

**School of Computer Science and Engineering (SCOPE)**

**B.Tech. Computer Science and Engineering**

**Specialisation in Artificial Intelligence and Machine Learning**

**BCSE303L**

**OPERATING SYSETMS**

SLOT: F2 & TF2

LAB REPORT ON

VIRTUAL MEMORY MANAGER

SUBMITTED BY:

DIVYANSHU PATEL

23BAI1214

SUBMITTED TO:

PROF SIVAGAMI M

**Table of Contents**

1. Introduction to Virtual Memory Manager
   1. Memory Allocation Techniques Simulation (Contiguous)
2. First-Fit Memory Allocation
3. Next-Fit Memory Allocation
4. Best-Fit Memory Allocation
5. Worst-Fit Memory Allocation
6. Comparative Analysis of Memory Allocation Techniques

1.2 Page Replacement Algorithm Simulation

1. FIFO (First-In-First-Out) Page Replacement
2. LRU (Least Recently Used) Page Replacement
3. Clock (Second Chance) Page Replacement
4. Optimal Page Replacement
5. LFU (Least Frequently Used) Page Replacement
6. MFU (Most Frequently Used) Page Replacement
7. Comparative Analysis of Page Replacement Algorithms

1.3 Frame Allocation Techniques Simulation

1. Equal Frame Allocation
2. Proportional Frame Allocation
3. Priority-Based Frame Allocation
4. Comparative Analysis of Frame Allocation Techniques
5. Introduction to Programming Projects

**Reference: OPERATING SYSTEM CONCEPTS ABRAHAM SILBERSCHATZ 10th. Edition**

(Standard Textbook as mentioned in syallabus)

1. Problem Statement given in **ABRAHAM SILBERSCHATZ Textbook**
2. Objectives and Deliverables of the Project
3. Background and Problem Description
4. Address Translation
5. Handling Page Faults
6. Test File
7. Elementary Requirements
8. Page Replacement
9. Statistics
10. Implementation Details and Methods
11. Basic Data Structure
12. Initialize Variables
13. Get and Parse Addresses
14. LRU Replacement in TLB
15. LRU Replacement in Memory
16. Main Part of code

1. Program Results
2. Compare result.txt with correct.txt
3. Statistics
4. Introduction to Virtual Memory Manager

A **Virtual Memory Manager (VMM)** is a system component that manages the mapping between virtual memory (used by programs) and physical memory (RAM). It allows programs to use more memory than physically available by swapping data between RAM and disk storage, ensuring efficient memory usage and isolation between processes. The VMM handles tasks like address translation, page allocation, and page replacement.

Key Concepts:

Memory Allocation Techniques

Page Replacement Algorithm

Frame Allocation Techniques

1.1 Memory Allocation Techniques Simulation

1. First-Fit Memory Allocation
2. Next-Fit Memory Allocation
3. Best-Fit Memory Allocation
4. Worst-Fit Memory Allocation
5. Buddy System

#*include* <stdio.h>

#*include* <stdbool.h>

#*include* <limits.h>

#*define* *MAX\_PARTITIONS* 5

#*define* *MAX\_PROCESSES* 4

void *displayAllocation*(int allocation*[]*, int processes*[]*, int partitions*[]*, int n\_processes) {

*for* (int i = 0; i < n\_processes; i++) {

*printf*("Process %d (%d): ", i + 1, processes[i]);

*if* (allocation[i] != -1) {

*printf*("Allocated to Partition %d (%d)\n",

                   allocation[i] + 1,

                   partitions[allocation[i]]);

        } *else* {

*printf*("Not Allocated\n");

        }

    }

}

// *Fixed Size Partitioning Algorithms*

void *firstFitFixed*(int partitions*[]*, int n\_partitions, int processes*[]*, int n\_processes) {

*bool* isAllocated[*MAX\_PARTITIONS*] = {*false*};

    int allocation[*MAX\_PROCESSES*];

*for* (int i = 0; i < n\_processes; i++)

        allocation[i] = -1;

*for* (int i = 0; i < n\_processes; i++) {

*for* (int j = 0; j < n\_partitions; j++) {

*if* (!isAllocated[j] && partitions[j] >= processes[i]) {

                allocation[i] = j;

                isAllocated[j] = *true*;

*break*;

            }

        }

    }

*printf*("\nFirst Fit (Fixed) Allocation:\n");

*displayAllocation*(allocation, processes, partitions, n\_processes);

}

void *nextFitFixed*(int partitions*[]*, int n\_partitions, int processes*[]*, int n\_processes) {

*bool* isAllocated[*MAX\_PARTITIONS*] = {*false*};

    int allocation[*MAX\_PROCESSES*];

    int lastAllocated = -1;

*for* (int i = 0; i < n\_processes; i++)

        allocation[i] = -1;

*for* (int i = 0; i < n\_processes; i++) {

        int j = (lastAllocated + 1) % n\_partitions;

        int count = 0;

*while* (count < n\_partitions) {

*if* (!isAllocated[j] && partitions[j] >= processes[i]) {

                allocation[i] = j;

                isAllocated[j] = *true*;

                lastAllocated = j;

*break*;

            }

            j = (j + 1) % n\_partitions;

            count++;

        }

    }

*printf*("\nNext Fit (Fixed) Allocation:\n");

*displayAllocation*(allocation, processes, partitions, n\_processes);

}

void *bestFitFixed*(int partitions*[]*, int n\_partitions, int processes*[]*, int n\_processes) {

*bool* isAllocated[*MAX\_PARTITIONS*] = {*false*};

    int allocation[*MAX\_PROCESSES*];

*for* (int i = 0; i < n\_processes; i++)

        allocation[i] = -1;

*for* (int i = 0; i < n\_processes; i++) {

        int bestIdx = -1;

        int minDiff = *INT\_MAX*;

*for* (int j = 0; j < n\_partitions; j++) {

*if* (!isAllocated[j] && partitions[j] >= processes[i]) {

*if* (partitions[j] - processes[i] < minDiff) {

                    minDiff = partitions[j] - processes[i];

                    bestIdx = j;

                }

            }

        }

*if* (bestIdx != -1) {

            allocation[i] = bestIdx;

            isAllocated[bestIdx] = *true*;

        }

    }

*printf*("\nBest Fit (Fixed) Allocation:\n");

*displayAllocation*(allocation, processes, partitions, n\_processes);

}

void *worstFitFixed*(int partitions*[]*, int n\_partitions, int processes*[]*, int n\_processes) {

*bool* isAllocated[*MAX\_PARTITIONS*] = {*false*};

    int allocation[*MAX\_PROCESSES*];

*for* (int i = 0; i < n\_processes; i++)

        allocation[i] = -1;

*for* (int i = 0; i < n\_processes; i++) {

        int worstIdx = -1;

        int maxDiff = -1;

*for* (int j = 0; j < n\_partitions; j++) {

*if* (!isAllocated[j] && partitions[j] >= processes[i]) {

*if* (partitions[j] > maxDiff) {

                    maxDiff = partitions[j];

                    worstIdx = j;

                }

            }

        }

*if* (worstIdx != -1) {

            allocation[i] = worstIdx;

            isAllocated[worstIdx] = *true*;

        }

    }

*printf*("\nWorst Fit (Fixed) Allocation:\n");

*displayAllocation*(allocation, processes, partitions, n\_processes);

}

// *Variable Size Partitioning Algorithms*

void *firstFitVariable*(int partitions*[]*, int n\_partitions, int processes*[]*, int n\_processes) {

    int allocation[*MAX\_PROCESSES*];

    int remainingSpace[*MAX\_PARTITIONS*];

*for* (int i = 0; i < n\_partitions; i++)

        remainingSpace[i] = partitions[i];

*for* (int i = 0; i < n\_processes; i++)

        allocation[i] = -1;

*for* (int i = 0; i < n\_processes; i++) {

*for* (int j = 0; j < n\_partitions; j++) {

*if* (remainingSpace[j] >= processes[i]) {

                allocation[i] = j;

                remainingSpace[j] -= processes[i];

*break*;

            }

        }

    }

*printf*("\nFirst Fit (Variable) Allocation:\n");

*displayAllocation*(allocation, processes, partitions, n\_processes);

}

void *nextFitVariable*(int partitions*[]*, int n\_partitions, int processes*[]*, int n\_processes) {

    int allocation[*MAX\_PROCESSES*];

    int remainingSpace[*MAX\_PARTITIONS*];

    int lastAllocated = -1;

*for* (int i = 0; i < n\_partitions; i++)

        remainingSpace[i] = partitions[i];

*for* (int i = 0; i < n\_processes; i++)

        allocation[i] = -1;

*for* (int i = 0; i < n\_processes; i++) {

        int j = (lastAllocated + 1) % n\_partitions;

        int count = 0;

*while* (count < n\_partitions) {

*if* (remainingSpace[j] >= processes[i]) {

                allocation[i] = j;

                remainingSpace[j] -= processes[i];

                lastAllocated = j;

*break*;

            }

            j = (j + 1) % n\_partitions;

            count++;

        }

    }

*printf*("\nNext Fit (Variable) Allocation:\n");

*displayAllocation*(allocation, processes, partitions, n\_processes);

}

void *bestFitVariable*(int partitions*[]*, int n\_partitions, int processes*[]*, int n\_processes) {

    int allocation[*MAX\_PROCESSES*];

    int remainingSpace[*MAX\_PARTITIONS*];

*for* (int i = 0; i < n\_partitions; i++)

        remainingSpace[i] = partitions[i];

*for* (int i = 0; i < n\_processes; i++)

        allocation[i] = -1;

*for* (int i = 0; i < n\_processes; i++) {

        int bestIdx = -1;

        int minDiff = *INT\_MAX*;

*for* (int j = 0; j < n\_partitions; j++) {

*if* (remainingSpace[j] >= processes[i]) {

*if* (remainingSpace[j] - processes[i] < minDiff) {

                    minDiff = remainingSpace[j] - processes[i];

                    bestIdx = j;

                }

            }

        }

*if* (bestIdx != -1) {

            allocation[i] = bestIdx;

            remainingSpace[bestIdx] -= processes[i];

        }

    }

*printf*("\nBest Fit (Variable) Allocation:\n");

*displayAllocation*(allocation, processes, partitions, n\_processes);

}

void *worstFitVariable*(int partitions*[]*, int n\_partitions, int processes*[]*, int n\_processes) {

    int allocation[*MAX\_PROCESSES*];

    int remainingSpace[*MAX\_PARTITIONS*];

*for* (int i = 0; i < n\_partitions; i++)

        remainingSpace[i] = partitions[i];

*for* (int i = 0; i < n\_processes; i++)

        allocation[i] = -1;

*for* (int i = 0; i < n\_processes; i++) {

        int worstIdx = -1;

        int maxDiff = -1;

*for* (int j = 0; j < n\_partitions; j++) {

*if* (remainingSpace[j] >= processes[i]) {

*if* (remainingSpace[j] > maxDiff) {

                    maxDiff = remainingSpace[j];

                    worstIdx = j;

                }

            }

        }

*if* (worstIdx != -1) {

            allocation[i] = worstIdx;

            remainingSpace[worstIdx] -= processes[i];

        }

    }

*printf*("\nWorst Fit (Variable) Allocation:\n");

*displayAllocation*(allocation, processes, partitions, n\_processes);

}

int *main*() {

    int partitions*[]* = {150, 350};

    int processes*[]* = {300, 25, 125, 50};

    int n\_partitions = sizeof(partitions) / sizeof(partitions[0]);

    int n\_processes = sizeof(processes) / sizeof(processes[0]);

    int tempPartitionsFixed[*MAX\_PARTITIONS*];

    int tempPartitionsVariable[*MAX\_PARTITIONS*];

*printf*("Fixed Size Partitioning:\n");

*for* (int i = 0; i < n\_partitions; i++)

        tempPartitionsFixed[i] = partitions[i];

*firstFitFixed*(tempPartitionsFixed, n\_partitions, processes, n\_processes);

*for* (int i = 0; i < n\_partitions; i++)

        tempPartitionsFixed[i] = partitions[i];

*nextFitFixed*(tempPartitionsFixed, n\_partitions, processes, n\_processes);

*for* (int i = 0; i < n\_partitions; i++)

        tempPartitionsFixed[i] = partitions[i];

*bestFitFixed*(tempPartitionsFixed, n\_partitions, processes, n\_processes);

*for* (int i = 0; i < n\_partitions; i++)

        tempPartitionsFixed[i] = partitions[i];

*worstFitFixed*(tempPartitionsFixed, n\_partitions, processes, n\_processes);

*printf*("\nVariable Size Partitioning:\n");

*for* (int i = 0; i < n\_partitions; i++)

        tempPartitionsVariable[i] = partitions[i];

*firstFitVariable*(tempPartitionsVariable, n\_partitions, processes, n\_processes);

*for* (int i = 0; i < n\_partitions; i++)

        tempPartitionsVariable[i] = partitions[i];

*nextFitVariable*(tempPartitionsVariable, n\_partitions, processes, n\_processes);

*for* (int i = 0; i < n\_partitions; i++)

        tempPartitionsVariable[i] = partitions[i];

*bestFitVariable*(tempPartitionsVariable, n\_partitions, processes, n\_processes);

*for* (int i = 0; i < n\_partitions; i++)

        tempPartitionsVariable[i] = partitions[i];

*worstFitVariable*(tempPartitionsVariable, n\_partitions, processes, n\_processes);

*return* 0;

}

**BUDDY SYSTEM :**

#*include* <stdio.h>

#*include* <math.h>

#*include* <stdbool.h>

void *displayAllocation*(int process\_size, int process\_num, int partition\_num) {

*if* (partition\_num != -1)

*printf*("Process %d (%d): Allocated to Partition %d\n",

               process\_num + 1, process\_size, partition\_num + 1);

*else*

*printf*("Process %d (%d): Not Allocated\n", process\_num + 1, process\_size);

}

int *getNextPowerOf2*(int n) {

*return* *pow*(2, *ceil*(*log2*(n)));

}

void *buddySystem*(int partitions*[]*, int n\_partitions, int processes*[]*, int n\_processes) {

    int temp\_partitions[n\_partitions];

*for*(int i = 0; i < n\_partitions; i++)

        temp\_partitions[i] = partitions[i];

*printf*("\nBuddy System Allocation:\n");

*for*(int i = 0; i < n\_processes; i++) {

        int required\_size = *getNextPowerOf2*(processes[i]);

        int allocated = -1;

*for*(int j = 0; j < n\_partitions; j++) {

*if*(temp\_partitions[j] >= required\_size) {

*while*(temp\_partitions[j] > required\_size &&

                      (temp\_partitions[j]/2) >= required\_size) {

                    temp\_partitions[j] /= 2;

                }

*if*(temp\_partitions[j] >= required\_size) {

                    allocated = j;

                    temp\_partitions[j] -= required\_size;

*break*;

                }

            }

        }

*displayAllocation*(processes[i], i, allocated);

    }

}

int *main*() {

    // *powers of 2 for buddy system*

    int partitions*[]* = {128, 512, 256, 256, 512};

    int processes*[]* = {212, 417, 112, 426};

    int n\_partitions = sizeof(partitions) / sizeof(partitions[0]);

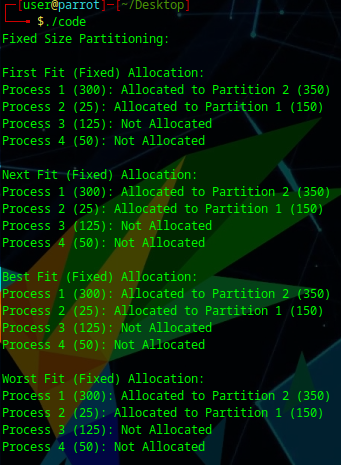
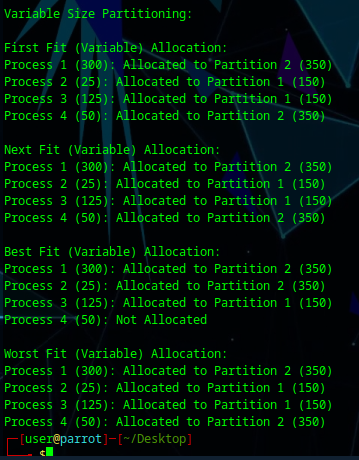
    int n\_processes = sizeof(processes) / sizeof(processes[0]);

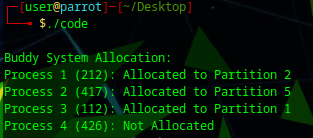
*buddySystem*(partitions, n\_partitions, processes, n\_processes);

*return* 0;

}

1. Comparative Analysis of Memory Allocation Techniques



1.2 Page Replacement Algorithm Simulation

1. FIFO (First-In-First-Out) Page Replacement

void *fifo*(int pages*[]*, int n\_pages, int n\_frames) {

    int frames[MAX\_FRAMES], current = 0, hits = 0, misses = 0;

*for* (int i = 0; i < n\_frames; i++) frames[i] = EMPTY;

*printHeader*("I. FIFO (First-In-First-Out)");

*for* (int i = 0; i < n\_pages; i++) {

*printf*("Page request %d: ", pages[i]);

*if* (!*pageExists*(frames, n\_frames, pages[i])) {

            frames[current] = pages[i];

            current = (current + 1) % n\_frames;

            misses++;

*printf*("Miss! ");

        } *else* {

            hits++;

*printf*("Hit! ");

        }

*displayFrames*(frames, n\_frames);

*printf*("\n");

    }

*printf*("Total Hits: %d, Misses: %d\n", hits, misses);

}

1. LRU (Least Recently Used) Page Replacement

void *lru*(int pages*[]*, int n\_pages, int n\_frames) {

    int frames[MAX\_FRAMES], last\_used[MAX\_FRAMES], hits = 0, misses = 0;

*for* (int i = 0; i < n\_frames; i++) {

        frames[i] = EMPTY;

        last\_used[i] = EMPTY;

    }

*printHeader*("II. LRU (Least Recently Used)");

*for* (int i = 0; i < n\_pages; i++) {

*printf*("Page request %d: ", pages[i]);

*if* (!*pageExists*(frames, n\_frames, pages[i])) {

            int lru\_idx = 0, min\_time = INT\_MAX;

*for* (int j = 0; j < n\_frames; j++) {

*if* (frames[j] == EMPTY) {

                    lru\_idx = j;

*break*;

                }

*if* (last\_used[j] < min\_time) {

                    min\_time = last\_used[j];

                    lru\_idx = j;

                }

            }

            frames[lru\_idx] = pages[i];

            last\_used[lru\_idx] = i;

            misses++;

*printf*("Miss! ");

        } *else* {

*for* (int j = 0; j < n\_frames; j++)

*if* (frames[j] == pages[i])

                    last\_used[j] = i;

            hits++;

*printf*("Hit! ");

        }

*displayFrames*(frames, n\_frames);

*printf*("\n");

    }

*printf*("Total Hits: %d, Misses: %d\n", hits, misses);

}

1. Clock (Second Chance) Page Replacement

void *clock*(int pages*[]*, int n\_pages, int n\_frames) {

    int frames[MAX\_FRAMES], second\_chance[MAX\_FRAMES], pointer = 0;

    int hits = 0, misses = 0;

*for* (int i = 0; i < n\_frames; i++) {

        frames[i] = EMPTY;

        second\_chance[i] = 0;

    }

*printHeader*("III. Clock (Second Chance)");

*for* (int i = 0; i < n\_pages; i++) {

*printf*("Page request %d: ", pages[i]);

*if* (!*pageExists*(frames, n\_frames, pages[i])) {

*while* (second\_chance[pointer]) {

                second\_chance[pointer] = 0;

                pointer = (pointer + 1) % n\_frames;

            }

            frames[pointer] = pages[i];

            second\_chance[pointer] = 1;

            pointer = (pointer + 1) % n\_frames;

            misses++;

*printf*("Miss! ");

        } *else* {

*for* (int j = 0; j < n\_frames; j++)

*if* (frames[j] == pages[i])

                    second\_chance[j] = 1;

            hits++;

*printf*("Hit! ");

        }

*displayFrames*(frames, n\_frames);

*printf*("\n");

    }

*printf*("Total Hits: %d, Misses: %d\n", hits, misses);

}

1. Optimal Page Replacement

void *optimal*(int pages*[]*, int n\_pages, int n\_frames) {

    int frames[MAX\_FRAMES], hits = 0, misses = 0;

*for* (int i = 0; i < n\_frames; i++) frames[i] = EMPTY;

*printHeader*("IV. Optimal");

*for* (int i = 0; i < n\_pages; i++) {

*printf*("Page request %d: ", pages[i]);

*if* (!*pageExists*(frames, n\_frames, pages[i])) {

            int replace\_idx = -1;

            int future[MAX\_FRAMES];

*for* (int j = 0; j < n\_frames; j++) future[j] = INT\_MAX;

*for* (int j = 0; j < n\_frames; j++) {

*if* (frames[j] == EMPTY) {

                    replace\_idx = j;

*break*;

                }

*for* (int k = i + 1; k < n\_pages; k++) {

*if* (frames[j] == pages[k]) {

                        future[j] = k;

*break*;

                    }

                }

            }

*if* (replace\_idx == -1) {

                int max\_future = -1;

*for* (int j = 0; j < n\_frames; j++) {

*if* (future[j] > max\_future) {

                        max\_future = future[j];

                        replace\_idx = j;

                    }

                }

            }

            frames[replace\_idx] = pages[i];

            misses++;

*printf*("Miss! ");

        } *else* {

            hits++;

*printf*("Hit! ");

        }

*displayFrames*(frames, n\_frames);

*printf*("\n");

    }

*printf*("Total Hits: %d, Misses: %d\n", hits, misses);

}

1. LFU (Least Frequently Used) Page Replacement

void *lfu*(int pages*[]*, int n\_pages, int n\_frames) {

    int frames[MAX\_FRAMES], frequency[MAX\_FRAMES], hits = 0, misses = 0;

*for* (int i = 0; i < n\_frames; i++) {

        frames[i] = EMPTY;

        frequency[i] = 0;

    }

*printHeader*("V. LFU (Least Frequently Used)");

*for* (int i = 0; i < n\_pages; i++) {

*printf*("Page request %d: ", pages[i]);

*if* (!*pageExists*(frames, n\_frames, pages[i])) {

            int lfu\_idx = 0, min\_freq = INT\_MAX;

*for* (int j = 0; j < n\_frames; j++) {

*if* (frames[j] == EMPTY) {

                    lfu\_idx = j;

*break*;

                }

*if* (frequency[j] < min\_freq) {

                    min\_freq = frequency[j];

                    lfu\_idx = j;

                }

            }

            frames[lfu\_idx] = pages[i];

            frequency[lfu\_idx] = 1;

            misses++;

*printf*("Miss! ");

        } *else* {

*for* (int j = 0; j < n\_frames; j++)

*if* (frames[j] == pages[i])

                    frequency[j]++;

            hits++;

*printf*("Hit! ");

        }

*displayFrames*(frames, n\_frames);

*printf*("\n");

    }

*printf*("Total Hits: %d, Misses: %d\n", hits, misses);

}

1. MFU (Most Frequently Used) Page Replacement

void *mfu*(int pages*[]*, int n\_pages, int n\_frames) {

    int frames[MAX\_FRAMES], frequency[MAX\_FRAMES], hits = 0, misses = 0;

*for* (int i = 0; i < n\_frames; i++) {

        frames[i] = EMPTY;

        frequency[i] = 0;

    }

*printHeader*("VI. MFU (Most Frequently Used)");

*for* (int i = 0; i < n\_pages; i++) {

*printf*("Page request %d: ", pages[i]);

*if* (!*pageExists*(frames, n\_frames, pages[i])) {

            int mfu\_idx = 0, max\_freq = -1;

*for* (int j = 0; j < n\_frames; j++) {

*if* (frames[j] == EMPTY) {

                    mfu\_idx = j;

*break*;

                }

*if* (frequency[j] > max\_freq) {

                    max\_freq = frequency[j];

                    mfu\_idx = j;

                }

            }

            frames[mfu\_idx] = pages[i];

            frequency[mfu\_idx] = 1;

            misses++;

*printf*("Miss! ");

        } *else* {

*for* (int j = 0; j < n\_frames; j++)

*if* (frames[j] == pages[i])

                    frequency[j]++;

            hits++;

*printf*("Hit! ");

        }

*displayFrames*(frames, n\_frames);

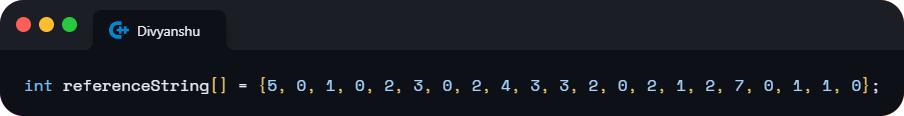
*printf*("\n");

    }

*printf*("Total Hits: %d, Misses: %d\n", hits, misses);

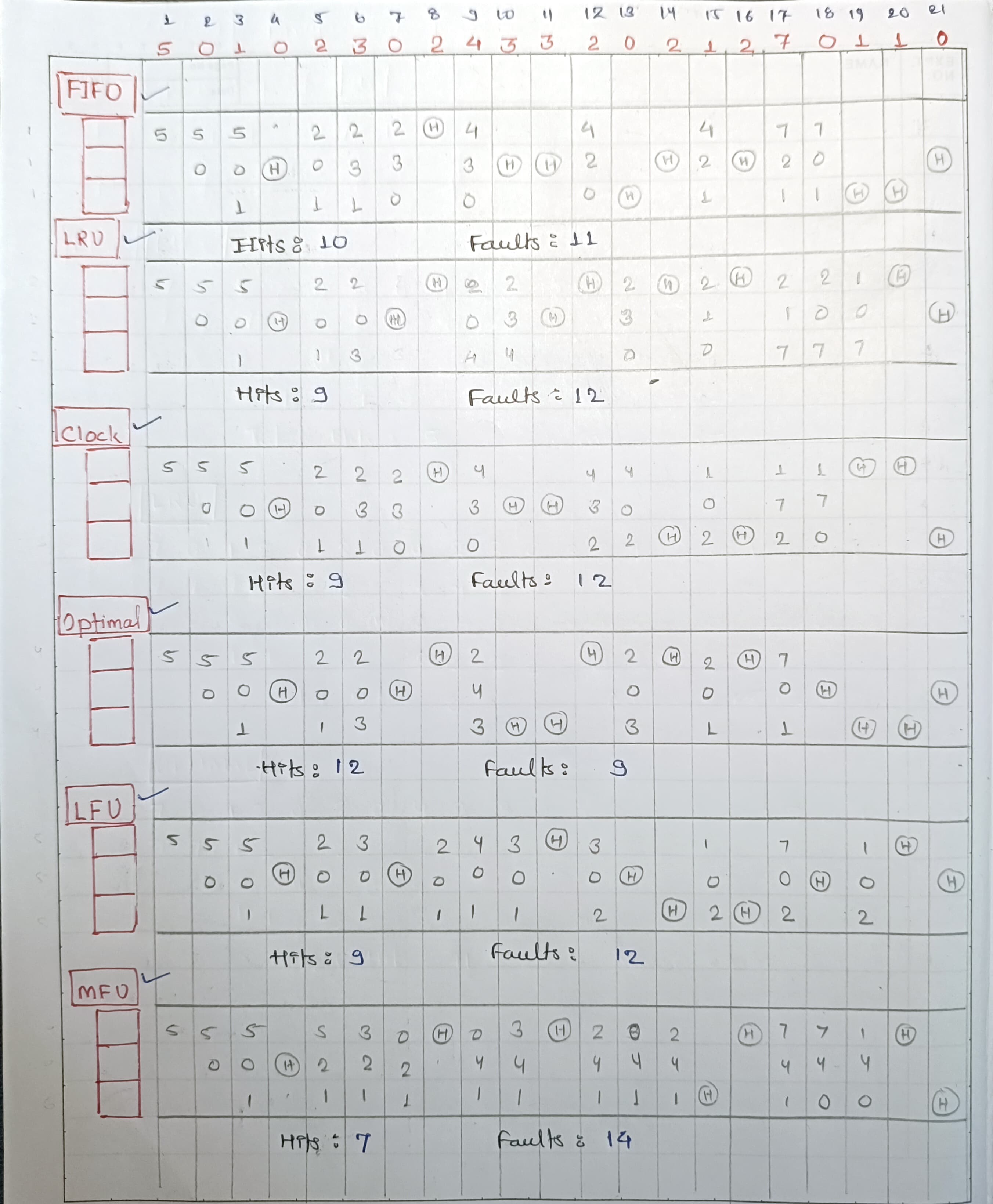
}

1. Comparative Analysis of Page Replacement Algorithms



A screenshot of a computer

Description automatically generated



1.3 Frame Allocation Techniques Simulation

1. Equal Frame Allocation

// *Equal Frame Allocation*

void *equalFrameAllocation*(Process processes*[]*, int processCount, int totalFrames) {

    int framesPerProcess = totalFrames / processCount;

*for* (int i = 0; i < processCount; i++) {

        processes[i].framesAllocated = framesPerProcess;

    }

*printf*("\nEqual Frame Allocation:\n");

*displayAllocation*(processes, processCount);

}

1. Proportional Frame Allocation

// *Proportional Frame Allocation*

void *proportionalFrameAllocation*(Process processes*[]*, int processCount, int totalFrames) {

    int totalRequiredFrames = 0;

*for* (int i = 0; i < processCount; i++) {

        totalRequiredFrames += processes[i].requiredFrames;

    }

*for* (int i = 0; i < processCount; i++) {

        processes[i].framesAllocated = (processes[i].requiredFrames \* totalFrames) / totalRequiredFrames;

    }

*printf*("\nProportional Frame Allocation:\n");

*displayAllocation*(processes, processCount);

}

1. Priority-Based Frame Allocation

// *Priority-Based Frame Allocation*

// *Frames Allocated to Process=(Process Priority / Total Priority) × Total Frames*

void *priorityBasedFrameAllocation*(Process processes*[]*, int processCount, int totalFrames) {

    int totalPriority = 0;

*for* (int i = 0; i < processCount; i++) {

        totalPriority += processes[i].priority;

    }

*for* (int i = 0; i < processCount; i++) {

        processes[i].framesAllocated = (processes[i].priority \* totalFrames) / totalPriority;

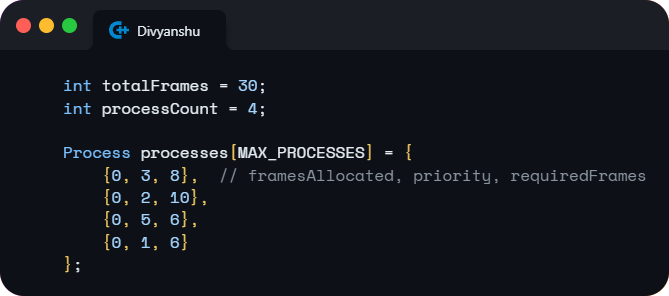
    }

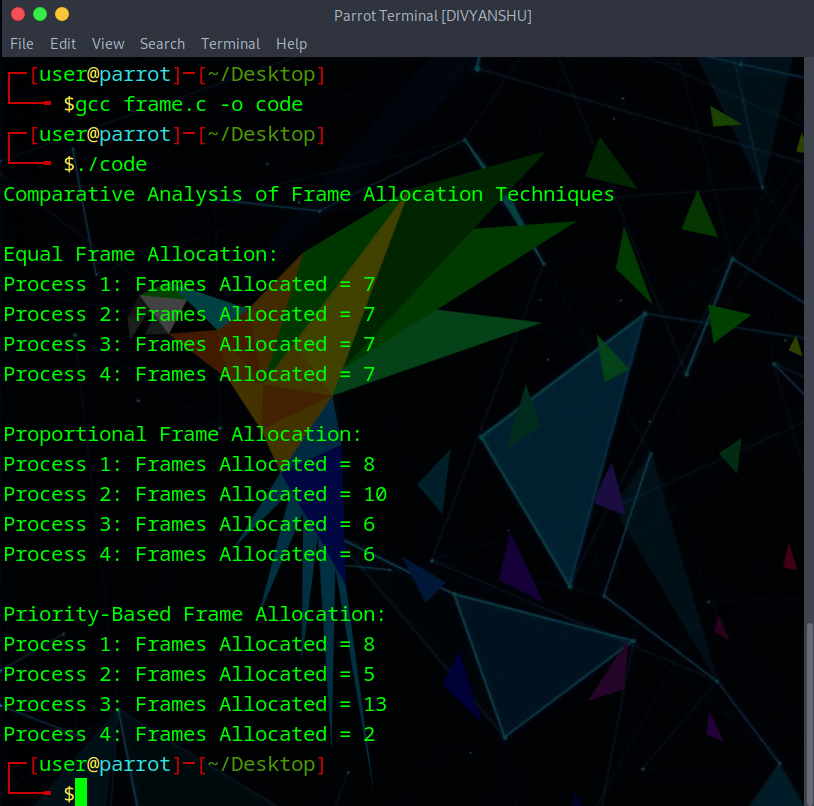
*printf*("\nPriority-Based Frame Allocation:\n");

*displayAllocation*(processes, processCount);

}

1. Comparative Analysis of Frame Allocation Techniques





1. Introduction to Programming Projects

**Reference: OPERATING SYSTEM CONCEPTS ABRAHAM SILBERSCHATZ 10th. Edition**

1. Problem Statement given in **ABRAHAM SILBERSCHATZ Textbook**

**(Chapter 9 Virtual Memory Pg. 458)**

Programming Projects: Designing a Virtual Memory Manager

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size 216 = 65,536 bytes. Your program will read from a file containing logical addresses and, using a TLB as well as a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. The goal behind this project is to simulate the steps involved in translating logical to physical addresses.

Specifics:

Your program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must mask the rightmost 16 bits of each logical address. These 16 bits are divided into (1) an 8-bit page number and (2) 8-bit page offset. Hence, the addresses are structured as shown in Figure 9.33. Other specifics include the following:

• 28 entries in the page table

• Page size of 28 bytes

• 16 entries in the TLB

• Frame size of 28 bytes

• 256 frames

• Physical memory of 65,536 bytes (256 frames × 256-byte frame size)

Additionally, your program need only be concerned with reading logical addresses and translating them to their corresponding physical addresses. You do not need to support writing to the logical address space.

1. Objectives and Deliverables of the Project

This project involves developing a program to convert logical to physical addresses for a virtual memory system with a 65,536-byte address space. Using a TLB and page table, the program will translate logical addresses, handling page faults with demand paging and managing memory with page-replacement algorithms (FIFO or LRU). Key goals include understanding address translation steps and collecting statistics like page-fault and TLB hit rates.

**Objectives:**

* Simulate logical-to-physical address translation using a TLB and page table.
* Resolve page faults with demand paging from BACKING\_STORE.bin.
* Implement FIFO or LRU page-replacement for memory management.
* Report page-fault rate and TLB hit rate.

**Deliverables:**

* **Program**: Translates addresses from addresses.txt and outputs byte values.
* **Output File**: output.txt with physical addresses and byte values.
* **Statistics**: Page-fault and TLB hit rates.

1. Background and Problem Description

This project simulates virtual memory management, translating logical to physical addresses using a page table and a Translation Lookaside Buffer (TLB) for faster access. When pages aren’t in memory, demand paging loads them from secondary storage, handling page faults efficiently. Limited memory prompts the use of FIFO or LRU page-replacement algorithms to decide which pages to replace, ensuring effective memory management.

1. Address Translation

Our program will translate logical to physical addresses using a TLB and page table.

First, the page number is extracted from the logical address, and the TLB is consulted as shown in Figure 1.

A blue rectangular object with black text

Description automatically generated

Figure 1. Address Structure

* In the case of a TLB hit, the frame number is obtained from the TLB.
* In the case of a TLB miss, the page table must be consulted.

In the latter case, either the frame number is obtained from the page table, or a page fault occurs. A visual representation of the address translation process is:

A diagram of a computer

Description automatically generated

Figure 2. A representation of the address-translation process

1. Handling Page Faults

The program uses demand paging to handle page faults by reading pages from BACKING\_STORE.bin, a 65,536-byte binary file. On a page fault (e.g., for page 15), the program reads a 256-byte page from BACKING\_STORE.bin and loads it into an available frame in physical memory. Once loaded, future accesses to this page are handled by the TLB or page table, speeding up access. The program treats BACKING\_STORE.bin as a random-access file and utilizes C library functions like fopen(), fread(), fseek(), and fclose() for I/O operations.

1. Test File

The program reads logical addresses from addresses.txt, each an integer between 0 and 65,535, which represents the virtual address space. For each address, it translates it to a physical address and retrieves the signed byte value stored there.

16916

62493

.....

12107

1. Elementary Requirements

The program runs as ./a.out addresses.txt and reads 1,000 logical addresses from addresses.txt (ranging from 0 to 65,535). For each address, it outputs:

1. The logical address read from addresses.txt.
2. The translated physical address.
3. The signed byte value stored at that physical address in memory.
4. Page Replacement

In this phase, physical memory is limited to 128 page frames (smaller than the virtual address space). The program will now track free frames and use a page-replacement policy (FIFO or LRU) to handle page faults when memory is full.

1. Implementation Details and Methods
2. Basic Data Structure

#*include*<stdio.h>

#*include*<stdlib.h>

#*define* *PAGE\_NUM* 256

#*define* *PAGE\_SIZE* 256

#*define* *FRAME\_NUM* 256// *change frame num*

#*define* *FRAME\_SIZE* 256

#*define* *TLB\_ENTRY\_NUM* 16

#*define* *PAGE\_TABLE\_SIZE* 256

FILE\* addresses;

FILE\* backing\_store;

FILE\* result\_file;

typedef struct PAGE\_TABLE\_ITEM{

    int valid;

    int frame\_id;

} page\_table\_item;

typedef struct TLB\_ITEM{

    int used\_time;

    int frame\_id;

    int page\_id;

} tlb\_item;

typedef struct MEMORY\_ITEM{

    int used\_time;

    char data[*FRAME\_SIZE*];

} memory\_item;

page\_table\_item page\_table[*PAGE\_TABLE\_SIZE*];

tlb\_item TLB[*TLB\_ENTRY\_NUM*];

memory\_item memory[*FRAME\_NUM*];

1. Initialize Variables

/\* *initialize page table* \*/

void *page\_table\_init*(){

*for*(int i=0; i<*PAGE\_TABLE\_SIZE*; i++){

        page\_table[i].valid = 0;// *set to not valid*

        page\_table[i].frame\_id = -1;// *set to None*

    }

}

/\* *initialize TLB* \*/

void *tlb\_init*(){

*for*(int i=0; i<*TLB\_ENTRY\_NUM*; i++){

        TLB[i].used\_time = -1;

        TLB[i].frame\_id = -1;

        TLB[i].page\_id = -1;

    }

}

/\* *initialize memory*\*/

void *memory\_init*(){

*for*(int i=0; i<*FRAME\_NUM*; i++){

        memory[i].used\_time = -1;

    }

}

/\* *initialize page table, TLB, memory* \*/

void *initialize*(){

*page\_table\_init*();

*tlb\_init*();

*memory\_init*();

*printf*("Initialize Finish.\n");

}

1. Get and Parse Addresses

/\* *get the page num, given address*\*/

int *get\_page*(int address){

*return* address >> 8;

}

/\* *get the offset, given address* \*/

int *get\_offset*(int address){

*return* address & 0xFF;

}

1. LRU Replacement in TLB

/\* *LRU replacement for TLB* \*/

void *TLB\_LRU\_Replacement*(int page\_id, int frame\_id, int time){

    int min\_time = time;

    int min\_idx = 0;

*for*(int i = 0; i < *TLB\_ENTRY\_NUM*; i++){

*if*(TLB[i].used\_time < min\_time){

            min\_time = TLB[i].used\_time;

            min\_idx = i;

        }

    }

    TLB[min\_idx].frame\_id = frame\_id;

    TLB[min\_idx].page\_id = page\_id;

    TLB[min\_idx].used\_time = time;

}

1. LRU Replacement in Memory

/\* *LRU Replacement for Memory*

*Updating new data*

*returning the frame id* \*/

int *memory\_LRU\_Replacement*(int page\_id, int time){

    int min\_time = time;

    int min\_idx = 0;

*for*(int i = 0; i < *FRAME\_NUM*; i++){

*if*(memory[i].used\_time < min\_time){

            min\_time = memory[i].used\_time;

            min\_idx = i;

        }

    }

    memory[min\_idx].used\_time = time;

    // *Find the old page id, and set invalid*

*for*(int i = 0; i < *PAGE\_TABLE\_SIZE*; i++){

*if*(page\_table[i].frame\_id == min\_idx){

            page\_table[i].valid = -1;

        }

    }

    // *Load data from backing store*

*fseek*(backing\_store, page\_id \* *PAGE\_SIZE*, *SEEK\_SET*);

*fread*(memory[min\_idx].data, sizeof(char), *FRAME\_SIZE*, backing\_store);

*return* min\_idx;

}

1. Main Part of code

int *main*(int argc, char\*argv*[]*){

    addresses = *fopen*(argv[1], "r");

    backing\_store = *fopen*("BACKING\_STORE.bin", "rb");

    result\_file = *fopen*("result.txt", "w");

    // *Initialize*

*initialize*();

    int address;

    int page\_id;

    int frame\_id;

    int offset;

    // *Counters*

    int count = 0;

    int tlb\_hit = 0;

    int page\_fault = 0;

    int tlb\_miss = 0;

    int page\_hit = 0;

    int time = 0;

*fscanf*(addresses, "%u", &address);

*while*(!*feof*(addresses)){

        count++;

        time++;

        // *Decode address*

        page\_id = *get\_page*(address);

        offset = *get\_offset*(address);

        // *Search in TLB*

        int tlb\_find = 0;

*for*(int i = 0; i < *TLB\_ENTRY\_NUM*; i++){

*if*(page\_id == TLB[i].page\_id){

                tlb\_hit++;

                tlb\_find = 1;

                frame\_id = TLB[i].frame\_id;

                memory[frame\_id].used\_time = time;

                TLB[i].used\_time = time;

*break*;

            }

        }

        // *Not found in TLB, search in page table*

        int page\_find = 0;

*if*(!tlb\_find){

            tlb\_miss++;

*if*(page\_table[page\_id].valid == 1){

                page\_find = 1;

                page\_hit++;

                frame\_id = page\_table[page\_id].frame\_id;

                memory[frame\_id].used\_time = time;

*TLB\_LRU\_Replacement*(page\_id, frame\_id, time);// *Update TLB*

            }

*else* {

                page\_fault++;

                frame\_id = *memory\_LRU\_Replacement*(page\_id, time);

                // *Update page table*

                page\_table[page\_id].frame\_id = frame\_id;

                page\_table[page\_id].valid = 1;

                // *Update TLB*

*TLB\_LRU\_Replacement*(page\_id, frame\_id, time);

            }

        }

        // *Calculate physical address and get data*

        int physical\_address = frame\_id \* *FRAME\_SIZE* + offset;

        int data = memory[frame\_id].data[offset];

        // *Output result*

*fprintf*(result\_file, "Virtual address: %d Physical address: %d Value: %d\n", address, physical\_address, data);

*fscanf*(addresses, "%u", &address);

    }

    // *Finish*

*fclose*(addresses);

*fclose*(backing\_store);

*fclose*(result\_file);

    // *Statistics*

*printf*("Execution Finish.\n");

*printf*("-----------------------------\n");

    double page\_fault\_rate = page\_fault / (double)count;

    double page\_hit\_rate = page\_hit / (double)count;

    double tlb\_hit\_rate = tlb\_hit / (double)count;

    double tlb\_miss\_rate = tlb\_miss / (double)count;

*printf*("Frame Num: %d\n", *FRAME\_NUM*);

*printf*("TLB Hits: %d\n", tlb\_hit);

*printf*("TLB Misses: %d\n", tlb\_miss);

*printf*("TLB Hit Rate: %f\n", tlb\_hit\_rate);

*printf*("TLB Miss Rate: %f\n", tlb\_miss\_rate);

*printf*("Page Hits: %d\n", page\_hit);

*printf*("Page Faults: %d\n", page\_fault);

*printf*("Page Hit Rate: %f\n", page\_hit\_rate);

*printf*("Page Fault Rate: %f\n", page\_fault\_rate);

*printf*("Allocation Technique: Demand Paging\n");

*printf*("Replacement Technique: LRU (Least Recently Used)\n");

*return* 0;

}

1. Program Results
2. Compare result.txt with correct.txt

When frame\_num = 256, we get the following result:

A black screen with white text

Description automatically generated

A screen shot of a computer screen

Description automatically generated

The output file is result.txt .

So whether result.txt is the same as correct.txt ?

We could use bash to compare them:

A screen shot of a computer

Description automatically generated

Enter:

A screen shot of a computer

Description automatically generated

Then we get:

A screenshot of a computer

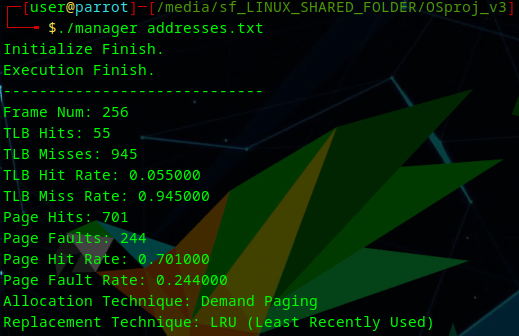
Description automatically generated

Thus, we succeed in getting the correct results!

1. Statistics

We change the frame num, to see the TLB hit rate and Page Fault Rate.

When frame\_num=256:



When frame\_num=128:

A screenshot of a computer

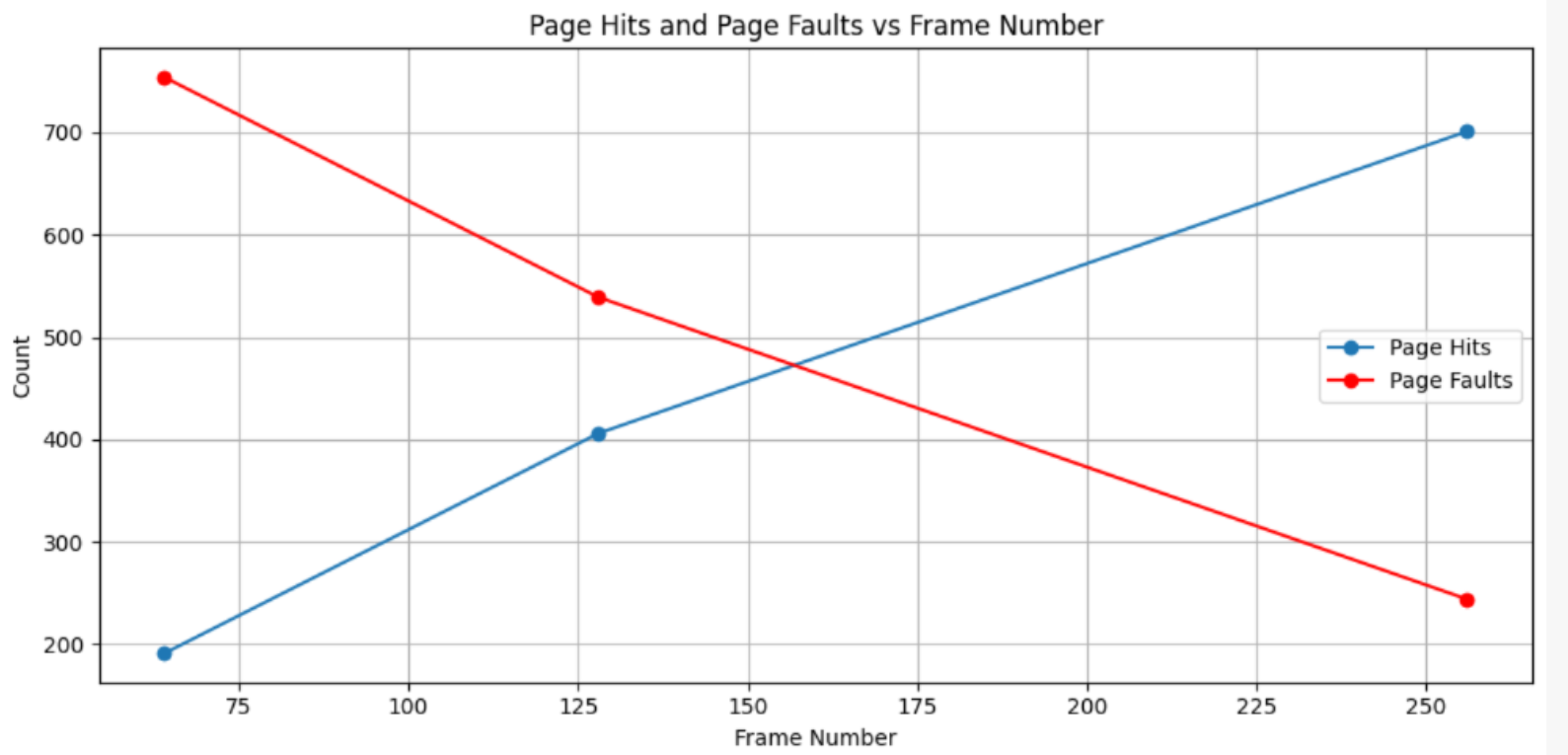
Description automatically generated

When frame\_num=64:

A screen shot of a computer

Description automatically generated

These results provide interesting insights into the behavior of our virtual memory system:



1. TLB Hit Rate: The TLB hit rate remains constant regardless of the number of frames. This is because the TLB's effectiveness depends more on the locality of reference in the address stream than on the size of physical memory.

2. Page Fault Rate: As the number of frames decreases, the page fault rate increases significantly. This demonstrates the impact of physical memory size on system performance. With fewer frames, pages need to be swapped in and out more frequently, leading to more page faults.

3. Trade-off: These results illustrate the trade-off between memory usage and system performance. While using fewer frames saves memory, it leads to more page faults, which can significantly slow down the system due to increased disk I/O.

**In conclusion, when the frame number gets smaller, the page fault rate goes higher, and**

**the TLB hit rate keeps low.**

4. **Conclusion and Thoughts**

This project on implementing a virtual memory manager serves as an excellent practical exercise in understanding and applying core operating system concepts related to memory management. It effectively bridges the gap between theoretical knowledge and practical implementation, providing valuable insights into the complexities and trade-offs involved in designing efficient memory management systems.

REFRENCES:

Problem Statement given in ABRAHAM SILBERSCHATZ Textbook

(Chapter 9 Virtual Memory Pg. 458)

Notes to Students (pg. xiii)

<http://www.wiley.com/college>

Wiley website (View student companion site)

<https://www.wiley.com/en-us/Operating+System+Concepts%2C+10th+Edition-p-9781119320913#related-resources-section>

Github (for supportive files mention in book)

<https://github.com/greggagne/osc10e/tree/master/ch10>