Lecture 25: Read/Write Pattern, Dining Philosophers

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

Acknowledgments: CMSC 330 U. Maryland, CS 444 (Clarkson), Dave Johnson (COMP 421), Ken Birman (Cornell)
Motivation for Read-Write Object-based isolation

1. Sorted List example
2. public boolean contains(Object object) {
3.     // Observation: multiple calls to contains() should not
4.     // interfere with each other
5.     return isolatedWithReturn(this, () -> {
6.         Entry pred, curr;
7.         ...
8.         return (key == curr.key);
9.     });
10. }
11.
12. public int add(Object object) {
13.     return isolatedWithReturn(this, () -> {
14.         Entry pred, curr;
15.         ...
16.         if (...) return 1; else return 0;
17.     });
18. }
Read-Write Object-Based Isolation

\[
\text{isolated(\text{readMode(obj1)}, \text{writeMode(obj2)}, \ldots, () \rightarrow \text{<body> });}
\]

- Programmer specifies list of objects as well as their read-write modes for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists such that one of the accesses is in \text{writeMode}

**Sorted List example**

1. public boolean contains(Object object) {
2.     return \text{isolatedWithReturn( readMode(this), () \rightarrow { }
3.     Entry pred, curr;
4.      ... 
5.      return (key == curr.key);
6.     });
7. }
8. }
9. public int add(Object object) {
10.    return \text{isolatedWithReturn( writeMode(this), () \rightarrow { }
11.      Entry pred, curr;
12.        ... 
13.      if (...) return 1; else return 0;
14.    });
15. }
Read-Write Concurrency Pattern

• Common pattern in concurrency

• HJLib Read-Write Object Isolation, Java ReentrantReadWriteLock, C++ Boost UpgradeLockable, sync.RW Mutex in Go

• Upgradeable/downgradeable
  • **Can upgrade Read access to Write access**
  • Could be tricky to implement and avoid deadlock
  • **Downgrade Write access to Read access**

• Priority policies
  • **Read-preferring**
    • Max concurrency
    • Could starve writers
  • **Write-preferring**
    • Less concurrency
    • More overhead
Liveness Recap

• **Deadlock**: task’s execution remains incomplete due to it being blocked awaiting some condition
• **Livelock**: two or more tasks repeat the same interactions without making any progress
• **Starvation**: some task is repeatedly denied the opportunity to make progress
• **Bounded wait (fairness)**: each task requesting a resource should only have to wait for a bounded number of other tasks to “cut in line”
• **Non-concurrency**: a task is prevented from making progress due to overly restrictive resource management
Deadlock Conditions

- Mutual Exclusion
  - At least one resource that must be held is in non-shareable mode
- Hold and wait
  - There exists a task holding a resource, and waiting for another
- No preemption
  - Resources cannot be preempted
- Circular wait
  - There exists a set of tasks \( \{T_1, T_2, \ldots, T_N\} \), such that
    - \( T_1 \) is waiting for \( T_2 \), \( T_2 \) for \( T_3 \), \ldots and \( T_N \) for \( T_1 \)
  - All four conditions must hold for deadlock to occur
The Dining Philosophers Problem

A classical Synchronization Problem devised by Dijkstra in 1965

Constraints

- Five philosophers either eat or think
- They must have two chopsticks to eat
- Can only use chopsticks on either side of their plate
- No talking permitted

Goals

- Progress guarantees
  - Deadlock freedom
  - Livelock freedom
  - Starvation freedom
  - Maximum concurrency (no one should starve if there are available forks for them)
General Structure of Dining Philosophers Problem: PseudoCode

1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5. async(() → {
6. while(true) {
7. Think ;
8. Acquire chopsticks;
9. // Left chopstick = chop[p]
10. // Right chopstick = chop[(p-1)%numChops]
11. Eat ;
12. } // while
13. }); // async
14.} // for
Solution 1: Using Java’s Synchronized Statement

1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5.     async(() -> {
6.         while(true) {
7.             Think ;
8.             synchronized(chop[p]) { // get the left chopstick
9.                 synchronized(chop[(p-1)%numChops]) { // get the right chopstick
10.                     Eat ;
11.                 }
12.             }
13.         } // while
14.     }); // async
15.} // for
Problems?

• What if everyone picks up the left chopstick at the same time?
• Deadlock!
• Starvation due to deadlock
• No livelock
• Non-concurrency due to deadlock
Solution 2: Using Java’s tryLock

1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5.     async(() -> {
6.         int first = p; int second = (p - 1) & numChops;
7.         while(true) {
8.             Think ;
9.             if (!chop[first].lock.tryLock()) continue;
10.            if (!chop[second].lock.tryLock()) {
11.                chop[first].lock.unlock(); continue;
12.            }
13.            Eat ;
14.            chop[first].lock.unlock();chop[second].lock.unlock();
15.        } // while
16.     }); // async
17. } // for
Problems?

• Everyone picks up the left chopstick at the same time, tries to pick up the right one, gives up, puts down the left one, and repeat
• Livelock!
• Starvation due to livelock!
• No deadlock
• Non-concurrency due to livelock
Solution 3: Using Global Isolated

1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5.   async(() -> {
6.     while(true) {
7.       Think ;
8.       isolated {
9.         Pick up left and right chopsticks;
10.        Eat ;
11.       }
12.     } // while
13.   }); // async
14.} // for
Problems?

• No deadlock or lovlock possible
• Starvation!
  • No guarantee that a philosopher will ever get to eat, if others are very hungry and “cut in line” all the time.
• Non-concurrency
  • Only one philosopher can eat at any time
Solution 4a: Impose Order

1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5.   async(() -> {
6.     int first = (p == 0)? (p - 1) % numChops : p
7.     int second = (p == 0)? p : (p - 1) % numChops
8.     while(true) {
9.       Think ;
10.      synchronized(first) {
11.         synchronized(second) {
12.           Eat ;
13.         }
14.      }
15.   } // while
16. }); // async
17.} // for
Preventing Deadlock by Ordering

It is not possible for all philosophers to have a chopstick

1. Two philosophers, A and B, must share a chopstick, X, that is “smaller” than all other chopsticks
2. One of them, A, has to pick up X first
3. B can’t pick up X at this point
4. B can’t pick up the “bigger” chopstick until X is released
5. So, 4 philosophers left, 5 chopsticks total
6. One philosopher must be able to have two chopsticks!
Solution 4b: Using tryLock

1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5.     async() → {
6.         int first = (p == 0)? (p - 1) % numChops : p
7.         int second = (p == 0)? p : (p - 1) % numChops
8.         while(true) {
9.             Think ;
10.            if (!chop[first].lock.tryLock()) continue;
11.            if (!chop[second].lock.tryLock()) {
12.                chop[first].lock.unlock(); continue;
13.            }
14.            Eat ;
15.                chop[first].lock.unlock();chop[second].lock.unlock();
16.            } // while
17.        }); // async
18.    } // for
Solution 4c: Using Object-Based Isolation

1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5.   async(() -> {
6.     while(true) {
7.       Think ;
8.       isolated (chop[p], chop[(p-1)%numChops]){
9.       Eat ;
10.     }
11.   } // while
12. }); // async
13. } // for
Problems for 4a, 4b and 4c?

• No deadlock or lovelock possible
• Starvation!
  • No guarantee that a philosopher will ever get to eat, if others are very hungry and “cut in line” all the time.
• Concurrency
  • 4a: still have a non-concurrency problem. If philosopher 0 is eating, philosophers 1-4 could all be holding their left chopstick waiting
  • 4b and 4c: If a philosopher is hungry, and his chopsticks are not used for eating, he’ll get to eat
Solution 5: Using Semaphores

1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ...; // Initialize array of chopsticks
4. Semaphore table = new Semaphore(3, true);
5. for (i=0;i<numChops;i++) chop[i].sem = new Semaphore(1, true);
6. for(p in 0 .. numPhilosophers-1) {
7.     async(() -> {
8.         Think;
9.         table.acquire(); // At most 3 philosophers at table
10.        p = empty place at the table that has nobody on the left
11.        chop[p].sem.acquire(); // Acquire left chopstick
12.        chop[(p-1)%numChops].sem.acquire(); // Acquire right chopstick
13.        Eat;
14.        chop[p].sem.release(); chop[(p-1)%numChops].sem.release();
15.        table.release();
16.     });// while
17. }); // async
18.} // for