ACADEMIC TASK – 2

Topic: VIRTUAL MEMORY MANAGEMENT TOOL

Subject: Operating System (CSE-316)

Submitted by:

Name: J V Purushotham

Registration no: 12303116

Roll No: 09

Section: K23TG

Group Members:

Registration No:	Name:
12306606	B Pranathi Sri (08)
12303116	J V Purushotham (09)
12301670	K Hari Krishna (10)

Submitted to: Mrs. Akash Pundir sir

1. Project Overview:

Virtual Memory Management Visualization Tool is an interactive learning tool to assist students in comprehending basic memory management concepts in operating systems. It models basic techniques like paging and segmentation, visually demonstrating memory allocation, page faults, and impacts of different page replacement algorithms in real time. Parameters of the memory can be set by the user and experiments tried with varying inputs to see how system performance would be affected based on user settings.

This tool is a useful learning tool for students, teachers, and hobbyists who want to learn about virtual memory operations and concepts by getting hands-on experience. Users will be able to experiment with different memory configurations and page replacement algorithms like Least Recently Used (LRU) and Optimal Page Replacement to observe their impact on system performance.

Objective:

- Visualize Paging and Segmentation: Enable dynamic, real-time visualization of memory allocation processes, demonstrating both internal and external fragmentation under various schemes.
- Simulate Page Faults & Demand Paging: Demonstrate the demand paging concept and emphasize the occurrence and recovery of page faults when a program runs.
- Facilitate Interactive User Inputs: Enable users to specify own memory configurations like memory size, segment size, page frame number, and reference strings to customize simulations.
- Implement and Compare Page Replacement Algorithms: Support several page replacement methods, such as Least Recently Used (LRU) and Optimal Page Replacement, to compare their efficiency and performance.
- Analyze Memory Fragmentation: Show and compare internal and external fragmentation in different memory allocation methods and provide insights into their effect on system performance.

2. Module-Wise Breakdown:

1. User Input Module
- Purpose: Accept and validate user-defined configurations.
- Functionality:
  - Input memory size, segment size, and page frame count.
  - Enter reference strings for memory access.
  - Choose memory management strategy (Paging or Segmentation).
  - Select page replacement algorithm (LRU or Optimal).

2. Memory Segmentation Module
- Purpose: Simulate memory allocation using segmentation.
- Functionality:
  - Allocate and deallocate memory segments.
  - Visualize segment placement in memory.
  - Show external fragmentation dynamically.

3. Paging Module
- Purpose: Simulate paging-based memory management.
- Functionality:
  - Implement demand paging mechanism.
  - Handle page requests using user-defined reference strings.
  - Simulate and track page faults.
  - Apply selected page replacement algorithm (LRU/Optimal).

4. Page Replacement Algorithms Module
- Purpose: Manage page replacement based on selected strategy.
- Functionality:
  - Implement Least Recently Used (LRU) algorithm.
  - Implement Optimal Page Replacement algorithm.

- Maintain and update page frames accordingly.

5. Visualization Module

- Purpose: Provide visual feedback and understanding.
- Functionality:
  - Display memory structure and allocation in real-time.
  - Animate memory accesses, page faults, and segment allocation.
  - Show page table updates and frame status.

6. Fragmentation Analysis Module

- Purpose: Analyse and display memory fragmentation.
- Functionality:
  - Calculate internal fragmentation in paging.
  - Calculate external fragmentation in segmentation.
  - Provide graphical/statistical output for analysis.

7. Evaluation & Summary Module

- Purpose: Summarize results and performance metrics.
- Functionality:
  - Display total page faults encountered.
  - Evaluate the performance of page replacement strategies.
  - Provide a summary of fragmentation and allocation efficiency.

8. Reset & Replay Module

- Purpose: Reset the simulation and allow experimentation.
- Functionality:
  - Clear all inputs and visualizations.
  - Allow users to reconfigure settings and rerun simulations.

3. Key Functionalities:

1. Simulation of Dynamic Memory Allocation

- Enables users to simulate dynamic memory allocation based on paging or segmentation.
- Graphic illustration of how memory is separated and allocated into pages or segments as specified by the user.

2. Page Fault Detection and Handling

- Detects and shows page faults automatically during memory access.
- Simulates the behaviour of the system when it handles page faults in demand paging.

3. Comparison of Page Replacement Algorithms

- Supports two principal page replacement algorithms:
  - Least Recently Used (LRU)
  - Optimal Page Replacement
- Allows users to toggle between algorithms and see their impact on page fault rates and memory performance.

4. Real-Time Visualization of Memory

- Displays interactive, real-time graphical visualization of:
  - Memory blocks and their status
  - Page tables and segment tables
  - Frame allocation and changes
  - Fragmentation (internal/external)

5. Performance Analysis and Metrics

- Shows primary performance indicators such as:
  - Page fault count and rate
  - Memory utilization efficiency
  - Fragmentation statistics
- Assists users in assessing the efficacy of varying memory management approaches.

4. Technology Used:

Programming Language

- **Python** – Used as the core programming language for implementing the simulation logic and GUI.

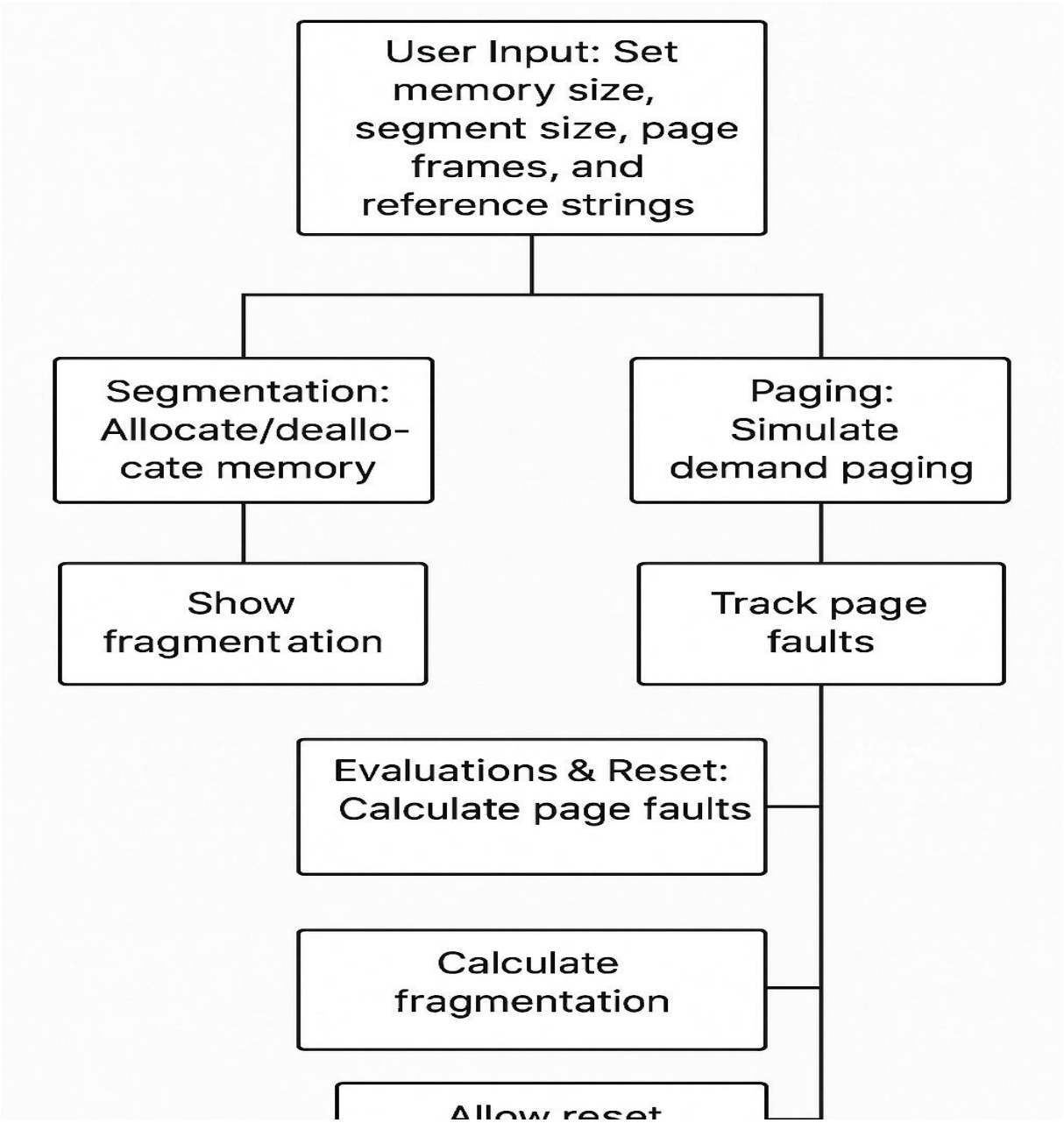
Libraries and Tools

- **PyQt5**
  - Utilized for developing the Graphical User Interface (GUI).
  - Provides essential components for creating interactive windows, buttons, labels, and custom graphics views.
- **sys**
  - Used for handling system-level interactions and application control flow.
- **QtWidgets** (from PyQt5)
  - Offers a wide range of UI components such as:
    - QMainWindow
    - QPushButton
    - QLabel
    - QGraphicsView
- **QtGui** (from PyQt5)
  - Supplies graphical elements like:
    - QColor for defining color properties.
    - QBrush for rendering filled graphical shapes.

5. Other Components:

- **Memory Allocation**
  - Accommodates a variety of strategies like First-Fit, Best-Fit, Worst-Fit, Dynamic. Allocation, and Compaction to showcase how memory is allocated under different schemes.
- **Page Replacement**
  - Introduces prominent page replacement techniques like FIFO, Clock Algorithm, Random Replacement, LRU, and Optimal, enabling users to compare their performance.
- **Custom Simulations**
  - Facilitates simulation of user-defined workloads, OS behaviour simulation, and batch processing examples for thorough understanding and experimentation.
- **Monitoring & Metrics**
  - Offers real-time monitoring of process states, memory consumption, and performance metrics through dynamic graphs and visual markers.
- **Interactive GUI**
  - Includes drag-and-drop process control, color-coded memory blocks, and step-by-step execution to facilitate user interaction and learning.
- **Paging & TLB**
  - Models sophisticated memory methods like multi-level paging and the utilization of a Translation Lookaside Buffer (TLB) for address translation.
- **Export & Reports**
  - Includes supports for saving and loading simulations, exporting memory states, and building analytical reports to document and analyse.
- **Support for Multi-Threading**
  - Models concurrent memory accesses and allocations to demonstrate the effects of multi-threading and concurrent handling in live operating systems.

6. Flow Diagram:



7. GitHub Revision Tracking:

To keep things transparent and to maintain consistent progress, version control and all project updates have been tracked via GitHub.

- **Repository Name:** Operating System Project
- **GitHub Link:** https://github.com/jvpurushotham/Operating_System_Project/tree/main

Overview of Commit History:

Project Setup: Set up the basic project directory structure and required files.
Paging & Segmentation: Coded basic visualization logic for memory segmentation and paging.
Page Faults & Demand Paging: Coded functionality to simulate demand paging and page faults.
Page Replacement Algorithms: Integrated major algorithms such as LRU, Optimal, FIFO, and

Clock.
Memory Allocation & Fragmentation: Facilitated user-defined handling of input and visualization of fragmentation.
GUI & Performance Metrics: Created a clear user interface and incorporated real-time stats monitoring.
Multi-Level Paging & Optimization: Improved simulation through the addition of multi-level paging and Translation Lookaside Buffer (TLB) processing for optimized performance.
Final Enhancements: Executed bug fixes, provided session-saving functions, and completed the project to release.

8. Conclusion

The **Virtual Memory Management Visualization Tool** successfully mimics and displays fundamental memory management principles such as paging, segmentation, page faults, demand paging, and various page replacement algorithms. With its interactive and user-friendly simulations, the tool enables users to:

- Visualize dynamic memory allocation,
- Understand internal and external fragmentation, and
- Compare the performance of different page replacement techniques like **Least Recently Used (LRU)** and **Optimal Replacement**.

This project serves as a robust educational tool for **students**, **system developers**, and **researchers**, providing hands-on, experiential learning to understand how memory is efficiently managed in operating systems.

Future Scope

1. **Incorporation of Additional Algorithms**
   Add support for advanced page replacement techniques such as NFU (Not Frequently Used) and the Working Set Model to enable comprehensive performance comparisons.

2. **GUI Enhancements & Real-Time Animations**
   Upgrade the graphical interface with smoother animations and an enriched user experience to boost interactivity and learning effectiveness.

3. **Multi-Threading & Concurrency Simulation**
   Simulate multi-threaded environments to demonstrate the behaviour of memory allocation and access under concurrent execution conditions.

4. **Integration with Real Operating Systems**
   Simulate the memory behaviour of popular operating systems (e.g., Linux, Windows) to bridge the gap between theoretical concepts and real-world applications.

5. **Cloud-Based Deployment**
   Develop a web-based version of the tool to enhance accessibility for remote learners and support collaborative simulations across different locations.

6. **AI-Based Optimization**
   Incorporate machine learning algorithms to analyse workload patterns and suggest optimal memory allocation strategies dynamically.

By implementing these future enhancements, the tool can evolve into a comprehensive, intelligent, and scalable platform—offering both academic and practical value in the domain of virtual memory management and operating systems.

9. References:

- **Operating System Concepts: Silberschatz**, Galvin, and Gagne (For virtual memory, paging, and segmentation concepts).
- **Modern Operating Systems**: Andrew S. Tanenbaum (For memory management techniques and page replacement algorithms).
- **Computer Organization and Design**: David A. Patterson & John L. Hennessy (For understanding memory hierarchy and TLB).
- **Linux Kernel Documentation**: Official Linux resources (For demand paging and memory management in real-world OS).

Appendix

AI-Generated Project Elaboration/Breakdown Report

Project Overview: A web-based Virtual Memory Management Visualization Tool that allows users to simulate and analyze paging, segmentation, memory allocation, and page replacement algorithms in an interactive environment**.**

Technology Stack: Developed using HTML, CSS, vanilla JavaScript for the front end and JavaScript-based data structures for memory simulation and management.

Memory Simulation: The system simulates virtual memory allocation, including paging, segmentation, demand paging, and page faults, ensuring an interactive learning experience.

Page Replacement Mechanism: Implements LRU, Optimal, FIFO, and Clock algorithms, allowing users to compare their efficiency and impact on memory performance.

User Interface: A graphical UI with a light theme, enabling users to visualize memory allocation, track page faults, and analyze memory fragmentation in real-time.

Automation: A step-by-step execution mode allows users to pause and analyze each stage of memory allocation and replacement, ensuring a clear understanding of memory dynamics.

Error Handling: Ensures accurate visualization of page faults, invalid memory accesses, and memory fragmentation, with appropriate UI updates**.**

Enhancements: Can be extended with multi-threading simulation, real-time OS memory behavior analysis, and AI-driven memory optimization.

Security Considerations: Future improvements include access control, secure data handling, and cloud-based simulations for broader accessibility**.**

Final Outcome: A fully interactive virtual memory simulation tool, providing valuable insights into memory management, page replacement strategies, and fragmentation handling in operating systems**.**

Project Breakdown

1. Core Features

- **Paging Simulation:** Supports LRU & Optimal page replacement, tracks page faults, and visualizes page frames dynamically.
- **Segmentation Simulation:** Allows memory allocation/deallocation, tracks fragmentation, and updates visuals in real-time.

2. Technology Stack

- **Frontend:** PyQt5 (QTabWidget, QGraphicsView, QPushButton, QLabel).
- **Backend:** Python-based **data structures** for memory management.

3. User Interaction

- **Paging:** Set frames, reference string, and algorithm.
- **Segmentation:** Define memory size & segment size dynamically.
- **Controls:** Start, Step, Reset, Allocate, Deallocate buttons with status bar updates.

4. Automation & Error Handling

- Real-time visualization for paging & segmentation.
- Prevents invalid memory operations & input errors.

5. Future Enhancements

- More page replacement algorithms (FIFO, NFU).
- Multi-level paging & TLB simulation.
- Cloud-based & AI-driven memory optimization.

10. Problem Statement:

Description In modern operating systems, virtual memory management plays a crucial role in optimizing memory usage and ensuring efficient process execution. However, understanding its internal mechanisms—such as paging, segmentation, page faults, and demand paging—can be complex.

This project aims to develop a visual memory management tool that allows users to experiment with memory allocation, observe page replacement strategies (LRU & Optimal), and analyze memory fragmentation dynamically. The tool provides an interactive visualization of memory allocation, helping users understand how operating systems manage memory efficiently and handle page faults and fragmentation.

By simulating real-world memory management behavior, this tool will serve as an educational resource for students and professionals, aiding in better comprehension of paging, segmentation, and page replacement algorithms.

7. Code of this project:

```
from PyQt5.QtGui import QColor, QBrush
import sys
from PyQt5.QtWidgets import (
    QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout, QLabel, QPushButton,
    QComboBox, QSpinBox, QLineEdit, QTabWidget, QGraphicsView, QGraphicsScene,
    QGraphicsRectItem, QGraphicsTextItem, QStatusBar
)

# Main application window
class MemoryVisualizer(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Memory Management Visualizer")  # Window title
        self.setGeometry(100, 100, 1000, 700)  # Window size and position

        main_widget = QWidget()
        self.setCentralWidget(main_widget)
        main_layout = QVBoxLayout(main_widget)
```

```
        self.tabs = QTabWidget()  # Tabbed layout
        main_layout.addWidget(self.tabs)

        self.paging_tab = QWidget()  # Paging tab
        self.segmentation_tab = QWidget()  # Segmentation tab
        self.tabs.addTab(self.paging_tab, "Paging")
        self.tabs.addTab(self.segmentation_tab, "Segmentation")

        self.statusBar = QStatusBar()  # Status bar
        self.setStatusBar(self.statusBar)
        self.statusBar.showMessage("Ready")

        self.setup_paging_tab()  # Setup paging UI
        self.setup_segmentation_tab()  # Setup segmentation UI

    # Setup for paging
    def setup_paging_tab(self):
        layout = QVBoxLayout(self.paging_tab)
        control_layout = QHBoxLayout()

        control_layout.addWidget(QLabel("Algorithm:"))
        self.algorithm_combo = QComboBox()  # Algorithm selection
        self.algorithm_combo.addItems(["LRU", "Optimal"])
        control_layout.addWidget(self.algorithm_combo)

        control_layout.addWidget(QLabel("Frames:"))
        self.frame_spin = QSpinBox()  # Frame size input
        self.frame_spin.setRange(1, 20)
        self.frame_spin.setValue(4)
        control_layout.addWidget(self.frame_spin)

        control_layout.addWidget(QLabel("Reference String:"))
        self.ref_string_input = QLineEdit()  # Reference string input
        self.ref_string_input.setPlaceholderText("e.g. 1,2,3,4,1,2,5")
        control_layout.addWidget(self.ref_string_input)

        self.start_btn = QPushButton("Start")  # Start button
        self.step_btn = QPushButton("Step")  # Step button
        self.reset_btn = QPushButton("Reset")  # Reset button
        control_layout.addWidget(self.start_btn)
        control_layout.addWidget(self.step_btn)
        control_layout.addWidget(self.reset_btn)

        layout.addLayout(control_layout)

        self.paging_view = QGraphicsView()  # Visual display
        self.paging_scene = QGraphicsScene()
        self.paging_view.setScene(self.paging_scene)
        layout.addWidget(self.paging_view)

        self.stats_label = QLabel("Page Faults: 0")  # Faults display
        layout.addWidget(self.stats_label)

        self.frames = []  # Frame storage
```

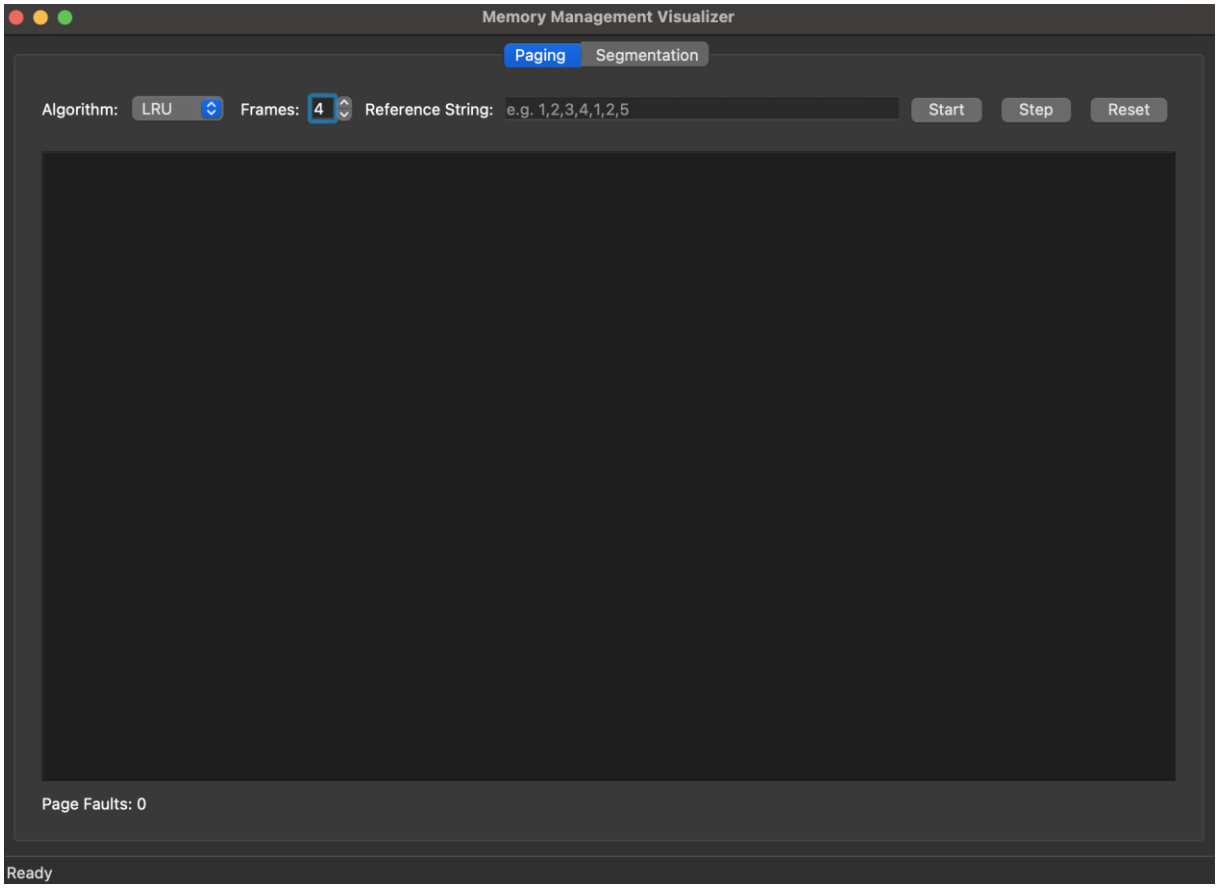
```
        self.ref_list = []  # Reference string list
        self.current_index = 0
        self.page_faults = 0

        self.start_btn.clicked.connect(self.start_paging)
        self.step_btn.clicked.connect(self.step_paging)
        self.reset_btn.clicked.connect(self.reset_paging)

    def start_paging(self):
        self.frames.clear()
        self.page_faults = 0
        self.current_index = 0
        self.paging_scene.clear()

        ref_string = self.ref_string_input.text().strip()
        if not ref_string:
            self.statusBar.showMessage("Error: Enter a reference string!")
            return

        try:
            self.ref_list = [int(x.strip()) for x in ref_string.split(',')]  # Parse input
        except ValueError:
            self.statusBar.showMessage("Error: Invalid input!")
            return

        self.statusBar.showMessage("Paging started!")
        self.step_paging()  # Start first step

    def step_paging(self):
        if self.current_index >= len(self.ref_list):  # End of string
            self.statusBar.showMessage("All references processed.")
            return

        page = self.ref_list[self.current_index]
        algorithm = self.algorithm_combo.currentText()
        frame_limit = self.frame_spin.value()

        if page not in self.frames:
            self.page_faults += 1  # Fault occurred
            if len(self.frames) < frame_limit:
                self.frames.append(page)
            else:
                if algorithm == "LRU":
                    self.frames.pop(0)  # Remove LRU page
                elif algorithm == "Optimal":
                    future = self.ref_list[self.current_index+1:]
                    indices = [future.index(f) if f in future else float('inf') for f in self.frames]
                    to_remove = indices.index(max(indices))  # Optimal replacement
                    self.frames.pop(to_remove)
                self.frames.append(page)
        else:
            if algorithm == "LRU":
                self.frames.remove(page)
                self.frames.append(page)  # Update LRU
```

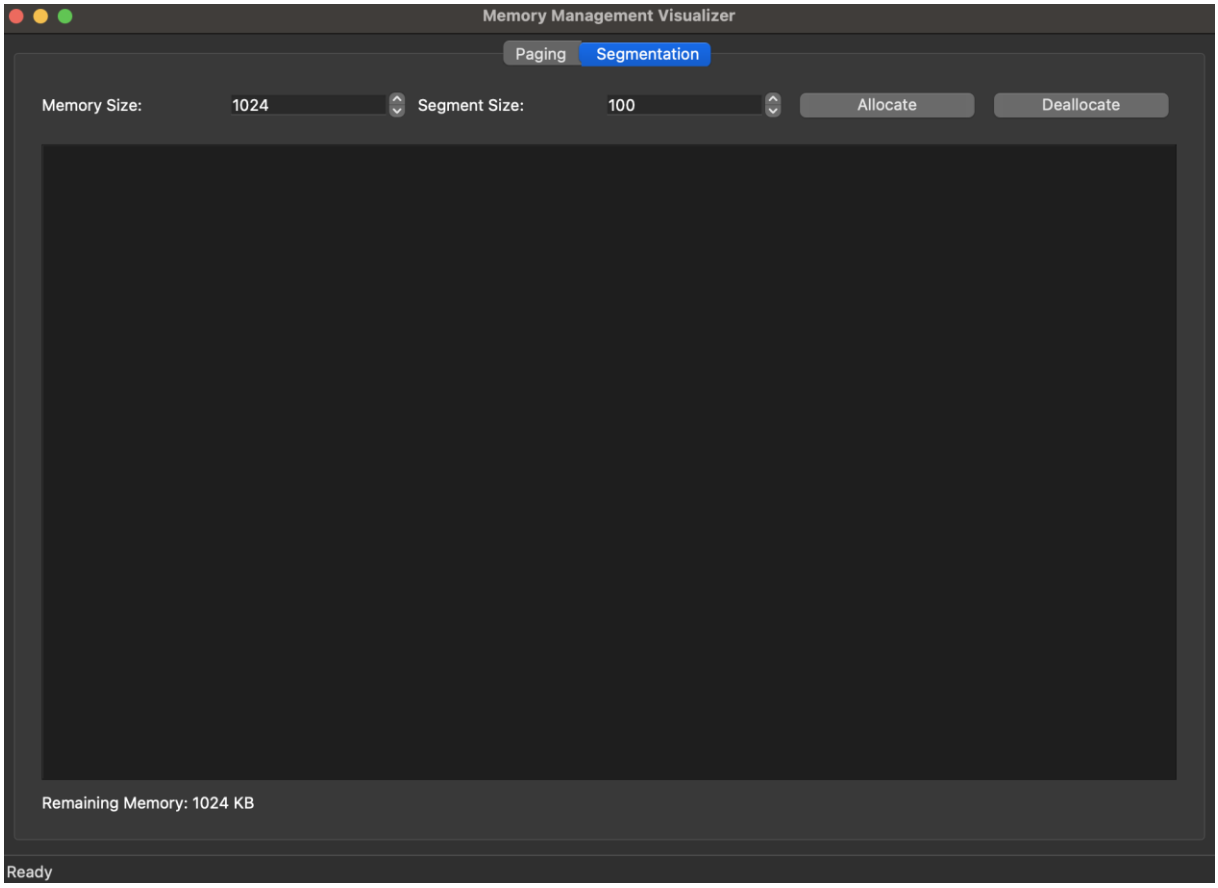
```
        self.current_index += 1
        self.stats_label.setText(f"Page Faults: {self.page_faults}")
        self.visualize_paging()  # Refresh visuals

    def reset_paging(self):
        self.frames.clear()
        self.ref_list.clear()
        self.page_faults = 0
        self.current_index = 0
        self.paging_scene.clear()
        self.stats_label.setText("Page Faults: 0")
        self.statusBar.showMessage("Paging reset!")

    def visualize_paging(self):
        self.paging_scene.clear()
        self.paging_scene.setBackgroundBrush(QBrush(QColor("#1e1e2f")))  # Background color

        for i, frame in enumerate(self.frames):
            rect = QGraphicsRectItem(50, i * 50, 100, 50)  # Frame block
            rect.setBrush(QBrush(QColor("#4CAF50")))
            self.paging_scene.addItem(rect)
            text = QGraphicsTextItem(str(frame))  # Frame text
            text.setPos(85, i * 50 + 12)
            self.paging_scene.addItem(text)

    # Setup for segmentation
    def setup_segmentation_tab(self):
        layout = QVBoxLayout(self.segmentation_tab)
        control_layout = QHBoxLayout()

        control_layout.addWidget(QLabel("Memory Size:"))
        self.mem_size_spin = QSpinBox()  # Total memory input
        self.mem_size_spin.setRange(100, 10000)
        self.mem_size_spin.setValue(1024)
        control_layout.addWidget(self.mem_size_spin)

        control_layout.addWidget(QLabel("Segment Size:"))
        self.seg_size_spin = QSpinBox()  # Segment size input
        self.seg_size_spin.setRange(10, 500)
        self.seg_size_spin.setValue(100)
        control_layout.addWidget(self.seg_size_spin)

        self.alloc_btn = QPushButton("Allocate")  # Allocate memory
        self.dealloc_btn = QPushButton("Deallocate")  # Deallocate memory
        control_layout.addWidget(self.alloc_btn)
        control_layout.addWidget(self.dealloc_btn)

        layout.addLayout(control_layout)

        self.seg_view = QGraphicsView()  # Segmentation view
        self.seg_scene = QGraphicsScene()
        self.seg_view.setScene(self.seg_scene)
        layout.addWidget(self.seg_view)

        self.frag_label = QLabel("Remaining Memory: 1024 KB")  # Memory remaining
```

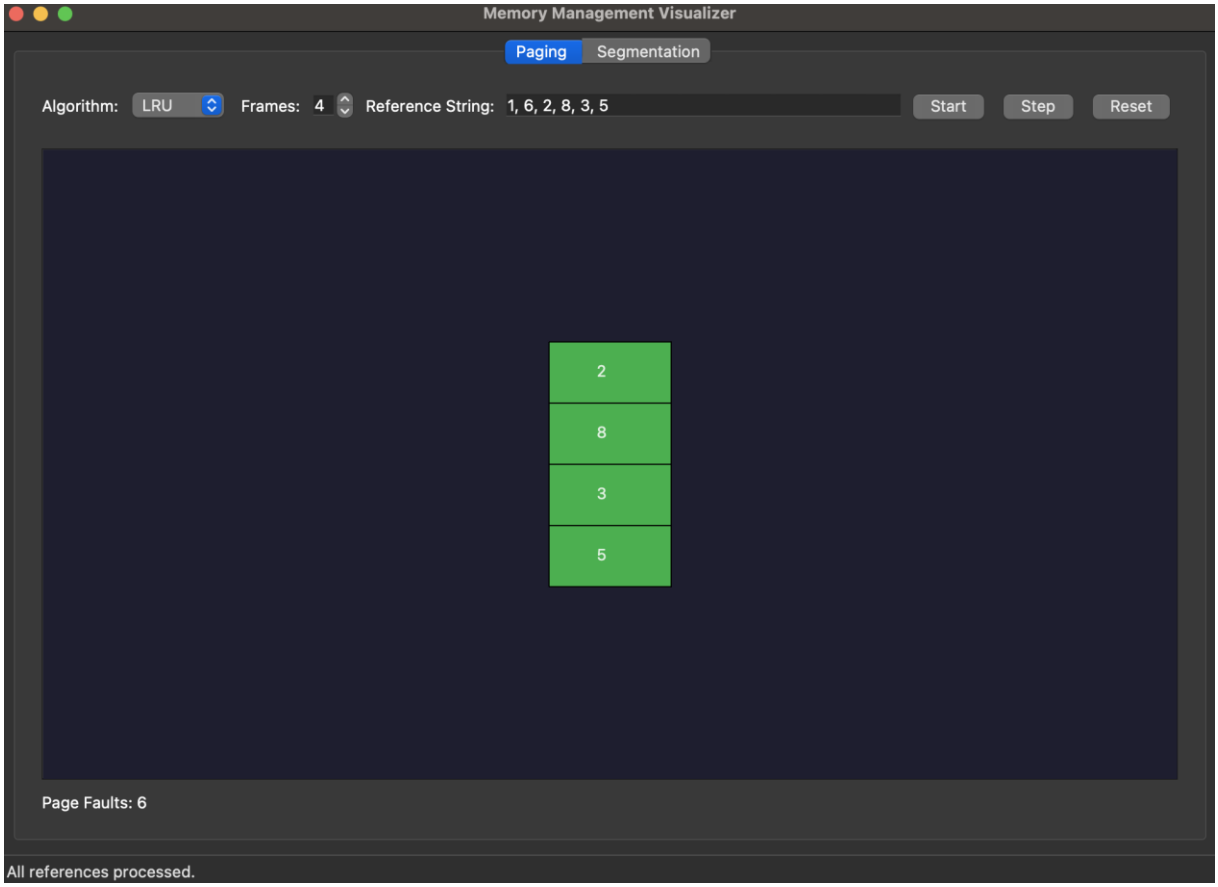
```
        layout.addWidget(self.frag_label)

        self.remaining_memory = self.mem_size_spin.value()
        self.allocated_blocks = []  # Store segments

        self.alloc_btn.clicked.connect(self.allocate_memory)
        self.dealloc_btn.clicked.connect(self.deallocate_memory)
        self.mem_size_spin.valueChanged.connect(self.reset_memory)

    def allocate_memory(self):
        segment_size = self.seg_size_spin.value()
        if segment_size <= self.remaining_memory:
            self.allocated_blocks.append(segment_size)  # Allocate segment
            self.remaining_memory -= segment_size
            self.update_segmentation_visuals()
            self.statusBar.showMessage(f"Allocated {segment_size} KB. Remaining:
{self.remaining_memory} KB")
        else:
            self.statusBar.showMessage("Error: Not enough memory available!")

    def deallocate_memory(self):
        if self.allocated_blocks:
            freed_size = self.allocated_blocks.pop()  # Deallocate last
            self.remaining_memory += freed_size
            self.update_segmentation_visuals()
            self.statusBar.showMessage(f"Deallocated {freed_size} KB. Remaining:
{self.remaining_memory} KB")
        else:
            self.statusBar.showMessage("Error: No segments to deallocate!")

    def reset_memory(self):
        self.allocated_blocks.clear()  # Clear all segments
        self.remaining_memory = self.mem_size_spin.value()
        self.update_segmentation_visuals()
        self.statusBar.showMessage("Memory reset!")

    def update_segmentation_visuals(self):
        self.seg_scene.clear()
        self.seg_scene.setBackgroundBrush(QBrush(QColor("#1b263b")))  # Background color
        total_memory = self.mem_size_spin.value()
        view_height = 400
        scale_factor = view_height / total_memory
        y_offset = 0
        colors = ["#e63946", "#a8dadc", "#457b9d", "#f36f53", "#ffb703"]  # Segment colors

        for i, segment in enumerate(self.allocated_blocks):
            scaled_height = max(segment * scale_factor, 5)  # Scale block
            rect = QGraphicsRectItem(50, y_offset, 200, scaled_height)
            rect.setBrush(QBrush(QColor(colors[i % len(colors)])))
            self.seg_scene.addItem(rect)

            text = QGraphicsTextItem(f"{segment} KB")  # Segment label
            text.setDefaultTextColor(QColor("#ffffff"))
            text.setPos(125, y_offset + scaled_height / 4)
            self.seg_scene.addItem(text)
```

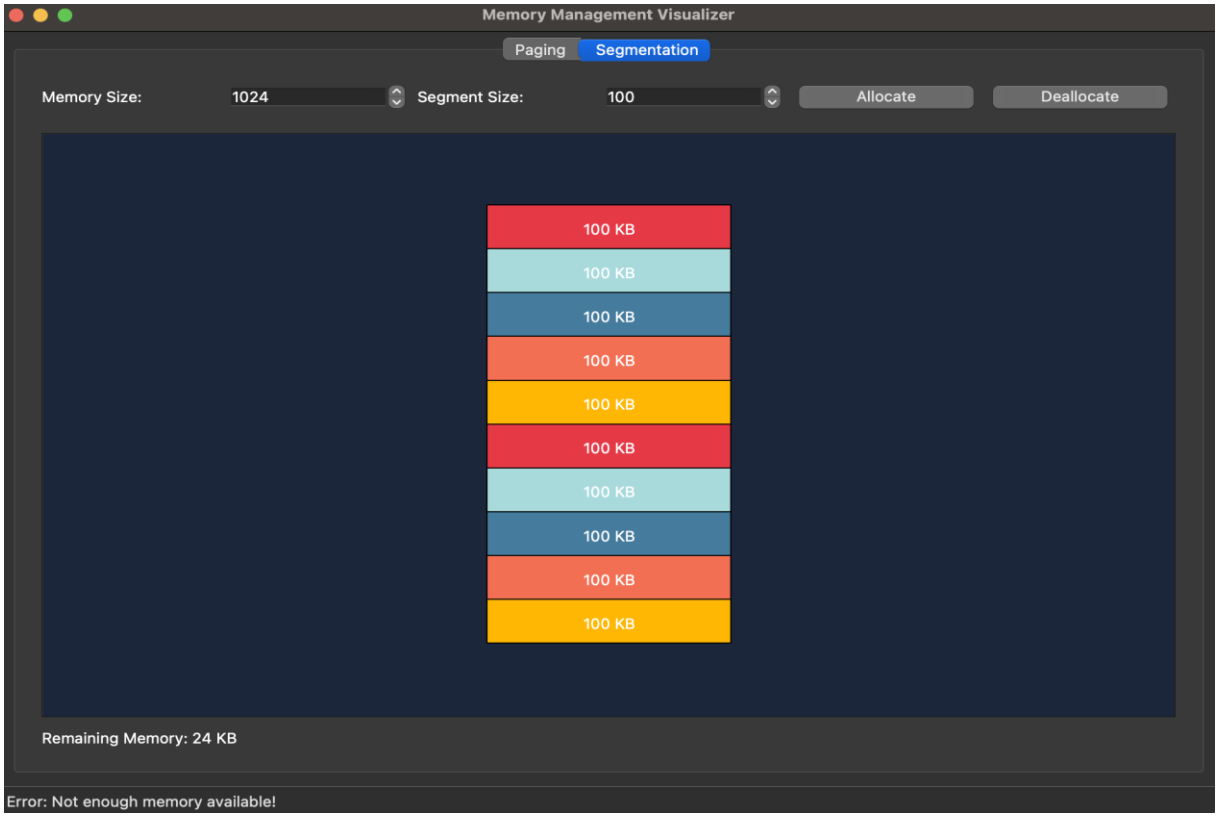
```
            y_offset += scaled_height

        self.frag_label.setText(f"Remaining Memory: {self.remaining_memory} KB")

# Entry point
if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MemoryVisualizer()
    window.show()
    sys.exit(app.exec_())
```

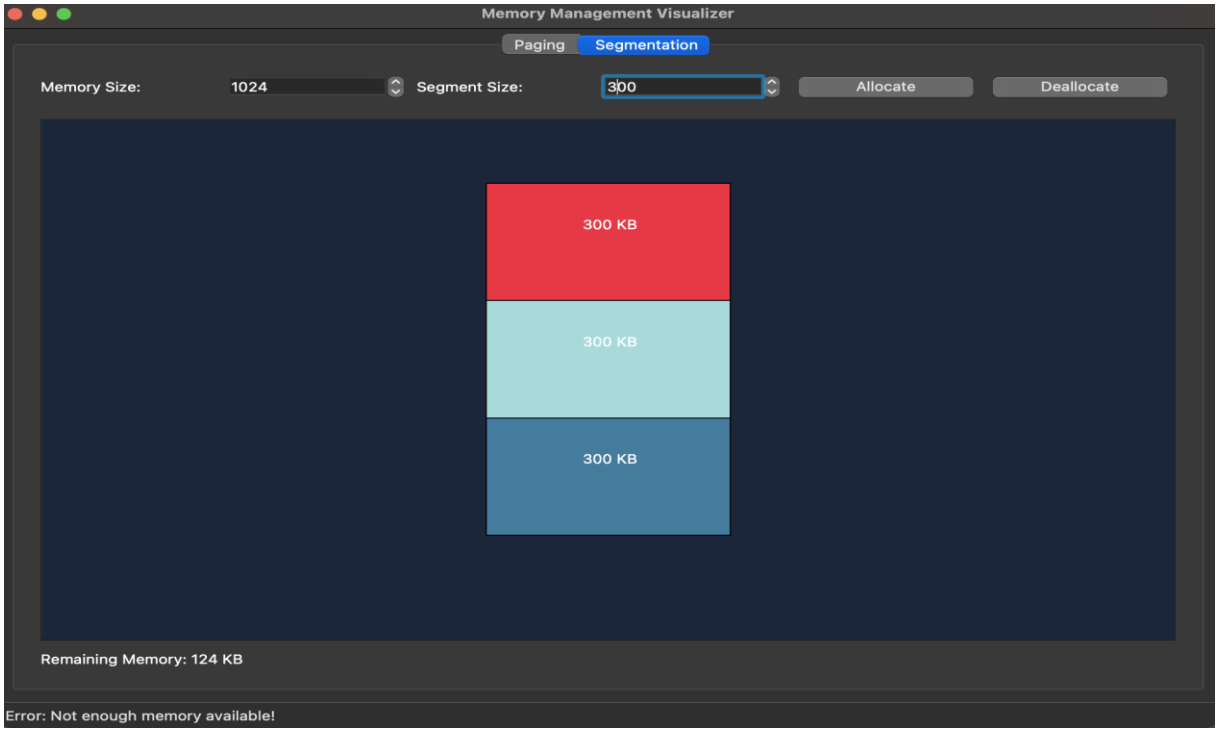
Output Screenshots:

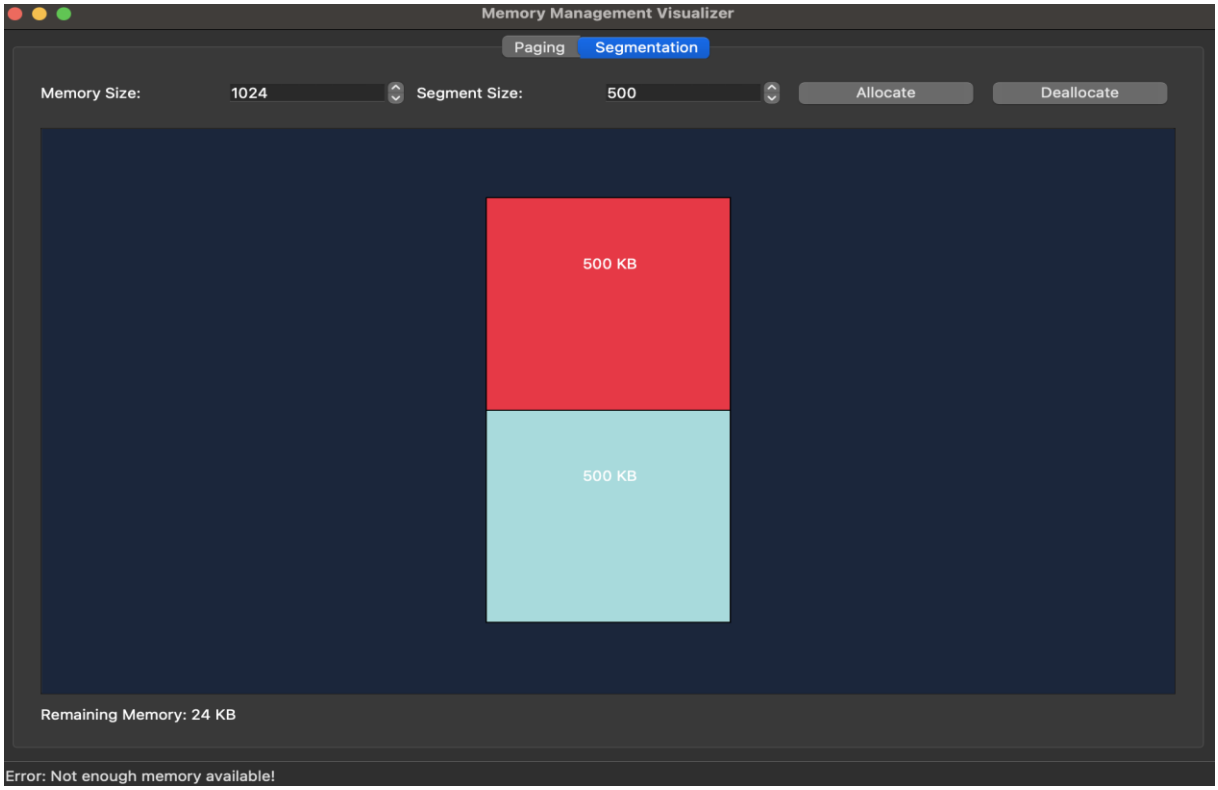



Memory Management Visualizer
Paging
Segmentation
Memory Size:
1024
Segment Size:
100
Allocate
Deallocate
Remaining Memory: 1024 KB
Ready


Memory Management Visualizer
Paging
Segmentation
Algorithm:
LRU
Frames:
4
Reference String:
1, 6, 2, 8, 3, 5
Start
Step
Reset
2
8
3
5
Page Faults: 6
All references processed.


Memory Management Visualizer
Paging
Segmentation
Memory Size:
1024
Segment Size:
100
Allocate
Deallocate
100 KB
100 KB
100 KB
100 KB
100 KB
100 KB
100 KB
100 KB
100 KB
100 KB
Remaining Memory: 24 KB
Error: Not enough memory available!


Memory Management Visualizer
Paging
Segmentation
Memory Size:
1024
Segment Size:
300
Allocate
Deallocate
300 KB
300 KB
300 KB
Remaining Memory: 124 KB
Error: Not enough memory available!


Memory Management Visualizer
Paging
Segmentation
Memory Size:
1024
Segment Size:
500
Allocate
Deallocate
500 KB
500 KB
Remaining Memory: 24 KB
Error: Not enough memory available!