Republic of the Philippines
Tarlac State University
COLLEGE OF COMPUTER STUDIES
Tarlac City, Tarlac
Tel. No. (045) 6068173

Page Replacement Algorithms Simulator

A Case Study Presented to

The Instructor of the Course

Operating System

In Partial Fulfillment

of the Requirements for the Subject

Operating System

Prepared by:

Soliman, Dennis Jr. M.

Submitted To:

Cura, Jo Anne S.

Instructor

May 21, 2025

Republic of the Philippines
Tarlac State University
COLLEGE OF COMPUTER STUDIES
Tarlac City, Tarlac
Tel. No. (045) 6068173

1. INTRODUCTION

1.1 Project Context

Operating systems use page replacement algorithms to manage limited physical memory resources efficiently. When all page frames are occupied and a new page is needed, these algorithms determine which existing page to evict. This study examines and simulates three page replacement algorithms which are:

- FIFO (First-In-First-Out): Follows a chronological rule in which the oldest ones get replaced
- LRU (Least Recently Used): Removes the least recently accessed page
- OPT (Optimal): Scans the pages forward before choosing the most optimal page to be replaced

2. Documentation

2.1 System Overview

The interactive simulator provides a customizable reference string generation and a random generation to allow other input. A configurable number for page frames and a comparative summary for the results

2.2 Interface Design

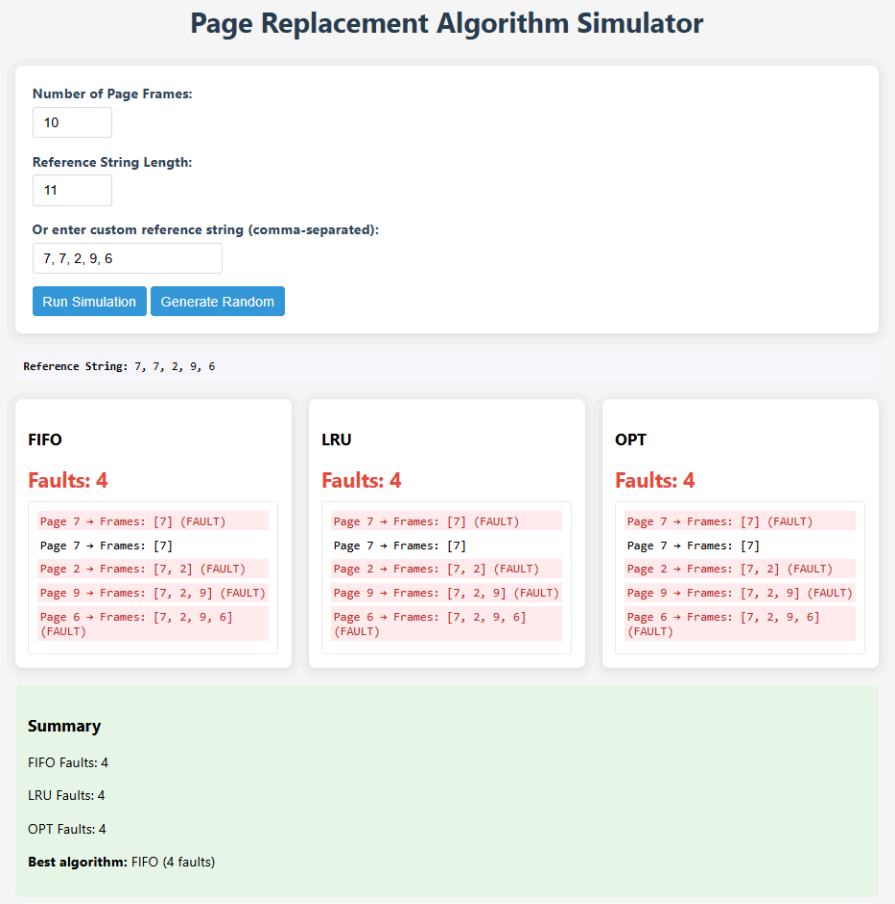

Figure 1: Page Replacement Algorithm Simulator interface

The program contains a program of three different instruments naming html, css and javascript to make a web page of a simulated page replacement algorithm. The design is made to

be simple and professional looking with the use of a box layered layout with color indicators to show expression of faults and the summary, red and green respectively.

3. Code Analysis

3.1 Core Algorithms

FIFO Implementation

```
// FIFO Implementation
function fifo(refString, numFrames) {
    const frames = [];
    let faults = 0;
    let pointer = 0;
    const steps = [];

    for (let i = 0; i < refString.length; i++) {
        const page = refString[i];
        const isFault = !frames.includes(page);

        if (isFault) {
            faults++;

            if (frames.length < numFrames) {
                frames.push(page);
            } else {
                frames[pointer] = page;
                pointer = (pointer + 1) % numFrames;
            }
        }

        steps.push({
            text: `Page ${page} → Frames: [${frames.join(', ')}]${isFault ? ' (FAULT)' : ''}`,
            isFault: isFault
        });
    }
    return { faults, steps };
}
```

Figure 2: FIFO algorithm snippet

This code snippet implements the First-In-First-Out page replacement algorithm using a circular pointer. When a page fault occurs, it replaces the oldest page in the frame (tracked by the pointer) and increments the pointer. The implementation tracks faults and generates step-by-step output showing frame states and fault occurrences.

LRU Implementation

```
// LRU Implementation
function lru(refString, numFrames) {
    const frames = [];
    const lastUsed = {};
    let faults = 0;
    let time = 0;
    const steps = [];

    for (let i = 0; i < refString.length; i++) {
        const page = refString[i];
        time++;
        const isFault = !frames.includes(page);

        if (isFault) {
            faults++;

            if (frames.length < numFrames) {
                frames.push(page);
            } else {
                let lruPage = frames[0];
                let minTime = lastUsed[lruPage];

                for (const framePage of frames) {
                    if (lastUsed[framePage] < minTime) {
                        minTime = lastUsed[framePage];
                        lruPage = framePage;
                    }
                }

                const index = frames.indexOf(lruPage);
                frames[index] = page;
            }
        }
        lastUsed[page] = time;
        steps.push({
            text: `Page ${page} → Frames: [${frames.join(', ')}]${isFault ? ' (FAULT)' : ''}`,
            isFault: isFault
        });
    }
    return { faults, steps };
}
```

Figure 3: LRU algorithm snippet

The code snippet implements the LRU algorithm which tracks page usage timestamps in a lastUsed dictionary. On page faults, it evicts the frame with the oldest timestamp, prioritizing recently accessed pages. A running time counter helps identify the least recently accessed page accurately. While more adaptive than FIFO, the timestamp maintenance adds slight overhead compared to simpler algorithms.

OPT Implementation

```javascript
// OPT Implementation
function opt(refString, numFrames) {
    const frames = [];
    let faults = 0;
    const steps = [];

    for (let i = 0; i < refString.length; i++) {
        const page = refString[i];
        const isFault = !frames.includes(page);
        if (isFault) {
            faults++;
            if (frames.length < numFrames) {
                frames.push(page);
            } else {
                let farthest = i;
                let replaceIndex = 0;
                for (let j = 0; j < frames.length; j++) {
                    const framePage = frames[j];
                    let k;
                    for (k = i + 1; k < refString.length; k++) {
                        if (refString[k] === framePage) break;
                    }
                    if (k === refString.length) {
                        replaceIndex = j;
                        break;
                    }
                    if (k > farthest) {
                        farthest = k;
                        replaceIndex = j;
                    }
                }
                frames[replaceIndex] = page;
            }
        }
        steps.push({
            text: `Page ${page} → Frames: [${frames.join(', ')}]${isFault ? ' (FAULT)' : ''}`,
            isFault: isFault
        });
    }
    return { faults, steps };
}
```

Figure 4: OPT algorithm snippet

This optimal algorithm code snippet requires scanning ahead in the reference string. It replaces the page that won't be used for the longest time or never used again. The implementation requires nested loops to analyze future page references, making it computer intensive but gives optimal results.

4. Results and Discussion

4.1 Sample Simulation

First Simulation

Page frames: 3

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Algorithm	Page Faults
FIFO	15
LRU	12
OPT	9

Best algorithm: OPT (9 faults)

Second Simulation

Page frames: 4

Reference String: 7, 0, 7, 9, 4, 1, 3, 0, 7, 6, 7, 6, 9, 0, 4, 1, 4, 5, 1, 2

Algorithm	Page Faults
FIFO	4
LRU	4
OPT	4

Best algorithm: FIFO (4 faults)

As shown on the two simulations above, it shows the program is working properly which compares the three page algorithms and calculates the best algorithm. The customizable reference string generation and a random generation allows more flexible input and the configurable number for page frames increases the number of possible solutions to the program.

5. Conclusion

This interactive case study successfully demonstrates the behavior and relative performance of fundamental page replacement algorithms. The web-based implementation provides an engaging way to explore algorithm behavior under different memory configurations. The development of this program allows future users and learners to visualize, interact and use it as a computational tool for page replacement algorithms.