Scrivere alla fine di un file

Vi sono due modi per scrivere alla fine di un file:

- usare `lseek` per spostarsi alla fine del file e poi scrivere:
```
lseek(filedes, (off_t)0, SEEK_END);
write(filedes, buf, BUFSIZE);
```

- usare `open` con il flag `O_APPEND`:
```
filedes = open("nomefile", O_WRONLY | O_APPEND);
write(filedes, buf, BUFSIZE);
```

Altri flag utili nell'utilizzo di `open` sono i seguenti:

- `O_RDONLY`: apre il file specificato in sola lettura.
- `O_RDWR`: apre il file specificato in lettura e scrittura.
- `O_CREAT`: crea un file con il nome specificato; con questo flag è possibile specificare come terzo argomento della `open` un numero **ottale** che rappresenta i permessi da associare al nuovo file (e.g., `0644`).
- `O_TRUNC`: tronca il file a zero.
- `O_EXCL`: flag "esclusivo"; un tipico esempio d'uso è il seguente:
```
filedes = open("nomefile", O_WRONLY | O_CREAT | O_EXCL, 0644);
```
che provoca un fallimento nel caso in cui il file `nomefile` esista già.

Eliminare un file

Per cancellare un file vi sono due system call a disposizione del programmatore:

```
#include <unistd.h>
int unlink(const char *pathname);

#include <stdio.h>
int remove(const char *pathname);
```

Entrambe le system call hanno un unico argomento: il pathname del file da eliminare.

Esempio:

```
remove("/tmp/tmpfile");
```

Differenze:

- `unlink` in certi sistemi non può rimuovere le directory, in altri, per eseguire questa operazione, può essere necessario avere i privilegi di `root`;
- `remove` richiama `unlink` per i file e `rmdir` per le directory; funziona anche con i link simbolici.

La chiamata di sistema `fcntl`

La system call `fcntl` permette di esercitare un certo grado di controllo su file già aperti:

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

int fcntl(int filedes, int cmd, ...);
```

I parametri dal terzo in poi variano a seconda del valore dell'argomento `cmd`. L'utilizzo più comune di `fcntl` si ha quando `cmd` assume i seguenti valori:

- `F_GETFL`: fa in modo che `fcntl` restituisca il valore corrente dei flag di stato (come specificati nella `open`).

- `F_SETFL`: imposta i flag di stato in accordo al valore del terzo parametro. Esempio:

```
if(fcntl(filedes, F_SETFL, O_APPEND) == -1)
  printf("fcntl error\n");
```

Esempio d'uso di `fcntl`

```
#include <fcntl.h>

int filestatus(int filedes) {
  int arg1;

  if((arg1 = fcntl(filedes, F_GETFL)) == -1) {
    printf("filestatus failed\n");
    return -1;
  }

  printf("File descriptor %d\n", filedes);

  switch(arg1 & O_ACCMODE) {
    case O_WRONLY:
      printf("write only");
      break;
```

Esempio d'uso di `fcntl` (...continua)

```
    case O_RDWR:
      printf("read write");
      break;
    case O_RDONLY:
      printf("read only");
      break;
    default:
      printf("No such mode");
      break;
  }

  if(arg1 & O_APPEND)
    printf(" - append flag set");

  printf("\n");
  return 0;
}
```

dove `O_ACCMODE` è una maschera appositamente definita in `<fcntl.h>`.

`stat` e `fstat`

Le informazioni e le proprietà dei file (dispositivo del file, numero di inode, tipo del file, numero di link, UID, GID, dimensione in byte, data ultimo accesso/ultima modifica, informazioni sui blocchi che contengono il file) sono contenute negli inode. Le chiamate di sistema `stat` e `fstat` permettono di accedere in lettura alle informazioni e proprietà associate ad un file:

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

int stat(const char *pathname, struct stat *buf);

int fstat(int filedes, struct stat *buf);
```

L'unica differenza fra le due system call consiste nel fatto che, mentre `stat` prende come primo argomento un pathname, `fstat` opera su un descrittore di file. Quindi `fstat` può essere utilizzata soltanto su file già aperti tramite la `open`.

La struttura `stat`

`stat` è una struttura definita in `<sys/stat.h>` che comprende i seguenti componenti (i tipi sono definiti in `<sys/types.h>`):

Tipo	Nome	Descrizione
`dev_t`	`st_dev`	logical device
`ino_t`	`st_ino`	inode number
`mode_t`	`st_mode`	tipo di file e permessi
`nlink_t`	`st_nlink`	numero di link non simbolici
`uid_t`	`st_uid`	UID
`gid_t`	`st_gid`	GID
`dev_t`	`st_rdev`	membro usato quando il file rappresenta un device
`off_t`	`st_size`	dimensione **logica** del file in byte
`time_t`	`st_atime`	tempo dell'ultimo accesso
`time_t`	`st_mtime`	tempo dell'ultima modifica
`time_t`	`st_ctime`	tempo dell'ultima modifica alle informazioni della struttura `stat`
`long`	`st_blksize`	dimensione del blocco per il file
`long`	`st_blocks`	numero di blocchi allocati per il file

Esempio (I)

Il programma `lookout.c`, data una lista di nomi di file, controlla ogni minuto se un file è stato modificato. Nel caso ciò avvenga termina l'esecuzione stampando un messaggio che informa l'utente dell'evento.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>

#define MFILE 10

void cmp(const char *, time_t);
struct stat sb;

main(int argc, char **argv) {
  int j;
  time_t last_time[MFILE+1];
```

Esempio (II)

```
if(argc<2) {
  fprintf(stderr, "uso: lookout file1 file2 ...\n");
  exit(1);
}

if(--argc>MFILE) {
  fprintf(stderr, "lookout: troppi file\n");
  exit(1);
}

for(j=1; j<=argc; j++) {
  if(stat(argv[j], &sb) == -1) {
    fprintf(stderr, "lookout: errore nell'accesso al file %s\n", argv[j]);
    exit(1);
  }

  last_time[j] = sb.st_mtime;
}
```

Esempio (III)

```
  for(;;) {
    for(j=1; j<=argc; j++)
      cmp(argv[j], last_time[j]);

    sleep(60);
  }

}

void cmp(const char *name, time_t last) {

  if(stat(name, &sb) == -1 || sb.st_mtime != last) {
    fprintf(stderr, "lookout: il file %s e‘ stato modificato\n", name);
    exit(0);
  }

}
```

Directory

Le **directory** unix sono **file**.

Molte system call per i file ordinari possono essere utilizzate per le directory. E.g. `open`, `read`, `fstat`, `close`.

Tuttavia le directory non possono essere create con `open`, `creat`.

Esiste un insieme di system call speciali per le directory.

Una directory è rappresentata in memoria da una **tabella**, con una entry per ogni file nella directory.

inode	name
⋮	⋮

Ogni entry è una struttura di tipo `dirent` definita in `<dirent.h>`:

```
ino_t d_ino;
char d_name[ ];
```

Il primo campo contiene l'inode del file, il secondo il nome del file. Se `d_ino`=0, allora lo slot è libero.

Creazione e apertura di una directory

Creazione di una directory:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir (const char *pathname, mode_t mode)
```

La system call `mkdir` restituisce 0 o -1 a seconda che termini con successo o meno. Al momento della creazione con `mkdir`, i link . e .. vengono inseriti nella tabella.

Apertura di una directory:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (const char *dirname);
```

La system call `opendir` ritorna un puntatore di tipo `DIR` oppure il null pointer, in caso di fallimento.

Lettura, riposizionamento, chiusura di una directory

Lettura di una directory:

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir (DIR *dirptr);
```

La system call `readdir` restituisce un puntatore alla struttura `dirent` dove è stata copiata la prossima entry nella tabella.

Riposizionamento:

```
void rewinddir (DIR *dirptr);
```

Chiusura:

```
#include <dirent.h>
int closedir (DIR *dirptr);
```

Esempio: lettura di una directory

```
#include <dirent.h>

int my_ls (const char *name)
{
  struct dirent *d;
  DIR *dp;

  /* apertura della directory */
  if ((dp=opendir(name)) == NULL)
     return (-1);

  /* stampa dei nomi dei file contenuti nella directory */
  while (d = readdir(dp))
  {
     if (d->d_ino != 0)
        printf("%s\n", d->d_name);
  }

  closedir(dp);
  return(0);
}
```

Tipo di un file

L'attributo `st_mode` di una struttura ottenuta con la system call `stat` contiene il **file mode**, cioè una sequenza di bit ottenuti facendo l'OR bit a bit della stringa che esprime i permessi al file e una costante che determina il tipo del file (regolare, directory, speciale, etc.).

Per scoprire se un file è una directory, si può usare la macro `S_ISDIR`:

```
/* buf e' il puntatore restituito da stat */
if (S_ISDIR (buf.st_mode))
    printf("It is a directory\n");
  else
    printf("It is not a directory\n");
```

Altre macro utili: `S_ISREG`, `S_ISCHR`, `S_ISBLK`, `S_ISLNK`.

Cambiamento della directory corrente

La directory corrente di un processo è quella in cui il processo è eseguito. Tuttavia un processo può cambiare la sua directory corrente con la system call

```
#include <unistd.h>
int chdir (const char *path);
```

dove `path` è il pathname della nuova directory corrente. Il cambiamento si applica solo al processo chiamante.

Attraversamento dell'albero di una directory

La system call `ftw` consente di eseguire un'operazione `fn` su tutti i file nella gerarchia della directory `dirpath`:

```
#include <ftw.h>
int ftw (const char *dirpath, int (*fn)(), int nopenfd);
```

Il parametro `nopenfd` controlla il numero di file descriptor usati da `ftw`. Più grande è il valore di `nopenfd`, meno directory devono essere riaperte, incrementando la velocità di esecuzione.

`fn` è una funzione definita dall'utente, che viene passata alla routine `ftw` come puntatore a funzione. Ad ogni chiamata, `fn` viene chiamata con tre argomenti: una stringa contenente il nome del file a cui `fn` si applica, un puntatore ad una struttura `stat` con i dati del file, un codice intero. Il prototipo di `fn` deve perciò essere:

```
int fn (const char *name, const struct stat *sptr, int type);
```

L'argomento `type` contiene uno dei seguenti valori (definiti in `<ftw.h>`), che descrivono il file oggetto:

`FTW_F`	l'oggetto è un file
`FTW_D`	l'oggetto è una directory
`FTW_DNR`	l'oggetto è una directory che non può essere letta
`FTW_SL`	l'oggetto è un link simbolico
`FTW_NS`	l'oggetto non è un link simbolico e su di esso `stat` fallisce

Esempio (I)

Il programma `rls` (recursive ls) prende come argomento sulla linea di comando una directory e visualizza su standard output tutti i file regolari e le directory che incontra attraversando il file system a partire da quest'ultima (evidenziando se si tratta di file ordinari o di directory).

```
#include <ftw.h>
#include <stdio.h>
#include <string.h>

int src(const char *name,const struct stat *sptr,int type);

main(int argc,char **argv) {

    if(argc!=2) {
        fprintf(stderr,"Utilizzo: rls <dir>\n");
        exit(1);
    }
```

Esempio (II)

```
    if(ftw(argv[1],src,5)==-1) {
        perror("Errore nell'esecuzione di ftw");
        exit(2);
    }

}

int src(const char *name,const struct stat *sptr,int type) {

    if(type==FTW_F || type==FTW_D) {
        printf("%s ",name);

        if(type==FTW_F)
            printf("(file ordinario)\n");
        else
            printf("(directory)\n");

    }

    return 0;
}
```

Esercizi

- Si scriva un programma che conti le modifiche ad un file (specificato come primo argomento sulla riga di comando) nell'arco di un intervallo di tempo (specificato in secondi come secondo argomento sulla linea di comando). Alla fine il programma deve produrre sullo schermo del terminale un istogramma che mostri il numero di modifiche (si utilizzi ad esempio il carattere `*`).

- Si scriva un programma C che realizza una versione semplificata del comando unix `find`. Il programma dovrà ricevere sulla linea di comando il nome di una directory *dir* ed una stringa *str* e dovrà visitare l'intero albero di directory e file che ha come radice *dir*, stampando su std output tutti i file i cui nomi hanno come suffisso la stringa *str*, segnalando se si tratta di directory o file ordinari.