CSC 412 - Prog05 Report

Ryan Brooks

This assignment challenged me managing multiple processes and forking with processor communication via pipes more complex than what we've seen in the lab. Throughout this report, I will discuss my process for implementing the assignment based on how I set up the project and how my design changed through the 5 versions as well as the difficulties encountered. Furthermore, I'll discuss how pipes were used throughout the program as well as the limitations.

Implementation Details & Difficulties

Version 1

Implementing version 1 was very straightforward. I used an object-oriented approach of a Server and Client architecture. The server knows how many clients are connected and via calls in the main function, the server handles the steps of the process while the Clients represent the distributors. They store all the files that belong to them at a given time and process them. The server first distributes subsets of files evenly to the clients. Then it calls another function to handle verifying which files belong to them in conjunction with methods of the Client class. Finally, the server calls processing to happen on the data files, which is executed by the clients and returns the reconstructed code blocks to the server, which are then written to a file. I also used a minorly improved version of my testing macros I created last assignment for debugging. Folder structure is explained in the README file.

Version 2

Version 2 exposed me to the first forking part of the assignment. Since all the architecture was pretty tight with interacting between the Server and Clients, I had to do some reorganization to allow writing and reading to files. The clients write to the files while the server reads and processes the data. All major sections/operations of the program are still executed through the Server. Major updates were to the verifyDataFilesDistribution and processDataFiles functions.

Before I handled it with a matrix for all verified files and since it was linear I could just update whatever row in the verified files matrix with the new file and at the end update the files list for all the clients with the ones in the verified matrix. I scrapped all of that and made the server functions that verified and processed data files fork and call child functions. These child functions would call methods of the Client class. These methods would handle the writing, terminate their processes and the server would facilitate reading the files and initializing each part of the process like normal. These were the main changes for this version alongside better usage of the main files array to pass by reference and have the child function use file indices wherever possible.

Version 3

Version 3 required more restructuring of the project than version 2 due to implementing the exec functions, which conflicted a bit with how data was sent from server to client. Previously the server would create the clients, various functions would be called in main, and eventually the server's method for distributing the data files would be called and a child function would be forked for each client off of that. Then it would run the client's method to handle that. Same for data processing. I had to modify that so the server's function forked and then called exec instead of calling a method on each client to be run by the child process; it would call exec on a program to handle distributing the data–still handled by each child. Then that program would

need arguments, creates a child function, and calls the child's method to handle the data distribution like before. Exact same idea for the data processing setup. So it amounted to adding

The main modifications happened in those two functions. Consequently, I had to change the way files were passed around in other parts of the program to support data being written to files from different sources--now the exec programs. One example of this is in the readDistributorTempFiles function. I needed specific functions for reading the data because it wasn't just a list of Clients being passed around; in the processor and distributor programs called by exec, I needed to write as arguments the subset of files they need to distribute, then the program recreates their own client for processing.

It was slightly tricky getting the arguments to be passed to the exec functions right (and then make sure those outputted the data correctly like in version 2) to finally have the server read back the data.

Initially I didn't like this because it added unneeded complexity and runtime bloat to the program with constructing the arguments, especially for the subset of files needed to be run by each client. However, Professor Hervé explained to me that exec is a good idea when there's already an existing architecture and it would be easy to just call exec and pass some data instead of integrating it in the system. This problem is very simple and already planned for the grand scope of what's happening so it feels a bit awkward to implement. This improved in later versions a lot.

The main limitation of this is that there's a limit to the amount of arguments based. If there's too few distributors then there would be too many files passed as arguments. The only solution I could think of was having the distributors themselves read the files, but since I wasn't running into this problem with the provided data sets, I chose not to because I knew it would take a hit to

the speed of the program. The other solution would be passing–somehow–file indices instead of full paths. This is better than the previous solution, but, again with more time I'd work out a better solution.

Version 4

For version 4, I started off by breaking the server's verifyDataFilesDistribution function into 2. 1 to handle the forking and another to prepare all the arguments for the exec call. This is because I had to set up pipes in this function between the server and all its clients, both reading and writing ones pipes. That's main purpose of this function now. Besides adding pipes to that function, I modified the distributor program, called by exec, to know when all its siblings have completed since now the distributor program would initialize the processor program, not the server. I was able to use the pipes to allow the distributor processes launched in the exec program to know when its siblings had finished to eventually call its grandchild process for data processing. I explain in the section below my choices for the types of pipes, architecture, and reading blocking.

I needed pipes since the server was no longer handling the next step in the program, the distributor program, just one of n clients, needed to know when its n siblings had all finished so it could move onto data processing. I kept track of the number of siblings complete by having the distributor program write to the server via pipes when it was done writing to the temporary files. The server would wait until all clients had signaled through reading pipes, and then write to the clients, signaling that they can continue, and closing the writing pipes.

The next step involved modifying the client class and distributor program further to call the processDataFiles function not in the server but in the client. This also included other functions in the server, so they all had to be moved to the client class now. The new process involves the

distributor program reading all the data files produced by it (now that all distributor clients are done), and it can then update the clients with the files it found as belonging to that client. Only minor changes were required to this function after moving it to the client's class. After that, I knew I needed to replicate a similar structure to verifying the data file's distribution like before.

Like normal, it would call client functions to handle this, so the old processDataFiles in the client that was called by the main server one is now the main one called by the distributor program as a method of the client class. It's renamed to initializeProcessor to now provide the main forking and launching of the processor process. Similarly, the old verifyDataFilesDistribution function the server was renamed to initializeDistributor for the same reason. The processor function would be called from the distributor function and launch the grandchild (processor) process. That program didn't require any changes.

Overall, I needed to do a lot of restructuring to get the pipes to work so the distributor processes could communicate and also to move a few functions that were handled by the server to the client, but the program now has two main interactions from main, to launching the server, whcih branches off of initializeDistributor, which then calls initializeProcessor. Nothing needed to change in the client's processData function because it just writes to files. That part didn't need pipes in this version.

The final main component of this version was getting the grandchild to notify the children when complete. The child can read the data files, and somehow the server needs to get that data. Before I simply had the server reading the data files once the children completed, so I knew I once again had to use pipes to write the reconstructed code blocks to the server. Once again some functions had to be moved around, but long, long story short, I was able to reuse the same pipes that were initially created by the server in the very first function. The distributor

program calls a function to do all the data processing and eventually we have the code blocks reconstructed from the files read. Now we can call the same pipes already sent as arguments to the distributor program and write to the server. I just had to change the spots in the program where I had closed the pipes to do that later.

Now the server's main initializeDistributor function calls a function to handle the distributor exec call, another to handle coordinating the distributor processes, and a final one to gather the results from the pipes, close them, and report back to the server.

Version 5

For Version 5, it was finally time to fully utilize pipes. This ended up being a bit harder than expected because of dealing with asynchronous issues and using pipes. I had to work out exactly what was going wrong, why the pipes weren't working when they were working in unit testing prior to integration testing. I figured it had to be an issue related to blocking reads or something like that because there were two points in the program where the server and distributor signal each other to keep going so data can be fully processed before moving onto the next step.

I eventually redid some of how the distributor process works so signaling to the server is more implied than in version 4. I think it was waiting for signals that didn’t always need to be there and could be solved just by adding a character to the end of the pipe. Now that's at the end of the field and client indices it sends to the server, built into the data it pipes since all the functions that relied on writing and reading to files are gone.

Consequently, I was able to remove both the helper functions in the distributor program to streamline the process of interacting with the server. Now it's all done after it pipes data to the

server, including the file paths and their corresponding client indices to the server, it adds the done signal, which is just a blank. Then because of the blocking read call, it will wait while the server processes all the incoming files that need to be sent around to other clients. It still won't complete until all clients have sent data back to the server, like version 4. Then it sends all of the files it found belonging to each client to each distributor process. That's handled by new functions in the server, all called under initlaizeDistributor function. The distributor program itself has a new function to handle all the pipes to keep that out of the Client class. It simply reads the file paths back from the server in identical means as the server reads the data that the distributor sent it (minus the client index since it's only going 1 place this time).

Back in the server, this process is handled by 2 new functions. The await distributors function has been updated to better integrate with pipes and there's a new function to send the data to the distributors. The new await function handles all the reading from pipes that was sent by the distributors it reads until it finds the 0-character, indicating the end and also indicating that the client has finished verifying everything. The function creates a vec of vecs of file paths in client index order and the redistributor function takes that vector and uses the pipes once again to send them back to the correct clients. This was pretty tricky to put together even though I had most of the pipe work in place from version 4, simply due me finding it tricky to manage working with quite a lot of pipes and having blocking issues.

I also wanted to mention that I did do 2*n pipes over just 2 pipes. Of course it's 2*n because they're unary so each distributor has a reading end pipe and a writing end pipe, so each has 2. While this solves the issues of pipe atomicity, it's probably pretty memory intensive and might be slower just due to the filesystem. I tried to minimize writing to more files than needed, opting for longer to parse a file but less files to parse. I think if given more time for the assignment I'd restructure some of the pipes, to reduce memory costs, and definitely addressing the issue of

pipe capacity and better scheduling sending data through the pipes. I think I would quite easily hit the 4kb limit of a pipe if given a slightly larger dataset than the test assignment ones.

Regardless, now I was in a place where the files were distributed, so I could launch the processor process. I thought this was going to be more work due to no pipes being used in this process yet, but I realized I could use the same pipes I'd been using the whole time since once the processor process. I ended up being able to remove a couple functions, too. First things first, I modified the processor program to take in the write end of the pipe as arguments. The distributor program calls the processor program, so this was straightforward. Then I modified the existing function that wrote the processed data to a tmp file to write to the pipe's Fd. This was the last modification because I already had a function to handle reading the combined code blocks that was called by the server due to version 4.

Interprocess Communication

This section details my choice of process communication throughout the program. This will be based off the Version 5 since it’s my final integration of pipes (includes notes from the extra credit version as well).

All of the pipes throughout the program I decided were to be created when it initializes the distributor process as that function’s the main server control for what it needs to facilitate for the entire program. I used unnamed pipes for this since all processes were related in this hierarchy of server → distributors → server → distributors → processors → distributors → server. Each client on the server gets 2 pipes. These are stored in vectors of child to parent pipes and parent

to child pipes. The unused ends are then closed. When this function forks and calls the distributor exec program, it passes that client's read and write file descriptors.

The child processes start running and verifying which files belong to themselves not. Then they send the wrong ones back to the server over that client's write pipe. During this, the server has been listening with a blocking read pipe until the client sends an empty byte signal at the end, indicating they're done. This is all done through the distributor process. There's another server function the parent process calls to handle this. It goes through all clients and listens at that client's read pipe and parses the data sent. Only when all processes have indicated they're done verifying their files does the server call another function to redistribute the files to the clients. In a very similar fashion, this function sends the file paths to each client using its parent to child writing pipe. This process also sends the empty byte signal to indicate all file paths have been sent. So the distributor process during this has been using a blocking read call to wait for the server. Finally, all files have been distributed and the distributor process can move onto processing.

The data processing component is pretty simple since it doesn't require any parent to child signaling or vice-versa. The distributor process runs the exec call to launch the processor process. It passes the same writing pipe file descriptor that the distributor process had, as this is now a grandchild of the distributor child. The processor process passes that child write file descriptor to a method of the Client class to process the data files. It simply writes the lines of code it sorted and reconstructed to the pipe and the processor process terminates. Consequently, the distributor process can also terminate. Now we're back in the server process. Again, this was a lot simpler because the act of the child processes termination allows the server process to continue (it's just waiting right now), so it can run the function that retrieves what was sent over the pipe for each Client, and then outputs the results to a file.

As I described in the first section, I did struggle a bit with implementing the pipes for version 5. I think this was due to the asynchronous nature of working with different levels of child processes, across exec calls, and trying to go back and forth between the server and children. The main problems I saw with this is a file path or a line of code can get very long and if there's not that many distributors working at the same time, I saw the data being sent easily being able to exceed pipe capacity or atomicity. I think I was pretty good about handling atomicity because while memory wise there are a lot of pipes so perhaps that slows things down and will become costly if there's a lot of distributors running at the same time. Otherwise, I didn't have to worry about data sent through the pipe being spliced with others writing at the same time, which reduced the potential problems to only being a synchronization problem with going back and forth between the server.

When implementing the extra credit, I created a communications file that had a function for reading and writing to pipes. I set an arbitrary, small limit on the pipe to limit capacity to mimic a real application, where there would be enough data to exceed pipe capacity. Then I went through everywhere I used the normal write and read functions and replaced it with my own–two small functions that include standardized error checking. They expect a message as a string. They'll compute the message's size and send that through the pipe along with the 0 byte to signify the end.