CSx55: Distributed Systems [Threads & Safety]

Threads have you in a bind?
With Objects and Concurrency at play
Are nerves about to fray?
Here's something to have those worries abate
It's just about access to shared, mutable state

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- When to use synchronized (implicit) vs explicit locks?
- When to use `tryLock()` over `lock()`?
- `Condition` on the `Lock` interface?

2

Topics covered in this lecture

- wait-notify
- Thread safety

3

4

Objects and communications

- Every object has a lock
- Every object also includes mechanisms that allow it to be a **waiting area**
  - Allows *communication* between threads

Conditions

- One thread needs a **condition** to exist
  - Assumes another thread will *create* that condition
- When another thread creates the condition?
  - It **notifies** the first thread that has been **waiting** for that condition

`wait(), notify()` and the `Object` class

```
public class Object {

   public void wait();

   public void wait(long timeout);

   public void notify();
}
```


`wait(), notify()` and the `Object` class

- Wait-and-notify mechanisms are available for every object
  - Accomplished by **method invocations**
- Synchronized mechanism is handled by using a *keyword*

Wait-and-notify relate to synchronization, but …

- It is more of a **communications mechanism**
- Allows one thread to communicate to another that a **condition** has occurred
  - Does not specify *what* that specific condition is

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT THREADS & THREAD SAFETY L6.9

9

Can wait-and-notify replace the synchronized mechanism?

- No
- <u>Does not</u> solve the race condition that the synchronized mechanism solves
- Must be used in **conjunction** with the synchronized lock
  - Prevents race condition that exists in the `wait-notify` mechanism itself

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT THREADS & THREAD SAFETY L6.10

10

A code snippet that uses wait-notify to control the execution of the thread

```
public class Tester implements Runnable {

   private boolean done = true;

   public synchronized run() {
      while (true) {
         if (done) wait();
         else { ... Logic ... wait(100);}

      }
   }

   public synchronized void setDone(boolean b) {
      done = b;
      if (!done) notify();
   }
}
```


About the `wait()` method

- When `wait()` executes, the synchronization lock is *released*
  - By the JVM
- When a notification is received?
  - The thread needs to *reacquire* the synchronization lock before returning from `wait()`

Integration of wait-notify and synchronization

- **Tightly integrated** with the synchronization lock
  - Feature not directly available to us
  - Not possible to implement this: native method
- This is typical of approach in other libraries
  - *Condition variables* for Solaris and POSIX threads require that a mutex lock be held

Details of the race condition in the wait-notify mechanism

- The first thread *tests the condition* and confirms that it must wait
- The second thread *sets the condition*
- The second thread calls `notify()`
  - This **goes unheard** because the first thread is not yet waiting
- The first thread calls `wait()`

How does the potential race condition get resolved?

- To call `wait()` or `notify()`
  - Obtain lock for the object on which this is being invoked
- It seems as if the lock has been held for the entire `wait()` invocation, but …
  1. `wait()` *releases lock prior to waiting*
  2. *Reacquires the lock just before returning* from `wait()`

Is there a race condition during the time `wait()` releases and reacquires the lock?

- `wait()` is **tightly integrated** with the lock mechanism
- Object lock is **not freed until** the waiting thread is in a *state in which it can receive notifications*
  - System prevents race conditions from occurring here

If a thread receives a notification, is it guaranteed that condition is set?

- No
- *Prior* to calling `wait()`, *test condition* while holding lock
- Upon *returning* from `wait()` *retest* condition to see if you should `wait()` again

What if `notify()` is called and no thread is waiting?

- Wait-and-notify mechanism has no knowledge about the condition about which it notifies
- If `notify()` is called when no other thread is waiting?
  - The notification is lost

What happens when more than 1 thread is waiting for a notification?

- Language specification does not define which thread gets the notification
  - Based on JVM implementation, scheduling and timing issues
- *No way to determine* which thread will get the notification

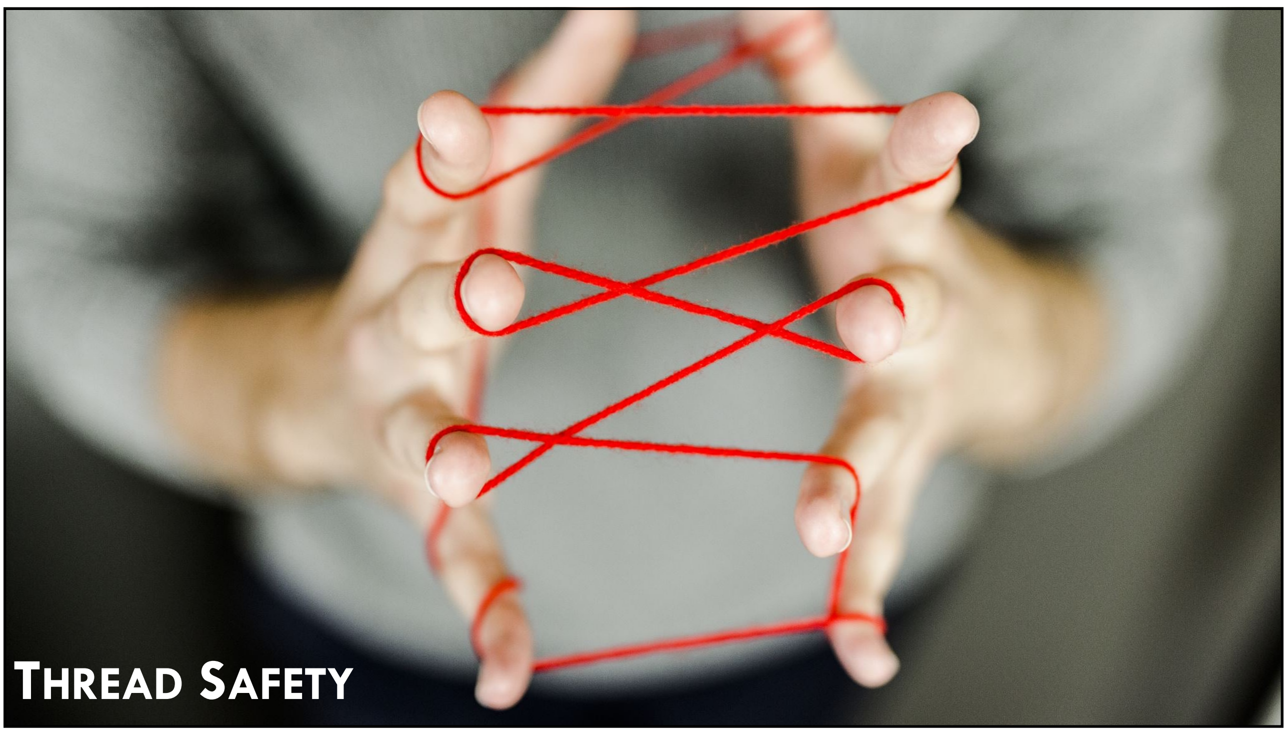
`notifyAll()`

- All threads that are waiting on an object are notified
- When threads receive this, they must work out
  1. Which thread should continue
  2. Which thread(s) should call `wait()` again
     - All threads wake up, but they **still have to reacquire the object lock**
     - Must wait for the lock to be freed

Threads and locks: Summary

- **Locks are held by threads**
  - A thread can hold <u>multiple locks</u>
    - Any thread that tries to obtains these locks? Placed into a wait state
    - If the thread deadlocks? It results in all locks that it holds becoming unavailable to other threads
- If a lock is held by some other thread?
  - The thread *must wait* for it to be free: **There is no preemption of locks**!
  - If the lock is unavailable (or held by a deadlocked thread) it blocks all the waiting threads

21



22

Race conditions

- Getting the right answer depends on lucky timing
  - E.g., check-then-act: When stale observations are used to make a decision on what to do next
- Real world example
  - Our previous example of 2 friends trying to meet up for coffee on campus without specifying which of the 2 locations

Racing and synchronization [1/3]

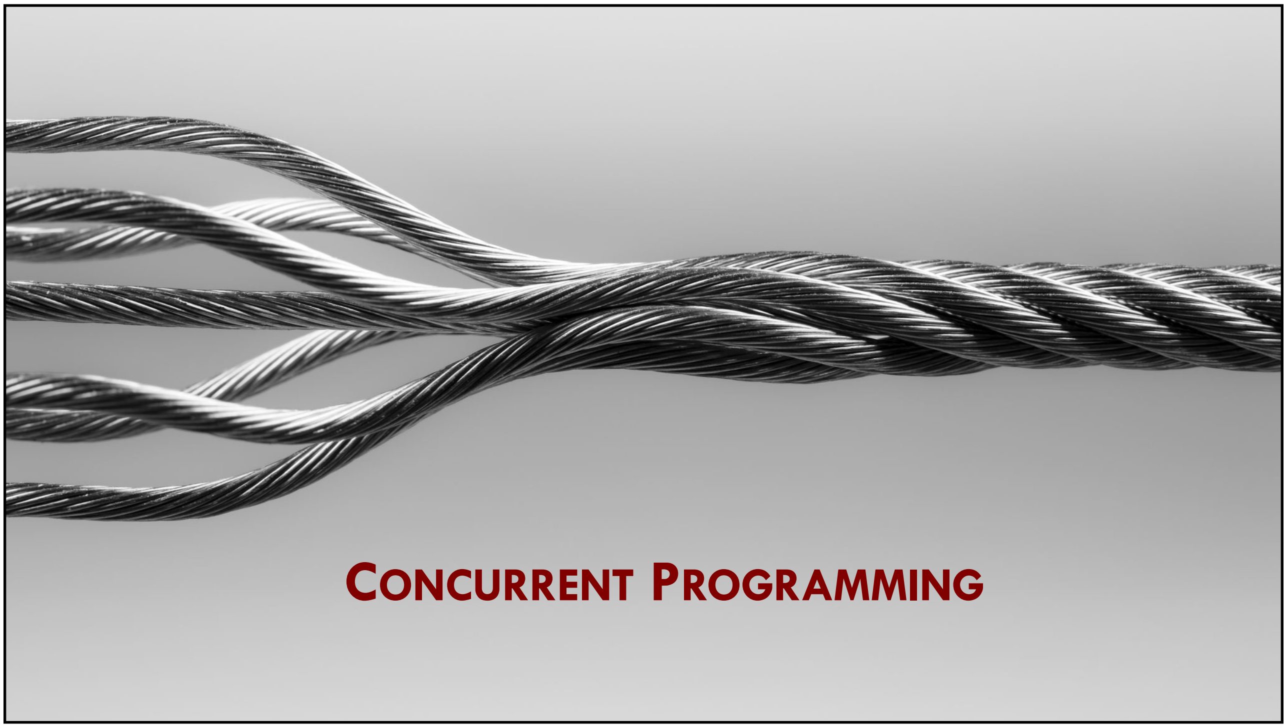
- Purpose of **synchronization**?
  - Prevent race conditions that can cause data to be found in either an inconsistent or intermediate state
- Threads are not allowed to race during sections of code protected by synchronization
  - But this does not mean outcome or order of execution of threads is deterministic
    - Threads may be racing prior to the synchronized section of code

Racing and synchronization [2/3]

- If threads are waiting on the same lock
  - The order in which the synchronized code is executed is determined by order in which lock is granted
    - Which is platform-specific and non-deterministic

Racing and synchronization [3/3]

- Not all races should be avoided
  - This is a subtle but important point: If you do this … everything is serialized!
  - **Only race-conditions within thread-unsafe sections of the code** are considered a problem
    1. Synchronize code that prevents race condition
    2. Design code that is thread-safe without the need for synchronization (or requires minimal synchronization)



27

Concurrent programming

- Concurrent programs require the **correct use** of threads and locks
- But these are just *mechanisms*

28

Object State

- Includes its **data**
  - Stored in instance variables or static fields
  - Fields from dependent objects
    - `HashMap`'s state also depends on `Map.Entry<K, V>` objects
- Encompasses any data that can affect its *externally visible* behavior

The crux of developing thread safe programs

- Managing access to **state**
  - In particular *shared, mutable state*
- Shared
  - Variables could be accessed by multiple threads
- Mutable
  - Variable's values change over its lifetime
- Thread-safety
  - **Protecting data from uncontrolled concurrent access**

When to coordinate accesses

- Whenever more than one thread accesses a state variable, and one of them **might write** to it?
  - They must all coordinate their access to it
- Avoid temptation to think that there are special situations when you can disregard this

When should an object be thread-safe?

- Will it be accessed from multiple threads?
- The key here is *how* the object is **used**
  - Not *what* it **does**

How to make an object thread-safe

- Use *synchronization* to **coordinate** access to mutable state
- Failure to do this?
  - Data corruptions
  - Problems that manifest themselves in myriad forms

Mechanisms for synchronization in Java

- One way to achieve this is via the `synchronized` keyword
  - Exclusive locking
- Other approaches include:
  - `volatile` variables
  - Explicit **locks**
  - **Atomic** variables

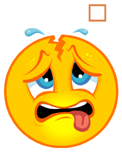
Programs that omit synchronizations

- Might work for some time
  - But it ***will break*** at some point
- Far easier to design a class to be thread-safe *from the start*
  - Retrofitting it to be thread-safe is extremely hard

35

Thread-safety: Encapsulate your state

- Fewer code should have access to a particular variable
  - Easier to reason about *conditions* under which it might be accessed
- **DON'T:**
  - Store state in `public` fields
  - Publish reference to an **internal** object



36

Fixing access to mutable state variables from multiple threads

- *Don't share* state variables across threads
- Make state variables *immutable*
- Use *synchronization* to coordinate access to the state variable

Correctness of classes

- Class conforms to **specification**
- **Invariants** constrain object's state
- **Post conditions** describe the effects of operations

A Thread-safe class

- **Behaves correctly** when accessed from multiple threads
- Regardless of *scheduling or interleaving* of execution of those threads
  - By the runtime environment
- No <u>additional synchronization or coordination</u> by the calling code

Really?

- Thread safe classes encapsulate *any needed* synchronization
- Clients <u>**should not**</u> have to provide their own

Stateless objects are always thread-safe

```
public class StatelessClass implements Servlet {

     public void factorizer(ServletRequest req,
                            ServletResponse resp) {
         BigInteger i = extractFromReq(req);
         BigInteger[] factors  = factorize(i);
         encodeIntoResponse(resp, factors);
     }

}
```


Stateless objects are always thread-safe

- **Transient state** for a particular computation exists solely in *local variables*
  - Stored on the thread's stack
  - Accessible only to the executing thread
- One thread cannot influence the result of another
  - The threads have no shared state

Atomicity

- Let's look at two operations **A** and **B**
- From the perspective of thread executing **A**
- When another thread executes **B**
  - Either all of **B** has executed or none of it has
- Operations **A** and B are **atomic** *with respect to each other*

Initializing Objects

```
public class LazyInitialization {

    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
       if (instance == null) {
         instance = new ExpensiveObject();
       }
       return instance;
    }

}
```

Thread-safe initialization

```
public class Singleton {
   private static final Singleton instance = new Singleton();

   // Private constructor prevents instantiation from other
   // classes
   private Singleton() { }

   public static Singleton getInstance() {
      return instance;
   }
}
```

The `final` keyword

- You cannot extend a `final` class
  - E.g., `java.lang.String`
- You cannot override a `final` method
- You can only initialize a `final` variable **once**
  - Either via an initializer or an assignment statement

Blank `final` instance variable of a class

- Must be assigned *within every constructor* of the class
- Attempting to set it outside the constructor will result in a compilation error
- The value of a `final` variable is not necessarily known at compile time

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]
- *Java Concurrency in Practice. Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Addison-Wesley Professional. ISBN: 0321349601/978-0321349606. [Chapters 1, 2, 3 and 4]*