

CSA0404-Operating System(slot-c)

PAGE REPLACEMENT ALGORITHM

Guided By,
Dr. G. Mary Valantina
(Course Faculty)
SSE,SIMATS.

Project By,
P. Sivanvitha(192224041)
T. Nikitha(192224053)
T. Naga Chandu(192211843)
SSE,SIMATS

ABSTRACT:

Page replacement algorithms are crucial components of virtual memory management systems, aiming to optimize memory utilization and minimize page faults. This paper presents a comparative study of three fundamental page replacement algorithms: First-In-First-Out (FIFO), Least Recently Used (LRU), and Optimal. FIFO operates on the principle of evicting the oldest page in memory when a new page needs to be brought in. LRU, on the other hand, evicts the page that has not been accessed for the longest period. Optimal, often considered the theoretical ideal, selects the page that will not be accessed for the longest time in the future. The study evaluates these algorithms across various metrics including page fault rates, computational complexity, and adaptability to different workload scenarios. Simulations are conducted to analyze the algorithm behavior under diverse conditions, ranging from sequential access patterns to irregular access distributions. Furthermore, the paper explores practical considerations such as implementation overhead and memory footprint associated with each algorithm. Insights gained from the study can aid system designers and developers in selecting the most appropriate page replacement strategy based on system requirements and constraints. By shedding light on the strengths and limitations of FIFO, LRU, and Optimal algorithms, this study contributes to the ongoing discourse on memory management optimization and provides valuable guidance for improving system performance in virtual memory environments.

INTRODUCTION:

Page replacement algorithms are critical components of memory management in computer operating systems, particularly in systems that utilize virtual memory. These algorithms are responsible for selecting which page to evict from memory when a new page needs to be loaded in. Among the various page replacement algorithms, three commonly used ones are First-In-First-Out (FIFO), Least Recently Used (LRU), and Optimal.

1.FIFO (First-In-First-Out): FIFO is one of the simplest page replacement algorithms. It operates on the principle of a queue: the page that has been in memory the longest is the one selected for replacement. When a page needs to be replaced, the oldest page in memory, i.e., the one that entered first, is removed.

2.LRU (Least Recently Used): LRU is based on the idea that the page that has not been accessed for the longest period of time is the one to replace. It requires keeping track of the time when each page was last accessed.

3.Optimal: The Optimal algorithm, also known as the "Belady's Optimal Algorithm," serves as a theoretical benchmark for comparing other page replacement algorithms. It operates by selecting the page that will not be used for the longest period of time in the future.

LITERATURE REVIEW:

- ▶ Numerous studies have been conducted to investigate the performance of page replacement algorithms in virtual memory systems.
- ▶ Early research focused on simple algorithms such as FIFO and LRU, examining their theoretical properties and practical implications.
- ▶ Subsequent work introduced more sophisticated algorithms and proposed optimizations to address the limitations of existing approaches. Comparative studies have been conducted to benchmark the performance of different algorithms using various metrics, including page fault rate, cache hit rate, and system overhead.
- ▶ Additionally, researchers have explored the impact of workload characteristics on the effectiveness of page replacement strategies, leading to the development of adaptive and hybrid approaches.

IMPLEMENTATION:

1.FIRST-IN-FIRST-OUT(FIFO):

FIFO is one of the simplest page replacement algorithms. It operates on the principle of evicting the oldest page in memory when a new page needs to be brought in. Here's a step-by-step explanation of how FIFO works:

1.Initialization: Maintain a queue (or array) of fixed size representing the frames in memory. Initially, all frames are empty.

2.Page Fault Handling:

1. When a page fault occurs (i.e., a requested page is not in memory):
2. If there is free space in memory (i.e., the number of frames used is less than the total number of frames available)
3. If memory is full:
 1. Select the page that was brought into memory first (i.e., the oldest page) for eviction.
 2. Remove the oldest page from the front of the queue.
 3. Bring the new page into memory and place it at the end of the queue.

3.Counting Page Faults: Increment a counter each time a page fault occurs.

FIFO's simplicity comes from its straightforward eviction strategy: it always evicts the page that has been in memory the longest. This algorithm is easy to implement and has a low overhead, making it suitable for systems with limited computational resources.

2.LEAST RECENTLY USED(LRU):

LRU operates on the principle of evicting the page that has not been accessed for the longest time. Here's how it works:

1.Initialization: Maintain a data structure (such as a linked list, queue) to keep track of the order in which pages are accessed. Initially, all frames are empty.

2.Page Access Handling:

1. When a page is accessed:
2. If the page is already in memory:
 - Update its position in the data structure to reflect that it was the most recently used page.
- 3.If the page is not in memory (i.e., a page fault occurs):
 - Bring the requested page into memory and place it at the front of the data structure to indicate that it's the most recently used page.
 - If memory is full, evict the page at the end of the data structure (the least recently used page).

3.Counting Page Faults: Increment a counter each time a page fault occurs.

The key idea behind LRU is to approximate the optimal page replacement strategy by assuming that the least recently used pages are less likely to be used in the near future.

3.OPTIMAL:

1.Initialization: Maintain a data structure (such as a hashmap or array) to keep track of future page accesses.

2.Predicting Future Page Accesses:

1. When a page is brought into memory, analyze the remaining reference string (sequence of page accesses) to predict when the page will be accessed next.
2. Select the page that will not be accessed for the longest time in the future for eviction.

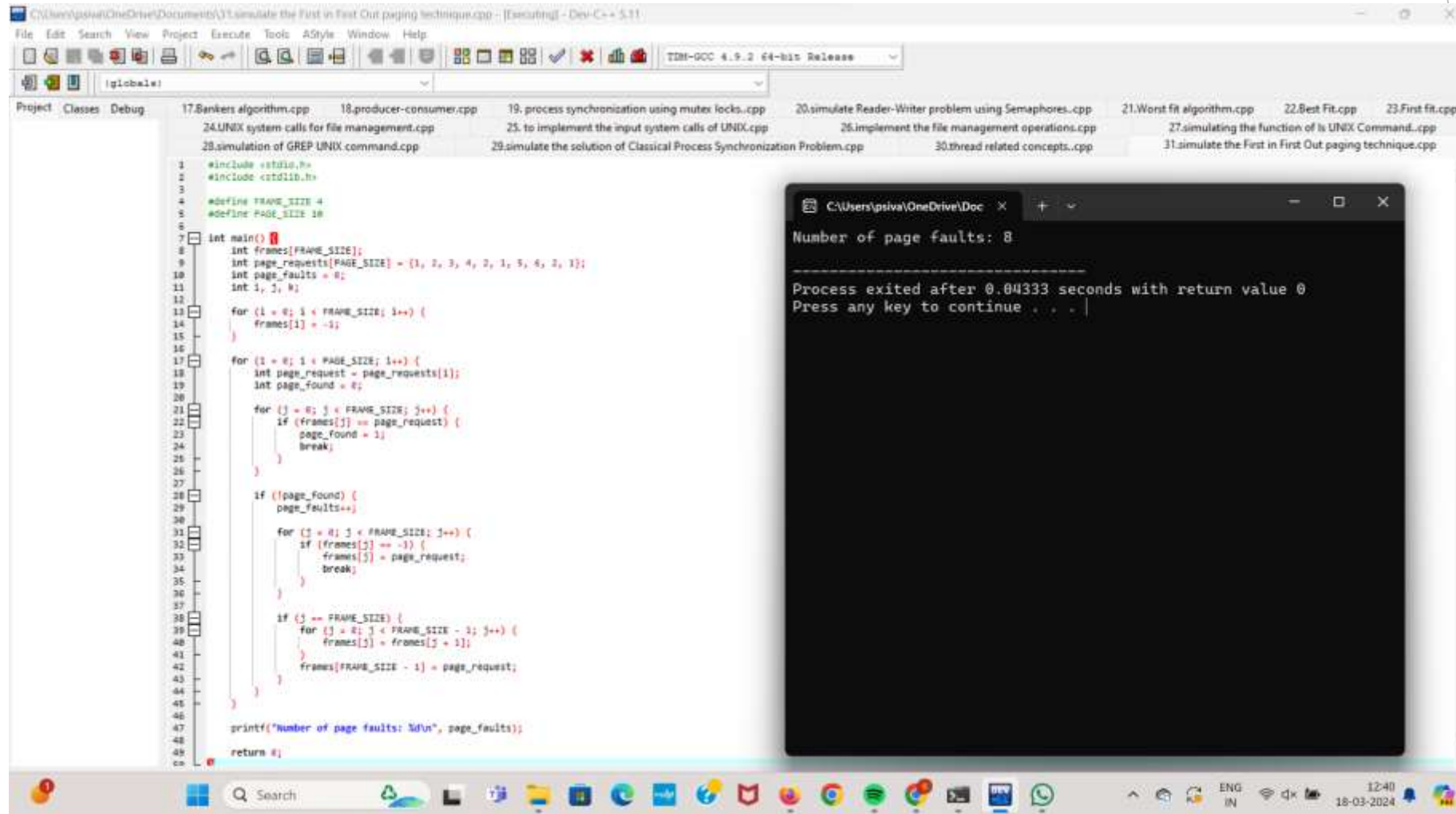
3.Page Fault Handling:

1. When a page fault occurs (i.e., a requested page is not in memory)
2. If there is free space in memory, simply bring the requested page into memory.
3. If memory is full:
 1. Predict future page accesses for each page in memory based on the remaining reference string.
 2. Evict the page that will not be accessed for the longest time in the future according to the predictions.
 3. Bring the new page into memory.

4.Counting Page Faults: Increment a counter each time a page fault occurs.

Implementing the Optimal algorithm involves simulating the future by scanning the remaining reference string from the current position to predict future page accesses.

OUTPUT FOR FIFO CODE:

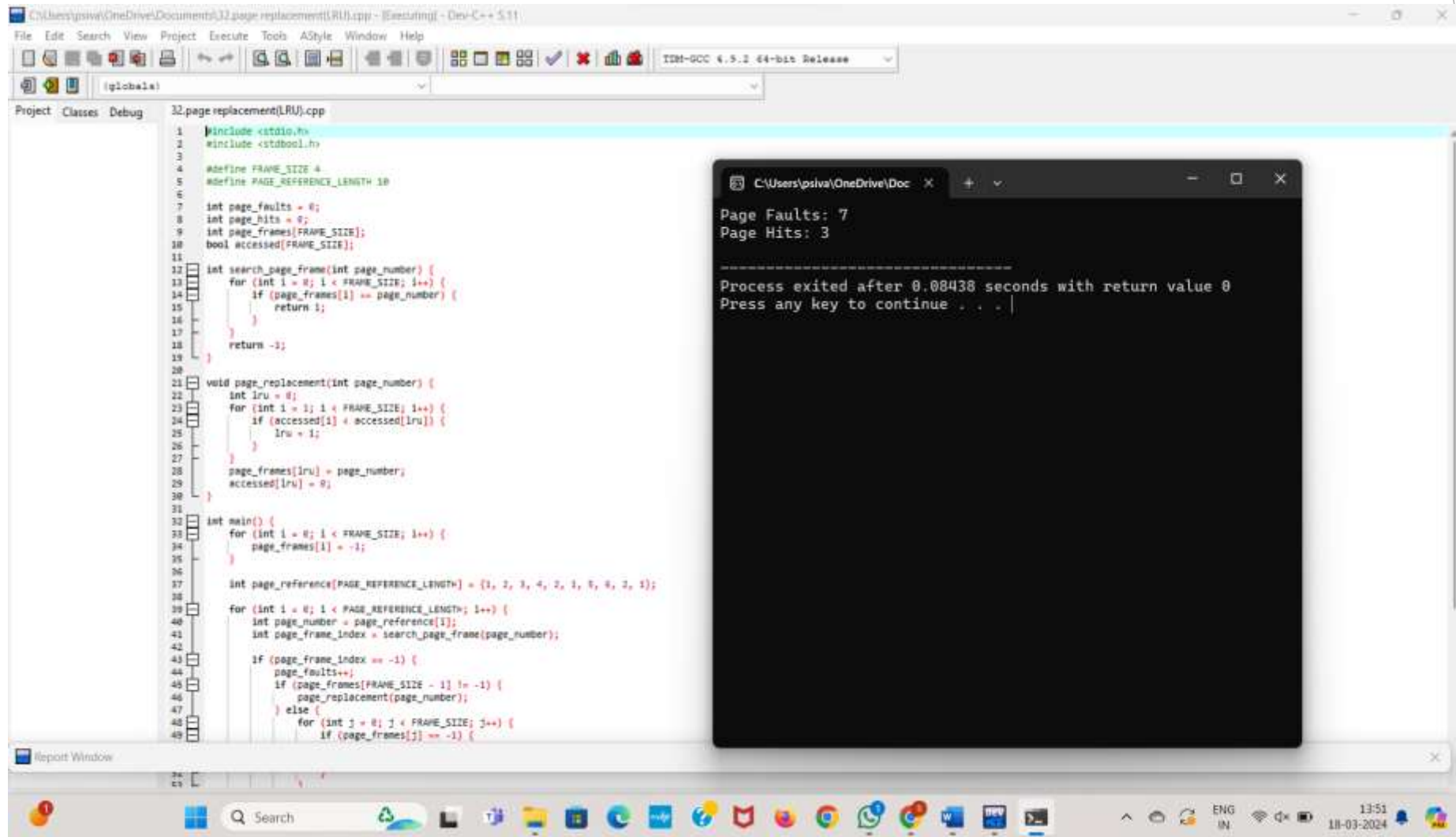


The screenshot displays a C++ IDE with the following components:

- File Explorer:** Shows a project structure with files like `17.Bankers algorithm.cpp`, `18.producer-consumer.cpp`, `19.process synchronization using mutex locks.cpp`, `20.simulate Reader-Writer problem using Semaphores.cpp`, `21.Worst fit algorithm.cpp`, `22.Best Fit.cpp`, `23.First fit.cpp`, `24.UNIX system calls for file management.cpp`, `25.to implement the input system calls of UNIX.cpp`, `26.implement the file management operations.cpp`, `27.simulating the function of ls UNIX Command.cpp`, `29.simulation of GREP UNIX command.cpp`, `29.simulate the solution of Classical Process Synchronization Problem.cpp`, `30.thread related concepts.cpp`, and `31.simulate the First in First Out paging technique.cpp`.
- Code Editor:** Contains the implementation of the FIFO paging algorithm. The code defines `FRAME_SIZE` as 4 and `PAGE_SIZE` as 10. It initializes an array `frames` of size `FRAME_SIZE` with -1. A sequence of page requests is provided: `{1, 2, 3, 4, 2, 1, 5, 6, 2, 1}`. The algorithm iterates through these requests, checking if the requested page is already in the frames. If not, it replaces the oldest page (the one at index 0) with the new request, incrementing the page fault count. If the page is already in the frames, no action is taken. The final output is "Number of page faults: 8".
- Output Window:** Displays the execution results: "Number of page faults: 8", "Process exited after 0.04333 seconds with return value 0", and "Press any key to continue . . .".

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define FRAME_SIZE 4
5 #define PAGE_SIZE 10
6
7 int main()
8 {
9     int frames[FRAME_SIZE];
10    int page_requests[PAGE_SIZE] = {1, 2, 3, 4, 2, 1, 5, 6, 2, 1};
11    int page_faults = 0;
12    int i, j;
13
14    for (i = 0; i < FRAME_SIZE; i++) {
15        frames[i] = -1;
16    }
17
18    for (i = 0; i < PAGE_SIZE; i++) {
19        int page_request = page_requests[i];
20        int page_found = 0;
21
22        for (j = 0; j < FRAME_SIZE; j++) {
23            if (frames[j] == page_request) {
24                page_found = 1;
25                break;
26            }
27        }
28
29        if (!page_found) {
30            page_faults++;
31
32            for (j = 0; j < FRAME_SIZE; j++) {
33                if (frames[j] == -1) {
34                    frames[j] = page_request;
35                    break;
36                }
37            }
38
39            if (j == FRAME_SIZE) {
40                for (j = 0; j < FRAME_SIZE - 1; j++) {
41                    frames[j] = frames[j + 1];
42                }
43                frames[FRAME_SIZE - 1] = page_request;
44            }
45        }
46    }
47
48    printf("Number of page faults: %d\n", page_faults);
49
50    return 0;
51 }
```


OUTPUT FOR LRU CODE:



The image shows a C++ IDE window titled "32.page replacement(LRU).cpp" with the following code:

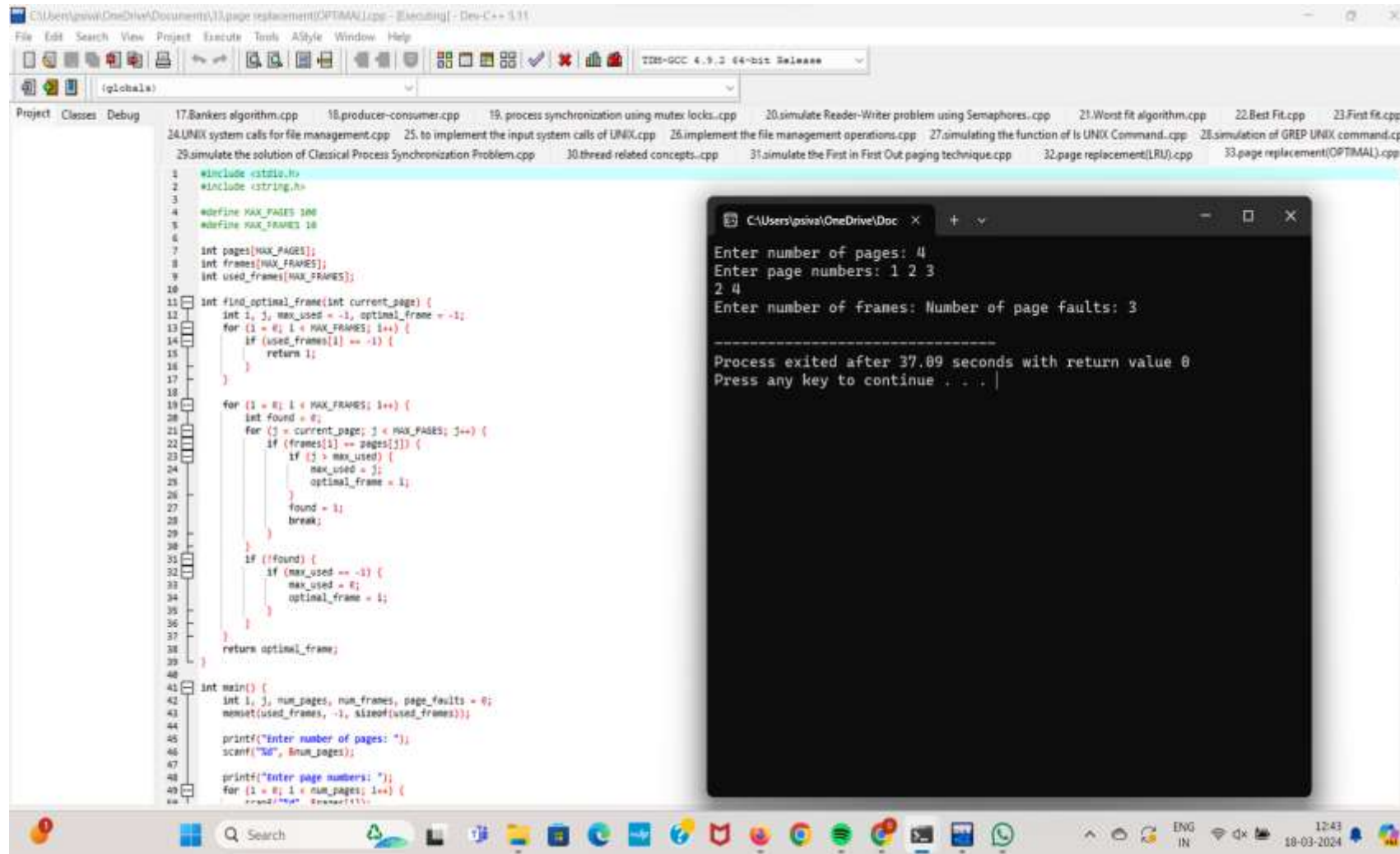
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define FRAME_SIZE 4
5 #define PAGE_REFERENCE_LENGTH 10
6
7 int page_faults = 0;
8 int page_hits = 0;
9 int page_frames[FRAME_SIZE];
10 bool accessed[FRAME_SIZE];
11
12 int search_page_frame(int page_number) {
13     for (int i = 0; i < FRAME_SIZE; i++) {
14         if (page_frames[i] == page_number) {
15             return i;
16         }
17     }
18     return -1;
19 }
20
21 void page_replacement(int page_number) {
22     int lru = 0;
23     for (int i = 1; i < FRAME_SIZE; i++) {
24         if (accessed[i] < accessed[lru]) {
25             lru = i;
26         }
27     }
28     page_frames[lru] = page_number;
29     accessed[lru] = 0;
30 }
31
32 int main() {
33     for (int i = 0; i < FRAME_SIZE; i++) {
34         page_frames[i] = -1;
35     }
36
37     int page_reference[PAGE_REFERENCE_LENGTH] = {1, 2, 3, 4, 2, 1, 5, 4, 3, 1};
38
39     for (int i = 0; i < PAGE_REFERENCE_LENGTH; i++) {
40         int page_number = page_reference[i];
41         int page_frame_index = search_page_frame(page_number);
42
43         if (page_frame_index == -1) {
44             page_faults++;
45             if (page_frames[FRAME_SIZE - 1] != -1) {
46                 page_replacement(page_number);
47             } else {
48                 for (int j = 0; j < FRAME_SIZE; j++) {
49                     if (page_frames[j] == -1) {
```

The output window shows the following results:

```
Page Faults: 7
Page Hits: 3

-----
Process exited after 0.08438 seconds with return value 0
Press any key to continue . . .
```

OUTPUT FOR OPTIMAL CODE:



The screenshot displays a C++ IDE with the source code for an optimal page replacement algorithm. The code defines a function `find_optimal_frame` that iterates through the list of pages to find the one that will not be used in the future, or the one that is used furthest in the future. The `main` function prompts the user for the number of pages and the sequence of page numbers, then calls `find_optimal_frame` to calculate the number of page faults.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 #define MAX_PAGES 100
5 #define MAX_FRAMES 10
6
7 int pages[MAX_PAGES];
8 int frames[MAX_FRAMES];
9 int used_frames[MAX_FRAMES];
10
11 int find_optimal_frame(int current_page) {
12     int i, j, max_used = -1, optimal_frame = -1;
13     for (i = 0; i < MAX_FRAMES; i++) {
14         if (used_frames[i] == -1) {
15             return i;
16         }
17     }
18     for (i = 0; i < MAX_FRAMES; i++) {
19         int found = 0;
20         for (j = current_page; j < MAX_PAGES; j++) {
21             if (frames[i] == pages[j]) {
22                 if (j > max_used) {
23                     max_used = j;
24                     optimal_frame = i;
25                 }
26                 found = 1;
27                 break;
28             }
29         }
30         if (!found) {
31             if (max_used == -1) {
32                 max_used = i;
33                 optimal_frame = i;
34             }
35         }
36     }
37     return optimal_frame;
38 }
39
40 int main() {
41     int i, j, num_pages, num_frames, page_faults = 0;
42     memset(used_frames, -1, sizeof(used_frames));
43
44     printf("Enter number of pages: ");
45     scanf("%d", &num_pages);
46
47     printf("Enter page numbers: ");
48     for (i = 0; i < num_pages; i++) {
49         scanf("%d", &pages[i]);
50     }
51 }
```

The terminal window shows the following output:

```
C:\Users\psiva\OneDrive\Doc x + v
Enter number of pages: 4
Enter page numbers: 1 2 3
2 4
Enter number of frames: Number of page faults: 3

-----
Process exited after 37.09 seconds with return value 0
Press any key to continue . . . |
```

FIRST-IN-FIRST-OUT(FIFO)

ADVANTAGES:

- Simple to implement.
- Low computational overhead.
- Fairly predictable behavior.

LIMITATIONS:

- Can suffer from the "Belady's Anomaly," where increasing the number of frames
- can lead to more page faults.

LEAST RECENTLY USED

ADVANTAGES:

- Attempts to approximate the optimal page replacement strategy by evicting the least recently used pages.
- Often performs better than FIFO in practice.

LIMITATIONS:

- May require more memory and computational overhead to maintain the access order of pages.
- Can be more complex to implement compared to FIFO.

OPTIMAL

ADVANTAGE:

- Provides a benchmark for evaluating the performance of other algorithms.

LIMITATIONS:

- Requires knowledge of future page accesses, which is typically not available in practice.
- Not feasible for practical implementations due to its reliance on future information.

CONCLUSION:

In conclusion, page replacement algorithms are fundamental to memory management in operating systems, determining the eviction and replacement of pages in memory. Each algorithm—FIFO, LRU, and Optimal—offers distinct advantages and limitations. FIFO, being the simplest, operates on a first-in-first-out basis but may suffer from the Belady's Anomaly, leading to increased page faults with more frames. LRU aims to approximate the optimal strategy by evicting the least recently used pages and generally outperforms FIFO, though it requires additional data structures and computational overhead. Optimal represents the theoretical best-case scenario but is impractical due to its reliance on future information. In practice, the choice of algorithm depends on system constraints and performance requirements, often involving a trade-off between simplicity, computational overhead, and performance.

Real-world implementations may combine strategies or employ variations of these algorithms to achieve optimal memory management and system performance. Understanding these algorithms' strengths, weaknesses, and practical implications is essential for designing efficient memory management systems in operating environments.

REFERENCES:

1. Bell, John. [*"Operating Systems Course Notes: Virtual Memory"*](#). University of Illinois at Chicago College of Engineering. [Archived](#) from the original on 23 September 2018. Retrieved 21 July 2017.
2. Torrez, Paul; et al. [*"CS111 Lecture 11 notes"*](#). UCLA Computer Science Department. Archived from [the original](#) on 9 January 2009.
3. Bahn, Hyokyung; Noh, Sam H. (12–14 February 2003). Characterization of Web reference behavior revisited: Evidence for Dichotomized Cache management. [*International Conference on Information Networking 2003*](#). Jeju, South Korea: Springer-Verlag. pp. 1018–1027. [doi:10.1007/978-3-540-45235-5_100](#). [ISBN 978-3-540-40827-7](#).
4. Tanenbaum, Andrew S. (2001). [*Modern Operating Systems*](#) (2nd ed.). Upper Saddle River, NJ, USA: Prentice-Hall. p. [218 \(4.4.5\)](#). [ISBN 978-0-13-031358-4](#). [LCCN 00051666](#). [OCLC 45284637](#). [OL 24214243M](#).