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

---

Frequently asked questions from the previous class survey

- Why is load atomic but not ++ or --?
- Does a thread gain locks in the order that it requests them?
- How do we fix Heisenbugs?
- Why is better the `Lock` interface or a `synchronized` block?
- Can you introduce consistency issues with `Lock`?

---

Topics covered in this lecture

- Thread safety
- Compound actions
- Reentrancy

---

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

```java
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();
    }
}
```

---

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. Recaptures 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

What if `notifyAll()` is called?

- 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

Threads and locks

- Locks are held by threads
  - A thread can hold multiple locks
  - Any thread that tries to obtain 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
THREAD SAFETY

February 14, 2019

CS455: Introduction to Distributed Systems [Spring 2019]
Dept. Of Computer Science, Colorado State University

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 example from last class 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 ... every thing 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 minimal synchronization)

CONCURRENT PROGRAMMING

February 14, 2019

CS455: Introduction to Distributed Systems [Spring 2019]
Dept. Of Computer Science, Colorado State University
Concurrent programming

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

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

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

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 additional synchronization or coordination by the calling code
Really?

- Thread safe classes encapsulate any needed synchronization
- Clients should not have to provide their own

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

```java
public class LazyInitialization {
    private ExpensiveObject instance = null;
    public ExpensiveObject getInstance() {
        if (instance == null) {
            instance = new ExpensiveObject();
        }
        return instance;
    }
}
```

Thread-safe initialization

```java
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

Atomicity with compound operations

```java
public class CountingFactorizer {
    private long count = 0;
    public long getCount() { return count; }
    public void factorizer(int i) {
        int[] factors = factor(i);
        count++;
    }
}
```

Atomicity with compound operations

```java
public class CountingFactorizer {
    private final AtomicLong count = new AtomicLong(0);
    public long getCount() { return count; }
    public void factorizer(int i) {
        int[] factors = factor(i);
        count.incrementAndGet();
    }
}
```

Compound actions & thread-safety

- Compound actions
  - Check-then-act
  - Read-modify-write
- Must be executed atomically for thread-safety

**Locks & Reentrancy**
Reentrancy

- When thread requests lock held by another thread?
  - Requesting thread blocks
- If a thread attempts to acquire a lock it already holds?
  - Succeeds
- Locks are acquired on a per-thread rather than on a per-invocation basis

How reentrancy works [1/2]

- For each lock two items are maintained
  - Acquisition count
  - Owning thread
- When the count is zero?
  - Lock is free
- If a thread acquires lock for the first time?
  - Count is one

How reentrancy works [2/2]

- If owning thread acquires lock again, count is incremented
- When owning thread exits synchronized block, count is decremented
  - If it is zero .... Lock is released

Does this result in a deadlock?

```java
public class Widget {
    public synchronized void doSomething() {
        ...
    }
}
public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() + "Calling doSomething ()");
        super.doSomething();
    }
}
```

No! Intrinsic locks are reentrant

Guarding state with locks

- A mutable, shared variable that may be accessed by multiple threads must be guarded by the same lock
- For every invariant that involves more than one variable?
  - All variables must be guarded by the same lock
Watch for indiscriminate use of synchronization

- Every method in `Vector` is synchronized
- But this does not render compound actions on `Vector` atomic

```java
if (!vector.contains(element)) {
    vector.add(element);
}
```

- Snippet has race condition even though `add` and `contains` are atomic
- Additional locking needed for compound actions

Pitfalls of over synchronization

- Number of simultaneous invocations?
  - Not limited by processor resources, but is limited by the application structure
  - Poor concurrency

Antidote for poor concurrency

- Control the scope of the lock
  - Too large: Invocations become sequential
  - Don’t make it too small either
    - Operations that are atomic should not be in synchronized block

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