Concurrency in Object Oriented Programs 2

Object-Oriented Software Development COMP4001
CSE UNSW Sydney

Lecturer: John Potter
Outline

- Atomic Operations
  - Use of Volatile
  - java.util.concurrent.atomic

- Overview of Concurrency Utilities
  - java.util.concurrent
  - Concurrent Collections
Atomic Operations

- Atomic actions are indivisible
  - no thread will see any intermediate state of an atomic action
    - i.e. no interleaving of atomic actions
- Java guarantees reading/writing of all basic types from/to memory is atomic:
  - for reference or primitive values
    - EXCEPT long or double
- Accessing variables of these types may not require synchronisation
  - but using them might (check-and-act)
Thread Caches

- Each Java thread has a working memory that interacts with main memory
  - Strict rules on how variables move between them
- Working memory acts like a cache
  - Locking a lock
    - Forces accessed variables to be loaded from main memory
  - Unlocking a lock
    - Forces all modified variables to be written to main memory
- Variables that are not modified in a section of code may only get loaded once e.g. loops
- In Java, threads only share memory via objects on the heap
  - so the only variables we care about for synchronization are fields (whether static or not)
The **volatile** Keyword

- Any variable declared as volatile:
  - Read from main memory on every access
  - Written to main memory on every assignment
  - Not cached in registers or in caches
- Make read/write atomic on long and double as well
  ```java
  public volatile long volaVar;
  ```
- Establish a happens-before relationship
  - A write on a volatile happens-before subsequent reads
  - A read on a volatile always returns the most recent write by any thread
- More efficient than synchronized code
  - Accessing a volatile variable performs no locking
When to Use volatile

- When the variable is shared by multiple threads
  - and you require the thread to always access the most current value

- If a thread accesses the variable more than once, be careful
  - Do you need to preserve the value between accesses, without interference by other threads?
    - If so, a volatile is not adequate

- Care is needed to avoid race conditions. Be concerned that
  - Writes to the variable do not depend on its current value
  - The variable does not participate in invariants with other state variables
  - Locking is not required for any other reason while the variable is being accessed
Disadvantages of Locking

- Threads acquire locks to gain exclusive access to shared variables
- Suspending/resuming threads has lot of overhead
  - Context switches, thread scheduling, etc.
- When a thread is waiting for a lock, it cannot do anything else
- Performance hazard
  - High priority thread is blocked by lower priority threads
- Liveness hazard
  - Deadlock, livelock, starvation
Limitations of volatile Variables

- Volatile variables are a lighter-weight synchronization mechanism
  - But rather limited compared to locking
- Cannot be used to construct atomic compound operations
  - When one variable depends on another
  - When the new value of a variable depends on its old value
- E.g., `counter++` is three distinct operations (read-modify-write)
  - how can we make that appear as a single, indivisible operation? i.e. how can we make the increment be atomic?
Outline

- Atomic Operations
  - Use of Volatile
  - java.util.concurrent.atomic

- Overview of Concurrency Utilities
  - java.util.concurrent
  - Concurrent Collections
Atomic Variables Classes

- The `java.util.concurrent.atomic` package
  - AtomicInteger, AtomicLong, AtomicBoolean and AtomicReference are most commonly used
- Support atomic operations on single variables
- `get()` and `set()` methods to read and write on volatile variables
  - Set/write has a happens-before relation with any subsequent get/read on the same variable
- `compareAndSet()` method
  - Has the memory effect of both reading and writing a volatile variable
Atomic Array Classes

- AtomicIntegerArray, AtomicLongArray and AtomicReferenceArray
- Elements can be updated atomically
  - Provide volatile access semantics to the elements of array
- Have `get(int index)` and `set(int index, int newValue)` methods to get and set element at given position atomically
  - Also have `compareAndSet`, `getAndSet` methods
- A volatile array has volatile semantics only for the array reference, not for its elements
AtomicIntegerArray

/** * Set the element at position i to the given value and return the
 * old value.
 *
 * @param i the index
 * @param newValue the new value
 * @return the previous value
 */

public final int getAndSet(int i, int newValue) {
    while (true) {
        int current = get(i);
        if (compareAndSet(i, current, newValue))
            return current;
    }
}
Counter Example

class Counter {
    private int c = 0;
    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}
Counter Example via Locking

class Counter {
    private int c = 0;
    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}

class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
Counter Example without Lock

class Counter {
    private int c = 0;
    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}

import java.util.concurrent.atomic.*;

class AtomicCounter {
    private AtomicInteger c
        = new AtomicInteger(0);
    public void increment() {
        c.incrementAndGet();
    }
    public void decrement() {
        c.decrementAndGet();
    }
    public int count() {
        return c.get();
    }
}
Outline

- Atomic Operations
  - Use of Volatile
  - java.util.concurrent.atomic

- Overview of Concurrency Utilities
  - java.util.concurrent
  - Concurrent Collections
Concurrency Utilities in Java

- Java library support for concurrency
  - Since Java 1.5
  - Doug Lea was key designer
- See `java.util.concurrent` package documentation
Concurrency Utilities – What?

- Atomic Variables
  - `java.util.concurrent.atomic` (see above)
- Concurrent Collections
- Task Scheduling Framework
  - `Task` model sits on top of threads
    - Uses thread pools
  - `Executor` control task execution
- Synchronizers
  - Semaphores, mutexes, latches, barriers
- Locks
  - Read-Write locks, Timeouts for Blocking, Condition Variables
Concurrency Utilities – Why?

- Easier concurrent programming
- Increased productivity
  - e.g. builds on existing Collections
- Increased performance
- Increased reliability
- Improved maintainability of concurrent apps

- Achieved by carefully reviewed implementations of standard concurrency abstractions in library
  - Don’t try to duplicate this effort yourself!
Outline

- Atomic Operations
  - Use of Volatile
  - java.util.concurrent.atomic

- Overview of Concurrency Utilities
  - java.util.concurrent
  - Concurrent Collections
Concurrent Collections

- Existing Collections framework in java.util
  - Standard interfaces List, Set, Map
  - Synchronization wrappers
- Concurrent Collections provide
  - High performance, concurrent implementations of standard interfaces
  - Interfaces Queue, BlockingQueue, BlockingDeque
    - With concurrent implementations
Queues

- Queue and Deque interfaces added to java.util
- Standard methods throw exceptions if precondition is violated
  - `add(e), remove(), element()`
  - e.g. add to full container, remove or get from empty
  - Not appropriate for concurrent access
    - Different threads do not know current state of queue
    - Need to check state before applying update
      - Need to avoid interference (a data race) between check and update
  - Special methods designed for concurrent access
    - `offer(e), poll(), peek()`
    - Return special value (usually `null`) to indicate no success
Queue

- **ConcurrentLinkedQueue** provides a concurrent implementation of `Queue`
  - Standard `List` classes also implement `Queue`
- **BlockingQueue** extends `Queue` interface
  - operations (put and take) block if preconditions are not satisfied
  - Implemented in `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue`, and `DelayQueue`
- Cover the most common usage contexts
  - producer-consumer, messaging, parallel tasking, and related concurrent designs (see later)
Concurrent Implementations

- ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, and CopyOnWriteArraySet.

- Prefix “Concurrent” here typically indicates:
  - Designed for concurrent access by multiple threads
    - Do not use a single exclusion lock / monitor
  - Usually allow concurrent reads
    - Sometimes also concurrent writes
  - In contrast, “Synchronized” collections are single-threaded
  - E.g. java.util.Hashtable and Collections.synchronizedMap(new HashMap()) are synchronized
    - They block attempted concurrent access
Synchronized vs Concurrent Collections

- Use synchronized collections to achieve thread-safety, but not for concurrent performance
  - Simple, single lock imposes single-threadedness
  - Does not scale well
    - Lock is a bottleneck when concurrent demand is high
    - We call this situation high contention
- Use concurrent collections to get multiple access
  - Provide both safety and good performance
- Use an unsynchronized collection if
  - Collection is not shared, or all access is protected (indirectly) by some other lock
ConcurrentHashMap

- Typical usage is for initial construction with many writes
  - Followed by many reads, with few if any writes
- Read operations are not blocked
  - May overlap with updates
- The hash table is “striped” to allow concurrent updates to different parts by hash index
  - Performance trade-off between number ("granularity") of locks and number of concurrent threads allowed access
    - Granularity can be set with a constructor
    - Check out javadoc and implementation source code in java.util.concurrent.ConcurrentHashMap
CopyOnWriteArrayList

- When the expected number of reads and traversals greatly outnumber the number of updates
  - A COWArrayList is preferable to a synchronized ArrayList
- Implementation guarantees that the underlying array is copied on each write (mutating) operation
- Its iterator takes the data array when it is constructed
  - This is a *snapshot* iterator
  - The iterator ignores any subsequent (overlapping) writes
Weakly Consistent Iterators

- Standard Java Collections (in java.util) provide fast-fail traversal iterators
  - Any use of an iterator after a modification of the data structure causes a concurrent access exception
- Most Concurrent Collections provide weakly consistent Iterators
  - Not fast-fail
- A weakly consistent iterator is thread-safe
  - Does not necessarily freeze the collection while iterating
  - It may (or may not) reflect any updates since the iterator was created