Laboratorio di sistemi operativi – T4

Pipe e FIFO

Il programma

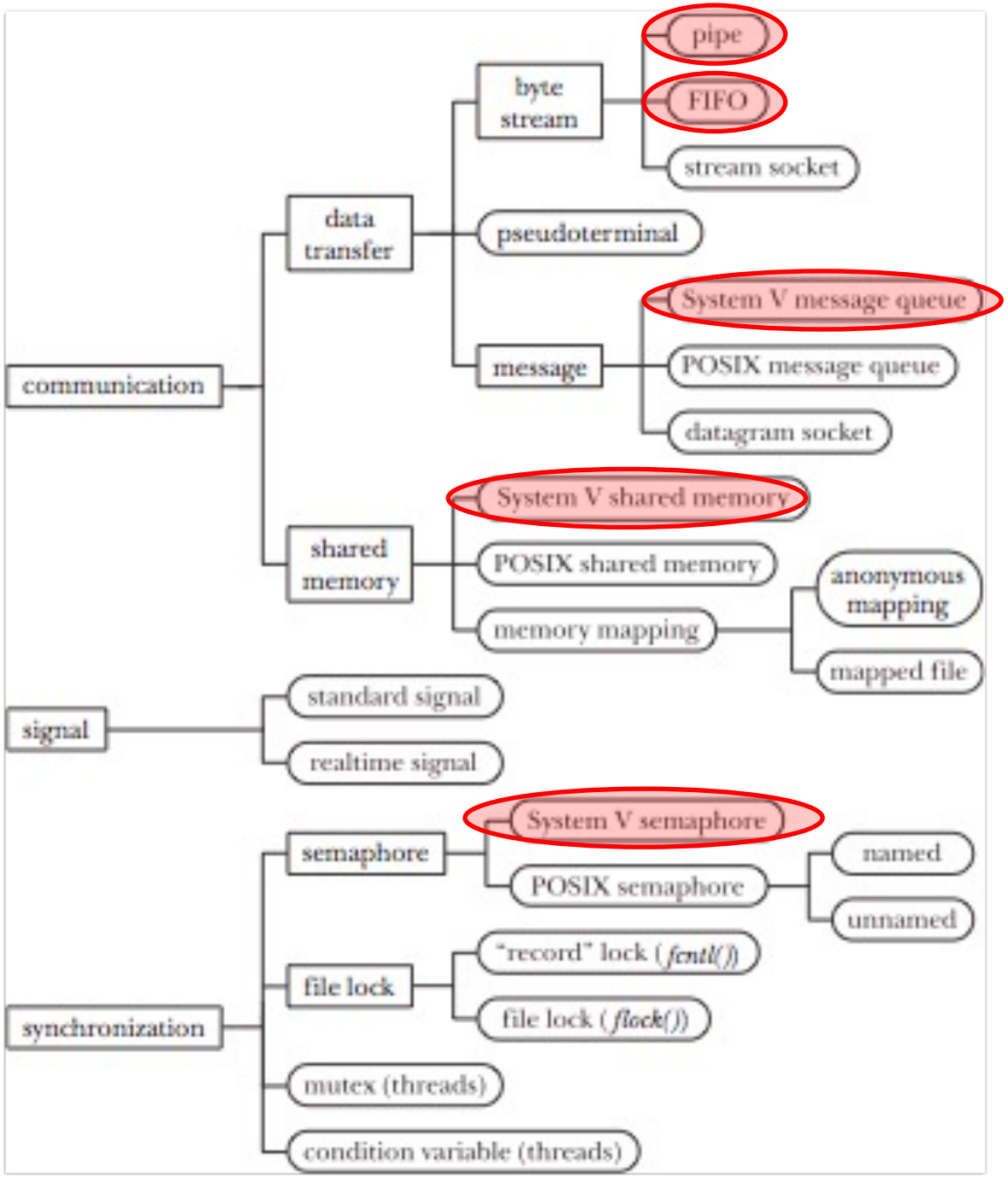
1. Introduzione a UNIX
2. Nozioni integrative del linguaggio C
3. controllo dei processi;
4. **pipe e fifo;**
5. code di messaggi;
6. memoria condivisa;
7. semafori;
8. segnali;
9. introduzione alla programmazione bash

Una breve panoramica su IPC

IPC facilities

- i vari strumenti che UNIX offre per la comunicazione e la sincronizzazione possono essere suddivisi in tre ampie categorie funzionali:
  - **Comunicazione**: facilities utilizzate per lo scambio di dati fra processi.
  - **Sincronizzazione**: facilities utilizzate per sincronizzare le azioni dei processi.
  - **Segnali**: sebbene i segnali siano nati prevalentemente con altri fini, in alcune circostanze possono essere utilizzati come strumenti di sincronizzazione.

Che cosa studieremo durante il corso?



Communication facilities

- **Data-transfer facilities**: l'elemento fondamentale che distingue questi strumenti è la nozione di scrittura e lettura.
  - per comunicare, un processo scrive i dati alla facility per l'IPC e un altro processo legge questi dati.
  - questi strumenti richiedono due trasferimenti dati fra la memoria utente e quella del kernel: un trasferimento durante la scrittura e un trasferimento durante la lettura.

Communication facilities

- **Memoria condivisa**: la memoria condivisa permette ai processi di scambiarsi le informazioni mettendole in una regione della memoria condivisa fra i processi.
  - un processo può rendere i dati disponibili per gli altri processi collocandoli in una regione di memoria condivisa.
  - poiché la comunicazione non richiede system call o trasferimento di dati fra la memoria utente e quella del kernel, la memoria condivisa è uno strumento di comunicazione molto veloce.

Data-transfer facilities

- Le data-transfer facilities possono essere ulteriormente suddivise nelle seguenti sottocategorie:
  - **Byte stream**: i dati scambiati per mezzo di pipe, FIFOs, e datagram sockets sono uno stream di byte.
    - ogni operazione di lettura può leggere un numero arbitrario di byte, senza considerare la dimensione dei blocchi scritti dallo scrivente.
    - questo modello riflette il tradizionale modello di UNIX in cui il file è visto come una sequenza di byte.

Data-transfer facilities

- Le data-transfer facilities possono essere ulteriormente suddivise nelle seguenti sottocategorie:
  - **Messaggio**: i dati scambiati con le code di messaggi, e i socket hanno la forma di messaggi delimitati.
    - ogni operazione di lettura legge un intero messaggio, così come scritto dal processo scrivente.
    - non è possibile leggere parzialmente un messaggio, lasciando il resto sulla IPC facility, e non è possibile leggere molteplici messaggi con una singola operazione di lettura.

Data-transfer facilities

- Le data-transfer facilities sono distinte dalla memoria condivisa per alcune caratteristiche generali:
  - Sebbene le data-transfer facilities possano avere molteplici lettori, le operazioni di lettura sono distruttive. Una operazione read consuma i dati, e i dati non sono più disponibili per altri processi.
  - La sincronizzazione fra processo lettore e scrittore è automatica. Se un lettore tenta di consumare dati da una facility che attualmente non ne contiene, (di default) l'operazione di lettura si bloccherà finché un processo non avrà scritto dei dati su quella facility.

Memoria condivisa

- Sebbene la memoria condivisa fornisca una comunicazione veloce, questo vantaggio è bilanciato dalla necessità di sincronizzare le operazioni sulla memoria condivisa.
  - per esempio, un processo non dovrebbe cercare di accedere a una struttura dati presente nella memoria condivisa mentre un altro processo la sta modificando.
- il semaforo è lo strumento di sincronizzazione abitualmente utilizzato con la memoria condivisa.
  - i dati presenti nella memoria condivisa sono visibili a tutti i processi che condividono quel segmento di memoria, diversamente dalla semantica distruttiva delle operazioni di lettura messe a disposizione dalle data-transfer facilities

Synchronization Facilities: semafori

- Un semaforo è un intero mantenuto dal kernel, il cui valore non può divenire minore di 0.
  - Un processo può decrementare o incrementare il valore di un semaforo. Se viene fatto un tentativo di decrementare il valore di un semaforo sotto lo 0, il kernel blocca l'operazione finché il valore del semaforo aumenta a un livello che permette di eseguire l'operazione.
  - In alternativa, il processo può richiedere una nonblocking operation; in questo caso, invece di bloccarlo, il kernel provoca una restituzione immediata con un errore che indica che non è stato possibile eseguire l’operazione immediatamente.

Pipes

Pipe

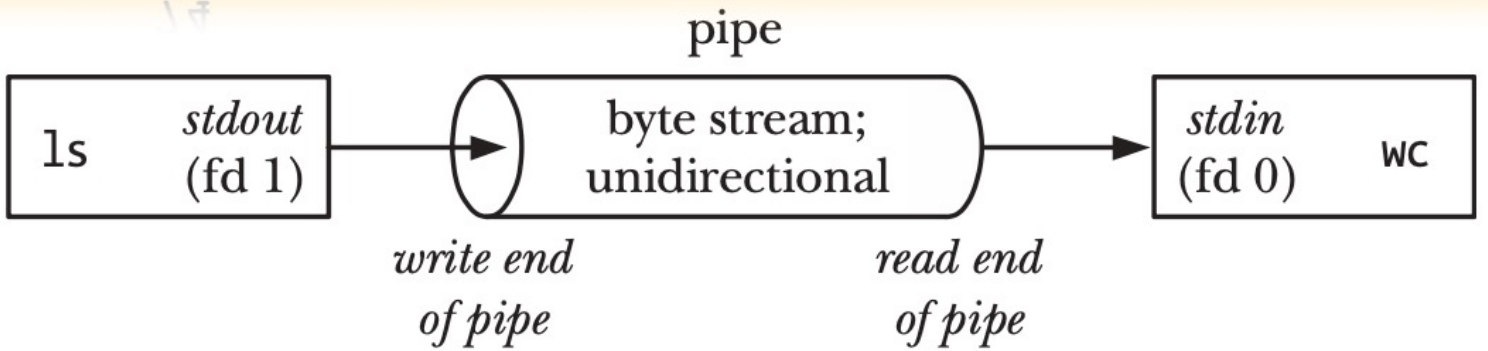
- Flusso di byte uni-direzionale
- le pipe forniscono una soluzione a un problema frequente: avendo creato due processi per eseguire programmi diversi (comandi), come può la shell fare in modo che l'output prodotto da un processo sia utilizzato come input per l'altro processo?
- Le FIFO sono una variazione del concetto di pipe. La differenza sostanziale è che le FIFO possono essere utilizzate per la comunicazione fra processi qualsiasi.

Pipe

```
$ ls -al | wc -l
      74
```

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

```
$ ls -al | wc -l
      74
```



- i due processi cono collegati alla pipe: il processo scrittore (ls) ha il proprio standard out (file descriptor 1) collegato con il write end del pipe, mentre il processo lettore (wc) ha il proprio standard input (file descriptor 0) collegato al read end del pipe.
  - NB: i due processi sono completamente ignari dell'esistenza del pipe: semplicemente leggono e scrivono da/su descrittori di file standard.

Le pipe sono flussi di byte

- una pipe è un flusso di byte: usando una pipe, non facciamo riferimento ad alcun concetto di messaggio o di delimitazione di messaggio.
  - il processo che legge da una pipe può leggere blocchi di qualsiasi dimensione, indipendentemente dalla dimensione dei blocchi scritti dal processo che scrive.
- i dati passano attraverso la pipe in sequenza: i byte sono letti nello stesso ordine in cui sono stati scritti. non è possibile accedere ai dati in maniera casuale (ad es. utilizzando `lseek()` si ottiene un errore con `errno=EPIPE`).

Lettura da una pipe

- I tentativi di leggere da una pipe vuota restano bloccati finché almeno un byte non è stato scritto sulla pipe.
- Se il write end di un pipe viene chiuso, un processo che legge dalla pipe riceverà il codice end-of-file (i.e., `read()` restituirà 0) una volta che avrà letto tutti i dati presenti nella pipe.
- Le pipe sono unidirezionali. I dati possono viaggiare solo in una direzione. Un'estremità (end) del pipe è utilizzato in scrittura, e l'altro in lettura.

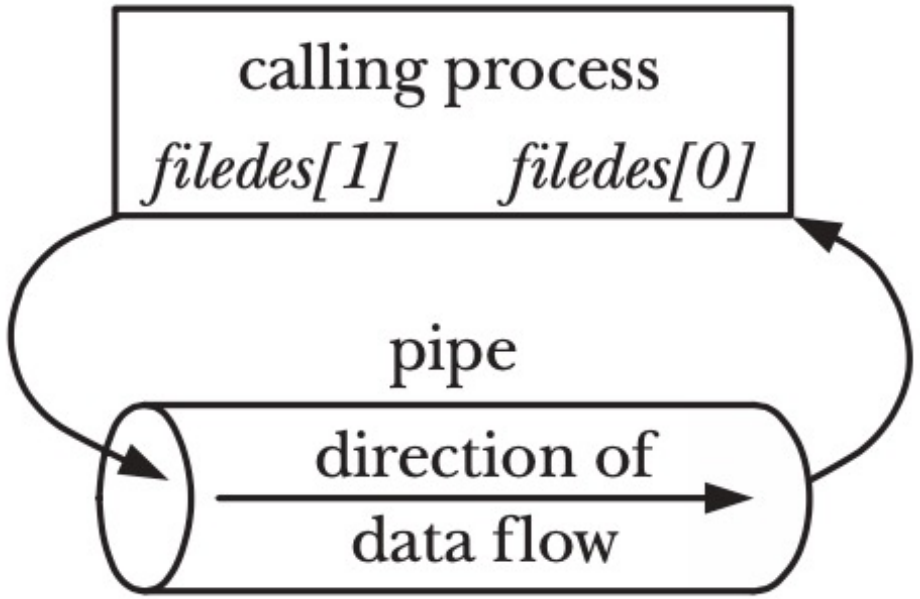
Capacità limitata

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

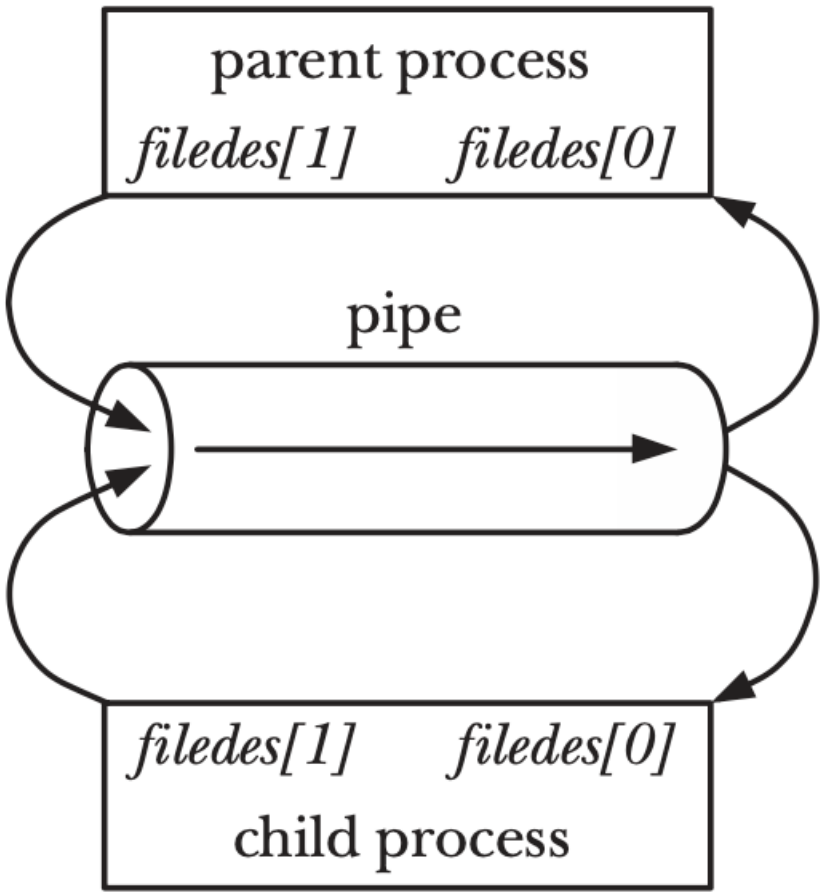
Capacità limitata

- In teoria, non ci sono motivi per cui una 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 la pipe, il kernel deve eseguire un context switch per consentire al lettore di essere schedulato in modo che possa prelevare qualche dato dalla pipe.
  - L'utilizzo di un buffer di dimensione maggiore comporta la riduzione del numero di context switch.

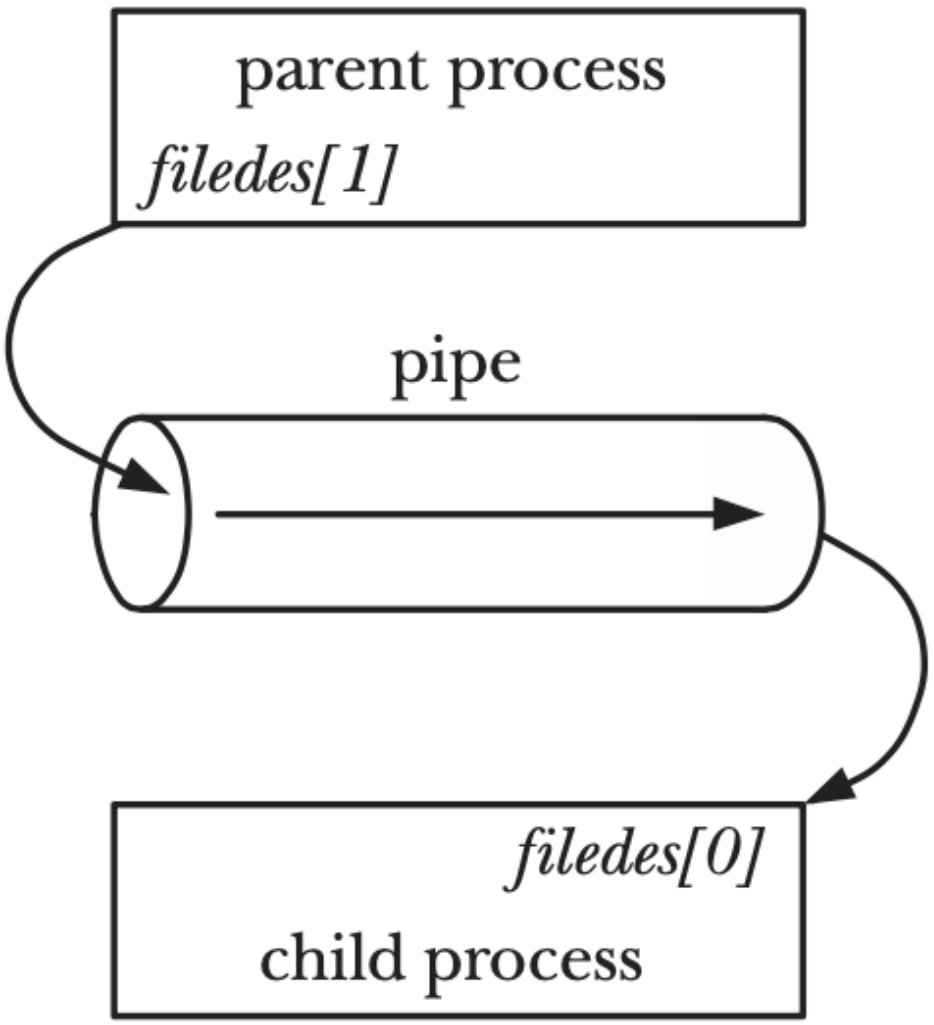
```
#include <unistd.h>

int pipe(int filedes[2]);
//Returns 0 on success, or -1 on error
```



- la system call `pipe()` crea una nuova 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 sulla pipe.



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

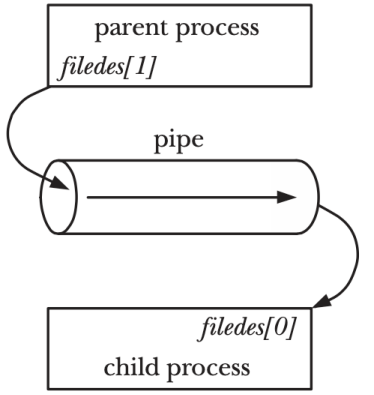


- tecnicamente sarebbe possibile leggere e scrivere sulla pipe per il genitore e il figlio, ma non è il modo standard di utilizzo
- subito dopo la `fork()`, un processo chiude il proprio descrittore per l'estremità in scrittura, e l'altro chiude il proprio descrittore per l'estremità in lettura.
  -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

Un esempio

```
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 di lettura ---
   break;
   default: // padre
   if (close(filedes[0]) == -1) // --- chiude il read end
      ddErrExit("close");

   // --- il padre compie qualche operazione di scrittura ---

   break;
}
```

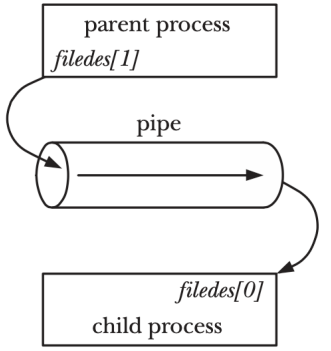
Lettura da una pipe



- Una operazione di lettura **consuma** i dati presenti nel buffer della pipe
  - Se i dati sono presenti, la read ritorna il numero di byte letti, che saranno memorizzati nella variabile buffer
  - Se i dati non sono presenti
    - Se almeno un file descriptor in scrittura è ancora aperto, allora la read è bloccante
    - Se nessun file descriptor in scrittura è aperto, allora la read ritorna il valore 0

```
char buffer[100];
int n_bytes;

n_bytes = read(filedes[0], buffer, sizeof(buffer));
```

Scrittura su una pipe



- Dopo una operazione di scrittura su una pipe
  - Se il buffer può contenere i dati passati come parametro, la `write` ritorna il numero di byte scritti
  - Se il buffer della pipe è pieno, la `write` è bloccante (in attesa di una `read`)
  - Se nessun file descriptor in lettura è ancora aperto, un segnale di `SIGPIPE` è generato (per notificare che i dati appena scritti non saranno mai letti!)

```
char buffer[100];
int n_bytes;

n_bytes = write(filedes[1], buffer, sizeof(buffer));
```

Chiusura dei descrittori inutilizzati (lettore)

- I file descriptors inutilizzati in lettura e in scrittura **devono** essere chiusi.
- Il processo che legge dalla pipe chiude il proprio write descriptor, così che quando l'altro processo completa il proprio output e chiude il proprio descrittore write, il lettore riceve, dopo aver consumato tutti i byte nel buffer, il valore di ritorno 0 (che indica un EOF)
  - Se invece il processo lettore non chiude la propria estremità aperta in scrittura, dopo che l'altro processo avrà chiuso il proprio descrittore write, il lettore non riceverà l'end-of-file neppure dopo avere letto tutti i dati dalla pipe.
  - In questo caso, una read() si bloccherebbe in attesa di dati (che sappiamo non arriveranno!) perché il kernel sa che c'è ancora almeno un descrittore di file aperto per la pipe

Chiusura dei descrittori inutilizzati (scrittore)

- Il processo scrittore chiude l'estremità aperta in lettura della pipe per una ragione diversa. Quando un processo tenta di scrivere su una pipe per il quale nessun processo ha un descrittore aperto in lettura, il kernel invia il segnale SIGPIPE al processo scrittore
- Un processo può organizzarsi per intercettare o ignorare tale segnale; in questo caso la write() sulla 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 della pipe, ed è la ragione per cui i descrittori aperti in lettura dovrebbero essere chiusi

Alcune considerazioni

- Le operazioni di scrittura e lettura sono atomiche (importante nel caso in cui si avessero molteplici scrittori e lettori)
- La dimensione del buffer della pipe è indicato in `PIPE_BUF` (includere `limits.h`)
- Se un processo è bloccato in attesa di una lettura o scrittura e viene interrotto da un segnale, l'operazione ritorna `-1` con `errno=EINTR`

Comunicazione tra processi non padre-figlio

- Le pipe possono essere usate per la comunicazione fra (due o più) processi parenti, supponendo che la pipe sia creata da un antenato comune e prima della serie di chiamate fork() con cui sono stati creati i vari figli.
  - Per esempio, una pipe potrebbe essere usata per mettere in comunicazione un processo e un processo nipote (granchild) del primo. Il primo processo crea la pipe, e quindi effettua una fork(), e il figlio effettua una ulteriore fork() creando così il nipote.
  - Un altro scenario frequente è l'utilizzo di una pipe per la comunicazione fra due processi fratelli (siblings): il loro genitore crea la pipe, e quindi crea i due figli.
    - E’ così facendo che la shell crea una pipeline.

«Copiare» un file descriptor su un altro

```
#include <unistd.h>

int dup2(int fd_src , int fd_dst)
```

- `dup2()` copia il file descriptor `fd_src` sul file descriptor `fd_dst`
- se `fd_dst` è stato precedentemente aperto, allora `dup2()` provvede a chiuderlo
- Esempio:
  - Se `fd[1]` è il write end di scrittura di una pipe, `dup2(fd[1],1)` redirige tutto quello che prima veniva diretto sullo stdout verso la pipe
  - Una `printf` avrà quindi l'effetto di scrivere sulla pipe e non su stdio

FIFO

FIFO e pipe

- Una FIFO è simile a una pipe. La principale differenza è che una FIFO ha un nome all'interno del file system ed è aperta nello stesso modo di un file.
- Questo permette a una FIFO di essere utilizzata per comunicazioni fra processi non imparentati (e.g., un client e un server).

FIFO e pipe

- Una volta che una FIFO è stata aperta possiamo utilizzare le stesse system call dell'I/O utilizzate con le pipe e gli altri file (cioè, `read()`, `write()`, e `close()`).
- Come per le pipe, anche le FIFO hanno un'estremità aperta in scrittura e una aperta in lettura, e come per le pipe, i dati sono letti nello stesso ordine in cui sono stati scritti
  - Questa caratteristica conferisce alle FIFO il loro nome: first in, first out. Le FIFO sono anche note come *named pipes*.

Creare una FIFO (da linea di comando)

- Possiamo creare una FIFO dalla shell con il comando mkfifo:
  - `mkfifo [ -m mode ] pathname`
- Il `pathname` è il nome della FIFO che si intende creare, e l'opzione `-m` specifica i permessi analogamente al comando `chmod`.
- Listato con il comando `ls -l`, una FIFO è mostrata con il carattere `p` nella prima colonna

```
$ mkfifo -m 644 prima_fifo
$ ls -l prima_fifo
prw-r--r--  1 schi  staff  0 Nov 23 21:29 prima_fifo
```

Creare una FIFO (in un programma C)

- La funzione `mkfifo()` crea una nuova FIFO con il `pathname` dato.
- L'argomento mode specifica i permessi per la nuova FIFO. Tali permessi sono specificati mettendo in OR (ORing) varie possibili costanti.

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

int mkfifo(const char *pathname, mode_t mode);
//Returns 0 on success, or -1 on error
```

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

Sincronizzazione con FIFO

- L'utilizzo di una FIFO serve ad avere un processo lettore e uno scrittore alle due estremità.
  - di default, l'apertura di una FIFO in lettura (flag `O_RDONLY` della `open()`) si blocca finché un altro processo apre la FIFO in scrittura (flag `O_WRONLY` della `open()`).
  - per contro, l'apertura della FIFO in scrittura si blocca finché un altro processo non apre la FIFO in lettura
  - In altri termini, l'apertura di una FIFO sincronizza i processi lettori e scrittori.
  - Se l'altra estremità della FIFO è già aperta (per esempio nel caso in cui una coppia di processi hanno già aperto ciascuna estremità della FIFO), la `open()` va immediatamente a buon fine.