

SISTEMI OPERATIVI

Esercitazione 3

1 **strace** : visualizzazione delle system call invocate da un processo

Il programma **strace** consente di visualizzare le system call (con i relativi argomenti) invocate da un processo in esecuzione. È uno strumento di debug molto istruttivo che permette di ottenere, in tempo reale su schermo o in un file di testo, numerose informazioni sull'esecuzione di un programma.

Può essere invocato in due modalità principali :

1. **strace ./nomeprogramma** [argomenti] (*esegue nomeprogramma*)
2. **strace -p pid_processo_in_esecuzione** (*visualizza le system call invocate dal processo con il PID specificato, già in esecuzione*) .

Per memorizzare su file la traccia delle system call invocate, utilizzare l'opzione **-o** : ad es. **strace -o traccialog.txt ./nomeprogramma** .

Per ottenere la traccia delle system call invocate da un processo e dai suoi processi figli, utilizzare l'opzione **-f** : ad es. **strace -f ./nomeprogramma** .

Si tratta di una funzionalità molto importante, difficilmente ottenibile in un normale debugger che può soltanto seguire il processo padre oppure il processo figlio (cfr. il comando **set follow-fork-mode** di **gdb**).

Per tracciare solo alcune system call utilizzare l'opzione **-e trace** : ad es.
strace -e trace=open,close,read,write nomeprogramma , oppure utilizzare i nomi delle categorie di system call, ad esempio: **strace -e trace=file ./nomeprogramma**
strace -e trace=signal ./nomeprogramma
strace -e trace=process ./nomeprogramma
strace -e trace=network ./nomeprogramma

Per le altre opzioni consultare il manuale di **strace**.

Si consiglia vivamente di eseguire almeno una volta con **strace** i programmi presentati nel seguito di questa e delle prossime esercitazioni.

Gli utenti Mac possono fare riferimento all'analogo comando **dtruss**, che va utilizzato con privilegi elevati mediante **sudo** (ad es. **sudo dtruss ./es3**)

NOTA BENE : Il contenuto dell'archivio zip, con i file da utilizzare nelle esercitazioni scaricato dal sito Elly del corso, va estratto e memorizzato in un'opportuna cartella del proprio *home directory*, utilizzando il comando **unzip** o il

file manager: *unzip so-esercitazioni.zip*. I file saranno memorizzati in una cartella denominata *so-esercitazioni* con sottocartelle *eserc1*, *eserc2*,...,*eserc7*. Nel seguito i riferimenti al percorso */temi_esame/Unix/so/* vanno sostituiti con il percorso in cui è stato estratto l'archivio zip scaricato dal sito del corso su Elly, ad esempio *~/so-esercitazioni*.

2 Chiamate di sistema per l'I/O

Quando un programma viene messo in esecuzione, possiede almeno tre file descriptor:

- 0 Standard input
- 1 Standard output
- 2 Standard error

Vediamo le principali chiamate di sistema che fanno uso dei descrittori di file. La maggior parte di queste chiamate restituisce -1 in caso di errore e assegna alla variabile *errno* il codice di errore. I codici di errore sono documentati nelle pagine di manuale delle singole chiamate di sistema e in quella di *errno*. La funzione *perror()* può essere utilizzata per visualizzare un messaggio di errore basato sul relativo codice.

2.1 write()

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

Quando si utilizza la chiamata di sistema *write()*, i primi *count* byte di *buf* vengono scritti nel file che è stato associato al file descriptor *fd*. La chiamata restituisce il numero dei byte scritti oppure -1 se si è verificato un errore (in questo caso si deve controllare la variabile *errno* o utilizzare *perror/strerror*).

Esercizio 1:

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

int main()
{
    char s[100];
    size_t sl;

    strcpy(s, "Questa stringa andra' sullo standard output\n");
    sl = strlen(s);

    if ((write(1, s, sl)) == -1)
        perror("write error");
    /* visualizza questa stringa e un messaggio descrittivo dell' errore */

    return 0;
}
```

Eseguire il programma anche con il debugger *ddd* o direttamente con *gdb*.

2.2 read()

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

Quando si utilizza la chiamata di sistema `read()`, i primi `count` byte di `buf` vengono letti dal file che è stato associato al file descriptor `fd`. La chiamata restituisce il numero di byte che sono stati letti.

Esercizio 2:

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

#define N 256

int main()
{
    char buffer[N];
    int nread;

    nread = read(0, buffer, N);
    if (nread == -1)
        perror("read error");

    if ((write(1, buffer, nread)) != nread)
        perror("write error");

    return 0;
}
```

Utilizzare i comandi di redirectione dello shell perchè il programma abbia come input (*stdin*) il suo stesso file sorgente.

Modificare il programma affinché effettui la conversione delle lettere maiuscole presenti nello *stdin* in lettere minuscole (si veda il manuale del predicato `isupper()`, definito in `ctype.h`, che si limita a verificare se un carattere è maiuscolo - la modifica di ogni carattere maiuscolo in minuscolo deve essere fatta esplicitamente, ad es. con `tolower()`).

2.3 open() e close()

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

La chiamata di sistema `open()` apre un file e restituisce un intero che ne rappresenta il file descriptor. Il parametro `pathname` è il nome del file che si vuole aprire. Il parametro `flags` è uno tra `O_RDONLY` (per aprire il file in modalità read-only), `O_WRONLY` (per aprire il file in modalità write-only), o `O_RDWR` (per aprire il file in modalità read e write); può essere

sommato logicamente con flag aggiuntivi (si veda il manuale per un elenco completo). Il terzo parametro serve quando si utilizza `open()` per creare un nuovo file (con il flag `O_CREAT`): specifica, in formato ottale, i permessi di accesso per user, group e others. Ad esempio, 0764 significa: user 111 (rwx), group 110 (rw), others 100 (r).

```
#include <unistd.h>
```

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

La chiamata di sistema `close()` serve a chiudere un file specificandone il file descriptor.

Esercizio 3 (copia un file preesistente in un nuovo file):

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

```
#include <stdlib.h>
```

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

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

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

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

```
#define PERM 0644
```

```
#define N 2048
```

```
int main(int argc, char **argv)
```

```
{
```

```
    int infile, outfile, nread, nwrite;
```

```
    char buffer[N];
```

```
    if(argc != 3)
```

```
    {
```

```
        fprintf(stderr, "Uso: %s nomefilesorgente nomefiledestinazione\n", argv[0]);
```

```
        exit(1);
```

```
    }
```

```
    if((infile = open(argv[1], O_RDONLY)) < 0)
```

```
    {
```

```
        fprintf(stderr, "Il file %s non e' accessibile : %s\n", argv[1], strerror(errno));
```

```
        exit(2);
```

```
    }
```

```
    if((outfile = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, PERM))
```

```
    {
```

```
        fprintf(stderr, "Il file %s non puo' essere creato :
```

```
%s\n", argv[2], strerror(errno));
```

```
        exit(3);
```

```

    }

    while ((nread = read(infile, buffer, N)) > 0)
        write(outfile, buffer, nread);

    close(infile);
    close(outfile);

    return 0;
}

```

Nel file sorgente `es3.c` aggiungere il controllo degli errori sulle chiamate per l'apertura, la creazione e la scrittura dei file. Come potete verificarne il comportamento ?

Si modifichi il precedente sorgente (*copia-readwrite.c*) per utilizzare, al posto di *N*, un valore passato come argomento di invocazione. Si crei anche una versione (*copia-sendfile.c*) basata sull'uso della system call `sendfile` vista a lezione e si confrontino i tempi di esecuzione tramite il comando `time`, ad es. :

```

time copia-readwrite filesorgente filedestinazione 1
time copia-readwrite filesorgente filedestinazione 512
time copia-readwrite filesorgente filedestinazione 2048
time copia-sendfile filesorgente filedestinazione

```

2.4 lseek()

```

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

```

La funzione `lseek()` imposta la posizione corrente di lettura e scrittura in corrispondenza del valore `offset`, all'interno del file referenziato dal descrittore `fildes`. A seconda del valore di `whence`, l'offset è relativo all'inizio (`SEEK_SET`), alla posizione corrente (`SEEK_CUR`), o alla fine del file (`SEEK_END`).

Esercizio 4 (occorre il file *test.dat* contenuto in */temi_esame/Unix/so/eserc3*):

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define REC_LEN 19

int main() {

```

```

int f;
long int length, cur_pos;
char buf[REC_LEN];

if((f = open("test.dat", O_RDWR)) < 0)
{
    perror('Apertura di test.dat impossibile');
    exit(1);
}

printf("This file contains %ld bytes\n", length = lseek(f, 0, SEEK_END));
/* Si riposiziona all'inizio */
cur_pos = lseek(f, 0, SEEK_SET);

/* Il file contiene dei record di REC_LEN Byte;
   ne viene letto uno ogni 6 */

while (cur_pos < length) {
    read(f, buf, REC_LEN);
    write(1, buf, REC_LEN); // Visualizza il record corrente
    /* Si sposta avanti di 5 record */
    cur_pos = lseek(f, 5*REC_LEN, SEEK_CUR);
}
close(f);

return 0;
}

```

Modificare il programma affinché legga (dal file) e scriva (sullo standard output) solo i primi $N < \text{REC_LEN}$ caratteri di ciascun record.

2.5 fstat()

```

#include <sys/stat.h>
#include <unistd.h>
int fstat(int filedes, struct stat *buf);

```

La chiamata di sistema `fstat()` restituisce informazioni sul file referenziato dal descrittore `filedes`, memorizzando il risultato nella struct `stat` (vedere il manuale) puntata da `buf`.

Esercizio 5:

```

#include <sys/stat.h>

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

```

```

int main(void)
{
    struct stat statbuf;
    int file;

    /* apre il file test.dat dell'esercizio 4 */
    file = open("test.dat", O_RDONLY);
    if (file == -1)
    {
        perror("file opening error");
        exit(1);
    }

    /* ottiene informazioni sul file */
    fstat(file, &statbuf);
    close(file);

    /* mostra le informazioni ottenute */
    if (statbuf.st_mode & S_IFCHR)
        printf("Handle refers to a device.\n");
    if (statbuf.st_mode & S_IFREG)
        printf("Handle refers to an ordinary file.\n");
    if (statbuf.st_mode & S_IRREAD)
        printf("User has read permission on file.\n");
    if (statbuf.st_mode & S_IWRITE)
        printf("User has write permission on file.\n");

    printf("Size of file in bytes: %ld\n", statbuf.st_size);
    printf("Time file last opened: %s\n", ctime(&statbuf.st_ctime));

    return 0;
}

```

Modificare l'esercizio scegliendo il file da esaminare dall'argomento di invocazione e utilizzando la system call `stat`.

2.6 `dup()` e `dup2()`

```

#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);

```

Le chiamate di sistema `dup()` e `dup2()` creano una copia del descrittore di file `oldfd`. Il nuovo descrittore è, nel caso della `dup()`, il descrittore non utilizzato con numero d'ordine più basso; nel caso della `dup2()`, è il `newfd` specificato dall'utente.

Esercizio 6 (redirezione dello standard output):

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

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

#define PERM 0644

int main()
{
    int fd;

    fd = open("file.txt", O_CREAT|O_WRONLY|O_TRUNC, PERM);
    if (fd == -1)
    {
        perror("open error");
        exit (1);
    }
    /* ora chiude lo standard output */
    close(1);
    /* poi duplica fd nel descrittore libero piu' basso (1!) */
    dup(fd);

    /* infine scrive in 1, che ora e' una copia del descrittore del file */
    write(1, "Hello from write\n", 17);
    /* anche printf() scrive sul file */
    printf("Hello from printf\n");

    return 0;
}
```

Modificare l'esempio precedente in modo da ottenere lo stesso risultato utilizzando la chiamata `dup2()`.

2.7 `link()` e `unlink()`

La chiamata di sistema `link()` crea un hard link (`alias_name`) al file il cui nome è passato in ingresso (`original_name`).

```
#include <unistd.h>
int link(char *original_name, char *alias_name);
```

Consultare il manuale per sapere in quali casi la chiamata fallisce.

Esercizio 7:


```
#include <unistd.h>

int main()
{
    if ((link("test.dat", "test2.dat")) == -1)
    {
        perror("link error");
        return 1;
    }
    return 0;
}
```

La chiamata di sistema opposta è `unlink()`, che rimuove un hard link (`alias_name`).

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

Consultare il manuale per sapere in quali casi la chiamata fallisce.

Esercizio 8:

```
#include <unistd.h>

int main()
{
    if ((unlink("test2.dat")) == -1)
    {
        perror("unlink error");
        return 1;
    }
    return 0;
}
```

2.8 Esercizio riepilogativo

Scrivere un programma C che esegua le seguenti operazioni:

1. creazione di un file contenente un vettore di strutture; ogni struttura deve contenere due campi di tipo stringa con dimensioni prefissate: *nome* e *età*, ad es. :

```
struct persona {
    char nome[16],
    char eta[3];
};
```

2. ricerca sul file e visualizzazione su standard output dei campi della *i*-esima struttura (record), dove *i* è un parametro definito dall'utente come argomento di invocazione del programma.