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.

```
minicat - concatenate and copy files

USAGE:

      minicat [-b ###] [-o outfile] infile1 [...infile2....]
      minicat [-b ###] [-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, minicat 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.

Any of the infiles can be the special name - (a single hyphen).  minicat
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, minicat reads from standard input until eof.

The optional -b### argument can be used to specify the buffer size in bytes.

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.
```

- Use UNIX system calls directly for opening, closing, reading and writing files. Do not use the stdio library calls such as fopen for this purpose. [You may use stdio functions for error reporting, argument parsing, etc.]
- 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.
- Experiment with different read/write buffer sizes. As part of your submission, compare the performance of your program with very small buffer sizes such as 1, 2, 4, 8 with that using a 4096 byte buffer. One way to measure the run time of the program is by prefixing its command line with the `time` shell command. Another way is to use system calls to measure the time of day at the start and end of the program.
- In order to meaningfully do these tests, you need to process a significant number of data, such that the run time under the slowest case (buffer size of 1) is on the order of many seconds. If you use a 1K test file, all of your results will be about the same, because of the granularity of the system clock.
- Note that I am not asking you to produce elaborate graphs, regressions and analyses in this assignment. Just tell me subjectively if increasing the buffer size improves performance. Speculate on why that might be.
- 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 them 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 unit 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: Your program must work for any number of files of any size and for "binary" files which contain non-printable characters. Test this by generating several large files filled with random bytes (look up `/dev/urandom` and the `dd` command) and `cat` them together to an output file, then do the same with your code. compare the results (look up the `sum, md5sum, sha1sum, sha256sum` commands)
- *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?*

Submission

Submit (at least) the following for problem #3, in hard-copy:

- Program source code listing, in black & white, monospaced font at least 10pt.
- Legible screenshot or cut-and-paste of text terminal window commands and responses demonstrating successful run of your program and error condition properly detected and reported.
- ditto, demonstrating proper binary file operation
- Experimental raw data (run times vs buffer sizes)
- Analysis of results. [A paragraph or two will suffice]

Due Date: Problem 3 would normally be due on 9/19. However, class will not meet that day. Therefore, the due date is extended until 11:59PM on 9/20. Please use the homework submission box adjacent to my office rm 810.