CS2106: Operating Systems
Lab 5 – stdio Library

Important:

- The deadline of submission through LumiNUS is **Sat, 16 Nov, 2pm**
- The total weightage is **5%:**
  - Exercise 1: 1%
  - Exercise 2: 2% **[Lab Demo Exercise]**
  - Exercise 3: 1%
  - Exercise 4: 1%
- The assignment write-up is adapted to be run on Ubuntu 16.04.
- Lab 5 will be tested on the lab machines (Ubuntu 16.04).

Section 1. Overview

The purpose of this lab is to deepen your understanding of the standard file operations and to introduce you to the concept of dynamic libraries. There are **four exercises** in this lab and solving them will produce your own mini-stdio library. This library creates a wrapper over the system calls `open`, `close`, `read`, `write`, and `lseek`. The purpose of having this library is to reduce the number of system calls made to the OS when working with files.

General outline of the exercises:

- Exercise 1: Defining and initializing the `MY_FILE` data structure
- Exercise 2: `my_fread()` + Demo
- Exercise 3: `my_fwrite()`
- Exercise 4: `my_fflush()` + `my_fseek()`

It is strongly recommended to read the whole document before starting to work on this assignment. Before you start, you also need to appropriately setup your environment (as described in this document) to avoid difficulties when compiling your code later.

1.1 Grading

Please submit your source files (**ex1.c**, **ex2.c**, **ex3.c**, and **ex4.c**) together with the header file **my_stdio.h**, placed inside a .zip folder as described at the end of this document. For each exercise, we will compile your code and run it against a different **runner.c**. We will use a different **runner** to ensure correct execution of each exercise. You will obtain the marks for an exercise if it passes our test cases (partial credit may be given if your code has bugs that only manifest on corner cases). One of the two marks allocated to exercise 2 will be awarded for the demonstration to your lab TA. Only changes made in **ex1-4.c** and **my_stdio.h** will be used for grading. You may change the

other files during your own testing as well, but those will not be taken into consideration during grading.

Note that you are not allowed to use the library calls `fopen, fclose, fread, fwrite, fseek, fflush` for this lab. You need to make use of the system calls (`open, close, read, write, lseek`) to implement your file operations.

1.2 Compilation and Running

We provide a `Makefile` to make the process of compilation easier. Running `make` in the main folder of the assignment compiles the source codes and produces the library **libmy_stdio.so**, as well as the `runner` and `demo` executables. You can then execute `./runner` to test your code against a set of tests and check whether your code is working as expected.

Steps in the `Makefile`:

1. Compiles the **ex1-4.c** source files into **libmy_stdio.so**. Note that only files **ex1-4.c** and **my_stdio.h** will be considered for grading and your code should only be added into those files.
2. Compiles **runner.c** and **demo.c** linking the **my_stdio** library. In general, if you want to link the library when compiling your code, you must specify this during the compilation of your code by adding: `-L<path_to_libmy_stdio.so> -lmy_stdio`
3. You can run `make clean` to remove the files that were produced during compilation.

Before you can successfully execute `./runner`, you must set the environment variable LD_LIBRARY_PATH to prompt the loader to search for libraries in the current directory as well when starting programs. Otherwise, if the loader cannot locate **libmy_stdio.so**, it won't be able to execute the programs that use it. We propose two ways to set LD_LIBRARY_PATH:

1. Updating your .bashrc file. From **lab5** folder (containing the library and the executable), run the following command (one time only):
   ```
   $ echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.' >> ~/.bashrc; source ~/.bashrc
   ```
   You can verify that .bashrc has been properly updated by running:
   ```
   $ tail –n 1 ~/.bashrc
   ```

   After this, you can compile and execute the **runner** after each change in your code:
   ```
   $ make
   $ ./runner < input1.in
   ```
   Note that **runner** and **libmy_stdio.so** must be in the same directory!

2. Calling **call_runner.sh** script. This script sets the LD_LIBRARY_PATH and then calls the runner. You may have to set 'execute' permission on the **call_runner.sh** script by running (one time only):
   ```
   $ chmod +x call_runner.sh
   ```

After this, you can compile and run the **runner** after each change in your code:

```
$ make
$ ./call_runner.sh input1.in
```

Note that **call_runner.sh** creates a new process that sets the `LD_LIBRARY_PATH` and then calls your **runner**. However, the variable is set only for that process and thus the modification is not persistent.

If you don't set up your LD_LIBRARY_PATH accordingly, you will encounter this error when trying to execute the runner:

```
./runner: error while loading shared libraries: libmy_stdio.so:
cannot open shared object file: No such file or directory
```

While the `Makefile` uses of the flag `-Wall`, marks will not be deducted if your code produces warnings when it is compiled.

1.3 Dynamic Libraries

Here we describe how to create and use dynamic libraries. Note that you can solve the assignment without this knowledge, and you may skip this subsection if it's not your cup of tea.

A dynamic or shared library is created with the purpose of being linked at run-time by other programs. The library can be linked by many programs at the same time, despite having only one instance of it loaded in memory - this can greatly reduce the memory consumption.

On Unix-like systems, dynamic libraries have the extension .so, from (dynamic) shared object, whereas their counterparts on Windows have the extension .dll, from dynamic-link library. Our focus will be on dynamic libraries for Unix-like systems.

Creating a dynamic library

To create a dynamic library, the `-fPIC` flag must be used during compilation. PIC stands for Position Independent Code and ensures that the generated machine code does not require to be located at a specific virtual memory address in order to work properly. This allows multiple processes to share the library code because they can map it anywhere in their own virtual address space without affecting the proper functionality of the library.

Let's say we want to create a dynamic library from our source file, `foo.c`. As you may expect, the first step is to compile it:

```
$ gcc -Wall -fPIC -o foo foo.c
```

Next, we have to turn the resulting object file `foo.o` into a shared library, which we shall call `libfoo.so`. To do so, we run:

```
$ gcc -shared -o libfoo.so foo.o
```

The `-shared` flag allows us to create a shared object that can later be linked by other files to form an executable.

Using a dynamic library
To allow `bar.c` to use functionalities defined in `libfoo.so`, we have to link `libfoo.so` during the compilation of `bar.c` using the `-l` option. In addition, we also have to specify where the library is located in the system using the `-L` option. Thus, the compilation command will look similar to this:
```
gcc -L/path/to/foo -Wall -o bar bar.c -lfoo
```

GCC assumes libraries to be starting with **lib** and end with **.so** or **.a**, thus `-lfoo` will look for `libfoo.so`.

The last step is to inform the loader (i.e., the part of the OS that's responsible for loading programs and libraries) that it should be looking in `/path/to/foo` as well when searching for libraries during the program loading. This can be done by adding to `/path/to/foo` to the LD_LIBRARY_PATH environment variable. Earlier, we instructed you to add ., i.e., the current directory, to the LD_LIBRARY_PATH, so whenever you try running the **runner** or **demo**, the directory from which you are running the command will also be searched for libraries.

Miscellaneous
`ldd`: The `ldd` command prints the dynamic libraries required by a program. You can test it on the **runner** executable before and after setting up LD_LIBRARY_PATH!
```
$ ldd runner
```

You may see where different libraries are mapped in a process' address space using $ `cat /proc/<PID>/maps`.

Section 2. Implementing the Assignment

The goal of this assignment is to produce a toy version of the stdio library, so all the function names and data structures have the same names defined by the stdio library prefixed by the string `my_`, capitalized when needed, e.g., instead of having `FILE *f = fopen()`, we will now have `MY_FILE *f = my_fopen()`. Your task is to implement **buffered file operations** that wrap around the primitive file operations. The buffered version essentially maintains an internal intermediate storage in memory (i.e. buffer) to store user read/write values from/to the file.

The data structure `MY_FILE` is defined in `my_stdio.h`, together with the prototypes of all the functions you have to implement.

2.1 Exercise 1: Defining and initializing your MY_FILE structure

You can find TODO comments in my_stdio.h and ex1.c to point out what you need to do for this exercise. You can add code in any parts of the code in my_stdio.h and ex1.c.

A `MY_FILE` structure has been defined in **my_stdio.h**; at the moment, the structure has only one member, `fd`. Your first job is to enhance this structure with whatever fields you deem necessary to solve the assignment. We strongly recommend reading all the requirements of the assignment before starting the implementation.

In **ex1.c**, the function `my_fopen` and `my_fclose` have been mostly implemented for you but there is still some work to be done. You are implementing buffered file operations. **The buffer size (capacity) is 4096 bytes!**

`MY_FILE *my_fopen(const char *pathname, const char *mode)`: Opens the file specified by `pathname` and associates a stream to it. The argument `mode` specifies whether a file is opened for reading ("r"), writing ("w") or appending ("a"). The difference between "w" and "a" is that files opened in append mode always perform their writings at the end of the file. An additional "+" character may be added to specify that a file is opened in what is called update mode ("r+", "w+" or "a+"), and both write and read operations can be performed on it (for "a+" files, the writing operations will be performed at the end of the file). The return value is a MY_FILE pointer if the call is successful, or NULL otherwise.

We took the necessary steps to create a file descriptor `fd` and associate it with the `MY_FILE` structure. Your job is to initialize the remaining fields of your structure based on how you define it.

`int my_fclose(FILE *stream)`: flushes the `stream` pointed to by `stream` and closes the underlying file descriptor. If successful, the function returns 0, otherwise it returns `MY_EOF`. The only thing you need to do related to this function is free any memory that your `MY_FILE` structure used.

You can modify `my_stdio.h` and `ex1.c` as you find necessary as long as you ensure that the `Makefile` still compiles your code successfully. We will use the same `Makefile` to create your **libmy_stdio.so** during grading.

2.2 Exercise 2: Implementing `my_fread()`

Part 1: my_fread() functionality (1 mark)

Your task for this exercise is to implement the functionality of the `fread`() function; we will call this newly defined function `my_fread()`.

`size_t my_fread(void *ptr, size_t size, size_t nmemb, MY_FILE *stream)`: The function reads `nmemb` items of data, each `size` bytes long, from the stream pointed to by `stream`, storing them at the location given by `ptr`. The function returns the number of items read, or `–1` if an error occurs.

The purpose of `my_fread()` is to use the buffer to reduce the number of `read()` system calls; a correct implementation of `my_fread()` should call `read()` only when needed.

Part 2: Demo (1 mark)
Do you remember the behaviour observed in Lab 2 ex1? In this lab, we will replace the `fread()` calls with `my_fread()` calls (done for you in **demo.c**). Your task is to explain what the `MY_FILE` data structure contains for the parent and child processes, and to specify what will be read for both CASE A and CASE B:

- CASE A: there is an `my_fread` call done before fork
- CASE B: no `my_fread` done before fork

demo.c is compiled and linked by the Makefile provided to you. To run the demo, call:

```
$ ./demo input1.in 4
```

Or

```
$ ./call_demo.sh input1.in 4
```

2.3 Exercise 3: Implementing `my_fwrite()`

Your next task is to implement `my_fwrite()` function such that it mimics the functionality of the standard `fwrite()` function.

`size_t my_fwrite(const void *ptr, size_t size, size_t nmemb, MY_FILE *stream)`: The function writes `nmemb` items of data from the location given by `ptr`, each item having `size` bytes, to the stream pointed to by `stream`.
Note that if a file is opened in append mode, all writes should be performed at the end of the file regardless of repositioning operations.
The function returns the number of items written, or `–1` if an error occurs.

The purpose of `my_fwrite()` is to use the buffer to reduce the number of `write()` system calls; a correct implementation of `my_write()` should call `write()` only when needed.

2.4 Exercise 4: Implementing `my_fflush()` and `my_fseek()`

Your task in this exercise is to implement `my_fflush()` and `my_fseek()` functions that mimic the `fflush()` and `fseek()` functionalities, respectively.

Part 1: my_fflush
`int my_fflush(MY_FILE *stream)`: The function forces the buffered data for the given output stream pointed by `stream` to be written via the stream's underlying write function (i.e., ~~`my_f`~~`write()`). For input streams, `my_fflush()` discards any buffered data that has been fetched from the file. The function should return 0 if the operation is successful, or `MY_EOF` otherwise.
You can see an instance of my_fflush() being used when my_fclose() is called.

Part 2: my_fseek

`int my_fseek(MY_FILE *stream, long offset, int whence)`: the function sets the file position indicator for the stream pointed to by `stream`. The new position, measured in bytes, is obtained by adding `offset` bytes to the position specified by `whence`. The `whence` can take 3 values:

- `SEEK_SET`: offset is relative to the start of the file
- `SEEK_CUR`: offset is relative to the current position indicator
- `SEEK_END`: offset is relative to the end-of-file

The standard `fseek()` function returns 0 upon success and –1 if an error occurs. However, the value returned by `my_fseek()` should indicate the new value of the file offset as measured from the beginning of the file, or –1 if an error occurs. You should implement `my_fseek()` using `lseek()` system calls.

The SEEK_SET, SEEK_CUR and SEEK_END values are defined in **unistd.h** and take the values 0, 1, and 2 respectively. Note that the **runner** expects these values to be given as numbers.

2.5 Testing Your Implementation

Nota bene: When a file allows both reading and writing operations, the standard specifies that:

- `fread()` cannot follow `fwrite()` without a prior call to `fflush()` or positioning function (e.g., `fseek()`)
- `fwrite()` cannot follow `fread()` without first calling a positioning function.

We will follow the same convention in testing your implementations for `my_fread` and `my_fwrite`.

You can test your implementation by using **runner.c**. The **runner** reads commands from an input file; the commands specify what file operation to be performed in the following template:

- `my_fopen <file-name> <mode>`
- `my_fclose <file-name>`
- `my_fread <file-name> <number-of-items>`
- `my_fwrite <file-name> <number-of-items>`
- `my_fflush <file-name>`
- `my_fseek <file-name> <position> <whence>`

Note that when writing, the runner will write the letters from `a` to `z` repeatedly until it writes as many bytes required by `my_fwrite()` function call (each letter is one byte).

Executing each command prints a message to STDOUT, prefixed by the letter S or F, specifying whether the command was executed successfully, or the command failed (i.e., an error has occurred). Note that an S does not necessarily mean the output is correct, it only specifies that no errors were encountered while executing the command. Similarly, an F does not mean the output is wrong, it simply indicates that

executing the command resulted in an error. There are a few errors that will immediately terminate your program, e.g., incomplete commands or if the **runner** is unable to allocate memory. Failure to allocate a `MY_FILE` is not considered a fatal error since it can also happen if the file is opened for reading but the file does not exist.

You can redirect the output of the runner to another file using >, e.g., you can call runner like this:

```
$ ./runner < input1.in > input1.out
```
or
```
$ ./call_runner input1.in > input1.out
```

The correct output for input1.in can be found in input1.out. You can check that your output matches the correct one using `diff`, e.g., `diff –b input1.out <your-output>`.

The **runner** used during the actual grading will also check that your implementation makes the appropriate system calls and marks will be deducted if the number of system calls is not appropriate. You can investigate what system calls are performed by your code using `strace`.

Input:

```
my_fopen runner.c r
my_fread runner.c 101
my_fopen rummer.c r
my_fopen test.txt w+
my_fwrite test.txt 5
my_fseek test.txt 3 0
my_fread test.txt 1
my_fseek test.txt 0 2
my_fwrite test.txt 4096
my_fflush test.txt
my_fclose test.txt
my_fopen test.txt a
my_fwrite test.txt 10
my_fflush test.txt
my_fclose test.txt
my_fclose runner.c
my_fopen test.txt r
my_fread test.txt 3
my_fclose test.text
my_fclose test.txt
```

Expected output follows:

```
S: File runner.c is now open
S: 101 bytes were read from file runner.c
F: Could not open file rummer.c
S: File test.txt is now open
S: 5 bytes were written to file test.txt
S: File offset of file test.txt is now at position 3
S: 1 bytes were read from file test.txt
S: File offset of file test.txt is now at position 5
S: 4096 bytes were written to file test.txt
S: File test.txt was flushed
S: File test.txt is now closed
S: File test.txt is now open
S: 10 bytes were written to file test.txt
S: File test.txt was flushed
S: File test.txt is now closed
S: File runner.c is now closed
S: File test.txt is now open
S: 3 bytes were read from file test.txt
F: File test.text is not open
S: File test.txt is now closed
```

Section 3. Submission

Zip the following files as E0123456.zip (**use your NUSNET id, NOT your student no A012…B, and use capital 'E' as prefix**):

a. **`my_stdio.h`**
b. **`ex1.c`**
c. **`ex2.c`**
d. **`ex3.c`**
e. **`ex4.c`**

Do **not** add additional folder structure during zipping, e.g. do not place the above in a "lab5\" subfolder etc.

Upload the zip file to the "Lab Assignment 5" folder on LumiNUS. Note the deadline for the submission is **Sat, 16 November, 2pm**.

Please ensure you follow the instructions carefully (output format, how to zip the files etc). **Deviations will be penalized.**