CS 4348 CPU/Memory Process Project Summary - Ethan Fischer

Project Purpose

This project simulates a barebones operating system consisting of a CPU that can read from and write to a memory array, and that's it. It's very fundamental, but not simple. The CPU and memory are separate processes that must communicate using pipes because the CPU doesn't have direct access to the memory. The CPU sends requests to the memory through a pipe and the memory process responds by writing to the memory array or sending the CPU a value from memory. The CPU fetches instructions until there is an error or an instruction in the program tells it to end execution. The objective of building this system was to develop an understanding of how multiple processes communicate and cooperate and important OS concepts such as processor and memory interaction, processor instruction behavior, the role of registers, stack processing, procedure calls, system calls and interrupt handling, memory protection, and I/O.

Project Implementation

I chose to implement my project using C, and with that comes fork() and pipe(). To split the program execution into two separate processes, I used the fork() method which returns a process id. If that process id is 0, the child process is running. Otherwise, the parent process is running (assuming that there was no error forking, if fork() produces an error it returns -1). In order to check which process code to run, my main() function is really just one big if/else statement.

```
if(pid == 0) {
    //memory
} else {
    //cpu
}
```

To allow the memory and cpu process to communicate, I used the pipe() function to create two pipes: one for sending data from memory to the cpu, and one for sending data from the cpu to memory. Using two pipes like this allows me to ensure that there is no data corruption in the pipe due to unpredictable execution order of read and write statements in both processes.

```
int p1[2]; // memory -> CPU pipe
int p2[2]; // CPU -> memory pipe
pipe(p1);
pipe(p2);
```

The memory process first fills up the memory array by reading from a given program file. Then it waits for the CPU to send a request. A CPU request is a struct I created with 4 descriptors: op, address, value, and debugPID. The op character tells memory if the cpu is requesting to read from memory or write to memory, or if it wants memory to break out of the read loop. The address is the address that the cpu wants to read

from or write to. The value is the value that the cpu wants to write into memory if it's writing. The debugPID descriptor doesn't matter and is purely for debugging purposes.

```
struct cpuRequest {
    char op;
    int address;
    int value;
    int debugPID;
}
```

The CPU first initializes its registers, PC - program counter, SP - stack pointer, IR - instruction register, AC - accumulator, and X and Y which are user registers. Then the CPU loops until there is an error that returns an error code or an instruction signifying the end of the program. At the beginning of each loop, the cpu checks for a timer interrupt based on an argument entered at the command line for execution. Then the PC is incremented and the next instruction is fetched from memory and stored in IR. The CPU determines the instruction meaning using a switch statement and does whatever the instruction tells it to before moving on to the next.

```
int PC, SP, IR, AC, X, Y;
while(true) {
    if(interrupt)
        // handle interrupt
    PC++
    cpuRead(PC)
    //do something
}
```

My Experience

In all honestly this project, particularly doing it in C, was a nightmare. I'm not one to put a project off until the last minute, and that was true of this one as well. I started back on September 14th and tried to work on it a couple times a week. Last week I was busy with exams and didn't get as much done as I would have liked but felt I was in a good enough position that I'd be able to wrap it up on Saturday. That was not the case. Everytime I thought I'd made some progress, some new issue arose that broke everything. Running sample 2 would uncover problems that hadn't been apparent in sample 1. I kept running into a mind boggling situation where literally the only thing I would change is adding or removing a printf statement for the sake of debugging and my output would change completely, which sounds like common sense I know, "duh, you added or removed output, your output is going to change," but I'm talking about something like "made it here" and suddenly the functionality of the actual logic in the code would produce a different output.

One of the biggest issues I dealt with was simply trying to extract the value from the txt file and ignore white space and non digit characters. I was constantly getting incorrect values in memory that made no sense.

Another issue I face was my switch case statements all worked perfect, but I wasn't incrementing PC or SP correctly in one spot and it would throw everything off. Creating separate functions to push and pop from

the stack helped solve the stack problems. I decided to increment PC before reading the next instruction, but that caused some issues because at the very beginning, if PC is initialized to 0, then the CPU starts reading from address 1 and not 0. So I had to initialize PC to -1 so that at the beginning of the loop it would increment back up to 0. I also had to decrement PC everytime I set PC to a different address so that when it increments at the beginning of the loop, it reads from the correct address. There might have been a cleaner solution, but this what I found that worked and I didn't have enough time to find a better one.

Another issue I experienced but didn't have time to solve was if you choose a interrupt timer that is a power of 2 (i.e. 2, 4, 8, 16, 32....) the cpu gets stuck in an infinite loop and you'll have to use ctrl+c to quit the program. I'm pretty sure that this is because of some weird math thing with the modulus interrupt check:

```
if(instructionCount%interruptTimer == 0 && !kernelMode) {
    //interrupt
}
```

My guess is that after a certain point the interrupt handler is called every cycle and the cpu keeps switching between user and kernel mode infinitely. I'm not sure why though, because in theory, say your interruptTimer is 4, as long as instructionCount is not evenly divisible by 4, this should work. When instructionCount is 17, 17%4 is 1. instructionCount is only ever incremented once after each instruction fetch. If I had more time, I'd find a better solution for checking for an interrupt but this works with correct output from all 4 samples in most cases.

Despite the many hours of headache (seriously, I'm writing this at 9:30pm on Sunday the day after it was due and have been working on it literally all weekend) and incredible stress, I learned a lot about C and grew a lot as an engineer. This project taught me so much about C and working with multiple processes and how to debug them (using A LOT of printf statements, as you'll see in my code). I learned the true value in keeping it simple when testing and how important it can be when you're working on a large project like this.