Teoria UNIX

Matteo Franchini

2 settembre 2023

Indice

1 Programmazione di sistema UNIX

1.1 Argomenti di programma

Un programma può accedere agli eventuali argomenti di invocazione attraverso i parametri della funzione principale **main**

```
main (int argc, char *argv[]) {
        int i;
        printf("Numero di argomenti = %d\n", argc);
        for (i = 0; i < argc; i++) {
                printf("Argomento %d (argv[%d]) = %s\n", i, i, argv[i]);
        }
}
```

si noti che `%d` si usa per indicare che in quel punto ci va un **intero**, mentre `%s` si usa per indicare che ci va una **stringa**

1.2 Compilazione

```
gcc -o mioprogramma mioprogramma.c
```

1.3 Eseguendo il programma

```
./mioprogramma 1 pippo pluto 4
```

1.4 Variabile di ambiente

```
main(int argc, char *argv[], char **envp) {
        int i;
        prinft("Numero di argomenti (argc) = %d\n", argc);
        for (i = 0; i < argc; i++) {
                printf("Argomento %d (argv[%d]) = %s\n", i, i, argv[i]);
        }
        while (*envp != NULL) { printf("%s\n", envp++); }
}
```

1.5 Perror e Streerror

`perror` e `strerror` permettono di visualizzare o di generare messaggi descritti dell'errore

```
if (syscall_N (..., ...) < 0)
{
        perror("Errore nella syscall_N");
        /*
        la descrizione dell'errore viene concatenata
        alla stringa argomento
        */
        exit(1); // terminazione del processo con errore
}
```

2 Operazioni sui file

2.1 Apertura file

Apertura ed eventuale creazione di un file

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

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

oppure

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

```
#include <fcntl.h>

fd = int open (const char *pathname, int flags, mode_t mode);
```

- `pathname` è il nome del percorso
- `flags` contiene il modo di accesso richiesto: uno tra `O_RDONLY`, `O_WRONLY`, `O_RDWR` più altre eventuali OR
  Esempio:

  ```
  O_WRONLY|O_CREAT|O_TRUNC
  ```

  per richiedere la creazione di un nuovo file o per azzerarlo se già esiste
- `mode` indica i **diritti di accesso**

Se l'**invocazione della primitiva** `open` ha successo, viene restituito al processo un valore intero ≥ 0 che costituisce il **file descriptor (fd)** per quel file

2.2 Duplicazione file descriptor

```
#include <unistd.h>

int dup (int oldfd);
int dup2(int oldfd, int newfd);
```

2.3 Chiusura di un file descriptor

```
#include <unistd.h>

int close (int fd);
```

2.4 Lettura e scrittura di un file descriptor

```
#include <unistd.h>

int read (int fd, void *buf, size_t count);
int write (int fd, void *buf, size_t count);
```

- `read` prova a leggere dall'oggetto a cui si riferisce `fd` fino a `count` byte, memorizzandoli a partire dalla locazione `buf`
- `write` prova a scrivere sull'oggetto a cui si riferisce `fd` fino a `count` byte, letti a partire dalla locazione `buf`|

2.5 Esempio di lettura/scrittura

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

#define BUFSIZ 4096

main () {
        char *f1 = "filesorg";
        char *f2 = "/tmp/filedest";
        char buffer[BUFSIZ];
        int infile, outfile; // file descriptor
        int nread;

        // apertura file sorgente

        if ((infile = open(f1, O_RONLY)) < 0) {
                perror("Apertura f1");
                exit(1);
        }

        /* open mi permette di aprire il file
         * quando scriviamo infile = ... stiamo assegnando
         * il valore restituito dalla chiamata open a infile
         * in modo da poter accedere al file utilizzando
         * direttamente infile, invece che il file descriptor
         * metto la condizione < 0 in quanto se abbiamo un
         * errore nell'apertura il fd < 0
         */

         // creazione file destinazione
         if ((outfile = open(f2, WRONLY|O_CREAT|O_TRUNC, 0644)) < 0) {
```

```
                perror("Creazione f2");
                exit(2);
        }

        /* ho messo O_WRONLY or O_CREAT or O_TRUNC
         * in quanto se esiste il file ci scrivo sopra,
         * altrimenti lo creo
         * oppure lo sovrascrivo
         */

        // ciclo di lettura/scrittura

        while ((nread = read(infile, buffer, BUFSIZ)) > 0) {
                if (write (outfile, buffer, nread) != read) {
                        perror("Errore write");
                        exit(3);
                }
                if (nread < 0) {
                        perror("Errore read");
                        exit(4);
                }
        }
        close(infile);
        close(outfile);
        exit(0);
}
```

2.6 Trasferire dati tra descrittori

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

ssize_t sendfile (int out_fd, int in_fd, off_t *offset, size_t count);
```

`sendfile` copia dati da un file descriptor all'altro **rimanendo all'interno del kernel**, quindi è più efficiente dell'uso combinato di read e write che trasferiscono dati tra spazio utente e kernel

- Se `offset` non è NULL, indica l'indirizzo di una variabile contenente lo spiazzamento da cui iniziare la lettura da `in_fd` che sarà modificata all'offset successivo all'ultimo byte letto; `count` è il numero di byte da copia

- Se `offset` non è NULL, allora `sendfile()` non modifica il file offset di `in_fd` altrimenti esso viene aggiustato per riflettere il numero di byte letti da `in_fd`

2.7 Copia di file con sendfile

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

int main (int argc, char* argv[]) {
        int read_fd, write_fd;
        struct stat stat_buf;
        off_t offset = 0;

        // apertura file input

        read_fd = open(argv[1], O_RDONLY);

        /* con fstat otteniamo le informazioni
         * del file che viene aperto e le salviamo
         * all'interno di stat_buf
         * in particolare in questo caso lo facciamo
         * per ottenere la dimensione del file
         */

        fstat(read_fd, &stat_buf);

        /* apriamo il file di lettura con gli stessi
         * permessi del file sorgente "stat_buf.st_mode"
         */

         write_fd = open (argv[2], O_WRONLY | O_CREAT, stat_buf.st_mode);

         sendfile (write_fd, read_fd, &offset, stat_buf.st_size);
```

```
    close (read_fd);
    close (write_fd);

    return 0;
}
```

2.8 Informazioni su file (ordinari, speciali, direttori)

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

int stat (const char *filename, struct stat *buf);
int fstat (int fd, struct stat *buf);
```

```
struct stat
   {
   dev_t         st_dev;     /* device */
   ino_t         st_ino;     /* inode */
   mode_t        st_mode;    /* protection */
   nlink_t       st_nlink;   /* number of hard links */
   uid_t         st_uid;     /* user ID of owner */
   gid_t         st_gid;     /* group ID of owner */
   dev_t         st_rdev;    /* device type (if inode device) */
   off_t         st_size;    /* total size, in bytes */
   unsigned long st_blksize; /* blocksize for filesystem I/O */
   unsigned long st_blocks;  /* number of blocks allocated */
   time_t        st_atime;   /* time of last access */
   time_t        st_mtime;   /* time of last modification */
   time_t        st_ctime;   /* time of last status change */
   };
```

Figura 1: Struttura buf

2.9 Cancellazione di file

```
#include <unistd.h>

int unlink (const char *filename);
```

Il file viene cancellato solo se: si tratta dell'ultimo link al file, non vi sono altri processi che lo hanno aperto

3 Primitive per la gestione degli accetti

3.1 Creazione di un nuovo processo

```
#include <unistd.h>

int fork (void);
```

Viene creato un nuovo processo (figlio) identico al processo padre che ha invocato `fork()`.
Solo il valore di uscita della `fork` è diverso per i due processi

```
pid = fork();
```

per il padre `pid` vale il pid del figlio, per il figlio `pid = 0`

3.2 Sistema di generazione

```
#include <unistd.h>

if (fork() == 0) {
        // codice del FIGLIO
}
else {
        // codice del PADRE
}
```

il padre può decidere se continuare la propria esecuzione concorrentemente a quella del figlio, oppure attendere che il figlio termini (**primitiva `wait`**)

3.3 Identificazione dei processi

```
#include <unistd.h>

pid_t getpid (void);
pid_t getppid (void);
```

La `getpid` ritorna al processo chiamante il suo PID, mentre `getppid` ritorna al processo chiamante l'identificatore di processo di suo padre (PID del PADRE)

3.4 Sincronizzazione tra padre e figlio

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

pid_t wait (int *status);
```

Il processo chiamante rimane bloccato in attesa della terminazione di uno tra i suoi figli.
Se la `wait` ha successo il valore di ritorno è il PID del processo figlio che è terminato

3.5 Uso della wait

```
int status;

if (fork () == 0) {
        // Codice del figlio
}
else {
        // Codice del padre
        ...
        // Attende che il figlio termini
        wait(&status);
}
```

Nel caso di più figli in esecuzione può essere necessario attenere **la terminazione di uno specifico figlio**

```
while (pid = wait(&status) != pidfiglio);
```

oppure direttamente

```
waitpid(pidfiglio, &status, NULL)
```

3.6 Terminazione volontaria di un processo

```
#include <stdlib.h>
void exit(int status);

#include <unistd.h>
void _exit(int status)
```

Un processo **termina volontariamente** invocando la primitiva `_exit` oppure la funzione `exit` della libreria standard I/O di C

3.7 Esecuzione di un programma

```
#include <unistd.h>

int execve (const char *pathname, char *const argv[], char *const envp[]);
```

Il processo chiamante passa ad eseguire il programma `filename` La fork **crea un nuovo processo identico al padre**, la exec permette **di modificare l'ambiente di esecuzione di un processo**

3.8 Esempio di uso della execve

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

int main () {
        int status;
        pid_t pid;
        char *env[] = {
                "TERM=vt100",
                "PATH=/bin:/usr/bin",
                (char *) 0
        };

        char *args[] = {
                "cat",
                "f1",
                "f2",
                (char *) 0
        };

        if ((pid=fork()) == 0) {
                // codice del figlio
                execve("bin/cat", args, env);

                /* si torna a questo punto solo
                 * nel caso in cui si verifichi un
                 * errore
                 */
                perror("excve")
                exit(1);
```

```
        }
        else {
                // codice del padre
                wait(&status);
                printf("Il processo %d e' terminato con %d\n",
                pid, WEXITSTATUS(status));
        }
        exit(0);
}
```

4 Primitive per la gestione dei segnali

Elenco delle figure