

**Problem 3 -- Use of system calls in a simple concatenation 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.

kitty - concatenate and copy files

**USAGE:**

```
kitty [-o outfile] infile1 [...infile2....]  
kitty [-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, kitty 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, kitty will use a read/write buffer size of 4096 bytes.

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

At the end of concatenating any file (including standard input), kitty will print a message to standard error giving the number of bytes transferred (for that file), and the number of read and write system calls made. If the file was observed to be a "binary" file, an additional warning message will be printed. The name of the file will also appear in this message. 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:**

kitty file1 - file 2

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

kitty -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.]
- 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: Your program must work for any number of files of any size and for "binary" files which contain non-printable characters. For our purposes, these characters are ASCII codes < 32 decimal (excepting 9 through 13) or >=127. Another definition is `!(isprint(c) || isspace(c))`. Test the behavior with binary files 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`, or `sha256sum` commands)
- 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?*