Project 1 Summary Zach Tang CS 4348.006

1. Project Purpose

The overall purpose of this project is to understand how multiple processes communicate through IPC (e.g. UNIX pipes) and understand important low-level concepts that will show up throughout the OS course. The project achieved this by having me implement a simple emulation of a system consisting of a CPU and memory. Many different low-level concepts from Computer Architecture were reviewed in this project, such as instruction set architecture, interrupts, timers, memory interfacing, stack maintenance, function calls, and registers.

2. How the project was implemented

I implemented the project using C++. I divided the CPU and Memory into 2 different classes Additionally, I defined the timer as a class inside the CPU as it was internal to the CPU. I ran the CPU and Memory on different processes and configured IPC in my main.cpp file to allow them to communicate.

The memory class is called MMU and this class handles parsing and loading the given program into a memory array. It also handles public requests for reads and writes to and from this memory array. Lastly, I created a function for dumping the memory array contents to help during debugging.

The CPU class handles implementing the fetch decode execute cycle. There are 3 parts to the fetch decode execute cycle: 1. Timer Ticks, 2. Fetching and Decoding Instruction, 3. Interrupt Handling.

For timer ticks, I designed an internal timer class in the CPU class that has a tick function that interrupts the CPU every X instructions. I instantiate this timer class as a member of the CPU class so I can tick it at the beginning of every fetch decode execute function run.

For fetching and decoding instructions, I created an individual function for each instruction of the instruction set. To map each instruction code to its corresponding member function, I created a Decode(address) helper function that returns a function pointer to the corresponding member function given an instruction code. I have MMU interfacing functions to read and write from and to memory. I use these to fetch the next instruction code. I then take this instruction code and get its corresponding member function using Decode(address). Finally, I run the member function which does the necessary tasks defined by the instruction set.

For interrupt handling, I have 3 flags: 1. Interrupt Enabled, 2. Interrupt Flag, 3. Timer Flag. The interrupt enabled flag determines if I'm allowed to handle interrupts in the current cycle (e.g. the CPU does not allow handling interrupts when in kernel mode). The interrupt flag determines if the Int() system call was made. The timer flag determines if the timer sent an interrupt. I use the interrupt flag and timer flag to determine the address (1000 or 1500) I will call to handle the interrupt.

The MMU and CPU run on different processes. I used the UNIX fork() call to create a parent and child process. I configured IPC via UNIX pipes to allow the MMU and CPU to communicate with each other. Whenever the CPU needs to read or write something from or to memory, it sends a request into the MMU's pipe file descriptor consisting of the necessary information for the MMU to read or write (e.g. address and value). The MMU then communicated back to the CPU by writing to the CPU's pipe file descriptor (e.g. writes the value of the integer read from memory).

3. Personal experience doing the project

This project provided a good challenge for me. I started out designing the system to organize my approach, which made it a lot more straightforward for me to do. I had to brush up on forking and piping and this project made me realize that I'm still uncomfortable with implementing that, so I should practice IPC more to get more familiar. Otherwise, I'm proud of the way I approached this project. Rather than get straight to coding, I outlined my design and created a clear plan in my head for steps to complete the project. This made the process much smoother for me and I was able to complete the project in a timely manner. The project itself was interesting. I imagine it's very similar to what virtualization tools like VMWare need to implement when virtualizing different systems. I'm going to explore this topic more in my free time.