Laboratorio Sistemi Operativi

Giovanni Tosini

Indice

Capitolo 1

Processi e programmi

Un processo è un'istanza di un programma eseguito. Come viene creato, il **Kernel** gli associa una certa struttura di memoria.

Program code: segmento in sola lettura contenente istruzioni in linguaggio macchina;

Initialized data: segmento contenente variabili globali e statiche;

Uninitialized data: segmente contenente variabili globali e statiche **non** inizializzate;

Heap: segmento contenente variabili allocate dinamicamente;

Stack: segmento contenente gli argomenti e le variabili interne delle funzioni.

Una delle strutture dati di supporto è il **file descriptor table**, conterrà tutti i file che il processo aprirà. Ogni processo contiene già 3 **file descriptor** associati ad esso:

1. Standard input
2. Standard output
3. Standard error

Ogni successivo file aperto verrà identificato con il valore minore disponibile. Il file descriptor table è visibile **solo** a runtime.

Capitolo 2

System call

Sono un punto di ingresso verso il Kernel, vengono utilizzate per richiedere dei servizi. Dallo User Level verranno fatte delle chiamate alla System Call Interface che a sua volta comunicherà al Kernel.

2.1 Gestione degli errori delle System Call

Nella sezione ERRORS del comando `man` si possono trovare tutti i possibili valori di ritorno di errore di una System Call. Tuttavia è possibile usare la variabile `errno` accedibile tramite l'uso della libreria `<errno.h>`. Ci permetterà di sapere l'errore effettivo causato in base al valore salvato al suo interno.

Esempio:

```
1         #include <errno.h>
2         ...
3         //system call to open a file
4         fd = open(pathname, flags, mode);
5         //Begin code handling errors
6         if(fd == -1){
7             if(errno == EACCES){
8                 //Handling not allowed access to the file
9             }
10            else{
11                //Some other error occured
12            }
13        }
14        //End code handling errors
15        ...
16
```

La maggior parte delle system call ritorna un -1 o NULL Pointer in caso di errore, alcune però usano il -1 come valore di ritorno anche in caso di non errore. Qui l'uso di `errno` acquista ulteriore valore. Esempio:

```
        #include <sys/resource.h>
        ...
        //Reset the errno variable to 0
        errno = 0;
        //System call getpriority gets the nice value of a
   process
        nice = getpriority(which, who);
        if((nice == -1) && (errno != 0)){
            //Handling getpriority errors
        }
        ...

```

Esistono altre funzioni che aiutano a gestire gli errori, come la funzione `perror()` che stampa su standard error la stringa che le viene fornita. Esempio:

```
        #include <stdio.h>
        ...
        //System call to open a file
        fd = open(pathname, flags, mode);
        if(fd == -1){
            perror("<Open>");
            //System call to kill the current process
            exit(EXIT_FAILURE);
        }
        ...

```

L'output sarà:

```
        <Open>: No such file or directory

```

La libreria `string.h` fornisce la funzione `strerror()` che prende in input il valore di `errno` e stampa l'errore effettivo. Esempio:

```
        #include <stdio.h>
        ...
        //System call to open a file
        fd = open(path, flags, mode);
        if(fd == -1){
            printf("Error opening (%s): \n\t%s\n", path,
   strerror(errno));
            //System call to kill the current process
            exit(EXIT_FAILURE);
        }
```

```
10          ...
11
```

L'output sarà il seguente:

```
1           Error opening (myFile.txt):
2               No such file or directory
3
```

Capitolo 3

Kernel data types

Sono delle `typedef` di tipi normali C, necessari per ovviare problemi di portabilità, per esempio il `pid_t` usando per identificare il process ID di un processo non è altro che un tipo definito come `typedef int pid_t`, quindi un intero.

Capitolo 4

Filesystem

4.1 File

4.1.1 open

Apre un file esistente e nel caso in cui non esistesse lo può creare tramite l'uso di specifiche flag, in caso di successo ritorna un file descriptor, quindi va aggiunta una riga alla file descriptor table. In caso di errore ritorna un -1.

```
1       #include <sys/stat.h>
2       #include <stdio.h>
3
4       //Returns file descriptor on successo, or -1 on error
5       int open(const char *pathname, int flags, .../*mode_t
    mode*/);
6
```

- il `pathname` può essere il nome del file o il suo eventuale path;
- la flag può essere un bit mask di una o più flag che definiscono l'accesso al file, possono essere ORate fra di loro tramite "|";
- le mode possono si comportano in maniera simile alle flag, definiscono i permessi che il file avrà.

Tabella con le flag disponibili:

Flag	Description
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_TRUNC	Truncate existing file to zero length
O_APPEND	Writes are always appended to end of file
O_CREAT	Create file if it doesn't already exist
O_EXCL	With O_CREAT, ensure that this call creates the file.

Tabella delle mode disponibili:

Flag	Description
S_IRWXU	user has read, write, and execute permission
S_IRUSR	user has read permission
S_IWUSR	user has write permission
S_IXUSR	user has execute permission
S_IRWXG	group has read, write, and execute permission
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	others has read, write, and execute permission
S_IROTH	others has read permission
S_IWOTH	others has write permission
S_IXOTH	others has execute permission

Se non vengono forniti i permessi cosa succederà al file? All'interno del SO esiste la `umask` con dei valori che di default non dà permessi allo user e solo scrittura a group e others, tale valore sarà 022. Di `umask` ne esiste una sola, andando a fornire dei permessi tramite la `open` i permessi che il file avrà saranno la `mode` con il negato della `umask` (mode and ∼umask. Vari esempi di utilizzo:

```
1       int fd;
2       //Open existing file for only writing
3       fd = open("myfile", O_WRONLY);
4
5       //Open new or existing file for reading/writing,
   truncating
6       // to zero bytes; file permissions read+write only
   for owner
7       fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC,
   S_IRUSR | S_IWUSR);
8
```

4.1.2 read

Prende in input il file descriptor ottenuto tramite la `open`, un `buffer` dove andremo a salvare quello che leggeremo dal file e un `size_t` che definisce il numero di byte che vogliamo leggere dal file. In caso di successo ritornerà un valore `ssize_t` che dovrebbe essere uguale o minore a `count`, in di errore tornerà un -1.

```
1       #include <stdio.h>
2
3       //Returns number of bytes read, or -1 on error
4       ssize_t read(int fd, void *buf, size_t count);
5
```

Esempio d'uso:

```
        //Open existing file for reading
        int fd = open("myfile", O_RDONLY);
        if(fd == -1)
            errExit("open");

        // A MAX_READ bytes buffer
        char buffer[MAX_READ + 1];

        //Reading up to MAX_READ bytes from myfile
        ssize_t numRead = read(fd, buffer, MAX_READ);
        if(numRead == -1)
            errExit("Read");

```

Un esempio di lettura da Standard Input:

```
        // A MAX_READ bytes buffer
        char buffer[MAX_READ + 1];

        //Reading up to MAX_READ bytes from STDIN
        ssize_t numRead = read(STDIN_FILENO, buffer, MAX_READ
    );
        if(numRead == -1)
            errExit("read");

        buffer[numRead] = '\0';
        printf("Input data: %s\n", buffer);

```

4.1.3 write

Ci permette di scrivere su un file descriptor

```
        #include <unistd.h>

        //Returns number of bytes written, or -1 on error
        sszie_t write(int fd, void *buf, size_t count);

```

Esempio di scrittura:

```
        //Open existing file fro writing
        int fd = open("myfile", O_WRONLY);
        int(fd == -1)
            errExit("open");

        //A buffer collecting the string
        char buffer[] = "Ciao Mondo";

        //Writing up to sizeof(buffer) bytes into myfile
```

```
10          ssize_t numWrite = write(fd, buffer, sizeof(buffer));
11          if(numWrite != sizeof(buffer))
12              errExit("write");
13
```

Per scrivere su terminale, come prima si userà `STDOUT_FILENO` al posto del file descriptor.

4.1.4 lseek

Una volta aperto un file, il kernel salva un file offset ovvero un indicatore un valore che identifica a quale punto di scrittura/lettura siamo arrivati. Per utilizzare tale cursore useremo la `lseek`.

```
1           #include <unistd.h>
2
3           //Returns the resulting offset location, or -1 on
    error
4           off_t write(int fd, off_t offset, int whence);
5
```

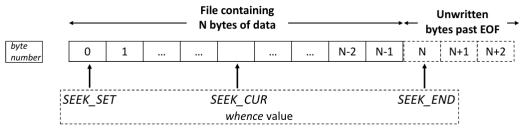
N.B.: `whence` indica la base di partenza dell'offset;

Esempio di utilizzo:

```
1           #include <unistd.h>
2
3           //Returns number of bytes written, or -1 on error
4           sszie_t write(int fd, void *buf, size_t count);
5
```

Alcuni esempi:

```
1           //first byte of the file
2           off_t current = lseek(fd1, 0, SEEK_SET);
3           //last byte of the file
4           off_t current = lseek(fd2, -1, SEEK_END);
5           //10th byte past the current offset location of the
    file
6           off_t current = lseek(fd3, -10, SEEK_CUR);
7           //10th byte after the current offset location of the
    file
8           off_t current = lseek(fd4, 10, SEEK_CUR);
9
```



4.1.5 close

```
1         #include <unistd.h>
2
3         //Returns 0 on success, or -1 on error
4         int close(int fd);
5
```

Tutti i file descriptor vengono chiusi quando un processo termina, ma è buona prassi chiudere sempre. La chiusura **non** elimina il file.

4.1.6 unlink

```
1         #include <unistd.h>
2
3         //Returns 0 on success, or -1 on error
4         it unlink(const char *pathname);
5
```

Prende in input il nome del file, perché il file descriptor può anche essere chiuso, se il file non ha altri symbolic link, viene rimosso.

Symbolic link: il collegamento su desktop, oppure il file è aperto da altri processi.

`unlink` **non** può rimuovere directory.

4.1.7 stat, lstat, fstat

```
1         #include <sys/stat.h>
2
3         //Returns 0 on success, or -1 on error
4         int stat(const char *pathname, struct stat *statbuf);
5         int lstat(const char *pathname, struct stat *statbuf)
    ;
6         int fstat(int fd, struct stat *statbuf);
7
```

In caso di succeso la `struct stat` viene popolata da varie informazioni. La differenza tra queste system call sono:

- `stat` ritorna informazioni relative a un file tramite il nome o path;
- `lstat` tramite symbolic link;
- `fstat` utilizza il file descriptor;

4.1.8 Mode

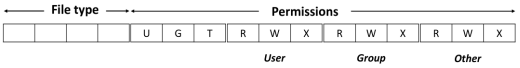
Si tratta di una bit mask, presente anche nella `struct stat` , che ci permette di definire i permessi dei file. Lunga 16 bit, i primi 9 sono per other, group e user, rispettivamente 3 a testa. Possiamo usarla per avere informazioni sulla tipologia del file. Esempio:

```
        char pathname[] = "/tmp/file.txt";
        struct stat statbuf;
        //Getting the attribute of /tmp/file.txt
        if(stat(pathname, &statbuf) == -1)
            errExit("stat");

        //Checking if /tmp/file.txt is a regular file
        if((statbuf.st_mode & S_IFMT) == S_IFREG)
            prinf("regular file!\n);

        //Equivalently, checking if /tmp/file.txt is a
        //regular file by S_ISREG macro
        if(S_ISREG(statbuf.st_mode))
            printf("regular file!\n");

```



I bit oltre i 9 dedicati a other, group e user hanno i seguenti significati:

- U identifica se l'utente che sta eseguendo quell'eseguibile è lo stesso utente proprietario dell'eseguibile;
- G verifica se il gruppo che sta eseguendo è il gruppo proprietario;
- T è lo sticky bit, funziona come un bit che non ci permette di cancellare quel file;

File e directory sono la stessa cosa, per modificare i permessi di una directory si usano le stesse mode dei file.

4.1.9 access

Controlla l'accessibilità di un file relativamento al nostro user id e group id.

```
        #include <unistd.h>

        //Returns 0 if all permissions are granted, otherwise
    -1
        int access(const char *pathname, int mode)

```

Le possibili mode che si possono usare insieme a questa system call sono queste:

Constant	Description
F_OK	Does the file exist?
R_OK	Can the file be read?
W_OK	Can the file be written?
X_OK	Can the file be executed?

4.1.10 chmod e fchmod

Permettono di cambiare i permessi di un file prendendo in input il pathname o il file descriptor più le eventuali mode di interesse.

```
1         //All return 0 on success, or -1 on error
2         #include <sys/stat.h>
3
4         int chmod(const char *pathname, mode_t mode);
5
6         #define _BSD_SOURCE
7         #include <sys/stat.h>
8
9         int fchmod(int fd, mode_t mode);
10
```

4.2 Directory

4.2.1 mkdir

Prende in input il pathname della directory e una mode.

```
1         #include <sys/stat.h>
2
3         //Returns 0 on success, or -1 on error.
4         int mkdir(const char *pathname, mode_t mode);
5
```

I parametri mode sono gli stessi della `open`, se la directory esistesse già, il valore di ritorno sarà sempre -1, ma la variabile `errno` conterrà il messaggio `EEXIST`, a indicare che tale directory è già presente.

4.2.2 rmdir

```
        #include <unistd.h>

        //Returns 0 on success, or -1 on error.
        int rmdir(const char *pathname);

```

Se esiste anche un solo file all'interno dello directory tale system call tornerà -1 di errore, per avere successso deve essere completamente vuota.

4.2.3 opendir, closedir e readdir

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

        //Returns directory stream handler, or NULL on error.
        DIR *opendir(const char *dirpath);

        //Returns 0 on success, or -1 on error
        int closedir(DIR *dirp);

```

Una volta aperta una directory per leggerla si userà

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

        //Returns pointer to an allocated structure
   describing the
        //next directory entry or NULL on end-of-directory or
    error
        struct dirent *readdir(DIR *dirp);

```

La struttura di riferimento per la `readdir`:

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

Tramite il `d_type` possiamo ottenere delle informazioni sul tipo di file che stiamo scorrendo:

Constant	File type
DT_BLK	block device
DT_CHR	character device
DT_DIR	directory
DT_FIFO	named pipe (FIFO)
DT_LNK	symbolic link
DT_REG	regular file
DT_SOCK	UNIX socket

Esempio d'uso:

```
1         DIR *dp = opendir("myDir");
2         if(dp == NULL)
3             return -1;
4
5         errno = 0;
6         struct dirent *dentry;
7         //Iterate until NULL is returned as a result
8         while((dentry = readdir(dp)) != NULL){
9             if(dentry->d_type == DT_REG)
10                printf("Regular file: %s\n", dentry->d_name);
11            errno = 0;
12        }
13        //NULL is returned on error, and when the end-of-
    directory is reached!
14        if(errno != 0)
15        printf("Error while reading dir.\n");
16        closedir(dp);
17
```