CISC 640 – Operating Systems

Dr. Gregory Simco

OS Problem Set Assignment

By:

Dunieski Otano

College of Engineering and Computing

Nova Southeastern University

June 3, 2020

**OS Problem Set**

Contents

[Introduction 3](#_Toc41967616)

[Methodology 3](#_Toc41967617)

[Scope: 4](#_Toc41967618)

[Data Design: 4](#_Toc41967619)

[Class and Object Design/Modules: 4](#_Toc41967620)

[Class Diagram 6](#_Toc41967621)

[Interface Design 6](#_Toc41967622)

# Introduction

The objective of this assignment is to gain a better understanding of the inner working of a computer central processing unit’s operations. How the internal computer architecture process opcode and one or more operands - values, memory address or registers as part of the fetch, decode and execute process cycle.

# Methodology

The approach I took after seeing the assigned OS Problem Set was to determine the programing language to use to produce the design results of the simulated virtual machine/operating system. In this case, I used C# programming language. Next, I designed a simple process to fetch, decode, and execute the process cycle.

# Scope:

This program is intended to run a SIMMAC machine and give a much deeper understanding of how operating systems work. Object-Oriented Programming is utilized to accomplish the objectives of this assignment.

After the SIMMAC programs being inputted by the user, the program outputs the results after adding, subtracting, loading in the accumulator as well as what values are stored in memory.

I believe one limitation is that I don’t have a UI where I can upload the file by clicking a button; instead, file names have to be typed so that the program can load them.

# Data Design:

This program uses Enumerations for both the process states (running, waiting, and ended) and OpCodes.

Class MemStruct utilizes mostly integers to declare the registers

Class ALU (Arithmetic Logic Unit) utilizes a list of MemStruct objects

Class PCB (Process Control Block) utilizes integers to declare StartsAt, EndsAt, processId, ProgramCounter, a State variable of Enumeration ProcessState type, a Registers variable of ALU type, and a list named Memory of MemStruct object type.

Class Program also utilizes integers (Int16) for variables procedId, instruction, and jbc as well integer (int) for variable psiar. Also, two lists are used, one named memory of type MemStruct, and the other one, a list of a list, of type PCB named blocks. They are both initialized to null. This class uses a Boolean variable named quit which is initialized to false, and list of program files named programFiles.

# Class and Object Design/Modules:

I implement a class named MemStruct which contains a method that will not dump memory slots that are unused. Also, I use the ALU (Arithmetic Logic Unit) class that contains all the registers and the different tasks that will be performed when the ALU class is instantiated in the PCB (Process Control Class). The PCB and ALU classes are serializable because I think that the state needs to be preserved when the thread is suspended.

The PCB class contains a method named ExecuteJob which, in turn, contains a while loop, which, in turn, contains a switch statement. All this process is an alternative to individual threads while experimenting with async. The HALT method is invoked to dump the registers and only memory that is relevant to the current PCB which will be in race StartsAt 🡪 EndsAt. If [this.Registers.PSIAR] == 1, it will be full memory dump. At the end of the method, all the program and memory are clear, so they don’t get dumped repeatedly. Another method named ClearProgram will also clear the program instructions.

The class Program contains the variables like processId, instruction, jbc, and psiar as well as lists of MemStruc and PBC objects such as memory and blocks. This is the class that loads the SIMMAC programs (program1.txt and program2.txt) in memory, initializes the system, runs the SIMMAC machine, executes each block collection in its own thread, iterates on Process Control Blocks, and executes each job in its own thread. The IterateBlocks method implements the join() method to prevent a race condition.

This is how the program works:

Once the Main method in the Program class is invoked upon running this project, the user will be prompted to enter a file name; once the file name is entered, the user will choose whether or not she or he wants to execute the program; if the answer is no, then another file name can be entered, else, the program will show the output.

The class Program contains the variables like processId, instruction, jbc, and psiar as well as lists of MemStruc and PBC objects such as memory and blocks. This is the class that loads the SIMMAC programs (program1.txt and program2.txt) in memory, initializes the system, runs the SIMMAC machine, executes each block collection in its own thread, iterates on Process Control Blocks, and executes each job in its own thread. The IterateBlocks method implements the join() method to prevent a race condition.

The system is initialized. If memory slots are less than 512, then it begins to add new objects of the class MemStruct which contains OpCodes and Operands; if the line count is greater than 512, then it will throw an exception (OutOfMemoryException).

Then, using a foreach loop, the program will iterate over each line of the program file and parse out opcodes and operands, using the split() method. As long as the halt() method is not invoked, the program continues taking instructions for both the opcodes and operands. When assuming that the halt() method has been invoked, the console prints end of job and increments job counter (jbc) by 1. Then the add() method from PCB class is invoked and creates a new object from the PCB class whose constructor takes two parameters (memory and processId coming from the Program class), the state is updated to waiting and the ProgramCounter is set to 0.

Then, then program instruction is incremented by 1. If the instruction is greater than 512, an OutOfMemoryException exception will be thrown. Once this happens, the blocks (object pcbs) will be added to the blocks list

Then, the runSIMMAC() method is invoked. The JOBS\_IN\_SYSTEM will be true as long as number of jobs is greater than 0. Next, a foreach loop is used to iterate over the number of blocks in blocks list. This is done to execute each block collection in its own thread. A new thread instance is created with every iteration and an arrow function is used to invoked the IterateBlocks() method in the current Program class. The IterateBlocks() method takes a list of PCB objects. Now, inside this method, there is another foreach that will iterate over every process control block of type PCB in the list of blocks. Another instance of the class Thread is created with each iteration to execute each job in its own thread and a join() method is used to prevent a race condition.

A Class Diagram is provided below to graphically portray the relation between the classes used in this program:

# Class Diagram

A screenshot of a cell phone

Description automatically generated

Interface Design***:***

The only human-computer interaction in this program happens when the user is prompted to enter the program name, choose whether or not to continue entering program names. Once the user is prompted with answering if they want to execute the program, then flow continues. If answer is yes, then the program shows the output.