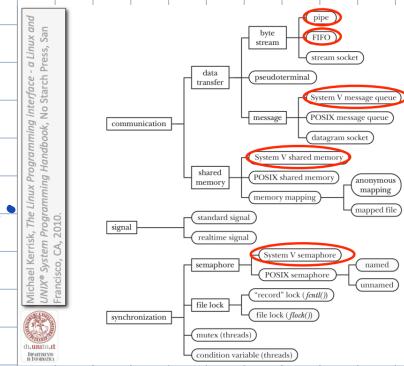


IPC FACILITIES

Strumenti messi a disposizione per:

- Comunicazione fra processi
- Sincronizzazione delle azioni dei processi.
- Segnali per comunicazione di eventi



IPC FACILITY: COMMUNICATION ➤

La comunicazione può avvenire tramite:

- file sharing: veloce ma necessita meccanismo di sincronizz.
- Data transfer:

Suddivise in

- Byte Stream (Pipe, FIFO e datagram sockets)

Le letture possono prendere un numero arbitrario di byte.

- Message (code di messaggi e sockets)

Le letture devono prendere l'intero messaggio, così come scritto dal consumer.

Si nota che le read sono distruttive e c'è la sync automatica. (Nel file sharing è il contrario)

Pipe / FIFO ➤

È un IPC byte stream che consente di trasferire l'output del 1° nel 2°.

Pipes

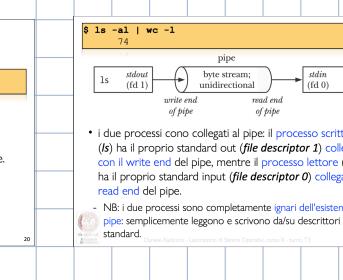
```
$ ls -al | wc -l
```

74

- per eseguire questi comandi, la shell crea due processi, che eseguono ls e wc, rispettivamente.
- questo è fatto utilizzando la fork() e la exec().



Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3



i pipe sono stream di byte

- un pipe è uno stream di byte: usando un pipe, non facciamo riferimento ad alcun concetto di messaggio o di delimitazione di messaggio.
- il processo che legge da un pipe può leggere blocchi di qualsiasi dimensione, indipendentemente dalla dimensione dei blocchi scritti dal processo che scrive.
- i dati passano attraverso il pipe in sequenza: i byte sono letti nello stesso ordine in cui sono stati scritti. non è possibile accedere ai dati in maniera casuale utilizzando seek().

Capacità limitata

- Un pipe è semplicemente un buffer mantenuto in memoria.
- Questo buffer ha una capacità massima. Una volta che un pipe è pieno, ulteriori tentativi di scrittura si bloccano finché il lettore rimuove alcuni dati dal pipe.
- In generale, un'applicazione non ha bisogno di conoscere la capacità del pipe.
- Se vogliamo evitare ai processi scrittori di restare bloccati, è necessario che i processi che leggono dai pipe siano progettati in modo da leggere i dati appena sono disponibili.

Capacità limitata

- In teoria, non ci sono motivi per cui un pipe non debba utilizzare capacità minime, fino al buffer costituito da un solo byte.
- La ragione per utilizzare buffer di dimensioni maggiori è l'efficienza: ogni volta che uno scrittore riempie il pipe, il kernel deve eseguire un context switch per consentire al lettore di essere 'scheduled' per prelevare qualche dato dal pipe.
- L'utilizzo di un buffer di dimensione maggiore comporta la riduzione del numero di context switch.

lettura da pipe

- I tentativi di leggere da un pipe vuoto restano bloccati finché almeno un byte non è stato scritto sul pipe.
- Se il write end di un pipe viene chiuso, un processo che legge dal pipe riceverà il codice *end-of-file* (i.e., read() restituirà 0), una volta che avrà letto tutti i dati presenti nel pipe.
- I pipe sono unidirezionali. I dati possono viaggiare solo in una direzione.

Un'estremità (end) del pipe è utilizzato in scrittura, e l'altra in lettura.

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

USAGE

unistd.h

Si crea con la funzione `pipe(arr[2])`

Se va a buon fine, alloca nell'array
2 file descriptors:

- `arr[0]`: per lettura
- `arr[1]`: per scrittura

COMUNICAZIONE PADRE-FIGLIO

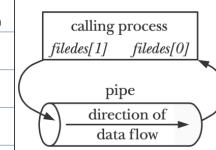
Di solito si:

1. crea il pipe
2. fa la fork. Il figlio avrà una copia dei file descriptors.
3. A seconda di chi è il consumer e chi è il producer, padre e figlio chiudono il canale non usato

Questo perché la comunicazione è unidirezionale per convenzione e necessaria per evitare attese infinite due a file descriptor ancora aperti.

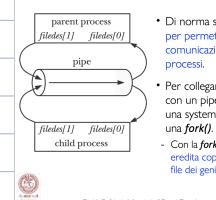
```
#include <unistd.h>
int pipe(int filedes[2]);
```

Returns 0 on success, or -1 on error



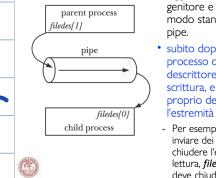
- la system call `pipe()` crea un nuovo pipe; se va a buon fine, la chiamata alloca un array (`filedes`) contenente due descrittori di file aperti.
- Un estremo è aperto in lettura (`filedes[0]`) e uno in scrittura (`filedes[1]`).
- Come con qualsiasi descrittore di file, possiamo utilizzare le system call `read()` e `write()` per eseguire le operazioni di I/O sul pipe.

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3



- Di norma si utilizzano i pipe per permettere la comunicazione fra due processi.
- Per collegare due processi con un pipe, eseguiamo prima una system call `pipe()` e poi una `fork()`.
- Con la `fork()`, il processo figlio eredita copia dei descrittori di file del genitore.

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3



- tecnicamente sarebbe possibile leggere e scrivere sul pipe per il genitore e il figlio, ma non è il modo standard di utilizzare i pipe.
- subito dopo la `fork()`, un processo chiude il proprio descrittore per l'estremità in scrittura, e l'altro chiude il proprio descrittore per l'estremità in lettura.
- Per esempio, se il genitore deve inviare dati al figlio, deve chiudere l'estremità aperta in lettura, `filedes[0]`, mentre il figlio deve chiudere `filedes[1]`.

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

```
int filedes[2];
if (pipe(filedes) == -1) // creazione pipe
    ddErrExit("pipe");
switch (fork()) {
    case -1:
        ddErrExit("fork");
    case 0: // ----- Child
        if (close(filedes[1]) == -1) // --- chiude il write end
            ddErrExit("close");
        // --- il figlio compie qualche operazione ---
        break;
    default: // padre
        if (close(filedes[0]) == -1) // --- chiude il read end
            ddErrExit("close");
        // --- il padre compie qualche operazione ---
        break;
}
```

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

Chiusura dei descrittori inutilizzati (lettore)

- I file descriptors inutilizzati in lettura e in scrittura devono essere chiusi.
- Il processo che legge dal pipe chiude il proprio `write` descriptor, così che quando l'altro processo completa il proprio output e chiude il proprio descriptor `write`, il lettore riceve un carattere terminatore, `end-of-file`.
- Se invece il processo lettore non chiude la propria estremità aperta in scrittura, dopo che l'altro processo avrà chiuso il proprio descriptor `write`, il lettore non riceverà l'`end-of-file` neppure dopo avere letto tutti i dati dal pipe.
- In questo caso, una `read()` si bloccerebbe in attesa di dati (che sappiamo non arriveranno) perché il kernel sa che c'è ancora almeno un descrittore di file aperto per il pipe.



Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

32

Chiusura dei descrittori inutilizzati (scrittore)

- Il processo scrittore chiude l'estremità aperta in lettura del pipe per una ragione diversa. Quando un processo tenta di scrivere su un pipe per il quale nessun processo ha un descrittore aperto in lettura, il kernel invia il segnale `SIGPIPE` al processo scrittore.
- Per default, tale segnale uccide il processo.
- Un processo può organizzarsi per intercettare o ignorare tale segnale; in questo caso la `write()` sul pipe fallisce con un errore `EPIPE` (broken pipe).
- Ricevere il segnale `SIGPIPE` o ricevere l'errore `EPIPE` è un'indicazione utile in merito allo `status` del pipe, ed è la ragione per cui i descrittori aperti in lettura dovrebbero essere chiusi.



Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

33

```
int main(int argc, char** argv) {
    int n;
    int fd[2];
    pid_t pid;
    char line[MAXLINE];
    if (pipe(fd) < 0)
        ddErrExit("pipe error");
    if ((pid = fork()) < 0) {
        ddErrExit("fork error");
    } else if (pid == 0) { // ----- figlio -----
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else { // ----- padre -----
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(EXIT_SUCCESS);
}
```

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

34

```
...
int main(int argc, char *argv[]) {
    int pdf[2]; /* pipe file descriptors */
    char buf[BUFSIZ];
    ssize_t numRead;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        /* stampa formato invocazione */
        ;
    if (pipe(pdf) == -1) /* creo il pipe */
        /* gestione errore */
    switch (fork()) {
        case -1:
            /* gestione errore */
            ...
pipe1.c
```

35

```
case 0: /* ----- figlio - legge dal pipe ----- */
    if (close(pdf[1]) == -1) /* chiusura write end */
        /* gestione errore */
    for (;;) { /* legge dal pipe, e scrive su stdout */
        numRead = read(pdf[0], buf, BUFSIZ);
        if (numRead == -1)
            /* gestione errore */
        if (numRead == 0)
            break; /* End-of-file */
        if (write(STDOUT_FILENO, buf, numRead) != numRead)
            /* gestione errore */
    }
    write(STDOUT_FILENO, "\n", 1);
    if (close(pdf[0]) == -1)
        /* gestione errore */
    exit(EXIT_SUCCESS);
}
pipe1.c
```

36

```
...
default: /* ----- padre - scrive sul pipe ----- */
    if (close(pdf[0]) == -1) /* chiusura del read end */
        /* gestione errore */
    if(write(pdf[1], argv[1], strlen(argv[1]))!=strlen(argv[1]))
        /* gestione errore */
    if (close(pdf[1]) == -1) /* il figlio riceverà EOF */
        /* gestione errore */
    wait(NULL); /* attesa della terminazione del figlio */
    exit(EXIT_SUCCESS);
}
// end switch
}
pipe1.c
```

37

POPE

Concludendo che crea un PIPE, fa la fork e poi nel figlio esegue il command nella coro specificato.

• command: Il comando da eseguire

• mode: Indica se il chiamante deve

leggere (r) o scrivere (w) nel pipe

Una volta finito, si usa `pclose()` per chiudere il PIPE.

```
#include <stdio.h>
int pclose(FILE *stream);
Returns termination status of child process,
or -1 on error
```

• Completate le operazioni di I/O, si utilizza la funzione `pclose()` per chiudere il pipe ed attendere che la shell figlia termini.

- In caso di successo, la `pclose()` ottiene lo status di terminazione della shell figlia (cioè lo status di terminazione dell'ultimo comando eseguito dalla shell, a meno che la shell sia uccisa da un segnale).

Daniele Radiconi - Laboratorio di Sistemi Operativi, corso B - turno T3

42

```
#include <stdio.h>
FILE *popen(const char *command, const char *mode);
Returns file stream, or NULL on error
```

parent → tp → cmdstring (child) → stdout

fp = fopen(cmdstring, "r")

Daniele Radiconi - Laboratorio di Sistemi Operativi, corso B - turno T3

43

convenience vs. efficiency

• L'utilizzo della `popen()` garantisce **semplicità d'uso**.

- La `popen()` crea il pipe, esegue la duplicazione dei descrittori, chiude i descrittori inutilizzati, e gestisce tutti i dettagli della `fork()` e della `exec()` per nostro conto.

• Tale semplicità d'uso ha un costo in termini di **efficienza**. Vengono creati almeno due processi in più: uno per la shell e uno o più per i comandi eseguiti dalla shell.

- Come per `system()`, `popen()` non dovrebbe essere utilizzata da programmi privilegiati.

Daniele Radiconi - Laboratorio di Sistemi Operativi, corso B - turno T3

43

```
#include <stdio.h>
#define PATH_MAX 1024
```

```
int main(int argc, char** argv) {
    FILE *fp;
    int status;
    char path[PATH_MAX];
```

```
    fp = fopen("ls", "r");
    if (fp == NULL)
        ; // gestione errore
```

```
    while (fgets(path, PATH_MAX, fp) != NULL)
        printf("%s\n", path);
```

```
    status = pclose(fp);
    if (status == -1) {
        ; // gestione errore
    } else {
        ; // analisi dell'exit status
    }
    return(0);
}
```

Daniele Radiconi - Laboratorio di Sistemi Operativi, corso B - turno T3

44

FIFO > o named pipes

Sono varianti delle pipe con nome nel FS ed aperto come un file, consentendo la comunicazione di proc. non imparentati

FIFO SU SHELL

Si creano con `mknod` in cui si possono specificare nome e permessi

FIFO SU C

Si creano con `mknod()`; specificando nome e permessi in OR

Di default, producer e consumer sono sincronizzati.

sincronizzazione con FIFO

• L'utilizzo di un FIFO serve ad avere un processo lettore e uno scrittore alle due estremità del FIFO.

- di default, l'apertura di un FIFO in lettura (flag `O_RDONLY` della `open()`) si blocca finché un altro processo apre il FIFO in scrittura (flag `O_WRONLY` della `open()`).

- per contro, l'apertura del FIFO in scrittura si blocca finché un altro processo non apre la FIFO in lettura.

- In altri termini, l'apertura di un FIFO **sincronizza i processi lettori e scrittori**.

- Se l'altra estremità del FIFO è già aperta (per esempio nel caso in cui una coppia di processi hanno già aperto ciascuna estremità del FIFO), la `open()` va immediatamente a buon fine.

Daniele Radiconi - Laboratorio di Sistemi Operativi, corso B - turno T3

```
#define FIFO_SIZE 128
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stropts.h>
char * myfifo = "/tmp/myfifo";
mknod(myfifo, 0666);
char arr1[FIFO_SIZE], arr2[FIFO_SIZE];

while (1) {
    int fd1;
    fd = open(myfifo, O_RDONLY); // apertura FIFO in lettura
    fget(fd, arr1, FIFO_SIZE, stdin); // lettura da stdin su arr1
    write(fd, arr2, strlen(arr2)+1); // scrittura e chiusura FIFO
    close(fd);

    fd = open(myfifo, O_WRONLY); // apertura FIFO in scrittura
    read(fd, arr1, sizeof(arr1)); // lettura da FIFO su arr1
    printf("User1: %s\n", arr1);
    close(fd); // chiusura FIFO
}
return 0;
}
```

Daniele Radiconi - Laboratorio di Sistemi Operativi, corso B - turno T3

51

```
#define FIFO_SIZE 128
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stropts.h>
char * myfifo = "/tmp/myfifo";
mknod(myfifo, 0666);
char arr1[FIFO_SIZE], arr2[FIFO_SIZE];

while (1) {
    int fd1;
    fd1 = open(myfifo, O_RDONLY); // apertura FIFO in lettura
    read(fd1, arr1, FIFO_SIZE); // lettura da FIFO su arr1 e chiusura
    close(fd1);
    printf("User2: %s\n", arr1);

    fd1 = open(myfifo, O_WRONLY); // apertura FIFO in scrittura
    fget(fd1, arr2, FIFO_SIZE, stdin); // lettura da stdin su arr2
    write(fd1, arr2, strlen(arr2)+1); // scrittura su FIFO
    close(fd1);
}
return 0;
}
```

Daniele Radiconi - Laboratorio di Sistemi Operativi, corso B - turno T3

52

TEE

È un comando usato per mandare 1 copia nello `stdout` ed una copia nel file (e quindi FIFO) specificata

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

```
FILE *popen(const char *command, const char *mode);
```

Returns file stream, or NULL on error

• Il valore di `mode` determina se lo standard output del comando eseguito è connesso all'estremità del pipe aperta in scrittura, o se il suo standard input è connesso all'estremità del pipe aperta in lettura.

- In caso di successo, `popen()` restituisce un file stream pointer che può essere gestito con le funzioni della libreria `stdio`.

- In caso di fallimento (e.g., il mode non è `r` o `w`, la creazione del pipe fallisce, o fallisce la `fork()` per creare il figlio), allora la `popen()` restituisce `NULL` e assegna `errno` indicando la causa dell'errore.

Daniele Radiconi - Laboratorio di Sistemi Operativi, corso B - turno T3

39



```

$ mkfifo myfifo
$ ls -l
total 24
-rw-r--r-- 1 radicion staff 0 Nov 11 18:00 file1
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file2
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file3
prw-r--r-- 1 radicion staff 0 Nov 11 18:00 myfifo
$ wc -l < myfifo &
[1]+ 24656  tee myfifo > sort -k5n
5
prw-r--r-- 1 radicion staff 0 Nov 11 18:01 myfifo
total 24
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file1
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file2
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file3
[1]+ Done
wc -l < myfifo

```

Diagram illustrating a pipeline:

```

graph LR
    A[ls] --> B[tee]
    B --> C[FIFO]
    C --> D[wc]

```

Detailed description: This diagram shows a command-line pipeline. It starts with 'ls' which lists files. The output goes to 'tee', which creates a FIFO named 'myfifo'. The output then goes to 'wc -l < myfifo &', which counts the lines in the FIFO. The FIFO is shown in yellow.

Daniele Radicchi - Laboratorio di Sistemi Operativi, corso B - turno T3

56

```

$ mkfifo myfifo
$ ls -l
total 24
-rw-r--r-- 1 radicion staff 0 Nov 11 18:00 file1
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file2
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file3
prw-r--r-- 1 radicion staff 0 Nov 11 18:00 myfifo
$ wc -l < myfifo &
[1] 24657
$ ls -l > tee myfifo | sort -k5n
5
prw-r--r-- 1 radicion staff 0 Nov 11 18:01 myfifo
total 24
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file1
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file2
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file3
[1]+ Done
wc -l < myfifo

```

Diagram illustrating a pipeline:

```

graph LR
    A[ls] --> B[tee]
    B --> C[FIFO]
    C --> D[sort]
    D --> E[wc]

```

Detailed description: This diagram shows a command-line pipeline. It starts with 'ls' which lists files. The output goes to 'tee', which creates a FIFO named 'myfifo'. The output then goes to 'sort -k5n', which sorts the files by the fifth field. Finally, the sorted output goes to 'wc -l < myfifo &', which counts the lines in the FIFO. The FIFO is shown in yellow.

Daniele Radicchi - Laboratorio di Sistemi Operativi, corso B - turno T3

57

```

$ mkfifo myfifo
$ ls -l
total 24
-rw-r--r-- 1 radicion staff 0 Nov 11 18:00 file1
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file2
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file3
prw-r--r-- 1 radicion staff 0 Nov 11 18:00 myfifo
$ wc -l < myfifo &
[1] 24658
$ ls -l > tee myfifo | sort -k5n
5
prw-r--r-- 1 radicion staff 0 Nov 11 18:01 myfifo
total 24
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file1
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file2
-rw-r--r-- 1 radicion staff 1 Nov 11 18:00 file3
[1]+ Done
wc -l < myfifo

```

Diagram illustrating a pipeline:

```

graph LR
    A[ls] --> B[tee]
    B --> C[FIFO]
    C --> D[sort]
    D --> E[wc]

```

Detailed description: This diagram shows a command-line pipeline. It starts with 'ls' which lists files. The output goes to 'tee', which creates a FIFO named 'myfifo'. The output then goes to 'sort -k5n', which sorts the files by the fifth field. Finally, the sorted output goes to 'wc -l < myfifo &', which counts the lines in the FIFO. The FIFO is shown in yellow.

Daniele Radicchi - Laboratorio di Sistemi Operativi, corso B - turno T3

58