COMP 322: Fundamentals of Parallel Programming

Lecture 20: Isolated (contd), Monitors, Java Concurrent Collections

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

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

Worksheet #19: Insertion of isolated for correctness

The goal of IsolatedPRNG is to implement a single Pseudo Random Number Generator object that can be shared by multiple tasks. Show the isolated statement(s) that you can insert in method nextSeed() to avoid data races and guarantee proper semantics.

```
1.class IsolatedPRNG {
2. private int seed;
3. public int nextSeed() {
4.   int retVal;
5.   isolated {
6.     retVal = seed;
7.     seed = nextInt(retVal);
8.   }
9.   return retVal;
10.} // nextSeed()
11.   . . .
12.} // IsolatedPRNG
```

```
main() { // Pseudocode
  // Initial seed = 1
  IsolatedPRNG r = new IsolatedPRNG(1);
  async { print r.nextSeed(); ... }
  async { print r.nextSeed(); ... }
} // main()
```

What might happen if only line 6 and/or line7 were enclosed in separate isolated statements?

Parallel Spanning Tree Algorithm using isolated statement (Recap)

```
1. class V  {
2.   V [] neighbors; // adjacency list for input graph
3.   V parent; // output value of parent in spanning tree
4.   boolean tryLabeling(V n) {
5.     isolated if (parent == null) parent=n;
6.     return parent == n; // return true for success
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.. . .
```

Example graph (root=1, spanning tree edge shown as arrow from child to parent)

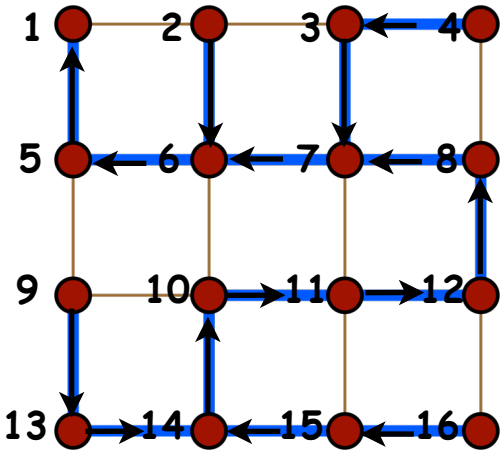


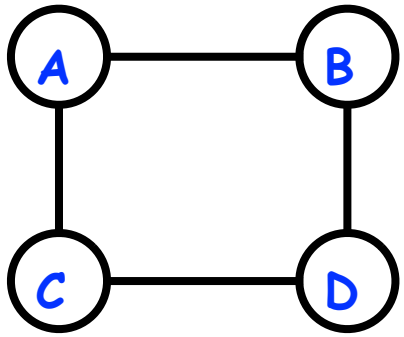
Figure source: http://en.wikipedia.org/wiki/Spanning_tree

Spanning Tree Definition

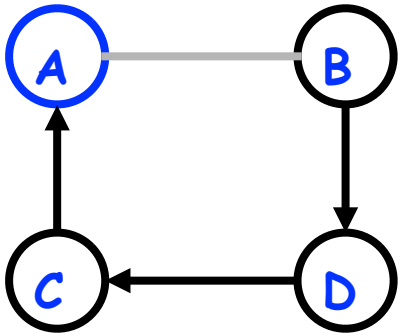
- A spanning tree, T, of a connected undirected graph G is
  - rooted at some vertex of G
  - defined by a parent map for each vertex
  - contains all the vertices of G, i.e. spans all vertices
  - contains exactly |v| - 1 edges
    - adding any other edge will create a cycle
  - contains no cycles (a 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

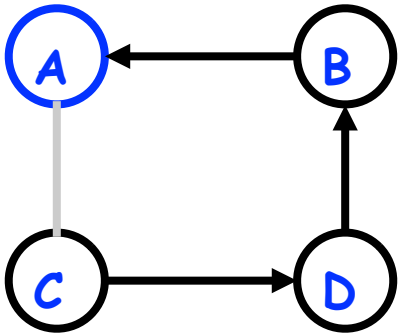
Example Graph:



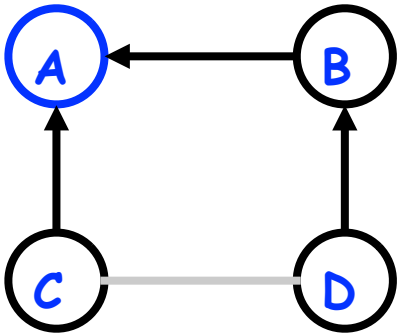
Spanning Trees:



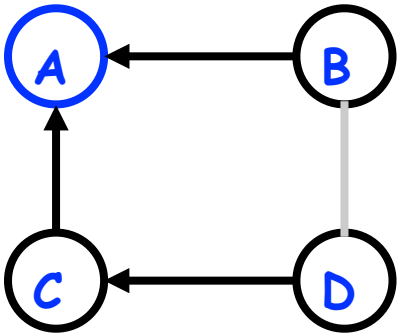
Vertex	Parent
A	null
B	D
C	A
D	C



Vertex	Parent
A	null
B	A
C	D
D	B



Vertex	Parent
A	null
B	A
C	A
D	B



Vertex	Parent
A	null
B	A
C	A
D	C

Parallel Spanning Tree Algorithm using Object-based isolation

```
1. class V  {
2.   V [] neighbors; // adjacency list for input graph
3.   V 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; // return true for success
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.. . .
```


java.util.concurrent. AtomicReference methods and their equivalent isolated statements

j.u.c.atomic Class and Constructors	j.u.c.atomic Methods	Equivalent HJ isolated statements
AtomicReference	Object o = v.**get**();	Object o; isolated (v) o = v.ref;
	v.**set**(newRef);	isolated (v) v.ref = newRef;
AtomicReference() // init = null	Object o = v.**getAndSet**(newRef);	Object o; isolated (v) { o = v.ref; v.ref = newRef; }

AtomicReference(init)	boolean b =
	v.**compareAndSet**
	(expect,update);
		boolean b;
		isolated (v)
		if (v.ref==expect) {v.ref=update; b=true;}
		else b = false;

Methods in java.util.concurrent.AtomicReference class and their equivalent HJ isolated statements. Variable v refers to an AtomicReference object in column 2 and to a standard non-atomic Java object in column 3. ref refers to a field of type Object.

AtomicReference<T> can be used to specify a type parameter.

Parallel Spanning Tree Algorithm using 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.   } // tryLabeling
7.   void compute() {
8.     for (int i=0; i<neighbors.length; i++) {
9.       V child = neighbors[i];
10.      if (child.tryLabeling(this))
11.          async child.compute(); //escaping async
12.     }
13.  } // compute
14.} // class V
15.. . .
16.root.parent = root; // Use self-cycle to identify root
17.finish root.compute();
18.. . .
```


Semantics of Exceptions and Async's within an Isolated Statement

```
1.  isolated {
2.    int t1 = p.x;
3.    p.x++;
4.    // Task execution terminates with NullPointerException
5.    // if q==null (as in non-isolated case)
6.    int t2 = q.x;
7.    q.x--;
8.    // Async creation (but not execution) is part of mutual
9.    // exclusion construct.  Async can logically be executed
10.   // after isolated statement.
11.   async { ... t1 ... t2 ... }
12.   . . .
13. } // isolated
```


Three cases of contention among isolated statements

1. Low contention: when isolated statements are executed infrequently
   - Use of global isolated statements is usually the best approach. No visible benefit from other techniques because they incur overhead that is not needed since contention is low.
2. Moderate contention (no variable is a "hot spot"): when serialization of all isolated statements limits performance, but serializing only interfering isolated statements results in good scalability
   - Atomic variables and object-based isolation usually do well in this scenario since the benefit obtained from reduced serialization outweighs any extra overhead incurred.
3. High contention (one or more variables are hot spots): when interfering isolated statements dominate the program execution time
   - Best approach in such cases is to find an alternative approach to isolated e.g., use of finish/phaser accumulators

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

Monitors – a Diagrammatic summary

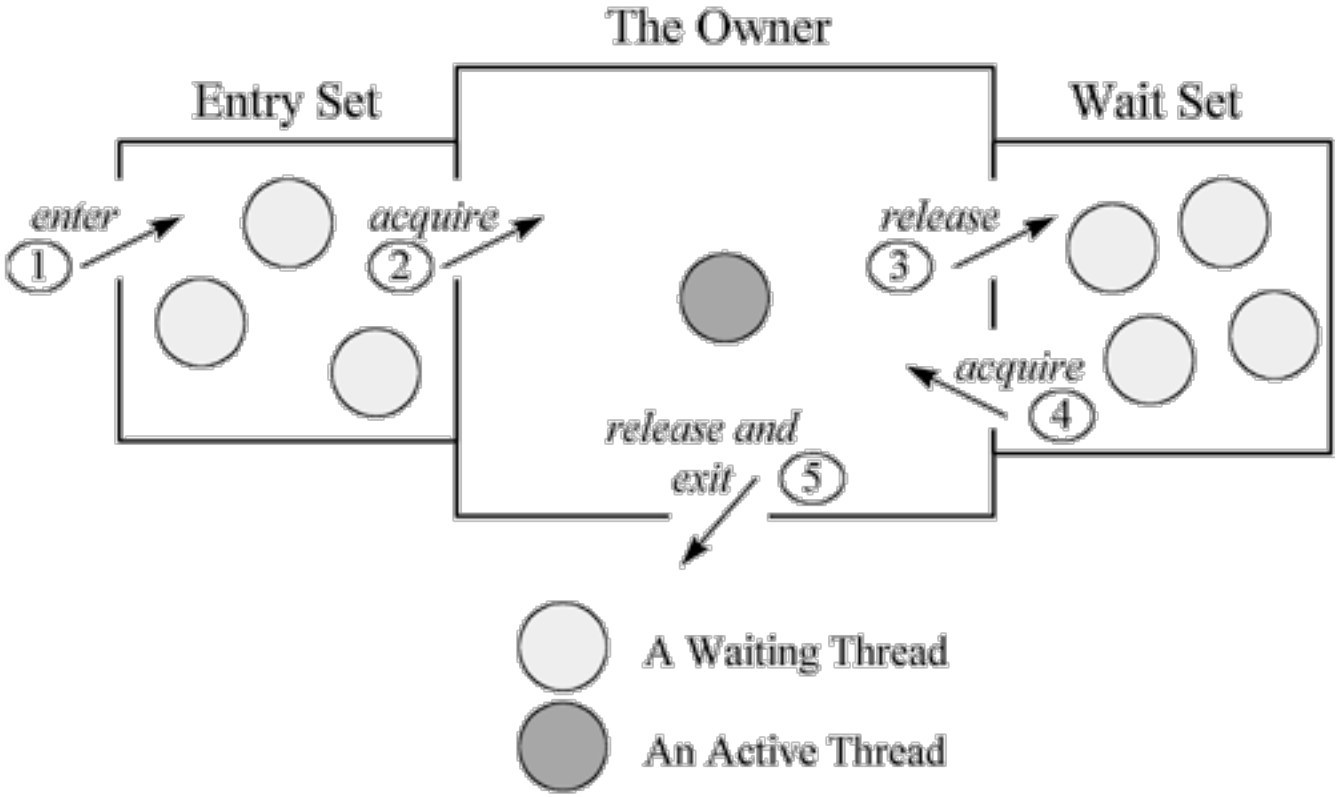


Figure 20-1. A Java monitor.

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, ConcurrentLinkedQueue, 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**

java.util.concurrent library

- Atomic variables
  - Efficient implementations of special-case patterns of isolated statements
- Concurrent Collections:
  - Queues, blocking queues, concurrent hash map, …
  - Data structures designed for concurrent environments
- Executors, Thread pools and Futures
  - Execution frameworks for asynchronous tasking
- Locks and Conditions
  - More flexible synchronization control
  - Read/write locks
- Synchronizers: Semaphore, Latch, Barrier, Exchanger, Phaser
  - Tools for thread coordination
- WARNING: only a small subset of the full java.util.concurrent library can safely be used in HJ programs
  - Atomic variables and some concurrent collections are part of the safe subset
  - We will study the full library later this semester as part of Java Concurrency

The Java Map Interface

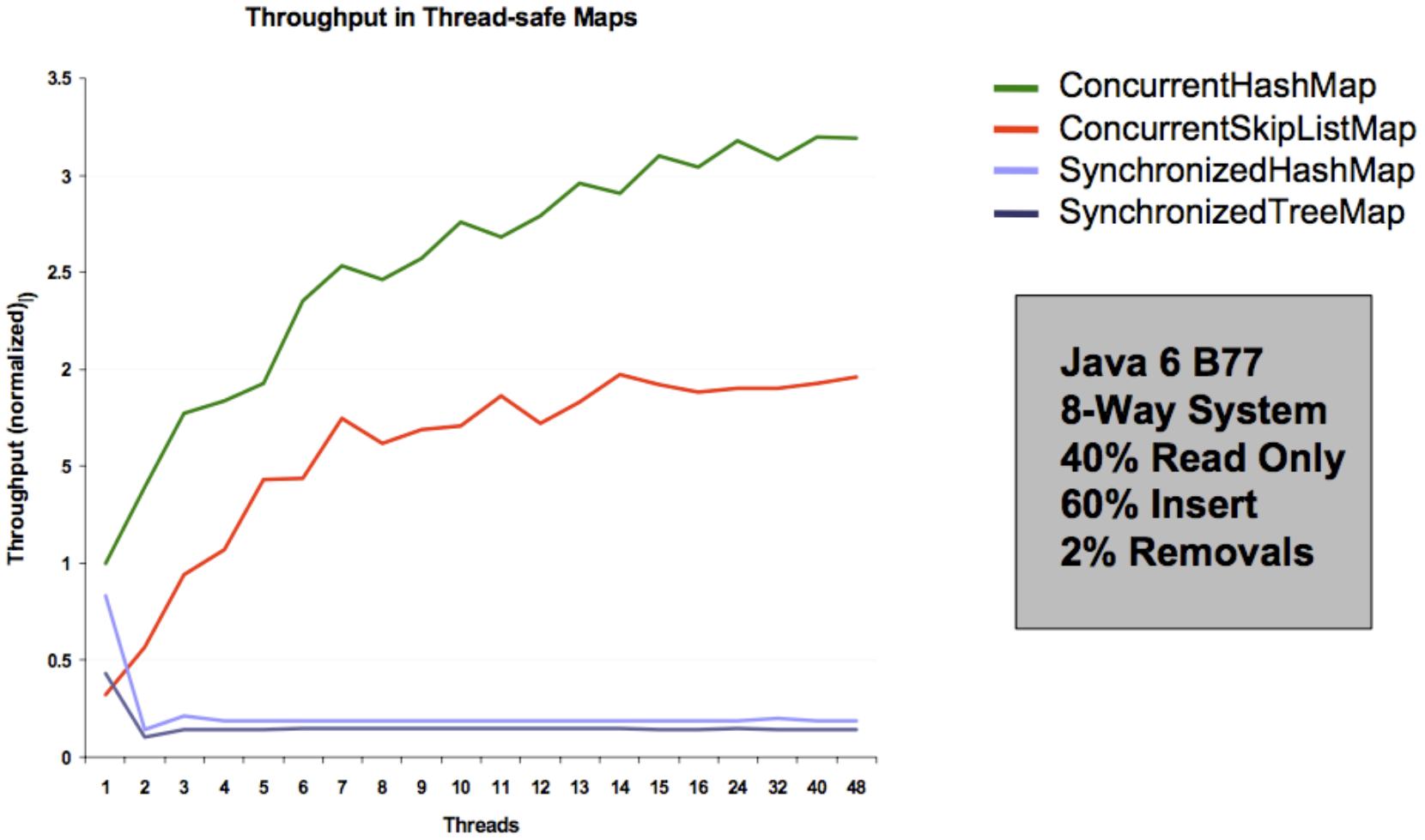
- Map describes a type that stores a collection of key-value pairs
- A Map associates a key with a value
- The keys must be unique
  - the values need not be unique
- Useful for implementing software caches (where a program stores key-value maps obtained from an external source such as a database), dictionaries,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



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.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`

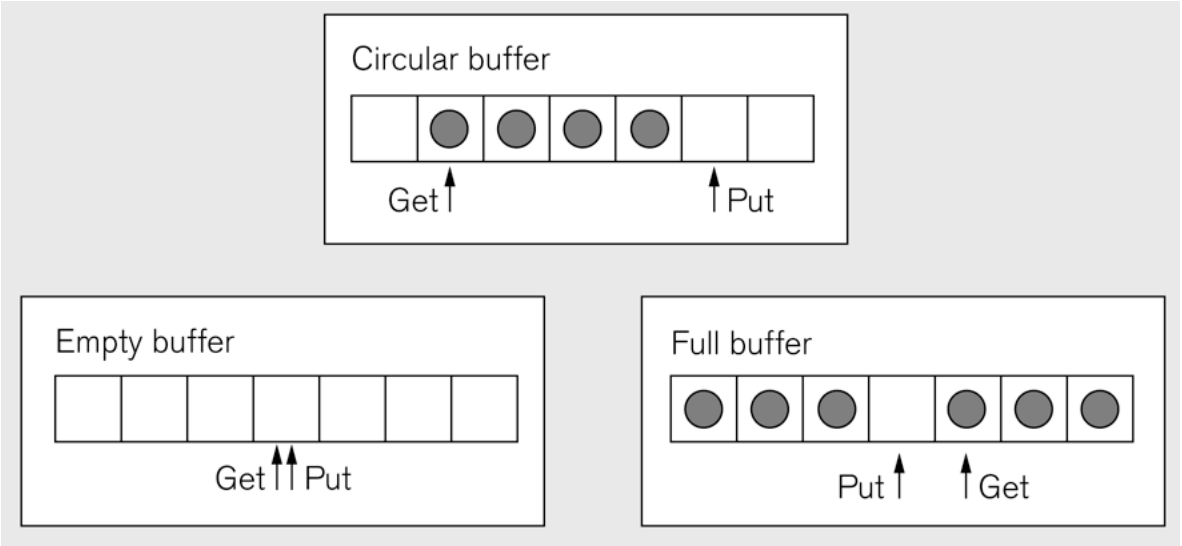
Example usage of ConcurrentLinkedQueue in org.apache.catalina.tribes.io.BufferPool15Impl

```
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

Single-Producer Single-Consumer Bounded Buffer Problem (Recap)

A bounded buffer with a single producer and a single consumer. The Put and Get cursors indicate where the producer will insert the next item and where the consumer will remove its next item.



We will revisit this problem with multiple producers and consumers later in the course

- Requires nondeterministic merge in general

Single-Producer Single-Consumer Bounded Buffer using Bounded Phaser (Recap)

```
1. finish {
2.   phaser ph = new phaser(<SIG_WAIT>, bound_size);
3.   async phased (ph<SIG>)
4.     while (…) { insert(); next; } // producer
5.   async phased (ph<WAIT>)
6.    while (…) { next; remove(); } // consumer
7. }
```

- *How would this code behave if there was no bound specified for the phaser?*

Got the idea? Let's try an example with ConcurrentLinkedQueue in Worksheet 20.

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

Worksheet #20: java.util.concurrent.ConcurrentLinkedQueue

Name 1: _____ Name 2: _____

Consider the code below that uses a ConcurrentLinkedQueue to solve the *unbounded* buffer problem with a single producer and a single consumer.

Can any async task in this version get into an infinite loop without producing or consuming an item? Explain why or why not. Also say what might happen if these two async tasks are only permitted to run on one (the same) HJ worker.

Worksheet #20 (contd)

```
1. q = new ConcurrentLinkedQueue();
2. finish {
3.   async while (true) {
4.     o = new ... ; // allocate item
5.     q.offer(o);
6.   } // producer
7.   async while (true) {
8.     o = q.poll(); // remove item
9.     if (o != null) o.process();
10.  } // consumer
11. }
```