Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Concurrency

Introduction to concurrency, part 3

Concurrency primitives, libraries, and design patterns

Charlie Garrod  Bogdan Vasilescu
Administrivia

• Homework 5b due 11:59 p.m. Tuesday
  – Turn in by Wednesday 9 a.m. to be considered as a Best Framework

• Optional reading due today:
  – Java Concurrency in Practice, Chapter 10
Key concepts from Tuesday
Bad news: some simple actions are not atomic

- Consider a single 64-bit `long` value

- Concurrently:
  - Thread A writing high bits and low bits
  - Thread B reading high bits and low bits

Precondition:

```plaintext
long i = 10000000000;
```

Thread A:

```plaintext
i = 42;
```

Thread B:

```plaintext
ans = i;
```

<table>
<thead>
<tr>
<th>ans:</th>
<th>01001...00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>ans:</td>
<td>00000...00101010</td>
</tr>
<tr>
<td>ans:</td>
<td>01001...00101010</td>
</tr>
</tbody>
</table>

(100000000000)  
(42)  
(100000000042 or ...)
Avoiding deadlock

• **The *waits-for graph* represents dependencies between threads**
  – Each node in the graph represents a thread
  – An edge T1->T2 represents that thread T1 is waiting for a lock T2 owns

• **Deadlock has occurred iff the *waits-for graph* contains a cycle**

• **One way to avoid deadlock: locking protocols that avoid cycles**
Encapsulating the synchronization implementation

```java
public class BankAccount {
    private long balance;
    private final long id = SerialNumber.generateSerialNumber();
    private final Object lock = new Object();

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        BankAccount first = source.id < dest.id ? source : dest;
        BankAccount second = first == source ? dest : source;
        synchronized (first.lock) {
            synchronized (second.lock) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }

    ...

}  
```
An aside: Java Concurrency in Practice annotations

@ThreadSafe
class BankAccount {
    @GuardedBy("lock")
    private long balance;
    private final long id = SerialNumber.generateSerialNumber();
    private final Object lock = new Object();

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        BankAccount first = source.id < dest.id ? source : dest;
        BankAccount second = first == source ? dest : source;
        synchronized (first.lock) {
            synchronized (second.lock) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }
}
An aside: Java Concurrency in Practice annotations

- @ThreadSafe
- @NotThreadSafe
- @GuardedBy
- @Immutable
Today

• Strategies for safety
• Java libraries for concurrency
• Building thread-safe data structures
  – Java primitives for concurrent coordination
• Coming Tuesday: Program structure for concurrency
Policies for thread safety

• Thread-confined
• Shared read-only
• Shared thread-safe
  – Objects that perform internal synchronization
• Guarded
  – Objects that must be synchronized externally
Stack confinement

• Primitive local variables are never shared between threads
Thread confinement with java.lang.ThreadLocal<T>

• Sharable variable that confines state to each thread
  – Internally similar to a Map<Thread, T>

ThreadLocal<T>:

  T get(); // gets value for current thread
  void set(T value); // sets value for current thread
Shared read-only

- Immutable data is always safe to share
Shared thread-safe

- "Thread-safe" objects that perform internal synchronization
- Build your own, or know the Java concurrency libraries
java.util.concurrent is BIG (1)

- Atomic variables: java.util.concurrent.atomic
  - Support various atomic read-modify-write ops
- Executor framework
  - Tasks, futures, thread pools, completion service, etc.
- Locks: java.util.concurrent.locks
  - Read-write locks, conditions, etc.
- Synchronizers
  - Semaphores, cyclic barriers, countdown latches, etc.
java.util.concurrent is BIG (2)

- Concurrent collections
  - Shared maps, sets, lists
- Data exchange collections
  - Blocking queues, deques, etc.
- Pre-packaged functionality: java.util.Arrays
  - Parallel sort, parallel prefix
The `java.util.concurrent.atomic` package

- Concrete classes supporting atomic operations, e.g.:
  - `AtomicLong`
    ```java
type long get();
void set(long newValue);
long getAndSet(long newValue);
long getAndAdd(long delta);
long getAndIncrement();
boolean compareAndSet(long expectedValue, long newValue);
long getAndUpdate(LongUnaryOperator updateFunction);
long updateAndGet(LongUnaryOperator updateFunction);
...
```
public class SerialNumber {
    private static AtomicLong nextSerialNumber = new AtomicLong();

    public static long generateSerialNumber() {
        return nextSerialNumber.getAndIncrement();
    }
}

AtomicLong example
Overview of java.util.concurrent.atomic

- Atomic{Boolean,Integer,Long}
  - Boxed primitives that can be updated atomically
- AtomicReference<T>
  - Object reference that can be updated atomically
- Atomic{Integer,Long,Reference}Array
  - Array whose elements may be updated atomically
- Atomic{Integer,Long,Reference}FieldUpdater
  - Reflection-based utility enabling atomic updates to volatile fields
- LongAdder, DoubleAdder
  - Highly concurrent sums
- LongAccumulator, DoubleAccumulator
  - Generalization of adder to arbitrary functions (max, min, etc.)
Concurrent collections

- Provide high performance and scalability

<table>
<thead>
<tr>
<th>Unsynchronized</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashMap</td>
<td>ConcurrentHashMap</td>
</tr>
<tr>
<td>HashSet</td>
<td>ConcurrentHashMap</td>
</tr>
<tr>
<td>TreeMap</td>
<td>ConcurrentSkipListMap</td>
</tr>
<tr>
<td>TreeSet</td>
<td>ConcurrentSkipListSet</td>
</tr>
</tbody>
</table>
java.util.concurrent.ConcurrentHashMap

- Implements java.util.Map<K,V>
  - High concurrency lock striping
    - Internally uses multiple locks, each dedicated to a region of hash table
    - Externally, can use ConcurrentHashMap like any other map…
Atomic read-modify-write methods

- V putIfAbsent(K key, V value);
- boolean remove(Object key, Object value);
- V replace(K key, V value);
- boolean replace(K key, V oldValue, V newValue);
- V compute(K key, BiFunction<...> remappingFn);
- V computeIfAbsent(K key, Function<...> mappingFn);
- V computeIfPresent (K key, BiFunction<...> remapFn);
- V merge(K key, V value, BiFunction<...> remapFn);
java.util.concurrent.BlockingQueue

- Implements java.util.Queue\<E\>
- java.util.concurrent.SynchronousQueue
  - Each put directly waits for a corresponding poll
- java.util.concurrent.ArrayBlockingQueue
  - put blocks if the queue is full
  - poll blocks if the queue is empty
The CopyOnWriteArrayList

• Implements java.util.List<E>
• All writes to the list copy the array storing the list elements
Example: adding concurrency to the observer pattern

// Not thread safe. Contains a subtle bug.

private final List<Observer<E>> observers = new ArrayList<>();
public void addObserver(Observer<E> observer) {
    synchronized(observers) { observers.add(observer); }
}
public boolean removeObserver(Observer<E> observer) {
    synchronized(observers) { return observers.remove(observer); }
}
private void notifyOf(E element) {
    synchronized(observers) {
        for (Observer<E> observer : observers) {
            observer.notify(this, element);
        }
    }
}
Example: adding concurrency to the observer pattern

```java
private final List<Observer<E>> observers = new ArrayList<>();
public void addObserver(Observer<E> observer) {
    synchronized(observers) { observers.add(observer); }
}
public boolean removeObserver(Observer<E> observer) {
    synchronized(observers) { return observers.remove(observer); }
}
private void notifyOf(E element) {
    synchronized(observers) {
        for (Observer<E> observer : observers)
            observer.notify(this, element); // Risks liveness and
                // safety failures!
    }
}
```
One solution: \textit{snapshot iteration}

```java
private void notifyOf(E element) {
    List<Observer<E>> snapshot = null;

    synchronized(observers) {
        snapshot = new ArrayList<>(observers);
    }

    for (Observer<E> observer : snapshot) {
        observer.notify(this, element); // Safe
    }
}
```
A better solution: CopyOnWriteArrayList

private final List<Observer<E>> observers =
    new CopyOnWriteArrayList<>();

public void addObserver(Observer<E> observer) {
    observers.add(observer);
}

public boolean removeObserver(Observer<E> observer) {
    return observers.remove(observer);
}

private void notifyOf(E element) {
    for (Observer<E> observer : observers)
        observer.notify(this, element);
}
Defining your own thread-safe objects

- Identify variables that represent the object's state
- Identify invariants that constrain the state variables
- Establish a policy for maintaining invariants with concurrent access to state
Policies for thread safety (again)

- Thread-confined
- Shared read-only
- Shared thread-safe
  - Objects that perform internal synchronization
- Guarded
  - Objects that must be synchronized externally
A toy example: Read-write locks (a.k.a. *shared* locks)

Sample client code:

```java
private final RwLock lock = new RwLock();

lock.readLock();
try {
    // Do stuff that requires read (shared) lock
} finally {
    lock.unlock();
}

lock.writeLock();
try {
    // Do stuff that requires write (exclusive) lock
} finally {
    lock.unlock();
}
```
An aside: More Java primitives, for coordination

- **Goal**: *guarded suspension without spin-waiting*
  
  ```java
  volatile boolean ready = ...;
  while (!ready); // loop until ready...
  ```

- **Object methods for coordination**:
  ```java
  void wait();
  void wait(long timeout);
  void notify();
  void notifyAll();
  ```
A toy example: Read-write locks (implementation 1/2)

@ThreadSafe
public class RwLock {
    // State fields are protected by RwLock's \textit{intrinsic} lock

    /** Num threads holding lock for read. */
    @GuardedBy("this")
    private int numReaders = 0;

    /** Whether lock is held for write. */
    @GuardedBy("this")
    private boolean writeLocked = false;

    public synchronized void readLock() throws InterruptedException {
        while (writeLocked) {
            wait();
        }
        numReaders++;
    }
}
A toy example: Read-write locks (implementation 2/2)

```java
public synchronized void writeLock() throws InterruptedException {
    while (numReaders != 0 || writeLocked) {
        wait();
    }
    writeLocked = true;
}

public synchronized void unlock() {
    if (numReaders > 0) {
        numReaders--;
    } else if (writeLocked) {
        writeLocked = false;
    } else {
        throw new IllegalStateException("Lock not held");
    }
    notifyAll(); // Wake any waiters
}
```
Advice for building thread-safe objects

• Do as little as possible in synchronized region: get in, get out
  – Obtain lock
  – Examine shared data
  – Transform as necessary
  – Drop the lock

• If you must do something slow, move it outside the synchronized region
Documentation

- Document a class's thread safety guarantees for its clients
- Document a class's synchronization policy for its maintainers
- Use @ThreadSafe, @GuardedBy annotations
Summary of our RwLock example

• Generally, avoid wait/notify
• Never invoke wait outside a loop
  – Must check coordination condition after waking
• Generally use notifyAll, not notify
• Do not use our RwLock – it's just a toy
  – Instead, know the standard libraries...
    • Discuss: sun.misc.Unsafe