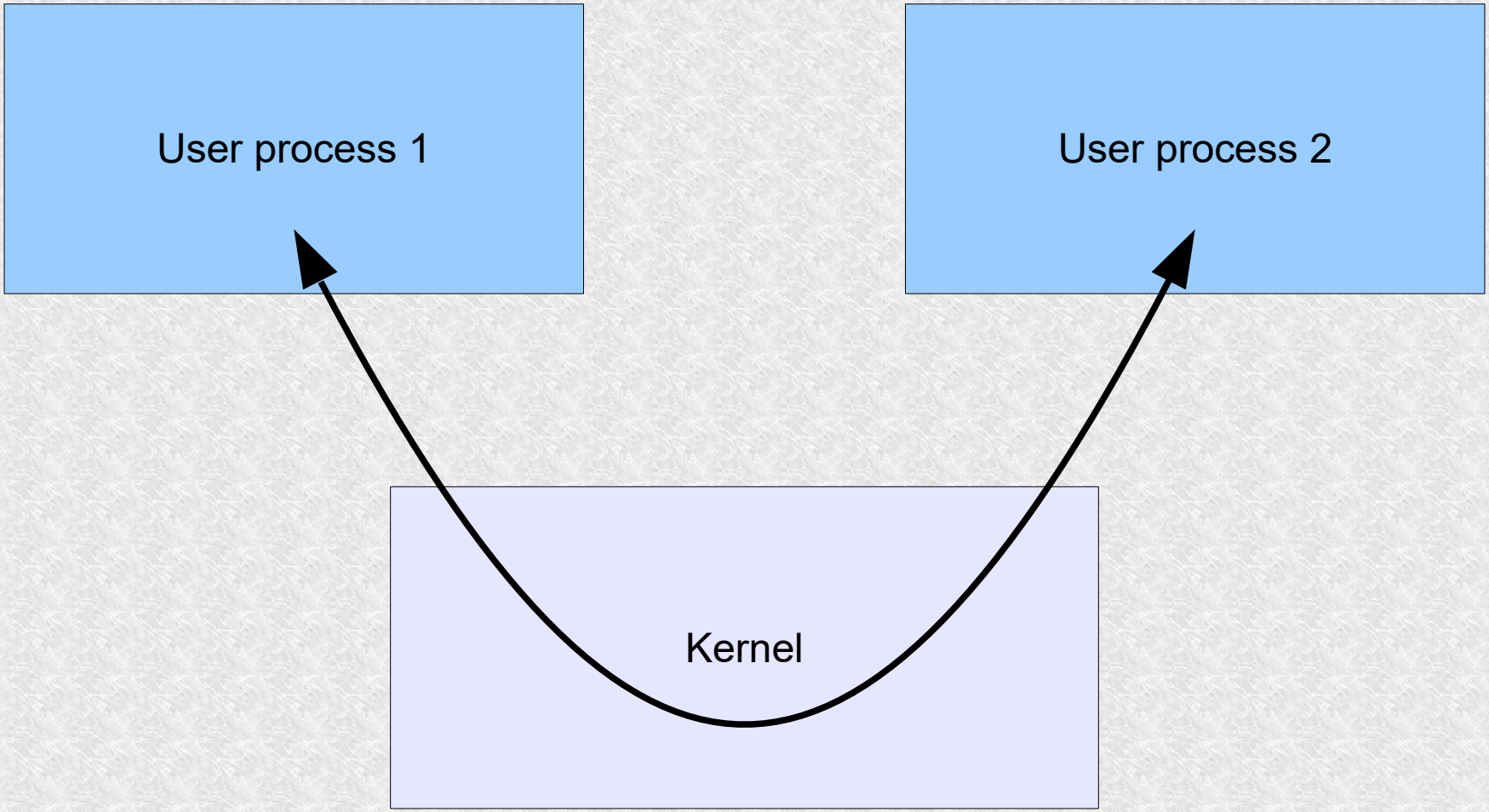
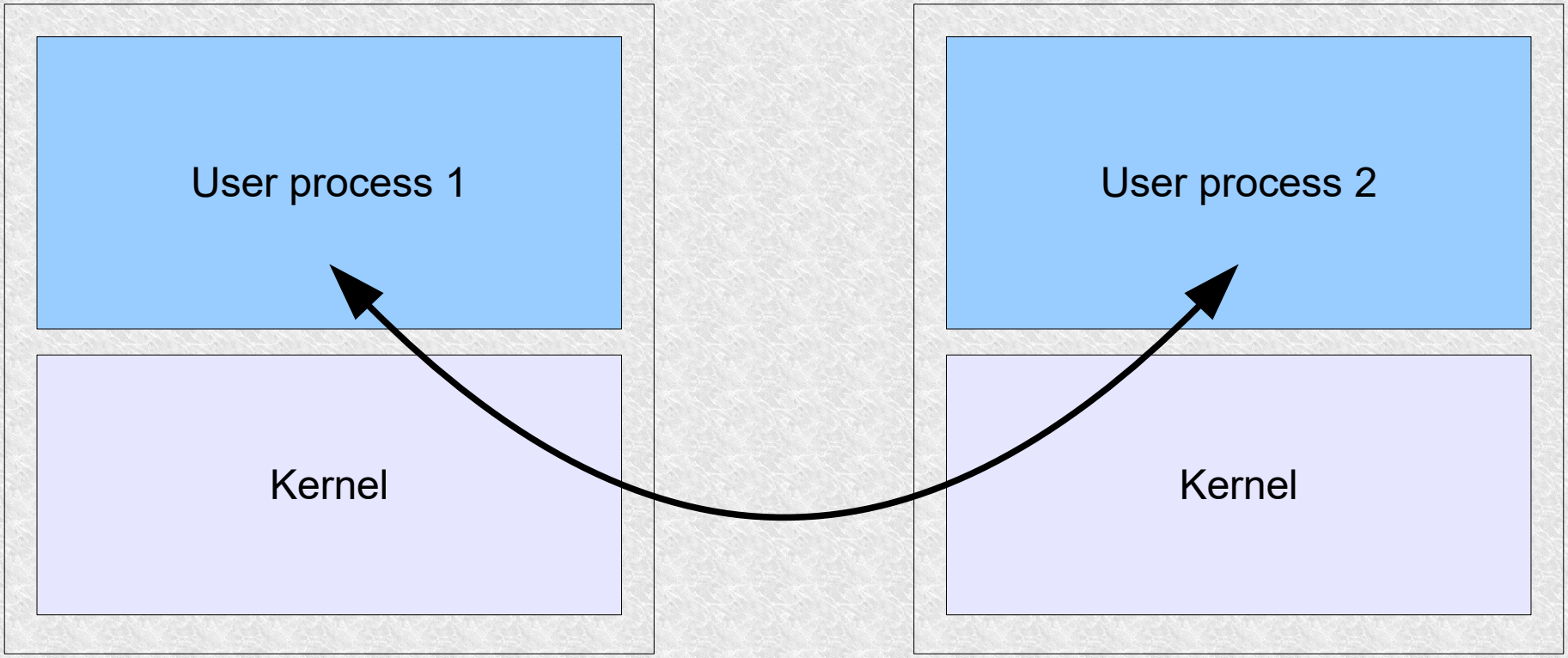
Concurrent Programing

In Linux/Unix

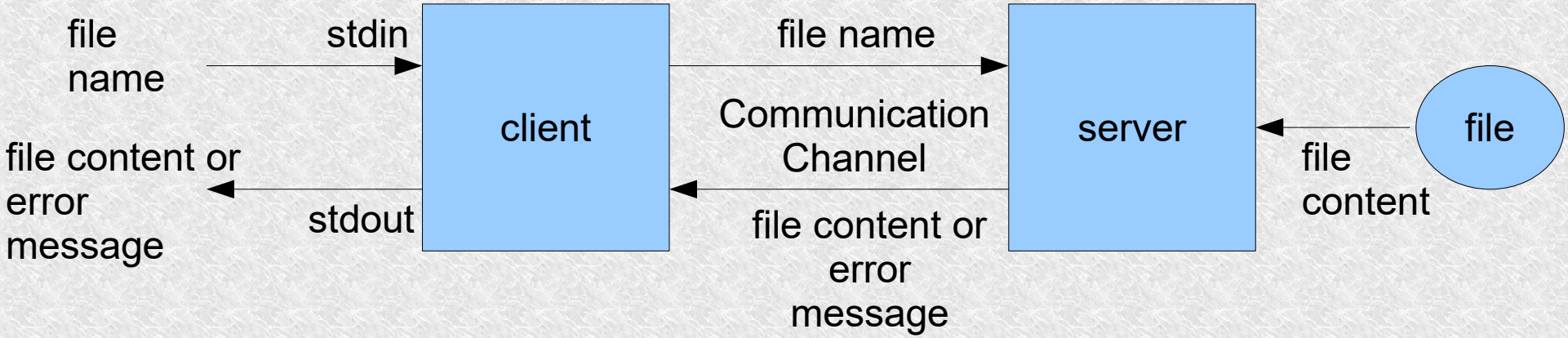
Communication



Communication



Communication



- the client gets the filename from stdin
- sends it to the server
- the server opens the file
- it is sent by the client (or info about the error)
- the client displays the content on stdout

PIPE links

- Unnamed lines (PIPEs) allow data to flow in one direction
- we create it using functions

```
int pipe(int *filedes);
```

- *filedes* is a two-element array
- *filedes[0]* is the file descriptor open for reading
- *filedes[1]* is the file descriptor open for writing

PIPE links

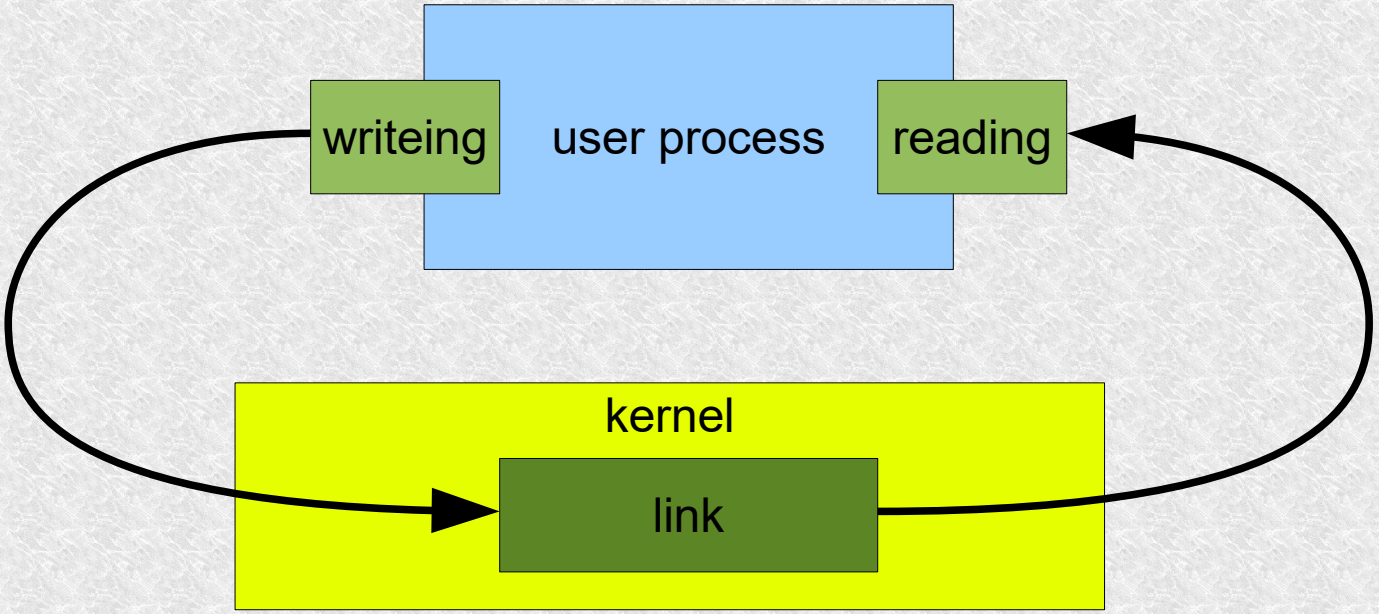
We usually use communication links to communicate between two processes, but the example below will be on the one

```
main()
{
int pipefd[2],n;
char buff[100];
if (pipe(pipefd) <0)
    perror("pipe error\n");
printf("read fd = %d, write fd = %d\n",pipefd[0],pipefd[1]);
if (write(pipefd[1],"hello world\n",12) !=12)
    perror("write error\n");
if ((n = read(pipefd[0],buff,sizeof(buff))) <=0)
    perror("read error\n");
write(1, buff, n); /* filedes=1 it is stdout */
exit(0);
}
```

PIPE links

it could be that "*hello world"* will come before "*read fd* = "... etc because printf uses a buffer and it is flushed at the end of the program.

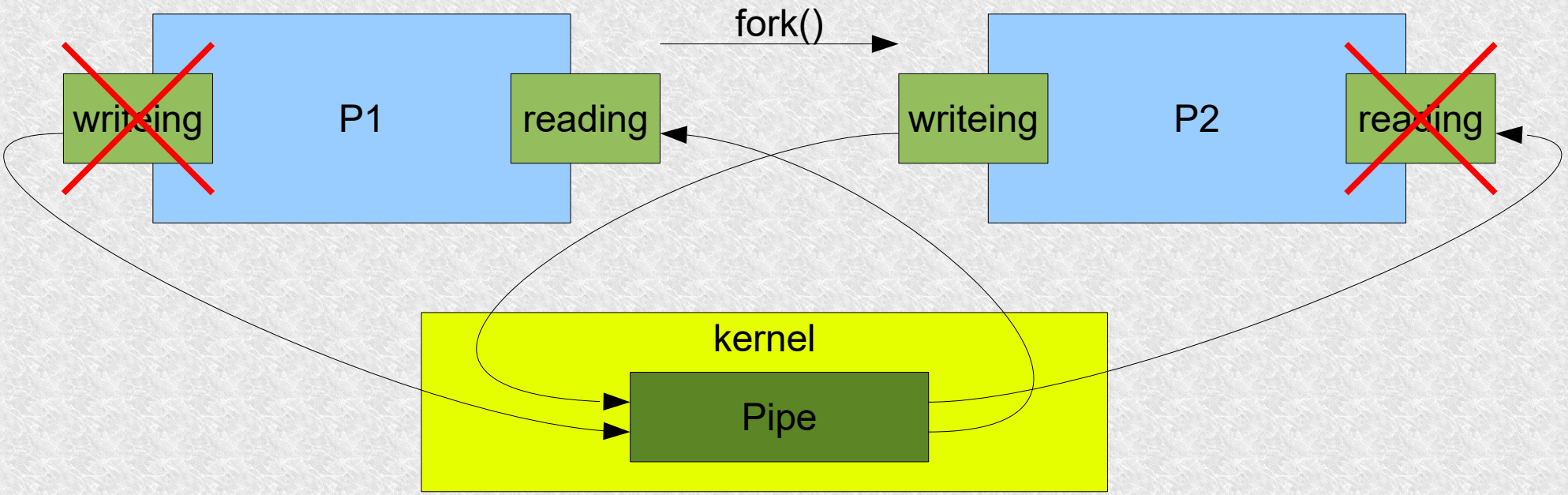
We created a link more or less like this:



PIPE link

To create a link between two processes, we must follow the following algorithm

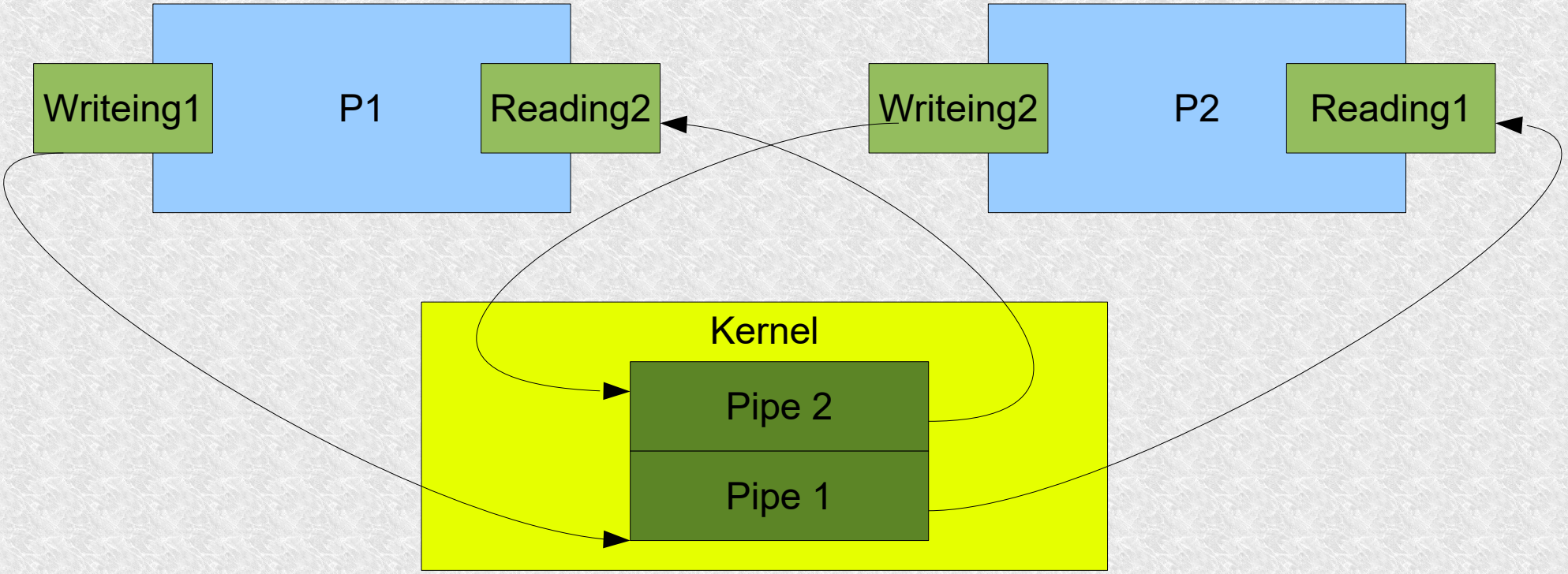
- the parent process creates the link
- then forks, this is where the descriptors are copied
- the parent process closes one writing or reading descriptor
- child process closes one reading or writing descriptor
  (Notice. When parent close reading, child must close writeing and vice versa)



PIPE link

these links are one-way, if we want communication in two directions, the following steps must be performed

- create two links (link1, link2)
- call a *fork*
- ancestor closes link 1 for reading;
- ancestor closes link 2 for writing;
- child closes link 1 for writing;
- child closes link 2 for reading;



PIPE link

standard I/O library contains functions creating a communication link and initiating execution of a second process:

```
#include <stdio.h>
FILE *popen(char *command, char *type);
```

- *command* stands for command prompt
- type ”r” or ”w”
- returns input or output, or NULL on failure

the function to close the stream opened by popen is:

```
#include <stdio.h>
int pclose(FILE *stream);
```

PIPE link

```
#include <stdio.h>
#define MAXLINE 1024
main ()
{
char line[MAXLINE],command[MAXLINE+10];
int n;
FILE *fp;

/* get file name from stdandard input */
if (fgets(line, MAXLINE, stdin) == NULL)
    perror("client: file name read error\n");

sprintf(command, "cat %s",line);
if ((fp = popen(command, "r")) == NULL)
    perror("error popen\n");
/* get data from file end send it to standard output */
while ((fgets(line, MAXLINE,fp)) != NULL)
    {
    n = strlen(line);
    if (write(1, line, n) != n) /* fd 1 = stdout */
    perror("client: write data error\n");
    }
if (ferror(fp)) /* checks the error flag for stream, return a non-zero value if
it is set */
    perror("blad fget\n");
pclose(fp);
exit(0);
}
```

PIPE link

The biggest disadvantage of PIPE links is that we can only create them between related processes.

FIFO links

Other - named links.

- They are similar to communication links
- they allow unidirectional flow
- the first byte read from the fifo will be the first one written there.
- a FIFO queue, unlike an unnamed link, has a name
- thanks to this, unrelated processes can access the same link

FIFO links

To create a FIFO is used the function.

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

- pathname – normal unix path name
- mode – access mode word which will be logically summed with the S_IFIFO tag from the file <sys/stat.h>
- dev – device, you can skip in case of FIFO queues

you can also use a command

```
/bin/mknod name p
```

once it is created, it must be opened for reading or writing, e.g. via *open* or *fopen*, *freopen*.
You can set the appropriate O_NDELAY tag
The result of setting the O_NDELAY marker is in the table:

FIFO links

Situation	O_NDELAY flag was not set	O_NDELAY flag was set
Open a FIFO for reading only, and no process has opened that queue for writing	Wait for the process to open the FIFO for writing	return immediately without error
open FIFO for writing only. No process has opened this queue for reading	wait for the process to open the FIFO queue for reading	return immediately, signal error, put ENXIO constant in errno
read from communication link (or FIFO), there are no data	wait until the data in the link (queue) appears or it is not open for writing for any process; pass zero as the value of the function, if no process has opened the link (queue) for writing, otherwise pass the number of data	return immediately, pass zero as the function value
write, communication link (or FIFO) full	wait until there is space, then write data	return immediately, pass zero as the function value

FIFO links

Rules:

- if the process requests to read a smaller amount of data than there is in the link, it will read as much as it wants and the rest by the way
- if a process requests more data than it is, it will get as much data as is in the link.
- if there is no data in the link and no process has opened it for writing, the read value will be zero, signifying the end of the file.
- if the process writes a smaller portion of data than the capacity of the link or FIFO (usually 4096B) then data integrity will be guaranteed
- if more than that, it is not guaranteed, because a second process writing to the same link may interleave the data
- if the process calls write to a link that no other process has opened for reading
    - it will get a SIGPIPE
    - *write* will return 0
    - *errno* will have a value EPIPE
    - if the process cannot handle the SIGPIPE signal it will be terminated

FIFO links

Imagine a daemon waiting for customer data on some link open for reading, when the client finishes writing and ends by eg exit, the link would have to be opened again and waited.

- the daemon should open the same link for both reading and writing
- he will never use to write but thanks to that he will not get EOF due to lack of a client to write.

FIFO links

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
extern int errno;
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define PERMS 0666

main()
{
int readfd, writefd;
/* open FIFO queues we assume that the server has already created them */
if ((writefd = open(FIFO1, 1)) < 0)
    perror("Client: can't open fifo1 for writeing");
if ((readfd = open(FIFO2, 0)) < 0)
    perror("Client: can't open fifo2 for reading");
client(readfd,writefd);
close(readfd);
close(writefd);
/* remove FIFO queue */
if (unlink(FIFO1) < 0)
    perror("Parent can't remove FIFO1 %s",FIFO1);
if (unlink(FIFO2) < 0)
    perror("Parent can't remove FIFO2 %s",FIFO2);
exit(0);
}
```

FIFO links

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
extern int errno;
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define PERMS 0666
main()
{
int readfd, writefd;
/* create FIFOs and then open them - one for reading  */
/* the other for writing */
if ((mknod(FIFO1, S_IFIFO | PERMS, 0) < 0) && (errno !=EEXIST))
  perror("can't create fifo 1: %s\n", FIFO1);
if ((mknod(FIFO2, S_IFIFO | PERMS, 0) < 0) && (errno !=EEXIST))
    {
    unlink(FIFO1);
    perror("can't create fifo 2: %s\n", FIFO2);
    }
if ((readfd = open(FIFO1 ,0)) < 0)
  perror("Server: can't open fifo1 for reading");
if ((writefd = open(FIFO2,1)) < 0)
  perror("Server: can't open fifo2 for writeing");
server(readfd,writefd);
close(readfd);
close(writefd);
exit(0);
}
```

Data streams and messages

- In the examples so far, we have used the concept of data stream
- there are no separate records or structures in it
- the system doesn't interprets what data it receives
- if we want to interpret, the reading and writing processes have to establish a common way of communicating data
- the simplest structure is the lines ending with ‘\n’
- or you can come up with more complex structures
- Structure with message type and length. For further examples, we will place this structure in the mesg.h file:

Data streams and messages

```
#define MAXMESGDATA (4096-16)
#define MESGHDRSIZE (sizeof(Mesg) - MAXMESGDATA)

typedef struct {
   int mesg_len;
   long mesg_type;
   char mesg_data[MAXMESGDATA];
} Mesg;
```

Data streams and messages

```
#include ”mesg.h” /* there is our structure*/

/* send the message using the file descriptor, the
structure fields must be filled in first by the
calling process*/
void mesg_send(int fd, Mesg *mesgptr)
{
int n;
/* prepare header */
n = MESGHDRSIZE + mesgptr->mesg_len;
if (write(fd, (char*) mesgptr, n ) !=n)
   perror(„writeing message error\n”);
}
```

Data streams and messages

```
/* get the message, using the file descriptor fill in the
Mesg structure fields and return the value of the mesg_len
field */

int mesg_recv(int fd, Mesg *mesgptr)
{
int n;
/* we get the header and check how much is still to
download*/
/* if EOF then return 0 */
if ((n = read(fd, (char*)mesgptr, MESGHDRSIZE)) == 0)
   return(0); /* end of file */
else if (n != MESGHDRSIZE)
   perror(„error in reading the message header\n”);
if ((n = mesgptr->mesg_len) > 0)
   if (read(fd, mesgptr->mesg_data, n) != n)
      perror(„error reading message data\n”);
return(n);
}
```

Data streams and messages

```
Mesg mesg;
void client(int ipcreadfd, int ipcwritefd)
{
int n;
if (fgets(mesg.mesg_data, MAXMESGDATA, stdin) == NULL)
    perror("error reading the file name\n");
n = strlen(mesg.mesg_data);
if (mesg.mesg_data[n-1] == '\n')
    n--; /*omit the newline character picked up by fgets() */
mesg.mesg_len = n;
mesg.mesg_type = 1L;
mesg_send(ipcwritefd, &mesg);

while (( n = mesg_recv(ipcreadfd, &mesg)) > 0)
    if (write(1, mesg.mesg_data, n) != n)
   perror("data write error\n");
if (n<0)
    perror("data read error\n");
}
```

Data streams and messages

```
extern int errno;

void server(int ipcreadfd, int ipcwritefd)
{
int n, filefd;
char errmesg[256];
mesg.mesg_type = 1L;
if ((n = mesg_recv(ipcreadfd,&mesgt)) <= 0)
    perror("server: error reading the file name\n");
mesg.mesg_data[n] = '\0';
if ((filefd = open(mesg.mesg_data, 0)) < 0)
    {   /* prepare an error message */
    sprintf(errmesg,":can't open %s\n",
strerror(errno));
    strcat(mesg.mesg_data,errmesg);
    mesg.mesg_len = strlen(mesg.mesg_data);
    mesg_send(ipcwritefd,&mesg);
    }
else //...cdn
```

Data streams and messages

```
//...cd

    {
    while (( n = read(filefd, mesg.mesg_data, MAXMESGDATA)) > 0)
   {
   mesg.mesg_len = n;
   mesg_send(ipcwritefd, &mesg);
   }
    close(filefd);
    if (n < 0)
   perror("server: reading error");
    }
/* send an empty message which means the processing is finished */
mesg.mesg_len = 0;
mesg_send(ipcwritefd,&mesg);
}
```

Namespaces

- links are unnamed but queues can be identified by Unix path names.
- the set of possible names for a given type of interprocess communication is called a namespace.
- all types of interprocess communication (except links) are through names.
- On next slide. List of naming conventions for different types of interprocess communication:

Namespaces

Type of inter-process communication	Namespace	Identification
communication link	(no name)	file descriptor
FIFO	path name	file descriptor
message queue	key key_t	identifier
shared memory	key key_t	identifier
semaphore	key key_t	identifier
socket – Unixa domain	path name	file descriptor
socket – other domain	(depends on domain)	file descriptor

Keys key_t

- these are identifiers typically 32b uniquely identifying message queues, shared memory, or semaphores.
- You can create yourself, e.g. 1234L
- But in more serious applications, it is better to use:

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, char proj);
```

- based on the path name and project number (*proj* is the number 8b), it creates an almost unique key for us. (it is a 32-bit number and is created on the basis of inode (32b) project number (8b) and the so-called small file system device number (8b))

Keys key_t

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

main()
{
key_t key;
key = ftok("/tmp/somename",4); //the file must
exist because if not, ftok will return -1
printf("ftok %d\n",key);
}
```