CMPE382

Operating Systems

Project #1: A Confusing Text Editor

Due Date: XX/XX/XXXX Friday

This project must be implemented in C (and not anything else)

It is advised that this project is implemented using gcc compiler (Native compiler for linux)

This project is assigned to each student individually.

Objectives

There are five objectives to this assignment:

- To familiarize yourself with the Linux programming environment.
- To develop your programming skills in C.
- To learn how processes are handled (i.e., starting and waiting for their termination).
- To be exposed to threaded execution, and performance issues.

Overview

This assignment is composed of two main tasks. First, you will implement a set of simple command-line programs in order to edit a text document. Second you combine them into a single program that provides additional functionality by combining them in various ways.

In the first task, you need to write several command-line programs that are taking inputs and execute on a **command line interpreter** or **shell.** If you are not familiar with what a shell (or a terminal in linux terms) is, I advise you to install a Linux distribution and examine the Linux terminal for basic understanding what it is capable of.

The programs that you need to write in the first phase are as follows:

- **search <keyword> [-c] <inFile>**

The search command (program in the first phase) always takes two arguments. A keyword, which is a sequence of characters (or a string in C, as C defines strings as an array of characters) and an input file. The command searches the input file for the occurrence of keyword and **prints each line that contains the keyword**. The The third argument "-c" is optional. If included, the command also prints the number of times the keyword has been found by printing the number on the screen.

- **replace <targetKeyword> <sourceKeyword> [-c] <inFile>**

Works similar to search, however replace takes two keywords, one target and one source keyword, replaces of all occurrences of targetKeyword with sourceKeyword.

- **insert <insertedkeyword> [-c] <-a OR -b> <targetkeyword> <inFile>**

Insert command inserts the "insertedKeyword" before or after each occurence of the targetKeyword. -a (after) or -b (before) arguments decide whether the insertedKeyword is placed before or after the targetKeyword.

- **lineCount <inFile>**

Counts the number of lines in the input File and prints it

- **split <CharacterCount> <inFile>**

Split command splits the lines in the input file into multiple lines. The output of the command is the same content as the input file, however each line cannot have more than "CharacterCount" characters. If a line within the input fi le contains more characters, it is divided into multiple lines as appropriate.

- **head <lineCount> <inFile>**

Prints the first "lineCount" lines in the input File

- **tail <lineCount> <inFile>**

Prints the last "lineCount" lines in the input File

- **mid <startLine> <endLine> <inFile>**

Prints the lines between "startline" and "endline" in the input File

```
prompt>
prompt> replace "cow" "chicken" myFile.txt
prompt> replace "cow" "chicken" -c myFile.txt
prompt> split 100 myFile.txt
prompt> mid 100 200 myFile.txt
```

For example, all of the above commands are legitimate. In the example the empty line, denoted by "prompt>" is the assumed shell in Linux. The first command (in the second line) replaces all occurrences of word cow with chicken and prints the contents. The second command replaces all occurrences also, but at the last output line it also prints the number of occurrences of cow in the document. The third command splits each line in myFile.txt into 100-line subtexts. The last command prints the contents in between 100th and 200th lines in the input file.

In order to implement these functions in C, you obviously will need string manipulation and file operations. I suggest you to check out the "string.h" library, along with fgets(), strtok(), and strsep() methods in the library. We will be going over a simple example in the class.

Also notice that you can write small programs for each of these functions even in the beginning of the class, without much dedicated time, without knowledge required in the course.

The program should operate in this basic way: when you type in a command (in response to its prompt), it creates a child process (a thread) that executes the command, or sub-command you have just entered, produces its output on the screen, and then prompts for more user input when it has finished.

ADDITIONAL FUNCTIONALITY AT A COMMAND LINE

As the second step, you will merge all of these functionalities into a single program. The program will ask the user to enter commands until the user enters quit, analyze the user command and act appropriately. In addition, the user will be able to merge multiple functionalities using 3 different operators. These operators are:

- ";": meaning a simultaneous run (in threaded mode)
- ":" meaning a sequentially (in normal mode)
- ">" signifying an output file the result can be written into.

Each user input line may contain multiple commands separated with one of the above characters. Each of the commands separated by a **:** should be run simultaneously, and by a **;** should be run concurrently (in threaded mode).

In any case, if multiple commands are provided, each sequence of commands get only one input file. For example, the following lines are all valid and have reasonable commands specified:

```
prompt>
prompt> replace "cow" "chicken" myFile.txt
prompt> replace "cow" "chicken" : split 100 myFile.txt
prompt> replace "cow" "chicken" > out.txt : split 100 > out2.txt myFile.txt
prompt> replace "cow" "chicken" ; split 100 myFile.txt
```

For example, on the third line, the commands `replace "cow" "chicken"` and `split 100` should all be running sequentially; so first replace command will be executed and then split. On the fifth line, the commands `replace "cow" "chicken"` and `split 100` should all be running simultaneously. As a result, you **should** see that their output is intermixed. Notice that all commands are using the same file as input: myFile.txt in all examples.

The fourth line denotes how output files will be handled. In the example, the replace command will be executed, and its output will be written to out.txt. Next, split command will be executed over the output of replace, and its result will be written into output2.txt. (check out the fork() command it can be very useful here.)

To exit the shell, the user can type `quit`. This should just exit the shell and be done with it (the exit() system call can be useful here). If the "quit" command is on the same line with other commands, you should ensure that the other commands execute (and finish) before you exit your shell.

This project is not as hard as it may seem at first reading; in fact, the code you write will be much smaller than this specification. Writing your shell in a simple manner is a matter of finding the relevant library routines and calling them properly. Your finished programs will probably be under 1000 lines, including comments. If you find that you are writing a lot of code, it probably means that you are doing something wrong and should take a break from hacking and instead think about what you are trying to do.

Program Specifications

Your C program must be invoked exactly as follows:

./editor

POINTERS AND SUGGESTIONS

Defensive programming is an important concept in operating systems: an OS can't simply fail when it encounters an error; it must check all parameters before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner; by "reasonable", we mean print an understandable error message and either continue processing or exit, depending upon the situation.

You should consider the following situations as errors; in each case, your shell should print a message and exit gracefully:

- An incorrect number of command line arguments to your program.
- A command without an input file specified.

For the following situation, you should print a message to the user and **continue** processing:

- A command does not exist or cannot be executed. (Optional for this project, but let's test our C programming skills.)

Optionally, to make coding your program easier, you may print an error message and continue processing in the following situation:

- A very long command line (for this project, over 512 characters including the '\n').

Your program should also be able to handle the following scenarios, which are **not** errors:

- An empty command line.
- Extra white spaces within a command line.

In no case, should any input or any command line format cause your program to crash or to exit prematurely. You should think carefully about how you want to handle oddly formatted command lines (e.g., lines with no commands between a **;**). In these cases, you may choose to print a warning message and/or execute some subset of the commands. However, in all cases, your shell should continue to execute!

Hints

Your program is basically a loop: it repeatedly prints a prompt, parses the input, executes the command specified on that line of input, and waits for the command to finish, if it is in the foreground. This is repeated until the user types "quit" or ends their input.

You should structure your loop such that it creates a new process (metod call? opening a thread? figure it yourselves) for each new command. There are two advantages of creating a new process. First, it protects the main shell process from any errors that occur in the new command. Second, it allows easy concurrency; that is, multiple commands can be started and allowed to execute simultaneously (i.e., in parallel style). For this, you need to work with threads or forks!

To simplify things for you in this first assignment, we will suggest a few library routines you may want to use to make your coding easier. You are free to use these routines if you want or to disregard our suggestions.

To find information on these library routines, look at the manual pages (using the Unix command **man**). You will also find man pages useful for seeing which header files you should include.

Parsing

For reading lines of input, you may want to look at **fgets().** To open a file and get a handle with type **FILE *** , look into **fopen().** Be sure to check the return code of these routines for errors! (If you see an error, the routine **perror()** is useful for displaying the problem.) You may find the **strtok() or strsep()**routines useful for parsing the command line (i.e., for extracting the arguments within a command separated by whitespace or a tab or ...).

Executing Commands

Look into `fork(), execvp(), wait/waitpid()`and `pthread's.`

The `fork()` system call creates a new process. After this point, two processes will be executing within your code. You will be able to differentiate the child from the parent by looking at the return value of `fork;` the child sees a 0, the parent sees the `pid` of the child.

You will note that there are a variety of commands in the `exec` family; **for this project, you can use** `execvp()` or the `pthread` functionalities that we have shown during the class (However, you possibly need to check out other methods of the pthread library for perfect functionality).

Another concept that you will be needing to implement the batch task is understanding how programs get input parameters in C. For a program the first argument specifies the program that should be executed, with the full path specified; this is straight-forward. The second argument, `char *argv[]` matches those that the program sees in its function prototype:

```
int main(int argc, char *argv[]);
```

Note that this argument is an array of strings, or an array of pointers to characters. For example, if you invoke a program with:

```
foo 205 535
```

then argv[0] = "foo", argv[1] = "205" and argv[2] = "535". **Important:** the list of arguments is terminated with a NULL pointer; that is, argv[3] = NULL.

The `wait()/waitpid()` system calls allow the parent process to wait for its children. Read the man pages for more details.

Miscellaneous Hints

- Remember to get the **basic functionality** of your program working before worrying about all of the error conditions and end cases. For example, first focus on getting a single command running. Next, try working on multiple jobs separated with the **:** character. Finally, make sure that you are correctly handling all of the cases where there is miscellaneous white space around commands or missing commands.
- If you want to execute multiple programs, you'll need to loop over each line and execute the programs one by one. But you can't use execvp because that immediately replaces the currently executing process (your C program) with the process executed via the shell, meaning that the rest of your C program will never be executed. You need to learn how to use fork() combined with execvp so you can execute child processes. You first call fork() to create a child process, and then from the child process you call execvp. Fork + Exec is a common strategy in UNIX environments to launch other processes from a parent process.
- We strongly recommend that you check the output of all your commands from the very beginning of your work. This will often catch errors in how you are invoking these new system calls.
- **Beat up your own code!** You are the best (and in this case, the only) tester of this code. Throw lots of junk at it and make sure it behaves well. Good code comes through testing -- you must run all sorts of different tests to make sure things work as desired. Don't be gentle -- other users certainly won't be. Break it now so we don't have to break it later.

- Keep versions of your code. More advanced programmers will use a source control system such as SVN or CVS. Minimally, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). By keeping older, working versions around, you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be.

- For testing your code, you will probably want to run commands that take a while to complete. when multiple copies are run in parallel you should see the output from each process interleaved.

Grading

For this project, you need to hand in 3 distinct items:

- Your source code (no object files or executables, please!, but a compile script using gcc)
- A Makefile for compiling your source code OR your code + user friendly instructions with no file placement requirements (everything should be in a single file, you may consider sharing a test input file also)
- A `README` file with some basic documentation about your code

Hand in your source code. Handing in a compressed folder would be the best strategy.

No late projects will be accepted!

Finally, we would like to see a file called `README` describing your code. This file should have contain the following three sections:

- Design overview: A few paragraphs describing the overall structure of your code and any important structures.
- Complete specification: Describe how you handled any ambiguities in the specification. For example, for this project, explain how your shell will handle lines that have no commands between semi-colons.
- Known bugs or problems: A list of any features that you did not implement or that you know are not working correctly

Finally, while you can develop your code on any system that you want, make sure that your code runs correctly on a machine that runs the Linux operating system.