Homework 3 Report

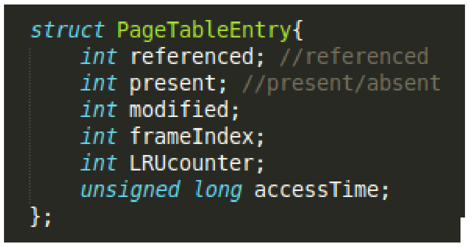
Elif Akgün
1801042251

1. Part 1

1.1 Page Table Structure

To keep page table entries, I used struct data type. It includes some entries for page replacement algorithms. I explained that why i keep these entries one by one below.

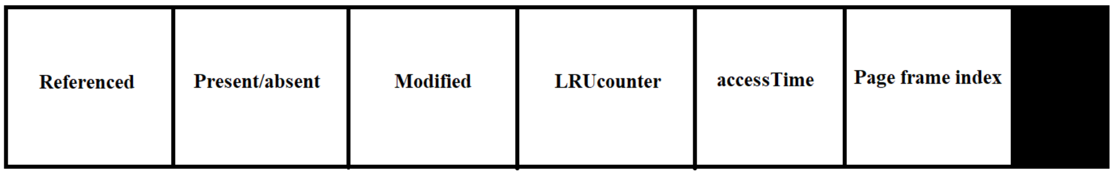
Figure 1: PageTableEntry Struct (line 57 in sortArrays.cpp)



```
struct PageTableEntry{
    int referenced; //referenced
    int present; //present/absent
    int modified;
    int frameIndex;
    int LRUcounter;
    unsigned long accessTime;
};
```

1. referenced: All page replacement algorithms except FIFO that are in this homework use R bits. So if a thread uses a page, this page's R bit becomes 1 to indicate some threads use it.
2. present: If the page is in physical memory, its present bit is 1. So I kept this entry to know whether a page is in memory or not.
3. modified: If the page is changed, its modified bit becomes 1. Whenever it is written to the disk, it becomes 0 again.
4. frameIndex: This entry indicates that the virtual page at which address in physical memory.
5. LRUcounter: I only used this entry in LRU page replacement algorithm. Because according to this algorithm, we need to keep a counter to know that what is the counter when it last accessed this page.
6. accessTime: I only used this entry in WSClock algorithm. This algorithm checks time difference between the current time and the last accessed time. So i kept last access time.

Figure 2: Page Table Entry

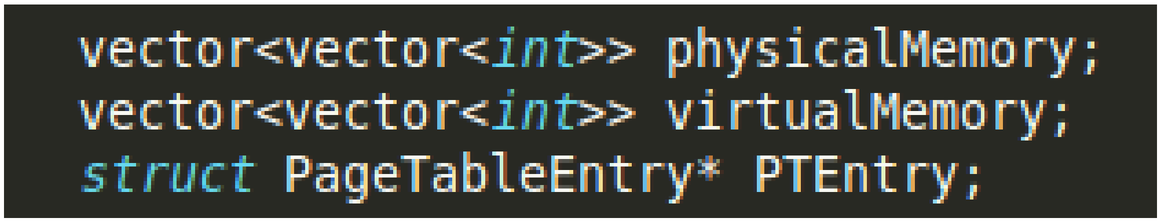


2. Part 2

2.1 Allocation Policy

I used different data types/structures for memories. I used 2D vector for physical memory and virtual memory, and I used an array for page table entries. The main reason of why I used vector data type is vectors can delete a specific row directly and insert a row(also vector) to a vector. Our page frames include some integers(also addresses in virtual memory). To facilitate page replacements I used vector data structure. Also there is a file to keep disk data.

Figure 3: Allocating Memories



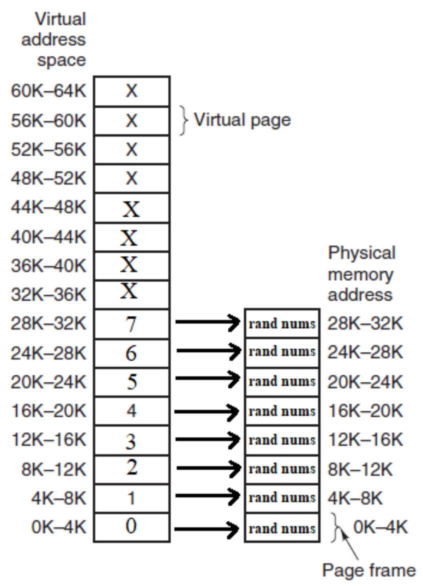
```
vector<vector<int>> physicalMemory;
vector<vector<int>> virtualMemory;
struct PageTableEntry* PTEntry;
```

physicalMemory vector's size is (physical memory page frame number * frame size) and virtualMemory vector's size is (virtual memory page frame number * frame size). PTEntry array's size equals to virtual memory page frame number. It keeps all pages' information.

In fillMemories function(line 188 in sortArrays.cpp), I filled memories with integer numbers. Numbers that are in virtual memory indicate addresses of physical memory. I also filled disk to keep data which are not in physical memory. The zeroth page in the virtual memory corresponds to the zeroth page in the physical memory. It goes on like this for as many as the physical number of pages.

For backing store I wrote a function: changeDisk(line 536 in sortArrays.cpp). It writes to disk the given value to given index as parameter. I used static swap area, so all data are in disk. Also, modified bit of current page becomes 0 because it is written to the disk.

Figure 4: Virtual Memory and Physical Memory System



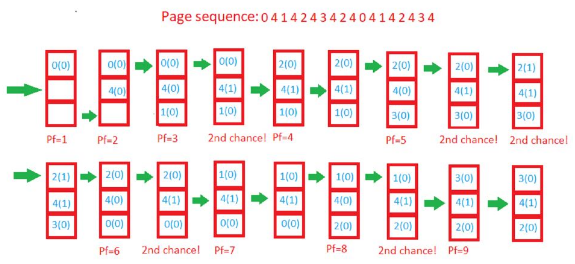
2.2 Page Replacement Algorithms

2.2.1 Second-Chance (SC)

This page replacement algorithm likes FIFO. Firstly, it checks first page. If it is referenced, then it removes it from top and puts it to end by changing its R bit with 0. It gives this page second change. Then it checks next page so it checks pages until find a page with R bit 0. When conditions are met, it replaces this page.

To implement this algorithm, I used a infinite while loop. It checks pages until conditions are met, then it replaces pages. Also I used an array which is pageFrameArr. Its size equals number of physical page frame and it keeps which pages are in physical memory. SC algorithm is on line 721 in sortArrays.cpp.

Figure 5: A Clear Example for SC Algorithm

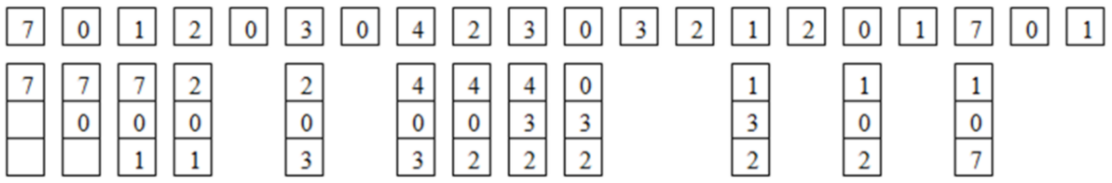


2.2.2 Least-Recently-Used (LRU)

This page replacement algorithm likes optimal page replacement algorithm. It keeps a counter for pages and always replace a page with the smallest counter. So it ensures that it removes least recently used page.

It is difficult that implement this algorithm in real life because we need hardware help. But this homework is a simulation. So to implement this algorithm, I kept LRUcounter entry for each page and a global LRUcounter. When it accesses a page, this page's LRUcounter becomes global LRUcounter. Global LRUcounter increases when a page is accessed. When a page fault occurs, I check all pages' LRUcounter value and I remove page with smallest counter which is least recently used page. LRU algorithm is on line 779 in sortArrays.cpp.

Figure 6: A Clear Example for LRU Algorithm

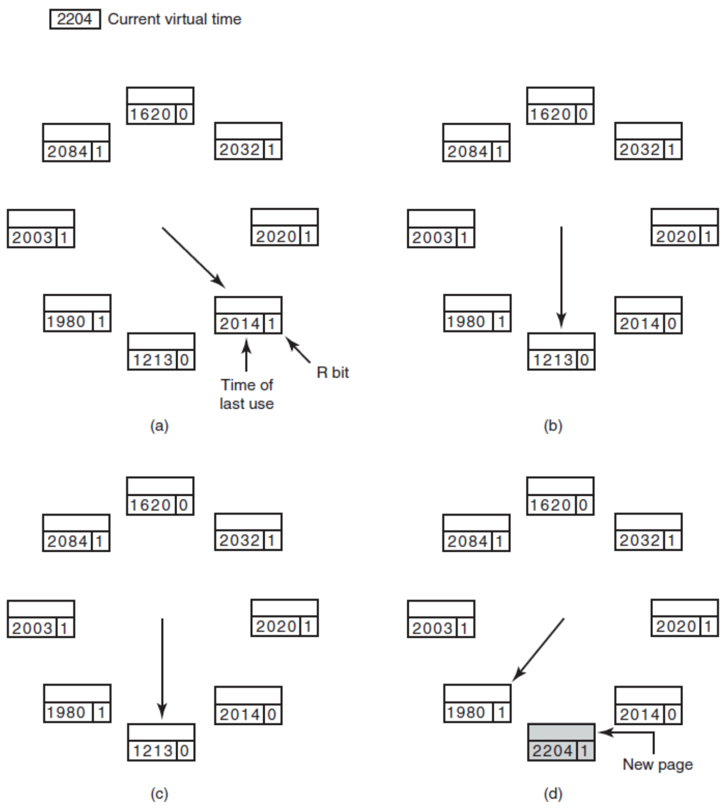


2.2.3 Working Set Clock (WSClock)

This page replacement algorithm uses the working set concept and likes SC algorithm. It keeps pages with a circular linked list. When page fault occurs, it checks R bits and last access time. If page's R bit is 1, then it changes it with 0 and checks next page. If a page's R bit is 0, then it checks page age. Page age equals (current virtual time - last access time). If it is bigger than threshold, it removes this page.

As I mentioned above, I kept a accessTime entry for all pages for this algorithm. I checked all pages in order. If the page's referenced entry is 1, then I changed it with 0 and continued. When a page's referenced entry is 0, I check page age. For this purpose, I wrote a function which is getCurrentTime((line 1057 in sortArrays.cpp). It returns current time in microseconds. And I kept a threshold T for page age. I set it to 100 microseconds. If the page's age bigger than 100 microseconds, I removed it. WSClock algorithm is on line 814 in sortArrays.cpp.

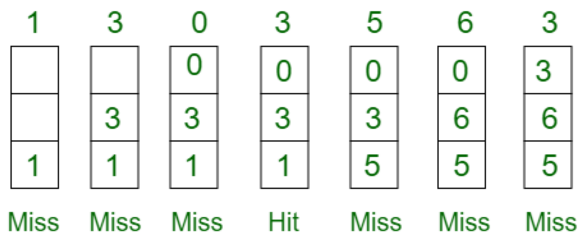
Figure 7: A Clear Example for WSClock Algorithm



2.2.4 The First-In, First-Out (FIFO)

This page replacement algorithm does not use any page table entry. It keeps all pages in the memory as a list. When page fault is occurred, it removes first page from page table and new page added to the end of the file. To implement this algorithm, I used pageFrameArr. When page fault occurred, I removed first element of this vector and I put new page to the end of the vector. FIFO algorithm is on line 865 in sortArrays.cpp.

Figure 8: A Clear Example for FIFO Algorithm



2.2.5 Not Recently Used (NRU)

This page replacement algorithm uses R bits and M bits. On clock interrupts, clears R bits. M bits are reseted when they are written to the disk. There are 4 classes in this algorithm: class 0, class 1, class2, class3.
class 0: not referenced, not modified
class 1: not referenced, modified
class 2: referenced, not modified
class 3: referenced, modified
When page fault is occurred, it removes a random page from the lowest class.

To implement this algorithm, I used referenced and modified page table entries. I kept 4 vector for classes. When page fault is occurred, I removed a random page from lowest class in the page table. NRU algorithm is on line 894 in sortArrays.cpp.

2.3 Implementation of Get Set Methods

2.3.1 get (line 586 in sortArrays.cpp)

This method has two parameters which are index and tName. index is virtual address of integer and tName is thread name. index takes a value between 0 and (virtual page frame number * page frame size). It gets an element from physical memory. As I mentioned before, physical memory is a 2D vector. So I need row and column indexes to access physical memory. row equals index/frameSize and column equals index%frameSize because I mapped memories like this. It checks this page's present bit. If it 1, it means this page is in physical memory and it returns its value. Otherwise it goes to disk, gets needed page and applies the page replacement algorithm.

This method has helper methods which are getDisk(line 642 in sortArrays.cpp), and pageReplace(line 706 in sortArrays.cpp). getDisk method gets data from disk and pageReplace method makes the page replacement operation with using chosen algorithm. Also get method uses mutex. Threads should not access at the same time to the disk because one thread can read while another thread is writing.

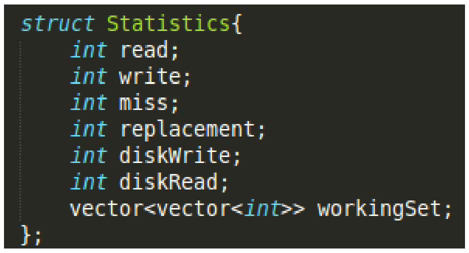
2.3.2 set (line 663 in sortArrays.cpp)

This method likes get method. It has three parameters which are index, value, tName. index is virtual address of integer, values is the value to be set, and tName is thread name. As in the get method, I accessed the physical memory and set the value. I updated this page's modified bit to indicate that this page is modified. There were synchronization issues as I mentioned in get method. So I used mutex in this method, too.

2.4 Statistics

I kept number of reads, number of writes, number of page misses, number of page replacements, number of disk page writes, number of disk page reads, and estimated working set function w statistics for each thread. I used struct data type.

Figure 9: Statistics Struct



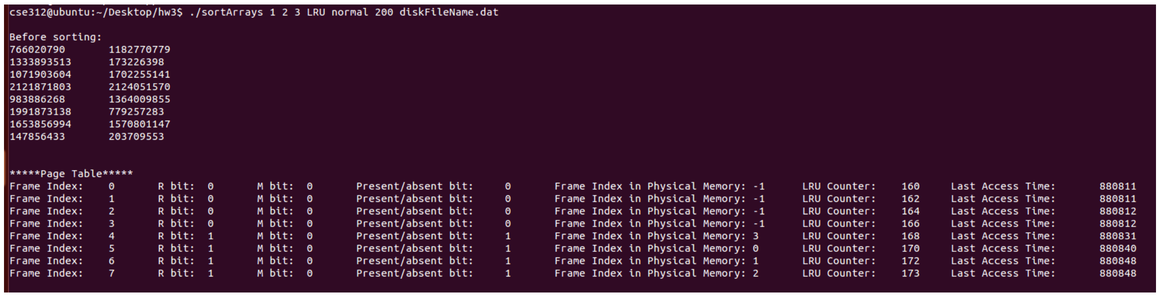
```
struct Statistics{
    int read;
    int write;
    int miss;
    int replacement;
    int diskWrite;
    int diskRead;
    vector<vector<int>> workingSet;
};
```

I kept an struct Statistics type array which is *stats* for threads. It has 6 elements. *stats*[0] is for bubble sort thread, *stats*[1] is for quick sort thread, *stats*[2] is for merge thread, *stats*[3] is for linear search thread, *stats*[4] is for binary search thread, and *stats*[5] is for fill. For statistics, I used some helper functions which are incReadNum(line 945 in sortArrays.cpp), incWriteNum(line 961 in sortArrays.cpp), incMissNum(line 977 in sortArrays.cpp), incRepNum(line 993 in sortArrays.cpp), incDiskWriteNum(line 1025 in sortArrays.cpp), incDiskReadNum(line 1009 in sortArrays.cpp), and addWorkingSet(line 1041 in sortArrays.cpp). These functions have a parameter which is tName. They increase relevant statistic of the thread. I wrote this functions to get rid of if else blocks in the get set functions.

2.5 Sample Output

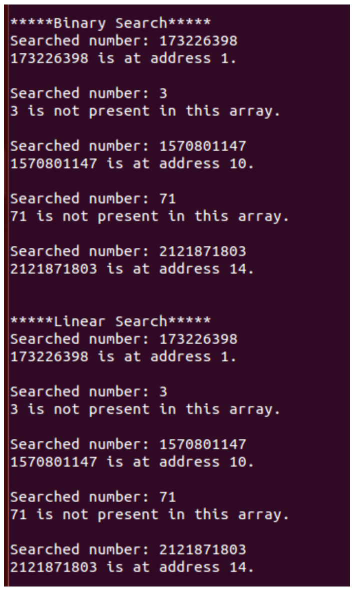
When I run the program with the ./sortArrays 1 2 3 LRU normal 200 diskFileName.dat command, the output is as follows.

Figure 10: Part 2 Output-1



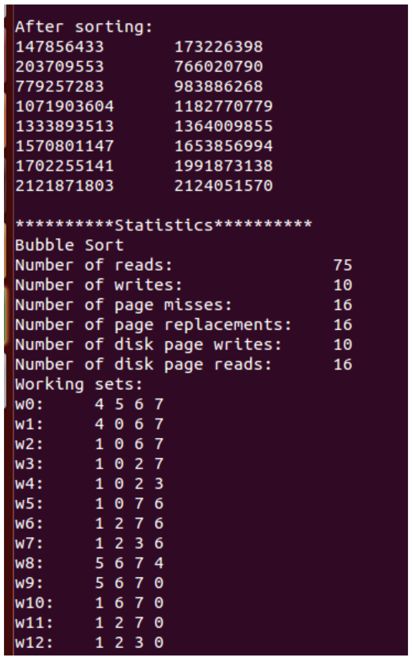
```
cse312@ubuntu:~/Desktop/hw3$ ./sortArrays 1 2 3 LRU normal 200 diskFileName.dat

Before sorting:
766020790       1182770779
1333893513      173226398
1071903604      1702255141
2121871803      2124051570
983886268       1364009855
1991873138      779257283
1653856994      1570801147
147856433       203709553


*****Page Table*****
Frame Index:    0       R bit:  0       M bit:  0       Present/absent bit:     0       Frame Index in Physical Memory: -1      LRU Counter:    160     Last Access Time:       880811
Frame Index:    1       R bit:  0       M bit:  0       Present/absent bit:     0       Frame Index in Physical Memory: -1      LRU Counter:    162     Last Access Time:       880811
Frame Index:    2       R bit:  0       M bit:  0       Present/absent bit:     0       Frame Index in Physical Memory: -1      LRU Counter:    164     Last Access Time:       880812
Frame Index:    3       R bit:  0       M bit:  0       Present/absent bit:     0       Frame Index in Physical Memory: -1      LRU Counter:    166     Last Access Time:       880812
Frame Index:    4       R bit:  1       M bit:  0       Present/absent bit:     1       Frame Index in Physical Memory: 3       LRU Counter:    168     Last Access Time:       880831
Frame Index:    5       R bit:  1       M bit:  0       Present/absent bit:     1       Frame Index in Physical Memory: 0       LRU Counter:    170     Last Access Time:       880840
Frame Index:    6       R bit:  1       M bit:  0       Present/absent bit:     1       Frame Index in Physical Memory: 1       LRU Counter:    172     Last Access Time:       880848
Frame Index:    7       R bit:  1       M bit:  0       Present/absent bit:     1       Frame Index in Physical Memory: 2       LRU Counter:    173     Last Access Time:       880848
```

Figure 11: Part 2 Output-2



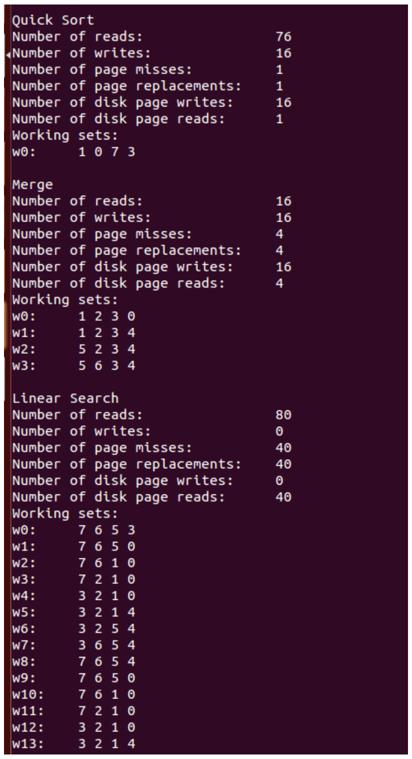
```
*****Binary Search*****
Searched number: 173226398
173226398 is at address 1.

Searched number: 3
3 is not present in this array.

Searched number: 1570801147
1570801147 is at address 10.

Searched number: 71
71 is not present in this array.

Searched number: 2121871803
2121871803 is at address 14.


*****Linear Search*****
Searched number: 173226398
173226398 is at address 1.

Searched number: 3
3 is not present in this array.

Searched number: 1570801147
1570801147 is at address 10.

Searched number: 71
71 is not present in this array.

Searched number: 2121871803
2121871803 is at address 14.
```

Figure 12: Part 2 Output-3



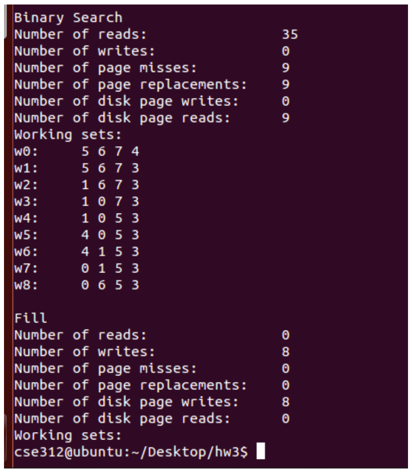
```
After sorting:
147856433       173226398
203709553       766020790
779257283       983886268
1071903604      1182770779
1333893513      1364009855
1570801147      1653856994
1702255141      1991873138
2121871803      2124051570

**********Statistics**********
Bubble Sort
Number of reads:                75
Number of writes:               10
Number of page misses:          16
Number of page replacements:    16
Number of disk page writes:     10
Number of disk page reads:      16
Working sets:
w0:     4 5 6 7
w1:     4 0 6 7
w2:     1 0 6 7
w3:     1 0 2 7
w4:     1 0 2 3
w5:     1 0 7 6
w6:     1 2 7 6
w7:     1 2 3 6
w8:     5 6 7 4
w9:     5 6 7 0
w10:    1 6 7 0
w11:    1 2 7 0
w12:    1 2 3 0
```

Figure 13: Part 2 Output-4



```
Quick Sort
Number of reads:                76
Number of writes:               16
Number of page misses:          1
Number of page replacements:    1
Number of disk page writes:     16
Number of disk page reads:      1
Working sets:
w0:     1 0 7 3

Merge
Number of reads:                16
Number of writes:               16
Number of page misses:          4
Number of page replacements:    4
Number of disk page writes:     16
Number of disk page reads:      4
Working sets:
w0:     1 2 3 0
w1:     1 2 3 4
w2:     5 2 3 4
w3:     5 6 3 4

Linear Search
Number of reads:                80
Number of writes:               0
Number of page misses:          40
Number of page replacements:    40
Number of disk page writes:     0
Number of disk page reads:      40
Working sets:
w0:     7 6 5 3
w1:     7 6 5 0
w2:     7 6 1 0
w3:     7 2 1 0
w4:     3 2 1 0
w5:     3 2 1 4
w6:     3 2 5 4
w7:     3 6 5 4
w8:     7 6 5 4
w9:     7 6 5 0
w10:    7 6 1 0
w11:    7 2 1 0
w12:    3 2 1 0
w13:    3 2 1 4
```

Figure 14: Part 2 Output-5



```
Binary Search
Number of reads:                35
Number of writes:               0
Number of page misses:          9
Number of page replacements:    9
Number of disk page writes:     0
Number of disk page reads:      9
Working sets:
w0:     5 6 7 4
w1:     5 6 7 3
w2:     1 6 7 3
w3:     1 0 7 3
w4:     1 0 5 3
w5:     4 0 5 3
w6:     4 1 5 3
w7:     0 1 5 3
w8:     0 6 5 3

Fill
Number of reads:                0
Number of writes:               8
Number of page misses:          0
Number of page replacements:    0
Number of disk page writes:     8
Number of disk page reads:      0
Working sets:
cse312@ubuntu:~/Desktop/hw3$ 
```

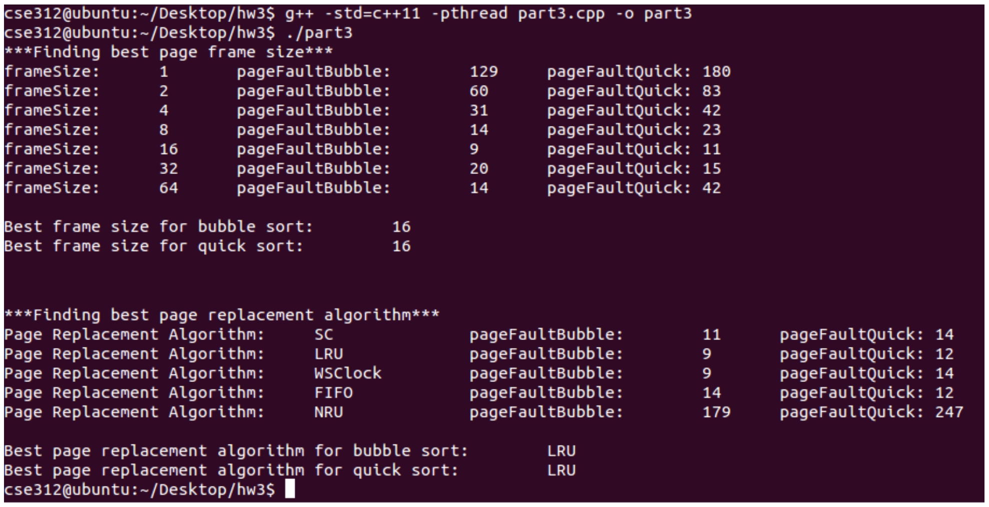
3. Part 3

It is written that ".. physical memory can hold 16K integers and the virtual memory can hold 128K integers." for this part in homework file. I set memories as given but it took a lot of time and program did not finish. So I set memories to smaller numbers. Physical memory can hold 64 integers and the virtual memory can hold 128 integers in my case. I did bonus part, too. I wrote a new program which is part3.cpp for this part. It almost same with sortArrays.cpp. I added two new functions: findBestFrameSize(line 96 in part3.cpp) and findBestAlgorithm(line 145 in part3.cpp).

findBestFrameSize function checks all page frame size in a for loop. This loop goes up to the physical memory size and it can be power of 2 so frame size can be 1, 2, 4, 8, 16, 32, 64 because physical memory size is 64. I kept 2 counters which are counter1 and counter2. counter1 keeps page fault count for bubble sort and counter2 keeps page fault count for quick sort. After each iteration, I kept results in pageReplBubble and pageReplQuick arrays. After the for loop, I found the best page frame size by finding minimum page fault count for each sorting thread.

findBestAlgorithm function checks all page replacement algorithms in a for loop. There are 5 different page replacement algorithm, so this loop goes up to five. If index equals 0, page replacement algotihm becomes SC; else if index equals 1, page replacement algotihm becomes LRU; if index equals 2, page replacement algotihm becomes WSClock, if index equals 3, page replacement algotihm becomes FIFO, else page replacement algotihm becomes LRU. Like findBestFrameSize function, I chechked all page fault counts for each sorting thread and I kept results in pageReplBubble and pageReplQuick arrays. After for loop, I found the best page replacement algorithm by finding minimum page fault count for each sorting thread.

Figure 15: Part 3 Output



```
cse312@ubuntu:~/Desktop/hw3$ g++ -std=c++11 -pthread part3.cpp -o part3
cse312@ubuntu:~/Desktop/hw3$ ./part3
***Finding best page frame size***
frameSize:      1       pageFaultBubble:        129     pageFaultQuick: 180
frameSize:      2       pageFaultBubble:        60      pageFaultQuick: 83
frameSize:      4       pageFaultBubble:        31      pageFaultQuick: 42
frameSize:      8       pageFaultBubble:        14      pageFaultQuick: 23
frameSize:      16      pageFaultBubble:        9       pageFaultQuick: 11
frameSize:      32      pageFaultBubble:        20      pageFaultQuick: 15
frameSize:      64      pageFaultBubble:        14      pageFaultQuick: 42

Best frame size for bubble sort:        16
Best frame size for quick sort:         16


***Finding best page replacement algorithm***
Page Replacement Algorithm:     SC              pageFaultBubble:        11      pageFaultQuick: 14
Page Replacement Algorithm:     LRU             pageFaultBubble:        9       pageFaultQuick: 12
Page Replacement Algorithm:     WSClock         pageFaultBubble:        9       pageFaultQuick: 14
Page Replacement Algorithm:     FIFO            pageFaultBubble:        14      pageFaultQuick: 12
Page Replacement Algorithm:     NRU             pageFaultBubble:        179     pageFaultQuick: 247

Best page replacement algorithm for bubble sort:        LRU
Best page replacement algorithm for quick sort:         LRU
cse312@ubuntu:~/Desktop/hw3$ 
```

X axis represents page faults, y axis represents frame size.

Figure 16: Bubble Sort Page Frame Size/Page Fault Graph

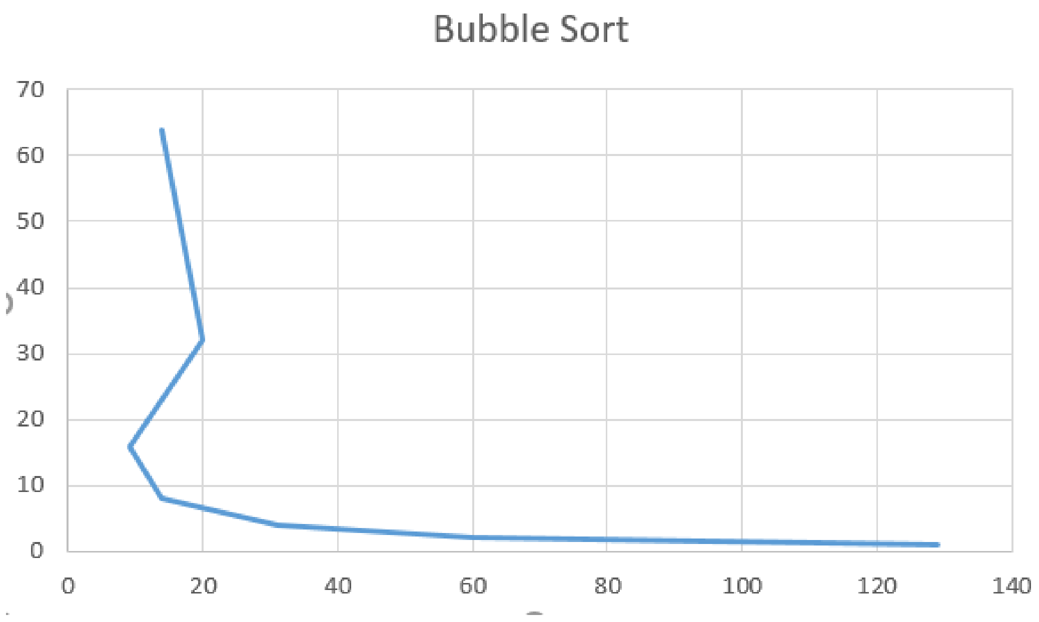
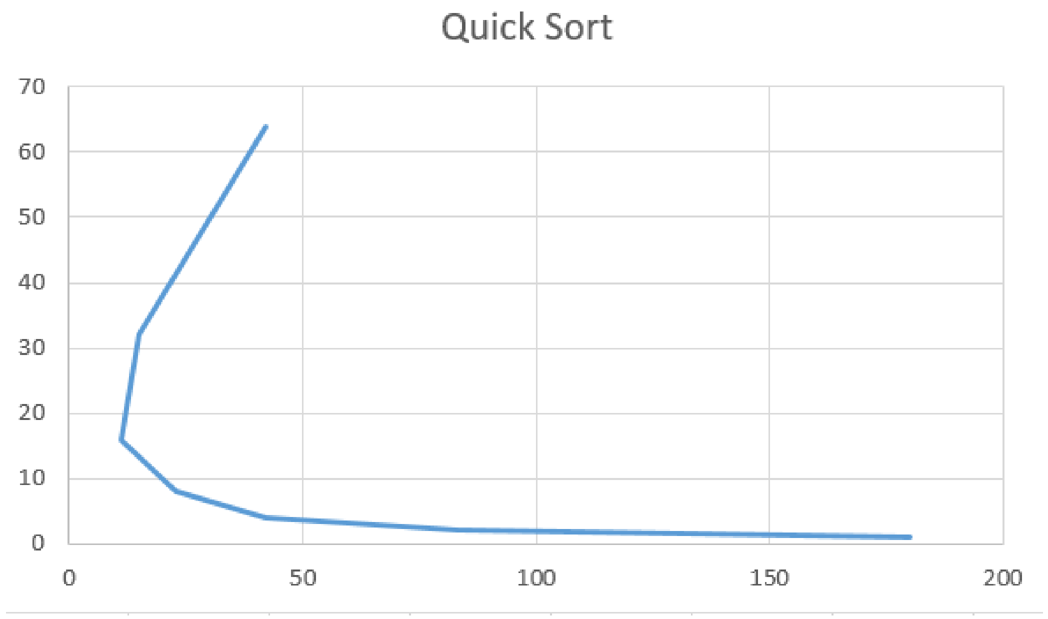


Figure 17: Quick Sort Page Frame Size/Page Fault Graph



4. Program Flow

Firstly, I parsed command line arguments in parseCL(line 139 in sortArrays.cpp) function. Then initPTEntry(line 173 in sortArrays.cpp) initializes page table entries and fillMemories(line 188 in sortArrays.cpp) fills virtual memory and physical memory with random integers. I wrote thrOperations(line 242 in sortArrays.cpp) function for thread operations. It creates 5 threads. First thread sorts first part by using bubble sort second thread sorts second part by using quick sort. Third thread merge two parts and fourth and fifth threads search with linear search and binary search respectively. Merge thread waits for sorting operations. For this purpose I created a conditional variable. When sorting threads are done, the signal is sent, then merge thread runs. Also linear search and binary search threads wait for merge operation. For this purpose I created another conditional variable. When merge thread is done, the signal is sent and searching threads run. After thread operations, it writes sorted integer numbers and statistics to console.