Java Concurrency In a Nutshell

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Why A Lecture On "Java Concurrency"?
2. Transitioning To Multi-Threaded Code
3. Race Conditions: The Problem(s)
4. Synchronization: The Solution
5. Concurrency Programming Guidelines
6. Creating & Managing Multiple Clients
7. Some Closing Thoughts

- Note: we will likely not finish this material in class.
- Because this material is not "core" *Database Implementation* material, am not allocating more than one lecture
- For your (project implementation etc) sake: review (and think about) all the material

Much material derived from lectures created by Professor Dan Grossman, Paul G. Allen School of Computer Science & Engineering, University of Washington with permission

Why A Lecture On “Java Concurrency”?

How Is “Concurrency” Relevant To “Database Implementation”?

- As you probably know, this course is not *Parallel Programming* ☺
- Q: so, why a lecture on Java concurrency?
- A: because databases, like all “service providing” software, must perform well
  - Or: they won’t have any customers ☺
- Throughput: a fundamental performance metric
  - *“How many client requests serviced per given time unit?”*
- Implication: your DBMS must support concurrent, multi-client access to its data
- Your implementation must therefore address a fundamental tension
  1. Must allow concurrent access (“the more clients, the better”)
  2. Must ensure that concurrent access doesn’t corrupt data-structures (“the fewer clients, the easier to implement”)

Why Java Concurrency?

- Having established that understanding concurrency issues & solutions is relevant to our course …
- Why: Java concurrency?
  - After all: many (most?) enterprise databases are written in C or C++
  - (See this discussion, for example)
- More importantly: the Java concurrency model is definitely different from C/C++
  - Java concurrency libraries are irrelevant to other languages ☹
- Frankly: was tempted to ignore the whole issue (as the textbook does ☺)
- But: you'll be implementing the course project in Java
  - To succeed, you must be competent in and comfortable with Java concurrency
  - Otherwise: your DBMS & my test cases will not get along ☹

Concurrency & “Database Implementation”

> You will have to teach yourself Java concurrency skills as you complete the course project

- This lecture will <u>not</u> teach you all or even much of what you need to know about Java concurrency
  - Consider taking *Parallel Programming* ☺
- Today’s lecture is a guided tour, tailored to helping you succeed in the course project
  - Make you aware of the pitfalls and some solution techniques
  - Steer you away from useless or incorrect information on the Internet

Broader Importance Of This Material

- Any server-side software must allow for (and correctly manage) concurrency
- Keep in mind
  - Java is a heavy-duty, industrial-strength language
  - Its approach to concurrency is one of its strengths
  - Being able to do concurrent programming is a key differentiator between amateur and professional programmers
- I want you to incorporate these skills into your software portfolio
- Final point:
  - Yes: the solutions we'll discuss today are specific to Java
  - But: the issues in (shared memory) concurrent programming that we're discussing are language independent

Historical Perspective (I)

- Primary goal of using concurrency is maximizing use of CPU (s)
- But: the strategy for maximizing CPU usage has changed over time
- Single-core era: use non-blocking I/O as an alternative to "blocking" code
  - Also: prioritized background tasks
- Multi-core era: Coarse-grained, task-based concurrency
  - Largely about throughput: pushing more requests through a server
  - That's what we'll focus on for this course
- Many-core era: Fine-grained data parallelism
  - Largely about latency: "use more cores to get the answer faster"

Credits to Brian Goetz, Java Language Architect for these insights

Hardware Trends Drive Software Trends

- Hardware (previous slide) shapes the languages, library, and frameworks we write
- Java 1: supported threads, locks, condition queues
- Java 5: added thread pools, blocking queues, concurrent collections
- Java 7: added the fork-join library
- Java 8: added parallel streams

> - You therefore want to focus on the Java 5 additions
> - `java.util.concurrent` and sub-packages (`atomic` and `locks`)

Transitioning To Multi-Threaded Code

Multi-Threading: Major Paradigm Shift (I)

- Since COM 1300, you've been "code-centric"
  1. You write code
  2. Computer executes that code "one line at a time"
- Your mind-set (for coding & debugging): *"I only have to worry about a single line of code at any given moment"*
  - Because the computer will execute exactly one line of code at any given moment

That mind-set is (mostly) incorrect ☺

Multi-Threading: Major Paradigm Shift (II)

- What was really going on:
  1. You write code
  2. You create a thread: an independent execution "engine"
  3. That thread executes your code "one line at a time"
- Key point:
  - You can create multiple threads, and then direct each thread to independently execute that code
- Until now:
  - By default: the `java` command creates only one thread to execute your program
  - That thread is labeled as `main`
  - Typically, one thread suffices to get your work done, but not any more ☺

Motivation: Performance! (I)

- I've already motivated this shift in your programming paradigm: performance, performance, performance
- Contrast to (for example) a *Data Structures* project
  - Your focus: write correct code that meets requirements
  - Elegant code is a bonus
- *Introduction to Algorithms* added the idea of "pick the right algorithm"
  - To break the "Big-O" barrier
  - But fundamentally: only one thread of computation is executing your code
- Now: your DBMS code is a "service", available to multiple, independent clients
  - In other words: each of these "independent clients" is (in OS terms) a thread (or a process)
  - Because **OATSdb** is an embedded database, each of these "independent clients" is a Java thread (not a process)

Motivation: Performance! (II)

- Theoretically: your DBMS could insist "I'll service only one client at a time"
  - This would definitely make it much easier to write "correct & elegant code" ☺
  - But: your throughput metric will be very disappointing
- Implication

  - Your development test-bed will have to create and manage multiple clients (threads)
  - Your DBMS implementation will have to ensure that these multiple threads don't corrupt your data-structures

- We say: *"your DBMS must support concurrency"*

Key "Take-Aways" At This Point

- Stop thinking of your code as a "live" entity that can do only one thing at a given moment
  - Example: a *Stack* which can do either `pop` or `push` but not both at the same time
- Instead: your code is a "static" entity that is "infused with life" by being executed by a Thread
- If multiple threads execute your code, any number of your code points can be executed at the same time
  - Example: `Stack.pop` and `Stack.push`
- Murphy's Law says: the worst possible combination of concurrent executions will occur ☹

1. Explain why multi-threaded programs require a paradigm shift on your part
    - In other words: the “concurrency problem”
2. Present solutions to the “concurrency problem”
3. Turn from the “server-side” DBMS issues to “client-side” test-bed issues

Race Conditions: The Problem(s)

Since COM 1300 you've become accustomed to rely on the notion of invariants to establish program correctness

> *In computer programming, specifically object-oriented programming, a class invariant (or type invariant) is an invariant used for constraining objects of a class. Methods of the class should preserve the invariant. The class invariant constrains the state stored in the object.*
>
> *Class invariants are established during construction and constantly maintained between calls to public methods. Code within functions may break invariants as long as the invariants are restored before a public function ends.*

Source: Wikipedia

Bad Interleavings: Method Boundaries May Not Preserve Invariants (II)

- Once we enter the world of multi-threaded programming, we can no longer rely on invariants to establish program correctness
- In reality, methods are not executed atomically
- $Thread_1$, executing $method_1$, may have its execution interleaved with $Thread_2$, executing $method_2$
- Implication
  - $thread_1$ may see “intermediate” state affected by $method_2$ execution
  - $thread_2$ may see “intermediate” state affected by $method_1$ execution

Bad Interleavings: Shared State Transitions Are Now Exposed

- The whole point of "object-oriented" programming is to encapsulate shared state
- Concurrent thread executions imply that (unless we're careful) we've broken encapsulation ☹
- Typical scenarios of different methods accessing shared state
  - One thread deposits money in a bank-account, other thread does a withdrawal
  - "Producer" thread enqueues a task in a queue, "consumer" thread dequeues a task
  - One thread does a *Hashtable.get*, other thread does a *Hashtable.put*
- Method abstraction conceals the fact that e.g., a single *Hashtable.get* is "really" multiple lower-level machine instructions
  - And thus can be interleaved with the execution of *Hashtable.put*
  - With disastrous results ☹

This code is absolutely correct in a single-threaded environment

```java
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.
} // class BankAccount
```

Multi-Threaded Environment: Interleaving

Scenario (*x* and *y* are *BankAccount* instances):

- Thread T_1 calls x.withdraw(100)
- Thread T_2 calls y.withdraw(100)

- If T_1 and T_2 executions overlap (to any extent) , we say the executions interleave
- Because of OS "time-slicing", can happen even with a single processor
- No problem if the threads are accessing different account instances
- But: if the threads access the same account, we have absolutely no idea what's going to happen

Interleaving: One Scenario

Thread 1	Thread 2
`int b = getBalance();`	
	`int b = getBalance();`
	`if(amount > b)`
	`  throw new …;`
	`setBalance(b - amount);`
`if(amount > b)`	
`  throw new …;`	
`setBalance(b - amount);`	

- Both withdrawals are successful, customer gets to “double-dip”
- This is a “lost-withdrawal” scenario, bank is very unhappy ☹

Rearranging Code Doesn’t Work

```
void withdraw(int amount) {
  if (amount > getBalance()) {
    throw new WithdrawTooLargeException();
  }
  setBalance(getBalance() - amount);
}
```

- Rearranging or repeating code does not solve interleaving problems
- Here: only narrows the exposure, but other thread can still change the balance between second call to *getBalance* and reducing by *amount*
  - May not even do that ☹…since compiler may optimize back to original version
- May even change to “negative balance” scenario
  - Second call to *getBalance* occurs after other thread has already reduced balance
- Again: you simply can’t assume that this won’t happen

“Race” Conditions (I)

A race condition occurs when the computation result depends on scheduling: specifically, how hardware & software interact to interleave the concurrent execution of threads

- I’m only introducing the term “race condition” because it’s so prevalent
- Unfortunately, it’s an overloaded term
  - It refers to “bad” interleavings (the problem we’ve been discussing so far)
  - And it refers to data races: a simultaneous read/write or write/write (by two threads) of the same memory location
- The common denominator of these two types of race conditions is that they’re concurrency bugs
  - If only one thread, then no problem manifests

“Race” Conditions (II)

The good news: we’ll use the same solution approach to both “bad interleavings” and “data races”

- (I’m therefore going to postpone discussion of “data race” problem until we’ve discussed this solution)
- Conceptual difference between “data race” and “bad interleaving” is that a “data race” condition is always an error!
- In contrast: only you (the programmer) can determine whether an interleaving is “bad”
- Depends on how you’ve defined your invariants (the “code specification”)
  - “Bad interleaving” exists *iff* your code exposes an intermediate state that violates an invariant

Segue: Solution For Race Conditions Is Mutual Exclusion

- Think of it this way: our concurrency problems are caused by allowing independent threads to run wild in our code
- Solution idea: impose restrictions on thread executions
  - Specifically: annotate our code to state *“only one thread can execute this block of code at a time”*
- Example: allow at most one thread to withdraw from account at a time
  - Note: this requires that we also exclude other concurrent operations such as *deposit*
- This technique is called mutual exclusion or the creation of critical sections

Synchronization: The Solution

Support For Mutual Exclusion

- Unfortunately: mutual exclusion cannot be enforced by the compiler ☹
  - The semantics are at a higher-level than the compiler understand
  - How <u>can</u> the compiler possibly understand what interleavings are OK and which ones are dangerous?
- We do need, and can get, support from Java language primitives
  - The language must be able to enforce mutual exclusion

One Approach: Locks or Mutex Variables

- Some languages and libraries use the concept of lock or mutex variable to provide mutual exclusion
- Conceptually: two methods
  - `acquire`
  - `release`
- A critical section (such as accessing an account's `balance` variable) is ...
- ...Preceded by a call to `acquire` (*thread*$_2$ will be forced to block if *thread*$_1$ currently holds the lock)
- ...Followed by a call to `release` (*thread*$_1$ will release the lock after it modifies `balance`, allowing *thread*$_2$ to acquire the lock and proceed)
- Locks require special hardware and OS support to implement
  - We're requiring that *"check if lock is held and if not held acquire the lock"* be an atomic operation

Lock Variables: Getting It Wrong

- Wrong: if you use different locks for `withdraw` and `deposit`
  - Mutual exclusion works only when the shared resource being protected (`balance` instance variable) is protected by the same lock
- Bad Performance: too coarse a lock granularity
  - Example: don't use one lock for all bank accounts ☺
- Wrong: if *thread*$_1$ "forgets" to release a lock, then it will block other threads forever
- Example:
  1. Thread acquires lock
  2. Thread invokes `withdraw`
  3. Withdrawal attempt triggers *WithdrawTooLargeException*
  4. The lock will never be released
  5. All threads are prevented from using this code forever

Locks: Reentrantcy Problem

- Assume you've used locks correctly: `withdraw` and `deposit` use the same lock
- Now you implement `getBalance` and `setBalance`
- Two scenarios
  1. The new code uses a different lock than the "withdraw" lock
     - Incorrect: you've enabled a race condition between `setBalance` and `withdraw`
  2. The new code uses the same lock as the "withdraw lock"
     - 2.1 `withdraw` acquires the lock
     - 2.2 `withdraw` (internally) invokes `getBalance`
     - 2.3 Thread blocks forever as it tries to acquire the `getBalance` which it already has ☹
  3. One solution: allow locks to be reentrant

Reentrant Locks: Semantics

- A reentrant lock (sometimes called a *recursive lock*)
  - "Remembers" <u>which</u> thread has acquired it
  - Maintains a `count` variable
- When locks goes from "not held" to "acquired" state ...
  - `count` is set to <u>1</u>
- If acquiring thread (re)invokes "acquire" on this lock ...
  - Thread does not block
  - Instead: `count` is incremented
- When thread invokes "release" on this lock ...
  - `count` is decremented
  - If `count` now reaches <u>0</u>, lock enters "not held" state

- I've included this discussion of the lock approach because
  - Many languages and libraries use it
  - Because this is the approach used in the JDK 5 *java.util.concurrent.locks.ReentrantLock* class(es) ☺
  - Because it will help you understand the Java language solution: object monitors
- My opinion only: the Java object monitor approach is superior for "vanilla" mutual exclusion
  - You can't beat built-in language support
  - Specifically "automatic `acquire` and `release`"
- Use the JDK 5 facilities for more complicated situations
  - And read the Javadoc carefully!

Java Object Monitors: Introduction

- In Java, every object instance and every class is associated with a monitor
  - A monitor is implicitly associated with a lock
  - Java manages the magic transparently
- When a thread acquires the monitor lock, all other threads are blocked from acquiring that lock
- How does a thread acquire the monitor lock?
  - Not by accessing the object's code!
  - By (literally) synchronizing on the object instance

```
synchronized (o) {
  // code goes here
}
```

- We say that the thread has synchronized on Object o
- JVM will block this thread if some other thread currently has "Object o lock"

Java Synchronization

```java
synchronized (o) {
  // code goes here
}
```

- Thread has the lock if it makes it past beginning of synchronized block
- JVM will automatically release that lock when thread executions moves past the closing curly brace of the synchronized block
  - Even if it exits via an Exception ☺
- Note (I can't stress this too much): complete decoupling of the thread …
  - A "live thing", something which **executes code**
- From the lock …
  - Which is associated with an object (or class) instance
  - Static code and associated mutable state

“Synchronization” Is Just a Lock

- I introduced Java synchronized keyword after discussing reentrant lock approach because
  - That’s essentially what a Java monitor is: a reentrant lock associated with an object or class instance
  - For which the language and JVM support provide automatic “acquire” and “release” semantics
- But conceptually the same
- With all the perils thereto ☺
  - You are responsible for ensuring that the lock is used consistently to protect the relevant critical section throughout your code

Synchronize On Private Object?

You can synchronize on explicit lock variable

```
class BankAccount {
  private int balance = 0;
  private Object lk = new Object();
  // etc

  void withdraw(int amount) {
      synchronized(lk) {
          // modify balance
      }
  }
}
```

But ...

- Makes it harder for other code to synchronize on this BankAccount's operations
- And ... why bother? Just synchronize on this
  - The BankAccount instance itself

```
class BankAccount {
  private int balance = 0;
  int getBalance() { synchronized (this) { return balance; }
      }
}
```

Can Synchronize On Method Granularity

```java
class BankAccount {
  private int balance = 0;
  synchronized int getBalance() { return balance; }
```

- Syntactic sugar ...
- Putting synchronized before a method declaration is equivalent to surrounding the method body with a `synchronized(this)`
- Looks nicer ☺
- May cost you if the method consists of multiple, “long-duration” critical sections involving different shared resources
  - Prevents other threads from accessing all of those resource

“Data Races”

- In addition to solving the “bad interleaving” problem, mutual exclusion solves the “data race” problem
- Key point: sensitize yourself to the existence of such problems

What is a Data Race?

The Thread Analyzer detects data-races that occur during the execution of a multi-threaded process. A data race occurs when:

1. *two or more threads in a single process access the same memory location concurrently, and*
2. *at least one of the accesses is for writing, and*
3. *the threads are not using any exclusive locks to control their accesses to that memory.*

When these three conditions hold, the order of accesses is non-deterministic, and the computation may give different results from run to run depending on that order. Some data-races may be benign (for example, when the memory access is used for a busy-wait), but many data-races are bugs in the program.

Sun Studio 12: Thread Analyzer User’s Guide

Data Race: This Code Is Incorrect

```
class Stack<E> {
  private E[] array = (E[]) new Object[SIZE];
  int index = -1;
  boolean isEmpty() { // unsynchronized: wrong?
    return index==-1;
  }

  synchronized void push(E val) {
    array[++index] = val;
  }

  synchronized E pop() {
    return array[index--];
  }

  E peek() { // unsynchronized: wrong!
    return array[index];
  }
}
```

- Perhaps *isEmpty* and *peek* can “get away” with dropping synchronized?
  - After all: looks like the state changes written or read by these methods are atomic

Data Race Problems: You Must Synchronize

- Even "tiny steps" may require multiple steps in the compiled code implementation
  - `array[++index] = val` (probably) takes at least two steps
- Useful to think of a "data race" as a "simultaneous access error"
- Compared to "bad interleavings", a "data race" is a "lower-level" error
  - Absolutely, fundamentally, a problem
  - You simply cannot reason about the behavior of code that has data races
  - From the C++ reference: "If a data race occurs, the behavior of the program is undefined"
- Not going into how data races actually cause problems
  - That's a discussion for a compiler and/or computer architecture course ☺
- See the `java.util.concurrent.atomic` classes if you want "atomic" operations on individual variables

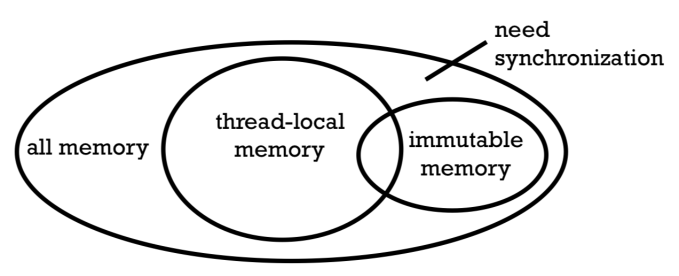
A Word About Deadlocks

- Students' reaction to discussion so far is often *"no problem, I'll wrap all my methods with `synchronized`"* ☺
- One issue: may hurt performance because other threads can't make progress while this thread is executing a given method
  - They can execute neither this method nor any other method of this object
  - We'll elaborate on this issue later
- More fundamental problem: deadlock
  1. $Thread_1$ acquires the lock but blocks before releasing (e.g., "data structure is full")
  2. $Thread_2$, waiting to empty the data structure, can't acquire the lock
  3. Both threads block forever
- Classic example: a "producer/consumer" class
- Lesson: don't blindly "synchronize" everything

Concurrency Programming Guidelines

- Previous material is about making you aware of the multi-threading paradigm shift
- Why perfectly correct sequential code can break as soon as it's deployed in a multi-threaded environment
- The need for using mutual exclusion techniques to keep concurrent threads from stepping on one another
- Now: some guidelines as you begin writing your own concurrent code

Categorize Memory Into Three Buckets



Every memory location in your program should have (at least) one of the following properties

- Be thread-local: only one thread ever accesses it
- Be immutable: (after being initialized), memory is only read, never written
- Be synchronized: locks are used to ensure there are no race conditions

Thread-Local Memory

- Simplest way to avoid data races is for only one thread to have access to memory ☺
- So: whenever possible, do not share resources among threads
- Definitely encourage you to read the `java.lang.ThreadLocal` Javadoc!
- If multiple threads need to access a resource, see if you can give each a copy of that resource
  - Example: `java.util.Random`
  - Only consider allowing threads to share a mutable resource if no other choice
- Note: this is why you don't need to synchronize on a method's local variables ...
- The runtime's call-stack is (effectively) thread-local!

Immutable Memory

- Whenever possible, do not update objects
  - Instead: create new objects
- This is one of the key tenets of functional programming
  - Always (even in sequential environment) a good idea to avoid side-effects
  - An invaluable idea in a concurrent environment!
- Key point: concurrent read operations can never cause data races
- Programming tip: we're biased to "reusing" (mutating) object instances
  - We justify that bias by talking about "efficiency" ☺
  - Try to correct for that bias

The Rest of The Venn Diagram

- After you've minimized the amount of state shared & mutable state, the burden of providing mutual exclusion falls on you
- Some guidelines for these scenarios …
- Guideline #0: No data races
  - Never allow two threads to read/write or write/write the same location at the same time
- (Remember: avoiding "data races" will not necessarily fix "bad interleavings")
- Guideline #1: For each piece of state that requires mutual exclusion, associate a lock that must be acquired before thread can read or write that location
  - We say the lock guards the location

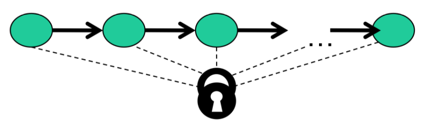
Consistent Locking

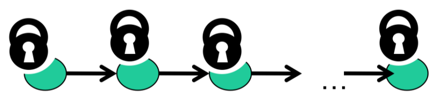
- Perfectly fine to use the same lock to guard multiple pieces of state
  - In Java, the guard is often the object containing the location
  - `this` inside an object's methods
  - But you can guard a larger data-structure with its own lock to provide more granular mutual exclusion
- Remember to document the guard for each location
- You – the programmer – get to decide how to partition the control of your "shared-and-mutable" state lock

Flexibility: Change The Locking Protocol Dynamically

- We already have three techniques to avoid problems
  1. Thread-local
  2. Immutable
  3. Consistent locking
- Your code can dynamically switch between one technique and another
  - As long as all threads agree on the protocol being used at any given time
- Example: data-structure initialization may use locking technique because multiple threads are used for performance reasons
  - Then kill all threads but “main”: now protected by “thread-local” technique
  - Or: once initialized, data-structure becomes immutable

“Coarse-Grained” Versus “Fine-Grained” Locking





- Fewer locks, or “more objects per lock”
  - Example: One lock for entire data structure (e.g., array)
  - Example: One lock for all bank accounts

- More locks, or “fewer objects per lock”
  - Example: One lock per data element (e.g., array index)
  - Example: One lock per bank account

Not a dichotomy: this distinction is a continuum!

“Coarse-Grained” Versus “Fine-Grained”: Tradeoffs

- Coarse-grained advantages
  - Simpler to implement
  - Faster/easier to implement operations that access multiple locations
    - Because all guarded by the same lock
- Fine-grained advantages
  - Increase possibilities for concurrent access
  - This can improve performance in situations where coarse-grained locking leads to unnecessary blocking
- Guideline #2: Start with coarse-grained (simpler) and move to fine-grained (performance) only if contention on the coarser locks becomes an issue
  - Use a profiler before trusting your intuition!
  - Can guarantee that (at least initially) you’ll introduce lots of bugs ☹

“Critical-Section” Granularity

- We’ve been discussing lock granularity: how many objects per lock
- Orthogonal issue: critical-section granularity
  - How much “work” in the code that must be done while holding locks
- Critical sections are too long?
  - Performance loss because other threads are blocked
- Critical sections are too short?
  - Bugs! You’ve written code where other threads will be able to see intermediate state
- Guideline #3: Make an effort to not do expensive computation or I/O in critical sections
  - And take even more care not to introduce race conditions ☺

Understanding The “Critical-Section Granularity” Issue

- Tempting (and often a good idea) to simply wrap a chunk of code with a single lock
  - Example: code only contains operations like assignment statements and “cheap” method
    - Code runs so fast that it’s not worth splitting into multiple critical sections
- Assumption fails when critical section will contain “expensive” computation
- Scenario: you need to replace a Hashtable value by performing some computation on its old value
  - We want other threads to see either the old value or the new value
  - We also need the old value in order to perform the expensive code required to compute the new value

This Critical-Section Is Too Large

- Assume that `lock` guards the Hashtable
- The code below locks the entire table during the expensive computation

```
synchronized(lock) {
  v1 = table.get(k);
  v2 = expensive(v1);
  table.put(k, v2);
}
```

This Critical-Section Is Too Short

```
synchronized(lock) {
  v1 = table.get(k);
}

v2 = expensive(v1);

synchronized(lock) {
  table.put(k, v2);
}
```

- The above code is buggy!
- Scenario: $thread_2$ updates to map $k \rightarrow v3$ while $thread_1$ was executing `expensive(v1)`
- If $thread_2$ does the update after $thread_1$, then table will end up with $k \rightarrow v3$ (not v2)
- If $thread_2$ does the update before $thread_1$, then table will end up with $k \rightarrow$ expensive(v3) (not v2)
- Desired semantics: $k \rightarrow v2$

This Critical-Section Is Just Right

```java
boolean loop_done = false;
while(!loop_done) {
  synchronized(lock) {
    v1 = table.get(k);
  }

  v2 = expensive(v1);

  synchronized(lock) {
    // If true, we already have correctly
    // computed ``v2''
    //
    // If false, we know that some other
    // thread has updated the map: throw out
    // pre-computed value and start over
    if (table.get(k) == v1) {
      loop_done = true;
      table.put(k, v2);
    }
  } // while-loop
```

Note: this approach assumes that it doesn't matter if some arbitrary sequence of Hashtable operations occurred between the two critical section. All we care about is that we change mapping from $k \to v1$ to $k \to v2$

Design For Atomicity

- Operations on your data-structures should be atomic
  - No other thread can see the “partial execution” of that operation
- Guideline #4: Think about atomicity first and locks second
  - “What operations need to be atomic?”
  - Make critical sections just long enough to preserve atomicity
  - Then design the locking protocol to implement the critical sections correctly

Don't "Roll Your Own" (I)

- You need to understand today's material because you'll be writing new code
- But: make sure you're aware of the JDK 5 `jdk.util.concurrent` concurrent collection APIs
  - These are high-performance concurrent implementations of standard collection interfaces such as *List*, *Queue*, and *Map*
  - To provide high concurrency, these implementations manage their own synchronization internally
  - Implication: you can't exclude concurrent activity from a concurrent collection
  - Locking a concurrent collection only slows your code down, accomplishes nothing!
- Take the time to understand what these classes do and how they do it before coding your own data-structure
  - See if you can re-purpose the concurrent collection classes even if it doesn't immediately look like it addresses your problem

Don't "Roll Your Own" (II)

```java
// Code by Bloch, Item #81: "Simulate the
// behavior of String.intern:"
private static final ConcurrentMap<String, String> map =
    new ConcurrentHashMap<>();

public static String intern(String s) {
  // ConcurrentHashMap is optimized for
  // retrieval operations, such as
  // get. Therefore, it is worth invoking
  // get initially and calling putIfAbsent
  // only if get indicates that it is
  // necessary:
  String result = map.get(s);
  if (result == null) {
    result = map.putIfAbsent(s, s);
    if (result == null)
      result = s;
  }
  return result;
}
```

Concurrent Collections: Bloch's Observations

- (FWIW, my own performance tests confirm Bloch's statements below)
- "On my machine, the intern method above is over six times faster than `String.intern`"
- "Concurrent collections make synchronized collections largely obsolete"
- "For example, use ConcurrentHashMap in preference to Collections.synchronizedMap"
- "Simply replacing synchronized maps with concurrent maps can dramatically increase the performance of concurrent applications"

Blocking Operations

- In JDK 5, some of the collection interfaces were extended with blocking operations
  - That is: if a method can't be performed at the time that a thread invoked it …
  - The thread will (automatically) block until the internal state of the collection permits the method to continue
- Example: `BlockingQueue extends Queue` and adds several methods, including `take`
  - `take` removes and returns the head element from the queue, waiting if the queue is empty
- Blocking queues can be used to implement work queues (also known as "producer-consumer" queues)
  - One or more producer threads enqueue work items
  - One or more consumer threads dequeue and process items as they become available
  - Threads block as long as queue state isn't appropriate for desired semantics

- Synchronizers are objects that enable threads to wait for one another
  - Use synchronizers when you need to coordinate decoupled activities
- Typically, the *CountDownLatch* and *Semaphore* classes suffice
  - Sometimes you need the more powerful (and complicated) *CyclicBarrier* and *Exchanger* classes
  - Or even the *Phaser*

CountDownLatch (I)

- Countdown latches are "single-use" barriers
  - They allow one or more threads to wait for one or more other threads to do something else
  - CountDownLatch constructor takes an `int`: specifies the number of times the `countDown` method must be invoked on the latch before all waiting threads are allowed to proceed
- Scenario (see next slide for code)
  - You want to "time" code execution for a given "concurrency level"
  - That implies that you only start the clock when all worker threads are "ready to go"
  - Implementation: one latch is initialized to the given concurrency level, each worker decrements the count when ready
  - That latch prevents timer from starting until all are ready
  - Another latch ensures that the timer measurement is only taken when all workers have completed

CountDownLatch (II)

```java
// Code from Goetz and Bloch
public class ConcurrentTimer {
  private ConcurrentTimer() { } // Noninstantiable

  public static long time(Executor executor, int concurrency,
                          Runnable action) throws InterruptedException {
    CountDownLatch ready = new CountDownLatch(concurrency);
    CountDownLatch start = new CountDownLatch(1);
    CountDownLatch done  = new CountDownLatch(concurrency);

    for (int i = 0; i < concurrency; i++) {
      executor.execute(() -> {
          ready.countDown(); // Tell timer we're ready
          try {
            start.await(); // Wait till peers are ready
            action.run();
          } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
          } finally {
            done.countDown();  // Tell timer we're done
          }
        });
    }

    ready.await();     // Wait for all workers to be ready
    long startNanos = System.nanoTime();
    start.countDown(); // And they're off!
    done.await();      // Wait for all workers to finish
    return System.nanoTime() - startNanos;
  }
}
```

- Old-style discussions of how to create multiple, independent, tasks focus on *"should I extend `Thread` or should I implement `Runnable`?"*
- That's really "old-style": almost always, refrain from working directly with threads!
- When you work directly with threads, a Thread serves as both a "unit of work" (or "task") and the mechanism for executing the unit of work
- JDK 5 provides the *Executor* framework, which separates the task concept from the execution mechanism
- Two kinds of tasks: *Runnable* and its close cousin, *Callable*
    - A *Callable* is like *Runnable*, except that it returns a value and can throw arbitrary exceptions

Executor Service

- The general mechanism for executing tasks is the JDK 5 (interface-based) executor service
- Think in terms of tasks (not Threads!), let an executor service execute those tasks for you
  - You'll gain the flexibility of scheduling different execution policies, concurrency levels, and other tunable knobs
  - Read the Javadoc!
- Some examples:
  - You can wait for any or all of a collection of tasks to complete (see `invokeAny` and `invokeAll`)
  - You can wait for the executor service to terminate (see `awaitTermination`)
  - You can retrieve the results of tasks one by one as they complete (using an *ExecutorCompletionService*)
  - You can schedule tasks to run at a particular time or to run periodically (using a *ScheduledThreadPoolExecutor*)

Executor Example

```java
// Code from Java Concurrency In Practice (Goetz)
public class PrimeGenerator implements Runnable {
  private static ExecutorService exec = Executors.newCachedThreadPool();
  private final List<BigInteger> primes = new ArrayList<BigInteger>();
  private volatile boolean cancelled;

  public void run() {
    BigInteger p = BigInteger.ONE;
    while (!cancelled) {
      p = p.nextProbablePrime();
      synchronized (this) {
        primes.add(p);
      }
    }
  }
  public void cancel() {
    cancelled = true;
  }
  public synchronized List<BigInteger> get() {
    return new ArrayList<BigInteger>(primes);
  }
  static List<BigInteger> aSecondOfPrimes() throws InterruptedException {
    PrimeGenerator generator = new PrimeGenerator();
    exec.execute(generator);
    try {
      SECONDS.sleep(1);
    } finally {
      generator.cancel();
    }
    return generator.get();
  }
}
```

Some Closing Thoughts

“Concurrent” Programming

- Concurrency: the problem of correctly and efficiently managing concurrent access to shared resources from multiple clients
- Requires coordination between threads
  - Typically done by requiring other threads to block until “first thread is no longer accessing the shared resource”
  - Compare to the “fork/join” framework from *Introduction to Algorithms*
    - Concurrency coordination is very different from “fork/join” coordination
    - *join* semantics: $thread_1$ blocks until $thread_2$ has completely finished its execution
    - Concurrency coordination: $thread_1$ blocks until $thread_2$ has finished some “arbitrary” section of code execution
- Major challenge: correct concurrent programs are non-deterministic
  - Non-repeatability → much more complicated to test & debug than single-threaded environments ☹

Parallelism *Versus* Concurrency (I)

- We've previously discussed (in *Introduction to Algorithms*) using multiple threads to achieve parallelism
  - Defined as "how to break up a computation into parts that can be executed in parallel"
  - Remember the Fork/Join framework?
- These parallel algorithms all had a very simple structure that avoided the race conditions discussed in today's lecture
- Each thread is assigned its "own section of memory"
  - Example: algorithm partitions array into sub-arrays, one sub-array per thread
- Yes: all threads have access to the same shared memory ("entire array")
  - But algorithm ensures that threads agree as to what's "mine" and what's "theirs"
- On `fork`, algorithm completely assigns ownership of sub-array to "forkee"
- Main thread only resumes ownership when it does a `join` on the "forkee" thread

Parallelism *Versus* Concurrency (II)

- Unfortunately for us, the approach we used for fork/join parallelism won't work when
  - Memory is accessed by threads in overlapping or unpredictable ways
  - Threads are doing un-coordinated tasks but still need concurrent access to the same resources
  - Such scenarios raise the mutual-exclusion issues discussed in today's lecture
- Thought experiment: can you "reduce" implementation of a data-structure (let alone a DBMS) to fork/join parallelism?
- In a nutshell:
  - Parallelism is about making algorithms run faster
  - Concurrency will not make "an algorithm" run faster

Concurrency: Not Just About Throughput

- We've previously explained that concurrency is about achieving good throughput
  - Alternatively: enabling as many client threads to use a service (or data-structure) at the same time (while preserving "correctness")
- But note that concurrency also improves another performance metric: latency
- No time to pursue this idea, but here are some examples
  - Respond to GUI events in one thread while another thread performs an expensive computation
  - Web-application services another request while another thread does I/O for previous request
  - Failure in $task_1$ does not bring down $task_2$

Today's Lecture: Wrapping it Up

Why A Lecture On "Java Concurrency"?

Transitioning To Multi-Threaded Code

Race Conditions: The Problem(s)

Synchronization: The Solution

Concurrency Programming Guidelines

Creating & Managing Multiple Clients

Some Closing Thoughts

Readings

- I've already strongly recommended Java Concurrency in Practice by *Brian Goetz et al* Amazon
- This lecture overlaps considerably with Chapters 2 & 3
- Am not assigning "homework", just a course project ☺