Name: Devraj Pravin Lokhande Roll No.- 64

PRN: 12310051 Operating Systems SY-AIML Div: A

**LAB - 6**

1. Placement Strategies

#include <iostream>

#include <vector>

#include <iomanip>

using namespace std;

void printBlockMatrix(const vector<int>& originalBlock, const vector<int>& remainingBlock, const string& strategy) {

    cout << "\n=== " << strategy << " Block Allocation Matrix ===\n";

    cout << left << setw(15) << "Block No" << setw(20) << "Original Size" << "Remaining Size\n";

    for (int i = 0; i < originalBlock.size(); i++) {

        cout << left << setw(15) << i + 1 << setw(20) << originalBlock[i] << remainingBlock[i] << endl;

    }

}

void printAllocationMatrix(const vector<int>& allocation, const vector<int>& processSize) {

    cout << left << "\nProcess Allocation:\n";

    cout << left << setw(15) << "Process No" << setw(20) << "Process Size" << "Allocated Block\n";

    for (int i = 0; i < allocation.size(); i++) {

        cout << left << setw(15) << i + 1 << setw(20) << processSize[i];

        if (allocation[i] != -1)

            cout << allocation[i] + 1;

        else

            cout << "Not Allocated";

        cout << endl;

    }

}

void firstFit(const vector<int>& originalBlockSize, const vector<int>& processSize) {

    vector<int> blockSize = originalBlockSize;

    vector<int> allocation(processSize.size(), -1);

    for (int i = 0; i < processSize.size(); i++) {

        for (int j = 0; j < blockSize.size(); j++) {

            if (blockSize[j] >= processSize[i]) {

                allocation[i] = j;

                blockSize[j] -= processSize[i];

                break;

            }

        }

    }

    cout << "\n======== First Fit Strategy ========";

    printAllocationMatrix(allocation, processSize);

    printBlockMatrix(originalBlockSize, blockSize, "First Fit");

}

void nextFit(const vector<int>& originalBlockSize, const vector<int>& processSize) {

    vector<int> blockSize = originalBlockSize;

    vector<int> allocation(processSize.size(), -1);

    int j = 0;

    for (int i = 0; i < processSize.size(); i++) {

        int count = 0;

        while (count < blockSize.size()) {

            if (blockSize[j] >= processSize[i]) {

                allocation[i] = j;

                blockSize[j] -= processSize[i];

                break;

            }

            j = (j + 1) % blockSize.size();

            count++;

        }

    }

    cout << "\n======== Next Fit Strategy ========";

    printAllocationMatrix(allocation, processSize);

    printBlockMatrix(originalBlockSize, blockSize, "Next Fit");

}

void bestFit(const vector<int>& originalBlockSize, const vector<int>& processSize) {

    vector<int> blockSize = originalBlockSize;

    vector<int> allocation(processSize.size(), -1);

    for (int i = 0; i < processSize.size(); i++) {

        int bestIdx = -1;

        for (int j = 0; j < blockSize.size(); j++) {

            if (blockSize[j] >= processSize[i]) {

                if (bestIdx == -1 || blockSize[j] < blockSize[bestIdx])

                    bestIdx = j;

            }

        }

        if (bestIdx != -1) {

            allocation[i] = bestIdx;

            blockSize[bestIdx] -= processSize[i];

        }

    }

    cout << "\n======== Best Fit Strategy ========";

    printAllocationMatrix(allocation, processSize);

    printBlockMatrix(originalBlockSize, blockSize, "Best Fit");

}

void worstFit(const vector<int>& originalBlockSize, const vector<int>& processSize) {

    vector<int> blockSize = originalBlockSize;

    vector<int> allocation(processSize.size(), -1);

    for (int i = 0; i < processSize.size(); i++) {

        int worstIdx = -1;

        for (int j = 0; j < blockSize.size(); j++) {

            if (blockSize[j] >= processSize[i]) {

                if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx])

                    worstIdx = j;

            }

        }

        if (worstIdx != -1) {

            allocation[i] = worstIdx;

            blockSize[worstIdx] -= processSize[i];

        }

    }

    cout << "\n======== Worst Fit Strategy ========";

    printAllocationMatrix(allocation, processSize);

    printBlockMatrix(originalBlockSize, blockSize, "Worst Fit");

}

int main() {

    vector<int> blockSize = {100, 500, 200, 300, 600};

    vector<int> processSize = {212, 417, 112, 426};

    firstFit(blockSize, processSize);

    nextFit(blockSize, processSize);

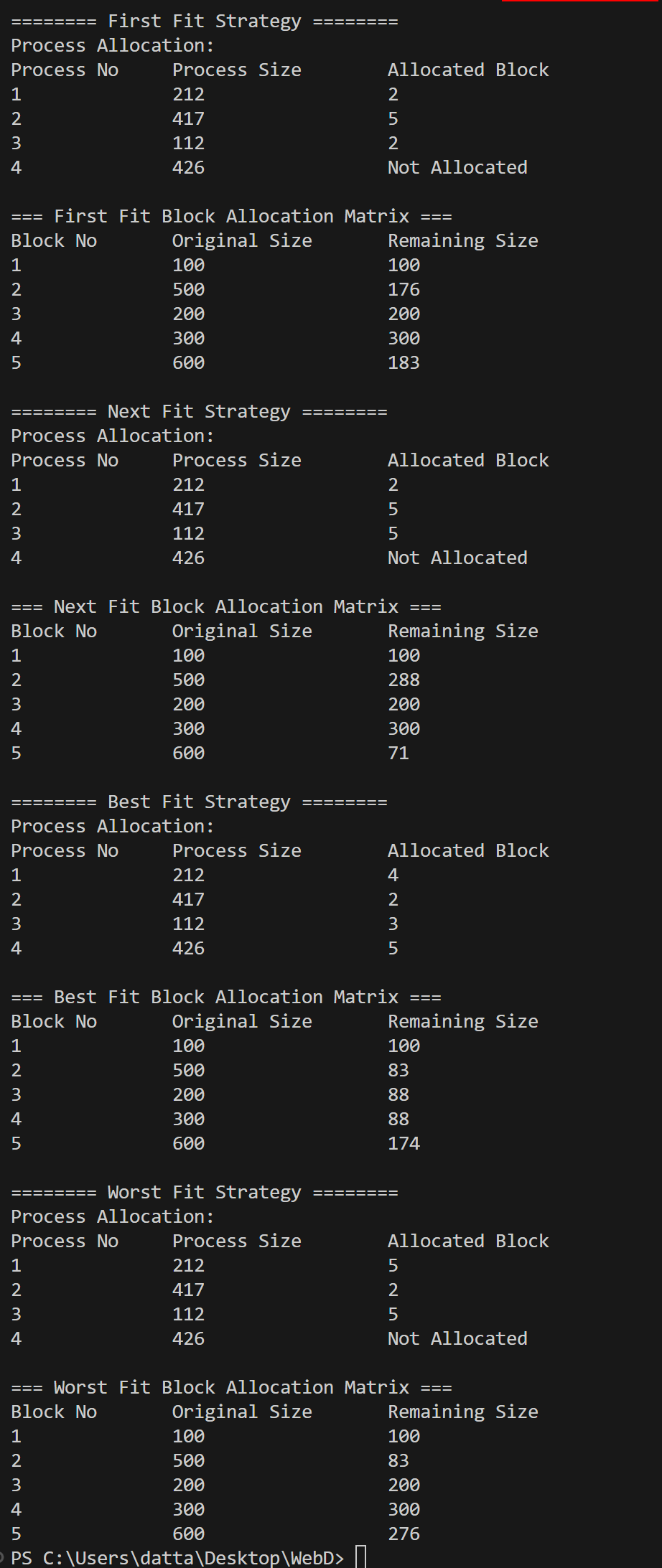
    bestFit(blockSize, processSize);

    worstFit(blockSize, processSize);

    return 0;

}

**OUTPUT:**



1. Buddy System

import java.util.concurrent.Semaphore;

#include <iostream>

#include <cmath>

#include <map>

#include <vector>

#include <iomanip>

using namespace std;

int nextPowerOf2(int n) {

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

}

class BuddySystem {

private:

int totalSize;

map<int, vector<int>> freeList;

public:

BuddySystem(int size) {

totalSize = nextPowerOf2(size);

freeList[totalSize].push\_back(0);

}

void allocate(int processID, int size, map<int, pair<int, int>>& allocation) {

int blockSize = nextPowerOf2(size);

bool allocated = false;

for (int s = blockSize; s <= totalSize; s \*= 2) {

if (!freeList[s].empty()) {

int startAddr = freeList[s].back();

freeList[s].pop\_back();

while (s > blockSize) {

s /= 2;

freeList[s].push\_back(startAddr + s);

}

allocation[processID] = {startAddr, blockSize};

allocated = true;

break;

}

}

if (!allocated) {

cout << "Process " << processID << " of size " << size << " couldn't be allocated\n";

}

}

void printAllocationMatrix(map<int, pair<int, int>>& allocation) {

cout << "\n=== Buddy System Allocation Matrix ===\n";

cout << left << setw(15) << "Process ID" << setw(15) << "Start Addr" << "Block Size\n";

for (auto& entry : allocation) {

cout << left << setw(15) << entry.first << setw(15) << entry.second.first << entry.second.second << "\n";

}

}

void printFreeListMatrix() {

cout << "\n=== Free List Matrix ===\n";

cout << left << setw(15) << "Block Size" << "Free Start Addresses\n";

for (auto& block : freeList) {

cout << left << setw(15) << block.first;

for (auto addr : block.second)

cout << addr << " ";

cout << "\n";

}

}

};

int main() {

int memorySize = 1024;

BuddySystem buddy(memorySize);

vector<int> processes = {100, 50, 200, 300, 400, 500};

map<int, pair<int, int>> allocation;

for (int i = 0; i < processes.size(); i++) {

buddy.allocate(i + 1, processes[i], allocation);

}

buddy.printAllocationMatrix(allocation);

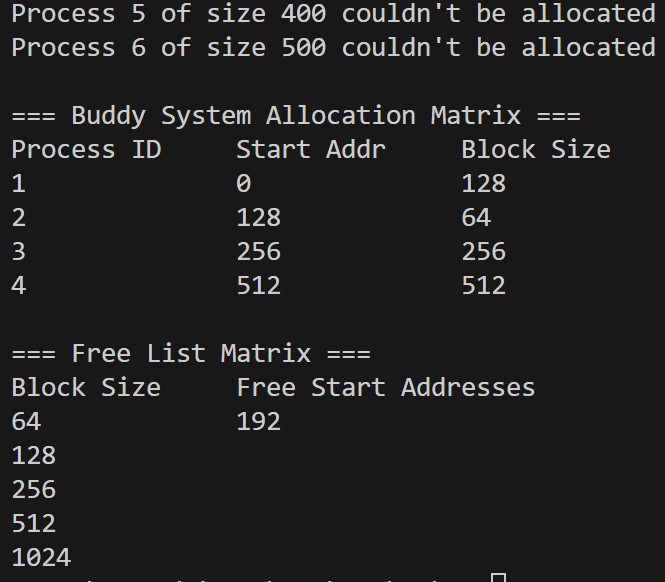
buddy.printFreeListMatrix();

return 0;

}

}

**OUTPUT:**

****

1. Logical/Virtual Address Conversion Using Paging and Segmentation (C++)

import java.util.concurrent.Semaphore;

#include <iostream>

#include <vector>

#include <iomanip>

using namespace std;

void paging() {

    cout << "\n=== Paging ===\n";

    int pageSize, numPages;

    cout << "Enter page size: ";

    cin >> pageSize;

    cout << "Enter number of pages: ";

    cin >> numPages;

    vector<int> pageTable(numPages);

    cout << "Enter frame number for each page:\n";

    for (int i = 0; i < numPages; i++) {

        cout << "Page " << i << ": ";

        cin >> pageTable[i];

    }

    int logicalAddress;

    cout << "Enter logical address: ";

    cin >> logicalAddress;

    int pageNumber = logicalAddress / pageSize;

    int offset = logicalAddress % pageSize;

    if (pageNumber >= numPages) {

        cout << "Invalid logical address!\n";

        return;

    }

    int frameNumber = pageTable[pageNumber];

    int physicalAddress = frameNumber \* pageSize + offset;

    cout << "Physical Address: " << physicalAddress << "\n";

}

void segmentation() {

    cout << "\n=== Segmentation ===\n";

    int numSegments;

    cout << "Enter number of segments: ";

    cin >> numSegments;

    vector<int> base(numSegments), limit(numSegments);

    cout << "Enter base and limit for each segment:\n";

    for (int i = 0; i < numSegments; i++) {

        cout << "Segment " << i << " - Base: ";

        cin >> base[i];

        cout << "Segment " << i << " - Limit: ";

        cin >> limit[i];

    }

    int segmentNumber, offset;

    cout << "Enter logical address (segment\_number offset): ";

    cin >> segmentNumber >> offset;

    if (segmentNumber >= numSegments || offset >= limit[segmentNumber]) {

        cout << "Invalid segment or offset!\n";

        return;

    }

    int physicalAddress = base[segmentNumber] + offset;

    cout << "Physical Address: " << physicalAddress << "\n";

}

int main() {

    paging();

    segmentation();

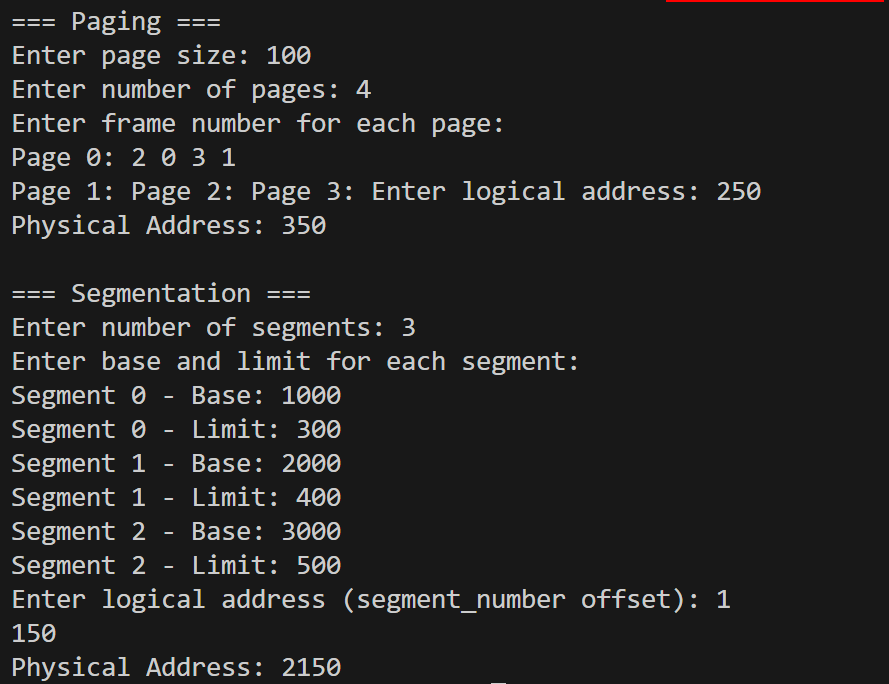
    return 0;

}

    }

}

**OUTPUT:**

****

**4.  Page Replacemet algorithms (FIFO, LRU, Optimal)**

#include <vector>

#include <iostream>

#include <queue>

#include <unordered\_map>

#include <algorithm>

#include <iomanip>

using namespace std;

void printMatrix(const vector<vector<int>>& mat, const vector<int>& ref) {

    cout << "\n=== Page Replacement Matrix ===\n   ";

    for (int r : ref)

        cout << setw(3) << r;

    cout << "\n";

    for (int i = 0; i < mat.size(); ++i) {

        cout << "F" << i + 1 << " ";

        for (int j = 0; j < mat[0].size(); ++j) {

            if (mat[i][j] == -1) cout << "  -";

            else cout << setw(3) << mat[i][j];

        }

        cout << "\n";

    }

}

void FIFO(vector<int> pages, int frames) {

    cout << "\n=== FIFO Page Replacement ===\n";

    queue<int> q;

    unordered\_map<int, bool> inMemory;

    vector<vector<int>> matrix(frames, vector<int>(pages.size(), -1));

    int pageFaults = 0, idx = 0;

    for (int i = 0; i < pages.size(); i++) {

        if (!inMemory[pages[i]]) {

            pageFaults++;

            if (q.size() == frames) {

                int out = q.front(); q.pop();

                inMemory[out] = false;

                for (int j = 0; j < frames; j++)

                    if (matrix[j][i - 1] == out) { matrix[j][i] = pages[i]; break; }

            } else {

                for (int j = 0; j < frames; j++) {

                    if (matrix[j][i] == -1 || (i > 0 && matrix[j][i - 1] == -1)) {

                        matrix[j][i] = pages[i];

                        break;

                    }

                }

            }

            q.push(pages[i]);

            inMemory[pages[i]] = true;

        }

        for (int j = 0; j < frames; j++) {

            if (i > 0 && matrix[j][i] == -1)

                matrix[j][i] = matrix[j][i - 1];

        }

    }

    printMatrix(matrix, pages);

    cout << "Total Page Faults (FIFO): " << pageFaults << "\n";

}

void LRU(vector<int> pages, int frames) {

    cout << "\n=== LRU Page Replacement ===\n";

    vector<vector<int>> matrix(frames, vector<int>(pages.size(), -1));

    unordered\_map<int, int> lastUsed;

    vector<int> mem;

    int pageFaults = 0;

    for (int i = 0; i < pages.size(); i++) {

        if (find(mem.begin(), mem.end(), pages[i]) == mem.end()) {

            pageFaults++;

            if (mem.size() == frames) {

                int lru = mem[0];

                for (int p : mem)

                    if (lastUsed[p] < lastUsed[lru]) lru = p;

                replace(mem.begin(), mem.end(), lru, pages[i]);

            } else mem.push\_back(pages[i]);

        }

        lastUsed[pages[i]] = i;

        for (int j = 0; j < frames; j++) {

            if (j < mem.size())

                matrix[j][i] = mem[j];

            else if (i > 0)

                matrix[j][i] = matrix[j][i - 1];

        }

    }

    printMatrix(matrix, pages);

    cout << "Total Page Faults (LRU): " << pageFaults << "\n";

}

void Optimal(vector<int> pages, int frames) {

    cout << "\n=== Optimal Page Replacement ===\n";

    vector<vector<int>> matrix(frames, vector<int>(pages.size(), -1));

    vector<int> mem;

    int pageFaults = 0;

    for (int i = 0; i < pages.size(); i++) {

        if (find(mem.begin(), mem.end(), pages[i]) == mem.end()) {

            pageFaults++;

            if (mem.size() == frames) {

                int index = -1, farthest = i;

                for (int j = 0; j < mem.size(); j++) {

                    int k;

                    for (k = i + 1; k < pages.size(); k++)

                        if (pages[k] == mem[j])

                            break;

                    if (k > farthest) {

                        farthest = k;

                        index = j;

                    }

                }

                if (index == -1) mem[0] = pages[i];

                else mem[index] = pages[i];

            } else mem.push\_back(pages[i]);

        }

        for (int j = 0; j < frames; j++) {

            if (j < mem.size())

                matrix[j][i] = mem[j];

            else if (i > 0)

                matrix[j][i] = matrix[j][i - 1];

        }

    }

    printMatrix(matrix, pages);

    cout << "Total Page Faults (Optimal): " << pageFaults << "\n";

}

int main() {

    int n, f;

    cout << "Enter number of pages: ";

    cin >> n;

    vector<int> pages(n);

    cout << "Enter page reference string: ";

    for (int i = 0; i < n; i++) cin >> pages[i];

    cout << "Enter number of frames: ";

    cin >> f;

    FIFO(pages, f);

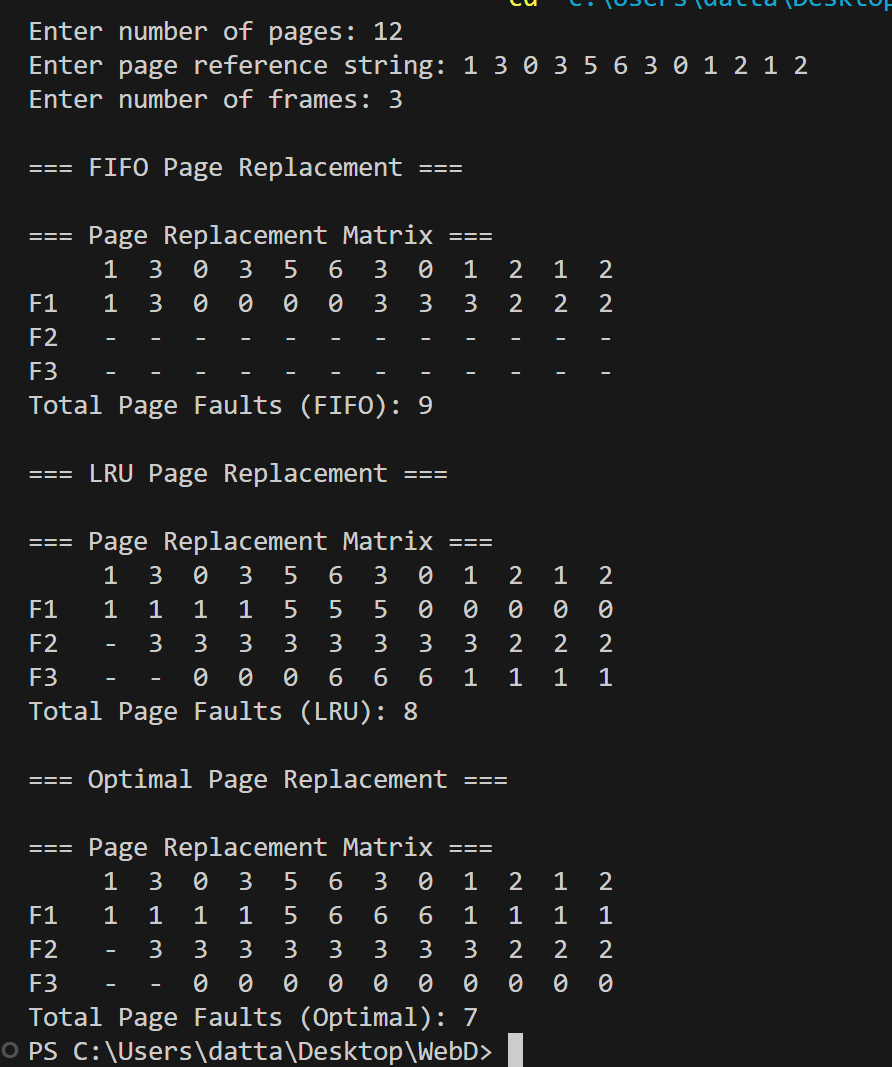
    LRU(pages, f);

    Optimal(pages, f);

    return 0;

}

**OUTPUT :**

****