King Abdulaziz University
Faculty of Computing & Information Technology
Computer Science Department

Memory Management: implementation of Main Memory & Virtual Memory

CPCS-361 Operating Systems 1 Project – Winter 2023

Prepared for:

Dr. Mai Ahmed Fadel

Department of Computer Science

Prepared by (Group 5) members:

Student name	Student ID	Section
Hadeel Abuhamous		B2
Ro'aa Altunsi		B3
Dima Kanawati		B2
Ashwaq Khayat		B2

Feb 19, 2023

Table of Contents

Table of Figures

List of Tables

1. Introduction

Memory management is an operating system functionality that manages memory and transfers processes back and forth between the main memory and disk during program execution (OS Memory Management, n.d.). Our project's idea is to implement a simulation of two kinds of memory used in computers which are the Main memory & Virtual memory, and their management of process allocation.

1.1 Main Memory

The project has two sections, first section is about main memory. Main memory is the primary, internal workspace in the computer, commonly known as RAM (random access memory). The purpose of the project is to design and implement a program that simulates some of managing a contiguous region of memory where addresses ranges from 0 to MAX -1. Our program can also allow the user to request and release for a contiguous block of memory, compact unused holes of memory into one single block, and report the regions of free and allocated memory.

1.2 Virtual Memory

As for the Second section of the project, it's about the Virtual memory. Virtual memory is a common technique used in a computer's operating system (OS). Virtual memory uses both hardware and software to enable a computer to compensate for physical memory shortages, temporarily transferring data from random access memory (RAM) to disk storage. The purpose of the project is to create a program that reads from a file containing logical addresses and translate each logical address to its corresponding physical address. The program can also calculate the page fault implement a page replacement.

1.3 Programming Tools & System Specifications

Compiler name & version	Internal API of javac used by NetBeans Apache 16 IDE
Operating system name & version	Windows 10 Home 64-bit
Processor (CPU)	Intel® Core™ i7-7700HQ CPU 2.80GHz
Main memory (RAM)	16.0 GB

Table 1: Programming Tools & System Specifications

2. Test cases

2.1 Main Memory

- At startup our program will be passed the initial amount of memory. And present the prompt: allocator>

```
run:
|| Welcome to our Memory Management Program ||
Please enter the memory size
./allocator |
```

Figure 1: Program Startup output

Then we can use one of the 4 commands: RQ, RL, C, STAT:

- The first command is "RQ" referring to Request, it will allocate memory using one of the three approaches: first fit, best fit, and worst fit depending on the flag (F, B, W).

```
run:
|| Welcome to our Memory Management Program ||
Please enter the memory size
./allocator 1048576

allocator>RQ P0 36462 F

allocator>
```

Figure 2: Request Command Output

- The second command is “RL” referring to Release, it will release or remove a process from the memory leaving a hole in its address/place.

```
Addresses [0:3462] Process P5
Addresses [3463:26996] Process P3
Addresses [26997:64530] Process P9
Addresses [64531:234421] Unused
Addresses [234422:326806] Process P1
Addresses [326807:1048575] Unused

allocator>RL P3

allocator>RL P5

allocator>STAT
Addresses [0:26996] Unused
Addresses [26997:64530] Process P9
Addresses [64531:234421] Unused
Addresses [234422:326806] Process P1
Addresses [326807:1048575] Unused
```

Figure 3: Release Command Output

- The third command is “C” referring to Compact, it will compact the set of holes into one larger hole to exploit unused spaces and use them to allocate more processes.

```
allocator>stat
Addresses [0:3545] Process P0
Addresses [3546:4789] Unused
Addresses [4790:7324] Process P3
Addresses [7325:8324] Process P4
Addresses [8325:1048575] Unused

allocator>c

allocator>stat
Addresses [0:3545] Process P0
Addresses [3546:6080] Process P3
Addresses [6081:7080] Process P4
Addresses [7081:1048575] Unused
```

Figure 4: Compact Command Output

- The fourth and last command is “STAT” referring to Status, it will display the memory status information such as the name of allocated processes, their base address, end addresses, and holes in memory.

```
allocator>STAT
Addresses [0:3462] Process P5
Addresses [3463:26996] Process P3
Addresses [26997:64530] Process P9
Addresses [64531:234421] Unused
Addresses [234422:326806] Process P1
Addresses [326807:1048575] Unused
```

Figure 5: Status Command Output

2.2 Virtual memory

- Ensure that the program can retrieve from the physical memory the expected value for the chosen logical address:

```
—————————————————————— Statistics ——————————————————————

Length of address string is 80

35826 10842 45515 6091  5868  63258 10392 25086 30705 53502
28931 44954 42252 21395 32541 45688 5129  34561 36922 692
48128 54253 969   58969 20163 3884  43328 50552 51079 43765
49294 13730 49655 44770 8620  56657 12893 4970  59240 58882
43121 20259 28964 20094 5003  44059 51476 43218 38174 41118
19358 3784  54388 27444 15757 38336 17665 28483 46919 21583
33134 21398 8090  60086 13533 3067  42932 5491  18145 40891
33405 36529 1220  22514 58554 32756 64196 31649 22914 50227

■ The Number of page faults: 21
■ The percentage of address references that resulted in page faults is %26.25
■ The Number of page hits: 59
```

Figure 6: Physical Memory Value Retrieval Test Output

- Ensure that the program can correctly deal with partially loaded process by identifying the occurrence of page faults.

```
—————————————————— The Five Test Cases ————————————————————

Logical Address     Page #     Offset     Frame #     Value      Same as model answer

27966               109        62         16          27         Yes
53683               209        179        242         108        Yes
14557               56         221        85          0          Yes
16916               66         20         67          0          Yes
61006               238        78         110         59         Yes
```

Figure 7: Partially Loading Processes Test Output

3. Page Replacement

Page replacement is a policy that is used to prevent memory over-allocation by including it with page-fault service (Silberschatz, Beran, Gagne, & Galvin, 2008), it works by selecting an allocated frame as a victim using one of the page replacement algorithms, swap its assigned page out to the backing store, and reallocate a new page to that frame.

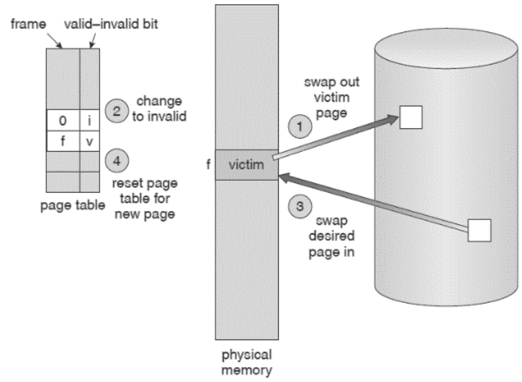


Figure 8: Page Replacement Policy

In our project we used the FIFO page replacement policy, which selects the victim based on the page from memory that is first loaded among the available pages (Silberschatz, Beran, Gagne, & Galvin, 2008).

3.1 Test Cases

- Ensure that your program can correctly replace an old page when the requested logical address is part of a new page, and the physical memory is full.

——————————— The testing of the page replacement #1 ———————————

Logical Address	New Page #	Victim Page #	Reused Frame #
22048	86	72	53
26867	104	17	43
31880	124	98	2
18815	73	4	23
33958	132	26	101

——————————— The testing of the page replacement #2 ———————————

Logical Address	Page #	Offset	Frame #	Value	Same as model answer
22048	86	32	53	40	Yes
26867	104	243	43	-65	Yes
31880	124	136	2	-115	Yes
18815	73	127	23	-53	Yes
33958	132	166	101	-119	Yes

Figure 9: Page Replacement Test Outputs

4. Source Code

4.1 Main Memory's Source Code

In order to implement the main memory in an efficient and practical way, we used Object-oriented programming concepts (classes and objects) to build our memory. So, we implemented four classes: the MemoryManagement class as a Main class, the Memory class, the Partition class, and the Process class.

- MemoryManagement Class

```
import static java.lang.System.exit;
import java.util.Scanner;

public class MemoryManagement {
    static Memory mainMemory;
    public static void main(String[] args) throws
        IndexOutOfBoundsException{
        Scanner input; String command; String[] commandInfo;
        System.out.println("|| Welcome to our Memory Management
                Program ||\n./allocator 1048576");
        input = new Scanner(System.in);
        System.out.println("Please enter the memory size");
        System.out.print("./allocator ");
        int memorySize = Integer.parseInt(input.nextLine());
        mainMemory = new Memory(memorySize);

        while(true){
            // Read command from user
            System.out.print("\nallocator>");
            command = input.nextLine().toUpperCase();
            commandInfo = command.split(" ");
            switch(commandInfo[0]){
                case "X": exit(0);  // Terminate the program

                case "RQ":  // Call RQ method
                    if(commandInfo.length != 4){
                        System.out.println("Error! Please enter a
                                complete request (e.g., RQ P0 400
                                F)");
                        continue;
                    }  else if(mainMemory.findProcess
                       (commandInfo[1])!= null){
                        System.out.println("Error! The process " +
                                commandInfo[1] + " is already
                                allocated in memory");
                        continue;
                    } else if (!commandInfo[2].matches("[0-9]+")){
                        System.out.println("Error! The process size
                                should be a number!");
```

```
                    continue;
                } RequestMemLocation(commandInfo[1],
            Integer.parseInt(commandInfo[2]), commandInfo[3]);
                break;
            case "RL":  // Call RL Method
                if (commandInfo.length != 2) {
                    System.out.println("Error! Please enter a
                            complete request (e.g., RL P0)");
                    continue;
                }
                ReleaseMemLocation(commandInfo[1]);
                break;

            case "C":  // Call C Method
                compact();
                break;

            case "STAT":  // Call STAT Method
                printStatusReport();
                break;

            default: // The user enters invalid command
                System.out.println("\nYou are allowed only to
                   enter:");
                System.out.printf("%-20s\n%-20s\n%-20s\n%-
                   20s\n%-20s\n",
                   "X >> Exit", "RQ >> Request", "RL >>
                   Release", "C >> Compact",
                   "STAT >> Status Report");
            }
        }
    }
//------------------ Request Memory Space -------------------
    public static void RequestMemLocation(String name, int size,
String policy){

        Partition hole = null;
        if(mainMemory.getMaxHoleSize() < size){   // Not enough
        space
            System.out.println("Error! Memory is full now");

        } else {  // Allocate the process
            Processes proc = new Processes(name, size);
            switch (policy) {
                case "F":    // First Fit
                    hole = mainMemory.getFirstFitHole(proc);
                    break;
                case "B":    // Best Fit
                    hole = mainMemory.getBestFitHole(proc);
                    break;
                case "W":    // Worst Fit
```

```java
                    hole = mainMemory.getWorstFitHole(proc);
                    break;
                default:
                    System.out.println("Error! Please enter one of
                       these Policies: F B W");
                    return;
            }
            mainMemory.allocate(hole ,proc);
        }
    }
//-------------------- Release Memory Space -----------------
    public static void ReleaseMemLocation(String name){

        // Check if the process exists in the memory
        Partition processPart = mainMemory.findProcess(name);
        if(processPart==null){
            // If process isn't exist, print error message
            System.out.println("Error, process is not found.");
        } else {
            // If process exists, release it
            mainMemory.deallocate(processPart);
        }
    }
//------------------------ Compaction --------------------
    public static void compact(){
        if(mainMemory.getNumberOfHoles() == 1)
            System.out.println("No need to comapct, there is only
               one hole");
        else
            mainMemory.compaction();
    }
//---------------------- Status Report -------------------
    public static void printStatusReport() {
        mainMemory.statusReport();
    }}
```

- Memory Class

```java
import java.util.ArrayList;
public class Memory {

    private final int MAX = 1048576;
    ArrayList<Partition> memory;

    // Constructors
    public Memory(){
        memory = new ArrayList<>(MAX);
        memory.add(new Partition(0, 1048575, MAX));
    }
```

```java
    // Get the size of the largest hole
    public int getMaxHoleSize(){
        int maxHoleSize = 0;
            for (Partition part : memory)
                if(part.isHole() && part.getPTsize() > maxHoleSize)
                    maxHoleSize = part.getPTsize();
        return maxHoleSize;
    }

    // Get the total number of holes
    public int getNumberOfHoles(){
        int total = 0;
        for(Partition part : memory)
            if(part.isHole())
                total++;
        return total;
    }

    // Get first fit hole
    public Partition getFirstFitHole(Processes proc){
        for (Partition part : memory)
            if(part.isHole() && part.getPTsize() >=
               proc.getPSsize())
                return part;
        return null;
    }
    // Get best fit
    public Partition getBestFitHole(Processes proc){
        int procSize = proc.getPSsize();
        Partition bestFit = getFirstFitHole(proc);

        for(Partition part : memory)
            if(part.isHole() && part.getPTsize() >= procSize &&
                    part.getPTsize() < bestFit.getPTsize())
                bestFit = part;
        return bestFit;
    }

    // Get Worst fit (hole with the largest size)
    public Partition getWorstFitHole(Processes proc){
        int procSize = proc.getPSsize();
        Partition worstFit = getFirstFitHole(proc);

        for(Partition part : memory)
            if(part.isHole() && part.getPTsize() >= procSize &&
               part.getPTsize() > worstFit.getPTsize())
                worstFit = part;
        return worstFit;
    }
```

```java
// Allocate memory to a specific process
public void allocate(Partition part, Processes process){
    int index = memory.indexOf(part);
    int endAddress;

    // holeSize = ProcessSize
    if(part.getPTsize() == process.getPSsize()){
        part.setHole(false);
        part.setProcess(process);
    } else {
        endAddress = part.getBase() + (process.getPSsize()-1);
        memory.add(index, new Partition(process,
            part.getBase(), endAddress));
        part.setBase(endAddress+1);
        part.setPTsize(part.getEndAddress()-(part.getBase()-
            1));
    }
}

// Deallocate a specific process from memory
public void deallocate(Partition allocatedPart){
    int index = memory.indexOf(allocatedPart);
    Partition p = allocatedPart; // p is a partition pointer
    p.setHole(true);

    // The second index is already decremented
    while(((--index) > -1) && (memory.get(index).isHole())){
        p = memory.get(index);
    }
    int sumHolesSize = 0;
    int endAddress = p.getEndAddress();

    int nextIndex = memory.indexOf(p)+1;
    while((nextIndex)<memory.size() &&
            memory.get(nextIndex).isHole()){
        sumHolesSize += memory.get(nextIndex).getPTsize();
        endAddress = memory.get(nextIndex).getEndAddress();
        memory.remove(nextIndex);
    }

    // Combine the holes size into one element in the array &
    update values
    p.setPTsize(p.getPTsize() + sumHolesSize);
    p.setEndAddress(endAddress);
    p.setHole(true);
    p.setProcess(null);

}
```

```java
    // Search for a process in memory
    public Partition findProcess(String name){
        Partition p=null; //the partition which has the process
        for (Partition temp : memory){ //loop through the memory
            if(!temp.isHole() &&
        temp.getProcess().getName().equalsIgnoreCase(name)){
                p = temp;
            }
        }
        return p;
    }
    // Compact unused holes of memory into one region
    public void compaction() {
        // Calculate the holes sizes.
        int total = 0;
        Partition part;

        //Loop for every partition in the memory
        for (int k=0; k < memory.size(); k++) {
            part = memory. get(k);
            if (part.isHole()) {
                total += part.getPTsize();
                // The index after the first hole.
                int nextIndex = memory.indexOf(part) + 1;
                // Loop from the next partition.
                for (int i = nextIndex; i < memory.size(); i++) {
                    Partition nextPart = memory.get(i);
                    // Make the processes go back one step.
                    nextPart.setBase(nextPart.getBase() -
                       part.getPTsize());
                    nextPart.setEndAddress(nextPart.getEndAddress()
                       - part.getPTsize());
                }
                memory.remove(part);
            }
        }
        // Get the last partition by getting the memory size.
        Partition lastPartition = memory.get(memory.size()-1);
        // Get the last base: which is the last partition end
        address + 1
        int lastBase = lastPartition.getEndAddress()+1;
        memory.add( new Partition(lastBase, MAX - 1) );
    }
    // Print type of partition (Hole or Process)
    public void printMemory(){
        for(Partition part: memory){
            if(part.isHole()){
                System.out.print("Hole, ");
```

```
            } else {
                System.out.print(part.getProcess().getName() + ",
                        ");
            }}}
    // Print a memory status report
    public void statusReport() {
        memory.forEach((part) -> {
            System.out.println("Addresses [" + part.getBase() + ":"
                + part.getEndAddress() + "] "
                 + (part.isHole() ? "Unused" : ("Process " +
                        part.getProcess().getName())));
        });
    }
}
```

- Partition Class

```
public class Partition {

    private boolean hole = true;        // Default case
    private int base;
    private int endAddress;
    private int PTsize;
    private Processes process;


    // Constructor
    // Defaulf constructor
    public Partition(int base, int endAddress, int size){
        this.base = base;
        this.endAddress = endAddress;
        this.PTsize = size;
    }

    public Partition(Processes proc, int base, int endAddress){
        hole = false;
        process = proc;
        this.base = base;
        this.endAddress = endAddress;
        this.PTsize = endAddress - base + 1;
    }

    // Constructor for compact method
    public Partition(int base, int endAddress){
        this.base = base;
        this.endAddress = endAddress;
        this.PTsize = endAddress - base + 1;
    }
```

```java
// Getters & Setters
    public boolean isHole() {
        return hole;
    }

    public int getBase() {
        return base;
    }

    public int getEndAddress() {
        return endAddress;
    }

    public int getPTsize() {
        return PTsize;
    }

    public Processes getProcess() {
        return process;
    }

    public void setBase(int base) {
        this.base = base;
    }

    public void setHole(boolean hole) {
        this.hole = hole;
    }

    public void setEndAddress(int endAddress) {
        this.endAddress = endAddress;
    }

    public void setProcess(Processes process) {
        this.process = process;
    }

    public void setPTsize(int PRsize) {
        this.PTsize = PRsize;
    }
}
```

- Process Class

```java
public class Processes {

    private String name;
    private int PSsize;
```

```
     // Getters & Setters
    public String getName() {
        return name;
    }

    public int getPSsize() {
        return PSsize;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setPSsize(int PSsize) {
        this.PSsize = PSsize;
    }

}
```

4.2 Virtual Memory’s Source Code

In the implementation of the virtual memory, we used the same Object-oriented programming concepts. So, we implemented four classes: the VirtualMemory class as a Main class, the Memory class, the Frame class, and the Address class.

- VirtualMemory Class

```
// CPCS361 Group Project - Winter 2023
// Virtual Memory Class (Main Class)
package cpcs361group5part2;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import java.util.Scanner;

public class VirtualMemory {

    // Declare variables for page size & number of frames
    static int pageSize = 256;
    static int numberOfFrames;
```

```java
public static void main(String[] args) throws
IndexOutOfBoundsException , FileNotFoundException {

        //========= Specifications ==========
        // initialize number of frames
        numberOfFrames = 256;

        // Implement the Page Table as an Array of 2^8 entries
        int[] pageTable = new int[256];
        int logicalAddress, memValue, savedMemValue, maskedAddress,
                offset, pageNo, frameNo=-1, translatedAddress;

        // Create new random object to generate random numbers
within a range
        Random randNo = new Random();

        // initialize the page table values with -1 (empty)
        initArr(pageTable);

        // Create the physical memory (Page Replacement -> new size
= 128 page*256 page size)
        int memSize = pageSize * numberOfFrames;    // PageSize
(256) X frames (256) = 65,536
        Memory physicalMem = new Memory(memSize , numberOfFrames ,
pageSize);

        // Create two arrays for test cases
        int[][] testCase1 = new int[30][2]; //[logical
addresses][value]
        int[][] testCase2 = new int[80][2]; //[logical
addresses][value]

        // Create an address string array to store the 80 address
(used later for printing)
        String[] addressString = new String[80];

        //======== Files Section =========
        // Input & Output Files
        File memAddressesFile = new File("addresses.txt");
        File memValuesFile = new File("correct.txt");
        File memAddr4PRFile = new File("addresses for Page
Replacements.txt");
        File outputFile = new File("output.txt");

        // Files existance check
        fileCheck(memAddressesFile);
        fileCheck(memValuesFile);
        fileCheck(memAddr4PRFile);
        // Input Scanner objects to read from files
        Scanner inputAddress = new Scanner(memAddressesFile);
        Scanner inputMemValue = new Scanner(memValuesFile);
        Scanner memAddr4PR = new Scanner(memAddr4PRFile);
```

```
// Output PrintWriter object to write on files
        PrintWriter output = new PrintWriter(outputFile);

        // ======== Address Translation ==========
        // Read 100 logical addresses from addresses.txt file one-
by-one
        for (int i = 0; i < 100; i++) {

            // Read the logical address and mask the rightmost 16-
bits
            logicalAddress = (inputAddress.nextInt()) & 65535;
            // Read memory value
            memValue = inputMemValue.nextInt();

            // Extract the page number & The offset from the
logical address
            pageNo = findPageNo(logicalAddress);    // 1st 8-bits
for pageNo
            offset = findOffset(logicalAddress);    // Last 8-bits
for offset

            // 1- Make sure that the page number is not in the Page
Table
            // 2- if it is, give a random number as a frame number
            if (pageTable[pageNo] == -1) {
                frameNo = randNo.nextInt(numberOfFrames);
                pageTable[pageNo] = frameNo;
            }

            // Compute the physical address ((frame number * frame
size) + offset)
            translatedAddress = transLA(pageTable[pageNo] ,
pageSize , offset);

            // Access the frame from physical memory using the
frameNo
            Frame frame = physicalMem.findFrame(pageTable[pageNo]);
            // Access the frame address from physical memory using
the offset
            Address physAddress = frame.getAddress(offset);

            // If Phys.Address. is previously allocated find a new
random number as a frame number
            while(physAddress.getValue()!=-1){
                frameNo = randNo.nextInt(numberOfFrames);
                pageTable[pageNo] = frameNo;
                frame = physicalMem.findFrame(frameNo);
                physAddress = frame.getAddress(offset);
            }
            // Add the value from correct file to the physical
address
            physAddress.setValue(memValue);
```

```
            // Print the Signed Byte in the output file
            output.println(physAddress.getValue());
            if (i < 30) {
                testCase1[i][0] = logicalAddress;
                testCase1[i][1] = memValue;
            } else if (i < 80) {
                testCase2[i - 30][0] = logicalAddress;
                testCase2[i - 30][1] = memValue;
            }
        }
        // ============= Five Test Cases ===============
        // ============= The First Test ==============

        System.out.println("\n———————————— The Five Test Cases ——————
————————\n");
        System.out.printf("%-20s%-10s%-10s%-10s%-10s%-15s\n\n" ,
                "Logical Address", "Page #", "Offset", "Frame #",
"Value", "Same as model answer");

        for (int i = 0; i < 5; i++) {
            // Test random address
            int index = randNo.nextInt(30);

            // Get the logical address
            logicalAddress = (testCase1[index][0]) & 65535;
            // Save the memory value from file correct
            memValue = testCase1[index][1];

            // Find the page number
            pageNo = findPageNo(logicalAddress);
            // Find the offset
            offset = findOffset(logicalAddress);

            // Find the frame number from the page table
            frameNo = pageTable[pageNo];

            // Translate the logical address to the physical
address
            translatedAddress = transLA(frameNo, pageSize, offset);

            // Find the frame & offset using frameNo
            Frame frame = physicalMem.findFrame(frameNo);
            Address physAddress = frame.getAddress(offset);

            // Find the saved memory value from frame
            savedMemValue = physAddress.getValue();

            // Check if the stored Signed Byte in phys. Mem is
equal to the test Signed Byte
            String validation = (memValue == savedMemValue) ? "Yes"
: "No";
```

```java
            System.out.printf("%-20d%-10d%-10d%-10d%-10d%-15s\n" ,
                    logicalAddress, pageNo, offset, frameNo,
savedMemValue, validation);
        }

        // ============ Statistics ============
        // ============ The Second Test ============

        // Skip 800 readings just to make sure to read new logical
addresses in the next test case
        int endIndex = randNo.nextInt(800);
        for (int i = 0; i < endIndex; i++) {
            logicalAddress = inputAddress.nextInt();//skip
        }

        // Read another 30 addresses different from the 100
adresses populated before
        for (int i = 50; i < 80; i++) {
            // Read address
            logicalAddress = inputAddress.nextInt();
            // Read memory value
            memValue = inputMemValue.nextInt();

            testCase2[i][0] = logicalAddress;
            testCase2[i][1] = memValue;
        }

        // Both types of addresses should be intermixed
        List<int[]> pair = new ArrayList<>();
        pair.addAll(Arrays.asList(testCase2));
        Collections.shuffle(pair);

        // Store logical addresses in the Adresses string (for
printing)
        for (int i = 0; i < pair.size(); i++) {
            addressString[i] = "" + pair.get(i)[0];
        }

        // Print Results
        System.out.println("\n\n—————————————— Statistics ——————————
—————");
        // Create counters for page faults & hits
        int pageFault = 0;
        int pageHit = 0;
        for (int i = 0; i < 80; i++) {
            pageNo = findPageNo(pair.get(i)[0]);
            if (pageTable[pageNo] == -1) {
                pageFault++;
            } else {
                pageHit++; }}
```

```
        System.out.println("\nLength of address string is 80\n");
        for (int i = 0; i < 80; i++) {
            if (i % 10 == 0 && i != 0) { // Print 10 addresses per
line
                System.out.println();
            }
            System.out.printf("%-5s ", addressString[i]);

        }
        System.out.println();
        System.out.println("\n■ The Number of page faults: " +
pageFault
                + "\n■ The percentage of address references that
resulted in page faults is %"
                + (pageFault / 80.0) * 100.0 + "\n■ The Number of
page hits: " + pageHit);


        // ========= Page Replacement =========
        // Change the size of the physical memory to 128. generate
133 logical addresses
        Memory pageReplacementMemory = new Memory(128*256 , 128,
256); // New memory size is 32768

        // Create a new MyTestData2 (key-value pair): store
addresses in random order
        // (i.e. 133 entry in the vector), for each address assign
        // A random signed byte value
        int[][] MyTestData2 = new int[133][2];
        int[] pageTableTestData2 = new int[256];
        // Initialize pageTable with -1s
        initArr(pageTableTestData2);
        ArrayList<Integer> FIFO = new ArrayList<>();
        int victim;

        System.out.printf("\n────────────── The testing of the page
replacement #1 ──────────────\n"
                + "\n%-20s%-15s%-15s%-15s\n\n" ,
                "Logical Address" , "New Page #" , "Victim Page #"
, "Reused Frame #");

        List<Integer> frameNoArr = generateRand(128);

        for (int i = 0; i < 133; i++) {
            // Read logical address & signed byte
            MyTestData2[i][0] = memAddr4PR.nextInt();
            MyTestData2[i][1] = memAddr4PR.nextInt();
            // Extract Page number & offset
            maskedAddress = MyTestData2[i][0] & 65535;
            pageNo = findPageNo(maskedAddress);
            offset = findOffset(maskedAddress);
```

```
            // If memory is full, perform page replacement
            if (isFull(pageReplacementMemory)) {
                victim = FIFO.remove(0);
        // Choose random page to replace
                frameNo = pageTableTestData2[victim];
        // Store the frameNo of the victim
                pageTableTestData2[victim] = -1;
        // Set the frame# of that page to -1
                System.out.printf("%-20d%-15d%-15d%-15d\n" ,
                        maskedAddress , pageNo , victim , frameNo);
                pageTableTestData2[pageNo] = frameNo;
        // Add the frameNo to the page table
                FIFO.add(pageNo);
            } else {
                frameNo = frameNoArr.get(i);
        // Get the randomized frameNo
                pageTableTestData2[pageNo] = frameNo;
        // Store that frameNo in the page table

pageReplacementMemory.findFrame(frameNo).setUsed(true);
        //Set the frame status to 'used' in the phys.mem.
                FIFO.add(pageNo);
        // Add the pageNo to the Queue for future replacement
            }

            translatedAddress = transLA(pageTableTestData2[pageNo],
256, offset);
            Frame frame =
pageReplacementMemory.findFrame(pageTableTestData2[pageNo]);
            Address physAddress = frame.getAddress(offset);
            physAddress.setValue(MyTestData2[i][1]);
        // Store the signed byte value in the PhysMem
        }

        // =========== Print Results ============
        System.out.printf("\n-------------- The testing of the page
replacement #2 --------------\n"
                +"\n%-20s%-15s%-15s%-15s%-15s%-20s\n" ,
                "Logical Address" , "Page #" , "Offset" , "Frame #"
, "Value" , "Same as model answer");

        for (int i = 128; i < 133; i++) {
            logicalAddress = MyTestData2[i][0];
            memValue = MyTestData2[i][1];

            maskedAddress = logicalAddress & 65535;
            pageNo = findPageNo(maskedAddress);
            offset = findOffset(maskedAddress);
            frameNo = pageTableTestData2[pageNo];
```

```
            System.out.printf("\n%-20d%-15d%-15d%-15d%-15d%-20s" ,
                maskedAddress , pageNo , offset , frameNo ,
savedMemValue , validation);
        }

        System.out.println();
        inputAddress.close();
        output.close();
    }

    // Initialize an array elements with -1
    public static void initArr(int[] arr) {
        //initialize page taple values with -1
        for (int i = 0; i < arr.length; i++) {
            arr[i] = -1;
        }
    }

    // Extract page number from 16-bit
    public static int findPageNo(int address) {
        return address / pageSize;
    }

    // Extract offset from 16-bit
    public static int findOffset(int address) {
        return address % pageSize;
    }

    // Translate Logical address to Physical address.
    public static int transLA(int frameNo , int pageSize , int
offset) {
        return ((frameNo * pageSize) + offset);
    }

    // Generate a shuffled array with numbers ranged between (0-
limit)
    public static List<Integer> generateRand(int limit) {

        List<Integer> numbers = new ArrayList<>();
        for (int i = 0; i < limit; i++) {
            numbers.add(i);
        }
        Collections.shuffle(numbers);
        return numbers;
    }
    // Check if the memory is full or not.
    public static boolean isFull(Memory memory) {
        for (int i = 0; i < memory.getMemorySize(); i++) {
            if (!memory.findFrame(i).getUsed()) {
                //there is at least one frame free
                return false;
```

```
            }
        }
        // memory is full
        return true;
    }

    // A method to check whether a specific file exists or not.
    public static void fileCheck(File f){
        if (!f.exists()) {
            System.out.println("File " + f.getName() + " doesn't
exist");
            System.exit(0);
        }
    }
}
```

- Memory Class

```
package cpcs361group5part2;

import java.util.ArrayList;

public class Memory {

    ArrayList<Frame> memory;

    // Constructors
    public Memory(int memSize , int numberOfFrames , int frameSize)
{
        memory = new ArrayList<>(numberOfFrames);

        // create frames
        int base = 0;
        int endAddress = frameSize;
        for (int i = 0; i < numberOfFrames; i++) {
            memory.add(new Frame(i, base, endAddress, frameSize));
            base += frameSize;
            endAddress += frameSize;
        }
    }

    // Other methods
    // Get memory size (number of partitions)
    public int getMemorySize() {
        return memory.size();
    }

    // Get frame by frame number
    public Frame findFrame(int frameNo) {
```

```java
        for (Frame frame : memory) { //loop through the memory
            if (frame.getFrameNo() == frameNo) {
                return frame;
            }
        }
        return null;
    }

    public boolean isFull() {
        for (int i = 0; i < memory.size(); i++) {
            if (!memory.get(i).getUsed()) {
                //there is at least one frame free
                return false;
            }
        }
        // memory is full
        return true;
    }
}
```

- Frame Class

```java
package cpcs361group5part2;

public class Frame {

    private int frameNo;
    private int base;
    private int endAddress;
    private int frameSize;
    private Address[] addresses;
    private boolean used = false;


    // Constructor
    public Frame(int frameNo, int base, int endAddress, int
frameSize){
        this.frameNo = frameNo;
        this.base = base;
        this.endAddress = endAddress;
        this.frameSize = frameSize;

        // set addresses value
        addresses = new Address[this.frameSize];
        int address = base;
        for (int i = 0; i < frameSize; i++) {
            addresses[i] = new Address(address++);
        }
    }
```

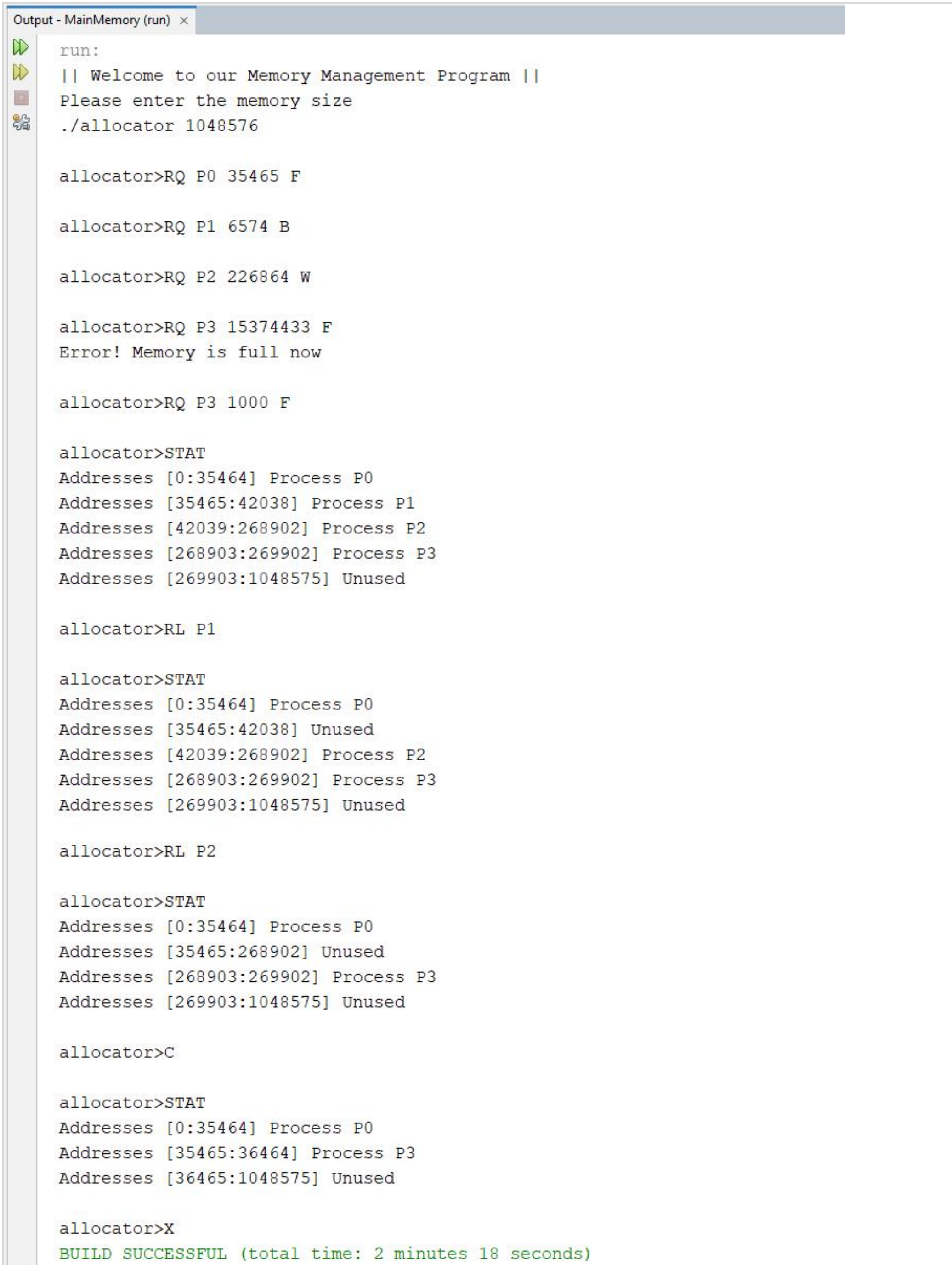
```java
    // Getters & Setters
    public int getBase() {
        return base;
    }

    public int getEndAddress() {
        return endAddress;
    }

    public int getSize() {
        return frameSize;
    }

    public Address getAddress(int index) {
        return addresses[index];
    }

    public int getFrameNo() {
        return this.frameNo;
    }

    public boolean getUsed() {
        return this.used;
    }

    public void setUsed(boolean used) {
        this.used = used;
    }

    public void setFrameNo(int frameNo) {
        this.frameNo = frameNo;
    }

    public void setBase(int base) {
        this.base = base;
    }

    public void setEndAddress(int endAddress) {
        this.endAddress = endAddress;
    }

    public void setAddress(int index, Address newAddress) {
        this.addresses[index] = newAddress;
    }

    public void setSize(int size) {
        this.frameSize = size;
    }

}
```

- Address Class

```
package cpcs361group5part2;

public class Address {

    private int addressNo;
    private int value;
    private int frameNo;

    // Constructor
    public Address(int addressNo) {
        this.addressNo = addressNo;
        this.value = -1;
    }

    public Address(int addressNo , int value) {
        this.addressNo = addressNo;
        this.value = value;
    }

    // Getters & Setters
    public int getAddressNo() {
        return addressNo;
    }

    public int getValue() {
        return value;
    }

    public int getFrameNo() {
        return frameNo;
    }

    public void setFrameNo(int frameNo) {
        this.frameNo = frameNo;
    }

    public void setAddressNo(int addressNo) {
        this.addressNo = addressNo;
    }

    public void setValue(int value) {
        this.value = value;
    }

}
```

5. Program Outputs

5.1 Main Memory Output (NetBeans IDE)



Output - MainMemory (run)

```
run:
|| Welcome to our Memory Management Program ||
Please enter the memory size
./allocator 1048576

allocator>RQ P0 35465 F

allocator>RQ P1 6574 B

allocator>RQ P2 226864 W

allocator>RQ P3 15374433 F
Error! Memory is full now

allocator>RQ P3 1000 F

allocator>STAT
Addresses [0:35464] Process P0
Addresses [35465:42038] Process P1
Addresses [42039:268902] Process P2
Addresses [268903:269902] Process P3
Addresses [269903:1048575] Unused

allocator>RL P1

allocator>STAT
Addresses [0:35464] Process P0
Addresses [35465:42038] Unused
Addresses [42039:268902] Process P2
Addresses [268903:269902] Process P3
Addresses [269903:1048575] Unused

allocator>RL P2

allocator>STAT
Addresses [0:35464] Process P0
Addresses [35465:268902] Unused
Addresses [268903:269902] Process P3
Addresses [269903:1048575] Unused

allocator>C

allocator>STAT
Addresses [0:35464] Process P0
Addresses [35465:36464] Process P3
Addresses [36465:1048575] Unused

allocator>X
BUILD SUCCESSFUL (total time: 2 minutes 18 seconds)
```

Figure 10: Main Memory Output using NetBeans IDE.

5.2 Virtual Memory Output (NetBeans IDE)

```
Output - VirtualMemory (run)

run:

——————————————— The Five Test Cases ———————————————

Logical Address     Page #    Offset    Frame #    Value     Same as model answer

64243               250       243       172        -68       Yes
27966               109       62        96         27        Yes
6727                26        71        90         -111      Yes
7591                29        167       203        105       Yes
57982               226       126       77         56        Yes


—————————————————— Statistics ——————————————————

Length of address string is 80

24380 43121 17665 10385 44770 33405 63829 27444 20259 8565
51476 19602 43765 8800  44954 42034 969   34561 45688 45457
58554 38775 45737 6091  5003  2919  19498 3067  10842 49294
50227 30711 42932 31649 58882 3884  59240 16399 41934 48128
41118 40891 12453 21395 59667 5129  63258 692   30705 56657
19358 28290 50552 17661 10392 28964 8620  8362  49640 3784
17884 33856 38336 52782 47894 46919 54388 15757 18145 32541
36529 36922 44817 42252 22514 60086 57998 28928 52335 52704

■ The Number of page faults: 17
■ The percentage of address references that resulted in page faults is %21.25
■ The Number of page hits: 63

—————————————————— The testing of the page replacement #1 ——————————————————

Logical Address     New Page #     Victim Page #   Reused Frame #

22048               86             72              28
26867               104            17              11
31880               124            98              0
18815               73             4               35
33958               132            26              72

—————————————————— The testing of the page replacement #2 ——————————————————

Logical Address     Page #         Offset          Frame #         Value           Same as model answer

22048               86             32              28              40              Yes
26867               104            243             11              -65             Yes
31880               124            136             0               -115            Yes
18815               73             127             35              -53             Yes
33958               132            166             72              -119            Yes
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 11: Virtual Memory Output using NetBeans IDE.

To check the output.txt file (Click Here)

To check Test cases output files (Test case 1) (Test case 2) (Test case 3)

6. References

Operating System - Memory Management. (n.d.). Retrieved Feb 15, 2023, from TutorialsPoint: https://www.tutorialspoint.com/operating_system/os_memory_management.htm

Silberschatz, A., Beran, J., Gagne, G., & Galvin, P. (2008). Operating System Concepts. United States of America.