COEN 346 Programming Assignment #3

The main objective of this assignment is to simulate operating system's virtual memory management and concurrency control. The virtual memory being managed consists of a main memory and a large disk space. Both are divided into 'pages'. The number of pages of the main memory is limited while disk space has unlimited number of pages. Each page stores a variable in the form (id, value), such that id is the identifier of the variable and value is the value associated with the variable. To manage (store, search, and remove) variables, the virtual memory manager offers three APIs to the processes:

a. **Store (string variableId, unsigned int value)**: This instruction stores the given variable id and its value in the first unassigned spot in the memory.
b. **Release (string variableId)**: This instruction removes the variable id and its value from the memory so the page which was holding this variable becomes available for further storage.
c. **Lookup (string variableId)**: This instruction checks if the given variableId is stored in the memory and returns its value or -1 if it does not exist. If the Id exists in the main memory it returns its value. If the Id is not in the main memory but exists in disk space (i.e. page fault occurs), then it should move this variable into the memory and release the assigned page in the virtual memory. Notice that if no spot is available in the memory, program needs to swap this variable with the least recently accessed variable, i.e. the variable with smallest Last Access time, in the main memory.

Example: consider a virtual memory manager managing a main memory that can contain 2 pages, and an unlimited disk space. A process P arrives at t=0, at t=1 it stores variable '1' with value '10'. The main memory now has one filled page (the first page) with ('1','10'). ('1','10') has a last access value equal to 1. At t=2, P stores a variable '4' with value '5'. Since, the main memory still has an empty spot ('4','5') will be stored in the main memory. ('4','5') has a last access value equal to 2. At t=3, P stores ('3','100'), since the main memory is full this variable will be stored in the disk space. At t=4, P performs a lookup for variable '1', since '1' is in the main memory, the value '10' will be returned and last access of ('1','10') will be updated to 4. At t=5, P looks up '3', '3' is not in the main memory since it was stored in the disk space. Therefore, the virtual memory manager must swap it with the variable with the smallest values of last access, namely ('4','5'), as a result, ('3','100') will be moved to the second page of the main memory, ('4','5') will be moved to the disk. And the value '100' will be returned, and last access associated with ('3','100') will be set to 5.

Input: input will consist of a memconfig.txt file which contains the number of pages in the main memory. A processes.txt file which contains, the number of cores C, the number of processes N followed by N lines each contains the process start time and duration. A commands.txt file which contains the commands to be executed by the processes. Any process should continually pick one command from the list of commands. The command that will be picked at any time is the next command (the one after the

command last picked by any process). If all the commands have been executed, the process will simply start over from the beginning. To invoke the commands the processes should call the suitable API of virtual memory manager based on the picked command. The process should wait for a random time (from 1 to 1000 milliseconds) between each API call to simulate the time each API call may take. The commands in commands.txt look like the following:

a. **Store** [variable ID] [value]: e.g. "Store 10 5"
b. **Release** [variable ID]: "Release 10"
c. **Lookup** [variable ID]: "Lookup 10"

Output: file output.txt that captures the following events with their associated information:

- The time each process has started/finished: with the value of the clock at that time, the id of the started process, and the name of the event (Started or Finished).
- Each command executed by a process: the value of the clock at the time the command was invoked, the name of the command, its arguments, its output if any (in the case of lookup, c.f. example below).
- Other events of the memory manager: mainly the swapping when needed within a lookup. The string tracing this event should also contain the Ids of the variables being swapped, the first variableId being the variable id of the variable that was in the disk space, the space being the variableId of the variable that was in the main memory.

Implementation requirements:

The following issues should be considered for simulation:

1- Processes should be simulated by threads. Each process has a starting time and total duration. The thread simulating the process (one thread per process) should run for a time equal to the duration of its associated process as indicated in the input. It should start the first time the scheduler is supposed to schedule its associated process (when its arrival time has come, and there is at most one process in the CPU at the time). The algorithm used to schedule the processes is FIFO (First In, First Out). Note that the maximum number of processes that can run at the same time equals the number of cores given as input in processes.txt.
2- The FIFO scheduler should be running on its own thread.
3- Your program will need a clock. The clock should be running on its own thread.
4- Virtual memory: the pages in the main memory should be simulated by an array in the actual computer physical memory, while disk pages are simulated by an array located into a text file such as "vm.txt", **which must be accessed every time we need to access the disk**.
5- Processes try to store, release and retrieve "variables" from/to the memory. Each page contains only one variable Id and its value. Of course, working with memory can only happen when the process is running. Each variable in the memory has a Last Access property which is a time stamp indicating the last access time to this variable.
6- Virtual memory manager should be running in a separate thread that receive the API calls from other processors and execute them.

Programming Tips:

- In order to protect critical sections and ensure mutual exclusion, use appropriate tools, semaphores for instance, within the body of each function.
- The program must work with arbitrary number of processes and arbitrary size of memory.

Following are the list of sample input / output files.

Input Files:

- processes.txt

```
2
3
2       3
1       2
4       3
```

- memconfig.txt

```
2
```

- "commands.txt"

```
Store 1 5
Store 2 3
Store 3 7
Lookup 3
Lookup 2
Release 1
Store 1 8
Lookup 1
```

Output File:

```
Clock: 1000, Process 2: Started.
Clock: 1010, Process 2, Store: Variable 1, Value: 5
Clock: 1730, Process 2, Store: Variable 2, Value: 3
Clock: 2000, Process 1: Started.
Clock: 2010, Process 1, Store: Variable 3, Value: 7
Clock: 3000, Process 2: Finished.
Clock: 3020, Memory Manager, SWAP: Variable 3 with Variable 1
Clock: 3030, Process 1, Lookup: Variable 3, Value: 7
Clock: 3100, Process 1, Lookup: Variable 2, Value: 3
Clock: 3800, Process 1, Release: Variable 1
Clock: 4000, Process 3: Started
Clock: 4200, Process 3, Store: Variable 1, Value 8
Clock: 4400, Memory Manager, SWAP: Variable 1 with Variable 3
Clock: 4410, Process 1, Lookup: Variable 1, Value 8
Clock: 5000, Process 1: Finished.
Clock: 7000, Process 3: Finished.
```

The assignment should be done in a team of three students. The deliverable consists of a well-commented code and a report. Do not include your code in your report.
The report should be at least two pages with the following sections:

- Name of group members
- High level description of the code (description of the methods/functions/threads/data structures and the flow of the program).
- A detailed conclusion, discussing you experience with concurrency control in simulating virtual memory management. You also need to analyze your program and discuss how you enable mutual exclusion in your program and the efficiency of your technique.

This assignment will take 50% of the total mark for programming assignments. Also, for this assignment 80% of the mark is dedicated to your code and 20% to your report.

The code and the report should be submitted in one zip file through Moodle by Monday April 12th, 2021 (11:59 pm).

Also, you will have to **demonstrate your work as a team during the week of April 12**. This will be done through Zoom.