Project Report

##

Use-Cases Demonstrating the Utility of Multithreaded Programming and Synchronization

1. Concurrency in User Requests

- Multithreading handles multiple user requests at the same time, improving the responsiveness of systems like e-commerce platforms.

2. Thread Synchronization

- **Mutexes** and **semaphores** are used to ensure that threads accessing shared resources (e.g., Inventory) do not cause data inconsistency.

3. Deadlock Prevention

- Multithreading techniques(Dining Philosophers Problem) ensure that the system avoids deadlocks, keeping operations flowing without interruptions.

4. Improved Resource Utilization

- Maximizes system resource use, allowing better throughput and efficiency, especially under high traffic conditions.

5. Real-Time Inventory Updates

- Multithreading keeps inventory data updated in real-time, reflecting Inventory changes immediately after changes are made.

6. Asynchronous Order Processing

- Orders are processed concurrently, reducing user wait times and enhancing system performance.

7. Scalable Performance

- Systems can scale by adding more threads, accommodating increased user activity without degrading performance.

##

Architecture and Implementation

AUTHENTICATION LOGIC :

- Reads users.db to authenticate users.

INVENTORY MANAGEMENT :

- Reads/writes to inventory.db.

- Allows only the Registered Users to add, delete, view and modify items.

ORDER PROCESSING:

- Users can add, view, update, and delete the ordered items.

- Threads simulate multiple customers ordering concurrently.

- Each order updates Inventory in a synchronized manner.

DINING PHILOSOPHERS SIMULATION

--

- Simulates the classical OS problem.

- Uses threads to represent philosophers.

- Avoids deadlocks by managing resource access order**.**

DATABASE HANDLING:

- inventory.db and users.db are flat files storing items and authenticating users.

DOCUMENTATION:

- Instructions.txt provides usage details.

- Workflow.png outlines the system process.

##

Core Operating System Concepts Demonstrated

The OS-Ecommerce project showcases several fundamental Operating System concepts by simulating an e-commerce platform. It is designed to educate and demonstrate how multithreading, synchronization, and deadlock avoidance are implemented in real-world-like scenarios.

Core OS Concepts Used

1. Multithreading (Concurrency)

- Uses POSIX threads (pthreads) to simulate multiple users acting concurrently.
- Each customer or philosopher runs as a separate thread.

2. Synchronization

- Utilizes mutexes to prevent race conditions.
- Ensures thread-safe access to shared resources like the inventory database.
- Semaphores and other synchronization primitives are used where necessary.

3. Critical Section Management

- Demonstrates handling of critical sections—code that accesses shared data—using mutex locks.

4. Deadlock Avoidance

- Implements the Dining Philosophers Problem to simulate resource allocation challenges.
- Avoids deadlocks using ordered resource access.

5. Process Simulation

- Threads simulate process-like behavior to execute operations concurrently.

6. Thread Communication and Coordination

- Threads coordinate to manage tasks such as placing orders and updating inventory.
- Reflects real OS thread scheduling and cooperation.

###

Dry Run

Initial State:

- philosopher_active[0..9] = true (All philosophers active)
- current_philosopher = 0 (Philosopher 0 starts)
- active_users = 10
- forks[0..9] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1} (All forks available)
- (condition_variable i.e cv): Used to synchronize philosopher threads. Each philosopher waits until `current_philosopher == id` before proceeding. `cv.notify_all()` is called after each philosopher completes their turn to wake the next one.

Philosopher 0's Turn

Prompt: Philosopher 0, do you want to exit? (-1 to exit, any other number to continue): 1

====== Inventory Menu ======
1. Add Inventory
2. Update Inventory
3. Display Inventory
4. Remove Inventory
Enter your choice: 1
...
Menu interaction completed for Philosopher 0

- Philosopher 0 is thinking. (2-second delay)
- Picks up Forks 0 and 1 (forks[0] = 0, forks[1] = 0)
- Philosopher 0 is eating. (3-second delay)
- Releases Forks 0 and 1 (forks[0] = 1, forks[1] = 1)
- Updates current_philosopher = 1 and notifies all

Philosopher 1's Turn

Prompt: Philosopher 1, do you want to exit? (-1 to exit, any other number to continue): 6

====== Inventory Menu ======
1. Add Inventory
2. Update Inventory
3. Display Inventory
4. Remove Inventory
Enter your choice: 2
...
Menu interaction completed for Philosopher 1

- Philosopher 1 is thinking. (1-second delay)
- Picks up Forks 1 and 2 (forks[1] = 0, forks[2] = 0)
- Philosopher 1 is eating. (2-second delay)
- Releases forks[1] and forks[2]
- Updates current_philosopher = 2 and notifies all

Philosopher 2's Turn

Prompt: Philosopher 2, do you want to exit? (-1 to exit, any other number to continue): 1

====== Inventory Menu ======
...
Menu interaction completed for Philosopher 2

- Philosopher 2 is thinking. (3-second delay)
- Picks up Forks 2 and 3 (forks[2] = 0, forks[3] = 0)
- Philosopher 2 is eating. (1-second delay)
- Releases Forks 2 and 3
- Updates current_philosopher = 3

Philosopher 3's Turn

Prompt: Philosopher 3, do you want to exit? (-1 to exit, any other number to continue): 1

====== Inventory Menu ======
...
Menu interaction completed for Philosopher 3

- Philosopher 3 is thinking. (2-second delay)
- Picks up Forks 3 and 4 (forks[3] = 0, forks[4] = 0)
- Philosopher 3 is eating. (3-second delay)
- Releases Forks 3 and 4
- Updates current_philosopher = 4

Philosopher 4's Turn

Prompt: Philosopher 4, do you want to exit? (-1 to exit, any other number to continue): 1

====== Inventory Menu ======
...
Menu interaction completed for Philosopher 4

- Philosopher 4 is thinking. (2-second delay)
- Picks up Forks 4 and 5 (forks[4] = 0, forks[5] = 0)
- Philosopher 4 is eating. (3-second delay)
- Releases Forks 4 and 5
- Updates current_philosopher = 5

Philosopher 5 Exits

Prompt: Philosopher 5, do you want to exit? (-1 to exit, any other number to continue): -1
Philosopher 5 has exited.

- philosopher_active[5] = false
- active_users = 9

Execution continues similarly for remaining philosophers until all have exited.

Note: Output may interleave due to multithreaded `cv.notify_all()`, causing later philosophers to show input/output during prior philosopher's "eating" state.

##

Conclusion

The **OS-Ecommerce** project is a practical and educational simulation that demonstrates the implementation of core operating system concepts in a simplified e-commerce environment. By leveraging **POSIX threads**, **mutexes**, and **semaphores**, the system effectively models concurrent user interactions such as inventory management and order processing.

The integration of the **Dining Philosophers Problem** showcases thoughtful handling of **deadlock avoidance** and **resource management**, further reinforcing the application of theoretical OS principles. The use of **flat-file databases** and a **command-line interface** adds realism and modularity, making it an excellent learning tool.

Overall, this project serves as a comprehensive demonstration of OS topics including **multithreading, synchronization, critical section management, deadlock handling**, and **file I/O**, all within the context of a familiar and practical application: an online shopping platform. It balances simplicity and depth, making it ideal for both academic exploration and foundational systems programming experience.