Gestione Dei Processi In Unix

Capitolo 8 - Stevens

Vitiello Autilia, PhD Student
Facoltà di Scienze MM.FF.NN.
Università degli Studi di Salerno
vitiello@dia.unisa.it
http://www.dia.unisa.it/dottorandi/avitiello/

Sommario

- Creazione di nuovi processi
- Processo di terminazione
- Esecuzione di programmi
- Altro….

Identificatori di processi

- Ogni processo ha un identificatore unico non negativo
- Process Id = 1 ⇨ il cui file di programma è contenuto in **/sbin/init**
  - invocato dal kernel alla fine del boot, legge il file di configurazione **/etc/inittab** dove ci sono elencati i file di inizializzazione del sistema (rc files) e dopo legge questi rc file portando il sistema in uno stato predefinito (multi user)
- non muore mai.
- è un processo utente (cioé non fa parte del kernel) di proprietà di root e ha quindi i privilegi del superuser

Identificatori di processi

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

`pid_t getpid (void);`	process ID del processo chiamante
`pid_t getppid (void);`	process ID del padre del processo chiamante
`uid_t getuid (void);`	real user ID del processo chiamante
`uid_t geteuid (void);`	effective user ID del processo chiamante
`gid_t getgid (void);`	real group ID del processo chiamante
`gid_t getegid (void);`	effective group ID del processo chiamante

Esempio

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

int main(void)
{
   printf("pid del processo = %d\n", getpid() );
   return (0);
}
```

Creazione di Processi

- L'unico modo per creare nuovi processi è attraverso la chiamata della funzione ***fork*** da parte di un processo già esistente.

- Spazio di indirizzi:
  - il figlio è una copia dello spazio degli indirizzi del padre;
  - Si può ottenere che il figlio esegua un nuovo programma attraverso la chiamata alla ***exec.***

- Esecuzione:
  - i processi padre e figlio continuano concorrentemente eseguendo le istruzioni successive alla fork;
    - non si sa se il figlio è eseguito prima del padre, questo dipende dall'algoritmo di scheduling usato dal kernel
  - Il processo padre può anche attendere uno o più figli usando le due funzioni ***wait***.

Funzione Fork

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

pid_t fork(void);
```

- Restituisce due volte:
  - 0 nel figlio,
  - pid del figlio nel padre
  - -1 in caso di errore

- un processo può avere più figli e non c'e' nessuna funzione che può dare al padre il pid dei suoi figli
- Il figlio può conoscere il pid del padre con la funzione `getppid();`

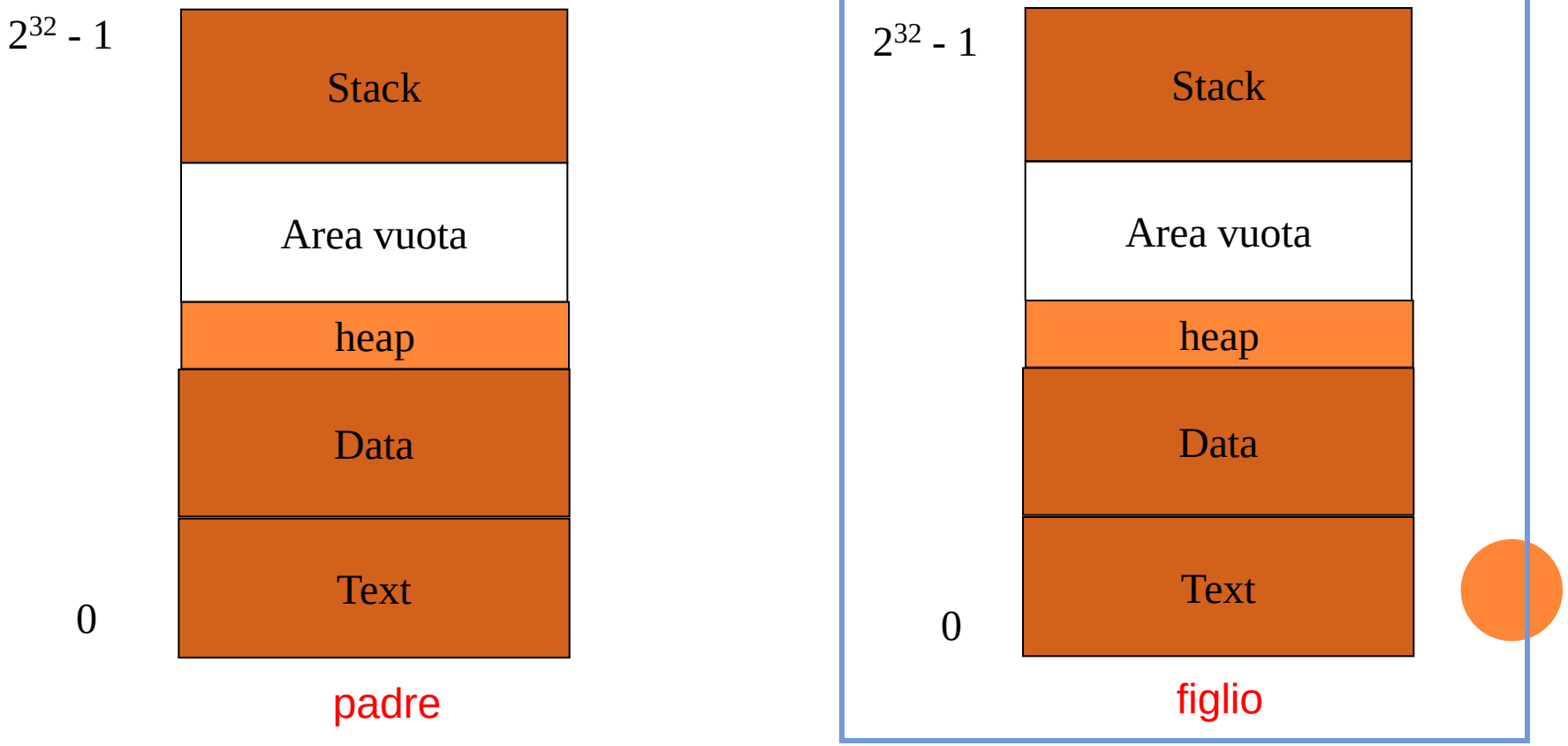
Funzione Fork: Copia dello Spazio degli indirizzi

- Nel caso in cui il figlio è una *copia* del padre
  - figlio prende una copia dei *dati*, *stack* e *heap;*
  - Non è una condivisione;

- ***Copy on write***: il padre e il figlio condividono data, stack e heap e il kernel li protegge settando i permessi a *read-only*. Solo se uno dei processi tenta di modificare una di queste regioni, allora essa viene copiata.

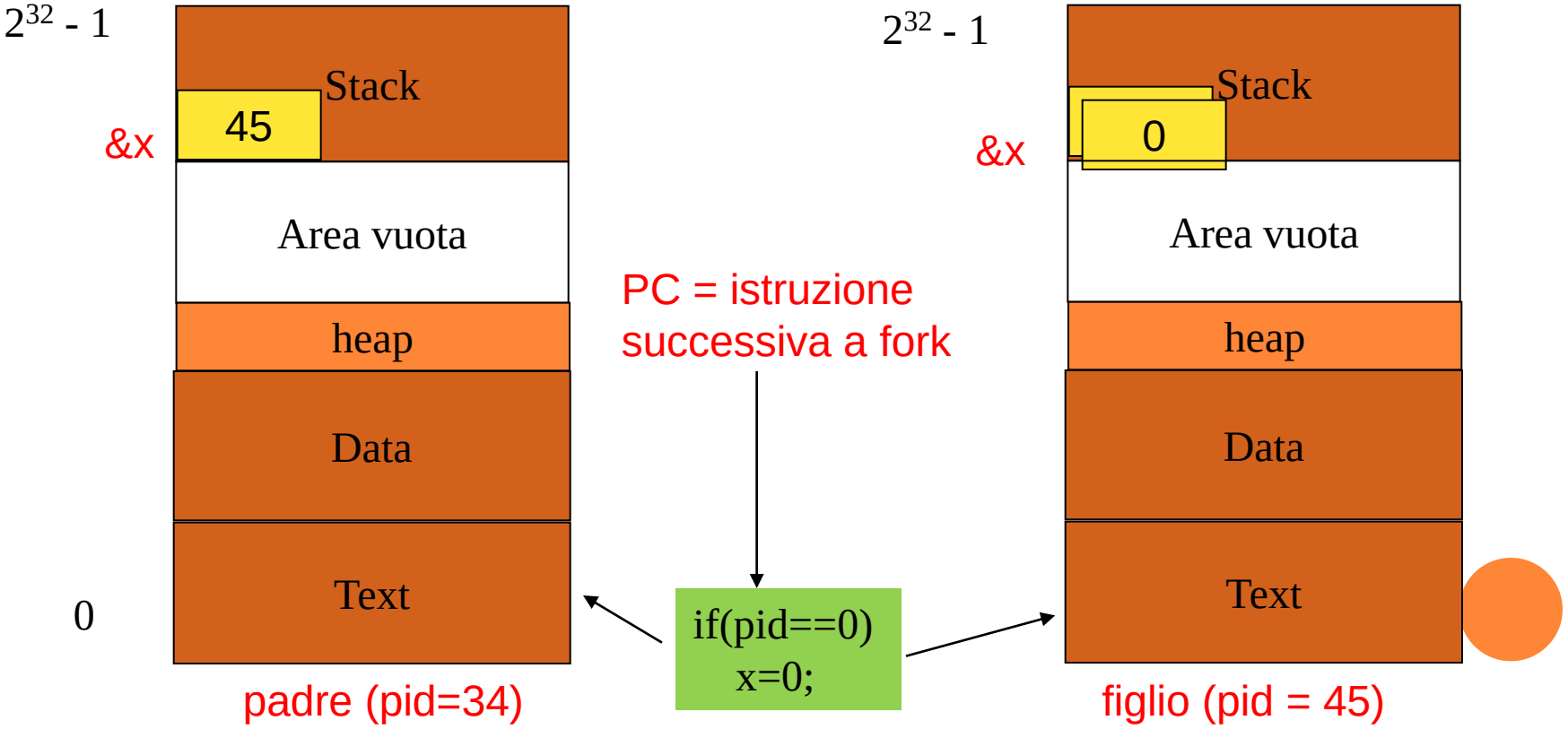
Funzione Fork: Copia dello Spazio degli indirizzi -1

- Spazio di indirizzamento di padre e figlio dopo una `fork` terminata con successo



Funzione Fork: Copia dello Spazio degli indirizzi -2

- Come prosegue l'esecuzione nei processi padre e figlio?



Esempio Fork

```
#include <sys/types.h>
int glob=10; /* dati inizializzati */
int main(void)
{
    int var=100;   /* vbl sullostack */
    pid_t pippo;
    printf("prima della fork\n");
      if( (pippo=fork()) == 0) {glob++; var++;}
        else sleep(2);
    printf("pid=%d, glob=%d, var=%d\n",getpid(),glob,var);
   exit(0);
}
```

Esempio fork

```
#include <sys/types.h>
int glob=10; /* dati inizializzati */
int main(void)
{
    int var=100;   /* vbl sullostack */
    pid_t pippo;
    printf("prima della fork\n");
      if( (pippo=fork()) == 0) {glob++; var++;}
        else sleep(2);
    printf("pid=%d, glob=%d, var=%d\n",getpid(),glob,var);
   exit(0);
}
```

```
$ a.out
prima della fork
pid=227, glob=11, var=101
pid=226, glob=10, var=100
$
```

Figlio

Padre

Conclusioni sulla Fork:

- Motivi di fallimento:
  - Si è raggiunto il limite per il numero di processi per il sistema;
  - Si è raggiunto il numero massimo (`CHILD_MAX`) di processi per l'utente (real user ID)

- Usi della fork:
  - un processo attende richieste (p.e. da parte di client nelle reti) allora si duplica
    - il figlio tratta la richiesta
    - il padre si mette in attesa di nuove richieste
  - un processo vuole eseguire un nuovo programma (p.e. la shell si comporta così) allora si duplica e il figlio lo esegue (chiamando una `exec`.)

Funzione vfork

- Ha la stessa chiamata e gli stessi valori di ritorno della funzione `fork`.

- Differenze:
  - `vfork` crea un nuovo processo ma senza copiare lo spazio di indirizzi del padre.
    - Pertanto è usata quando il figlio chiama subito la exec.
    - Bisogna stare attenti perché fin quando il figlio non esegue la exec, il figlio usa lo spazio di indirizzi del padre.
  - `vfork` garantisce che il figlio esegua prima del padre, fin quando non si trova ad eseguire la `exec`. Quando il figlio chiama la `exec` il padre viene risvegliato
    - Può creare deadlock se il figlio ha operazione che dipendono dal padre.

- Usata per ottimizzare

Terminazione di un Processo

- Terminazione ***normale***
  - ritorno dal `main` (chiamare la funzione return);
  - chiamata a `exit` (effettua operazioni di pulizia come chiusura degli stream aperti e poi ritorna al kernel)
  - chiamata a `_exit` (ritorna immediatamente al kernel)

- Terminazione ***anormale***
  - chiamata `abort` (sottocaso del caso successivo perché si genera un segnale `SIGABORT`);
  - arrivo di un segnale
    - interrupt generati da altri processi o dal kernel (p.e. quando si divide per zero)

Terminazione dei Processi

- quando un processo termina il kernel manda al padre il segnale `SIGCHLD`;
- il padre può ignorare il segnale (default) oppure lanciare una funzione (signal handler);
- in ogni caso il padre può chiedere informazioni sullo stato di uscita del figlio chiamando le funzioni **`wait`** e **`waitpid`**.

Funzione WAIT

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

pid_t wait (int *statloc);
```

- Descrizione: chiamata da un processo padre ottiene in **`statloc`** lo stato di terminazione di un figlio
- Restituisce:
  - pid del processo terminato se OK,
  - -1 in caso di errore (se il processo non ha figli)

Funzione Waitpid

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

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

- Descrizione: richiede la memorizzazione in **statloc** dello stato di terminazione del figlio specificato dal **pid** nel 1° argomento; il processo chiamante può bloccarsi o meno, in dipendenza del contenuto di **options**

- Restituisce:
  - pid del processo terminato se OK
  - 0 (vedremo dopo…)
  - -1 in caso di errore (se il processo non ha figli, oppure se lo specificato processo non esiste o non è figlio del processo chiamante)

Paramentri di Waitpid

- **Pid**:
  - pid == -1 (il processo aspetta per un qualsiasi figlio)
  - pid > 0 (il processo aspetta per il pid del figlio specificato)
  - pid == 0 (il processo aspetta per un qualsiasi processo con group ID pari a quello del processo chiamante)
  - pid < -1 (il processo aspetta per un qualsiasi processo con group ID = |pid|)

- **Options**
  - 0 (niente… come wait il processo si blocca in attesa)
  - WNOHANG (non si blocca se il figlio specificato dal pid non è disponibile immediatamente. In questo caso il valore di ritorno è 0.)

Differenze tra Wait e Waitpid

- `Wait` blocca sempre il processo fino a quando un figlio termina mentre `waitpid` ha una opzione che può prevenire la fase di bloccaggio

- `Waitpid` non aspetta il primo figlio che termina perché usa un argomento (il primo) per controllare quale processo attendere.

Stato di Terminazione del Figlio

- Le funzioni `wait` e `waitpid` hanno l'argomento **`statloc`** che è un puntatore ad intero.

- Se l'argomento `statloc` è non nullo, lo stato di terminazione del processo figlio che è appena terminato è memorizzato in esso
  - Se non ci interessa lo stato di terminazione del processo figlio settiamo questo argomento a NULL.

Stato di Terminazione del Figlio - 2

- Per comprendere lo stato di terminazione del processo figlio è possibile usare le seguenti tre funzioni:

- `WIFEXITED(status)`
  - True se status fa riferimento ad un figlio terminato regolarmente
- `WIFSTOPPED(status)`
  - True se status fa riferimento ad un figlio stopped
- `WIFSIGNALED(status)`
  - True se status fa riferimento ad un figlio terminato in maniera anormale in seguito ad un segnale

Cosa succede quando un processo termina?

- Se un figlio termina prima del padre, allora il padre ottiene lo status del figlio con `wait`.

- Se un figlio termina prima del padre, ma il padre non utilizza `wait`, allora il figlio diventa uno ***zombie.***

- Se un padre termina prima del figlio, allora il processo ***init*** diventa il nuovo padre.
  - Quando un figlio del processo *init* termina non diventa uno zombie perché *init* è scritto in modo tale da chiamare sempre una funzione `wait` quando i suoi figli terminano.

ESEMPIO: ZOMBIE.C

```
int main()
{
pid_t pid;
if ((pid=fork()) < 0)
       printf("fork error");
else if (!pid) /* figlio */
       exit(0);

sleep(2); /* padre */

system("ps"); /* dice che il figlio è zombie... STAT Z*/
exit(0);
}
```

Funzione System

```
#include <stdlib.h>

int system (const char *cmdstring);
```

- Serve ad eseguire un comando shell dall'interno di un programma.
- esempio: system("ls - a");

Funzione Exec

- L'unico modo per creare un processo è attraverso la **fork**
- L'unico modo per eseguire un programma è attraverso la **exec**
- La chiamata ad **exec** reinizializza un processo: il segmento istruzioni ed il segmento dati utente cambiano (viene eseguito un nuovo programma) mentre il segmento dati di sistema rimane invariato

Funzione Exec

```
#include <unistd.h>

int execl (const char *path, const char *arg0, ../* (char *) 0 */);

int execv (const char *path, char *const argv[ ]);

int execle (const char *path, const char *arg0, ../*(char *) 0, char *const envp[ ] */);

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

int execlp (const char *file, const char *arg0, ../*(char *)0 */);

int execvp (const char *file, char *const argv[ ]);
```

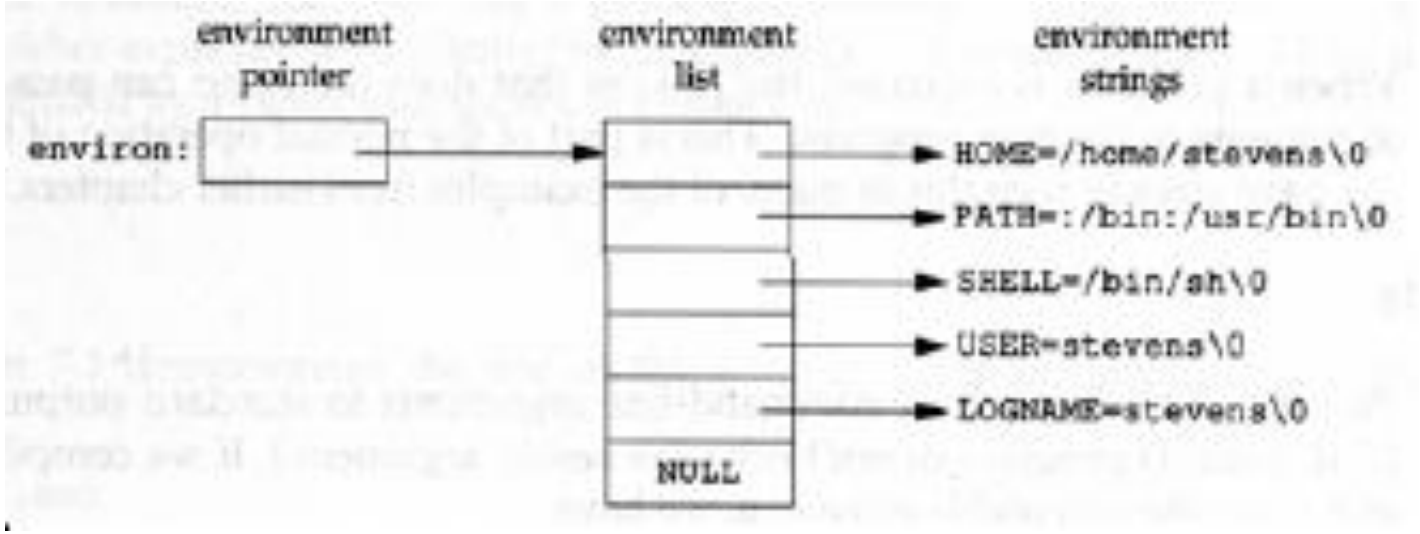
- Restituiscono:
  - -1 in caso di errore
  - non ritornano se OK.

Funzione Exec: differenze

- Nel nome delle `exec` ***l*** sta per list mentre ***v*** sta per vector
  - `execl`, `execlp`,`execle` prendono come parametro la lista degli argomenti da passare al file da eseguire
  - `execv`, `execvp`,`execve` prendono come parametro l'array di puntatori agli argomenti da passare al file da eseguire
- `execl`***p*** ed `execv`***p*** prendono come primo argomento un file e non un pathname, questo significa che il file da eseguire e' ricercato in una delle directory specificate in PATH
- `execl`***e*** ed `execv`***e*** passano al file da eseguire la enviroment list; un processo che invece chiama le altre `exec` copia la sua variabile environ per il nuovo file (programma)

Enviromental List

- Ad ogni programma è passata una lista di variabili d'ambiente sotto forma di un array di puntatori
- extern char **environ



Esempi

```
#include <unistd.h>……
printf("Sopra la panca \n");
execl("/bin/echo","echo","la","capra","campa",NULL);
……
```

```
$a.out
Sopra la panca
la capra campa
```

```
#include <unistd.h>……
char **parametri={"echo", "la", "capra", "campa",
   (char*)0};
printf("Sopra la panca \n");
execv ("/bin/echo", parametri);
……
```

Esempio 1

```
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"

int main(void)
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
            /* parent will wait for the child to complete
    */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

Esempio 2

- Scrivere un programma C che prenda in input da linea di comando il nome di un file. Il programma deve creare un figlio che visualizzi su standard output il contenuto del file e stampare a video se il figlio ha terminato correttamente (ovviamente per farlo deve aspettare la sua terminazione).

```
int main(int argc, char *argv[])
{
pid_t  pid;
int status;
      /* fork another process */
      pid = fork();
      if (pid < 0) { /* error occurred */
                  fprintf(stderr, "Fork Failed");
                  exit(-1);
      }
      else if (pid == 0) { /* child process */
                  execlp("/bin/cat", “cat", argv[1]);
/*oppure
argv[0]=“cat”;
execvp("/bin/cat", argv);
*/
      }
      else { /* parent process */
                  /* parent will wait for the child to complete */
                  wait (& status);
                  if(WIFEXITED(status))
                    printf (“Child correctly completed");
                  exit(0);
      }
}
```

Esercizi

1. Scrivete un programma C che crea un numero di processi specificati in input. Poi:
   - Ogni processo figlio deve stampare su terminale il proprio PID e terminare.
   - Il processo padre deve aspettare la terminazione di tutti i figli e terminare.

2. Scrivete un programma C che crea un numero di processi specificati in input. Poi:
   - Ogni processo figlio deve stampare su terminale il proprio PID e terminare.
   - Il processo padre deve aspettare la terminazione di tutti i figli *nell'ordine in cui sono stati creati* e terminare.