Corso di Sistemi Operativi
A.A. 2008-2009
-
CHIAMATE DI SISTEMA PER FILE E SEMAFORI

Fabio Buttussi

System call per l'accesso a file

Nome	Significato
`open`	apre un file in lettura e/o scrittura o crea un nuovo file
`close`	chiude un file precedentemente aperto
`read`	legge da un file
`write`	scrive su un file
`lseek`	sposta il puntatore di lettura/scrittura ad un byte specificato
`creat`	crea un file nuovo
`remove`	rimuove un file
`unlink`	rimuove un file
`stat`	controlla gli attributi associati ad un file

La system call `open`

La system call `open` permette l'**apertura** di un file: recupera l'i-node, verifica i permessi, inserisce il file nella tabella dei file aperti e restituisce l'indice (`int`) del descrittore del file nella tabella. Se fallisce, ritorna `-1`.

```
int open(const char *pathname, int flags);
```

Il parametro `pathname` è un nome di file (relativo o assoluto), mentre il parametro `flags` consiste, invece, in una delle seguenti costanti o in una loro combinazione (con l'operatore |):

- `O_RDONLY`: apre il file specificato in sola lettura.
- `O_WRONLY`: apre il file specificato in sola scrittura.
- `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_APPEND`: scrive in coda al file.
- `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.

Le system call `close`, `read`, `write`

```
int close (int fd)
```

Libera le aree di occupate nelle varie tabelle e provoca la scrittura su file di eventuali buffer non pieni.

```
int lung = read(int fd, void *buffer, int n)
```

Legge al pi‘u n byte a partire da quello corrente, li scrive in un buffer in memoria, aggiorna la posizione corrente nel file e ritorna il numero di byte letti.

```
int lung = write(int fd, void *buffer, int n)
```

Legge al pi‘u n byte da un buffer in memoria, li scrive a partire dalla posizione corrente nel file, aggiorna la posizione corrente nel file e ritorna il numero di byte scritti.

La system call `lseek`

La system call `lseek` permette l'**accesso random** ad un file, cambiando il numero del prossimo byte da leggere/scrivere.

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int start_flag);
```

Il parametro `filedes` è un descrittore di file.
Il parametro `offset` determina la nuova posizione del puntatore in lettura/scrittura.
Il parametro `start_flag` specifica da dove deve essere calcolato l'`offset`. `startflag` può assumere uno dei seguenti valori simbolici:

`SEEK_SET` (0)	:	`offset` è misurato dall'inizio del file
`SEEK_CUR` (1)	:	`offset` è misurato dalla posizione corrente del puntatore
`SEEK_END` (2)	:	`offset` è misurato dalla fine del file

`lseek` ritorna la nuova posizione del puntatore.

Offset validi e non validi

Il parametro `offset` può essere **negativo**, cioè sono ammessi **spostamenti all'indietro** a partire dalla posizione indicata da `start_flag`, purchè però non si vada oltre l'inizio del file.
Tentativi di spostamento prima dell'inizio del file generano un errore.

È possibile spostarsi **oltre la fine del file**.
Ovviamente non ci saranno dati da leggere in tale posizione.
Futuri accessi tramite la `read` ai byte compresi tra la vecchia fine del file e la nuova posizione danno come risultato il carattere ASCII null.

Esempio:

```
off_t newpos;
⋮
newpos = lseek(fd, (off_t)-16, SEEK_END);
```

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);
```

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.

Le system call `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

Directory

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

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

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

Esiste un insieme di system call speciali per le directory (es., `opendir`, `mkdir`).

Interprocess communication: Pipe

- Affinché due processi possano cooperare, è spesso necessario che **comunichino** fra loro dei dati.

- Una prima possibile soluzione a questo problema consiste nell'utilizzo condiviso dei file (e.g., leggendo e scrivendo in un file comune). Tuttavia tale approccio risulta inefficiente; inoltre vi è la possibilità che si verifichino dei problemi di contesa della risorsa condivisa.

- UNIX, per risolvere il problema, mette a disposizione una primitiva, detta **pipe**, che consiste in un canale **unidirezionale** di comunicazione che collega un processo ad un altro, generalizzando il concetto di file.

- È possibile inviare dati in una pipe attraverso la system call `write` e leggere dalla pipe attraverso la system call `read`.

La system call `pipe`

Per creare una pipe, esiste l'apposita system call:

```
#include <unistd.h>

int pipe(int filedes[2]);
```

dove `filedes` è un array di due interi che conterrà i descrittori di file identificanti la pipe. Il primo (`filedes[0]`) serve a leggere dalla pipe, mentre il secondo (`filedes[1]`) serve a scrivervi.

La politica utilizzata da una pipe per gestire i messaggi è di tipo FIFO e non può essere cambiata.

L'utilità di una pipe diventa evidente quando è utilizzata in congiunzione alla chiamata di sistema `fork`, in quanto i descrittori di file rimangono aperti dopo la `fork` stessa.

Esempio (I)

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

#define MSGSIZE 16

char *msg1="hello, world #1";
char *msg2="hello, world #2";
char *msg3="hello, world #3";

main() {
  char inbuf[MSGSIZE];
  int p[2], j;
  pid_t pid;

  if(pipe(p)==-1) {
    perror("pipe call");
    exit(1);
  }
```

Esempio (II)

```
  switch(pid=fork()) {
    case -1:
      perror("fork call");
      exit(2);
    case 0:                      /* processo figlio */
      close(p[0]);               /* chiusura del descrittore di lettura */
      write(p[1], msg1, MSGSIZE);
      write(p[1], msg2, MSGSIZE);
      write(p[1], msg3, MSGSIZE);
      break;
    default:                     /* processo padre */
      close(p[1]);               /* chiusura del descrittore di scrittura */
      for(j=0; j<3; j++) {
        read(p[0], inbuf, MSGSIZE);
        printf("%s\n", inbuf);
      }
      wait(NULL);
  }
  exit(0);
}
```

Segnali

- I segnali in Unix sono un meccanismo semplice per inviare degli interrupt software ai processi.
- Solitamente sono utilizzati per gestire errori e condizioni anomale, piuttosto che per trasmettere dati.
- Relativamente ai segnali, un processo può compiere tre tipi di azioni:
  1. eseguire una opportuna funzione per trattare l'errore (signal handling);
  2. bloccare il segnale;
  3. inviare il segnale ad un altro processo.
- I segnali sono definiti da delle costanti simboliche dichiarate nel file header `<signal.h>`.
- Alcuni segnali causano una terminazione normale del processo che li riceve, mentre altri provocano la terminazione anormale del processo a cui sono indirizzati.
- In `<sys/wait.h>` sono definite delle macro per determinare la causa della terminazione di un processo figlio.

Semafori

Il concetto di semaforo è stato introdotto da E.W. Dijkstra per risolvere il problema della sincronizzazione fra processi. Un semaforo è una variabile intera sem su cui sono ammesse le seguenti operazioni:

1. wait(sem) (o p(sem)):

```
if(sem!=0)
  sem=sem-1;
else
  /* attendi finche' sem!=0 e decrementa sem di un'unita` */
```

2. signal(sem) (o v(sem)):

```
sem=sem+1;
/* fai ripartire il primo fra i processi in coda d'attesa */
```

Le due operazioni precedenti sono **atomiche**, i.e., soltanto un processo può alterare il valore di un semaforo ad un dato istante.
La creazione e la gestione dei semafori avviene mediante speciche system call.

Informazioni sui semafori

Ogni semaforo ottenuto dalla system call `semget` si ritrova associate le seguenti informazioni:

- `semval`: il valore del semaforo (intero positivo): non può essere manipolato direttamente, ma soltanto attraverso le system call apposite;
- `sempid`: il PID del processo che ha operato per ultimo sul semaforo;
- `semcnt`: numero di processi in attesa che il semaforo assuma un valore maggiore di quello attuale;
- `semzcnt`: numero di processi in attesa che il semaforo assuma il valore `0`.