Operating Systems

**CS4348/CS5348**

## Project #1: Exploring Multiple Processes and IPC

**Due Date: Saturday, February 27, 2016**

## I. Project Organization

You should do the following pieces to complete your project. Each piece is explained below:

* Code 60 points
* Output 30 points
* Summary 10 points

Each piece is separately graded. A missing piece will result in losing all of the points for that piece.

# Code

The actual code of your program should be in this section. It should be nicely formatted with plenty of comments. The code should be easy to read, properly indented, employ good naming standards, good structure, etc.

Attached source code.

# Output

Output will be graded by running your program on four sample programs posted on eLearning, plus one that you have written. The one you write should be different from the samples and at least as complex. Each is 6 points.

# Summary

The summary section should discuss three things: (1) project purpose, (2) how the project was implemented, and (3) your personal experience in doing the project. It should be at least one page in length. A summary that is lacking will not receive full credit.

**Project Summary**

In this project, we simulated a simple processor interacting with memory. We also demonstrated how processes exchange information with one another when implementing this project. The two processes that were executed were CPU and memory. The memory would contain all the instructions that the CPU needed to fetch and execute. We also gained more understanding of basic operating system like processor interaction with memory, instruction and register behaviors, instruction execution and interrupt handling. Overall the project very well demonstrated the concepts that we have learned in the first unit of this course, and it was good to physically put them into practice.   
 My project was done in C++, so in order for the two separate processes to execute and communicate with each other I had to use the fork and pipe calls. When the fork is called, it creates two processes, a parent process, and a child process that run at the same time. The pipe call is used to send information back and forth between the parent and the child process. The child process was the memory, and once it was called, the program file was read into a memory array of 2000, which represented my simple memory that my CPU would fetch instructions from. After memory was instantiated, the only thing that memory did was read instructions to CPU, or write to the memory array based on the information given by the CPU process. The CPU process was a little more complicated because it consisted of keeping track of all the registers needed in the system, and incrementing the program counter, so we could fetch and execute the instructions in the proper order. So with a program count, and access to reading/writing to memory the CPU was able to read instructions in order to execute them accordingly, and write data to memory as was necessary. The CPU also successfully handled interrupts and system calls. The CPU would process interrupts by entering kernel mode, and transitioning to the system stack, rather than the user stack in user mode. The CPU needed to save all the old registers to the stack before entering the interrupt, so that once the interrupt was processed, the CPU would go back to execution normally.

I do feel that I have learned a lot from this project. Before this I have not really used fork and pipes to this extent, however at the end of this I feel much more comfortable running processes and communicating between them using pipes. I also have a better understanding of the basics of how a processor and memory interact with each other. I did learn concepts like this in computer architecture but they did not stick with me, but after completing this assignment, since I had to manually code the sequence of how CPU interacts with memory myself the concept is much more ingrained in my knowledge. I have to admit I did not have a concrete understanding of jump instructions and interrupts before this assignment, but now these concepts seem almost easy to me!

## II. Project Description

**Language/Platform**

The project must be written in C, C++, or Java.

If using C or C++, you must use a Unix fork to create processes and a Unix pipe for communication.

If using Java, you must use the Runtime exec method to create processes and streams for communication.

Your project will receive no credit if not using processes or if using threads instead of processes.

All code must run successfully on our cs1.utdallas.edu server or csgrads1.utdallas.edu server.

Any other method requires instructor approval.

### Problem Overview

The project will simulate a simple computer system consisting of a CPU and Memory.

The CPU and Memory will be simulated by separate processes that communicate.

**Objectives**

1. Learn how multiple processes can communicate and cooperate.
2. Understand low-level concepts important to an operating system.

|  |  |
| --- | --- |
| 1. Processor interaction with main memory. 2. Processor instruction behavior. 3. Role of registers. 4. Stack processing. 5. Procedure calls. | 1. System calls. 2. Interrupt handling. 3. Memory protection. 4. I/O. |

**Problem Details**

CPU

It will have these registers: PC, SP, IR, AC, X, Y.

It will support the instructions shown on the next page of this document.

It will run the user program at address 0.

Instructions are fetched into the IR from memory. The operand can be fetched into a local variable.

Each instruction should be executed before the next instruction is fetched.

The user stack resides at the end of user memory and grows down toward address 0.

The system stack resides at the end of system memory and grows down toward address 0.

There is no hardware enforcement of stack size.

The program ends when the End instruction is executed. The 2 processes should end at that time.

The user program cannot access system memory (exits with error message).

Memory

It will consist of 2000 integer entries, 0-999 for the user program, 1000-1999 for system code.

It will support two operations:

read(address) - returns the value at the address

write(address, data) - writes the data to the address

Memory will initialize itself by reading a program file.

Timer

A timer will interrupt the processor/ after every X instructions, where X is a command-line parameter.

Interrupt processing

There are two forms of interrupts: the timer and a system call using the int instruction.

In both cases the CPU should enter kernel mode.

The stack pointer should be switched to the system stack. system stack 999

SP and PC registers should be saved on the system stack. (The handler may save additional registers).

A timer interrupt should cause execution at address 1000.

The int instruction should cause execution at address 1500.

Interrupts should be disabled during interrupt processing to avoid nested execution.

The iret instruction returns from an interrupt.

**Instruction set**

|  |  |
| --- | --- |
| 1 = Load value  2 = Load addr  3 = LoadInd addr    4 = LoadIdxX addr    5 = LoadIdxY addr  6 = LoadSpX  7 = Store addr  8 = Get  9 = Put port  10 = AddX  11 = AddY  12 = SubX  13 = SubY  14 = CopyToX  15 = CopyFromX  16 = CopyToY  17 = CopyFromY  18 = CopyToSp  19 = CopyFromSp  20 = Jump addr  21 = JumpIfEqual addr  22 = JumpIfNotEqual addr  23 = Call addr  24 = Ret  25 = IncX  26 = DecX  27 = Push  28 = Pop  29 = Int  30 = IRet  50 = End | Load the value into the AC  Load the value at the address into the AC  Load the value from the address found in the given address into the AC  (for example, if LoadInd 500, and 500 contains 100, then load from 100).  Load the value at (address+X) into the AC  (for example, if LoadIdxX 500, and X contains 10, then load from 510).  Load the value at (address+Y) into the AC  Load from (Sp+X) into the AC (if SP is 990, and X is 1, load from 991).  Store the value in the AC into the address  Gets a random int from 1 to 100 into the AC  If port=1, writes AC as an int to the screen  If port=2, writes AC as a char to the screen  Add the value in X to the AC  Add the value in Y to the AC  Subtract the value in X from the AC  Subtract the value in Y from the AC  Copy the value in the AC to X  Copy the value in X to the AC  Copy the value in the AC to Y  Copy the value in Y to the AC  Copy the value in AC to the SP  Copy the value in SP to the AC  Jump to the address  Jump to the address only if the value in the AC is zero  Jump to the address only if the value in the AC is not zero  Push return address onto stack, jump to the address  Pop return address from the stack, jump to the address  Increment the value in X  Decrement the value in X  Push AC onto stack  Pop from stack into AC  Perform system call  Return from system call  End execution |

**Input File Format**

Each instruction is on a separate line, with its operand (if any) on the following line.

The instruction or operand may be followed by a comment which the loader will ignore.

Anything following an integer is a comment, whether or not it begins with //.

A line may be blank in which case the loader will skip it without advancing the load address.

A line may begin by a period followed by a number which causes the loader to change the load address.

Your program should run correctly with the any valid input files.

**Sample Programs**

The input program filename and timer interrupt value should be command line arguments, for example:

java Project1 program.txt 30

Here are two sample programs for illustration purposes:

This program gets 3 random integers and sums them, then prints the result.

Note that each line only has one number.

8 // Get

14 // CopyToX

8 // Get

16 // CopyToY

8 // Get

10 // AddX

11 // AddY

9 // Put 1

1

50 // End

This program prints HI followed by a newline to the screen. To demonstrate a procedure call, the newline is printed by calling a procedure.

## 1 // Load 72=H

## 72

## 9 // Put 2

## 2

## 1 // Load 73=I

## 73

## 9 // Put 2

## 2

## 23 // Call 11

## 11

## 50 // End

## 1 // Load 10=newline

## 10

## 9 // Put 2

## 2

24 // Return

## IV. Project Guidelines

### Submitting

Submit your project on eLearning. Include in your submission the following files:

1. A Word or text document for the summary.
2. Your source files.
3. The sample5.txt file you created.
4. A “readme” file listing your files, a description of each file, and how to compile and run your project.

### Partial or Missing Submissions

It is your responsibility to upload all of the right files on time. It is recommended that you double-check the files you upload to make sure they are the right ones. Once the deadline passes, changes to the submission are not accepted without a late penalty.

### Academic Honesty

This is an individual project. All work must be your own. Comparison software will be used to compare the work of all students. Similar work will be reported to the Office of Judicial Affairs for investigation.

### Grading

The written portions will be graded subjectively based on completeness and quality. The code will be graded based on points allocated for each key part of the processing as determined by the instructor. The output will be graded based on expected results for the input programs.

### Resources

Examples were given in class and are available on eLearning. Code from these examples may be freely used in your project. The web also has many good articles on this topic. You may also find information in books on Unix or Linux programming.