Processi

Comunicazione tra processi

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

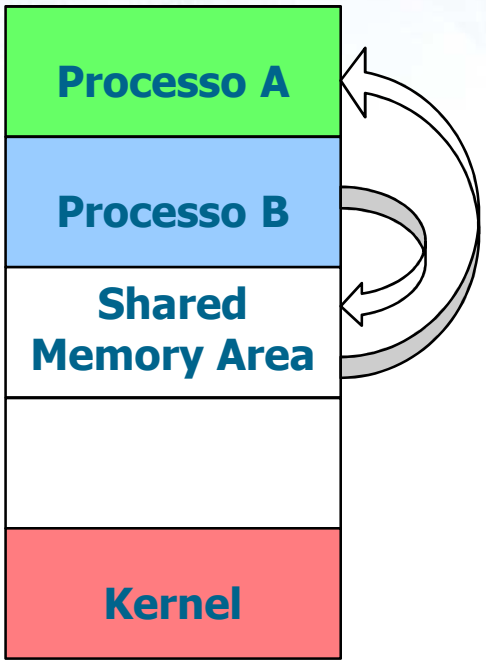
Processi independenti e cooperanti

- I processi concorrenti possono essere
  - Indipendenti
  - Cooperanti
- Un processo è **indipendente** se
  - **Non** può essere infuenzato dagli altri processi
  - **Non** può influenzare l'esecuzione di altri processi
- Un processo è **cooperante** in caso contrario
  - La cooperazione può avvenire solo tramite lo **scambio** oppure la **condivisione** di dati
  - Scambio e condivisione di dati richiedono l'implementazione di meccanismi di sincronizzazione opportuni

Comunicazione tra processi

- La condivisione di informazioni si denota spesso con il termine **IPC InterProcess Communication**
- I modelli di comunicazione principali sono basati su
  - Memoria condivisa
  - Scambio di messaggi

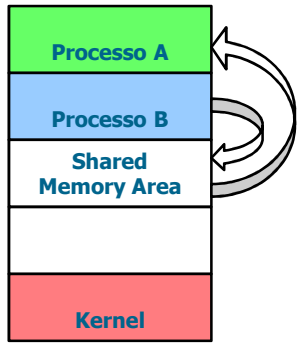
Modelli di comunicazione



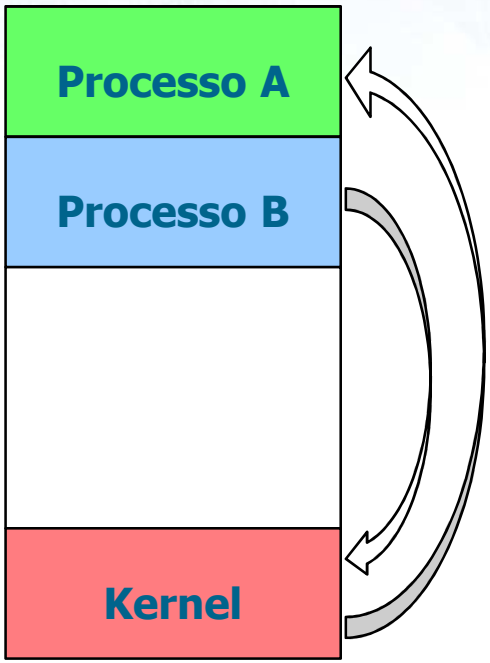
- Memoria condivisa
  - Condivisione di una zona di memoria
  - R/W dei dati in tale area
    - Normalmente il SO **impedisce** a un processo di accedere alla memoria di un altro processo
    - I processi devono accordarsi su
      - Diritti di accesso (R, W, R/W)
      - Strategie di gestione
        - Produttore – consumatore con buffer limitato o illimitato

Modelli di comunicazione

- I metodi con buffer condiviso più comuni prevedono l'utilizzo di
  - File
    - Si condivide il nome o il puntatore del file prima di effettuare una fork o una exec
  - **File mappati** in memoria
    - Ogni processo associa una regione della propria memoria centrale a un file
    - Ogni processo accede alla sua zona di memoria, operando di fatto sullo stesso file
- La tecnica è adatta per condividere quantità elevate di dati



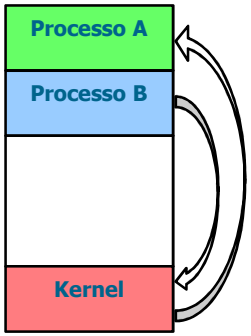
Modelli di comunicazione



- Scambio di messaggi
  - La comunicazione avviene mediante lo scambio di messaggi
  - Occorre instaurare un canale di comunicazione
  - Richiede l'utilizzo di system calls
    - La chiamata di una system call richiede l'intervento del kernel
    - L'intervento del kernel causa un rallentamento nei tempi di esecuzione
  - Adatta allo scambio di dati in quantità ridotta

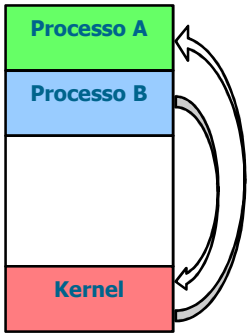
Canali di comunicazione

- Un canale di comunicazione viene usualmente caratterizzato da
  - Denominazione (naming)
    - La comunicazione può essere diretta o indiretta
    - La comunicazione è diretta se viene effettuata specificando esplicitamente il destinatario (in trasmissione) e il mittente (in ricezione)
      - send (dest, messaggio)
      - receive (src, &messaggio)
    - La comunicazione è indiretta se avviene tramite **mailbox**
      - send (mailboxAddress, messaggio)
      - receive (mailBoxAddress, &messaggio)



Canali di comunicazione

- Sincronizzazione
  - Tanto l'invio quanto la ricezione di messaggi può essere
    - Sincrona, i.e., bloccante
    - Asincrona, i.e., non bloccante
- Capacità
  - La coda utilizzata per la comunicazione può avere lunghezza (capacità)
    - Pari a zero: il canale non può avere messaggi in attesa, no c'è buffering; il sender si blocca in attesa del receiver; il receiver in attesa del sender
    - Limitata: il sender si blocca nel caso la coda sia piena; il receiver in caso di coda vuota
    - Infinita: il sender non si blocca mai; il receiver in caso di coda vuota



Canali di comunicazione

- UNIX prevede
  - **Half-duplex pipes**
  - FIFOs
  - Full-duplex pipes
  - Named full-duplex pipes
  - Message queues
  - **Semaphores**
  - Sockets
  - STREAMS

Estensioni delle pipe analizzate

Analizzate per la gestione della sincronizzazione tra processi

Comunicazione processi in rete. Ogni processo è indentificato da un socket a cui è associato un indirizzo di rete

Utilizzati da UNIX System V

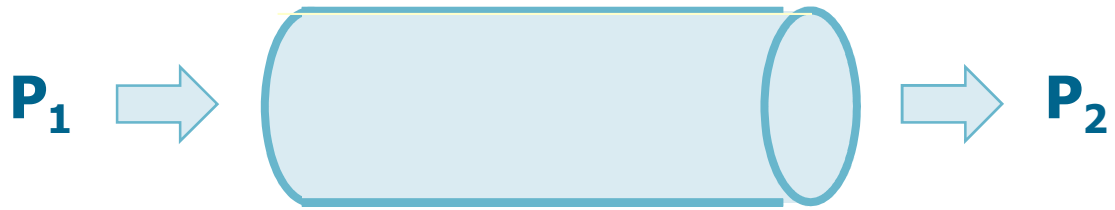
Non tutti le tipologie di comunicazione sono supportate da tutte le versioni di UNIX

Le pipe

- Sono la più vecchia forma di comunicazione nei SO UNIX
- Forniscono un canale di comunicazione
  - Diretto
  - Asincrono
  - A capacità limitata
- Le pipe "vivono" in memoria e sono più efficienti dell'utilizzo di file

Le pipe

- Una **pipe** consiste in un **flusso dati** tra due processi
  - Una pipe si gestisce in maniera simile a un file
  - Ogni pipe è rappresentata mediate due descrittori (interi), uno per ciascun estremo
  - Un processo (P_1) scrive a una estremità e l'altro processo (P_2) legge dall'altra estremità



Le pipe

- Storicamente una pipe
  - È **half-duplex**
    - I dati possono fluire in entrambe le direzioni (da P_1 a P_2 oppure da P_2 a P_1) ma **non** contemporaneamente
    - Meccanismi più potenti (e.g., full-duplex) sono nati più recentemente e hanno portabilità più limitata
  - Può essere utilizzata per la comunicazione tra processi con un **parente comune**
    - I file descriptor devono essere comuni ai due processi comunicanti e quindi tali processi devono avere un antenato comune

Di fatto per problemi di sincronizzazione assumono modalità simplex

Simplex: Monodirezionale
Half-Duplex: Bidirezionale ma non alternata (walkie-talkie)
Full-Duplex: Bidirezionale (telefono)

System call pipe ( )

```
#include <unistd.h>

int pipe (int fileDescr[2]);
```

- La system call pipe crea un pipe
- La funzione ritorna due descrittori di file nel vettore **fileDescr**
  - fileDescr[0]: Aperto per la lettura della pipe
  - fileDesrc[1]: Aperto per la scrittura della pipe
  - L'output effettuato su fileDescr[1] corrisponde all'input ricevuto su fileDescr[0]

I descrittori delle pipe sono degli interi

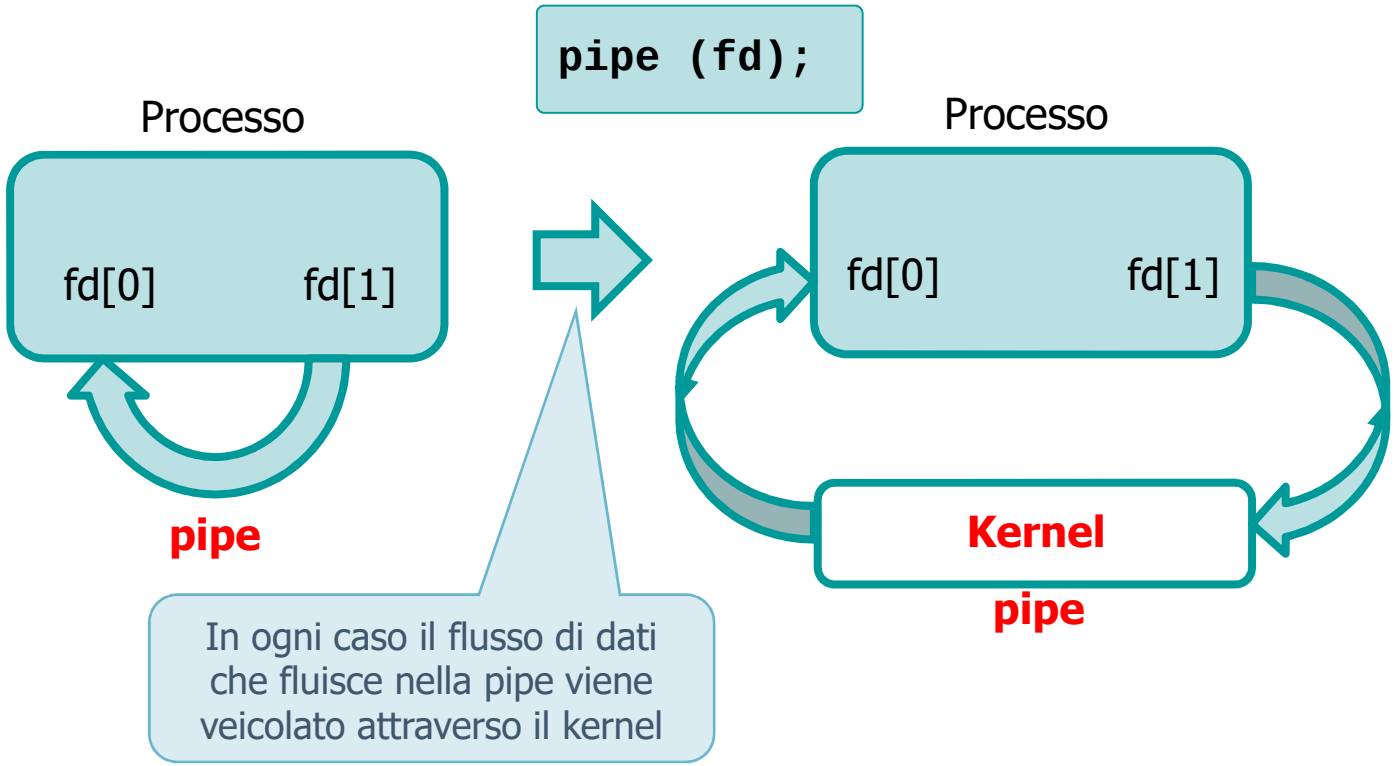
System call pipe ( )

- Parametri
  - Il vettore di descrittori di dimensione 2
- Valore di ritorno
  - Il valore 0, se l'operazione ha avuto successo
  - Il valore -1, in caso si sia verificato un errore
- Le risorse associate a una pipe sono rilasciate quando tutti i processi coinvolti
  - Hanno chiuso i propri terminali
  - Sono terminati

```
int pipe (int fileDescr[2]);
```

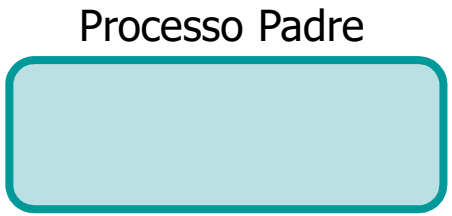
System call pipe ( )

- Una pipe all'interno dello stesso processo è pressoché inutile



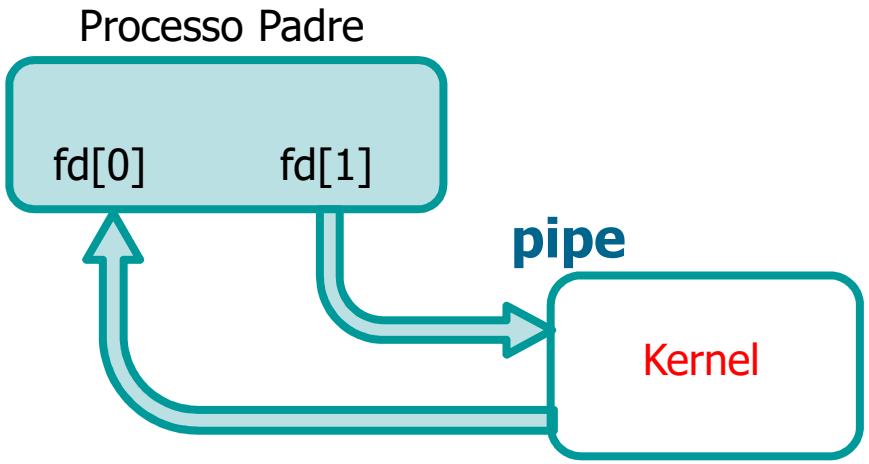
System call pipe ( )

- Occorre quindi creare una pipe che metta in comunicazione due processi (padre e figlio)
- Questo viene ottenuto mediante clonazione del padre **dopo** aver creato la pipe



System call pipe ( )

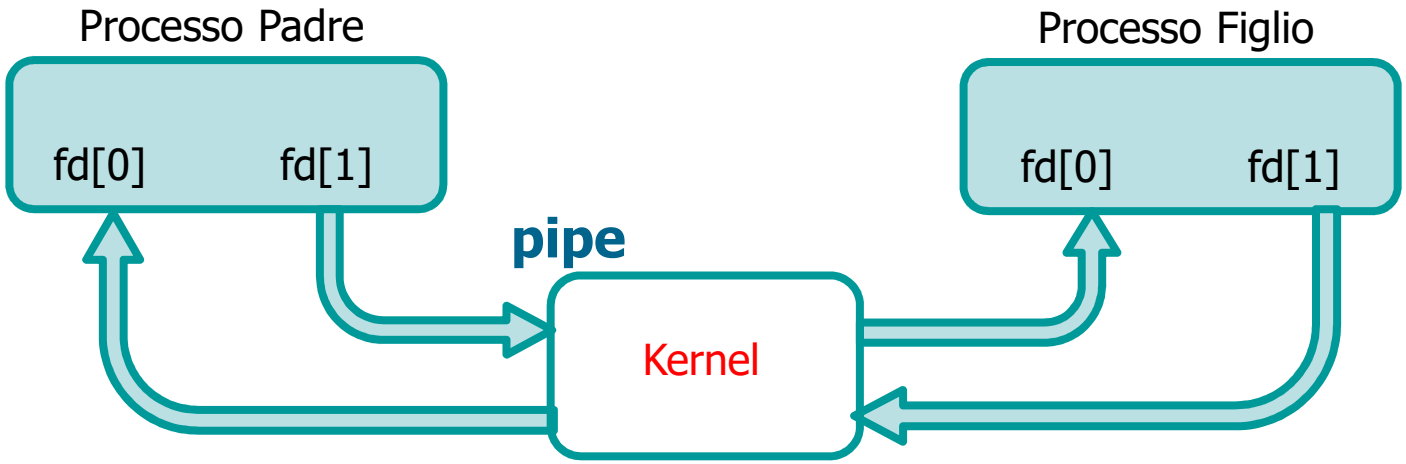
- Il processo "padre" crea una pipe



System call pipe ( )

- Il processo "padre" crea una pipe
- Effettua una fork
- Il processo figlio **eredita** i descrittori dei file

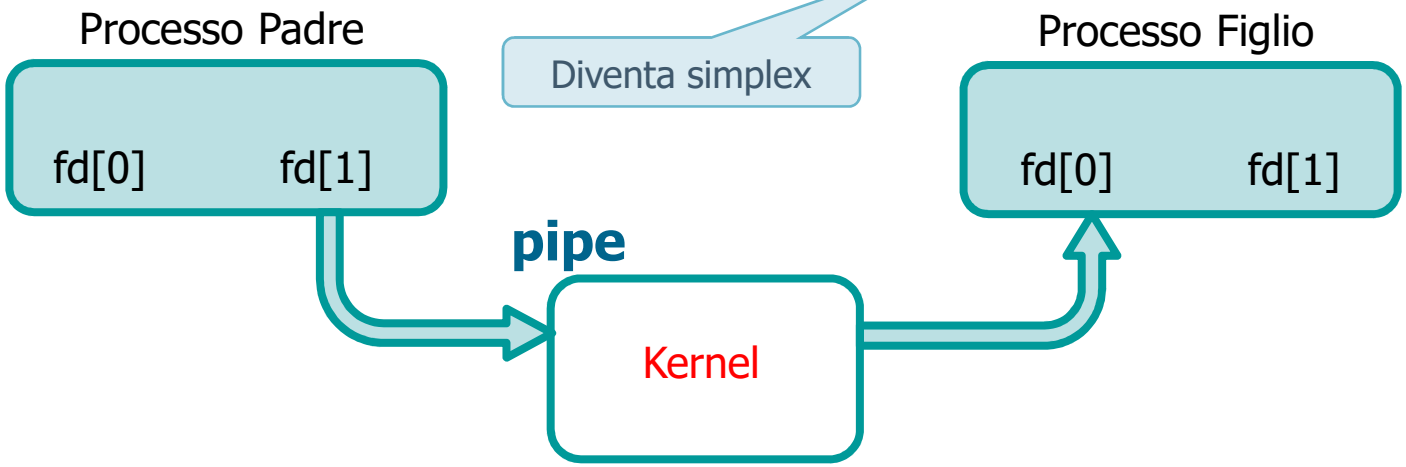
Se la pipe venisse fatta **dopo** la fork i descrittori **non** sarebbero ereditati



System call pipe ( )

- Il processo "padre" crea una pipe
- Effettua una fork
- Il processo figlio **eredita** i descrittori dei file
- Uno dei due processi (e.g., il padre) scrive nella pipe mentre l'altro (e.g., il figlio) legge dalla pipe
- Il descrittore non utilizzato può essere chiuso

Modalità half-duplex



I/O su pipe

I descrittori delle pipe sono degli interi

- Le pipe si manipolano (R/W) in maniera identica ai file UNIX
  - Si utilizzano le primitive **read** e **write**
    - La configurazione standard è quella di avere un unico scrittore e un unico lettore
    - In presenza di più scrittori i dati scritti risultano interallacciati
      - L'operazione di **write** è atomica (la pipe ha **lock** interni per gestirne il suo l'utilizzo)
      - L'atomicità però può essere interrotta quando una **write** tenta di agire su una pipe (quasi) piena e quindi l'operazione viene completa in più fasi
    - In presenza di più lettori risulta indeterminato chi è il lettore di un dato specifico

I/O su pipe

- La system call **read**
  - Si blocca se la pipe è vuota (**è bloccante**)
  - Ritorna solo i caratteri disponibili, se la pipe contiene meno caratteri di quanto richiesto
  - Ritorna 0 se la pipe è stata chiusa all'altra estremità
    - Perché la read ritorni 0 occorre **non** ci siano più writer (anche fittizzi)
    - Occorre quindi **tutti** gli estremi in scrittura vengano chiusi, anche quello all'interno del processo che tenta di leggere

I/O su pipe

- La system call **write**
  - Si blocca se la pipe è piena (**è bloccante**)
  - La dimensione di una pipe dipende dalle architetture a dall'implementazione
    - La costante PIPE_BUF definisce il numero di caratteri scrivibili atomicamente in una pipe
    - Il valore standard di PIPE_BUF varia da 4KByte a 128 Kbyte e può essere modificato
  - Ritorna SIGPIPE se l'altra estremità è stata chiusa
  - Se la fine delle operazioni di scrittura non si vogliono verificare in base al valore di ritorno della read è sempre possibile trasferire un dato "sentinella" (end-of-message marker)

Esercizio

- Si effettui la comunicazione di un dato tra un processo padre e un processo figlio, ovvero
  - Si crei una pipe mettendola in comune tra un processo padre e un processo figlio
  - Si trasferisca un singolo carattere dal processo padre al processo figlio
- Flusso logico
  - Creazione della pipe
  - Clonazione del processo
  - Chiusura del terminale inutilizzato della pipe
  - Operazioni di read e write ai due estremi

Soluzione

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main () {
  int n;
  int file[2];
  char cR;
  char cW = 'X';
  pid_t pid;
  if (pipe(file) == 0) {
    pid = fork ();
    if (pid == -1) {
      fprintf(stderr, "Fork failure");
      exit(EXIT_FAILURE);
    }
```

Prima si crea la pipe

Poi si clona il processo

Soluzione

```
  if (pid == 0) {
      // Child reads
      close (file[1]);
      n = read (file[0], &cR, 1);
      printf ("Read %d bytes: %c\n", n, cR);
      exit (EXIT_SUCCESS);
    } else {
      // Parent writes
      close (file[0]);
      n = write (file[1], &cW, 1);
      printf ("Wrote %d bytes: %c\n", n, cW);
    }
  }
  exit(EXIT_SUCCESS);
}
```

L'estremo non utilizzato viene chiuso

Il figlio legge

Il padre scrive

Lettura e scrittura sono bloccanti. Quindi i processi si sincronizzano.

Esercizio

- Le pipe hanno dimensione infinita?
  - Ovvero, qual è la dimensione di una pipe?
- Dato che la **write** è una system call bloccante è possibile comprendere la dimensione di una pipe effettuando operazioni di **write** sino a quando l'operazione si blocca

Soluzione

```
...
#define SIZE  524288

int fd[2];


int main () {
...
int i, n, nR, nW;
char c = '1';
setbuf (stdout, 0);

...
pipe(fd);
n = 0;
```

Prima si crea la pipe

Soluzione

```
if (fork()) {
  fprintf (stdout, "\nFather PID=%d\n", getpid());
  sleep (1);
  for (i=0; i<SIZE; i++) {
    nW = write (fd[1], &c, 1);
    n = n + nW;
    fprintf (stdout, "W %d\r", n);
  }
} else {
  fprintf (stdout, "Child  PID=%d\n", getpid());
  sleep (10);
  for (i=0; i<SIZE; i++) {
    nR = read (fd[0], &c, 1);
    n = n + nR;
    fprintf (stdout, "\t\t\t\tR %d\r", n);
  }
}
```

Poi si clona il processo

Il padre scrive SIZE caratteri

Il figlio legge altrettanto

\r = CR = Carriage Return (non a capo)

Esempio

```
➢ ./pgrm
Father PID=2272
Child  PID=2273
W 0

...

W 65536

...

W 65536 R 0

...

W 524288 R 524288
```

Il numero di caratteri scritti aumenta sino alla dimensione della pipe

Quando la pipe è piena la write si blocca

Dopo 10 secondi si incomincia a leggere e si incomincia a svuotare la pipe

R & W avvengono in parallelo sino a raggiungere SIZE caratteri

Esercizio

- Nell'esempio precedente si scrivono e si leggono **esattamente** SIZE caratteri
- Come si procede per la scrittura di un numero di caratteri variabile?
  - Si gestisce il valore di ritorno della **read**
  - Si gestisce un dato "sentinella" (end-of-message marker)
    - Provare …
    - La trasmissione di informazioni complesse richiede la gestione di un qualche tipo di protocollo di comunicazione

Soluzione

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  char c;
  int n, fd[2];
  pid_t p;

  setbuf (stdout, 0);

  pipe(fd);
  fprintf (stdout, "Reading from %d; ", fd[0]);
  fprintf (stdout, "Writing to %d\n", fd[1]);

  p = fork();
```

Prima si crea la pipe

Poi si clona il processo

Soluzione

Buggy ...

```
  if (p > 0) {
    write (fd[1], "Hi Child!", 9);
    wait (NULL);
    fprintf (stdout, " - Parent ends\n");
  } else {
    while ((n = read (fd[0], &c, sizeof (char))) > 0) {
      fprintf (stdout, "%c", c);
    }
    fprintf (stdout, " - Child ends\n");
  }

  return 0;
}
```

Parent

Child

Esempio di esecuzione

```
➢ ./pgrm
Reading from 3; Writing to 4
Hi Child!
```

Il programma non termina !
La **read** è **bloccante**

Soluzione

Corretta ...

```
  if (p > 0) {
    close (fd[0]);
    write (fd[1], "Hi Child!", 9);
    close (fd[1]);
    wait (NULL);
    fprintf (stdout, " - Parent ends\n");
  } else {
    close (fd[1]);
    while ((n = read (fd[0], &c, sizeof (char))) > 0) {
      fprintf (stdout, "%c", c);
    }
    fprintf (stdout, " - Child ends\n");
  }

  return 0;
}
```

I terminali non utilizzati vanno chiusi

Anche nel figlio (in lettura)

Esempio di esecuzione

```
➢ ./pgrm
Reading from 3; Writing to 4
Hi Child!
 - Parent ends
```

Il programma termina !

Esempio

- Che cosa succede se non si rispetta la gestione half-duplex di una pipe?
  - È possibile invertire le operazioni di lettura e di scrittura?
    - In generale in risultato è indefinito ... però è possibile ottenere un risultato corretto nel primo caso ...
  - È possibile avere lettori e/o scrittori multipli?
    - Provare ...

Esempio

Il programma riceve una stringa nel parametro argv[1]

```
int fd[2];
setbuf (stdout, 0);
pipe (fd);
if (fork()!=0) {
  while (1) {
    if (strcmp(argv[1],"F")==0||strcmp(argv[1],"FC")==0) {
      c = 'F';
      fprintf (stdout, "Father Write %c\n", c);
      write (fd[1], &c, 1);
    }
    sleep (2);
    if (strcmp(argv[1],"C")==0||strcmp(argv[1],"FC")==0) {
      read (fd[0], &c, 1);
      fprintf (stdout, "Father Read %c\n", c);
    }
    sleep (2);
  }
  wait ((int *) 0);
}
```

Se argv[1] è "F"
il padre scrive solo e il figlio legge solo

Se argv[1] è "C"
il figlio scrive solo e il padre legge solo

Esempio

```
} else {
  while (1) {
    if (strcmp(argv[1],"F")==0||strcmp(argv[1],"FC")==0) {
      read (fd[0], &c, 1);
      fprintf (stdout, "Child Read %c\n", c);
    }
    sleep (2);
    if (strcmp(argv[1],"C")==0||strcmp(argv[1],"FC")==0) {
      c = 'C';
      fprintf (stdout, "Child Write %c\n", c);
      write (fd[1], &c, 1);
    }
    sleep (2);
  }
  exit (0);
}
```

Se argv[1] è "FC"
padre e figlio scrivono alternandosi

Esempio

```
➢ ./pgrm F
Father Write F
Child Read F
...
^C
➢ ./pgrm C
Child Write C
Father Read C
...
^C
➢ ./pgrm FC
Father Write F
Child Read F
Child Write C
Father Read C
...
^C
```

Solo il padre scrive
...OK

Solo il figlio scrive ...
OK

Padre e figlio scrivono
alternandosi
ogni 2 secs...
OK

Però come si alternano
nei casi reall?