Short Report on Process Scheduling and Synchronization Simulations

CSC 4320/6320, Spring 25

Operating Systems

Roya Hosseini

Team Members: Shruti Dabhi, Samuel Holison

Overview: We worked on implementing and simulating three key concepts in operating systems using Java:

1. **Process Scheduling Simulation**
2. **Dining Philosophers Problem**
3. **Readers-Writers Synchronization Problem**

Our goal was to understand how threading and synchronization are handled in concurrent Java programs and to build simulations that demonstrate realistic behavior.

Problems and Solutions

- **Process Scheduling Simulation**
    - *Problem*: The program needed to read a list of processes from a file and simulate each one running based on its CPU burst time.
    - *Solution*: We created a ProcessThread class that reads each line from processes.txt, extracts the burst time, and starts a separate thread to simulate the process execution

using Thread.sleep(). We ensured each process completed before the program exited by using join().

- **Dining Philosophers Problem**
    - *Problem*: Preventing deadlock when philosophers try to pick up forks (resources).
    - *Solution*: We implemented an asymmetrical solution where even-numbered philosophers pick up the left fork first and odd-numbered ones pick up the right fork first. We used ReentrantLock to manage each fork as a resource. This approach helped us avoid circular wait conditions.
- **Readers-Writers Problem**
    - *Problem*: Ensuring writers have exclusive access while allowing multiple readers concurrently when no writer is active.
    - *Solution*: We implemented a custom ReadWriteLock class that uses a ReentrantLock and a Condition variable. It ensures readers wait when a writer is active and vice versa. Readers increment and decrement a shared counter, and writers wait until there are no active readers.

Challenges Faced

- Understanding when and how to use lock(), unlock(), and condition.await() vs condition.signalAll() was initially tricky.
- Avoiding deadlock in Dining Philosophers required testing different fork-picking strategies.

Conclusion

We successfully implemented all three simulations (Process Scheduling Simulation, Dining Philosophers Simulation and Readers-Writers Simulation) with proper synchronization and thread handling. The

project gave us valuable hands-on experience with Java concurrency, race condition prevention, and design patterns for safe thread execution.

We tested all modules to ensure expected behaviors, and interrupt mechanisms were used to end simulations gracefully. The program is user-interactive, allowing us to choose which simulation to run from the console.