

SOLUZIONI ESERCITAZIONE DI VENERDÌ 30/04/2021

NOTA BENE:

- a) *Per ognuno dei programmi C che devono essere scritti per questa esercitazione (e per le prossime) deve essere scritto anche il makefile che produce il file eseguibile controllando tutti i possibili WARNING di compilazione (opzione -Wall) che devono sempre essere eliminati, come chiaramente anche gli errori di compilazione e di linking!*
- b) *Una volta ottenuto l'eseguibile, va verificato il funzionamento; in caso di passaggio di parametri va verificato anche che i controlli funzionino correttamente!*

1. Scrivere un programma in C **padreFiglioConStatus.c** che per prima cosa deve riportare su standard output il pid del processo corrente (processo padre) e poi deve creare un processo figlio: ricordarsi che il padre deve controllare il valore di ritorno della fork per assicurarsi che la creazione sia andata a buon fine. Il processo figlio deve riportare su standard output il suo pid e il pid del processo padre. Quindi, il processo figlio deve calcolare, in modo random, un numero compreso fra 0 e 99.

Al termine, il processo figlio deve ritornare al padre il valore random calcolato e il padre deve riportare su standard output il PID del figlio e il valore ritornato.

OSSERVAZIONE: per generare numeri random usare

- a) Chiamata alla funzione di libreria per inizializzare il seme:

```
#include <time.h>
srand(time(NULL));
```

- b) Funzione che calcola un numero random compreso fra 0 e n-1:

```
#include <stdlib.h>
int mia_random(int n)
{
    int casuale;
    casuale = rand() % n;
    return casuale;
}
```

```
$ cat padreFiglioConStatus.c
```

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

```
int mia_random(int n)
{
    int casuale;
    casuale = rand() % n;
    return casuale;
}
```

```
int main(int argc, char **argv)
{
```

```
    int pid;           /* per valore di ritorno della fork */
    int pidFiglio;     /* per valore di ritorno della wait */
    int status;        /* per usarlo nella wait */
    int ritorno;       /* per filtrare valore di uscita del figlio */
```

```
    printf("Sono il processo padre con pid %d\n", getpid());
```

```
    if ((pid = fork()) < 0)
```

```

{
    printf("ERRORE nella fork\n");
    exit(1);
}

if (pid == 0)
{
    /* figlio */
    int r; /* per valore generato random */

    printf("Sono il processo figlio con pid %d e sono stato generato
dal processo padre con pid %d\n", getpid(), getppid());
    srand(time(NULL)); /* inizializziamo il seme per la generazione
random di numeri */
    r=mia_random(100);
    /* il figlio deve tornare il valore random calcolato */
    exit(r);
}

/* padre */
/* il padre aspetta il figlio */
if ((pidFiglio=wait(&status)) < 0)
{
    printf("ERRORE nella wait %d\n", pidFiglio);
    exit(2);
}

if (pid == pidFiglio) printf("Terminato figlio con PID = %d\n",
    pidFiglio);
else
{
    /* problemi */
    printf("Il pid della wait non corrisponde al pid della fork!\n");
    exit(3);
}

if ((status & 0xFF) != 0)
    printf("Figlio terminato in modo involontario\n");
else
{
    ritorno=(int)((status >> 8) & 0xFF);
    printf("Il figlio con pid=%d ha ritornato %d\n", pidFiglio,
    ritorno);
}

exit(0);
}

```

2. Scrivere un programma in C **myGrepConFork-ridStError.c** che, partendo dal programma **myGrepConFork.c** visto a lezione, vada ad operare la ridirezione anche dello standard error.

```

$ cat myGrepConFork-ridStError.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

```

```

/* FILE: myGrepConFork-ridStError.c */
int main (int argc, char** argv)
{
    int pid;                                /* per fork */
    int pidFiglio, status, ritorno;         /* per wait padre */

    if (argc != 3)
    {
        printf("Errore nel numero di parametri che devono essere due
(stringa da cercare e nome del file dove cercare): %d\n", argc);
        exit(1);
    }

    /* generiamo un processo figlio dato che stiamo simulando di essere il
processo di shell! */
    pid = fork();
    if (pid < 0)
    {
        /* fork fallita */
        printf("Errore in fork\n");
        exit(2);
    }

    if (pid == 0)
    {
        /* figlio */
        printf("Esecuzione di grep da parte del figlio con pid %d\n",
getpid());
        /* ridirezioniamo lo standard output su /dev/null perche' ci
interessa solo se il comando grep ha successo o meno */
        close(1);
        open("/dev/null", O_WRONLY);
        /* ridirezioniamo anche lo standard error su /dev/null perche' ci
interessa solo se il comando grep ha successo o meno */
        close(2);
        open("/dev/null", O_WRONLY);
        execlp("grep", "grep", argv[1], argv[2], (char *)0);

        /* si esegue l'istruzione seguente SOLO in caso di fallimento della
execlp */
        /* ATTENZIONE SE LA EXEC FALLISCE NON HA SENSO FARE printf("Errore
in execlp\n"); DATO CHE LO STANDARD OUTPUT E' RIDIRETTO SU /dev/null */
        exit(-1); /* torniamo al padre un -1 che sara' interpretato come
255 e quindi identificato come errore */
    }

    /* padre aspetta subito il figlio appunto perche' deve simulare la shell
e la esecuzione in foreground! */
    pidFiglio = wait(&status);
    if (pidFiglio < 0)
    {
        printf("Errore wait\n");
        exit(3);
    }
    if ((status & 0xFF) != 0)
        printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
    else
    {

```

```

        ritorno=(int)((status >> 8) & 0xFF);
        printf("Il figlio con pid=%d ha ritornato %d (se 255 problemi!)\n",
pidFiglio, ritorno);
    }

    exit (0);
}

```

3. Scrivere un programma in C **myGrepConFork-ridStErrorEInput.c** che, partendo dal programma **myGrepConFork-ridStError.c** precedente, vada ad operare la ridirezione anche dello standard input in modo che venga letto il contenuto del file il cui nome è passato come secondo parametro.

```

$ cat myGrepConFork-ridStErrorEInput.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

/* FILE: myGrepConFork-ridStError.c */
int main (int argc, char** argv)
{
    int pid;                                /* per fork */
    int pidFiglio, status, ritorno;         /* per wait padre */

    if (argc != 3)
    {
        printf("Errore nel numero di parametri che devono essere due
(stringa da cercare e nome del file dove cercare): %d\n", argc);
        exit(1);
    }

    /* generiamo un processo figlio dato che stiamo simulando di essere il
processo di shell! */
    pid = fork();
    if (pid < 0)
    {
        /* fork fallita */
        printf("Errore in fork\n");
        exit(2);
    }

    if (pid == 0)
    {
        /* figlio */
        printf("Esecuzione di grep da parte del figlio con pid %d\n",
getpid());
        /* ridirezioniamo lo standard input in modo da leggere dal file
passato come secondo parametro */
        close(0);
        if (open(argv[2], O_RDONLY))
        {
            printf("Errore in apertura file %s\n", argv[2]);
            exit(-1);
        }
        /* ridirezioniamo lo standard output su /dev/null perche' ci
interessa solo se il comando grep ha successo o meno */
        close(1);
        open("/dev/null", O_WRONLY);
    }
}

```

```

        /* ridirezioniamo anche lo standard error su /dev/null perche' ci
interessa solo se il comando grep ha successo o meno */
        close(2);
        open("/dev/null", O_WRONLY);
        execlp("grep", "grēp", argv[1], (char *)0); /* in questo caso
passiamo come parametro solo la stringa da cercare */

        /* si esegue l'istruzione seguente SOLO in caso di fallimento della
execlp */
        /* ATTENZIONE SE LA EXEC FALLISCE NON HA SENSO FARE printf("Errore
in execlp\n"); DATO CHE LO STANDARD OUTPUT E' RIDIRETTO SU /dev/null */
        exit(-1); /* torniamo al padre un -1 che sara' interpretato come
255 e quindi identificato come errore */
    }

/* padre aspetta subito il figlio appunto perche' deve simulare la shell
e la esecuzione in foreground! */
pidFiglio = wait(&status);
if (pidFiglio < 0)
{
    printf("Errore wait\n");
    exit(3);
}
if ((status & 0xFF) != 0)
    printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
else
{
    ritorno=(int)((status >> 8) & 0xFF);
    printf("Il figlio con pid=%d ha ritornato %d (se 255 problemi!)\n",
pidFiglio, ritorno);
}

exit (0);
}

```

4. Scrivere un programma in C **padreFigliMultipli.c** che deve prevedere un singolo parametro che deve essere considerato un numero intero N che deve essere strettamente maggiore di 0 e strettamente minore di 255 e che per prima cosa deve riportare su standard output il pid del processo corrente (processo padre) insieme con il numero N . Il processo padre deve generare N processi figli. Ognuno di tali figli P_i deve riportare su standard output il suo pid insieme con il proprio indice d'ordine (i). Al termine, ogni processo figlio P_i deve ritornare al padre il proprio indice d'ordine e il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

```
$ cat padreFigliMultipli.c
```

```

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

int main (int argc, char **argv)
{
    int N;                /* numero di figli */
    int pid;              /* pid per fork */
    int i;                /* indice */
    int pidFiglio, status, ritorno; /* per wait e valore di
ritorno figli */
}

```

```

/* controllo sul numero di parametri: esattamente uno */
if (argc != 2)
{
    printf("Errore numero di parametri: %s vuole solo un numero\n",
argv[0]);
    exit(1);
}

/* convertiamo il parametro in numero */
N=atoi(argv[1]);
if (N <= 0 || N >= 255)
{
    printf("Errore l'unico parametro non e' un numero strettamente
positivo o non e' strettamente minore di 255: %d\n", N);
    exit(2);
}

printf("Sono il processo padre con pid %d e sto per generare %d figli\n",
getpid(), N);

/* creazione figli */
for (i=0; i < N; i++)
{
    if ((pid=fork())<0)
    {
        printf("Errore creazione figlio per l'indice %d\n", i);
        exit(3);
    }
    else if (pid==0)
    {
        /* codice figlio */
        printf("Sono il figlio %d di indice %d\n", getpid(), i);
        /* ogni figlio deve tornare il suo indice ordine */
        exit(i);
        /* al di la' del valore ritornato, e' ASSOLUTAMENTE
INDISPENSABILE che ci sia la presenza di una exit per fare in modo che il
processo figlio termini e NON esegua il ciclo for, che invece deve essere
eseguito solo dal padre */
    }
} /* fine for */

/* codice del padre */
/* Il padre aspetta i figli */
for (i=0; i < N; i++)
{
    pidFiglio = wait(&status);
    if (pidFiglio < 0)
    {
        printf("Errore in wait\n");
        exit(4);
    }
    if ((status & 0xFF) != 0)
        printf("Figlio con pid %d terminato in modo anomalo\n",
pidFiglio);
    else
    {
        ritorno=(int)((status >> 8) & 0xFF);
    }
}

```

```

        printf("Il figlio con pid=%d ha ritornato %d\n",
pidFiglio, ritorno);
    }
}

exit(0);
}

```

5. Scrivere un programma in C **padreFigliConSalvataggioPID.c** che deve prevedere un singolo parametro che deve essere considerato un numero intero N che deve essere strettamente maggiore di 0 e strettamente minore di 155 e che per prima cosa deve riportare su standard output il pid del processo corrente (processo padre) insieme con il numero N . Il processo padre deve generare N processi figli. Ognuno di tali figli P_i deve riportare su standard output il suo pid insieme con il proprio indice d'ordine (i) e quindi deve calcolare in modo random (vedi sopra) un numero compreso fra 0 e 100+i. Al termine, ogni processo figlio P_i deve ritornare al padre il valore random calcolato e il padre deve stampare su standard output il PID di ogni figlio, insieme con il numero d'ordine derivante dalla creazione, e il valore ritornato.

```

$ cat padreFigliConSalvataggioPID.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

int mia_random(int n)
{
    int casuale;
    casuale = rand() % n;
    return casuale;
}

int main (int argc, char **argv)
{
    int N; /* numero di figli */
    int *pid; /* array di pid per fork */

    int i, j; /* indici */
    int pidFiglio, status, ritorno; /* per wait e valore di ritorno figli */

    /* controllo sul numero di parametri: esattamente uno */
    if (argc != 2)
    {
        printf("Errore numero di parametri: %s vuole un numero\n",
argv[0]);
        exit(1);
    }

    /* convertiamo il parametro in numero */
    N=atoi(argv[1]);
    if (N <= 0 || N >= 155)
    {
        printf("Errore l'unico parametro non e' un numero strettamente
positivo o non e' strettamente minore di 155: %d\n", N);
        exit(2);
    }
}

```

```

}

/* allocazione pid */
if ((pid=(int *)malloc(N*sizeof(int))) == NULL)
{
    printf("Errore allocazione pid\n");
    exit(3);
}

printf("Sono il processo padre con pid %d\n", getpid());

/* creazione figli */
for (i=0;i<N;i++)
{
    if ((pid[i]=fork())<0)
    {
        printf("Errore creazione figlio\n");
        exit(4);
    }
    else if (pid[i]==0)
    {
        /* codice figlio */
        int r;
        srand(time(NULL)); /* per valore generato random */
        /* inizializziamo il seme per la
generazione random di numeri */
        printf("Sono il figlio %d di indice %d\n", getpid(), i);
        r=mia_random(100+i);
        /* ogni figlio deve tornare il numero random calcolato */
        exit(r);
    }
} /* fine for */

/* codice del padre */

/* Il padre aspetta i figli */
for (i=0; i < N; i++)
{
    pidFiglio = wait(&status);
    if (pidFiglio < 0)
    {
        printf("Errore in wait %d\n", pidFiglio);
        exit(5);
    }
    if ((status & 0xFF) != 0)
        printf("Figlio con pid %d terminato in modo anomalo\n",
pidFiglio);
    else
    {
        ritorno=(int)((status >> 8) & 0xFF);
        for (j=0; j < N; j++)
            if (pidFiglio == pid[j])
            {
                printf("Il figlio con pid=%d di indice %d
ha ritornato %d\n", pidFiglio, j, ritorno);
                break;
            }
    }
}
}

```



```
exit(0);
}
```

6. Scrivere un programma in C **padreFigliConConteggioOccorrenze.c** che deve prevedere un numero variabile $N+1$ di parametri: i primi N (con N maggiore o uguale a 2, da controllare) che rappresentano N nomi di file (F1, F2. ... FN), mentre l'ultimo rappresenta un singolo carattere Cx (da controllare). Il processo padre deve generare N processi figli (P0, P1, ... PN-1): i processi figli P_i (con i che varia da 0 a $N-1$) sono associati agli N file F_f (con $f = i+1$). Ogni processo figlio P_i deve leggere dal file associato contando le occorrenze del carattere Cx .

Al termine, ogni processo figlio P_i deve ritornare al padre il numero di occorrenze (*NOTA BENE: si può supporre minore di 255*) e il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

```
$ cat padreFigliConConteggioOccorrenze.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>

int main (int argc, char **argv)
{
    char Cx;                                /* carattere che i figli devono
cercare nel file a loro associato */
    int N;                                  /* numero di figli */
    int pid;                                /* pid per fork */
    int i;                                  /* indice */
    int totale=0;                            /* serve per calcolare il numero
di occorrenze: in questo caso abbiamo usato un semplice int perche' la
specifica dice che si può supporre minore di 255 */
    int fd;                                  /* per la open */
    char c;                                  /* per leggere i caratteri dal
file */
    int pidFiglio, status, ritorno; /* per valore di ritorno figli */

    /* controllo sul numero di parametri: almeno due nomi file e un carattere */
    if (argc < 4)
    {
        printf("Errore numero di parametri: i parametri passati a %s sono
solo %d\n", argv[0], argc);
        exit(1);
    }

    /* controlliamo che l'ultimo parametro sia un singolo carattere */
    if (strlen( argv[argc-1]) != 1)
    {
        printf("Errore ultimo parametro non singolo carattere dato che e'
%s\n", argv[argc-1]);
        exit(2);
    }

    /* individuiamo il carattere da cercare */
    Cx = argv[argc-1][0];

    /* individuiamo il numero di file/processi */
```

```

N=argc-2;

printf("Sono il processo padre con pid %d e creero' %d processi figli che
cercheranno il carattere %c nei file passati come parametri\n", getpid(),
N, Cx);

/* creazione figli */
for (i=0;i<N;i++)
{
    if ((pid=fork())<0)
    {
        printf("Errore creazione figlio\n");
        exit(3);
    }
    else if (pid==0)
    {
        /* codice figlio */
        printf("Sono il figlio %d di indice %d\n", getpid(), i);
        /* apriamo il file (deleghiamo ad ogni processo figlio, il
controllo che i singoli parametri (a parte l'ultimo) siano nomi di file */
        /* notare che l'indice che dobbiamo usare e' i+1 */
        /* in caso di errore decidiamo di ritornare -1 che sara'
interpretato dal padre come 255 e quindi un valore non valido! */
        if ((fd = open(argv[i+1], O_RDONLY)) < 0)
        {
            printf("Errore: FILE %s NON ESISTE\n", argv[i+1]);
            exit(-1);
        }
        /* leggiamo il file */
        while (read (fd, &c, 1) != 0)
            if (c == Cx) totale++;          /* se troviamo il
carattere incrementiamo il conteggio */

        /* ogni figlio deve tornare il numero di occorrenze e
quindi totale */
        exit(totale);
    }
} /* fine for */

/* codice del padre */
/* Il padre aspetta i figli */
for (i=0; i < N; i++)
{
    pidFiglio = wait(&status);
    if (pidFiglio < 0)
    {
        printf("Errore in wait\n");
        exit(4);
    }
    if ((status & 0xFF) != 0)
        printf("Figlio con pid %d terminato in modo anomalo\n",
pidFiglio);
    else
    {
        ritorno=(int)((status >> 8) & 0xFF);
        printf("Il figlio con pid=%d ha ritornato %d (se 255
problemi!)\n", pidFiglio, ritorno);
    }
}

```

```

}

exit(0);
}

```

7. Scrivere un programma in C **padreFigliNipotiConExec.c** che deve prevedere un numero variabile N di parametri (con N maggiore o uguale a 3, da controllare) che rappresentano nomi di file (F1, F2. ... FN). Il processo padre deve generare N processi figli (P0, P1, ... PN-1): i processi figli P_i (con i che varia da 0 a N-1) sono associati agli N file F_f (con $f = i+1$). Ogni processo figlio P_i deve, per prima cosa, creare un file F_{Out} il cui nome deve risultare dalla concatenazione del nome del file associato F_f con la stringa ".sort". Quindi, ogni processo figlio P_i deve creare a sua volta un processo nipote PP_i : ogni processo nipote PP_i esegue concorrentemente e deve ordinare il file F_f secondo il normale ordine alfabetico usando in modo opportuno il filtro sort di UNIX/Linux riportando il risultato di tale comando sul file F_{Out} . Al termine, ogni processo nipote PP_i deve ritornare al figlio il valore ritornato dal comando `sort` (in caso di insuccesso nella esecuzione del `sort` deve essere tornato il valore -1) e, a sua volta, ogni processo figlio P_i lo deve ritornare al padre. Il padre deve stampare, su standard output, i PID di ogni figlio con il corrispondente valore ritornato.

```

$ cat padreFigliNipotiConExec.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>

#define PERM 0644

int main (int argc, char **argv)
{
    int N;                /* numero di figli */
    int pid;              /* pid per fork */
    int i;                /* indice */
    char *FOut;           /* nome de file da creare da parte
dei figli */
    int fdw;              /* per la creat */
    int pidFiglio, status, ritorno; /* per valore di ritorno figli */

    /* controllo sul numero di parametri: almeno tre nomi file */
    if (argc < 4)
    {
        printf("Errore numero di parametri: i parametri passati a %s sono
solo %d\n", argv[0], argc);
        exit(1);
    }

    /* individuiamo il numero di file/processi */
    N=argc-1;

    printf("Sono il processo padre con pid %d e creero' %d processi figli che
generanno ognuno un nipote\n", getpid(), N);

    /* creazione figli */
    for (i=0;i<N;i++)
    {
        if ((pid=fork())<0)
        {

```

```

        printf("Errore creazione figlio %d-esimo\n", i);
        exit(3);
    }
    else if (pid==0)
    {
        /* codice figlio */
        printf("Sono il figlio %d di indice %d\n", getpid(), i);
        /* in caso di errore (sei nei figli che nei nipoti)
decidiamo di ritornare -1 che sara' interpretato dal padre come 255 e
quindi un valore non valido! */
        /* i figli devono creare il file specificato */
        FOut=(char *)malloc(strlen(argv[i+1]) + 6); /* bisogna
allocare una stringa lunga come il nome del file + il carattere '.' + i
caratteri della parola sort (4) + il terminatore di stringa */
        if (FOut == NULL)
        {
            printf("Errore nelle malloc\n");
            exit(-1);
        }
        /* copiamo il nome del file associato al figlio */
        strcpy(FOut, argv[i+1]);
        /* concateniamo la stringa specificata dal testo */
        strcat(FOut, ".sort");
        fdw=creat(FOut, PERM);
        if (fdw < 0)
        {
            printf("Impossibile creare il file %s\n", FOut);
            exit(-1);
        }
        /* chiudiamo il file creato che il figlio non usa */
        close(fdw);

        if ( (pid = fork()) < 0) /* ogni figlio crea un nipote */
        {
            printf("Errore nella fork di creazione del
nipote\n");
            exit(-1);
        }
        if (pid == 0)
        {
            /* codice del nipote */
            printf("Sono il processo nipote del figlio di
indice %d e pid %d sto per eseguire il comando sort per il file %s\n", i,
getpid(), argv[i+1]);
            /* chiudiamo lo standard input */
            close(0);
            /* apriamo il file associato in sola lettura */
            if (open(argv[i+1], O_RDONLY) < 0)
            {
                printf("Errore: FILE %s NON ESISTE\n",
argv[i+1]);
                exit(-1);
            }
            /* chiudiamo lo standard output */
            close(1);
            /* apriamo il file creato in sola scrittura */
            if (open(FOut, O_WRONLY) < 0)
            {

```

```

        printf("Errore: FILE %s NON si riesce ad
aprire in scrittura\n", FOut);
        exit(-1);
    }
    /* Il nipote diventa il comando sort: bisogna usare
le versioni dell'exec con la p in fondo in modo da usare la variabile di
ambiente PATH: NON serve alcun parametro */
    execlp("sort", "sort", (char *)0);

    /* Non si dovrebbe mai tornare qui!!*/
    /* usiamo perror che scrive su standard error,
dato che lo standard output e' collegato alla pipe */
    perror("Problemi di esecuzione del sort da parte
del nipote");
    exit(-1);
}
/* il figlio deve aspettare il nipote per ritornare al
padre il valore tornato dal nipote */
pid = wait(&status);
if (pid < 0)
{
    printf("Errore in wait\n");
    exit(-1);
}
if ((status & 0xFF) != 0)
{
    printf("Nipote con pid %d terminato in modo
anomalo\n", pid);
    exit(-1);
}
else
    ritorno=(int)((status >> 8) & 0xFF);
/* il figlio ritorna il valore ricevuto dal nipote */
exit(ritorno);
}
} /* fine for */

/* codice del padre */
/* Il padre aspetta i figli */
for (i=0; i < N; i++)
{
    pidFiglio = wait(&status);
    if (pidFiglio < 0)
    {
        printf("Errore in wait\n");
        exit(4);
    }
    if ((status & 0xFF) != 0)
        printf("Figlio con pid %d terminato in modo anomalo\n",
pidFiglio);
    else
    {
        ritorno=(int)((status >> 8) & 0xFF);
        printf("Il figlio con pid=%d ha ritornato %d (se 255
problemi!)\n", pidFiglio, ritorno);
    }
}
}

```

```
exit(0);  
}
```