Principles of Software Construction: Objects, Design, and Concurrency

Part 5: Concurrency

Introduction to concurrency, part 3

Concurrency primitives, libraries, and design patterns

Charlie Garrod    Michael Hilton
Administrivia

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

• 17-355/17-665/17-819: Program Analysis
Key concepts from Tuesday
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
• Building thread-safe data structures
  – Java primitives for concurrent coordination
• Java libraries for concurrency
• Program structure for concurrency
Shared mutable state requires concurrency control

• Three basic choices:
  1. Don't mutate: share only immutable state
  2. Don't share: isolate mutable state in individual threads
  3. If you must share mutable state: *limit concurrency to achieve safety*
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>
  
  ```java
  ThreadLocal<T>:
  T get(); // gets value for current thread
  void set(T value); // sets value for current thread
  ```
Defining 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

• Thread-confined
• Shared read-only
• Guarded
  – Objects that must be synchronized externally
• Shared thread-safe
  – Objects that perform internal synchronization
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 (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();
}
```
A toy example: Read-write locks (implementation 1/2)

@ThreadSafe
class RwLock {
    // State fields are protected by RwLock's *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
}
```
Basic 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
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
    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>ConcurrentHashSet</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.concurrentBlockingQueue

- 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: *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
    }
}
```
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);
}
Today

- Strategies for safety
- Building thread-safe data structures
  - Java primitives for concurrent coordination
- Java libraries for concurrency
- Program structure for concurrency
Producer-consumer design pattern

• Goal: Decouple the producer and the consumer of some data
• Consequences:
  – Removes code dependency between producers and consumers
  – Producers and consumers can produce and consume at different rates
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 fork-join pattern

if (my portion of the work is small)
   do the work directly
else
   split my work into pieces
   invoke the pieces and wait for the results
The membrane pattern

- Multiple rounds of fork-join, each round waiting for the previous round to complete
Summary

• Reuse, don't build: know the j.u.c. libraries
• Use common patterns for program structure
  – Decompose work into independent tasks