CSE 332: Concurrency and Locks

Richard Anderson
Spring 2016

Announcements

- Project 3 available
- Wednesday and Friday: Concurrency
- Next week: Algorithms

Really sharing memory between Threads

Heap for all objects and static fields, shared by all threads

A bad interleaving

Interleaved withdraw(100) calls on the same account
- Assume initial balance == 150

How to fix?

No way to fix by rewriting the program
- can always find a bad interleaving → violation
- need some kind of synchronization

Class BankAccount

```java
public class BankAccount {
    private int balance = 0;
    public int getBalance() { return balance; }
    public void setBalance(int x) { balance = x; }
    public void withdraw(int amount) {
        int b = getBalance();
        if (amount > b) throw new WithdrawTooLargeException();
        setBalance(b - amount);
    }
    // other operations like deposit, etc.
}
```

Thread 1
```
x.withdraw(100);
```

Thread 2
```
x.withdraw(100);
```

Banking

Two threads both trying to withdraw(100) from the same account:
- Assume initial balance 150

```java
class BankAccount {
    private int balance = 0;
    public int getBalance() { return balance; }
    public void setBalance(int x) { balance = x; }
    public void withdraw(int amount) {
        int b = getBalance();
        if (amount > b) throw new WithdrawTooLargeException();
        setBalance(b - amount);
    }
    // other operations like deposit, etc.
}
```

Thread 1
```
x.withdraw(100);
```

Thread 2
```
x.withdraw(100);
```
Race Conditions

A race condition: program executes incorrectly due to unexpected order of threads

Two kinds
1. data race:
   - two threads write a variable at the same time
   - one thread writes, another reads simultaneously
2. bad interleaving: wrong result due to unexpected interleaving of statements in two or more threads

Concurrency

Concurrency:
Correctly and efficiently managing access to shared resources from multiple possibly-simultaneous clients

Requires coordination
- synchronization to avoid incorrect simultaneous access:
  - make others block (wait) until the resource is free

Concurrent applications are often non-deterministic
- how threads are scheduled affects what operations happen first
- non-repeatability complicates testing and debugging
  - must work for all possible interleavings!!

Concurrency Examples

• Bank Accounts
• Airline/hotel reservations
• Wikipedia
• Facebook
• Databases

Locks

• Allow access by at most one thread at a time
  - "mutual exclusion"
  - make others block (wait) until the resource is free
  - called a mutual-exclusion lock or just lock, for short

• Critical sections
  - code that requires mutual exclusion
  - defined by the programmer (compiler can’t figure this out)

Lock ADT

We define Lock as an ADT with operations:
- new: make a new lock, initially “not held”
- acquire: blocks if this lock is already currently “held”
  - Once "not held", makes lock "held" (one thread gets it)
- release: makes this lock “not held”
  - If >= 1 threads are blocked on it, exactly 1 will acquire it
  - Allow access to at most one thread at a time

How can this be implemented?
- acquire (check “not held” -> make “held”) cannot be interrupted
- special hardware and operating system-level support

Basic idea (note Lock is not an actual Java class)

```java
class BankAccount {
    private int balance = 0;
    private Lock lk = new Lock();
    ...
    void withdraw(int amount) {
        lk.acquire(); // may block
        int b = getBalance();
        if(amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
        lk.release();
    }
    // deposit would also acquire/release lk
}
```
Common Mistakes

- Forgetting to release locks
  - e.g., because of Throws (previous slide)
- Too few locks
  - e.g., all bank accounts share a single lock
- Too many locks
  - separate locks for deposit, withdraw

What Do We Lock?

- Class
  - e.g., all bank accounts?
- Object
  - e.g., a particular account?
- Field
  - e.g., balance
- Code fragment
  - e.g., withdraw

Synchronized: Locks in Java

Java has built-in support for locks

```java
synchronized (expression) {
    statements
}
```

1. *expression evaluates to an object*
   - Any object (but not primitive types) can be a lock in Java
2. Acquires the lock, blocking if necessary
   - If you get past the {}, you have the lock
3. Releases the lock at the matching }
   - even if control leaves due to throw, return, etc.
   - so impossible to forget to release the lock

BankAccount in Java

```java
class BankAccount {
    private int balance = 0;
    private Object lk = new Object();
    int getBalance() {
        synchronized (lk) { return balance; }
    }
    void setBalance(int x) {
        synchronized (lk) {
            balance = x;
        }
    }
    void withdraw(int amount) {
        synchronized (lk) {
            int b = getBalance();
            if(amount > b) {
                throw ...
            }
            setBalance(b - amount);
        }
    }
    // deposit would also use synchronized(lk)
}
```

Shorthand

Usually simplest to use the class object itself as the lock

```java
synchronized (this) {
    statements
}
```

This is so common that Java provides a shorthand:

```java
synchronized {
    statements
}
```

Final Version

```java
class BankAccount {
    private int balance = 0;
    synchronized int getBalance() {
        return balance;
    }
    synchronized void setBalance(int x) {
        balance = x;
    }
    synchronized void withdraw(int amount) {
        int b = getBalance();
        if(amount > b) {
            throw ...
        }
        setBalance(b - amount);
    }
    // deposit would also use synchronized
}
```
Stack Example

```java
class Stack<E> {
    private E[] array = (E[])new Object[SIZE];
    int index = -1;
    boolean isEmpty() {
        return index==-1;
    }
    void push(E val) {
        array[++index] = val;
    }
    E pop() {
        if(isEmpty())
            throw new StackEmptyException();
        return array[index--];
    }
}
```

Why Wrong?

- isEmpty and push are one-liners. What can go wrong?
  - ans: one line, but multiple operations
  - array[++index] = val probably takes at least two ops
  - data race if two pushes happen simultaneously

Stack Example (fixed)

```java
class Stack<E> {
    private E[] array = (E[])new Object[SIZE];
    int index = -1;
    synchronize boolean isEmpty() {
        return index==-1;
    }
    synchronize void push(E val) {
        array[++index] = val;
    }
    synchronize E pop() {
        if(isEmpty())
            throw new StackEmptyException();
        return array[index--];
    }
}
```

Lock everything? No.

For every memory location (e.g., object field), obey at least one of the following:

1. **Thread-local**: only one thread sees it
2. **Immutable**: read-only
3. **Shared-and-mutable**: control access via a lock

Thread local

Whenever possible, do **not** share resources
- easier to give each thread its own local copy
- only works if threads don’t need to communicate via resource

In typical concurrent programs, the vast majority of objects should be thread local: shared memory should be rare—minimize it

Immutable

If location is read-only, no synchronizatin is necessary

Whenever possible, do **not** update objects
- make new objects instead!
- one of the key tenets of functional programming (CSE 341)

In practice, programmers usually over-use mutation—minimize it
The rest: keep it synchronized

Other Forms of Locking in Java

- Java provides many other features and details. See, for example:
  - Chapter 14 of CoreJava, Volume 1 by Horstmann/Cornell
  - Java Concurrency in Practice by Goetz et al