HW 1: **Lists**

CS 162

Due: September 16, 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. Along the way, you will gain experience with the list data structure used widely in Pintos, but in a user program running on Linux. We hope that completing this assignment will prepare you to begin Project 1, 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.

This assignment is due at 11:59 pm on 9/16/2020.

1 Getting Started

Log in to your Vagrant Virtual Machine and run:

```
$ cd ~/code/personal/
$ git pull staff master
$ cd hw1
```

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

1.1 Overview of Source Files

Below is an overview of the starter code:

`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].

`word_count_l.c`
This file is the starter code for your implementation of the `word_count` interface specified in `word_count.h`, using the Pintos list data structure. We have already provided the type declarations for this in `word_count.h`. You must use those. Notice how the list element is embedded into the `struct`, rather than the `next` pointer. Also, the `Makefile` provides the `#define PINTOS_LIST` as a flag to `cc`. `words.c` is unchanged, as is its behavior. This exercise will cement your understanding of how to traverse and manipulate the kinds of lists that are used throughout Pintos. Your implementation of the `word_count` interface in `word_count_l.c`, when linked with the driver in `words.c`, should result in an application, `lwords` that behaves identically to `words`, but internally uses the Pintos list data structure to keep track of word counts.

`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

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

coordinate their work.

`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. This implementation of the `word_count` interface will be linked with your code in `pwords.c` to produce the `pwords` application. You will need to complete it.

`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 output.

2 Using Pintos Lists to Count Words

The Pintos operating system makes heavy use of a particular linked list library taken directly from Linux. Familiarity with this library will make it easier to understand Pintos, and you will need to use it in your solution for the projects. The objective of this exercise is to build familiarity with the Pintos linked list library in userspace, where issues are easier to debug than in the Pintos kernel.

First, read `list.h` to understand the API to library.

Then, complete `word_count_l.c` so that it properly implements the new `word_count` data structure with the Pintos list representation. You **MUST** use the functions in `list.h` to manipulate the list.After you finish making this change, `lwords` should work properly.

The `wordcount_sort` function sorts the wordcount list according to the comparator passed as an argument. Although `words` and `lwords` sort the output using the `less_count` comparator declared in `word_helpers.h`, the `wordcount_sort` function that you write should work with any comparator passed to it as the argument `less`. For example, passing the `less_word` function in `word_helpers.h` as the comparator should also work. If you're having trouble with function pointer syntax when implementing this, here[2] is a good tutorial.

Hint #1: We provide a `Makefile` that will build `lwords` based on these source files. It compiles your program with the `-DPINTOS_LIST` flag, which is equivalent to putting a `#define PINTOS_LIST` at the top of the file. This selects a definition of the word count structure that uses Pintos lists. We recommend reading the `word_count.h` file to understand the new structure definition so you can see how the Pintos list structure is being used.

Hint #2: The provided `Makefile` uses the `words.o` and `lwords.o` object files we have given you to provide the `main()` function in both the `words` and `lwords` programs. To ensure that your code works with this `main()` function (which we have not given you the source code of, but only the object files), you should ensure that your implementation of `word_count_l.c` adheres to the interface contained in `word_count.h`.

[2] https://www.geeksforgeeks.org/function-pointer-in-c/

3 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?

4 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[3], 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 it might alert you to subtle concurrency bugs that may not manifest for smaller inputs.

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

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.**

5 Additional Questions

Answer the following additional questions on Gradescope:

1. Briefly compare the performance of `lwords` and `pwords` when run on the Gutenberg dataset. How might you explain the performance differences?
2. 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`?

6 Autograder and Submission

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.

Your written responses submitted to Gradescope will not be graded by the autograder. It will be graded manually based on effort.