Kevin Chen
nkc160130
Due: 28 September 2019
SE 4348
Prof. Ozbirn

Project 1

In the first project for the Operating Systems class, we explored using multiple processes and IPC. Though we're not actually writing directly to the CPU or main memory of our computers, we use Java/C/C++ to emulate a CPU, with addresses such as PC, AC, X and Y, as well as all the execution logic. The main memory, meanwhile, only has two operations: read and write. The main memory is where both user instructions, system instructions, and user stack and system stack are stored. Also, the system is to implement interrupts; that is, if there is a system call or a set number of instructions is reached, the system should switch to kernel mode, by saving the PC and AC of the user program and switching both to its system locations. Interrupts cannot be nested within each other, and in this project system calls will not ever call an interrupt and vice versa.

To implement this project, I utilized C, because I felt that that was the programming language most appropriate, given that the CPU and memory are working on a very low level and C is about as low-level as you get without directly coding with Assembly. To start, I checked to see if the user has input a file and a timer value. If not, the program immediately exits. Then, I initialized two pipes for communication between the CPU and memory, and vice versa. Next, I forked the process into a parent, the CPU, and a child, the memory, to implement multiple processes. For the memory, I initialized a 2000-size array, read the file provided by the command line input line by line, skipping the comments and spaces, and then read it into the array. The memory then goes into an infinite loop to listen for read and write requests from the pipe. The read request is implemented by reading in a string, such as "r100", and writing to the pipe the value at location 100. The write request is implemented by reading in a string, such as "w100", and reading in another value to set the array at index 100 equal to. The CPU, meanwhile, after initializing buffers and registers, immediately goes into an infinite loop as well, and houses all the logic in if statements. It first fetches the PC instruction from memory, and then checks for a timer interrupt. If the timer interrupt is due, or if there is a timer interrupt waiting, it executes it. The load, store, pop, push, etc. commands are all implemented by performing a combination of read and write commands sent through the pipe.

Kevin Chen
nkc160130
Due: 28 September 2019
SE 4348
Prof. Ozbirn

I learned a lot from doing this project. It took many grueling hours of trying to figure out what was in the stack, creating debug outputs and setting watches for different variables, etc. and watching the program hang, with no output, with no obvious reason why. First and foremost, I learned that dealing with strings in C is a huge pain, and I would personally have preferred to use C++, especially because strings are much easier to handle. This project also reinforced and expanded my knowledge of pipes and forking processes that I was introduced to in SE 3377 (C/C++ Programming in a Unix Environment). Through coding the interrupts and interrupt handlers, I learned just how complex interrupt handing processing can be, even in a simple form that this project had. I also got a lot better at debugging my code, using watches to monitor the values of variables and verify to see if it was as it should have been. But it was cool to see that I had created a basic, albeit somewhat powerful, processor out of various C variables, arrays, etc. It was eye-opening as to just how complex real-world, modern processors can be, and all the hard work that must be done for the processor to function correctly.