Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Concurrency

Introduction to concurrency, part 3

Concurrency primitives, libraries, and design patterns

Josh Bloch          Charlie Garrod
Administrivia

- HW 5b due today, 11:59EDT (framework & plugin implementation)
- Optional reading due today: JCiP Chapter 12
Key concepts from Tuesday
Lock splitting for increased concurrency

Review: what’s the bug in this code?

```java
public class BankAccount {
    private long balance;

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

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        synchronized(source) {
            synchronized(dest) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }

    ...
}
```
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
Avoiding deadlock by ordering lock acquisition

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

    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) {
            synchronized (second) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }
}
```
Using a private lock to encapsulate synchronization

```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;
            }
        }
    }
}
```
Java Concurrency in Practice annotations

@ThreadSafe
public 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;
            }
        }
    }
}
Today

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

1. **Thread-confined state** – mutate but don’t share
2. **Shared read-only state** – share but don’t mutate
3. **Shared thread-safe** – object synchronizes itself internally
4. **Shared guarded** – client synchronizes object(s) externally
1. Three kinds of thread-confined state

• **Stack-confined**
  – Primitive local variables are *never* shared between threads
  – Fast and cheap

• **Unshared object references**
  – The thread that creates an object must take action to share (“publish”)  
  – e.g., put it in a shared collection, store it in a static variable

• **Thread-local variables**
  – Shared object with a separate value for each thread
  – Rarely needed but invaluable (e.g., for user ID or transaction ID)

```java
class ThreadLocal<T> {
    ThreadLocal() ;  // Initial value for each thread is null
    static <S> ThreadLocal<S> withInitial(Supplier<S> supplier);
    void set(T value); // Sets value for current thread
    T get();           // Gets value for current thread
}
```
2. Shared read-only state

- Immutable data is always safe to share
- So is mutable data that isn’t mutated
3. Shared thread-safe state

• Thread-safe objects that perform internal synchronization
• You can build your own, but not for the faint of heart
• You’re better off using ones from java.util.concurrent
• j.u.c also provides skeletal implementations
4. Shared guarded state

- Shared objects that must be locked by user
  - Most examples in the last two lectures. e.g., BankAccount
- Can be error prone: burden is on user
- High concurrency can be difficult to achieve
  - Lock granularity is the entire object
- You’re generally better off avoiding guarded objects
Outline

I. Strategies for safety
II. Building thread-safe data structures
III. Java libraries for concurrency (java.util.concurrent)
wait/notify – a primitive for cooperation

The basic idea is simple...

- State (fields) are guarded by a lock
- Sometimes, a thread can’t proceed till state is right
  - So it waits with `wait`
  - Automatically drops lock while waiting
- Thread that makes state right wakes waiting thread(s) with `notify`
  - Waking thread must hold lock when it calls `notify`
  - Waiting thread automatically acquires lock when it wakes up
But the devil is in the details

*Never invoke wait outside a loop!*

- Loop tests condition **before and after** waiting
- Test before skips **wait** if condition already holds
  - Necessary to ensure **liveness**
  - Without it, thread can wait forever!
- Testing after waiting ensure **safety**
  - Condition may not be true when thread wakes up
  - If thread proceeds with action, it can destroy invariants!
All of your waits should look like this

```java
synchronized (obj) {
    while (<condition does not hold>) {
        obj.wait();
    }

    ... // Perform action appropriate to condition
}
```
Why can a thread wake from a wait when condition does not hold?

- Another thread can slip in between notify & wake
- Another thread can invoke notify accidentally or maliciously when condition does not hold
  - This is a flaw in Java locking design!
  - Can work around flaw by using private lock object
- Notifier can be liberal in waking threads
  - Using notifyAll is good practice, but can cause extra wakeups
- Waiting thread can wake up without a notify(!)
  - Known as a spurious wakeup
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
A toy example: Read-write locks (a.k.a. shared/exclusive 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)

```java
@ThreadSafe public class RwLock {
    /** Number of threads holding lock for read. */
    @GuardedBy("this") // Intrinsic lock on RwLock object
    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
  - And any prose that is required
Summary of our RwLock example

• Generally, avoid wait/notify
  – Java.util.concurrent provides better alternatives
• 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
Outline

I. Strategies for safety
II. Building thread-safe data structures
III. Java libraries for concurrency (java.util.concurrent)
java.util.concurrent is BIG (1)

1. Atomic variables: java.util.concurrent.atomic
   – Support various atomic read-modify-write ops

2. Concurrent collections
   – Shared maps, sets, lists

3. Data exchange collections
   – Blocking queues, deques, etc.

4. Executor framework
   – Tasks, futures, thread pools, completion service, etc.

5. Synchronizers
   – Semaphores, cyclic barriers, countdown latches, etc.

6. Locks: java.util.concurrent.locks
   – Read-write locks, conditions, etc.
java.util.concurrent is BIG (2)

• Pre-packaged functionality: java.util.Arrays
  – Parallel sort, parallel prefix

• **Completable futures!**
  – Multistage asynchronous concurrent computations

• Flows
  – Publish/subscribe service

• And more
  – It just keeps growing
1. 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.)
Example: AtomicLong

class AtomicLong {   // We used this in generateSerialNumber()
    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);
    ...
}
2. 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>
You can’t prevent concurrent use of a concurrent collection

- This works for synchronized collections...
  ```java
  Map<String, String> syncMap =
      Collections.synchronizedMap(new HashMap<>());
  synchronized(syncMap) {
      if (!syncMap.containsKey("foo"))
          syncMap.put("foo", "bar");
  }
  ```

- But not for concurrent collections
  - They do their own internal synchronization
  - Never synchronize on a concurrent collection!
Instead, use 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);
Concurrent collection example: canonicalizing map

private final ConcurrentMap<T, T> map = new ConcurrentHashMap<>();

public T intern(T t) {
    String previousValue = map.putIfAbsent(t, t);
    return previousValue == null ? t : previousValue;
}
java.util.concurrent.ConcurrentHashMap

- Uses **many** techniques used to achieve high concurrency
  - Over 6,000 lines of code
- The simplest of these is *lock striping*
  - Multiple locks, each dedicated to a region of hash table
Aside: the producer-consumer 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
3. Data exchange collections summary

Hold elements for processing by another thread (producer/consumer)

- **BlockingQueue** – Supports blocking ops
  - ArrayBlockingQueue, LinkedBlockingQueue
  - PriorityBlockingQueue, DelayQueue
  - SynchronousQueue

- **BlockingDeque** – Supports blocking ops
  - LinkedBlockingDeque

- **TransferQueue** – BlockingQueue in which producers may wait for consumers to receive elements
  - LinkedTransferQueue
Summary of BlockingQueue methods

<table>
<thead>
<tr>
<th></th>
<th>Throws exception</th>
<th>Special value</th>
<th>Blocks</th>
<th>Times out</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td>add(e)</td>
<td>offer(e)</td>
<td>put(e)</td>
<td>offer(e, time, unit)</td>
</tr>
<tr>
<td><strong>Remove</strong></td>
<td>remove()</td>
<td>poll()</td>
<td>take()</td>
<td>poll(time, unit)</td>
</tr>
<tr>
<td><strong>Examine</strong></td>
<td>element()</td>
<td>peek()</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>
# Summary of BlockingDeque methods

## First element (head) methods

<table>
<thead>
<tr>
<th>Insert</th>
<th>AddFirst(e)</th>
<th>OfferFirst(e)</th>
<th>PutFirst(e)</th>
<th>OfferFirst(e, time, unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove</td>
<td>RemoveFirst()</td>
<td>PollFirst()</td>
<td>TakeFirst()</td>
<td>PollFirst(time, unit)</td>
</tr>
<tr>
<td>Examine</td>
<td>GetFirst()</td>
<td>PeekFirst()</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

## Last element (tail) methods

<table>
<thead>
<tr>
<th>Insert</th>
<th>AddLast(e)</th>
<th>OfferLast(e)</th>
<th>PutLast(e)</th>
<th>OfferLast(e, time, unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove</td>
<td>RemoveLast()</td>
<td>PollLast()</td>
<td>TakeLast()</td>
<td>PollLast(time, unit)</td>
</tr>
<tr>
<td>Examine</td>
<td>GetLast()</td>
<td>PeekLast()</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>
4. Executor framework overview

- **Flexible interface-based task execution facility**
- **Key abstractions**
  - Runnable, Callable&lt;T&gt; - kinds of tasks
- **Executor** – thing that executes tasks
- **Future&lt;T&gt;** – a promise to give you a T
- **Executor service** – Executor that
  - Lets you manage termination
  - Can produce Future instances
Executors – your one-stop shop for executor services

- `Executors.newSingleThreadExecutor()`
  - A single background thread
- `newFixedThreadPool(int nThreads)`
  - A fixed number of background threads
- `Executors.newCachedThreadPool()`
  - Grows in response to demand
A very simple (but useful) executor service example

• Background execution of a long-lived worker thread
  – To start the worker thread:
    ```java
    ExecutorService executor = Executors.newSingleThreadExecutor();
    ```
  – To submit a task for execution:
    ```java
    executor.execute(runnable);
    ```
  – To terminate gracefully:
    ```java
    executor.shutdown(); // Allows tasks to finish
    ```
Other things you can do with an executor service

• Wait for a task to complete
  
    Foo foo = executorSvc.submit(callable).get();

• Wait for any or all of a collection of tasks to complete
  
    invoke{Any,All}(Collection<Callable<T>> tasks)

• Retrieve results as tasks complete
  
    ExecutorCompletionService

• Schedule tasks for execution a time in the future
  
    ScheduledThreadPoolExecutor

• etc., ad infinitum
The fork-join pattern

if (my portion of the work is small)
do the work directly
else
    split my work into pieces
    recursively process the pieces
ForkJoinPool: executor service for ForkJoinTask

*Dynamic, fine-grained parallelism with recursive task splitting*

class SumOfSquaresTask extends RecursiveAction {
    final long[] a; final int lo, hi; long sum;
    SumOfSquaresTask(long[] array, int low, int high) {
        a = array; lo = low; hi = high;
    }

    protected void compute() {
        if (hi - lo < THRESHOLD) {
            for (int i = lo; i < hi; ++i)
                sum += a[i] * a[i];
        } else {
            int mid = (lo + hi) >>> 1;
            SumOfSquaresTask left = new SumOfSquaresTask(a, lo, mid);
            left.fork(); // pushes task
            SumOfSquaresTask right = new SumOfSquaresTask(a, mid, hi);
            right.compute();
            right.join(); // pops/runs or helps or waits
            sum = left.sum + right.sum;
        }
    }
}
5. Overview of synchronizers

- **CountDownLatch**
  - One or more threads to wait for others to count down
- **CyclicBarrier**
  - A set of threads wait for each other to be ready
- **Semaphore**
  - Like a lock with a maximum number of holders ("permits")
- **Phaser** – Cyclic barrier on steroids
- **AbstractQueuedSynchronizer** – roll your own!
6. Overview of java.util.concurrency.locks (1/2)

- **ReentrantReadWriteLock**
  - Shared/Exclusive mode locks with tons of options
    - Fairness policy
    - Lock downgrading
    - Interruption of lock acquisition
    - Condition support
    - Instrumentation

- **ReentrantLock**
  - Like Java’s intrinsic locks
  - But with more bells and whistles
Overview of `java.util.concurrent.locks` (2/2)

- **Condition**
  - `wait/notify/notifyAll` with multiple wait sets per object
- **AbstractQueuedSynchronizer**
  - Skeletal implementation of locks relying on FIFO wait queue
- **AbstractOwnableSynchronizer, AbstractQueuedLongSynchronizer**
  - Fancier skeletal implementations
ReentrantReadWriteLock example

*Does this look vaguely familiar?*

```java
private final ReentrantReadWriteLock rwl =
    new ReentrantReadWriteLock();

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

rwl.writeLock().lock();
try {
    // Do stuff that requires write (exclusive) lock
} finally {
    rwl.writeLock().unlock();
}
```
Summary

- `java.util.concurrent` is big and complex
- But it’s well designed and engineered
  - Easy to do simple things
  - Possible to do complex things
- Executor framework does for execution what collections did for aggregation
- This lecture just scratched the surface
  - But you know the lay of the land and the javadoc is good
- Always better to use `j.u.c` than to roll your own!