Pipes are a simple, synchronised way of passing information between processes.

Differences between files and pipes:

- **pipes have the bounded size** - the maximum capacity of a pipe is referenced by the constant `PIPE_BUF` (usually limited to 5120 bytes.)

- **sequential access** – one can only read or write from and to the pipe, the pointer of the current position can not be moved (`lseek` is unacceptable)

- `write` appends data to the input of a pipe while `read` reads any data from output of a pipe but:

- data that is read are **removed** from the pipe

- if the pipe is empty and at least one descriptor for reading is open, then the read is blocked until some data are written to the pipe or until the pipe descriptor is closed

- the process is blocked if it wants to write and the pipe is full

Pipes can be divided into two categories:

- unnamed pipes

- named pipes

Named pipes exist as directory entries and they can be used by unrelated processes provided that the processes know the name and location of the entry.

Unnamed pipes

Unnamed pipes may be only used with related processes (parent/child or child/child having the same parent). They exist as long as their descriptors are open.

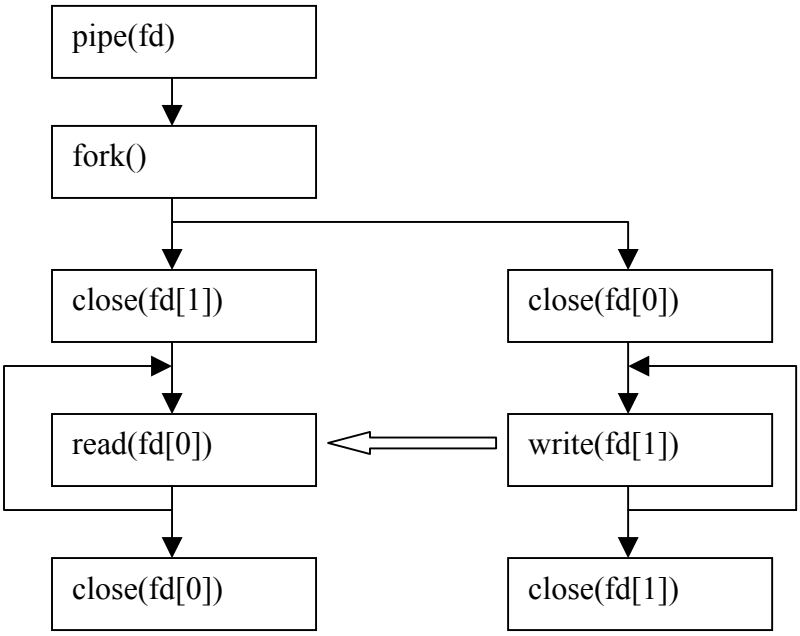
```
#include <unistd.h>

int pipe(int fd[2])
```

```
RETURNS: success : 0
         error: -1
```

1. `fd [0]` – descriptor used for reading
2. `fd [1]` – descriptor used for writing

Function creates the pipe and opens it for reading and writing.

Scheme of communication through pipe:



Example 1 Communication via an unnamed pipe

```
main() {
int fd[2];

if (pipe(fd) == -1){
     perror("Create pipe error");
     exit(1);
}

switch(fork()){
case -1:
     perror("Failure on creating a process ");
     exit(1);
case 0:
     close (fd[0]);                          //child process
     if (write(fd [1], "Hallo!", 7) == -1){
          perror("Writting to pipe error");
          exit(1);
     }
     exit(0);
default: {                                    //parent process
     char buf[10];
     close (fd[1]);
     if (read(fd[0], buf, 10) == -1){
          perror("Reading from pipe error");
          exit(1);
     }
     printf("Data from the pipe: %s\n", buf);
}
}
}
```

Example2 The problem of reading from the empty pipe

```
main() {
int fd[2];


pipe(fd);
if (fork() == 0){   // child process
  write(fd [1], "Hallo!", 7);
  exit(0);
}
else {              //parent process
  char buf[10];
  read(fd [0], buf, 10);
  read(fd[0], buf, 10);
  printf("Data from pipe: %s\n", buf)
}
}
```

Example 3 : Displaying result of `ls` in capital letters

```
#define MAX 512
main(int argc, char* argv[]) {

int fd[2];

if (pipe(fd) == -1){
     perror("Creating pipe");
     exit(1);
}

switch(fork()){
case -1:
     perror("Creating a process ");
     exit(1);
case 0:
     dup2(fd[1], 1);
     execvp("ls", argv);
      perror("program ls");
     exit(1);
default: {
      char buf[MAX];
     int nb, i;
      close(fd[1]);
     while ((nb=read(fd[0], buf, MAX)) > 0){
          for(i=0; i<nb; i++)
               buf[i] = toupper(buf[i]);
          if (write(1, buf, nb) == -1){
          perror ("Writting to stdout");

      exit(1);
}
}
if (nb == -1){
     perror("Reading from pipe");
     exit(1);
}
}
}
```

Example 4 : `ls |tr 'a-z' 'A-Z'` with unnamed pipes

```
main(int argc, char* argv[]) {
int fd[2];

if (pipe(fd) == -1){
  perror("Creating pipe");
  exit(1);
}
switch(fork()){
 case -1:
     perror("Creating process");
     exit(1);
 case 0:                 // child process
     dup2(fd[1], 1);
     execvp("ls", argv);
     perror("command ls");
     exit(1);
 default: {              // parent process
     close(fd[1]);
     dup2(fd[0], 0);
     execlp("tr", "tr", "a-z", "A-Z", 0);
     perror("command tr");
     exit(1);
 }
}
}
```

Named pipes – FIFO

- The named pipe has a directory entry.
- The directory entry allows using the FIFO for any process which knows its name, unlike unnamed pipes which are restricted only to related processes.

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

int mkfifo(char * path, mode_t mode)
```

```
RETURNS: success : 0
         error: -1
```

`path` points to the pathname of a file
`mode` access rights

Function creates the pipe but **does not open it**

```
#include <fcntl.h>
main(){
    mkfifo("abc", 0600);
    open("abc", O_RDONLY);
}
```

`open` is blocked until another process opens FIFO in a complementary manner!

Example 5 : Creating and using named pipe

```
#include <fcntl.h>

main() {
 int fd;

 if (mkfifo("/tmp/fifo", 0600) == -1){
     perror("creating FIFO");
     exit(1);
 }
 switch(fork()){
   case -1:
     perror("Creating process");
     exit(1);
   case 0:
     fd= open("/tmp/fifo", O_WRONLY);
     if (fd== -1){
      perror("Opening FIFO for writing");
     exit(1);
     }
     if (write(fd, "Hallo!", 7) == -1){
          perror("writing to FIFO");
          exit(1);
      }
     exit(0);
  default: {
     char buf[10];
     fd = open("/tmp/fifo", O_RDONLY);
      if (fd== -1){
          perror("Opening FIFO for reading");
          exit(1);
      }
     if (read(fd, buf, 10) == -1){
          perror("Reading from FIFO");
          exit(1);
     }
     printf("Data read from FIFO: %s\n", buf);
      }
     }
}
```

Example 6 : `ls |tr 'a-z' 'A-Z'` with named pipes

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

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

 if (mkfifo("/tmp/fifo", 0600) == -1){
     perror("Tworzenie kolejki FIFO");
     exit(1);
 }
 switch(fork()){
     case -1:
          perror("Creating process");
          exit(1);
     case 0:
          close(1);
          fd= open("/tmp/fifo", O_WRONLY);
          if (fd == -1){
               perror("Opening FIFO for writing");
               exit(1);
          }
          else if (fd!= 1){
              fprintf(stderr, "Uncorrect write descriptor \n");
              exit(1);
          }
          execvp("ls", argv);
          perror("Running ls command");
          exit(1);
      default: {
          close(0);
          fd= open("/tmp/fifo", O_RDONLY);
          if (fd== -1){
               perror("Opening FIFO for reading");
               exit(1);
          }
          else if (fd!= 0){
              fprintf(stderr, "Uncorrect write descriptor \n");
              exit(1);
          }
          execlp("tr", "tr", "a-z", "A-Z", 0);
          perror("Running tr command");
          exit(1);
     }
   }
}
```

Common failures

Example 7 : Failuers - unnamed pipes

```
#define MAX 512

main(int argc, char* argv[]) {
  int fd[2];
  if (pipe(fd) == -1){
          perror("Tworzenie potoku");
          exit(1);
  }
  if (fork() == 0){
     dup2(fd[1], 1);
     execvp("ls", argv);
     perror("Running ls");
     exit(1);
  }
  else {
     char buf[MAX];
     int nb, i;
     close(fd[1]);
     wait(0);
     while ((nb=read(fd[0], buf, MAX)) > 0){
          for(i=0; i<nb; i++)
          buf[i] = toupper(buf[i]);
          write(1, buf, nb);
 }
}
}
```

Example 8 : Failuers - named pipes

```
#include <fcntl.h>
#define MAX 512

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

  if (mkfifo("/tmp/fifo", 0600) == -1){
     perror("Creating FIFO");
     exit(1);
  }
  if (fork() == 0){
     close(1);
     open("/tmp/fifo", O_WRONLY);
     execvp("ls", argv);
     perror("Running ls");
     exit(1);
  }
  else {
     char buf[MAX];
     int nb, i;

     wait(0);
     fd= open("/tmp/fifo", O_RDONLY);
     while ((nb=read(fd, buf, MAX)) > 0){
          for(i=0; i<nb; i++)
               buf[i] = toupper(buf[i]);
               write(1, buf, nb);
     }
 }
}
```

Exercises:

Realize using unnamed and named pipes following commands:

1. `finger | cut -d' ' -f1`
2. `ls -l | grep ^d | more`
3. `ps -ef | tr -c \ \: | cut -d\: -f1 | sort | uniq -c | sort -n`