Problem 1 -- What is kernel mode / what is user mode?

Explain for each of the following items whether they can be accomplished entirely within user mode, or require a system call? If the latter, identify what specific system call or calls would be used. If the answer is "it depends," tell me on what it depends!

a) reading one char from a file on disk using fgetc

b) calling a function

c) `struct whack *s = malloc (sizeof *s);`

d) getting the current time of day

e) `double e = exp(1.0);`

Problem 2 -- What happens if...

For each of the following code fragments, tell me "what happens if". Explain your answer as appropriate. For example, if there is a system call error, what specific error and why? If a variable is assigned to, what value winds up in that variable? You're allowed to run these examples to see what happens.

```
a) n=write(1,"XYZ",3);
```

```
b) fd=open("/ooopsy",O_WRONLY|O_CREAT|O_TRUNC,0666);
```

```
c) char buf[3]; for(;;) printf("%d\n",read(fd,buf,3)); /* fd is a file on disk
open for reading; the associated file is 10 bytes long */
```

```
d) n=write(1,NULL,1);
```

```
e) n=close(-1);
```

Problem 3 -- Use of system calls in a simple concatentation program

The objective of this assignment is to write a simple C program which is invoked from the command line in a UNIX environment, to utilize the **UNIX system calls** for file I/O, and to properly handle and report error conditions.

The program is described below as a "man page", similar to that which describes standard UNIX system commands. The square brackets [ ] are not to be typed literally, but indicate optional arguments to the command.

This program is similar to the standard, common `cat` program which is found on any UNIX system.

```
meow - concatenate and copy files

USAGE:

      meow [-o outfile] infile1 [...infile2....]
      meow [-o outfile]

DESCRIPTION:

This program opens each of the named input files in order, and concatenates
the entire contents of each file, in order, to the output.  If an
outfile is specified, meow opens that file (once) for writing, creating it
if it did not already exist, and overwriting the contents if it did.
If no outfile is specified, the output is written to standard output, which
is assumed to already be open.

During the concatenation, meow will use a read/write buffer size of
4096 bytes.

Any of the infiles can be the special name - (a single hyphen).  meow
will then concatenate standard input to the output, reading until end-of-file,
but will not attempt to re-open or to close standard input.  The hyphen can be
specified multiple times in the argument list, each of which will cause
standard input to be read again at that point.

If no infiles are specified, meow reads from standard input until eof.

At the end of concatenating any file (including standard input), meow
will print a message to standard error (not standard output)
giving the name of the file, the number of bytes transferred, and the
number of lines.
In the case of standard input, the name will appear as <standard input>


EXIT STATUS:
```

```
program returns 0 if no errors (opening, reading, writing or closing)
were encountered.
Otherwise, it terminates immediately upon the error, giving a proper
error report, and returns -1.

EXAMPLES:

meow file1 - file 2

 (read from file1 until EOF, then standard input until EOF, then file 2,
  output to standard output)

meow -o output - - file3

 (read from standard input until EOF, then read again from standard input
  until EOF, then read file3 until EOF, all output to file "output")
```

IMPORTANT NOTES

- **Read the man pages!**
- Use UNIX system calls directly for opening, closing, reading and writing files. Do not use the stdio library calls such as fread for this purpose. [You may use stdio functions for error reporting, argument parsing, etc.]
- This is a command-line or "batch" program. It is not a user-friendly interactive program. As such, you should not issue prompts such as "enter the file name."
- As part of your assignment submission, show sample runs which prove that your program properly detects the failure of system calls, and makes appropriate error reports to the end user. For example, you can test the `open` system call error handling by specifying an input file that does not exist. Read, write and close errors are harder to generate at this stage of the course -- you could optionally try using a USB memory device that you yank out while the program is running -- but regardless you must still properly check for and report errors on these system calls.
- As a matter of programming elegance and style, avoid cut-and-paste coding! *E.g. the case of reading from standard input vs reading from a file* The program should be around 100 lines of C code. Programs which are say 200 lines long are inelegant and will be graded accordingly.
- Make sure to consider unusual conditions, such as "partial writes," even though you will not necessarily be able to generate these conditions during testing. The lecture notes contain discussion of this "feature" of the write system call. Will your program handle this correctly?
- Your program must have proper error reporting (what/how/why) as discussed in class and/or lecture notes.
- Exit status: we'll cover this formally in units 3 and 4. For now, exit status is the value passed to the `exit` system call, or the value returned from the function `main`
- Binary files: A "binary" file is not meant to be looked at directly in the terminal. It is the opposite of a "text file." If you accidentally `cat a.out`, your screen will fill up with garbage and possibly your session may lock up. This is because the seemingly random bytes present in any binary file will wind up being non-printable control characters.

  Your program must work for any number of files of any size and for "binary" files. Your testing must include this. One way to verify that your program is concatenating each file as "just a bunch of bytes" is to use a "checksum" or "hashing" command, e.g.

```
$ cat a.out foo.jpg   # NOOOO!!!! Don't do this!!!!
$ cat a.out foo.jpg | sha1sum
4efaa1fe1c674ad9380c6e7152cd712f5377a4ca  -
$ cat a.out foo.jpg >output.bin
$ sha1sum output.bin
4efaa1fe1c674ad9380c6e7152cd712f5377a4ca  output.bin
$ meow a.out foo.jpg | sha1sum
4efaa1fe1c674ad9380c6e7152cd712f5377a4ca  -
```

You can use other utilities such as md5sum, sum, sha256sum, etc. You can also use wc to see the total byte count of the output.

If you don't understand the command lines above, read up on them!! There are many UNIX reference sources on the 'net which will be useful.

- Make sure your program works correctly when there are multiple instances of the single hyphen (standard input) in the argument list
- It is assumed that you already know how to parse arguments in C. However, it is not desirable that you get bogged down with this detail. Look at the `getopt` library function for a quick and easy way to parse arguments.
- *Question to ponder: How can you specify an input file which is literally a single hyphen, and not have it be confused with a command-line flag or the special symbol for standard input?*