School of Engineering and Technology

Operating Systems

Laboratory Exercise No.7

Fundamentals of Operating System

JACOB S. SANTOS **11/4/24**

BSCOE 4-1 **Engr. Elizier Obamos**

Objective

- To understand and implement the Optimal Page Replacement Algorithm.
- To analyze and compare the performance of the optimal page replacement algorithm with other algorithms like FIFO and LRU.

Problem Description

In an operating system, page replacement algorithms decide which memory pages to remove to accommodate new pages. The Optimal Page Replacement Algorithm is a theoretical model that replaces the page that will not be needed for the longest period in the future. Although it's challenging to implement in real-time systems, it provides a benchmark to evaluate the efficiency of other algorithms.

Instructions:

- Explanation of the Optimal Page Replacement Algorithm:
- Explain that the algorithm predicts future page requests to decide which page to replace, minimizing page faults.
- Discuss the limitations of using the optimal algorithm in real-world applications but its value in comparing other algorithms' efficiencies.

Example Scenario:

Consider a sequence of page requests: [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2].

Assume a memory with a limited capacity (e.g., 3 frames).

Walk through each page request and determine which page should be replaced according to the optimal page replacement policy.

Task 1: Manual Trace

- Given a specific page sequence and frame size, students will manually trace the optimal page replacement algorithm.
- They should record each step, note which pages are loaded into frames, and track the number of page faults.

Task 2: Program Implementation

Write a program that simulates the optimal page replacement algorithm.

The program should:

- Accept a list of page requests and the number of frames as input.

- Implement logic to identify which page will not be needed for the longest period and replace it.
- Track and output the number of page faults

Task 3: Comparison with Other Algorithms

Analysis Questions:

Why is the optimal page replacement algorithm considered impractical in real systems?

Though theoretically efficient, the optimal page replacement algorithm is impractical in real systems as well because it requires knowledge of all future memory accesses ahead of making the decision to replace. A real operating system cannot predict an exact sequence of memory requests a program will make such that it cannot decide ahead of time which page would be used farthest into the future. Although the optimal algorithm may minimize page faults, its realization is impossible for a dynamic, nondeterministic workload with perfect foresight. As such, systems use approximation algorithms, such as Least Recently Used (LRU), that rely on past behavior to estimate future needs in the absence of impossible knowledge of the future.

How could knowing the access pattern in advance affect system performance?

Knowing the access pattern beforehand would significantly improve the system performance since the OS would be able to make optimal decisions on memory management, thus minimizing page faults and maximizing cache efficiency. The OS would pre-load the pages in the memory which are required the next time and remove those that will not be needed the longest time. This would reduce the number of slower memory swaps between disk and RAM and thus lead to faster execution of programs, lower latencies, and better responsiveness of systems. The system would essentially work closer to a theoretical ideal in memory management, with predictive access patterns that could be translated into improved performance, especially for high-demand and real-time applications.

Submit:

- The manual trace of the optimal page replacement.

7	7	7	7	-	**3**	-	3	-	-	3	-	-
	0	0	0	-	0	-	**4**	-	-	**0**	-	-
		1	**2**	-	2	-	2	-	-	2	-	-

Step 1-3: Fill page requests to the number of frames specified 7, 0 , 1

Step 4 Replace 1 in 7, 0, 1 to 7, 0, 2 as it is the one not needed in the longest time in the future

Step 5: Retain value 7, 0, 2 because the next request is 0

Step 6: Replace 7 in 7, 0, 2 to 3, 0, 2 as it is the one not needed in the longest time in the future

Step 7: Retain value 3, 0, 2 because the next request is 0

Step 8: Replace 0 in 3, 0, 2 to 3, 4 ,2 as it is the one not needed in the longest time in the future

Step 9-10: Retain value 3, 4, 2 because the next page requests is in the frame namely 2, 3

Step 11: Replace 4 in 3, 4, 2 with 3, 0, 2 as it is the one not needed in the longest time in the future

Step 12-13: Retain Values because the next page requests is the same in the frame

Final State of Frames: 3, 0, 2

Number of Frames = 3

Page Faults = 7

- Results of the program, including the number of page faults.

```
Enter page requests (comma-separated): 7,0,1,2,0,3,0,4,2,3,0,3,2
Enter number of frames (1-3): 3
Step 1: [7] - Page fault
Step 2: [7, 0] - Page fault
Step 3: [7, 0, 1] - Page fault
Step 4: [7, 0, 2] - Page fault
Step 5: [7, 0, 2]
Step 6: [3, 0, 2] - Page fault
Step 7: [3, 0, 2]
Step 8: [3, 4, 2] - Page fault
Step 9: [3, 4, 2]
Step 10: [3, 4, 2]
Step 11: [3, 0, 2] - Page fault
Step 12: [3, 0, 2]
Step 13: [3, 0, 2]
Final state of frames: [3, 0, 2]
Total page faults: 7
```

- Source code for the optimal page replacement program.

```
package optimal;

import java.util.*;

public class OptimalPageReplacement {

    private static int pageFaults = 0;

    public static void main(String[] args) {
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.print("Enter page requests (comma-separated): ");
            String[] input = scanner.nextLine().split(",");
            List<Integer> pages = new ArrayList<>();
            for (String page : input) {
                pages.add(Integer.parseInt(page.trim()));
            }

            System.out.print("Enter number of frames (1-3): ");
            int frameCount = Integer.parseInt(scanner.nextLine().trim());

            if (frameCount < 1 || frameCount > 3) {
                System.out.println("Number of frames must be between 1 and
3.");
                return;
            }

            List<Integer> finalFrames = optimalPageReplacement(pages,
frameCount);

            System.out.println("Final state of frames: " + finalFrames);
            System.out.println("Total page faults: " + pageFaults);
```

```
            } catch (NumberFormatException e) {
                e.printStackTrace();
            }
        }

        private static List<Integer> optimalPageReplacement(List<Integer> pages,
    int frameCount) {
            List<Integer> frames = new ArrayList<>();

            for (int step = 0; step < pages.size(); step++) {
                int page = pages.get(step);
                boolean pageFaultOccurred = false;

                if (!frames.contains(page)) {
                    pageFaults++;
                    pageFaultOccurred = true;

                    if (frames.size() < frameCount) {
                        frames.add(page);
                    } else {
                        int pageToReplace = findPageToReplace(frames, pages,
    step);
                        frames.set(frames.indexOf(pageToReplace), page);
                    }
                }

                System.out.print("Step " + (step + 1) + ": " + frames);
                if (pageFaultOccurred) {
                    System.out.println(" - Page fault");
                } else {
                    System.out.println();
                }
            }

            return frames;
        }

        private static int findPageToReplace(List<Integer> frames, List<Integer>
    pages, int currentIndex) {
            int farthestIndex = -1;
            int pageToReplace = frames.get(0);

            for (int frame : frames) {
                int nextUseIndex = findNextUseCyclic(pages, currentIndex + 1,
    frame);

                if (nextUseIndex > farthestIndex) {
                    farthestIndex = nextUseIndex;
                    pageToReplace = frame;
                }
            }

            return pageToReplace;
        }

        private static int findNextUseCyclic(List<Integer> pages, int startIndex,
    int page) {
            int size = pages.size();
            for (int i = startIndex; i < startIndex + size; i++) {
                int index = i % size;
                if (pages.get(index) == page) {
                    return i;
                }
            }
            return Integer.MAX_VALUE;
        }
    }
```