**Project Documentation: Deadlock Prevention Techniques**

**Overview**

This project focuses on implementing and testing various deadlock prevention techniques in a multi-threaded environment using Java. Deadlocks are a common issue in concurrent programming where two or more threads are blocked forever, waiting for each other to release resources. This project provides a set of techniques to prevent deadlocks and includes test cases to verify their effectiveness.

---

**Features of the Code**

1. Deadlock Detection

- \*\*Method\*\*: `detectDeadlock()`

- \*\*Description\*\*:

- This method detects deadlocks by checking for cycles in the resource allocation graph.

- It uses a depth-first search (DFS) approach to detect cycles.

- \*\*How It Works\*\*:

- A resource allocation graph is maintained using a `Map<Integer, List<Integer>>`.

- The `detectCycle()` method recursively checks for cycles in the graph.

- \*\*Overcoming Deadlocks\*\*:

- If a cycle is detected, the system can take corrective actions like preempting resources or killing processes.

---

2. \*\*Hold and Wait Prevention\*\*

- \*\*Method\*\*: `allocateAllResourcesAtOnce(List<Lock> resources)`

- \*\*Description\*\*:

- This technique ensures that a thread requests all required resources at once.

- If all resources are not available, the thread releases all acquired resources and retries.

- \*\*How It Works\*\*:

- The method attempts to lock all resources in a single operation using `tryLock()`.

- If any resource cannot be locked, all previously locked resources are released.

- \*\*Overcoming Deadlocks\*\*:

- Prevents the "hold and wait" condition, one of the four necessary conditions for deadlock.

---

3. \*\*No Preemption Prevention\*\*

- \*\*Method\*\*: `preemptResources(List<Lock> resources)`

- \*\*Description\*\*:

- This technique forcefully releases resources if a deadlock is detected or likely to occur.

- \*\*How It Works\*\*:

- The method attempts to lock resources and immediately unlocks them to simulate preemption.

- \*\*Overcoming Deadlocks\*\*:

- Ensures that resources can be forcibly taken away from threads, preventing deadlocks.

---

4. \*\*Circular Wait Prevention\*\*

- \*\*Method\*\*: `enforceResourceOrdering(List<Lock> resources)`

- \*\*Description\*\*:

- This technique enforces a global ordering of resources to prevent circular waits.

- \*\*How It Works\*\*:

- Resources are sorted based on their identity hash codes.

- Threads must always request resources in this predefined order.

- \*\*Overcoming Deadlocks\*\*:

- Prevents the "circular wait" condition by ensuring that threads cannot wait for resources in a cyclic manner.

---

5. \*\*Timeout-Based Deadlock Prevention\*\*

- \*\*Method\*\*: `acquireWithTimeout(Lock resource, long timeoutMs)`

- \*\*Description\*\*:

- This technique uses a timeout mechanism to prevent threads from waiting indefinitely for resources.

- \*\*How It Works\*\*:

- The method attempts to lock a resource within a specified timeout period using `tryLock(timeoutMs)`.

- \*\*Overcoming Deadlocks\*\*:

- If the lock cannot be acquired within the timeout, the thread can retry or take alternative actions.

---

6. \*\*Priority-Based Allocation\*\*

- \*\*Method\*\*: `priorityBasedAllocation(Map<Integer, Lock> processLocks)`

- \*\*Description\*\*:

- This technique assigns priorities to processes and ensures that low-priority processes release their locks if necessary.

- \*\*How It Works\*\*:

- Processes are sorted by priority, and locks are acquired in priority order.

- \*\*Overcoming Deadlocks\*\*:

- Prevents resource starvation and ensures that high-priority processes can always acquire resources.

---

7. \*\*Simulated Database Locking\*\*

- \*\*Method\*\*: `simulateDatabaseLocking(Lock tableLock, Lock rowLock)`

- \*\*Description\*\*:

- This technique simulates database row-level locking to prevent deadlocks in database systems.

- \*\*How It Works\*\*:

- The method first locks the table and then attempts to lock the row.

- If the row cannot be locked, the table lock is released.

- \*\*Overcoming Deadlocks\*\*:

- Ensures that locks are acquired in a hierarchical manner, reducing the likelihood of deadlocks.

---

8. \*\*Distributed Deadlock Prevention\*\*

- \*\*Method\*\*: `distributedDeadlockPrevention(List<String> resourceIds)`

- \*\*Description\*\*:

- This technique prevents deadlocks in distributed systems by using unique global IDs for resources.

- \*\*How It Works\*\*:

- Resources are sorted globally, and locks are acquired in a consistent order across all nodes.

- \*\*Overcoming Deadlocks\*\*:

- Ensures that all nodes follow the same resource ordering, preventing distributed deadlocks.

---

9. \*\*Starvation Prevention\*\*

- \*\*Method\*\*: `starvationPrevention()`

- \*\*Description\*\*:

- This technique prevents thread starvation by using fair locks.

- \*\*How It Works\*\*:

- A `ReentrantLock` with fairness enabled ensures that the longest-waiting thread gets access to the resource.

- \*\*Overcoming Deadlocks\*\*:

- Ensures that no thread is starved of resources, reducing the likelihood of deadlocks.

---

10. \*\*Additional Techniques\*\*

- \*\*Resource Ordering with Priority\*\*:

- \*\*Method\*\*: `resourceOrderingWithPriority(List<Lock> resources, List<Integer> priorities)`

- \*\*Description\*\*: Combines resource ordering with process priorities to prevent deadlocks.

- \*\*Wait-Die Scheme\*\*:

- \*\*Method\*\*: `waitDie(int timestamp, int otherTimestamp)`

- \*\*Description\*\*: Older processes wait for younger ones, while younger processes die if they cannot acquire resources.

- \*\*Wound-Wait Scheme\*\*:

- \*\*Method\*\*: `woundWait(int timestamp, int otherTimestamp)`

- \*\*Description\*\*: Older processes preempt younger ones, while younger processes wait if they cannot acquire resources.

---

**Test Cases Explanation**

1. \*\*Hold and Wait Prevention Test\*\*

- \*\*Objective\*\*: Verify that resources are allocated all at once.

- \*\*Test Case\*\*:

- Two resources (`resource1` and `resource2`) are created.

- The `allocateAllResourcesAtOnce()` method is called to lock both resources.

- \*\*Expected Result\*\*: If both resources are locked successfully, the test passes.

---

2. \*\*Circular Wait Prevention Test\*\*

- \*\*Objective\*\*: Verify that resources are locked in a global order.

- \*\*Test Case\*\*:

- The `enforceResourceOrdering()` method is called with a list of resources.

- \*\*Expected Result\*\*: Resources are locked and unlocked in a sorted order, preventing circular waits.

---

3. \*\*Timeout-Based Prevention Test\*\*

- \*\*Objective\*\*: Verify that a resource is locked within a timeout period.

- \*\*Test Case\*\*:

- The `acquireWithTimeout()` method is called with a timeout of 1000ms.

- \*\*Expected Result\*\*: If the resource is locked within the timeout, the test passes.

---

4. \*\*Priority-Based Allocation Test\*\*

- \*\*Objective\*\*: Verify that locks are acquired based on process priority.

- \*\*Test Case\*\*:

- A map of process IDs and locks is created.

- The `priorityBasedAllocation()` method is called to lock resources in priority order.

- \*\*Expected Result\*\*: Resources are locked and unlocked in priority order.

---

5. \*\*Database Locking Test\*\*

- \*\*Objective\*\*: Verify that table and row locks are acquired hierarchically.

- \*\*Test Case\*\*:

- The `simulateDatabaseLocking()` method is called with a table lock and a row lock.

- \*\*Expected Result\*\*: The table lock is acquired first, followed by the row lock.

---

6. \*\*Distributed Deadlock Prevention Test\*\*

- \*\*Objective\*\*: Verify that resources are locked in a global order.

- \*\*Test Case\*\*:

- The `distributedDeadlockPrevention()` method is called with a list of resource IDs.

- \*\*Expected Result\*\*: Resources are locked in a sorted order.

---

7. \*\*Starvation Prevention Test\*\*

- \*\*Objective\*\*: Verify that fair locks prevent thread starvation.

- \*\*Test Case\*\*:

- The `starvationPrevention()` method is called to lock and unlock a fair lock.

- \*\*Expected Result\*\*: The lock is acquired and released without starvation.

---

8. \*\*Deadlock Detection Test\*\*

- \*\*Objective\*\*: Verify that deadlocks are detected in the resource allocation graph.

- \*\*Test Case\*\*:

- Edges are added to the resource graph to create a cycle.

- The `detectDeadlock()` method is called to detect the cycle.

- \*\*Expected Result\*\*: If a cycle is detected, the test passes.

---

**Conclusion**

This project demonstrates various deadlock prevention techniques and provides test cases to verify their effectiveness. By implementing these techniques, developers can build more robust and deadlock-free concurrent systems. The test cases ensure that each technique works as expected and can be used to simulate real-world scenarios.

**THANKING YOU SIR**