pwords

TABLE OF CONTENTS

Implementation

`words` and `lwords` operate in a single thread, opening, reading, and processing each file one after another. With `pwords`, your task is to provide the same end-to-end functionality while using multiple threads. This means you are **not allowed to materialize your intermediate results** (i.e. write each thread's results to a separate file and aggregate these). Make sure to read through `word_count.h` and `Makefile` to see what macros are used for `pwords`. In particular, the `word_count_list_t` will differ from `lwords`.

`pwords.c` will serve as the driver program, meaning it will manage the creation and upkeep of the threads you create. Each file should be processed in a separate thread. We recommend you reference `pthread.c` to draw inspiration and clues as to how you should structure your code.

Synchronization

When implementing `words_count_p.c`, keep in mind the functionality of all these methods are identical to what you did in `words_count_1.c`. However, you must use synchronization techiques to ensure coordination amongst different threads to prevent race conditions.

Your synchronization must be fine-grained. Different threads should be able to open and read their respective files concurrently, serializing only their modifications to shared data. In particular, it is unacceptable to use a global lock around the call to the `count_words` function in `pwords.c`, since it would prevent multiple threads from concurrently reading files. We reserve the right to deduct points from such implementations. Instead, you should only synchronize access to the word count list data structure in `word_count_p.c`. You will need to ensure all such modifications are complete before printing the result or terminating the process.

We recommend that you start by just implementing the thread-per-file aspect (i.e. without synchronization). This will be quite similar to the code you've written in `word_count_1.c`. Your program might not even error, since multithreaded programs with synchronization bugs may *appear* to work properly much of the time. Once you're confident in the logic of your methods, then add in the necessary synchronization.

To help you find subtle synchronization bugs in your program, we have provided a decently large input for your words program in the `gutenberg/` directory. These files were generated from select stories from Project Gutenberg, making sure to choose short stories so that the word count program does not take too long to run. Make sure to compare the results of running `pwords` to running words to check if your output is correct. As stated before, this does not ensure your synchronization is correct, but it might alert you to subtle synchronization bugs that may not manifest for smaller inputs.

Gradescope questions

After correctly implementing `pwords` (i.e. passing autograder tests), compare `lwords` and `pwords` by answering the following questions.

1. Briefly compare the performance of `lwords` and `pwords` when run on the Gutenberg dataset. How might you explain their relative performance?

2. Under what circumstances would `pwords` perform better than `lwords`? Under what circumstances would `lwords` perform better than `pwords`? Is it possible to use multithreading in a way that always performs better than `lwords`?
