Assignment 1

CS 301 - Operating Systems

Due date: 26th August, 2021

1 Introduction

In this assignment, we will work with threads and processes from the perspective of a user program to understand how these concepts are used in the operating system. We will also gain experience with the list data structure that is used widely in Pintos, but in the context of a user program running on Linux, and would prepare you for Project-Module 1, where you will work with the implementations of these constructs in the Pintos kernel. This assignment is due on 26th August, 2021, end of day.

To get started, access the `assignment1` link assigned to you on GitHub classroom. After cloning the repo on your virtual machine, run `make` inside the folder to generate the binaries: `words`, and `lwords`.

2 Overview of Source Files

Below is an overview of the starter code:

`list.c, list.h`: These files form 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`. Do 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`.

`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. You must use the type declarations provided for this in `word_count.h`. 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`. Your implementation of the `word_count` interface in `word_count_l.c`, when linked with the driver in `words.o`, should result in an application, `lwords` that behaves identically to the frequency mode of words, but internally uses the Pintos list data structure to keep track of word counts.

3 Using 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 user-space, where issues are easier to debug than in the Pintos kernel.

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

Note 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.

Note 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, you should ensure that your implementation of `word_count_l.c` adheres to the interface contained in `word_count.h`.

4 Bonus Credit: Using Multiple Threads to Count Words

The `words` program operates in a single thread, opening, reading, and processing each file one after another. Can you write a version of this program that opens, reads, and processes each file in a separate thread? 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. You may start by just implementing the thread-per-file aspect, without synchronizing updates to the word count list.

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.

Before attempting this exercise, please create a copy of the `words` program and make the changes in that copy. Also, modify the `Makefile` to compile this copy for checking your changes.