Lecture 31: Concurrency I

CS 62
Spring 2019
William Devanny & Alexandra Papoutsaki

Some slides based on those from Dan Grossman, U. of Washington
Concurrent Programming

- Allowing simultaneous or interleaved access to shared resources from multiple clients.
- Requires coordination, particularly synchronization to avoid incorrect simultaneous access: make somebody block
  - `join` is not what we want
  - `block` until another thread is “done using what we need” not “completely done executing”
Very complicated, very quickly

- Concurrent code gets very complicated very quickly. Why?
- Concurrency introduces non-determinism!
- In sequential programming, when you run the same program multiple times, you get the same result.
- This is no longer true for concurrent programs. Threads can run in any order giving unpredictable results.
- How threads are scheduled affects *what* operations from other threads they see and *when* they see them.
- Non-repeatability complicates testing and debugging.
Examples

• Multiple threads:
  • Processing different bank-account operations
  • What if 2 threads change the same account at the same time?

• Using a shared cache of recent files
  • What if 2 threads insert the same file at the same time?

• Creating pipeline with queue for handing work to next thread in sequence?
  • What if enqueuer and dequeuer adjust a circular array queue at the same time?
Threads again?!

• Not about speed, but code structure for responsiveness
• Example: Respond to GUI events in one thread while another thread is performing an expensive computation
• Processor utilization (mask I/O latency)
  • If 1 thread “goes to disk,” have something else to do
• Failure isolation
  • Convenient structure if we want to interleave multiple tasks and don’t want an exception in one to stop the other
Sharing is caring

- Common to have different threads access the same resources in an unpredictable order or even at about the same time
- But program correctness requires that simultaneous access be prevented using synchronization
- Simultaneous access is rare
  - Makes testing difficult
  - Must be much more disciplined when designing / implementing a concurrent program
  - We will discuss common idioms known to work
class BankAccount {
    private int balance = 0;
    int getBalance() {
        return balance;
    }
    void setBalance(int x) {
        balance = x;
    }
    void withdraw(int amount) {
        int b = getBalance();
        if(amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
    }
    // ... other operations like deposit, etc.
}
Canonical Example - Bad interleavings

Interleaved `withdraw(100)` calls on the same account
Assume initial `balance` is 150

```java
Thread 1
-------
int b = getBalance();

Thread 2
-------
int b = getBalance();
if(amount > b)
    throw new ...;
setBalance(b - amount); // sets balance to 50
if(amount > b) // no exception: b holds 150
    throw new ...;
setBalance(b - amount);
```
Interleaving is the problem

• Suppose:
  • Thread T1 calls `withdraw(100)`
  • Thread T2 calls `withdraw(100)`

• If second call starts before first finishes, we say the calls interleave
  • Could happen even with one processor since a thread can be pre-empted at any point for time-slicing

• If x and y refer to different accounts, no problem
  • “You cook in your kitchen while I cook in mine”
  • But if x and y alias, possible trouble...
First attempt to fix the problem

It is tempting and almost always **wrong** to fix a bad interleaving by rearranging or repeating operations, such as:

```java
void withdraw(int amount) {
    if (amount > getBalance())
        throw new WithdrawTooLargeException();
    // maybe balance changed, so get the new balance
    setBalance(getBalance() - amount);
}
```

Just because statement is on one line does not mean it happens all at once!
What we want: Mutual exclusion

• The fix: Allow at most one thread to withdraw from account $A$ at a time
  • Exclude other simultaneous operations on $A$ too (e.g., deposit)
• Called mutual exclusion:
  • One thread using a resource (here: a bank account) means another thread must wait
  • We call the area of code that we want to have mutual exclusion (only one thread can be there at a time) a critical section.
• Programmer (you!) must implement critical sections:
  • “The compiler” has no idea what interleavings should or should not be allowed in your program
  • But you need language primitives to do it!
Our own mutual-exclusion protocol?

- Say we tried to coordinate it ourselves using a boolean `busy`

```java
class BankAccount {
    private int balance = 0;
    private boolean busy = false;
    void withdraw(int amount) {
        while(busy) { /* spin-wait */ }
        busy = true;
        int b = getBalance();
        if(amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
        busy = false;
    }
    // deposit would spin on same boolean
}
```

- We can check that `busy` is false, but then it might get set to true before we have a chance to set it to true ourselves.
What we need

• **Mutual-Exclusion Locks** (aka *Mutex*, or just *Lock*)
  • Still on a conceptual level at the moment, *Lock* is not a Java class (though Java’s approach is similar)

• We will 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*” [all at once!]
    Checking & setting happen together, and cannot be interrupted – Fixes problem we saw before!!
  • `release`: makes this lock “*not held*”
    If >= 1 threads are blocked on it, exactly 1 will acquire it
Why that works?

• The lock implementation ensures that given simultaneous acquires and/or releases, a correct thing will happen

• Example:
  • If we have two acquires: one will “win” and one will block

• How can this be implemented?
  • Need to “check if held and if not make held” “all-at-once”
  • Uses special hardware and O/S support
  • More in upper division classes on computer-architecture or operating-systems
  • Here, we will use a language primitive
Almost-correct pseudocode

```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();
    }
}
```

- Problem occurs if `amount > b`. An exception is thrown and lock is never released. Stuck in forever-waiting land
- Assuming `getBalance` and `setBalance` are public, they should also acquire and release the lock.
Re-entrant Lock idea

• A re-entrant lock (a.k.a. recursive lock)

• The idea: Once acquired, the lock is held by the Thread, and subsequent calls to acquire in that Thread won’t block

• Result: withdraw can acquire the lock, and then call setBalance, which can also acquire the lock
  • Because they’re in the same thread & it’s a re-entrant lock, the inner acquire won’t block!!
Re-entrant Lock

• ”Remembers”
  • The thread (if any) that currently holds it
  • a count

• When the lock goes from not-held to held, the count is set to 0

• If (code running in) the current holder calls acquire:
  • it does not block
  • it increments the count

• On release:
  • if the count is > 0, the count is decremented
  • if the count is 0, the lock becomes not-held
Re-entrant locks work

- This simple code works fine provided \( \text{lk} \) is a re-entrant lock
- Okay to call `setBalance` directly
- Okay to call `withdraw` (won’t block forever)

```c
int setBalance(int x) {
    lk.acquire();
    balance = x;
    lk.release();
}
void withdraw(int amount) {
    lk.acquire();
    ...
    setBalance(b - amount);
    lk.release();
}
```
Java’s re-entrant locks

- `java.util.concurrent.locks.ReentrantLock`
- Has methods `lock()` and `unlock()`
- Conceptually owned by the Thread, and shared within that thread
- Important to guarantee that lock is always released!!!
- Recommend something like this:
  ```java
  myLock.lock();
  try {
    // method body
  } finally {
    myLock.unlock();
  }
  ```
- Despite what happens in `try`, the code in `finally` will execute afterwards
Synchronized in Java

- Java has built-in support for re-entrant locks
- You can use the `synchronized` statement as an alternative to declaring a `ReentrantLock`
- `synchronized (expression) { statements }`
- 1. Evaluates expression to an object
   - Every object (but not primitive types) “is 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!
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 new WithdrawTooLargeException();
            setBalance(b - amount);
        }
    }
    // deposit and other operations would also use synchronized(lk)
}
What’s the problem?

• As written, the lock is private
  • Might seem like a good idea
  • But also prevents code in other classes from writing operations that synchronize with the account operations

• More idiomatic is to synchronize on this...
  • Also more convenient: no need to have an extra object!
class BankAccount {
    private int balance = 0;
    int getBalance() {
        synchronized (this) {
            return balance;
        }
    }
    void setBalance(int x) {
        synchronized (this) {
            balance = x;
        }
    }
    void withdraw(int amount) {
        synchronized (this) {
            int b = getBalance();
            if(amount > b)
                throw new WithdrawTooLargeException();
            setBalance(b - amount);
        }
        // deposit and other operations would also use synchronized(this)
    }
}
• Version #2 is slightly poor style because there is a shorter way to say the same thing
• Putting `synchronized` before a method declaration means the entire method body is surrounded by `synchronized(this){...}
• Therefore, version #3 (next slide) means exactly the same thing as version #2 but is more concise
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 new WithdrawTooLargeException();
        setBalance(b - amount);
    }
    // deposit and other operations would also be declared synchronized
}