Threads, Concurrency and Synchronization

William Pugh, Univ. of Maryland
Doug Lea, SUNY Oswego
David Holmes, DLTeCH Pty Ltd

Why You Are Here

Learn strategies, patterns and gotcha’s in using the building blocks for concurrent Java applications.

Learning Objectives

- Understand the meaning of `synchronized`
- Understand the purpose of `interrupt`
- Understand use of `wait/notify/notifyAll`
- Spot some cases where attempts to be clever with threads and synchronization are wrong
  - Although powerful and useful, threads are subtle and dangerous

Speaker’s Qualifications

- David Holmes is a co-author of “The Java Programming Language, 3rd edition”
- Doug Lea is author of “Concurrent Programming in Java”, and has served on many JSRs
- William Pugh does research on the interaction of threads, programming languages and compilers
- Doug and David have presented tutorials on Java concurrency at various conferences over the past 5 years, including OOPSLA and ECOOP

Past, Present and Future

- Won’t talk about deprecated methods
  - Or problems in earlier VM’s
- Existing thread specification broken
  - Being revised as JSR-133
  - Revision is fairly close to existing practice
  - Some VM’s already compliant
- New Concurrency abstractions in JSR-166
  - Targeted for 1.5

These are Building Blocks

- In many cases, far more effective to use higher level concurrency abstractions
  - See Doug Lea’s concurrency library
  - And JavaOne talk on Concurrency JSR-166
- This talk describes the building blocks on which higher level concurrency abstractions are built
Synchronization

Java Synchronization

- Each Java object has an associated lock
- Use `synchronized(obj) { ... }` to acquire lock for duration of block
  - Locks automatically released
  - Locks are recursive
    - A thread can acquire the lock on an object multiple times
- Provides mutually exclusive access to code/data protected using the same lock

Three Aspects of Synchronization

- Atomicity
  - Prevention of interference through locking
- Visibility
  - Ensuring that changes made in one thread are seen in other threads
- Ordering
  - Ensuring that you aren’t surprised by the order in which statements are executed

What Synchronization Does

- Provides atomicity through mutual exclusion
- All code blocks synchronizing on the same object are ordered
  - Happen in some particular order
- Everything in one block occurs before and is visible to everything in a later block

When are actions visible and ordered with other Threads?

```
Thread 1
x = 1
unlock M
Thread 2
lock M
i = x
```

Agenda

- Synchronization
  - The meaning of `synchronized` (JSR-133)
  - Guidelines for synchronization
- Thread Communication
  - Thread interruption
  - Using `wait/notify/notifyAll`
- A few common idioms and pitfalls
When are actions visible and ordered with other Threads?

Thread 1

\[ y = 1 \]
\[ \text{lock } M \]
\[ x = 1 \]
\[ \text{unlock } M \]

Everything before the unlock

Thread 2

\[ \text{lock } M \]
\[ i = x \]
\[ \text{unlock } M \]
\[ j = y \]

Is visible to everything after the matching lock

What this means

- Don’t have to hold the lock while performing all the writes you want to be visible
  - Can prepare object first
  - Then hold lock while adding to shared buffer
- Association between the object locked and the data changed is up to the programmer
  - Obtaining a lock on an object doesn’t prevent other threads from modifying it
  - You can use a separate, private Object to protect a data structure

Holding Locks

- Use lock ordering if acquiring multiple locks
  - Avoids deadlock
- Think twice and thrice before holding locks on multiple objects at same time
  - Potential deadlock
  - \text{wait} only gives up lock on one object
  - Can block other threads for a long time
- Try to avoid holding locks while:
  - Sleeping or performing blocking I/O
  - Calling code you don’t control

Volatile Fields

- If you are going to access a shared field without using synchronization
  - It needs to be \text{volatile}
- Semantics for \text{volatile} have been strengthened in JSR-133
  - Many VM’s already compliant
- If you don’t try to be too clever
  - Declaring it \text{volatile} just works

Using Volatile

- A one-writer/many-reader value
  - Simple control flags:
    - \textit{volatile} boolean done = false;
  - Keeping track of a “recent value” of something

Misusing Volatile

- Incrementing a volatile field doesn’t work
  - In general, writes to a volatile field that depend on the previous value of that field don’t work
- A volatile reference to an object isn’t the same as having the fields of that object be volatile
  - No way to make elements of an array volatile
- Can’t keep two volatile fields in sync
Thread Communication

Interrupting Threads
- No way to force another thread to stop executing
  - Could leave things in inconsistent state
- Must ask nicely
  - Use `Thread.interrupt()`
- Other thread must be prepared to be interrupted
  - Code must be “interrupt friendly”

Checking for Interrupts
- `static boolean interrupted()`
  - True if current thread had been interrupted
  - Resets interrupted status
- `boolean isInterrupted()`
  - True if specific thread has been interrupted
  - Does not reset interrupted status
- Methods that throw `InterruptedException`
  - `wait`, `join`, `sleep`
  - Some I/O methods could

Don’t Hide Interrupts
- For library code
  - Any code that might be used by someone else
- Don’t hide the fact that an interrupt occurred
  - Rethrow the exception
    - Make interruption part of the method’s semantics
    - Or re-interrupt yourself
      - `Thread.currentThread().interrupt()`
      - You can ignore, but your caller can process

Using wait/notify/notifyAll
- Allows threads to `wait` for a condition to hold, until informed by another thread that it may hold
  - `notify` informs a single waiting thread
  - `notifyAll` informs all waiting threads
- Must hold the object’s lock before using
  - Protects access to the condition being checked
  - Lock is released while waiting

wait/notify/notifyAll Example
```java
class OnePlaceBuffer{
p  rivate Object o;
  p ublic synchronized Object take() throws InterruptedException {
    while (o == null) wait();
    Object result = o;
    o = null;
    notifyAll();
    return result;
  }
  p ublic synchronized void put(Object v)
    throws InterruptedException {
    while (o != null) wait();
    o = v;
    notifyAll();
  }
}
```
wait/notify/notifyAll Gotcha’s

- **wait** must be in a loop
  - Can’t assume that just because **wait** has returned, conditions are suitable to proceed
- **notify** rather than **notifyAll** is difficult
  - Optimization to reduce potential wake-ups
    - Only one thread can benefit, and
    - Right thread will always get selected, and
  - The above holds in subclasses
    - Lots of subtle problems can arise
- Avoid holding other locks when waiting
  - **wait** only gives up locks on that object

Idioms and Pitfalls

Cost of Synchronization

- Synchronization isn’t free
  - But is pretty cheap on modern VM’s
  - Cost really a factor only when wrapped around small methods
- Few suffer from the cost of synchronizing shared accesses
  - But some classes synchronize all access, even when not shared by threads

Idioms & Pitfalls

- Synchronization Cost
  - Avoiding (the need for) synchronization
- Fairness
- Transactions
- Finalizers
- Being overly clever
- Safety issues

Avoiding cost of Synchronization

- Classes that synchronize all accesses
  - 1.0 collection classes (Vector, HashTable)
  - java.io streams
  - StringBuffer
- Use 1.2 collection classes instead
  - Use synchronization wrappers if needed
- Do bulk reads/writes to I/O streams
  - Or use new I/O channels (java.nio)

Avoiding need for Synchronization

- Avoid sharing unnecessarily
  - Constrain objects to a single thread
  - ThreadLocals
- No changing state means no interference
  - Stateless objects: e.g., java.lang.Math
  - Immutable objects: e.g., Integer and String
    - Declare fields to be final
- Containment restricts access
  - Guarantee exclusive control of internal components
  - Synchronize at the container level
ThreadLocal

- Allows you to create a variable
  - With a separate value for each thread
  - Never needs synchronization
  - Can be used to cache values from shared structures
- ThreadLocal’s have gotten much faster

Being Overly Clever

- People sometimes try to carefully reason about cause-and-effect and ordering to find a way to communicate between threads without using synchronization or volatile fields
- Almost all such attempts are wrong
  - E.g., Double-checked locking
- Compilers and processors do reordering and transformations that thwart such reasoning

No Promises About Fairness

- Priorities might not mean anything
- May not have preemptive multithreading
  - Need to use yield/sleep to force context switch
- Yield/sleep may not mean anything
- But in a good VM, you will get some form of fairness
  - But don’t depend on the details

Transactions

- Be careful about relying on the built-in synchronization of library classes
- Often, you need to perform a series of actions atomically:
  - Check for element in Map;
    - if it doesn’t exist, add new entry
  - Iterate through collection
  - Write a full record to a stream
- Need external synchronization

Finalizers

- Lots of potential dangers in finalizers
- Finalizer may run while components of object are still in use
- Finalizers may be simultaneously invoked on different components of an unreachable object graph
- Finalizers are dubious in general
  - Cause lots of GC problems
  - When you really need them, use synchronization

Safety Issues

- Particularly for safety critical code
- Worry about security attacks through data races
- Synchronize carefully
Safe Immutable Objects

- Immutable objects can be subject to attack
  - One thread creates object
  - Passes reference to object to another thread via a race
  - Second thread might not see all writes done by constructor
- Potential problem with `java.lang.String`
- Fixed by declaring all fields final
  - Under new JSR-133 semantics

Summary

- Correct multithreaded code is always harder than you think it is
  - Even for us
- Write code as through the threads are conspiring to get you
- Synchronization is your friend
- Consider using libraries if you need more complicated concurrency abstractions

If You Only Remember One Thing...

Keep synchronization simple, follow the basic rules and don’t try to be too clever