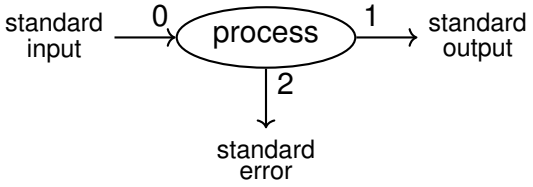
Chapter 7: File Input/Output

A Unix process can do pretty much whatever it wants with its virtual CPU and virtual memory, but all other hardware resources are only accessible by the kernel. In particular, reading or writing to/from persistent storage (i.e. HDD or flash) happens through a set of dedicated system calls. Similary, terminal input/output is performed by reading or writing "files".

1 Working with File Descriptors

Opening a file Before you can read or write a file, you must inform the kernel of your intention and ask for permission. This operation is called *opening the file*, and happens through a syscall named `open()` which takes two arguments: the desired **path** and some flags describing the desired access mode. When successful, `open()` returns a small, non-negative integer called a **file descriptor** (FD). This allows a process to have several open files simultaneously: every input/output syscall takes a file descriptor argument. Every process starts with three file descriptors already open: 0 is standard input, 1 is standard output, and 2 is standard error.



Exercise 1 Read K&R §8.1 for more information about file descriptors and K&R §8.3 for more about `open`. Type `man 2 open` to read the technical details, and click on the following links:

- `https://en.wikipedia.org/wiki/File_descriptor`
- `https://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html`

Reading and Writing Actual Input/Output between the process and the outside world happens through syscalls named `read()` and `write()`, which take just three arguments: a file descriptor, a pointer to a byte array, and a length (maximum number of bytes to be transferred). Read K&R §8.2 for more information about the read/write syscalls, then type `man 2 read`, `man 2 write` and click on the following link:

- `https://www.gnu.org/software/libc/manual/html_node/I_002fO-Primitives.html`

Exercise 2 The Unix command `cat` reads one or more text files and prints them[1] on standard output. Using syscalls `open()`, `close()`, `read()`, and `write()`, implement a program `cat-readwrite.c` which does the same, one character at a time.

2 Using buffered streams from <stdio.h>

The C language offers direct access to input/output system calls, but it also provides higher-level input/output facilities called **streams** (cf K&R Chap. 7). This improves portability of C source code to non-UNIX platforms (e.g. windows) with completely different system calls. But even on Unix systems, streams offer more flexibility than using low-level I/O and typically better performance thanks to buffering.

[1] The name is derived from its function to concatenate files.

Syntax A standard stream is a variable with type `FILE*` i.e. a pointer to some opaque type named `FILE`. Just like with the underlying file descriptors, the program has no access to the internal state of the stream, just an opaque handle. The `stdio.h` library provides three built-in `FILE*` named `stdin`, `stdout`, and `stderr`. The `printf()` function prints to `stdout`.

- `https://www.gnu.org/software/libc/manual/html_node/I_002f0-on-Streams.html`

Exercise 3 Write a program `cat-stio.c` with the same structure as the previous one, but using the stdio API. Use `fopen()/fclose()` to open and close files (cf first link below) When successful, `fopen()` returns a valid `FILE*` otherwise it returns `NULL` (an invalid pointer). Use `fgetc()` and `fputc()` to read and write one byte at a time (cf links below).

- `https://www.gnu.org/software/libc/manual/html_node/Opening-Streams.html`
- `https://www.gnu.org/software/libc/manual/html_node/Character-Input.html`
- `https://www.gnu.org/software/libc/manual/html_node/Simple-Output.html`

Remark: Notice that `fgetc()` returns an **`int`** and not a **`char`**. This is because it will return the special value `EOF` when the stream is finished.

3 Performance assessment

We're now going to compare the speed of our two programs.

Exercise 4 Prepare a big text file with command `ls -R / | head -c 1000000 > big.txt`

Remark: in the command-line above we use **file redirection** syntax (see chapter 3, section 2, as well as K&R §7.1):

Exercise 5 Read `man head` to understand what we did with `head -c 1000000`.

Exercise 6 Go read K&R §7.1 about file redirections. Ask us for clarifications if needed.

Exercise 7 Use the `time` command to measure execution times: `time ./cat-stdio big.txt` and `time ./cat-readwrite big.txt`. Even on modern systems, the cost of doing this many syscalls is so high that the `readwrite` version can be visibly slower. In contrast, the `stdio` version uses line-buffered output: `fputc()` accumulates the characters in an internal buffer, and only does one `write()` when it reaches a `'\n'` byte. Also, `stdio` detects that `bla.txt` is a file and not a terminal, so it uses a large buffer to reduce the number of calls to `read()`.
Do the same experiment but with a `>` to redirect stdout to a file. Compare the execution times.

Exercise 8 Execute the same four commands but with `strace` instead of `time`. This tool intercepts all system calls performed by a command and displays their arguments in a readable fashion.

- Observe that our call to `fopen()` results in a `open()` system call. What file descriptor is returned by the kernel ?
- Look at the `read()` and `write()` system calls. What is the size of `stdio`'s internal buffer for non-interactive `FILE*` streams ?

Remark: Remember from chapter 3 that you can also redirect the standard error of any command (e.g. strace) with the `2>` operator.

4 More fun with stdio

Exercise 9 In file `minigrep.c` write a program which reads every line of its standard input (with `fgets()` cf K&R §7.7, `man fgets` or link below) and prints only lines that match a certain pattern passed as a command-line argument, as illustrated below. You can use for instance `strstr()` for string searching, and `fputs()` for line output. Type `man strstr` and `man fputs` to read the docs.
Links:

- https://www.gnu.org/software/libc/manual/html_node/Line-Input.html#index-fgets
- https://www.gnu.org/software/libc/manual/html_node/Simple-Output.html#index-fputs
- https://www.gnu.org/software/libc/manual/html_node/Search-Functions.html#index-strstr

```
cat *.c | ./minigrep include
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

Exercise 10 Augment your minigrep with a new feature: when there are multiple arguments, don't read stdin but treat all subsequent arguments as file names to be searched e.g. `minigrep inclu *.c`.