



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

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

1.1 Memory Allocation Techniques Simulation (Contiguous)

- I. First-Fit Memory Allocation
- II. Next-Fit Memory Allocation
- III. Best-Fit Memory Allocation
- IV. Worst-Fit Memory Allocation
- V. Comparative Analysis of Memory Allocation Techniques

1.2 Page Replacement Algorithm Simulation

- I. FIFO (First-In-First-Out) Page Replacement
- II. LRU (Least Recently Used) Page Replacement
- III. Clock (Second Chance) Page Replacement
- IV. Optimal Page Replacement
- V. LFU (Least Frequently Used) Page Replacement
- VI. MFU (Most Frequently Used) Page Replacement
- VII. Comparative Analysis of Page Replacement Algorithms

1.3 Frame Allocation Techniques Simulation

- I. Equal Frame Allocation
- II. Proportional Frame Allocation
- III. Priority-Based Frame Allocation
- IV. Comparative Analysis of Frame Allocation Techniques

2. Introduction to Programming Projects

Reference: OPERATING SYSTEM CONCEPTS ABRAHAM SILBERSCHATZ 10th. Edition

(Standard Textbook as mentioned in syllabus)

- I. Problem Statement given in **ABRAHAM SILBERSCHATZ Textbook**
- II. Objectives and Deliverables of the Project
- III. Background and Problem Description
- IV. Address Translation
- V. Handling Page Faults
- VI. Test File
- VII. Elementary Requirements
- VIII. Page Replacement
- IX. Statistics

3. Implementation Details and Methods

- I. Basic Data Structure
- II. Initialize Variables
- III. Get and Parse Addresses
- IV. LRU Replacement in TLB
- V. LRU Replacement in Memory
- VI. Main Part of code

4. Program Results

- I. Compare result.txt with correct.txt
- II. Statistics

1. 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

- I. First-Fit Memory Allocation
- II. Next-Fit Memory Allocation
- III. Best-Fit Memory Allocation
- IV. Worst-Fit Memory Allocation
- V. 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;
}

```

I. Comparative Analysis of Memory Allocation Techniques

```

[user@parrot] ~/Desktop
$ ./code
Fixed Size Partitioning:

First Fit (Fixed) Allocation:
Process 1 (300): Allocated to Partition 2 (350)
Process 2 (25): Allocated to Partition 1 (150)
Process 3 (125): Not Allocated
Process 4 (50): Not Allocated

Next Fit (Fixed) Allocation:
Process 1 (300): Allocated to Partition 2 (350)
Process 2 (25): Allocated to Partition 1 (150)
Process 3 (125): Not Allocated
Process 4 (50): Not Allocated

Best Fit (Fixed) Allocation:
Process 1 (300): Allocated to Partition 2 (350)
Process 2 (25): Allocated to Partition 1 (150)
Process 3 (125): Not Allocated
Process 4 (50): Not Allocated

Worst Fit (Fixed) Allocation:
Process 1 (300): Allocated to Partition 2 (350)
Process 2 (25): Allocated to Partition 1 (150)
Process 3 (125): Not Allocated
Process 4 (50): Not Allocated

Variable Size Partitioning:

First Fit (Variable) Allocation:
Process 1 (300): Allocated to Partition 2 (350)
Process 2 (25): Allocated to Partition 1 (150)
Process 3 (125): Allocated to Partition 1 (150)
Process 4 (50): Allocated to Partition 2 (350)

Next Fit (Variable) Allocation:
Process 1 (300): Allocated to Partition 2 (350)
Process 2 (25): Allocated to Partition 1 (150)
Process 3 (125): Allocated to Partition 1 (150)
Process 4 (50): Allocated to Partition 2 (350)

Best Fit (Variable) Allocation:
Process 1 (300): Allocated to Partition 2 (350)
Process 2 (25): Allocated to Partition 2 (350)
Process 3 (125): Allocated to Partition 1 (150)
Process 4 (50): Not Allocated

Worst Fit (Variable) Allocation:
Process 1 (300): Allocated to Partition 2 (350)
Process 2 (25): Allocated to Partition 1 (150)
Process 3 (125): Allocated to Partition 1 (150)
Process 4 (50): Allocated to Partition 2 (350)

```

```

[user@parrot] ~/Desktop
$ ./code

Buddy System Allocation:
Process 1 (212): Allocated to Partition 2
Process 2 (417): Allocated to Partition 5
Process 3 (112): Allocated to Partition 1
Process 4 (426): Not Allocated

```

1.2 Page Replacement Algorithm Simulation

I. 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);
}
```

II. 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);
}

```

III. 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);
}

```

IV. 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);
}
```

V. 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);
}

```

VI. 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;

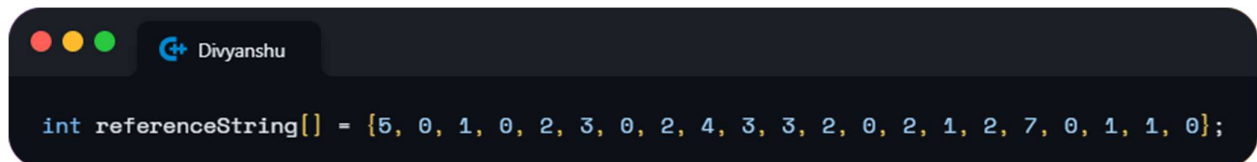
```

```

    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);
}

```

VII. Comparative Analysis of Page Replacement Algorithms



```

int referenceString[] = {5, 0, 1, 0, 2, 3, 0, 2, 4, 3, 3, 2, 0, 2, 1, 2, 7, 0, 1, 1, 0};

```



```
Parrot Terminal [DIVYANSHU]
File Edit View Search Terminal Help
[user@parrot]-[~/Desktop]
$gcc replacement.c -o code
[user@parrot]-[~/Desktop]
$./code
Enter the number of frames: 3
Algorithm Comparison:
-----
| Algorithm | Hits | Faults | Hit Rate | Fault Rate |
-----
| FIFO | 11 | 10 | 0.52 | 0.48 |
| LRU | 9 | 12 | 0.43 | 0.57 |
| Optimal | 12 | 9 | 0.57 | 0.43 |
| Second Chance | 9 | 12 | 0.43 | 0.57 |
| LFU | 9 | 12 | 0.43 | 0.57 |
| MFU | 9 | 12 | 0.43 | 0.57 |
-----
[user@parrot]-[~/Desktop]
$
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
 5 0 1 0 2 3 0 2 4 3 3 2 0 2 1 2 7 0 1 1 0

FIFO ✓



5	5	5		2	2	2	(H)	4		4		4		7	7					
	0	0	(H)	0	3	3		3	(H)	(H)	2	(H)	2	(H)	2	0			(H)	
		1		1	1	0		0		0	(H)	1		1	1	(H)	(H)			

LRU ✓



Hits: 10 Faults: 11

5	5	5		2	2		(H)	2		(H)	2	(H)	2	(H)	2	2	1	(H)		
	0	0	(H)	0	0	(H)		0	3	(H)	3		1		1	0	0		(H)	
		1		1	3			4	4		0		0		7	7	7			

Clock ✓



Hits: 9 Faults: 12

5	5	5		2	2	2	(H)	4		4	4		1		1	1	(H)	(H)		
	0	0	(H)	0	3	3		3	(H)	(H)	3	0	0		7	7				
		1		1	1	0		0		2	2	(H)	2	(H)	2	0			(H)	

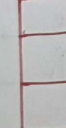
Optimal ✓



Hits: 12 Faults: 9

5	5	5		2	2		(H)	2		(H)	2	(H)	2	(H)	7					
	0	0	(H)	0	0	(H)		4		0	0	0	0	0	(H)			(H)		
		1		1	3			3	(H)	(H)	3		1		1		(H)	(H)		

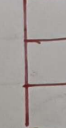
LFU ✓



Hits: 9 Faults: 12

5	5	5		2	3		2	4	3	(H)	3		1		7		1	(H)		
	0	0	(H)	0	0	(H)	0	0	0		0	(H)	0		0	(H)	0		(H)	
		1		1	1		1	1	1		2	(H)	2	(H)	2		2			

MFU ✓



Hits: 7 Faults: 14

5	5	5		5	3	0	(H)	0	3	(H)	2	0	2		(H)	7	7	1	(H)	
	0	0	(H)	2	2	2		4	4		4	4	4			4	4	4		
		1		1	1	1		1	1		1	1	1	(H)		1	0	0		(H)

1.3 Frame Allocation Techniques Simulation

I. 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);
}
```

II. 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);
}
```

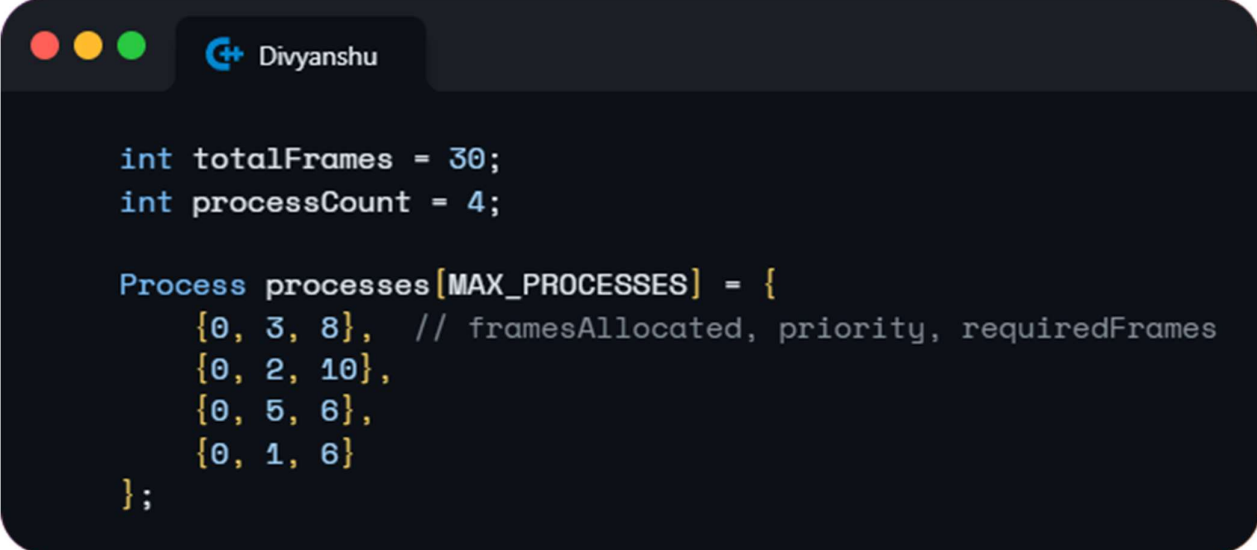
III. Priority-Based Frame Allocation

```
// 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);
}
```

IV. Comparative Analysis of Frame Allocation Techniques



```
int totalFrames = 30;
int processCount = 4;

Process processes[MAX_PROCESSES] = {
    {0, 3, 8}, // framesAllocated, priority, requiredFrames
    {0, 2, 10},
    {0, 5, 6},
    {0, 1, 6}
};
```

```
Parrot Terminal [DIVYANSHU]
File Edit View Search Terminal Help
[user@parrot]-[~/Desktop]
$gcc frame.c -o code
[user@parrot]-[~/Desktop]
$./code
Comparative Analysis of Frame Allocation Techniques

Equal Frame Allocation:
Process 1: Frames Allocated = 7
Process 2: Frames Allocated = 7
Process 3: Frames Allocated = 7
Process 4: Frames Allocated = 7

Proportional Frame Allocation:
Process 1: Frames Allocated = 8
Process 2: Frames Allocated = 10
Process 3: Frames Allocated = 6
Process 4: Frames Allocated = 6

Priority-Based Frame Allocation:
Process 1: Frames Allocated = 8
Process 2: Frames Allocated = 5
Process 3: Frames Allocated = 13
Process 4: Frames Allocated = 2
[user@parrot]-[~/Desktop]
$
```


2. Introduction to Programming Projects

Reference: OPERATING SYSTEM CONCEPTS ABRAHAM SILBERSCHATZ 10th. Edition

- I. 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 $2^{16} = 65,536$ 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 \times 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.

- II. 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.

III. 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.

IV. 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.

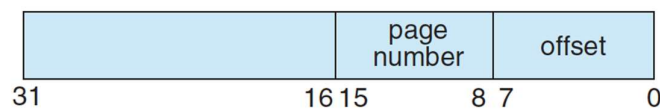


Figure 1. Address Structure

- 🚦 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:

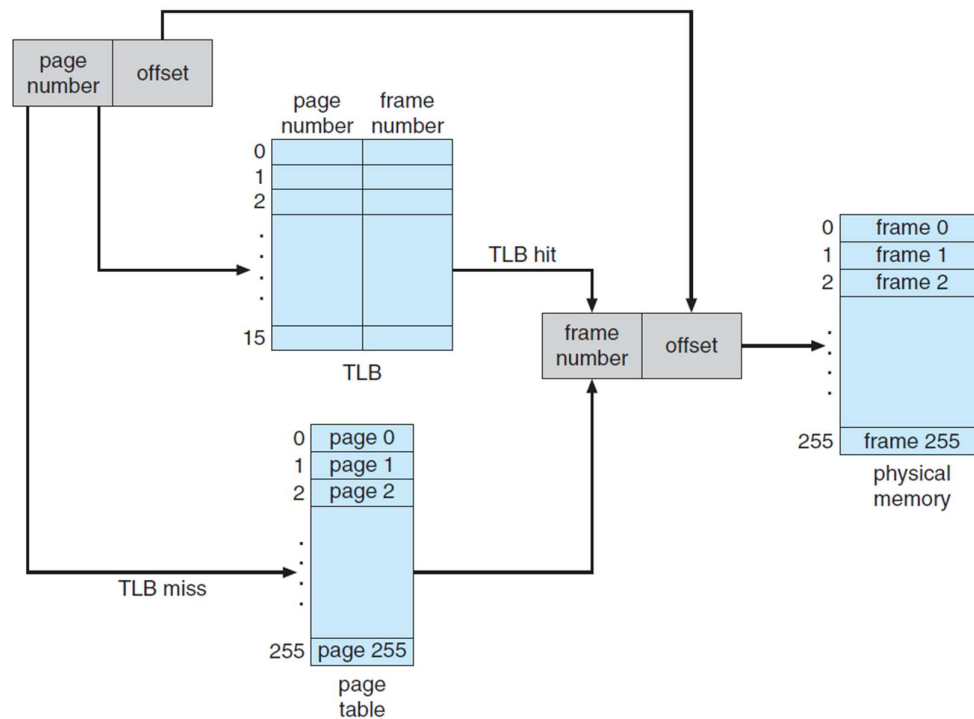


Figure 2. A representation of the address-translation process

V. 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.

VI. 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
```


VII. 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.

VIII. 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.

3. Implementation Details and Methods

I. 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];

```

II. 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");
}

```

III. 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;
}

```

IV. 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;
}
```

V. 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;
}
```

VI. 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;
}

```

4. Program Results

I. Compare result.txt with correct.txt

When frame_num = 256, we get the following result:



Command : To Complile Code

```
# complie
```

```
> make
```

```
# clean
```

```
> make clean
```

```
# output result
```

```
> ./manager addresses.txt
```

Divyanshu

```
ParrotTerminal [DIVYANSHU]
File Edit View Search Terminal Help

[user@parrot]-[/media/sf_LINUX_SHARED_FOLDER/OSproj_v3]
→ $make clean
rm manager
[user@parrot]-[/media/sf_LINUX_SHARED_FOLDER/OSproj_v3]
→ $make
gcc manager.c -o manager
[user@parrot]-[/media/sf_LINUX_SHARED_FOLDER/OSproj_v3]
→ $./manager addresses.txt
Initialize Finish.
Execution Finish.
-----
Frame Num = 256:
TLB Hit Rate: 0.055000
Page Fault Rate: 0.244000
[user@parrot]-[/media/sf_LINUX_SHARED_FOLDER/OSproj_v3]
→ $make clean
rm manager
[user@parrot]-[/media/sf_LINUX_SHARED_FOLDER/OSproj_v3]
→ $make
gcc manager.c -o manager
[user@parrot]-[/media/sf_LINUX_SHARED_FOLDER/OSproj_v3]
→ $./manager addresses.txt
Initialize Finish.
Execution Finish.
-----
Frame Num: 256
TLB Hits: 55
TLB Misses: 945
TLB Hit Rate: 0.055000
TLB Miss Rate: 0.945000
Page Hits: 701
Page Faults: 244
Page Hit Rate: 0.701000
Page Fault Rate: 0.244000
Allocation Technique: Demand Paging
Replacement Technique: LRU (Least Recently Used)
[user@parrot]-[/media/sf_LINUX_SHARED_FOLDER/OSproj_v3]
→ $
```

The output file is result.txt .

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

We could use bash to compare them:

```
compare.sh

if cmp -s "correct.txt" "result.txt"
then
    echo "The files match"
else
    echo "The files are different"
fi
```

Divyanshu

Enter:

```
Command : To Compare Result

# Normalize the line endings in both
> unix2dos correct.txt result.txt

# compare results
> sh compare.sh
```

Divyanshu

Then we get:



```
[user@parrot]~/media/sf_LINUX_SHARED_FOLDER/OSproj_v3
$ sh compare.sh
The files match
[user@parrot]~/media/sf_LINUX_SHARED_FOLDER/OSproj_v3
$
```

Thus, we succeed in getting the correct results!

II. Statistics

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

When frame_num=256:

```
[user@parrot]~/media/sf_LINUX_SHARED_FOLDER/OSproj_v3
$ ./manager addresses.txt
Initialize Finish.
Execution Finish.
-----
Frame Num: 256
TLB Hits: 55
TLB Misses: 945
TLB Hit Rate: 0.055000
TLB Miss Rate: 0.945000
Page Hits: 701
Page Faults: 244
Page Hit Rate: 0.701000
Page Fault Rate: 0.244000
Allocation Technique: Demand Paging
Replacement Technique: LRU (Least Recently Used)
```

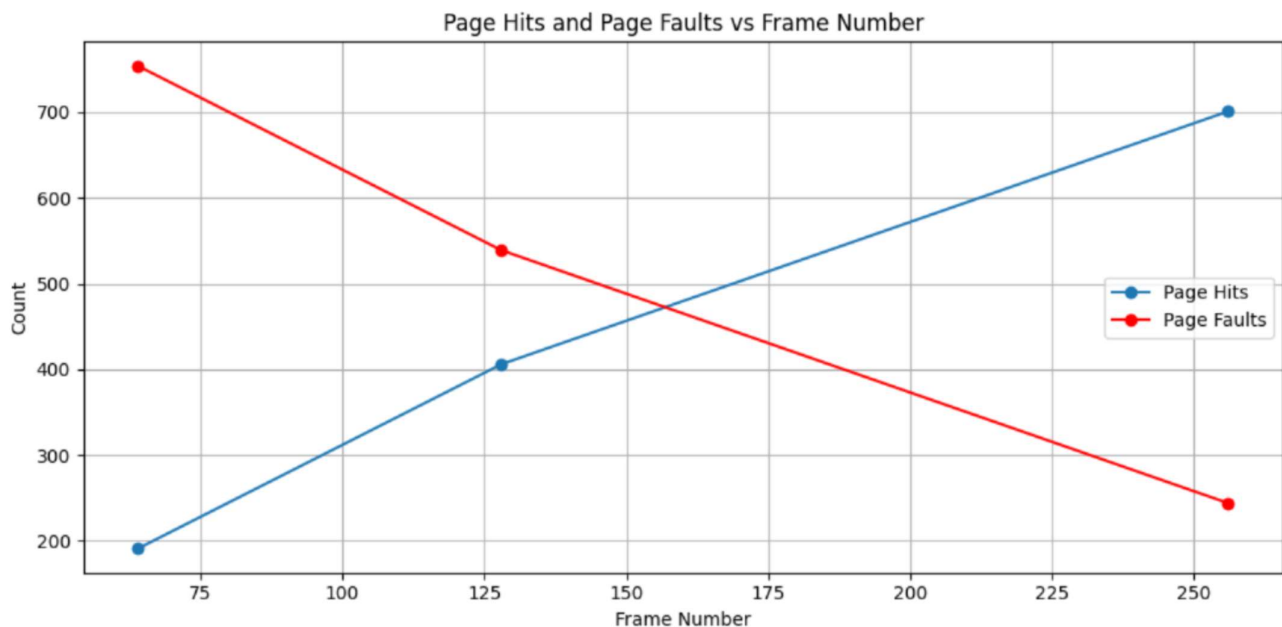
When frame_num=128:

```
[user@parrot]~/media/sf_LINUX_SHARED_FOLDER/OSproj_v3
$ ./manager addresses.txt
Initialize Finish.
Execution Finish.
-----
Frame Num: 128
TLB Hits: 55
TLB Misses: 945
TLB Hit Rate: 0.055000
TLB Miss Rate: 0.945000
Page Hits: 406
Page Faults: 539
Page Hit Rate: 0.406000
Page Fault Rate: 0.539000
Allocation Technique: Demand Paging
Replacement Technique: LRU (Least Recently Used)
```

When frame_num=64:

```
[user@parrot]-[/media/sf_LINUX_SHARED_FOLDER/OSproj_v3]
$ ./manager addresses.txt
Initialize Finish.
Execution Finish.
-----
Frame Num: 64
TLB Hits: 55
TLB Misses: 945
TLB Hit Rate: 0.055000
TLB Miss Rate: 0.945000
Page Hits: 191
Page Faults: 754
Page Hit Rate: 0.191000
Page Fault Rate: 0.754000
Allocation Technique: Demand Paging
Replacement Technique: LRU (Least Recently Used)
```

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.