CSE312 OPERATING SYSTEMS

HW2 REPORT

COŞKUN HASAN ŞALTU

1801042631

1) PROBLEM SOLUTION APPROACH

Virtual Memory

Virtual memory is a memory management technique used in computer systems to provide the illusion of having more memory than is physically available. It allows programs to operate as if they have access to a large, contiguous block of memory, even if the physical memory is limited.

In a virtual memory system, the memory space is divided into fixed-size blocks called pages. Similarly, the physical memory is divided into fixed-size blocks called frames. The virtual memory and physical memory are mapped to each other through a page table, which keeps track of the mapping between virtual pages and physical frames.

When a program requests memory, it operates on virtual addresses, which are translated to physical addresses by the memory management unit (MMU) using the page table. If the required page is already present in physical memory, the MMU translates the virtual address to the corresponding physical address, and the program can access the data directly. This scenario is known as a "page hit."

If a requested page is not present in physical memory, it leads to a "page fault." In this case, the operating system needs to bring the required page from the secondary storage (usually the disk) into a free frame in physical memory. This involves evicting a page from physical memory if all frames are occupied. The evicted page is typically written back to disk if it has been modified (dirty).

Virtual memory provides several advantages, including:

- Increased memory capacity: It allows programs to address more memory than is physically available.

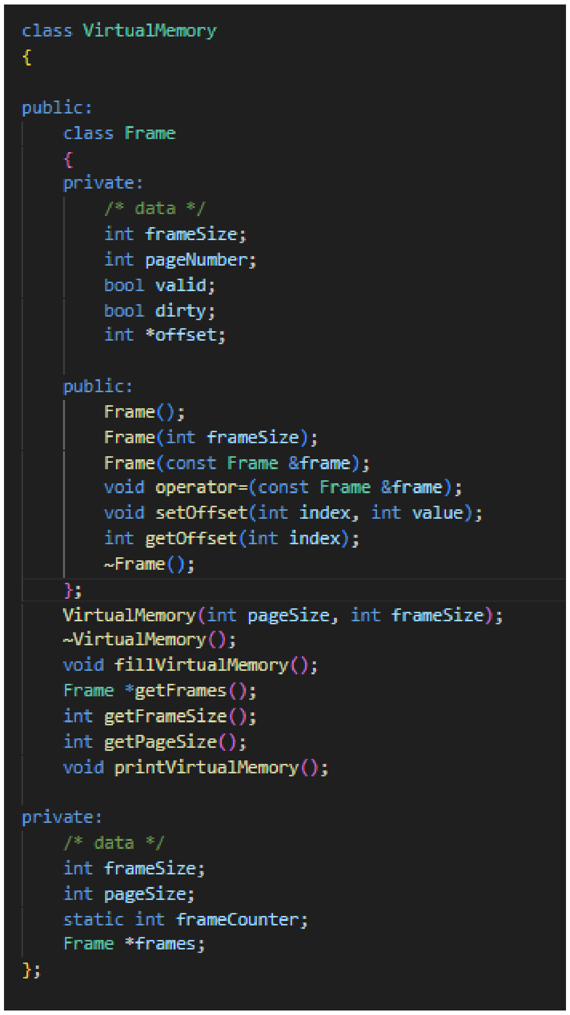
- Memory isolation: Each program operates within its own virtual memory space, protecting it from interference by other programs.

- Simplified memory management: Programs can be loaded and executed without worrying about the availability of contiguous free memory blocks.

- Efficient memory allocation: Memory can be allocated in small, fixed-size units (pages), reducing internal fragmentation.

However, virtual memory also introduces additional complexities, such as the overhead of page table lookups and the potential for increased disk I/O due to page faults. Proper management of virtual memory, including page replacement algorithms and disk swapping strategies, is essential for optimizing system performance.

Virtual memory class:



```cpp
class VirtualMemory
{

public:
    class Frame
    {
    private:
        /* data */
        int frameSize;
        int pageNumber;
        bool valid;
        bool dirty;
        int *offset;

    public:
        Frame();
        Frame(int frameSize);
        Frame(const Frame &frame);
        void operator=(const Frame &frame);
        void setOffset(int index, int value);
        int getOffset(int index);
        ~Frame();
    };
    VirtualMemory(int pageSize, int frameSize);
    ~VirtualMemory();
    void fillVirtualMemory();
    Frame *getFrames();
    int getFrameSize();
    int getPageSize();
    void printVirtualMemory();

private:
    /* data */
    int frameSize;
    int pageSize;
    static int frameCounter;
    Frame *frames;
};
```

The `VirtualMemory` class represents a virtual memory system. It contains a nested `Frame` class, which represents a single frame within the virtual memory.

The `VirtualMemory` class has the following member functions and variables:

- `Frame` class: This nested class represents a frame within the virtual memory. It has private member variables such as `frameSize` (the size of the frame), `pageNumber` (the page number associated with the frame), `valid` (a flag indicating if the frame is valid), `dirty` (a flag indicating if the frame has been modified), and `offset` (an array storing the data within the frame). It also has member functions such as the default constructor, parameterized constructor, copy constructor, assignment operator, `setOffset` (to set the value at a specific offset within the frame), `getOffset` (to retrieve the value at a specific offset), and the destructor.

- `frameSize`: An integer variable representing the size of each frame within the virtual memory.

- `pageSize`: An integer variable representing the size of each page within the virtual memory.

- `frameCounter`: A static integer variable used to keep track of the number of frames created in the virtual memory system.

- `frames`: An array of `Frame` objects representing the frames in the virtual memory.

The `VirtualMemory` class has the following member functions:

- `VirtualMemory(int pageSize, int frameSize)`: A constructor that initializes the virtual memory with the given page size and frame size. It dynamically allocates memory for the frames array based on the number of frames required.

- `~VirtualMemory()`: A destructor that frees the dynamically allocated memory for the frames array.

- `fillVirtualMemory()`: A function that fills the virtual memory with data. This function is not implemented in the provided code snippet, but it is likely used to populate the virtual memory with initial values.

- `getFrames()`: A function that returns a pointer to the array of `Frame` objects representing the frames in the virtual memory.

- `getFrameSize()`: A function that returns the size of each frame in the virtual memory.

- `getPageSize()`: A function that returns the size of each page in the virtual memory.

- `printVirtualMemory()`: A function that prints the contents of the virtual memory. This function is not implemented in the provided code snippet, but it can be used to display the data stored in each frame of the virtual memory.

Page Table

A page table is a data structure used in virtual memory systems to manage the mapping between virtual memory addresses and physical memory addresses. It is an essential component of memory management in modern computer systems.

In a virtual memory system, the memory address space of a process is divided into fixed-size blocks called pages. Similarly, the physical memory (RAM) is divided into fixed-size blocks called frames. The page table maintains the mapping between each virtual page and its corresponding physical frame.

The page table consists of page entries, with each entry representing a single page of virtual memory. Each page entry typically contains the following information:

- Virtual page number: The unique identifier for the virtual page.

- Physical frame number: The identifier for the physical frame where the corresponding page is stored.

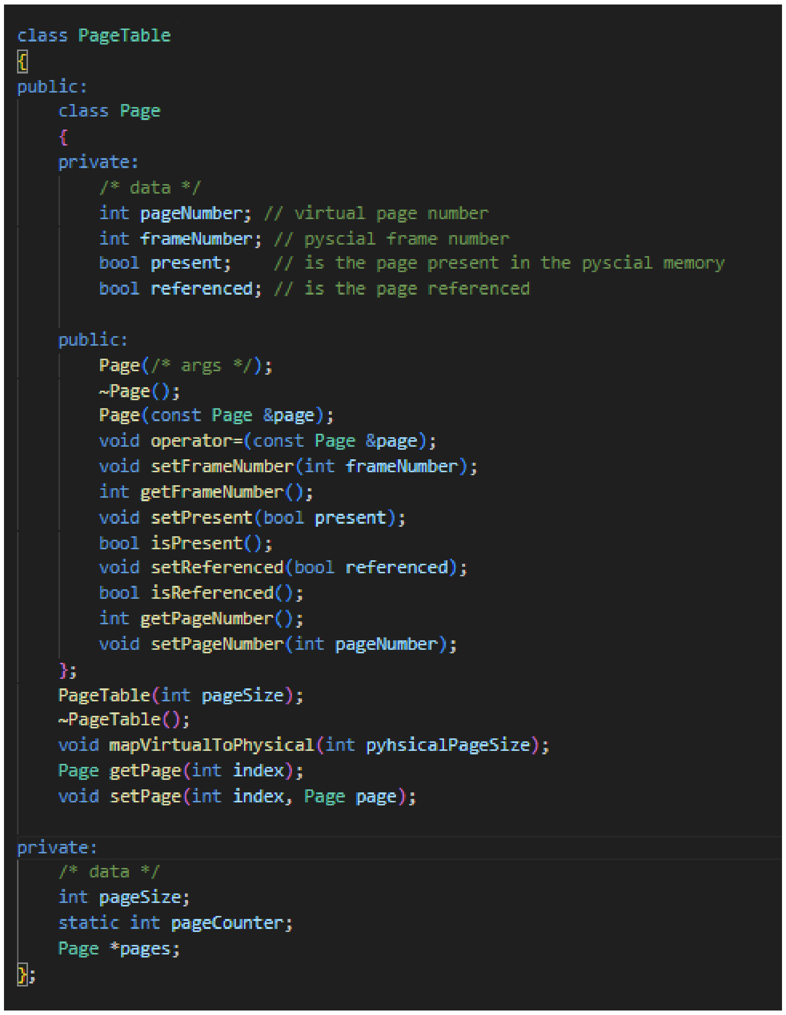
- Present bit: A flag indicating whether the page is currently present in physical memory.

- Referenced bit: A flag indicating whether the page has been referenced (accessed) recently.

When a program references a virtual memory address, the memory management unit (MMU) uses the page table to translate the virtual address to the corresponding physical address. If the requested page is present in physical memory (indicated by the present bit), the MMU retrieves the physical address and accesses the data directly. This is known as a "page hit." If the requested page is not present in physical memory (indicated by the absence of the present bit), it results in a "page fault," and the operating system needs to bring the required page from secondary storage (e.g., disk) into an available physical frame.

The page table is typically maintained by the operating system on a per-process basis. The OS is responsible for managing page table entries, updating them during page faults, and allocating physical frames when necessary. Various page replacement algorithms, such as LRU (Least Recently Used) or FIFO (First-In-First-Out), are used to select which pages to evict from physical memory when all frames are occupied.

Page Table Class:



```
class PageTable
{
public:
    class Page
    {
    private:
        /* data */
        int pageNumber; // virtual page number
        int frameNumber; // pyscial frame number
        bool present;    // is the page present in the pyscial memory
        bool referenced; // is the page referenced

    public:
        Page(/* args */);
        ~Page();
        Page(const Page &page);
        void operator=(const Page &page);
        void setFrameNumber(int frameNumber);
        int getFrameNumber();
        void setPresent(bool present);
        bool isPresent();
        void setReferenced(bool referenced);
        bool isReferenced();
        int getPageNumber();
        void setPageNumber(int pageNumber);
    };
    PageTable(int pageSize);
    ~PageTable();
    void mapVirtualToPhysical(int pyhsicalPageSize);
    Page getPage(int index);
    void setPage(int index, Page page);

private:
    /* data */
    int pageSize;
    static int pageCounter;
    Page *pages;
};
```

The `PageTable` class represents a page table used in a virtual memory system. It contains a nested `Page` class, which represents a single page entry within the page table.

The `PageTable` class has the following member functions and variables:

- `Page` class: This nested class represents a page entry within the page table. It has private member variables such as `pageNumber` (the virtual page number associated with the entry), `frameNumber` (the physical frame number mapped to the page), `present` (a flag indicating if the page is present in physical memory), and `referenced` (a flag indicating if the page has been referenced). It also has member functions such as the default constructor, destructor, copy constructor, assignment operator, `setFrameNumber` (to set the physical frame number), `getFrameNumber` (to retrieve the physical frame number), `setPresent` (to set the presence flag), `isPresent` (to check if the page is present), `setReferenced` (to set the referenced flag), `isReferenced` (to check if the page has been referenced), `getPageNumber` (to retrieve the virtual page number), and `setPageNumber` (to set the virtual page number).

- `pageSize`: An integer variable representing the size of each page in the page table.

- `pageCounter`: A static integer variable used to keep track of the number of pages in the page table.

- `pages`: An array of `Page` objects representing the page entries in the page table.

The `PageTable` class has the following member functions:

- `PageTable(int pageSize)`: A constructor that initializes the page table with the given page size. It dynamically allocates memory for the page entries based on the number of pages required.

- `~PageTable()`: A destructor that frees the dynamically allocated memory for the page entries.

- `mapVirtualToPhysical(int physicalPageSize)`: A function that maps the virtual page numbers to physical frame numbers. This function is not implemented in the provided code snippet, but it would typically involve setting the `frameNumber` for each page entry based on the physical page size.

- `getPage(int index)`: A function that returns a specific `Page` object from the page table based on the given index.

- `setPage(int index, Page page)`: A function that sets a specific `Page` object in the page table based on the given index.

Physical Memory

Physical memory, also known as main memory or RAM (Random Access Memory), refers to the actual hardware component in a computer system where data and instructions are stored for immediate access by the CPU (Central Processing Unit). It is a volatile form of memory, meaning its contents are lost when the computer is powered off.

Physical memory is organized into fixed-size units called memory cells or memory locations, each capable of storing a certain amount of binary data. These memory cells are further grouped into contiguous blocks called memory addresses. The size of a memory cell is typically represented in bytes, and the capacity of physical memory is measured in terms of the total number of memory cells it can accommodate.

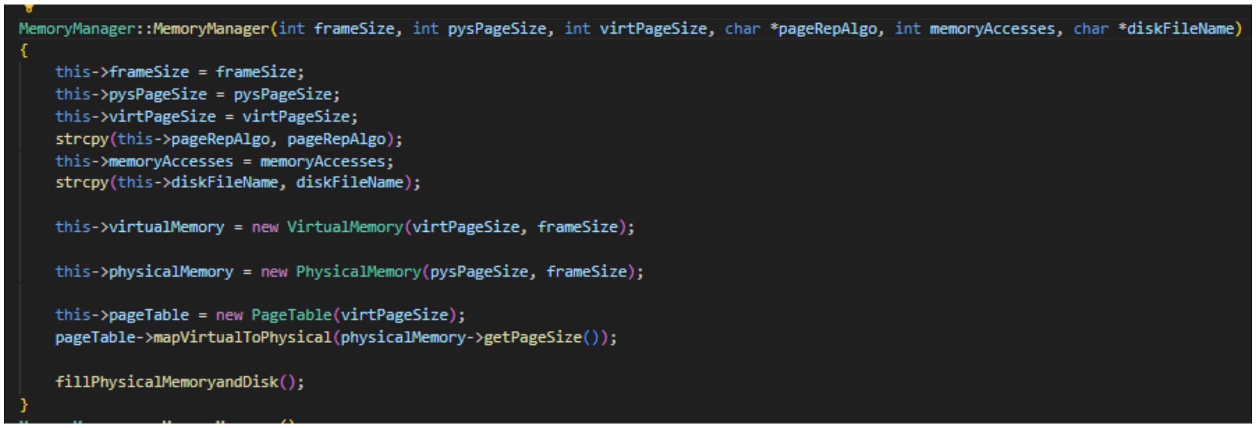
The purpose of physical memory is to hold the data and instructions that the CPU needs to process during program execution. When a program is running, its instructions and data are loaded into physical memory from secondary storage (e.g., hard disk) to facilitate faster access and execution. The CPU interacts with physical memory by reading from and writing to specific memory addresses to retrieve or store data.

Physical memory operates at a much faster speed compared to secondary storage devices, such as hard drives or solid-state drives, allowing for quick access to data by the CPU. The memory cells in physical memory are arranged in a linear fashion, and each cell is uniquely identified by a memory address. The CPU uses memory addresses to access and manipulate data in physical memory.

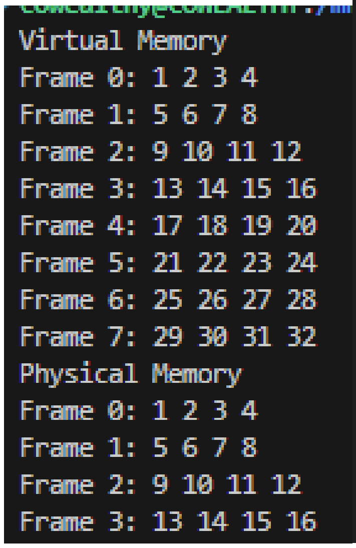
To manage physical memory efficiently, the operating system employs techniques such as memory allocation and deallocation, memory paging, and caching. These techniques ensure that the available physical memory is utilized optimally and that data is stored and retrieved in the most efficient manner possible.

Road Map

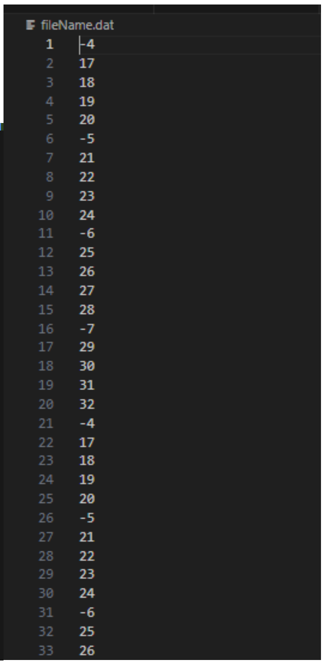
First of all, memories were created according to the determined sizes. Then the virtual memory is filled with random numbers. Then, which virtual page will display which pyshical page is mapped in the page table. Finally, pyhsical memory is filled according to virtual memory and addresses that do not fit in pyscial memory are moved to disk.



```cpp
MemoryManager::MemoryManager(int frameSize, int pysPageSize, int virtPageSize, char *pageRepAlgo, int memoryAccesses, char *diskFileName)
{
    this->frameSize = frameSize;
    this->pysPageSize = pysPageSize;
    this->virtPageSize = virtPageSize;
    strcpy(this->pageRepAlgo, pageRepAlgo);
    this->memoryAccesses = memoryAccesses;
    strcpy(this->diskFileName, diskFileName);

    this->virtualMemory = new VirtualMemory(virtPageSize, frameSize);

    this->physicalMemory = new PhysicalMemory(pysPageSize, frameSize);

    this->pageTable = new PageTable(virtPageSize);
    pageTable->mapVirtualToPhysical(physicalMemory->getPageSize());

    fillPhysicalMemoryandDisk();
}
```



```
Virtual Memory
Frame 0: 1 2 3 4
Frame 1: 5 6 7 8
Frame 2: 9 10 11 12
Frame 3: 13 14 15 16
Frame 4: 17 18 19 20
Frame 5: 21 22 23 24
Frame 6: 25 26 27 28
Frame 7: 29 30 31 32
Physical Memory
Frame 0: 1 2 3 4
Frame 1: 5 6 7 8
Frame 2: 9 10 11 12
Frame 3: 13 14 15 16
```



fileName.dat

```
-4
17
18
19
20
-5
21
22
23
24
-6
25
26
27
28
-7
29
30
31
32
-4
17
18
19
20
-5
21
22
23
24
-6
25
26
```

Then, it was necessary to pull data from the disk in case of page faults. **Second Chance** page replacement algorithm is used to extract data from the disk.

Second Chance

The Second Chance algorithm is a page replacement algorithm used in virtual memory systems to determine which pages should be replaced when there is a page fault (i.e., a requested page is not present in the physical memory).

The algorithm is based on the idea of giving each page a "second chance" before being selected for replacement. It uses a reference bit, typically associated with each page in the page table, to track whether a page has been recently referenced or not.

The basic steps of the Second Chance algorithm are as follows:

1. Each page in the page table is assigned a reference bit, which is initially set to 0 or false.

2. When a page fault occurs and a new page needs to be brought into physical memory, the page replacement algorithm examines the pages in a specific order (e.g., based on a queue or a clock hand).

3. The algorithm checks the reference bit of the current page being examined. If the reference bit is set to 1 or true, it means the page has been recently referenced.

4. In this case, the reference bit is cleared (set to 0 or false), indicating that the page has received a "second chance" and should not be immediately replaced.

5. The algorithm then moves to the next page and repeats the process until it finds a page with a reference bit of 0 or false.

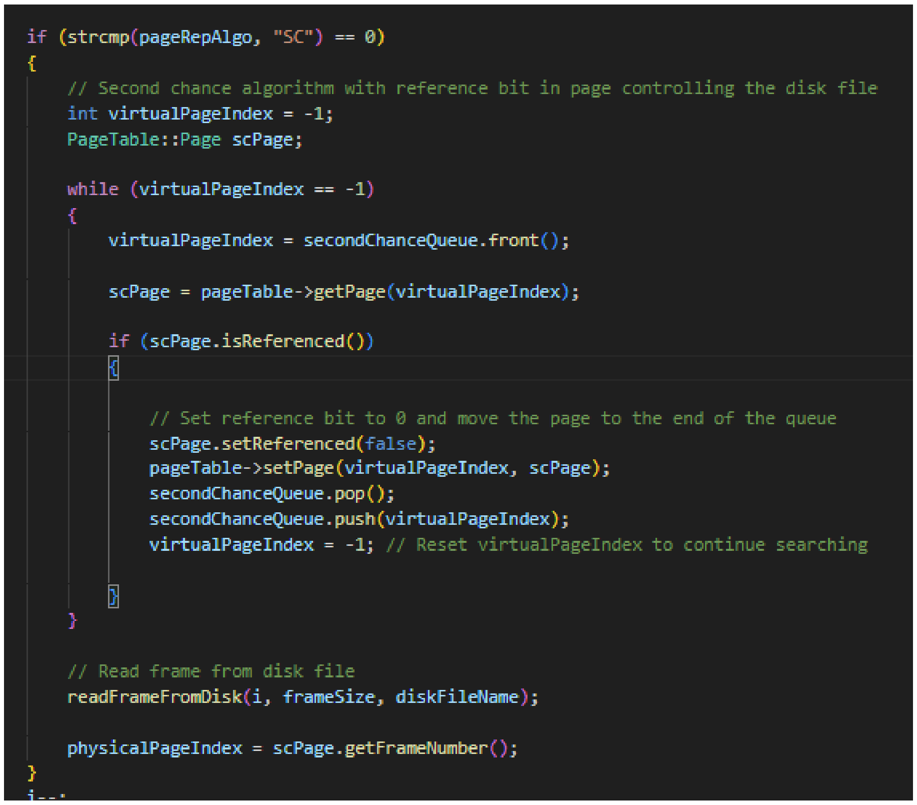
6. When a page with a reference bit of 0 is found, it is selected for replacement, and the new page is brought into the freed frame in the physical memory.

7. The replaced page may be written back to disk if it has been modified (dirty bit is set).

8. Finally, the reference bit of the replaced page is set to 0, as it will be given a fresh chance if it is referenced again in the future.

The Second Chance algorithm combines the advantages of both FIFO (First-In-First-Out) and LRU (Least Recently Used) algorithms. It gives recently referenced pages a second opportunity to be accessed before being replaced, effectively mimicking a FIFO strategy but with some level of preference for recently referenced pages.

My Algorithm:



```
if (strcmp(pageRepAlgo, "SC") == 0)
{
    // Second chance algorithm with reference bit in page controlling the disk file
    int virtualPageIndex = -1;
    PageTable::Page scPage;

    while (virtualPageIndex == -1)
    {
        virtualPageIndex = secondChanceQueue.front();

        scPage = pageTable->getPage(virtualPageIndex);

        if (scPage.isReferenced())
        {

            // Set reference bit to 0 and move the page to the end of the queue
            scPage.setReferenced(false);
            pageTable->setPage(virtualPageIndex, scPage);
            secondChanceQueue.pop();
            secondChanceQueue.push(virtualPageIndex);
            virtualPageIndex = -1; // Reset virtualPageIndex to continue searching

        }
    }

    // Read frame from disk file
    readFrameFromDisk(i, frameSize, diskFileName);

    physicalPageIndex = scPage.getFrameNumber();
}
i--;
```

The algorithm works as follows:

1. Check if the page replacement algorithm being used is "SC" (Second Chance) by comparing the `pageRepAlgo` variable with the string "SC".

2. If the algorithm matches, initialize the `virtualPageIndex` variable to -1 and create a `PageTable::Page` object called `scPage` to hold the current page information.

3. Enter a loop that continues until a suitable virtual page is found.

4. Get the front element of the `secondChanceQueue`, which represents the index of a virtual page that needs to be examined.

5. Retrieve the corresponding page from the page table using `pageTable->getPage(virtualPageIndex)` and assign it to the `scPage` object.

6. Check if the reference bit of the `scPage` is set to true (referenced).

7. If the reference bit is set, it means the page has been referenced, so its reference bit is cleared (set to false), the updated page is stored back in the page table using `pageTable->setPage(virtualPageIndex, scPage)`, and the page index is moved to the end of the `secondChanceQueue` to give it another chance for reference.

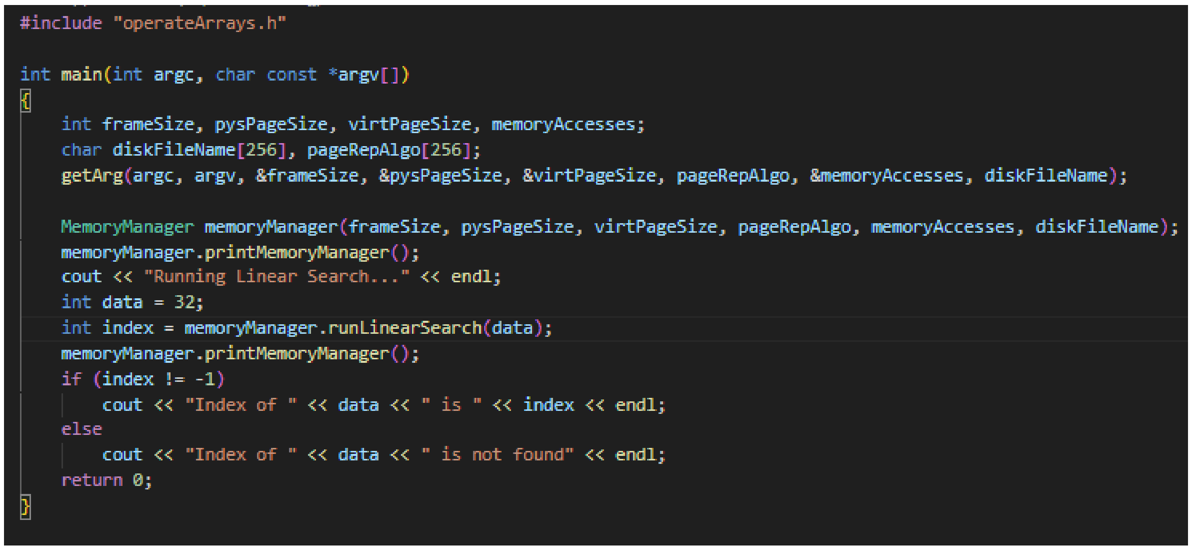
8. Reset the `virtualPageIndex` to -1 to continue searching for a suitable page.

9. Once a suitable virtual page is found, read the corresponding frame from the disk file using the `readFrameFromDisk` function, passing the `i` index, `frameSize`, and `diskFileName` as parameters.

10. Set the `physicalPageIndex` variable to the frame number of the `scPage`, which represents the physical page index where the page will be stored in physical memory.

2) Test Cases

This is the driver:



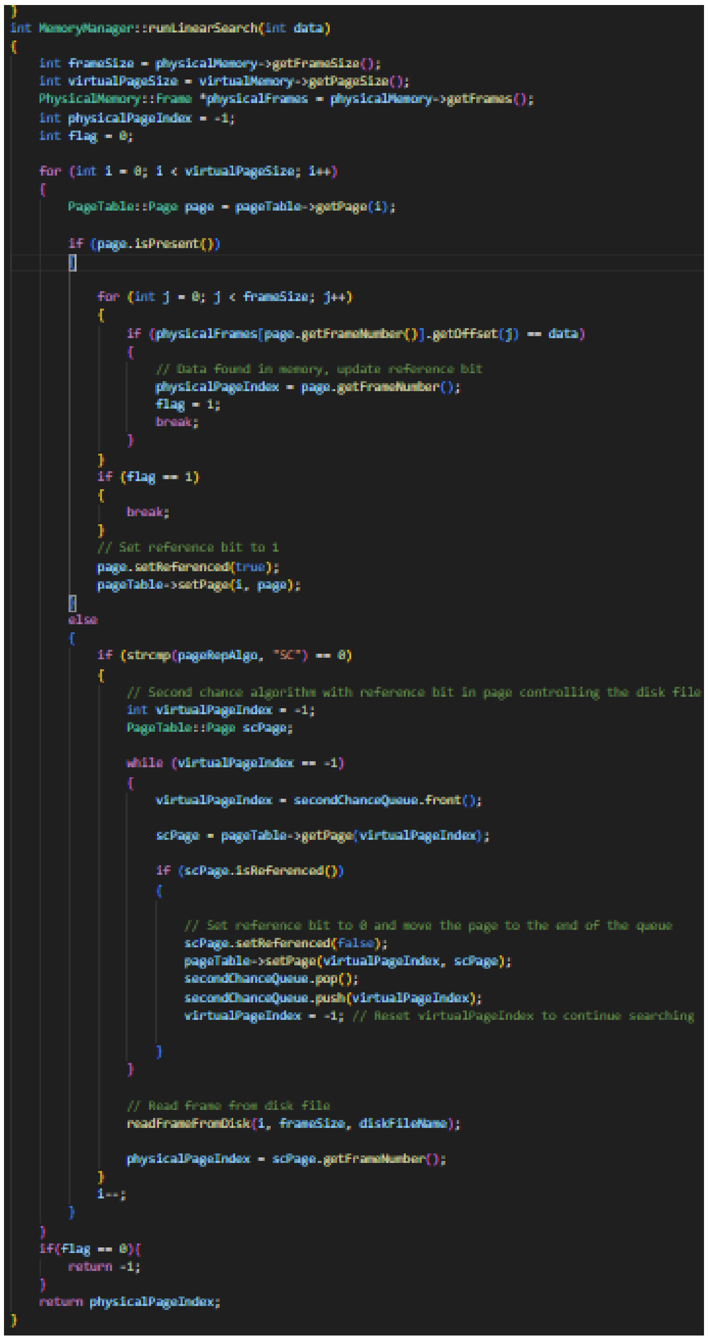
```
#include "operateArrays.h"

int main(int argc, char const *argv[])
{
    int frameSize, pysPageSize, virtPageSize, memoryAccesses;
    char diskFileName[256], pageRepAlgo[256];
    getArg(argc, argv, &frameSize, &pysPageSize, &virtPageSize, pageRepAlgo, &memoryAccesses, diskFileName);

    MemoryManager memoryManager(frameSize, pysPageSize, virtPageSize, pageRepAlgo, memoryAccesses, diskFileName);
    memoryManager.printMemoryManager();
    cout << "Running Linear Search..." << endl;
    int data = 32;
    int index = memoryManager.runLinearSearch(data);
    memoryManager.printMemoryManager();
    if (index != -1)
        cout << "Index of " << data << " is " << index << endl;
    else
        cout << "Index of " << data << " is not found" << endl;
    return 0;
}
```

The usage of this program is:

"Usage: ./operateArrays <frameSize> <physicalMemorySize> <virtualMemorySize> <pageReplacementAlgo> <memoryAccesses> <diskFileName>"

The program executable just Second Chance algorithm.

The program is tried with lineer search:



```
}
int MemoryManager::runLinearSearch(int data)
{
    int frameSize = physicalMemory->getFrameSize();
    int virtualPageSize = virtualMemory->getPageSize();
    PhysicalMemory::Frame *physicalFrames = physicalMemory->getFrames();
    int physicalPageIndex = -1;
    int flag = 0;

    for (int i = 0; i < virtualPageSize; i++)
    {
        PageTable::Page page = pageTable->getPage(i);

        if (page.isPresent())
        {

            for (int j = 0; j < frameSize; j++)
            {
                if (physicalFrames[page.getFrameNumber()].getOffset(j) == data)
                {
                    // Data found in memory, update reference bit
                    physicalPageIndex = page.getFrameNumber();
                    flag = 1;
                    break;
                }
            }
            if (flag == 1)
            {
                break;
            }
            // Set reference bit to 1
            page.setReferenced(true);
            pageTable->setPage(i, page);
        }
        else
        {
            if (strcmp(pageRepAlgo, "SC") == 0)
            {
                // Second chance algorithm with reference bit in page controlling the disk file
                int virtualPageIndex = -1;
                PageTable::Page scPage;

                while (virtualPageIndex == -1)
                {
                    virtualPageIndex = secondChanceQueue.front();

                    scPage = pageTable->getPage(virtualPageIndex);

                    if (scPage.isReferenced())
                    {

                        // Set reference bit to 0 and move the page to the end of the queue
                        scPage.setReferenced(false);
                        pageTable->setPage(virtualPageIndex, scPage);
                        secondChanceQueue.pop();
                        secondChanceQueue.push(virtualPageIndex);
                        virtualPageIndex = -1; // Reset virtualPageIndex to continue searching

                    }
                }

                // Read frame from disk file
                readFrameFromDisk(i, frameSize, diskFileName);

                physicalPageIndex = scPage.getFrameNumber();
            }
            i--;
        }
    }
    if(flag == 0){
        return -1;
    }
    return physicalPageIndex;
}
```

Here is an explanation of the algorithm:

1. Initialize variables such as `frameSize` (size of a frame in the physical memory), `virtualPageSize` (number of pages in the virtual memory), `physicalFrames` (array of frames in the physical memory), `physicalPageIndex` (index of the found data in the physical memory), and `flag` (a flag to track if the data is found).

2. Iterate through the virtual pages from 0 to `virtualPageSize - 1`.

3. Check if the page at the current index is present in the physical memory (using the `isPresent()` method of the `PageTable::Page` class).

4. If the page is present, iterate through the offsets in the frame to search for the desired data. If the data is found, update `physicalPageIndex` with the frame number and set `flag` to 1 to indicate that the data is found.

5. If the data is not found, set the reference bit of the page to 1 (using the `setReferenced(true)` method of the `PageTable::Page` class) to indicate that the page has been referenced.

6. If the page is not present in the physical memory, and the page replacement algorithm chosen is "SC" (Second Chance), execute the Second Chance algorithm.

7. In the Second Chance algorithm, repeatedly retrieve the front page index from the `secondChanceQueue` and get the corresponding page from the page table.

8. Check if the page's reference bit is set to 1. If so, reset the reference bit to 0 (using the `setReferenced(false)` method), move the page to the end of the queue by popping and pushing it again, and reset the `virtualPageIndex` to -1 to continue the search.

9. If the page's reference bit is 0, indicating it has not been recently referenced, proceed with page replacement.

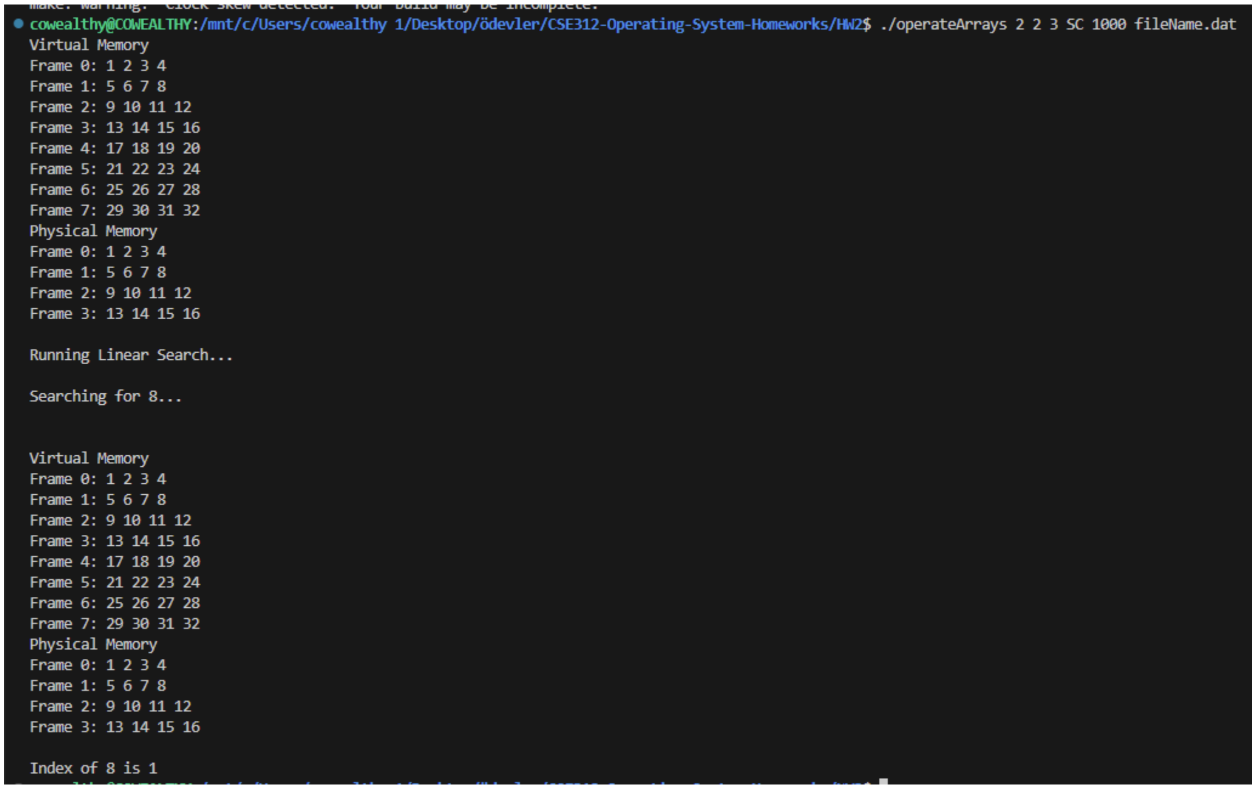
10. Read the frame from the disk file into the physical memory using the `readFrameFromDisk` function.

11. Update `physicalPageIndex` with the frame number of the replaced page obtained from `scPage.getFrameNumber()`.

12. After finding the data or replacing a page, if `flag` is still 0, it means the data was not found in the physical memory. In this case, return -1 to indicate that the data was not found.

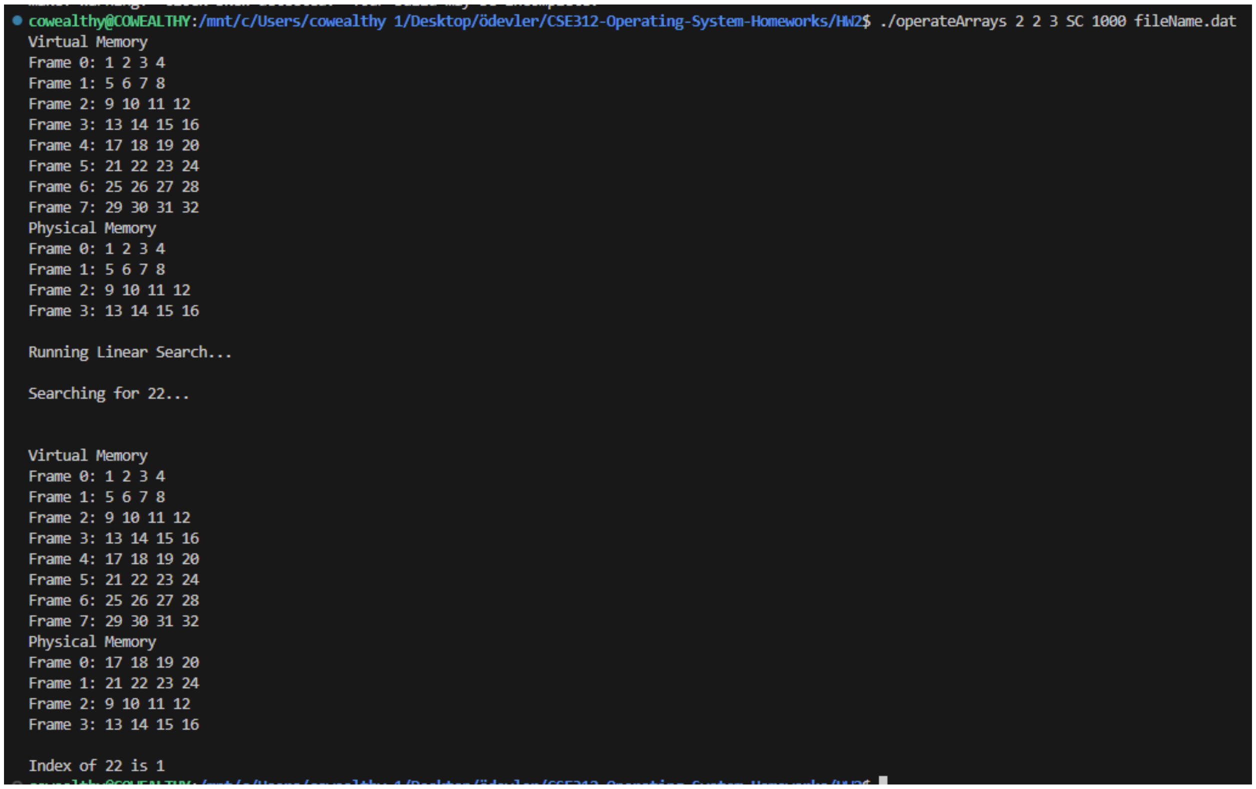
13. Return `physicalPageIndex`, which holds the index of the found data in the physical memory.

The program is tried lineer search with 8. 8 is already putted in physical memory. So, there is no page fault. Physical memory is not updated. The program returns just index of the 8 addres.



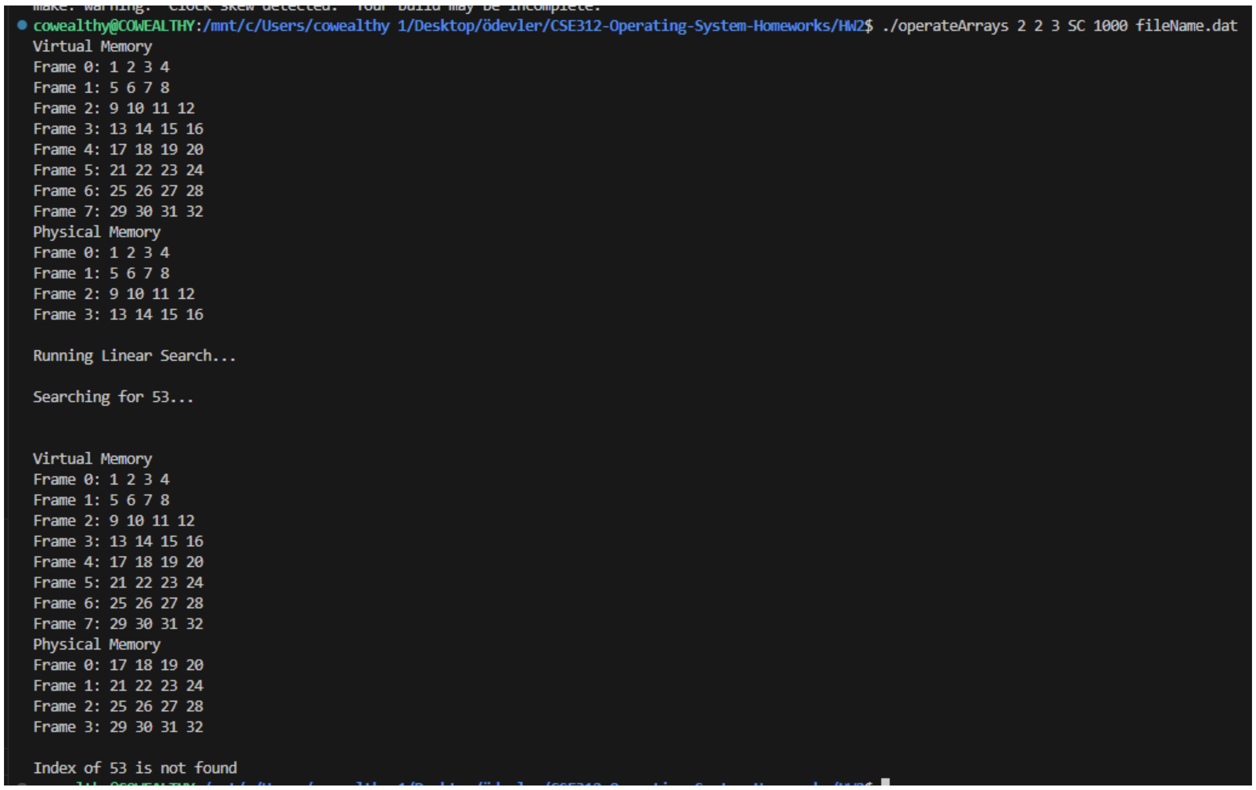
```
● cowealthy@COWEALTHY:/mnt/c/Users/cowealthy 1/Desktop/ödevler/CSE312-Operating-System-Homeworks/HW2$ ./operateArrays 2 2 3 SC 1000 fileName.dat
  Virtual Memory
  Frame 0: 1 2 3 4
  Frame 1: 5 6 7 8
  Frame 2: 9 10 11 12
  Frame 3: 13 14 15 16
  Frame 4: 17 18 19 20
  Frame 5: 21 22 23 24
  Frame 6: 25 26 27 28
  Frame 7: 29 30 31 32
  Physical Memory
  Frame 0: 1 2 3 4
  Frame 1: 5 6 7 8
  Frame 2: 9 10 11 12
  Frame 3: 13 14 15 16

  Running Linear Search...

  Searching for 8...


  Virtual Memory
  Frame 0: 1 2 3 4
  Frame 1: 5 6 7 8
  Frame 2: 9 10 11 12
  Frame 3: 13 14 15 16
  Frame 4: 17 18 19 20
  Frame 5: 21 22 23 24
  Frame 6: 25 26 27 28
  Frame 7: 29 30 31 32
  Physical Memory
  Frame 0: 1 2 3 4
  Frame 1: 5 6 7 8
  Frame 2: 9 10 11 12
  Frame 3: 13 14 15 16

  Index of 8 is 1
```

The program is tried lineer search with 22. 22 is not putted in the physical memory. So, some page faults occurred. The necessary address is taken from the Disk thanks to second algorithm. Physical memory is updated.



```
cowealthy@COWEALTHY:/mnt/c/Users/cowealthy 1/Desktop/ödevler/CSE312-Operating-System-Homeworks/HW2$ ./operateArrays 2 2 3 SC 1000 fileName.dat
Virtual Memory
Frame 0: 1 2 3 4
Frame 1: 5 6 7 8
Frame 2: 9 10 11 12
Frame 3: 13 14 15 16
Frame 4: 17 18 19 20
Frame 5: 21 22 23 24
Frame 6: 25 26 27 28
Frame 7: 29 30 31 32
Physical Memory
Frame 0: 1 2 3 4
Frame 1: 5 6 7 8
Frame 2: 9 10 11 12
Frame 3: 13 14 15 16

Running Linear Search...

Searching for 22...


Virtual Memory
Frame 0: 1 2 3 4
Frame 1: 5 6 7 8
Frame 2: 9 10 11 12
Frame 3: 13 14 15 16
Frame 4: 17 18 19 20
Frame 5: 21 22 23 24
Frame 6: 25 26 27 28
Frame 7: 29 30 31 32
Physical Memory
Frame 0: 17 18 19 20
Frame 1: 21 22 23 24
Frame 2: 9 10 11 12
Frame 3: 13 14 15 16

Index of 22 is 1
```

Last, program is tried lineer search with 53. 53 is not putted in the memory and disk. So, the program gives some page fault, look for address to disk. The physical memory is updated. However, the address is not founded.



```
make: warning:  Clock skew detected.  Your build may be incomplete.
cowealthy@COWEALTHY:/mnt/c/Users/cowealthy 1/Desktop/ödevler/CSE312-Operating-System-Homeworks/HW2$ ./operateArrays 2 2 3 SC 1000 fileName.dat
Virtual Memory
Frame 0: 1 2 3 4
Frame 1: 5 6 7 8
Frame 2: 9 10 11 12
Frame 3: 13 14 15 16
Frame 4: 17 18 19 20
Frame 5: 21 22 23 24
Frame 6: 25 26 27 28
Frame 7: 29 30 31 32
Physical Memory
Frame 0: 1 2 3 4
Frame 1: 5 6 7 8
Frame 2: 9 10 11 12
Frame 3: 13 14 15 16

Running Linear Search...

Searching for 53...


Virtual Memory
Frame 0: 1 2 3 4
Frame 1: 5 6 7 8
Frame 2: 9 10 11 12
Frame 3: 13 14 15 16
Frame 4: 17 18 19 20
Frame 5: 21 22 23 24
Frame 6: 25 26 27 28
Frame 7: 29 30 31 32
Physical Memory
Frame 0: 17 18 19 20
Frame 1: 21 22 23 24
Frame 2: 25 26 27 28
Frame 3: 29 30 31 32

Index of 53 is not found
```