Lezione 5: File I/0

5.1 UNIX: Programmazione di Sistema

Per utilizzare i servizi offerti da unix, tipo la creazione dei file, duplicazione di processi e comunicazione tra processi i programmi applicativi devono interagire con il sistema operativo.

Per fare ciò devono usare un insieme di routine dette System Call, che costituiscono l’interfaccia funzionale del programmatore col nucleo di unix.

Le SC sono simili alle routine di libreria C ma eseguono una chiamata di subroutine direttamente nel nucleo di unix.

5.2 UNIX: SC

Le SC costituiscono un entry point per il kernel, il programmatore chiama la funzione utilizzando la sintassi usuale delle funzionio C

```
int open(const char *path, int mode)
```

La funzione invoca nel modo opportuno il servizio del sistema operativo.

Salva gli argomenti della SC ed un numero identificativo della SC stessa.

5.3 UNIX: Funzioni di libreria

Le funzioni di libreria forniscono servizi di utilità generale al programmatore.

Non sono entry poin del kernel, anche se possono fare uso di SC per realizzare il proprio servizio.

Ad esempio

printf → può utilizzare la SC write per stampare.

strcpy, atoi → non coinvolgono il sistema operativo.

Possono essere sostituite con altre funzioni che realizzano lo stesso compito.

5.4 Tipi di sistemi primitivi

Storicamente certi tipi di dati C sono stati associati con certe variabili di sistema unix.

Ad esempio i valori per i major e minor device number erano memorizzati in numeri interi short a 16 bit. (8 riservati per i major e gli altri 8 per i minor device)

Si possono creare seri problemi di portabilità quando ci si sposta da sistema a sistema o da un'architettura ad un altra

La tecnica adottata è definire tipi di dati dipendenti dall'implementazione chiamati tipi di dati di sistema primitivi <sys/types.h> sono definiti negli header mediante la typedef. La maggior parte termina in _t

Tutte le funzioni di libreria di solito non fanno riferimento ai tipi elementari dello standard del linguaggio C, ma ad una serie di tipi primitivi del sistema.

Si evita nei nostri programmi il riferimento a dettagli implementaivi che possono cambiare da sistema a sistema.

5.5 Gestione dei file

Le SC per la gestione dei file permettono di manipolare:

- file regolari
- directory
- file speciali

Tra i file speciali troviamo:

- link simbolici
- dispositivi (terminali, stampanti)
- meccanismi di IPC (pipe e socket)

5.6 Chiamate di sistema di I/O

Le SC descritte prima realizzano le operazioni di base per la gestione dei file:

- open()
- read()
- write()
- lseek()
- close()

Spesso a queste funzioni ci si riferisce come I/O non bufferizzato. Ogni read o write invoca una SC del kernel.

5.7 Sequenze tipiche di operazioni con file

```
int fd; // dicbiara un descrittore di file
fd = open(filename, ...) // apre un file

if (fd == -1){} // gestione di errore

read(fd, ...); // legge dal file
write(fd, ...); // scrive nel file
lseek(fd, ...); // si sposta all'iterno del file
close(fd); // chiusura del file liberando il descrittore
unlink(filename); // rimuove il file
```

5.8 Chiamate di sistema di I/O

I file aperti sono gestiti dal kernel mediante descrittori di file, un descrittore di file è un intero non negativo.

I descrittori di file pososno variare da 0 a OPEN_MAX.

Le prime versioni dei sistemi avevano un limite superiore di 19, consentendo un massimo di 20 file aperti per processo. Molti sistemi hanno incrementato tale limite a 63.

Alla richiesta di aprire un file esistente i du creare un nuovo file il kernel, ritorna un descrittore di file al processo chiamante.

Quando si vuole leggere o scrivere su un file si passa come argomento a read o write il descrittore ritornato da open.

Per convenzione:

Il descrittore 0 viene associato allo standard input.

Il descrittore 1 allo standard output.

Il descrittore 2 allo standard error.

Per conformità allo standard Posix, i numeri 0,1 e 2 possono essere sostituiti dalle costanti STDIN_FILENO, STDOUT_FILENO e STDERR_FILENO, definite nell'header <unistd.h>

5.9 Chiamata di sistema open

```
#include <fnctl.h>
int open(const char *path, int oflag, mode_t);
```

Funzione per aprire o creare file

path è il nome del file da creare o da aprire.

il terzo argomento viene utilizzato solo quando si crea un file.

Ritorna il descrittore del file, -1 in caso di errore.

oflag può assumere diversi valori:

- apri solo in lettura O_RDONLY
- apri solo in scrittura O_WRONLY
- apri in lettura e scrittura O_RDWR

CI SONO COSTANTI OPZIONALI:

- O_APPEND esegue un aggiunta alla fine del file per ciascuna scrittura
- O_CREAT crea il file se non esiste
- O_EXCL se utilizzata con O_CREAT ritorna un errore se il file esiste
- O_TRUNC se il file esiste ed é aperto con successo per sola scrittura o per letturura-scrittura lo tronca a lunghezza 0
- O_NOCTTY se path é un terminal device non lo rende il terminale di controllo del processo
- O_NONBLOCK se path è una FIFO un file a blocchi o a caratteri apre in maniera non bloccante sia in lettura che scrittura

mode definisce i bit di permesso di accesso ai file.

5.10 Chiamata di sistema creat

Nelle precedenti versioni di unix il secondo argomento della open poteva essere solo 0,1 o 2

Non c'era modo di aprire un file che non esisteva, fu introdotto così creat per la creazione di nuovi file.

Con le opzioni O_CREAT e O_TRUNC la open é in grado di aprire nuovi file e quindi non c'è bisogno di questa funzione.

Il problema di questa funzione è che un file si apre solo in scrittura.

5.11 Chiamata di sistema close

```
#include <unistd.h>
int close(int fildes);
```

chiude un file

ritorna -1 in caso di errore

quando un processo termina tutti i file aperti vengono automaticamente chiusi.

5.12 Chiamata di sistema lseek

Ad ogni file aperto è associato un valore intero non negativo, detto offset corrente del file, che misura il numero di byte dall'inizio del file.

```
#include <unistd.h>
off_t lseek (int fildes, off_t offset, int whence);
```

Ritorna -1 in caso di errore.

Quando il file viene aperto l'offset viene inizializzato a zero, a meno che non si specifichi l'opzione O_APPEND.

L’argomento whence può assumere i seguenti valori:

- SEEK_SET l'offset viene posto a offset byte dall'inizio del file
- SEEK_CUR viene aggiunto offset all'offset corrente
- SEEK_END l'offset viene posto alla fine del file, più offset

Poiché una chiamata a lseek andata a buon fine restituisce il nuovo offset del file, per determinare l’offset corrente:

Si cercano zero byte dalla posizione corrente

```
off_t currpos;

currpos = lseek(fd, 0, SEEK, cur);
```

Tale tecnica è usata anche per determinare se un file è in grado di essere “cercato”

Se il descrittore del file si riferisce a pipe, fifo o socket, lseek restituisce -1 e errno è
impostata al valore ESPIPE

5.12.1 Esempio

```
#include <sys/types.h>
#include “apue.h"

int main(void){
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

5.13 Chiamata di sistema lseek (cont)

L’offset di un file può essere più grande della dimensione corrente del file.

- La write successiva estende il file
- Crea un buco
- Qualsiasi byte nel file che non è stato scritto è letto come 0

- Non è richiesto che ai buchi sia allocato un blocco su disco

5.13.1 Esempio

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include “apue.h"

char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";

int main(void) {
    int fd;
    if ( (fd = creat("file.hole", FILE_MODE)) < 0) err_sys("crea
    if (write(fd, buf1, 10) != 10) err_sys("buf1 write error");
    /* offset ora = 10 */
    if (lseek(fd, 40, SEEK_SET) == -1) err_sys("lseek error");
    /* offset ora = 40 */
    if (write(fd, buf2, 10) != 10) err_sys("buf2 write error");
    /* offset ora = 50 */
    exit(0);
}
```

5.14 Chiamata di sistema read

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbytes);
```

Legge dal fildes, nbytes byte in buf a partire dalla posizione corrente

Aggiorna la posizione corrente

RItorna il numero di byte effettivamente letti 0 se alla fine del file, -1 in caso di errore.

Ci sono diversi casi in cui read legge un numero di byte minore di quanto richiesta.

Se è raggiunta la fine di un file prima che sia letta la quantità di byte richiesti.

Quando si legge da un terminale.

Quando si legge da una rete.

Quando si legge da un pipe o FIFO.

Quando si è interrotti da un segnale ed è stata letta una quantità parziale di dati.

5.15 Chiamata di sistema write

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbytes);
```

Scrive nel file fildes, nbytes byte da buf, a partire dalla posizione corrente.

Aggiorna la posizione corrente

Restituisce il numero di byte effettivamente scritti, o –1 in caso di errore

5.15.1 Esempio (copia un file)

```
#define BUFFS 4096

int main(void){
 int n;
 char buf[BUFFS];

 while ((n=read(STDIN_FILENO, buf, BUFFS)) > 0)
     if (write(STDOUT_FILENO, buf, n) != n)
            printf("write error");
     if (n < 0)
        printf("read error");
```

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

5.16 Condivisione di file

Unix supporta la condivisione del file aperti tra differenti processi

Il kernel usa tre strutture dati per rappresentare un file aperto.

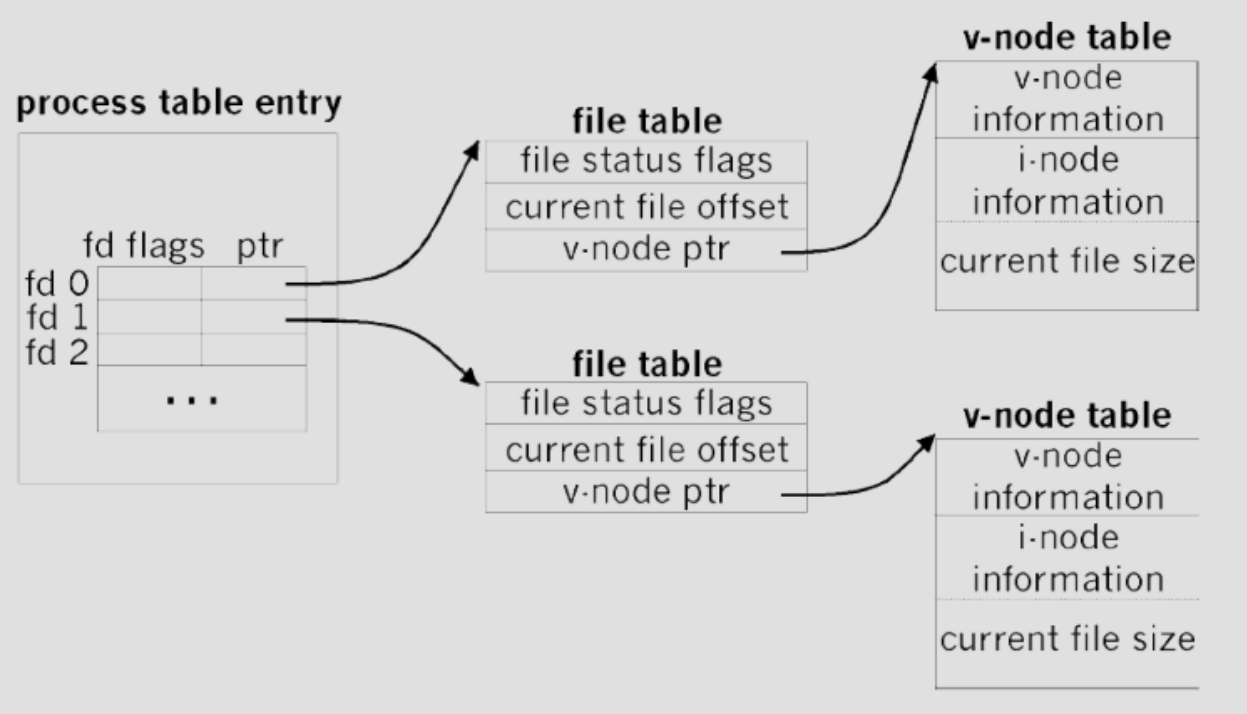
Ogni processo ha un'entrata nella tabella dei processi.

All'interno di ogni entrata della tabella dei processi c'è una tabella dei descrittori di file aperti. A ciascun descrittore sono associati: flag e puntatore ad un'entrata della tabella dei file.

Il kernel mantiene una tabella di file per tutti i file aperti. Ogni entrata contiene: flag, offset corrente del file e un puntatore alla entrata della tabella dei v-node per il file.

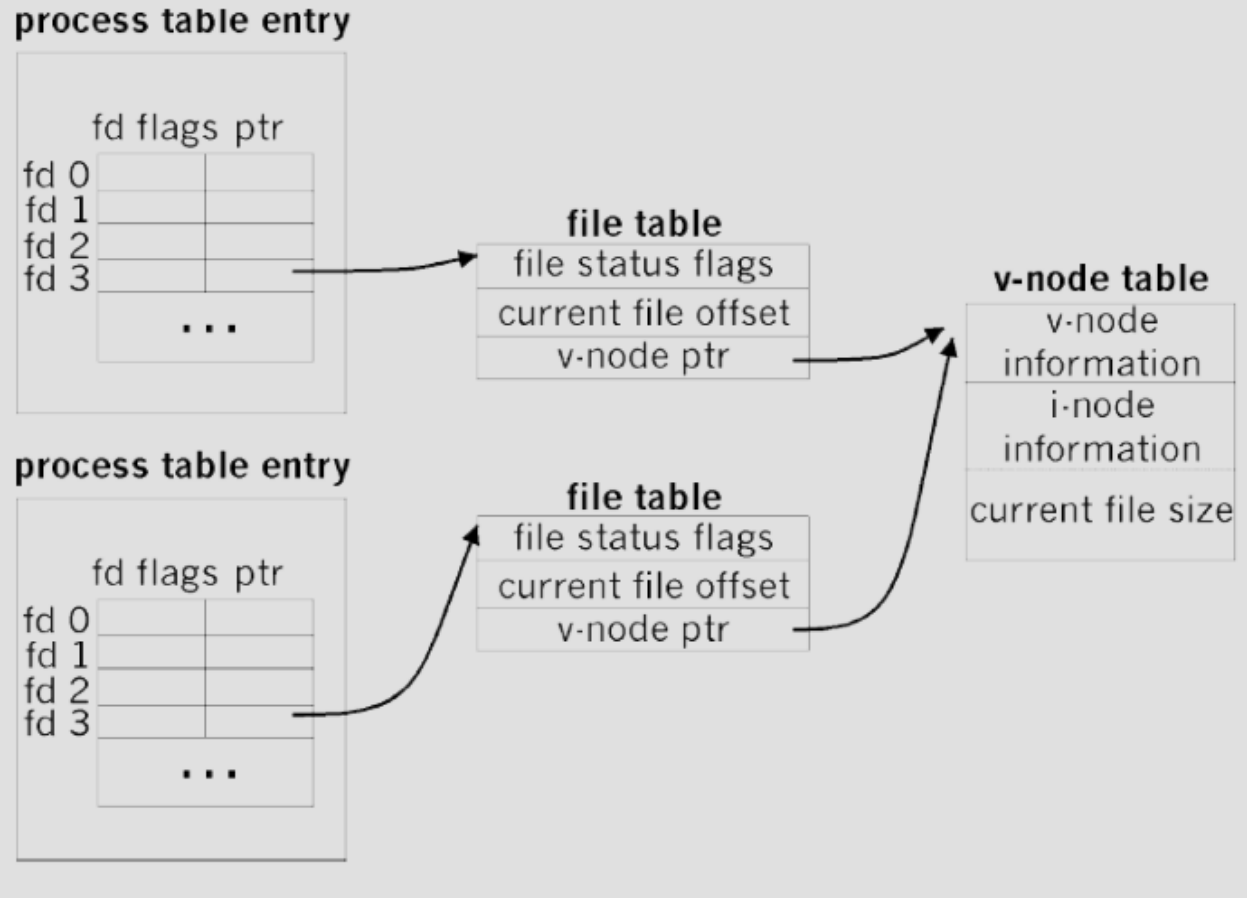
Ogni file aperto ha una struttura v-node che contiene informazioni sul tipo del file e puntatori a funzioni che operano sul file. Il v-node contiene anche l'i-node per il file. Queste sono informazioni lette dal disco quando il file é aperto.

Per esempio vediamo le tre tabelle per un singolo processo con due differenti file aperti. Un file aperto sullo standard input, un file aperto sullo standard output.



Con due processi indipendenti che hanno lo stesso file aperto il primo processo ha il file aperto sul descrittore di file 3, il secondo processo ha lo stesso file aperto sul descrittore 4. Ogni processo che apre il file ha la propria entrata nella tabella dei file, sebbene sia richiesta per un dato file solo una singola entrata della tabella dei v-node.

La ragione per cui ciascun processo ha la propria entrata della tabella dei file è che ogni processo ha il proprio offset per il file.



Cosa accade quando un processo cerca di accedere ad un file?

Quando un processo accede ad un file mediante una write l'elemento della tabella dei file relativo all'offset viene aggiornato e se necessario viene aggiornato l'inode.

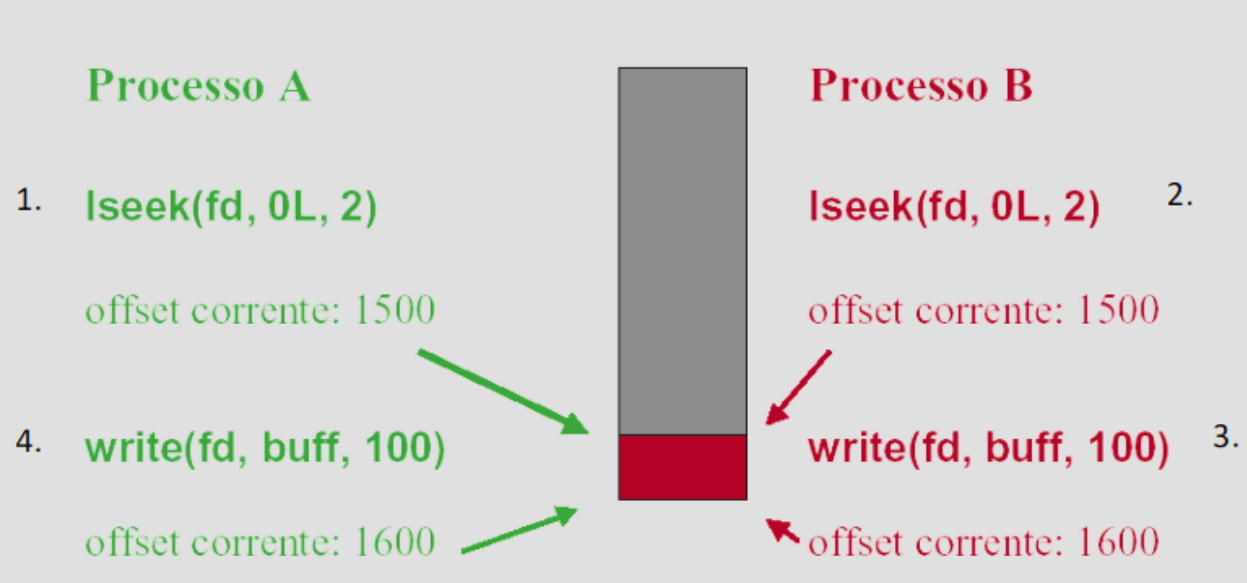
Se il file è aperto con O_APPEND é messo nella tabella dei file un flag corrispondente, una chiamata ad lseek modifica solo l'offset corrente del file e non viene eseguita nessuna operazione di I/O. Se si chiede di posizionarsi alla fine del file, il valore corrente dell'offset nella tabella dei file viene preso dal campo della tavola di i-node che descrive la dimensione del file.

5.17 Operazioni atomiche

Questo tipo di operazione non comporta alcun problema per unico processo, se più processi concorrenti impiegano questa tecnica per aggiungere dati alla fine di

un file possono verificarsi dei problemi, supponiamo di avere due processi A e B che aggiungono byte alla fine di uno stesso file.

Ognuno ha aperto il file senza l'operazione O_APPEND, supponiamo che l'attuale fine del file abbia offset pari a 1500.



Il processo A ha sovrascritto quello che ha scritto B.

L'operazione posizionati alla fine del file e scrivi richiede due azioni distinte.

Una qualsiasi operazione che richieda più di una chiamata a funzione può essere interrotta.

Il modo per eseguire questa operazione in maniera atomica è di utilizzare il flag O_APPEND quando si apre il file.

Il kernel posiziona l'offset alla fine del file corrente prima di ogni write.

5.18 Duplicare i descrittori di file

La SC dup() e dup2() permettono di duplicare un file descriptor.

Creano un nuovo file descriptor che punta alla stessa entry della tabella dei file aperti.

5.18.1 Le funzioni dup() e dup2()

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

dup() trova il più piccolo descrittore non utilizzato e lo fa riferire alla stessa file descriptor entry a cui fa riferimento filedes.

dup2() il secondo argomento filedes2 è il nuovo descrittore. Se filedes2 é attualmente attivo, lo chiude e quindi lo fa riferire allo stesso file a cui fa riferimento filedes.

Il descrittore di file originale e quello copiato condividono lo stesso puntatore interno al file e le stesse modalità di accesso.

Ritornano il nuovo descrittore se hanno successo e -1 in caso di errore.

5.18.2 Esempio duplicare i descrittori di file

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

int main (void) {
    int fd1, fd2, fd3;

    fd1 = open ("test.txt", O_CREAT | O_RDWR, 0600);
    printf ("fd1 = %d\n", fd1);
```

```
    write (fd1, “Cosa sta", 8);

    fd2 = dup (fd1); /* Effettua una copia di fd1 */
    printf ("fd2 = %d\n", fd2);

    write (fd2, " accadendo", 10);
    close (0); /* Chiude lo standard input */

    fd3 = dup (fd1); /* Effettua un’altra copia di fd1 */
    printf ("fd3 = %d\n", fd3);

    write (0, " al contenuto", 13);
    dup2 (3, 2); /* Duplica il canale 3 sul canale 2 */
    write (2, “?\n", 2);
    return 0;
}
```

5.19 Gestione degli errori: perror()

Una SC ritorna -1 se fallisce, per gestire gli errori originati dalle SC i due principali ingredienti da utilizzare sono:

- errno → variabile globale che contiene il codice numerico dell’ultimo errore generato da una SC
- perror() → subroutine che mostra una descrizione testuale dell’ultimo errore generato dall’invocazione di una SC

Variabile globale errno

Inizializzata a 0

Se si verifica un errore dovuto ad una SC ad errno è assegnato un codice numerico corrispondente.

errno.h → contiene codici di errore predefiniti:

```
#define EPERM 1 /* Operation not permitted */
#define ENOENT 2 /* No such file or directory */
#define ESRCH 3 /* No such process */
#define EINTR 4 /* Interrupted system call */
#define EIO 5 /* I/O error */
```

```
void perror (char *str)
```

mostra la stringa str seguita da : e da una stringa che descrive il valore corrente di errno.

SE non ci sono errori da riportare viene mostrata la stringa Error 0.

Non è una SC ma una routine di libreria.

Per accedere alla variabile errno ed invocare perror() occorre includere il file errno.h

I progrmmi dovrebbero controllare se il valore ritornato da una Sc é-1 e in questo caso invocare perror() per una descrizione dell'errore.

5.19.1 Esempio: showErrno.c

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

int main(void) {
    int fd;

    /* Apre un file inesistente per causare un errore */
    fd = open ("nonexist.txt", O_RDONLY);
```

```
    if (fd == -1) { /* fd=-1 => si è verificato un errore */
        printf ("errno = %d\n", errno);
        perror ("main");
    }

    fd = open ("/", O_WRONLY); /* Forza un errore diverso */

    if (fd == -1) {
        printf ("errno = %d\n", errno);
        perror ("main");
    }

    /* Esegue una system call con successo */
    fd = open ("nonexist.txt", O_WRONLY | O_CREAT, 0644);
    printf("errno = %d\n", errno);

    /* Visualizza dopo la chiamata */
    perror ("main");
    errno = 0; /* Reset manuale variabile di errore */
    perror ("main");
    return 0;
}
```