CSx55: Distributed Systems [Threads]

The Tangible Lock

Have you a synchronized method?
    The acquisition's implicit
        With the lock hiding in plain slight

Care for the lock to be tactile?
    Use the `Lock` instead
        But with responsibilities galore

A recourse when drowning in bugs?
    Tread carefully with how you lock() and unlock()
        and … reckon with them exceptions

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Can a thread call another thread's methods?
- Can we use stop/suspend/resume if know how to use them "correctly"?
- Pinning threads to cores? N.B.: also known as thread affinity
- Can we know if the core utilization by a given application (with a large number of cores) is high?
- When a thread "sleeps" who is awaking the thread when time elapses?
- Why is the run() method public, if we should call start()? What's the point?
- What happens when you interrupt a thread, but that thread has no blocking calls?
- Where is the thread (i.e., the object associated with it stored)?

Topics covered in this lecture

- Locks
- Notifications
- Wait-notify

Two friends plan to meet at Starbucks
But there are two Starbucks on College Avenue

	@ the First Starbucks Store	**@ the Second Starbucks Store**
12:10	**A** is looking for friend **B**	**B** is looking for friend **A**
12:15	**A** leaves for the second store	**B** leaves for the first store
12:20	**B** arrives at store	**A** arrives at store
12:30	**B** is Looking for friend **A**	**A** is looking for friend **B**
12:40	**B** leaves for the second store	**A** leaves for the first store

Both friends are now frustrated and undercaffeinated!

Data Synchronization

Why sharing data between threads is problematic

- **Race conditions**
  - Correct outcome depends on lucky timing of uncontrollable events
- Threads attempt to access data more or less *simultaneously*
  - A thread may change the value of data that some other thread is operating on

Example code with race condition

```
public class MyThread extends Thread {
   private byte[] values;
   private int position;

   public void
      modifyData(byte[] newValues, int newPosition) {
      ... Modify values and position
   }

   public void utilizeDataAndPerformFunction() {
      ... Use values and position
   }

   public void run() {
     ... Main logic
   }
}
```

In the previous snippet a race condition exists because …

- The thread that calls `modifyData()` is **accessing the same data** as the thread that calls `utilizeDataAndPerformFunction()`

- `utilizeDataAndPerformFunction()` and `modifyData()` **are not atomic**
  - It is possible that `values` and `position` are changed *while they are being used*

What is atomic?

- The code cannot be interrupted during its execution
  - Accomplished in hardware or *simulated* in software
- Code that cannot be found in an *intermediate state*

Eliminating the race condition using the `synchronized` keyword

- If we declared both `modifyData()` and `utilizeDataAndPerformFunction()` as **synchronized**?
  - Only one thread gets to call *either* method at a time
    - Only one thread accesses data at a time
  - When one thread calls one of these methods, while another is executing one of them?
    - The second thread must *wait*

Example code with no race conditions by using the synchronized keyword

```
public class MyThread extends Thread {
   private byte[] values;
   private int position;

   public void synchronized
      modifyData(byte[] newValues, int newPosition) {
      ... Modify values and position
   }

   public void synchronized
      utilizeDataAndPerformFunction() {
      ... Use values and position
   }

   public void run() {
     ... Main logic
   }
}
```

Revisiting the mutex lock

- **Mut**ually **ex**clusive lock
- If two threads try to grab a mutex?
  - Only one succeeds
- In Java, every object has an associated **lock**

When a method is declared `synchronized` ...

- The thread that wants to execute the method must **acquire** a lock
- Once the thread has acquired the lock?
  - It executes method and **releases** the lock
- When a method returns, the lock is released
  - Even if the return is because of an exception

Locks and objects

- There is only **one lock per object**
- If two threads call synchronized methods of the same object?
  - Only one can execute immediately
    - The other has to wait until the lock is released

Afraid of what the truth might bring
He locks his doors and never leaves
Desperately searching for signs
To terrify, to find a thing
He battens all the hatches down
And wonders why he hears no sound
Frantically searching his dreams
He wonders what it's all about

Telescope, Cage the Elephant

SYNCHRONIZATION PITFALLS

Another code snippet to look at …

```
public class MyThread extends Thread {
   private boolean done = false;

   public void run() {
     while (!done) {
        ... Main logic
     }
   }

   public void setDone(boolean isDone) {
     done = isDone;
   }
}
```

Can't we just synchronize the two methods as we did previously?

- If we synchronized both `run()` and `setDone()` ?
  - `setDone()` would never execute!
- The `run()` method does not exit until the `done` flag is set
  - But the `done` flag cannot be set because `setDone()` cannot execute till `run()` completes
- Uh oh …

The problem stems from the scope of the lock

- **Scope of a lock**
  - Period between grabbing and releasing a lock
- Scope of the `run()` method is too large!
  - Lock is grabbed and never released
- We will look at techniques to *shrink the scope* of the lock
- But let's look at another solution for now

Let's look at operations performed on the data item (`done`)

- The `setDone()` method stores a value into the flag
- The `run()` method reads the value

- In our previous example:
  - Threads were accessing *multiple* pieces of data
  - No way to update multiple data items *atomically* without the `synchronized` keyword

But Java specifies that the loading and storing of variables is atomic

- Except for `long` and `double` variables
- The `setDone()` should be atomic
  - The `run()` method has only one read operation of the data item
- The race condition <u>should not</u> exist
  - But why is it there?

Threads are allowed to hold values of variables in registers

- When one thread changes the value of the variable?
  - Another thread *may not see* the changed variable

- This is particularly true in loops controlled by a variable
  - Looping thread **loads value of variable in register** and *does not notice* when value is changed by another thread

Two approaches to solving this

- Providing setter and getter methods for variable and using the `synchronized` keyword
  - *When lock is acquired*, temporary values stored in registers are *flushed* to main memory
- The **`volatile`** keyword
  - Much cleaner solution

If a variable is marked as `volatile`

- Every time it is used?
  - Must be read from main memory
- Every time it is written?
  - Must be written to main memory
- Load and store operations are **atomic**
  - Even for `long` and `double` variables

Some more about volatile variables

- Prior to JDK 1.2 variables were always read from main memory
  - Using volatile variables was moot

- Subsequent versions introduced memory models and optimizations

Synchronization and the volatile keyword

- Can be used *only* when operations use a **single load and store**
  - Operations like `++`, `--`?
    - Load-change-store …
- The `volatile` keyword forces the JVM to not make temporary copies of a variable
- Declaring an array `volatile`?
  - The reference becomes volatile
  - The individual elements are not volatile

Synchronized methods & Locks

Synchronizing methods

- **Not possible** to execute the same method in one thread while …
  - Method is running in another thread
- If two different synchronized methods in an object are called?
  - They both require the lock of the same object
- Two or more synchronized methods of the same object *can never run in parallel* in separate threads

A lock is based on a specific instance of an object

- Not on a particular method or class
- Suppose we have 2 objects: `objectA` and `objectB` with synchronized methods `modifyData()` and `utilizeData()`
- One thread can execute `objectA.modifyData()` while another executes `objectB.utilizeData()` *in parallel*
  - Two different locks are grabbed by two different threads
  - No need for threads to wait for each other

How does a synchronized method behave in conjunction with an unsynchronized one?

- Synchronized methods try to grab the object lock
  - Only 1 synchronized method in an object can run at a time … *provides data protection*
- Unsynchronized methods
  - Don’t grab the object lock
  - Can *execute at any time* … *by any thread*
    - Regardless of whether a synchronized method is running

For a given object, at any time …

- **Any number** of *unsynchronized methods* may be executing
- But only **1 synchronized method** can execute

Synchronizing static methods

- A lock can be obtained for each class
  - The **class lock**

- The class lock is the *object lock* of the **`Class` object** that models the class
  - There is only 1 `Class` object per class
  - Allows us to achieve synchronization for static methods

Object locks and class locks

- Are **not operationally related**
- The class lock can be grabbed and released *independently* of the object lock
- If a non-static synchronized method calls a static synchronized method?
  - It acquires both locks

Empty stares, from each corner of a shared prison cell
One just escapes, one's left inside the well
And he who forgets, will be destined to remember

Nothingman, Eddie Vedder & Jeffrey Ament, Pearl Jam

EXPLICIT LOCKING

The synchronized keyword

- *Serializes accesses* to synchronized methods in an object
- Not suitable for *controlling lock scope* in certain situations
- Can be *too primitive* in some cases

Many synchronization schemes in J2SE 5.0 onwards implement the `Lock` interface

- Two important methods
  - `lock()` and `unlock()`
- Similar to using the synchronized keyword
  - Call `lock()` at the start of the method
  - Call `unlock()` at the end of the method
- Difference: we have an *actual object* that **represents** the lock
  - Store, pass around, or discard

Semantics of the using Lock

- If another thread *owns* the lock
  - Thread that attempts to acquire the lock must wait until the other thread calls `unlock()`
- Once the waiting thread acquires the lock, it returns from the `lock()` method

Using the `Lock` interface

```
public class DataOpertor {
   private Lock dataLock = new ReentrantLock();
   public void
      modifyData(byte[] newValues, int newPosition) {
     try {
       dataLock.lock();
        ... Modify values and position
     } finally {
         dataLock.unlock();
     }
   }

   public void utilizeDataAndPerformFunction() {
      try {
       dataLock.lock();
        ... Use values and position
     } finally {
         dataLock.unlock();
     }
}
```

Advantages of using the Lock interface

- Grab and release locks *whenever* we want
- Now possible for **two objects to share the same lock**
    - Lock is no longer attached to the object whose method is being called
- Can be *attached to data, groups of data*, etc.
    - Not objects containing the executing methods

Advantages of explicit locking

- We can move them anywhere to **adjust lock scope**
  - Can span from a line of code to a scope that encompasses multiple methods and objects

- Lock at scope *specific to problem*
  - Not just the object

Synchronized Blocks

Much of what we accomplish with the `Lock` we can do so with the `synchronized` keyword

```
public class DataOperator {

   public void
      modifyData(byte[] newValues, int newPosition) {
     synchronized(this) {
        ... Modify values and position
     }
   }

   public void utilizeDataAndPerformFunction() {
      synchronized(this) {
         ... Use values and position
      }
   }
}
```

Synchronized methods vs. Synchronized Blocks

- Possible to use only the **synchronized block** mechanism to synchronize whole method
- You decide when it's best to synchronize a block of code or the whole method
- RULE: **Establish as small a lock scope as possible**

The `Lock` interface [java.util.concurrent.locks]

```
public interface Lock {

   public void lock();

   public void lockInterruptibly()
                         throws InterruptedException;

   public boolean tryLock();
   public boolean tryLock(long time, TimeUnit unit)
                         throws InterruptedException;

   public void unlock();

   public Condition newCondition();
```

Lock Fairness

- `ReentrantLock` allows locks to be granted **fairly**
  - Locks are granted as close to arrival order as possible
  - Prevents *lock starvation* from happening

- Possibilities for granting locks
  1. First-come-first-served
  2. Allows servicing the maximum number of requests
  3. Do what's best for the platform

The contents of this slide-set are based on the following references

- *Java Threads. Scott Oaks and Henry Wong. . 3rd Edition. O'Reilly Press. ISBN: 0-596-00782-5/978-0-596-00782-9.* [Chapters 3, 4]