John Kucera

Prof. Suzanna Schmeelk

CMSC 412 Operating Systems

28 February 2021

Week 8: Final Project (REPORT)

**Demand Paging Virtual Memory Simulator (Java)**

**Introduction**:

**Virtual Memory** is a memory scheme that expands the amount of data that RAM can have access to. With this feature, you can treat chunks of storage in the HDD as “RAM” in addition to the memory that RAM already has in physical form. For example, if a program requires more memory than is already on RAM physically, the HDD will use its empty storage to host “virtual memory” that the RAM can use to continue its processes.

There are several virtual memory management methods to help organize and optimize the data that is kept on RAM, the data that is kept on HDD, and how they are swapped between each other as processes continue. One of these methods is **Demand Paging**. With this method, a process can be broken down and its piece get split between RAM and HDD. The optimal parts can stay in RAM to continue functioning, and the ones not currently in use can rest in HDD. So, when RAM needs a memory page that is in HDD’s virtual memory, it must request that data to be swapped into use. When RAM accesses a page that is not currently in use (and therefore becomes in use), it raises an exception called a page fault. If there was data in a frame that needed to be swapped out of physical memory to make room for the new page, that frame is called the victim frame.

The question is: how do you decide exactly which frames to swap out when other data needs to be accessed? This is where **page-replacement algorithms** come in (hence the name). Demand paging uses these algorithms to make decisions for page replacement.

**Project Overview**:

This project simulates some of the page-replacement algorithms that Demand Paging uses to manage virtual memory. It is a Java program that, after user input to store a reference string, runs that reference string through one of the page-replacement algorithms. The hypothetical computer can have N=0 to N=8 physical frames and up to 10 virtual memory frames. The reference string can either be input by the user or randomly generated. Using an algorithm generates a plaintext diagram that shows each physical frame, the string values stored in those physical frames, page faults that occur, and which frames become victim frames. As the user presses *enter* to continue, the diagram updates in a way that simulates that specific algorithm. Here are the page-replacement algorithms simulated in this project:

* “First in, First Out”, or **FIFO**. With FIFO page replacement, *the frame that was brought into memory FIRST will also be the first one to be swapped out* and become the victim frame (when other data is swapped in).
* “Optimal Page-Replacement”, or **OPT**. With OPT algorithm, it is already known what future values will be referenced to (in other words, it already looks at the entire reference string). When data is swapped in, it “looks in the future” and *the victim frame is the one whose value is not going to be referenced for the longest amount of time*.
* “Least Recently Used”, or **LRU**. With LRU page replacement, *the frame whose value has not been referenced for the longest time* becomes the victim frame and gets swapped out (when other data is swapped in).
* “Least Frequently Used”, or **LFU**. With LFU page replacement, *the frame whose value has been referenced the least amount of times becomes the victim frame* and gets swapped out (when other data is swapped in).

**Design Description & Implementation**:

To achieve the requirements of this project, I created one Java class (Main.java) which includes the following methods:

* **Main() constructor**: Prints the menu that prompts user to select 1 of the 8 options: read a reference string, generate a reference string, display current reference string, simulate FIFO, simulate OPT, simulate LRU, simulate LFU, and exit. This also uses a while loop to keep the user looping through the menu after tasks are finished or errors are raised (until the user selects “0 – exit”). If no reference string has been set yet, options 3 through 7 will notify the user and loop straight back to the menu. If a non-integer OR integer that is outside of 0 to 7 is selected, an error is raised, and the user is looped straight back to the menu to try again.
* **readReferenceString()**: Called when option 1 is selected on the menu. First, the user is prompted to enter the length of the reference string they will input (inputting a non-positive or non-integer will raise an error and make the user try again). Then, using an ArrayList<Integer> to store the reference string, the user is prompted to enter the string values 1 at a time so that each can be validated/verified. If an entry is not an integer from 0 to 9 (the only values that this virtual memory holds), an error is raised and the user must try again. After each entry, the reference string is printed for the user to see. When it is all done, the method returns the whole reference string as an ArrayList<Integer> object.
* **generateReferenceString()**: Called when option 2 is selected on the menu. First, the user is prompted to enter the length of the reference string they will input (inputting a non-positive or non-integer will raise an error and make the user try again). Then, using java.util.Random, random integers from 0 to 9 are generated that create the reference string. The method returns the whole reference string as an ArrayList<Integer> object.
* **createDiagram()**: Called in options 4-7. This method takes the number of physical frames and the reference string and uses them to create the String[][] object that will later be printed into the simulation diagram. Elements of the diagram, such as the leftmost labels and the reference string in the uppermost row, are inserted into the String[][] object. The method returns the String[][] object.
* **printDiagram()**: Called in options 4-7. This method takes the String[][] created by createDiagram() and prints it into a neat, proper-looking diagram that is meant to look identical to the diagrams in Module 3.
* **getPhysicalFrames()**: Called in the beginning of options 4-7. This method prompts the user to enter the number of physical frames they want in their algorithm simulation – there can be 1 to 8 physical frames. If the user enters anything outside of this range or a non-integer, an error is raised and they have to try again. After successful entry, the method returns the number as an integer.
* **Algorithms**: These 4 methods are called when options 4-7 are selected and have many similarities. They all require the reference string as a parameter. First, an ArrayList<Integer> object is created that is meant to store current data for the physical frames. A String[][] object is created using createDiagram() and is then printed initially. For each step in the diagram, the user must press enter to continue. The steps are: for every value in the reference string, IF it is not yet in memory, it gets added in to the next empty frame and a page fault is generated. IF there is no empty frame, the algorithm is used to decide what becomes victim frame (will be explained shortly). IF the value is already in memory, nothing happens, and a page fault is generated. After the step is complete, the String[][] diagram is updated with information from physical memory, page faults, and victim frames. It then calls printDiagram() to print it with updated information. After full completion, total number of faults is printed. The user loops back to the menu.
  + **FIFO**: Keeps track of which frame came in first based on a count during each step (compared to the location of current frame), and that frame will become victim frame. New value replaces it.
  + **OPT**: Allowed to “look into the future” so this method creates a duplicate copy of the reference string. It looks at this and keeps counts of which values will be used in the future and when it will happen. The frame that will be referenced LATEST, becomes the victim frame. New value replaces it.
  + **LRU**: Creates an array of the physical frames and keeps a count of which frames do not get used, resetting the current frame when it is used. The frame that has the highest count (which means it has not been used recently) becomes the victim frame. New value replaces it.
  + **LFU**: Creates a HashMap that stores the count that each value has occurred. When assessing which frame gets swapped out, the frame whose value has occurred the LEAST (according to the HashMap) becomes the victim frame. **When there is a tie between values with equal frequencies, the victim frame is the one with the lowest physical frame number.**
* **main method**: Creates the menu with Main() constructor.

The flow of the program is like this: when the user presses run, the main method uses the Main() constructor to create the menu. The user can choose option 0, which is to exit the application; option 1, which is to input in a reference string; option 2, which is to generate a random reference string; option3, which is to read the current reference string; option 4, which is to simulate FIFO on the current reference string; option 5, which is to simulate OPT on the current reference string; option 6, which is to simulate LRU on the current reference string; or option 7, which is to simulate LFU on the current reference string. After each task or menu error, the application loops back to the menu to allow the user to continue. For each algorithm simulation, the user presses enter to continue through each step.

This simulator could be implemented in a classroom setting that teaches about virtual memory. Beginner students often have trouble visualizing how each algorithm works and exactly which frames get swapped out would benefit from seeing it step-by-step in this simulation. It could also be used for computer scientists who need to see how their applications get managed through virtual memory in different operating systems, and they could write the code based on certain algorithms that are best for their application.