CSE312 Homework2 Report

Yeşim Yalçın-200104004094

1-) Problem Definition and My Solution Approach

A-) Virtual Memory Creation

The problem was to design a virtual memory and in this virtual memory, demonstrate FIFO, FIFO Second Chance, LRU page replacement algorithms on sorting algorithms Bubble Sort, Quick Sort, Insertion Sort.

To be able to solve this problem, I firstly had to create the virtual memory in which I could use page replacement algorithms. **I followed these steps to do that:**

- **Create a RAM** which had operations to save data in blocks.
- **Create a Disk** which had operations to save data in blocks.
- Have a way to know **which blocks are free, which are full, a way to determine a block's address quickly**.
- **Have two page tables**. One for ram, one for the disk. Design the page tables in a way that the pages hold enough info **to do required virtual memory lookup operations.**
- **Have a memory management unit (MMU)** to do all the management of RAM, Disk and page table operations.

1-) Creating a RAM

To create the RAM I used the class **MemoryManager**. This class was introduced in the 16th video. However, I edited this class according to my needs.

MemoryManager class uses MemoryChunks to hold block information. Thanks to MemoryChunk class, we know all the needed details of a block.

- It holds the **size of the block** to be able to properly iterate through RAM.
- It holds a variable named **allocated** to know if a block is free.
- It holds pointers to the **next and previous block** to be able to iterate through RAM easily.
- It holds the **frameNo** of the block. This frameNo is the **physical address** of the block.

I added a page table to this class. **This page table is responsible to do conversion of virtual addresses to physical addresses in RAM**. However, there was a problem. MemoryManager class gives opportunity to have different sized blocks. If there are different sized blocks, then we can never know how many blocks RAM can hold. **In reality, the actual page sizes and page frame sizes are always equal**. Therefore we know how many blocks RAM can hold and have that many pages in the ram page table. **In our case, we cannot know how many pages we can put into the page table.** That's why in my program, I set the block size of both RAM and disk as **20*1024**. 20 is because of the sizeof MemoryChunk. 1024 is to symbolize KB.

In my program RAM size can be changed according to the parameters of MMU constructor. However, block sizes cannot be changed. **Page table sizes are calculated according to the given RAM size and predetermined block size**.

2-) Creating a Disk
To create the Disk, I used the class **MemoryManager2**. **It operates exactly the same as MemoryManager class**. Only difference is that their sizes, page tables, maximum page table page amounts and therefore pointed addresses etc are different.

3-) Block Operations
As I have mentioned before, thanks to the **MemoryChunks** I insert for each block in RAM/Disk, I know all required information related to a block.

4-) Page Tables
As I have mentioned before, I have page tables in both RAM and Disk. **The one in the RAM is responsible of the addresses in RAM. The one in the disk is responsible of the addresses in Disk.** Their sizes are calculated according to RAM/Disk sizes and the predetermined block size.

Page Tables hold pages. This pages have:

- An **index** variable to hold the virtual address
- A **frameNo** variable to hold the physical address
- A **no** variable to hold the time they were inserted into RAM. (For the ones in the disk, this one does not matter)
- A **referenced** variable to show is referenced or not. (1 if referenced, 0 if not) (For the ones in the disk, this one does not matter)
- A **counter** to hold the time they were last referenced. (For the ones in the disk, this one does not matter.

Index and frameNo are held for the virtual address to physical address conversion. The rest are held for the page replacement algorithms. No is used in FIFO, referenced is used in Second Chance, counter is used in LRU.

5-) MMU
To create the MMU, I used the class MMU. This class has:

- A **MemoryManager** class for **RAM**
- A **MemoryManager2** class for **Disk**
- **fifo, chance, lru** variables to know which one to use (Whichever has the value 1, it is used)
- A **totalCounter** which is incremented by one whenever there is a RAM or Disk access. This is used in LRU to determine the time counter.
- A **resetRefCounter** which is incremented by one whenever there is a RAM or Disk access. This is used to reset the referenced bits after a while in second chance algorithm. It is reseted when it reaches 10.
- **totalHit and totalMiss** variables to count hits and misses.
- **hitCounter and missCounter** to count hits and misses in each determined interval. They are reseted when the resetRefCounter reaches 10.

MMU is responsible of controlling all operations related to RAM and Disk.
When it is initialized, **it firstly allocates enough memory for both RAM and Disk**. However, if the given sizes for RAM and Disk exceeds the OS Limit (64MB), it uses default sizes for RAM and Disk (4 MB RAM, 16MB Disk). Afterwards, **it calculates the maximum**

block sizes of RAM and Disk then sets up their page tables. It also sets the given algorithm to use whenever there is a page replacement needed.

MMU has operations to insert and item to the storage also get an item from the storage. To insert an item, it firstly searches the RAM. If there is enough space in RAM, it puts the item into the RAM. If there is not enough space in RAM, the item is put inside the Disk. Their page tables are also updated whenever an item is placed.

To be able to retrieve an item from the storage, a virtual address is needed as an input. The given virtual address is firstly searched in the page table in RAM. If it is in RAM, the page frame number is retrieved. The page frame is searched and then the item is found in it. **However, if the virtual address cannot be found in RAM then it is a page fault.** The virtual address is searched inside the disk and the page frame number is retrieved. The page frame is searched and then the item is found in it.

Whenever a page fault happens, the page in the disk will be brought to RAM from disk. However this means RAM is full. That’s why **a page replacement must happen.** Previously selected page replacement algorithm is applied and one page is brought to RAM, the other one is put into the Disk.

Moreover, whenever a page is referenced, their referenced bit is set to 1 during these operations. Their counter is updated as well. If a page replacement happens, no variable of pages are also updated accordingly.

B-) Page Replacement Algorithms

I do the page replacements in MMU therefore I implemented them inside MMU class.

1-) FIFO

For this algorithm, I have a function named **fifoOperation**. In thisI funtion, I check no variables of pages inside ram page table. **I choose the page with the smallest no** because having the smallest no means that page entered ram page table the earliest. From this function, I retrieve the index of this page and then swap this page of ram page table with the faulted page in the disk with **pageReplacementOperation** function.

2-) Second Chance FIFO

For this algorithm, I have a function named **chanceOperation**. This function is very similar to fifoOperation. **However, when it founds the page to choose, it looks if their referenced bit is 1. If it is 1, it makes it 0 and tries to find another page to choose**. From this function, I retrieve the index of this page and then swap this page of ram page table with the faulted page in the disk with **pageReplacementOperation** function.

3-) LRU

Fort his algorithm, I have a function named **lruOperation**. **This function searches for the page which has the smallest counter**. Because having the smallest counter means, that page was referenced the earliest. From this function, I retrieve the index of this page and then swap this page of ram page table with the faulted page in the disk with **pageReplacementOperation** function.

C-) Page Replacement Testing

To test my virtual memory and page replacement algorithms, I implemented three requested sorting algorithms. **My program works this way:**

- **Makes the user choose a page replacement algorithm**. User should enter 0 for FIFO, 1 for Second Chance, 2 for LRU.
- **Makes the user choose a sorting algorithm**. User should enter 0 for Bubble Sort, 1 for Quick Sort, 2 for Insertion Sort.
- **Checks if the user input is valid**. If the user input is not valid then the default 0, 0 is used as inputs. Meaning FIFO and Bubble Sort will be applied.
- Sets up the array.
- **Inserts array elements into RAM and Disk with MMU operations.**
- **During insertion if array size exceeds both RAM and Disk, exception occurs.**
- **Initial array is printed** out from the storage. Making sure insertion was properly done.
- Array size is calculated in respect to the RAM size. Here RAM size is assumed by it's max block count.
- **If the calculated size is less than %100 then it means no Disk will be required. Exception happens** because we are trying to test page replacement algorithms. If there is no Disk usage, then there is no page replacement.
- Given algorithm is applied. For these algorithms, I have the functions bubble, quick and insertion.
- **During these algorithms, all array element accesses are done through the storage. Therefore some page replacements etc are demonstrated.**
- **Outcome array is printed** out to show that the sorting is done properly while having all MMU operations.
- **Miss/Hit etc outputs are printed.**

D-) Miss/Hit Outputs

I print out all info related to:

- Total misses,
- Total hits,
- Miss rate,
- Hit rate,
- Total pages written to the disk count,
- Total disk accesses count

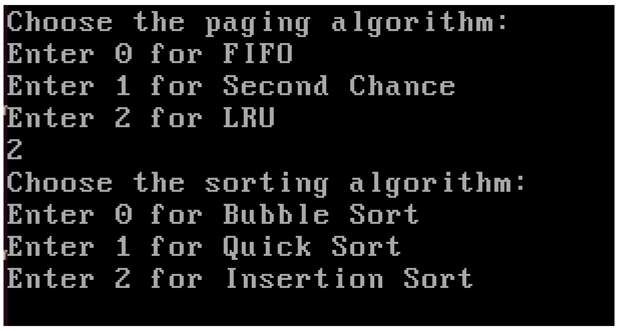
I do this at the end of the program. However, I also do this during the program run as well. I use the **resetRefCounter** inside the MMU as a timer to print these information.

E-) Exceptions I Solved

- I never let the given RAM and Disk size exceed the maximum OS space (64MB).
- I never let the given array size overflow my storage space.
- I never let to operate with an array smaller than RAM size.
- I never let the user input invalid input for page replacement algorithms and for sorting algorithms.
- I have equal blocksizes so that there is never ambiguity for page table sizes.
- I update page frame data if an array index content is changed. This is essential because during sorting, there is a lot of swapping between array indexes.

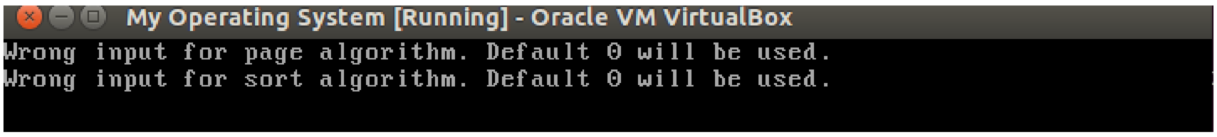
F-) Tests

1-) User Selection For Algorithms



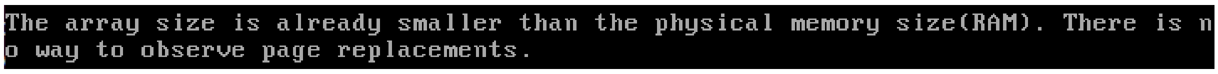
```
Choose the paging algorithm:
Enter 0 for FIFO
Enter 1 for Second Chance
Enter 2 for LRU
2
Choose the sorting algorithm:
Enter 0 for Bubble Sort
Enter 1 for Quick Sort
Enter 2 for Insertion Sort
```

2-) If User Inputs Wrong Numbers



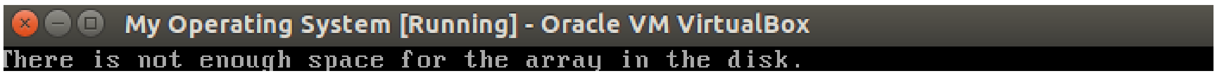
```
Wrong input for page algorithm. Default 0 will be used.
Wrong input for sort algorithm. Default 0 will be used.
```

3-) If the Array Size is Too Small



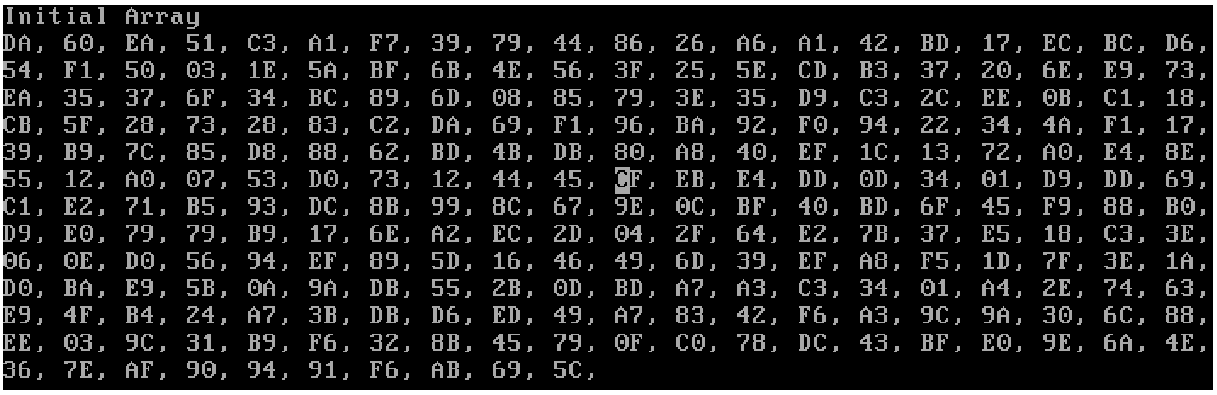
```
The array size is already smaller than the physical memory size(RAM). There is n
o way to observe page replacements.
```

4-) If the Array Size if Too Big



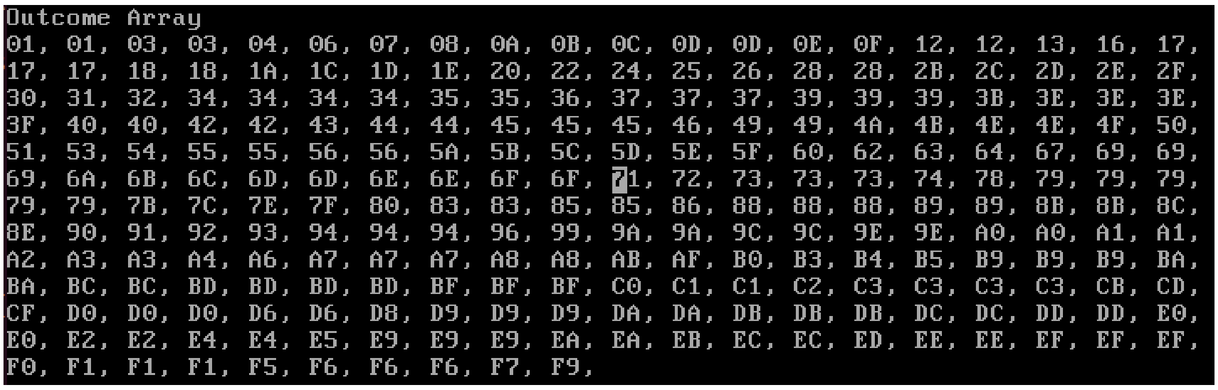
```
There is not enough space for the array in the disk.
```

5-) Initial Array



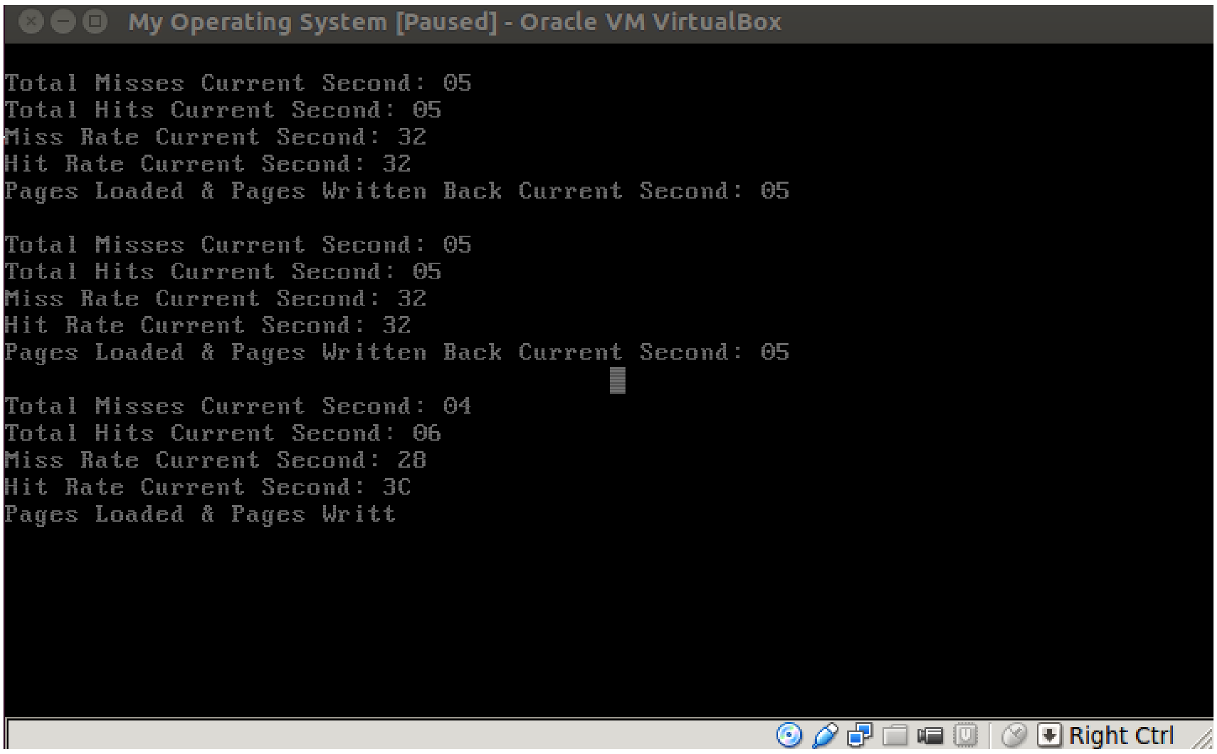
```
Initial Array
DA, 60, EA, 51, C3, A1, F7, 39, 79, 44, 86, 26, A6, A1, 42, BD, 17, EC, BC, D6,
54, F1, 50, 03, 1E, 5A, BF, 6B, 4E, 56, 3F, 25, 5E, CD, B3, 37, 20, 6E, E9, 73,
EA, 35, 37, 6F, 34, BC, 89, 6D, 08, 85, 79, 3E, 35, D9, C3, 2C, EE, 0B, C1, 18,
CB, 5F, 28, 73, 28, 83, C2, DA, 69, F1, 96, BA, 92, F0, 94, 22, 34, 4A, F1, 17,
39, B9, 7C, 85, D8, 88, 62, BD, 4B, DB, 80, A8, 40, EF, 1C, 13, 72, A0, E4, 8E,
55, 12, A0, 07, 53, D0, 73, 12, 44, 45, CF, EB, E4, DD, 0D, 34, 01, D9, DD, 69,
C1, E2, 71, B5, 93, DC, 8B, 99, 8C, 67, 9E, 0C, BF, 40, BD, 6F, 45, F9, 88, B0,
D9, E0, 79, 79, B9, 17, 6E, A2, EC, 2D, 04, 2F, 64, E2, 7B, 37, E5, 18, C3, 3E,
06, 0E, D0, 56, 94, EF, 89, 5D, 16, 46, 49, 6D, 39, EF, A8, F5, 1D, 7F, 3E, 1A,
D0, BA, E9, 5B, 0A, 9A, DB, 55, 2B, 0D, BD, A7, A3, C3, 34, 01, A4, 2E, 74, 63,
E9, 4F, B4, 24, A7, 3B, DB, D6, ED, 49, A7, 83, 42, F6, A3, 9C, 9A, 30, 6C, 88,
EE, 03, 9C, 31, B9, F6, 32, 8B, 45, 79, 0F, C0, 78, DC, 43, BF, E0, 9E, 6A, 4E,
36, 7E, AF, 90, 94, 91, F6, AB, 69, 5C,
```

6-) Sorted Array



```
Outcome Array
01, 01, 03, 03, 04, 06, 07, 08, 0A, 0B, 0C, 0D, 0D, 0E, 0F, 12, 12, 13, 16, 17,
17, 17, 18, 18, 1A, 1C, 1D, 1E, 20, 22, 24, 25, 26, 28, 28, 2B, 2C, 2D, 2E, 2F,
30, 31, 32, 34, 34, 34, 34, 35, 35, 36, 37, 37, 37, 39, 39, 39, 3B, 3E, 3E, 3E,
3F, 40, 40, 42, 42, 43, 44, 44, 45, 45, 45, 46, 49, 49, 4A, 4B, 4E, 4E, 4F, 50,
51, 53, 54, 55, 55, 56, 56, 5A, 5B, 5C, 5D, 5E, 5F, 60, 62, 63, 64, 67, 69, 69,
69, 6A, 6B, 6C, 6D, 6D, 6E, 6E, 6F, 6F, 71, 72, 73, 73, 73, 74, 78, 79, 79, 79,
79, 79, 7B, 7C, 7E, 7F, 80, 83, 83, 85, 85, 86, 88, 88, 88, 89, 89, 8B, 8B, 8C,
8E, 90, 91, 92, 93, 94, 94, 94, 96, 99, 9A, 9A, 9C, 9C, 9E, 9E, A0, A0, A1, A1,
A2, A3, A3, A4, A6, A7, A7, A7, A8, A8, AB, AF, B0, B3, B4, B5, B9, B9, B9, BA,
BA, BC, BC, BD, BD, BD, BD, BF, BF, BF, C0, C1, C1, C2, C3, C3, C3, C3, CB, CD,
CF, D0, D0, D0, D6, D6, D8, D9, D9, D9, DA, DA, DB, DB, DB, DC, DC, DD, DD, E0,
E0, E2, E2, E4, E4, E5, E9, E9, E9, EA, EA, EB, EC, EC, ED, EE, EE, EF, EF, EF,
F0, F1, F1, F1, F5, F6, F6, F6, F7, F9,
```

7-) Example Outputs



```
Total Misses Current Second: 05
Total Hits Current Second: 05
Miss Rate Current Second: 32
Hit Rate Current Second: 32
Pages Loaded & Pages Written Back Current Second: 05

Total Misses Current Second: 05
Total Hits Current Second: 05
Miss Rate Current Second: 32
Hit Rate Current Second: 32
Pages Loaded & Pages Written Back Current Second: 05

Total Misses Current Second: 04
Total Hits Current Second: 06
Miss Rate Current Second: 28
Hit Rate Current Second: 3C
Pages Loaded & Pages Writt
```

Important Note: The function **printfHex** can print up to 250 in decimal. Therefore if there is more than 250 hits or misses, the output number can be wrong. The main reason why I chose 250 as array size was this. However, it is still possible to have more than 250 hits or misses in total.