pipes

Pipes and Named Pipes allow processes to communicate using streams of data. A process can send data through/write to the pipe, and another process can receive/read data from the pipe.
A pipe acts like a synchronous finite buffer:

- if a process tries to **write** to a pipe that is **full**, it blocks;
- if a process tries to **read** from a pipe that is **empty**, is blocks.

unnamed pipes

characteristics

- communication between processes that are hierarchically related (i.e. father-child relationship)
- pipes must be created prior to creating child process
- whenever a pipe is created, two file descriptors are opened:
  - `fd[0]` : for reading
  - `fd[1]` : for writing
  - unused file descriptors should be closed
- pipes are **unidirectional**

functions

```
int pipe(int[2] fd);
// creates an unnamed pipe
```

```
- fd: where the file descriptors will be stored
- returns 0 on success, -1 on error
```

```
ssize_t read(int fd, void *buf, size_t size);
// read from pipe
```

```
ssize_t write(int fd, const void *buf, size_t count);
// write to pipe
```

```
int close(int fd);
// close a file descriptor
```

notes

- if you try to write less that `PIPE_BUF` bytes into a pipe, you are guaranteed that it will be written atomically
- if you try to write more, you have no guaranties! If several processes are writing at the same time, the writes can be interleaved
- also, when a process tries to read from a pipe, you are not guaranteed that it will be able to read everything
- you must synchronize your writes when you're writing a lot of data and ensure that you read complete messages

> **Notes**
> `PIPE_BUF` is a system variable defined in `limits.h`

- **all file descriptors are closed and some process tries to write**:
  - the write function will return -1 and set errno to `EPIPE`. Moreover, the process will receive a `SIGPIPE` signal. It can be potentially dangerous, as, by default, this signal kills the process. To prevent such a situation, ignore the signal, block it, or handle it
- **no one can write anymore but a process still wants to read**:
  - trying to read from a pipe whose writing end is closed, we can still read all the data remaining inside, and then, the read function just returns 0, to indicate end-of-file. No signal is sent nor any error occurs

- when all the descriptors of a pipe are closed, it is not possible to use it anymore and all the data inside it is lost, so it is recommended to read all of them before closing the last

output to avoid losing any data

- closing pipes incautiously is dangerous, but remaining them unnecessarily open could also pose a risk. If we do not use the pipe, but we forget to close any writing end (e.g. in some reading process), the read function can block our program still waiting for data

controlling file descriptors

- each process has a file descriptor table. By default, entries 0, 1 and 2 are: `stdin`, `stdout`, `stderr`
- each time a file is opened, an entry is added to this table. Each time a file is closed, the corresponding entry becomes available
- the process descriptor table, in fact, contains only references to the OS global file descriptor table

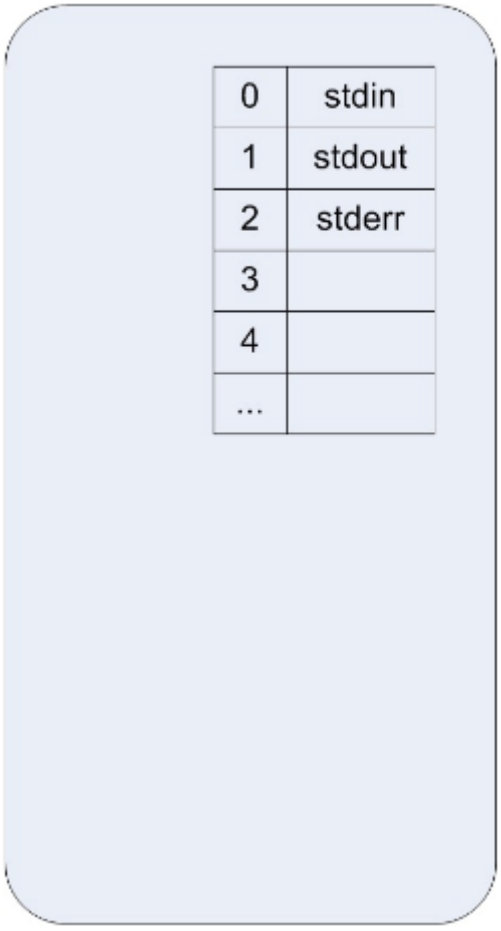
```
int dup(int fd);
// duplicates file descriptor “fd” on the first available position of the file descriptor table

int dup2(int fd, int newfd);
// duplicates file descriptor “fd” on the “newfd” position, closing it if necessary
```

> **Note**
>
> - after a file descriptor is duplicated, the original and the duplicate can be used interchangeably - they share the file pointers, the buffers, locks, etc.
> - **closing one file descriptor doesn’t close all other that have been duplicated**

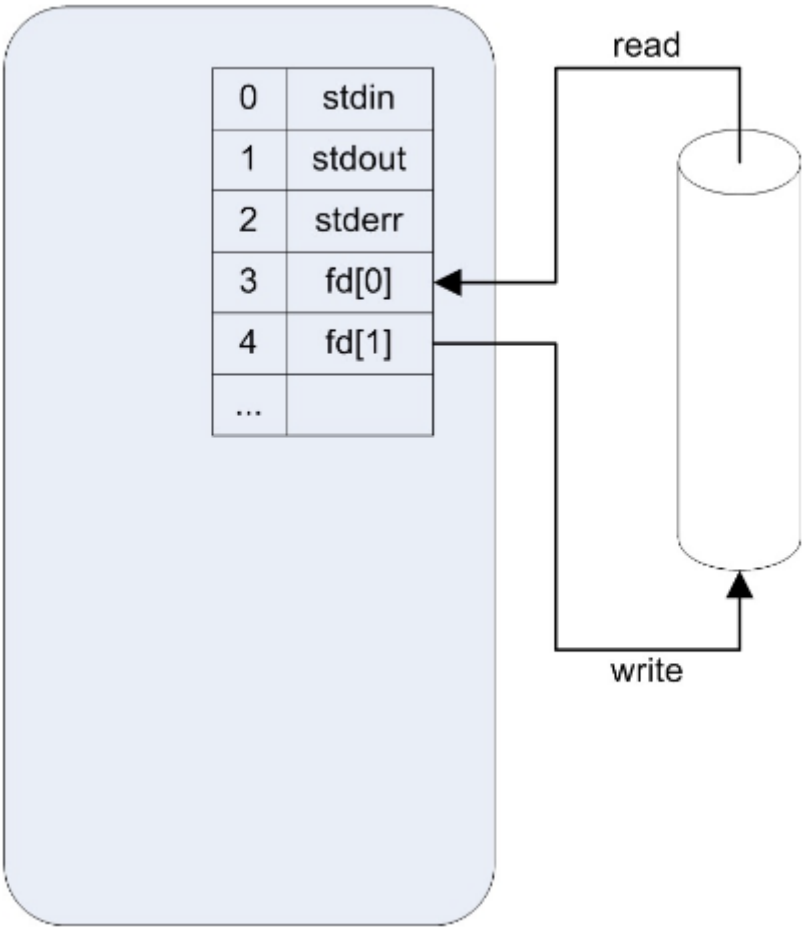
implementing a pipe

> **Note**
>
> - implementing a pipe is basically associating de standard output of a process to the standard input of another process
> - in the shell, a pipe is implemented using `|`: `ps –e | grep process`

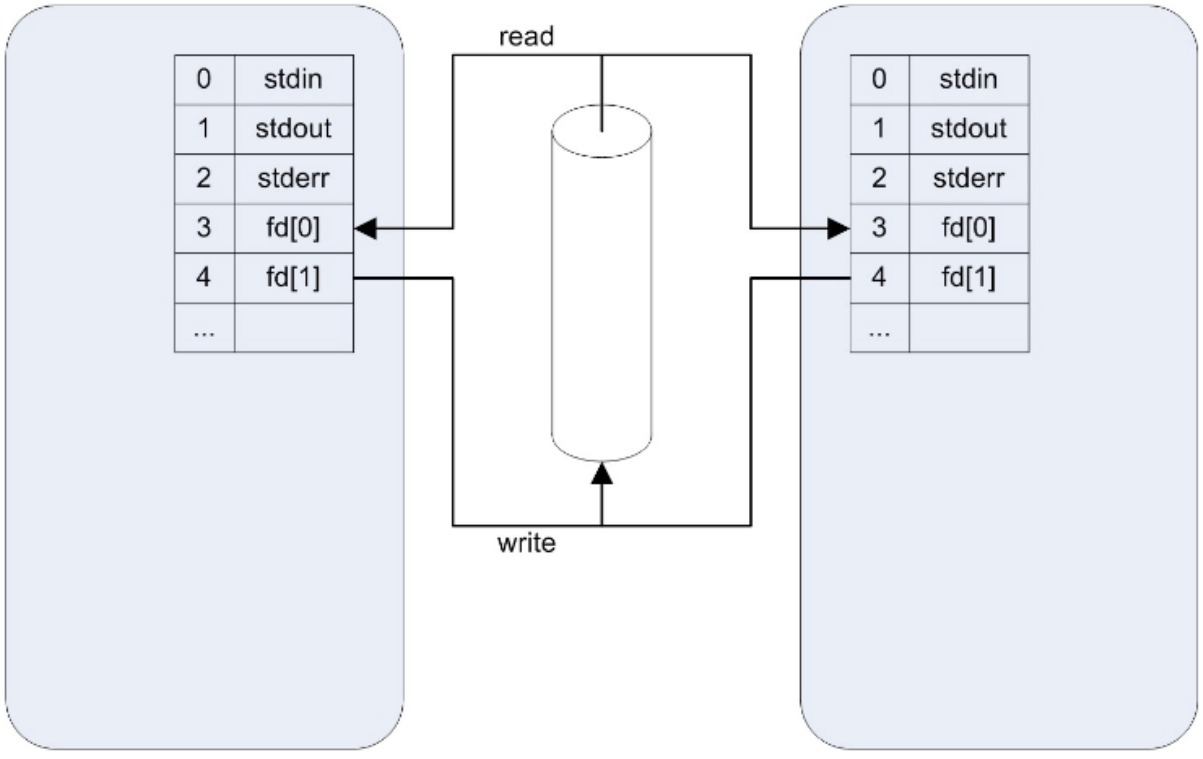
1. create a process



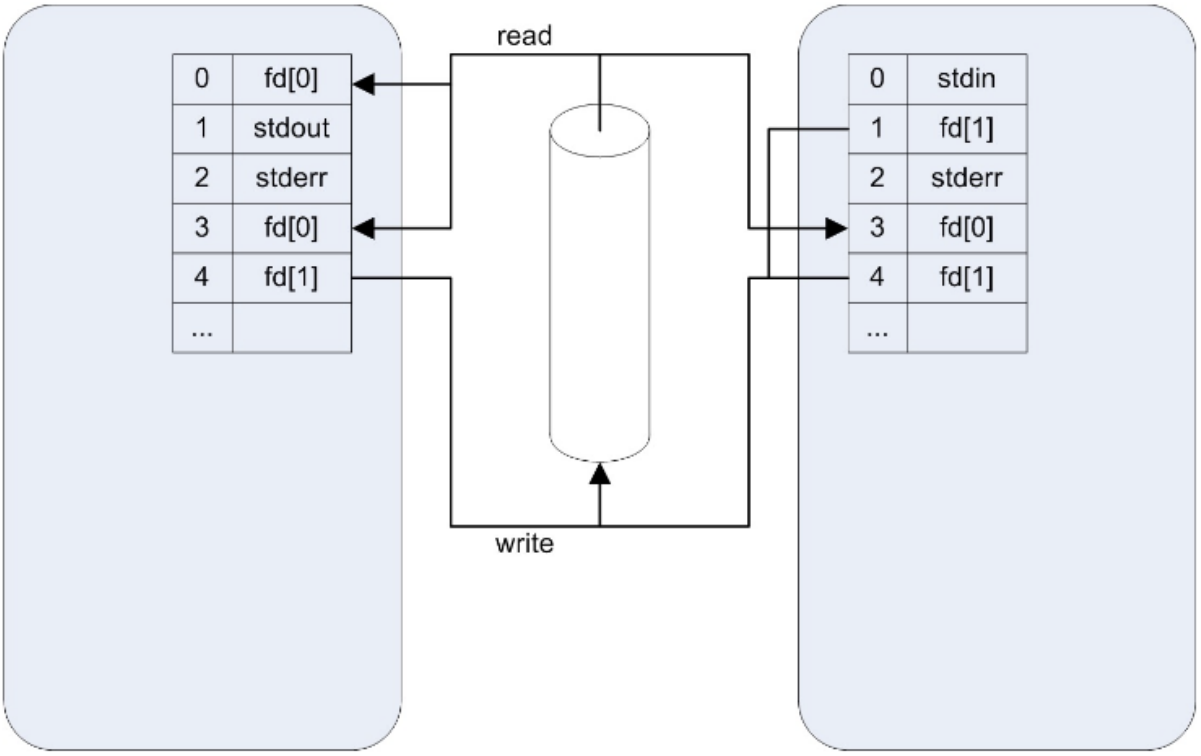
2. create a pipe (pipe())



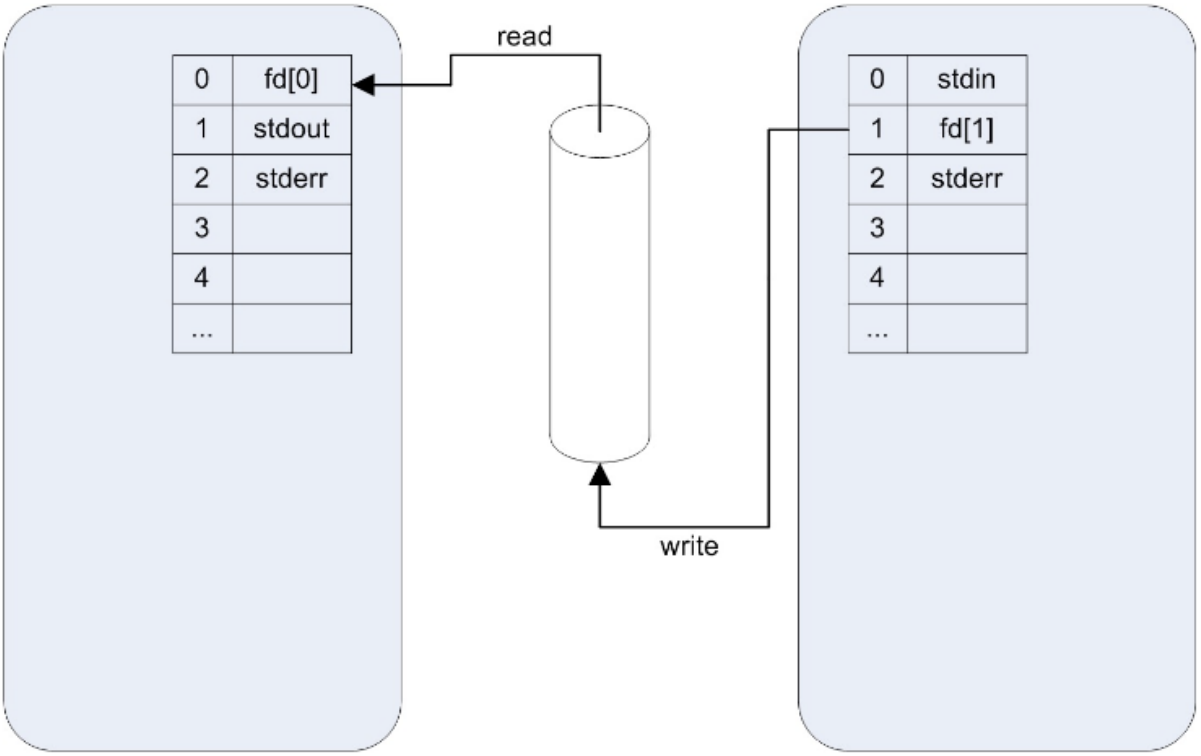
3. create a child process (fork())



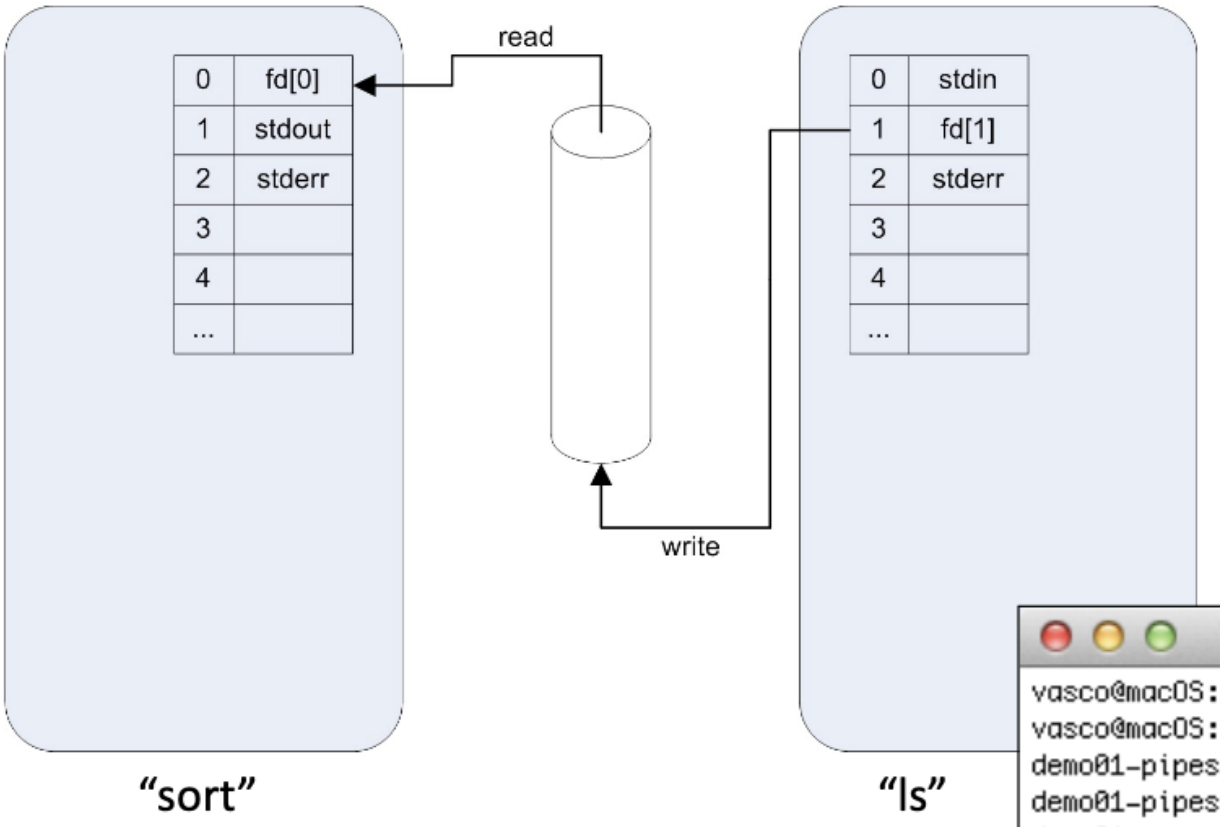
4. redirect file descriptors (dup2())



5. close file descriptors that are not needed



6. execute the code



named pipes

characteristics

similar to pipes but allow communication between unrelated processes

- each pipe has a name
- the pipe is written persistently in the file system

- They are **half-duplex** – each process writes or reads but not both
- They are opened like files, but they **are not files**
- You cannot `fseek()` a named pipe
- `write()` always appends to the pipe
- `read()` always returns data from the beginning of the pipe.
- After data is read from the named pipe, it's no longer there. It's not a file, it's an object in the Unix kernel.

functions

shell

```
mkfifo pipe_name -m<permissions>
# create a named pipe
```

```
- pipe_name: file/pipe name
- -mXXX: permissions (in octal)
```

```
ls -l pipe_name
# list the pipe in the folder (output begins with a `p`)
```

```
file file_name
# determine the file type
```

```
rm pipe_name
# delete named pipe
```

> **Note**
>
> use commands like `tail` or `cat` to see pipe contentes
> use `echo "string" > pipe_name` to write to a pipe

C

```
int mkfifo(const char *filename, mode_t mode);
// create a named pipe
```

```
- mode: O_CREAT | O_EXCL | <permissions in octal>
- returns -1 on error
```

```
- sets `errno` to `EEXIST` if that pipe name already exists
```

```
int open(const char *filename, int flags);
```

```
- flags:
        - `O_RDONLY`: Open for reading only.
        - `O_WRONLY`: Open for writing only.
        - `O_RDWR`: Open for reading and writing.
        - `O_CREAT`: Create the file if it doesn't exist.
        - `O_TRUNC`: Truncate the file to zero length if it already
exists.
        - `O_APPEND`: Append to the end of the file.
        - `O_EXCL`: Ensure that this call creates the file. If the file
already exists and `O_CREAT` and `O_EXCL` are used together, `open()` will
fail.
        - `O_NONBLOCK`: Open the file in non-blocking mode.
```

> **Note**
>
> The `read()` and `write()` functions were already explained for unnamed pipes.

```
int unlink(const char *pathname);
// delete a named pipe
```

I/O multiplexing

- If you get a SIGPIPE signal, this means that you are trying to read/write from a closed pipe

- A named pipe is a connection between two processes. A process blocks until the other party opens the pipe...
- Being it an open for reading or writing (`O_RDONLY` , `OWRONLY`)
- It is possible to bypass this behaviour (open it non-blocking – `O_NONBLOCK`), but be very, very careful: if not properly programmed, it can lead to busy waiting. If a named pipe is open

non-blocking, EOF is indicated when `read()` returns 0.
- When designing a client/server multiple client application, this means that either the pipe is re-opened after each client disconnects, or the pipe is open read-write.
- If opened “read-write”, the server will not block until the other party connects (since the server itself is also another party)

select()

- in a scenario where we have several pipes and we need to check them all, we would be blocked sometime when one of them was empty
- I/O multiplexing allow to examine several fole descriptors at the same time
- `select()` and `pselect()`

```
int select(int n,                    // greatest file descriptor number + 1
                  fd_set *readfd,            // reading activity
                  fd_set *writefd,           // writing activity
                  fs_set *exceptfd,          // out-of-band activity
                  struct timeval *timeout)
```

```
- blocks until activity is detected or a timeout occurs
- fd_set variables are input/output. upon return, they indicate if there was activity in a certain file descriptor or not
```

fd_set

It's a bit set:

- we have a list of file descriptors
- basically, we are saying that, if `[n] = 1` the n-th file descriptor will be present in our set

- `FD_ZERO(fd_set *set)`
  - Cleans up the file descriptor set
- `FD_SET(int fd, fd_set *set)`
  - Sets a bit (*sets to 1*) in the file descriptor set
- `FD_CLEAR(int fd, fd_set *set)`
  - Clears a bit (*sets to 0*) in the file descriptor set
- `FD_ISSET(int fd, fd_set *set)`

- Tests if a file descriptor is set