Matthew Towles
CMSC 412 – Operating Systems

Report Document – DemandPagingSimulator

Overview

This program is a demand paging virtual memory simulator implemented with Java. It simulates four different page replacement schemes:

- **FIFO** – next page to replace is the first one in among the current pages
- **OPT** – next page to replace is the one that will not be used for the longest amount of time
- **LRU** – next page to replace has been last used least recently among current pages
- **LFU** – next page to replace has been used the least in the past among current pages

The user, within a range of one to seven, can dictate the amount of physical frames in this simulator. The simulated single process that is running will always have a virtual memory of ten frames which means that the reference string values will range from zero to nine.

After the program starts, a menu prompts the user with eight different options. This menu will continue to prompt the user after executing the option chosen by the user until the user selects the *exit* option. In addition to *exit* and an option for each of the four paging algorithms, the other three options on the menu prompt are:

- *Read reference string* – reads user input for the reference string from the keyboard
- *Generate reference string* – randomly generates a reference string
    - Reference string length is given by the user
- *Display reference string* – simply outputs the current reference string

The reference string can only contain integers and white space. A reference string value can only be from zero to nine. Therefore, the reference string “12 3” is effectively the same as “1 2 3” as “12” in the first reference string will be evaluated as two separate numbers, not as twelve (since this is out of range of the virtual memory frames.) If a reference string input given by the user contains any characters other than white space, which is ignored, or integers then the user is notified and prompted to provide a new reference string. This is done as opposed to ignoring invalid characters in case the user made a mistake by entering a non-valid character.

There is no GUI for this program. All output and input are done via the console or command line.

main – Starting the DemandPagingSimulator

An integer, from one to seven, is expected as a command line argument by the program.

`setFrameCount(int count)`
This method sets the physical frame count. After the command line argument(s) are evaluated, if an argument has been provided, the argument is passed to this method as an integer. If no argument provided in command line, then this method is called with the maximum amount of physical frames allowed by default for convenience (especially if testing/using with an IDE.) This method calls `validFrameCount(int count)` to see if the count given is in the valid range. If not, then the default amount is used.

`validFrameCount(int count)`
This simply returns a Boolean on whether count is within the valid physical frame count range (0 – 7).

At this point, the frame count has been set and is displayed as confirmation to the user. Next, the referenceString and frames class variables are initialized as empty ArrayList and LinkedList, respectively. Finally, `promptUser()` is called.

`promptUser()` – DemandPagingSimulator in Action

`promptUser()`
This method displays the menu of options, captures the user's selection and calls the `processSelection` method with this selection. This is all in a try/catch block to catch any invalid input given by the user. The try/catch also has a finally block which recursively calls this method to keep it in a loop until the user explicitly exits the program by choosing the exit option.

`processSelection(int selection)`
This method interprets the selection made by the user, which is passed as the selection parameter and calls the corresponding method. Before any method is called, this method checks if the method to be called will require the `referenceString` class variable to be set. Any option chosen after option 2 will require the `referenceString`. In the event that the `referenceString` is not set and an option requiring it is selected, an `IOException` is thrown. An `IOException` is also thrown if the user chooses an invalid option.

Option 0 - Exit

If 0 is selected then the program exits.

Option 1 – Read Reference String

If 1 is selected then `getUserRefString()` is called followed by the `initVars()` method.

`initVars()`
Called by option 1 and option 2, which is after a `referenceString` has been created. This method initializes three class variables with the size of the `referenceString`. This is done so that there can be `null` or empty values in each for the output table. The variables are:

1. `faults`: an array list that keeps track of the faults during a paging algorithm
2. `victims`: an array list that keeps track of victims during a paging algorithm
3. `snapShots`: an array list which saves the state of the physical frames, the class variable `frames`, at each step of a paging algorithm.

`getUserRefString()`

This method clears the current `referenceString` if it exists and saves a new one based on user input. Each character is evaluated to ensure that the `referenceString` provided by the user is valid (contains only whitespace and/or integers.) It calls itself recursively after it outputs an error message if an invalid character is detected. Therefore, this method will continually prompt the user for a `referenceString` until a valid one is provided.

Option 2 – Generate Reference String

If option 2 is selected then the `getRandomRefString()` method is called.

`getRandomRefString()`
This method clears the current `referenceString` if it exists. Then the user is prompted for a preferred length for the reference string, which will be randomly generated. `Math.floor` and `Math.random` are used to generate the random reference string.

Option 3 – Display Current Reference String

Option 3 calls `displayReferenceString()` which simply prints the `referenceString` to output.

Options 4 - 7 are the four paging algorithms. All four of the methods start by calling `reset()` which clears the class variables `faults`, `snapShots`, `frames`, and `victims`. It also calls the `initVars()` method reference above.

Option 4 - `simFIFO()`

This checks each value in the `referenceString`, first to see if the value is present in `frames`. If the value is not in `frames`: there is a page fault and a `TRUE` value is added to `faults`. Then the value is added to the beginning of `frames`. Then the size of `frames` is evaluated after the value is added. If `frames` has too many values, the last one must be removed since this is a first-in-first-out algorithm. The removed value is saved to the `victims` array.

If the value already existed in `frames`, then none of the above code will execute. Since the value is already in `frames`, there is no victim - `null` added to `victims` - and there is no fault - `FALSE` added to `faults`.

Next, a copy of `frames` is saved and added to `snapShots`. Also, the output table is displayed and the user is prompted to continue. All of this occurs for each value in the `referenceString`.
After the `referenceString` is evaluated, the final output table is displayed during the last loop and after the loop, the total amount of faults are displayed.

Options 5 - 7 each call `simOPT`, `simLRU`, and `simLFU`. Each of these methods behave similarly to the `simFIFO()` method in all but one way in which all of these algorithms differ. The difference is how each of these methods choose the victim frame.

Option 5 - `simOPT()` and `findVictimOPT(int index)`

The method, `simOPT()` is called by option 5. This method calls `findVictimOPT()` to choose the next victim frame. The index passed as the parameter is the current index in the `referenceString` - the value currently being evaluated.

`findVictimOPT(int index)`
This method implements the optimal algorithm which looks at the values not evaluated and chooses next victim frame based on which of the values currently in `frames` will not be used or will be used last among the other values in `frames`. It does this by first making a copy of `frames` so that each time a value in `frames` is found, it can be removed from the copy until only one is left - which will then be the victim frame. Only values after the current value in `referenceString` are evaluated.

In the event that there is no optimal choice, then the top or first value in `frames` is chosen. This may occur if none of the values in `frames` occur again in the reference string or if the last value in the reference string is being evaluated and a page fault occurs, for example.

Option 6 – `simLRU()` and `findVictimLRU(int index)`

The method, `simLRU()` is called by option 6. This method calls `findVictimLRU()` to choose the next victim frame. The index passed as the parameter is the current index in the `referenceString`.

<u>`findVictimLRU(int index)`</u>
This method implements the least recently used algorithm which chooses its next victim based on which of the values in `frames` was last used less recently than the other values currently in `frames`. This is done by making a copy of `frames` which we can remove potential victims from until the victim is left as `referenceString` is traversed from the index before the current value all the way to the first value in the `referenceString`. As each value within the `framesCopy` is found, the value is removed. Once only one value remains or the rest of the `referenceString` is traversed, the remaining value is returned as the victim or one of the remaining values is returned as the next victim, respectively.

Option 7 - `simLFU()` and `findVictimLFU(int index)`

The method, `simLFU()` is called by option 7. This method calls `findVictimLFU()` to choose the next victim frame. The index passed as the parameter is the current index in the `referenceString`.

<u>`findVictimLFU(int index)`</u>
This method implements the least frequently used algorithm which chooses the next victim among the values in frames based on which value has occurred the least in the `referenceString` so far. This is done by using an int array names `frameFrequencies` to keep track of how many times each value occurs in the `referenceString`. Each index in the array corresponds with a value in the reference string. The value held at the index of `frameFrequencies[i]` is the number of occurrences of the value `i` in `referenceString`. The `referenceString` is iterated from the beginning up to the current index of the `referenceString` (not beyond -- only the OPT algorithm gets to see the future.)

The first frame in frames is the default victim if there is not a least frequently used value in `frames`. For each value in `frames`, the corresponding value in `frameFrequencies` is checked against the current least frequently used value, the local variable `lfu`, and also checked against the number of occurrences of each value in `frames`. The `lfu` variable is updated each time a less frequently occurring value is found until all comparisons are made and finally `lfu`, the victim frame, is returned.

Other methods

`findIndex(int victim)`

This method is used to find the index of the chosen victim in `frames`. Since `frames` is a linked list, a design choice based on the FIFO algorithm, there needed to be a way to find the index of a specified value in `frames`. Since `frames` can never exceed a size of seven, this is an acceptable work around with this data structure as any hit to performance is negligible.

`displayTable()`

Displays the output table to the console/command line by calling methods to display the `referenceString`, `frames`, `faults`, and `victims`.

`displayFrames()`

This method displays each physical frame and the value that each physical frame has held at each iteration of the `referenceString`. This requires two loops, the outer loop to set up the row for each physical frame. The inner loop is then used to generate the index, `j`, to loop through the `snapShots`, each of which is a value for each physical frame value. The index, `i`, is used to obtain the physical frame value in each of the `snapShots`.

Since this is somewhat confusing, if one is viewing the output table, the `i` index corresponds to each physical frame row. The `j` index corresponds to each snap shot in `snapShots` or to each value in `referenceString`. For example, if `j` = 2 and `i` = 3 then: `snapShots[2][3]` which would be the third value that the fourth physical frame held.

`displayFaults() & displayVicitims()`

These methods display the `faults` and `victims`, respectively, formatted for the output table.

It should be noted that the `displayTable()` method uses the same method as option 3 to print out the reference string.

`displayTotalFaults()`

This method adds up the total number of `TRUE` values in the `faults` class variable and displays this number as the total number of page faults. This is called after each of the algorithms is run.

This concludes the report and documentation on the DemandPagingSimulator.