1. PARALLELISMO E PROCESSI

1.1 La necessità del parallelismo

Un programma eseguibile, generato ad esempio da un programma C, è rigorosamente sequenziale, nel senso che viene eseguito una istruzione alla volta e, dopo l'esecuzione di un'istruzione, è univocamente determinata la prossima istruzione da eseguire.

In base a quanto noto finora, l'esecuzione di N programmi da parte di un calcolatore dovrebbe anch'essa essere rigorosamente sequenziale, in quanto, dopo avere eseguito la prima istruzione di un programma dovrebbero essere eseguite tutte le successive fino alla fine del programma prima di poter iniziare l'esecuzione della prima istruzione del programma successivo. Questo modello sequenziale è molto comodo per il programmatore, perchè nella scrittura del programma egli sa che non ci saranno "interferenze" da parte di altri programmi, in quanto gli altri programmi che verranno eseguiti dallo stesso esecutore saranno eseguiti o prima o dopo l'esecuzione del programma considerato.

Tuttavia, il modello di esecuzione sequenziale non è adeguato alle esigenze della maggior parte dei sistemi di calcolo; questa inadeguatezza è facilmente verificabile pensando ai seguenti esempi:

- server WEB: un server WEB deve poter rispondere a molti utenti contemporaneamente; non sarebbe accettabile che un utente dovesse attendere, per collegarsi, che tutti gli altri utenti si fossero già scollegati
- calcolatore multiutente: i calcolatori potenti vengono utilizzati da molti utenti contemporaneamente; in particolare, i calcolatori centrali dei Sistemi Informativi (delle banche, delle aziende, ecc…) devono rispondere contemporaneamente alle richieste di moltissimi utilizzatori contemporanei
- applicazioni multiple aperte da un utente: quando un utente di un normale PC tiene aperte più applicazioni contemporaneamente esistono diversi programmi che sono in uno stato di esecuzione già iniziato e non ancora terminato

In base ai precedenti esempi risulta necessario un modello più sofisticato del

sistema; si osservi che tale modello deve garantirci due obiettivi tendenzialmente contrastanti:

- fornire parallelismo, cioè permettere che l'esecuzione di un programma possa avvenire senza attendere che tutti gli altri programmi in esecuzione siano già terminati;
- garantire che ogni programma in esecuzione sia eseguito esattamente come sarebbe eseguito nel modello sequenziale, cioè come se i programmi fossero eseguiti uno dopo l'altro, evitando "interferenze" indesiderate tra programmi diversi (vedremo che in alcuni casi esistono anche interferenze "desiderate", ma per ora è meglio non considerarle).

1.2 La nozione di processo

Per ottenere un comportamento del sistema che soddisfa gli obiettivi indicati sopra la soluzione più comune è quella rappresentata in figura 1.1, nella quale si vede che sono stati creati tanti "esecutori" quanti sono i programmi che devono essere eseguiti in parallelo. Nel contesto del SO LINUX (e in molti altri) gli esecutori creati dinamicamente per eseguire diversi programmi sono chiamati **Processi**. I processi devono essere considerati degli esecutori completi, e quindi la struttura di figura 1.1 risponde in maniera evidente ad ambedue i requisiti contrastanti definiti al precedente paragrafo: al primo requisito, perchè diversi processi eseguono diversi programmi in parallelo, cioè senza che uno debba attendere la terminazione degli altri, e al secondo requisito, perchè, essendo i diversi processi degli esecutori indipendenti tra loro, non c'è interferenza tra i diversi programmi (come se fossero eseguiti su diversi calcolatori).

I processi possono essere considerati come dei calcolatori o macchine **virtuali**, nel senso che sono calcolatori realizzati dal software (sistema operativo) e non esistono in quanto Hardware, anche se ovviamente il SO ha a sua volta bisogno di essere eseguito da un calcolatore reale. La definizione dei processi come macchine virtuali non contraddice, ma estende, la precedente definizione dei processi come programmi in esecuzione. In effetti, ogni processo deve possedere ad ogni istante un unico programma in esecuzione; pertanto, la comunicazione tra due processi coincide con la comunicazione tra i due corrispondenti programmi in esecuzione.

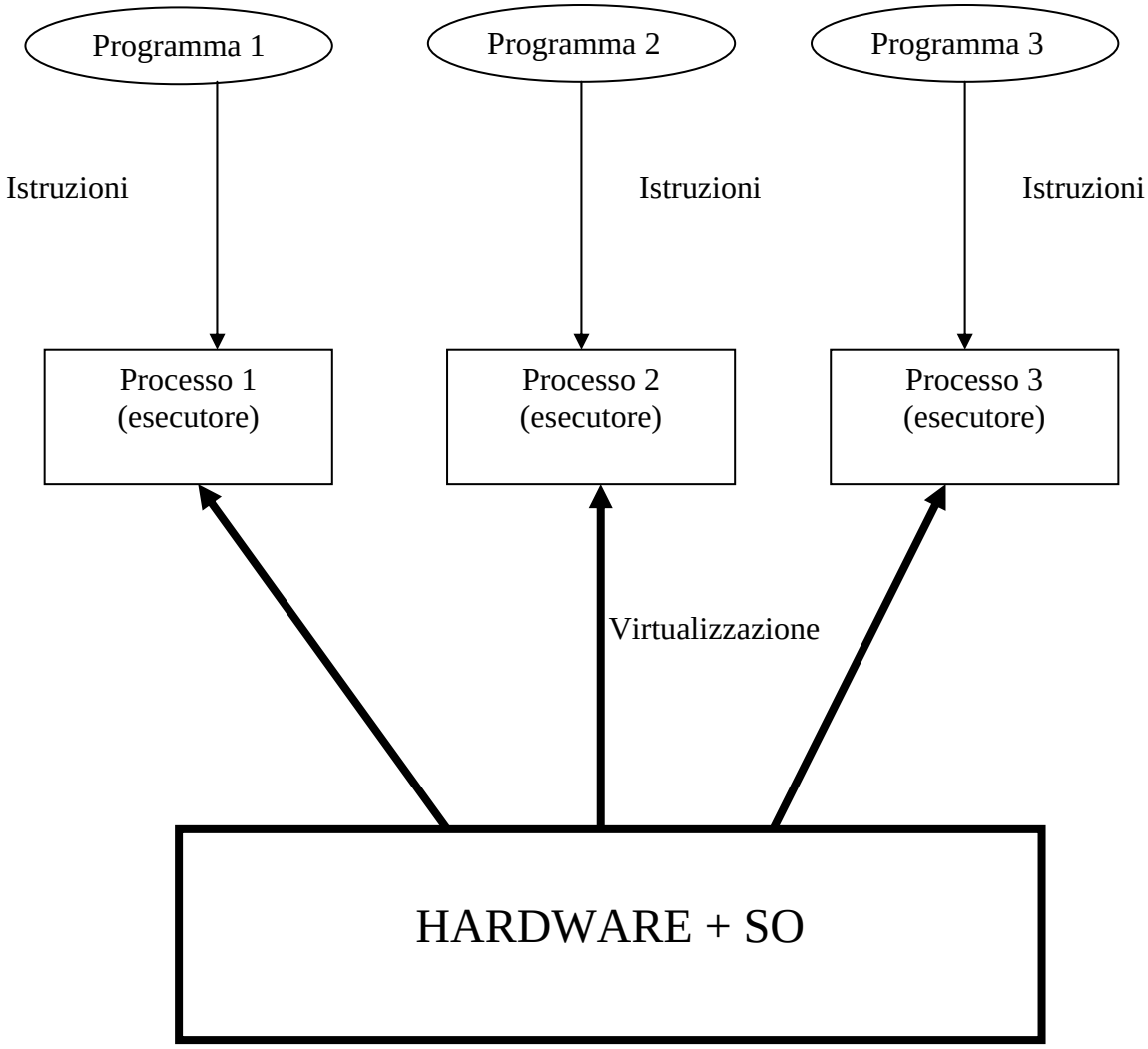


Figura 1.1 – Processi e programmi eseguiti dai processi

Tuttavia, la nozione di processo come macchina virtuale è più ampia e mette in luce alcune importanti caratterisitche del processo che estendono la normale nozione di programma in esecuzione; in particolare:

- il processo possiede delle risorse, per esempio una memoria, dei file aperti, un terminale di controllo, ecc…
- il programma eseguito da un processo può essere sostituito senza annullare il processo stesso, ovvero un processo può passare dall'esecuzione di un programma all'esecuzione di un diverso programma (attenzione, questo

passaggio è sequenziale, cioè il primo programma viene totalmente abbandonato per passare al nuovo);

Si osservi che, dato che il processo rimane lo stesso anche se cambia il programma da esso eseguito, le risorse del processo non si annullano quando si passa all'esecuzione di un nuovo programma e quindi il nuovo programma potrà utilizzarle; ad esempio, il nuovo programma lavorerà sullo stesso terminale di controllo del programma precedente.

Sorge però una domanda: come è possibile realizzare diversi processi che procedono in parallelo se l'Hardware del calcolatore è uno solo ed è sequenziale? A questa domanda si risponderà trattando la struttura interna del SO; per ora basta tenere presente che il SO è in grado di virtualizzare diversi processi indipendenti e che dal punto di vista del programmatore quello di figura 4.1 è il modello di riferimento da utilizzare.

1.3 Caratteristiche generali dei processi

Tutti i processi sono identificati da un apposito identificatore (PID = Process Identifier).

Tutti i processi (ad eccezione del primo, il processo "init", creato all'avviamento dal SO) sono creati da altri processi. Ogni processo, a parte il processo init, possiede quindi un processo padre (parent) che lo ha creato.

La memoria di ogni processo è costituita (dal punto di vista del programmatore di sistema) da 3 parti (dette anche "segmenti"):

1. il **segmento codice** (text segment): contiene il codice eseguibile del programma;
2. il **segmento dati** (user data segment): contiene tutti i dati del programma, quindi sia i dati statici, sempre presenti, sia i dati dinamici, che a loro volta si dividono in dati allocati automaticamente in una zona di memoria detta **pila** (variabili locali delle funzioni) e dati dinamici allocati esplicitamente dal programma tramite la funzione "malloc( )" in una zona di memoria detta **heap**.
3. il **segmento di sistema** (system data segment): questo segmento contiene dati che non sono gestiti esplicitamente dal programma in esecuzione, ma dal SO; un esempio è la "tabella dei files aperti" del processo, che contiene

i riferimenti a tutti i file e a tutti i socket che il processo ha aperto durante l'esecuzione e quindi permette al SO di eseguire i servizi che il programma richiede con riferimento (tramite il descrittore) ai file e ai socket.

Il sistema operativo LINUX mette a disposizione del programmatore un certo numero di servizi di sistema che permettono di operare sui processi. I principali servizi che tratteremo in questo testo consentono di:

- generare un processo **figlio** (child), che è copia del processo **padre** (parent) in esecuzione;
- attendere la terminazione di un processo figlio;
- terminare un processo figlio restituendo un codice di stato (di terminazione) al processo padre;
- sostituire il programma eseguito da un processo, cioè il segmento codice e il segmento dati del processo, con un diverso programma.

1.4 Generazione e terminazione dei processi: le funzioni fork ed exit

La funzione **fork** crea un processo figlio identico al processo padre; il figlio è infatti una copia del padre all'istante della fork. Tutti i segmenti del padre sono duplicati nel figlio, quindi sia il codice e le variabili (segmenti codice e dati), sia i file aperti e i socket utilizzati (segmento di sistema) sono duplicati nel figlio.

L'unica differenza tra figlio e padre è il valore restituito dalla funzione fork stessa:

- nel processo padre la funzione fork restituisce il valore del pid del processo (figlio) appena creato
- nel processo figlio la funzione fork restituisce il valore 0

In questo modo dopo l'esecuzione di una fork è possibile sapere se siamo nel processo padre oppure nel figlio interrogando tale valore.

Prototipo della funzione *fork*.

pid_t fork( ) (pid_t è un tipo predefinito)

Il risultato restituito dalla fork assume dopo l'esecuzione il seguente valore:

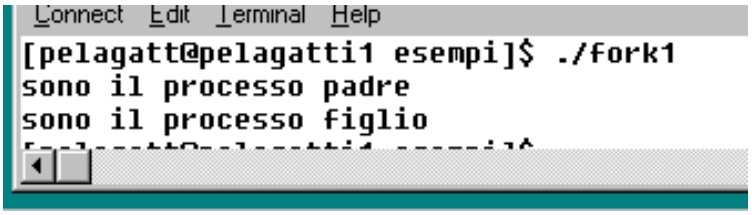
- 0 nel processo figlio;
- diverso da 0 nel processo padre; normalmente tale valore indica il pid del

figlio, tranne -1, che indica un errore e quindi che la fork non è stata eseguita;

La funzione **exit** pone termine al processo corrente (più avanti vedremo che exit può avere un parametro). Un programma può terminare anche senza una invocazione esplicita della funzione exit; in tal caso exit viene invocata automaticamente dal sistema (questo comportamento è simile a quello del comando return in una funzione C: esso permette la terminazione della funzione in un punto qualsiasi, ma una funzione può anche terminare raggiungendo la fine, senza return esplicita)

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

void main( )
{       pid_t pid;

        pid=fork( );

        if (pid==0)
                {printf("sono il processo figlio\n");
                 exit( );
                 }
        else    {printf("sono il processo padre\n");
                 exit( ); /* non necessaria */
                 }
}
```

a)il programma fork1



b)risultato dell'esecuzione di fork1

Figura 1.2

In figura 1.2 è mostrato un programma che utilizza i servizi fork ed exit e il risultato della sua esecuzione. Si noti che *l'ordine nel quale sono state eseguite le due printf è casuale; dopo una fork può essere eseguito per primo sia il processo padre*

che il processo figlio. Costituisce un grave errore di programmazione ipotizzare un preciso ordine di esecuzione di due processi, perché essi evolvono in parallelo.

Si osservi che, a causa della struttura particolarmente semplice dell'esempio le due exit non sarebbero necessarie, perché comunque il programma terminerebbe raggiungendo i punti in cui sono le exit.

Infine, si osservi che ambedue i processi scrivono tramite printf sullo stesso terminale. Questo fatto è un esempio di quanto asserito sopra relativamente alla duplicazione del segmento di sistema nell'esecuzione di una fork: dato che la funzione printf scrive sullo standard output, che è un file speciale, e dato che la tabella dei file aperti è replicata nei due processi in quanto appartenente al segmento di sistema, le printf eseguite dai due processi scrivono sullo stesso standard output.

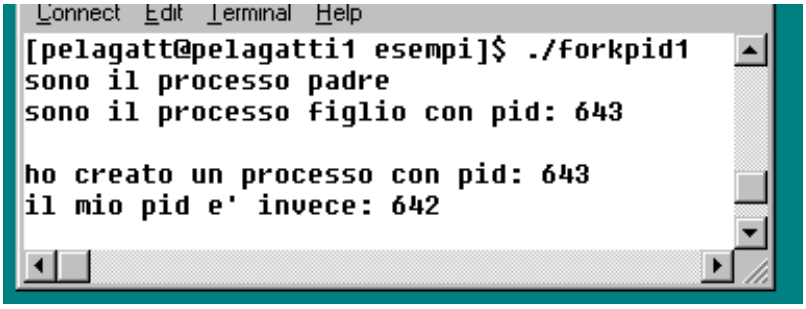
Possiamo arricchire l'esempio precedente stampando il valore del pid dei due processi padre e figlio; a questo scopo possiamo utilizzare una funzione di libreria che restituisce al processo che la invoca il valore del suo pid. Tale funzione si chiama **getpid** ed ha il seguente prototipo:

```
pid_t getpid( )
```

In figura 1.3 sono riportati il testo ed il risultato dell'esecuzione del programma forkpid1, che è simile al programma fork1 ma stampa i pid dei processi coinvolti. Si osservi che le printf eseguite dai due processi sono mescolate tra loro in ordine casuale, per i motivi discussi sopra. Per interpretare il risultato è quindi necessario osservare bene il testo delle printf nel codice.

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

void main( )
{       pid_t pid,miopid;

        pid=fork( );
        if (pid==0)
                {miopid=getpid( );
                 printf("sono il processo figlio con pid: %i\n\n",miopid);
                 exit( );
                 }
        else    {printf("sono il processo padre\n");
                 printf("ho creato un processo con pid: %i\n", pid);
                 miopid=getpid( );
                 printf("il mio pid e' invece: %i\n\n", miopid);
                 exit( ); /* non necessaria */
                 }
}
```

a)il programma forkpid1



b)risultato dell'esecuzione di forkpid1

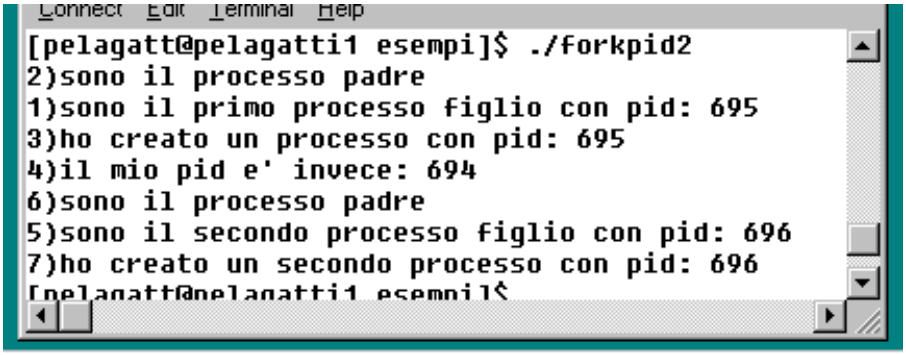
Figura 1.3

Un processo può creare più di un figlio, e un figlio può a sua volta generare dei figli (talvolta si usa la dizione di processi "nipoti" per i figli dei figli), ecc… Si viene a creare in questo modo una struttura gerarchica tra processi in cima alla quale è situato, come già detto, il primo processo generato dal sistema operativo durante l'inizializzazione.

In figura 1.4 sono riportati il testo e il risultato dell'esecuzione del programma forkpid2, che estende forkpid1 facendo generare al processo padre un secondo figlio. Dato che i 3 processi scrivono sullo stesso terminale in ordine casuale, per rendere il risultato più intelligibile tutte le stampe sono state numerate.

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

void main( )
{       pid_t pid,miopid;

        pid=fork( );

        if (pid==0)
                {miopid=getpid( );
                 printf("1)sono il primo processo figlio con pid: %i\n",miopid);
                 exit( );
                 }
        else    {printf("2)sono il processo padre\n");
                 printf("3)ho creato un processo con pid: %i\n", pid);
                 miopid=getpid( );
                 printf("4)il mio pid e' invece: %i\n", miopid);
                 pid=fork( );
                 if (pid==0)
                         {miopid=getpid( );
                          printf("5)sono il secondo processo figlio con pid: %i\n",miopid);
                          exit;
                         }
                 else {printf("6)sono il processo padre\n");
                          printf("7)ho creato un secondo processo con pid: %i\n", pid);
                          exit( ); /* non necessaria */
                          }
                 }
}
```

a)il programma forkpid2



b)risultato dell'esecuzione di forkpid2

Figura 1.4

1.5 Attesa della terminazione e stato restituito da un processo figlio: la funzione wait e i parametri della funzione exit

La funzione **wait** sospende l'esecuzione del processo che la esegue ed attende la terminazione di un qualsiasi processo figlio; se un figlio è terminato prima che il padre esegua la wait, la wait nel padre si sblocca immediatamente al momento dell'esecuzione.

Prototipo della funzione *wait*

```
pid_t wait(int *)
```

Esempio di uso:

```
pid_t pid;
int stato;
pid = wait(&stato);
```

Dopo l'esecuzione la variabile pid assume il valore del pid del figlio terminato; la variabile stato assume il valore del codice di terminazione del processo figlio. Tale codice contiene una parte (gli 8 bit superiori) che può essere assegnato esplicitamente dal programmatore tramite la funzione exit nel modo descritto sotto; la parte restante è assegnata dal sistema operativo per indicare particolari condizioni di terminazione (ad esempio quando un processo viene terminato a causa di un errore).

Prototipo della funzione *exit*

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

Esempio: exit(5)

termina il processo e restituisce il valore 5 al padre

Se il processo che esegue la exit non ha più un processo padre (nel senso che il processo padre è terminato prima), lo stato della exit viene restituito all'interprete comandi.

Dato che il valore restituito dalla exit è contenuto negli 8 bit superiori, lo stato ricevuto dalla wait è lo stato della exit moltiplicato per 256.

In figura 1.5 sono riportati il testo e il risultato dell'esecuzione del programma forkwait1, che crea un processo figlio e pone il processo padre in attesa della terminazione di tale figlio. Il processo figlio a sua volta termina con una exit restituendo il valore di stato 5. Dopo la terminazione del figlio il padre riprende

l'esecuzione e stampa l'informazione ottenuta dalla wait. Si osservi che per stampare correttamente il valore dello stato restituito dal figlio è necessario dividerlo per 256 e che il padre riceve anche il pid del figlio terminato; quest'ultimo dato è utile quando un processo padre ha generato molti figli e quindi ha bisogno di sapere quale dei figli è quello appena terminato.

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

void main( )
{       pid_t pid, miopid;
        int stato_exit, stato_wait;

        pid=fork( );
        if (pid==0)
                { miopid=getpid( );
                  printf("sono il processo figlio con pid %i \n", miopid);
                  printf("termino \n\n");
                  stato_exit=5;
                  exit(stato_exit);
                }
        else
                { printf("ho creato un processo figlio \n\n");
                  pid=wait (&stato_wait);
                  printf("terminato il processo figlio \n");
                  printf("il pid del figlio e' %i, lo stato e' %i\n",pid,stato_wait/256);
                }
}
```

a)il programma forkwait1

```
Connect  Edit  Terminal  Help
[pelagatt@pelagatti1 esempi]$ ./forkwait1
ho creato un processo figlio
sono il processo figlio con pid 769
termino


terminato il processo figlio
il pid del figlio e' 769, lo stato e' 5
[pelagatt@pelagatti1 esempi]$
```

b)risultato dell'esecuzione di forkwait1

Figura 1.5

Una variante di wait: la funzione **waitpid**

Esiste una variante di wait che permette di sospendere l'esecuzione del processo padre in attesa della terminazione di uno specifico processo figlio, di cui viene fornito il pid come parametro. Per la sintassi di waitpid si veda il manuale.

1.6 Sostituzione del programma in esecuzione: la funzione *exec*

La funzione **exec** sostituisce i segmenti codice e dati (di utente) del processo corrente con il codice e i dati di un programma contenuto in un file eseguibile specificato, ma il segmento di sistema non viene sostituito, quindi ad esempio i file aperti rimangono aperti e disponibili.

Il processo rimane lo stesso e mantiene quindi lo stesso pid.

La funzione exec può passare dei parametri al nuovo programma che viene eseguito. Tali parametri possono essere letti dal programma tramite il meccanismo di passaggio standard dei parametri al main(argc, argv).

Esistono molte varianti sintattiche di questo servizio, che si differenziano nel modo in cui vengono passati i parametri.

La forma più semplice è la execl, descritta di seguito.

Prototipo della funzione *execl*

*execl (char *nome_programma, char *arg0, char *arg1, …NULL);*

Il parametro nome_programma è una stringa che deve contenere l'identificazione completa (pathname) di un file eseguibile contenente il nuovo programma da lanciare in esecuzione.

I parametri arg0, arg1, ecc… sono puntatori a stringhe che verranno passate al main del nuovo programma lanciato in esecuzione; l'ultimo puntatore deve essere NULL per indicare la fine dei parametri.

Ricevimento di parametri da parte di un main( )

Prima di analizzare come *exec* passa i parametri al main è opportuno richiamare come in un normale programma C si possano utilizzare tali parametri.

Quando viene lanciato in esecuzione un programma main con la classica intestazione

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

il significato dei parametri è il seguente:

argc contiene il numero dei parametri ricevuti

argv è un vettore di puntatori a stringhe, ognuna delle quali è un parametro.

Per convenzione, il parametro argv[0] contiene il nome del programma lanciato in esecuzione.

Quando si lancia in esecuzione un programma utilizzando la shell, facendolo seguire da eventuali parametri, come nel comando

>gcc nomefile

la shell lancia in esecuzione il programma gcc e gli passa come parametro la stringa “nomefile”. Il programma gcc trova in argv[1] un puntatore a tale stringa e può quindi sapere quale è il file da compilare.

Modalità di passaggio dei parametri al Main da parte di exec

Abbiamo già visto che quando viene lanciato in esecuzione un programma dotato della classica intestazione

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

il parametro argc è un intero che contiene il numero di parametri ricevuti e il parametro argv[ ] è un vettore di puntatori a stringhe. Ognuna di queste stringhe è un parametro. Inoltre, per convenzione, il parametro argv[0] contiene sempre il nome del programma stesso.

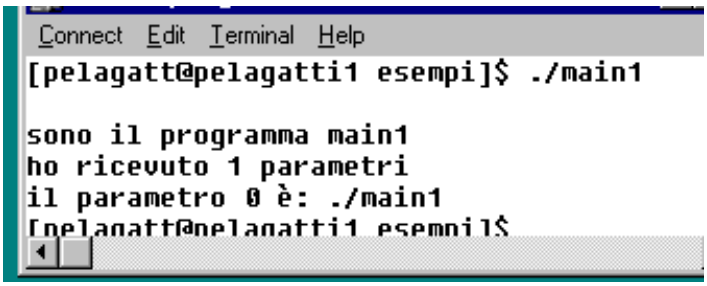
Al momento dell’esecuzione della funzione

execl (char *nome_programma, char *arg0, char *arg1, …);

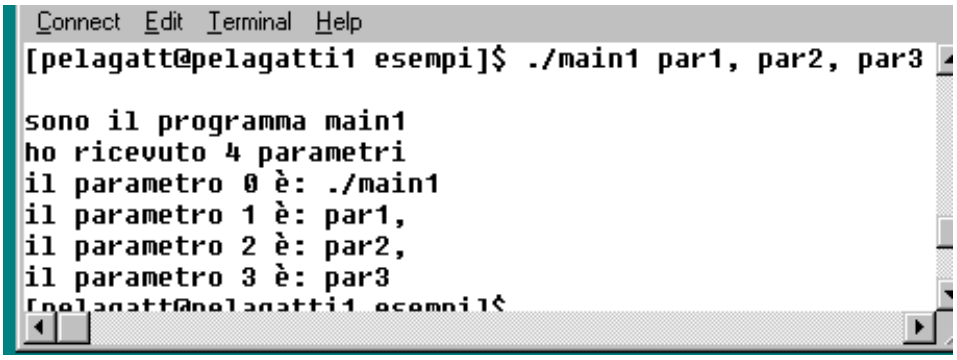
i parametri arg0, arg1, ecc… vengono resi accessibili al nuovo programma tramite il vettore di puntatori argv.

```
#include <stdio.h>
void main (int argc, char *argv[ ] )
{       int i;

        printf("\nsono il programma main1\n");
        printf("ho ricevuto %i parametri\n", argc);

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

}
```

a)il programma main1



b)risultato dell'esecuzione di main1 da riga di comando, senza parametri



c) risultato dell'esecuzione di main1 da riga di comando, con 3 parametri

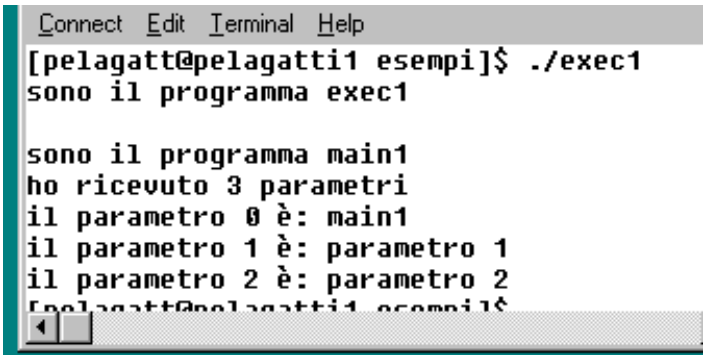
Figura 1.6

Un esempio di uso della funzione execl è mostrato nelle figure 1.6 e 1.7. In figura 1.6 sono riportati il codice e il risultato dell'esecuzione di un programma main1 che stampa il numero di parametri ricevuti (argc) e i parametri stessi. In particolare, in figura 1.6b si mostra l'effetto dell'esecuzione di main1 senza alcun parametro sulla

riga di comando; il risultato mostra che la shell ha passato a main1 un parametro costituito dal nome stesso del programma, in base alle convenzioni già richiamate precedentemente. In figura 1.6c invece sono stati passati 3 parametri sulla riga di comando e il programma li ha stampati.

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

void main( )
{
        char P0[ ]="main1";
        char P1[ ]="parametro 1";
        char P2[ ]="parametro 2";

        printf("sono il programma exec1\n");

        execl("/home/pelagatt/esempi/main1", P0, P1, P2, NULL);

        printf("errore di exec"); /*normalmente non si arriva qui!*/
}
```

a)il programma exec1



b)risultato dell'esecuzione di exec1

figura 1.7

In figura 1.7 sono invece mostrati il codice e il risultato dell'esecuzione del programma exec1, che lancia in esecuzione lo stesso programma main1 passandogli alcuni parametri. Si noti che main1 scrive sullo stesso terminale di exec1, perché lo standard output è rimasto identico, trattandosi dello stesso processo. Si noti che in questo caso è stato il programma exec1 a seguire la convenzione di passare a main1 il nome del programma come primo parametro argv[0]; questo fatto conferma che si

tratta solamente di una convenzione seguita in particolare dagli interpreti di comandi (shell) e non di una funzione svolta direttamente dal servizio di exec.

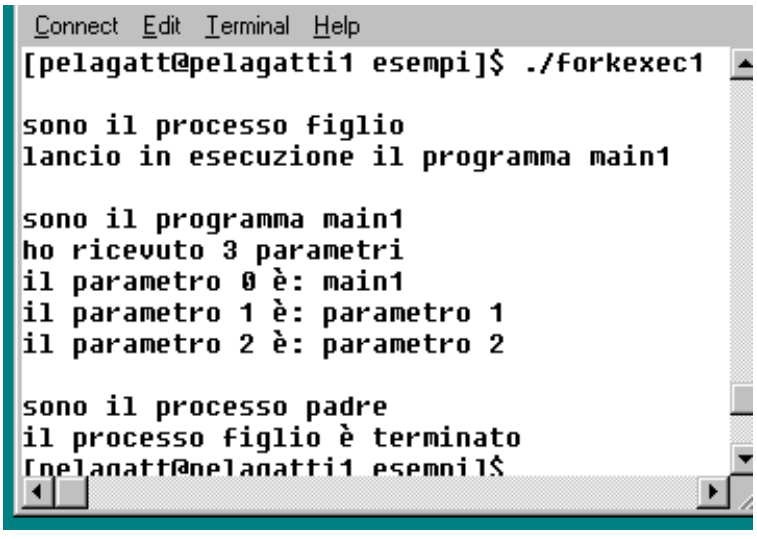
Il servizio exec assume particolare rilevanza in collaborazione con il servizio fork, perchè utilizzando entrambi i servizi è possibile per un processo far eseguire un programma e poi riprendere la propria esecuzione, come fanno ad esempio gli interpreti di comandi. In figura 1.8 sono mostrati il testo e il risultato dell'esecuzione di un programma forkexec1, nel quale viene creato un processo figlio che si comporta come il precedente programma exec1, mentre il processo padre attende la terminazione del figlio per proseguire.

Altre versioni della funzione exec

Esistono altre versioni della funzione exec che differiscono tra loro nel modo in cui vengono passati i parametri al programma lanciato in esecuzione. In particolare la **execv** permette di sostituire la lista di stringhe dalla execl con un puntatore a un vettore di stringhe, in maniera analoga al modo in cui i parametri sono ricevuti nel main; altre 2 versioni (**execlp** e **execvp**) permettono di sostituire il nome completo (pathname) dell'eseguibile con il semplice nome del file e utilizzano il direttorio di default per cercare tale file; infine 2 ulteriori versioni (**execle** e **execve**) hanno un parametro in più che si riferisce all'ambiente di esecuzione (environment) del processo. Per la sintassi di queste versioni si veda il manuale.

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

void main( )
{       pid_t  pid;
        int stato_wait;
        char P0[ ]="main1";
        char P1[ ]="parametro 1";
        char P2[ ]="parametro 2";

        pid=fork( );

        if (pid==0)
        {       printf("\nsono il processo figlio \n");
                printf("lancio in esecuzione il programma main1\n");
                execl("/home/pelagatt/esempi/main1", P0, P1, P2, NULL);
                printf("errore di exec"); /*normalmente non si arriva qui!*/
                exit( );
        }
        else
        {
                wait(&stato_wait );
                printf("\nsono il processo padre\n");
                printf("il processo figlio è terminato\n");
                exit( );
        }
}
```

a)il programma forkexec1



b)risultato dell'esecuzione di forkexec1

Figura 1.8

1.7 Esercizio conclusivo

Con riferimento al programma di figura 1.9 si devono riempire 3 tabelle, una per il processo padre e una per ognuno dei due processi figli, aventi la struttura mostrata in figura 1.10, indicando, negli istanti di tempo specificati, il valore delle variabili i, j, k, pid1 e pid2, e utilizzando le seguenti convenzioni:

- nel caso in cui al momento indicato la variabile non esista (in quanto non esiste il processo) riportare NE;
- se la variabile esiste ma non se ne conosce il valore riportare U;
- si suppone che tutte le istruzioni fork abbiano successo e che il sistema operativo assegni ai processi figli creati valori di pid pari a 500, 501, ecc…;

Attenzione: la frase “dopo l’istruzione x” definisce l’istante di tempo immediatamente successivo all’esecuzione dell’istruzione x da parte di un processo (tale processo è univoco, data la struttura del programma); a causa del parallelismo, può essere impossibile stabilire se gli altri processi hanno eseguito certe istruzioni e quindi se hanno modificato certe variabili – in questi casi è necessario utilizzare la notazione U per queste variabili, perchè il loro valore nell’istante considerato non è determinabile con certezza.

La soluzione è riportata in figura 1.11. Nel processo padre le variabili esistono sempre, il valore di pid1 dopo l’istruzione 6 è assegnato a 500 e non cambia nelle istruzioni successive, il valore di pid2 è sempre indeterminato, le variabili j e k mantengono sempre il loro valore iniziale. Più complessa è la valutazione dello stato della variabile i dopo le istruzioni 9 e 11, perchè queste istruzioni sono eseguite dai processi figli e quindi non possiamo sapere se il processo padre ha già eseguito in questi istanti l’istruzione 18, che modifica i. questo è il motivo per l’indicazione U nelle corrispondenti caselle. Dopo l’istruzione 19 ovviamente i vale 11.

Nel primo processo figlio l’ultima riga vale NE perchè sicuramente in quell’istante tale processo è sicuramente terminato. I valori di pid1, pid2, j e k sono ovvi. Il valore di i è indeterminato sostanzialmente per lo stesso motivo indicato per il processo padre.

Nel secondo processo figlio le motivazioni sono derivabili da quelle fornite per i casi precedenti.

```
01:     main()
02:     {
03:             int i, j, k, stato;
04:             pid_t pid1, pid2;
05:             i=10; j=20; k=30;
06:             pid1 = fork(); /*creazione del primo figlio /
07:             if(pid1 == 0) {
08:                             j=j+1;
09:                             pid2 = fork(); /*creazione del secondo figlio */
10:                             if(pid2 == 0) {
11:                                             k=k+1;
12:                                             exit();}
13:                             else {
14:                                     wait(&stato);
15:                                     exit();   }
16:                             }
17:             else {
18:                     i=i+1;
19:                     wait(&stato);
20:                     exit();   }
21:     }
```

Figura 1.9

Istante	Valore delle variabili				
	pid1	**pid2**	**i**	**j**	**k**
Dopo l'istruzione 6					
dopo l'istruzione 9					
dopo l'istruzione 11					
dopo l'istruzione 19					

figura 1.10 - Struttura delle 3 tabelle da compilare

Istante	Valore delle variabili				
	pid1	pid2	i	j	k
Dopo l'istruzione 6	500	U	10	20	30
dopo l'istruzione 9	500	U	U	20	30
dopo l'istruzione 11	500	U	U	20	30
dopo l'istruzione 19	500	U	11	20	30

1. Valore delle variabili nel processo padre.

Istante	Valore delle variabili				
	pid1	pid2	i	j	k
dopo l'istruzione 6	0	U	10	20	30
dopo l'istruzione 9	0	501	10	21	30
dopo l'istruzione 11	0	501	10	21	30
dopo l'istruzione 19	NE	NE	NE	NE	NE

2. Valore delle variabili nel primo processo figlio.

Istante	Valore delle variabili				
	pid1	pid2	i	j	k
dopo l'istruzione 6	NE	NE	NE	NE	NE
dopo l'istruzione 9	0	0	10	21	30
dopo l'istruzione 11	0	0	10	21	31
dopo l'istruzione 19	NE	NE	NE	NE	NE

3. Valore delle variabili nel secondo processo figlio

Figura1.11 – Soluzione dell'esercizio