CSci 4061: Intro. to Operating Systems

Assignment 4: Multithreaded Vim Clone

Due:

December 3rd by midnight. You should work in groups of two.

Purpose:

To develop a vim like `editor` program in C that will allow users to edit text files from the command line. The purpose of the lab is to expose you to mulithreaded programming. The editor will use multiple threads for auto-saving and a seamless UI experience. The UI interface is implemented with the curses library, we provide all the functions necessary to handle interactions with the user. Curses is a text-based windowing system that does **not** need X windows. You will need to include both the `-lpthread` & `-lcurses` library flags in your makefile (make sure these libraries are linked at the end of the compile line). For further information about the curses library can be found at `http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Curses.pdf`

Description:

Your main program `editor` will contain three threads; a UI thread, an auto-save thread, and a router main thread. To store the text from the open text file you need to implement a text buffer, `textbuff`.

`textbuff`

`textbuff` is responsible for storing the text from an open text file with each line stored in a node in an internal linked list. Additionally you must implement the specified methods for accessing and modifying the text buffer:

Required Methods:

`void init_textbuff(char* file)` Initialize the data structures necessary for using the text buffer. Takes as argument the path to a `file` used to initialize the `textbuff`

`int getLine(int index, char** returnLine)` Fetches the line `index` from the text buffer and places it in `returnLine`

`int insert(int row, int col, char text)` Inserts `text` into `textbuff` at line `row` and inside of that line position `col`. If after insertion the length of the line is longer than `LINEMAX` then split the line into two.

`int getLineLength()` Returns the number of lines in `textbuff`

`int deleteCharacter(int row, int col)` Delete a single character from line `row` and position `col`

`void deleteBuffer()` Removes everything from `textbuff`. **Note:** remember to free any dynamically allocated memory and make sure getLineLength() returns the correct values.

Suggested Methods: The following are not required but may help in the above implementations.

`int deleteLine(int index)` Removes the line `index` from `textbuff` `int insertLine(int row, char* text)` Inserts `text` into `textbuff` as line `row`. This does not delete the old line `row` it moves it down one line.

Note: We've provided a `node` data structure at the beginning of `textbuff.h` it is not required that you use this but it is recommended.
Note: Any indexing starts at 0 so $(row, col) \Rightarrow (0, 0)$ refers the the first character on the first line.

We've provided a mutex `text_` to provide thread safe access to the `textbuff`. Our code use this mutex inside of redraw which uses the `getLine` text buffer method to display the text on the screen. If you use a different mutex to control access to the `textbuff` then you **MUST** change the mutex in the `redraw` method. Remember to synchronize all access to shared variables as needed.

Router

The router is responsible for reading messages from the message queue and acting on them. The message queue is a thread safe queue which passes messages. You are not required to use our message data structure but it is recommended. The editor interfaces with the message queue using `pop` and `push`, you must complete these methods in a thread safe manner. The router must block on an empty queue until an item is inserted. Similarly, the UI thread should block if the queue is full (reaches a max limit).

The Message queue passes messages with a `command` field which can take one of four named constants; `EDIT`, `SAVE`, `QUIT`, `DEL`. These correspond to The four different types of commands the router must handle. The UI thread will place messages of the appropriate type into the message queue and the router must handle the request.

Note: The maximum number of messages in the message queue is defined to be `QUEUEMAX`. When the router receives a `SAVE` request it should delete the temporary save file if it's present.

UI Thread

The UI thread is started with `start_UI`. Your tasks are to add code in `loop` and `input_mode` methods to pass commands to the router thread, specifically: insert, delete, save, and quit.

Auto-save Thread

The Auto-save thread periodically saves the `textbuff` to a temporary file. For example if the program was run using `./editor file.txt` then the temporary file would be `file.txt~`. The temporary file should be removed after the file is saved.

Error Handling:

You are expected to check the return value of all system calls that you use in your program and respond to the error appropriately. If your program encounters an error, a useful error message should be printed to the screen. Your program should be robust; it should try to recover if possible. If the error prevents your program from functioning, then it should exit after printing the error message. (The use of the `perror` function for printing error messages is encouraged.)

Documentation:

You *must* include a README file which describes your program. It needs to contain the following:

- The purpose of your program
- How to compile the program
- How to use the program from the shell (syntax)
- What exactly your program does

The README file does not have to be very long, as long as it properly describes the above points. Proper in this case means that a first-time user will be able to answer the above questions without any confusion.

Within your code you should use one or two sentences to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to increase readability.

At the top of your README file and main C source file please include the following comment:

```
/*login: itlabs_login_name
* date: mm/dd/yy
* name: full_name1, full_name2
* id: id_for _first_name, id_for_second_name */
```

Grading:

5% README file
20% Documentation within code, coding, and style (indentations, readability of code, use of defined constants rather than numbers)
75% Test cases (correctness, error handling, meeting the specifications)