HOMEWORK

## Project: Operating System Simulator

## I. Project Description

### Problem Overview

The project will simulate a simple computer system consisting of a CPU and Memory. The simulator illustrates some important low-level concepts of 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 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 instruction, 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.

The stack is switched to the system stack.

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  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  Set system mode, switch stack, push SP and PC, set new SP and PC  Restore registers, set user mode  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 //.

Any line without instruction or operand at the beginning will be skiped

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 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 the program file must contain one number per line.

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 // ReturnII. Project Guidelines

### Using GitHub

Each group creates a project on GitHub and commit code to this project. Last commit is before 23:59:00 Tuesday 8 November 2016.

### Committing

All members have to contribute to the project via committing. Each commit needs clear and detailed comments.

Commit your project according to the following steps:

1. First time when the project is created
2. Second time, with skeleton codes only (just declaration, no implementation) with comments (class header, function header, variables explanation)
3. Third time, with comments inside functions and test cases for important functions (in Excel)
4. The rest are for implementation, refactoring, debugging etc.

### Acceptance Tests

Program should run all test files without errors, produce correct output.

### Grading

* Program runs all test files (20%)
* Coding convention (20%)
* Thinking before writing (20%)
* Clean code rules (20%)
* Exception handling (10%)
* Group work (10%)