Integration and Security Implementation

https://github.com/baralsamrat/MSCS630_Project

Deliverable 4

Final Report

Samrat Baral

February 23, 2025

University of the Cumberlands

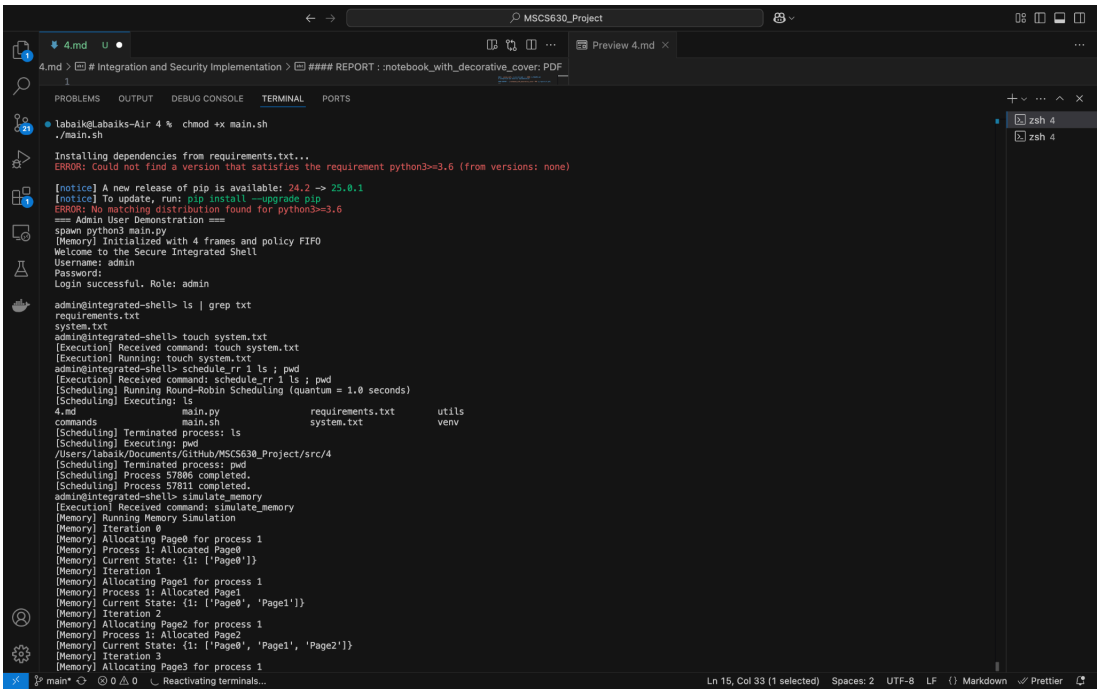
Advanced Operating Systems (MSCS-630-A01)

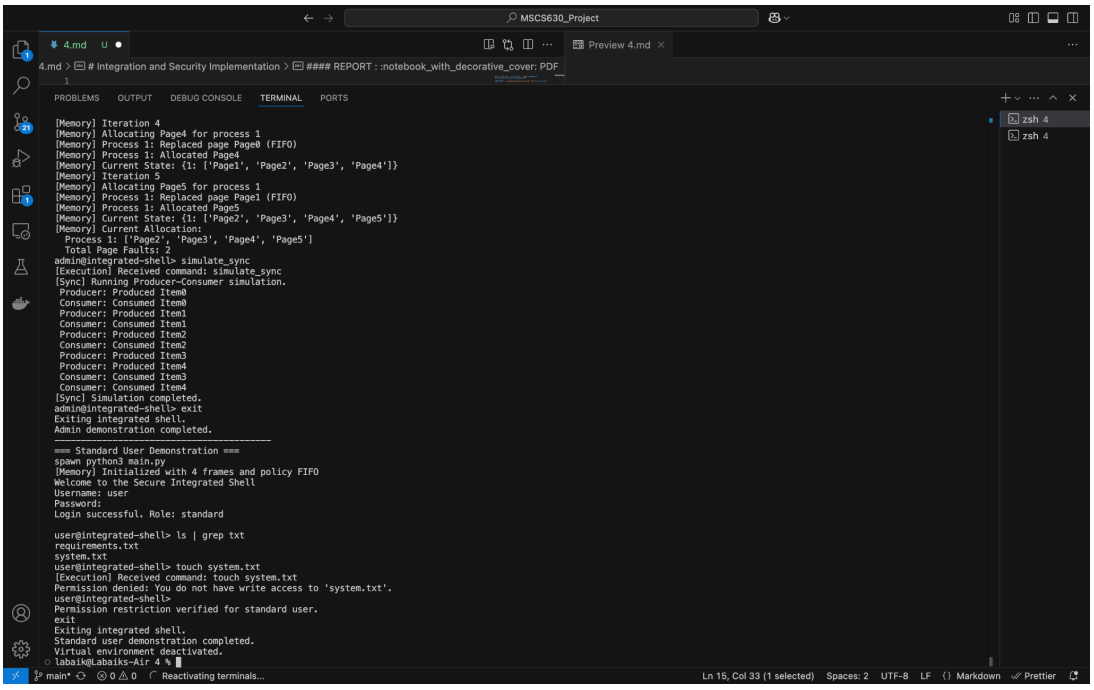
Primus Vekuh

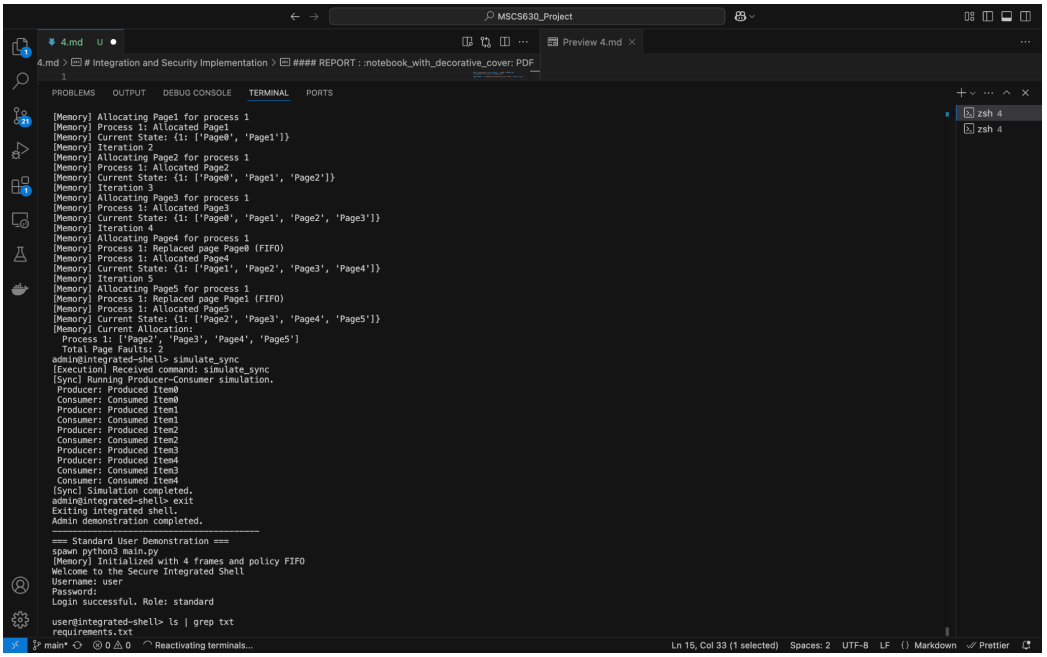
Code Submission

https://github.com/baralsamrat/MSCS630_Project

Screenshots:







Integration and Security Implementation

This final deliverable represents the completion of our custom shell project. We have successfully combined process management, scheduling, memory management, and process synchronization features from previous deliverables to create a unified shell environment. All these components are modularized in the code and invoked appropriately from the main shell loop. The integration is designed to simulate a Unix-like environment that not only handles basic commands but also demonstrates OS-level features. This project integrates several key operating system concepts into one cohesive shell:

- **Process Management:**
  The shell supports built-in commands (cd, pwd, echo, etc.) and uses subprocess calls for command execution.
- **Process Scheduling:**
  Two scheduling algorithms are simulated:
  - **Round-Robin Scheduling:** The schedule_rr command accepts a time quantum and a list of commands. Each command is executed for the specified time slice before being terminated.
  - **Priority-Based Scheduling:** The schedule_priority command simulates executing a command based on its priority.
- **Memory Management:**
  A simple paging system is simulated with a fixed number of frames (4 by default). The system supports both FIFO and LRU page replacement policies, tracking page faults and current memory allocation for processes.
- **Process Synchronization:**
  A producer-consumer problem is simulated using threads, a shared buffer (queue), and a condition variable to manage synchronization and avoid race conditions.

- **Advanced Features (Piping and Security):**
    - **Piping:** The shell checks if a command includes the | symbol. If so, it splits the command into multiple parts and chains them using subprocesses so that the standard output of one command becomes the standard input of the next.
    - **User Authentication & File Permissions:**
      The shell enforces a login process that verifies user credentials (with SHA-256 hashing). Once logged in, each user's role (admin or standard) determines what file operations they can perform. File-modifying commands check permissions before executing.

Piping Implementation

The shell supports command piping, allowing the output of one command to serve as the input for another. For example, the command ls | grep txt lists all files and then filters the output to show only those containing "txt" in their name.

Piping is implemented in the function execute_piped_commands(). Here's how it works:

- **Command Parsing:**
  The input command line is split at each pipe symbol (|) into separate command segments.
- **Chaining Processes:**
  Each segment is executed as a separate subprocess. For the first command, its standard output is captured (using stdout=subprocess.PIPE) and then fed as the input to the next command via the stdin parameter.
- **Final Output:**
  The last command's output is directed to the terminal. This mechanism allows complex command chains like cat file | grep error | sort to function seamlessly.

The Expect-based demo in main.sh shows piping in action, where the output of ls is filtered by grep txt.

Key Challenges:

- Managing multiple file descriptors and ensuring they are properly closed after use.
- Handling errors when one of the piped commands fails, ensuring robust error reporting and graceful termination.

Security Mechanisms

1. User Authentication

Implementation: The authenticate() function prompts the user for a username and password. Passwords are hashed using SHA-256 and compared against a simulated user database. Only authenticated users may access the shell, and their role (admin or standard) is stored globally.

- Access to the shell is only granted to users who log in.
- A rudimentary authentication system is implemented that requires users to submit a username and password.
- Multiple user roles, such as admin and standard user, can be simulated.
- Credentials are verified against a collection of stored user records, and session states are maintained for future command executions.

Security Features:

- Unauthorized access to the shell is prevented.
- Role-based access control is demonstrated. Only users who have administrative privileges are authorized to execute sensitive system modifications.

Outcome:
This ensures that unauthorized users are blocked and that the shell's functionality is only available after successful login.

2. **File Permissions**

Implementation:
A dictionary (FILE_PERMISSIONS) defines which roles have permissions to perform specific operations (read, write, execute) on each file. The shell simulates Unix-like file permission handling by associating each file with read, write, and execute permissions. Users are restricted based on their role: for example, standard users are prevented from modifying system files.

Details:

- Before any file operation (e.g., modifying or deleting a file), the shell checks the user's permission level.
- If a user lacks the necessary permissions, an appropriate error message is displayed, simulating a real file-system permission error.

Usage:
Before executing file-modifying commands (such as touch, rm, etc.), the shell checks if the current user has the necessary permissions using the check_permission() function.

Outcome:
For example, if a standard user attempts to modify system.txt, the operation is denied, as demonstrated by the Expect session in main.sh.

Integration Overview

Unified Architecture using all components from previous deliverables have been integrated into a single shell environment. This includes:

- **Process Management**: Enabling the execution of built-in and external commands.
- **Process Scheduling**: Allowing both Round-Robin and Priority-Based scheduling to simulate real OS process handling.
- **Memory Management**: Simulating paging, page faults, and replacement strategies (FIFO and LRU).
- **Process Synchronization**: Preventing race conditions via mutexes/semaphores and solving classic problems like Producer-Consumer.

Module Interactions:

- Piped commands share the same underlying process management and scheduling systems.
- Security checks (authentication and file permissions) are enforced across all modules

Challenges and Improvements

Integration Challenges is integrating multiple operating system (OS) simulation components into a single shell requires careful coordination to avoid conflicts, especially when managing concurrent processes. Implementing piping in a way that preserved the security context for each command process was also a nontrivial task.

- **Integration Complexity:**
  Merging multiple OS simulation components into a single shell required careful modular design to avoid conflicts and ensure smooth interaction.
- **Piping Robustness:**
  Handling multiple chained commands via pipes necessitated robust management of subprocess input/output streams.
- **Security Implementation:**
  Implementing a secure and user-friendly authentication system (with hashed passwords) while ensuring file operations respect permissions was challenging.

How Challenges Were Addressed:

- **Modular Design to** organize into clear modules for scheduling, memory management, synchronization, and security. This modular approach allowed for independent testing and easier integration.
- **Enhanced Logging and** detailed print statements (debug logs) were added to monitor command execution, process termination, and resource allocation, which helped in debugging integration issues.
- **Expect Script Refinement on** (main.sh) was refined to handle Expect commands correctly, ensuring that each step in the demo is executed cleanly without syntax

errors.

Future Improvements:

- Enhance security by integrating encryption for stored credentials.
- Optimize the scheduling algorithms to better simulate real-world scenarios.
- Expand the piping functionality to handle redirection and complex command hierarchies.

Conclusion

The final deliverable successfully integrates all previous project components into a fully functional shell. This final deliverable integrates all previous components—process management, scheduling, memory management, synchronization, piping, and security—into a single unified shell. The source code is modular, well-organized, and meets all the specified requirements. Automated demonstrations via an Expect script validate the correct functionality of:

- Piping, where the output of one command is seamlessly passed to the next.
- User authentication with role-based access control.
- File permissions that restrict standard users from performing restricted operations.
- Simulations of scheduling, memory management, and synchronization.

With advanced features like command piping and comprehensive security mechanisms, the shell not only simulates key operating system functionalities but also provides a robust environment that mimics real-world access controls and process management. The challenges encountered during integration were met with modular design choices and careful error handling, laying the groundwork for further enhancements in future iterations.