Principles of Software Construction: Concurrency, Pt. 3 – java.util.concurrent

Josh Bloch                Charlie Garrod
Administrivia

• Homework 5b due Tuesday 11:59 p.m.
  – Turn in your work by Wednesday 9 a.m. to be considered as a Best Framework
Can you find the bug?

Public service announcement (1/1)

/* From Linux 2.3.99 drivers/block/radi5.c */
static struct buffer_head * get_free_buffer(
    struct stripe_head *sh, int b_size) {
    struct buffer_head *bh;
    unsigned long flags;

    save_flags(flags);
    cli();
    if ((bh = sh->buffer_pool) == NULL)
        return NULL;
    sh->buffer_pool = bh->b_next;
    bh->b_size = b_size; restore_flags(flags);
    return bh;
}
Can you write a program to find the bug?

Public service announcement (2/2)

• Take Program Analysis (17-355) and learn (e.g.):
  – Abstract interpretation, a theory for reasoning about programs even before you know their input
  – Concolic testing, combines symbolic execution with randomized testing to exercise hard-to-reach corner cases
  – And more: interprocedural analysis, control-flow analysis, shape analysis, and dynamic analysis
• Then build awesome tools to find bugs, verify security properties, and generate tests!
• New course, Spring 2017
  – T/Th 10:30 in GHC 4102 Prof. Jonathan Aldrich
Key concepts from Tuesday...

- Never use wait outside of a while loop!
  - Think twice before using it at all
- Neither an under- nor an over-synchronizer be
  - Under-synchronization causes safety (& liveness) failures
  - Over-synchronization causes liveness (& safety) failures
- Two things that I should have said Tuesday...
1. Do as little as possible in synchronized regions

- Get in, get done, and get out
  - Obtain lock
  - Examine shared data
  - Transform as necessary
  - Drop lock

- If you must do something slow, move it outside synchronized region
2. Avoiding deadlock

- Deadlock caused by a cycle in waits-for graph
  - T1: `synchronized(a){ synchronized(b){ ... } }`
  - T2: `synchronized(b){ synchronized(a){ ... } }

- To avoid these deadlocks:
  - When threads have to hold multiple locks at the same time, **all threads obtain locks in same order**
java.util.concurrent is BIG (1)

I. Atomic vars - java.util.concurrent.atomic
   - Support various atomic read-modify-write ops

II. Executor framework
    - Tasks, futures, thread pools, completion service, etc.

III. Locks - java.util.concurrent.locks
     - Read-write locks, conditions, etc.

IV. Synchronizers
    - Semaphores, cyclic barriers, countdown latches, etc.
java.util.concurrent is BIG (2)

V. Concurrent collections
   – Shared maps, sets, lists

VI. Data Exchange Collections
   – Blocking queues, deques, etc.

VII. Pre-packaged functionality - java.util.arrays
   – Parallel sort, parallel prefix
I. Overview of java.util.atomic

- **Atomic{Boolean,Integer,Long}**
  - Boxed primitives that can be updated atomically
- **AtomicReference<T>**
  - Object reference that can be updated atomically
  - Cool pattern for state machine AtomicReference<StateEnum>
- **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.)
AtomicInteger example (review) [EJ Item 66]

public class SerialNumber {
    private static AtomicInteger nextSerialNumber = new AtomicInteger();

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

VI. Executor framework Overview

• Flexible interface-based task execution facility
• Key abstractions
  – Runnable, Callable<\text{T}> - kinds of tasks
• Executor – thing that executes tasks
• Future<\text{T}> – a promise to give you a \text{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 executor service example

• Background execution on 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
    ```
• Better replacement for our runInBackground and WorkQueue examples from previous lectures.
Other things you can do with an executor service

• Wait for a task to complete
  ```java
  Foo foo = executorSvc.submit(callable).get();
  ```
• Wait for any or all of a collection of tasks to complete
  ```java
  invoke{Any,All}(Collection<Callable<T>> tasks)
  ```
• Retrieve results as tasks complete
  ```java
  ExecutorCompletionService
  ```
• Schedule tasks for execution in the future
  ```java
  ScheduledThreadPoolExecutor
  ```
• etc., ad infinitum
ForkJoinPool: executor service for ForkJoinTask instances

class SumSqTask extends RecursiveAction {
    final long[] a; final int lo, hi; long sum;
    SumSqTask(long[] array, int low, int high) {
        a = array; lo = low; hi = high;
    }
    protected void compute() {
        if (h - l < THRESHOLD) {
            for (int i = l; i < h; ++i)
                sum += a[i] * a[i];
        } else {
            int mid = (lo + hi) >>> 1;
            SumSqTask left = new SumSqTask(a, lo, mid);
            left.fork(); // pushes task
            SumSqTask right = new SumSqTask(a, mid, hi);
            right.compute();
            right.join(); // pops/runs or helps or waits
            sum = left.sum + right.sum;
        }
    }
}

II. Overview of j.u.c.locks (1)

- **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 j.u.c.locks (2)

• Condition
  – wait/notify/notifyAll with multiple wait sets per object

• AbstractQueuedSynchronizer
  – Skeletal implementation of locks relying on FIFO wait queue

• AbstractOwnableSynchronizer, AbstractQueuedLongSynchronizer
  – More skeletal implementations
ReentrantReadWriteLock example

Does this look vaguely familiar?

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();
}
III. 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!
CountDownLatch example

Concurrent timer [EJ Item 69]

```java
public static long time(Executor executor, int nThreads,
                        final Runnable action) throws InterruptedException {
    CountDownLatch ready = new CountDownLatch(nThreads);
    CountDownLatch start = new CountDownLatch(1);
    CountDownLatch done = new CountDownLatch(nThreads);
    for (int i = 0; i < nThreads; i++) {
        executor.execute(() -> {
            ready.countDown(); // Tell timer we're ready
            try {
                start.await(); // Wait till peers are ready
                action.run();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                done.countDown(); // Tell timer we're done
            }
        });
    }
    ready.await(); // Wait for all workers to be ready
    long startNanos = System.nanoTime();
    start.countDown(); // And they're off!
    done.await(); // Wait for all workers to finish
    return System.nanoTime() - startNanos;
}
```
IV. 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 exclude concurrent activity from 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!
Concurrent collections have prepackaged 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

```java
private static final ConcurrentMap<String, String> map =
    new ConcurrentHashMap<String, String>();

// This implementation is OK, but could be better
public static String intern(String s) {
    String previousValue = map.putIfAbsent(s, s);
    return previousValue == null ? s : previousValue;
}
```
An optimized canonicalizing map
[EJ Item 69]

- ConcurrentHashMap optimized for read
  - So call get first, putIfAbsent only if necessary

// Good, fast implementation!
public static String intern(String s) {
    String result = map.get(s);
    if (result == null) {
        result = map.putIfAbsent(s, s);
        if (result == null)
            result = s;
    }
    return result;
}
Concurrent observer pattern requires open calls

This code is prone to liveness and safety failures!

```java
private final List<SetObserver<E>> observers =
    new ArrayList<SetObserver<E>>();
public void addObserver(SetObserver<E> observer) {
    synchronized(observers) {
        observers.add(observer);
    }
}
public boolean removeObserver(SetObserver<E> observer) {
    synchronized(observers) {
        return observers.remove(observer);
    }
}
private void notifyElementAdded(E element) {
    synchronized(observers) {
        for (SetObserver<E> observer : observers)
            observer.notifyAdded(this, element); // Callback!
    }
}
```
A decent solution: *snapshot iteration*

private void notifyElementAdded(E element) {
    List<SetObserver<E>> snapshot = null;

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

    for (SetObserver<E> observer : snapshot) {
        observer.notifyAdded(this, element); // Open call
    }
}
A better solution: CopyOnWriteArrayList [EJ Item 67]

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

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

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

private void notifyElementAdded(E element) {
    for (SetObserver<E> observer : observers)
        observer.notifyAdded(this, element);
}
V. Data exchange collections summary

*Hold elements for processing by another thread*

- **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></th>
<th>Throws exception</th>
<th>Special value</th>
<th>Blocks</th>
<th>Times out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>addFirst(e)</td>
<td>offerFirst(e)</td>
<td>putFirst(e)</td>
<td>offerFirst(e, time, unit)</td>
</tr>
<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></th>
<th>Throws exception</th>
<th>Special value</th>
<th>Blocks</th>
<th>Times out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>addLast(e)</td>
<td>offerLast(e)</td>
<td>putLast(e)</td>
<td>offerLast(e, time, unit)</td>
</tr>
<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>
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 framework did for aggregation
• This talk 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!**