CSC 412 - Programming Assignment 05, Fall 2019

Jordan Coats

***Abstract*— The objective of this assignment is to use fork(), create a server-client architecture, blocking/non-blocking system calls, Doxygen.**

I. INTRODUCTION

This was a really interesting project to create and very educational. Throughout this process I have rewritten every version of the program as I learn more from the sequential versions.

II. DATASTRUCT

To start with I would like discuss the datastruct.h file I created for this project as it is used in every version. As per advice from Chris, instead of worrying about keeping track of each file's data I combined it all into a simple struct that reads and catalogs every big from each text file to keep it very simple. At first I wasn't going to include the filename but it did prove to be useful in the later versions.

III. VERSION 1

This was the simplest version of the process and is pretty straight forward. This is where I came up with the functions that defined the rest of the version such as readData, distributor, processor, and SplitVector. The function filenamePath was created by Herve. To keep the temp files in order and out of the way my program creates it's own directory and then removes it at the end. If you'd like to see the created files please comment out this line.

```
nftw("tempV*",
rmFiles,
10,
FTW_DEPTH | FTW_MOUNT | FTW_PHYS);
```

The nftw and rmFiles function wasn't a part of the overall design but did make cleaning the files up afterwards very simple, it came from a StackOverflow posting explaining the process of removing directories. The min and SplitVector functions were pulled from 212 assignments and either heavily modified or left alone.

IV. VERSION 2

The biggest change between V1 and V2 is the introduction of child processes by using fork(). Other than the fact that it is in a child process the code is virtually identical for distributor. The additional code handles the PID and waiting, the first loop creates each child then a second loop after makes sure every child has finished before moving on to the processor step. Processor is also virtually identical in the same regard. The wait for process just tells the parent that the output text files it creates are ready to be combined.

V. VERSION 3

This version adds execvp() calls to launch a compiled Distributor and Processor program instead of using just a child process alone. The biggest change is no longer using a vector to pass to distributor for each process but the parent actually creates a text file that distributor can read from that has the exact same format as the vector would have. An issue I had with this version was going from a string to a char*. Prof Herve helped me write code to accomplish that:

```
std::string k_s = std::to_string(k);
std::string n_s = std::to_string(n);

char *args[] = {"./distributor",
(char *)(k_s.c_str()),
(char *)(n_s.c_str()),
NULL};

int status = execvp(args[0], args);

printf("STATUS CODE=%d\n", status);
```

With that it handles itself very close to V2 as both the distributor and processor are again almost identical.

VI. BASH SCRIPT

The bash script is expected to be in the same folder as the compiled prog05 program but otherwise it takes 2 args which are the input data directory and an output filename. It can be in a folder but it must be made before, as the bash script doesn't create any folders. Then it goes through each text file looking for CR, if it finds them it will replace them with a "/n" character. Then it goes through and reads just the first character and compares it to it's counter, if that process index exceeds it will replace the counter. Then at the end since we start at 0 it will add one, because we start from 0.

VII. CONCLUSION

Unfortunately I was not able to finish the V4 and V5 with the pipes. My idea was to implement:

```
int pipe[procNum * 2][2];
```

As that would create two pipes (read/write) for each process that we would need. Then it's multiplied by 2 as there would be the distributor and processor that need them. From there the distributor would instead create the a processor that uses the the index to know which lot of files it would sort. Once all the distributors are finished they talk to the parent to give the completion signal. From there the parent sends a signal to the first distributor to begin its processor. When the first

processor finishes it will tell the parent distributor. When the parent distributor knows that it will relay it to the parent to let the n+1 processor know it can start. From there it would continue until they all give the signal and the parent knows it can combine all the output files back into the code.