CSE 312
OPERATING SYSTEMS

FINAL PROJECT

PART 1

ÖMER ÇEVİK
161044004

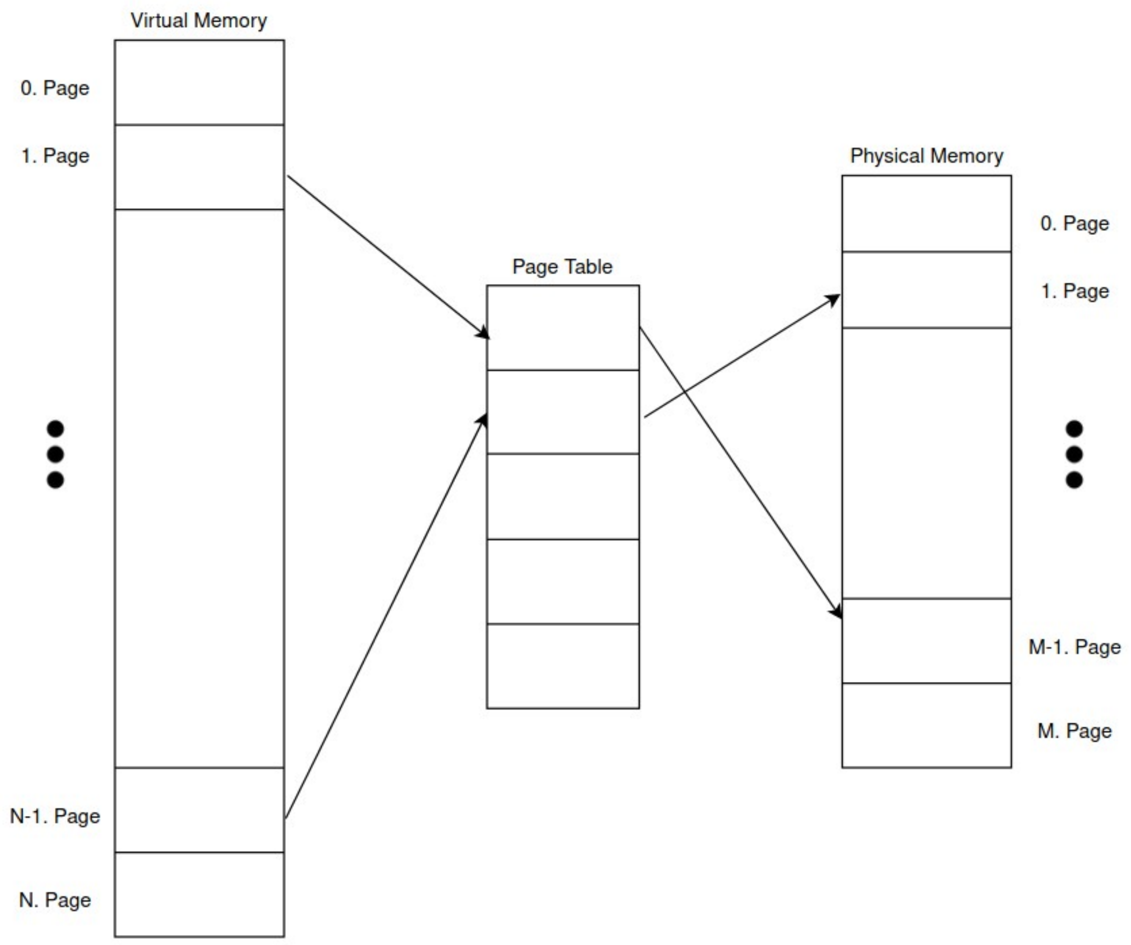
1. Virtual Memory Structure

To declare the virtual memory structure there is something to say firstly, the virtual memory is greater than physical memory and in main function there are 4 different threads which are sorting these virtual memory as a C array that all quarter of virtual memory is sorted by different 4 sorting algorithms: bubble, quick, merge, index. First quarter is going to be sorted by *Bubble Sort*, second quarter is going to be sorted by *Quick Sort*, third quarter is going to be sorted by *Merge Sort* and fourth quarter of virtual memory is going to be sorted by *Index Sort* as page by page.

There is connection between physical memory using a page table as bridge. Page frame size of virtual memory and physical memory is same. The page table also keeps pages and frame sizes of the page table are same with physical and virtual memories.

Each page of virtual memory is going to point a page table entry and each page table entry is going to point the physical memory page. All pages can not be in page table so there must be changing between pages which is be provided by page replacement algorithms.

In the beginning of the program, there is a checker method (*checkConstraints()*)defined in *VirtualMemory* class (virtualMemory.h) which checks the allocation policy is *local* or *global*. If the allocation policy is ***local*** then each sorting thread can reach to their quarter indexes in virtual memory. If the allocation policy is ***global*** then each sorting thread can reach to all indexes of virtual memory.



1. Virtual Memory Structure

2. <u>Page Table Structure</u>

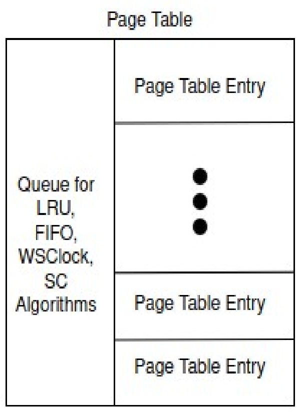
In my page table structure, I kept a class named *PageTable* (pageTable.h), that keeps page table entries which is represented by an object array of *PageTableEntry* class (pageTable.h) and these array’s size is maximized by *total pages size of physical memory* / *page frame size (ex: 32 / (4096/1024) = 32 / 4 = 8)*.

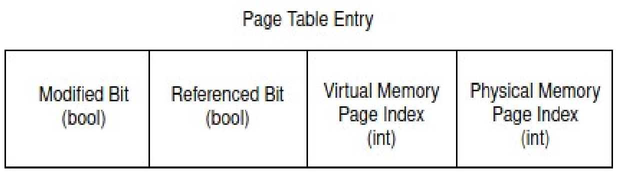
The *PageTable* class also keeps a queue which is used for FIFO, LRU, Second Chance and WSClock Page Replacement Algorithms. That queue keeps the page’s indexes as time ordered for LRU and WSClock algorithms. For FIFO and Second Chance algorithms it keeps as replaced order. So if FIFO algorithm is used then the first index at queue is going to be removed from page table. For other algorithms that kind of structure will be provided using that queue. That queue is declared as C++ integer *vector*.

Also there is a variable named *wsClockPointer* which represents the current page as a pointer for WSClock page replacement algorithm.

Finally, there is a variable named *tableCounter* that represents the size for page table entry size.

In *PageTableEntry* class, there are some special bits which are going to be used by page replacement algorithms: modified bit (NRU), referenced bit (Second-Chance, NRU, LRU and WSClock). Also there is the page index of virtual memory which points to it. Page replacement algorithms are going to be detailed in next topic.




Page Table Entry
Modified Bit (bool)
Referenced Bit (bool)
Virtual Memory Page Index (int)
Physical Memory Page Index (int)

2. Page Table Structure

3. Page Replacement Algorithms

There are five different page replacement algorithms used in that program: FIFO, Second Chance, LRU, NRU and WSClock. Each of them uses different bits or structure to provide page replacement.

FIFO

The FIFO (First In First Out) algorithm is a page replacement algorithm what uses the knowledge of which page is came first to page table and then it will be out from page table when it is full. The basic idea and basic structure of page replacement algorithm is provided by using a queue of page indexes. The FIFO algorithm is defined in *PageReplacement* class (pageReplacement.h) and implemented using the queue in page table (pageReplacement.cpp). It doesn’t check any bits just removes the front of queue and replace the new page into physical memory.

Second Chance

The Second Chance algorithm is a page replacement algorithm what uses the knowledge of which page is came first to page table and the referenced bit is *true* then it will be ignored to replace just for once and next element of queue will be replaced. Second Chance algorithm is defined in *PageReplacement* class (pageReplacement.h) and implemented using the queue in page table (pageReplacement.cpp). If the referenced bit is **not** *true* then it will be replaced as FIFO structure.

LRU

The LRU (Least Recently Used) algorithm is a page replacement algorithm what uses the knowledge of which page is least recently used in page table. There are many ways to implement that kind of structure. I used the queue to implement least recently used page inserted in queue. It replace the location of pages when they got the hit as end of the queue. The least recently used page will be represented in zero indexed page of queue. LRU algorithm is defined in *PageReplacement* class (pageReplacement.h) and implemented using the queue in page table (pageReplacement.cpp).

NRU

The NRU (Not Recently Used) algorithm is a page replacement algorithm what uses the knowledge of which page is **not** recently used in page table. It checks whole page table which has referenced bit as zero and modified bit as zero then it will be replaced. Otherwise it will replace the page in zero-indexed page of page table. NRU algorithm is defined in *PageReplacement* class (pageReplacement.h) and implemented using the queue in page table (pageReplacement.cpp).

WSClock

The WSClock (Working Set Clock) algorithm is a page replacement algorithm what uses the knowledge of which page is least recently used in page table. There are many ways to implement that kind of structure as LRU. I used the queue to implement least recently used page inserted in queue. It replace the location of pages when they got the hit as end of the queue. The least recently used page will be represented in zero indexed page of queue and if the least recently used page's referenced bit is *true* then it will be have chance to not replace and checks for next least recently used page. That will be in a loop as a clock. WSClock algorithm is defined in *PageReplacement* class (pageReplacement.h) and implemented using the queue in page table (pageReplacement.cpp).

4. <u>Part 2</u>

In this part, the flow of part2 algorithm will be explained. First of all, in main method (part2.cpp) the *VirtualMemory* class (virtualMemory.h) object is created as pointer to the object. Also there is a *Sorter* class (sorter.h) which is going to sort the virtual memory page by page. The *set()* and *get()* functions are defined and implemented in sorter pages (sorter.h and sorter.cpp). Each sorting algorithms uses the *get()* and *set()* functions to swap values in virtual memories. But there is a critical region between threads so I used a *mutex* to not allow other threads' reach to virtual memory. That creates a safe way to sort. In main function all threads are sorting with their algorithms and waits in *join()* to finish their job. After finishing sorting main function closes the disk file, checks the all sorts are sorted well, prints the statistics and deletes the virtual memory object.

All sorting algorithms are declared in sorter.h and implemented in sorter.cpp. All page replacement algorithms are declared in pageReplacement.h and implemented in pageReplacement.cpp. All page table algorithms and design are defined in pageTable.h and implemented in pageTable.cpp. All physical memory design and algorithms are defined in physicalMemory.h and implemented in physicalMemory.cpp. All virtual memory definitions and algorithms are defined in virtualMemory.h and implemented in virtualMemory.cpp.

In *get()* and *set()* functions, I used *pageTablePrintInt* to accessing physical memory in page fault counted as frame size. Because accessing memory is not only for one value in page fault. In *hit*, it counts only as one. Because only one memory accessing is succeed.

5. <u>Part 3</u>

In this part, the flow of part3 algorithm will be explained. First of all main method (part3.cpp) the *VirtualMemory* class (virtualMemory.h) object is created as pointer to the object for every step of changing frame size. Frame size is initialized as 2 in the beginning. Then it grows as multiplying by 2 for each iteration until 4096. 4096 is the number that the optimal frame size while virtual memory size 128K and physical memory size 16K. The page table size is defined as 128K / 16K = 8 to simulate an optimal sized page table in beginning. For 4096 frame size the page table size is 4. Because the physical memory page size is also 8 when frame size is 4096. It keeps approximately 5 – 10 minutes to run the simulation for each frame size. Then it prints the results to console for each sorting algorithm.

In **bonus part**, I created a loop for each page replacement algorithm and a nested loop to fill virtual memory with random values 10 times and sorted using that 4 threads and saved their each page replacement sizes in vectors. Then evaluated the averages of that 10 page replacement sortings of each algorithm and printing to console. It also takes approximately 10 minutes. It also prints the which page replacement is best for each sorting algorithms with their page replacements sizes.