The purpose for this project was to expand my current knowledge of IPC and how to achieve communication between two processes. Seeing as how this is an Operating Systems course, finding out and understanding how to develop a program that is able to spawn and connect to another program is monumental. Not only that, but I also gained techniques in implementing certain facets of operating systems such as timers, stacks, and interrupts. I focused on building these fundamental concepts to further give insight onto the inner workings of an operating system and how I can create personal projects that are far more complex than anything I have made thus far. Achieving a system that is able to fulfill the requirements that the CPU and memory modules needs reinforces my knowledge as a programmer in my personal ability to construct complex systems. So not only was it important for me to get a better grasp of how Operating Systems work, but it is also imperative in a pedagogical sense that as programmers, I now have better confidence in my skills to complete projects that seem far greater than my current ability to comprehend.

I chose to implement this project with Java as my choice of language. To be honest, I only chose Java because I felt my C skills were not up to the test with this project and that it has been years since I’ve dabbled with C++. I also have the most confidence in my skills with Java as I feel it is far more intuitive to program with than the other two I mentioned. Starting off, I utilize the Runtime and Process APIs to create a new process which will serve as the memory and pass the file which was entered in from command line. Afterwards, I create input and output pipes along with PrintWriter and Scanner objects as a means of communicating between the two processes. I then instantiate my registers with all zeroes except the SP which starts at 1000, timer which is assigned the value of the second argument passed from command line, and a flag initialized as false indicating whether the CPU is in kernel mode. The CPU then enters a while loop, which waits to fetch an instruction from the memory process. For fetching, the CPU enters a method which checks if the CPU is in user mode and if it is, checks the index which it is trying to fetch and should the index be greater than 999, it prints an error statement and terminates itself and the memory program. On the flipside, when the memory is first spawned, it begins to process the file passed from the CPU program by parsing each line and extracting only the instructions along with the index that it should reside in memory. Then the memory program enters a while loop which waits for a fetch command from CPU. Once it receives the fetch command, it reads the index passed on and returns the data from the integer array. The CPU receives the data and stores it in the IR register where it becomes decoded in a switch statement. The program enters the corresponding case and begins the execution phase of the CPU. There are many cases in the switch statement ranging from simple increments, fetching and assigning to the AC register, simple arithmetic, and other instructions such as printing out the AC as an integer or character. Certain instructions require writing into memory and the way I implemented that is by doing the same initial steps when reading which is to check if the user is violating system memory, then passing the data and index to the memory program which then detects that it has been passed two integers instead of one and then writes it into the memory array. One of the cases involves pushing a value onto the stack register which first decrements the SP then stores the value in memory at whatever index the SP is at. The case involved in popping the stack does the reverse and resets the value to zero wherever the SP is currently at in the memory index and then increments the SP. Certain instructions involves branching to different addresses in memory and is achieved by first pushing the PC onto the stack and then setting the PC to whatever the next value is in the memory array from where the PC is currently at. There is even a special pop instruction which pops the value at the top of the stack and sets the PC to that value. The final and critical aspect in the execution phase is if the instruction being executed is a system call which then pushes the SP and PC on the top of the stack, sets the PC to the next value in the memory array, and changes the flag register to true indicating that the CPU is currently in kernel mode. It executes the set of instructions pertaining to the system memory and once it finally reads a certain instruction letting the CPU know it is finished executing the system code, it then pops the values pushed form the initial values of PC and SP and resets them to where they previously were. The program sets the flag to false indicating that the CPU is in user mode and then continues executing the user program input file. After the CPU is done executing the instruction, it increments the PC register so that it may fetch the next instruction. Not only that, but it also decrements the timer and sees if it has which zero. If it has, then it does the same procedures as the system call. The CPU pushes the PC and SP values to the stack and changing their values to the system memory, changes the flag to true, and executes the system code until it reaches the end where it does the reverse process so that it may continue working on the user program.

Going into this project, I had lacked the necessary skills needed prior to this class pertaining to IPC. After taking Systems Programming in UNIX Environment, I felt the knowledge gained afterwards on processes were slightly lackluster. So, in an attempt to come back and understand these concepts, I had to really prepare myself and study the techniques needed to get a good understanding of IPC. The project really started off really rocky as I couldn’t figure out that the memory essentially waited for input to be sent to it which goes the same for the CPU program. The problem I was encountering though was that I didn’t realize I could have the CPU send the fetch instruction, had the memory return the data, then execute like normal while the memory simply waited for another read or write instruction. It was essentially playing ping pong between the two programs. I think that if I had known that Java does blocking I/O that maybe I would have come to the conclusion faster, but I was really just messing around for a good minute trying to find out myself. After figuring that part out, I had gotten IPC under my belt and began the nits and grits of the project. I had trouble implementing some of the instructions, some were silly mistakes and others were simply because I had the wrong idea such as the stack. At first, I had the SP start at 999 and my push method would write the value then decrement. The problem here was that if the CPU were to execute the instruction that copied the value of the SP into the AC, it would pass a 0 since the value pushed would be at the index right above the SP. I had to really draw a picture on my iPad to figure out this major bug which allowed my program to successfully run sample3.txt. The timers weren’t that big of a hassle as I thought which made me relieved. Overall, I gained a lot of valuable experience with this project and has made me confident that I am able to achieve things greater than I can imagine.