When I started with the program, I aimed for it to be scalable from the start. I found this easier than trying to start with a set number of processes and then making it scalable from there. I designed my program to run with up to 10 filenames, but it could be changed slightly to make it work with more. When the program starts, it creates 2 pipes, one that will be used to communicate the keyword to the children, and one that will be used to communicate the results of the search from the children to the parent. It sets up a handle for handling an interrupt (CTRL+c), as well as setting up a fd\_set to store the file descriptors that will be watched for data.

The parent then runs a loop up to 10 times asking for a filename input from the user. If the filename was ‘$$$’ it exits the loop. When the user enters a filename, the parent forks a child. The child then saves the filename to its own memory space. It also sets up the pipes properly so that it can read in a keyword from the parent, write results to the parent as well as print to the screen. It then breaks out of the loop, so it doesn’t create its own forked processes.

Once all the files have been entered, the parent makes sure that at least one file was entered, and then finalizes its connections to the pipes. It then enters an infinite loop, constantly checking for input from either the user (a keyword) or from the child (results of a search). If the results come in as a negative number, then that was a flag that says the file couldn’t be opened, so the parent reduces the number of children count by one (because the child is going to kill itself). If the results don’t contain a negative number, it prints out the results and checks if all the children have replied. If all the children have replied, its gives a total results value.

The child does something like the parent, except it sits in an infinite loop waiting to receive a keyword from the parent. Once it receives a keyword, its searches through the file it was given earlier and send the results to the parent inside of a struct. The struct was a nice way to store the child’s id, number of occurrences, filename, and keyword in a bundle to send over the pipe. This was the parent could just read a ‘results’ struct size of information into a struct and not have to worry about mixing up which info is from which child. If the results returned a negative number, then the child would still send it to the parent, so the parent is aware it couldn’t open the file, and then the child kills itself and closes the appropriate file descriptors.

If the user enters a CTRL+c, both the parent and the children have the signal handle to the interruptHandler function and they both will close the appropriate pipes and then exit.

Nathan Kelderman

Professor Wolffe

CIS 457 Multi-process Text Analysis