COMP 322: Fundamentals of Parallel Programming

Lecture 20: Java Concurrent Collections

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

Announcements

- **Graded midterm exams can be picked up from Amanda Nokleby in Duncan Hall room 3137**
- **Homework 5 will be sent out by tomorrow**
  - **Homework 6 dates will be adjusted accordingly**

Acknowledgments for Today’s Lecture

- Lecture 20 handout
- “Java’s Collection Framework” slides by Rick Mercer
- “Introduction to Concurrent Programming in Java”, Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - Contributing authors: Doug Lea, Brian Goetz
- “Java Concurrency Utilities in Practice”, Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - Contributing authors: Doug Lea, Tim Peierls, Brian Goetz

Table 2: Examples of common isolated statement idioms and their equivalent AtomicInteger implementations (Corrected version)

1) Rank computation:	

`rank = new ...; rank.count = 0;`
`. . .`
`isolated r = ++rank.count;`
	`AtomicInteger rank = new AtomicInteger();`
	`. . .`
	`r = rank.incrementAndGet();`

2) Work assignment:	

`rem = new ...; rem.count = n;`
`. . .`
`isolated r = rem.count--;`
`if ( r > 0 ) . . .`
	`AtomicInteger rem = new AtomicInteger(n);`
	`. . .`
	`r = rem.getAndDecrement();`
	`if ( r > 0 ) . . .`

3) Counting semaphore:	

`sem = new ...; sem.count = 0;`
`. . .`
`isolated r = ++sem.count;`
`. . .`
`isolated r = --sem.count;`
`. . .`
`isolated s = sem.count; isZero = (s==0);`
	`AtomicInteger sem = new AtomicInteger();`
	`. . .`
	`r = sem.incrementAndGet();`
	`. . .`
	`r = sem.decrementAndGet();`
	`. . .`
	`s = sem.get(); isZero = (s==0);`

4) Sum reduction:	

`sum = new ...; sum.val = 0;`
`. . .`
`isolated sum.val += x;`
	`AtomicInteger sum = new AtomicInteger();`
	`. . .`
	`sum.addAndGet(x);`

Java Collection Framework

- The *Java Collections Framework* is a unified architecture for representing and manipulating collections. It has:
  - Interfaces: abstract data types (ADTs) representing collections of objects
  - Implementations: concrete implementations of the collection interfaces
  - Algorithms: methods that perform useful computations, such as searching and sorting

    These algorithms are said to be *polymorphic*: the same method can be used on different implementations

Java Collection interfaces

Collection

Map

Set

List

Queue

SortedMap

SortedSet

Implementations of Collection Interfaces

- A collection class
  - implements an ADT as a Java class
  - implements all methods of the interface
  - selects appropriate instance variables
  - can be instantiated
- Some well-known collection classes used in sequential Java programs
  - `List: ArrayList, LinkedList, Vector`
  - `Map: HashMap, TreeMap`
  - `Set: TreeSet, HashSet`

Working with Collections in a Parallel Program

Different approaches:

1. Restrict access to a single task ➔ no modification needed
2. Ensure that each call to a public method is "synchronized" (isolated) with respect to other calls ➔ excessive synchronization
3. Use specialized implementations that minimize serialization across public methods ➔ Java Concurrent Collections

- We will focus on three java.util.concurrent classes that can be used freely in HJ programs, analogous to Java Atomic Variables
  - ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArraySet
- Other j.u.c. classes can be used in standard Java, but not in HJ
  - ArrayBlockingQueue, CountDownLatch, CyclicBarrier, DelayQueue, Exchanger, FutureTask, LinkedBlockingQueue, Phaser PriorityBlockingQueue, Semaphore, SynchronousQueue

Java Collection interfaces

Collection
Set
List
Queue
SortedSet
Map
SortedMap

The Java Map Interface

- Map describes a type that stores a collection of *key-value* pairs
- A Map associates (maps) a key the it's value
- The keys must be unique
  - the values need not be unique
- Useful for implementing software caches (where a program stores key-value maps obtained from an external source such as a database), dictionaries, sparse arrays, …

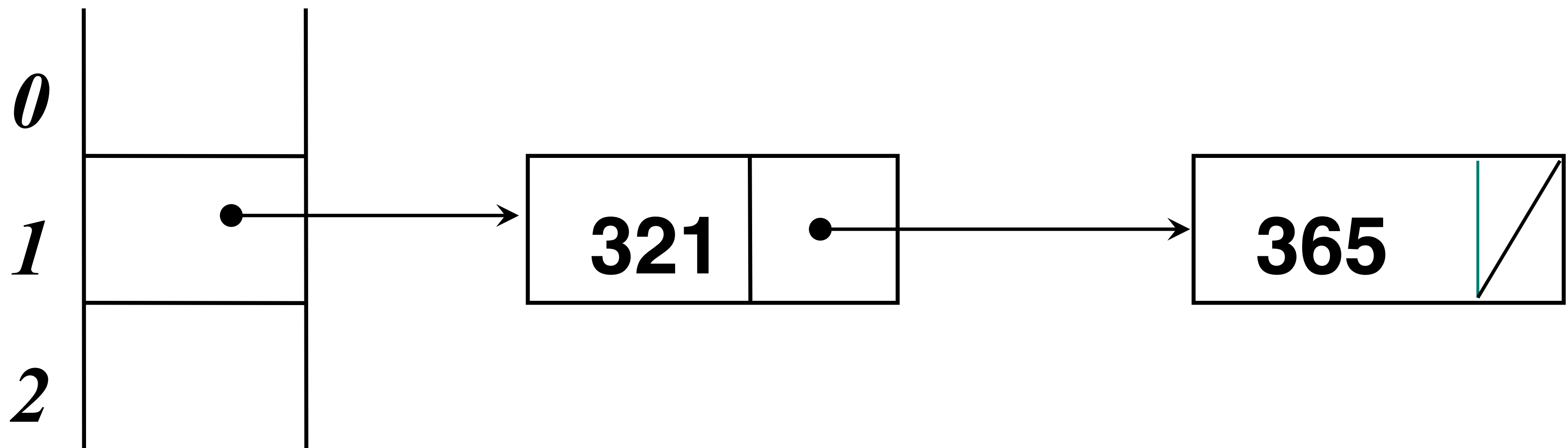
- A Map is often implemented with a hash table (HashMap)
- Hash tables attempt to provide constant-time access to objects based on a key (String or Integer)
  - key could be your Student ID, your telephone number, social security number, account number, …
- The direct access is made possible by converting the key to an array index using a *hash function* that returns values in the range 0 … ARRAY_SIZE-1, typically by using a (mod ARRAY_SIZE) operation

Collisions

- A good hash method
  - executes quickly
  - distributes keys equitably
- But you still have to handle collisions when two keys have the same hash value
  - the hash method is not guaranteed to return a unique integer for each key
    - example: simple hash method with "baab" and "abba"
- There are several ways to handle collisions
  - Consider separate chaining hashing

An Array of LinkedList Objects (to support Collisions)

An array of linked lists



java.util.concurrent.concurrentHashMap

- Implements ConcurrentMap sub-interface of Map
- Allows read (traversal) and write (update) operations to overlap with each other
- Some operations are atomic with respect to each other e.g.,
  - get(), put(), putIfAbsent(), remove()
- Aggregate operations may not be viewed atomically by other operations e.g.,
  - putAll(), clear()
- Expected degree of parallelism can be specified in ConcurrentHashMap constructor
  - ConcurrentHashMap(initialCapacity, loadFactor, concurrencyLevel)
  - A larger value of concurrencyLevel results in less serialization, but a larger space overhead for storing the ConcurrentHashMap

Concurrent Collection Performance

Throughput in Thread-safe Maps

Throughput (normalized)

3.5
3
2.5
2
5
1
0.5
0

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 24 32 40 48

Threads

- ConcurrentHashMap
- ConcurrentSkipListMap
- SynchronizedHashMap
- SynchronizedTreeMap

Java 6 B77
8-Way System
40% Read Only
60% Insert
2% Removals

Example usage of ConcurrentHashMap in org.mirrorfinder.model.BaseDirectory

```
1  public abstract class BaseDirectory extends BaseItem implements Directory {
2    Map files = new ConcurrentHashMap();
3    . . .
4    public Map getFiles() {
5      return files;
6    }
7    public boolean has(File item) {
8      return getFiles().containsValue(item);
9    }
10   public Directory add(File file) {
11     String key = file.getName();
12     if (key == null) throw new Error(. . .);
13     getFiles().put(key, file);
14     . . .
15     return this;
16   }
17   public Directory remove(File item) throws NotFoundException {
18     if (has(item)) {
19       getFiles().remove(item.getName());
20       . . .
21     } else throw new NotFoundException("can't remove unrelated item");
22   }
23 }
```

Listing 1: Example usage of ConcurrentHashMap in org.mirrorfinder.model.BaseDirectory [1]

Java Collection interfaces

Collection

Map

Set

List

Queue

SortedMap

SortedSet

java.util.concurrent.ConcurrentLinkedQueue

- `Queue` interface added to `java.util`
  - interface Queue extends Collection and includes

```
boolean offer(E x); // same as add() in Collection
E poll(); // remove head of queue if non-empty
E remove(o) throws NoSuchElementException;
E peek(); // examine head of queue without removing it
```

- Non-blocking operations
  - Return `false` when full
  - Return `null` when empty
- Fast thread-safe non-blocking implementation of Queue interface: `ConcurrentLinkedQueue`

Example usage of ConcurrentLinkedQueue in org.apache.catalina.tribes.io.BufferPool15Impl

```
1  class BufferPool15Impl implements BufferPool.BufferPoolAPI {
2    protected int maxSize;
3    protected AtomicInteger size = new AtomicInteger(0);
4    protected ConcurrentLinkedQueue queue = new ConcurrentLinkedQueue();
5    . . .
6    public XByteBuffer getBuffer(int minSize, boolean discard) {
7      XByteBuffer buffer = (XByteBuffer) queue.poll();
8      if ( buffer != null ) size.addAndGet(-buffer.getCapacity());
9      if ( buffer == null ) buffer = new XByteBuffer(minSize,discard);
10     else if ( buffer.getCapacity() <= minSize ) buffer.expand(minSize);
11     . . .
12     return buffer;
13   }
14   public void returnBuffer(XByteBuffer buffer) {
15     if ( (size.get() + buffer.getCapacity()) <= maxSize ) {
16       size.addAndGet(buffer.getCapacity());
17       queue.offer(buffer);
18     }
19   }
20 }
```

Listing 2: Example usage of ConcurrentLinkedQueue in org.apache.catalina.tribes.io.BufferPool15Impl

Java Collection interfaces

Collection

Map

Set

List

Queue

SortedMap

SortedSet

java.util.concurrent.CopyOnWriteArraySet

- Set implementation optimized for case when sets are not large, and read operations dominate update operations in frequency
- This is because update operations such as add() and remove() involve making copies of the array
  - Functional approach to mutation
- Iterators can traverse array "snapshots" efficiently without worrying about changes during the traversal.

Example usage of CopyOnWriteArraySet in org.norther.tammi.spray.freemarker.DefaultTemplateLoader

```
1  public class DefaultTemplateLoader implements TemplateLoader, Serializable
2  {
3    private Set resolvers = new CopyOnWriteArraySet();
4    public void addResolver(ResourceResolver res)
5    {
6      resolvers.add(res);
7    }
8    public boolean templateExists(String name)
9    {
10       for (Iterator i = resolvers.iterator(); i.hasNext();) {
11         if (((ResourceResolver) i.next()).resourceExists(name)) return true;
12       }
13       return false;
14   }
15   public Object findTemplateSource(String name) throws IOException
16   {
17     for (Iterator i = resolvers.iterator(); i.hasNext();) {
18       CachedResource res = ((ResourceResolver) i.next()).getResource(name);
19       if (res != null) return res;
20     }
21     return null;
22   }
23 }
```

Listing 3: Example usage of CopyOnWriteArraySet in org.norther.tammi.spray.freemarker.DefaultTemplateLoader