American University of Central Asia

Software Engineering Department

Operating Systems (COM 341)

Course Project

Part 1

Task #1: The CPU

The task is to create a CPU emulator for an imaginary instruction set architecture. You need to devise registers, create a set of instructions, implement a basic fetch/decode cycle, and create an interrupt generation and processing facilities. Students can base their work on any real CPU architecture.

The instruction set should include at least one command for loading values into registers, one for branching, and one for software interrupt generation.

You need to implement at least the following set of objects from the following hierarchy:

Kernel

- List of Processes
- Machine
    - Programmable Interrupt Controller (PIC)
        - Interrupt Service #1
        - Interrupt Service #2
        - Interrupt Service #3
        - ...
        - Interrupt Service #6
    - Programmable Interval Timer (PIT)
        - PIC object reference
    - Memory
        - Physical Memory (RAM)

        - CPU
            - Registers
            - Memory object reference
            - PIC object reference

Process (PCB)

- ID
- Registers
- State (Running, Ready, Blocked)
- Memory position

The kernel acts as a simulator of a real OS kernel. The kernel is written in a high-level language. It contains a list of processes and can create them by loading executable binaries (created by *vmasm*) from a file system into the physical memory of the virtual machine sequentially. It processes the timer interrupt to perform preemptive scheduling and responds to the first software interrupt to unload processes. The kernel can start and stop the machine execution.

The machine acts as a cycle generator. It asks the processor to fetch and perform the next operation from memory and tells the timer to respond to the next processor cycle. After each consecutive step, the processor adjusts its registers appropriately (e.g. changes the general-purpose registers, advances the program counter). The processor can generate a software interrupt through the PIC. The PIC contains vectors for hardware and software interrupts. The kernel adds appropriate interrupt handlers to the PIC during initialization and starts the machine. The function/method pointers or function objects (with or without lambda expressions) can be used to implement interrupt routines. The timer can constantly generate an interrupt through the PIC after a predefined number of cycles. The memory object contains a set of fixed-size arrays representing different memory regions or mappings (RAM, video memory, etc.).

The process object encapsulates the execution state of the processor. It stores the ID, state flag, and its memory position.

Students should implement at least four different scheduling algorithms (discussed in class and in section #2.4 from the course book). An appropriate data structure should be used for each of them.

Required

- Round-Robin Scheduling
- Priority Scheduling

Plus, any additional two from the following list:

- First-Come First-Served

- Shortest Job First
- Multiple Queues (+0.2 extra points)

The scheduling algorithm can be selected with a command-line flag passed to the application. The list of processes to load and execute and (+priorities for certain schedulers) should follow immediately.

The following command starts the kernel with the round robin scheduler and three processes loaded from the file system:

```
svm /scheduler:rr change_registers_and_exit.vmexe change_registers_and_exit.vmexe
change_registers_and_exit.vmexe
```

Students should also implement the exit system call in their kernel. The exit call unloads the current process and performs a context switch.

The implementation is defined as function/method pointers or function objects (with or without the lambda expression syntax) for respective hardware or software interrupts.

Major operations of the virtual machine should be outlined as a set of log messages.

Students must document their CPU. The documentation should include names of every register, list of operation codes, respective mnemonics, and meaningful description for every element.

Sample Documentation

VMCPU

32-bit ISA

Registers

Four general-purpose 32-bit registers

Two 32-bit special registers: SP, PC

A 32-bit *flags* register

General-Purpose

- *A, B, C, D*

Program Counter (Instruction Pointer)

- *PC*

Stack Pointer

- *SP*

Flags

- *FLAGS*

[31: N|30: Z|29: C|28: O|27–0: Reserved]

N (bit 31): negative condition flag

Z (bit 30): zero condition flag

C (bit 29): carry condition flag

O (bit 28): overflow condition flag

Instructions

mov reg imm32

Moves an immediate value to a register

- *0x10: mov a imm32*
- *0x11: mov b imm32*
- *0x12: mov c imm32*
- *0x13: mov d imm32*

jmp imm32

Jumps to a relative address forward or backward

- *0x20: jmp imm32*

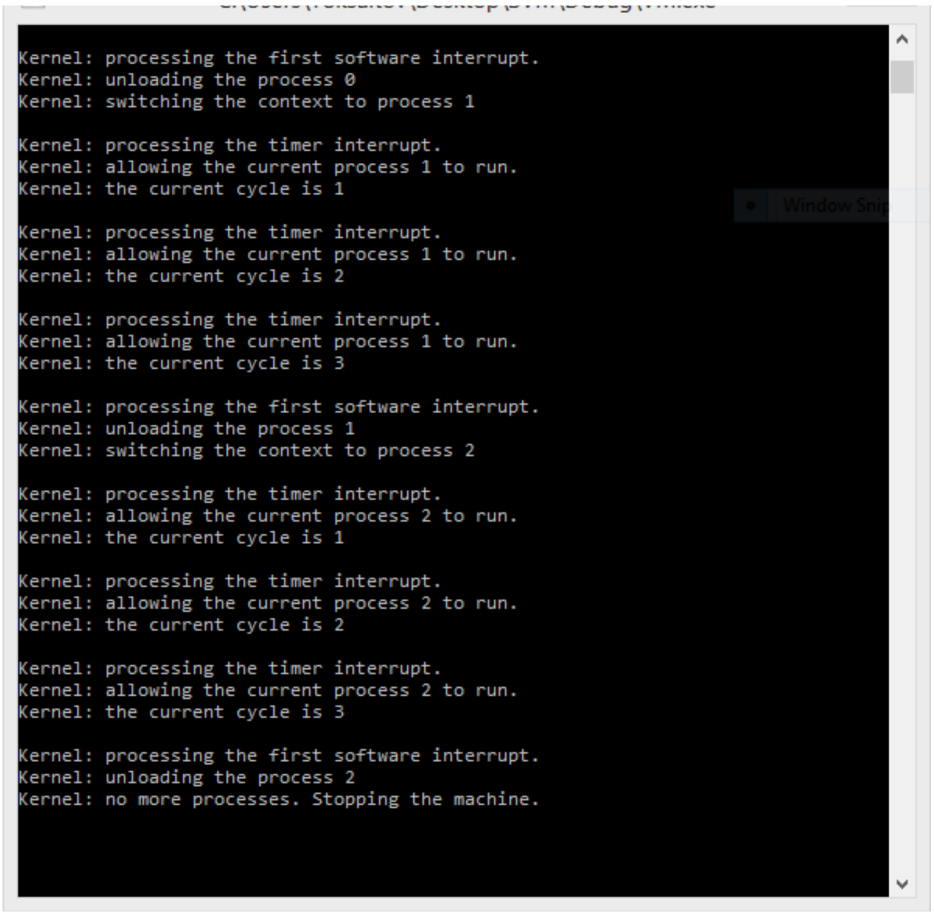
int imm32

Causes a software interrupt specified by the immediate value

- *0x30: int imm32*

Logging

Example #1



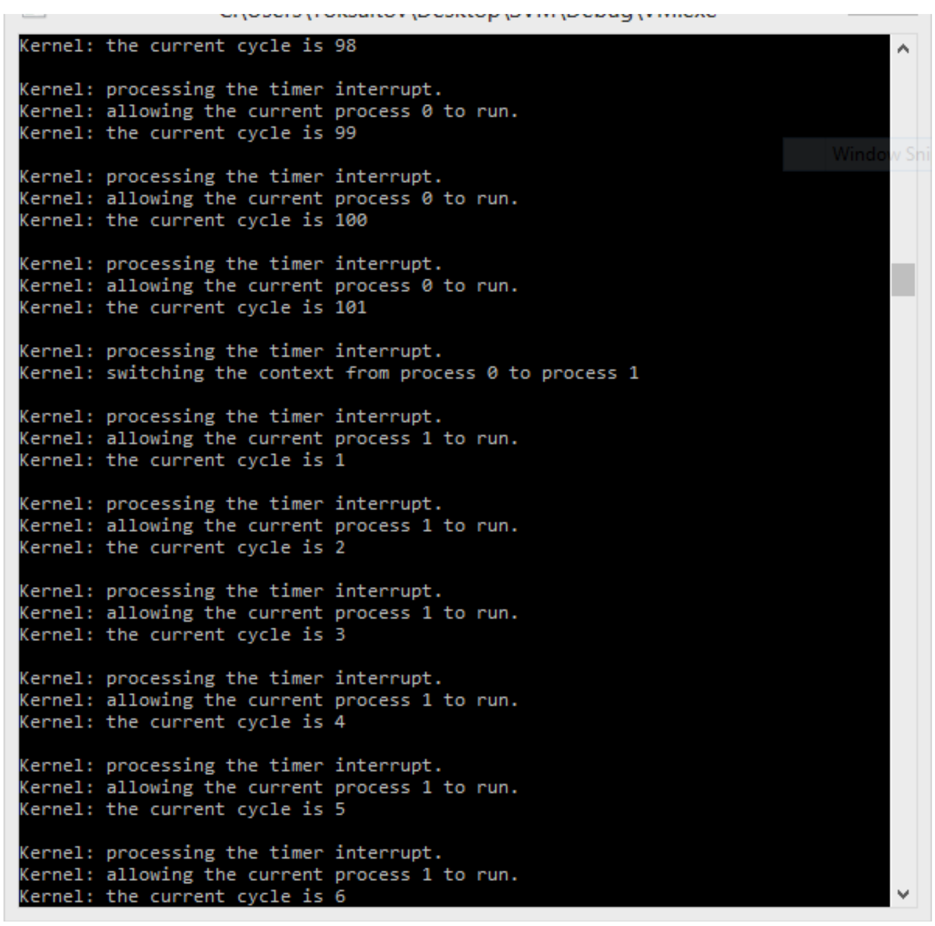
```
Kernel: processing the first software interrupt.
Kernel: unloading the process 0
Kernel: switching the context to process 1

Kernel: processing the timer interrupt.
Kernel: allowing the current process 1 to run.
Kernel: the current cycle is 1

Kernel: processing the timer interrupt.
Kernel: allowing the current process 1 to run.
Kernel: the current cycle is 2

Kernel: processing the timer interrupt.
Kernel: allowing the current process 1 to run.
Kernel: the current cycle is 3

Kernel: processing the first software interrupt.
Kernel: unloading the process 1
Kernel: switching the context to process 2

Kernel: processing the timer interrupt.
Kernel: allowing the current process 2 to run.
Kernel: the current cycle is 1

Kernel: processing the timer interrupt.
Kernel: allowing the current process 2 to run.
Kernel: the current cycle is 2

Kernel: processing the timer interrupt.
Kernel: allowing the current process 2 to run.
Kernel: the current cycle is 3

Kernel: processing the first software interrupt.
Kernel: unloading the process 2
Kernel: no more processes. Stopping the machine.
```

Sample kernel with three processes in a queue with IDs 0, 1, 2 created from the following code:

```
mov a 42 # test that we can change register values
mov a 1  # system call numbers go to the register a
mov b 0  # the first parameter goes into the register b
int 1    # perform the system call (tell the kernel to unload the current process)
```

The round robin scheduling was used with a quantum value set to 10 timer ticks.

Example #2



```
Kernel: the current cycle is 98

Kernel: processing the timer interrupt.
Kernel: allowing the current process 0 to run.
Kernel: the current cycle is 99

Kernel: processing the timer interrupt.
Kernel: allowing the current process 0 to run.
Kernel: the current cycle is 100

Kernel: processing the timer interrupt.
Kernel: allowing the current process 0 to run.
Kernel: the current cycle is 101

Kernel: processing the timer interrupt.
Kernel: switching the context from process 0 to process 1

Kernel: processing the timer interrupt.
Kernel: allowing the current process 1 to run.
Kernel: the current cycle is 1

Kernel: processing the timer interrupt.
Kernel: allowing the current process 1 to run.
Kernel: the current cycle is 2

Kernel: processing the timer interrupt.
Kernel: allowing the current process 1 to run.
Kernel: the current cycle is 3

Kernel: processing the timer interrupt.
Kernel: allowing the current process 1 to run.
Kernel: the current cycle is 4

Kernel: processing the timer interrupt.
Kernel: allowing the current process 1 to run.
Kernel: the current cycle is 5

Kernel: processing the timer interrupt.
Kernel: allowing the current process 1 to run.
Kernel: the current cycle is 6
```

Another kernel configuration with three processes in a queue with IDs 0, 1, 2 created from the following code:

```
mov a 42 # test that we can change register values
jmp -2   # jump back to the previous operation code by setting the PC/IP register
```

The quantum value for the round robin scheduling was set to 100 timer ticks.

Task #2: The Assembler

Create a command-line assembler for the created instruction set. The assembler should be able to process an input source file, parse it, translate tokens into a set of operation codes with data, and write it to the output binary file. The file names are provided as command-line arguments. The assembler will turn the source assembly code for your CPU architecture into a binary file containing a stream of operation codes and data. The compiled binary should be used in the virtual machine as an executable process. Each file should be loaded into a random access memory of a virtual machine sequentially for processing.

```
vmasm source.vmasm output.vmexe
```

Notes

The name of the emulator is *svm*. The name of the assembler is *svmasm*.

The following compilers are allowed: *VC (>=10)*, *Clang (>=3.1)*, and *GCC (>=4.2)*

Homework

Supporting Documents

Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C, Chapter #3

ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition, Chapter A2

Reading

Operating Systems Design and Implementation, Third Edition by Andrew S. Tanenbaum, Section #2.4 (AUCA Library Call Number: QA76.76.O63 T35 2006)

Operating System Concepts by Abraham Silberschatz, Peter B. Galvin, Greg Gagne (AUCA Library Call Number: QA76.76.O63 S558 2009)

Language Reference

The C++ Programming Language, Third Edition by Bjarne Stroustrup (AUCA Library Call Number: QA 76.73.C153 S773 2005)

C++ Language and Library Reference <http://en.cppreference.com>

Submission

Students have two weeks to finish the task. The kernel source tree should be packed and sent to toksaitov_d@auca.kg (pack the solution or a set of source files into a *zip* or *tar* archive).

It is recommended to use a version control system (VCS) such as Git or Mercurial to record you changes. You will get 0.1 extra points if you will provide your kernel source tree via the local repository or through a public web-based one (GitHub, Bitbucket, etc.).

Compose project description properly in a formal way. Describe the approach used for the solution.

The structure of the archive/repository:

```
+ <last_name>_<first_name>_project_1
|--+ svm
|  |--- Project/Solution files and directories
|  |--- ...
|  |--+ assemblies
|     |--- test_exit_syscall.vmasm
|     |--- test_infinite_loop.vmasm
|     |--- ...
|--- CPU_Documentation.<extension>
```

Grading

To be eligible for a certain grade you need to implement the following set of scheduling algorithms. You final grade can be lower if your solution is not properly decomposed into a set of classes and methods or has an inconsistent code style (e.g., different naming conventions used for variables and methods, inconsistent code indentation).

None: F

First-Come First-Served: D

+ Shortest Job First: C

+ Round-Robin Scheduling: B

+Priority Scheduling or Multiple Queues: A

Rules

Students are required to follow the rules of conduct of the Software Engineering Department and American University of Central Asia.

Team work is NOT encouraged. Equal blocks of code or similar structural pieces in separate works will be considered as academic dishonesty and all parties will get zero for the task.