CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Advanced Concurrency

Department of Computer Science
University of Maryland, College Park

Excellent Reference on Concurrency

- Reference: “Java Concurrency in Practice” by Brian Goetz

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

Synchronized Collections

- Achieve thread safety by allowing access to only one thread at a time
- Examples
  - **Vector**
  - **Hashtable**
  - **Synchronized wrapper classes** created by *Collections.synchronizedXxx*
- Example: synchronized set
- http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#synchronizedSet(java.util.Set)
- Disadvantage of this approach: poor concurrency

Concurrent Collections

- Designed to allow concurrent access by multiple threads
  - Blocking only when they “conflict”
- Higher space overhead
  - Not much time overhead
- Many of the concurrent collections do not allow null keys or values
- Examples
  - **ConcurrentHashMap**
    - Replacement for synchronized hash-based Map implementations
  - **CopyOnWriteArrayList**
    - Replacement for synchronized List implementations (where traversal is the predominant operation)

Concurrent HashMap

- Allows simultaneous reads, and by default up to 16 simultaneous writers
  - Can increase the number of simultaneous writers
- **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

CopyOnWriteArrayList

- Suitable only if updates rare and iteration occurs often
- Iteration uses a snapshot of the array
- Iterators keep a reference to the backing array current at the beginning of the iteration
- When an update occurs a new array copy is created and published
- Important use case
  - Keeping track of listeners to an Observable
  - While iterating through list of listeners (delivering a notification), one of them might ask to be unsubscribed

Concurrent Skip Lists

- Skip Lists are a probabilistic alternative to balanced trees
  - Stores sorted list of items using layers of linked lists
- Invented in 1988 by Prof. Bill Pugh
- **Examples**
  - **ConcurrentSkipListMap**
    - http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html
  - **ConcurrentSkipListSet**
    - http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListSet.html
  - Above classes are concurrent replacements for a synchronized SortedMap or SortedSet (e.g., TreeMap, TreeSet wrapped with synchronizedMap)

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

- **BlockingQueue**
  - http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html
  - BlockingQueue implementations are thread-safe
  - BlockingQueue implementations designed for used in producer-consumer queues
- BlockingQueue methods can handle in different ways operations that cannot be satisfied immediately. The options are
  - Throwing an exception
  - Returning a special value (null or false)
  - Blocking the thread until the operation can succeed
    - E.g., waiting for space to become available
  - Blocking the thread for a given period of time before giving up

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

Synchronizers

- **Synchronizer**
  - Any object that coordinates control flow of threads
  - They allow threads arriving at synchronizer to pass or to wait
- **Examples**
  - Semaphores
  - Latches
  - Barriers
  - Blocking queues can act as synchronizers

Semaphore

- Controls number of activities accessing a resource or performing an action
- Contains a count of the number of permits available
- You can acquire or release permits
- acquire method - blocks if not enough permits are available
- release method – returns permit to the semaphore

CountDownLatch

- Act as a gate that is open once a set of events have taken place
- Has a counter that can be decremented (never incremented)
- countDown method - decrements counter indicating event has taken place
- await method – wait for the counter to reach zero
  - Blocks until counter reaches zero

Barrier

- Allows set of threads to wait for each other to reach a common point
- await method – blocks until all threads have reached the barrier
- Example: CyclicBarrier
  - http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/CyclicBarrier.html

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

Atomic Classes

- java.util.concurrent.atomic
  - Toolkit of classes that support lock-free thread-safe programming on single variables
- **AtomicInteger** class
  - Encapsulates an integer
  - Supports atomic operations:
    - int getAndIncrement()
    - int decrementAndGet()
    - boolean compareAndSet(int expect, int update)
- There is an AtomicX class for every primitive type
- The atomic operations are very efficient
  - Most processors provide some kind of atomic compare and swap instruction

Executor

- An object that executes submitted Runnable tasks, rather than starting a thread for each task (e.g., new Thread(new(RunnableTask())).start()))
- **You ask an executor to do it**

  Executor executor = *create executor*;
  executor.execute(new RunnableTask1());
  executor.execute(new RunnableTask2());
- **An executor can be simple or complex**
  - 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