Summary Anqi Chen

Project purpose

In this project, we learned how two processes can be created and executed at the same time and how these two processes can communicate with each other. One of the processes is used to simulate memory which stores program instructions and data for user and system. The other process is used to simulate CPU which can execute a series of instructions. This project includes some low-level concepts that are important to an operating system. After finishing this project, we can get a better understanding of how CPU interacts with main memory, namely how CPU can read data or instruction from memory and also write data back into memory. Also we get a better understanding of how instructions are being fetched and decoded, and how several instructions can be jumped over. In addition, we get to learn how procedure call is being performed, how the stack pointer is going to change during the process of procedure call and also the importance of saving registers into stack before jumping into the procedure address. And most importantly, the project helps us better understand the differences of executing instructions in user mode and kernel mode.

How the project was implemented

The two processes for simulating main memory and CPU respectively were first created using folk().

The program code is then loaded into the main memory and being fetched by CPU one by one. Since these are two separate processes, the only way to let them communicate with each other is through pipe. The CPU will decode each instruction and decide what type of instruction it is going to execute. The simplest type of instruction does not include any interaction with main memory. For example, the load value instruction or the addX instruction. This type of instructions only updates the value in the registers. In addition, the jump address instruction only updates the PC value, which does not include interactions with main memory either. For other instruction like load address or store address, it is required to either read from memory or write to memory, which is performed using pipe. Since CPU needs to send data to memory as well as receive data from memory, and vice versa, we need to create two pipes. One is for CPU to send data to memory and memory to receive data from CPU, the other one is for CPU to receive data from memory and memory to send data to CPU. Other instructions like Call address or Ret are more complicated, since it includes interactions with memory as well as pushing registers into stack, which means we need to change the stack pointer as well. Here it is important to save PC registers into stack since later on when we return from procedure call, we have to know which address we will return to. Note that other registers like AC should also be pushed into stack since they may be used in procedure calls so their value may be changed.

When we switch from user mode to kernel mode, we have to push both PC and SP register into system stack since when system call returns, these two registers need to be restored. There are some differences executing instruction in user mode or kernel mode. First, when in user mode, we do not have access to system memory, whereas in system mode, we have full access to the main memory. Also when in user mode we can do system call either through the Int instruction or the timer, but in kernel mode, the system call is disabled.

Finally, when End instruction is executed, the process simulating CPU will send a signal to the process simulating memory through pipe, telling the memory that the program has reached to the end so that the process should exit as well.

Personal experience in doing the project

I had no experience of writing program with two parallel processes, so it looks difficult for me at the beginning. After I learned how the fork(), pipe(), read() and write() instructions work, I got some basic ideas of how this project should be implemented. My first step is to make my two processes able to communicate with each other. After that, I was able to write the code for the first couple of instruction set including load value, load addr, and so on. Other instructions like AddX, AddY were pretty easy once I figured out how to make the two processes communicate through pipe. What takes me a long time was the instruction that has to do with procedure calls. In this case, if the SP points to the wrong memory location, you will never get the right results since the procedure call cannot return to the right address. It also takes me quite a long time to figure out how to set the PC values correctly. It seems to be very simple at the beginning: just increment it by 1 after executing each instruction. However, soon I found that for instruction that has an operand, PC should be incremented by 2. In general, I got a lot of practice using pipe in this project and had a better idea of how operating system works.