IPC: Pipe e FIFO

Contiene lucidi tratti da: 2006-2007 Marco Faella, Clemente Galdi, Giovanni Schmid (Università di Napoli Federico II), 2005-2007 Francesco Pedullà, Massimo Verola (Uniroma2), 2001-2005 Renzo Davoli (Università di Bologna), Alberto Montresor (Università di Bologna).

Interprocess Communication (IPC)

- Per cooperare, i processi hanno bisogno di comunicare
- I segnali sono un primo modo di farlo
  - Il messaggio consiste nel “numero del segnale” ed, eventualmente, le informazioni in siginfo_t.
- Vogliamo però trasmettere informazioni arbitrarie

Le pipe

Le pipe forniscono un meccanismo attraverso il quale l'output di un processo diviene l'input per un altro processo.

Il più semplice esempio di tale meccanismo è fornita dall'operatore di pipe | di una shell:

ls | grep old

è una pipeline grazie alla quale l'output del comando ls viene passato in input al comando grep, senza un file intermedio di appoggio.

La comunicazione tra i due comandi è unidirezionale (da ls a grep, il viceversa non è possibile), e la sincronizzazione è ottenuta arrestando grep quando non c'è nulla da leggere e arrestando ls quando la pipe è piena.

Le pipe

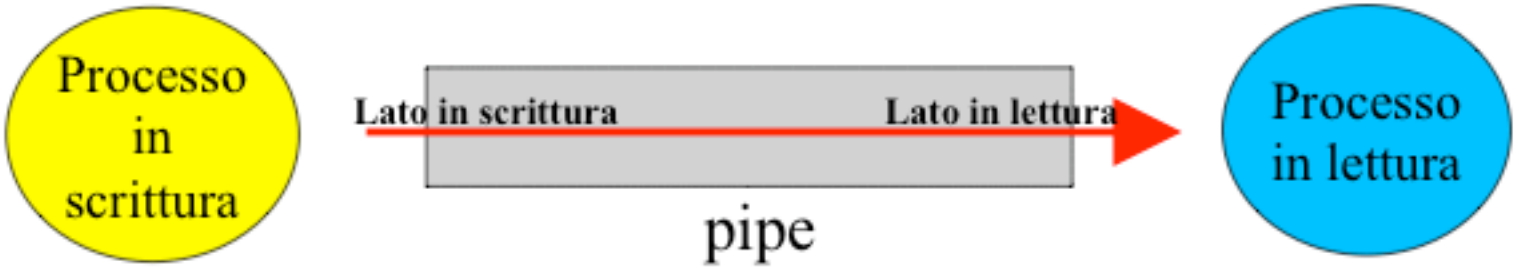
- Le pipe (*tubi*) sono canali di comunicazione a senso unico tra due processi
- I processi devono essere imparentati
  - tipicamente, un padre e un figlio
- Un processo scrive sulla pipe (usando write)
- Un altro processo legge dalla stessa pipe (usando read)

Le pipe

- **Cos'è un pipe?**
  - E' un canale di comunicazione che unisce due processi
- **Caratteristiche:**
  - La più vecchia e la più usata forma di *interprocess communication* utilizzata in Unix
  - Limitazioni
    - Sono half-duplex (comunicazione in un solo senso)
    - Utilizzabili solo tra processi con un "antenato" in comune
  - Come superare queste limitazioni?
    - Gli *stream pipe* sono full-duplex
    - *FIFO* (*named pipe*) possono essere utilizzati tra più processi
    - *named stream pipe* = stream pipe + FIFO

Le pipe

Le pipe tra processi, realizzate attraverso la chiamata di sistema pipe o la funzione popen della libreria STDIO, sono del tutto analoghe: i dati immessi da un processo nella pipe con successive scritture sono accodati nell'attesa che un altro processo le legga; la coda è gestita con criterio FIFO.



Una pipe ha una dimensione finita, il cui valore è definito dalla costante PIPE_BUF: soltanto un numero massimo di byte può essere scritto o letto ogni volta.

Una pipe è subordinata all'organizzazione gerarchica dei processi: affinchè due processi possano comunicare in pipeline è necessario un antenato comune che abbia predisposto una pipe a questo fine.

La funzione Pipe

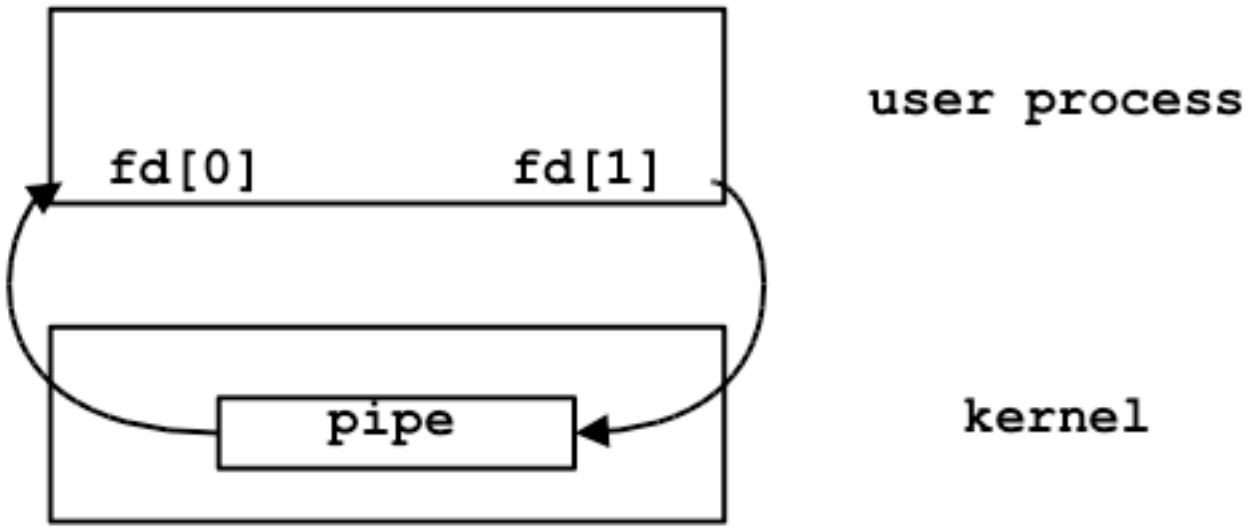
```
#include <unistd.h>

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

- l'argomento ***filedes*** è costituito da due descrittori di file:
  - ***filedes[0]*** è aperto in lettura e rappresenta il lato in lettura della pipe ;
  - ***filedes[1]*** è aperto in scrittura e rappresenta il lato in scrittura della pipe;
  - l'output di ***filedes[1]*** è l'input per ***filedes[0]***;

- restituisce 0 in caso di successo, -1 altrimenti.

Le pipe

- **System call:** `int pipe(int filedes[2]);`
  - Ritorna due descrittori di file attraverso l'argomento **`filedes`**
    - **`filedes[0]`** è aperto in lettura
    - **`filedes[1]`** è aperto in scrittura
  - L'output di **`filedes[1]`** (estremo di write del pipe) è l'input di **`filedes[0]`** (estremo di read del pipe)



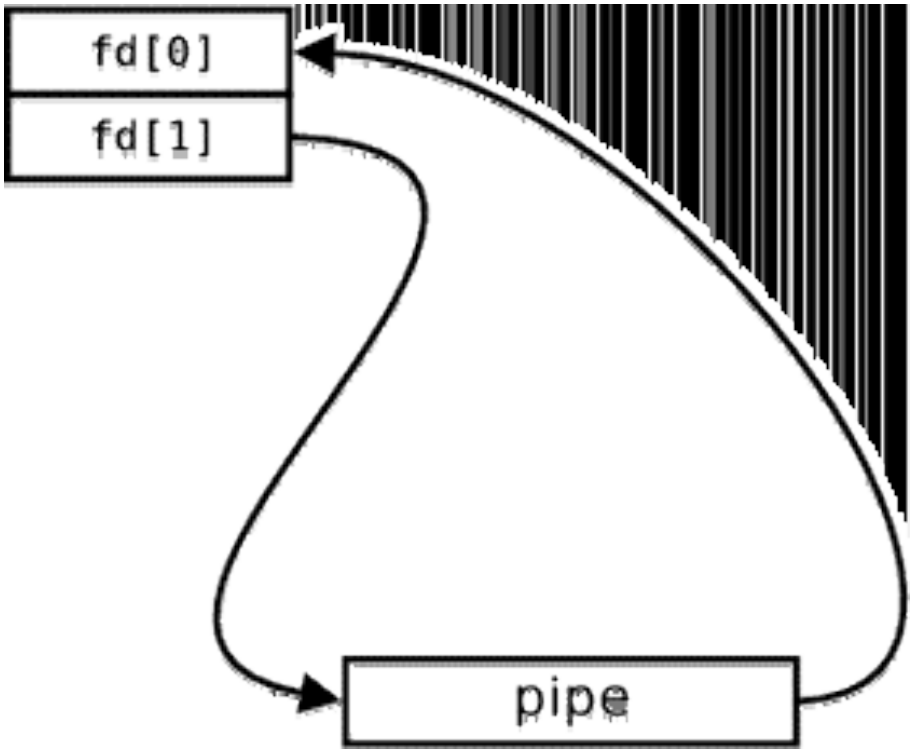
Le pipe

- Consideriamo il frammento:

```
int fd[2];

if (pipe(fd) < 0)
        perror("pipe"), exit(1);
```

- dopo la sua esecuzione:
  - fd[0] e' il descrittore per **leggere** dalla pipe
  - fd[1] e' il descrittore per **scrivere** sulla pipe

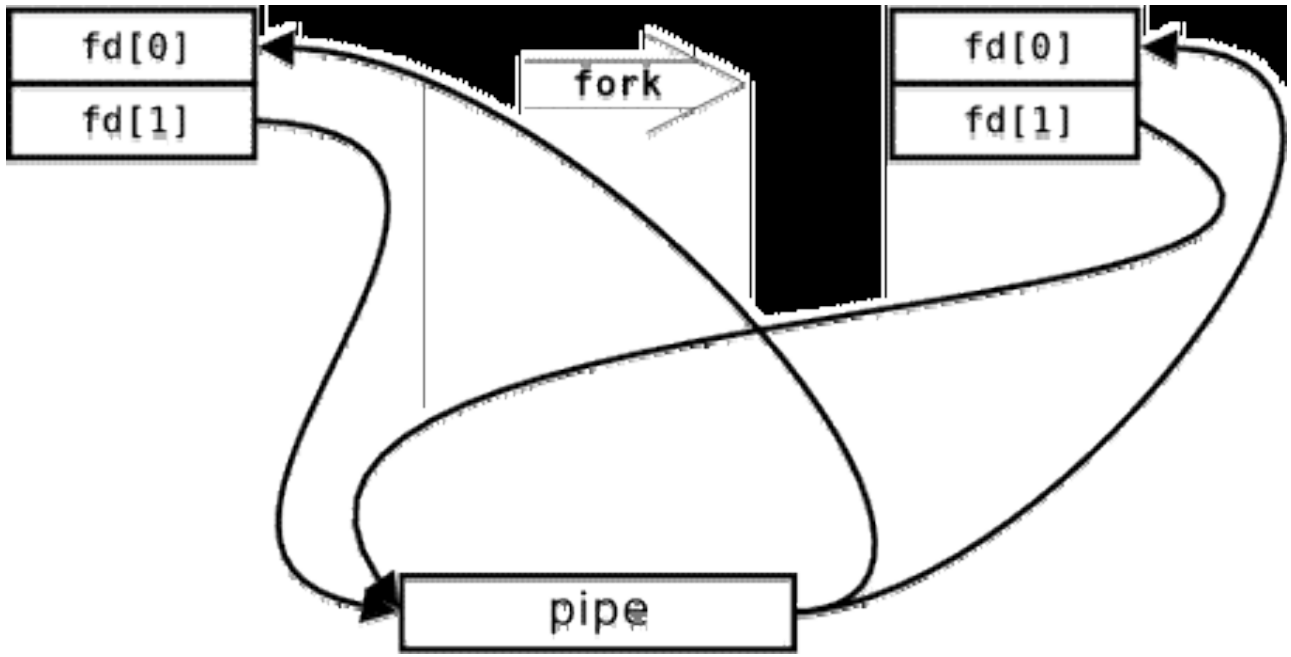
Le pipe



Generata da un singolo processo ha poca utilità …

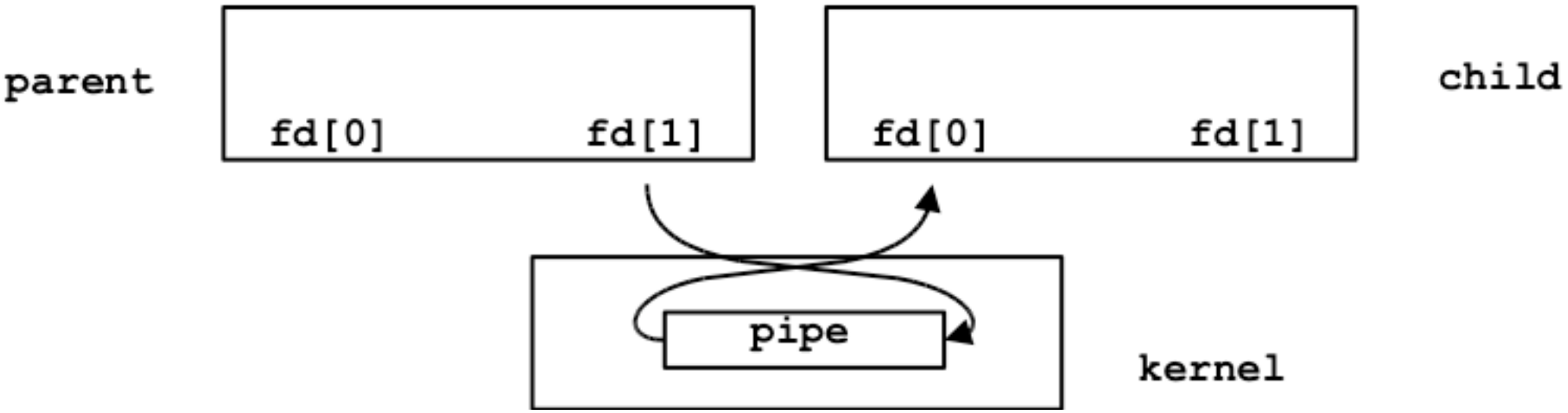
Utilizzo pipe

- Tipicamente, un processo crea una pipe e poi chiama fork



Utilizzo pipe

- **Come utilizzare i pipe?**
  - Cosa succede dopo la **`fork`** dipende dalla direzione dei dati
  - I canali non utilizzati vanno chiusi
- **Esempio: parent → child**
  - Il parent chiude l'estremo di read (**`close(fd[0]);`**)
  - Il child chiude l'estremo di write (**`close(fd[1]);`**)



Utilizzo pipe: read

- **Come utilizzare i pipe?**
  - Una volta creati, è possibile utilizzare le normali chiamate **`read/write`** sugli estremi
- **La chiamata `read`**
  - se l'estremo di write è aperto
    - restituisce i dati disponibili, ritornando il numero di byte
    - successive chiamate si bloccano fino a quando nuovi dati non saranno disponibili
  - se l'estremo di write è stato chiuso
    - restituisce i dati disponibili, ritornando il numero di byte
    - successive chiamate ritornano 0, per indicare la fine del file

Utilizzo pipe: write

- **La chiamata `write`**
  - se l'estremo di read è aperto
    - i dati in scrittura vengono bufferizzati fino a quando non saranno letti dall'altro processo
  - se l'estremo di read è stato chiuso
    - viene generato un segnale **`SIGPIPE`**
      - ignorato/catturato: write ritorna **`-1`** e **`errno=EPIPE`**
      - azione di default: terminazione
- **Esercizio:**
  - Due processi: parent e child
  - Il processo parent comunica al figlio una stringa, e questi provvede a stamparla

Una pipe tra padre e figlio

```
int fd[2];

if (pipe(fd) < 0)
        perror("pipe"), exit(1);
if ( (pid=fork()) < 0 )
        perror("fork"), exit(1);
else if (pid>0) {   // padre
        close(fd[0]);
        write(fd[1], "ciao!", 5);
} else {            // figlio
        close(fd[1]);
        n = read(fd[0], buf, sizeof(buf));
        write(STDOUT_FILENO, buf, n);
}
```

```c
/* pipe1: invio di dati da un genitore ad un figlio */

#include <stdio.h>
#include <unistd.h>
#define MAXLINE 64

int main(void)
{
  int n, fd[2];
  pid_t pid;
  char line[MAXLINE];

  if (pipe(fd) < 0) perror("pipe"), exit(1);

  if ( (pid = fork()) < 0) perror("fork"), exit(1);

  else if (pid > 0) { /* genitore */
    close(fd[0]);
    write(fd[1], "hello world\n", 12);
  }
  else { /* figlio */
    close(fd[1]);
    n = read(fd[0], line, MAXLINE);
    write(STDOUT_FILENO, line, n);
  }
  exit(0);
}
```

Leggere e scrivere sulle pipe

- All'inizio una pipe è vuota
- write aggiunge dati alla pipe
- read legge *e rimuove* dati dalla pipe
  - non si possono leggere piu' volte gli stessi dati da una pipe
  - non si puo' chiamare lseek su una pipe
  - i dati si ottengono in ordine First In First Out
- una pipe con una estremita' chiusa si dice rotta (broken)

Leggere e scrivere sulle pipe

- Scrivere: write aggiunge i suoi dati alla pipe
  - se la pipe e' rotta, viene generato il segnale SIGPIPE e write restituisce un errore
- Leggere: read(fd[0], buf, 100)
  - meno di 100 bytes nella pipe: read legge l'intero contenuto della pipe
  - piu' di 100 bytes nella pipe: read legge i primi 100 bytes
  - pipe vuota: read si blocca in attesa di dati
  - pipe vuota e rotta: read restituisce 0

Esempio

- **Problema**: Consideriamo un programma **`prog1`** che scrive su standard output. Come si puo' fare in modo che l'output venga visualizzato una pagina alla volta, senza pero' modificare il programma stesso?
- **Soluzione:** si scrive un altro programma che:
  - crea un pipe e poi genera un processo child mediante `fork`
  - nel codice del parent chiude l'estremo di read del pipe e lo stdout, e riassegna mediante **`dup2`** il fd dello **`stdout (1)`** sull'estremo di write del pipe
  - nel codice del child chiude l'estremo di write del pipe e lo stdin, e riassegna mediante **`dup2`** il fd dello **`stdin (0)`** sull'estremo di read del pipe
  - il parent mediante **`exec`** lancia il programma **`prog1`**
  - il child mediante **`exec`** lancia un programma tipo di paginazione dell'output tipo **`more`** o **`less`**

Pipe e Dup

- Nell’esempio precedente lettura e scrittura direttamente su pipe descriptor
- Interessante è l’uso della duplicazione dei pipe descriptors su stdin o stout
- Es. quando programmi scrivono o leggono su stdin o stdout

Pipe tra due programmi

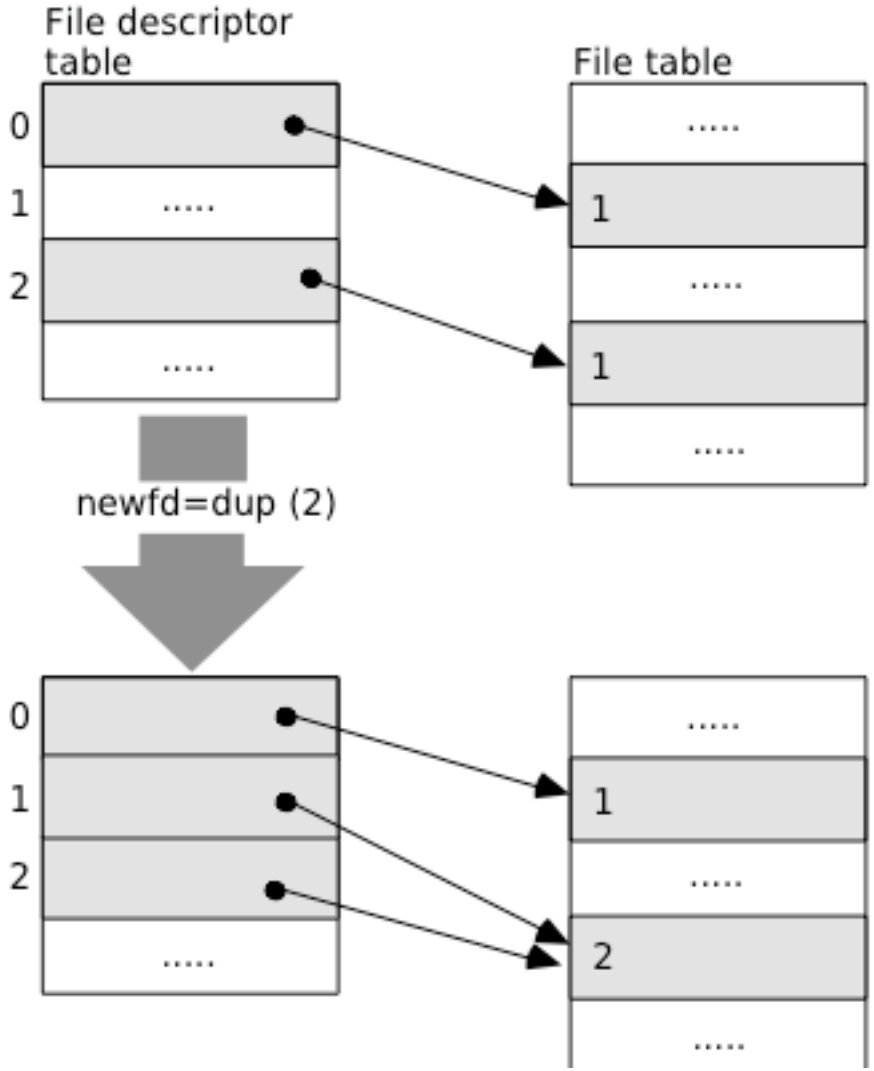
Per associare lo stdout o lo stdin di un programma, rispettivamente, al lato in scrittura o a quello in lettura di una una pipe, si può utilizzare la funzione seguente:

```
#include <unistd.h>

int  dup2 ( int  filedes,  int filedes2 );
```

- duplica il descrittore di file ***filedes***, nel nuovo descrittore ***filedes2*** ;
- se ***filedes2*** è aperto, esso viene chiuso prima della duplicazione;
- restituisce il nuovo descrittore ***filedes2*** in caso di successo, -1 altrimenti.

Duplicazione (vedi File System)

- Un file descriptor esistente viene duplicato da una delle seguenti funzioni:
  - `int dup(int filedes);`
  - `int dup2(int filedes, int filedes2);`
- Entrambe le funzioni “duplicano” un file descriptor, ovvero creano un nuovo file descriptor che punta alla stessa file table entry del file descriptor originario
- Nella file table entry c’e’ un campo che registra il numero di file descriptor che la “puntano”



Duplicazione (vedi File System)

- Funzione `dup`
  - Seleziona il più basso file descriptor libero della tabella dei file descriptor
  - Assegna la nuova file descriptor entry al file descriptor selezionato
  - Ritorna il file descriptor selezionato
- Funzione `dup2`
  - Con **`dup2`**, specifichiamo il valore del nuovo descrittore come argomento **`filedes2`**
  - Se **`filedes2`** è già aperto, viene chiuso e sostituito con il descrittore duplicato
  - Ritorna il file descriptor selezionato

Pipe tra due programmi

Per associare lo stdout o lo stdin di un programma, rispettivamente, al lato in scrittura o a quello in lettura di una una pipe, si può utilizzare la funzione seguente:

```
#include <unistd.h>

int  dup2 ( int  filedes,  int filedes2 );
```

- duplica il descrittore di file ***filedes***, nel nuovo descrittore ***filedes2*** ;
- se ***filedes2*** è aperto, esso viene chiuso prima della duplicazione;
- restituisce il nuovo descrittore ***filedes2*** in caso di successo, -1 altrimenti.

Esempio

Legge un file indicato sulla linea di comando e lo visualizza utilizzando un pager.

```
#define DEF_PAGER       "/bin/more"     /* default pager program */

int main(int argc, char *argv[])
{
        if (argc != 2)
                err_quit("usage: a.out <pathname>");

        if ((fp = fopen(argv[1], "r")) == NULL)
                err_sys("can't open %s", argv[1]);
        if (pipe(fd) < 0)
                err_sys("pipe error");
```

Esempio

```
if ((pid = fork()) < 0) {
        perror("fork error");
} else if (pid > 0) { /* padre  */
        close(fd[0]);          /* chiude la pipe il lettura */

        /* il padre copia il file argv[1] sulla pipe  */

        while (fgets(line, MAXLINE, fp) != NULL) {
                n = strlen(line);
                if (write(fd[1], line, n) != n)
                        perror("write error to pipe");
        }
        if (ferror(fp))
                perror("fgets error");

        close(fd[1]);   /* chiude la pipe in scrittura */

        if (waitpid(pid, NULL, 0) < 0)
                err_sys("waitpid error");
        exit(0);
```

Esempio

```
else {                  /* figlio */
        close(fd[1]);   /* chiude la pipe in scrittura*/
        if (fd[0] != STDIN_FILENO) {/* controlla che non  sia gia' ok*/
                if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                        perror("dup2 error to stdin");
                close(fd[0]);   /* non necessario dopo dup2*/
        }

        /* costruisce gli argomenti per execl() */
        if ((pager = getenv("PAGER")) == NULL)
                pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, '/')) != NULL)
                argv0++;      /* puntatore ad ultimo '/' in pager,
                                 estrae il nome del comando */
        else
                argv0 = pager;  /* non ci sono "/" in pager */

        if (execl(pager, argv0, (char *)0) < 0)
                err_sys("execl error for %s", pager);
}
exit(0);
```

Pipe tra due programmi

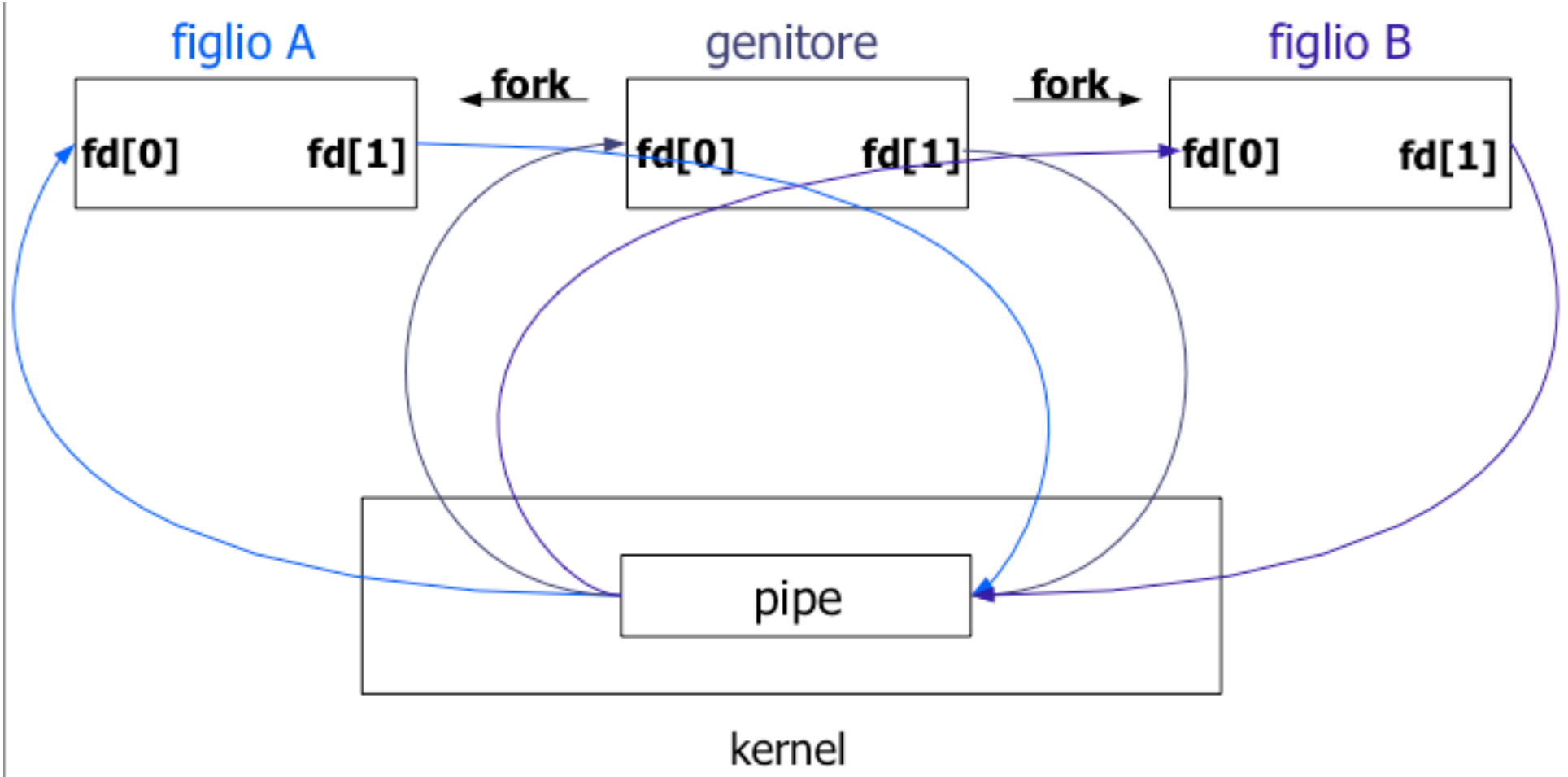
Per realizzare una pipe tra due programmi, come

ls | grep old

la sequenza di eventi è precisamente la seguente:

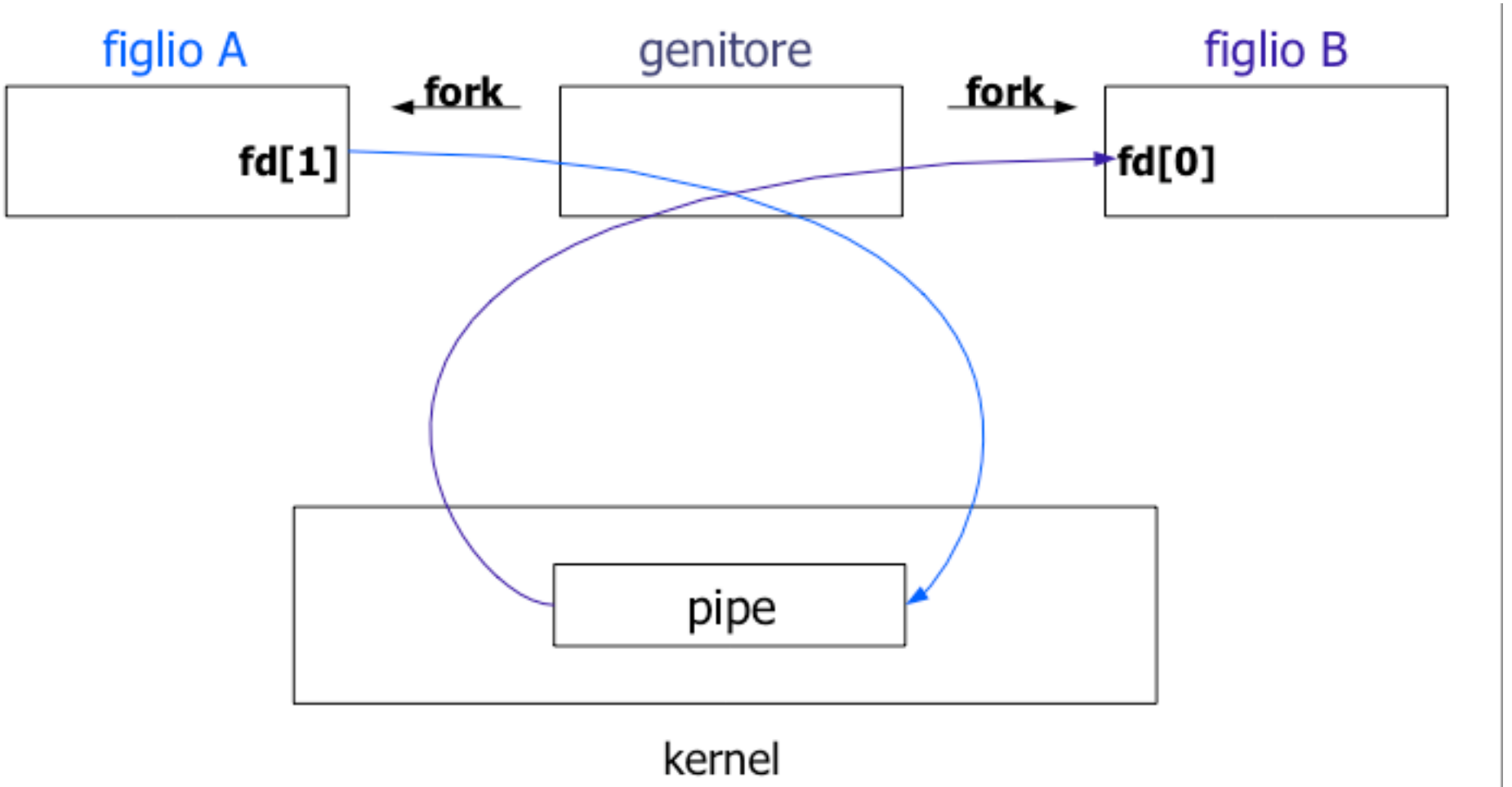
1. Il genitore crea una pipe usando la funzione pipe;
2. Il genitore crea due figli con la funzione fork, dopodichè chiude entrambi i lati della pipe;
3. il figlio scrittore chiude il lato in lettura della pipe ed associa il proprio stdout al lato in scrittura della pipe;
4. il figlio lettore chiude il lato in scrittura della pipe ed associa il proprio stdin al lato in lettura della pipe;
5. ciascuno dei figli carica con una exec il proprio programma;
6. Al termine della comunicazione, lo scrittore e il lettore chiudono il lato della pipe di loro competenza.

Pipe tra due programmi



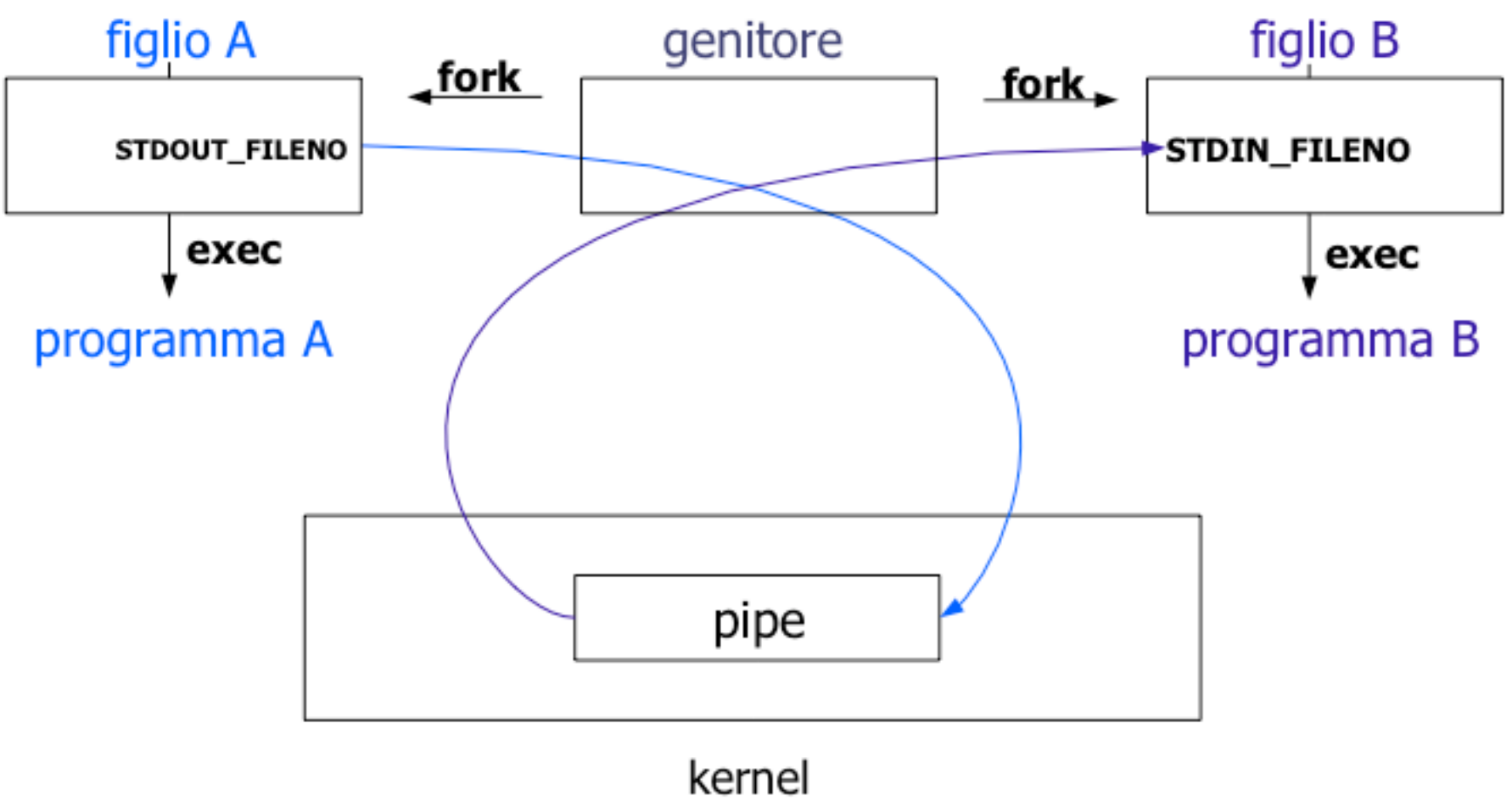
Pipe half-duplex dopo due chiamate a fork

Pipe tra due programmi



Figlio A scrittore e figlio B lettore

Pipe tra due programmi



Programma A scrittore e programma B lettore

```c
/* implementa la pipe  "who | wc" mediante la syscall pipe */

#include<unistd.h>
#include<stdio.h>
#include<sys/types.h>
#include<errno.h>

int main(void)
{
   int mypipe[2];
   pid_t pid1, pid2; /* 2 figli, 1 x ciascun programma */

   if (pipe(mypipe)<0) perror("pipe"), exit(1);

   if ((pid1=fork())<0) perror("fork"), exit(1);

   else if (pid1 == 0){ /* 1st child, reader */
     close(mypipe[1]);

     if(dup2(mypipe[0], STDIN_FILENO)!= STDIN_FILENO)
       perror("dup2"), exit(1);

     close(mypipe[0]); /* opzionale */

     if (execl("/usr/bin/wc","wc",NULL)<0)
       perror("execl"), exit(1);
   }
```

```
else{  /* pid1>0, parent */

 if ((pid2=fork())<0) perror("fork"), exit(2);

 else if (pid2 == 0){ /* 2nd child, writer */
   close(mypipe[0]); /* opzionale */

   if(dup2(mypipe[1], STDOUT_FILENO)!= STDOUT_FILENO)
            perror("dup2"), exit(1);

   close(mypipe[1]);  /* opzionale */

   if (execlp("who","who",NULL)<0)
     perror("execlp"), exit(1);
 }
else{ /* pid1>0, pid2>0, parent */
   close(mypipe[0]);  /* opzionale */
   close(mypipe[1]);

   /* attende per i 2 figli */
   waitpid(pid1,NULL,0);
   waitpid(pid2,NULL,0);

   exit(0);
   }
 }
```

popen e pclose

L'esecuzione di un comando da parte di un processo in modo che quest'ultimo possa riceverne l'output o inviargli l'input è una operazione molto comune, per la quale è dunque preferibile avere delle funzioni di più alto livello.

La libreria standard di IO fornisce invero le funzioni popen e pclose, che consentono al programmatore di evitare le azioni esplicite di creazione di una pipe, generazione di un figlio, chiusura del lato della pipe non utilizzato da parte dei processi scrittore e lettore, exec di una shell per l'esecuzione del comando e, infine, attesa della terminazione di quest'ultimo.

Più precisamente, popen crea una pipe, crea un figlio, chiude i lati non utilizzati della pipe ed esegue il comando; mentre pclose attende che l'esecuzione del comando sia terminata e chiude la pipe.

popen e pclose

```
#include <stdio.h>

FILE  *popen (const char *cmdstring , const char *type );
int  pclose ( FILE *filepointer );
```

La funzione popen esegue il comando *cmdstring* e restituisce un puntatore a file per il processo chiamante. Se *type* è uguale a "w" il puntatore a file è connesso allo stdin del comando, se invece *type* è uguale a "r" tale puntatore è connesso allo stdout.

popen e pclose

Il comando *cmdstring* è eseguito come sh -c *cmdstring*, ossia la shell espande i caratteri speciali. Ad es., si può scrivere:

```
fp=popen("ls *.c", "r");
```

(eseguita da bourne shell)

La funzione popen restituisce un puntatore a file in caso di successo, il puntatore NULL in caso di errore.

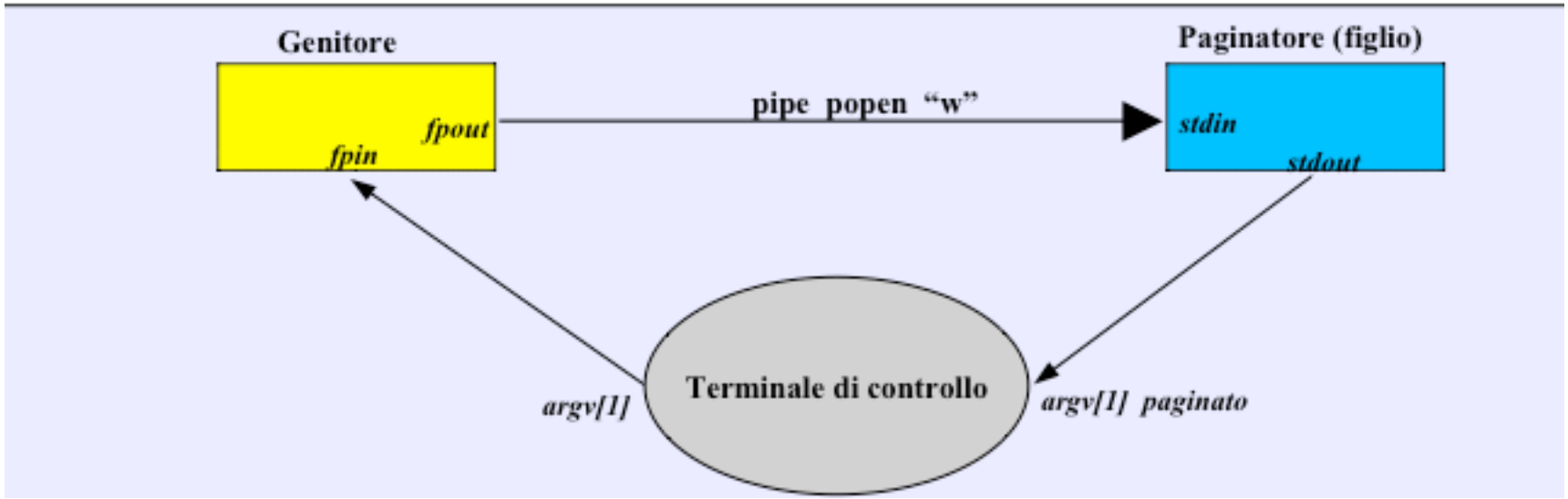
La funzione pclose chiude lo stream di I/O standard e attende che il comando sia terminato, restituendo lo stato di terminazione della shell. Se la shell non può essere eseguita, lo stato di terminazione restituito da pclose equivale all'esecuzione da parte della shell di exit(127).

Esempio: pager

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

#define MAXLINE  4096
#define PAGER "${PAGER:-more}" /* var. d'ambiente, o default */

int main(int argc, char *argv[ ])
{
  char line[MAXLINE];
  FILE *fpin *fpout;

  if (argc != 2)
    printf("usage: %s <pathname>", argv[0]), exit(1);

  if ( (fpin = fopen(argv[1], "r")) == NULL)
    perror("fopen"), exit(1);

  /* crea un figlio "paginatore", collegandosi al suo stdin */
  if ((fpout=popen(PAGER, "w") ) == NULL)
    perror("popen"), exit(1);
```

Esempio: pager

```
/* copia argv[1] al paginatore */
while (fgets(line, MAXLINE, fpin) != NULL)
  if (fputs(line, fpout) == EOF)
    perror("fputs"), exit(1);

if (ferror(fpin))
  perror("fgets"), exit(1);

if (pclose(fpout) == -1)
  perror("pclose"), exit(1);

exit(0);
}
```



FIFO (o named pipe)

- Le pipe possono essere utilizzate solo se due processi hanno un “antenato” comune
  - Un processo crea la pipe e qualche discendente la usa.
- Le FIFO possono essere utilizzate per consentire la **comunicazione tra due processi arbitrari**.
  - Devono condividere solo il “nome” della FIFO

FIFO (named pipe)

I file speciali FIFO (pipe con nome) consentono di superare alcune delle limitazione delle pipe. Essi difatti, rispetto a queste ultime, offrono i seguenti vantaggi:

- una volta creati, esistono nel file sistem fintanto che non vengono esplicitamente cancellati;
- possono essere usati da processi che non hanno un comune antenato.

I file FIFO possono essere creati in due modi:

- attraverso la shell, con il comando mkfifo;
- all'interno di un programma, con la chiamata alla funzione mkfifo.

Una volta creato un file FIFO, su di esso si possono effettuare le operazioni usuali di IO su file (open, read, write, close, ...)

FIFO (o named pipe)

- Dopo la creazione della FIFO, puo' essere usata come “un file”
  - utilizzando open, read, write, close
  - NON la lseek
- E' possibile che piu' processi scrivano sulla stessa FIFO
  - Se il numero di byte scritti sulla FIFO e' inferiore a PIPE_BUF, le scritture sono “atomiche”
- L'utilizzo di O_NONBLOCK consente di non bloccare le operazioni di open/read/write.
  - Attenzione ad errori e SIGPIPE!

FIFO (named pipe)

Come per le pipe, se si esegue una write su di un file FIFO che nessun processo ha aperto in lettura, è generato il segnale SIGPIPE .

E' comune la situazione in cui più processi scrivono su di una stesso file FIFO: affinché i dati non si mischino è necessario utilizzare operazioni atomiche di scrittura.

Come per le pipe, la costante PIPE_BUF stabilisce il massimo numero di byte che possono essere scritti in maniera atomica in un file FIFO.

FIFO

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

- crea un FIFO dal **pathname** specificato
- la specifica dell'argomento **mode** è identica a quella di **open**, **creat** (mode codifica i permessi di accesso al file mediante un numero ottale, ad esempio `0644 = rw-r--r--`)

- **Come funziona un FIFO?**
  - una volta creato un FIFO, le normali chiamate **open**, **read**, **write**, **close**, possono essere utilizzate per leggere il FIFO
  - il FIFO può essere rimosso utilizzando **unlink**
  - le regole per i diritti di accesso si applicano come se fosse un file normale

FIFO: open

- **Chiamata open**
  - File aperto senza flag O_NONBLOCK
    - Se il FIFO è aperto in sola lettura, la chiamata **si blocca** fino a quando un altro processo non apre il FIFO in scrittura
    - Se il FIFO è aperto in sola scrittura, la chiamata **si blocca** fino a quando un altro processo non apre il FIFO in lettura
  - File aperto con flag O_NONBLOCK
    - Se il FIFO è aperto in sola lettura, la chiamata **ritorna immediatamente**
    - Se il FIFO è aperto in sola scrittura, e nessun altro processo lo ha aperto in lettura, la chiamata **ritorna un messaggio di errore**

FIFO: write

- **Chiamata `write`**
  - se nessun processo ha aperto il file in lettura viene generato un segnale **`SIGPIPE`**:
    - ignorato/catturato: write ritorna **`-1`** e **`errno=EPIPE`**
    - azione di default: terminazione
- **Atomicità**
  - Quando si scrive su un pipe, la costante **`PIPE_BUF`** (in genere pari a 4096, vedi `/usr/include/linux/limits.h`) specifica la dimensione del buffer del pipe
  - Chiamate **`write`** di dimensione inferiore a **`PIPE_BUF`** vengono eseguite in modo atomico
  - Chiamate **`write`** di dimensione superiore a **`PIPE_BUF`** possono essere eseguite in modo non atomico
  - La presenza di piu' scrittori può causare interleaving tra chiamate

Operazioni e Modalità

Operazione corrente	Status del descrittore complementare	Comportamento modalità bloccante	Comportamento modalità non bloccante
apertura FIFO in sola lettura	FIFO aperta in scrittura	ritorna con successo	ritorna con successo
	FIFO chiusa in scrittura	blocca finchè la FIFO è aperta in scrittura	ritorna con successo
apertuta FIFO in sola scrittura	FIFO aperta in lettura	ritorna con successo	ritorna con successo
	FIFO chiusa in lettura	blocca finchè la FIFO è aperta in lettura	ritorna con l'errore ENXIO
lettura da pipe o FIFO vuote	pipe o FIFO aperta in scrittura	blocca finchè sono immessi dati o il lato in scrittura viene chiuso	ritorna con l'errore EAGAIN
	pipe o FIFO chiusa in scrittura	ritorna col valore 0 (fine del file)	ritorna col valore 0 (fine del file)
scrittura su pipe o FIFO	pipe o FIFO aperta in lettura	Se #byte<= PIPE_BUF, scrive in modo atomico, bloccando se non c'è spazio disponibile. Se #byte>PIPE_BUF scrive in modo non atomico.	Se #byte<=PIPE_BUF, scrive in modo atomico, ritornando con EAGAIN se non c'è spazio disponibile. Se #byte>PIPE_BUF e non c'è almeno 1 byte disponibile, ritorna con EAGAIN; altrimenti scrive solo ciò che può.
	pipe o FIFO chiusa in lettura	genera SIGPIPE	genera SIGPIPE

Esempio: comunicazione FIFO

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define MAX_BUF_SIZE 1000

int main(int argc, char *argv[]){

int fd, ret_val, count, numread;
char buf[MAX_BUF_SIZE];

/* Create the named - pipe */
ret_val = mkfifo("miafifo", 0666);
    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe");
        exit (1);    }

/* Open the pipe for reading */
    fd = open("miafifo", O_RDONLY);

    /* Read from the pipe */
    numread = read(fd, buf, MAX_BUF_SIZE);
    buf[numread] = '0';
    printf("Server : Read From the pipe : %s\n", buf);
```

```
/* Convert to the string to upper case */
    count = 0;
    while (count < numread) {
        buf[count] = toupper(buf[count]);
        count++;
    }
  printf("Server : Converted String : %s\n", buf);

}
```

Codice Server

Esempio: comunicazione FIFO

Codice Client

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

int main(int argc, char *argv[])
{
    int fd;

    /* Check if an argument was specified. */

    if (argc != 2) {
        printf("Usage : %s <string to be sent to the server>n", argv[0]);
        exit (1);
    }

    /* Open the pipe for writing */
    fd = open("miafifo", O_WRONLY);

    /* Write to the pipe */
    write(fd, argv[1], strlen(argv[1]));
}
```

Esempio: comunicazione FIFO

Esempio esecuzione:

- $./servFifo &
- $./clientFifo prova
- Server : Read From the pipe : prova
- Server : Converted String : PROVA

Esercizio 1

- Scrivere un programma C che crea un figlio
  - il **padre** invia al figlio 10 numeri interi casuali al ritmo di uno al secondo, e poi termina
  - il **figlio** riceve i numeri dal padre e li stampa sul terminale
  - il **figlio** termina dopo aver ricevuto il decimo numero
- I due processi comunicano tramite una pipe

- Opzionale: modificare il figlio in modo che termini dopo aver ricevuto 5 interi

Esercizio 2

- Scrivere un programma C che crea un figlio
  - il **padre** entra in un ciclo in cui legge da terminale un numero intero x e manda al figlio il numero x^2
  - il **padre** esce dal ciclo e termina quando l'utente immette il numero 0
  - il **figlio** riceve i numeri dal padre e li stampa sul terminale
  - il **figlio** termina quando riceve 0 dal padre
- I due processi comunicano tramite una pipe

I Thread

- processi “leggeri”
- un processo puo' avere diversi thread
- i thread di uno stesso processo condividono la memoria ed altre risorse
  - facile comunicare tra thread!
- pthread = “POSIX thread”

Identificare i thread

- processo process id (pid) pid_t
- thread thread id (tid)
  pthread_t

pthread_t **pthread_self**(void);

- restiuisce il tid del thread corrente

int **pthread_equal**(pthread_t t1, pthread_t t2);

- Ritorna non zero se uguali, 0 se diversi

Creare un thread

```
typedef void (*thread_start)(void *);

int pthread_create(pthread_t           *tid,
                   const pthread_attr_t *attributes
                   thread_start         start,
                   void                 *argument);
```

- Restituisce 0 se OK, un codice d'errore altrimenti
- tid = argomento di ritorno, conterrà il tid del nuovo thread
- attributes = attributi del thread (vedere dopo)
- start = indirizzo della funzione da cui partire
- argument = l'argomento passato alla funzione start

Trattare gli errori

- siccome i thread condividono la memoria, e' meglio non usare una variabile globale (come errno) per i codici d'errore
- quindi, le funzioni pthread restituiscono direttamente un codice d'errore

---

char ***strerror**(int n);

- restituisce un messaggio corrispondente al codice d'errore n

Risorse condivise

- I thread di uno stesso processo condividono:
  - la memoria
  - il pid e il ppid
  - i file descriptor
  - le reazioni ai segnali
    - cioe', le chiamate a signal influenzano tutti i thread
- I thread non condividono: lo stack

Terminare un thread

- invocare exit() fa terminare *l'intero processo!*
- per terminare solo il thread corrente, si puo':
  - invocare return
  - invocare pthread_exit
  - far si che un altro thread chiami pthread_cancel

Terminare un thread

void **pthread_exit**(void *ret);

- termina il thread corrente, con valore di uscita ret
- altri thread possono leggere il valore di uscita usando pthread_join (vedere dopo)
- fare attenzione che i dati puntati da ret sopravvivano alla terminazione del thread!
  - ret non deve puntare allo stack (no variabili locali)
  - si variabili globali o allocate dinamicamente

Aspettare la terminazione di un thread

int **pthread_join**(pthread_t tid, void **ret);

- attende che il thread specificato da tid termini
  - se quel thread e' gia' terminato, ritorna subito (come wait)
- restituisce 0 se OK, un codice d'errore altrimenti
- ret e' un parametro di ritorno usato per restituire il valore d'uscita dell'altro thread
  - occhio al doppio puntatore!
- se il valore di uscita non ci interessa, passiamo NULL al posto di ret

Esempio

```
typedef struct foo{
    int a;
    int b;
} myfoo;

myfoo test; // Variabile GLOBALE


void stampa(char *st, struct foo *test){
  printf("%s: tid=%d a=%d b=%d\n", st, pthread_self(),test->a, test->b);
}

void *fun1(void *arg){
    myfoo test2 = {1,2}; // Variabile LOCALE
    printf("%s %d\n", arg, pthread_self());
    stampa(arg, &test2);
    pthread_exit((void *)&test2);
}
```

Esempio

```
void *fun2(void *arg){
   test.a = 3;
   test.b = 4; // Variabile GLOBALE
   printf("%s %d\n", arg, pthread_self());
   stampa(arg, &test);
   pthread_exit((void *)&test);
}

void *fun3(void *arg){
   myfoo *test3;
   test3=malloc(sizeof(struct foo)); // Variabile allocata dinamicamente  test3->a = 5;
   test3->b = 6;
   printf("%s %d\n", arg, pthread_self());
   stampa(arg, test3);
   pthread_exit((void *)test3); //c
}
```

Esempio

```
int main(void){
    char st[100];
    pthread_t tid1;
    pthread_t tid2;
    pthread_t tid3;

    myfoo *b; // PUNTATORE alla struttura (non allocata)

    pthread_create(&tid1, NULL, fun1, "Thread 1"); // Locale
    pthread_join(tid1, (void *)&b);
    stampa("Master  ", b);

    pthread_create(&tid2, NULL, fun2, "Thread 2"); // Globale
    pthread_join(tid2, (void *)&b);
    stampa("Master  ", b);

    pthread_create(&tid3, NULL, fun3, "Thread 3"); // Dinamica
    pthread_join(tid3, (void *)&b);
    stampa("Master  ", b);
}
```

Esempio

Thread 1: 1077283760
// Locale
Thread 1: a=1 b=2
Master : a=1075156600 b=1077281896

Thread 2: 1077283760
// Globale
Thread 2: a=3 b=4
Master : a=3 b=4

Thread 3: 1077283760
// Dinamica
Thread 3: a=5 b=6
Master : a=5 b=6

Cancellare un thread

int **pthread_cancel**(pthread_t tid);

- chiede che il thread specificato da tid venga terminato
  - non *aspetta* la terminazione
- restituisce 0 se OK, un codice d'errore altrimenti
- il valore di uscita di un thread cancellato e' dato dalla costante PTHREAD_CANCELED

Thread e fork

- Se un thread chiama fork, nasce un nuovo processo con un solo thread
- Potenziali problemi con i mutex in possesso di altri thread

Thread e segnali

- Le chiamate a signal influenzano tutti i thread
- Se arriva un segnale a un processo, succede che:
  - se il processo ha impostato un handler, il segnale arriva ad *uno qualunque* dei thread (che esegue l'handler)
  - se invece la reazione al segnale consiste nel terminare il processo, *tutti i thread* vengono terminati

Inviare un segnale a un thread

int **pthread_kill**(pthread_t tid, int signo);

- manda il segnale signo al thread specificato da tid
  - se e' impostato un handler, viene eseguito nel thread tid
  - se non e' impostato un handler, e il comportamento di default e' di terminare il processo, vengono comunque terminati tutti i thread
- restituisce 0 se OK, un codice d'errore altrimenti

Esempio

```
signal(SIGUSR1, usr1);
pthread_create(&tid1, NULL, fun, "Thread 1");
pthread_create(&tid2, NULL, fun, "Thread 2");
pthread_create(&tid3, NULL, fun, "Thread 3");
sleep(1);
pthread_kill(tid1, SIGUSR1);
pthread_kill(tid2, SIGUSR1);
pthread_kill(tid3, SIGUSR1);

sigemptyset(&set); // Configura la maschera SOLO nel master thread     sigaddset(&set,
 SIGUSR1);
sigprocmask(SIG_SETMASK, &set, NULL);
sleep(1);
while (i++<10){
   sleep(1);
   kill(pid, SIGUSR1); // il segnale e' intercettato da un thread
}
```

Esempio

Thread id=1077283760 ricevuto segnale
Thread id=1079385008 ricevuto segnale
Thread id=1081486256 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale

Esempio

```
signal(SIGUSR1, usr1);
pthread_create(&tid1, NULL, fun, "Thread 1");
pthread_create(&tid2, NULL, fun, "Thread 2");
pthread_create(&tid3, NULL, fun, "Thread 3");
sleep(1);
pthread_kill(tid1, SIGUSR1);
pthread_kill(tid2, SIGUSR1);
pthread_kill(tid3, SIGUSR1);

sigemptyset(&set); // Configura la maschera SOLO nel master thread     sigaddset(&set,
 SIGUSR1);
sigprocmask(SIG_SETMASK, &set, NULL);
sleep(1);
while (i++<10){
   // sleep(1);  // ELIMINANDO LA SLEEP ??
   kill(pid, SIGUSR1); // il segnale e' intercettato da un thread
}
```

Note su Linux

- Linux supporta lo standard dalla versione 2.6 del kernel
  - per sapere la versione del kernel, usare “uname -a”
- La versione 2.4 invece si discosta dallo standard
  - i thread hanno pid diversi!

Riferimenti

- Advanced Programming in the Unix Environment (Second Ed.)
  - Pipe e FIFO: 15.1, 15.2, 15.5
  - Threads: 11.1, 11.2, 11.3, 11.4, 11.5