CPS 3250

Project Report

Virtual Memory Explorer:

An Interactive Educational and Performance Analysis Platform

Kaiyu Liu	**1195058**	liukai@kean.edu
Fan Yu	**1195235**	yufa@kean.edu
Yiyang Hu	**1194116**	huyiya@kean.edu
Lei Xia	**1195180**	xial@kean.edu

Overview:

This project is a JavaFX application that simulates a virtual memory system for educational purposes. It features a user-friendly graphical interface that allows users to see how memory management works in an operating system through various interactive activities.

Key Features:

1. Memory Grid Visualization: This part of the application shows a visual representation of memory where users can see different processes and their pages.
2. Process Management: Users can create new processes and remove them, which helps to show how memory is allocated and then cleared.
3. Page Replacement Algorithms: The simulator includes several common algorithms for replacing pages in memory, which helps users understand how they operate in a real system.
4. Interactive GUI: The interface is straightforward, making it easy for users to interact with the system and see the results of their actions on memory in real-time.

Project Objective and Documentation:

The main purpose of this project is to teach and provide insight into how virtual memory systems work. The interactive elements and visual setup aim to make learning about operating system memory management clear and engaging for students and anyone interested in this area.

Code Explanation:

1. Main.java

- **Purpose**: Serves as the entry point of the application.
- **Overview**: The **main** method checks the Java version and launches the application. It ensures compatibility with Java 11 or higher, indicating the application's reliance on newer Java features, possibly JavaFX for the GUI.

2. Memory.java

- **Purpose**: Simulates the memory class.
- **Overview**: This class initializes the memory with a specified number of frames, represented by the **Frame** class. It appears to manage memory allocation at the frame level, crucial for simulating memory management.

3. MemorySimulatorGUI.java

- **Purpose**: Manages the graphical user interface of the application.
- **Overview**: This file uses JavaFX imports, indicating a GUI with various controls and layout management. It likely includes visual elements like buttons, text fields, and visualization areas to interact with and display the state of the virtual memory.

4. OS.java

- **Purpose**: Represents the operating system's role in memory management.
- **Overview**: Defines constants like memory size, page size, maximum segment number, and others. These constants are vital for simulating various aspects of an operating system's memory management, like paging and segmentation.

5. PCB.java

- **Purpose**: Represents the Process Control Block.
- **Overview**: Manages process-related information like process ID, segment table, resident set count, and frame numbers. This class is essential for tracking and managing process-specific data, crucial in a memory management system.

6. Shell.java

- **Purpose**: Provides a command-line interface for the simulation.
- **Overview**: Includes commands for process creation, destruction, and displaying memory usage. It acts as an interface for user interaction with the memory management system, allowing for command-line-based control.

Title of the Figures （GUI）

List of Figures (GUI)

1. **Figure 1: Main GUI Layout**
    - Description: This figure illustrates the overall layout of the Virtual Memory Simulator's interface.
    - Code Reference: In the showMainGUI method, the arrangement of the VBox and HBox elements, including memoryGrid, pageTableBox, controlPanel, pageReplacementPanel, and destroyPanel, defines the GUI layout.
2. **Figure 2: Memory Grid Visualization**
    - Description: A detailed view of the 8x8 memory grid representing memory frames.
    - Code Reference: The memory grid is created in the createMemoryGrid method. The grid is populated with StackPane elements representing each memory frame, as seen in the createFrameVisual method.
3. **Figure 3: Page Table Views**
    - Description: Screenshots of the loaded and unloaded page tables.
    - Code Reference: The createPageTable method is responsible for setting up the structure of these tables. The toggling between loaded and unloaded pages is managed in the createPageTableContainer method using a ComboBox.

4. **Figure 4: Control Panels**
    - Description: The control panel interface for creating processes, destroying processes, and page replacement.
    - Code Reference: This is detailed in the createControlPanel, createDestroyProcessPanel, and createPageReplacementPanel methods, where various TextField and Button elements are instantiated and event handlers are set up.
5. **Figure 5: Dialogs and Alerts**
    - Description: Examples of dialog boxes used for policy selection and operation feedback.
    - Code Reference: The policy selection dialog is created in the showPolicySelectionDialog method. The showAlert method handles the display of informational alerts.
6. **Figure 6: Instruction Window**
    - Description: The layout and content of the instruction window.
    - Code Reference: This window is created in the openInstructionWindow method, which sets up a TextArea with instructional text.

7. **Figure 7: Real-Time Statistics Display**
    - **Description:** This figure will showcase the new GUI component that displays the total number of pages and requests. It is a crucial feature that provides users with live feedback on the state of the virtual memory system.
    - **Code Reference:** The display for these statistics is set up in the `updateStatsDisplay` method, which is called within various parts of the code where pages are loaded or requests are made.
8. **Figure 8: Random Page Replacement Feature**
    - Description: A depiction of the GUI element allowing users to trigger a random page replacement. This feature adds an instructional dimension to the simulation, demonstrating the unpredictability of certain memory management scenarios.
    - Code Reference: The functionality for random page replacement is implemented in the `randomPageReplace` method. The corresponding button is set up in the `createControlPanel` method and is bound to an event handler that invokes this feature.

Code Integration for GUI Design:

```
public void showMainGUI(Stage primaryStage) {
    this.primaryStage = primaryStage;

    // Header with title and policy selection
    VBox header = createHeader();

    // 8x8 grid for memory representation
    VBox memoryGrid = createMemoryGrid();

    // VBox containing the page table and the ComboBox for column selection
    VBox pageTableBox = createPageTable();

    // Layout configuration
    HBox mainLayout = new HBox(10);
    mainLayout.getChildren().addAll(memoryGrid, pageTableBox);
    mainLayout.setAlignment(Pos.CENTER);

    // The control panel is now an HBox
    HBox controlPanel = createControlPanel();
    HBox destroyPanel = createDestroyProcessPanel(); // New control panel for destroying processes
    HBox pageReplacementPanel = createPageReplacementPanel(); // Control panel for replacing
pages

    // Create the question mark button
    Button questionButton = new Button("?");
    questionButton.setOnAction(event -> {
        // Event handling code goes here
    });

    // Additional layout and scene setting code not shown
}
```

Key Class Overview: PCB

1. **Attributes**:
    - **memory**: A static reference to the **Memory** object, suggesting a shared memory model across all PCB instances.
    - **id**: The unique identifier for each process.
    - **STable**: An array of **SegmentEntry** objects representing the segment table of the process.
    - **residentSetCount**: The count of pages in the resident set.
    - **residentSet**: An array of frame numbers of the pages in the resident set.
    - **policy**: The page replacement policy specific to this process.
    - **loadQueue**: A queue to track the order of page loads into memory, crucial for FIFO (First-In-First-Out) page replacement policy.
2. **Constructor**:
    - Initializes the process ID, segment table, and page replacement policy.
    - The segment table is filled based on the segment sizes passed during the instantiation of a PCB.
    - The resident set count is calculated based on the segment table, adhering to the maximum resident set number defined in **OS.java**.
3. **Static Method - setMemory**:
    - This method sets the **Memory** object reference for the PCB class, indicating that all PCB instances will interact with the same memory instance.

Nested Classes: SegmentEntry and PageEntry

1. **SegmentEntry**:
    - Represents a segment in the memory.
    - Contains attributes like **segmentNum**, **segmentSize**, and **PTable** (Page Table).
    - The constructor calculates the size of the page table based on the segment size and page size.
2. **PageEntry**:
    - Represents a page in a segment.
    - Attributes include **pageNum**, **load** (indicates if the page is loaded into memory), **frameNum** (the frame number where the page is loaded), **usedTime** (for LRU - Least Recently Used strategy), and additional information like protection settings.
    - Methods **setLoad** and **setUnload** manage the loading and unloading of pages from memory.

Functionality Overview:

- The **PCB** class and its nested classes (**SegmentEntry** and **PageEntry**) provide a comprehensive model for process representation in the memory management system.
- It facilitates tracking each process's memory usage, segment and page information, and adherence to specific memory policies.
- The design indicates a sophisticated approach to simulating memory management at both segment and page levels, crucial for understanding modern operating systems' memory management.

Key Class Overview: OS

1. **Constants**:
    - **memorySize**: Specifies the total size of the memory.
    - **pageSize**: Defines the size of each page, a crucial factor in page-based memory management.
    - **maxSegmentNum**: The maximum number of segments a program can have.
    - **maxSegmentSize**: The maximum size of each segment.
    - **maxResidentSetNum**: The maximum number of pages in a process's resident set.
    - **REPLACE_POLICY**: An enum defining page replacement policies (FIFO, LRU).
2. **Attributes**:
    - **ReplacePolicy**: The default replacement policy, set to LRU (Least Recently Used).
    - **processes**: A map storing **PCB** objects, keyed by process ID, representing all active processes.
    - **memory**: An instance of the **Memory** class, representing the simulated memory.
3. **Constructor**:
    - Initializes the **memory** object and sets it in the **PCB** class, ensuring a shared memory model for all processes.

4. **Process Validation and Creation**:
    - **validate**: Validates the legality of creating a process, checking for duplicate names and segment constraints.
    - **createProcess**: Creates a new process if it passes validation, allocating memory and setting up its PCB.
5. **Process Destruction**:
    - **destroyProcess**: Removes a process from the system, freeing up its memory.
6. **Memory Management**:
    - Methods like **showMemoryUsage** and **PageReplace** to manage memory allocation and apply page replacement policies.
7. **Utility Methods**:
    - **getProcesses** and **getAllActivePCBs**: Provide access to the active processes and their PCBs.

Functionality Overview:

- The **OS** class serves as the central hub for managing processes and memory in the simulator.
- It handles process creation and destruction, ensuring legal operations and memory allocation.
- The class also manages page replacement policies, essential for simulating real-world memory management scenarios in operating systems.
- The **showMemoryUsage** method likely provides a detailed view of the memory usage by each process, crucial for understanding memory allocation and paging.

Conclusion

The application represents a comprehensive Virtual Memory Simulator designed to visualize memory management strategies. It adeptly translates the intricacies of memory allocation and page replacement strategies, such as FIFO and LRU, into a visual and engaging format. With its intuitive GUI, the simulator invites users to actively engage with the memory management process through features like process creation and termination, and dynamic page replacement. The inclusion of informative dialog boxes and alert systems contributes to a seamless user experience, reinforcing learning through interaction. The recent enhancements, including real-time statistics and random page replacement, further enrich the simulator's capability to illustrate the operational behaviors of a virtual memory system. Altogether, this simulator is not just a tool but an interactive gateway, opening up the complex world of virtual memory systems to learners and enthusiasts alike.

Acknowledgement

This documentation and the associated Virtual Memory Simulator application owe their development to the collaborative efforts of **Group#666**, including **Kaiyu Liu, Yiyang Hu**, **Lei Xia** and **Yu Fan**. Special thanks to the JavaFX library for providing the graphical components and to the broader Java community for resources and support. Appreciation is also extended to the **Dr. Rashid** for his valuable insights and feedback throughout the development process.