Lecture 20: Isolated statement (contd), Monitors, Actors

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Acknowledgments

- Wikipedia – Spanning Tree
- Wolfram Mathworld – Spanning Tree
- Inside the Java Virtual Machine, Chapter 20: Thread Synchronization
- Concurrency Tutorial: Guarded Blocks
  http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html
- “Actor-based Programming for Scalable Parallel and Distributed Systems”, Gul Agha
  http://dl.dropbox.com/u/27020702/actors/Actors.pptx
Outline

- Spanning Tree Example
- Monitors
- Actors
Spanning Tree

• A spanning tree, \( T \), of a connected undirected graph \( G \) is
  • rooted at some vertex of \( G \)
  • defined by a parent map for each vertex
  • contains all the vertices of \( G \), i.e. spans all vertices
  • contains exactly \(|v| - 1\) edges
    • adding any other edge will create a cycle
  • contains no cycles (a tree!)
    • implies the edges involved in \( T \) is a subset of the edges in \( G \)
An Example Graph with 4 possible spanning trees rooted at vertex A

Example Graph:

Spanning Trees:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>null</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>null</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>null</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>null</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
</tbody>
</table>
Parallel Spanning Tree Algorithm using isolated statement

1. class V {
2.     V [] neighbors; // adjacency list for input graph
3.     AtomicReference parent; // output value of parent in spanning tree
4.     boolean tryLabeling(V n) {
5.         isolated if (parent == null) parent=n;
6.         return parent == n;
7.     } // tryLabeling
8.     void compute() {
9.         for (int i=0; i<neighbors.length; i++) {
10.            V child = neighbors[i];
11.            if (child.tryLabeling(this))
12.                async child.compute(); // escaping async
13.        }
14.     } // compute
15. } // class V
16.
17. root.parent = root; // Use self-cycle to identify root
18. finish root.compute();
19. . . .
Parallel Spanning Tree Algorithm using object-based isolation

1. class V {
2. V [] neighbors; // adjacency list for input graph
3. AtomicReference parent; // output value of parent in spanning tree
4. boolean tryLabeling(V n) {
5.     isolated(this) if (parent == null) parent=n;
6.     return parent == n;
7. } // tryLabeling
8. void compute() {
9.     for (int i=0; i<neighbors.length; i++) {
10.        V child = neighbors[i];
11.        if (child.tryLabeling(this))
12.            async child.compute(); //escaping async
13.     }
14. } // compute
15.} // class V
16.
17.root.parent = root; // Use self-cycle to identify root
18.finish root.compute();
19.

Parallel Spanning Tree Algorithm using java.util.concurrent.atomic.AtomicReference

1. class V {
2.    V [] neighbors; // adjacency list for input graph
3.    AtomicReference parent; // output value of parent in spanning tree
4.    boolean tryLabeling(V n) {
5.        return parent.compareAndSet(null ,n);
6.    }
7.    } // tryLabeling
8.    void compute() {
9.        for (int i=0; i<neighbors.length; i++) {
10.           V child = neighbors[i];
11.           if (child.tryLabeling(this))
12.              async child.compute(); // escaping async
13.        }
14.    } // compute
15.} // class V
16.. . .
17.root.parent = root; // Use self-cycle to identify root
18. finish root.compute();
19.. . .
Performance trade-offs for Isolated, Object-based Isolated, and Atomic Variables

- Atomic variables have the best performance of all three cases
  - Limitations:
    - Body of critical section must match existing method in atomic variable’s interface
    - Context-switching needs to be disabled in the middle of an atomic operation

- Standard isolated (“isolated-all”) performs better than object-based isolated in low contention
  - HJ’s standard isolated uses a single lock. The additional parallelism from object-based isolation does not make a measurable difference if contention is low, while the additional overhead for object-based isolation can be significant.

- Standard isolated (“isolated-all”) performs better than object-based isolated in high contention on a single object
  - Object-based isolation incurs extra overhead but provides no extra benefit when contention is on a single object.

- Object-based isolation performs better than standard isolated (“isolated-all”) if critical sections are distributed across a wide range of objects, there is sufficient contention to make standard isolated perform poorly, there isn’t too much contention on a single object to limit object-based contention, and there is enough work in the isolated statement to justify the overhead
  - HJ’s object-based isolation uses one global read-write lock, combined with built-in locks for all Java objects
Outline

- Spanning Tree Example
- Monitors
- Actors
Monitors --- an object-oriented approach to isolation

- A monitor is an object containing
  - some local variables (private data)
  - some methods that operate on local data (monitor regions)
- Only one task can be active in a monitor at a time, executing some monitor region
  - Analogous to a critical section
- Monitors can also be used for
  - Mutual exclusion
  - Cooperation
Mutual Exclusion with Monitors: an Analogy

- A building, many people can enter the building at the same time
  - Entering building == entering the monitor
  - Leaving building == exiting the monitor
- Special room which can be occupied by a single person at a time
- The room contains some data which can be used/modified
- People must queue up in the hall and compete to enter the room
  - Entering room == acquiring the monitor
  - Occupying the room == owning the monitor
  - Leaving room == releasing the monitor
Monitors Cooperation - Analogy

- Cooperation == waiting for some condition to be true before executing the monitor region

- Analogy:
  - A fastidious person will only work in the room if it is clean
  - If the room is unclean, he will move from the room to a waiting area, and wait for the room to become cleaner before trying to re-enter
  - Hopefully, a cleaner will come along, gain access to the room and clean it
  - Cleaner needs to notify people waiting after room is clean

- We will revisit this concept when we study “condition variables” later in the course
Monitors – a Diagrammatic summary

Figure 20-1. A Java monitor.

Figure source: http://www.artima.com/insidejvm/ed2/images/fig20-1.gif
Converting Standard Java Libraries to Monitors

Different approaches:

1. Restrict access to a single task → no modification needed

2. Ensure that each call to a public method is isolated → excessive serialization

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, ConcurrentHashMap, CopyOnWriteArraySet

- Other j.u.c. classes can be used in standard Java, but not in HJ because they may perform blocking operations
  - ArrayBlockingQueue, CountDownLatch, CyclicBarrier, DelayQueue, Exchanger, FutureTask, LinkedBlockingQueue, Phaser, PriorityBlockingQueue, Semaphore, SynchronousQueue
The Java Map Interface

- Map describes a type that stores a collection of key-value pairs
- A Map associates a key with a 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
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

- 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

```java
public abstract class BaseDirectory extends BaseItem implements Directory {
    Map files = new ConcurrentHashMap();

    . . .
    public Map getFiles() {
        return files;
    }

    public boolean has(File item) {
        return getFiles().containsValue(item);
    }

    public Directory add(File file) {
        String key = file.getName();
        if (key == null) throw new Error( . . . );
        getFiles().put(key, file);
        . . .
        return this;
    }

    public Directory remove(File item) throws NotFoundException {
        if (has(item)) {
            getFiles().remove(item.getName());
            . . .
        } else throw new NotFoundException("can't_remove_unrelated_item");
    }
}
```

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`

```java
class BufferPool15Impl implements BufferPool.BufferPoolAPI {
    protected int maxSize;
    protected AtomicInteger size = new AtomicInteger(0);
    protected ConcurrentLinkedQueue queue = new ConcurrentLinkedQueue();

    public XByteBuffer getBuffer(int minSize, boolean discard) {
        XByteBuffer buffer = (XByteBuffer) queue.poll();
        if (buffer != null) size.addAndGet(-buffer.getCapacity());
        if (buffer == null) buffer = new XByteBuffer(minSize, discard);
        else if (buffer.getCapacity() <= minSize) buffer.expand(minSize);

        return buffer;
    }

    public void returnBuffer(XByteBuffer buffer) {
        if ((size.get() + buffer.getCapacity()) <= maxSize) {
            size.addAndGet(buffer.getCapacity());
            queue.offer(buffer);
        }
    }
}
```

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

```java
public class DefaultTemplateLoader implements TemplateLoader, Serializable {
    private Set resolvers = new CopyOnWriteArraySet();

    public void addResolver(ResourceResolver res) {
        resolvers.add(res);
    }

    public boolean templateExists(String name) {
        for (Iterator i = resolvers.iterator(); i.hasNext();)
            if (((ResourceResolver) i.next()).resourceExists(name)) return true;
        return false;
    }

    public Object findTemplateSource(String name) throws IOException {
        for (Iterator i = resolvers.iterator(); i.hasNext();)
            CachedResource res = ((ResourceResolver) i.next()).getResource(name);
            if (res != null) return res;
        return null;
    }
}
```

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

• Spanning Tree Example
• Monitors
• Actors
Actors as concurrent objects

• An actor is an autonomous, interacting component of a parallel system.

• An actor has:
  — an immutable identity (name, virtual address)
  — a mutable local state (encapsulated)
  — procedures to manipulate local state (provide an interface)
  — a thread of control
The Actor Model: Fundamentals

- An actor may:
  - process messages
  - send messages
  - change local state
  - create new actors
Communication is asynchronous: no assumption can be made about order of message delivery.
Actor anatomy

Actors = encapsulated state + behavior (methods) +
thread of control + mailbox