Project 3, due Nov 14th

• parallel graph exploration from a start node
• find all reachable nodes
• perform a computation at each node
• both finding the neighbors of a node, and performing the computation, might be expensive
• want to use concurrency to accomplish this quickly
public interface Node
<N extends Node<N, V>, V> {
    V compute();
    Collection<N> neighbors();
}
public interface Explorer
    <N extends Node<N, V>, V> {

    public void explore(N start, int parallelism)
        throws InterruptedException;

    public Collection<N> reached();

    public V getValue(N n);

}
Implementing explore

- I’ll give you a sequential implementation of explore
- You have to write a concurrent one
- explore is only called once on an explorer
- can return before all computation is done
- but all computation should be queued up for execution
- if interrupted, computation should be aborted promptly
Mock Framework

- We’ll provide a mock framework
- can configure graph layout
- time required for call to neighbors() and compute()
- can mock processors
  - want to mock 600 processors?
Concurrent correctness

- We will be executing these projects on a system with 32 cores
- We will ask you to explain why certain bad things can’t happen
Efficiency matters

- There will be some test cases that check that your implementation isn’t too inefficient
- And that it also reaches moderate level of parallelism
- Some sort of non-grade prize for the most efficient solution
Simplicity

• You don’t need to create threads, synchronize, use Locks, Conditions or volatile variables to complete this project

• I encourage you not to

• We might require it, or dock you points if you do
Concurrency without threads

- You can write concurrent applications that don’t use explicit threads or synchronization
- Use built-in abstractions that support coordination and parallel execution
- A lot of this is covered in Section I-5.1 of JCIP
Java 6 alert

• Some of the features I’m going to be talking about are available only in Java 6+, not in Java 5.

• If you’ve got an Apple computer, we’re still screwed

• Leopard doesn’t provide Java 6, as many people had hoped/expected
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
ReadWriteLock

- You can use a ReadWriteLock implementation
  - e.g., ReentrantReadWriteLock
- Protect read-only operations with a read lock
  - allow simultaneous reads
- BE CAREFUL: some operations you might think are read-only are actually read/write operations
  - e.g., WeakHashMap.get()
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
ConcurrentHashMap

• 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
  • something I invented back in 1988

• 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
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?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Throws exception</th>
<th>Returns special value</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>add(e)</td>
<td>offer(e)</td>
<td>put(e)</td>
</tr>
<tr>
<td>remove</td>
<td>remove()</td>
<td>poll()</td>
<td>take()</td>
</tr>
<tr>
<td>examine</td>
<td>element</td>
<td>peek()</td>
<td></td>
</tr>
</tbody>
</table>
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
- FutureTask
- Semaphore
- Barriers
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
FutureValue\(<V>\>

- implements Runnable
- constructor takes a Callable\(<V>\>
  - like Runnable, except it returns a value of type V
- need to ask someone to run the FutureValue
- then any thread can call get() on the FutureValue
- blocks until result is available
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 the Blocking queue example from the midterm

• 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
Fair semaphores

• We could have used a fair semaphore for our blocking queue example
Barriers

- Coordinate multiple threads
- Create a barrier for 6 threads
  - once 6 threads are waiting at the barrier,
  - all 6 threads are allowed to go at once,
  - and the barrier is reset for another go
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
ThreadLocal
Avoiding contention

• All the stuff we’ve shown you so far is about managing contention
• But even better is to avoid it all together
• ThreadLocal is great for that.
Example

- You want to give each thread its own Random number generator
ThreadLocal<E>

E initialValue()
E get()
void set(E e)
void remove()
Defining a thread local

```java
static
    ThreadLocal<Random>
myRandomGenerator
    = new ThreadLocal<Random>() {

        public Random initialValue() {
            return new Random();
        }
    }
```
Using a thread local

myRandomGenerator.get().nextInt()
Executors
Executor

• An object that executes submitted Runnable tasks
• Rather than starting a thread for each task
  
  ```java
  new Thread(new(RunnableTask())).start()
  ```
• You ask an executor to do it
  
  ```java
  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
ExecutorService

- Allows you to shutdown executor (stop executing more tasks) and await termination
- Also allows you to submit a Callable and get back a Future
- Most work with a queue of jobs waiting to be run
java.util.concurrent.Executors

• provides many factory and utility methods for executors

• newFixedThreadPool(int nThreads)

• newCachedThreadPool()
  • creates threads as needed, reuses them
ThreadPoolExecutor

- implementation class for thread pool executors
- lots of dials and knobs
Why thread pools?

• Some overhead to starting a thread
  • If your task takes a second to run, the overhead to negligible

• Running 100,000 threads is a bad idea
  • unless you have a monster machine