Homework 2

Hanlin He (hxh160630)

October 8, 2018

Chapter 7

Exercise 85

Consider a thread acquired and release the lock for the first time. Let $qnodeX$ be the node created for the thread. After the thread released the lock, we will have:

- $tail = qnodeX$
- $qnodeX.locked = false$

If the same thread tries to acquire the lock again, the execution of `lock` method would be:

```
1 public void lock() {
2     Qnode qnode = myNode.get();          // qnode ← qnodeX
3     qnode.locked = true;                 // qnodeX.locked ← true
4     Qnode pred = tail.getAndSet(qnode); // pred ← qnodeX
5     while (pred.locked) {}       // get stuck in pred.locked forever
6 }
```

At line 4, return tail from `tail.getAndSet(qnode)` will return $qnodeX$ itself. And the *locked* value was already set to *true*, so the **`while`** loop will never break.

Exercise 91

- `testAndSet()` spin lock:

```
public boolean isLocked() {
    return state.get();
}
```

- CLH queue lock:

```
public boolean isLocked() {
    return tail.get().locked;
}
```

- MCS queue lock:

```java
public boolean isLocked() {
    return tail.get() != null;
}
```

Chapter 8

Exercise 95

```java
/**
 * 95-0. Interface definition.
 */
public interface Account {
  void deposit(int k);
  void withdraw(int k);
  int getBalance();
}
```

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 95-1. Implement this savings account using locks and conditions.
 */
public class SavingsAccount implements Account {

  private int balance;

  final Lock mutex = new ReentrantLock();

  private final Condition canWithdraw = this.mutex.newCondition();

  public SavingsAccount() {
    this(0);
  }

  public SavingsAccount(int balance) {
    this.balance = balance;
  }

  /**
   * Return current balance.
   */
  @Override
  public int getBalance() {
    this.mutex.lock();
    try {
      return this.balance;
    } finally {
      this.mutex.unlock();
```

```java
    }
  }

  /**
   * deposit(k) adds k to the balance
   */
  @Override
  public void deposit(int k) {
    this.mutex.lock();
    try {
      this.balance += k;
      this.canWithdraw.signalAll();
    } finally {
      this.mutex.unlock();
    }
  }

  /**
   * withdraw(k) subtracts k from balance, if the balance is at least k,
   * and otherwise blocks until the balance becomes k or greater
   */
  @Override
  public void withdraw(int k) {
    this.mutex.lock();
    try {
      while (k > this.balance) {
        this.canWithdraw.awaitUninterruptibly();
      }
      this.balance -= k;
    } finally {
      this.mutex.unlock();
    }
  }
}
```

```java
import java.util.concurrent.locks.Condition;

/**
 * 95-2. Implement two kinds of withdrawals for SavingsAccount: ordinary and preferred.
 */
public class SavingsAccountWithPreferredWithdraw extends SavingsAccount {

  private final Condition canOrdinaryWithdraw = this.mutex.newCondition();
  private int preferredWithdrawCount = 0;

  /**
   * ordinaryWithdraw perform withdrawal that yield to preferred withdrawal.
   */
  public void ordinaryWithdraw(int k) {
    // if preferred withdrawals waiting, wait for them to finish
    this.mutex.lock();
```

```java
    try {
      while (this.preferredWithdrawCount > 0) {
        this.canOrdinaryWithdraw.awaitUninterruptibly();
      }
      // if no preferred withdrawals waiting, perform withdraw
      this.withdraw(k);
    } finally {
      this.mutex.unlock();
    }
  }

  /**
   * preferredWithdraw perform withdrawal with higher priority.
   */
  public void preferredWithdraw(int k) {
    // increment preferred withdraw count using mutex
    this.mutex.lock();
    try {
      this.preferredWithdrawCount++;
    } finally {
      this.mutex.unlock();
    }

    // perform withdraw
    this.withdraw(k);

    // decrement preferred withdraw count using mutex
    this.mutex.lock();
    try {
      this.preferredWithdrawCount--;
      this.canOrdinaryWithdraw.signalAll();
    } finally {
      this.mutex.unlock();
    }
  }
}
```

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 95-3. Add a transfer() method that transfers a sum from one account
    to another.
 */
public class SavingsAccountWithTransfer extends SavingsAccount {

  private final Lock transferMutex = new ReentrantLock();

  /**
   * transfer() method transfers a sum from one account to another
   */
  public void transfer(int k, Account reserve) {
    this.transferMutex.lock();
    try {
      reserve.withdraw(k);
      this.deposit(k);
```

```
    } finally {
      this.transferMutex.unlock();
    }
  }

  /*
  If there are less than 10 threads for each account, all transfer
  methods called at 1:00 are guaranteed to return at 2:00.
   */
}
```

Chapter 8

Exercise 24

For the first history

Let H_1 denotes the history, α denotes the operation $A : r.\text{read}(1)$, β and γ denote the operations $B : r.\text{write}(1)$ and $B : r.\text{read}(2)$, and δ denotes the operation $C : r.\text{write}(2)$:

- Quiescently Consistent: Yes.

  After execution of α, β and γ, the program became quiescent. Thus, the history would be quiescently consistent if the result of δ appear after α, β and γ. Suppose γ was the last operation to take effect among α, β and γ, the history would be legal and quiescently consistent.

- Sequentially Consistent: Yes.

  Consider the sequential history shown in eq. (1).

  $$S_1 \equiv \beta \to \alpha \to \delta \to \gamma \tag{1}$$

  S_1 is apparently equivalent with H_1. And the object subhistory of r is legal. Thus, the history H_1 is sequentially consistent.

- Linearizable: Yes.

  Consider the sequential history in eq. (1) again. The preceding relations of H_1 is

  $$\to_{H_1} \equiv \{\alpha \to \gamma, \beta \to \gamma, \delta \to \gamma, \} \tag{2}$$

  Apparently, $\to_{H_1} \subseteq \to_{S_1}$. Thus, the history H is linearizable.

For the second history

Let H_2 denotes the history, α denotes the operation $A : r.\text{read}(1)$, β and γ denote the operations $B : r.\text{write}(1)$ and $B : r.\text{read}(1)$, and δ denotes the operation $C : r.\text{write}(2)$:

- Quiescently Consistent: Yes.

  After execution of α, β and γ, the program became quiescent. Thus, the history would be quiescently consistent if the result of δ appear after α, β and γ. Suppose β was the last operation to take effect among α, β and γ, the history would be legal and quiescently consistent.

- Sequentially Consistent: Yes.

  Consider the sequential history S_2 shown in eq. (3).

  $$S_2 \equiv \delta \rightarrow \beta \rightarrow \alpha \rightarrow \gamma \tag{3}$$

  S_2 is apparently equivalent with H_2. And the object subhistory of r is legal. Thus, the history H_2 is sequentially consistent.

- Linearizable: Yes.

  Consider the sequential history in eq. (3) again. The preceding relations of H_2 is

  $$\rightarrow_{H_2} \equiv \{\alpha \rightarrow \gamma, \beta \rightarrow \gamma, \delta \rightarrow \gamma, \} \tag{4}$$

  Apparently, $\rightarrow_{H_2} \subseteq \rightarrow_{S_2}$. Thus, the history H_2 is linearizable.

Exercise 27

Since Java does not guarantee linearizability, the definition of buffer would lead to linearizability problems.

```java
T[] items = (T[]) new Object[Integer.MAX_VALUE];
```

The `items` was defined as an array of generic type `T`, instead of an array of `AtomicReference<T>` or simply `AtomicReferenceArray<T>`. The changes of one element in `items` within one thread `items[slot] = x;` may not be noticeable to other threads immediately. So it is possible to have two threads T_1 and T_2 with a history H that:

$$H : T_1.enq(1) \rightarrow T_2.deq() \text{ throw empty exception} \tag{5}$$

because `value = items[slot]` in T_2 was still **`null`** after $T_1.enq(1)$. However, the legal sequence history for eq. (5) is:

$$S : T_2.deq() \text{ throw empty exception} \rightarrow T_1.enq(1) \tag{6}$$

Apparently, $\rightarrow_H \not\subseteq \rightarrow_S$, the implementation is not linearizable.