

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");
```

Inizialmente esisteva soltanto `unlink`, mentre `remove` è stata aggiunta in seguito come specifica dello standard ANSI C per l'eliminazione dei file regolari.

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", filedes);

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

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

```
case O_RDWR:
    print("read write");
    break;
case O_RDONLY:
    print("read only");
    break;
default:
    print("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_time != last) {
        fprintf(stderr, "lookout: il file %s e' stato modificato\n");
        exit(0);
    }

}
```

Memory mapped I/O

- Quando è necessario eseguire molte operazioni di I/O su disco, si genera un forte carico di lavoro sul sistema in quanto i dati su disco vengono copiati prima nei buffer interni del kernel e poi nelle strutture dati che sfruttano lo spazio di memoria dedicato al processo che ha eseguito la richiesta.
- Per evitare questo **overhead** si può mappare i file direttamente nello spazio di memoria riservato al processo.
- La memoria di un processo può essere manipolata mediante le seguenti funzioni:

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

```
void * memset(void *buf, int character, size_t size);  
void * memcpy(void *buf1, const void *buf2, size_t size);  
void * memmove(void *buf1, const void *buf2, size_t size);  
void * memcmp(const void *buf1, const void *buf2, size_t size);  
void * memchr(const void *buf, int character, size_t size);
```

La chiamata di sistema `mmap`

Il prototipo della system call `mmap` è definito come segue:

```
#include <sys/mman.h>
```

```
void * mmap(void *address, size_t length, int protection,  
            int flags, int filedes, off_t offset);
```

dove:

- `address` è l'indirizzo del punto in cui inizierà il mapping (se si passa 0, è il sistema a decidere e `mmap` restituisce l'indirizzo risultante);
- `length` è il numero di byte da mappare;
- `protection` determina se i dati mappati possono essere letti (`PROT_READ`), scritti (`PROT_WRITE`), eseguiti (`PROT_EXEC`) o se non si può accedere ad essi (`PROT_NONE`);
- `flags` determina se i cambiamenti ai dati mappati siano visibili agli altri processi ed eventuali modifiche scritte su disco (`MAP_SHARED`) oppure no (`MAP_PRIVATE`);
- `filedes` è il descrittore del file da mappare;
- `offset` rappresenta il punto nel file da cui iniziare il mapping.

La chiamata di sistema `munmap`

Il mapping viene eliminato automaticamente ed eventuali modifiche salvate su disco (nel caso in cui si sia specificato come protezione `MAP_SHARED`), quando il processo termina. Per eliminare manualmente il mapping si può usare la system call `munmap`:

```
#include <sys/mman.h>
```

```
void * munmap(void *address, size_t length);
```

Si noti che `munmap` non chiude il file mappato (per ottenere ciò bisogna eseguire una `close`).

Esempio: memory mapped file copy (I)

Il programma `copyfile.c` copia il file passato come primo argomento nel file passato come secondo argomento:

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>

main(int argc, char **argv) {
    int input, output;
    size_t filesize;
    void *source, *target;
    char endchar='\0';

    if(argc != 3) {
        fprintf(stderr, "utilizzo: copyfile file1 file2\n");
        exit(1);
    }
```

Esempio: memory mapped file copy (II)

```
if((input = open(argv[1], O_RDONLY)) == -1) {
    fprintf(stderr, "errore: impossibile aprire il file %s\n", argv[1]);
    exit(1);
}

if((input = open(argv[2], O_RDWR | O_CREAT | O_TRUNC, 0666)) == -1) {
    close(input);
    fprintf(stderr, "errore: impossibile aprire il file %s\n", argv[2]);
    exit(2);
}

filesize = lseek(input, 0, SEEK_END);
lseek(output, filesize-1, SEEK_SET);
write(output, &endchar, 1);

if((source = mmap(0, filesize, PROT_READ, MAP_SHARED, input, 0)) ==
    (void *)-1) {
    fprintf(stderr, "Errore durante il mapping del primo file\n");
    exit(1);
}
```

Esempio: memory mapped file copy (III)

```
if((target = mmap(0, filesize, PROT_WRITE, MAP_SHARED, output, 0)) ==  
    (void *)-1) {  
    fprintf(stderr, "Errore durante il mapping del secondo file\n");  
    exit(2);  
}
```

```
memcpy(target, source, filesize);  
munmap(source, filesize);  
munmap(target, filesize);  
close(input);  
close(output);  
exit(0);  
}
```


Esercizi

- Scrivere un programma che riporta le modifiche di dimensione di 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 delle modifiche (si utilizzi ad esempio il carattere *).
- Scrivere un programma che copia il primo file passato come argomento sulla linea di comando nel file passato come secondo argomento, utilizzando le chiamate di sistema `open`, `read` e `write`. Si confrontino le prestazioni di questo programma con quelle di `copyfile.c` che sfrutta il memory mapped I/O.