Korakot Santiudommongkol

CSC 412 Assignment 5 Report

Bash Script

I check the number of arguments to make sure that the right amount of argument is passed.Then I check if the data directory exists and if it isn't empty. If it doesn't exist or/and is empty then exit the script. I then check for if the data directory path includes a '/' at the end if it does remove it. I remove it because I did the **Extra Credit portion** where I made a directory with fixed at the end of the directory name and check and add an index until reached a directory that does not exist. Next I create variable n which will represent the max index from all of the files in the data direct. I traverse through the data directory checking the index of the file getting it by using grep and comparing the index to variable n. To make sure all of the files had the unix LF character I check for files using the CRLF format by using "cat -v" command which gets the line of the text file and then visualize the CRLF in form of ^M, I then write into the new data directory using the same filename the line without the ^M which represents the CRLF character.

Version 1

Similar to the other version the main function puts all the arguments to variable, first argument is an int so used atoi to convert char* to int, next is the data directory name stored in const char* because it needs to be a const char* to use dirent to find all of the filenames inside the directory, and used a string for output file. First thing to do is check for output file to make sure it has the .c extension and add it if it does not have the extension. Next is to get the list of filenames in the data directory using the dirent library. Next I do the distributing and processing, distributing files equally to each n process. The process is really just to get the information in the processStruct which store the index, line number, and code fragments. The main function gets a 2d vector in return, turn the 2d vector into one big vector and then sort the vector using line num. And finally write only the code fragments into the outputfile.

Version 2

All data validation is similar in all versions. The first round of n child process creation is distributing the files to the next gen of children to do the processing. The distribution and process child uses a temp file #define as datalist concatenate the i value so I which each child should use. The distribution takes a peak at the index of each file and write to the temporary file path of the file whose index matches i. The process stores the information on the text file each child is responsible for in a struct which is stored into a vector, then the vector is sorted using the line number. Then overwrite the temp file used to store the file names, and overwrite with the code fragment with the correct line number order. The parent will wait for gen 1 children to finish first before starting processing. The parent will wait for each process to finish one at the time and then will read the same temp file and adding the content of each file into the output file.

Version 3

Same as Version 2 except for using execvp running 2 different cpp program distributor.cpp and processor.cpp for distributing and processing. I used execvp over execlp because the number of files to passed to the function is not a fixed number, so there will be a different number of arguments each time, so it is easier to create a char* array to hold all of the arguments.

Version 4

The only difference from the previous version is the way my distributor and processor work. I use n times to create n child process which will become the distributor. That distributor will look at all the file index and add the file name to a vector for each corresponding index equals to child i. Then the distributor will fork to create the processor, the child of the distributor is the processor. The processor takes the vector of file names and read each of the file and store it into a vector of struct, sort the vector using line number. Then the processor will write into the file naming scheme from the previous version only the code fragment in the right order into the file. The parent the processor child process is going to wait for its child process to finish then read the file the processor created appending each line together into one big string. The distributor will be sending concatenated to the root process, but since the root does not know how big the string is going to be. I decided to send the length of the concatenated string to the root first, so the root can create a char array with enough size of the string + 1 for the terminating 0 value. Then the distributor will send the string and then the root process will write into the output file the concatenated string of that each processor created. I decided to use unnamed pipe since I choose to create only a one way connection between distributor and processor, I found that it was not necessary to create a two way connection. I choose a blocking read over a non-blocking read because I decide to send one big string to the root process rather than x amount of string.

Version 5

Same as Version 4 except pipes between distributor and processor also. I used the same pipe method between distributor and root process. For the distributor and processor, the processor sends messages to the distributor of the code fragments in the right order one line at a time, and the distributor would do a non blocking read for each line sent and append it to a big string with “/n” each time since each line passed is its only individual line in the file. The distributor will take the completed string and sends it to the root to let it write it in the output file.

Limitation

For all version the larger the amount of data inputted, the slower the program is. For **Version5** in particular The non blocking read was most troublesome for me since I wanted to get the right amount of time I need to wait in between each write and each non-blocking read. If the usleep time was too little the non-blocking read will miss some messages sent to it. I used usleep over sleep because waiting even one second in between each write or read is way to long if given a large dataset. For **Version1** I made the assumption that the file should be split up evenly between each process rather than split it by index so the sorting of the line number comes at the root instead of the processor.