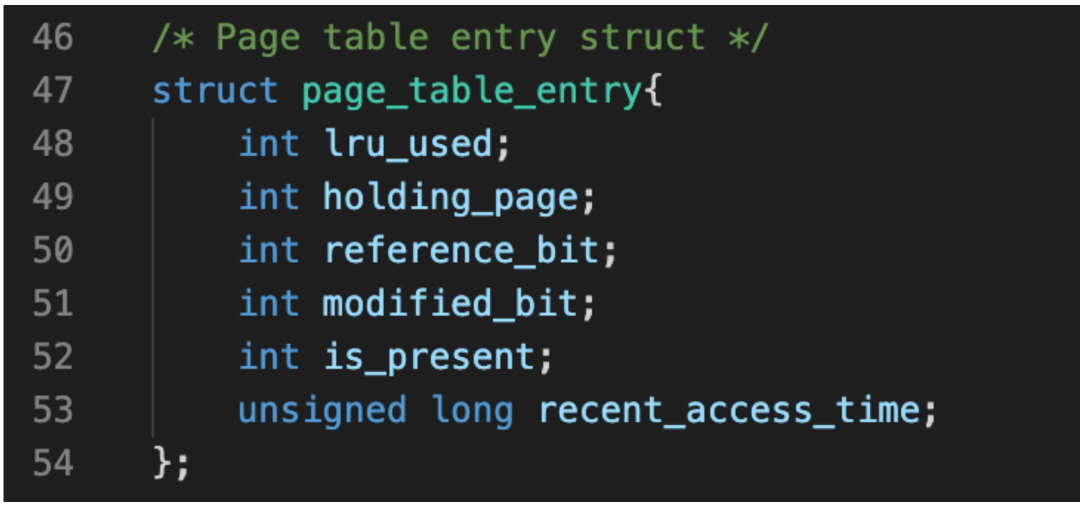
CSE312 FINAL PROJECT REPORT

Fatih Selim YAKAR - 161044054

CSE 312 FINAL PROJECT REPORT

• Page Table Structure

I generally used 2 different parallel data structures to express virtual and physical memory. A C integer array to hold addresses in Virtual memory and a struct array for Virtual memory page table entries. In this context, the C integer array is up to you as frame_size * memory_frames, while the parallel struct array is up to you as memory_frames.



```c
/* Page table entry struct */
struct page_table_entry{
    int lru_used;
    int holding_page;
    int reference_bit;
    int modified_bit;
    int is_present;
    unsigned long recent_access_time;
};
```

Figure 1 - Page table entry struct (sortArrays.c : 46)

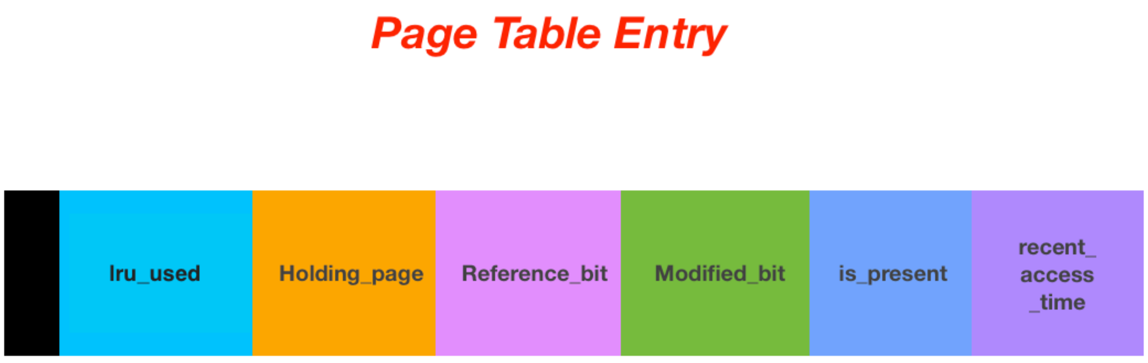


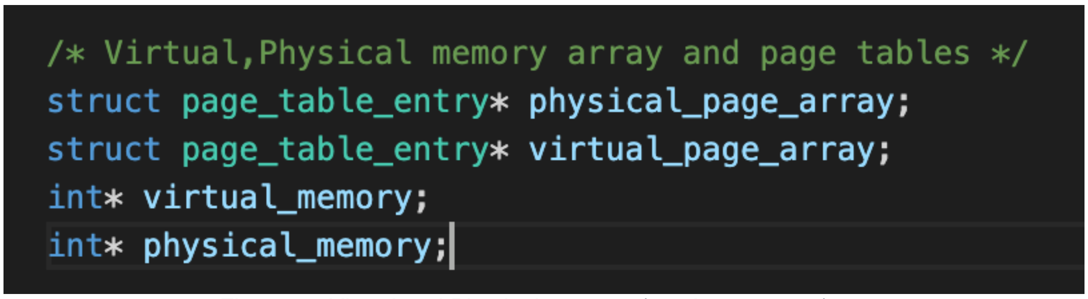
Figure 2 - Typical page table entry figure

Page table entry pieces

1. **Lru_used:** Counter to create a certain amount of time limit for least recently used algorithm.
2. **Holding_page:** Indicates which page the page in this index holds.
3. **Reference_bit:** It is the reference bit used by page replacement algorithms, such as Second Chance, WSClock, NRU. By checking this bit, page replacement conditions are determined.
4. **Modified_bit:** It is a modified bit used by page replacement algorithms, such as NRU. By checking this bit, page replacement conditions are determined.
5. **Is_present:** It is the check bit that indicates whether the page retained is present in physical memory.
6. **Recent_access_time:** Keeps the time to reach the page that was held last. It is used to find the oldest page in LRU and similar replacement algorithms.

• Overall Memory System

Generally, the virtual memory system consists of 3 main data structures. These are virtual memory consisting of integer array and structure, physical memory consisting of integer array and structure and disk, which is a file.



```
/* Virtual,Physical memory array and page tables */
struct page_table_entry* physical_page_array;
struct page_table_entry* virtual_page_array;
int* virtual_memory;
int* physical_memory;
```

Figure 3 - Virtual and Physical memory (sortArrays.c : 66)

Virtual memory actually contains as many addresses as the total number of random integers, and these addresses point to the indexes in disk and physical memory. The physical memory and disk are the random values of the virtual memory. If the is_present bit is not 1 in virtual memory, it means that the value of this page or index is not available in physical memory. If the is_present bit is 1, it means that it exists in physical memory.

Since virtual memory is logically much larger than physical memory, physical memory is not enough to show all the values of virtual memory. Uses a disk as secondary storage for insufficient situations. In other words, physical memory (integer array) is used as the first storage of virtual memory (integer array) and disk (file) is used as secondary storage. Physical memory is much faster but smaller. The disk is much larger but slower. Therefore, it is much more profitable to keep more accessed pages in physical memory.

VIRTUAL MEMORY SYSTEM

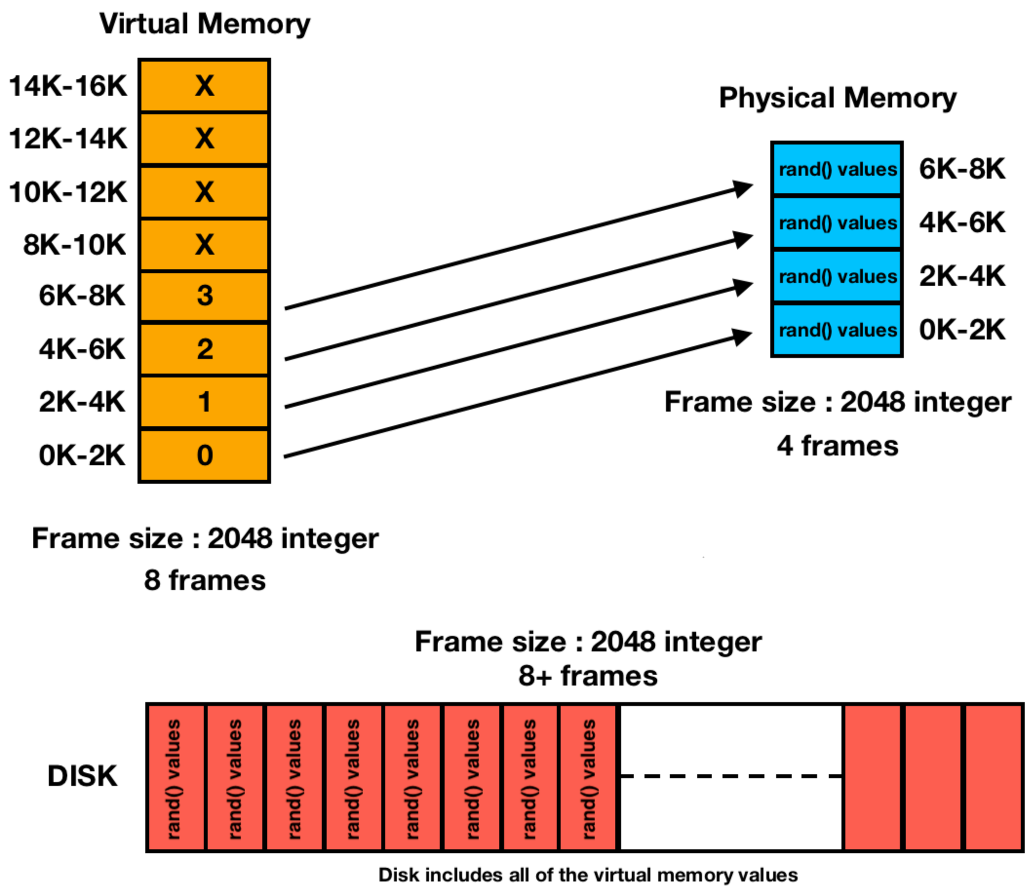


Figure 4 - Virtual memory system figure

Searching for value or page

First of all, the value of the page or the address of the page is found on virtual memory. In order to reach the value of the address found, it is first checked if it exists in physical memory. If there is, it becomes a “hit” and that value is returned directly. If not found, this will be “page fault”. In this case, since the value sought in physical memory cannot be found, the page containing the value sought from the secondary storage, ie disk, should be brought to physical memory. The best page to replace with page replacement algorithms is found and replaced with a new one. After the changed page, the value called from physical memory is taken and returned.

During this trade, of course, the values of the bits in the page table are updated and changed in the access times and the kept page variables. In other words, if the 30th page is kept in the 0th page of physical memory, for example, the holding page parameter changes since the 10th page will be kept after the rapport. Time, reference bit, modified bit, first_index and is_present bits are updated as appropriate. All this is done in get and set functions.

• Page Replacement Algorithms

In case of a page fault, it is necessary to bring a page containing physical storage, that is, the value sought from the disk. In other words, page replacement is required. There are different algorithms to find the most suitable page to replace from physical memory for page replacement. In this project these are:

1. **NRU (Not recently used) algorithm (sortArrays.c : 345)**
2. **FIFO (First in first out) algorithm (sortArrays.c : 213)**
3. **SC (Second chance) algorithm (sortArrays.c : 234)**
4. **LRU (Least recently used) algorithm (sortArrays.c : 276)**
5. **WSClock algorithm (sortArrays.c : 311)**

Not Recently Used Algorithm

In the not recently used algorithm, a comparison is made according to two bits. These are the referenced and modified bits. According to these bits, there are 4 cases from 2^2 and 4 are examined as classes. There are:

- **Class 0 : Not referenced, Not modified**
- **Class 1 : Not referenced, Modified**
- **Class 2 : Referenced, Not Modified**
- **Class 3 : Referenced, Modified**

The smallest numbered class is selected according to these classes and the page with that class feature is replaced.

The implementation of this in the code is as follows. There are 4 for loops according to the class priority and it controls the passes by scrolling in the pages in physical memory. The first loop is 0 0, the second loop is 0 1, the third loop is 1 0, and the fourth loop is for 1 1 state. Accordingly, if the specified bits match those searched, that page is selected to be replaced and replaced with the required page.

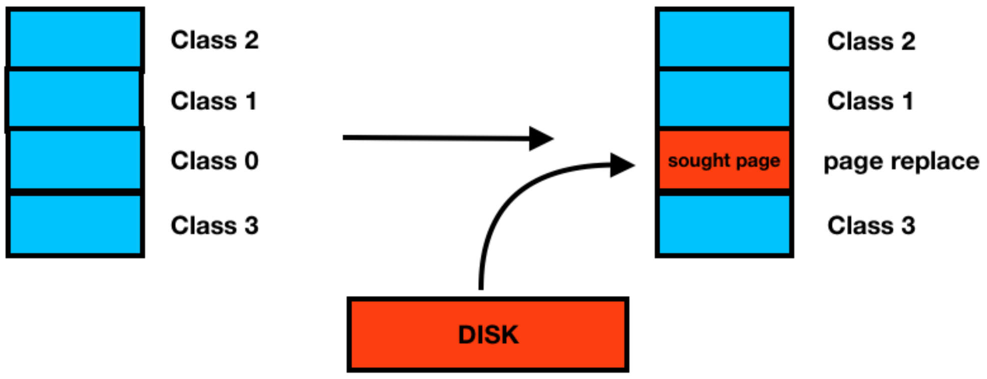


Figure 5 - Sample NRU replacement figure

First In First Out Algorithm

Pages in memory are treated like a queue. There are new pages in the head of the queue and old pages in the tail. Whenever a page replacement is required, the old page is removed from the tail.

I implemented this as follows. I fill the physical memory pages from beginning to end. In this context, physical memory is always the newest at the very end and always the oldest. I followed this up with an integer counter called fifo_index. The fifo_index counter, which starts at 0, increases by 1 each time a page replacement is made, so I always replace the oldest one with a new page. If it comes to the last index, it takes the mod again and returns to the beginning of the physical memory pages.

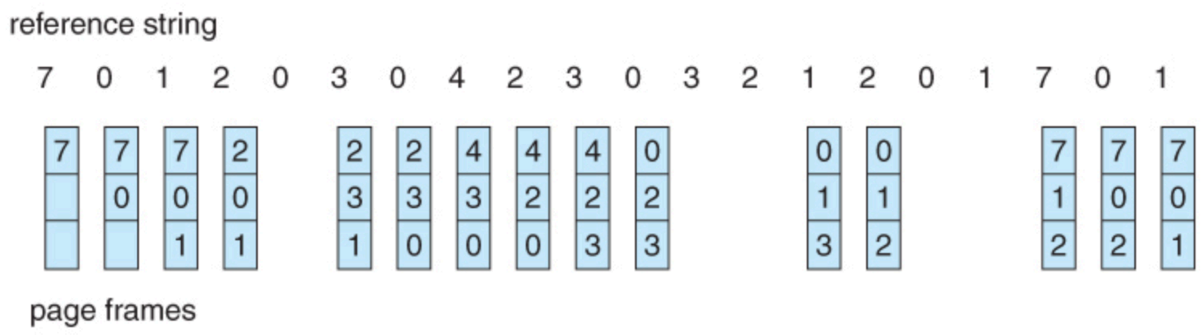


Figure 6 - Sample FIFO replacement figure

Second Chance

Second chance is a fifo algorithm based algorithm. There are also old ones in the tail section and new ones in the head section. However, unlike the fifo, reference bits are also checked when selecting the old page that will replace it. If the reference bit is 1, it is taken back to the queue's head, giving a second chance. If the reference bit is 0, it is normally replaced.

I implemented this as follows. I kept a sc_index counter just like I did in the fifo. And again I advanced it like in fifo. In addition to fifo. I checked the reference bits while selecting the page to delete. If it is 1, I brought that bit to 0 and I gave it a second chance. If it is 0 again, I brought a new page from the disk as in fifo.

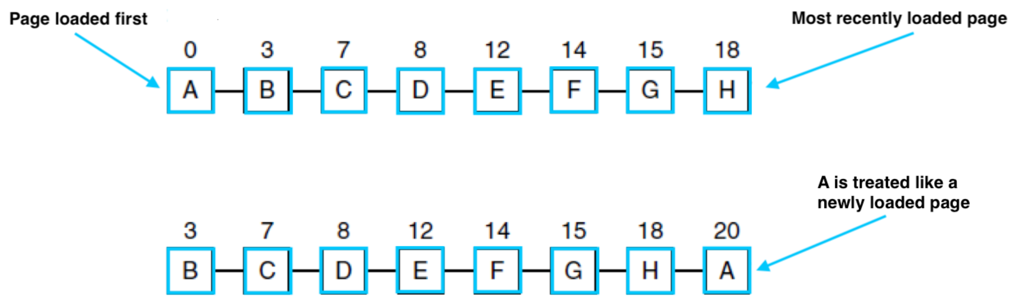


Figure 7 - Sample SC replacement figure

Least Recently Used Algorithm

There is a minimum number of used pages for a certain period of time and that page is replaced. To implement this, I went through all the physical memory pages to understand the usage amounts, then found the least used and replaced it.

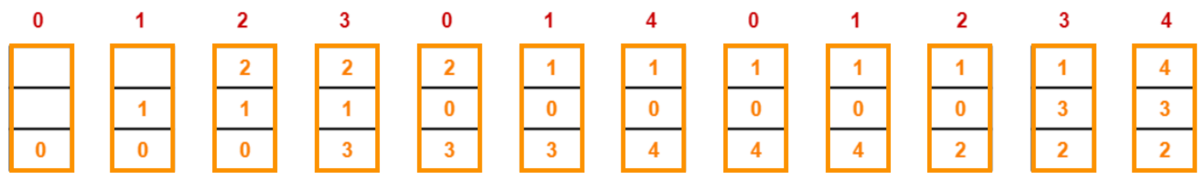


Figure 8 - Sample LRU replacement figure

WSClock Algorithm

In a data structure such as a circular list or clock, in case of page fault, it deletes the oldest and reference bit 0 and replaces it with a new page. If the oldest reference bit is 1, pull it to 0 and delete it in the next round.

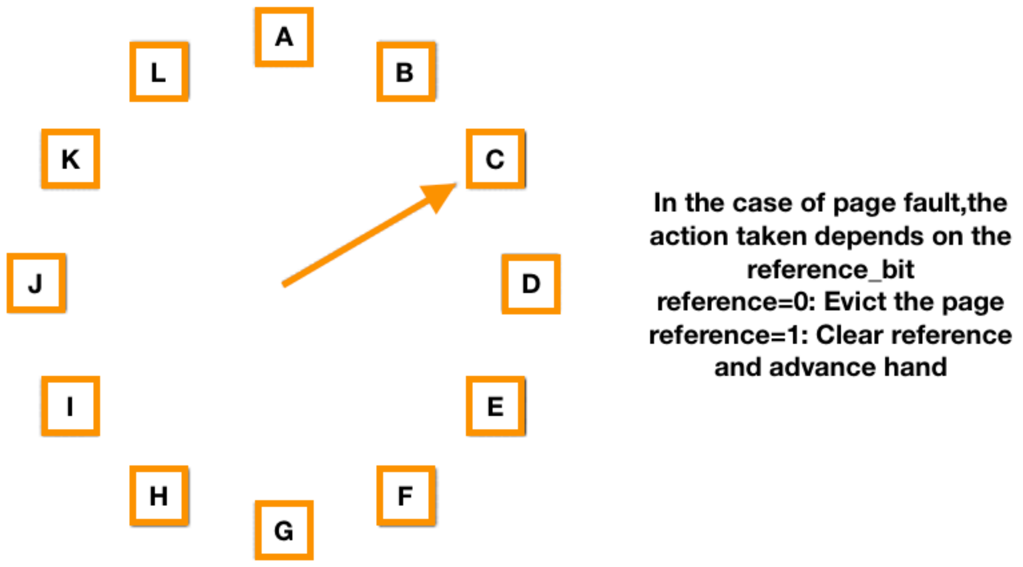

A
B
C
D
E
F
G
H
I
J
K
L
In the case of page fault,the action taken depends on the reference_bit
reference=0: Evict the page
reference=1: Clear reference and advance hand

Figure 9 - Sample WSClock replacement figure

• Set/Get functions

All of the get and set functions are between the mutex lock for accessing same data.

Get function (sortArrays.c : 472)

The get function works in two different ways, tName being index, fill and check and others (merge, bubble, quick). In the first part, namely for index, fill and check: It only reads the required address from physical memory for Check and Fill. In the index case, when it returns only indexes (address in vm), it does not encounter any page miss or page status because dictation can reach and return it as virtual_array [i]. In the second part, namely merge, bubble and quick, it firstly checks whether the desired address in physical memory has a value and if it exists, it returns directly. If it does not, it executes the page replacement algorithm given as input and returns the values of the desired address to physical memory and finally returns it from physical memory. Meanwhile, if the memory_access parameter is increased in each get and set, it reaches the number given as input, and makes a print page table. Also it updates the page table checking variables like reference bit,modified bit etc.

Set function (sortArrays.c : 407)

The set function works in 2 different ways. In case of "fill" as tName, it calls the fill function that fills the array and the disk in one move. Since it fills the arrays with rand (), there will not be any page fault, page miss, only memory and disk write. On the other hand, if "fill" does not come, it first checks whether there is an index given in the physical part of memory. If it is in the physical part, it replaces both it and the disk. If it is not in the physical part, it only changes the data on the disk. During these changes, it makes page table and variable updates.

• Allocation Policy

While doing a page replacement, I arranged 2 according to the allocation policy. While choosing the page to be replaced in the global one, I applied the replacement algorithm on all memory. But in case it is local, I set the memory to its own, regardless of which process is running for which quarter and made a replacement accordingly. In other words, while global replacements used all memory, local replacement only used the section reserved for it. Therefore, local replacements run slower. Global replacements work faster. As an example, while using all memory while bubble sort was in global policy, it was able to use only 1/4 of memory while saying local policy.

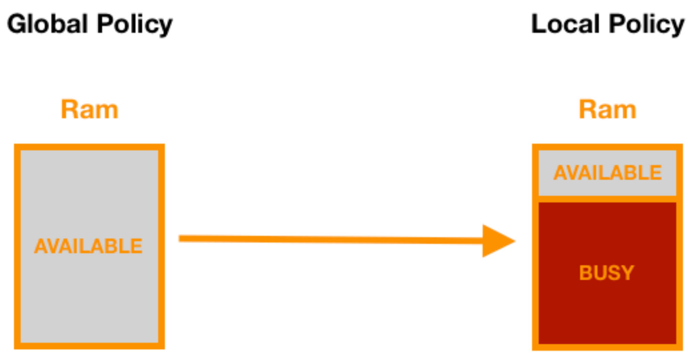


Figure 10 - Allocation policy figure

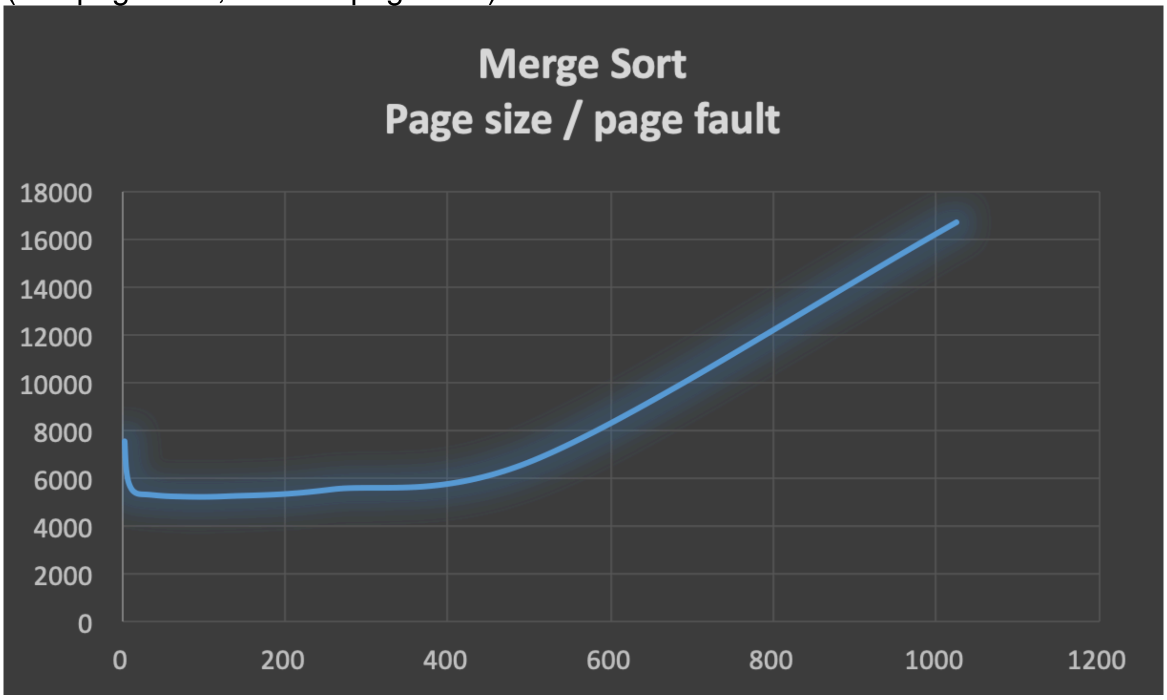
Part3

I set almost the same system in part2 for part3. But in this system, I used array instead of file for disk and used global policy, not local policy. In short, I simplified the complex parts in part2. Although it is generally the same code, I edited it more suitably for part3 purpose.

For the first section of Part3, I have set fixed virtual memory integer size and fixed physical memory size. I then wrote a reinitialize function to initialize the arrays. In this way, when the function works, it is filled again according to the new frame size. I combined all this to calculate the number of minimum page faults by changing the number of frames in a loop, from frame size (2 ^ 1) to (2 ^ i) physical memory frame count. This is how I found the optimum frame size.

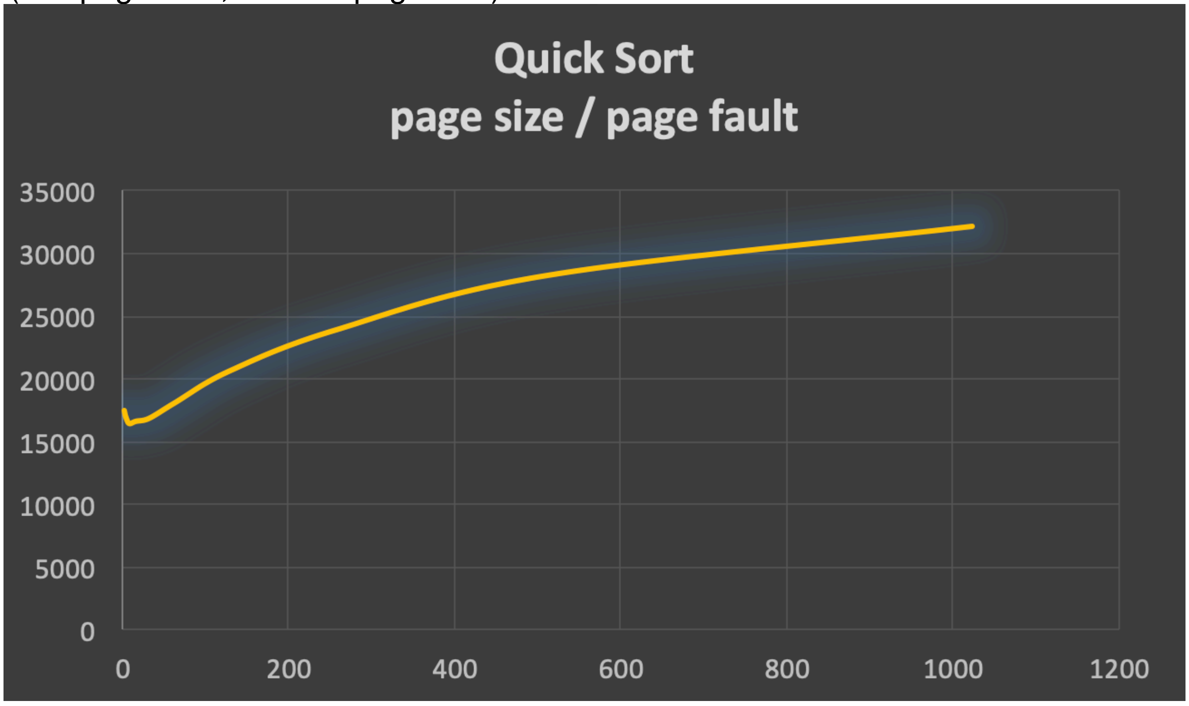
For the second section (bonus 1) I tried to find the minimum page fault by changing the replacement algorithm in the loop this time with the same system above. So I found the best replacement algorithm.

- **First section's Merge Sort (Optimal page size is 64)**
  (left: page fault, bottom: page size)



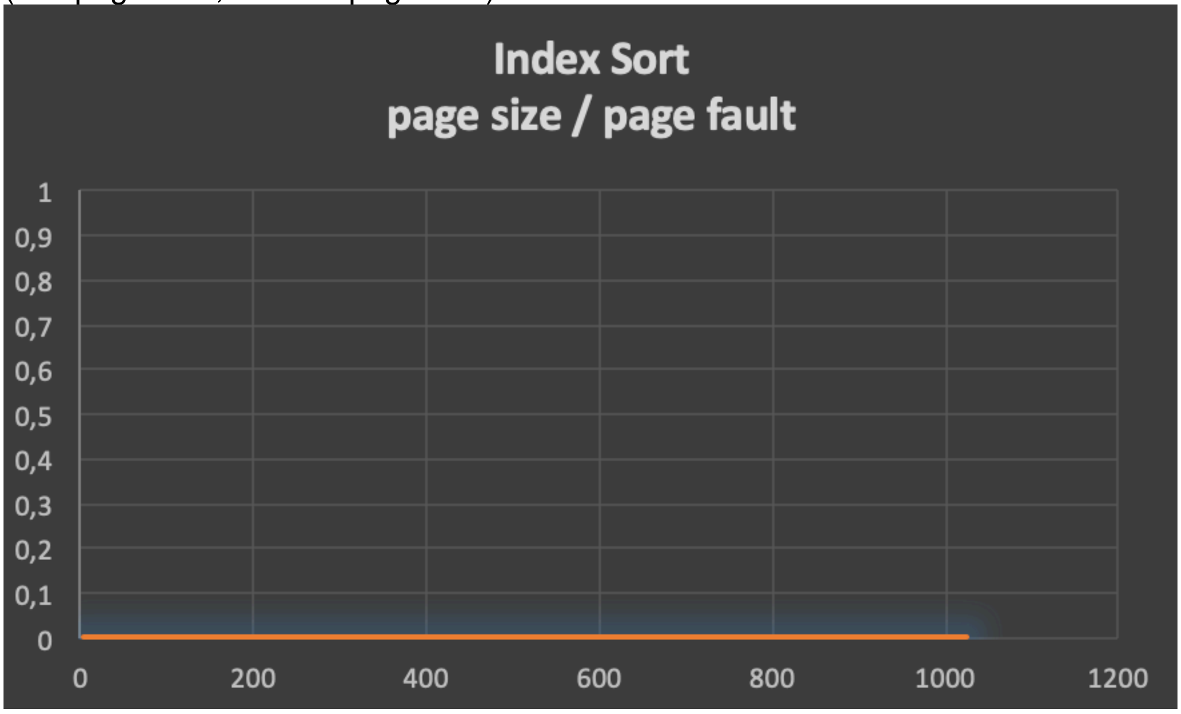
- **First section's Quick Sort (Optimal page size is 8)**

(left: page fault, bottom: page size)



- **First section's Index Sort (Optimal page size is 2)**

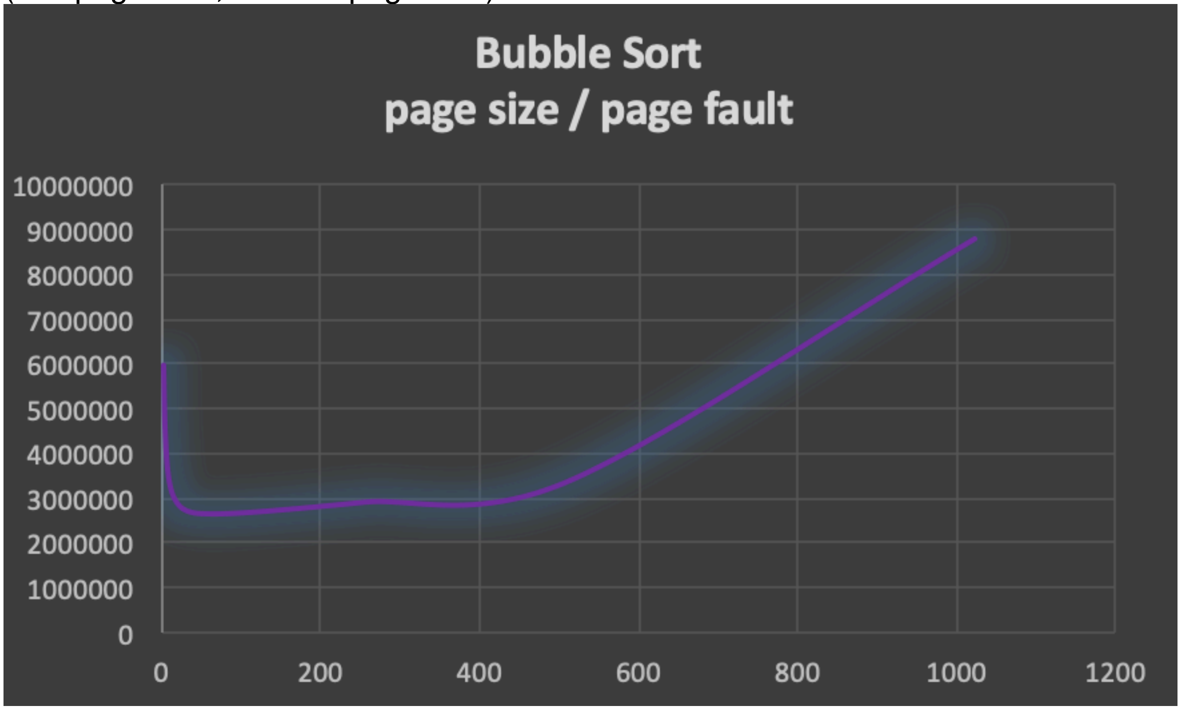
(left: page fault, bottom: page size)



Please readMe file for description

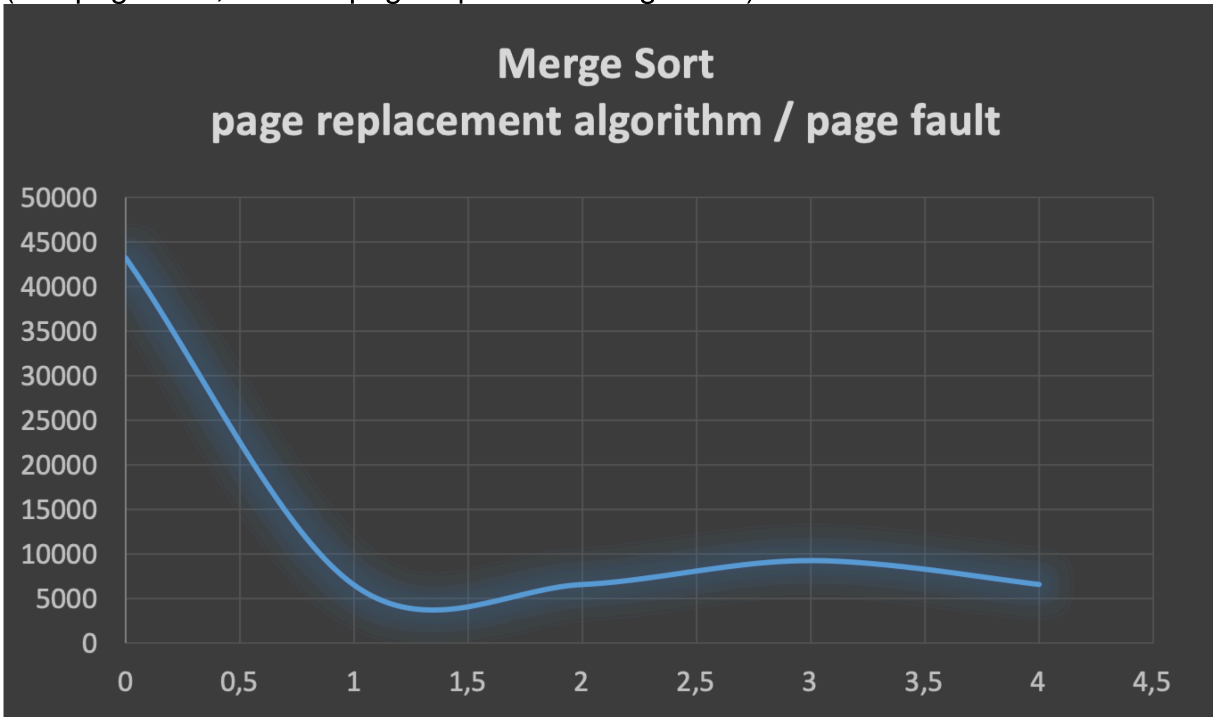
- **First section’s Bubble Sort (Optimal page size is 64)**

(left: page fault, bottom: page size)



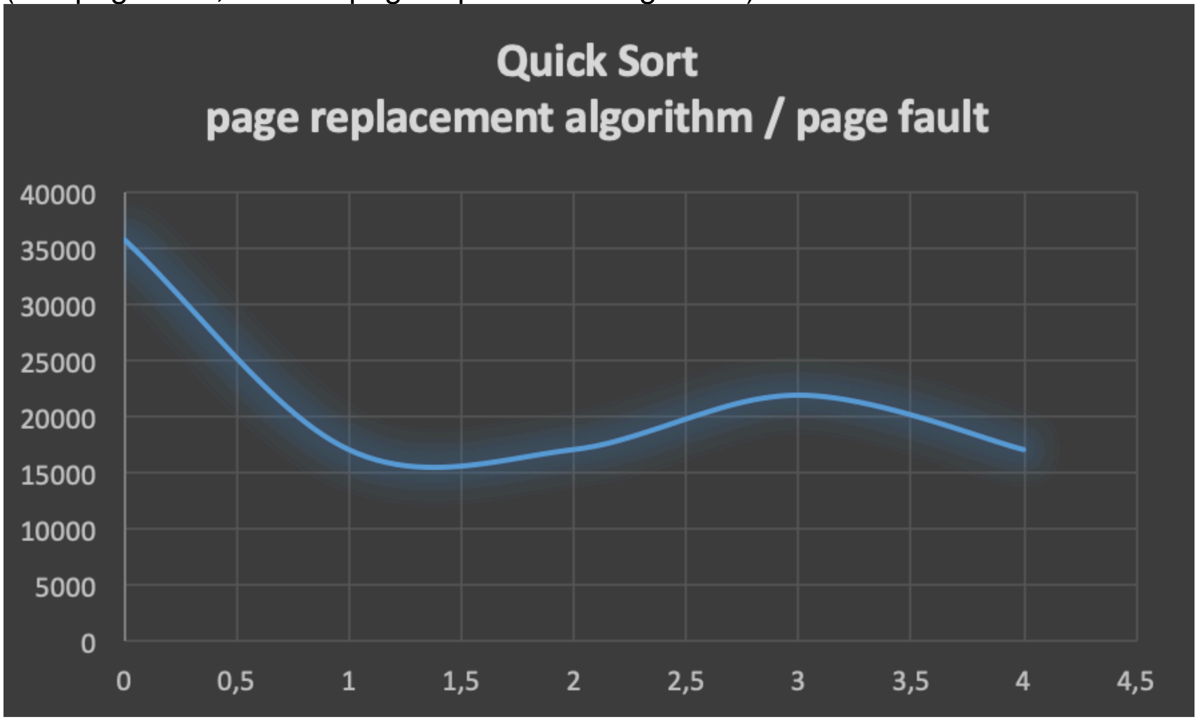
- **Second section’s (Bonus part) Merge Sort (Best p.r.a is FIFO(1))**

(left: page fault, bottom: page replacement algorithm)



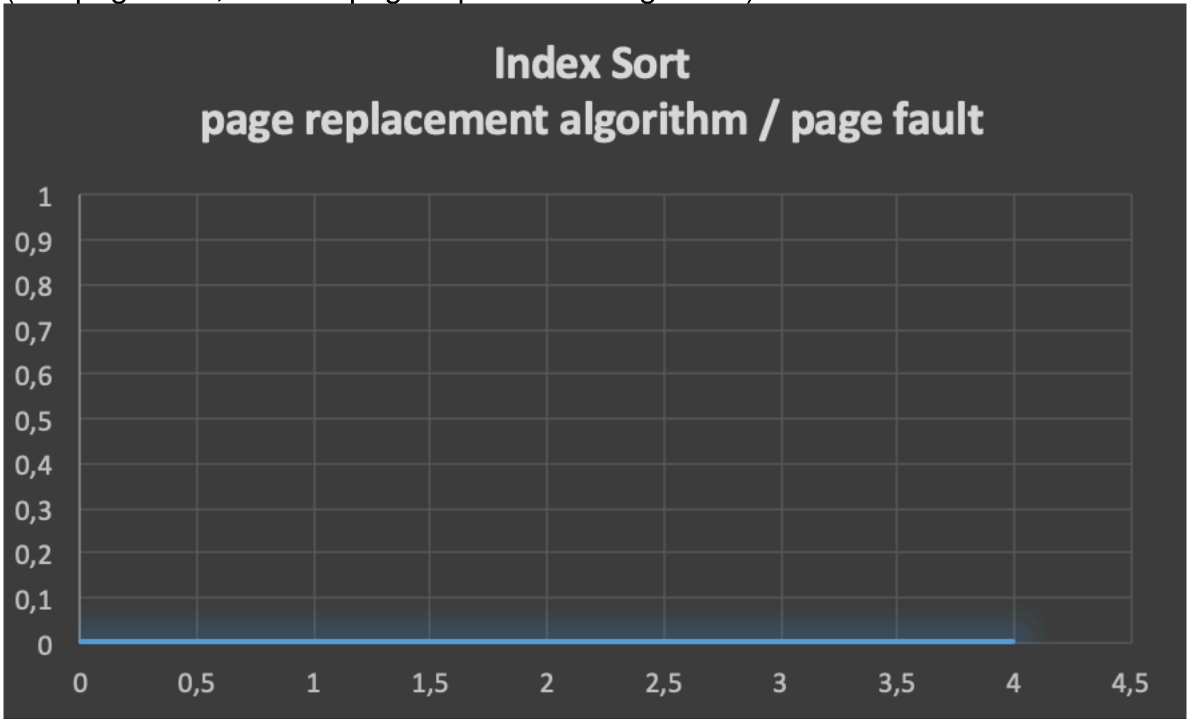
- **Second section's (Bonus part) Quick Sort (Best p.r.a is WSClock(4))**

(left: page fault, bottom: page replacement algorithm)



- **Second section's (Bonus part) Index Sort (Best p.r.a is NRU(0))**

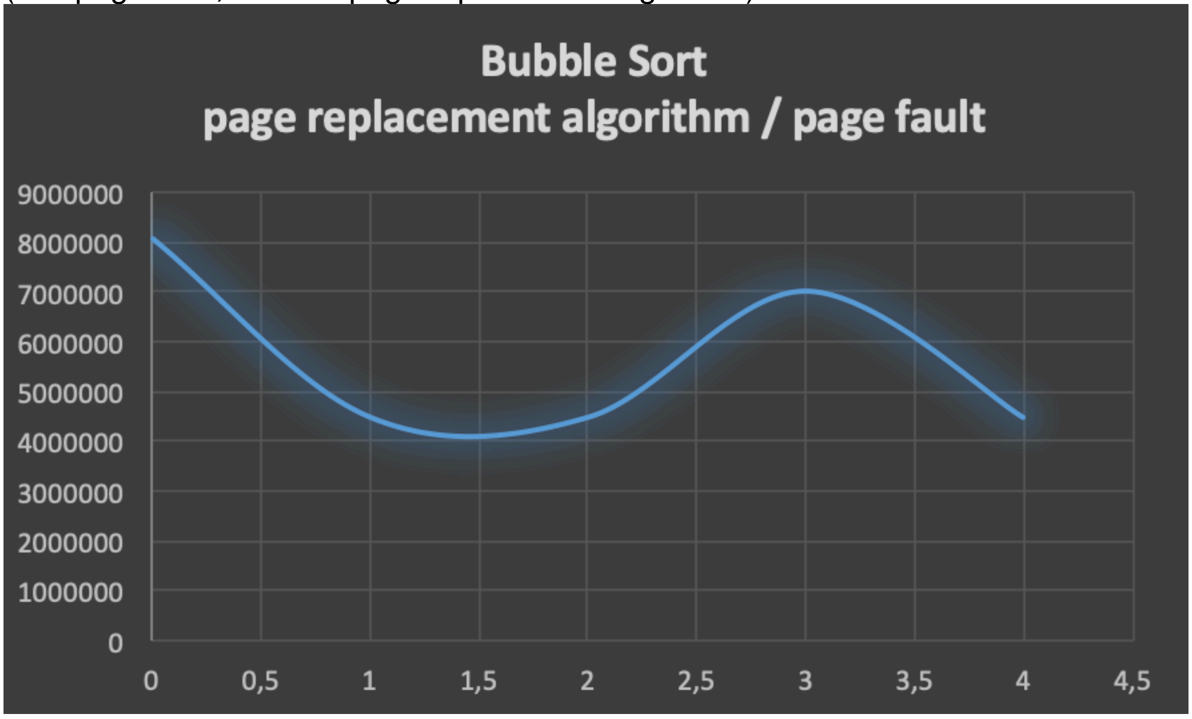
(left: page fault, bottom: page replacement algorithm)



Please readMe file for description

- **Second section's (Bonus part) Bubble Sort (Best p.r.a is WSClock(4))**

(left: page fault, bottom: page replacement algorithm)


Bubble Sort
page replacement algorithm / page fault