CSE 312 FINAL REPORT

Project Overview

What is this project about ?

This project is about simulating basic virtual memory operations and implementing various page replacement algorithms. After implementing those operations and algorithms we'll run experiments with different parameters to analyze performance of those algorithms.

What is the virtual memory ?

Virtual memory is kind of abstract term that refers to a technique that can benefit on ***transparency***, ***efficiency*** and ***protection***. I will explain these terms and how I used in my simulation in later pages for now we can think virtual memory as some kind of system that responsible illusion of large and private address space to programs.

Figure 1: Tom Kilburn, inventor of virtual memory technique.

What will be kept in virtual memory ?

Since this is a simple simulation we're not going to hold everything like(program text, stack variables, heap variables etc.). We'll think our memory as cells that can hold only "integer" values.

Page Table Implementation

In order to understand page table implementation first we need to understand what is page and what are basic terms like frame, frame size etc.

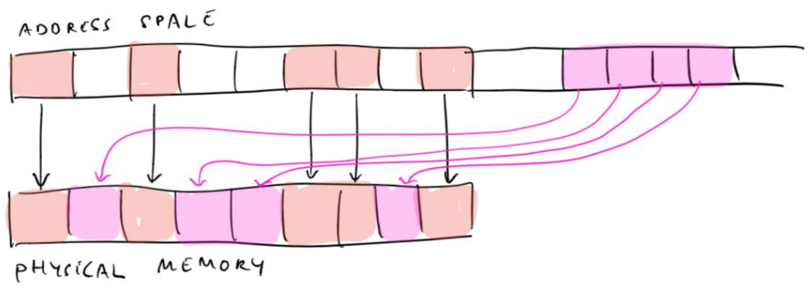


Figure 2: Virtual and Physical Memory

Pages:

We can think pages as bunch of memory cells that collected in one place. In our simulation pages are take shaped from integers.

Page Frames:

Corresponding units in the real physical memory called page frames. Generally page frames and virtual pages have the same size.

Physical Memory:

In real world physical memory is meaning RAM space. In our implementation this will be a raw C integer array.

Disk:

If we promised the user more memory space than RAM then we need to keep the other data in some place which is disk. In our simulation this will be a linux file.

Page replacement:

If the specified page is not present in memory then we need to replace that page from disk. This is called page replacement. All steps will be covered with references to my source code.

Allocation Policies:

Page replacement can be either local or global. Local page replacement isolates each process' pages to themselves. Global page replacements doesn't do that so it can be more efficient but also less scalable. In our simulation we'll use both policies.

Since we understand the basic terms now I can talk about page table implementation. There is a saying from David Wheeler, he says that : “All problems in computer science can be solved by another level of indirection.”

Without a surprise we’ll use indirection in virtual memory too. Page table will be the key struct to succeed that indirection. Page table is consists of page table entries. These entries are kept as 32 bit space. There are many fields inside of it like Global, Dirty etc. In our simulation I used a struct to represent those entries:

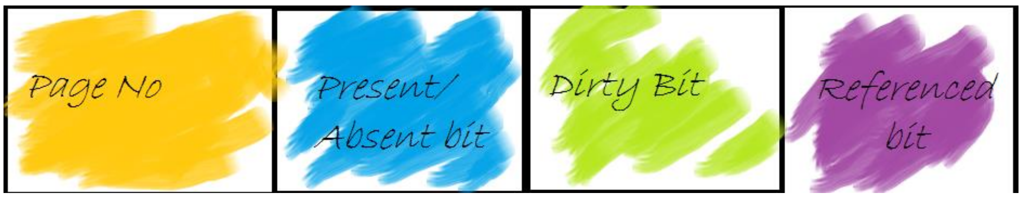


Figure 3: Page table entry (virtual_mem.h – line 14)

Page Frame Number:

This is the integer value that holds which physical address that entry maps.

Present/Absent:

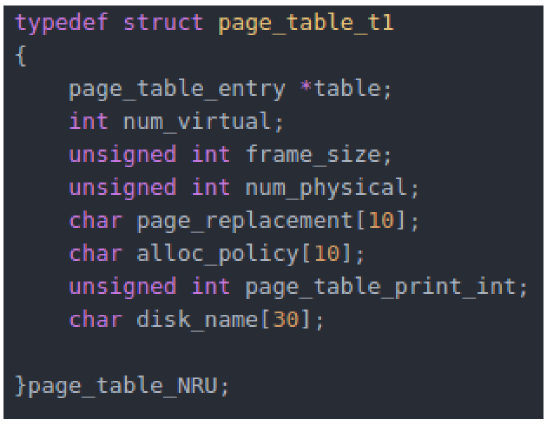
This is the integer value that specifies the current page is in disk or not.

Dirty Bit:

This is the bit for specifying the page is modified or not. We need this bit because we don’t want to make unnecessary disk writes.

Referenced:

This is bit will be valid when the current page is referenced. We’ll use this bit in algorithms like NRU.



```
typedef struct page_table_t1
{
    page_table_entry *table;
    int num_virtual;
    unsigned int frame_size;
    unsigned int num_physical;
    char page_replacement[10];
    char alloc_policy[10];
    unsigned int page_table_print_int;
    char disk_name[30];

}page_table_NRU;
```

Figure 4: Page table struct inside virtual_mem.h

This is the page table struct that holds array of page table entries called table. It also holds the program input parameters. I kept those values in order to make it easier to reach those variables inside thread functions.

Program Flow

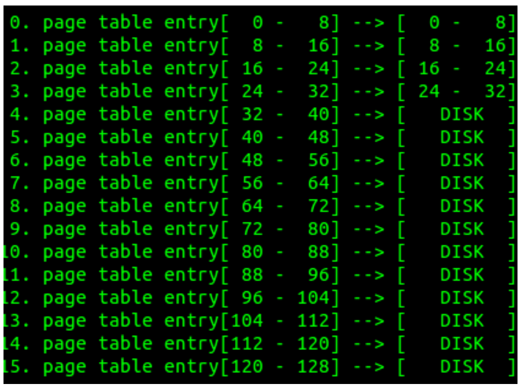
Allocating resources and checking parameters:

When the program starts executing as main thread parameters are taken from command line arguments. Those arguments are checked via “check_arguments” function (sortArrays.c, 133.line). If the parameters meaningless like frame_size = 0, num_physical > num_virtual, program gracefully terminated with error message.

After parameters are checked necessary resources allocated on heap. The allocated resources are physical memory, page table, bitmap matrix for LRU algorithm and queue etc. Also threads and mutexes created with using pthread library of POSIX.

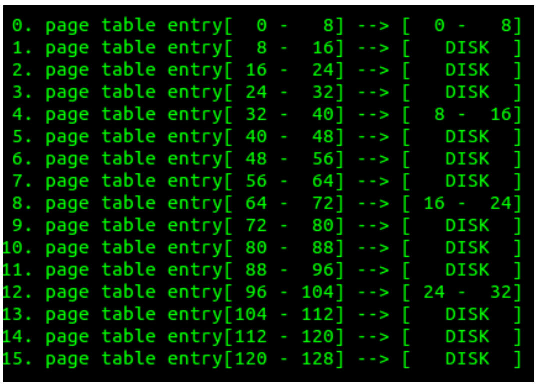
Filling virtual memory as main thread:

Main thread starts filling virtual memory with random integers. What we mean here virtual memory is physical memory + DISK. If physical memory has not empty spaces the filling is done to DISK. If physical memory have spaces it’s done to physical memory. If the allocation policy global the filling process is straight forward but if the allocation process is local we need to make sure that each ¼ . quarter of memory has at least one page. (fill_virtual_memory function, virtual_mem.c)



```
0. page table entry[  0 -   8] --> [  0 -   8]
1. page table entry[  8 -  16] --> [  8 -  16]
2. page table entry[ 16 -  24] --> [ 16 -  24]
3. page table entry[ 24 -  32] --> [ 24 -  32]
4. page table entry[ 32 -  40] --> [   DISK   ]
5. page table entry[ 40 -  48] --> [   DISK   ]
6. page table entry[ 48 -  56] --> [   DISK   ]
7. page table entry[ 56 -  64] --> [   DISK   ]
8. page table entry[ 64 -  72] --> [   DISK   ]
9. page table entry[ 72 -  80] --> [   DISK   ]
10. page table entry[ 80 -  88] --> [   DISK   ]
11. page table entry[ 88 -  96] --> [   DISK   ]
12. page table entry[ 96 - 104] --> [   DISK   ]
13. page table entry[104 - 112] --> [   DISK   ]
14. page table entry[112 - 120] --> [   DISK   ]
15. page table entry[120 - 128] --> [   DISK   ]
```

Figure 5: Filling with global policy



```
0. page table entry[  0 -   8] --> [  0 -   8]
1. page table entry[  8 -  16] --> [   DISK   ]
2. page table entry[ 16 -  24] --> [   DISK   ]
3. page table entry[ 24 -  32] --> [   DISK   ]
4. page table entry[ 32 -  40] --> [  8 -  16]
5. page table entry[ 40 -  48] --> [   DISK   ]
6. page table entry[ 48 -  56] --> [   DISK   ]
7. page table entry[ 56 -  64] --> [   DISK   ]
8. page table entry[ 64 -  72] --> [ 16 -  24]
9. page table entry[ 72 -  80] --> [   DISK   ]
10. page table entry[ 80 -  88] --> [   DISK   ]
11. page table entry[ 88 -  96] --> [   DISK   ]
12. page table entry[ 96 - 104] --> [ 24 -  32]
13. page table entry[104 - 112] --> [   DISK   ]
14. page table entry[112 - 120] --> [   DISK   ]
15. page table entry[120 - 128] --> [   DISK   ]
```

Figure 6: Figure 7: Filling with local policy

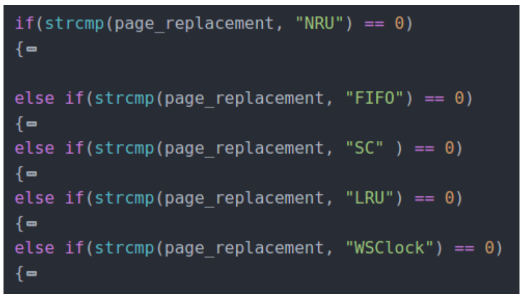
Joining sort threads:

After filling whole virtual memory. We're ready to sort those random values with 4 different threads. Threads will be: Bubble Sort, Quick Sort, Merge Sort, Index Sort in order. Those thread functions contains sort algorithms that you can find in virtual_mem.c (6441. line)

Of course those sort algorithms will try to access our virtual memory with get and set functions. If get and set functions work properly the virtual array will be sorted.

Get Method:

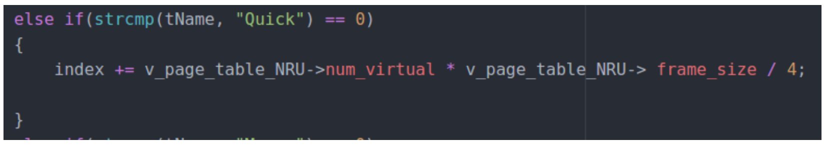
1. Lock the mutex. So shared variables won't cause any problems.(virtual_mem.c 642.line)
2. There are 5 different page replacement algorithms create 5 different sections of code with if conditions.(Unfortunately a lot of code duplicated here.)



```
if(strcmp(page_replacement, "NRU") == 0)
{

else if(strcmp(page_replacement, "FIFO") == 0)
{
else if(strcmp(page_replacement, "SC" ) == 0)
{
else if(strcmp(page_replacement, "LRU") == 0)
{
else if(strcmp(page_replacement, "WSClock") == 0)
{
```

Figure 8: Conditions for per page replacement algorithm.

3. Check the thread name for translate address space. We know that virtual memory provides same address space to each process but behind the scenes true address is converted. I did the same thing here.

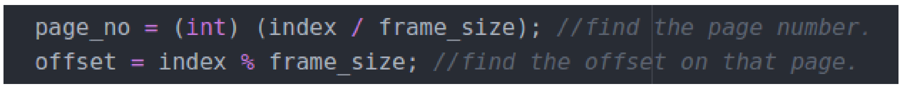


```
else if(strcmp(tName, "Quick") == 0)
{
    index += v_page_table_NRU->num_virtual * v_page_table_NRU-> frame_size / 4;

}
```

Figure 9: Translating address space for quick sort.

4. As we know get function has one input parameter and that's index. In my implementation I didn't work with binary values. Instead of working with binary numbers I used decimal numbers to simplify the process. We know that virtual address consists of offset and page number bits. In my implementation these are two separate integers. Let's say user's input was x for get function and frame size is y. We can simply find the

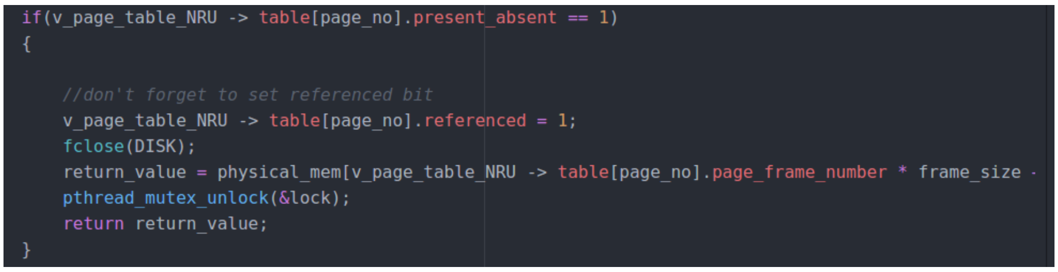
page no with x /y (automatic type casting to integer) and offset with x % y.



```
page_no = (int) (index / frame_size); //find the page number.
offset = index % frame_size; //find the offset on that page.
```

Figure 10: Finding page number and offset.(virtual_mem.c 716)

5. After finding the page number and offset we can check the if the specified page is present in disk or not.



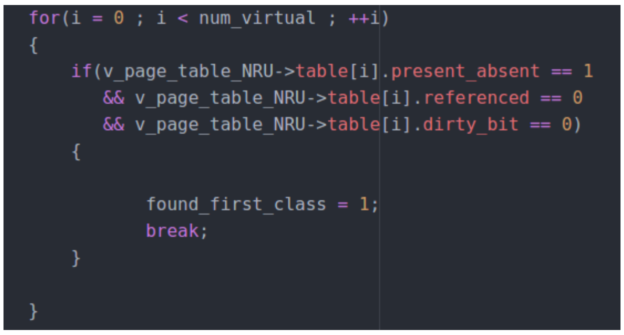
```
if(v_page_table_NRU -> table[page_no].present_absent == 1)
{

    //don't forget to set referenced bit
    v_page_table_NRU -> table[page_no].referenced = 1;
    fclose(DISK);
    return_value = physical_mem[v_page_table_NRU -> table[page_no].page_frame_number * frame_size
    pthread_mutex_unlock(&lock);
    return return_value;
}
```

Figure 11:If the page is present in disk, simply return the specified offset.(virtual_mem.c - 720. line)

6. If the specified page is not present in disk then we increment the page_misses variable and apply one of the page replacement algorithms to find victim page for both local and global policies.
7. There are 4 different page replacement algorithms that I implemented .
    a. **NRU**:
    In Not Recently Used algorithm we divide the pages to four classes and make a selection from lowest class
    Classes are:
        - Not referenced, not modified
        - Not referenced, modified
        - Referenced, not modified
        - Referenced, modified

    This page replacement algorithm requires to flags (Dirty and referenced bits) on page table entries that I already have. We're dividing those classes because if the page is not referenced more likely not referenced again soon so it's not important. If the page is not modified we don't have to write it back to disk therefore cost of eviction is lower. Also we need to refresh all the referenced bits when clock interrupt occurs. I simulated this clock interrupt interval with access count. For example pages referenced bits are refreshed every 10 accesses.

    In my implementation those 4 classes are 4 loops with check. First I'm checking all of the pages for class 0 then for class 1 etc. Implementation can be found in virtual_mem.c – 747.line.



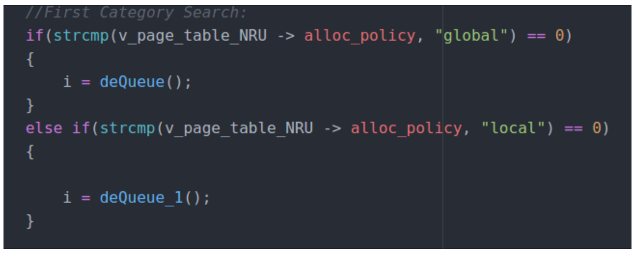
```
for(i = 0 ; i < num_virtual ; ++i)
{
    if(v_page_table_NRU->table[i].present_absent == 1
       && v_page_table_NRU->table[i].referenced == 0
       && v_page_table_NRU->table[i].dirty_bit == 0)
    {

           found_first_class = 1;
           break;
    }

}
```

Figure 12: First class check for NRU

b. FIFO:

This algorithm is simple to implement but not efficient. What OS does is keeps the present pages in Queue in FIFO manner. Since usage of the page is ignored it has poor performance. Also "Belady's Anomaly" can be clearly experienced with FIFO algorithm.

I implemented myQueue with simple C array that holds front and rear locations. When we decide to find a victim page simply we'll dequeue that page from queue and we're node.

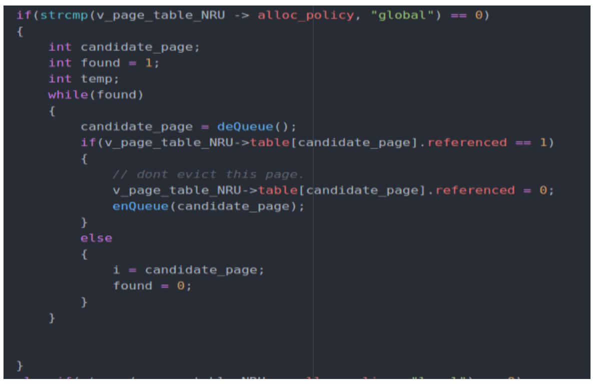


```
//First Category Search:
if(strcmp(v_page_table_NRU -> alloc_policy, "global") == 0)
{
    i = deQueue();
}
else if(strcmp(v_page_table_NRU -> alloc_policy, "local") == 0)
{

    i = deQueue_1();
}
```

Figure 13: Finding the victim page with FIFO manner.

c. Second Chance:

This algorithm is very similar to FIFO the only difference is when choosing the victim page if the dequeued page has valid R bit then **don't** evict it. Reset the R bit of that page and add back to queue again. In other words move that page to rear of the queue. With this mechanism we provide a second chance. This algorithms worst case if the all pages are referenced.



```
if(strcmp(v_page_table_NRU -> alloc_policy, "global") == 0)
{
    int candidate_page;
    int found = 1;
    int temp;
    while(found)
    {
        candidate_page = deQueue();
        if(v_page_table_NRU->table[candidate_page].referenced == 1)
        {
            // dont evict this page.
            v_page_table_NRU->table[candidate_page].referenced = 0;
            enQueue(candidate_page);
        }
        else
        {
            i = candidate_page;
            found = 0;
        }
    }


}
```

Figure 14: Second chance algorithm(virtual_mem.c 2324. line)

d. LRU:

Least Recently Used algorithm focuses on choosing victim that page has been unused for the longest time. LRU is good algorithm but impractical to implement on software. I implemented LRU algorithm with the help of page matrix in Tanenbaum's book. (virtual_mem.c 6667. line) The algorithms is:

- For n page entries keep n * n integer array.
- If a reference made to page i, set all i. row with 1's and columns to 0
- When we decide to evict a page we'll check the fewest 1's.

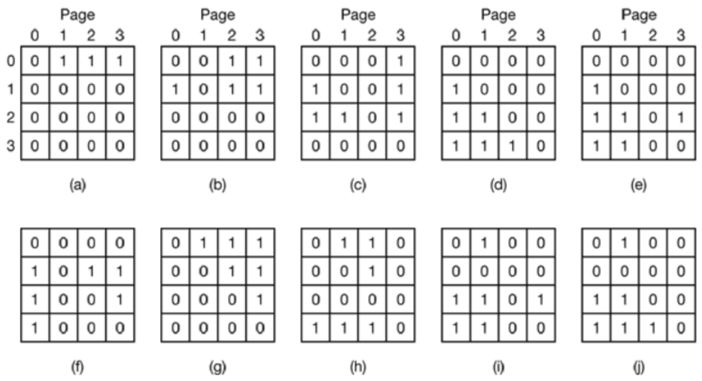
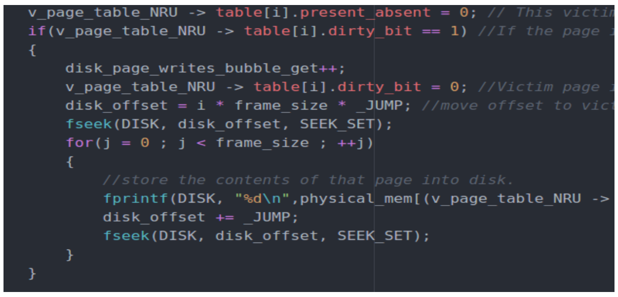


Figure 15: Tanenbaum's way to handle LRU (from book)

8. After finding the victim page we will replace that page with our new page. There are two options, we can always write back the page to disk which is easy to implement either we can write victim page to disk if it's not modified. I didn't make unnecessary writes therefore checked the dirty bit and wrote on disk if it's necessary.



```
v_page_table_NRU -> table[i].present_absent = 0; // This victi
if(v_page_table_NRU -> table[i].dirty_bit == 1) //If the page
{
    disk_page_writes_bubble_get++;
    v_page_table_NRU -> table[i].dirty_bit = 0; //Victim page
    disk_offset = i * frame_size * _JUMP; //move offset to vic
    fseek(DISK, disk_offset, SEEK_SET);
    for(j = 0 ; j < frame_size ; ++j)
    {
        //store the contents of that page into disk.
        fprintf(DISK, "%d\n",physical_mem[(v_page_table_NRU ->
        disk_offset += _JUMP;
        fseek(DISK, disk_offset, SEEK_SET);
    }
}
```

Figure 16: Writing victim page to disk if necessary (virtual_mem.c 916)

Set Method:

Set method's code section is 99 % same as get method so all 8 matters of get method is same for set method.

Printing results and freeing resources with check function:

After threads are finished executing main threads reports reads, writes, page misses, page replacements, disk page writes, disk page reads for each sorting algorithm. It also prints the sorted array for checking purposes. Since my fill function works as main thread page_misses and page_replacements will always have same value. Because threads starting working when physical memory is full. The missed page must be replaced so.

Allocation Policy issues:

When filling the virtual memory with another thread I realized that most of times one quarter of virtual memory doesn’t even have one page. Since local allocation policy is focuses on isolation it only replaces the pages from it’s own section. When that process can’t find any available present pages on memory then it can do nothing. In order to deal with problem for local allocation policy. I distributed pages evenly for each quarters.

Simulating Clock Interrupt

You might realise that I don’t have any field about clock values in my page table entry struct. There are no fields like that because I decided to simulate that interrupt when constant amount of references are made. I used *pageTablePrintInt* to set that interval. For example if that interval is 100. Every 100. Reference clock interrupt will occur and it will refresh those referenced bits.

Source Code Design and Usage:

There is a driver code named with *sortArrays.c* that starts the threads and reports the statistics at the end of program. Structures like page table and helper function’s prototypes kept in a header file named with *virtual_mem.h* and all implementations of threads, these functions etc. kept in *virtual_mem.c* source file. Also Makefile provided for building the program.

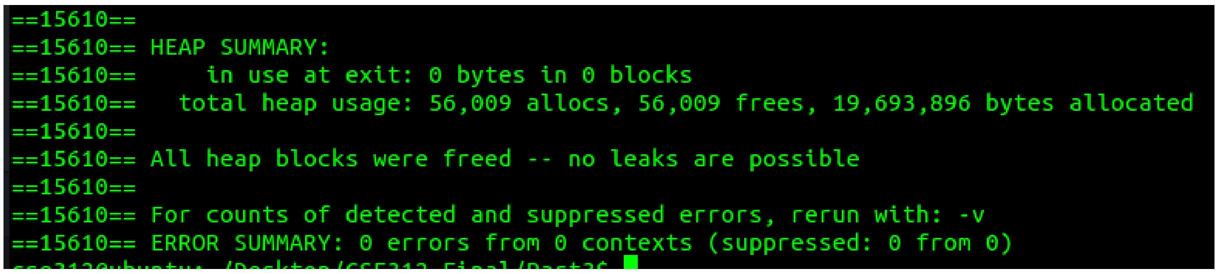
After building program with command *make* You can run the program as : *./sortArrays frameSize numPhysical numVirtual pageReplacement allocPolicy pageTablePrintInt filename.*

Simulating Disk File:

In order to store pages in disk and jump to the specified offsets. I kept my integers in file with my own format. I used a lot of standart library functions like fseek etc. I wrote single integer to file then seeked for a known byte count. This created some kind of boundary for each integer.(Like cells in memory.)

Program Quality:

Unfortunately there are many duplicated code in common get/set functions but program is not vulnerable to bad input's or some kind of memory leaks.



```
==15610==
==15610== HEAP SUMMARY:
==15610==     in use at exit: 0 bytes in 0 blocks
==15610==   total heap usage: 56,009 allocs, 56,009 frees, 19,693,896 bytes allocated
==15610==
==15610== All heap blocks were freed -- no leaks are possible
==15610==
==15610== For counts of detected and suppressed errors, rerun with: -v
==15610== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

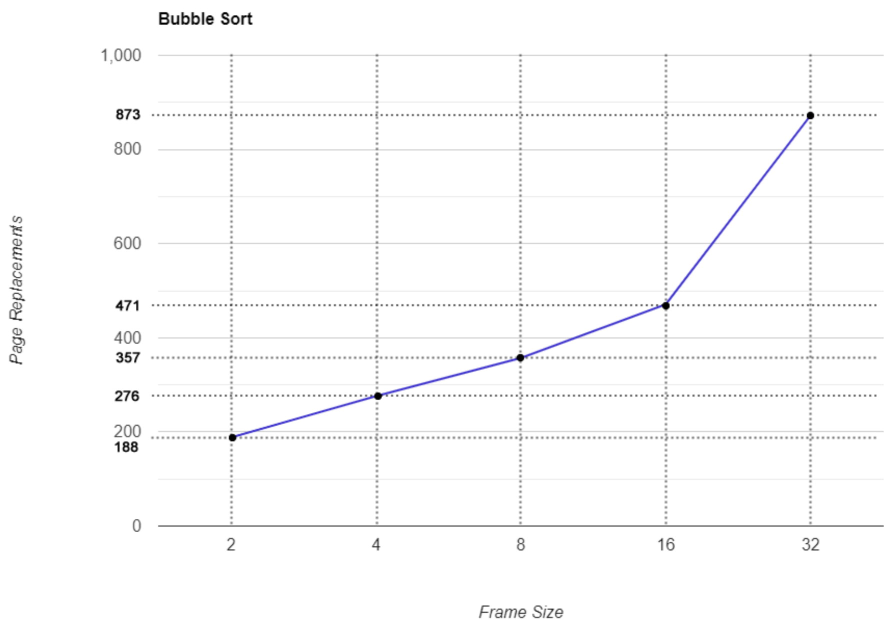
Figure 17: Valgrind check for sortArrays program

Optimal Page Size for Sorting Algorithms

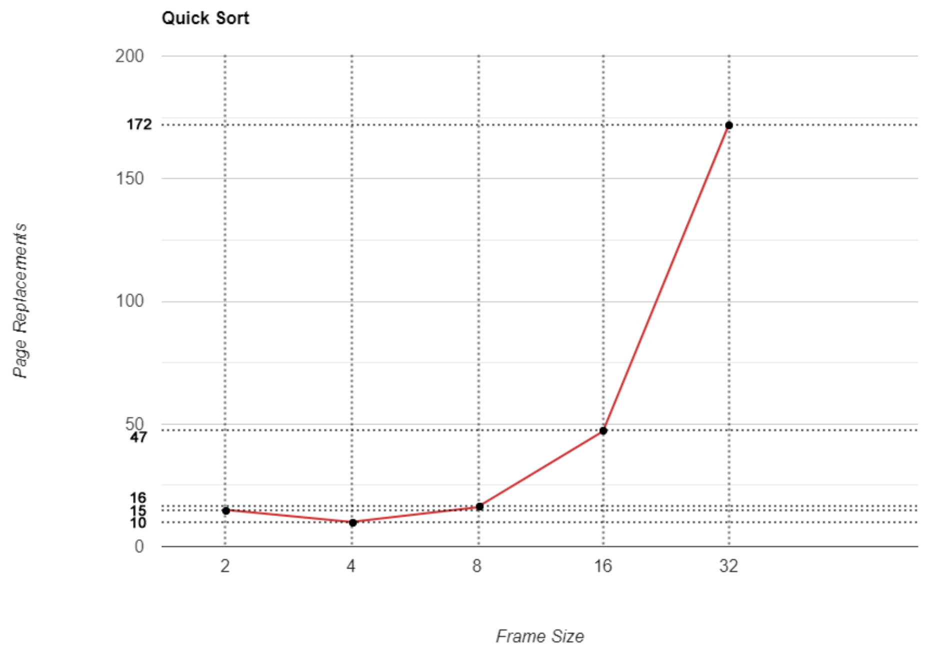
I ran 50 different page size experiments to find optimal one. In order to get quick results, my inputs were not that big but it's also works 128k virtual and 16k physical pages.

Graphs for Each Sorting Algorithms:

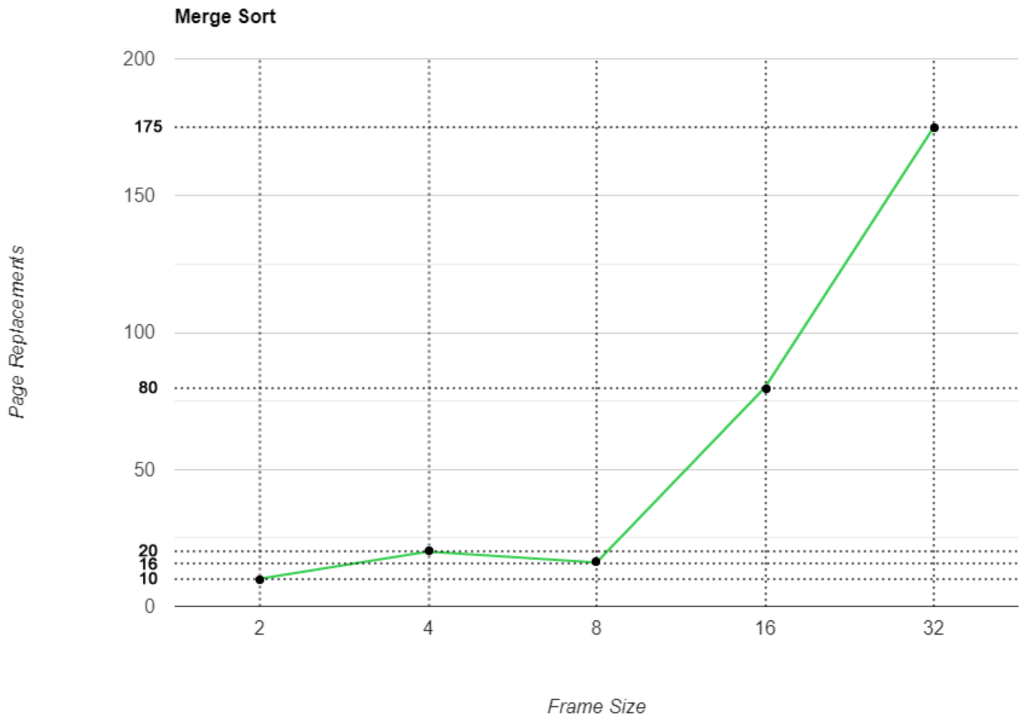
Bubble Sort:



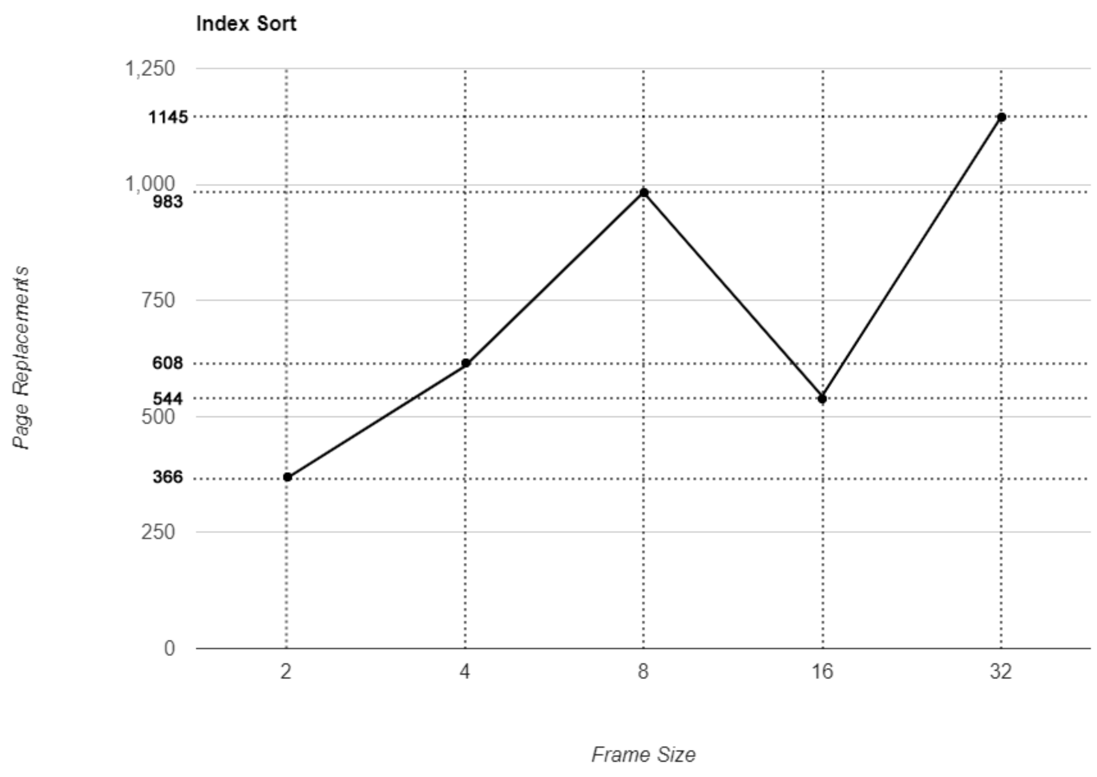
Quick Sort:



Merge Sort:



Index Sort



Discussion About Graphs:

The graphs are accurate with program in part3. In my experiment for quick and merge sort larger page sizes are not great. For bubble and index sort it's exponentially increases with frame size increase.

References:

- https://ourmachinery.com/post/virtual-memory-tricks/
- https://cs.nyu.edu/courses/spring03/V22.0202-002/lecture-09.html
- https://slideplayer.com/slide/1659993/