Homework 1: List

Due: Februrary 9, 2022

Contents

1 Overview

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. Along the way, you will gain experience with the list data structure used widely in Pintos, but in the context of a user program running on Linux. We hope that completing this assignment will prepare you to begin Project Userprog, where you will work with the implementations of these constructs in the Pintos kernel. Our goal is to give you experience with how to use them in userspace, to understand the abstractions they provide in an environment where bugs are relatively easy to debug, before having to work with them in Pintos. It will also 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.

1.1 Getting Started

To get started, log in to your development environment and get the starter code.

```
> cd ~/code/personal/
> git pull staff master
> cd hw-list
```

To build the code, run `make` which should create four binaries: `pthread`, `words`, `pwords`, and `lwords`. Make sure not to commit and push any binaries when submititng to the autograder. You can get rid of unnecessary binaries using `make clean`.

1.2 Skeleton

You'll notice that for some files, only the object files without the source are provided. Part of being able to program means being able to work with the abstractions you're provided, so we've left out implementations which are not necessary for you to complete this assignment.s

`list.h` provides the Pintos list abstraction which is taken directly from the Pintos source code. `list.c` provides the implementations, but you should be able to use this library solely based on the API given in `list.h`. **You must not modify these files**.

`word_count.h` defines the API you will implement. We have already provided necessary data structures `word_count_t` and `word_count_list_t` which **you must use**.

`words.o` and `word_count.o` provide compiled implementations of methods necessary to run the `words` program from Homework Intro. You can use the outputs of these programs as sanity checks on what your other programs that you'll build should output.

`word_count_l.c` will house your implementation of the the API in `word_count.h` using Pintos lists. The `Makefile` will provide the macro definition of `PINTOS_LIST` when compiling. When `word_count_l.c` is linked with the driver in `lwords.o` and compiled, it should result in an application `lwords` that behaves identically to the frequency mode of `words` but internally using Pintos lists instead of traditional linked lists as seen in Homework Intro.

Similarly, `word_count_p.c` will house your implementation of the API in `word_count.h` using Pintos lists and proper synchronization of the `word_count` data structure for a multithreaded program. Unlike `lwords`, you'll need to write the driver program in `pwords.c`. When `word_count_p.c` and `pwords.c` are put together and compiled, it should create an application `pwords` that behaves identically as `lwords` and frequency mode of `words` but internally uses multiple threads.

`word_helpers.h` provides an API for parsing and counting words. `word_helpers.o` provides compiled implementations for these methods.

`pthread.c` implements an example application that creates multiple threads andprints out certain memory addresses and values. You may find it helpful to base your `pwords.c` implementation off of `pthread.c`. While you won't be writing any code in `pthread.c`, you will be reading and analyzing it.

2 lwords

First, read `list.h` to understand the API. Focus on the examples given in the big docstring in the beginning of the file.

Next, thoroughly read through the data structures and methods in `word_count.h`. In particular, pay attention to the `word_count_t` and `word_count_list_t` structs. You may find it beneficial to see the compiler flags used in the `Makefile` and how that affects the struct definitions.

Finally, complete `word_count_l.c` to properly implement the `word_count` API given in `word_count.h`. **You must use the Pintos list API**. After you finish making this change, `lwords` should work properly (i.e. exhibit the same behavior as frequency mode of `words`).

The `wordcount_sort` function sorts the wordcount list according to the comparator passed as an argument. Although `lwords` uses the `less_count` function from `word_helpers.h` as the `less` argument, the `wordcount_sort` function should be general enough to work with any valid comparator passed in as the `less` argument. For example, passing the `less_word` function from `word_helpers.h` as the `less` parameter should. Check out some basics on function pointers[1] if you're having trouble understanding and writing the syntax.

[1] https://denniskubes.com/2013/03/22/basics-of-function-pointers-in-c/

3 pthread

Read `pthread.c` carefully. Then, run compile and run `pthread` multiple times and observe its output. Answer the following questions based on your observations.

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?

4 pwords

Implement `pwords` by completing `pwords.c` and `words_count_p.c`. `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.

When implementing `words_count_p.c`, keep in mind the functionality of all these methods are identical to what you did in `words_count_p.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 `count_words` function in `pwords.c`, since it would prevent multiple threads from concurrently reading files. 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 synchroniztion). This will be quite similar to the code you've written in `word_count_l.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[2], 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.

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`?

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

5 Submission

To submit and push to the autograder, push your changes to your repo which should trigger the autograder unless you're using slip days in which case you need to manually run it. Within a few minutes you should receive an email from the autograder. If you don't receive an email from the autograder within half an hour, please notify staff via a private post on Piazza. Your code should not include extraneous or debugging print statements as this will interefere with the autograder. Your written responses should be submitted to the Gradescope and will not be graded by the autograder.