Ian Pascoe

Jean-Yves Hervé

CSC 412

5 November 2018

Prog04 Report

Program 4 is centered around multi-process programming. Using the fork() call, a program is able to create multiple processes to handle multiple work-loads. For this assignment there are multiple versions to be created. Version 1 is a single process version of the program. I, first, get the file paths of every text file in the data set using the directory operations given in class and exercised in the previous assignment. Then, I figure out how big each group should be per process. This is done using simple arithmetic. This is found by dividing the total source count by the number of processes to create. Unfortunately, using this number will create groups too large if the total source count is an odd number. I could not figure out a way to construct a lopsided 2D array of strings to account for this, so this array will contain some “system garbage” naturally accrued by an uninitialized string. Once this 2D array of strings is created, I go through each file in the 2D array and replace the file path with the actual contents of the file. This is done using simple file reading functions (fgets). From there, I take the 2D array of contents and put them in a temporary 1D array so that way it can be sorted. I sort these strings using the index of the contents, which is the second number in the line. These sorted contents are then put back in the 2D array and printed to the output file “Version 1/Output/output.txt”.

Version 2 is implemented using multi-process programming. It reused a lot of the algorithms I created for version 1 but these algorithms were run amongst different processes. This splits the work-load up just about evenly per process. Each process groups the files according to the process count, reads the files, sorts the contents it has, then writes these contents to a temporary text file. From there, main reads this text file and puts the contents into a master list of the contents and sorts these contents, eventually writing them to the output file (“Version 2/Output/output.txt”). Unfortunately, since most of the code is reused from version 1, it has just about the same short comings.

For version 3, much of the code from version 2 is reused. Unfortunately, it has the same discrepancies as versions 1 and 2 but it uses multiple files to execute the same code as those versions. Using the execvp() function, I split the work up amongst 3 files. The main for version 3, helper.c (essentially the distributor), and helper2.c (the processor). Once I figured out how to call the exec function, the coding really was quite simple. The only difference was, instead of using arrays to pass the data, I used text files that were in turn read by the child processes. These files are arbitrarily named by the process number (1.txt, 2.txt, etc.). Once the distributing and processing is done, the processes write back to another temp.txt file which is sorted by main and then written to a final Output/output.txt file.

The script that was written for this assignment is rather simple. It reads the first character of each line of the data files and keeps track of the largest process index it finds. This way, it knows how many processes to created when calling the version that it does. The line with the call to the executable could be changed to v2 or v3, depending on which version you want to use. Since the C program itself handles the output file, it has no need to specify the output.