Advanced Shell Simulation with Integrated OS Concepts

Project Description: In this project, you will build a custom shell that simulates a Unix-like Operating System (OS) environment. The shell will integrate key operating system concepts such as process management and scheduling, memory management, process synchronization, and security mechanisms. Unlike traditional shells that rely heavily on the operating system to manage these tasks, your shell will internally manage these features, simulating how an operating system works at a high level. You will develop a shell that allows users to interact with processes, memory, and synchronization features while also implementing job control and basic security mechanisms.

You are free to choose a programming language for your implementation, such as C++, Java, or Python. The system call functions and examples provided later in this project description (e.g., fork(), exec(), wait()) are specific to C++ and Unix-like systems. Depending on the language you choose, you will use different system calls or libraries to achieve similar functionality (e.g., *ProcessBuilder* in Java or *subprocess* in Python).

The project will be split into *four* deliverables. Each deliverable will be accompanied by a report that discusses the relevant operating system concepts, analyzes your design and implementation, and evaluates the shell's performance.

Deliverable 1: Basic Shell Implementation and Process Management

In this deliverable, you will implement a basic shell that:

- Accepts and executes user commands.
- Implements built-in shell commands such as cd, pwd, exit, echo, clear, ls, cat, mkdir, rmdir, rm, touch, and kill.
- Manages foreground and background execution of commands using system calls like fork(), exec(), wait(), and exit().
- Tracks the status of running processes and provides basic job control.

Specific Commands and Utilities to Implement:

1. Built-in Commands:
    - cd [directory]: Change the current working directory.
    - pwd: Print the current working directory.
    - exit: Terminate the shell.
    - echo [text]: Print the specified text to the terminal.
    - clear: Clear the terminal screen.
    - ls: List the files in the current directory.
    - cat [filename]: Display the contents of a file.
    - mkdir [directory]: Create a new directory.
    - rmdir [directory]: Remove an empty directory.
    - rm [filename]: Remove a file.
    - touch [filename]: Create an empty file or update the timestamp of an existing file.

- kill [pid]: Terminate a process by its Process ID.

2. Process Management and Job Control:
    - Support foreground and background execution of commands.
    - Implement basic job control commands:
        - jobs: List all background jobs.
        - fg [job_id]: Bring a background job to the foreground.
        - bg [job_id]: Resume a stopped job in the background.

Report:

- Code Submission:
    - Submit the complete source code for your shell, including the implementation of built-in commands and process management.
- Screenshots:
    - Provide screenshots demonstrating:
        - Execution of built-in commands/utilities
        - Foreground and background process management (including how processes are created and managed).
        - Error handling for invalid commands and inputs.
- Process Management:
    - Explain how processes are created and managed in your shell. Discuss how foreground and background processes are handled.
- Error Handling:
    - Describe how your shell handles errors, such as invalid inputs or commands, and how it provides feedback.
- Challenges and Improvements:
    - Discuss any challenges you encountered during the implementation of Deliverable 1 and how you addressed them.

Deliverable 2: Process Scheduling

In this deliverable, you will extend your shell to internally manage process scheduling using two different algorithms: *Round-Robin Scheduling* and *Priority-Based Scheduling*. The goal is to simulate the process scheduling mechanisms found in an operating system, allowing your shell to manage a queue of processes and switch between them according to the scheduling algorithm.

Integration Requirements:

1. Round-Robin Scheduling:
    - Implement Round-Robin Scheduling, where each process is assigned a time slice (quantum).
    - After each time slice, switch to the next process in the queue.
    - The time slice should be configurable, allowing the user to specify the time slice when running the scheduling algorithm.
    - If a process completes before its time slice is finished, it should be removed from the queue, and the next process should be scheduled.

- Simulate process execution using a timer (e.g., time.sleep() in Python) to represent how long each process runs.

2. Priority-Based Scheduling:
   - Implement Priority-Based Scheduling, where each process is assigned a priority. The highest-priority process should always be selected to run first.
   - If two processes have the same priority, handle them in a First-Come, First-Served (FCFS) manner.
   - If a higher-priority process is added while a lower-priority process is running, the shell should preempt the lower-priority process and run the higher-priority process.
   - Manage the priority queue using a data structure like a heap (priority queue) to ensure that processes are selected based on their priority.

3. Timers:
   - Use timers to simulate the execution of each process. For example, use time.sleep() to simulate a process running for a given duration.
   - Ensure that processes are switched based on the configured time slice in Round-Robin Scheduling, and switch processes immediately when using Priority-Based Scheduling.

Report:

- Code Submission:
  - Submit the complete source code for your shell's scheduling algorithms, including both Round-Robin and Priority-Based Scheduling.
- Screenshots:
  - Provide screenshots demonstrating:
    - Execution of Round-Robin Scheduling, including time slice configuration and process switching.
    - Execution of Priority-Based Scheduling, showing proper handling of process priority and preemption.
    - Performance metrics (waiting time, turnaround time, response time) for processes being scheduled.
- Scheduling Algorithms:
  - Explain how your shell implements both Round-Robin Scheduling (with configurable time slices) and Priority-Based Scheduling (with priority handling and preemption).
- Performance Analysis:
  - Measure and analyze the performance of your scheduling algorithms using the metrics: waiting time, turnaround time, and response time.
- Challenges and Improvements:
  - Discuss any challenges you encountered during the implementation of Deliverable 2 and how you addressed them.

Deliverable 3: Memory Management and Process Synchronization

In this deliverable, you will integrate memory management and process synchronization into your shell. You will simulate how an operating system handles memory allocation using paging and page replacement algorithms, and how it synchronizes processes to prevent race conditions when accessing shared resources.

Integration Requirements:

1. Memory Management:
   - Paging System:
     - Implement a basic paging system to simulate memory allocation and deallocation for processes. Each process should be assigned pages, and the shell should manage memory in fixed-size page frames.
     - The shell should be able to handle page faults when a process requests a page that is not currently in memory.
   - Page Replacement Algorithms:
     - Implement two page replacement algorithms:
       - First-In-First-Out (FIFO): Pages are replaced in the order they were loaded into memory.
       - Least Recently Used (LRU): The least recently used page is replaced when memory overflows.
     - Ensure that when the memory is full, a page replacement algorithm is invoked to free up space for new pages.
   - Track Memory Usage:
     - Track the amount of memory used by each process and simulate memory overflow scenarios where page replacement is necessary.
     - Track the number of page faults that occur when a process requests a page that is not in memory.

2. Process Synchronization:
   - Mutexes or Semaphores:
     - Implement mutexes or semaphores to synchronize access to shared resources between processes.
     - Ensure that processes accessing shared resources (e.g., shared memory or a file) do not create race conditions by allowing simultaneous access.
   - Classical Synchronization Problem:
     - Implement a solution to a classical synchronization problem such as:
       - Producer-Consumer: One or more producers add items to a buffer, and one or more consumers remove them.
       - Dining Philosophers: Philosophers alternately think and eat, sharing limited resources (forks) in a way that prevents deadlock.
     - Demonstrate how your synchronization mechanism prevents race conditions and ensures proper access control.

Report:

- Code Submission:
  - Submit the complete source code for your shell's memory management and process synchronization mechanisms.
- Screenshots:
  - Provide screenshots demonstrating:
    - Memory allocation and deallocation using the paging system.
    - Page faults and page replacement in action (showing both FIFO and LRU algorithms).
    - Proper synchronization of processes accessing shared resources using mutexes or semaphores.
    - Screenshots should clearly show the synchronization problem you implemented (e.g., Producer-Consumer or Dining Philosophers).
- Memory Management:
  - Explain how memory management (paging and page replacement) is handled in a typical operating system and compare it to your implementation.
  - Discuss how your shell handles memory overflow, page faults, and page replacement (using FIFO and LRU algorithms).
  - Analyze the performance of the page replacement algorithms, including the number of page faults and the memory usage of processes.
- Process Synchronization:
  - Explain the use of mutexes or semaphores in your implementation and how they prevent race conditions.
  - Describe the classical synchronization problem you implemented and how your solution prevents deadlock, ensures resource availability, and avoids race conditions.
- Challenges and Improvements:
  - Discuss any challenges you encountered while implementing memory management and process synchronization, and how you addressed them.

Deliverable 4: Integration and Security Implementation

In this final deliverable, you will integrate all components from previous deliverables into a cohesive shell. Additionally, you will implement advanced features such as piping and security mechanisms (user authentication and file permissions).

Integration Requirements:

- Piping:
  - Implement command piping, where the output of one command is used as the input to another.
  - Example: *ls | grep txt* should list files and filter them using grep to show only those containing "txt" in the name.

    - The shell should support chaining multiple commands using pipes (e.g., cat file | grep error | sort).

- Security:
    - User Authentication:
        - Implement a user authentication system, where users need to log in to access the shell. Allow users to have different credentials (username and password).
        - Simulate multiple users with different permission levels (e.g., admin, standard user).
- File Permissions:
    - Implement file permission handling where users have varying access rights (read, write, execute) to files.
    - Simulate how file access is restricted based on user permissions (e.g., standard users cannot modify system files).

Final Report:

- Code Submission:
    - Submit the complete source code for the fully integrated shell, including piping and security mechanisms.
- Screenshots:
    - Provide screenshots demonstrating:
        - The shell's piping functionality, showing how output from one command is passed as input to another.
        - User authentication: Logging in with different user credentials and accessing the shell.
        - File permissions: Showing users with different permission levels accessing files and demonstrating restrictions based on their roles.
- Integration Overview:
    - Provide a detailed overview of how all the components from previous deliverables (process management, scheduling, memory management, synchronization, etc.) were integrated into a single shell.
- Piping Implementation:
    - Discuss how piping was implemented in your shell. Explain how the output of one command is passed to the input of another command.
- Security Mechanisms:
    - Describe your implementation of user authentication and file permissions. Discuss how access restrictions are handled based on user roles and file permissions.
- Challenges and Improvements:
    - Discuss any challenges you encountered during the integration of the various components and how you addressed them.