**TARLAC STATE UNIVERSITY**

**COLLEGE OF COMPUTER STUDIES**

**Operating Systems - OS**

**2nd Semester – A.Y. 2024-2025**

***Simulation of Page-Replacement Algorithms***

***in a Virtual Memory System***

A Project

Submitted to the Faculty of the College of Computer Studies

San Isidro Campus, Tarlac City,

Tarlac State University

In Partial Fulfillment of the Requirements for the Subject

Operating Systems

Academic Year 2024-2025

Submitted By:

**Nierras, Stephen Jan V.**

**Jo Anne G. Cura**

Subject Instructor

Date of submission: May 21, 2025

**Table of Contents**

**I. Project Overview3**

**II. Code Explanation4**

2.1. Libraries Used4

2.2. FIFO Algorithm4

2.3. LRU Algorithm4

2.4. OPT Algorithm5

2.5. String Generator6

2.6. Codes for the GUI6

**III. Results8**

3.1. Results8

3.2. Documentation9

**IV. Conclusion11**

**I. Project Overview**

This project is a Python-based interactive simulator designed to facilitate a deeper understanding of page replacement algorithms commonly used in operating systems. It specifically focuses on three classic strategies: **First-In-First-Out (FIFO)**, **Least Recently Used (LRU)**, and the **Optimal Replacement Algorithm**.

The simulator employs a graphical user interface (GUI) built using **Tkinter**, allowing users to interact with the program intuitively. Users can generate a random page-reference string and specify the number of memory frames to be used in the simulation. The program then applies all three algorithms to the same reference string and displays the corresponding number of **page faults** each one produces.

By enabling users to observe and compare the behavior of these algorithms under identical conditions, the project serves as an effective educational tool. It transforms abstract theoretical concepts—such as memory management, page faults, and replacement strategies—into a more tangible and engaging learning experience. This hands-on approach supports students and learners in developing a clearer and more practical understanding of how operating systems handle memory allocation and page replacement decisions.

**II. Code Explanation**

This section presents the source code of the application. The source code has five main parts, which will be briefly explained thereafter; Libraries used, the FIFO algorithm, the LRU algorithm, the OPT algorithm, and the code for the GUI.

**2.1 Libraries Used**

Below are the libraries ued for the program.

**# Libraries used**

**import tkinter as tk  
import random**

The *tkinter* library lets the program create windows, buttons, labels, text boxes, and other GUI elements that builds up the desktop application with a graphical interface. The *random* library lets the application generate random integer values for the program.

**2.2 FIFO Algorithm**

Below is the block of code for the First-In-First-Out algorithm.

**# FIFO  
def fifo(pages, frames):  
 memory = []  
 page\_faults = 0  
 for page in pages:  
 if page not in memory:  
 page\_faults += 1  
 if len(memory) >= frames:  
 memory.pop(0)  
 memory.append(page)  
 return page\_faults**

This function maintains a queue, which is the memory variable. A page fault occurs when the memory is full and if the page is not yet in memory.

**2.3 LRU Algorithm**

The following block of codes are for the Last-Recently-Used algorithm.

**# LRU  
def lru(pages, frames):  
 memory = []  
 page\_faults = 0  
 for page in pages:  
 if page not in memory:  
 page\_faults += 1  
 if len(memory) >= frames:  
 memory.pop(0)  
 else:  
 memory.remove(page)  
 memory.append(page)  
 return page\_faults**

This function keeps track of the order of the pages used. A page fault occurs whenever a page is not yet in memory. If a page is already in memory, it removes it from its old position and re-adds it at the end, also, it removes the least recently used page when the memory is full.

**2.4 OPT Algorithm**

Below is the block of code for the Optimal Page Replacement algorithm.

**# OPT  
def optimal(pages, frames):  
 memory = []  
 page\_faults = 0  
 for i in range(len(pages)):  
 page = pages[i]  
 if page not in memory:  
 page\_faults += 1  
 if len(memory) < frames:  
 memory.append(page)  
 else:  
 future = pages[i + 1:]  
 indexes = []  
 for m in memory:  
 if m in future:  
 indexes.append(future.index(m))  
 else:  
 indexes.append(float('inf'))  
 victim\_index = indexes.index(max(indexes))  
 memory[victim\_index] = page  
 return page\_faults**

This function is just a theoretical simulation of the OPT algorithm wherein it uses the knowledge of the future page references of the program. If the memory is full and a page is still needed to be processed, it will replace the pafe that won’t be used for the time period.

**2.5 String Generator**

Below is the function for the string generator.

**# To Generate REference Strings  
def generate\_reference():  
 global reference\_string  
 reference\_string = [random.randint(0, 9) for \_ in range(20)]  
 reference\_label.config(text=f"Page Reference: {reference\_string}")**

This generates 20 random integer with the help of the *random* library.

**2.6 Codes for the GUI**

Below are the lines of code to produce results and the graphical user interface of the application.

**# The following codes are for the GUI  
def run\_simulation():  
 try:  
 frames = int(frames\_entry.get())  
 if frames <= 0:  
 raise ValueError  
 except ValueError:  
 animate("⚠ Please enter a valid positive number of frames.", result\_label)  
 return  
 if reference\_string is None:  
 animate("⚠ Please generate a page reference string first.", result\_label)  
 return  
 fifo\_faults = fifo(reference\_string, frames)  
 lru\_faults = lru(reference\_string, frames)  
 opt\_faults = optimal(reference\_string, frames)  
 result\_text = (  
 f"🧠 FIFO Faults: {fifo\_faults}\n"  
 f"📚 LRU Faults: {lru\_faults}\n"  
 f"🚀 OPT Faults: {opt\_faults}"  
 )  
 animate(result\_text, result\_label)  
def animate(text, label, index=0):  
 if index <= len(text):  
 label.config(text=text[:index])  
 label.after(30, lambda: animate(text, label, index + 1))  
def styled\_button(master, text, command):  
 btn = tk.Button(master, text=text, command=command, font=FONT\_NORMAL,  
 bg=BTN\_COLOR, fg="white", activebackground=BTN\_HOVER,  
 activeforeground="white", bd=0, padx=15, pady=5, cursor="hand2")  
 return btn  
reference\_string = [random.randint(0, 9) for \_ in range(20)]  
root = tk.Tk()  
root.title("Page Replacement Simulator")  
root.geometry("650x430")  
root.configure(bg="#1e1e1e")  
FONT\_HEADER = ("Segoe UI", 20, "bold")  
FONT\_NORMAL = ("Segoe UI", 12)  
FONT\_RESULT = ("Consolas", 13)  
FG\_PRIMARY = "#f0f0f0"  
FG\_SECONDARY = "#a9a9a9"  
BG\_DARK = "#1e1e1e"  
BG\_ACCENT = "#333"  
BTN\_COLOR = "#3a7bd5"  
BTN\_HOVER = "#2b5fa2"  
tk.Label(root, text="Page Replacement Simulator", font=FONT\_HEADER, bg=BG\_DARK, fg=FG\_PRIMARY).pack(pady=15)  
frame\_input = tk.Frame(root, bg=BG\_DARK)  
frame\_input.pack(pady=10)  
tk.Label(frame\_input, text="Number of Frames:", font=FONT\_NORMAL, bg=BG\_DARK, fg=FG\_SECONDARY).pack(side=tk.LEFT, padx=5)  
frames\_entry = tk.Entry(frame\_input, width=5, font=FONT\_NORMAL, bg=BG\_ACCENT, fg=FG\_PRIMARY, insertbackground="white", justify="center", bd=0)  
frames\_entry.pack(side=tk.LEFT, padx=5)  
styled\_button(frame\_input, "🎲 Generate String", generate\_reference).pack(side=tk.LEFT, padx=10)  
reference\_label = tk.Label(root, text=f"Page Reference: {reference\_string}", font=FONT\_NORMAL, bg=BG\_DARK, fg=FG\_PRIMARY)  
reference\_label.pack(pady=10)  
styled\_button(root, "▶ Run Simulation", run\_simulation).pack(pady=10)  
result\_label = tk.Label(root, text="", font=FONT\_RESULT, bg=BG\_DARK, fg=FG\_PRIMARY, justify="left", anchor="w")  
result\_label.pack(pady=15)  
root.mainloop()**

These lines of code for the GUI are not yet mastered by the project owner, and are, thus, assumed to be produced by an artificially intelligent chatbot. Hehe.

**III. Results**

This section presents the results obtained from running the simulator with randomly generated reference strings across varying frame sizes. For each test case, the number of page faults for the FIFO, LRU, and OPT algorithms was recorded and compared.

**3.1. Results**

Below are three sample runs of the program using different reference strings and frame sizes. The page faults produced by each algorithm are as follows:

**Run 1**

* Reference String: [1, 8, 1, 1, 4, 9, 2, 3, 6, 1, 7, 7, 4, 3, 5, 6, 3, 2, 8, 4]
* Frames: 5
* FIFO: 14 faults
* LRU: 15 faults
* OPT: 10 faults

**Run 2**

* Reference String: [1, 8, 1, 1, 4, 9, 2, 3, 6, 1, 7, 7, 4, 3, 5, 6, 3, 2, 8, 4]
* Frames: 7
* FIFO: 11 faults
* LRU: 10 faults
* OPT: 9 faults

**Run 3**

* Reference String: [3, 1, 0, 9, 3, 3, 7, 5, 3, 2, 5, 3, 4, 9, 6, 3, 6, 0, 6, 4]
* Frames: 3
* FIFO: 15 faults
* LRU: 14 faults
* OPT: 11 faults

Upon analyzing the results, the Optimal algorithm consistently produced the fewest page faults, as expected, since it uses future knowledge to make the best possible replacement decisions. The LRU algorithm followed closely, performing better than FIFO in most cases due to its ability to adapt to recent usage patterns and retain pages likely to be used again soon. In contrast, the FIFO algorithm, while the simplest to implement, generally resulted in the most page faults because it removes the oldest page without considering its future use, making it less efficient in scenarios where pages are frequently reused.

**3.2. Documentation**

The images below are the documentation of the three sample runs mentioned on chapter 3.1.

**Run 1**

A screenshot of a computer

AI-generated content may be incorrect.

**Run 2**

A screenshot of a computer

AI-generated content may be incorrect.

**Run 3**

A screenshot of a computer

AI-generated content may be incorrect.

The images show the graphical user interface of the application, which features the generated page references and the showcase of the number of faults for each algorithm.

**IV.** **Conclusion**

This project successfully demonstrated the working of three fundamental page replacement algorithms through an interactive and user-friendly Python application. By allowing users to generate random reference strings and simulate algorithm performance with different frame sizes, the project helped bridge the gap between theoretical understanding and practical application.

From the results, it is evident that **Optimal Replacement** yields the least number of page faults. However, since it relies on future knowledge, it is impractical for real-time systems. **LRU**, though more complex to implement than FIFO, proves to be a better approximation of optimal behavior in real-world scenarios. **FIFO**, while simple and easy to implement, does not consider page usage frequency or recency, often leading to inefficient memory management.

It was also observed that increasing the number of frames generally leads to fewer page faults, especially in algorithms like LRU and OPT. This highlights the importance of memory size in system performance. If provided more time and resources, this simulator could be further enhanced by integrating additional algorithms such as **Clock** or **Second Chance**, allowing manual entry of reference strings, and including detailed graphical representations of memory state transitions to enrich the learning experience.