# CSE332: Data Abstractions

# Lecture 22: Shared-Memory Concurrency and Mutual Exclusion

Tyler Robison

Summer 2010

# Toward sharing resources (memory)

So far we've looked at parallel algorithms using fork-join

ForkJoin algorithms all had a very simple *structure* to avoid race conditions

- Each thread had memory "only it accessed"
  - Example: array sub-range
  - Array variable itself was treated as 'read-only' in parallel portion
- Result of forked process not accessed until after join() called
- So the structure (mostly) ensured that bad simultaneous access wouldn't occur

Strategy won't work well when:

- Memory accessed by threads is overlapping or unpredictable
- Threads are doing independent tasks needing access to same resources (rather than implementing the same algorithm)

We'll need to coordinate resources for them to be of use

# What could go wrong?

```
enqueue(x) {
   if(back==null){
        back=new Node(x);
        front=back;
   }
   else{
        back.next = new Node(x);
        back = back.next;
   }
}
```

- Imagine 2 threads, running at the same time, both with access to a shared linked-list based queue (initially empty)
- Each own program counter (and heap, etc.)
- Queue is shared, so they both indirectly use the same ‘front’ and ‘back’ (which is the whole point of sharing the queue)
- We have no guarantee what happens first between different threads; can (and will) arbitrarily ‘interrupt’ each other
- Many things can go wrong: say, one tries to enqueue “a”, the other “b”, and both verify that back is ‘null’ before other sets back
  - Result: One assignment of back will be ‘forgotten’
- In general, any ‘interleaving’ of results is possible if enqueue were called at the same time for both

# Concurrent Programming

***Concurrency***: Allowing simultaneous or interleaved access to shared resources from multiple clients

Requires *coordination*, particularly synchronization to avoid incorrect simultaneous access: make somebody *block* (wait) until resource is free

- `join` isn't going to work here
- We want to block until another thread is "done using what we need" not "completely done executing"

Even correct concurrent applications are usually highly non-deterministic

- How threads are scheduled affects what operations from other threads they see when
- Non-repeatability complicates testing and debugging

# Why threads?

Use of threads not always to increase performance (though they can be)

Also used for:

- *Code structure for responsiveness*
  - Example: Respond to GUI events in one thread while another thread is performing an expensive computation
- *Failure isolation*
  - Convenient structure if want to *interleave* multiple tasks and don't want an exception in one to stop the other

# Canonical example

- Simple code for a bank account
- Correct in a single-threaded world

```
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.
}
```

# Interleaving

Suppose we have 2 threads, T1 & T2:

- Thread **T1** calls **`x.withdraw(100)`**
- Thread **T2** calls **`y.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
  - T1 runs for 50 ms, pauses somewhere, T2 picks up for 50ms

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, weird things can occur

# A bad interleaving

Imagine two interleaved **`withdraw(100)`** calls on the same account

- Assume initial **`balance`** 150
- From the code we saw before, this ***should*** cause a `WithdrawTooLarge` exception

Thread 1

```
int b = getBalance();




if(amount > b)
  throw new …;
setBalance(b – amount);
```

Thread 2

```
int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b – amount);
```

Time

Instead of an exception, we have a "Lost withdraw"

**But if we had 'if(amount>getBalance())' instead, this wouldn't have happened… right?**

# Incorrect "fix"

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

```
void withdraw(int amount) {
  if(amount > getBalance())
    throw new WithdrawTooLargeException();
  // maybe balance changed
  setBalance(getBalance() – amount);
}
```

This fixes nothing!

- Narrows the problem by one statement
- (Not even that since the compiler could turn it back into the old version because you didn't indicate need to synchronize)
- And now a negative balance is possible – why?

# Mutual exclusion

The sane fix: At most one thread withdraws from account **A** at a time

- Exclude other simultaneous operations on **A** too (e.g., deposit)
- Other combinations of simultaneous operations on 'balance' could break things
- 'One at a time' is embodied in the idea of 'mutual exclusion'

***Mutual exclusion***: One thread doing something with a resource (here: an account) means another thread must wait

- Define 'critical sections'; areas of code that are mutually exclusive

Programmer (that is, *you*) must implement critical sections

- "The compiler" has no idea what interleavings should or shouldn't be allowed in your program
- Buy you need language primitives to do it!
- Like with Thread start() & join(), you can't implement these yourself in Java

# Wrong!

Why can't we implement our own mutual-exclusion protocol?

- Say we tried to coordinate it ourselves, using 'busy':

```
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
}
```

# Still just moved the problem!

Thread 1

```
while(busy) { }

busy = true;

int b = getBalance();




if(amount > b)
   throw new …;
setBalance(b - amount);
```

Thread 2

```

while(busy) { }

busy = true;

int b = getBalance();
if(amount > b)
   throw new …;
setBalance(b - amount);
```

Time

"Lost withdraw" – unhappy bank

**Time does elapse between checking 'busy' and setting 'busy'; can be interrupted there**

# What we need

- To resolve this issue, we'll need help from the language

- One basic solution: Locks
  - Still on a conceptual level at the moment, 'Lock' is not a Java class*

- An ADT with operations:
  - `new`: make a new lock
  - `acquire`: If lock is *"not held"*, makes it *"held"*
    - Blocks if this lock is already *"held"*
    - Checking & setting happen together, and cannot be interrupted
    - Fixes problem we saw before
  - `release`: makes this lock *"not held"*
    - If multiple threads are blocked on it, exactly 1 will acquire it

# Why that works

- Lock: ADT with operations **`new`**, **`acquire`**, **`release`**
- 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 and update” “all-at-once”
  - Uses special hardware and O/S support
    - See CSE471 and CSE451
  - In CSE332, we take this as a primitive and use it

# Almost-correct pseudocode

**Note: 'Lock' is not an actual Java class**

```
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
}
```

**One problem with this code…**

# Some potential Lock mistakes

- A lock is a very primitive mechanism
  - Still up to you to use correctly to implement critical sections
  - Lots of little things can go wrong, and completely break your program
- Incorrect: Forget to release a lock (blocks other threads forever!)
  - Previous slide is wrong because of the exception possibility!

```
if(amount > b) {
  lk.release(); // hard to remember!
  throw new WithdrawTooLargeException();
}
```

- Incorrect: Use different locks for **`withdraw`** and **`deposit`**
  - Mutual exclusion works only when using same lock
  - With one lock for each, we could have a simultaneous withdraw & deposit; could still break
- Poor performance: Use same lock for every bank account
  - No simultaneous withdrawals from different accounts

# Other operations

- If **`withdraw`** and **`deposit`** use the same lock (and they use it correctly), then simultaneous calls to these methods are properly synchronized

- But what about **`getBalance`** and **`setBalance`**?
  - Assume they're **`public`**, which may be reasonable

- If they don't acquire the same lock, then a race between **`setBalance`** and **`withdraw`** could produce a wrong result

- If they do acquire the same lock, then **`withdraw`** would block forever because it tries to acquire a lock it already has

# One (not very good) possibility

```
int setBalance1(int x) {
  balance = x;
}
int setBalance2(int x) {
  lk.acquire();
  balance = x;
  lk.release();
}
void withdraw(int amount) {
  lk.acquire();
  …
  setBalanceX(b - amount);
  lk.release();
}
```

- Can't let outside world call **`setBalance1`**
- Can't have **`withdraw`** call **`setBalance2`**
- Could work (if adhered to), but not good style; also not very convenient
- Alternately, we can modify the meaning of the Lock ADT to support *re-entrant locks*
  - Java does this

# Re-entrant lock

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
- "Remembers"
  - the thread (if any) that currently holds it
  - a *count*
- When the lock goes from *not-held* to *held*, the count is 0
- If code in the holding Thread 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*
- 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

# Java's Re-entrant Lock

- **`java.util.concurrent.ReentrantLock`**
- Has methods `lock()` and `unlock()`
- As described above, it is conceptually owned by the Thread, and shared within that
- Important to guarantee that lock is ***always*** released; recommend something like this:

```
lock.lock();
try { // method body }
finally { lock.unlock(); }
```

- Despite what happens in 'try', the code in finally will execute afterwards

# Synchronized: A Java convenience

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, uses it as a lock
   - 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

# Example of Java's **synchronized**

```
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)
}
```

# Improving the Java

- 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 common is to synchronize on `this`…
  - Also, it's convenient; don't need to declare an extra object

# Java version #2

```
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 …
      setBalance(b – amount);
    }
  }
  // deposit would also use synchronized(this)
}
```

# Syntactic sugar

`synchronized (this)` is sufficiently common that there is an even simpler way to do it in Java:

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

# Java version #3 (final version)

```
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
}
```