

LabSO 2024

Laboratorio Sistemi Operativi - A.A. 2023-2024

Michele Grisafi - michele.grisafi@unitn.it

Pipe su shell

—

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. Esempio:

```
ls . | sort -R          #stdout -> stdin
ls nonExistingDir |& wc  #stdout e stderr -> stdin
cat /etc/passwd | wc | less  #out -> in, out-> in
```

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 (right) si sospende fino ad avere i dati

Esempio

```
$ gcc src.c -o pip.out  
$ (sleep 2 ; echo "hi how are you") | ./pip.out
```

```
#define MAXBUF 10                                //pip.c  
#include <stdio.h>  
#include <string.h>  
  
int main() {  
    char buf[MAXBUF];  
    printf("This is the output of pip\n");  
    fgets(buf, sizeof(buf), stdin); // may truncate!  
    printf("Read '%s'\n", buf);  
    return 0;  
}
```

Cosa succede se il secondo programma termina prima?

Example

```
$ gcc src.c -o inv.out  
$ ls /tmp | ./inv.out
```

```
#include <stdio.h>  
  
int main() {  
    int c, d;  
    // loop into stdin until EOF (as CTRL+D)  
    // read from stdin  
    while ((c = getchar()) != EOF) {  
        d = c;  
        if (c >= 'a' && c <= 'z') d -= 32;  
        if (c >= 'A' && c <= 'Z') d += 32;  
        putchar(d); // write to stdout  
    };  
    return (0);  
}
```

Esempio di una semplice applicazione che legge da stdin e stampa su stdout, invertendo i caratteri minuscoli con quelli maiuscoli.

Esempio

```
// output.out
#include <stdio.h>
#include <unistd.h>
int 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>
int main() {
    char msg[50]; int n=3;
    while((n--)>0){
        int c = read(0,msg,49);
        if (c>0) {
            msg[c]=0;
            fprintf(stdout,
                "Read: '%s' (%d)\n",msg,c);
        };
    };
};
```

```
$ ./output.out | ./input.out
```

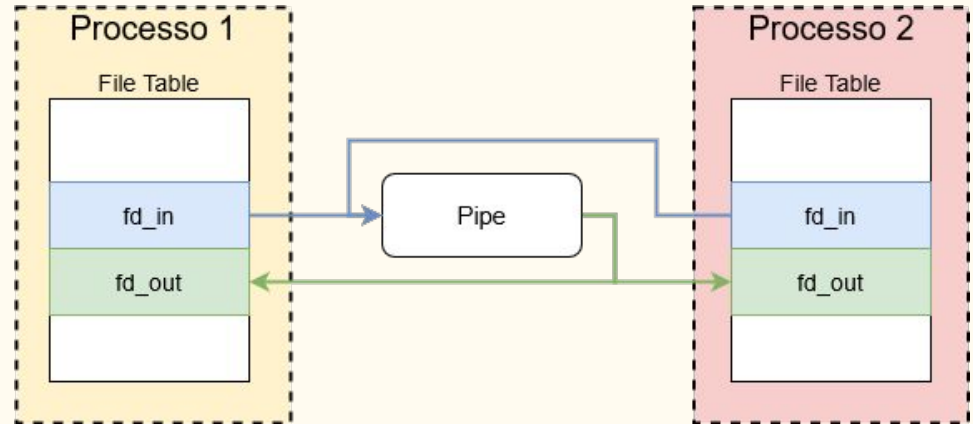
Pipe anonime

—

Pipe anonime

Le pipe anonime, come quelle usate su shell, ‘uniscono’ due processi aventi un antenato comune (oppure tra padre-figlio). Il collegamento è unidirezionale ed avviene utilizzando un buffer di dimensione finita.

Per interagire con il buffer (la pipe) si usano due file descriptors: uno per il lato in scrittura ed uno per il lato in lettura. Visto che i processi figli ereditano i file descriptors, questo consente la comunicazione tra i processi.



Creazione pipe

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

L'array passato come argomento sarà popolato con due nuovi file descriptors (i primi due disponibili). Il primo elemento sarà usato per la lettura della pipe, il secondo per la scrittura.

Normalmente un processo usa il lato in lettura, ed un altro processo usa il lato in scrittura (unidirezionale!)

Esempio creazione pipe

```
#include <stdio.h>                                     //pipe.c
#include <unistd.h>
int main(){
    int fd[2];
    pipe(fd) //Create unnamed pipe using 2 file descriptors
    int isChild = !fork();
    if(isChild){
        //Use one side of the pipe
    }else{
        //use the other side of the pipe
    }
}
```

Esempio limite creazione pipe

```
#include <stdio.h> //pipe.c
#include <unistd.h>
int main(){
    int fd[2], cnt = 0;
    while(pipe(fd) == 0){ //Create unnamed pipe using 2 file descriptors
        cnt++;
        printf("%d,%d,",fd[0],fd[1]);
    }
    printf("\n Opened %d pipes, then error\n",cnt);
    int op = open("/tmp/tmp.txt",O_CREAT|O_RDWR,S_IRUSR|S_IWUSR);
    printf("File opened with fd %d\n",op);
}
```

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

Legge al massimo `num` bytes di dati dalla pipe identificata dal file descriptor `fd[0]`, salvandoli in `data`.

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 lato di scrittura è stato chiuso (da ogni processo), con il buffer vuoto restituisce 0, altrimenti restituisce il numero di bytes letti.
- 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 recuperati solo quelli presenti

Esempio lettura pipe

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

Scrittura: `int write(int fd[1], char * data, int num)`

La scrittura della pipe tramite il comando `write` restituisce il numero di bytes scritti. Tuttavia, se il lato in lettura è stato chiuso viene inviato un segnale `SIGPIPE` allo scrittore (default handler quit) e la chiamata restituisce `-1`.

In caso di scrittura, se vengono scritti meno bytes di quelli che ci possono stare (`PIPE_BUF`) la scrittura è “atomica” (tutto assieme), in caso contrario non c’è garanzia di atomicità e la scrittura sarà bloccata (in attesa che il buffer venga svuotato) o fallirà se il flag `O_NONBLOCK` viene usato.

Per modificare le proprietà di una pipe, possiamo usare la system call `fcntl`, la quale manipola i file descriptors.

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

Esempio scrittura pipe

```
#include <unistd.h> <stdio.h> <signal.h> <errno.h> <stdlib.h> //writePipe.c
void handler(int signo){
    printf("SIGPIPE received\n");    perror("Error in handler");
}
int main(void){
    signal(SIGPIPE,handler);
    int fd[2]; char buf[50];
    int esito = pipe(fd); //Create unnamed pipe
    close(fd[0]); //Close read side
    printf("Attempting write\n");
    int numOfWritten = write(fd[1],"writing",8);
    printf("I've written %d bytes\n",numOfWritten);
    perror("Error after write");
}
```

Esempio 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 di comunicare.

Esempio unidirezionale

```
#include <stdio.h> <unistd.h> <sys/wait.h>           //uni.c
int main(){
    int fd[2]; char buf[50];
    pipe(fd); //Create unnamed pipe
    int p2 = !fork();
    if(p2){
        close(fd[1]);
        int r = read(fd[0],&buf,50); //Read from pipe
        close(fd[0]); printf("Buf: '%s'\n",buf);
    }else{
        close(fd[0]);
        write(fd[1],"writing",8); // Write to pipe
        close(fd[1]);
    }
    while(wait(NULL)>0);
}
```

Esempio 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.

Esempio bidirezionale

```
#include <stdio.h> <unistd.h> <sys/wait.h>    #define READ 0 #define WRITE 1    //bi.c
int 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);
}
```

Esercizi

- Impostare una comunicazione bidirezionale tra due processi con due livelli di complessità:
 - Alternando almeno due scambi ($P1 \rightarrow P2$, $P2 \rightarrow P1$, $P1 \rightarrow P2$, $P2 \rightarrow P1$)
 - Estendendo il caso a mo' di “ping-pong”, fino a un messaggio convenzionale di “fine comunicazione”

Gestire la comunicazione

Per gestire comunicazioni complesse c'è bisogno di definire un “protocollo”. Esempio:

- Messaggi di lunghezza fissa (magari inviata prima del messaggio)
- Marcatore di fine messaggio (per esempio con carattere NULL o newline)

Più in generale occorre definire la sequenza di messaggi attesi.

Esempio: reindirige lo stdout di cmd1 sullo stdin di cmd2

```
#include <stdio.h> <unistd.h> #define READ 0 #define WRITE 1 //redirect.c
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("error"); // 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("error"); // Should never execute
    }
}
```

Pipe con nome/FIFO

FIFO

Le pipe con nome, o FIFO, sono delle pipe che 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, dall'esistenza del file stesso.

Le FIFO sono interpretate come dei file, perciò si possono usare le funzioni di scrittura/lettura dei file viste nelle scorse lezioni. Restano però delle pipe, con i loro vincoli e le loro capacità. NB: non sono dei file effettivi, quindi lseek non funziona, il loro contenuto è sempre vuoto e non vi ci si può scrivere con un editor!

Normalmente aprire una FIFO blocca il processo finchè anche l'altro lato non è stato aperto. Le differenze tra pipe anonime e FIFO sono solo nella loro creazione e gestione.

Creare una FIFO

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

Crea una FIFO (un file speciale), solo se non esiste già, con il nome **pathname**. Il parametro **mode** può definire i privilegi del file nella solita maniera.

Una volta creata, per essere usata la fifo deve essere aperta con `open()`, usando le flag `O_RDONLY` o `O_WRONLY` per gestire il lato di apertura.

```
#include <sys/stat.h><sys/types.h><unistd.h><fcntl.h><stdio.h> //fifo.c
int main(void){
    char * fifoName = "/tmp/fifo1";
    mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if it doesn't exist
    perror("Created?");
    int fd = open(fifoName,O_RDONLY);
}
```

Comunicazione bloccata

```
#include <sys/stat.h><sys/types.h><unistd.h><fcntl.h><stdio.h> //fifo.c
int main(void){
    char * fifoName = "/tmp/fifo1";
    mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if it doesn't exist
    perror("Created?");
    if (fork() == 0){ //Child
        open(fifoName,O_RDONLY); //Open READ side of pipe...stuck!
        printf("Open read\n");
    }else{
        sleep(3);
        open(fifoName,O_WRONLY); //Open WRITE side of pipe
        printf("Open write\n");
    }
}
```

Esempio comunicazione: writer (1/2)

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

//fifoWriter.c
int 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 it 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);
}
```

Esempio comunicazione: reader (2/2)

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

//fifoReader.c
int 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);
}
```

Comunicazione sul terminale

È possibile usare le FIFO da terminale, leggendo e scrivendo dati tramite gli operatori di redirectione.

```
echo "message for pipe" > /path/nameOfPipe
```

```
cat /path/nameOfPipe
```

NB: non si possono scrivere dati usando editor di testo! Una volta consumati i dati, questi non sono più presenti sulla fifo.

Pipe anonime vs FIFO

	pipe	FIFO
Rappresentazione	Buffer	Buffer
Riferimento	anonimo	File
Accesso	2 File descriptors	1 File descriptor
Persistenza	Eliminata alla terminazione di tutti i processi	Esiste finchè esiste il file
Vincoli accesso	Antenato comune	Permessi sul file
Creazione	pipe()	mkfifo()
Apertura	Automatica con pipe()	open()
Max bytes per atomicità	PIPE_BUF = 4096 on Linux, minimo 512 Bytes POSIX	

Esercizi

- Impostare una comunicazione bidirezionale tra due processi con due livelli di complessità:
 - Alterna due scambi ($P1 \rightarrow P2$, $P2 \rightarrow P1$, $P1 \rightarrow P2$, $P2 \rightarrow P1$)
 - Mantenendo una comunicazione fino alla ricezione di un carattere di terminazione (da decidere)
- Creare un programma che prenda come argomento 'n' il numero di figli da generare. Ogni figlio creato comunicherà al genitore (tramite pipe) un numero casuale e il genitore calcolerà la somma di tutti questi numeri, inviandola a stdout.

CONCLUSIONI

PIPE e FIFO (“named pipes”) sono sistemi di comunicazione tra processi (“parenti”, tipicamente padre-figlio, nel primo caso e in generale nel secondo caso) che consentono scambi di informazioni (messaggi) e sincronizzazione (grazie al fatto di poter essere “bloccanti”).