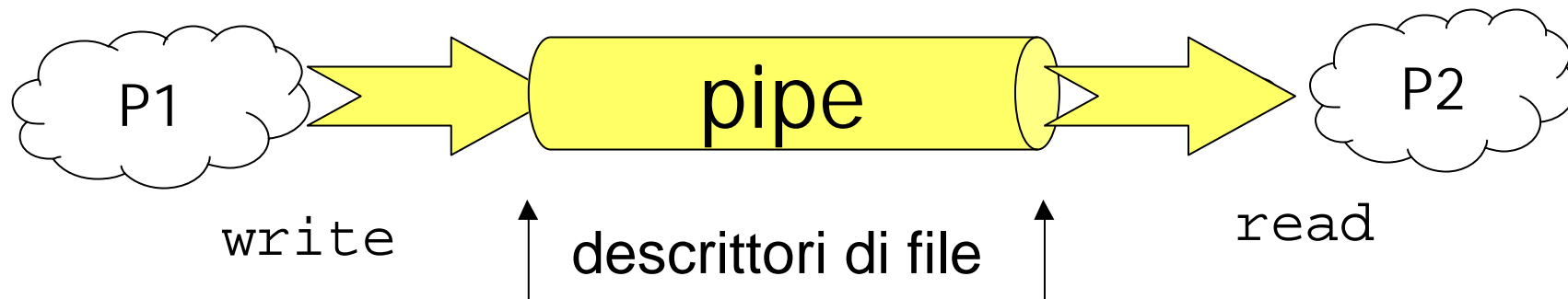


Meccanismi di Comunicazione Interprocesso (IPC)

- segnali
- ⇒ pipe (anonymous o FIFO)
- shared memory
 - socket
-
- sincronizzazione tra processi
 - semafori

Pipes

- Meccanismo IPC:
 - semplice modello di comunicazione monodirezionale
 - ◆ un processo P1 scrive su un estremo di una pipe i dati che vuole inviare ad un altro processo P2. Questo può riceverli leggendoli dall'altro estremo della pipe.
 - la sincronizzazione dei processi comunicanti è molto semplice perché si basa su primitive di scrittura/lettura non bufferizzate e bloccanti
- Anonime o FIFO (named pipe)



Low Level I/O (1)

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

int creat(char *path, mode_t mode);
int unlink(char *path);
int rename(char *old, char *new);
int open(char *path, int oflag, ...);
int close(int fd);
int read(int fd, char *buf, unsigned nbyte);
int write(int fd, char *buf, unsigned nbyte);
off_t lseek(int fd, off_t offset, int whence);
int link(char *path1, char *path2);
int chmod(char *path, mode_t mode);
int stat(char *path, struct stat *buf);
```

Low Level I/O (2)

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

man 2 open

```
int creat(char *path, mode_t mode);
int open(char *path, int oflag);
int open(char *path, int oflag, mode_t mode);
int close(int fd);
```

oflag:
O_RDONLY
O_WRONLY
O_CREAT
O_TRUNC

- La `open` riceve il nome del file da aprire o creare, i permessi `mode` e la modalità di apertura `oflag`
- restituisce un descrittore di file

mode:

S_IRUSR	0400
S_IWUSR	0200
S_IXUSR	0100
S_IRGRP	040
S_IWGRP	020
S_IXGRP	010
S_IROTH	04
S_IWOTH	02
S_IXOTH	01

Descrittori di File

- Il SO tiene traccia dei file aperti da ciascun processo mantenendo aggiornata una tabella dei descrittori di file
- Un descrittore di file è un intero usato dal SO come indice di una tabella dei descrittori
- Una volta ottenuto un descrittore tramite la `open`, la `create`, o la `pipe`, per le operazioni successive si specifica il file mediante il suo descrittore
 - 0 → *stdin*
 - 1 → *stdout*
 - 2 → *stderr*
- N.B. le funzioni che creano nuovi descrittori scelgono sempre il più piccolo intero disponibile
- N.B. un processo eredita dal padre la tabella dei descrittori

Low Level I/O (3)

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

man 2 read
man 2 write

```
int read(int fd, char *buf, unsigned nbyte);
```

- Legge dal file `fd` e copia in `buf` un max di `nbyte` bytes
- restituisce 0 per EOF, -1 se c'è un errore
- l'input dalla tastiera normalmente è trasmesso solo con un carattere di new line

```
int write(int fd, char *buf, unsigned nbyte);
```

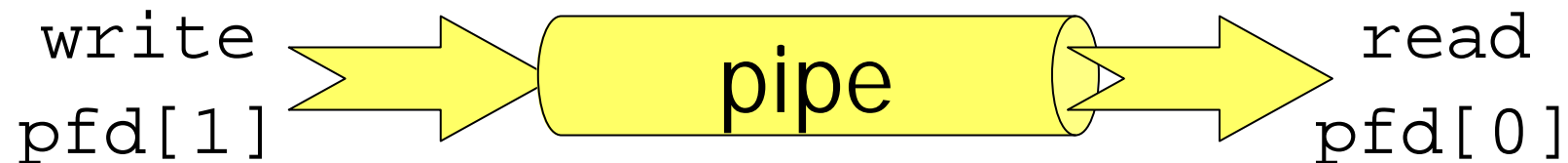
- Preleva `nbyte` bytes da `buf` e li scrive in `fd`

Come Creare una Pipe

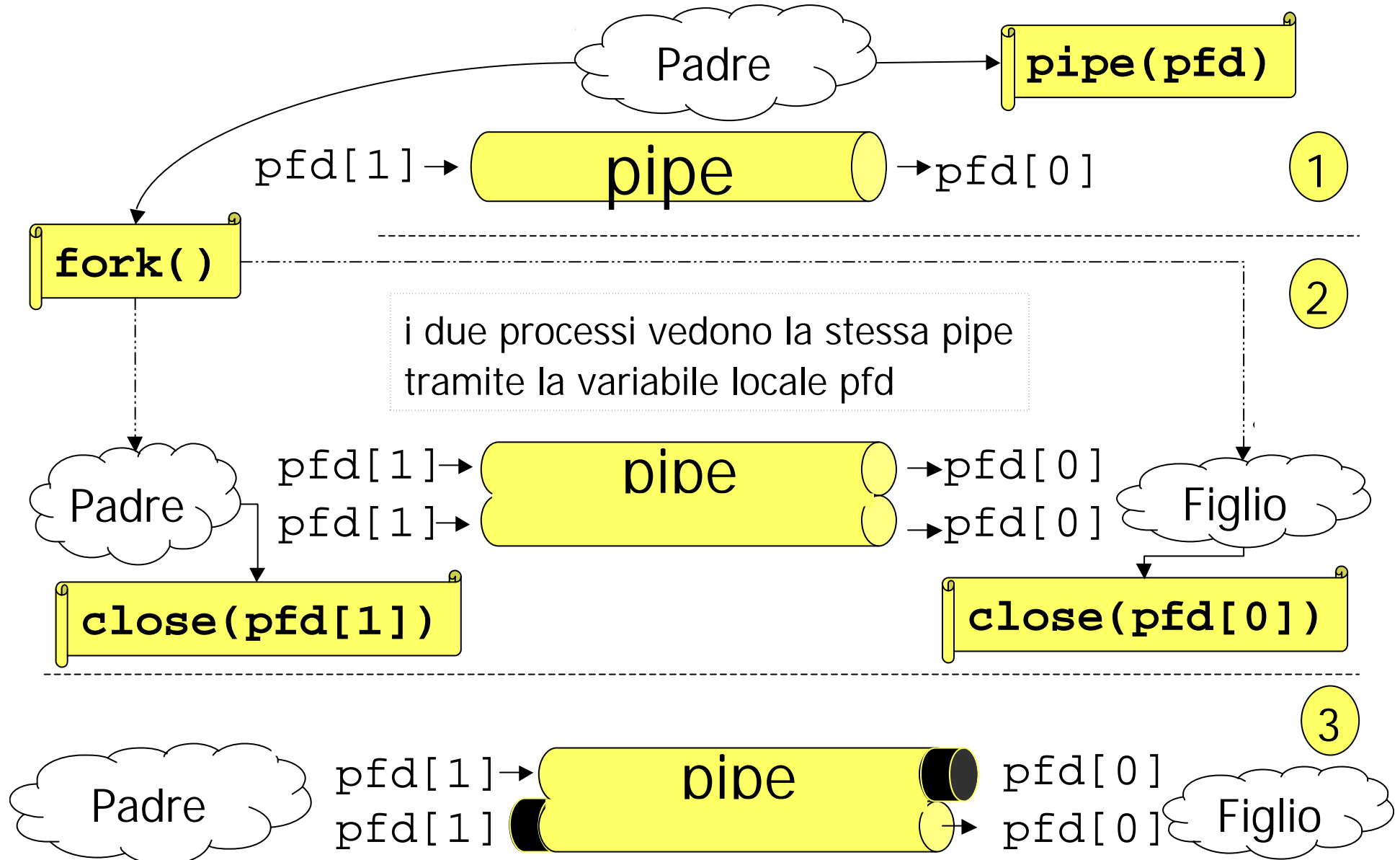
```
#include <stdio.h>
```

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

- crea una coppia di descrittori di file restituendoli in nell'array puntato da `pfd`.
 - `pfd[0]` è per la lettura,
 - `pfd[1]` è per la scrittura.
- Restituisce 0 se tutto va bene; -1 altrimenti



Come Usare una Pipe Anonima



pipe.c (1)

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

```
#define SIZE 1024
```

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

```
    int pfd[2]; ←
```

```
    int nread;
```

```
    int pid;
```

```
    char buf[SIZE];
```

```
    if (pipe(pfd) == -1) {  
        perror("pipe() failed");  
        exit(1);  
    }
```

```
    if ((pid=fork()) < 0 ) {  
        perror("fork() failed");  
        exit(2);  
    }
```

```
...
```

pipe.c (2)

...

```
if (pid==0) {      /* child */  
    → close(pfd[1]);  
    while ( (nread=read(pfd[0], buf, SIZE)) != 0) {  
        printf("child reads %s\n", buf);  
    }  
    → close(pfd[0]);  
  
    } else {        /* parent */  
  
    → close(pfd[0]);  
    strcpy(buf, "I'm your father!");  
    /* include null terminator in write */  
    write(pfd[1], buf, strlen(buf)+1);  
    → close(pfd[1]);  
    }  
    exit(0);  
}
```

Comunicazione Bidirezionale con Pipes

Per una comunicazione bidirezionale sono necessarie due pipes distinte: `pipe2ways.c`

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

```
#define SIZE 128
```

```
int main(int argc, char** argv) {  
    int pfd1[2], pfd2[2]; ←  
    int nread, pid;  
    char inBuf[SIZE], outBuf[SIZE];  
  
    if (pipe(pfd1) != 0 || pipe(pfd2) != 0) {  
        perror("pipe() failed");  
        exit(1);  
    }  
    if ((pid=fork()) < 0 ) {  
        perror("fork() failed");  
        exit(2);  
    }  
}
```

...

... pipe2ways.c...

...


```
if (pid==0) { /* child */
    printf("Child   pipe descriptors:
           pfd1[0] %d, pfd2[0] %d,
           pfd1[1] %d, pfd2[1] %d\n",
           pfd1[0],pfd2[0],pfd1[1],pfd2[1]);
    /* get a message from the father */
    close(pfd1[1]); close(pfd2[0]);
    nread=read(pfd1[0], inBuf, SIZE);
    inBuf[nread]='\0';
    close(pfd1[0]);
    printf("Child received: %s\n", inBuf);
    /* now bounce the message back */
    write(pfd2[1],inBuf,strlen(inBuf));
    close(pfd2[1]);
} else { /* parent */...
```

```

... } else { /* parent */ ...pipe2ways.c
    printf("Parent pipe descriptors:
           pfd1[0] %d, pfd2[0] %d,
           pfd1[1] %d, pfd2[1] %d\n",
           pfd1[0],pfd2[0],pfd1[1],pfd2[1]);
    close(pfd1[0]); close(pfd2[1]);
    sleep(1);
    printf("Message? ");          // read a msg
    fgets(outBuf,SIZE,stdin);     // from user
    write(pfd1[1], outBuf, strlen(outBuf));
    close(pfd1[1]);
    /* get the message from the child */
    nread = read(pfd2[0],inBuf,SIZE);
    inBuf[nread]='\0';
    close(pfd2[0]);
    printf("Parent received: %s\n",inBuf);
    wait();
}
exit(0);
}

```

?



pipe2ways.c: Esempio di Output

```
[valter../prove]$ ./pipe2ways
```

```
Parent pipe descriptors: pfd1[0] 3, pfd2[0] 5, pfd1[1] 4, pfd2[1] 6
```

```
Child pipe descriptors: pfd1[0] 3, pfd2[0] 5, pfd1[1] 4, pfd2[1] 6
```

```
Message? Messaggio di prova
```

```
Child received: Messaggio di prova
```

```
Parent received: Messaggio di prova
```

Esercizio: riscrivere un programma che esegua comunicazione bidirezionale tramite pipe tra tre processi: il primo legge un messaggio dall'utente e lo invia al secondo che una volta ricevuto lo inoltra al terzo. Questo lo fa "rimbalzare" al secondo che lo restituisce al primo.

Pipes ed `exec ()`

- I due processi comunicanti devono conoscere i descrittori della pipeline
- Finora lo sdoppiamento della tabella dei descrittori di file dal padre al figlio ha permesso di sfruttare le variabili locali
- Supponiamo di voler implementare: `$ ls | wc -w`
 - il padre esegue `fork()`
 - il padre esegue `exec()` per `'ls'`
 - il figlio esegue `exec()` per `'wc -w'`
- La `exec ()` non cambia la tavola dei descrittori ma si perdono le variabili locali: come fare per accedere alla stessa pipe se gli unici riferimenti comuni erano in variabili locali con lo stesso valore?

dup e dup2

```
include <unistd.h>
```

```
int dup(int oldfd);
```

```
int dup2(int oldfd, int newfd);
```

- dup crea una nuova copia del descrittore (usando il più piccolo descrittore disponibile)
- dup2 crea una copia del descrittore oldfd in newfd
 - da quel momento in poi coindividono lo stesso file
 - modifiche fatte tramite uno si leggono anche tramite l'altro

dup2.c

```
#include <stdio.h>

#define SIZE 1024

int main(int argc, char** argv){
    int pfd[2], pid;

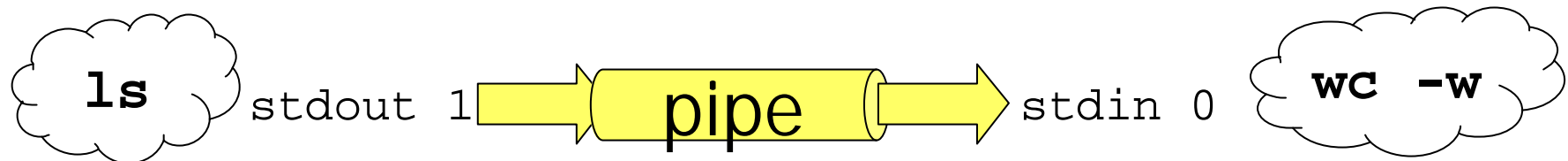
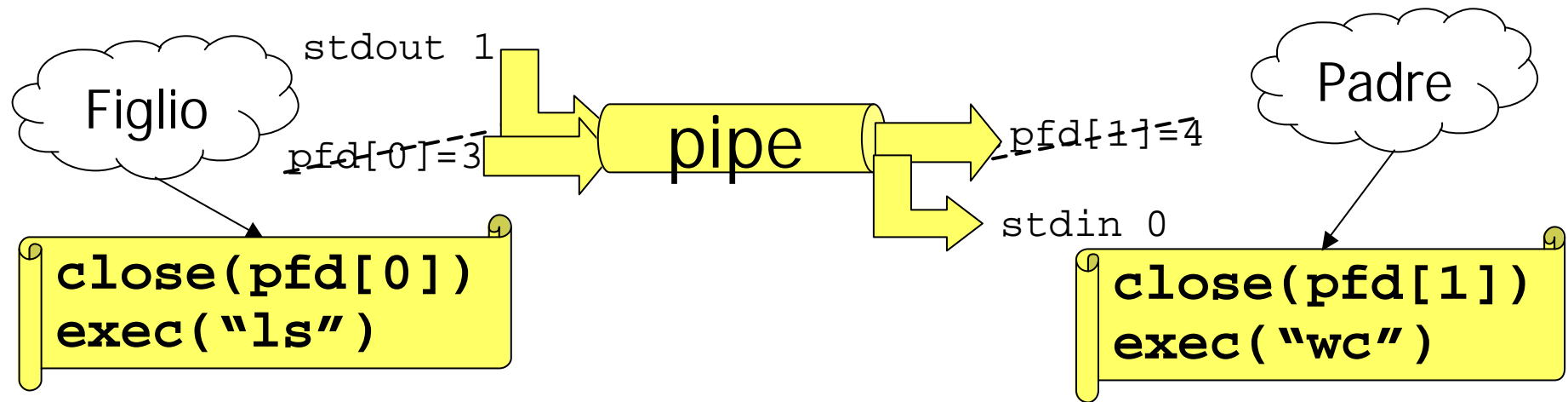
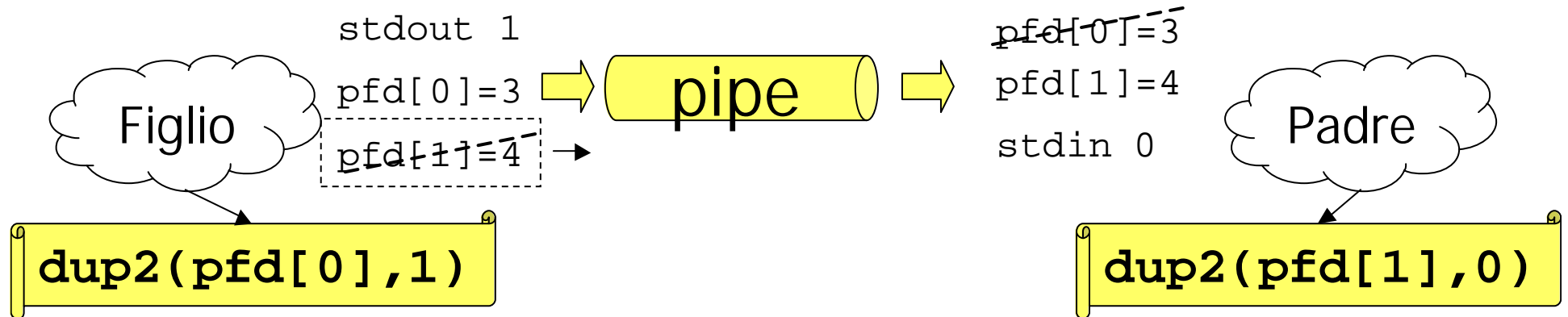
    if (pipe(pfd) == -1) {
        perror("pipe() failed");
        exit(1);
    }
    if ((pid=fork()) < 0 ) {
        perror("fork() failed");
        exit(2);
    }
    ...
}
```

Esempio di Output

```
[valter../prove]$ ./dup
[valter../prove]$          26
```

```
...
if (pid==0) { /* child */
    close(pfd[1]);
    → dup2(pfd[0],0);
    close(pfd[0]);
    execlp("wc","wc","-w",NULL);
    perror("wc failed");
    exit(3);
} else { /* parent */
    close(pfd[0]);
    → dup2(pfd[1],1);
    close(pfd[1]);
    execlp("ls","ls",NULL);
    perror("ls failed");
    exit(4);
}
exit(0);
}
```

dup2 e Pipelining



Esercizi

Esercizio: riscrivere un programma che esegua il comando di shell `ls -R | grep <pat>` dove *<pat>* è un pattern inserito dall'utente

Esercizio: scrivere un programma che esegua il comando di shell `ls | sort | grep <pat>` con tre processi distinti

dup e Redirezione

- Esempio di redirezione dello *stdout* al file `/tmp/output`:

...

```
close(1);
```

```
open( "/tmp/output", O_WRONLY, 0 );
```

...

- Esempio di redirezione dello *stdout* al canale di scrittura della pipe:

...

```
pipe( fd );
```

```
close(1);
```

```
dup( fd[1] );
```

```
close( fd[1] );
```

...

Esercizio: riscrivere `dup2.c` utilizzando solo `dup`

Popen

man 2 popen

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

```
FILE *popen(const char* cmd, const char* type);
```

- Restituisce un normale stream che va chiuso con:

```
pclose(FILE *fp)
```

- crea una pipeline monodirezionale mediante una sequenza:

1. `fork()`, per creare un processo figlio

2. `exec()`, per invocare una shell nel figlio

3. `cmd` viene eseguito

- Lo stream:

- In lettura restituisce lo *stdout* del comando invocato
- Utilizzato in scrittura scrive sullo *stdin* del comando invocato

popen.c

```
#include <stdio.h>
#define MAXSIZE 256
main (int argc, char *argv[]) {
    char line[MAXSIZE];
    FILE *fp;
    if ((fp=popen("ls","r"))==NULL) {
        perror("popen() error");
        exit(1);
    }
    while (fgets(line,MAXSIZE,fp)!=NULL)
        printf("%s",line);
    pclose(fp);
    exit(0);
}
```

Esercizio: scrivere un programma che attraverso l'utilizzo del comando di shell `wc` e la funzione `popen()` conteggi il numero di parole di una frase inserita dall'utente

Conclusioni sulle Pipes

- La comunicazione tramite pipe è monodirezionale; per comunicare nei due sensi sono necessarie due pipe
- La chiamata `pipe()` viene effettuata prima della `fork()`
- Perché due processi possano comunicare tramite una pipe anonima, devono condividere un antenato comune
- Le named pipe (FIFO) eliminano questa necessità perché le pipe hanno un nome esplicito a cui far riferimento

Named Pipes o FIFO

- In tutto analoghe alle pipe anonime tranne per il fatto che viene creato un file speciale nel file system
- Un qualsiasi processo può referenziarle direttamente utilizzando il nome del file (named pipes)
- I file speciali di tipo FIFO utilizzano blocchi *diretti* e buffer circolare

```
[valter../prove]$ mknod myfifo p
```

```
[valter../prove]$ ls -l myfifo
```

```
prw-rw-r-- 1 valter valter 0 dic 10 19:38 myfifo|
```

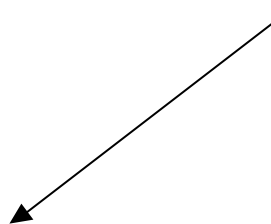
Creare FIFO: mknod

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

```
int mknod(char* pathname, mode_t mode, dev_t dev);
```

- restituisce -1 in caso di errore e 0 in caso di successo
- pathname è il nome del file speciale, incluso il path
- mode specifica i permessi e tipo del nuovo file
- (dev indica il minor ed il major number per file di tipo S_IFCHR o S_IFBLK)

mode:
S_IFREG
S_IFCHR
S_IFBLK
S_IFIFO
...*permessi*



man 2 mknod
man 1 mknod

umask: Maschera di Permessi per i Nuovi File

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

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

man 2 umask

```
mode_t umask(mode_t mask);
```

- I permessi assegnati ad ogni nuovo file, e quindi anche a quelli creati con la `mknod()`, vengono filtrati dalla *umask* di sistema
- *final_mask = requested_permission & ~ original_umask*
- Per evitare che la *umask* di sistema cambi i permessi desiderati si utilizza la `umask()` prima di creare la FIFO:

```
umask( 0 );
```

Apertura delle FIFO

Creazione di una FIFO:

```
umask( 0 );
```

```
mknod( "/tmp/myfifo", S_IFIFO|0666, 0 );
```

- crea un file speciale nella dir `/tmp` di nome `myfifo` con i permessi settati a `0666`, ovvero lettura e scrittura per tutti (*original_umask* è posta a 0)
- Le FIFO devono essere esplicitamente aperte dalla `open()`
- Al contrario le pipe anonime risiedono nel kernel e non hanno bisogno della chiamata `open()`

fifoserver.c

```
/** ***** fifoserver.c ***** */
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFOFILENAME "myfifo"
#define SIZE 80

int main(int argc, char *argv[]) {
    FILE *fp;
    char buf[80];

    /* create the FIFO if it does not exist */

    umask(0);
    mknod(FIFOFILENAME, S_IFIFO|0666,0);

    while(1) {
        fp=fopen(FIFOFILENAME,"r");
        fgets(buf,SIZE,fp);
        printf("Received: %s\n",buf);
        fclose(fp);
    }
    exit(0);
}
```

fifoclient.c

```

/***** fifoclient.c *****/
#include <stdio.h>
#include <stdlib.h>

#define FIFOFILENAME "myfifo"

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

    if (argc!=2) {
        printf("Usage: fifoclient <string>\n");
        exit(1);
    }

    if ((fp=fopen(FIFOFILENAME, "w" )) ==NULL) {
        perror("fopen() failed");
        exit(2);
    }
    fputs(argv[1],fp);
    fclose(fp);
    exit(0);
}

-----
[valter../provel$ gcc fifoserver.c -o fifoserver
[valter../provel$ gcc fifoclient.c -o fifoclient
[valter../provel$ ./fifoserver &
[2] 5091
[valter../provel$ ./fifoclient "messaggio di prova"
Received: messaggio di prova

```

`read()` e `write()` con FIFO

- Un processo che apre una FIFO in lettura si blocca fino a che qualche altro processo non la apre in scrittura
- Si può disabilitare tale comportamento utilizzando il flag `O_NONBLOCK` durante l'`open()` l'apertura della FIFO
- `read()` e `write()` bloccanti

Esercizi

Esercizio: scrivere due programmi diversi, server e client, tali che il client legge continuamente dei numeri dall'utente e li invia al server. Questo ne stampa il quadrato.

Esercizio: scrivere due programmi diversi, server e client, tali che il server scrive continuamente, ad intervalli di tempo regolari, l'orario di sistema tramite il comando di shell `date`.

Il client accetta input dall'utente e permette di interrompere temporaneamente o permanentemente il server; inoltre l'utente può specificare al client l'intervallo di tempo, in sec, con cui il server deve continuare a scandire l'orario.