Processi

C. Bolchini

Il *processo* è un'istanza del programma in esecuzione ed ha un insieme di informazioni ad esso associate che lo caratterizzano.

La libreria di sottoprogrammi per accedere alle informazioni sui processi e manipolarle è la libreria `unistd.h`.

1 Caratteristiche

All'avvio il sistema operativo crea un primo processo, `init`, da cui vengono creati tutti gli altri processi. Si crea quindi una gerarchia ad albero, in cui ogni processo ha un processo padre e zero o più processi figli (Figura 1).

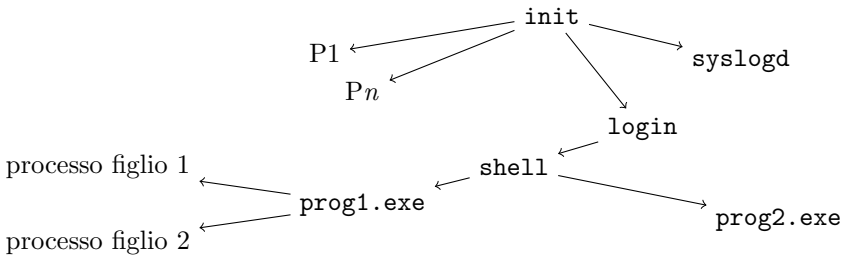

Figura 1: Gerarchia di processi.

come detto, un processo è un programma in esecuzione e il programma che il processo sta eseguendo si chiama l'*immagine del processo*. Ogni processo ha associato alcuni elementi, di seguito elencati (Figura 2):

Blocco di Controllo del Processo : (Process Control Block – PCB) insieme di informazioni utilizzate dal sistema operativo per controllare i processi.

System Stack : Pila del processo, utilizzata per memorizzare parametri e indirizzi delle chiamate di sistema.

Spazio utente (user space) : contiene il codice del programma che il processo esegue, e lo spazio dati (sia statici che dinamici).

Spazio condiviso (shared space) : spazio condiviso tra questo e altri processi.

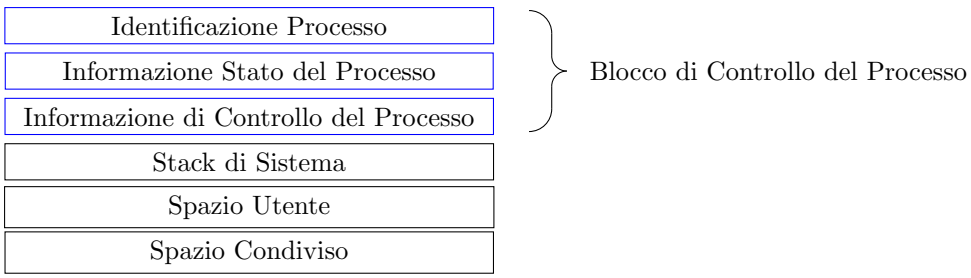

Figura 2: Immagine del processo.

Il PCB contiene a) informazioni per identificare il processo, b) informazioni sullo stato del processo e c) informazioni di controllo del processo.

Le informazioni per identificare il processo sono: a1) l'identificatore univoco del processo, a2) l'identificatore univoco del processo che ha creato questo processo (il processo *padre*), e a3) identificativo dell'utente responsabile del processo. Nell'insieme di informazioni del processo, ci sono: b1) registri visibili all'utente, b2) registri di stato e di controllo (per esempio, il registro Program Status Word - PSW) e b3) gli Stack Pointer del processo. L'ultimo insieme di informazioni include ulteriori dati per la gestione del processo, quali ad esempio lo scheduling usato, la priorità, i privilegi, gli elementi per la gestione della comunicazione tra processi le informazioni per la gestione della memoria e le risorse utilizzate.

Per riassumere, le informazioni più importanti sono:

ID processo		non ereditato dal padre
ID processo padre		non ereditato dal padre
ID utente	proprietario del processo	ereditato dal padre
ID gruppo	gruppo del proprietario del processo	ereditato dal padre
Stato del processo	`RUNNING, STOPPED, ZOMBIE, DEAD, INTERRUPTIBLE, UNINTERRUPTIBLE`	
Livello di priorità		ereditata dal padre
Informazioni scheduling	tempo di esecuzione, attesa accumulata, $\cdots$	ereditate dal padre
Permessi		ereditati dal padre
Descrittori file aperti	elenco dei file aperti	ereditati dal padre
Variabili d'ambiente		ereditate dal padre

Figura 3: Blocco di Controllo del Processo.

Dunque, il processo creato è una copia del progetto padre, ad eccezione del proprio identificatore (e dell'identificatore del padre, visto che il processo padre non può avere se stesso come processo padre).

1.1 ID del processo

Come detto, ogni processo è identificato univocamente tramite un *identificatore di processo* (*process ID* o *pid*), costituito da un numero di 16 bit, assegnato in modo sequenziale dal sistema operativo man mano che si creano nuovi processi.

Ogni processo ha anche un processo padre (a parte il processo speciale `init`, o `launchd` in MacOSX), identificato anch'esso da un *pid*, che prende il nome di *ppid*. In realtà per rappresentare l'identificatore di un processo, è definito nella libreria standard un tipo di dato apposito `pid_t`. Il processo `init` ha *pid* pari a 1.

È possibile accedere all'informazione del *pid* utilizzando il sottoprogramma della libreria di sistema `getpid()`, il cui prototipo è riportato di seguito:

```
pid_t getpid();
```

che restituisce l'identificatore del processo che effettua la chiamata. In modo simile, il sottoprogramma `getppid()` restituisce l'identificatore del processo padre. Il programma qua di seguito riportato (Listato 1) utilizza i due sottoprogrammi citati, mentre in Figura 4 è riportato ciò che viene visualizzato.

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

int main(int argc, char * argv[])
{
   printf("Il processo ha ID: %d\n", (int) getpid());
   printf("Il processo padre ha ID: %d\n", (int) getppid());

   return 0;
}
```

Listing 1: Identificatori di processo

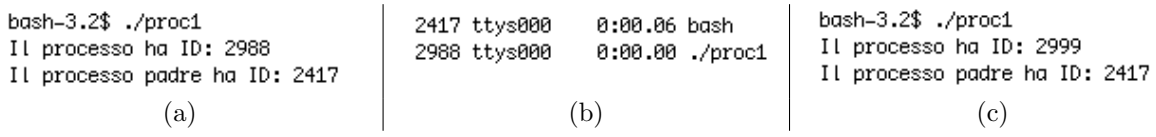

Figura 4: Identificatori di processo

Il programma si chiama `proc1`. Alla prima esecuzione il processo ha *pid* `2988` (Figura 1a); in Figura 1 l'elenco dei processi mostra il processo shell (`bash` da cui è stato invocato il programma in esecuzione (`proc1`)). Ogni volta che il programma viene eseguito, viene creato un nuovo processo, mentre il processo padre è sempre lo stesso (Figura 4c) se si usa sempre la stessa shell.

Per vedere quali sono i processi attivi, dalla shell è necessario eseguire il comando `ps`, che produce la lista dei processi controllati dal terminale della shell in cui si è invocato il comando. Utilizzando le opzioni offerte è possibile farsi elencare anche l'identificatore del processo padre (ad esempio con `ps -e -o pid,ppid,command`).

1.2 Esecuzione di un programma in *foreground* o *background*

Quando dall'interprete comandi (o *shell*) viene eseguito un comando, la shell crea un nuovo processo per l'esecuzione del programma e si mette in attesa della sua terminazione. È anche possibile fare in modo che la shell continui la propria esecuzione (restituendo il controllo all'utente che può quindi eseguire altri programmi) senza attendere la terminazione, bensì mandando in *background*, il programma eseguito, aggiungendo al termine del comando, il carattere `&`. In Figura 5 sono mostrate rispettivamente a) l'esecuzione normale (in *foreground*) del programma `proc_from_shell`, b) l'esecuzione in *background* in cui viene indicato l'identificativo del processo che è stato creato (`5268` nell'esempio) e viene restituito il prompt all'utente che può così eseguire altri comandi, e c) il messaggio che indica che il processo è terminato. È anche possibile interrompere l'esecuzione di un programma eseguito in *foreground*, per poi farlo riprendere in *background*, utilizzando rispettivamente il comando `Ctrl+z` (visualizzato come `^Z` in Figura 5d) e poi `bg` per far riprendere l'esecuzione in *background* (come mostrato in Figura 5e). È anche possibile far riprendere l'esecuzione in *foreground* con il comando `fg` (Figura 5f).

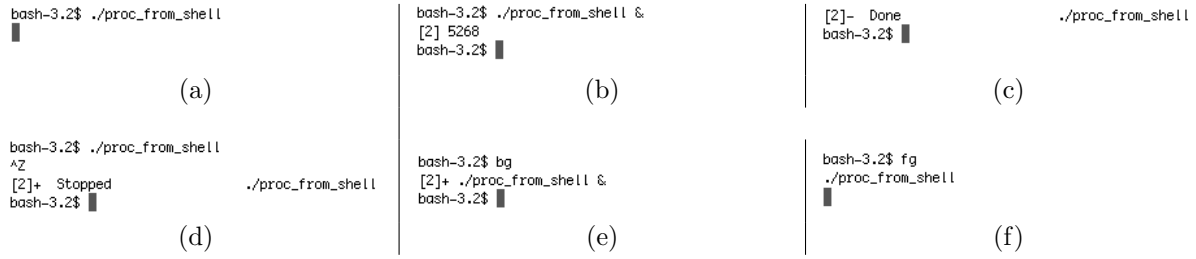

Figura 5: Esecuzione di un programma da interprete comandi

2 Creare processi

In questa sezione verranno mostrati gli aspetti di creazione e impiego dei processi, da programma.

2.1 Operazioni per la creazione di un processo

I passi che il kernel svolge per la creazione di un processo sono i seguenti:

- Assegnazione di un identificatore univoco e aggiunta di una nuova voce nella tabella dei processi;

- Allocazione dello spazio per il processo e la sua immagine;
- Inizializzazione del Blocco di Controllo del Processo e degli Stack Pointer;
- Inizializzazione di ulteriori riferimenti e altre azioni.

La "vita" di un processo si conclude nel momento in cui il suo termine viene comunicato al processo padre; a quel punto, tutte le risorse del processo, incluso il suo identificatore, vengono liberate.

2.2 `fork()`

Linux offre una funzione di libreria che consente ad un processo di creare un processo figlio che è una copia esatta del processo padre: `fork()`. Il processo padre continua ad esistere e procede nell'esecuzione normalmente, con le istruzioni successive alla chiamata della `fork()`. Il processo figlio, è un nuovo processo, con un nuovo identificatore, che eseguirà la parte di codice di sua competenza. Di fatto, esiste un unico programma che ha al suo interno, sia il codice che deve essere eseguito dal padre, sia il codice che deve essere eseguito dal figlio.

È possibile distinguere il codice padre da quello del figlio, perchè il sottoprogramma `fork()` restituisce al chiamante l'identificatore del processo che è stato creato (il padre conosce così il *pid* del processo figlio, altrimenti ignoto), mentre tale valore è 0 per il processo figlio. Il programma riportato nel Listato 2 mostra come viene gestita la creazione di un processo figlio, e la distinzione dei due processi, in Figura 6 è riportata l'esecuzione del programma che dà luogo ai due processi.

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

#define N 50

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

   printf("processo originale - ID: %d\n", (int) getpid());
   /* chiamata per la creazione di un processo figlio */
   childpid = fork();
   if (childpid != 0){
      /* questo e' il processo padre, che ha il pid del figlio */
      printf("\n----------------------------------------");
      printf("\nIo sono il processo padre con ID: %d", (int) getpid());
      printf("\nIl processo creato ha ID: %d", (int) childpid);
      printf("\nio stampo *\n");
      fflush(stdout);
      for(i = 0; i < N; i++){
         printf("*");
         fflush(stdout);
      }
      printf("\nfine processo padre\n");
      fflush(stdout);
   } else { /* questo e' il processo figlio */
       printf("\n----------------------------------------");
      printf("\nIo sono il processo figlio con ID: %d", (int) getpid());
      printf("\nIl processo padre ha ID: %d", (int) getppid());
      printf("\nio stampo +\n");
      fflush(stdout);
      for(i = 0; i < N; i++){
         printf("+");
         fflush(stdout);
      }
      printf("\nfine processo figlio\n");
      fflush(stdout);
      exit(0);
   }
   return 0;
}
```

Listing 2: Creazione di processi

L'esecuzione del programma (chiamato `proc2`) dà luogo alla traccia di Figura 6. Si ricordi che, fintanto i due processi sono entrambi attivi, è la politica di scheduling a stabilire quale dei due vada in esecuzione, quindi, in generale, una volta effettuata la chiamata al sottoprogramma `fork()` potrebbe venire sospeso il processo padre ed eseguito il processo figlio. In generale, è sbagliato fare ipotesi sull'ordine di esecuzione, e far dipendere la correttezza del proprio algoritmo da tali considerazioni.

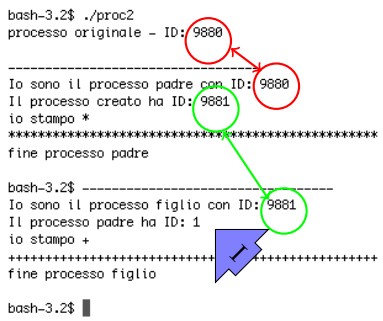

Figura 6: Creazione di processo: esecuzione. I) Il processo padre è terminato.

La freccia di Figura 6 mette in evidenza che il processo figlio, identificato dal *pid* `3934`, ha come processo padre il processo identificato dal *pid* `1` invece che da `3933`, come ci si sarebbe aspettati. Questo è legato al fatto che il processo padre in realtà è già terminato, quindi quando viene mandato in esecuzione il processo figlio, esso ha come processo padre, il processo padre di tutti, quello con *pid* 1 (il processo `init`). Ulteriori considerazioni verranno fatte parlando di processi *zombie* (Paragrafo 2.4).

Per vedere gli effetti della politica di scheduling di *time sharing*, che utilizza il meccanismo di *round robin*, e che sospende periodicamente il processo in esecuzione a favore di uno di quelli in attesa, è necessario incrementare il numero di `+` e `-`, ad esempio cambiando

```
#define N 5000
```

In tal caso si ottiene la traccia dell'esecuzione riportata in Figura 7 (sono state omesse alcune parti ...); in cui si nota che – casualmente – termina prima il processo figlio.

È possibile creare un numero qualsiasi di processi, utilizzando più volte il sottoprogramma `fork()`; il programma riportato nel Listato 3 effettua la creazione di due processi figli, utilizzando lo stesso meccanismo.

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

int main(int argc, char * argv[])
{
   int  cpid, cpid2;
   cpid = fork();
   if (cpid != 0) {
      fprintf(stdout, "father here!\n");
      cpid2 = fork();
      if (cpid2 == 0)
         fprintf(stdout, "second child here\n");
   } else
      fprintf(stdout, "first child here!\n");

   return 0;

}
```

Listing 3: Creazione di due processi

Un processo appena creato mediante il sottoprogramma `fork()` esegue lo stesso codice del processo padre, a partire dall'istruzione `fork()` in poi. Si usa quindi il valore restituito per distinguere ciò che fa un processo (il processo padre) da ciò che fa l'altro (il processo figlio). In realtà, avere tanti processi che eseguono lo stesso programma non è una cosa particolarmente utile; spesso si usa per consentire al processo figlio di eseguire un programma diverso, tramite i sottoprogrammi `exec` (presentati nel Paragrafo 4).

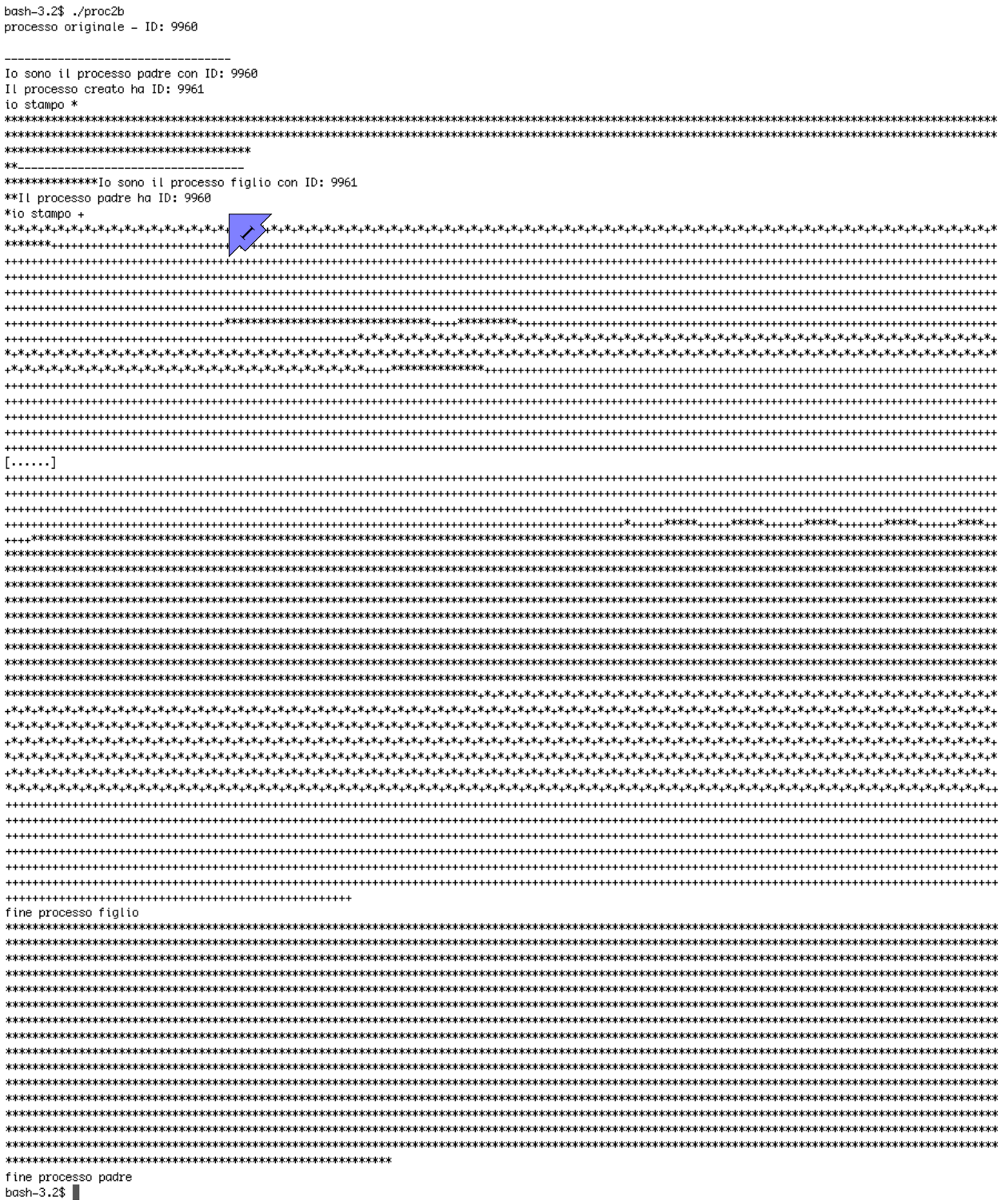

Figura 7: Creazione di processo: esecuzione. I) Identificatore del processo padre, ancora attivo.

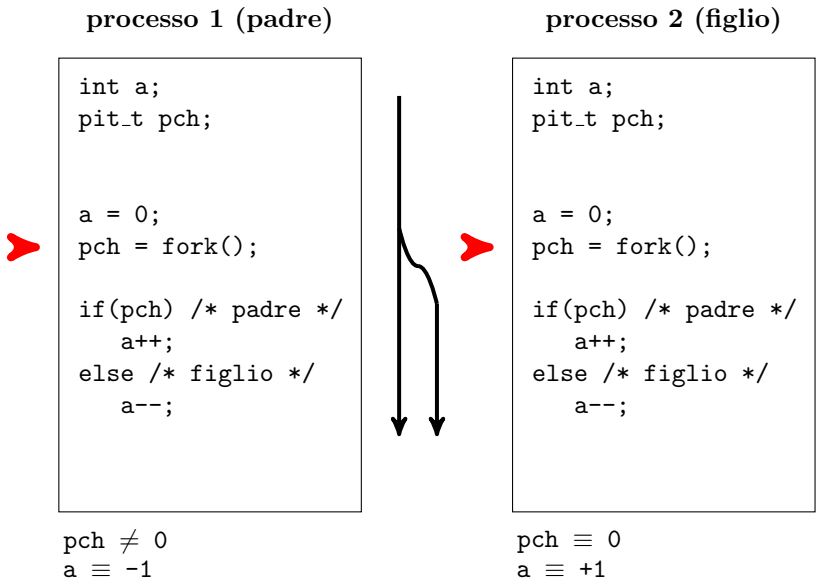

Figura 8: Creazione di un processo figlio da programma.

Poichè la creazione di un processo figlio genera una copia dell'immagine del processo padre, entrambi i processi hanno lo stesso codice, quindi le stesse variabili e lo stesso valore nelle variabili, all'atto della chiamata del sottoprogramma `fork()`. Dopo la chiamata, i due programmi procedono in modo indipendente, ognuno aggiornando i valori delle proprie variabili. Quindi, se anche "sembra" si stia eseguendo lo stesso codice, ognuno dei due processi esegue un "proprio" codice, in cui tutte le variabili hanno un valore uguale, ad eccezione della variabili che memorizza il valore (*pid*) restituito da `fork()`, che è 0 nel processo figlio creato, e diverso da zero nel processo padre. Dalla `fork()` in poi le strade si separano, e ogni processo procede in modo autonomo, per ciò che concerne il valore delle variabili (Figura 8 e Figura 9).

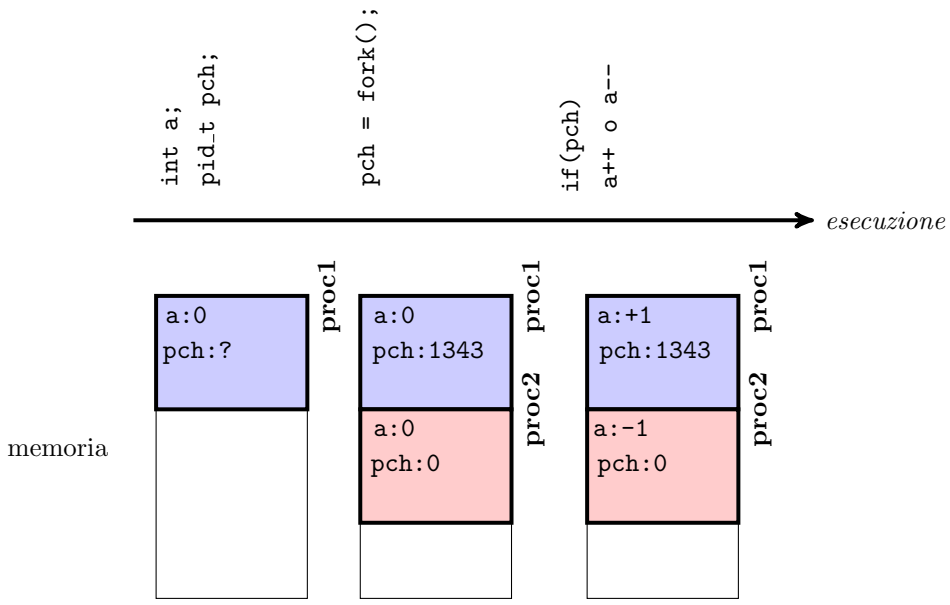

Figura 9: Creazione di un processo figlio da programma: immagini in memoria.

I processi hanno una propria copia delle variabili, ciascuna contenuta nel proprio spazio utente, con il proprio valore. Questo vale per tutti i tipi di variabili: globali, locali statiche e dinamiche; sono variabili allocate nell'area di memoria del processo (gestita dal *gestore della memoria*). Quindi, anche se i riferimenti, gli

indirizzi, della memoria *sembrano* uguali, essi di fatto sono diversi, in quanto sono l'indirizzo *logico* `XXXXX` del processo 1, che è diverso fisicamente dallo stesso indirizzo *logico* `XXXXX` del processo 2.

Per fare qualche esperimento è possibile utilizzare il programma riportato nel Listato 4 per visualizzate i valori degli indirizzi delle variabili e i loro valori e la cui traccia di esecuzione è mostrata in Figura 10.

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


int main(int argc, char * argv[])
{
   pid_t childpid;
   int * prima, * dopo;
   int a;
   int status;

   printf("Indirizzi variabili\n");
   printf("&prima:\t%p\n&prima:\t%p\n&a:\t%p\n", &prima, &dopo, &a);

   prima = (int *)malloc(sizeof(int));
   printf("prima:\t%p\n", prima);

   /* chiamata per la creazione di un processo figlio */
   childpid = fork();
   if(childpid == 0){
      /* figlio */
      a--;
      *prima = 10;
      dopo = (int *)malloc(sizeof(int));
      *dopo = 100;
      printf("*****figlio\n&prima:\t%p\nprima:\t%p\n*prima:\t%d\n",&prima, prima, *prima);
      printf("&dopo:\t%p\ndopo:\t%p\n*dopo:\t%d\n&a:\t%p\n", &dopo, dopo, *dopo, &a);
      _exit(0);
   } else {
      /* padre */
      a++;
      *prima = -10;
      dopo = (int *)malloc(sizeof(int));
      *dopo = -100;
      printf("******padre\n&prima:\t%p\nprima:\t%p\n*prima:\t%d\n", &prima, prima, *prima);
      printf("&dopo:\t%p\ndopo:\t%p\n*dopo:\t%d\n&a:\t%p\n", &dopo, dopo, *dopo, &a);
      wait(&status);
   }
   return 0;
}
```

Listing 4: La memoria del processo: indirizzi e valore delle variabili

Il discorso è invece diverso per quanto riguarda i file. In questo caso, se due processi aprono il file `esempio.txt`, essi accedono entrambi allo stesso elemento logico e fisico. Il programma mostrato nel Listato 5 dà luogo a due processi che accedono entrambi al file `leggimi.txt` tramite la variabile `fp`. Ogni qualvolta si effettua un'operazione di lettura (ma ciò è vero anche in scrittura) la variabile `fp` viene "spostata" e dunque ciò che legge un processo non viene letto dall'altro processo. Nel caso di un accesso in scrittura troveremmo nel file creato ciò che scrive sia il processo padre sia il processo figlio. In Figura 11 è riportato una traccia di esecuzione del programma, mentre il contenuto del file `leggimi.txt` è riportato in Figura 12.

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

#define MAXSTR 100

int main(int argc, char * argv[])
{
   FILE *fileptr1;
   char temp1[MAXSTR+1];
   int status;
   int pid2;

   fileptr1 = fopen("./leggimi.txt","r");
```

```
bash-3.2$ ./var
Indirizzi variabili
&prima: 0xbffff968
&prima: 0xbffff964
&a:     0xbffff960
prima:  0x100160
******padre
&prima: 0xbffff968
prima:  0x100160
*prima: -10
&dopo:  0xbffff964
dopo:   0x100170
*dopo:  -100
&a:     0xbffff960
*****figlio
&prima: 0xbffff968
prima:  0x100160
*prima: 10
&dopo:  0xbffff964
dopo:   0x100170
*dopo:  100
&a:     0xbffff960
bash-3.2$
```

Figura 10: La memoria del processo: indirizzi e valore delle variabili.

```
    setvbuf(fileptr1, (char *)NULL, _IONBF, 0); /* niente buffer */
    pid2=fork();
    printf("%i in esecuzione\n",getpid());  /* parent and child */
    fgets(temp1,MAXSTR,fileptr1);
    while(!feof(fileptr1))
    {
        if (pid2){
            printf("padre -- legge: %s", temp1);
            sleep(2);
        } else {
            printf("figlio -- legge: %s", temp1);
            sleep(1);
        }
        fgets(temp1,MAXSTR,fileptr1);
    }
    if(!pid2)
        _exit(0);
    else
        wait(&status);
    fclose(fileptr1);
    return 0;
}
```

Listing 5: Accesso a file da parte di più processi

Poichè quindi i file sono condivisi tra processi è necessario porre particolare attenzione nell'accesso, ricordando che non è corretto fare poi ipotesi sull'ordine con cui verranno eseguiti i processi.

2.3 Sincronizzare tra processi parenti

Il sistema operativo mette a disposizione anche i meccanismi per consentire una sincronizzazione tra processi: in generale, nel caso di un sistema multiprogrammato (multi-tasking) non è possibile stabilire da parte dell'utente quale processo venga eseguito prima e quale dopo, nel momento in cui entrambi sono attivi. In particolare, se un processo crea un processo figlio, il programmatore non è in grado di determinare l'ordine e l'eventuale alternanza con cui vengono eseguiti i processi. Per questo motivo, si dispone di sottoprogrammi che consentano una qualche forma limitata di sincronizzazione. Il sottoprogramma `wait` sospende l'esecuzione del processo che effettua la chiamata, fino a quando un processo figlio termina o fino a quando esso non venga

```
bash-3.2$ ./proc_fileread
3333 in esecuzione
padre - legge: 1: riga uno
3334 in esecuzione
figlio - legge: 2: riga due
figlio - legge: 3: riga tre
padre - legge: 4: riga quattro
figlio - legge: 5: riga cinque
figlio - legge: 6: riga sei
padre - legge: 7: riga sette
figlio - legge: 8: riga otto
figlio - legge: 9: riga nove
padre - legge: 10: riga dieci
figlio - legge:
bash-3.2$
```

Figura 11: Accesso a file da parte di più processi.

```
1: riga uno
2: riga due
3: riga tre
4: riga quattro
5: riga cinque
6: riga sei
7: riga sette
8: riga otto
9: riga nove
10: riga dieci
```

Figura 12: Contenuto del file `leggimi.txt`.

terminato da qualche altro processo. Se un processo figlio è già terminato (un cosiddetto processo "zombie"), il sottoprogramma ritorna immediatamente e le risorse del processo figlio vengono liberate. Il prototipo è il seguente:

```
pid_t wait(int *status);
```

Come si vede dal prototipo, il sottoprogramma `wait` non riceve in ingresso l'identificatore del processo di cui rimanere in attesa, bensì rimane in attesa che uno qualsiasi dei processi figli termini. È il sottoprogramma a restituire il *pid* del processo figlio che è terminato, trasmettendo anche nel parametro passato per indirizzo *status*, lo stato (il codice) con cui tale processo è terminato, indicato dall'esecuzione dell'istruzione `exit` del processo figlio. Lo stato restituito può poi essere valutato tramite alcune macro, la cui spiegazione è reperibile tramite `man waitpid`.

C'è anche un altro modo di gestire l'attesa o di accedere allo stato di un processo figlio, ed è tramite il sottoprogramma `waitpid`, che consente di richiedere lo stato e di rimanere in attesa di un processo specifico, identificato tramite *pid*. Il prototipo è il seguente:

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

In questo caso, `waitpid` viene utilizzata per conoscere lo stato (trasmesso tramite il parametro *status*) di un processo figlio tramite il suo identificatore *pid*. Di solito il processo che effettua la chiamata rimane così sospeso fino a quando il processo *pid* restituisce lo stato terminando (tramite la chiamata della `exit`). Il primo parametro in realtà può assumere i seguenti valori:

- **>0**: significa che il chiamante ha richiesto lo stato del processo figlio identificato dal *pid*;
- **0**: significa che il chiamante ha richiesto lo stato di un qualsiasi processo cha abbia stesso identificatore di gruppo;
- **-1** o **WAIT_ANY**: significa che il chiamante ha richiesto lo stato di un qualsiasi processo figlio;
- **<-1**: significa che il chiamante ha richiesto lo stato di un qualsiasi processo cha abbia identificatore di gruppo uguale al *pid* indicato.

Il parametro `options` è una maschera di bit in OR per indicare quale comportamento deve avere il chiamante in relazione alla richiesta dello stato del processo. In particolare, se si utilizza la costante simbolica `WNOHANG`, il chiamante riceverà l'informazione sullo stato del processo, ma poi proseguirà senza rimanere in attesa della sua terminazione e in tal caso il sottoprogramma restituisce 0. Utilizzando la costante simbolica `WUNTRACED` il chiamante non viene bloccato ma si ritorna subito dalla chiamata se il processo è sospeso. La costante simbolica `WCONTINUED` permette al chiamante di proseguire se il processo figlio è stato fatto proseguire dopo essere stato sospeso.

Visti i valori possibili per i parametri del sottoprogramma `waitpid()`, le due chiamate seguenti hanno lo stesso significato e comportamento.

```
wait(&statusproc);
```

e

```
waitpid(WAIT_ANY, &statusproc, 0);
```

2.4 Zombie

Un processo *zombie* è un processo che ha terminato la propria esecuzione, ma che non è stato ancora "ripulito" (è ancora in memoria). È compito dei processi padri, eliminare tutte le informazioni dei processi figli terminati. In particolare, un processo figlio che è terminato ma non è stato atteso (tramite una `wait()` o una `waitpid()`), diventa un processo "zombie". Ciò significa che il kernel mantiene un insieme minimo di informazioni per il processo (`PID`, stato di terminazione, informazioni sull'utilizzo delle risorse) per consentire poi al processo padre di accedere a tali informazioni. Sostanzialmente ci si aspetta che il processo padre non termini senza aver controllato la terminazione dei processi figli. Finché i processi *zombie* non vengono rimossi dal sistema mediante una `wait()`, essi consumano uno slot nella tabella dei processi del kernel; nel caso in cui tale tabella si riempia, non consente la creazione di ulteriori processi.

Nel caso in cui un processo padre termini senza attendere i processi figli, il sistema operativo assegna al processo `init` i processi *zombie* che sono stati così creati; periodicamente il processo `init` esegue una `wait()` per ripulire così il sistema.

Il Listato 6 mostra un programma che consente la creazione di un processo *zombie*, contando sul fatto che la sospensione del padre per un minuto consentirà al processo figlio di essere eseguito e terminato prima del processo padre.

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

int main ()
{
  pid_t child_pid;

  /* Crea processo figlio.  */
  child_pid = fork ();
  if (child_pid > 0) {
    /* padre: Sleep per un minuto.  */
    sleep (60);
  }
  else {
    /* Figlio: termina subito.  */
    exit (0);
  }
  return 0;
}
```

Listing 6: Processo *zombie*

2.5 Esempio

Il seguente programma crea due processi figli e ciascuno dei processi modifica il valore delle proprie variabili. È possibile provare a determinare, prima di eseguire il programma, cosa venga visualizzato (a parte l'esatto valore dei *pid*).

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

int main(int argc, char * argv[])
{
   int a, b, c;
   pid_t c1, c2;
   int status;

   a = 10;
   b = 20;
   c = 30;
   c1 = fork();
   if(c1 == 0){
      b++;
      c2 = fork();
      if(c2 == 0){
         c++;
         printf("procX: c1: %d, c2: %d, a: %d, b: %d, c: %d\n", c1, c2, a, b, c);
      } else {
         wait(&status);
         printf("procY: c1: %d, c2: %d, a: %d, b: %d, c: %d\n", c1, c2, a, b, c);
      }
      exit(0);
   }
   a++;
   wait(&status);
   printf("procZ: c1: %d, c2: %d, a: %d, b: %d, c: %d\n", c1, c2, a, b, c);
   return 0;
}
```

Listing 7: Creazione di processi

3 Le variabili d'ambiente

ll sistema operativo offre ad ogni programma in esecuzione un *ambiente*, ossia una collezione di coppie variabile-valore, costituite da stringhe. Convenzionalmente, le variabili d'ambiente hanno nomi tutti maiuscoli. Tra le variabili più note ci sono:

- `USER` il nome dell'utente;
- `HOME` il percorso alla directory home dell'utente;
- `PATH` il percorso in cui vengono cercati i programmi quando invocati.

La stessa shell ha un ambiente e offre comandi per visualizzare e aggiungere e modificare il valore delle variabili d'ambiente (il nome dei comandi può cambiare in base al tipo di shell). Nella shell di tipo `bash`, i comandi sono i seguenti:

- `echo $NOMEVARIABILE` visualizza il valore della variabile d'ambiente `NOMEVARIABILE`;
- `export NOMEVARIABILE=valore` assegna alla variabile d'ambiente `NOMEVARIABILE` il valore `valore`; se la variabile esisteva già il valore viene aggiornato, altrimenti la variabile viene creata. Nel caso in cui la variabile consista di una lista e si voglia aggiungere una o più voci, si procede così: `export PATH=$PATH:_nuovopercorso_`.

La Figura 13 riporta una schermata con esempi d'uso dei comandi citati.

```
bash-3.2$ echo $HOME
/Users/bolk
bash-3.2$ echo $USER
bolk
bash-3.2$ env
MANPATH=/usr/share/man:/usr/local/share/man:/usr/local/man:/Library/TeX/Distributions/.DefaultTeX/Contents/Man:/usr/X11/man
TERM_PROGRAM=Apple_Terminal
SHELL=/bin/bash
TERM=xterm-color
TMPDIR=/var/folders/zr/zrNNPbnp2RmUdU+F75VobU+++TM/-Tmp-/
Apple_PubSub_Socket_Render=/tmp/launch-m9B4lc/Render
TERM_PROGRAM_VERSION=240
USER=bolk
COMMAND_MODE=unix2003
SSH_AUTH_SOCK=/tmp/launch-Rsx2ET/Listeners
__CF_USER_TEXT_ENCODING=0x1F6:0:0
PATH=/usr/local/bin:/opt/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/texbin:/usr/X11/bin
PWD=/Users/bolk
EDITOR=vim
LANG=en_US.UTF-8
HOME=/Users/bolk
SHLVL=2
LOGNAME=bolk
DISPLAY=/tmp/launch-ub7ZFo/:0
SECURITYSESSIONID=b9f2a0
_=/usr/bin/env
bash-3.2$ export ARCH=i86
bash-3.2$
```

Figura 13: Valore di alcune variabili d'ambiente (visualizzate mediante il comando `echo`), e lista di tutte le variabili d'ambiente (visualizzata mediante il comando `env`).

3.1 Accesso da programma

È possibile accedere alle variabili d'ambiente e al loro valore tramite il sottoprogramma `getenv()` presente nella libreria `stdlib.h`, il cui prototipo è il seguente:

```
char * getenv(char * var);
```

Il sottoprogramma riceve in ingresso il nome della variabile *var* di cui si desidera conoscere il valore, che viene restituito come stringa; nel caso in cui la variabile non esista, il sottoprogramma restituisce `NULL`. Per impostare il valore di variabili o eliminarle sono disponibili rispettivamente i sottoprogrammi `setenv()` e `unsetenv()`.

```
int setenv(char * var, char * val, int modify);
int unsetenv(char * var);
```

Il sottoprogramma `setenv()` riceve in ingresso, nell'ordine, il nome della variabile *var*, il valore della variabile *val* e un intero *modify* che indica se sovrascrivere il valore della variabile nel caso esista già (parametro messo a 1) oppure no (parametro messo a 0). Per quanto riguarda invece l'eliminazione, il sottoprogramma riceve in ingresso il nome della variabile da eliminare. Entrambi restituiscono un codice che indica il buon esito dell'operazione (0) oppure no (valore diverso da 0).

Il Listato 8 riporta un programma per la visualizzazione del valore di una variabile d'ambiente, specificata dall'utente (in Figura 14 è riportata la traccia di esecuzione del programma stesso).

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

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

   char * nomevar; /* nome della variabile di cui si vuole visualizzare il valore */
   char * valore;

   /* il nome della variabile e' dato sulla linea di comando */
   if (argc != 2){
      printf("Sintassi del comando: %s NOMEVARIABILE\n", argv[0]);
      exit (0);
   }
```

```
    /* il primo parametro e' il nome del programma */
    /* il secondo parametro e' il nome della variabile */
    nomevar = argv[1];
    valore = getenv(nomevar);

    if(valore)  /* se la variabile d'ambiente esiste */
       printf("Il valore della variabile d'ambiente %s e': %s\n", nomevar, valore);
    else
       printf("Variabile %s inesistente\n", nomevar);

    return 0;
}
```

Listing 8: Accesso alle variabili d'ambiente

```
mac-bolchini-2:processi bolk$ ./mostravariabile USER
Il valore della variabile d'ambiente USER e': bolk
mac-bolchini-2:processi bolk$ ./mostravariabile HOME
Il valore della variabile d'ambiente HOME e': /Users/bolk
mac-bolchini-2:processi bolk$ ./mostravariabile PROVA
Variabile PROVA inesistente
```

Figura 14: Accesso alle variabili d'ambiente.

4 Esecuzione di un programma su file

Sono disponibili due approcci per l'esecuzione da codice di programmi residenti nel file system. Il primo di questi è piuttosto semplice ma offre pochi controlli e può essere rischioso, il secondo metodo invece consente maggior flessibilità, velocità e sicurezza.

4.1 Utilizzo di `system()`

Il sottoprogramma `system()` presente nella libreria standard C offre un modo immediato per eseguire un comando che corrisponde ad un programma, dall'interno di un altro programma, come se il nome del comando fosse stato scritto direttamente nella shell, l'interprete comandi. Infatti, il sottoprogramma `system()` crea un processo che esegue la shell (`/bin/sh`) e lancia poi il comando in tale shell. Il sottoprogramma `system()` restituisce il codice d'uscita del comando della shell. Il Listato 9 invoca il comando `env` visto prima per visualizzare tutte le variabili d'ambiente.

```
#include <stdio.h>

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

    retval = system("env");
    printf("\nreturn: %d\n", retval);
    /* il programma restituisce cio' che ha restituito */
    /* la chiamata di system */
    return retval;

}
```

Listing 9: Esecuzione di programmi da codice: esempio

Ciò che viene visualizzato dall'esecuzione del programma è riportato in Figura 15.
Una variante dello stesso programma, che fa uso di quanto visto in precedenza è riportato nel Listato 10 e l'esecuzione in Figura 16.

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

```
mac-bolchini-2:processi bolk$ ./systemcall
MANPATH=/usr/share/man:/usr/local/share/man:/usr/local/man:/Library/TeX/Distributions/.DefaultTeX/Contents/Man:/usr/X11/man
TERM_PROGRAM=Apple_Terminal
SHELL=/bin/bash
TERM=xterm-color
TMPDIR=/var/folders/zr/zrNNPbnp2RmUdU+F75VobU+++TM/-Tmp-/
Apple_PubSub_Socket_Render=/tmp/launch-m9B4lc/Render
TERM_PROGRAM_VERSION=240
USER=bolk
COMMAND_MODE=unix2003
SSH_AUTH_SOCK=/tmp/launch-Rsx2ET/Listeners
__CF_USER_TEXT_ENCODING=0x1F6:0:0
PATH=/usr/local/bin:/opt/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/texbin:/usr/X11/bin
_=/usr/bin/env
PWD=/Users/bolk/Documents/Corsi/Politecnico/2008-2009/fi1/processi
LANG=en_US.UTF-8
HOME=/Users/bolk
SHLVL=2
LOGNAME=bolk
DISPLAY=/tmp/launch-ub7ZFo/:0
SECURITYSESSIONID=b9f2a0

return: 0
```

Figura 15: Esecuzione di programmi da codice: esempio.

```
int main(int argc, char *argv[])
{
   int retval, status;
   pid_t pchild;

   pchild = fork();
   if(pchild == 0){
      printf("***************\nchild: %d\n", (int) getpid());
      retval = system("env");
      /* termina restituendo l'esito avuto da system */
      exit(retval);
   }
   /* padre */
   printf("***************\nfather: %d and child: %d\n", (int) getpid(), pchild);
   wait(&status);
   if (WIFEXITED(status))
      printf("***************\nreturn: %d\n", WEXITSTATUS(status));
   else
      printf("terminato in modo anomalo\n");

   return 0;
}
```

Listing 10: Esecuzione di programmi da codice: esempio con uso di `fork()`

Il limite di questo approccio consiste nel fatto che l'esecuzione del programma da shell va soggetto a tutti i problemi delle shell.

4.2 Utilizzo di `exec`

Quando si parla di sottoprogrammi `exec` si fa riferimento ad una famiglia di sottoprogrammi della libreria di sistema il cui nome inizia per `exec` seguito da una o più lettere, che svolgono la stessa funzionalità sostanzialmente, ma differiscono – di poco – in base ai parametri che ricevono in ingresso ed al comportamento (tra cui, ad esempio, `execv()` o `execl()`) I sottoprogrammi `exec` **sostituiscono** il programma che sta eseguendo prima della chiamata con un altro programma, quello indicato nel parametro passato ad `exec`. Quando si esegue una `fork()`, il processo continua ad eseguire il codice del programma che ha effettuato la chiamata, che viene replicato nei due processi, padre e figlio. Quando invece si chiama una `exec`, il processo immediatamente non esegue più il codice del programma che ha effettuato la chiamata, quanto inizia ad eseguire dall'inizio il codice del programma chiamato (a meno che la `exec` non fallisca). Inoltre, poiché il programma in esecuzione viene sostituito con quello da eseguirsi, di fatto non viene fatta alcuna `return`, a meno che qualcosa non funzioni male.

```
bash-3.2$ ./systemcall2
****************
father: 6778 and child: 6779
****************
child: 6779
MANPATH=/usr/share/man:/usr/local/share/man:/usr/local/man:/Library/TeX/Distributions/.DefaultTeX/Contents/Man:/usr/X11/man
TERM_PROGRAM=Apple_Terminal
TERM=xterm-color
SHELL=/bin/bash
TMPDIR=/var/folders/zr/zrNNPbnp2RmUdU+F75VobU+++TM/-Tmp-/
Apple_PubSub_Socket_Render=/tmp/launch-m9B4lc/Render
TERM_PROGRAM_VERSION=240
USER=bolk
COMMAND_MODE=unix2003
SSH_AUTH_SOCK=/tmp/launch-Rsx2ET/Listeners
__CF_USER_TEXT_ENCODING=0x1F6:0:0
PATH=/usr/local/bin:/opt/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/texbin:/usr/X11/bin
_=/usr/bin/env
PWD=/Users/bolk/Documents/Corsi/Politecnico/2008-2009/fi1/processi
EDITOR=vim
LANG=en_US.UTF-8
SHLVL=3
HOME=/Users/bolk
LOGNAME=bolk
ARCH=i86
DISPLAY=/tmp/launch-ub7ZFo/:0
SECURITYSESSIONID=b9f2a0
****************
return: 0 _
```

Figura 16: Esecuzione di programmi da codice: secondo esempio con uso di `fork()`.

Di solito, si utilizza una `exec` dopo aver effettuato un `fork()`, avendo così creato un nuovo processo per l'esecuzione del programma da riga di comando, un po' come fatto nel Listato 10.

La famiglia è costituita da sottoprogrammi il cui nome inizia con *exec* ed è seguito da una o più lettere, ognuna delle quali indica un'ulteriore specifica funzionalità della `exec`. In particolare:

- i sottoprogrammi che contengono la lettera *p* accettano come parametro il nome del programma da eseguire e cercano tale nome nel percorso indicato dalla variabile `$PATH`. Se la lettera *p* non è contenuta nel nome, è necessario fornire il nome del programma da eseguirsi **completo** di percorso.
- i sottoprogrammi che contengono la lettera *v* accettano come parametro la lista dei parametri da dare in ingresso al programma da eseguirsi, intesa come un array di puntatori a stringhe, di cui l'ultimo punta a `NULL`. Se invece c'è la lettera *l* allora la lista viene passata utilizzando il meccanismo del linguaggio C di una sequenza di stringhe specificate individualmente. L'ultimo argomento deve essere un puntatore a `NULL`. Nel passare i parametri del programma da eseguire, è necessario ricordarsi di passare anche il nome del programma stesso (senza percorso), che è sempre il primo parametro quando si esegue il programma da shell.
- i sottoprogrammi che contengono la lettera *e* accettano come parametro un array di variabili d'ambiente (come array terminato da un `NULL` di puntatori a stringhe, in cui ogni elemento è del tipo `VARIABILE=valore`).

Le possibilità offerte e i sottoprogrammi della famiglia `exec` sono riportati in Tabella 1 e nella Figura 17.

Una volta effettuata la chiamata del sottoprogramma `exec()`, viene creata una nuova immagine del processo che cambia completamente il contenuto della memoria, copiando solamente le stringhe dei parametri passati e dell'ambiente in una nuova posizione. Alcuni attributi del processo rimangono però invariati:

- l'ID del processo e del processo padre;
- il proprietario del processo (ID utente) e del gruppo (ID gruppo);
- descrittori di file aperti;

Il Listato 11 mostra un esempio di utilizzo dei sottoprogrammi `exec`.

prefisso	modalità argomenti	percorso eseguibile	variabili d'ambiente	nome
exec	l			execl
	l	p		execlp
	l		e	execle
	v			execv
	v	p		execvp
	v		e	execve

Tabella 1: La famiglia di sottoprogrammi `exec`.

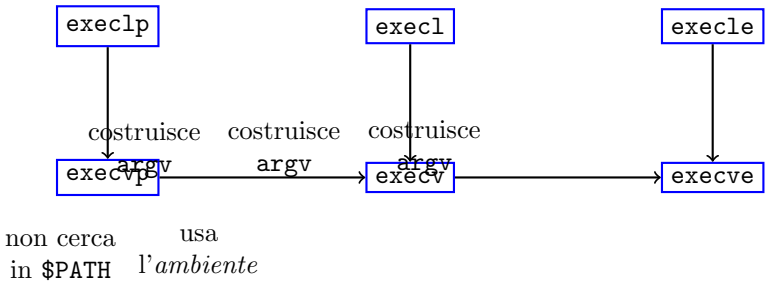

Figura 17: Famiglia di sottoprogrammi `exec`.

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

int main(int argc, char * argv[])
{
   int child_status;
   int cpid;
   cpid = fork( );
   if (cpid == 0)     //figlio
      execlp("ls", "ls", "-l", NULL);
   else {

      fprintf(stdout,"Processo padre!\n");
      wait(&child_status);
      if(WIFEXITED(child_status))
            printf("Figlio terminato normalmente, exitcode %d\n", WEXITSTATUS(child_status));
      else
            printf("Il processo figlio e' terminato in modo anomalo\n");

       printf ("Fine del main\n");
  }

  return 0;
}
```

Listing 11: Esecuzione di programmi da codice: esempio di `execlp()` con uso di `fork()`

Nel programma del Listato 11 viene usata il sottoprogramma `execlp()`, in cui il programma da eseguirsi verrà cercato nel percorso specificato dalla variabile d'ambiente `$PATH` (lettera *p*) e gli argomenti del programma verranno specificati come un elenco di stringhe, terminate da `NULL` (lettera *l*).

```
execlp("ls", "ls", "-l", NULL);
```

`ls` è il primo parametro e corrisponde al nome del programma da eseguire. A questo punto segue l'elenco degli argomenti con cui viene chiamato, il primo dei quali **deve essere** il nome del programma stesso, privo di qualsiasi percorso, ovvero `ls`. Con il terzo parametro iniziano gli argomenti veri e propri passati al programma da eseguire; nel caso specifico si passa un solo argomento, `-l`, e si mette il `NULL` a chiusura dell'elenco. Tutto ciò equivale ad eseguire da shell il comando `ls -l`.

Volendo vedere com sia possibile passare gli argomenti per il programma da eseguire, usando quindi i sottoprogrammi con la lettera *v*, si riscrive il codice prima vista come mostrato nel Listato 12.

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

int spawn (char* program, char** arg_list)
{
  pid_t child_pid;

  /* Duplica il processo  */
  child_pid = fork ();
  if (child_pid != 0)
    /* Padre  */
    return child_pid;
  else {
    /* esegue il PROGRAMMA richiesto, cercandolo in $PATH.  */
    execvp (program, arg_list);
    /* Si arriva qua solo se va in errore la execvp  */
    fprintf (stderr, "Errore in execvp\n");
    abort ();
  }
}

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

  /* Lista degli argomenti da passare al programma ls  */
  char* arg_list[] = {
    "ls",     /* argv[0], il nome del programma.  */
    "-l",
    NULL      /* Fine.  */
  };

  /* Crea un figlio ed esegue "ls" - Ignora il pid restituito.  */
  spawn ("ls", arg_list);

  wait(&child_status);
  if(WIFEXITED(child_status))
    printf("Figlio terminato normalmente, exitcode %d\n", WEXITSTATUS(child_status));
  else
    printf("Il processo figlio e' terminato in modo anomalo\n");

  printf("Fine del main\n");

  return 0;
}
```

Listing 12: Esecuzione di programmi da codice: esempio di `execvp()` con uso di `fork()`

In questo caso viene creata una variabile `char* arg_list[]` che è un array di puntatori a stringhe, inizializzato con i valori di seguito indicati

```
char* arg_list[] = {
    "ls",     /* argv[0], il nome del programma.  */
    "-l",
    NULL    /* Fine.  */
  };
```

L'inizializzazione immediata dell'array consente di non specificarne la dimensione (3 elementi); gli elementi sono scritti uno per riga per consentire di inserire i commenti; una scrittura equivalente è:

```
char* arg_list[] = {"ls","-l",NULL};
```

5 Riferimenti

Il materiale qua riportato è il risultato della traduzione e rielaborazione di materiale preso dalle fonti qui elencate e da altri appunti:

- Mark Mitchell, Jeffrey Oldham, and Alex Samuel, "Advanced UNIX Programming with Linux", Capitolo 3, che può essere scaricato all'indirizzo `http://www.advancedlinuxprogramming.com/`.
- Mauro Negri and Giuseppe Pelagatti, "I processi e le funzioni di Linux per la gestione dei processi"
- Harvey M. Deitel, Paul J. Deitel and David R. Choffnes, "Operating Systems", Pearson, Prentice Hall.