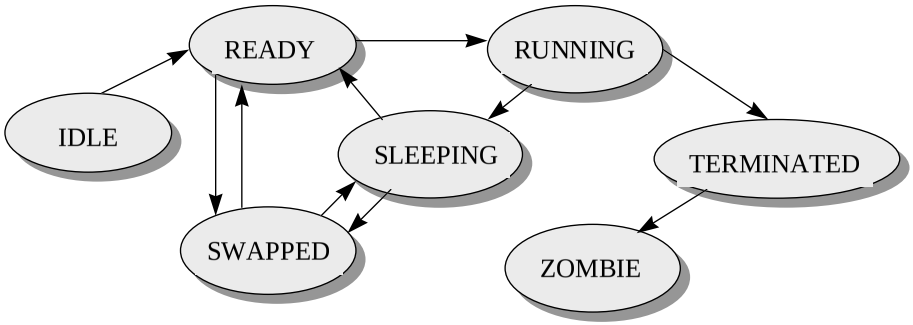
Eccezioni:

- il codice può essere condiviso
- il file system rappresenta un ambiente condiviso



Stati interni possibili di un processo UNIX

IDLE	stato iniziale
READY	pronto per l'esecuzione
RUNNING	in esecuzione
SWAPPED	immagine copiata su disco



SLEEPING	attesa di un evento per proseguire
TERMINATED	terminato
ZOMBIE	terminato ma ancora presente

Attributi di un Processo UNIX

- **pid** (**p**rocess **id**entifier)
- **ppid** (**p**arent **p**rocess **id**entifier)
- **pgid** (**p**rocess **g**roup **id**)

Un processo è lanciato da un utente, informazione di cui si tiene traccia in:

- **real uid** (real user identifier)
- **real user gid** (real user group identifier)

che corrispondono allo uid e gid dell'utente che ha lanciato il processo.

Altre informazioni:

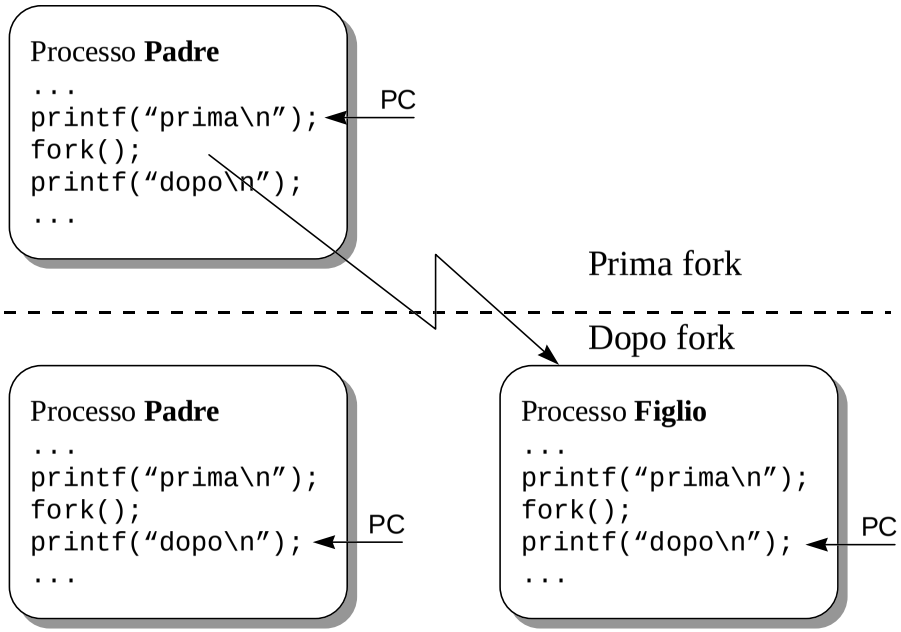
- **environment** (stringhe nome=valore)
- **current working directory**
- **dimensione massima** dei file creabili
- **maschera dei segnali**
- **controlling terminal**
- **priorità processo** (nice)

Primitive per la Gestione dei Processi

Creazione

FORK pid = **fork** ( );
pid_t pid;

Un processo ne genera un altro ➔ 2 processi concorrenti e indipendenti:
- il **parent** (processo padre), quello originario
- il **child** (processo figlio), quello generato.



Effetti della FORK

1. crea un nuovo **processo**
2. duplica i **dati** e **stack** sia **parte utente** sia **parte kernel;**
3. **stesso codice** per padre e figlio

fork restituisce l'**identificatore** del **processo creato (PID)** al padre
in caso di errore la fork restituisce al padre il valore **-1**
(limite al numero max di processi per utente e per sistema)

*Figlio eredita **tutti** attributi del processo padre, uniche **differenze** tra i due processi:*
- fork restituisce zero nel *figlio*, il pid del figlio nel *padre*
- il pid del figlio è diverso da quello del padre
- il pid del padre è diverso

NB. variabili e puntatori del padre sono *copiati* e ***non* vengono *condivisi*** da padre e figlio ma *duplicati*.

Esecuzioni differenziate del padre e del figlio

```
...
if(fork()==0) {
   ...   /* codice eseguito dal figlio */
   ...
} else {
   ...   /* codice eseguito dal padre */
   ...
}
```

Dopo la generazione del figlio il padre può decidere
se operare **contemporaneamente** ad esso
oppure
se **attendere** la sua terminazione (wait)

Esecuzione di un Programma
(primitiva EXEC)

exec trasforma il processo chiamante caricando un nuovo programma nel suo spazio di memoria

- **NON** si prevede di tornare al **programma chiamante**.
- **exec non produce nuovi processi** ma solo il cambiamento dell'ambiente del processo interessato, sia come codice, sia come dati.

Sono disponibili molte funzioni della famiglia **exec**:

exec**l**, exec**le**, exec**lp**, exec**v**, exec**ve**, exec**vp**

p ➔ la funzione prende un nome di file come argomento e lo cerca nei direttori specificati in PATH;
l ➔ la funzione riceve una lista di argomenti (NULL terminata);
v ➔ la funzione riceve un vettore argv[];
e ➔ la funzione riceve anche un vettore envp[] invece di utlizzare l'environment corrente.

Alcuni esempi di exec

EXECl **execl** (path, arg0, arg1, ..., argn, (char *) 0);
char *path, *arg0, *arg1, ..., *argn;
/* path è un pathname, assoluto o relativo*/
/* arg0 è il nome del file, seguono argomenti*/

EXECVE **execve** (path, argv, envp);
char *path, *argv[], *envp[];

Il nuovo file può essere un file eseguibile o un file di dati per un interprete shell.

Nota: In alcuni sistemi operativi, solo una exec (execve) è una system call, le altre sono chiamate di libreria che invocano la execve.

Utilizzo della primitiva exec

(differenziare comportamento del padre da quello del figlio)

```
pid = fork();

if (pid == 0){ /* figlio */
        printf("Figlio: esecuzione di ls\n");
        execl("/bin/ls", "ls", "-l", (char *)0);
        perror("Errore in execl\n");
        exit(1);
}

if (pid > 0){ /* padre */
        ...
        printf("Padre ....\n");
        exit(0);
}

if (pid < 0){ /* fork fallita */
        perror("Errore in fork\n");
        exit(1);
}
```

Il figlio esegue immediatamente una **exec** e passa a **eseguire** un altro programma

si carica il nuovo codice, i nuovi dati e tutto lo stato del nuovo programma

Si noti che il figlio **non** ritorna al suo stato precedente (GO TO)

Caratteristiche del processo dopo exec

Ci sono alcuni attributi che il processo che esegue l'exec mantiene:

- pid
- parent pid
- process gid
- session id

- real uid
- real user gid

- file mode creation mask
- current working directory
- root directory

- maschera dei segnali
- controlling terminal

- fd:
  i **file descriptor** sono ***conservati*** **MA**
  ogni descrittore di file aperto ha un flag close-on-exec,
  il default è tenerlo aperto (cambiabile con fcntl())

- Cosa succede all'effective uid ?

Sincronizzazione tra padre e figlio

WAIT pid= **wait** (&status);
int status;

```
...

if ((pid = fork()) == 0) {
   ...   /* codice eseguito dal figlio */
   ...
} else {
   ...   /* codice eseguito dal padre */
   ...
   wait(&status);
```

In caso di più figli **while (rid = wait (&status) != pid);**

Operazione di wait

Quando un processo termina, il nucleo notifica la terminazione al processo padre (mandandogli un segnale).

Il padre riceve lo stato di uscita del figlio invocando la wait.

Il processo padre che esegue la **wait:**

- ***si sospende*** se nessun processo figlio è terminato
- ***non si sospende*** se almeno un processo figlio è terminato (zombie)

Esempio di uso della WAIT

```
/*  il figlio scrive su un file;  il padre torna all'inizio e legge */
#include <stdio.h>
#include <fcntl.h>

int procfile (f1)
char  *f1;    /* file di comunicazione */
{int nread, nwrite = 0, atteso, status, fd, pid;
 char *st1 = "              ",  st2 [80];

if (fd = open (f1, O_RDWR | O_CREAT, 0644)<0) {
     perror("open"); exit(1); }

if ((pid = fork()) < 0)  {perror("fork"); exit(1);}

if (pid == 0) {  /*  FIGLIO */
     scanf  ("%s", st1);
     nwrite = write  (fd, st1, strlen(st1));
     exit (0);
}

else {    /* PADRE */
     atteso = wait (&status);     /* attesa del figlio */
     lseek (fd, 0L, 0);
     nread = read (fd, st2, 80);
     printf ("Il padre ha letto la stringa  %s\n", st2);
     close (fd);
     return (0); }
}

main  (argc, argv)  ... {  int integi;
   integi = procfile (file1);
   exit (integi);
                              }
```


Terminazione di un processo

modo *volontario*
modo *forzato*

VOLONTARIO
- operazione primitiva **"exit"** o
- alla conclusione del programma main;

EXIT

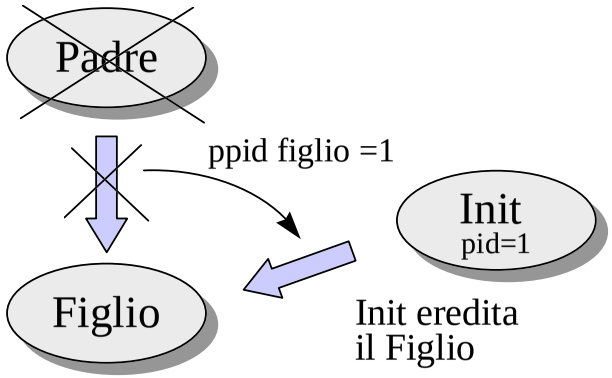
```
void  exit (status);
int status;
```

- **chiude tutti i file aperti** del processo che termina
- altre azioni tipo **scarico buffer** standard I/O library
- **status** viene passato al **processo padre**, se questo attende il processo che termina (esegue cioè la **wait**())

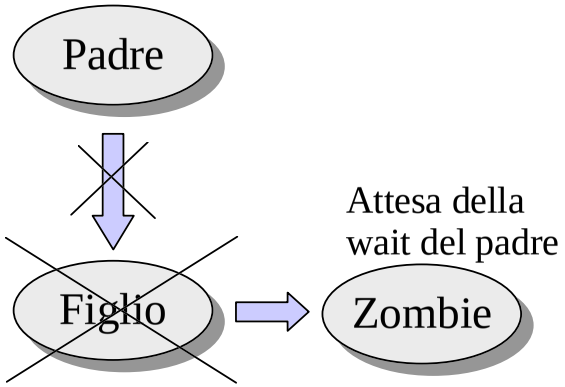
FORZATO
a seguito di:
- **azioni non consentite** (come riferimenti a indirizzi scorretti o tentativi di eseguire codici di operazioni non definite, che generano segnali “sincroni”)
- **segnali generati dall'utente** da tastiera e ricevuti dal processo, oppure
- **segnali spediti da un altro processo** tramite la system call **kill**.

Parentela Processi e Terminazione

Terminazione del Padre



Terminazione del Figlio: Processi Zombie



ESERCIZIO (esecuzione di comandi)

```
#include <stdio.h>

main  (argc, argv) {
    int stato, atteso,  pid;
    char st[80];

    for (;;) {
        if ((pid = fork()) < 0)  {perror("fork"); exit(1);}

        if (pid == 0) {  /*  FIGLIO: esegue i comandi */
            printf("inserire il comando da eseguire:\n");
            scanf  ("%s", st);
            execlp(st, (char *)0);
            perror("errore");
            exit (0);

        } else {    /* PADRE */
            atteso=wait (&stato); /*attesa figlio: sincronizzazione */
            printf ("eseguire altro comando? (si/no) \n");
            scanf  ("%s", st);
            if (strcmp(st, "si") exit(0);
        }
    }
}
```

Gestione degli errori

In caso di fallimento, le System Call ritornano **-1**

In più, UNIX assegna alla variabile globale **`errno`** il codice di errore occorso alla system call

`/usr/include/sys/errno.h` per le corrispondenze codici di errori e loro descrizione

(definire `extern int errno` nel programma)

perror( )

routine utilizzata nella gestione degli errori, stampa (su **standard error**) una stringa definita dall'utente, seguita dalla descrizione dell'`errno` avvenuto

Esempio:

```
perror("stringa descrittiva");
```

può stampare

```
stringa descrittiva: No such file or directory
```

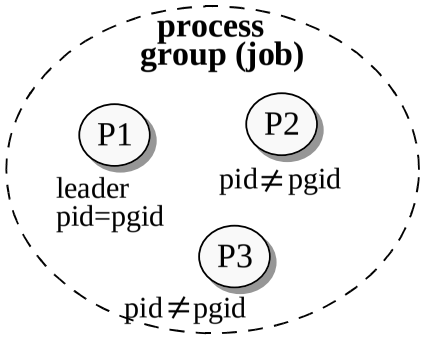

I gruppi di processi

Tutti i processi UNIX sono identificati dal **pid** e appartengono a un **gruppo** (identificato da **pgid**)

L'insieme di processi appartenenti a uno stesso gruppo è detto anche **job**

esempio:

```
cesare lia00 ~/ 10>P1 | P2 | P3
```



In un gruppo di processi c'è un solo **group leader**, che ha *pid = pgid*

Uso dei gruppi di processi :

- **segnali** mandati a tutti i processi in un gruppo (es. segnali da terminale)
- shell con **job control**

Gruppi di processi in foreground (ricevono input da tastiera e segnali, es. il ctrl-C) e in background.

I File e la primitiva FORK

variabili e puntatori del processo padre sono *copiati* e ***non*** vengono ***condivisi*** da padre e figlio ma ***duplicati***.

I **fd** vengono ***duplicati*** ➔ I **file aperti** dal processo padre sono aperti anche per il figlio

Attenzione: in questo caso, ogni puntatore a file (**I/O pointer**) è ***condiviso*** tra padre e figlio.

I/O pointer ***si sposta*** per entrambi in seguito a letture o scritture eseguite dal padre o dal figlio.