CSE 332: Data Abstractions

Lecture 22:
Deadlock
Readers/Writer Locks

Ruth Anderson
Winter 2015
Outline

Done:
- Programming with locks and critical sections
- Key guidelines and trade-offs

Now:
- Another common error: Deadlock
- Other common facilities useful for shared-memory concurrency
  - Readers/writer locks
Motivating Deadlock Issues

Consider a method to transfer money between bank accounts

```java
class BankAccount {
    ...
    synchronized void withdraw(int amt) {...}
    synchronized void deposit(int amt) {...}
    synchronized void transferTo(int amt,
    BankAccount a) {
        this.withdraw(amt);
        a.deposit(amt);
    }
}
```

Potential problems?
Motivating Deadlock Issues

Consider a method to transfer money between bank accounts

```java
class BankAccount {
    ...
    synchronized void withdraw(int amt) {...}
    synchronized void deposit(int amt) {...}
    synchronized void transferTo(int amt, BankAccount a) {
        this.withdraw(amt);
        a.deposit(amt);
    }
}
```

Notice during call to `a.deposit`, thread holds two locks
- Need to investigate when this may be a problem
The Deadlock

Suppose \( x \) and \( y \) are static fields holding accounts.

Thread 1: \( x\.transferTo(1,y) \)

- acquire lock for \( x \)
- do withdraw from \( x \)
- block on lock for \( y \)

Thread 2: \( y\.transferTo(1,x) \)

- acquire lock for \( y \)
- do withdraw from \( y \)
- block on lock for \( x \)
**Ex: The Dining Philosophers**

- 5 philosophers go out to dinner together at an Italian restaurant
- Sit at a round table; one fork per setting
- When the spaghetti comes, each philosopher proceeds to grab their right fork, then their left fork, then eats
- ‘Locking’ for each fork results in a **deadlock**
Deadlock, in general

A deadlock occurs when there are threads $T_1, \ldots, T_n$ such that:
- For $i=1,\ldots,n-1$, $T_i$ is waiting for a resource held by $T(i+1)$
- $T_n$ is waiting for a resource held by $T_1$

In other words, there is a cycle of waiting
  - Can formalize as a graph of dependencies with cycles bad

Deadlock avoidance in programming amounts to techniques to ensure a cycle can never arise
Back to our example

Options for deadlock-proof transfer:

1. Make a smaller critical section: `transferTo` not synchronized
   - Exposes intermediate state after `withdraw` before `deposit`
   - May be okay here, but exposes wrong total amount in bank

2. Coarsen lock granularity: one lock for all accounts allowing transfers between them
   - Works, but sacrifices concurrent deposits/withdrawals

3. Give every bank-account a unique number and always acquire locks in the same order
   - *Entire program* should obey this order to avoid cycles
   - Code acquiring only one lock can ignore the order
class BankAccount {
    ...
    private int acctNumber; // must be unique
    void transferTo(int amt, BankAccount a) {
        if (this.acctNumber < a.acctNumber) {
            synchronized(this) {
                synchronized(a) {
                    this.withdraw(amt);
                    a.deposit(amt);
                }
            }
        } else {
            synchronized(a) {
                synchronized(this) {
                    this.withdraw(amt);
                    a.deposit(amt);
                }
            }
        }
    }
}
Another example

From the Java standard library

class StringBuffer {
    private int count;
    private char[] value;
    ...
    synchronized append(StringBuffer sb) {
        int len = sb.length();
        if (this.count + len > this.value.length)
            this.expand(...);
        sb.getChars(0, len, this.value, this.count);
    }
    synchronized getChars(int x, int y,
        char[] a, int z) {
        "copy this.value[x..y] into a starting at z"
    }
}
Two problems

Problem #1: Lock for `sb` is not held between calls to `sb.length` and `sb.getChars`
   – So `sb` could get longer
   – Would cause `append` to throw an `ArrayBoundsException`

Problem #2: Deadlock potential if two threads try to `append` in opposite directions, just like in the bank-account first example

Not easy to fix both problems without extra copying:
   – Do not want unique ids on every `StringBuffer`
   – Do not want one lock for all `StringBuffer` objects

Actual Java library: fixed neither (left code as is; changed javadoc)
   – Up to clients to avoid such situations with own protocols
Perspective

• Code like account-transfer and string-buffer append are difficult to deal with for deadlock

• Easier case: different types of objects
  – Can document a fixed order among types
  – Example: “When moving an item from the hashtable to the work queue, never try to acquire the queue lock while holding the hashtable lock”

• Easier case: objects are in an acyclic structure
  – Can use the data structure to determine a fixed order
  – Example: “If holding a tree node’s lock, do not acquire other tree nodes’ locks unless they are children in the tree”
Outline

Done:
• Programming with locks and critical sections
• Key guidelines and trade-offs

Now:
• Another common error: Deadlock
• Other common facilities useful for shared-memory concurrency
  – Readers/writer locks
Reading vs. writing

Recall:
- Multiple concurrent reads of same memory: *Not* a problem
- Multiple concurrent writes of same memory: Problem
- Multiple 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:
- Could still allow multiple simultaneous readers!
Example

Consider a hashtable with one coarse-grained lock
- So only one thread can perform operations at a time
- Won’t allow simultaneous reads, even though it’s ok conceptually

But suppose:
- There are many simultaneous **lookup** operations
- **insert** operations are very rare
- It’d be nice to support multiple reads; we’d do lots of waiting otherwise

Note: Important that **lookup** does not actually mutate shared memory, like a move-to-front list operation would
Readers/writer locks

A new synchronization ADT: The readers/writer lock

- A lock’s states fall into three categories:
  - “not held”
  - “held for writing” by one thread
  - “held for reading” by *one or more* threads

- **new**: make a new lock, initially “not held”
- **acquire_write**: block if currently “held for reading” or “held for writing”, else make “held for writing”
- **release_write**: make “not held”
- **acquire_read**: block if currently “held for writing”, else make/keep “held for reading” and increment *readers count*
- **release_read**: decrement readers count, if 0, make “not held”
Pseudocode example (not Java)

class Hashtable<K,V> {
    ...
    // coarse-grained, one lock for table
    RWLock lk = new RWLock();
    V lookup(K key) {
        int bucket = hasher(key);
        lk.acquire_read();
        ... read array[bucket] ...
        lk.release_read();
    }
    void insert(K key, V val) {
        int bucket = hasher(key);
        lk.acquire_write();
        ... write array[bucket] ...
        lk.release_write();
    }
}
Readers/writer lock details

• A readers/writer lock implementation ("not our problem") usually gives *priority* to writers:
  – Once a writer blocks, no readers *arriving later* will get the lock before the writer
  – Otherwise an *insert* could *starve*
    • That is, it could wait indefinitely because of continuous stream of read requests

• Re-entrant?
  – Mostly an orthogonal issue
  – But some libraries support *upgrading* from reader to writer

• Why not use readers/writer locks with more fine-grained locking, like on each bucket?
  – Not wrong, but likely not worth it due to low contention
In Java

Java’s `synchronized` statement does not support readers/writer

Instead, library

`java.util.concurrent.locks.ReentrantReadWriteLock`

- Different interface: methods `readLock` and `writeLock` return objects that themselves have `lock` and `unlock` methods

- Does *not* have writer priority or reader-to-writer upgrading
  - Always read the documentation
Concurrency summary

- Concurrent programming allows multiple threads to access shared resources (e.g. hash table, work queue, grid in project 3)
- Introduces new kinds of bugs:
  - Data races
  - Critical sections too small
  - Critical sections use wrong locks
  - Deadlocks
- Requires synchronization
  - Locks for mutual exclusion (common, various flavors)
  - Condition variables for signaling others (less common, covered in notes)
- Guidelines for correct use help avoid common pitfalls
- Shared Memory model is not only approach, but other approaches (e.g., message passing) are not painless