PROGRAMMING ASSIGNMENT #3 REPORT

Simulating Virtual Memory Management

Name: Nabilah Rouf (40095893)
Thomas Tran (40095654)
Date: April 16, 2020
Prof. Ferhat Khendek
COEN 346

Objective

The purpose of this assignment is to be able to simulate an operating system's virtual memory management while taking into account the scheduling and synchronization of the processes that are to be executed. In essence, a file is read in order to identify which functions the processes are meant to call, however, storage space for variables is required. Since virtual memory in this assignment is made up of a fixed size array for main memory and unlimited disk space in the form of a list, the variables may not always find space in the main memory and must go to the list, which is then written into a file. It is important to take into account that the simulated system has two CPUs, meaning that two processes can run at the same time.

High Level Description

General Description

The written code is divided into multiple classes: Process, Scheduler, Clock, Variable, Commands, VMM and OSVMM. The main class, OSVMM, creates the lists, such as the process list. The processes read from the "processes.txt" file are initially inserted in the processes list. The "commands.txt" file is also read and inserted into a list, and the main memory array is created according to the indicated size in "memconfig.txt". The scheduler and VMM threads are started as well.

Commands and Variables class

The commands class includes the properties for each command: its function, id and value along with the appropriate getters and setters. The commands are added to a list in OSVMM, as mentioned earlier. The variable class includes the properties associated with each variable: its value, ID and last access time. These variables are to be added and removed from virtual memory and manipulated using the methods in the VMM class.

VMM class

The VMM class includes the methods: memLookup, memStore and memFree. The lookup function works by looking through the main memory array and checking to see if any of the variable IDs match the id that was passed as a parameter. If it is found, then the value of this variable is returned, else the variable with the passed ID is looked for in the disk space, which is in the form of a list. If it is found and there is an empty space in main memory, then the variable is added into the array. Otherwise, the memoryArray is searched in order to find the variable with the least access time. Once this is found, the variable is swapped out from the list and added to the array in this variable's place. If the variable is not found, the method returns -1.

To add, the memStore method works by taking a variable's ID and value and storing it as a variable in the main memory array if there is space, or in the disk space if there is no space. The memFree method on the other hand works by searching the main memory array for the variable ID that was passed, and removing it from the array. If it is not found in main memory, and it is found in the disk space, then it is removed from the list. Otherwise, a string is outputted that reads that the variable does not exist.

The commands are called in the VMM class according to what is read. The process class and the VMM class should work in sync so as to let one process access the VMM at a time. This was done using locks and busy wait loops. For instance, while the lock is equal to 1, the process can run and call the commands. However, after the process is done running, the lock is set to 0. In the VMM on the other hand, when the lock is set to 0, it cannot run, but when it is set to 1 it can proceed.

The process reads the command and passes the command to the VMM and sets the VMM lock so that the other process cannot access VMM at the same time. The process also waits for that lock to be unset so that it knows that its request has been served and moves on. When both processes finish, the scheduler can break its busy wait loop.

Process class

Each process has its own arrival time, burst time (which is continuously updated), a number in order to identify it, and access to a key (if available). In the run() method in the process class, the commands are called in accordance to what is read from the commands list. The burstTime of that process is decremented by the time quantum (1000 ms was used in this case). However, before the process can access the commands, a lock is set to the processNumber. If the lock is not the processNumber, then it continues busy waiting. Once the process is finished running, the lock is set back to 0, so that another process can now run. In addition, the CPUAccess key is incremented again because this process has finished its time with the CPU for now.

Scheduler class

The scheduler class, which extends thread, includes its constructor, a move method (ex. for when the processes must move from the processList to the readyList) and the run method. The scheduler class includes 4 lists, processes (for when the processes first arrive), ready, running and terminated. The scheduler works by continuously looping until the size of the terminated list is the same as the number of processes that there are. This means, that all of the processes have finished executing as they are now all in the “terminated” state. However, when the size of this list is smaller than the number of processes, then the processes list is first checked to see if it is empty. If it is not empty, then the arrival time of the first process in this list is compared to the clock to see if it is its time to execute. If its arrival time has come, then the number of CPUAccess keys is checked to see if there are enough. If it is able to obtain a key, then the process moves to the running list, decrements the number of keys and then it is started. The processes list is checked again (as this is in a while loop) to see if there’s another process whose arrival time has come. If so, then this process is also moved to the running list and started, otherwise, it checks the ready list. However, the ready list is only checked if it is not empty and if there are enough keys available (if there aren’t already 2 processes running).

The output is not read into the output.txt file because the while loop is never broken due to the fact that the program does not run properly. As seen in the image below, which was obtained from using System.out.println(), the clock stops at 3000, and two processes are resumed at the same time. Please disregard the random numbers in between used to debug.

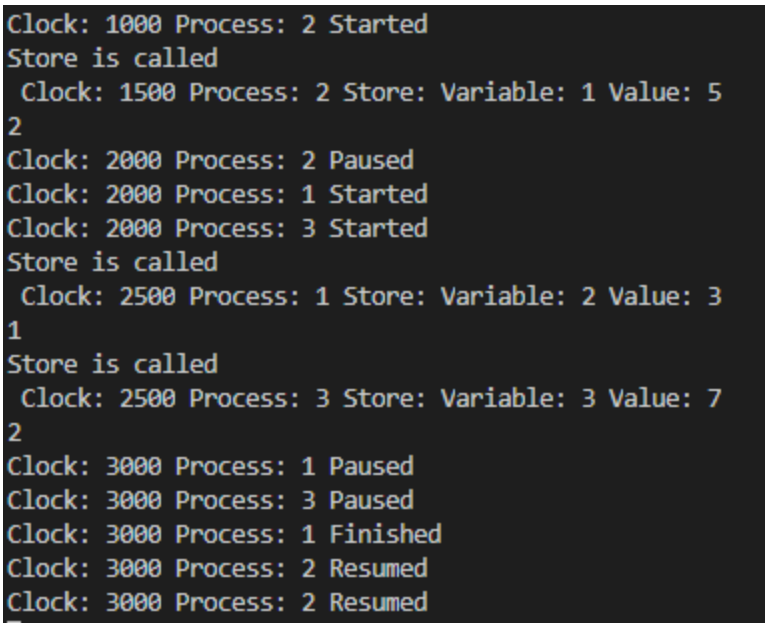

Figure 1. Output using System.out.println(“”)

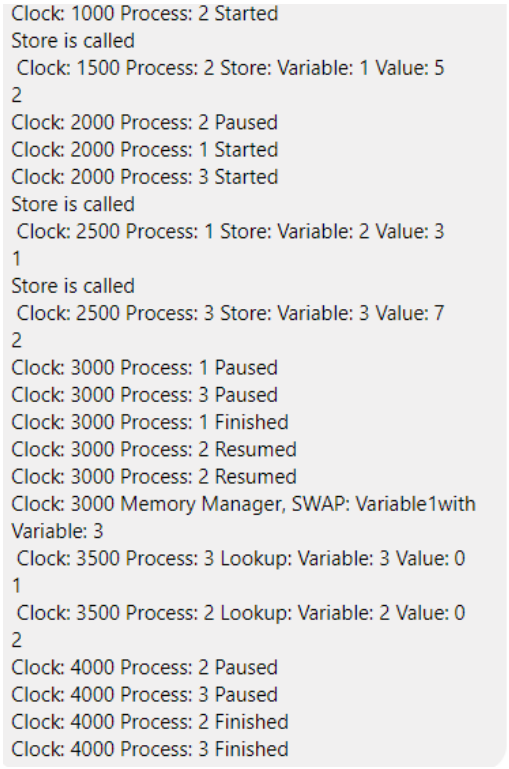

Figure 2. Another output obtained from the code (with the input from the Assignment).

From the debugging, we have found that certain threads are stuck in the locks when they shouldn’t be, so synchronization was a big issue in this assignment, as will be explained in the conclusion. The program gets stuck multiple times because sometimes certain threads are stuck in the locks. The lock CPUAccess, lock (checking if its equal to processNumber), the lock for

the VMM and the process, are all stuck in circles, which is why it's not proceeding properly when it runs sometimes, and does not complete the output. For example, since the while(CPUAccess.get() <2) is checked in the scheduler, and clock is only incremented after this, the clock is not able to increment to 2000 because the CPUAccess key is not incremented to 2 due to the other locks. Therefore, the next processes are not able to start. This is why sometimes the output is not as expected.

Conclusion

The concept of virtual memory as seen in class was further delved into with this programming assignment as the concepts of main memory and external disks were used in order to solve the assignment.

Process synchronization was a little more difficult to implement for this assignment considering that there are two processes that can be executed at the same time, since two CPU cores are available for use. However, seen as how using suspend() and resume() methods did not go according to plan for the previous assignment, we tried using semaphores instead. However, although implementing using semaphores was easier to understand in terms of the flow of the program, it was difficult to understand which thread will gain access to the semaphore once one was available. Therefore, in order to keep the same key concept as with a semaphore, an atomic integer was used instead in order to proceed with synchronization of the processes. Each process holds the key when it has access to one of the two CPUs, otherwise it busy waits. An atomic integer was used instead of a regular integer in order to prevent multiple threads from incrementing or decrementing its count at the same time (thus enabling mutual exclusion).

There had to be synchronization between the process and the VMM as well, and this was done using busy wait loops and a lock. We have struggled with synchronizing the program so that the VMM outputs properly, and therefore, there are times when the output shows that multiple processes have started, resumed and paused while other times, it gets stuck after the first process is started. Therefore, the written program is not efficient as outputs are not consistent.