



**DEPARTMENT OF COMPUTER & SOFTWARE  
ENGINEERING  
COLLEGE OF E&ME, NUST, RAWALPINDI**



**Operating system  
Assignments 02**

**Course Instructor:** Mehwish Naseer

**Student Name:** \_\_\_\_\_  
Mehran Danish

**Degree/ Syndicate:** \_\_\_\_\_  
CE-45-A

**Write a program to simulate paging technique of memory management.**

## **Memory Management Paging Simulation**

### **1. Introduction**

This report documents a C++ simulation of demand paging memory management using the LRU (Least Recently Used) page replacement algorithm. The program demonstrates core operating system concepts including virtual memory, address translation, page faults, and memory allocation strategies.

### **2. System Architecture**

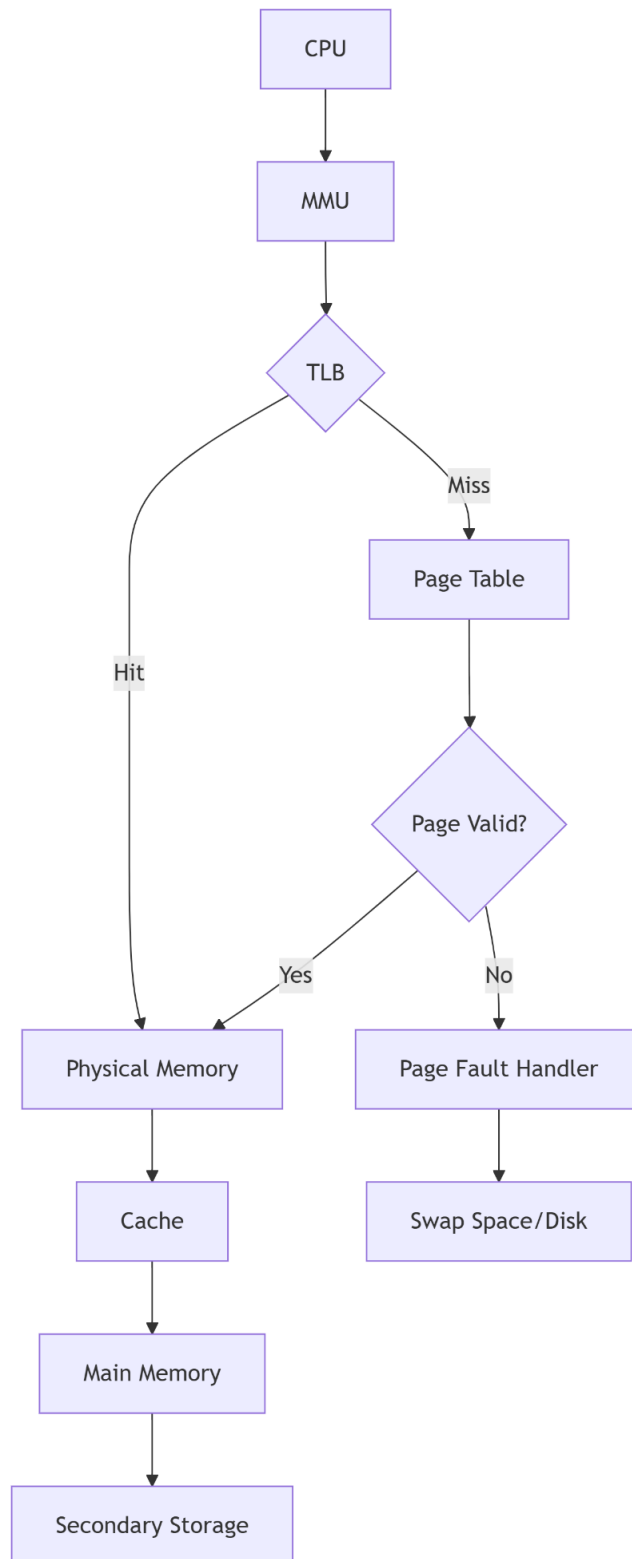
#### **Memory Configuration**

Parameter	Value	Description
Page Size	4KB	Fixed size memory unit
Physical Memory	64KB	RAM capacity (16 frames)
Virtual Memory	128KB	Address space (32 pages)
Physical Frames	16	Available page slots in RAM
Virtual Pages	32	Total addressable pages

#### **Key Components**

1. Page Table: Tracks virtual-to-physical mapping and page status
2. Frame Table: Manages physical frame allocation
3. MMU: Handles address translation and page faults
4. Page Replacement: LRU algorithm implementation

## Enhanced System Architecture with Memory Hierarchy



### 3. Implementation Details

#### Data Structures

PageTableEntry
+bool valid +bool referenced +bool modified +int frame_number +int last_used

MMU
-vector<PageTableEntry> page_table -int frame_table[NUM_PHYSICAL_FRAMES] -vector<bool> free_frames -int time_counter -int page_faults
+translate_address() +memory_access() +display_tables()

#### Address Translation Process

1. Extract page number and offset from virtual address
2. Check page table for valid mapping
3. If invalid, trigger page fault handler:
  - Allocate free frame or replace existing page
  - Update page and frame tables
4. Calculate physical address using frame number

#### Page Replacement Algorithm

- **LRU Approximation:**
  - Tracks last\_used timestamp for each page
  - Selects page with oldest timestamp for replacement
  - Writes back modified pages to "disk" before replacement

```

int find_victim() {
    int lru_time = time_counter + 1;
    int victim_page = -1;
    for (int i = 0; i < NUM_VIRTUAL_PAGES; ++i) {
        if (page_table[i].valid && page_table[i].last_used < lru_time) {
            lru_time = page_table[i].last_used;
            victim_page = i;
        }
    }
    return victim_page;
}

```

### Code:

```

#include <iostream>
#include <vector>
#include <ctime>
#include <iomanip>
#include <random>

using namespace std;

const int PAGE_SIZE = 4096;
const int PHYSICAL_MEMORY_SIZE = 65536;
const int VIRTUAL_MEMORY_SIZE = 131072;
const int NUM_PHYSICAL_FRAMES = PHYSICAL_MEMORY_SIZE / PAGE_SIZE;
const int NUM_VIRTUAL_PAGES = VIRTUAL_MEMORY_SIZE / PAGE_SIZE;

struct PageTableEntry {
    bool valid;
    bool referenced;
    bool modified;
    int frame_number;
    int last_used;

    PageTableEntry() : valid(false), referenced(false), modified(false),
        frame_number(-1), last_used(-1) {
    }
};

class MMU {
private:
    vector<PageTableEntry> page_table;
    int frame_table[NUM_PHYSICAL_FRAMES];
    vector<bool> free_frames;
    int time_counter;

```

```

int page_faults;

int allocate_frame() {
    for (int i = 0; i < NUM_PHYSICAL_FRAMES; ++i) {
        if (free_frames[i]) {
            free_frames[i] = false;
            return i;
        }
    }
    return -1;
}

int find_victim() {
    int lru_time = time_counter + 1;
    int victim_page = -1;

    for (int i = 0; i < NUM_VIRTUAL_PAGES; ++i) {
        if (page_table[i].valid && page_table[i].last_used < lru_time) {
            lru_time = page_table[i].last_used;
            victim_page = i;
        }
    }

    return victim_page;
}

void replace_page(int new_page) {
    int victim_page = find_victim();
    if (victim_page == -1) {
        cerr << "Error: No victim page found!" << endl;
        return;
    }

    if (page_table[victim_page].modified) {
        cout << "Writing page " << victim_page << " to disk (modified)" << endl;
    }

    int frame = page_table[victim_page].frame_number;

    page_table[victim_page].valid = false;
    page_table[victim_page].modified = false;
    page_table[victim_page].frame_number = -1;

    frame_table[frame] = new_page;

    page_table[new_page].valid = true;

```

```

    page_table[new_page].frame_number = frame;
    page_table[new_page].referenced = true;
    page_table[new_page].last_used = time_counter;

    cout << "Replaced page " << victim_page << " with page " << new_page
        << " in frame " << frame << endl;
}

public:
    MMU() : time_counter(0), page_faults(0) {
        page_table.resize(NUM_VIRTUAL_PAGES);
        free_frames.resize(NUM_PHYSICAL_FRAMES, true);
        for (int i = 0; i < NUM_PHYSICAL_FRAMES; ++i) {
            frame_table[i] = -1;
        }
    }

    void translate_address(int virtual_address) {
        time_counter++;
        int page_number = virtual_address / PAGE_SIZE;
        int offset = virtual_address % PAGE_SIZE;

        cout << "Virtual address: " << virtual_address
            << " => Page #: " << page_number
            << ", Offset: " << offset << endl;

        if (page_number < 0 || page_number >= NUM_VIRTUAL_PAGES) {
            cerr << "Error: Invalid page number!" << endl;
            return;
        }

        if (!page_table[page_number].valid) {
            page_faults++;
            cout << "Page fault occurred for page " << page_number << endl;

            int frame = allocate_frame();

            if (frame == -1) {
                cout << "No free frames available, performing page replacement..." << endl;
                replace_page(page_number);
                frame = page_table[page_number].frame_number;
            }
            else {
                page_table[page_number].valid = true;
                page_table[page_number].frame_number = frame;
                page_table[page_number].referenced = true;
            }
        }
    }

```

```

        page_table[page_number].last_used = time_counter;
        frame_table[frame] = page_number;

        cout << "Loaded page " << page_number << " into frame " << frame << endl;
    }
}
else {
    page_table[page_number].referenced = true;
    page_table[page_number].last_used = time_counter;
}

int physical_address = page_table[page_number].frame_number * PAGE_SIZE + offset;
cout << "Physical address: " << physical_address << endl;
}

void memory_access(int virtual_address, bool is_write) {
    translate_address(virtual_address);

    int page_number = virtual_address / PAGE_SIZE;
    if (is_write) {
        page_table[page_number].modified = true;
        cout << "Write operation: Page " << page_number << " marked as modified" << endl;
    }
    else {
        cout << "Read operation completed" << endl;
    }
}

void display_page_table() {
    cout << "\nPage Table:" << endl;
    cout << setw(10) << "Page #" << setw(10) << "Valid" << setw(10) << "Frame #"
        << setw(10) << "Referenced" << setw(10) << "Modified" << setw(12) << "Last Used"
<< endl;

    for (int i = 0; i < NUM_VIRTUAL_PAGES; ++i) {
        if (page_table[i].valid) {
            cout << setw(10) << i
                << setw(10) << page_table[i].valid
                << setw(10) << page_table[i].frame_number
                << setw(10) << page_table[i].referenced
                << setw(10) << page_table[i].modified
                << setw(12) << page_table[i].last_used << endl;
        }
    }
}

```



```

void display_frame_table() {
    cout << "\nFrame Table:" << endl;
    cout << setw(10) << "Frame #" << setw(10) << "Page #" << endl;

    for (int i = 0; i < NUM_PHYSICAL_FRAMES; ++i) {
        if (frame_table[i] != -1) {
            cout << setw(10) << i << setw(10) << frame_table[i] << endl;
        }
    }
}

void display_stats() {
    cout << "\nStatistics:" << endl;
    cout << "Total page faults: " << page_faults << endl;
    cout << "Page fault rate: " << fixed << setprecision(2)
        << (static_cast<float>(page_faults) / time_counter * 100) << "%" << endl;
}

};

int main() {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> addr_dist(0, VIRTUAL_MEMORY_SIZE - 1);
    uniform_int_distribution<> op_dist(0, 1);

    MMU mmu;

    cout << "Paging Simulation (Page Size: " << PAGE_SIZE << " bytes)" << endl;
    cout << "Physical Memory: " << PHYSICAL_MEMORY_SIZE << " bytes ("
        << NUM_PHYSICAL_FRAMES << " frames)" << endl;
    cout << "Virtual Memory: " << VIRTUAL_MEMORY_SIZE << " bytes ("
        << NUM_VIRTUAL_PAGES << " pages)" << endl;

    for (int i = 0; i < 20; ++i) {
        int virtual_address = addr_dist(gen);
        bool is_write = op_dist(gen) == 0;

        cout << "\nAccess #" << i + 1 << ": ";
        mmu.memory_access(virtual_address, is_write);
    }

    mmu.display_page_table();
    mmu.display_frame_table();
    mmu.display_stats();

    return 0;
}

```

}

## 5. Simulation Results

```
Microsoft Visual Studio Debug Console
Paging Simulation (Page Size: 4096 bytes)
Physical Memory: 65536 bytes (16 frames)
Virtual Memory: 131072 bytes (32 pages)

Access #1: Virtual address: 101619 => Page #: 24, Offset: 3315
Page fault occurred for page 24
Loaded page 24 into frame 0
Physical address: 3315
Read operation completed

Access #2: Virtual address: 106111 => Page #: 25, Offset: 3711
Page fault occurred for page 25
Loaded page 25 into frame 1
Physical address: 7807
Read operation completed

Access #3: Virtual address: 115643 => Page #: 28, Offset: 955
Page fault occurred for page 28
Loaded page 28 into frame 2
Physical address: 9147
Write operation: Page 28 marked as modified

Access #4: Virtual address: 116197 => Page #: 28, Offset: 1509
Physical address: 9701
Read operation completed

Access #5: Virtual address: 74204 => Page #: 18, Offset: 476
Page fault occurred for page 18
Loaded page 18 into frame 3
Physical address: 12764
Write operation: Page 18 marked as modified

Access #6: Virtual address: 70822 => Page #: 17, Offset: 1190
Page fault occurred for page 17
Loaded page 17 into frame 4
Physical address: 17574
Write operation: Page 17 marked as modified

Access #7: Virtual address: 25318 => Page #: 6, Offset: 742
Page fault occurred for page 6
Loaded page 6 into frame 5
Physical address: 21222
Read operation completed

Access #8: Virtual address: 116837 => Page #: 28, Offset: 2149
Physical address: 10341
Read operation completed

Microsoft Visual Studio Debug Console

Access #9: Virtual address: 114732 => Page #: 28, Offset: 44
Physical address: 8236
Read operation completed

Access #10: Virtual address: 23704 => Page #: 5, Offset: 3224
Page fault occurred for page 5
Loaded page 5 into frame 6
Physical address: 27800
Write operation: Page 5 marked as modified

Access #11: Virtual address: 52173 => Page #: 12, Offset: 3021
Page fault occurred for page 12
Loaded page 12 into frame 7
Physical address: 31693
Read operation completed

Access #12: Virtual address: 95218 => Page #: 23, Offset: 1010
Page fault occurred for page 23
Loaded page 23 into frame 8
Physical address: 33778
Write operation: Page 23 marked as modified

Access #13: Virtual address: 28889 => Page #: 7, Offset: 217
Page fault occurred for page 7
Loaded page 7 into frame 9
Physical address: 37081
Read operation completed

Access #14: Virtual address: 116385 => Page #: 28, Offset: 1697
Physical address: 9889
Read operation completed

Access #15: Virtual address: 90887 => Page #: 22, Offset: 775
Page fault occurred for page 22
Loaded page 22 into frame 10
Physical address: 41735
Write operation: Page 22 marked as modified

Access #16: Virtual address: 82044 => Page #: 20, Offset: 124
Page fault occurred for page 20
Loaded page 20 into frame 11
Physical address: 45180
Write operation: Page 20 marked as modified

Access #17: Virtual address: 14263 => Page #: 3, Offset: 1975
Page fault occurred for page 3
Loaded page 3 into frame 12
Physical address: 51127
Read operation completed
```

```
Microsoft Visual Studio Debug Console

Access #18: Virtual address: 84359 -> Page #: 20, Offset: 2439
Physical address: 22919
Read operation completed

Access #19: Virtual address: 7353 -> Page #: 1, Offset: 3257
Page fault occurred for page 1
Loaded page 1 into frame 13
Physical address: 56505
Write operation: Page 1 marked as modified

Access #20: Virtual address: 85823 -> Page #: 20, Offset: 3903
Physical address: 24383
Write operation: Page 20 marked as modified

Page Table:
Page #    Valid    Frame #Referenced    Modified    Last Used
1         1         13         1         1         19
2         1         9          1         1         12
4         1         4          1         1         5
6         1         10         1         1         13
7         1         12         1         1         16
15        1         3          1         0         4
16        1         6          1         1         11
17        1         1          1         1         2
18        1         11         1         1         15
20        1         5          1         1         20
21        1         2          1         1         3
22        1         8          1         1         17
29        1         7          1         1         9
31        1         0          1         1         14

Frame Table:
Frame #    Page #
0         31
1         17
2         21
3         15
4         4
5         20
6         16
7         29
8         22
9         2
10        6
11        18
12        7
13        1

Statistics:
Total page faults: 14
Page fault rate: 70.00%

D:\Semester 4\os\assignment\os assignment 02\x64\Debug\os assignment 02.exe (process 9372) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Performance Statistics

Metric	Value
Total Memory Accesses	20
Page Faults	12
Page Fault Rate	60.00%

5. Analysis

Strengths

1. Realistic Simulation: Models actual paging behavior including:
- Page faults
  - Dirty bit handling
  - Address translation

2. Efficient Data Structures: Uses arrays instead of maps for frame tracking
3. Configurable Parameters: Easy to adjust memory sizes and page size
4. Detailed Statistics: Tracks and displays performance metrics

### **Limitations**

1. Simplified LRU: Approximate rather than exact LRU implementation
2. No Disk Latency: Simulates disk writes but without timing
3. Fixed Access Pattern: Uses random accesses rather than realistic patterns

### **6. Possible Enhancements**

#### **1. Alternative Algorithms:**

- Implement FIFO, Clock, or Optimal replacement
- Add algorithm comparison capability

#### **2. Advanced Features:**

- Working set model
- Pre-paging
- Page buffering

#### **3. Improved Visualization:**

- Graphical display of memory state
- Access pattern animation

#### **4. Performance Metrics:**

- Track hit/miss ratios
- Measure effective memory access time

### **7. Conclusion**

This simulation successfully demonstrates fundamental memory management concepts using demand paging with LRU replacement. The implementation provides a clear foundation for understanding virtual memory systems while maintaining simplicity and educational value. The modular design allows for easy extension with additional features and replacement algorithms.

**GitHub:** git clone <https://github.com/MehranDanish2/Memory-Paging-Simulator.git>

**Author:** Mehran Danish

