Benjamin Marquis

CMSC 412

3/8/2022

**Demand Paging Report**

**Introduction:**

The purpose of this project is to simulate virtual memory paging. The different algorithms that are used are examples of virtual paging algorithms that are intended to make the usage of virtual memory as efficient as possible when simulating physical memory. The algorithms included in this project are First-In-First-Out, Optimal-Page-Replacement, Least-Recently-Used, and Least-Frequently-Used.

First-In-First-Out is the simplest algorithm, whenever a frame in physical memory needs to be cleared to create space for a reference the first reference that was added to the physical memory is the one that is removed. This is not as efficient as possible and typically less than desirable but it is very efficient with low overhead.

The Optimal-Page-Replacement algorithm is an idealized version of a demand paging algorithm. It is technically impossible to implement because there is no way for the computer to know what frames will be used in the future. It works by removing the reference that will not be reused for the longest time from the physical frame it is occupying in order to create space for the next reference.

The Least-Recently-Used algorithm simply removes the reference that was used least recently from its physical frame in order to create space for a new reference. It is likely the most practical and efficient algorithm for real world application because this is actually possible to know and decently efficient to implement.

The Least-Frequently-Used algorithm removes the reference that is used least frequently throughout the reference string. It is difficult to know which reference will be used the most often ahead of time so this algorithm is also less possible to implement. It is also only a slight improvement over the Least-Recently-Used algorithm.

**Overview:**

The design of this Java program is very simple. The main method of the program is an infinite loop that runs until the user indicates that they would like to exit. At each iteration the user is prompted to either enter in a reference string manually, generate a random reference string, view the current reference string, or simulate the usage of one of the demand paging algorithms.

If the user selects Read reference string, they will be prompted for the length of the reference string that they would like to enter and then the program will accept that many integers for input. If the user enters something that is not an integer an error message is displayed so that the user can try again.

If the user selects Generate reference string, they are prompted for the length of the reference string only. Then the program fills a reference string array with random numbers of the desired length. This reference string can then be used for the algorithm of the user’s choice.

Display current reference string will show the user the reference string that is currently in the program’s reference string array. If there is no reference string in the array yet because it has not been generated or read in manually the program displays an error message prompting the user to generate or read in a reference string first.

Each of the remaining options represents the simulation of the algorithm described. Whenever any of these options are selected the reference string is loaded and the algorithm is executed on it until completion of the reference string. Each iteration of the execution is shown one at a time until completion.

**Approach:**

The overarching approach to writing this program involved breaking down each step of each algorithm to its basic components. These components were then coalesced into the most efficient methods possible which were then utilized appropriately whenever necessary. This program contains 20 different methods for solving the given algorithms. This extensive framework was chosen to mitigate the need for unnecessary repetition in code where ever feasible.

**Design and Implementation:**

FIFO:

The Simulate FIFO option begins the simulation of the First-In-First-Out algorithm. First, it clears any faults or victims that are in the faults and victims array, then it prompts the user for a number of physical frames. The number of physical frames along with the length of the reference string are used to generate the frame data table, then the frames array is initialized and filled with the default value of -1. The program enters into a loop for the entire length of the reference string and makes a FIFO reference for each member of the reference string and displays all updated information at that time.

The FIFO reference method receives the integer value representing the iteration that the program is currently at. It then creates a temporary array that is one size larger than the array of frames that the user selected. It fills the temporary array with the data from the frames and then fills the last space with the value -1. At this time if the reference in the reference string at the current iteration is already in the physical frames array nothing is done and the method is exited. If the reference in the reference string is not present it is placed using the FIFO algorithm.

The FIFO algorithm places the reference that is not currently in the array of frames by shifting each item in the temporary array down one position. It adds the current iteration to the list of faulted frames. It then inserts the reference at position 0. It adds the new last member of the temporary array to the victims list and then updates the frames array without the removed item. The last step is for the algorithm to add the frames array to the array of arrays of frames data.

OPT:

Simulating the OPT algorithm begins by clearing any faults or victims in the faults and victims arrays. It then prompt the user for the number of physical frames and generates the frame table and frames arrays. The method then executes another method called generateReferenceProximityDictArr. This method creates an array of dictionaries of the same length as the reference string. Each dictionary in the array corresponds to a position in the reference string. Each dictionary is filled with the references that have been made up to that point by the program as keys, as well as the distance to the next occurrence of that reference as the value. Calculating the distance to the next occurrence is handled by the generateDistanceToNextOccurence method.

The program then enters into a loop that continually makes OPT references for each reference in the reference string. This is accomplished with the makeOPTReference method. It works by searching the current set of frames for the presence of the desired reference, if it is not present a fault is added to the fault array. Then the method checks to see if there is any space available in the frames. If there is space the reference is added in that position. If there is no space the method checks the appropriate dictionary in the array of dictionaries and looks for the frame that is holding the reference that will not be used for the longest amount of time. When that frame is found its contents are added to the victims array and then replaced by the reference that needed to be added. Then the frame data is added.

LRU:

At the start of the process for simulating the Least-Recently-Used algorithm the faults and victims array are cleared. Then the user is prompted for the number of physical frames in the simulation. Frame data is generated as well as the frame array. Then the method generateReferenceRecencyDictArr is called. It works in a similar manner to the generateReferenceProximityDictArr. It works by creating an array of dictionaries of the same length as the length of the array of references. Each dictionary is filled with the references that have been made up until that point of the reference string as keys, the values associated with the keys are the amount of references that have passed since the last usage of that reference. Hence the name Reference Recency Dictionary Array.

At this point the loop begins to cover every member of the reference string array. At each iteration a call to makeLRUReference is made. MakeLRUReference works by searching the frames for the presence of the reference that is being made at that time. If it is present nothing is done and the method is exited. If it is not present the method then generates a fault and searches for an open space to place the reference into the frames. If there is a space the reference is inserted. If there is no space the method uses the reference Recency Dictionary Array to select the frame that has been used least recency to make its insertion there. Then the victims array is updated with the victim that was removed and the frame data is updated with the new frames.

LFU:

The Least-Frequently-Used process starts the same way as the other algorithms. First the faults and victims are cleared, then the user is prompted for information to create the frames and frame data tables. The simulateLFU method then calls the makeReferenceFrequencyDict method. This method simply constructs one dictionary which holds the references in the reference array string as keys and sets their value to the number of occurrences that are made of that reference.

The simulateLFU method then enters into a loop for the number of references in the reference array. At each iteration a call to makeLFUReference is made. It operates by searching the current frames for the presence of the reference that is being made. If the reference is present nothing is done and the method is exited. If the reference is not present a fault is generated and the method searches for a space to put the reference into the physical frames. If no space is available the method then searches the Reference Frequency Dictionary to see which reference in the physical frames is used least often in the reference string array. When that physical frame is determined, its contents are added to the victim array and then replaced by the reference that is being made. Then the frame data is updated appropriately for the new physical frames.

**Conclusion:**

Overall this java code contains 20 different methods for accomplishing each of the various tasks demanded by the algorithms. Pseudocode was used extensively during the design of these algorithms. The data structures used by this code were chosen to be as efficient as possible, whenever possible arrays were used because of their low computational overhead. Dictionaries were also used extensively because of their fast look up times and efficient usage of memory. ArrayLists and other more convoluted data structures were avoided intentionally to keep the efficiency as high as possible.

Lessons learned:

1. Extensive and accurate pseudocode was instrumental in the design of these algorithms. Ensuring that pseudocode was algorithmically correct ahead of time and the designing the code to match the pseudocode as closely as possible was the only way to make these algorithms function in the coding environment.
2. Breaking down each step of the algorithm into smaller methods and separating those methods out individually and writing them as efficiently as possible was also a massive boon to this implementation. Theoretically, as always, the entire program could have been written in the main method but it would have resulted in a massive amount of rewritten code. Having readily accessible and understandable methods available made the amount of reworking minimal.
3. Using good variable and method names also helped immensely. Except for the array named nums, which was the reference string array, everything was named as closely as possible to describing its usage clearly to the programmer. That way methods and variables could be selected easily and utilized like tools in a toolbox.
4. There is probably a handful of small loops that were iterated through that could be merged to become more efficient, These little inefficiencies are difficult to find and typically do not result in a massive improvement to performance as they are not a very large computational demand. For instance, there is probably a better way to loop through the physical frames at any iteration and find either the presence of the reference or a space to put it in at the same time. This would yield a possible improvement.
5. Another potential improvement is a method for formatting the output. That was redone in each simulation entirely, where a repeated method call that implemented the display reference string and print frame data methods would have resulted in a smoother implementation process.