CSE 312 Spring 2023

Homework 2

Report

1801042614

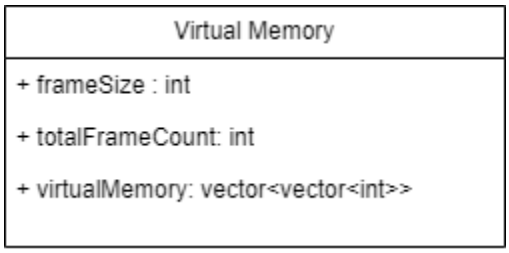
Samet Nalbant

Contents

1. Page Table Structure

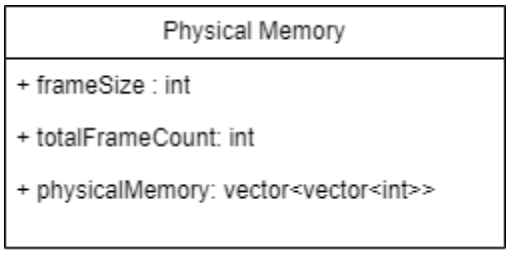
In this homework, to provide virtual memory, physical memory, page table and disk file, I create several classes.

- Virtual Memory Class



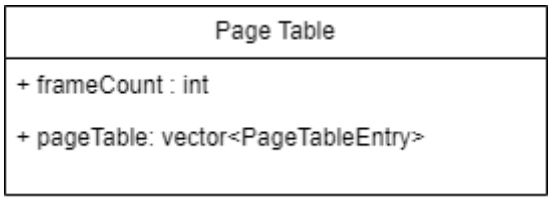
In this class, I hold the frame size, frame count and the offset values to access physical memory according to using index of virtual memory vector and offset that belong to it. Each dimension in the

- Physical Memory Class



This class also works like the Virtual Memory class. Instead of the containing offsets, values that belongs to the physical memory kept in 2 dimensional vector. Each dimension representing frames, and each frames contains set of values acording to frame size.

- Page Table Class



This class contains the set of PageTableEntry objects, that represents the mapping between virtual frames and physical frames. PageTableEntry objects also contains the required bits.

- Page Table Entry Class

Page Table Entry
+ pageFrameNumber : int
+ present : bool
+ modified : bool
+referenced: bool

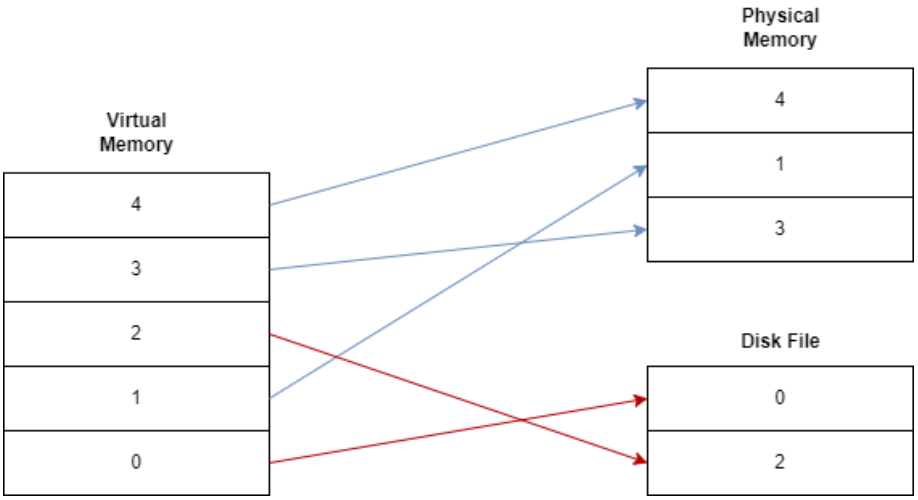
This class contains the required referenced, modified, present bits and also contains the physical page frame number to map virtual memory and physical memory.

- Disk File Strcuture

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

```

Disk file holds the virtual frames that are not mapped the physical memory due to the low size of physical memory. Each frame present hold in different line and the first number of each line shows the virtual frame number.

Overall structure of design shown figure in the below:



Page table provides the mapping between virtual memory and physical memory. This the example demonstration of page table mapping according to previous figure.

Page Table Mapping

4	2
3	0
2	-
1	1
0	-

2. Initilization of Physical Memory, Virtual Memory and Disk File

To manage the memories, disk file and to handle accesing memory, writing memory, page replacement operations there is a class name MemoryManager. When the MemoryManager object is created, It is intialize the physical memory, virtual memory and disk file according to its paremeters. Paremeters consist on:

- Frame Size
- Virtual Frame Count
- Physical Frame Count
- Disk File Name
- Page Replacement Algorithm

Initilization process starts with the mapping virtual frames and physical frames randomly. It also sets the necessary bits and values that will be used by page replacement algorithms.

```
void MemoryManager::initializePageTable(){
    vector<int> virtualFrames;
    for (int i = 0; i < this->virtualFrameCount; i++) {
        virtualFrames.push_back(i);
    }
    random_device rd;
    mt19937 g(rd());
    shuffle(virtualFrames.begin(), virtualFrames.end(), g);

    for (int i = 0; i < this->physicalFrameCount; i++) {
        int virtualFrameNumber = virtualFrames[i];
        this->pageTable->setPhysicalFrameNumber(virtualFrameNumber, i);
        this->pageTable->setPresentBit(virtualFrameNumber,true);
        secondChanceQueue.push(virtualFrameNumber);
        leastRecentlyUsed->setLRUEntry(i, virtualFrameNumber);
        wsClock->setWSClockEntry(i, virtualFrameNumber);
    }

    this->pageTable->printPageTable();
}
```

After the page table is initialized, physical memory is initialized and the values from 1 to frame size is setted for each page frame in the physical memory.

```
void MemoryManager::intializePhysicalMemory(){
    vector<int> frame;
    for (int j = 0; j < this->frameSize; j++) {
        frame.push_back(j);
        numberOfWrites++;
    }
    for (int i = 0; i < this->physicalFrameCount; i++) {
        this->physicalMemory->setFrame(i, frame);
    }
    this->physicalMemory->printPhysicalMemory();
}
```

After the physical memory is intialized, disk file is initialized with the virtual frames that are not mapped to the physical memory.

```
void MemoryManager::initializeDiskFile(){
    ofstream diskFile;
    diskFile.open(this->diskFileName);
    for(int i = 0; i < this->virtualFrameCount; i++){
        if(pageTable->getPhysicalFrameNumber(i) != -1)
            continue;
        else{
            diskFile << i << " ";
            for(int j = 0; j < this->frameSize; j++){
                diskFile << j << " ";
            }
            diskFile << endl;
        }
    }
    diskFile.close();
}
```

After all the others are intialized, virtual memory is initialized with values from 1 to frame size as a design choice for implementation.

```
void MemoryManager::initializeVirtualMemory(){
    for(int i = 0; i < this->virtualFrameCount; i++){
        vector<int> frame;
        for(int j = 0; j < this->frameSize; j++){
            frame.push_back(j);
        }
        this->virtualMemory->setFrame(i, frame);
    }
}
```

3. Reading and Writing Values From / To Physical Memory

To access physical memory, functions that belongs to MemoryManager is used. MemoryManager functions takes virtual adresses and get values using the page table information.

- Reading values from physical memory

To read values from physical memory, getValueFromPhysicalMemory function is used. First of all function controls whether the virtual page is present in the physical memory or not. If it is not present in the physical memory, according to the selected page replacement algorithm It does the page replacement and after that It returns the value.

```
int MemoryManager::getValueFromPhysicalMemory(int virtualFrameNumber, int offset){
    int physicalFrameNumber = pageTable->getPhysicalFrameNumber(virtualFrameNumber);
    if(physicalFrameNumber == -1){ // page fault is occured
        numberOfPageFaults++;
        numberOfPageReplacements++;
        switch (pageReplacementAlgorithm)
        {
        case WS_CLOCK_PAGE_REPLACEMENT:
            wsClockPageReplacement(virtualFrameNumber);
            break;
        case SECOND_CHANCE_PAGE_REPLACEMENT:
            secondChancePageReplacement(virtualFrameNumber);
            break;
        case LEAST_RECENTLY_USED_PAGE_REPLACEMENT:
            leastRecentlyUsedPageReplacement(virtualFrameNumber);
            break;
        default:
            break;
        }
        physicalFrameNumber = pageTable->getPhysicalFrameNumber(virtualFrameNumber);
    }
    vector<int> frame = physicalMemory->getFrame(physicalFrameNumber);
    pageTable->setReferenced(virtualFrameNumber, true);
    if(pageReplacementAlgorithm == WS_CLOCK_PAGE_REPLACEMENT){
        wsClock->updateLastAccessTime(physicalFrameNumber);
    }

    if(pageReplacementAlgorithm == LEAST_RECENTLY_USED_PAGE_REPLACEMENT)
        leastRecentlyUsed->incrementLRUEntryCounter(physicalFrameNumber);
    numberOfReads++;
    return frame[offset];
}
```

- Writing values to physical memory

To write values to physical memory, writeValueToPhysicalMemory function is used. Function takes virtual adress and the value that will be write to the memory. Function first determines whether the virtual page is present in the physical memory or not. If it is not present, It makes necessary page replacement operation according to chosen algorithm. After that, It writes value to the physical memory.

```
void MemoryManager::writeValueToPhysicalMemory(int virtualFrameNumber, int offset, int value){
    int physicalFrameNumber = pageTable->getPhysicalFrameNumber(virtualFrameNumber);
    if(physicalFrameNumber == -1){ // page fault is occured
        numberOfPageFaults++;
        numberOfPageReplacements++;
        switch (pageReplacementAlgorithm)
        {
        case WS_CLOCK_PAGE_REPLACEMENT:
            wsClockPageReplacement(virtualFrameNumber);
            break;
        case SECOND_CHANCE_PAGE_REPLACEMENT:
            secondChancePageReplacement(virtualFrameNumber);
            break;
        case LEAST_RECENTLY_USED_PAGE_REPLACEMENT:
            leastRecentlyUsedPageReplacement(virtualFrameNumber);
            break;
        default:
            break;
        }
        physicalFrameNumber = pageTable->getPhysicalFrameNumber(virtualFrameNumber);
    }

    pageTable->setReferenced(virtualFrameNumber, false);
    physicalMemory->setValue(physicalFrameNumber, offset, value);
    pageTable->setModified(virtualFrameNumber, true);
    if(pageReplacementAlgorithm == WS_CLOCK_PAGE_REPLACEMENT)
        wsClock->updateLastAccessTime(physicalFrameNumber);
    numberOfWrites++;
}
```

4. Reading and Writing Frames From / To Disk File

When page fault is encountered, frames should be loaded from disk file and replaced with the selected frames from the page replacement algorithm. To handle these implementations several functins are added to the MemoryManager class.

- Writing frames to disk file

To write frame to disk file, writeFrameToDiskFile function is used. It takes virtual frame number and the values belong to that frames. It appends frame to end of the disk file.

```
bool MemoryManager::writeFrameToDiskFile(int virtualFrameNumber, vector<int> frame){
    ofstream outputFile(diskFileName, ios::app);
    if (outputFile.is_open()) {
        outputFile << virtualFrameNumber << " ";
        for (auto i : frame) {
            outputFile << i << " ";
        }
        outputFile << endl;
        outputFile.close();
        numberOfDiskPageWrites++;
        return true;
    }
    return false;
}
```

- Reading frames from disk file

To read frame from disk file for replacement, getFrameFromDiskFile function is used. It takes the virtual frame number and returns the frame belong to that virtual frame number. Before the return opeartion, It removes the frame from disk file.

```
vector<int> MemoryManager::getFrameFromDiskFile(int virtualFrameNumber){
    ifstream inputFile(diskFileName);
    vector<int> integers;
    int frameNumber;
    if (inputFile.is_open()) {
        string line;
        while (getline(inputFile, line)) {
            istringstream iss(line);
            if (iss >> frameNumber) {
                if (frameNumber == virtualFrameNumber) {
                    int num;
                    while (iss >> num) {
                        integers.push_back(num);
                    }
                    break;
                }
            }
        }
        inputFile.close();
    }
    deleteFrameFromFile(virtualFrameNumber);
    numberOfDiskPageReads++;
    return integers;
}
```

- Deleting frames from disk file

To delete frame from file, deleteFrameFromFile function is used. It takes the virtual frame number and removes the frame belong the that virtual frame number from disk file.

```
void MemoryManager::deleteFrameFromFile(int virtualFrameNumber) {
    ifstream inputFile(diskFileName);
    ofstream tempFile("temp.txt");
    if (inputFile.is_open() && tempFile.is_open()) {
        string line;
        while (getline(inputFile, line)) {
            istringstream iss(line);
            int frameNumber;
            if (iss >> frameNumber) {
                if (frameNumber != virtualFrameNumber) {
                    tempFile << line << endl;
                }
            }
        }
        inputFile.close();
        tempFile.close();

        remove(this->diskFileName.c_str());
        rename("temp.txt", diskFileName.c_str());
    }
}
```

5. Page Replacement Algorithms

5.1. Second Chance Page Replacement

To implement a Second Change Page Replacement algorithm, queue named with secondChanceQueue is used to hold virtual frames that are present in the physical memory. To minimize the memory usage, It holds the indexes of the virtual frames. Algorithm checks the front item of queue. If it is not referenced and since it is the oldest page, It was replaced immediatly with the requested virtual page. If it is referenced, It moves to the next page and makes it unreferenced. This operation is done until find suitable page.

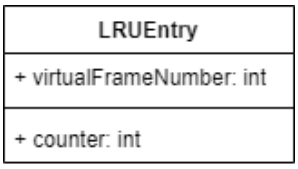
```
void MemoryManager::secondChancePageReplacement(int virtualFrameNumber){
    int frontPage = secondChanceQueue.front();
    while(pageTable->isReferenced(frontPage)){
        pageTable->setReferenced(frontPage, false);
        secondChanceQueue.pop();
        secondChanceQueue.push(frontPage);
        frontPage = secondChanceQueue.front();
    }

    secondChanceQueue.pop();
    secondChanceQueue.push(virtualFrameNumber);

    int physicalFrameNumberWillBeReplaced = deleteFrameFromPhysicalMemory(frontPage);
    vector<int> frameFromFile = getFrameFromDiskFile(virtualFrameNumber);
    physicalMemory->setFrame(physicalFrameNumberWillBeReplaced, frameFromFile);
    pageTable->initializePageTableEntry(virtualFrameNumber, physicalFrameNumberWillBeReplaced);

}
```

5.2. Least Recently Used Page Replacement

To implement Least Recently Used Page Replacement algorithm, several data structure is used.

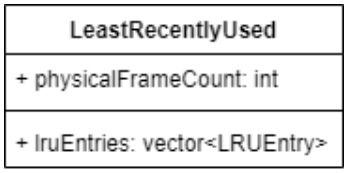
- LRUEntry

This class holds the virtual frame number and counter that will be used in Least Recently Used Page Replacement Algorithm.



- LeastRecentlyUsed

This class holds the virtual frames list and its counters to be used in Least Recently Used Page Replacement Algorithm.



Thanks the data structure that holds in the MemoryManager class, least recently used physical page is found by its counter. Counter is increased at every access. After the least recently used frame is found it is replaced with the requested frame. Requested frame is reading from disk and replaced frame is writing to the disk.

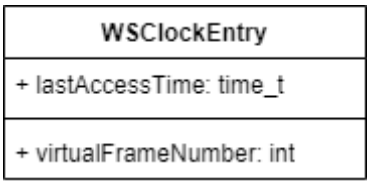
```
void MemoryManager::leastRecentlyUsedPageReplacement(int virtualFrameNumber){
    int virtualFrameNumberWillBeReplaced = leastRecentlyUsed->getLeastRecentlyUsedVirtualFrameNumber();
    leastRecentlyUsed->resetLRUEntryCounter(leastRecentlyUsed->getPhysicalFrameNumber(virtualFrameNumberWillBeReplaced));
    leastRecentlyUsed->setLRUEntryVirtualFrameNumber(leastRecentlyUsed->getPhysicalFrameNumber(virtualFrameNumberWillBeReplaced), virtualFrameNumber);
    int physicalFrameNumberWillBeReplaced = deleteFrameFromPhysicalMemory(virtualFrameNumberWillBeReplaced);
    vector<int> frameFromFile = getFrameFromDiskFile(virtualFrameNumber);
    physicalMemory->setFrame(physicalFrameNumberWillBeReplaced, frameFromFile);
    pageTable->initializePageTableEntry(virtualFrameNumber, physicalFrameNumberWillBeReplaced);
}
```

5.3 WSClock Page Replacement

To implement WSClock Page Replacement algorithm, several data structure and variables are used.

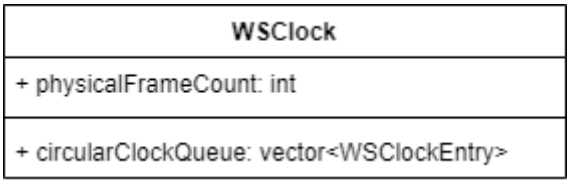
- WSClockEntry

This class holds the last access time and virtual frame number that will be used in WSClock page replacement algorithm.



- WSClock

This class holds the list of virtual frames in the physical memory and its last access time inside the circular queue form of vector.



- wsClockIndex

This class member variable is used for the circular queue data structure. Thanks to this variable, circular form provided using vector.

To implement WS Clock Page Replacement algorithm, R bit is checked, If it is 0 then the last access time is compared with the threshold value previosly deciced, If the last access time is passed according to given threshold, then replacement is done. If the R bit is 1, then the page is unreferenced and last acces time is updated. If, after whole circular queue is searched and no suitable page is found for replacement, since the oldest page is the first page before the search, is replaced.

```
void MemoryManager::wsClockPageReplacement(int virtualFrameNumber){
    for (int i = 0; i < physicalFrameCount; i++) {
        int startingIndex = (i + wsClockIndex) % physicalFrameCount;
        int virtualFrameIndex = pageTable->getVirtualFrameNumber(startingIndex);
        if (!pageTable->isReferenced(virtualFrameIndex)) { // R bit 0 ise ne yapılcak
            time_t currentTime = time(0);
            if (difftime(wsClock->getWSClockEntryLastAccessTime(startingIndex), currentTime) > threshold) {
                int virtualFrameNumberWillBeReplaced = wsClock->getWSClockEntryVirtualFrameNumber(startingIndex);
                int physicalFrameNumberWillBeReplaced = deleteFrameFromPhysicalMemory(virtualFrameNumberWillBeReplaced);
                vector<int> frameFromFile = getFrameFromDiskFile(virtualFrameNumber);
                physicalMemory->setFrame(physicalFrameNumberWillBeReplaced, frameFromFile);
                pageTable->initializePageTableEntry(virtualFrameNumber, physicalFrameNumberWillBeReplaced);
                wsClockIndex = startingIndex+1;
                return;
            }
            else{
                continue;
            }
        }
        else{
            pageTable->setReferenced(virtualFrameIndex, false);
            wsClock->updateLastAccessTime(virtualFrameIndex);
        }
    }
    // if all bits are 1, then we have to do some replacement operations
    int virtualFrameNumberWillBeReplaced = wsClock->getWSClockEntryVirtualFrameNumber(wsClockIndex);
    wsClock->setWSClockEntry(wsClockIndex, virtualFrameNumber);
    int physicalFrameNumberWillBeReplaced = deleteFrameFromPhysicalMemory(virtualFrameNumberWillBeReplaced);
    vector<int> frameFromFile = getFrameFromDiskFile(virtualFrameNumber);
    physicalMemory->setFrame(physicalFrameNumberWillBeReplaced, frameFromFile);
    pageTable->initializePageTableEntry(virtualFrameNumber, physicalFrameNumberWillBeReplaced);
    wsClockIndex = (wsClockIndex + 1) % physicalFrameCount;
}
```

6. Operations on Memory

6.1. Linear Search

In this search, whole virtual adresses are checked to found target value. Whole virtual memory accepted as a one dimensional array. If the target value is found, index of printed on terminal.

```
void linearSearchOperation(MemoryManager *memoryManager, int frameSize, int virtualFrameCount, int physicalFrameCount, int targetValue){
    int count = -1;
    bool isFound = false;
    for(int i=0; i < virtualFrameCount; i++){
        if(isFound)
            break;
        for(int j=0; j < frameSize; j++){
            int readedValue = memoryManager->getValueFromPhysicalMemory(i, j);
            if(readedValue == targetValue){
                count++;
                isFound = true;
                break;
            }
            count++;
        }
    }
    if(!isFound)
        cout << "Target value not found!" << endl;
    else
        cout << "Target value found at index: " << count << endl;
}
```

6.2. Array Summation

In this summation operation, whole virtual memory accepted as one dimensional array and the whole array is summed and the result printed on the screen.

```
void arraySummationOpeartion(MemoryManager *memoryManager, int frameSize, int virtualFrameCount, int physicalFrameCount){
    int sum = 0;
    for(int i=0; i < virtualFrameCount; i++){
        for(int j=0; j < frameSize; j++){
            sum += memoryManager->getValueFromPhysicalMemory(i, j);
        }
    }
    cout << "Sum: " << sum << endl;
}
```

6.3. Matrix Vector Multiplication

In this matrix vector multiplication, whole virtual memory is accepted as 4k and divided as 3k matrix, k is vector. After that matrix vector multiplication is realized. Firstly necessary values of matrix and vector are multiplied and overwrited to the matix. After that matrixes traversed to found multiplication result.

```cpp
void matrixMultiplicationOperation(MemoryManager *memoryManager, int frameSize, int virtualFrameCount, int physicalFrameCount){
    int vectorSize = virtualFrameCount / 4;
    //matrix
    int vectorIndex = 0;
    for(int i=vectorSize; i< virtualFrameCount; i++){
        if(vectorIndex >= vectorSize)
            vectorIndex = 0;
        for(int j=0; j<frameSize; j++){
            int result = memoryManager->getValueFromPhysicalMemory(i, j) * memoryManager->getValueFromPhysicalMemory(vectorIndex, j);
            memoryManager->writeValueToPhysicalMemory(i, j, result);
        }
        vectorIndex++;
    }

    cout << "--------------------------" << endl;

    vector<int> resultVector;
    vectorIndex = 0;
    int sum = 0;
    for(int i=vectorSize; i< virtualFrameCount; i++){
        if(vectorIndex >= vectorSize){
            resultVector.push_back(sum);
            sum = 0;
            vectorIndex = 0;
        }
        for(int j=0; j<frameSize; j++){
            sum += memoryManager->getValueFromPhysicalMemory(i, j);
        }
        vectorIndex++;
    }
    resultVector.push_back(sum);
    cout  << "Result Matrix: " << endl;
    for(int i=0; i<3; i++){
        cout << resultVector[i] << endl;
    }

    cout << "--------------------------" << endl;
    memoryManager->printPageTable();
}
```

6.4 Binary Seearch

In binary search algorithm, whole virtual memory is accepted one dimensional array. First the array is sorted by using the bubble sort. After the sort operation is realized, target value is searched by using binary search algorithm.

```cpp
void binarySearchOperation(MemoryManager* memoryManager, int frameSize, int virtualFrameCount, int physicalFrameCount, int targetValue) {

    for(int i=0; i < virtualFrameCount; i++){
        for(int j=0; j < frameSize; j++){
            for(int k=0; k < virtualFrameCount; k++){
                for(int l=0; l < frameSize; l++){
                    if(memoryManager->getValueFromPhysicalMemory(i, j) < memoryManager->getValueFromPhysicalMemory(k, l)){
                        int temp = memoryManager->getValueFromPhysicalMemory(i, j);
                        memoryManager->writeValueToPhysicalMemory(i, j, memoryManager->getValueFromPhysicalMemory(k, l));
                        memoryManager->writeValueToPhysicalMemory(k, l, temp);
                    }
                }
            }
        }
    }

    int left = 0;
    int right = virtualFrameCount * frameSize - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        int row = mid / frameSize;
        int column = mid % frameSize;

        int readedValue = memoryManager->getValueFromPhysicalMemory(row, column);

        if (readedValue == targetValue) {
            cout << "Target value found at index: " << mid << endl;
            return;
        } else if (readedValue < targetValue) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    cout << "Target value not found!" << endl;
```

7. Test Results

For physical memory hold 16K integer, for virtual memory hold 128K integer

Second Chance vs. Least Recently Used

- Linear Search

```
Least Recently Used Statistics:
Number of page faults: 31
Number of reads: 131072
Number of writes: 4096
Number of disk page writes: 55
Number of disk page reads: 31
Number of page replacements: 31
---------------------------------
Second Chance Statistics:
Number of page faults: 29
Number of reads: 131072
Number of writes: 4096
Number of disk page writes: 53
Number of disk page reads: 29
Number of page replacements: 29
```

- Array Summation

```
Least Recently Used Statistics:
Number of page faults: 31
Number of reads: 131072
Number of writes: 4096
Number of disk page writes: 55
Number of disk page reads: 31
Number of page replacements: 31
---------------------------------
Second Chance Statistics:
Number of page faults: 32
Number of reads: 131072
Number of writes: 4096
Number of disk page writes: 56
Number of disk page reads: 32
Number of page replacements: 32
---------------------------------
```

- Matrix - Vector Multiplication

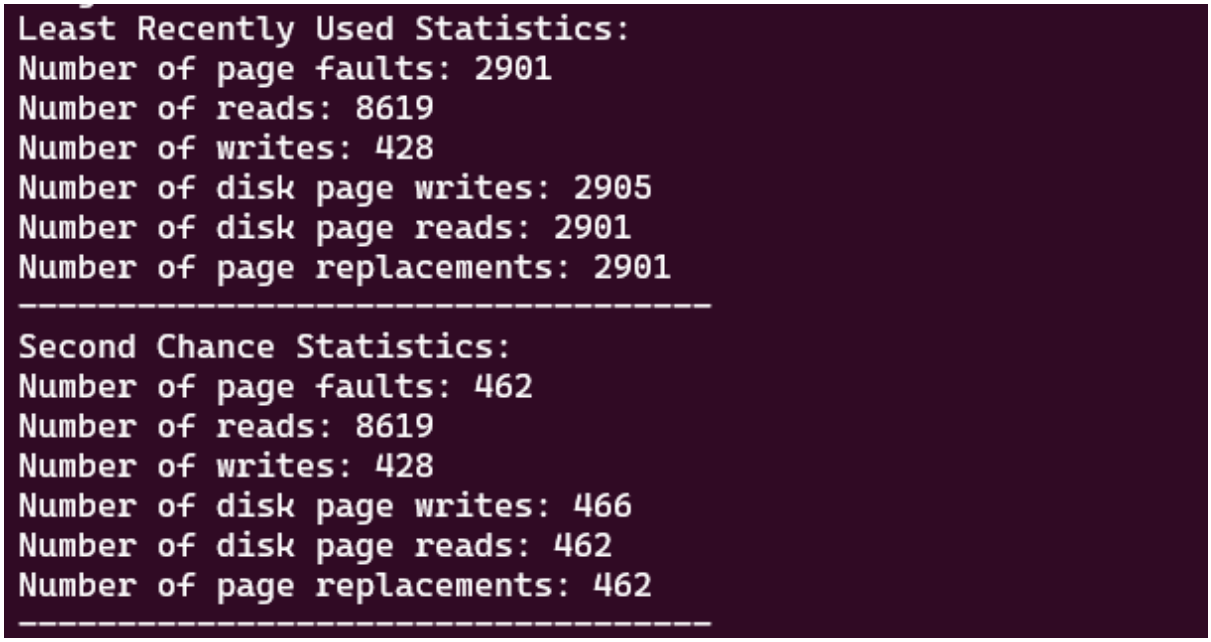
```
Least Recently Used Statistics:
Number of page faults: 44
Number of reads: 294912
Number of writes: 98304
Number of disk page writes: 44
Number of disk page reads: 44
Number of page replacements: 44
```

```
Second Chance Statistics:
Number of page faults: 51
Number of reads: 294912
Number of writes: 98304
Number of disk page writes: 51
Number of disk page reads: 51
Number of page replacements: 51
---------------------------------
```

- Matrix – Vector Multiplication

```
Least Recently Used Statistics:
Number of page faults: 11
Number of reads: 576
Number of writes: 224
Number of disk page writes: 15
Number of disk page reads: 11
Number of page replacements: 11
---------------------------------
Second Chance Statistics:
Number of page faults: 14
Number of reads: 576
Number of writes: 224
Number of disk page writes: 18
Number of disk page reads: 14
Number of page replacements: 14
---------------------------------
```

- Binary Search



To compare Second Chance and Least Recently Used algorithm, In array summation and linear search both of them gives close results. However in matrix vector multiplication and binary search algorithm Least Recently USED algorithm gives worse performance. Since, the soring takes long long time in binary search algorithm, It is tested with small numbers.