

Appunti di sistemi operativi


Appunti per il corso universitario di sistemi operativi, riferito a sistemi Unix/Windows.
Si discute su problemi di sincronizzazione, memoria e scheduling dei processi.


LogMeIn Pro²


Accedi al tuo PC ovunque ti trovi


Con LogMeIn Pro², puoi:

- Condividere file e foto con altri
- Stampare documenti a distanza

 **Stampa**

 **Condividi**

 **Ascolta**

 **Incontra**

Provalo gratis

ARGOMENTI

[INTRODUZIONE](#)

[INPUT/OUTPUT](#)

[GESTIONE DEI PROCESSI](#)

[ALGORITMI DI SCHEDULING](#)

[SCHEDULING MULTI CPU](#)

[SISTEMI REAL TIME](#)

[SCHEDULING SU LINUX](#)

[SCHEDULING SU WINDOWS](#)

[OPERAZIONI SUI PROCESSI](#)

[COMUNICAZIONE TRA PROCESSI](#)

[THREAD](#)

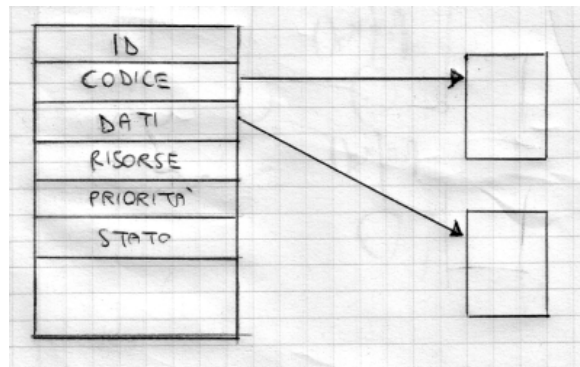
[SINCRONIZZAZIONE TRA PROCESSI](#)

[GESTIONE MEMORIA](#)

OPERAZIONI SUI PROCESSI

CREAZIONE

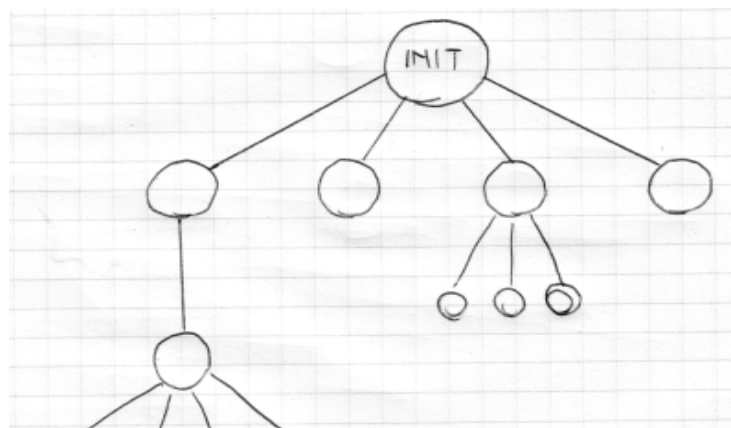
- viene creato un nuovo PD (process descriptor)



nel sistema ogni processo ha sempre un **processo padre** (non è l'utente a creare direttamente i processi, ma interagisce con i processi per crearne altri)

si crea quindi un albero di processi

al momento dello shutdown della macchina, il processo padre si occupa di terminare prima tutti i processi figli -> si chiudono tutti i processi dell'albero (**terminazione a cascata**)



la presenza di un legame tra i processi è evidente provando a eseguire un processo P dalla shell di linux: P non permette di usare il terminale fino alla sua terminazione, e se chiudo il terminale, si chiude anche P; la stessa cosa accade anche se eseguo il processo con il comando <nome processo>& (anche se ci viene restituito il controllo)

DEMONI

sono **processi di sistema** che rimangono attivi finché il sistema non viene chiuso; al momento della loro creazione questi vengono dissociati dal processo padre e “adottati” dal processo radice: non possono infatti rimanere orfani allo shutdown del sistema (altrimenti non potrebbero terminare regolarmente)

comandi utili per visualizzare parentele: ps axjf (vedi man ps per maggiori informazioni)

RELAZIONI DI CONTENUTO

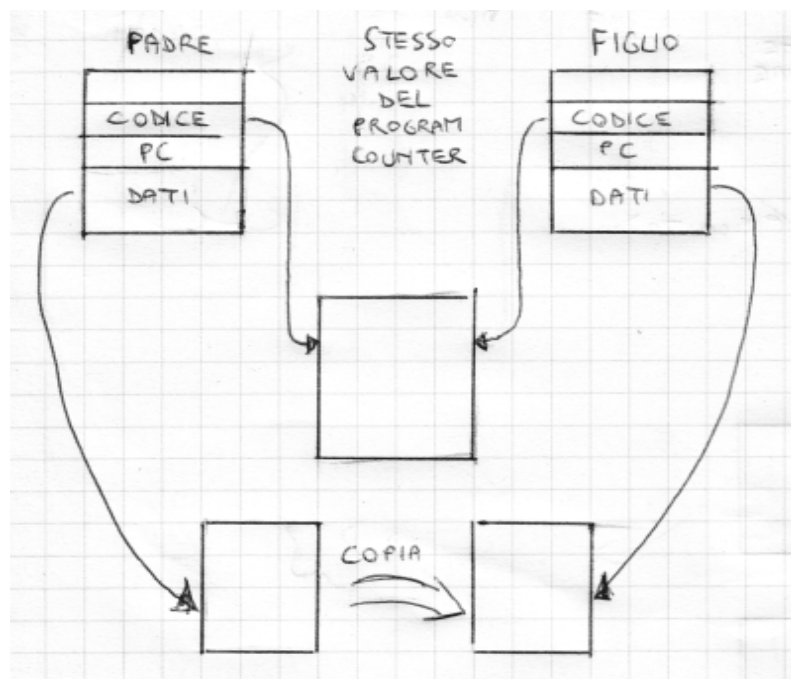
ho 2 possibilità:

- figlio è una **copia** del padre (ossia è composto da una copia degli indirizzi del padre; es. **unix** si comporta così di default, ma ci sono sistemi per ottenere un processo diverso)
- figlio è un **processo diverso** (es. **windows**)

UNIX

in unix la generazione è più a basso livello, più passo-passo

fork() = chiamata a sistema che crea un processo figlio



a questo punto ho 2 processi uguali, cosa me ne faccio?

```

pid = getpid();
fork()
if (getpid()==pid)
<padre>
else
<figlio>

```

a questo punto abbiamo due processi che eseguono lo stesso codice, per differenziarli si utilizza la chiamata `execvp`, che sostituisce lo spazio di memoria del processo chiamante con un nuovo programma

quando la `fork()` fallisce?

- può essere finita la memoria
- può essere finita la memoria del kernel (che non deve utilizzare tanta memoria per essere efficiente, non può toglierla al resto)

la `fork` ritorna:

- `< 0` in caso di fallimento
- `= 0` se mi trovo nel processo figlio
- `> 0` se mi trovo nel processo padre

versione rivista del codice precedente:

```

f = fork();
if (f < 0)
<errore>
else if (f == 0)
<figlio>
else
<padre>

```

codice:

```

f = fork();
if (f < 0)
    printf("Fallimento\n");

```

```

        printf("fallimento\n");
    else if(f == 0)
        printf("sono il processo figlio\n");
    else
        printf("Sono il processo padre e ho creato %d\n", f);
    printf("Termino pid = %i\n", getpid());

```

output provato: interessante il fatto di come si comporta lo scheduler con bash (per questo root@www cambia posizione)

```

root@www:~/prova# ./a.out
Sono il processo padre e ho creato 8657
Termino pid = 8656
root@www:~/prova# sono il processo figlio
Termino pid = 8657

```

```

root@www:~/prova# ./a.out
Sono il processo padre e ho creato 8667
sono il processo figlio
Termino pid = 8667
Termino pid = 8666

```

comportamento in caso di fallimento:

```

> test

fallimento

termino pid = 1812

```

comportamento in caso di riuscita:

```

> test

Sono il processo padre e ho creato 1813

termino pid 1812

termino pid 1813

```

ROBA CHE IN COMPITO SBAGLIANO TUTTI

```

fork()

fork()

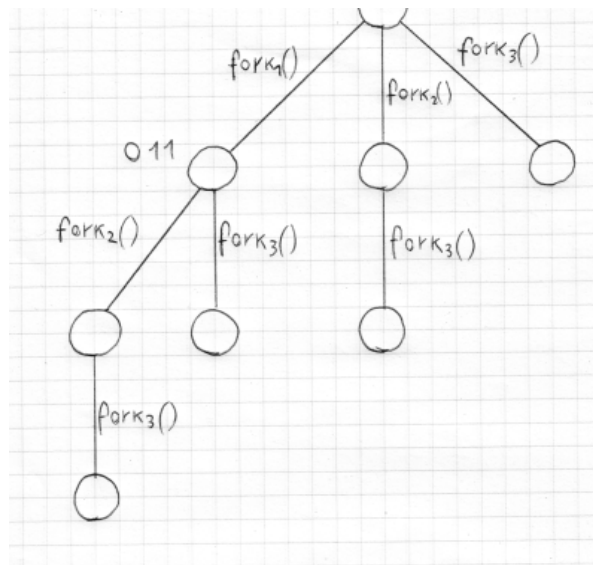
fork()

printf("%i %i %i", (f1 > 0)(f2 > 0)(f3 > 0));

```

quanti processi genera?

8



verranno tutti numeri binari di 3 bit, e tutti gli 8 processi si differenziano dagli altri per 1 bit

10 fork in fila genereranno 2^{10} processi (provando il codice l'ordine di stampa cambia ad ogni esecuzione, perché entra in gioco lo scheduling)

Es. realizzazione di una shell in linux

```

char comando[128];

int esito;

while (TRUE) {
    printf(">");
    scanf("%s", comando);
    esito = fork();
    if (esito < 0) <errore>

    if (esito == 0) {
        execl(comando, comando, NULL); // mettiamo NULL per semplicità,
                                         in realtà dovrebbe esserci una lista
                                         di parametri del comando (es. -h)

        <errore exec>
    }
}

```

se dopo un errore non termina il processo due shell vengono eseguite in maniera simultanea (quindi appaiono 2 cursori)

> pippo (exec fallisce)

> (padre)

> (figlio)

WINDOWS...

windows ha un approccio diverso: la fork NON ESISTE in windows (possiamo comunque ottenere gli effetti della fork clone con altri metodi):

- `execl (file, arg1, arg2, ..., argn, 0) = execute and leave: execl(file, argv);` eseguono il

- `execv(file, argv, argv2, ..., argvn, 0) = execute and leave, execv(file, argv)`. Eseguito il programma <file>

la `execv` può fallire, ad esempio se il file non c'è; se ha successo il codice

se ad esempio

TERMINAZIONE DEI PROCESSI

generalmente un processo termina quando ha eseguito la sua ultima istruzione e chiede al sistema operativo di essere cancellato usando la chiamata `exit()`

a questo punto il sistema operativo:

- rilascia le risorse del processo
- elimina il PD (process descriptor)
- solitamente segnala la terminazione al processo padre

UNIX

abbiamo 2 chiamate a sistema:

- **`exit(int stato)`** = termina il processo attivo; gli passiamo un numero intero per indicare lo stato della terminazione (`EXIT_FAILURE`; `EXIT_SUCCESS`; in genere corrispondono ad 1 e 0); tale numero viene restituito dopo la terminazione del processo in cui è contenuta la `exit`
- **`process_id wait (int *stato)`** = forza un processo padre ad aspettare che un processo figlio si fermi oppure termini; ritorna l'identificatore (il PID) del processo terminato (oppure -1 in caso di errore), `stato` è un vettore di bit che contiene anche altri bit di stato, che descrivono come è terminato il processo)
- macro che mi fanno gestire il valore di `stato` (ossia di verificare come se il programma è terminato in maniera regolare o meno): `WIFEXITED (stato)` processo è terminato tramite `exit`; `WIFEXITSTATUS (stato)`

es. di processo che fornisce informazioni in caso di errore:

```
char comando[128];
int esito;

while(TRUE) {
    printf(">");
    scanf("%s", comando);
    esito = fork();
    if (esito < 0) <errore>

    if (esito == 0) {
        execl(comando, comando, NULL);
        perror("errore exec"); // printerror = stampa la stringa e stampa di fianco
                               // l'ultimo errore avvenuto (es. "file not found")
        exit(EXIT_FAILURE);
    }
}
```

se voglio far sì che la shell aspetti la fine dell'esecuzione del comando prima di continuare la sua devo utilizzare `wait` (altrimenti proseguono entrambe regolate dallo scheduler); ecco un'esempio di shell leggermente più complesso:

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <wait.h>
#include <string.h>

int main(void){
    char comando[128];
    char completo[128];
    int esito;
    int pid;
    int stato;

    while(1){
        printf(" > ");
        scanf("%s", comando);

        // sistemiamo il path (per evitare di dover scrivere sempre
        // l'indirizzo dei comandi da eseguire)

        strcpy(completo, "/bin/");
        strcat(completo, comando);

        esito = fork();

        if (esito < 0)
            printf("errore\n");

        if (esito == 0){
            execl(completo, 0);
            // stampa l'ultimo errore avvenuto
            perror("errore di esecuzione");
            exit(EXIT_FAILURE);
        }
        else{
            pid = wait(&stato);
            // controlla se il figlio ha terminato correttamente la sua esecuzione
            if (WIFEXITED(stato))
                printf("%d uscito correttamente\n", pid);
            else
                printf("Qualcosa è andato storto\n");
        }
    }
}

```

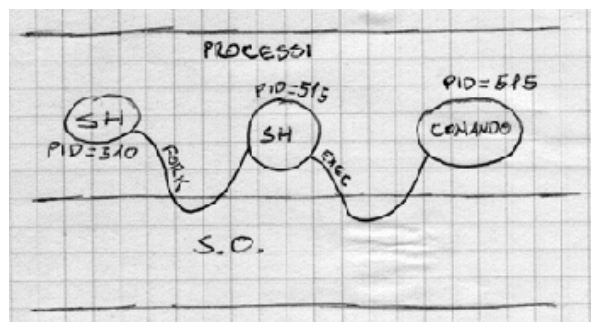
output approssimativo:

```

> ls      (attende la terminazione)
>
pippo
pluto
>

```

cosa accade durante l'esecuzione di un comando?



padre crea un nuovo processo figlio il quale a sua volta fa un exec; quindi mi trovo 2 processi che stanno eseguendo lo stesso codice sugli stessi dati, ma facendo l'exec si differenziano e il secondo diventa un **comando** (la fork crea un nuovo processo, la exec sostituisce i dati e il codice di quel processo, che resta però con lo stesso PID, ossia non è un ulteriore nuovo processo); ogni comando che eseguo dalla shell è un processo nuovo, la sequenzialità è simulata dalla wait

a seconda dell'ambito posso avere diversi tipi di processi

[continua..](#)

[Ritorna sopra](#) | [Home page](#) | [Xelon](#)