Problem 1 -- Kernel vs user mode

We can make system calls from C (or C++) programs. They look just like function calls. But as we'll see later in this course, there is some magic stuff (special assembly language) inside these system call functions which triggers the actual system call and transition to kernel mode.

Below is a table of functions you can call in the standard C library. Also included are some things which look like functions, but are something else. Some of these are actual system calls. Some are entirely inside user mode. Some always trigger another, underlying system call. Finally, some of these things could remain in user mode, or require a system call, depending on circumstances.

Evaluate each item, and complete the table. Is it a true system call? If yes, mark a Y and move on. But if no, does it trigger an underlying system call (always, sometimes, never)? And finally, if it always or sometimes triggers a system call, explain what system call and under what circumstances. Use a separate paragraph attached below the table if needed for the explanation.

Sources of information to complete this assignment: Unit 1 lecture notes. The "man pages:" look at the `-f` option to `man`. Run `"man man"` and/or refer to the lecture notes to learn what the various man page numbered sections mean.

name	syscall (Y/N)?	If No, triggers syscall (always/sometimes/never)	triggers what?
`getchar`			
`write`			
`strcpy`			
`strdup`			
`fdopen`			
`strerror`			
`isprint`			
`atoi`			
`break`			
`pow`			

Problem 2 -- Uh oh, something went wrong

Each of the situations below results in an error which sets `errno`. For each one, explain:

i) Why did this cause an error?

ii) What will `errno` be set to? State this answer in terms of the symbolic macro names for the integer error codes. e.g. `ETIMEDOUT`

iii) If `perror("")` is called immediately after the error, list the human-readable error message that will be printed to `stderr` e.g. "`Connection timed out`"

Hint: the above examples are to illustrate the format of the responses and are not the correct answers for any of the situations below! While the man pages for each system call document the error codes which are returned for different error conditions, they do not directly answer part (iii). You can write a simple test program to see what each

E code produces. You can also write test programs to cause these error conditions. But, you'll probably just ask the internet.

2A) `close(-1);`

2B) `int n=write(0,"41CS",4);`

2C) `int n=read(fd,NULL,1);` *fd is a valid file descriptor opened O_RDONLY*

2D) `int fd=open("/etc/passwd",O_TRUNC|O_CREAT|O_WRONLY,0666);`

Submission

Problems 1 and 2 will be submitted on paper. This will be the only unit where we have paper submissions. After this, all submissions (including problem 3 of this problem set) will be submitted via the ece357 virtual machine environment.

Please do not hand-write your answers, *including submitting printouts of your One-Note sheets. Compose a new document in PDF or text format with the answers, and print it out.*

Problem 3 -- Writing a simple stdio library

In this assignment, you will write a small library, i.e. code which is meant to be used by other programs. The library will mimic the stdio library, with a greatly reduced feature set. We'll then use the library in a simple test program. The library will implement the following 5 functions:

```
struct MYSTREAM *myfopen(const char *pathname, const char *mode);

struct MYSTREAM *myfdopen(int filedesc, const char *mode);

int myfgetc(struct MYSTREAM *stream);

int myfputc(int c,struct MYSTREAM *stream);

int myfclose(struct MYSTREAM *stream);
```

`myfopen` takes a pathname and a `mode` string which is either "r" or "w" (same as the `fopen`). However, only these two values need to be supported for this assignment. If you want to support things like "w+" knock yourself out, but it isn't extra credit. The `myfopen` function will: 1) verify that the mode string is either "r" or "w" and if not, set `errno` to EINVAL and return NULL.

2) dynamically allocate a `struct MYSTREAM` using malloc. This pointer is also the return value from the function. If `myfopen` fails at any point, you will return NULL, just like fopen. *Note:* the layout of `struct MYSTREAM` is up to you as this is an opaque or "private" (if C had such a notion) data structure for the library's use only.

3) Allocate a BUFSIZ-sized buffer for this stream. You can either do that with a separate malloc, or you can make

the buffer part of MYSTREAM.

4) Call the open system call with the proper and required parameters to open the file. Save the file descriptor within your data structure. If the underlying open fails, this will set errno for you. So just return NULL signifying error. Note that it is not appropriate for your library to print any error messages! Applications give error messages to the user, not libraries.

5) perform any other needed initialization of your data structure, and return

myfdopen works the same as myfopen except it takes a file descriptor that is already open, e.g. the result of the application calling open, or one of the 3 standard file descriptors (0,1,2) that is open when any program starts. It is assumed that the file descriptor was opened with a mode that is identical to or compatible with mode.

myfgetc is equivalent to the standard library function fgetc. It returns the next character from the stream (promoted to an int, so that EOF is distinct from character code \xFF) What happens depends on the buffer. If the buffer contains at least one character, simply return that next character. But if the buffer is empty (it will always be empty when the stream is first opened) you must first call the system call read to obtain a BUFSIZ worth of bytes. This doesn't mean that all these bytes will actually be read, so you must keep track of how many bytes are actually in the buffer! When the buffer is empty and read returns an end-of-file condition, myfgetc will return the value EOF (-1) and make sure errno=0. If the read system call returns an error, also return -1, and it is up to the application to read errno and make a valid error report.

myfputc functions the same as fputc with the important caveat that the stream is always in file-buffered mode (i.e. unbuffered and line-buffered are not supported.) Take the supplied character and place it into the buffer. If the buffer is now full then call the write system call to flush the entire buffer. If write returns an error, return the value -1 (and note that errno will already contain the error code for the application). If write returns 0, which will not happen under ordinary circumstances, treat this as an error. If write is not able to write out the entire buffer, that is a **partial write**, which is described in the lecture notes. For this assignment, we'll treat a partial write as an error too (although that is not proper) because you already have enough work to do in the first week! The return value of a successful myfputc is the same as the parameter c.

myfclose: If the stream was opened for reading, myfclose simply calls close on the underlying file descriptor, returning 0 if this succeeds, or -1 if there is an error. Any characters that had been in the buffer but were not read with myfgetc are silently discarded. After the close, call free to free the dynamically allocated memory. If the stream was opened for writing, you must first call write to flush the buffer, before you close the file descriptor, and a failure of either write or close must result in returning -1.

A note about libraries and compilation: Although this material is part of a pre-requisite course, it is worth repeating how libraries work in C. You should place your miniature stdio-like library in a C source file such as mylib.c and also create a file mylib.h which defines the structure struct MYSTREAM and provides prototypes for the 5 public library functions defined above. To use your library for example in the C program app.c, you must have a #include "mylib.h" line in that file. Then compile the "library" together with the "app" e.g.

```
gcc -o app app.c mylib.c
```

You should **not** have something like #include "mylib.c" in your application!

This is not intended as a complete tutorial and there are other methods, such as Makefiles, which you are permitted to use. Just make sure, if you are using more complicated tools, that you actually understand what the tools are doing!

Problem 4 -- Testing your stdio library

Now write a simple test program which processes input one character at a time and replaces all occurrences of the TAB character with four spaces. It can be invoked in any of the following ways:

```
tabstop -o outfile infile

tabstop -o outfile

tabstop infile

tabstop
```

In the first syntax, the file `outfile` is opened for writing, and `infile` is opened for reading. In the second syntax, the input is standard input. In the third syntax, an input file is specified but the output is standard output. In the last syntax, both standard input and standard output are used.

Use only myfopen, myfdopen, myfgetc, myfputc, and myfclose to handle file I/O including reading/writing standard input/output. Do not use any other library functions, nor direct system calls, for this purpose. Note that it will in fact be necessary to use all 5 of these functions for a complete and correct tabstop program!

Exhibit proper error handling and reporting, as defined in the lecture notes! You are permitted to use the stdio library such as `fprintf` and of course `perror` or `strerror` for error reporting and argument processing. It is acceptable for you to terminate the program upon the first error encountered and not make any attempt at graceful recovery.

Your program should return an exit status (we'll learn more about this in unit 3) of 0 if everything was OK, or 255 if anything went wrong.

Since your library functions have the same interface as the stdio library, you could test your tabstop program first by writing it with fopen, fgetc, etc. and then changing things to use your own library.

SUBMISSION: For problems 3/4 (and 5 if attempted) place all relevant source code, demonstration of running and results, and any requested writeups, in the `hwsub/ps1p3` directory on the VM. If you do the extra-credit, just submit one assignment, i.e. there is no need to submit a separate set of code and results for the non-EC version.

Problem 5 -- EXTRA CREDIT (+1): Buffer Sizes and Performance

For extra credit, experiment with buffer sizes other than the default BUFSIZ (which is 4096). To do this, change the definitions of the two open functions to the following (this breaks compatibility with fopen/fdopen):

```
struct MYSTREAM *myfopen(const char *pathname, const char *mode, int bufsiz)

struct MYSTREAM *myfdopen(int fd, const char *mode, int bufsiz)
```

Also, add a `-b buffer_size` command-line option to your `tabstop` program. You can now use a script (or a series of manual command invocations) with varying values e.g. `tabstop -b 256` to test the performance using different buffer sizes. For your test values, it is recommended to use powers of 2 starting from 1 and going up to 64K.

Invoke your tabstop program using the `time` command to determine the run time of your program. Make a table or graph of the run time vs buffer size. Present a brief narrative explaining your results and your explanation for what you observed.

Important: in order to obtain meaningful results with respect to the measuring granularity and precision of the time command, you'll need to make sure your program takes a few seconds. Create a test file of at least 1MB. Use files for input and output, not the terminal, as the slowness of the terminal will distort the results.