Lezione 7 - 26/4/21 (Errori in C, Pipe anonime)

Gestione errori in C

Gestione errori in C

Durante l'esecuzione di un programma ci possono essere diversi tipi di errori: system calls che falliscono, divisioni per zero, problemi di memoria etc..

Alcuni errori non fatali possono essere indagati attraverso la variabile **errno**. Questa variabile globale contiene **l'ultimo** codice di errore generato dal sistema.

Funzioni utili

```
char *strerror(int errnum); //converte l'errore da numerico a stringa
```

```
/*stampa su stderr la stringa passatagli come argomento
separata da ':' e concatenata con strerror(errno)*/
void perror(const char *str);
```

Es:

```
#include <stdio.h> <errno.h> <string.h>
extern int errno;
void main(){
  FILE *pf;
  pf=fopen("nonExistingFile.boh","rb");
  if(pf==NULL){
    fprintf(stderr, "errno= %d\n", errno);
    perror("Error printed by perror"); //stampa success se non ci sono errori
    fprintf(stderr,"strerror: %s\n", strerror(errno));
  } else {
    fclose(pf);
  }
}
```

Piping

Il piping connette l'output (stdout e stderr) di un comando all'input (stdin) di un altro comando, consentendo dunque la comunicazione tra i due.

I processi sono eseguiti in concorrenza utilizzando un buffer:

- Se pieno lo scrittore (left) si sospende fino ad avere spazio libero
- Se vuoto il lettore si sospende fino ad avere i dati

Come già visto il **piping** tra due processi/app si fa tramite '**|**' (bash).

Es:

```
//output.out
#include <stdio.h>
#include <unistd.h>
void main(){
  for(int i=0; i<3; i++){
    sleep(2);
    fprintf(stdout, "Written in buffer");
    fflush(stdout);
  }
}
```

```
//input.out
#include <stdio.h>
#include <unistd.h>
void main(){
  char msg[50];
  int n=3;
  while((n--)>0){
    int c=read(0,msg,50);
    if(c>0){
      msg[c]=0;
      fprintf(stdout, "Read: '%s' (%d)\n",msg,c);
    }
  }
}
```

Pipe anonime

Le pipe anonime sono come quelle usate su shell, sono unidirezionali e usano un buffer per comunicare.

Comandi base:

- **pipe():** Crea un buffer
- **write():** Scrive nel buffer
- **read():** Legge sul buffer
- **close():** Chiude il buffer

(Tutto avviene tramite file descriptors per questo serve una antenato comune)

ATTENZIONE!!

Esiste solo una pipe associata a più file descriptors

Creazione di pipe

```
int pipe(int pipefd[2]); //fd[0] lettura, fd[1] scrittura
```

```
//pipe.c
#include <stdio.h>
#include <unistd.h>
void main(){
  int fd[2];
  int esito=pipe(fd);
  if(esito==0){
    write(fd[1], "writing",8);
    char buf[50];
    int c=read(fd[0],&buf,50);
    printf("Read '%s' (%d)\n",buf,c);
  }
}
```

Lettura da pipe

La lettura della pipe tramite il comando read restituisce valori differenti a seconda della situazione:

- In caso di successo, read() restituisce il numero di bytes effettivamente letti

- Se il di scrittura è stato chiuso (da ogni processo) ed il buffer è vuoto restituisce 0
- Se il buffer è vuoto ma il lato di scrittura è ancora aperto (in qualche processo) il processo si sospende fino alla disponibilità dei dati o alla chiusura
- Se si provano a leggere più bytes (num) di quelli disponibili, vengono recupoerati solo quelli presenti

```
int read(int fd[0], char *data, int num)
```

```
//readPipe.c
#include <stdio.h>
#include <unistd.h>
void main(){
  int fd[2];
  char buf[50];
  int esito=pipe(fd); //create unnamed pipe
  if(esito==0){
    write(fd[1],"writing",8); //write to pipe
    int r = read(fd[0],&buf,50); //read from pipe
    printf("Last read %d. Recived '%s' \n",r,buf);
    //close(fd[1]); //hang when commented
    r=read(fd[0], &buf, 50); //read from pipe
    printf("last read %d. Recived '%s' \n",r,buf);
  }
}
```

Scrittura su pipe

```
int write(int fd[0], char *data, int num);
```

La scrittura della pipe tramite tramite comando **write** restituisce il numero di bytes scritti, -1 in caso di **errore** e 0 se non abbiamo **scritto nulla**.

Tuttavia, se il lato in lettura è stato chiuso viene inviato un segnale SIGPIPE allo scrittore (default handler quit).

In caso di scrittura, se vengono scritti meno bytes di quelli che ci possono stare la scrittura è "atomica" (tutto assieme), in caso contrario non c'è garanzia di atomicità e la scrittura sarà bloccata (o fallirà se il flag O_NONBLOCK viene usato).

```
int fcntl(int fd, F_SETFL, O_NONBLOCK);
```

Funzioni lettura scrittura

```
int write(int fd[0], char * data, int num);
```

Restituisce **il numero di byte** scritti, se il lato lettura è chiuso viene inviato un segnale **SIGPIPE** allo scrittore (default handler quit).

Comunicazione unidirezionale

Un tipico esempio di comunicazione unidirezionale tra un processo scrittore P1 ed un processo lettore P2 è il seguente:

- **P1** crea una **pipe()**
- **P1** esegue un **fork()** e crea P2
- **P1** chiude il lato **lettura**: **close(fd[0])**
- **P2** chiude il lato **scrittura**: **close(fd[1])**
- **P1 e P2** chiudono l'altro **fd** appena finiscono.

```
//uni.c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
void main(){
    int fd[2];
    char buf[50];
    pipe(fd); //P1 crea una pipe
    int p2 = !fork(); //P1 fa un fork e crea P2
    if(p2){
        close(fd[1]); //P2 chiude il lato scrittura
        int r = read(fd[0],&buf,50); //Read from pipe
        close(fd[0]); printf("Buf: '%s'\n",buf);
    }else{
        close(fd[0]); //P1 chiude il lato lettura
        write(fd[1],"writing",8); // Writes to pipe
        close(fd[1]);
    }
```

```
    while(wait(NULL)>0);
}
```

Comunicazione Bidirezionale

Un tipico esempio di comunicazione bidirezionale tra un processo scrittore P1 ed un processo lettore P2 è il seguente:

- P1 crea due pipe(), pipe1 e pipe2
- P1 esegue un fork() e crea P2
- P1 chiude il lato lettura di pipe1 ed il lato scrittura di pipe2
- P2 chiude il lato scrittura di pipe1 ed il lato lettura di pipe2
- P1 e P2 chiudono gli altri fd appena finiscono di comunicare.

```
//bi.c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#define READ 0
#define WRITE 1
void main(){
  int pipe1[2], pipe2[2];
  char buf[50];
  pipe(pipe1);
  pipe(pipe2); //Create two unnamed pipe
  int p2 = !fork();
  if(p2)
  {
    close(pipe1[WRITE]); close(pipe2[READ]);
    int r = read(pipe1[READ],&buf,50); //Read from pipe
    close(pipe1[READ]); printf("P2 received: '%s'\n",buf);
    write(pipe2[WRITE],"Msg from p2",12); // Writes to pipe
    close(pipe2[WRITE]);
  }
  else
  {
    close(pipe1[READ]); close(pipe2[1]);
    write(pipe1[WRITE],"Msg from p1",12); // Writes to pipe
    close(pipe1[WRITE]);
    int r = read(pipe2[READ],&buf,50); //Read from pipe
    close(pipe2[READ]); printf("P1 received: ‘%s’\n",buf);
  }
  while(wait(NULL)>0);
}
```

```
//redirect.c
#include <stdio.h>
#include <unistd.h>
#define READ 0
#define WRITE 1
int main (int argc, char *argv[]) {
  int fd[2];
  pipe(fd); // Create an unnamed pipe
  if (fork() != 0) { // Parent, writer
    close(fd[READ]); // Close unused end
    dup2(fd[WRITE], 1); // Duplicate used end to stdout
    close(fd[WRITE]); // Close original used end
    execlp(argv[1], argv[1], NULL); // Execute writer program
    perror("connect"); // Should never execute
  } else { // Child, reader
    close(fd[WRITE]); // Close unused end
    dup2(fd[READ], 0); // Duplicate used end to stdin
    close(fd[READ]); // Close original used end
    execlp(argv[2], argv[2], NULL); // Execute reader program
    perror("connect"); // Should never execute
  }
}
```

Pipe con nome (FIFO)

Le pipe con nome, o FIFO, corrispondono a dei **file speciali** nel filesystem grazie ai quali i processi, senza vincoli di gerarchia, possono comunicare. Un **processo** può **accedere** ad una di queste **pipe** se ha i **permessi sul file corrispondente** ed è vincolato,
ovviamente, all'**esistenza del file stesso.**

Essendo oggetti nel file system, si possono usare le funzioni di scrittura/lettura dei file viste nelle scorse lezioni. Una volta creata una pipe con nome, il file associato è persistente!

NB: al contrario di un normale file, una FIFO **deve essere aperta da entrambi i lati** per potervi interagire in **modo ragionevole**.

Creazioni pipe

```
int mkfifo(const char *pathname, mode_t mode);
```

Restituisce -1 in caso di errore.

```
//fifo.c
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
void main(){
    char * fifoName = "/tmp/fifo1";
    mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if doesn’t exist
    perror("Created?");
    if (fork() == 0){
        open(fifoName,O_RDONLY); //Open pipe in read only...stuck!
        printf("Open read\n");
    }else{
        sleep(1);
        open(fifoName,O_WRONLY); //Open pipe in write only
        printf("Open write\n");
    }
}
```

Esempio:

```
//fifoWriter.c
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
void main (int argc, char *argv[]) {
    int fd;
    char * fifoName = "/tmp/fifo1";
    char str1[80],* str2 = "I’m a writer";
    mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if doesn’t exist
    fd = open(fifoName, O_WRONLY); // Open FIFO for write only
    write(fd, str2, strlen(str2)+1); // write and close
    close(fd);
    fd = open(fifoName, O_RDONLY); // Open FIFO for Read only
    read(fd, str1, sizeof(str1)); // Read from FIFO
    printf("Reader is writing: %s\n", str1);
    close(fd);
}
```

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

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
void main (int argc, char *argv[]) {
    int fd;
    char * fifoName = "/tmp/fifo1";
    mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if doesn’t exist
    char str1[80], * str2 = "I'm a reader";
    fd = open(fifoName , O_RDONLY); // Open FIFO for read only
    read(fd, str1, 80); // read from FIFO and close it
    close(fd);
    printf("Writer is writing: %s\n", str1);
    fd = open(fifoName , O_WRONLY); // Open FIFO for write only
    write(fd, str2, strlen(str2)+1); // Write and close
    close(fd);
}
```