COMPSCI 230 S2C 2015
Software Design and Construction

Lecture 5 of Theme C

Locking, blocking, mutex; visibility, consistency

Learning Goals for Today

- Develop a "theoretical" understanding of concurrency
  - Learn a new word and a new concept: mutex, and mutual exclusion
  - Work through a few more examples
  - Know Goetz's advice on when to use, and when not to use, synchronization.
- Be able to diagnose a deadlocked system
  - Note: deadlock avoidance is beyond the scope of CompSci 230.
- Start to develop your own position on the relative importance of designing for features, correctness, security, and performance.
  - Agility and maintainability may be more important than some of the above… remember XP?
  - Consider taking Goetz's advice and my advice, then decide for yourself!

Atomicity isn't quite enough

- Up to now, we have talked about "threads running into each other", "stepping on each other", "over-writing each other's data", and "interrupting each other".
  - This is a vague statement of a major problem in concurrent programming.
  - Wikipedia has a nice definition: "In concurrent programming, an operation (or set of operations) is atomic… if it appears to the rest of the system to occur instantaneously."
- If only one thread is allowed to execute, then atomicity is trivial.
  - We need a "fairness" property too! Every thread should have a "fair chance" of entering a synchronized block.
  - The fairness properties of Java are outside the scope of CompSci 230.
    - Unfairness is observable in A4v0. Some workers are given more CPU time than others, and finish their task sooner; all tasks are of similar size.

Mutual Exclusion: A Solvable Problem

- Dijkstra (1965): "… computers have to be programmed in such a way that
  - at any moment only one of [their programs] is in its critical section…
  - [This is the problem of] mutual exclusion of critical-section execution…"
- Java' provides a solution to Dijkstra's "mutual exclusion problem".
  - Critical sections can be implemented, in Java, as synchronized blocks.
  - The monitor object (or lock) on a synchronized block is sometimes called a mutex.
- Dijkstra excludes static-priority schemes.
  - For example, a system in which "the main() thread controls the mutexes of all worker threads" does *not* provide mutual exclusion.
  - The fairness of a mutex is a contentious issue in co-design contracts:
    - Language designers versus OS designers, versus hardware designers;
    - Ease of programming versus performance, versus computational cost.

Synchronizing for Visibility (Goetz)

- "When an object acquires a lock, it first invalidates its cache,
  - so that it is guaranteed to load variables directly from main memory.
- "Similarly, before an object releases a lock, it flushes its cache,
  - forcing any changes made to appear in main memory.
- "In this way, two threads that synchronize on the same lock are
  - guaranteed to see the same values in variables modified inside a synchronized block.
- "The basic rule for synchronizing for visibility is that you must synchronize whenever you are:
  - "Reading a variable that may have been last written by another thread, [or]
  - "Writing a variable that may be read next by another thread."

My advice on `synchronized`

- Let the package-writers do the heavy-lifting – it's very difficult to write synchronized code.
  - `SwingWorkers` is pretty easy to use.
  - The `Concurrency` package will give you some additional options..
- There are no synchronized methods in `a4v0.jar`
  - Any field or variable written by a `SwingWorker`'s `doInBackground()` method is *not* reliably communicated to the model. So: each `SwingWorker` has a `MandelTask` object. Each worker writes its results into this object using its worker thread, that is, when its `doInBackground()` method is invoked.
  - Each `SwingWorker` writes its results into the model when its `done()` method is invoked – and this method is invoked on the EDT.
  - See https://docs.oracle.com/javase/tutorial/uiswing/concurrency/simple.html if you want to learn more, but this isn't examinable.

Synchronizing for Consistency (Goetz)

- "… you must also synchronize to ensure that consistency is maintained from an application perspective.
- "When modifying multiple related values,
  - you want other threads to see that set of changes atomically – either all of the changes or none of them.
- "This applies to
  - related data items (such as the position and velocity of a particle) and
  - metadata items (such as the data values contained in a linked list and the chain of entries in the list itself)."

- I'd say Goetz is giving good advice here, *if* your workers really must interact with each other in order to complete their task.
  - I occasionally use synchronization, but I only very rarely use lower-level constructs (e.g. volatile booleans), to ensure consistency and visibility.
  - "A spoonful of sugar helps the medicine go down"!
  - My advice: try to define tasks that can be done independently, so that you don't have to worry about workers getting consistent information from each other.

Hacking a stack – what's dangerous here?

```
public class UnsafeStack {
  public int top = 0;
  public int[] values = new int[ 1000 ];
  public void push( int n ) {
    values[ top++ ] = n;
  }
  public int pop() {
    return values[ --top ];
} }
```

- "What happens if more than one thread tries to use this class at the same time? It could be a disaster."

Hacking a stack – what's dangerous here?

```
public class UnsafeStack {
  public int top = 0;
  public int[] values = new int[ 1000 ];
  public void push( int n ) {
    values[ top++ ] = n;
  }
  public int pop() {
    return values[ --top ];
} }
```

- "Because there's no synchronization,
  - multiple threads could execute `push()` and `pop()` at the same time.
- What if one thread calls `push()` and another thread calls `push()` …
  - between the time top is incremented and it is used as an index into `values`?
- Then both threads would store their new value in the same location!"

Hacking a stack – what's dangerous here?

```
public class UnsafeStack {
  public int top = 0;
  public int[] values = new int[ 1000 ];
  public void push( int n ) {
    values[ top++ ] = n;
  }
  public int pop() {
    return values[ --top ];
} }
```

- "In this case, the cure is simple …
  - synchronize both `push()` and `pop()`, and you'll prevent one thread from stepping on another.
- "Note that using `volatile` would not have been enough…"
  - (Do you understand why?)

Incrementing a shared counter

```
public class Counter {
  private int counter = 0;
  public int get() {
    return counter;
  }
  public void set( int n ) {
    counter = n;
  }
  public void increment() {
    set( get() + 1 );
} }
```

- Is this class thread-safe?
  - Thread-safety won't be formally defined in CompSci 230.
  - Informal question: "If multiple threads access these class methods, will this expose defects of visibility, consistency, or fairness?"
  - Fairness defects are out of scope for CompSci 230. Let's consider visibility first.

Incrementing a shared counter: visibility?

```
public class Counter {
  private int counter = 0;
  public int get() {
    return counter;
  }
  public void set( int n ) {
    counter = n;
  }
  public void increment() {
    set( get() + 1 );
} }
```

- If a thread invokes the `set()` method on the same instance of `Counter`, either directly or indirectly through `increment()`,
  - The new value might not be visible to another thread.
  - Possible fixes:
  1. make counter `volatile`, or
  2. synchronize `set()`.

Incrementing a shared counter

```
public class Counter {
  private volatile int counter = 0;
  public int get() {
    return counter;
  }
  public void set( int n ) {
    counter = n;
  }
  public void increment() {
    set( get() + 1 );
} }
```

- Is this a thread-safe class?
  - No, concurrent invocations of `increment()` may expose an atomicity defect.
    - Both threads might `get()`; then both threads might add one and `set()`.

Incrementing a shared counter

```
public class Counter {
  private int counter = 0;
  public int get() {
    return counter;
  }
  public synchronized void set( int n ) {
    counter = n;
  }
  public void increment() {
    set( get() + 1 );
} }
```

- Is this thread-safe?
  - No, because concurrent invocations of `increment()` aren't atomic.

Incrementing a shared counter

```
public class Counter {
  private int counter = 0;
  public int get() {
    return counter;
  }
  public synchronized void set( int n ) {
    counter = n;
  }
  public synchronized void increment() {
    set( get() + 1 );
} }
```

- Is this thread-safe?
  - No, now there's a visibility problem with `get()` after `set()`.

Incrementing a shared counter

```
public class Counter {
  private int counter = 0;
  public synchronized int get() {
    return counter;
  }
  public synchronized void set( int n ) {
    counter = n;
  }
  public synchronized void increment() {
    set( get() + 1 );
} }
```

- Is this thread-safe?
  - Yes, I think so. (So does Goetz.)

Incrementing a shared counter

```
public class Counter {
  public int counter = 0;
  public synchronized int get() {
    return counter;
  }
  public synchronized void set( int n ) {
    counter = n;
  }
  public synchronized void increment() {
    set( get() + 1 );
} }
```

- Is this thread-safe?
  - NO!!!!

Immutability

- A class is immutable if it enforces a "write-once, before-any-read" property on its instances.
  - Many Java classes, including `String`, `Integer`, and `BigDecimal`, are immutable – their value cannot change after they are initialised.
- The advantage of immutability, from a thread-safety perspective, is that there is only one execution sequence to consider:
  - 1) instantiation; 2) the final write; 3) any number of concurrent reads.
  - Note: because there is only a single write, it can't be concurrent.
- A class is immutable if all of its fields are declared `final`.
  - This isn't always desirable.
  - Many immutable classes gain a performance advantage from having non-final fields which are invisible to the caller, such as `String.hashCode()`.
- Even if an entire class isn't immutable, you don't need to synchronize access to its `final` fields.
- Goetz's advice: `Final` is "thread-friendly". Use it as much as possible!

When you don't need to synchronize

- Goetz identifies a few safe harbours. "You don't need to synchronize your cross-thread data propagations
  - "When data is initialized by a static initializer (an initializer on a static field or in a `static{}` block)
  - "When accessing final fields
  - "When an object is created before a thread is created
  - "When an object is already visible to a thread that it is then joined with"

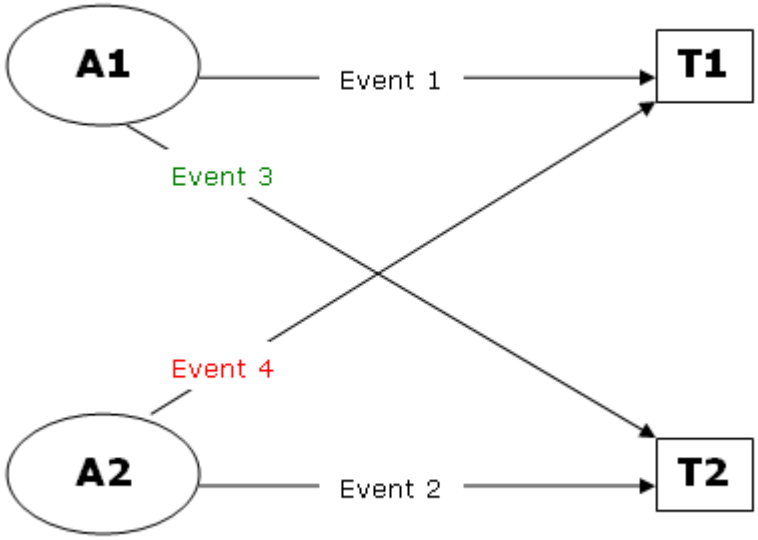
Deadlock

- "Whenever you have multiple processes contending for exclusive access to multiple locks, there is the possibility of deadlock.
  - "A set of processes or threads is said to be *deadlocked* when each is waiting for an action that only one of the others can perform.
- "The most common form of deadlock is when
  - "Thread 1 holds a lock on Object A and is waiting for the lock on Object B, and
  - "Thread 2 holds the lock on Object B and is waiting for the lock on Object A.
  - "Neither thread will ever acquire the second lock or relinquish the first lock. They will simply wait forever."
- Dijkstra, in EWD 123 (1968):
  - "This situation, when one process can only continue provided the other one is killed first, is called 'The Deadly Embrace'. "

Blocks escalating to deadlocks (Teratrax)

- "The following diagram shows the sequence of events leading to a deadlock.
  - Consider two applications (A1, A2) accessing two different [tables] (T1, T2):
    - Event 1: A1 places a lock on T1 inside its transaction [or synchronized block] and continues to execute other statements
    - Event 2: A2 places a lock on T2 inside its transaction and continues to execute other statements
    - Event 3: A1 attempts to place a lock on T2 ([because it needs] to access T2 before it can finish the transaction) but has to wait for A2 to release its lock

At this point, a block is created since A2 is blocking A1



- Event 4: While A1 is waiting, A2 attempts to place a lock on T1 ([because it needs] to access T1 before it can finish its own transaction)

A deadlock is created since two connections have blocked one another.

SQL Server automatically resolves the deadlock by choosing one of the connections as a deadlock victim and killing it."

Deadlock Avoidance

- Goetz: "To avoid deadlock, you should ensure that
  - when you acquire multiple locks, you always acquire the locks in the same order in all threads."
- This is sound advice
  - but it's beyond the scope of CompSci 230!
  - I don't expect you to be able to design a multi-threaded program with multiple locks.
  - I do expect you to realise that deadlock is possible, and that deadlock can be avoided in a program *if* its designer is able to construct a partial order on lock acquisition that will get the job done safely.

Deadlock Detection

- Deadlock detection is easier than avoidance
  - A program that is deadlocked is likely to be "hung" (non-responsive).
  - Any deadlocked program has at least two threads that are blocked or waiting.
- Discovering the state of a thread
  1. If you send a QUIT signal to a JVM (on most systems, typing CTRL-BREAK will do this) it will probably call its thread-dump method before exiting.
     - CTRL-C usually sends a SIGINT signal (an interrupt); to get a thread-dump, you want to send a SIGBREAK signal to the JVM.
  2. If you attach a debugger to a Java program (using the JDI interface to its JVM), you can see which threads are in the BLOCKED and WAITING states.
  3. You can invoke the getStackTrace() method of any thread you suspect of being deadlocked, at the point where you think it may be deadlocked.
     - You'll have to interrupt the deadlocked thread!
     - Use a try-catch structure, with the getStackTrace() in the catch handler.

Performance considerations

- Goetz: "There has been a lot written – much of it wrong – on the performance costs of synchronization.
  - It is true that synchronization, especially contended synchronization, has performance implications, but these may not be as large as is widely suspected.
- "Many people have gotten themselves in trouble by using fancy but ineffective tricks to try to avoid having to synchronize.
  - One classic example is the double-checked locking pattern (see Resources for several articles on what's wrong with it).
  - This harmless-looking construct purported to avoid synchronization on a common codepath, but was subtly broken, and all attempts to fix it were also broken."

Goetz's advice on performance-tuning

- "When writing concurrent code, don't worry so much about performance until you've actually seen evidence of performance problems.
  - "Bottlenecks appear in the places we often least suspect.
  - "Speculatively optimizing one code path
    - that may not even turn out to be a performance problem –
    - at the cost of program correctness –
    - is a false economy."
- I agree with Goetz regarding performance-tuning.
  - You should get the program features working *before* you try to optimise their performance.
  - But… maybe your experience will be different?
- In my experience, the most important objective is "features".
  - Then correctness, then security, then (if there's a problem) performance.
  - Maintainability and agility are also important!

Guidelines for synchronization (Goetz)

- "There are a few simple guidelines you can follow when writing synchronized blocks that will go a long way toward helping you to avoid the risks of deadlock and performance hazards:
- **Keep blocks short.**
  - Synchronized blocks should be short – as short as possible while still protecting the integrity of related data operations.
  - Move thread-invariant pre-processing and post-processing out of synchronized blocks.
- **Don't block.**
  - Don't ever call a method that might block, such as `InputStream.read()`, inside a synchronized block or method.
- **Don't invoke methods on other objects while holding a lock.**
  - This may sound extreme, but it eliminates the most common source of deadlock.

Learning Goals for Today

- Develop a "theoretical" understanding of concurrency
  - Learn a new word and a new concept: mutex, and mutual exclusion
  - Work through a few more examples
  - Know Goetz's advice on when to use, and when not to use, synchronization.
- Be able to diagnose a deadlocked system
  - Note: deadlock avoidance is beyond the scope of CompSci 230.
- Start to develop your own position on the relative importance of designing for features, correctness, security, and performance.
  - Agility and maintainability may be more important than some of the above… remember XP?
  - Consider taking Goetz's advice and my advice, then decide for yourself!