HW 2: **Lists and Threads**

CS 162

Due: February 14, 2020

Contents

In this homework, you will gain familiarity with threads and processes from the perspective of a user program, which will help you understand how to support these concepts in the operating system. Like last week, we hope that completing this assignment will prepare you to begin Project 1, and help you to see how you can do a lot of your development and testing of project code in a contained user setting before you drop it into the Pintos kernel.
This assignment is due at 11:59 pm on 2/14/2020.

1 Getting Started

Log in to your Vagrant Virtual Machine and run:

```
$ cd ~/code/personal/
$ git pull staff master
$ cd hw2
```

Run `make` to build the code. Five binaries should be created: `pthread`, `pwords`, and `lwords`.

1.1 Overview of Source Files

Below is an overview of the starter code:

`words.c`, `word_helpers.c`, `word_helpers.h`
These files contain a simple driver for parsing words, computing their aggregate count, and then writing the count to a stream, as in the previous homework.

`word_count.h`, `word_count_l.c`
These files provide the interface of and the implementation of the `word_count` abstraction using a Pintos list data structure for the representation. `word_count_l.c` is, in effect, the preferred solution to Homework 1. You must **not** modify these files.

`word_count_p.c`
This file is starter code to implement a version of `word_count_l.c` that not only uses the Pintos list data structure, but also provides proper synchronization when accessing the `word_count` data structure concurrently from multiple threads.

`pwords.c`
This file is starter code to implement the `pwords` application. This is a version of the `words` application, where each file is processed in a separate thread. You will need to modify it to spawn the threads and coordinate their work. It uses `word_count_p.c` to implement the `word_count` abstraction.

`list.c`, `list.h`
These files are the list library used in Pintos, which is based on the list library in Linux. You should be able to understand how to use this library based on the API given in `list.h`. You must **not** modify these files. If you're interested in learning about the internals of the list library, feel free to read `list.c` and the `list_entry` macro in `list.h`. You can find a good explanation of the `list_entry` macro here[1].

`pthread.c`
This file implements an example application that creates multiple threads and prints out certain memory addresses and values. In this assignment, you will answer some questions about this program and its

[1]https://stackoverflow.com/questions/15832301/understanding-container-of-macro-in-the-linux-kernel

output.

2 Observing a Multi-Threaded Program

The `pthread` application is an example application that uses multiple threads. First, read `pthread.c` carefully. Then, run `pthread` multiple times and observe its output. Answer the following questions on Gradescope:

1. Is the program's output the same each time it is run? Why or why not?
2. Based on the program's output, do multiple threads share the same stack?
3. Based on the program's output, do multiple threads have separate copies of global variables?
4. Based on the program's output, what is the value of `void *threadid`? How does this relate to the variable's type (`void *`)?
5. Using the first command line argument, create a large number of threads in `pthread`. Do all threads run before the program exits? Why or why not?

3 Using Multiple Threads to Count Words

The `words` program operates in a single thread, opening, reading, and processing each file one after another. In this exercise, you will write a version of this program that opens, reads, and processes each file in a separate thread.

First, read and understand `pwords.c`, which is a first cut at a program that intends to use multiple threads to count words.

Your task is to properly implement the `pwords` application. You will make changes to `pwords.c` and `word_count_p.c` to complete this task. It will need to spawn threads, open and process each file in a separate thread, and properly synchronize access to shared data structures when processing files. **Your synchronization must be fine-grained.** Different threads should be able to open and read their respective files concurrently, serializing only their modifications to the shared data structure. In particular, it is unacceptable to use a global lock around the call to `count_words()` in `pwords.c`, as such a lock would prevent multiple threads from reading the files concurrently. 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, without synchronizing updates to the word count list.Can you even detect the errors? Multithreaded programs with synchronization bugs may appear to work properly much of the time. But, the bugs are latent, ready to cause problems.

To help you find subtle synchronization bugs in your program, we have provided a somewhat large input for your `words` program in the `gutenberg/` directory. To generate these files, we selected some stories from among the most popular books made freely available by Project Gutenberg[2], making sure to choose short stories so that the word count program does not take too long to run. You should compare the result of running your `pwords` program on the Gutenberg dataset to the result of running `words` on the Gutenberg dataset and ensure they are same. This does not guarantee that your code is correct, but

[2]https://www.gutenberg.org/ebooks/search/?sort_order=downloads

it might alert you to subtle concurrency bugs that may not manifest for smaller inputs.

Hint #1: The `Makefile` that we provide will compile your `pwords.c` program with the two flags `-DPINTOS_LIST -DPTHREADS`, which select a definition of the word count structure that not only uses Pintos lists, but also includes a mutex that you may find useful for synchronization. Unlike the `lwords` exercise, in which the `word_count_t` structure was `typedef`'d to the Pintos list structure directly, the `word_count_t` structure now *contains* the Pintos list structure and a mutex. We expect your code in `word_count_p.c` to be similar to your code in `word_count_l.c`, with syntactic changes according to the new `word_count_t` structure and added synchronization to allow concurrent use of the `word_count` API as needed for `pwords`.

Hint #2: The multiple threads should aggregate their results without reading from or writing to any intermediate files. **Attempting to open or read from any files other than the ones passed as input to your program may cause autograder tests to fail or not be run.**

4 Additional Questions

Answer the following additional questions on Gradescope:

6. Briefly compare the performance of `lwords` and `pwords` when run on the Gutenberg dataset. How might you explain the performance differences?
7. Under what circumstances would `pwords` perform better than `lwords`? Under what circumstances would `lwords` perform better than `pwords`? Is it possible to use multiple threads in a way that always performs better than `lwords`?

5 Autograder and Submission

5.1 Autograder

To submit and push to autograder, first commit your changes, then do:

```
$ git push personal master
```

Within a few minutes you should receive an email from the autograder. (If you haven't received an email within half an hour, please notify the instructors via a private post on Piazza.) Please do not print anything extra for debugging, as this can interfere with the autograder.

Remember: Your final score in the autograder is not the maximum of your attempts, but rather your score for only your latest build. Any non-autograded components, like style and written portions, will be graded based on your last build for the assignment. Running a build after the deadline will consume slip days even if it doesn't change your score.

5.2 Written

Written portions of assignments will be submitted to Gradescope.

5.3 Survey

Remember to fill out your weekly survey! It will be due at the same time as this homework is due (February 14 at 11:59 PM). It will appear as a quiz on bCourses.