COEN 346

Lab Assignment #3 - Virtual Memory Manager

Jose Ricardo Monegro Quezada - 40087821

Karl Noory - 40059592

Yevhen Haydar - 40024141

Introduction:

During this programming assignment, the team has been tasked with simulating a virtual memory manager. To achieve this, there is a need for a simple scheduler which manages a list of processes in a FIFO approach, the virtual memory manager itself, and shared memory which processes can use to send commands that will then be executed by the memory manager. In addition to these major parts, a parser is needed for the various input files, a writer/logger to assist in generating an output file, and a timer class that simulates the execution clock. Due to independence of various logical blocks, the development was approached with a modular structure. Each part of the project could be developed and tested independently from one another and later integrated together to finalize the project. Additionally, care was taken to ensure that critical sections were well protected.

Discussions:

1. CommandBuffer (shared memory)

The CommandBuffer is a simple but important part of the program. It allows for the indirect communication between the processes and the virtual memory manager. It is basically an std::queue<Parser::Command> with thread safe (using mutex locks) push(...) and pop() commands. The class being thread safe allows multiple processes to push commands, while the virtual memory manager can pop those commands simultaneously without causing any weird errors related to racing conditions.

2. Parser

The parser is used to extract the information from the inputs. These inputs include the maximum number of pages for main memory (memconfig.txt), list of commands the virtual memory manager should execute (commands.txt) and process information (processes.txt). Once the program starts running, a Parser object is created using the "new Parser(root_file_path)" and three methods are called on it. The first method is parseMemConfig(), the second is parseProcess() and the third is parseCommands(). Once the parse functions complete their execution, all the necessary data will be stored in appropriate structs which can be retrieved later by calling the appropriate functions on the Parser object.

3. Writer

The writer class is needed to facilitate the output of the program to the output.txt file. The Writer object is created using "new Writer(writer_path, timer)" where we pass the output file path and a reference to the timer object. This Writer object, assuming it has successfully been created with no errors, will then be passed by reference to the scheduler and the virtual memory manager. To write to the output file, the scheduler and the virtual memory manager simply need to call the write(string) function to tell the Writer object to output a new line to the output.txt and also display it in the CLI. Since, we gave the Writer object a reference to the Timer, a timestamp will automatically be appended to the beginning of the each specified message. Since multiple parts of the program can write to the output at the same time (critical section), mutex locks were used to prevent any issues while writing to the common output file.

4. Scheduler

The scheduler that was implemented in this project operates simply as a FIFO scheduler. The scheduler is made aware of the number of available cores, number of processes as well as a list of said processes. The initial task of the scheduler is to sort the processes based on their arrival time. Once they have been sorted, the scheduler will execute its loop cycle. Each loop, the scheduler starts by managing the active processes, then it will see if new processes can be started if a core is empty. The goal is always to fill in all cores (maximize resource utilization), unless no more processes are available to start. Once no more processes are in the waiting queue and all active processes have finished, the scheduler will terminate.

Processes when made active, will generate a random number between 1 and 1000ms to wait to send a command to the virtual memory manager. If the wait time exceeds the stop time of the processes, the wait time is shrinked to prevent the process from running longer than it should. Once a command is ready to be executed, the process will send it to the shared memory, which is meant to exchange information indirectly between processes and virtual memory manager. The command list is looped infinitely.

5. Virtual Memory Manager

The virtual memory manager is composed of three parts, a file parser, a memory manager, and a thread controller. The file parser (vmm::vmm_file) takes care of the reading and writing operations done to the physical memory file (vmem.txt). It encodes and parses the data in a format that it can understand.

The memory manager (vmm::vmm_manager) handles the logic and rules for the rest of the manager. The store, release and lookup functions are implemented there, as well as logging and time keeping. To allow for better abstraction, the memory manager uses a set of callback functions to allow for better handling of itself by external classes.

The thread controller (vmm::vmm_thread) provides an option to execute the virtual memory manager thread on a separate thread from the rest of the system. It inherits its thread controlling mechanism from thread_controller (recycled from assignment 2). The thread controller uses the CommandBuffer object (shared memory) to retrieve the instructions that it will execute sequentially on a loop, and will remain idle (thanks to the use of a condition variable) if no more executions are left to be run.

Conclusion:

In this assignment, the concurrency between the processes and the virtual memory manager has been facilitated by implementing a shared memory object called CommandBuffer. Processes can push new commands in the queue that is the shared memory, and the virtual memory manager can retrieve commands from it when necessary. This was a great learning experience as it was crucial to pass information without interference between the multiple processes. The shared memory was implemented using a queue and a mutex, since multiple processes and the virtual memory manager can access and modify the shared memory simultaneously. At multiple other instances, there was a need for mutex implementation to protect variables that multiple threads had to access. This definitely helped cement our understanding of critical sections and showed us how we can simulate basic memory managers.