Assignment 02

If any of this is unclear, please get in touch with me ASAP. There are definitely things in the course I want you to struggle with. I absolutely do NOT want you struggling with understanding my expectations.

As discussed in class, using system calls directly can be horrifically inefficient. Specifically with regards to the I/O-related system calls, consider an imaginary program that produces 1 byte of output at a time, and does this continually throughout its lifetime. Calling `write(2)` every time it produces a single byte is going to be wasteful; it makes far more sense to save up those individual bytes of output and to only call `write(2)` once it has accumulated "enough" (to be defined later).

Saving intermediate results like this is called *buffering* and, not at all coincidentally, an arbitrary chunk of memory used for storing stuff is often called a *buffer*. This is directly related to the dreaded "Buffering..." message one sees from Netflix: your computer has a chunk of memory (a buffer) in which it stores the next few seconds of video. Your computer is simultaneously reading *from* that buffer to display the video and also receiving new data from Netflix *into* the buffer. Your video stops playing when the buffer is empty; the message indicates that Netflix has to send more data to catch up with the rate at which the buffer is emptied.

For this assignment, you will produce a collection of functions that insulate a program from the inefficiencies of I/O-related system calls using the method discussed in class. You will create functions analogous to the system calls `open(2)`, `close(2)`, `read(2)`, `write(2)`, and `lseek(2)`. You will also implement a function to *flush* a buffer: that is, to force all buffered data to be delivered to its destination (note that this only makes sense in the context of writing).

As implied above, the question of what constitutes "enough" is important. Traditionally, there are three options: unbuffered, in which data is immediately delievered to/from a process to the corresponding system call; line-buffered, in which the abstraction layer buffers a single line (ie, a string that ends in a newline character); and block buffered, in which the buffer is a fixed size (a "block"). For this assignment, you will implement a block-buffering scheme, whose block size is configurable at compile time.

In addition to the system calls enumerated above, you will likely find these library functions useful in your implementation:

- `memcpy(3)`
- `malloc(3)`
- `free(3)`

Goals

Abstractly, the goals of this assignment are for you to:

- internalize the potential inefficiency of system calls;
- refresh your facility with pointers and the heap;
- expand your use of Makefiles;
- continue to expand your experience with C and design/implementation of non-trivial programming projects.

Behavioral requirements

You will produce one source file, `myio.c`, that contains implementations of the following functions with the described behavior:

- `myopen`, analogous to `open(2)`. You must support these flags: `O_CREAT`, `O_WRONLY`, `O_RDONLY`, `O_RDWR`, `O_TRUNC`. In the case of `O_CREAT` and `O_TRUNC`, you may assume a mode of 0666 in all cases.
- `myclose`, analogous to `close(2)`.
- `myread`, analogous to `read(2)`.
- `mywrite`, analogous to `write(2)`.
- `myseek`, analogous to `lseek(2)`. We have not discussed this one, so read the manpage to understand its behavior. You need only implement the `SEEK_SET` and `SEEK_CUR` options.
- `myflush`, which forces any buffered data to its destination (eg, if the buffer has 12 bytes in it and is waiting for another 37 before it actually calls `write(2)`, a call to `myflush` will immediately cause those 12 bytes to be written, regardless of buffer occupancy).

Note that your `my*` functions *will* themselves issue system calls. For example, the only way to open a file for writing is to issue the `open(2)` system call, so presumably your `myopen` will (among other things) call `open(2)`. For further example, the program may call `mywrite` a bunch of times but the bytes being "written" will be saved up and only when enough have accumulated will `mywrite` actually call `write(2)` to push them to disk. This aspect is particularly important, so *please* talk to me if the desired behavior is not clear.

The various manpages imply but do not explicitly state one important fact: the same file offset is used for both reading and

writing. (Thus this is only applicable when a file is opened with the `O_RDWR` flag.) So if I open the file and then read 12 bytes (ie, bytes 0 through 11), the next operation (either reading *or* writing) will start at byte 12. In the case of a write operation, the bytes starting with the 12th will be overwritten.

Your implementation must allow a program to have an arbitrary number of files open simultaneously.

The size of the buffer must be easily configurable at compile time.

Testing

You will also produce any number of source files that *test* the behavior of the aforementioned functions. You might have one program to test simple reading, another to test simple writing, and others to test more complicated operations. The requirements for these programs are that the precise behavior they are testing must be documented.

You will find that these programs do not compile cleanly without function prototypes for all the `my*` functions. To solve this, create a *header file* called `myio.h` that contains those prototypes. Then, in the programs, you need only `#include <myio.h>`. (We will discuss header files in more detail when we get deeper into libraries, but for now this will do.)

Deliverable requirements

At least five files:

- `myio.c` and `myio.h`, whose contents and behavior are described above.
- At least one C program that tests the behavior of `myio.c`, the name(s) of which is/are at your discretion.
- `Makefile`, whose default target should build *all* the test programs. It must also have a `clean` target to remove all generated files. You may have additional targets if you like.
- (Optional) A shell script that executes all your tests.
- `README`, which will include a list of authors, a list of known bugs, and a list of resources you consulted in your work (a list of URLs is totally acceptable—if I can see the kinds of references you're using, I can get a better idea of how to tailor the instruction to your needs—just keep the README file open as you work and paste them there when you find something useful). The `README` file should also describe how to test your library.

Style requirements

Your code must exhibit:

- appropriate use of comments;
- appropriate use of stack, heap, and global variables;
- appropriate use of constants;
- appropriate decomposition;
- clear organization (eg, grouped header files, struct definitions, function prototypes);
- well-named variables, functions, and other identifiers;
- verification of command-line arguments, if applicable;
- checking and appropriate handling of all return values;
- consistent style (capitalization, indenting, etc).

Submission Process

Your submission files should be in a git repository on basin in this directory:

```
/home/<username>/<secret>/assignment2
```

When you wish for feedback, one member of your partnership should email me (CC'ing the other) with a description of the assignments and features upon which you would like my attention.

Last modified: 09/25/2022 12:01:10