CMSC 433
Thread safety
JCIP Ch. 2

Fall 2010

Administrivia

- The survey results are in. The policy will be
  - Pair programming with 35% projects, 50% exams
    - and 15% in-class exercises and participation
- This arrangement was nearly as popular as the “have your cake and eat it too” arrangement of pairs with 40% projects, 45% exams

Thread Safety

- A class is **thread safe** if it behaves *correctly* when *accessed from multiple threads*, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and *with no additional synchronization or other coordination on the part of the calling code*.
- Examples
  - thread safe: java.util.Vector
  - not thread safe: java.util.ArrayList

The key issue: mutable state

- *Thread safety errors arise when multiple threads interact with shared, mutable state*
- Corollaries: concurrency errors can be avoided by
  - Avoiding state altogether
  - Making state not shared (confining it)
  - Making shared state immutable

Atomicity

- Operations A and B are **atomic** with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either *all of B has executed or none of it has*.
- An **atomic operation** is one that is *atomic with respect to all operations*, including itself, that operate on the same state.
- Examples: atomic: x = y; not atomic: x = x + 1;
- Atomicity useful building block for reasoning about correctness

Race Conditions

- Race conditions often result in violations of atomicity. Occur when relative timing of two threads results in different answers
  - Common types:
    - *check-then-act*; e.g., lazy initialization
    - *read-modify-write*; e.g., update to counter
  - **Principle**: *Where practical, use existing thread-safe objects, like AtomicLong, to manage your class's state.*

Locking

- Goal: update related state variables atomically
  - Can't do this just with AtomicXXX values. Need to resort to locking to prevent simultaneous access.
- **Intrinsic locks** (monitor locks) are used in synchronized statements and methods
  - Only one thread may acquire a lock at a time
  - Intrinsic locks are *reentrant*, meaning the same thread may acquire the lock more than once

Guarding State

- If access to a state variable is coordinated by synchronization at one point, it must coordinated *everywhere the variable is accessed*.
  - We say that variable is **guarded by** a particular lock
    - Should make it clear to maintainers which lock
- For every invariant involving more than one variable, all the variables involved must be guarded by the *same* lock

Liveness and Performance

- Can have too much of a good thing: too much locking leads to performance degradation and/or deadlock.
  - For example, try to not hold a lock for a lengthy computation (e.g., input/output)
- But: be safe (simple) first, then optimize