Gebze Technical University

Department of Computer Engineering

CSE312/CSE504

Operating Systems

Spring 2020

Final Exam Project

Instructor: Yusuf Sinan AKGÜL

Student Name: Can BEYAZNAR

Student No: 161044038

Part 1

1) Structure of Page Table Entry

I used 4 parameters in total for my page table entry. This structure starts at **line 19**.

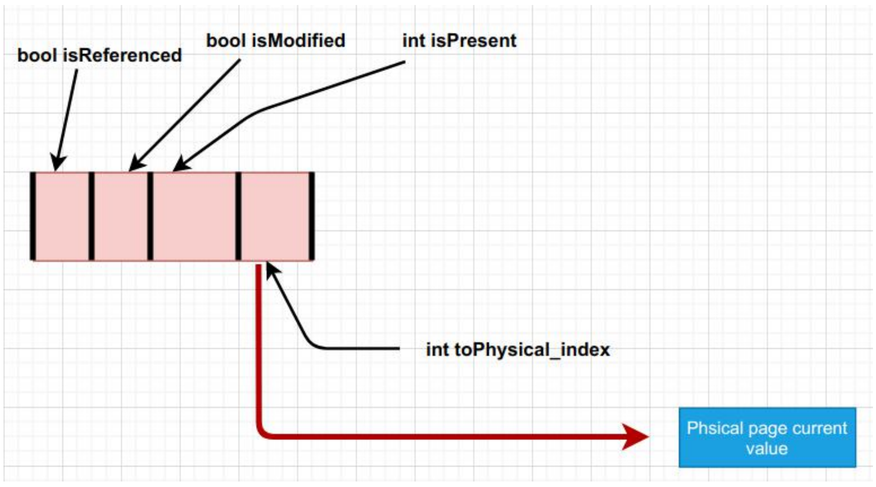
```
typedef struct _PageTableEntry
{
bool isReferenced;
bool isModified;
int isPresent;
int toPhysical_index;
} PageTableEntry;
```

bool isReferenced: Checks whether the page is referenced. This parameter is required for page replacement algorithms. If the page is read or printed by someone else, isReferenced is true.

bool isModified: the second parameter that follows the page usage. When a page is written, its modified bit is changed and set to true. This parameter is used in page replacement algorithms as in isReferenced.

int isPresent: Indicates whether the page in virtual memory is full. Thanks to this parameter, I find out whether the page points to pyhsical memory.

int toPhysical_index: This parameter holds the index of the physical memory that the page points to. If the page does not point to physical memory, its value is -1. This parameter is used to update data in page replacement and to be able to backup.



2) Problem Solution

I developed a simple algorithm to be able to quickly read and write data from disk. In order to be able to advance at **O(1)** time with fseek function in the file, I allocated the same size to each number in the file. In other words, each data takes a total of 17 bits in the file. For 16 bit integer, 1 bit for new line. Before I write the number that I generate randomly to the file, I add 0 per bit so that the data to be sent is 16 bits. Thus, I can find out exactly where the data I am looking for is in the file. I can explain the get function according to the structure of my file.

A) Implementation of get/set methods

In the **get** method **(line 639)**, I first check the tname. If tname fill or check, data is taken from the disk file according to the index given by the user. This condition works only at the beginning of the code and after sort algorithms are finished.

If tname is "quick", "bubble", "merge", "index", there are 2 possibilities. If the isPresent value of the index given by the user is 0, page replacement algorithms work. If it is 1, the value in the physical memory pointed to by virtual memory is returned. So **return (PhysicalMem[virtualMem[index]->toPhysical_index]);**

Under the condition isPresent == 0, the input that the user entered initially is checked. The page replacement algorithm changes according to the given input. Page replacement algorithms will be explained in detail later in the report. Before making a page replacement, the value is taken from the disk according to the index given by the user. Then, page replacement is done according to pageReplacement_type. And the value read from disk is returned after being written to physical memory.

I am increasing the memoryaccess_counter parameter (**for example line 1172**) whenever the get and set function is called. This parameter is used to check the pageTablePrintInt parameter entered by the user. If memoryaccess_counter is greater than or equal to pageTablePrintInt, the page table is printed and the information is printed. Then memoryaccess_counter parameter is assigned to 0.

The **set** method **(line 1214)** is simpler than the get function. First of all, I check if the given index is larger than the size of virtual memory. if it is greater than or equal, I quit the function. If it is small, I first calculate the position of the value that will be set according to the index parameter to write to the file (line 1220). Then I set the new value to the file.

Important note: the name of the set function in my program is _set. I had to change the name (because of the name) because the set function failed while writing my code.

B) Implementation of Page Replacement With NRU

To make a page replacement with NRU **(line 706)**, I kept a parameter of type vector int. While throwing random values in the fill section of the program, I pushed the indexes of virtual memories pointing to my physical memory into this vector. I keep the control and stage parameters for the loops that I will use in the NRU algorithm. I have a while loop that continues as long as control is 0. And I increase the stage parameter with each pass of this cycle. Stage parameter will be used for classification in NRU. Stage conditions are as follows.

Stage 0 -> R = 0, M = 0
Stage 1 -> R = 0, M = 1
Stage 2 -> R = 1, M = 0
Stage 3 -> R = 1, M = 1

Every time the stage increases, a for loop runs as far as vector's size. In this for world, the referenced and modified bits of virtual memory are checked according to the type of stage. For example, the stage parameter is 0 (line 723). The referenced and modified bits in the page table entry are checked. If both are 0, page replacement is done. While doing this operation, the old entry to be changed is backed up and its current data is set to its location in the file. Then the new entry is connected with its place in physical memory. And the index of the new entry to the vector is pushed. Thus, page replacement is completed. If R = 0 and M = 0 data are absent, the stage value increases and R = 0 M = 1 values are controlled.

C) Implementation of Page Replacement With FIFO

I used queue for FIFO **(line 867)** page replacement algorithm. As I explained in NRU, I push the index of virtual memory to queue as data is set to physical memory. In this algorithm, I am not looking at the R and M bits. I get the first data in the queue with the help of the front and find out which oldest virtual memory index is used. Then I do the page replacement process through this data. First of all, I back up the value that my old virtual memory index points to by restoring it to disk. Then I fill in the information of my page entry to be added and set the data I get from the disk to the free physical memory. And finally I push the new page entry index to queue and delete the old page entry index with the pop function from queue.

D) Implementation of Page Replacement With Second Chance

In SC page replacement algorithm **(line 913)**, I used queue as in FIFO algorithm. My goal is to get the oldest page entry from queue with the pop function, and check the R bit. If the R bit of the page entry is 0, I make the page replacement algorithm. If the R bit is 1, I assign the R bit of this page entry to 0 and push it back to the queue. Thus, by giving a second chance to the first entry, my cycle continues until I find a page entry with 0 of the R bit. In the page replacement section, I do the same operations that I have described in other algorithms. I save the old page entry data to disk. And I push the new incoming page entry to queue and write it to physical memory. And I'm getting out of the loop. This cycle continues until the page replacement is done. The worst case of this process is O (n).

E) Implementation of Page Replacement With LRU

In the LRU page replacement algorithm **(line 977)**, I used vector of type LRU_struct. LRU_struct is as follows.

```
typedef struct _LRU_struct
{
int day;
int hour;
int minute;
int second;

int id;
} LRU_struct;
```

In LRU page replacement algorithm **(line 977),** I used vector of type LRU_struct. LRU_struct is as follows.

A time-dependent comparison is required at the LRU. So before I add to this vectore in the fill section, I am calling the time and local functions, and I'm putting the second, minute, hour and day information of the page entry into the struct. In the page replacement algorithm, while selecting the entry I will change, I find the oldest entry in vector. Then I make my transactions over the oldest entry. To calculate this, I created a parameter called min_lru_entry. Then I make the comparison between min_lru_entry by assigning the data in vector to temp_lru_entry parameter in the for loop. If temp_lru_entry is older than min_lru_entry, I assign the temp_lru_entry parameter to the min_lru_entry parameter. Thus, the oldest page entry is found and page replacement is done. As explained in the previous algorithms, data to be deleted from physical memory is backed up on disk. The old page entry is deleted from the vector and the new incoming data is written to physical memory. And the addition time is written to the structure and pushed to the vector.

F) Implementation of Page Replacement With WSClock

In the WSClock algorithm **(line 1100)**, I used an int-type vector. Due to the structure of WSClock, I used vector in this algorithm as circular linked list. As in previous algorithms, I pushed the indexes of page entries added to physical memory to this vector. In the Page replacement algorithm, I created a while loop that continues until I find a page entry with a R bit of 0. In this loop I used the WSClock_currentpoint parameter in int value. This parameter is global and keeps track of which page entry was last. Thus, as described in the book, the WSClock algorithm becomes operational. By looking at the value in the WSClock_currentpoint index from the vector in the loop, the bit R is checked. If R bit is 0, page replacement is done and WSClock_currentpoint value is increased. If R bit is 1, R bit of the page entry is set to 0 and WSClock_currentpoint value is increased.

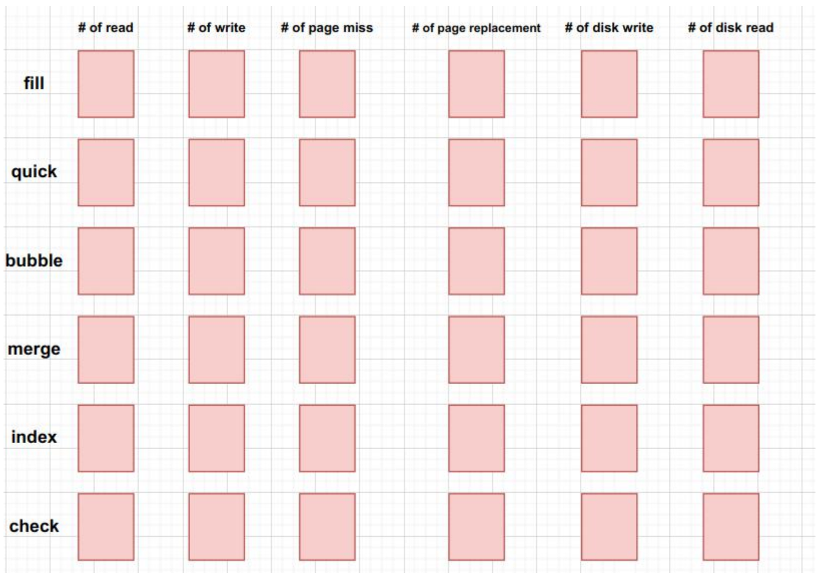
NOTE: If the WSClock_currentpoint value is greater than the size of the vector, WSClock_currentpoint is assigned to 0 for the clock to return to the beginning.

In case of R bit 0, page replacement is done. As in previous algorithms, backup and addition operations are done. And in vector, the new value is added (replaced) instead of the old value in the WSClock_currentpoint index. Thus, the program continues without changing Vector's size.

G) Backing Store

If we talk about the backing store part, I use 17 bits in total for each data in my disk file. I use it for 16 bit number and 1 bit for new line. for example, our data is 98765. This data is in the file as 0000000000098765\n. And thanks to this process, I can easily perform backup and installation operations. And so I know where the value I'm looking for is on disk. And I'm saving a lot of time. For backup, you can look at example **line 1123** on my code. For the loading part, you can look at **line 672**.

My statistic 2D array **(line 48)** is as follows.



	# of read	# of write	# of page miss	# of page replacement	# of disk write	# of disk read
fill						
quick						
bubble						
merge						
index						
check						

Note: Due to my lack of information, I was unable to take any action on allocation policy issues in my code.