CMSC 132:
Object-Oriented Programming II

Advanced Concurrency

Department of Computer Science

University of Maryland, College Park

Concurrency without Explicity Threads

- You can write concurrent applications that don’t use explicit threads or synchronization
- Use built-in abstractions that support coordination and parallel execution

Key Concepts

- thread-safe collections
- concurrent collections
- blocking queues
- synchronizers
- thread locals
- executors

Thread Safe Collections

- Standard collections or other abstractions that are intended to be thread safe
- Generally limited to one thread operating on them at a time (watch out for sequences that need to be atomic)
- Can use Collections wrapped methods

Concurrent Collections

- Designed to allow multiple simultaneous accesses and updates
  - Blocking only when they “conflict”
- Higher space overhead
  - Not much time overhead
- Many of the concurrent collections do not allow null keys or values

Concurrent HashMap

- Allows simultaneous reads, and by default up to 16 simultaneous writers
  - Can increase the number of simultaneous writers
- Use Collections. newSetFromMap to construct concurrent set

Special Methods

- V putIfAbsent(K key,V value)
  - Store the value only if the key has no mapping
  - Return old value (null if none)
- boolean remove(K key, V oldValue)
  - Remove mapping only if it has the specified value
- boolean replace(K key, V oldValue, V newValue)
  - Update the mapping only if it has the specified value

ConcurrentSkipLists

- Skip Lists are a probabilistic alternative to balanced trees
- Invented in 1988 by Prof. Bill Pugh
- ConcurrentSkipLists provide a concurrent sorted set implementation and lots of other API improvements over TreeMaps
- Java 6 only

CopyOnWriteArrayList

- Using locking to ensure only one thread can update it at a time
- Any update copies the backing array thus, read only operations don’t need any locks
- Iteration uses a snapshot of the array
  - Allows concurrent modification and update
- Suitable only if updates rare

Important Use Case

- Keeping track of listeners to an Observable
- While iterating through list of listeners, one of them might ask to be unsubscribed
- A “concurrent update”, even though we only have one thread

Waiting for Something to Happen

- We briefly talk about join (waits for another thread to terminate)
- There are lots of ways to have a thread wait until things are right for it to do something
  - wait/notify were the way to do this before Java 5
  - But now we have new ways that are often better: blocking queues and synchronizers

Blocking Queues and Dequeues

- A Queue is a first-in, first-out queue
- A dequeue is a Double-Ended Queue
  - Allows addition and removal at both ends
  - A dequeue can also serve as a stack

What Happens When It Can't Immediately Succeed?

	throws exception	returns special value	blocks
insert	add(e)	offer(e)	put(e)
remove	remove()	poll()	take()
examine	element	peek()	

Queue Notes

- Blocking queues also offer timed offer and poll methods
- Several different implementations, each with its own advantages
  - ConcurrentLinkedQueue
    - Doesn't support blocking, but allows for simultaneous addition/deletion
  - Array/Linked Blocking Dequeue/Queue

Synchronizers

- **Other ways to wait for some condition to be true**
- **CountDownLatch**
- **Semaphore**

CountDownLatch

- A variable that can be decremented
  - Never incremented
- You can wait for it to get to zero
- You can also find out the current value
  - Most of the time, you won't need to find out the current value

Semaphore

- Contains a count of the number of permits available
- You can acquire or release permits
  - No checking that you are releasing permits you have
  - Really, just a counter
- Acquire blocks if not enough permits are available

Fairness

- Consider a Blocking queue where you atomically remove multiple elements
- What happens if one person wants to atomically remove 10 elements from a queue that can contain up to 20 elements
  - But there is a constant stream of other threads that want to remove smaller number of elements?

Starvation!

Some Abstractions Have Fair Variants

- For example, fair semaphores and fair reentrant locks
- Generally, fair guarantees first-come, first-served
- But fair almost always reduces throughput
  - Over and above implementation cost
  - Letting running threads run improves throughput

java.util.concurrent.atomic

- **From Java API:**

  A small toolkit of classes that support lock-free thread-safe programming on single variables.

  http://java.sun.com/javase/6/docs/api/

AtomicInteger

- Encapsulates an integer
- Sort of like a volatile int
- But supports additional atomic operations:
  - int getAndIncrement()
  - int decrementAndGet()
  - boolean compareAndSet(int expect, int update)

Atomic Operations

- The atomic operations are very efficient
- Most processors provide some kind of atomic compare and swap instruction
  - Needed to efficiently implement locking

Lots of Atomic Classes

- There is an AtomicX class for every primitive type, and for references
- There are also classes that let you atomically update volatile fields, and ones that encapsulate arrays and allow you to perform atomic operations on array elements

Executor

- An object that executes submitted Runnable tasks
- Rather than starting a thread for each task

```
new Thread(new(RunnableTask())).start()
```

- You ask an executor to do it

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

Executors Can Be Simple

- The execute method might just run the task
- Or create and start thread
- Or do something more complicated

java.util.concurrent.Executors

- Provides many factory and utility methods for executors
- newFixedThreadPool(int nThreads)
- newCachedThreadPool()
  - creates threads as needed, reuses them

Why Thread Pools?

- Some overhead to starting a thread
- Running 100,000 threads is a bad idea
  - Unless you have a monster machine