Lecture 27: Read/Write Pattern, Java Locks - Soundness and Progress Guarantees

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
Recall that the use of synchronization is to protect interfering accesses

- Concurrent reads of same memory: Not a problem
- Concurrent writes of same memory: Problem
- Concurrent read & write of same memory: Problem

So far:

- If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time

But:

- This is unnecessarily conservative: we could still allow multiple simultaneous readers (as in object-based isolation)

Consider a hashtable with one coarse-grained lock

- Only one thread can perform operations at a time

But suppose:

- There are many simultaneous lookup operations and insert operations are rare
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

isolated(readMode(obj1),writeMode(obj2), …, () -> <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 writeMode

- Sorted List example

1. public boolean contains(Object object) {
2.     return isolatedWithReturn( readMode(this), () -> {
3.         Entry pred, curr;
4.         ...
5.         return (key == curr.key);
6.     });
7. }
8.  
9. public int add(Object object) {
10.    return isolatedWithReturn( writeMode(this), () -> {
11.        Entry pred, curr;
12.        ...
13.        if (...) return 1; else return 0;
14.    });
15. }
interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}

• Even though the interface appears to just define a pair of locks, the semantics of the pair of locks is coupled as follows
  — Case 1: a thread has successfully acquired writeLock().lock()
    – No other thread can acquire readLock() or writeLock()
  — Case 2: no thread has acquired writeLock().lock()
    – Multiple threads can acquire readLock()
    – No other thread can acquire writeLock()

• java.util.concurrent.locks.ReadWriteLock interface is implemented by java.util.concurrent.locks.ReadWriteReentrantLock class
class Hashtable<K,V> {
    ...
    // coarse-grained, one lock for table
    ReentrantReadWriteLock lk = new ReentrantReadWriteLock();
    V lookup(K key) {
        int bucket = hasher(key);
        lk.readLock().lock(); // only blocks writers
        ... read array[bucket] ...
        lk.readLock().unlock();
    }
    void insert(K key, V val) {
        int bucket = hasher(key);
        lk.writeLock().lock(); // blocks readers and writers
        ... write array[bucket] ...
        lk.writeLock().unlock();
    }
}
Read-Write Concurrency Pattern

• Common pattern in concurrency
• HJLib Read-Write Object Isolation, Java ReentrantReadWriteLock, C++ Boost UpgradeLockable, sync.RWmutex 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-prefering
    • Max concurrency
    • Could starve writers
  • Write-prefering
    • Less concurrency
    • More overhead
Safety vs Liveness

• In a concurrent setting, we need to specify both the safety and the liveness properties of an object

• Need a way to define
  —Safety: when an implementation is functionally correct (does not produce a wrong answer)
  —Liveness: the conditions under which it guarantees progress (completes execution successfully)

• Examples of safety
  • Data race freedom is a desirable safety property for parallel programs (Module 1)
  • Linearizability is a desirable safety property for concurrent objects (Module 2)
Liveness

• Liveness = a program’s ability to make progress in a timely manner

• Termination ("no infinite loop") is not necessarily a requirement for liveness
  • some applications are designed to be non-terminating

• Different levels of liveness guarantees (from weaker to stronger) for tasks/threads in a concurrent program
  1. Deadlock freedom
  2. Livelock freedom
  3. Starvation freedom
  4. Bounded wait
1. Deadlock-Free Parallel Program Executions

• A parallel program execution is *deadlock-free* if no task’s execution remains incomplete due to it being blocked awaiting some condition.

• Example of a program with a deadlocking execution:

```
// Thread T1
public void leftHand() {
    synchronized(obj1) {
        synchronized(obj2) {
            // work with obj1 & obj2
            ...
        }
    }
}

// Thread T2
public void leftHand() {
    synchronized(obj2) {
        synchronized(obj1) {
            // work with obj2 & obj1
            ...
        }
    }
}
```

• In this case, Task1 and Task2 are in a deadlock cycle.
  – Construct that can lead to deadlock in HJlib: async await
  – There are many constructs that can lead to deadlock cycles in other programming models (e.g., thread join, synchronized, Java locks)
2. Livelock-Free Parallel Program

• A parallel program execution exhibits *livelock* if two or more tasks repeat the same interactions without making any progress (special case of nontermination)

• Livelock example:

  ```java
  // Task T1
  incrToTwo(AtomicInteger ai) {
    // increment ai till it reaches 2
    while (ai.incrementAndGet() < 2);
  }

  // Task T2
  decrToNegTwo(AtomicInteger ai) {
    // decrement ai till it reaches -2
    while (ai.decrementAndGet() > -2);
  }
  ```

• Many well-intended approaches to avoid deadlock result in livelock instead
3. Starvation-Free Parallel Program Executions

A parallel program execution exhibits *starvation* if some task is repeatedly denied the opportunity to make progress

— Starvation-freedom is sometimes referred to as “lock-out freedom”

— Starvation is possible in HJ programs, since all tasks in the same program are assumed to be cooperating, rather than competing

  - If starvation occurs in a deadlock-free HJ program, the “equivalent” sequential program must be non-terminating (infinite loop)
4. Bounded Wait

• A parallel program execution exhibits bounded wait if each task requesting a resource should only have to wait for a bounded number of other tasks to “cut in line” i.e., to gain access to the resource after its request has been registered.

• If bound = 0, then the program execution is fair
Key Functional Groups in java.util.concurrent (j.u.c.)

- Atomic variables
  - The key to writing lock-free algorithms
- Concurrent Collections:
  - Queues, blocking queues, concurrent hash map, …
  - Data structures designed for concurrent environments
- Locks and Conditions
  - More flexible synchronization control
  - Read/write locks
- Executors, Thread pools and Futures
  - Execution frameworks for asynchronous tasking
- Synchronizers: Semaphore
  - Ready made tool for thread coordination
Semaphores

• Conceptually serve as “permit” holders
  —Construct with an initial number of permits
  —acquire(): waits for permit to be available, then “takes” one, i.e., decrements the count of available permits
  —release(): “returns” a permit, i.e., increments the count of available permits
  —But no actual permits change hands
    —The semaphore just maintains the current count
    —Thread performing release() can be different from the thread performing acquire()

• “fair” variant hands out permits in FIFO order

• Useful for managing bounded access to a shared resource