CMSC 433 – Programming Language Technologies and Paradigms

Java Concurrency Utilities
Overview

- Synchronized Collections
- Concurrent Collections
  - ConcurrentHashMap
  - CopyOnWriteArrayList
  - BlockingQueues
- Synchronizers
  - Latches
  - FutureTask
  - Semaphores
  - Barriers & Exchangers
Why Concurrency Utilities

• Java’s built-in concurrency primitives – wait(), notify(), and synchronized are limited
  – Hard to use correctly; Easy to use incorrectly
  – Too low level for many applications
  – Can perform poorly if used incorrectly
    • Less of an issue now than it used to be
  – Leave out useful concurrency constructs
java.util.concurrent Goals

- Provide efficient, correct & reusable concurrency building blocks
- Enhance scalability, performance, readability, maintainability, and thread-safety of concurrent Java applications
Synchronized Collections

- Many of the Collection classes are not thread-safe
- Can use methods in the Collections class to add some thread safety to these classes, e.g.,
  
  ```java
  List list = Collections.synchronizedList(new ArrayList());
  ```
public int size() {
    synchronized(mutex) {return c.size();}
}

public boolean isEmpty() {
    synchronized(mutex) {return c.isEmpty();}
}

public boolean contains(Object o) {
    synchronized(mutex) {return c.contains(o);}
}
Synchronized Collections

- Must handle compound actions manually
- For example,

```java
public static void deleteLast(List list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}
```
Synchronized Collections

- **Iterators must be manually synchronized**

```java
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e : c)
        foo(e);
}
```
List list = Collections.synchronizedList(new ArrayList());

synchronized(list) {
    Iterator i = list.iterator(); // Must be in synchronized block
    while (i.hasNext())
        foo(i.next());
}
Non-Obvious Use of Iterators

```java
public class HiddenIterator {
    private final Set<Integer> set = new HashSet<Integer>();
    public void foo() {
        System.out.println(set);
    }
}
```
ConcurrentModificationException

• Can’t modify a synchronized collection while iterating over it
  – Iterator may throw ConcurrentModificationException

• Need to manually lock collection
  – Or copy the collection and iterate over the copy
Concurrent Collections

• java.util.concurrent includes concurrent collection classes
• Allows multiple operations to overlap
  – Some differences in semantics
• Some examples
  – ConcurrentHashMap
  – CopyOnWriteArrayList
  – ArrayBlockingQueue
Concurrent Hash-based Map

- Uses lock striping, rather than a single lock
- Allows concurrent readers
- Readers operate concurrently with writers
- Limited number of writers can modify the map concurrently
ConcurrentHashMap Iterators

• Iterate over the elements that existed when the iterator was created
• Tolerate updates while iterating
  – No ConcurrentModificationException
• May or may not reflect updates that occur while iterating
ConcurrentHashMap Tradeoffs

- size() and isEmpty() are only approximations
- No built-in way to lock the entire map
- Supports several compound operations

```java
public interface ConcurrentMap<K,V> extends Map<K,V> {
    // Insert into map only if no value is mapped from K
    V putIfAbsent(K key, V value);
    // Remove only if K is mapped to V
    boolean remove(K key, V value);
    // Replace value only if K is mapped to oldValue
    boolean replace(K key, V oldValue, V newValue);
    // Replace value only if K is mapped to some value
    V replace(K key, V newValue);
}
```
Performance Comparison

- ConcurrentHashMap vs. Collections.synchronizedMap
- See HashMapPerfTest.java
- **Note**: incrementCount() is not safe
CopyOnWriteArrayList

- Concurrent List
- Effectively immutable
  - Creates and republishes a new copy of the collection every time it is modified
CopyOnWriteArrayList Iterators

- Iterates over elements that were contained in the CopyOnWriteArray at the start of iteration
- Tolerate updates while iterating
  - No ConcurrentModificationException
CopyOnWriteArrayList Tradeoffs

• Copying the backing array can be expensive
• Most effective when iteration is far more common than modification
Queues

• Queue interface added to java.util

interface Queue<E> extends Collection<E> {
    boolean offer(E x); // try to insert, return true if insert succeeded
    E poll(); // retrieve and remove. Return null if empty
    E remove() throws NoSuchElementException; // retrieve and remove
    E peek(); // retrieve, don’t remove. Return null if empty
    E element() throws NoSuchElementException; // retrieve, don’t remove
}

• One thread-safe, non-blocking implementation
  – ConcurrentLinkedQueue
Blocking Queues

• Extends Queue to provide blocking operations
  – Retrieval: wait for queue to become nonempty
  – Insertion: wait for capacity to be available
• Can be bounded or unbounded
• Implementations provided:
  – LinkedBlockingQueue (FIFO, may be bounded)
  – PriorityBlockingQueue (priority, unbounded)
  – ArrayBlockingQueue (FIFO, bounded)
  – SynchronousQueue (rendezvous channel)
Producer/Consumer Pattern

Producer(s)                      Work Queue                      Consumers(s)

P                                →

C₁
C₂
C₃
C₄
Producer/Consumer Tips

- Bound queue size
- Make put and take operations block
  - `BlockingQueue` provides this
- Makes programs more robust to overload
  - By throttling activities that can produce more work than consumers can handle
Producer-Consumer Examples

• See:
  – ProducerConsumerPrimitive.java (wait/notify)
  – ProducerConsumerConcUtil.java (BlockingQueue)
Synchronizers

• Utilities for coordinating access and control
• CountDownLatch
  – Allows threads to wait for a set of threads to complete an action
• Future & FutureTask
  – Represents an asynchronous computation
• Semaphore
  – Dijkstra counting semaphore, managing some number of permits
• CyclicBarrier
  – Allows a set of threads to wait until they all reach a specified barrier point
• Exchanger
  – Allows two threads to rendezvous and exchange data, such as exchanging an empty buffer for a full one
CountDownLatch

- Latching variables are conditions that once set never change
- Often used to start several threads, but have them wait for a signal before continuing
- See: CountDownLatchTest.java
Future Use Case

• Client initiates asynchronous computation
• Client receives a “handle” to the result
  – The Future
• Client does other work while waiting for result
• Client requests result from Future, blocking or polling until result is available
• Client uses result
FutureTask

- A cancellable asynchronous computation
- A base implementation of Future
- Can wrap a Callable or Runnable
Future and Callable

• Callable is functional analog of Runnable

```java
interface Callable<V> {
  V call() throws Exception;
}
```

• Future represents result of asynchronous computation

```java
interface Future<V> {
  V get() throws InterruptedException, ExecutionException;
  V get(long timeout, TimeUnit unit);
  boolean cancel(boolean mayInterruptIfRunning);
  boolean isCancelled();
  boolean isDone();
}
```
Future Example

• See: FutureTaskStringReverser.java
Another Future Example

- Implementing a cache with Future

```java
public class Cache<K, V>
    Map<K, Future<V>> map = new ConcurrentHashMap();
    public V get(final K key) {
        Future<V> f = map.get(key);  // null if key not found
        if (f == null) {
            Callable<V> c = new Callable<V>() {
                public V call() {// compute value associated with key}
            };
            f = new FutureTask<V>(c);
            Future old = map.putIfAbsent(key, f); // if key not found put(key,f) & return null
            if (old == null) {                    //  otherwise return get(key)
                new Thread(f).start();
            } else { f = old;}
        }
        return f.get();
    }
```
Semaphore

- Semaphore maintain a logical set of permits
- Two operations
  - acquire() blocks until a permit is free, then takes it
  - release() adds a permit & releases one blocking acquirer
- Can implement many synchronization protocols
- See:
  - SemaphoreTunnel.java
  - SemaphoreBuffer.java
CyclicBarrier

- Allows threads to wait at a common barrier point
- Useful when a fixed-sized party of threads must occasionally wait for each other
- Cyclic Barriers can be re-used after threads released
- Can execute a Runnable once per barrier point
  - After the last thread arrives, but before any are released
  - Useful for updating shared-state before threads continue
- See:
  - CyclicBarrierEx1.java
  - CyclicBarrierEx2.java
Exchanger

• Synchronization point where two threads exchange object(s)
  – i.e., a bidirectional SynchronizedQueue
• Each thread presents some object on entry to the exchange() method, and receives the object presented by the other thread on return
• See ExchangerTest.java
Locks and Lock Support

• High-level locking interface
• Adds non-blocking lock acquisition

interface Lock {
    void lock();
    void lockInterruptibly() throws IE;
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws IE;
    void unlock();
    Condition newCondition() throws UnsupportedOperationException;
}
ReentrantLock

• Flexible, high-performance lock implementation
• Implements a reentrant mutual exclusion lock (like Java intrinsic locks) but with extra features
  – Can interrupt a thread waiting to acquire a lock
  – Can specify a timeout while waiting for a lock
  – Can poll for lock availability
  – Can have multiple wait-sets per lock via the Condition interface
• Outperforms built-in monitor locks in most cases, but slightly less convenient to use (requires finally block to release lock)
Lock Example

- Locks not automatically released
  - Must release lock in `finally` block

```java
Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // perform operations protected by lock
} catch (Exception ex) {
    // restore invariants
} finally {
    lock.unlock();
}
```
ReadWrite Locks

- **ReadWriteLock** interface defines a pair of locks;
  - one for readers; one for writers

```java
interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

- **ReentrantReadWriteLock** class
  - Multiple readers, single writer
  - Allows writer to acquire read lock
  - Allows writer to downgrade to read lock
  - Supports “fair” and “non-fair” (default) acquisition
class RWDictionaryRWL {
    private final Map<String, Data> m = new TreeMap<String, Data>();
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();
    public Data get(String key) {
        r.lock();
        try {
            return m.get(key); }
        finally { r.unlock(); }
    }
    public Data put(String key, Data value) {
        w.lock();
        try {
            return m.put(key, value); }
        finally { w.unlock(); }
    }
}
Read/Write Lock Example

• See
  – `RWDictionaryIntrinsicLock.java`
  – `RWDictionaryRentrantLock.java`
  – `RWDictionaryRWL.java`
Condition

- Condition lets you wait for a condition to hold (like `wait`), but adds several features

```java
interface Condition {
    void await() throws IE;
    boolean await(long time, TimeUnit unit) throws IE;
    long awaitNanos(long nanosTimeout) throws IE;
    void awaitUninterruptibly();
    boolean awaitUntil(Date deadline) throws IE;
    void signal();
    void signalAll();
}
```
Condition

• Many improvements over `wait()`/`notify()`
  – Multiple conditions per lock
  – Absolute and relative time-outs
  – Timed waits tell you why you returned
  – Convenient uninterruptible wait
class BoundedBufferCond {
    Lock lock = new ReentrantLock();
    Condition notFull = lock.newCondition();
    Condition notEmpty = lock.newCondition();
    Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put( Object x) throws IE {
        lock.lock();
        try {
            while (count == items.length) notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally { lock.unlock(); }
    }
}
public Object take() throws IE {
    lock.lock();
    try {
        while (count == 0) notEmpty.await();
        Object x = items[takeptr];
        if (++takeptr == items.length) takeptr = 0;
        --count;
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
Condition Example

- Previous example in BoundedBufferCond.java
- See also: BoundedBufferPrim.java
Atomic Variables

- Holder classes for scalars, references and fields
- Supports atomic operations
  - Compare-and-set (CAS)
  - Get and set and arithmetic (where applicable)
- Some main classes: \{int, long, ref\} X \{value, field, array\}
  - E.g. AtomicInteger useful for counters, sequences, statistics
- Essential for writing efficient code on MPs
  - Nonblocking data structures & optimistic algorithms
  - Reduce overhead/contention updating “hot” fields
- JVM uses best construct available on platform
  - CAS, load-linked/store-conditional, locks
Atomic Variables

• See: CounterTest.java
Executor

- Standardizes asynchronous invocation
- Separates job submission from execution policy
  - `anExecutor.execute(aRunnable)`
- Rather than
  - `new Thread(aRunnable).start()`
- Two code styles supported:
  - Actions: Runnables
  - Functions: Callables
- Also has lifecycle mgmt: e.g., cancellation, shutdown
- Executor usually created via Executors factory class
  - Configures ThreadPoolExecutor
  - Customizes shutdown methods, before/after hooks, saturation policies, queuing
ExecutorService adds lifecycle management to Executor

```java
public interface Executor {
    void execute(Runnable command);
}

public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit);
    // other convenience methods for submitting tasks
}
```
• Executors factory methods

```
public class Executors {
    static ExecutorService newSingleThreadedExecutor();
    static ExecutorService newFixedThreadPool(int n);
    static ExecutorService newCachedThreadPool(int n);
    static ScheduledExecutorService newScheduledThreadPool(int n);
    // additional versions & utility methods
}
```
Thread per message Web Server

```java
class WebServer {
    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {handleRequest(connection);}
            };
            new Thread(r).start();
        }
    }
}
```
Executor Example

- Thread pool web server - better resource management

```java
class WebServer {
    Executor pool = Executors.newFixedThreadPool(7);
    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {handleRequest(connection);}
            };
            pool.execute(r);
        }
    }
}
```
ScheduledExecutorService

- For deferred and recurring tasks, can schedule
  - Callable or Runnable to run once with a fixed delay after submission
  - Schedule a Runnable to run periodically at a fixed rate
  - Schedule a Runnable to run periodically with a fixed delay between executions
- Submission returns a ScheduledFutureTask handle which can be used to cancel the task
- Like Timer, but supports pooling and is more robust
• It's the mutable state, stupid
  – All concurrency issues boil down to coordinating access to mutable state. The less mutable state, the easier it is to ensure thread safety

• Make fields final unless they need to be mutable

• Immutable objects are automatically thread safe
  – Immutable objects simplify concurrent programming tremendously
  – They are simpler and safer, and can be shared freely without locking or defensive copying

• Encapsulation makes it practical to manage complexity.
  – Encapsulating data within objects makes it easier to preserve their invariants
  – encapsulating synchronization within objects makes it easier to comply with their synchronization policy
Guard each mutable variable with a lock
- Guard all variables in an invariant with the same lock
- Hold locks for the duration of compound actions
- A program that accesses a mutable variable from multiple threads without synchronization is broken
- Don't rely on clever reasoning about why you don't need to synchronize
- Include thread safety in the design process
- Explicitly document that your class is/isn’t not thread safe
- Document your synchronization policy