# **Exercises** — Tiny Libstream

version **#7be580532266ed398481e31366afcc24b1950c2a**


**Assistants C/Unix 2023** <assistants@tickets.assistants.epita.fr>

# Copyright



# Contents



*https://intra.assistants.epita.fr

### File Tree

```
tinylibstream/
├─ Makefile   (to submit)
├─ include/
│  └─ libstream.h   (to submit)
├─ src/
│  └─ *   (to submit)
├─ stdin_buffering_test.c
└─ stdout_buffering_test.c
```

**Makefile**

- library: Produces the libstream.a library

**Authorized functions** : You are only allowed to use the following functions

- malloc(3)
- realloc(3)
- calloc(3)
- free(3)
- abort(3)
- isatty(3)
- open(2)
- write(2)
- read(2)
- close(2)
- lseek(2)

**Authorized headers** : You are only allowed to use the functions defined in the following headers

- err.h
- errno.h
- assert.h
- stddef.h
- string.h
- fcntl.h
- sys/stat.h

**Compilation** : Your code must compile with the following flags

- -std=c99 -pedantic -Werror -Wall -Wextra -Wvla

**Main function** : None

# 1 Introduction

The point of this exercise is to understand buffering, which is a major I/O performance optimization.

Without buffering:

- You would likely wait for seconds until each frame of your favorite internet videos appears on-screen.
- Writing and reading to and from files would be, in some cases, orders of magnitude slower.

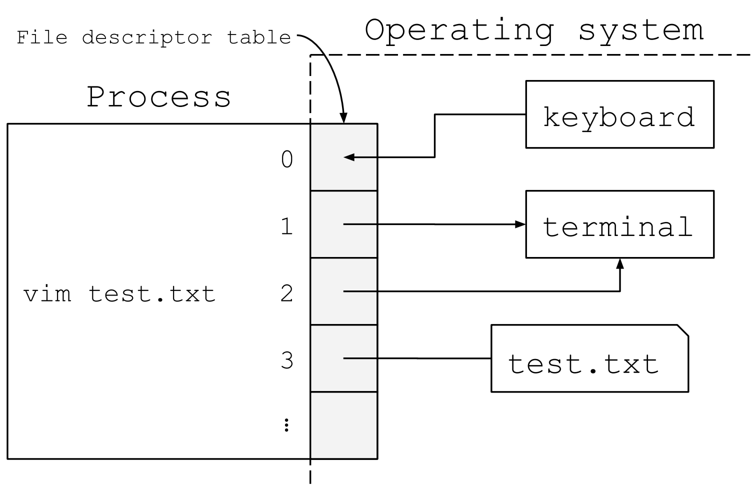
## 1.1 File descriptors

Processes need to deal with files, but can't just do it by themselves. Writing to files requires access to an actual storage medium, such as a hard disk: you wouldn't like any process to be able to wipe your disk, would you? Because of this reason and many others, the operating system manages it for you.

When a process opens a file, the operating systems returns a special identifier, called a file descriptor.

On the operating system side, each process has an array of open files, called the file descriptor table. A file descriptor[1] is the index of some file in this table.

When you `open(2)` a new file, the first unused cell of the array is used to store a pointer to the open file.




When you `open(2)` a new file, you get a file descriptor. You can then `read(2)` and `write(2)` to the file using your file descriptor. Once you're done working with the file, the file descriptor can be closed using `close(2)`.

[1] Programmers often shorten "File descriptor" as "fd".

## 1.2 Buffering

Unfortunately, calls to `read(2)` and `write(2)` are very, very slow, in a way that can't easily be fixed[2].

One could carefully program applications to minimize the number of calls to `read(2)` and `write(2)`. But there's an easier and safer solution: buffering.

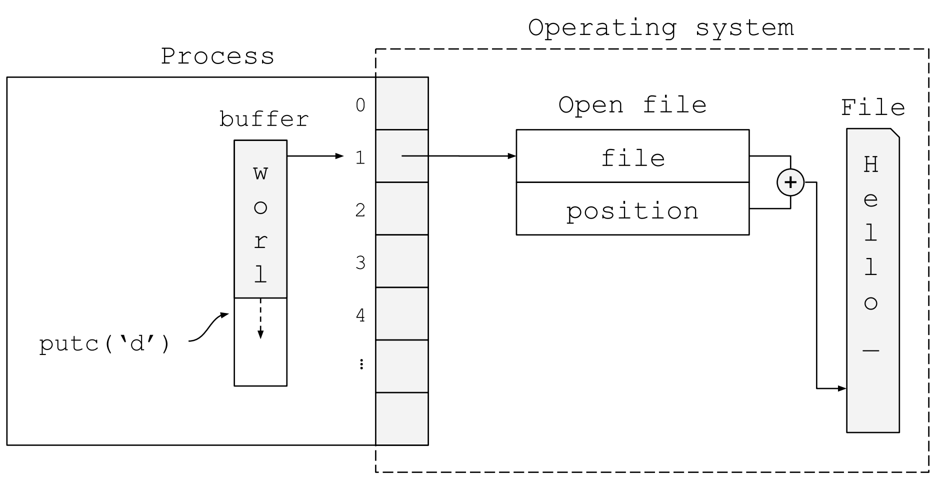
### Write buffering

Writing data is just like bringing clothes to the laundry: instead of doing a round trip to the laundry per article of clothing, you can carry batches of clothes, thus increasing your throughput.

Whenever some clothes need to be washed, put these in a basket of clothes that needs to be brought to the laundry. When the basket is full, you can bring it to the laundry. This technique reduces the time spent making round trips.

The same statement holds true for data and buffers:

Whenever some data needs to be written, put these in a buffer that needs to be written to the file. When the buffer is full, write its content to the file. This technique reduces the time spent making syscalls, which are round trips between the process and the operating system.




### Read buffering

Read buffering works slightly differently: it's about reading more than what the user asks, so that the next time your program needs data, it doesn't have to read once more.

It works just like Japanese restaurants: as customers don't know how hungry they are, they will fill a plate full of food just in case, eat from it, refill the plate when empty and repeat the process until they're not hungry anymore.

The same statement holds true for data and buffers:

As processes don't always know how many bytes they will need to read, they read a bunch of them into a buffer just in case, read from the buffer, refill the buffer when empty and repeat the process until done.

[2] `read(2)` and `write(2)` are system calls. System calls are slow because they must discard multiple processor caches, for architectural and security reasons.

Unlike Japanese food, unused data is disposable, and you won't pay extra for not using all of it.

## 1.3 Buffering modes

### Output buffering

Compile and run the provided `stdout_buffering_test.c`.

**Tips**

`make stdout_buffering_test` should do the trick for the compiling part.

As you can see and might already expect, the program outputs characters line by line. That's because when outputting data to a terminal, the program flushes its buffer as soon as a `\n` character is written. This mode of operation is called line buffering.

Now, run the following command:

```
42sh$ ./stdout_buffering_test | cat
```

What's going on? Well, `stdout_buffering_test` knows its output isn't going to an interactive terminal (it's sending data to `cat`). So it doesn't go through the hassle of buffering output line by line for you to read it.

Now, run the following command:

```
42sh$ stdbuf -oL ./stdout_buffering_test | cat
```

This command does the same thing, but `stdbuf` configures the buffering of stdout (`-o`) to flush on lines (`L`)[3].

```
42sh$ stdbuf -o0 ./stdout_buffering_test
```

This command runs `./stdout_buffering_test` with buffering disabled. Thus, characters aren't retained in the buffer and are immediately printed out.

```
42sh$ stdbuf -o3 ./stdout_buffering_test
```

This command runs the above program with a buffer of size 3. It's not line buffered, so it'll only flush when the buffer is full or the program terminates.

### Input buffering

Compile the provided `stdin_buffering_test.c`.

This program waits half a second each time it calls `read(2)`, and prints everything it gets from `stdin`.

```
42sh$ echo coucou | ./stdin_buffering_test
```

As the default buffer size is bigger than the input message, `stdin_buffering_test` prints all of it at once.

Try the following code samples:

[3] `stdbuf` runs programs with modified stdio buffering settings. It is sort of equivalent to calling `setvbuf(3)` from inside the program.

```
42sh$ echo coucou | stdbuf -i3 ./stdin_buffering_test
```

```
42sh$ echo coucou | stdbuf -iL ./stdin_buffering_test
```

Can you guess why `stdin` line buffering is meaningless?

Here is the line buffering workflow: as soon as a line ends, the process on the writing end of the pipe (`echo` in our case) writes the line. The process on the read end of the pipe doesn't know how much data was written, so it just reads as much as it could.

Input line buffering is meaningless because the reading process will get its data as soon as written anyway.

### 1.4 The file position indicator

When you open a file, there's a variable you don't have direct access to, called the file position indicator.

It stores where the next read or write would happen. When a file is opened, this indicator variable is positioned at the beginning of the file. If you `read(2)` or `write(2)` $n$ bytes of data, the indicator moves forward by the same amount of bytes.

**Be careful!**

`read(2)` and `write(2)` may not read or write as much as requested! These functions return the number of bytes actually read or written.

If you want to read or write exactly $n$ bytes, you may have to call the function multiple times.

The `lseek(2)` function can both change and return the file position indicator. `ftell(3)` and `fseek(3)` are the buffered equivalents, but you won't have to implement them.

**Tips**

You may need to set correctly the file position indicator when opening a file or when flushing. In those cases, use `lseek(2)`.

**Be careful!**

The file position indicator is also called the file offset.

## 2 Exercise requirements

You should implement every functions from the given `include/libstream.h` header file.

**Be careful!**

Your exercise should handle read and write buffering using a single buffer!

When your program needs to switch between read buffering and write buffering, it should flush the content of the buffer: if the buffer contains write-buffered data, it has to be written to the file descriptor. If it contains read-buffered data, it must be discarded, and the file position indicator must seek back to where the user would expect it to be if there were no buffering.

The given order might help you implementing them:

### 2.1 *lbs_fopen*

The `lbs_fopen` function should create a stream and associate a file to it. *open(2)* should be used and the stream's field *fd* need to be set to the returned file descriptor.

You'll only have to handle four `fopen(3)` modes and convert them into `open(2)` flags, thus the stream's field `flags` should be set accordingly. Those are the different modes you'll need to handle:

- *r*
- *r+*
- *w*
- *w+*

If *lbs_fopen* fail, `NULL` must be returned.

**Tips**

It is advised to take a closer look on `open(2)` and `fopen(3)` manpages for respective flags and mode equivalences.

**Tips**

By default, the **Buffering mode** should be set to *buffered*, you might want to change it when you're dealing with a terminal.

### 2.2 *lbs_fdopen*

The *lbs_fdopen* function works like `lbs_fopen` except it takes the *file descriptor* instead of the *path*.

Like `lbs_fopen`, if `lbs_fdopen` fails or the *file descriptor* is invalid, `NULL` must be returned.

**Tips**

You might want to use `lbs_fdopen` in `lbs_fopen` to make implementation clearer, since in `lbs_fopen` you use *open(2)* to have a *file descriptor* associated to a path.

**Tips**

To open standard I/O streams, `stdin`, `stdout` and `stderr`, you can use respectively $0$, $1$ and $2$. However if you don't want to use magic values, you can use preprocessor symbols defined in `unistd.h`, respectively `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`. For more infos, check `stdin(3)`.

### 2.3 *lbs_fflush*

The `lbs_fflush` function should flush the stream's buffer to the underlying file descriptor, making sure the stream position is correct. Since there is two operations, *writing* and *reading*, you'll need to handle these two operations differently:

- On writing operation, the data has to be **written**.
- On reading operation, the data needs to be **discarded** and the file descriptor must be put back to the position the user expects.

Globally, this function works just like `fflush(3)` except it **does not** flush all open output streams if the current stream is `NULL`.

On fail, it should set the error indicator.

**Tips**

To ensure the stream position is correct, you'll need to use `lseek(3p)` and the correct offset. You might want to take a closer look at the given functions.

### 2.4 *lbs_fclose*

The *lbs_fclose* function works just like `fclose(3)`. It flushes the stream, closes the associated *file descriptor* and frees the steam. The stream being freed, it can not be used no more.

On success, it should return 0, otherwise any other values.

### 2.5 *lbs_fputc*

The `lbs_fputc` function works just like `fputc(3)`. It writes a single character to the stream buffer. It may cause the stream to flush for different reasons:

- The buffer is full.
- The *buffering mode* is set to **unbuffered**.
- The *buffering mode* is set to **line buffered** and it encounters a '`\n`'.

On fail, it may set the error indicator and return $-1$, otherwise it return the written character.

**Tips**

The stream's current operation need to be switched to **writing** if needed.

**Be careful!**

It can flush multiple times on the same call.

### 2.6 *lbs_fgetc*

The `lbs_fgetc` works just like `fgetc(3)`, it reads a new character from the stream's buffer. It should refill the stream's buffer whenever it is empty.

It should return the read character, otherwise it may set the error indicator and return $-1$.

**Tips**

The stream's current operation need to be switched to **reading** if needed.

**Be careful!**

A refill should be done only once the buffer is empty. Be careful on the fact that `read(2)` may not read in one time the desired *count*.

# 3 Usefull resources

## 3.1 Glossary

**buffering**

The process of batching multiple operations together.

**buffer**

Among other things, a place where you store data awaiting to be processed, as part of buffering.

**buffer flush**

The process of synchronizing buffers with their underlying resource. For write buffers, it means writing the content of the buffer and emptying it. For read buffers, it means rewinding n characters back, with n being the number of characters already read by the user.

## 3.2 References

The C standard IO library is specified by http://pubs.opengroup.org/onlinepubs/9699919799/ and more helpfully the *Standard I/O Streams* section in the *General Information* chapter in the *System Interfaces* volume.

Also, the glibc manual https://www.gnu.org/software/libc/manual/ and the *Input/Output on Streams* chapter could be also an interesting read.

**Be careful!**

The POSIX specification document is a tough read. It is not meant to bring any sort of intuitive explanation of buffering.

*The way is lit. The path is clear. We require only the strength to follow it.*