Programmazione di Sistema – 3

Lucidi per il corso di Laboratorio di Sistemi Operativi tenuto da Paolo Baldan presso l'Università Ca' Foscari di Venezia, anno accademico 2004/2005. Parte di questo materiale è rielaborato dai lucidi del Corso di Laboratorio di Sistemi Operativi tenuto da Rosario Pugliese presso l'Università di Firenze, anno accademico 2001/02.

Gestione dei Processi

Gestione dei Processi

Un processo Unix si compone delle seguenti parti

code area
contiene il codice (text) eseguibile

data area
contiene i dati statici del processo

- *inizializzati*
- *non inizializzati*

stack area
contiene i dati dinamici o "temporanei" del processo (più precisamente environment, cmd line args, stack e heap)

Gestione dei Processi

Una tabella nel kernel, detta process table, contiene una entry, detta Process Control Block (PCB), per ciascun processo

- *PID (Process Id – identificatore unico del processo nel sistema)*
- *PPID (Parent PID)*
- *stato (running, runnable, sleeping, suspended, idle, zombie)*
- *real e effective user ID e GID, supplem. GIDs*
- *puntatori alla user area*
- *segnali pendenti*

Gestione dei Processi

user area

struttura dati (gestita dal kernel) che contiene informazioni per la gestione del processo

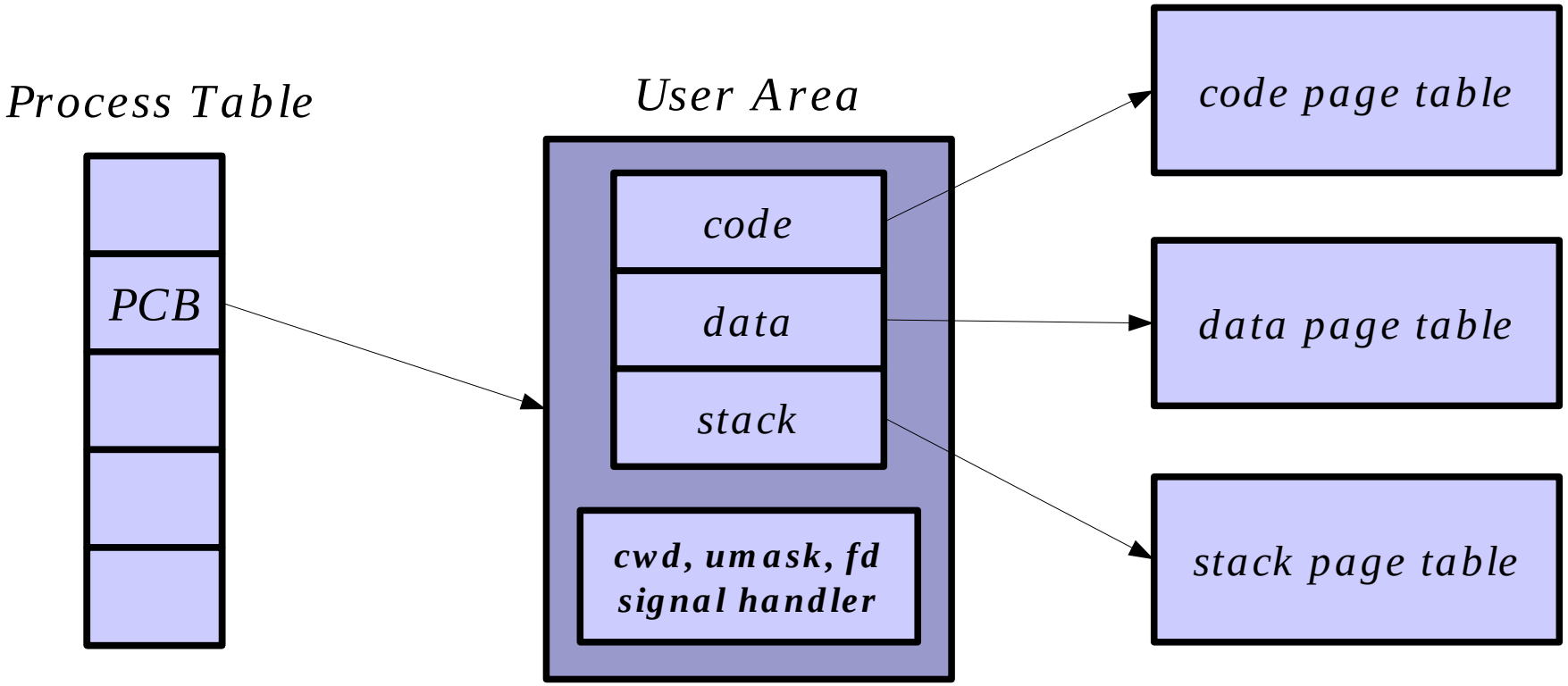
- *signal handler*
- *descrittori di file aperti*
- *uso CPU (per lo scheduling)*
- *puntatori a(lle page table di) code, data, stack area*

page table

usate dal sistema di gestione della memoria

Gestione dei Processi

Schema di massima ...



Gestione dei Processi

L'unico modo di creare un nuovo processo in Unix è duplicare un processo esistente (tramite una **`fork()`***)*

Quando un processo è duplicato, il processo padre ed il processo figlio sono virtualmente identici

- *il codice, i dati e la pila del figlio sono una copia di quelli del padre ed il processo figlio continua ad eseguire lo stesso codice del padre.*
- *differiscono per alcuni aspetti quali PID, PPID e risorse a run-time (es. segnali pendenti, file lock);*

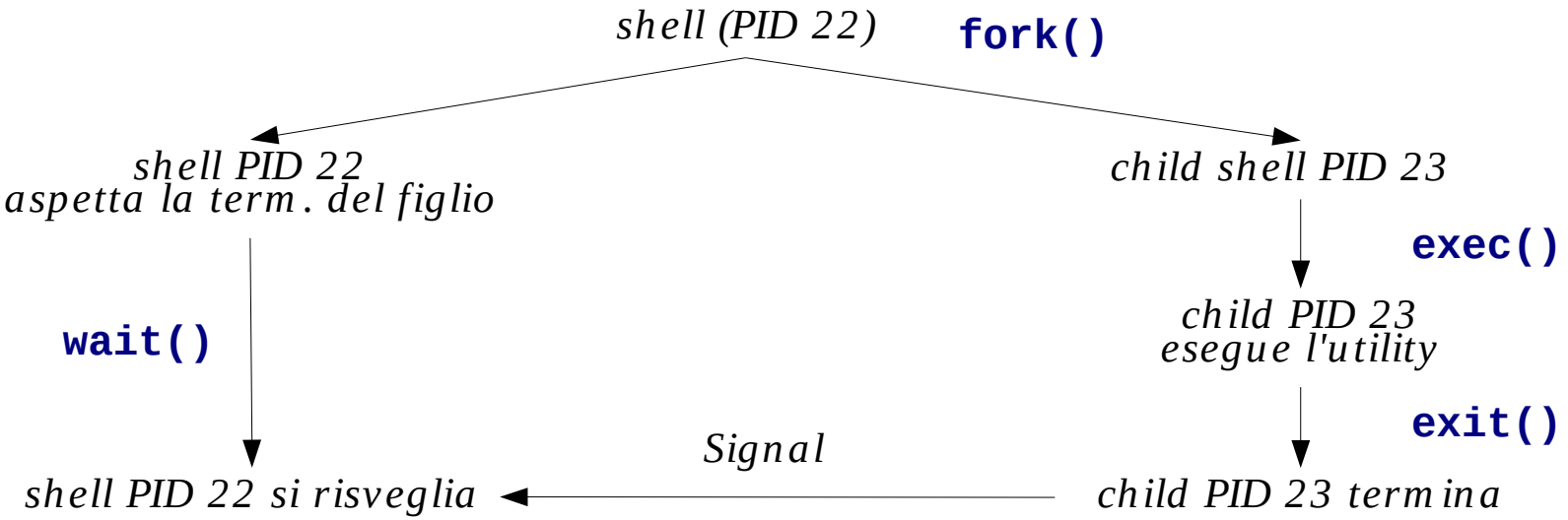
Quando un processo figlio termina (tramite una **`exit()`***), la sua terminazione è comunicata al padre (tramite un segnale) e questi si comporta di conseguenza.*

Gestione dei Processi

Un processo padre si può sospendere (tramite una **`wait()`***) in attesa della terminazione del figlio.*

Esempio: la shell che esegue un comando in foreground.

Un processo (figlio) può sostituire il proprio codice con quello di un altro file eseguibile (tramite una **`exec()`***), perciò differenziandosi dal padre.*



Gestione dei Processi

All'avvio del SO c'è un solo processo visibile chiamato **`init()`**, *con PID 1.*

Quindi **`init()`** *è l'antenato comune di tutti i processi esistenti in un dato momento nel sistema.*

Esempio:

> **`init()`** *crea i processi* **`getty()`** *responsabili di gestire i login degli utenti.*

Creare un nuovo processo: fork()

`pid_t fork (void)`

duplica il processo chiamante.

- *Il processo figlio è una copia quasi esatta del padre*
  - *eredita dal padre una copia del codice, dei dati, della pila, dei descrittori di file aperti e della tabella dei segnali...*
  - *ma ha numeri PID e PPID differenti.*
- *Se fork() ha successo, restituisce il PID del figlio al padre ed il valore 0 al figlio; altrimenti, restituisce -1 al padre e non crea alcun figlio.*

Si noti che **`fork()`** *è invocata da un processo, ma restituisce il controllo a due processi!*

Questi eseguono lo stesso codice in modo concorrente ma con aree dati e stack separate.

Ottenere il PID: getpid() e getppid()

```
pid_t getpid (void)
pid_t getppid (void)
```

`getpid()` *restituisce il PID del processo invocante.*

`getppid()` *restituisce il PPID (cioè il PID del padre) del processo invocante.*

Hanno sempre successo.

Il PPID di **`init()`** *(il processo con PID 1) è ancora 1.*

Esempio: myfork.c

```
#include <stdio.h>          /* Un programma che si sdoppia e
                               mostra il PID e PPID dei due
int main (void) {              processi componenti */
  int pid;
  printf ("I'm the original process with PID %d and PPID %d.\n",
            getpid (), getppid ());
  pid = fork (); /* Duplicate. Child and parent continue from here */
  if (pid != 0) { /* pid is non-zero, so I am the parent */
     printf ("I'm the parent process with PID %d and PPID %d.\n",
               getpid (), getppid ());
     printf ("My child's PID is %d.\n", pid); /* do not wait(); */
  }
  else { /* pid is zero, so I am the child */
     printf ("I'm the child process with PID %d and PPID %d.\n",
               getpid (), getppid ());
  }
  printf ("PID %d terminates.\n", getpid () );
                                    /* Both processes execute this */
  return 0;
}
```

Esempio: myfork.c

```
$ myfork
I'm the original process with PID 724 and PPID 572.
I'm the parent process with PID 724 and PPID 572.
I'm the child process with PID 725 and PPID 724. [ and PPID  1]
PID 725 terminates.
My child's PID is 725
PID 724 terminates.
$
```

Nell'esempio, il padre non aspetta la terminazione del figlio per terminare a sua volta.

Se un padre termina prima di un suo figlio, il figlio diventa orfano e viene adottato dal processo **`init()`**.

Terminazione di un processo

A meno che il sistema vada in crash, un processo in esecuzione può terminare nei seguenti modi

- *invoca la system call* **`exit()`**
- *invoca* **`return`** *durante l'esecuzione della funzione* **`main`**
- *termina implicitamente al completamento dell'esecuzione della funzione main*
- *riceve un segnale che ne causa la terminazione.*

Terminazione di un processo: exit()

`void exit (int status)`

chiude tutti i descrittori di file di un processo, dealloca le aree codice, dati e stack, e quindi fa terminare il processo;

non restituisce mai il controllo al chiamante.

exit(): Codice di Terminazione

Quando un figlio termina, il padre riceve un segnale **SIGCHLD** *ed il figlio aspetta che il suo codice di terminazione* **status** *sia accettato.*

- *Solo gli 8 bit meno significativi di* **status** *sono utilizzati, così i valori del codice vanno da 0 a 255.*
- *Un processo accetta il codice di terminazione di un figlio eseguendo una* **wait()**.

Il codice di terminazione di un processo figlio può essere usato dal processo padre per vari scopi.

- *Esempio: le shell interpretano l'exit status come valore logico e permettono di accedere al codice di terminaz. dell'ultimo processo figlio terminato tramite una variabile speciale (***bash** *usa* **?**, **C-shell** *usa* **status**).

Processi adottati

Se un processo padre termina prima di un figlio, quest'ultimo viene detto orfano.

> *Il kernel assicura che tutti i processi orfani siano adottati da* **`init()`** *ed assegna loro* **`PPID`** *1.*
>
> **`init()`** *accetta sempre i codici di terminazione dei figli.*

(vedi **`myfork.c`***)*

Processi zombie

Un processo che termina non scompare dal sistema fino a che il padre non ne accetta il codice di terminazione.

Un processo che sta aspettando che il padre accetti il suo codice di terminazione è chiamato processo zombie. (**<defunct>** *in* **ps**)

Se il padre non termina e non esegue mai una **wait()**, *il codice di terminazione non sarà mai accettato ed il processo resterà sempre uno zombie.*

Uno zombie non ha aree codice, dati o pila allocate, quindi non usa molte risorse di sistema ma continua ad avere un PCB nella Process Table (di grandezza fissa). Quindi la presenza di molti processi zombie potrebbe costringere l'amministratore ad intervenire.

Attendere la terminazione: wait()

`pid_t wait (int *status)`

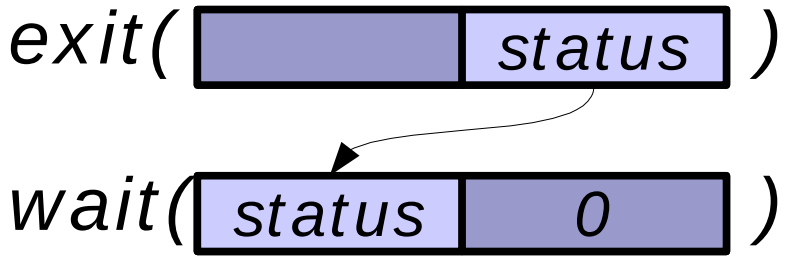
sospende un processo fino alla termin. di uno dei figli.

Se **`wait()`** *ha successo, restituisce il* **`PID`** *del figlio che è terminato e gestisce* **`status`**

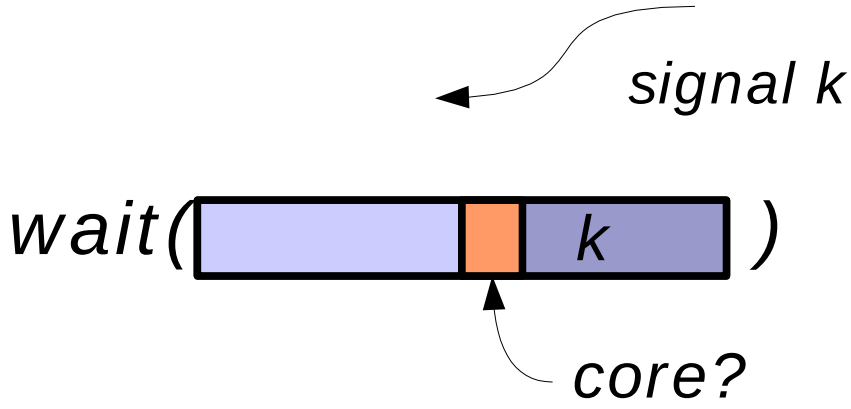
- *terminazione "normale": il figlio è terminato eseguendo* **`exit()`** *o* **`return()`** *(o fine del* **`main()`***)*
  - *il byte meno significativo di* **`status`** *è 0*
  - *il byte più significativo è il codice di terminazione del figlio (byte meno significativo del valore restituito);*
- *terminazione causata da un segnale*
  - *i 7 bit meno significativi di* **`status`** *sono posti al numero del segnale che ha causato la terminazione del figlio;*
  - *il bit rimanente del byte meno significativo è posto ad 1 se il figlio ha prodotto un core dump.*

Wait: graficamente

Nel caso di terminazione “normale”



Nel caso di terminazione per segnale



Attendere la terminazione: wait()

Se un processo esegue una **`wait()`** *e non ha figli, la* **`wait()`** *termina immediatamente con valore -1.*

Se un processo esegue una **`wait()`** *ed uno o più dei suoi figli sono già zombie,* **`wait()`** *termina immediatamente con lo stato di uno degli zombie.*

Variante (che permette di aspettare un figlio specifico)

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

Differenziare un processo: exec()

```
int execl (const char *path, const char *arg0,
           ..., const char *argn, NULL)

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

int execlp (const char *path, const char *arg0,
            ..., const char *argn, NULL)

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

Le system call della famiglia **exec()** *sostituiscono codice, dati e stack del processo invocante con quelli dell'eseguibile individuato dal pathname* **path** *ed eseguono il nuovo codice.*

PID *e* **PPID** *restano immutati*

Differenziare un processo: exec()

In effetti sono tutte routine di libreria C che invocano la vera system call **execve()**.

Se l'eseguibile non viene trovato, restituiscono -1

Una **exec()** *che ha successo non restituisce mai il controllo al chiamante (quando* **exec()** *termina, termina anche il chiamante).*

execl() *è identica a* **execlp()** *e* **execv()** *è identica a* **execvp()** *a parte per il fatto che*

- **execl()** *e* **execv()** *richiedono che sia fornito il pathname assoluto o quello relativo del file eseguibile*
- **execlp()** *e* **execvp()** *usano la variabile d'ambiente* **PATH** *per trovare l'eseguibile path.*

Differenziare un processo: exec()

`execl()` *e* **`execlp()`**

invocano l'eseguibile con argomenti **`arg1,..., argn`**.

la lista degli argomenti dev'essere terminata da **`NULL`**.

`arg0` *è il nome dell'eseguibile.*

`execv()` *e* **`execvp()`**

invocano l'eseguibile con argomenti **`arg[1]`**, ..., **`arg[n]`**.

`arg[0]` *è il nome dell'eseguibile.*

`arg[n+1]` *dev'essere* **`NULL`**.

Esempio: myexec.c

Il programma **myexec.c** *mostra un messaggio e quindi rimpiazza il suo codice con quello dell'eseguibile* **ls** *(con opzione* **-l***). Si noti che non viene invocata nessuna* **fork()**.

La **execl()**, *eseguita con successo, non restituisce il controllo all'invocante (quindi il secondo* **printf()** *non è mai eseguito).*

```
#include <stdio.h>

int main (void) {
  printf("I'm process %d and I'm about to exec an ls -l\n", getpid());
  execl("/bin/ls", "ls", "-l", NULL); /* Execute ls -l */
  printf("This line should never be executed\n");
}
```

Differenziare un processo: esempio

Esempio di esecuzione ...

```
$ myexec
I'm process 797 and I'm about to exec an ls -l
total 3324
drwxr-xr-x   3 rossi   users        4096 May 16 19:14 Glass
-rwxr-xr-x   1 rossi   users       22199 Mar 17 18:34 lez1.sxi
-rwxr-xr-x   1 rossi   users       25999 May 21 17:14 lez20.sxi
$
```

Elaborazione in background: background.c

Programma **background cmd args:**
Usa **fork()** *ed* **exec()** *per eseguire il comando* **cmd** *con i suoi argomenti in background.*

La lista degli argomenti è passata ad **execvp()** *tramite il secondo argomento* **&argv[1]**. *Si ricordi che* **execvp()** *permette di usare la variabile* **PATH** *per trovare l'eseguibile.*

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  if (fork () == 0) {      /* Child */
     execvp (argv[1], &argv[1]); /* Execute other program */
     fprintf (stderr, "Could not execute %s\n", argv[1]);
  }
}
```

Elaborazione in background: background.c

```
$ background sleep 5
$ ps
  PID TTY          TIME CMD
  822 pts/0    00:00:00 bash
  925 pts/0    00:00:01 emacs
  965 pts/0    00:00:00 sleep
  969 pts/0    00:00:00 ps
$
```

La lista degli argomenti è chiusa, come necessario, da **NULL** *perché, in un programma C, vale sempre* **argv[argc]=NULL**.

Cambiare directory: chdir()

```
int chdir (const char *pathname)
int fchdir (int fd)
```

Ogni processo ha associata una working directory, usata per interpretare pathname relativi.

Un processo figlio eredita la working dir. del padre.

`chdir()` *modifica la working directory di un processo in* **`pathname`**.

`fchdir()` *opera come* **`chdir()`** *ma prende come argomento un descrittore di file aperto* **`fd`**.

Perché **`[f]chdir()`** *abbia successo, il processo che la invoca deve avere permessi di esecuzione (x) su tutte le directory nel path.*

Modificare le priorità: nice()

`int nice (int delta)`

aggiunge **`delta`** *al valore corrente della priorità del processo invocante.*

- *I valori legali della priorità vanno da -20 a +19; se viene specificato un delta che porta oltre questi limiti il valore è troncato al limite.*
- *La priorità di un processo influenza la quantità di tempo di CPU che è allocata al processo (nice minore corrisponde a priorità maggiore).*
  - *Solo i processi del nucleo del SO o di un superutente possono avere priorità negativa.*
  - *Le shell di login cominciano con una priorità pari a 0.*
- *Se* **`nice()`** *ha successo, restituisce il nuovo valore della priorità; altrimenti restituisce -1 (possibile ambiguità con la corrispondente priorità!).*

Esempio: count.c (Uso del Disco)

Il programma **`count.c`** *conta il numero dei file che non sono directory all'interno di una gerarchia.*

- *l'argomento è la radice della gerarchia da analizzare*
- *se è una directory*
  - *crea un processo per ogni entry della directory*
  - *termina restituendo la somma degli exit status dei figli.*
- *altrimenti*
  - *termina con valore 1.*
- *Ogni figlio ripete lo stesso comportamento del padre.*

Svantaggi di questa tecnica: crea molti processi e può contare fino ad un massimo di 255 file per ogni directory (usa rappresentazioni ad 8 bit).

Esempio: Uso del Disco

```
/* count.c */
#include <stdio.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>

int processFile (char *);
int processDirectory (char *);

int main (int argc, char *argv[]) {
  int count;
  count = processFile(argv[1]);
  printf ("Total number of non-directory files is %d\n", count);
  return (/* EXIT_SUCCESS */ 0);
}
```

Esempio: Uso del Disco

```
int processFile (char *name) {
  struct stat statBuf; /* To hold the return data from stat () */
  mode_t mode;
  int result;
  result = stat (name, &statBuf); /* Stat the specified file */
  if (result == -1) return (0); /* Error */
  mode = statBuf.st_mode; /* Look at the file's mode */
  if (S_ISDIR (mode)) /* Directory */
      return (processDirectory(name));
  else
      return (1); /* A non-directory file was processed */
}
```

Esempio: Uso del Disco

```
int processDirectory (char *dirName) {
  DIR *dd;
  int fd, children, i, childPid, status, count, totalCount;
  char fileName [100];
  struct dirent *dirEntry;

  dd = opendir(dirName); /* Open directory */
  children = 0; /* Initialize child process count */
  while (!(dirEntry = readdir(dd))) { /* Scan directory */
     if (strcmp (dirEntry -> d_name, ".") != 0 &&
         strcmp (dirEntry -> d_name, "..") != 0) {
         if (fork () == 0) { /* Create a child to process dir entry */
             sprintf (fileName, "%s/%s", dirName, dirEntry -> d_name);
             count = processFile (fileName);
             exit (count);
         } else
           ++children; /* Increment count of child processes */
     }
  }
```

Esempio: Uso del Disco

```
/* continua ...*/

  closedir (dd); /* Close directory */
  totalCount = 0; /* Initialize file count */
  for (i = 0; i < children; i++) { /* Wait for children to terminate*/
     childPid = wait (&status); /* Accept child's termination code */
     totalCount += (status >> 8); /* Update file count */
  }
  return (totalCount); /* Return number of files in directory */
}
```

```
$ count /home/rossi/Sys
Total number of non-directory files is 213
$
```

Esempio: redirect.c (Redirezione)

`redirect file cmd arg1 ... argn`

esegue il comando **`cmd`** *con argomenti* **`arg1 ... argn`** *e redirige lo standard output del programma nel file* **`file`** *passato come parametro.*

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

int main (int argc, char *argv[]) {
  int fd;
  /* Open file for redirection */
  fd = open (argv[1], O_CREAT | O_TRUNC | O_WRONLY, 0600);
  dup2 (fd, 1); /* Duplicate descriptor to standard output */
  close (fd); /* Close original descriptor to save descriptor space */
  execvp (argv[2], &argv[2]); /* Invoke program; will inherit stdout */
  perror ("main"); /* Should never execute */
}
```

Esempio: redirect.c

```
$ redirect lista ls -l
$ cat lista
-rw-------   1 rossi   users          0 May 22 18:02 lista
-rwxr-xr-x   1 rossi   users       4314 May 22 18:02 redirect
-rw-r--r--   1 rossi   users        432 May 22 09:12 redirect.c
-rwxr-xr-x   1 rossi   users       6477 May 19 17:52 reverse
-rw-r--r--   1 rossi   users       5125 May 19 17:52 reverse.c
$
```