CIS*3110 Assignment 2
Threaded spell-checker
Due date: Wednesday, March 27, 11:59pm

Introduction

In this assignment, you will practice using threads and thread synchronization. Your program will have a text-based menu, which will run on the main thread. The user will be able to start a potentially long-running, computationally intensive task. This task will be a spellchecker, which will be described below. Your program will start each task in a separate "worker" thread, while the menu will continue executing on the main thread. You may also spawn additional threads if you think you need them.

Each task will run independently. Once it completes, it will save its results to a file. The file save must be properly synchronized.

This will allow you to practice running worker tasks independently of the UI, while updating shared resources based on the tasks in separate threads. This is a very common pattern in many concurrent programs.

Task - spellchecking

The task will be spell-checking a text file. This task will requite two inputs: the name of the text file and the name of the dictionary file. File names for each task will be passed through the menu, as user-provided inputs.

It will then check every word in the text file against the word list in the dictionary. A standard American English dictionary file that comes with most Linux distributions will be provided to you for reference.

When the spellchecker completes, it will return:
- the total number of spelling mistakes
- the three most frequently occurring "unknown" or 'incorrect" words - i.e. three words that occur in the input text file, but are not found in the dictionary
  - This means that your spell-checker will need to keep track of each misspelled word, so it can count them

Your program will save the output for each spellchecker task into a specific file (see below for details). It will also create the final summary, which will be either displayed on exit, or saved into a final log file (again, details are provided below).

Menu

The menu will be used to start spell-checking tasks and display the output of the spell-checker. When your program starts, it will display two options - we will call this the "main menu":
1. Start a new spellchecking task
2. Exit

If the user starts a new task, your program will display a submenu asking the user:
- first to to enter for the dictionary file

- then enter input text file

Once these file names have been entered, your program will start a new spell-checking task in a separate thread, and will "return to the main menu" - i.e. re-display the main menu.

The user must also have an option to return to the main menu from this sub-menu without starting a new task - i.e. cancel the creation of a new task. How you do this is up to you. However, your UI must clearly state how the user should do this. Also, you must return to the main menu without terminating the program.

Exiting

In the main menu, the user also has the Exit option. If the user selects this option, your program must check if there are current tasks running. If there are none, it must state that all threads have finished running before it exits. If there are still threads running, it must display the number of running threads before exiting.

In addition, when the program exists, it must display a summary of the completed spell-checking tasks. This summary must include the following:

- The total number of files processed
- The total number of spelling errors encountered by the spell checker tests
- The three most common misspellings. These will be computed based on the three most common misspellings retired by each spell-checker tasks. For each word, display the numb of times if occurred in the summaries of individual tasks.

  For example, let's assume you spell checker examines 4 files and finds the following misspellings:
  - File 1: `foo`, `blorg`, `blarg`
  - File 2: `faa`, `blorg`, `blurg`
  - File 3: `foo`, `blorg`, `blerg`
  - File 4: `foo`, `blorg`, `blerg`

    Then the three most common misspellings in the final summary will be:
    `blorg` (4 times), `foo` (3 times), and `blerg` (2 times).

    If you run into a time - multiple works occurring `n` times - it does not matter which one you include in the summary.

The overall summary should look like this:

```
Number of files processed: XXX
Number of spelling errors: YYY
Three most common misspellings: word1 (A times), word2 (B times), word3 (C times)
```

Command line argument

Your program will take an optional argument, `-l`. When your program is executed with this argument, e.g. `./A2 -l`, the final summary discussed about is not displayed on exit. Instead, then the program exits, the summary is saved into the file called `username_A2.out` - the same file that each individual spellchecker output is also written to, as discussed below.

If the `-l` argument is not provided, the summary is displayed on the screen, as stated above. The contents and format of the summary must be the same, regardless of whether his saved to a file or displayed on the screen.

Memory leaks

If the user decides to exit while there are still running threads, we can get memory leaks. Proper cleanup here would involve thread cancellation, which is outside the scope of this assignment. Leak tests for A2 submission will keep this in mind. Penalties for memory leaks in A2 will be applied as follows:
- If your program terminates with no threads running and there are leaks - penalty is applied
- If your program terminates with some thread running and there are leaks - penalty is not applied

Error handling

Your program must do error handling - if either the input file or the dictionary file cannot be opened, your program must display an error message clearly stating the problem. This should be done on the main thread, so a thread for processing invalid files is not created.

Spellchecking task details

After each spell-checking task completes, your program must save the result of the test to the file named `username_A2.out`, where `username` is your UofG username. The output from each spell-checked file will be a single line and it will contain 5 items:
- The name of the spellchecked file
- number of errors
- most frequently misspelled words 1, 2 and 3. Sort these word in order of misspellings, i.e. the most frequently misspelled word comest first, then the second word, etc.
- pus single white spaces between each field and a new line at the end of the last field

A single line in `username_A2.out` will look something like this

```
someFile.txt 32 someword badword2 badword3
```

- this indicates that `someFile.txt` had 32 spelling mistakes in total. The most frequently misspelled words were `someword` (most common misspelling) `badword2` (2nd most common misspelling), and `badword3` (3rd most common misspelling).

File writing from multiple tasks completing simultaneously must not overlap. Each result must be saved independently and correctly.

In essence, task results will be entered into a queue and removed as they are saved to a file.

Keep in mind that the order of the output lines in `username_A2.out` does not have to match the order in which the tasks were started. They will be inserted in the order the tasks complete. So if we spell-check a long `file1.txt` and then a short `file2.txt,` the results for `file2.txt` might appear first.

Hints

Each spellchecker task will run independently, but saving the output to a file will require proper use of thread synchronization primitives to avoid race conditions. Updating the running totals for final summary is another place you have to be careful.

You will need to identify the specific critical sections in your code and use Pthreads tools such as mutexes and condition variables to ensure correct execution of your code. You can use the sample code provided in class, which implements some of the things you will need. The message queue will definitely be useful here.

You must also avoid busy waits in your code.

Other files

Include a `Makefile` that compiles all your source code. The `Makefile` must have the following targets:

- `A2checker`: creates the executable `A2checker` that runs your program. You are not creating a library in this assignment - just a normal stand-alone executable.
- `clean`: the target that deletes the executable

Your `Makefile` must compile your all code with `-Wall -g -std=c11 -lpthread`.

Also, please include a `README` indicating known issues / bugs, as well as whatever additional instructions we may need to run your code when grading it.

Grading

As always, your code must compile and run. Code that does not compile will receive an automatic grade of **zero** (**0**). Code that compiles with warnings will result in penalties.

When your code is graded, we will use Valgrind DRD tool to monitor your program once it runs. We will observe thread creation and termination, the use of thread synchronization tools, lock contention (one thread waiting for a lock for too long), data races (race conditions), incorrect use of Pthreads tools, and other issues. We will also use Valgrind to check for memory leaks and errors.

Valgrind will also be used to check for memory errors and leaks.

Submission and evaluation

Submit the assignment using Moodle. Submit only the source code, report, and the makefile. Bundle everything in a Zip archive.

The assignment will be marked using the standard CIS*3110 Docker image provided on the course website and discussed in class. We will download and run a fresh image, create container, and use Docker to run and grade your code. Make sure you test your code using the CIS*3110 Docker image!

The TAs will unpack your code and use the makefile to compile your `A2checker`. They will then try to run your executable. If the Makefile is missing, the make command does not work, or the program does not execute, you will lose marks. As a result, it is always a good idea to unpack and test the file you are submitting to be sure that what you submit actually compiles.

This assignment is individual work and is subject to the University Academic Misconduct Policy. See course outline for details.