Multiprocessor Programming Course

Dario Bahena Tapia - Ricardo Zavaleta Vazquez - Isai Barajas Cicourel

October 2015

Chapter 1

Mutual Exclusion

1.1 Filter Lock

1.1.1 Particular Case

In this experiment, the particular case we are trying to study is mutual exclusion where several threads try to use a shared resource.

We are trying to do so using waiting rooms called levels that a thread must traverse before acquiring the lock.

1.1.2 Solution

We now consider two mutual exclusion protocols that work for n threads, where n is greater than 2. The Filter lock, is a direct generalization of the Peterson lock to multiple threads.

The Filter lock, creates $n-1$ "waiting rooms", called levels, that a thread must traverse before acquiring the lock. The Peterson lock uses a two-element boolean flag array to indicate whether a thread is trying to enter the critical section. The Filter lock generalizes this notion with an n-element integer $level[]$ array, where the value of $level[A]$ indicates the highest level that thread A is trying to enter.

1.1.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to increment a common counter. All threads have to cooperate to increase this counter from 0 to 1024. Each of the threads will increase by one the counter 128 times. The expected result is that regardless of the order in which each thread executes the increment, at the end the counter must stay at 1024. If that is not the case, then it means that mutual exclusion did not work.

1.1.4 Observations and Interpretations

When executing the test on a dual core machine there was a a low chance of failure, but when executed on a Quad core and Six core machines the problems where more consistent.

```
Testcase: testParallel(mutex.FilterTest):        FAILED
expected:<1023> but was:<1024>
```

1.1.5 Proposed changes to fix the problem

Even do the victim and level arrays are *volatile*, the atomicity is not guarantee and so to fix this I added the *synchronized* reserved word to the *sameOrHigher()* method in order to guarantee atomicity while checking the levels.

```
// Is there another thread at the same or higher level?
private synchronized boolean
  sameOrHigher(int me, int myLevel) {
  for (int id = 0; id < size; id++)
    if (id != me && level[id] >= myLevel) {
      return true;
    }
    return false;
}
```

Once the change is made the execution works fine on every equipment used.

```
Script Directory: .
Using OS: Linux
Linux Java 6 Runtime: ./jre/linux/bin
Will execute test from input: mutex.FilterTest
Classpath: .:./lib/*:./output/:./build/
Running JUnit using Linux
JUnit version 4.12
.
Time: 0.021

OK (1 test)
```

1.1.6 Proposed solution

Once the comparing of the level was made atomic, the spin made for conflicting threads was properly working and able to cooperate to increase the counter to 1024.

One thing that is worth mentioning is that the *volatile* keyword seems to either have an implementation problem or not documented restriction, since it is not helping guarantee the use of the waiting rooms causing additional

increases in the counter, thus ending with a different number rather than the one expected.

1.2 Queue Lock

1.2.1 Particular Case

In this experiment, the particular case we are trying to study is mutual exclusion where several threads try to use a queue.

We are trying to do so using bounded partial queue.

1.2.2 Solution

According to the theory the *enq()* and *deq()* methods operate on opposite ends of the queue, so as long as the queue is neither full nor empty, an *enq()* call and a *deq()* call should, in principle, be able to proceed without interference, guarantee mutual exclusion for the methods.

Here, we implement a bounded queue as a linked list of entries as shown by the Entry Class.

```
/**
 * Individual queue item.
 */
protected class Entry {
  /**
   * Actual value of queue item.
   */
  public T value;
  /**
   * next item in queue
   */
  public Entry next;
  /**
   * Constructor
   * @param x Value of item.
   */
  public Entry(T x) {
    value = x;
    next = null;
  }
}
```

1.2.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to *enq()* and *deq()* a range of numbers. All threads have to cooperate to add and remove elements from the queue. Each of the threads will enqueue and dequeue values into the queue, if everything works according to the test there will be mutual exclusion and the mapping of elements will corresponds to the queue elements. If that is not the case, a duplicate fail will be raised.

1.2.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since the ReentrantLock is used to acquire an explicit lock, the mutual exclusion is guarantee by the Java java.util.concurrent.locks package.

```
/**
 * Lock out other enqueuers (dequeuers)
 */
ReentrantLock enqLock, deqLock;
```

Chapter 2

Foundations of Shared Memory

2.1 Regular Boolean MRSW Register

2.1.1 Particular Case

A register, is an object that encapsulates a value that can be observed by a *read()* method and modified by a *write()* method

For Boolean registers, the only difference between safe and regular arises when the newly written value x is the same as the old. A regular register can only return x, while a safe register may return either Boolean value.

2.1.2 Solution

A register that implements the *Register¡Boolean¿* interface is called a Boolean register (we sometimes use 1 and 0 as synonyms for true and false). This register uses the *ThreadLocal* Java java.lang package, which provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. *ThreadLocal* instances are typically private static fields in classes that wish to associate state with a thread

```
public class RegBooleanMRSWRegister
implements Register<Boolean> {
  ThreadLocal<Boolean> last;
  private boolean s_value;
  RegBooleanMRSWRegister(int capacity) {
    this.last = new ThreadLocal<Boolean>() {
      protected Boolean initialValue() { return false;
         };
    };
  }
  public void write(Boolean x) {
    if (x != last.get()) {  // if new value different
      ...
      last.set(x);          // remember new value
      s_value =x;           // update register
    }
  }
  public Boolean read() {
    return s_value;
```

```
    }
}
```

2.1.3 Experiment Description

The test creates 8 threads that reads the value of a register. All threads have to be able to read the current value of the register despite the previous value is contrary to the current one. The expected result is must be a 1 or true value.

2.1.4 Observations and Interpretations

The tests executed as expected and no errors where found.

2.2 Sequential Snapshot

2.2.1 Particular Case

In order to read multiple register values atomically we use an atomic snapshot. An atomic snapshot constructs an instantaneous view of an array of atomic registers. We construct a wait-free snapshot, meaning that a thread can take an instantaneous snapshot of memory without delaying any other thread.

2.2.2 Solution

Each thread repeatedly calls *collect()*, and returns as soon as it detects a clean double collect (one in which both sets of timestamps were identical). This construction always returns correct values. The *update()* calls are wait-free, but *scan()* is not because any call can be repeatedly interrupted by *update()*, and may run forever without completing. It is however obstruction-free, since a *scan()* completes if it runs by itself for long enough.

```
public class SeqSnapshot<T> implements Snapshot<T> {
  private T[] a_value;
  public SeqSnapshot(int capacity, T init) {
    a_value = (T[])new Object[capacity];
    for (int i = 0; i < a_value.length; i++) {
      a_value[i] = init;
    }
  }
  public synchronized void update(T v) {
    a_value[ThreadID.get()] = v;
  }
  public synchronized T[] scan() {
    T[] result = (T[]) new Object[a_value.length];
    for (int i = 0; i < a_value.length; i++)
      result[i] = a_value[i];
    return result;
  }
}
```

2.2.3 Experiment Description

The test creates 2 threads that write the register twice with the the values *FIRST* and *SECOND* and later parallel reads the values.

2.2.4 Observations and Interpretations

When executing the test the test fails on the parallel part due to a index out of bound exception.

```
Testcase: testSequential(register.SeqSnapshotTest):
        Caused an ERROR
2
java.lang.ArrayIndexOutOfBoundsException: 2
        at register.SeqSnapshot.update(SeqSnapshot.java
           :26)
        at register.SeqSnapshotTest.testSequential(
           SeqSnapshotTest.java:40)
```

2.2.5 Proposed changes to fix the problem

By resetting the thread id at the beginning of the test the id is guarantee to be the experted while doing the parallel test.

```
  class MyThread extends Thread {
    public void run() {
      ThreadID.reset();
      instance.update(FIRST);
      instance.update(SECOND);
      Object[] a = instance.scan();
      for (Object x : a) {
        Integer i = (Integer) x;
        int me = ThreadID.get();
        for (int j = 0; j < THREADS; j++) {
          results[me][j] = (Integer)a[j];
        }
      }
    }
  }
```

Once the change is made the execution works fine on every equipment used.

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 0.097 sec

------------- Standard Output ---------------
sequential
parallel
---------------- ---------------- ----------------

test-single:
BUILD SUCCESSFUL (total time: 0 seconds)
```

2.2.6 Proposed solution

Once the ThreadID is reset before changing from the sequential to the parallel test the execution never fails, since the out of bound happens due to the id being incremental.

Chapter 3

Spin Locks and Contention

3.1 Array Lock

3.1.1 Particular Case

The ALock, is a simple array-based queue lock. In a queue, each thread can learn if its turn has arrived by checking whether its predecessor has finished.

3.1.2 Solution

The threads share an *AtomicInteger* tail field, initially zero. To acquire the lock, each thread atomically increments tail. Call the resulting value the threads slot. The slot is used as an index into a Boolean flag array. If *flag[j]* is true, then the thread with slot j has permission to acquire the lock.

```
public class ALock implements Lock {
  // thread-local variable
  ThreadLocal<Integer> mySlotIndex = new ThreadLocal<
     Integer> (){
    protected Integer initialValue() {
      return 0;
    }
  };
  volatile AtomicInteger tail;
  volatile boolean[] flag;
  int size;
  /**
   * Constructor
   * @param capacity max number of array slots
   */
  public ALock(int capacity) {
    size = capacity;
    tail = new AtomicInteger(0);
    flag = new boolean[capacity];
    flag[0] = true;
  }
  public void lock() {
    int slot = tail.getAndIncrement() % size;
    mySlotIndex.set(slot);
    while (! flag[mySlotIndex.get()]) {}; // spin
```

```
  }
  public void unlock() {
    flag[mySlotIndex.get()] = false;
    flag[(mySlotIndex.get() + 1) % size] = true;
  }
  // any class implementing Lock must provide these
     methods
  public Condition newCondition() {
    throw new java.lang.UnsupportedOperationException()
       ;
  }
  public boolean tryLock(long time,
      TimeUnit unit)
      throws InterruptedException {
    throw new java.lang.UnsupportedOperationException()
       ;
  }
  public boolean tryLock() {
    throw new java.lang.UnsupportedOperationException()
       ;
  }
  public void lockInterruptibly() throws
     InterruptedException {
    throw new java.lang.UnsupportedOperationException()
       ;
  }
}
```

3.1.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to increment a common counter. All threads have to cooperate to increase this counter from 0 to 1024. Each of the threads will increase by one the counter 128 times. The expected result is that regardless of the order in which each thread executes the increment, at the end the counter must stay at 1024. If that is not the case, then it means that mutual exclusion did not work.

3.1.4 Observations and Interpretations

When executing the test values where lost from the queue.

```
1009
1010
1011
1013
1014
1015
```

3.1.5 Proposed changes to fix the problem

By adding the volatile keyword in the ALock class the execution was fixed.

```
  volatile AtomicInteger tail;
  volatile boolean[] flag;
  int size;
```

Once the change is made the execution works fine on every equipment used.

3.1.6 Proposed solution

Apparently the AtomicInteger keyword is not guarantee that the values are propagated through the memory hierarchy and by making the value atomic the execution works.

3.2 Composite Fast Path Lock

3.2.1 Particular Case

In this experiment, we extend the *CompositeLock* algorithm to encompass a fast path in which a solitary thread acquires an idle lock without acquiring a node and splicing it into the queue.

3.2.2 Solution

We use a *FASTPATH* flag to indicate that a thread has acquired the lock through the fast path. Because we need to manipulate this flag together with the tail fields reference, we take a high-order bit from the tail fields integer stamp. The private *fastPathLock()* method checks whether the tail fields stamp has a clear *FASTPATH* flag and a null reference. If so, it tries to acquire the lock simply by applying *compareAndSet()* to set the FASTPATH flag to true, ensuring that the reference remains null. An uncontended lock acquisition thus requires a single atomic operation. The *fastPathLock()* method returns true if it succeeds, and false otherwise.

3.2.3 Experiment Description

The test creates 2 threads that need to be coordinate in order to increase a counter. All threads have to cooperate in order to increase the counter into the desire value, this lock implements a explicit lock that the test calls. If that is not the case, an assertion fail will be raised.

3.2.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since the *CompositeFastPathLock* is used to acquire an explicit lock, the mutual exclusion is guarantee.

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 0.895 sec

test-single:
BUILD SUCCESSFUL (total time: 3 seconds)
```

3.3 Time Out Lock

3.3.1 Particular Case

In this experiment we consider the Java Lock interface which includes a *tryLock()* method that allows the caller to specify a *timeout* or maximum duration the caller is willing to wait to acquire the lock. The TOLock implements this mechanism in order to wait for a resource.

3.3.2 Solution

The *tryLock()* method creates a new QNode with a null pred field and appends it to the list as in the CLHLock class. If the lock was free, the thread enters the critical section. Otherwise, it spins waiting for its predecessors QNodes pred field to change. If the predecessor thread times out, it sets the pred field to its own predecessor, and the thread spins instead on the new predecessor.

```
while (System.nanoTime() - startTime < patience) {
  QNode predPred = pred.pred;
  if (predPred == AVAILABLE) {
    return true;
  } else if (predPred != null) {  // skip
     predecessors
    pred = predPred;
  }
}
// timed out; reclaim or abandon own node
if (!tail.compareAndSet(qnode, pred))
  qnode.pred = pred;
```

3.3.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to increase a counter. All threads have to cooperate in order to increase the counter into the desire value, this lock implements a explicit lock that the test calls, but with the addition of a time out. If that is not the case, an assertion fail will be raised.

3.3.4 Proposed changes to improve execution

By making volatile the QNode's in the TOLock class the execution finishes faster.

```
  volatile static QNode AVAILABLE = new QNode();
  volatile AtomicReference<QNode> tail;
  ThreadLocal<QNode> myNode;
```

3.3.5 Observations and Interpretations

The tests executed as expected and no errors where found. But the time out generates some waiting depending on the number of cores the machine has.

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 22.551 sec

------------- Standard Output ---------------
locking
Created Threads
Started Threads
Finished Threads
timeout
timeouts: 8191
---------- ---------------- ----------------

test-single:
BUILD SUCCESSFUL (total time: 24 seconds)
```

Chapter 4

Monitors and Blocking Synchronization

4.1 Non-Reentrant Mutual Exclusion

4.1.1 Particular Case

In this experiment we use a re-entrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.

4.1.2 Solution

The Mutex lock extends a *AbstractQueuedSynchronizer*, we have a *tryAcquire()* that only acquire the lock when the state is FREE, an *tryRelease()* that releases the lock by setting the state to FREE and a method to verify the current state of the lock. Due to the AbstractQueued we use a framework for implementing blocking locks and related synchronizers (semaphores, events, etc) that rely on first-in-first-out (FIFO) wait queues. This extended class is designed to be a useful basis for most kinds of synchronizers that rely on a single atomic int value to represent state, thus the object does all the work and we forward the conditions.

```
  private final LockSynch sync = new LockSynch();

  public void lock() {
    sync.acquire(0);
  }
  public boolean tryLock() {
    return sync.tryAcquire(0);
  }
  public void unlock() {
    sync.release(0);
  }
  public Condition newCondition() {
    return sync.newCondition();
  }
  public boolean isLocked() {
    return sync.isHeldExclusively();
  }
  public boolean hasQueuedThreads() {
    return sync.hasQueuedThreads();
```

```
  }
  public void lockInterruptibly() throws
     InterruptedException {
    sync.acquireInterruptibly(0);
  }
  public boolean tryLock(long timeout, TimeUnit unit)
     throws InterruptedException {
    return sync.tryAcquireNanos(1, unit.toNanos(timeout
       ));
  }
```

4.1.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to increase a counter. All threads have to cooperate in order to increase the counter into the desire value, this lock implements a explicit lock that the test calls, but with the addition of a re-entrant lock sync, which in addition to implementing the Lock interface, this class defines methods *isLocked()* and *hasQueueThreads()*, as well as some associated protected access methods that may be used for instrumentation and monitoring. If that is not the case, an assertion fail will be raised.

4.1.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since we extend the AbstractQueuedSynchronizer, we are mainly using the Java java.util.concurrent.locks package.

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 0.382 sec

------------- Standard Output ---------------
parallel
---------- ---------------- ----------------

test-single:
BUILD SUCCESSFUL (total time: 1 second)
```

Chapter 5

Linked Lists: The Role of Locking

5.1 Fine-Grained Synchronization List

5.1.1 Particular Case

The Fine-Grained synchronization improves concurrency by locking individual nodes, rather than locking the list as a whole. Instead of placing a lock on the entire list, we add a Lock to each node, along with *lock()* and *unlock()* methods.

5.1.2 Solution

Just as in the coarse-grained list, *remove()* makes currA unreachable by setting predA next field to currA successor. To be safe, *remove()* must lock both predA and currA. To guarantee progress, it is important that all methods acquire locks in the same order, starting at the head and following next references toward the tail. The *FineList* class *add()* method uses hand-over-hand locking to traverse the list. The finally blocks release locks before returning.

```
public boolean add(T item) {
  int key = item.hashCode();
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        return false;
      }
      Node newNode = new Node(item);
      newNode.next = curr;
      pred.next = newNode;
```

```
        return true;
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
```

```
  public boolean remove(T item) {
    Node pred = null, curr = null;
    int key = item.hashCode();
    head.lock();
    try {
      pred = head;
      curr = pred.next;
      curr.lock();
      try {
        while (curr.key < key) {
          pred.unlock();
          pred = curr;
          curr = curr.next;
          curr.lock();
        }
        if (curr.key == key) {
          pred.next = curr.next;
          return true;
        }
        return false;
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
```

5.1.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to add and remove values from a list. All threads have to cooperate in order to parallel add and remove values from the list while avoiding duplicate remove of the values. If that is not the case, a fail will be raised.

5.1.4 Observations and Interpretations

The tests executed as expected and no errors where found.

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 0,183 sec

------------- Standard Output ---------------
sequential add, contains, and remove
parallel add
parallel remove
parallel both
---------- ---------------- ----------------

test-single:
BUILD SUCCESSFUL (total time: 2 seconds)
```

5.2 Optimistic Synchronization

5.2.1 Particular Case

In this experiment, the particular case we are trying to study an Optimistic Synchronization which to reduce synchronization costs we take a chance by, search without acquiring locks, lock the nodes found, and then confirm that the locked nodes are correct.

5.2.2 Solution

The *OptimisticList add()* method traverses the list ignoring locks, acquires locks, and validates before adding the new node, we must validate and guarantee freedom from interference by checking that predA points to currA and is reachable from head.

```
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    Entry pred = this.head;
    Entry curr = pred.next;
    while (curr.key <= key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) { // present
          return false;
        } else {               // not present
          Entry entry = new Entry(item);
          entry.next = curr;
          pred.next = entry;
          return true;
        }
      }
    } finally {                // always unlock
      pred.unlock(); curr.unlock();
    }
```

```
    }
  }
```

The *remove()* method traverses ignoring locks, acquires locks, and validates before removing the node.

```
  public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
      Entry pred = this.head;
      Entry curr = pred.next;
      while (curr.key < key) {
        pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
          if (validate(pred, curr)) {
            if (curr.key == key) { // present in list
              pred.next = curr.next;
              return true;
            } else {               // not present in
               list
              return false;
            }
          }
      } finally {                  // always unlock
        pred.unlock(); curr.unlock();
      }
    }
  }
```

5.2.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to add and remove values from a list. All threads have to cooperate in order to parallel add and remove values from the list while avoiding duplicate remove of the values. If that is not the case, a fail will be raised.

5.2.4 Observations and Interpretations

When executing the test duplicate removes where found.

```
junit.framework.AssertionFailedError: RemoveThread:
   duplicate remove:
```

5.2.5 Proposed changes to fix the problem

The test approach is naive, since it assumes that data would exist always prior deletion, thus not handling the case when there is no data to remove. By doing a retry until there is data to remove the problem seems to be fix.

```
  public boolean remove(T item) {
    int key = item.hashCode();
    boolean bRetry = true;
    while (true) {
      Entry pred = this.head;
      Entry curr = pred.next;
      while (curr.key < key) {
        pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
        while (bRetry) {
          if (validate(pred, curr)) {
            bRetry = false;
            if (curr.key == key) { // present in list
              pred.next = curr.next;
              return true;
            } else {               // not present in
               list
              return false;
            }
          }
        }
      } finally {                  // always unlock
        pred.unlock(); curr.unlock();
      }
```

```
    }
  }
}
```

Once the change is made the execution works fine on every equipment used.

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 0,071 sec

------------- Standard Output ---------------
sequential add, contains, and remove
parallel add
parallel remove
parallel both
---------------- ---------------- ----------------

test-single:
BUILD SUCCESSFUL (total time: 3 seconds)
```

5.2.6 Proposed solution

This error may imply a bad scheduling of the list. Apparently old JVM depending on the OS, executed using a logic that avoided this from happening by working in a round robin fashion.

Chapter 6

Concurrent Queues and the ABA Problem

6.1 HW Queue

6.1.1 Particular Case

The HW Queue uses the Atomic values defined by the Java language in order to implement a locking free queue.

6.1.2 Solution

The unbounded queue implementation used by the *HWQueue* class is blocking, meaning that the *deq()* method does not return until it has found an item to dequeue.

```
public T deq() {
  while (true) {
    int range = tail.get();
    for (int i = 0; i < range; i++) {
      T value = items[i].getAndSet(null);
      if (value != null) {
        return value;
      }
    }
  }
}
```

The *enq()* method uses the *AtomicInteger getAndIncrement()* method that atomically incremented the counter.

```
public void enq(T x) {
  int i = tail.getAndIncrement();
  items[i].set(x);
}
```

6.1.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to add and remove values from a list. All threads have to cooperate in order to parallel add and remove values from the list while avoiding duplicate remove of the values. If that is not the case, a fail will be raised.

6.1.4 Observations and Interpretations

The tests executed as expected and no errors where found.

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 0,121 sec

------------- Standard Output ---------------
sequential push and pop
parallel enq
parallel deq
parallel both
---------------- --------------- ---------------

test-single:
BUILD SUCCESSFUL (total time: 1 second)
```

6.2 Bounded Partial Queue

6.2.1 Particular Case

In this experiment we implement a bounded queue as a linked list and using the Java *ReentrantLock* implementation for the enqueue-er and dequeue-er explicit call of the lock.

6.2.2 Solution

According to the theory there is a limited concurrency we can expect a bounded queue implementation with multiple concurrent enqueue-ers and dequeue-ers to provide, but can extend the enq() and deq() methods to operate on opposite ends of the queue, so as long as the queue is neither full nor empty, an enq() call and a deq() call should, in principle, be able to proceed without interference.

The *deq()* method reads the size field to check whether the queue is empty. If so, the dequeue-er must wait until an item is enqueued. The dequeue-er waits on *notEmptyCondition*, which temporarily releases *deqLock*, and blocks until the condition is signaled.

```
public T deq() {
  T result;
  boolean mustWakeEnqueuers = true;
  deqLock.lock();
  try {
    while (size.get() == capacity) {
      try {
        notEmptyCondition.await();
      } catch (InterruptedException ex) {}
    }
    result = head.next.value;
    head = head.next;
    if (size.getAndIncrement() == 0) {
      mustWakeEnqueuers = true;
    }
  } finally {
    deqLock.unlock();
  }
```

```
    if (mustWakeEnqueuers) {
      enqLock.lock();
      try {
        notFullCondition.signalAll();
      } finally {
        enqLock.unlock();
      }
    }
    return result;
  }
```

The *enq()* method thread acquires the *enqLock*, and reads the size field. While that field is equal to the capacity, the queue is full, and the enqueue-er must wait until a dequeue-er makes room. The enqueue-er waits on the *notFullCondition* field, releasing the enqueue lock temporarily, and blocking until that condition is signalled.

```
  public void enq(T x) {
    if (x == null) throw new NullPointerException();
    boolean mustWakeDequeuers = false;
    enqLock.lock();
    try {
      while (size.get() == 0) {
        try {
          notFullCondition.await();
        } catch (InterruptedException e) {}
      }
      Entry e = new Entry(x);
      tail.next = e;
      tail = e;
      if (size.getAndDecrement() == capacity) {
        mustWakeDequeuers = true;
      }
    } finally {
      enqLock.unlock();
    }
    if (mustWakeDequeuers) {
      deqLock.lock();
      try {
```

```
        notEmptyCondition.signalAll();
      } finally {
        deqLock.unlock();
      }
    }
  }
```

6.2.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to *enq()* and *deq()* a range of numbers. All threads have to cooperate to add and remove elements from the queue. Each of the threads will enqueue and dequeue values into the queue, if everything works according to the test there will be mutual exclusion and the mapping of elements will corresponds to the queue elements. If that is not the case, a duplicate fail will be raised.

6.2.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since the ReentrantLock is used to acquire an explicit lock, the mutual exclusion is guarantee by the Java java.util.concurrent.locks package.

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 0,076 sec

------------- Standard Output ---------------
sequential push and pop
parallel enq
parallel deq
parallel both
------------- ---------------- ---------------

test-single:
BUILD SUCCESSFUL (total time: 1 second)
```

6.3 Unbounded Queue

6.3.1 Particular Case

In this experiment we implement a unbounded queue, the representation is the same as the bounded queue, except there is no need to count the number of items in the queue, or to provide conditions on which to wait.

6.3.2 Solution

According to the theory an item is actually enqueued when the *enq()* call sets the last nodes next field to the new node, even before *enq()* resets tail to refer to the new node. After that instant, the new item is reachable along a chain of the next references. The actual head is the successor of the node referenced by head, and the actual tail is the last item reachable from the head. Both the *enq()* and *deq()* methods are total as they do not wait for the queue to become empty or full.

```
public T deq() throws EmptyException {
  T result;
  deqLock.lock();
  try {
    if (head.next == null) {
      throw new EmptyException();
    }
    result = head.next.value;
    head = head.next;
  } finally {
    deqLock.unlock();
  }
  return result;
}
```

```
public void enq(T x) {
  if (x == null) throw new NullPointerException();
  enqLock.lock();
  try {
    Node e = new Node(x);
    tail.next = e;
```

```
      tail = e;
    } finally {
      enqLock.unlock();
    }
  }
```

This queue cannot deadlock, because each method acquires only one lock, either *enqLock* or *deqLock* and is much simpler than the bounded queue.

6.3.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to *enq()* and *deq()* a range of numbers. All threads have to cooperate to add and remove elements from the queue. Each of the threads will enqueue and dequeue values into the queue, if everything works according to the test there will be mutual exclusion and the mapping of elements will corresponds to the queue elements. If that is not the case, a duplicate fail will be raised.

6.3.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since the *ReentrantLock* is used to acquire an explicit lock, the mutual exclusion is guarantee by the Java java.util.concurrent.locks package. As a interesting note, once each 100 executions the running time increases exponentially as shown below.

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 10,102 sec

------------- Standard Output ---------------
sequential push and pop
parallel enq
parallel deq
parallel both
------------- ---------------- ---------------

test-single:
BUILD SUCCESSFUL (total time: 11 seconds)
```

Chapter 7

Counting, Sorting, and Distributed Coordination

7.1 Bitonic Counting Network

7.1.1 Particular Case

In this experiment we implement a *Bitonic* counting network, a *Bitonic* network is a comparison-based sorting network that can be run in parallel. It focuses on converting a random sequence of numbers into a bitonic sequence, one that monotonically increases, then decreases. We represent this network as switches in a counting network, but as we use shared-memory multiprocessor however, we represent this network as objects in memory mimicking a real network.

7.1.2 Solution

According to the theory each balancer is an object, whose wires are references from one balancer to another. Each thread repeatedly traverses the object, starting on some input wire and emerging at some output wire, effectively shepherding a token through the network. The *Balancer* class has a single Boolean field: toggle. The synchronized *traverse()* method complements the toggle field and returns as output wire, either 0 or 1.

```
public synchronized int traverse(int input) {
  try {
    if (toggle) {
      return 0;
    } else {
      return 1;
    }
  } finally {
    toggle = !toggle;
  }
}
```

The *Merger* class has three fields; the *width* field must be a power of 2, *half[]* is a two-element array of half-width *Merger* objects, and *layer[]* is an array of width/2 balancers implementing the final network layer. The class provides a *traverse(i)* method, where i is the wire on which the token enters. If the token entered on the lower width/2 wires, then it passes through *half[0]*, otherwise *half[1]*.

```
public Merger(int _size) {
  size = _size;
  layer = new Balancer[size / 2];
  for (int i = 0; i < size / 2; i++) {
    layer[i] = new Balancer();
  }
  if (size > 2) {
    half = new Merger[]{new Merger(size/2), new
       Merger(size/2)};
  }
}

public int traverse(int input) {
  int output = 0;
  if (size > 2) {
    output = half[input % 2].traverse(input / 2);
  }
  return output + layer[output].traverse(0);
}
```

The *Bitonic* class provides a *traverse(i)* method. If the token entered on the lower width/2 wires, then it passes through *half[0]*, otherwise *half[1]*. A token that emerges from the half-merger subnetwork on wire i then traverses the final merger network from input wire i.

7.1.3 Experiment Description

The test creates a 256 array that get values mapped to it and later checks that the steps where properly executes, by validating the result in the *position-1* of each elements the bitonic sorted should have accommodated the array into a proper sequence. If the result doesn't match the mapping a new exception is thrown.

7.1.4 Observations and Interpretations

The tests executed as expected and no errors where found. The bitonic counting network as represented through software is replicating hardware

that is well understood and easy to follow.

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
    elapsed: 0,052 sec

test:
Deleting: /var/folders/hn/
    vlmcc1gj1kl7bb9rpqb8tdqr0000gn/T/TEST-counting.
    BitonicTest.xml
BUILD SUCCESSFUL (total time: 0 seconds)
```

Chapter 8

Concurrent Hashing and Natural Parallelism

8.1 Cuckoo Hashing

8.1.1 Particular Case

Cuckoo hashing is a (sequential) hashing algorithm in which a newly added item displaces any earlier item occupying the same slot. In this test we modify the sequential Cuckoo hashing in order to change the sequential hashing algorithm to concurrent hashing.

8.1.2 Solution

We break up each method call into a sequence of phases, where each phase adds, removes, or displaces a single item x. We use a two-entry array *table[]* of tables, and two independent hash functions, (denoted as *hash0()* and *hash1()* in the code) mapping the set of possible keys to entries in the array.

```
public TCuckooHashSet(int capacity) {
  locks = new Lock[2][LOCKS];
  table = (T[][]) new Object[2][capacity];
  size = capacity;
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < LOCKS; j++) {
      locks[i][j] = new ReentrantLock();
    }
  }
}
private final int hash0(Object x) {
  return Math.abs(x.hashCode() % size);
}
private final int hash1(Object x) {
  random.setSeed(x.hashCode());
  return random.nextInt(size);
}
```

To test whether a value x is in the set, *contains(x)* tests whether either table[0][h0(x)] or *table*[1][$h1(x)$] is equal to x. Similarly, *remove(x)* checks whether x is in either *table*[0][$h0(x)$] or *table*[1][$h1(x)$], and removes it if found.

```
public boolean remove(T x) {
```

```
    if (x == null) {
      throw new IllegalArgumentException();
    }
    int h0 = hash0(x);
    Lock lock0 = locks[0][h0 % LOCKS];
    try {
      lock0.lock();
      if (x.equals(table[0][h0])) {
        table[0][h0] = null;
        return true;
      } else {
        int h1 = hash1(x);
        Lock lock1 = locks[1][h1 % LOCKS];
        try {
          lock1.lock();
          if (x.equals(table[1][h1])) {
            table[1][h1] = null;
            return true;
          }
          return false;
        } finally {
          lock1.unlock();
        }
      }
    } finally {
      lock0.unlock();
    }
  }
```

The *add(x)* method is the most interesting. It successively removes conflicting items until every key has a slot. To add x, the method swaps x with y, the current occupant of $table[0][h0(x)]$. If the prior value y was null, it is done, otherwise, it swaps the newly nestles value y for the current occupant of $table[1][h1(y)]$ in the same way and continues swapping entries (alternating tables) until it finds an empty slot.

8.1.3 Experiment Description

The test creates 8 threads that perform *enq()* and *deq()* operations in sequential and parallel. The dequeue values are revised and if a value is missing a fail assertion is thrown.

8.1.4 Observations and Interpretations

When executing the test the test fails on the parallel part due to a missing value when executing the parallel both test.

```
Exception in thread "Thread-5" Exception in thread "
    Thread-7" junit.framework.AssertionFailedError: Th 0
            DeqThread: missing value: 3
```

8.1.5 Proposed changes to fix the problem

By adding terminal output through System.out in the *remove()* method or Debugging the file the execution works, thus becoming a problem of observation, since the fail assertion only happens when its not observed.

```
compile-test:
.sequential add, contains, and remove
.parallel add
.parallel remove
.parallel both

Time: 0.071

OK (4 tests)
```

8.1.6 Proposed solution

It is not clear why the observation would affect the execution, thus it is possible that the issue is a propagation of the values in the memory which are forced to be executed constantly through the debug implementation.

8.2 Refinable Hash Set

8.2.1 Particular Case

In this test we want to refine the granularity of locking as the table size grows, so that the number of locations in a stripe does not continuously grow by using a globally shared owner field that combines a *Boolean* value with a reference to a thread.

8.2.2 Solution

While a resizing is in progress, the *owner* Boolean value is true, and the associated reference indicates the thread that is in charge of resizing. We use the owner as a mutual exclusion flag between the *resize()* method and any of the *add()* methods, so that while resizing, there will be no successful updates, and while updating, there will be no successful resizes.

```
public void resize() {
  int oldCapacity = table.length;
  int newCapacity = 2 * oldCapacity;
  Thread me = Thread.currentThread();
  if (owner.compareAndSet(null, me, false, true)) {
    try {
      if (table.length != oldCapacity) {  // someone
         else resized first
        System.out.println("Someone else resizd first
           ");
        return;
      }
      quiesce();
      List<T>[] oldTable = table;
      table = (List<T>[]) new List[newCapacity];
      for (int i = 0; i < newCapacity; i++)
        table[i] = new ArrayList<T>();
      locks = new ReentrantLock[newCapacity];
      for (int j = 0; j < locks.length; j++) {
        locks[j] = new ReentrantLock();
      }
      initializeFrom(oldTable);
```

```
      } finally {
        owner.set(null, false);        // restore prior
           state
      }
    }
  }
```

The *acquire()* and the *resize()* methods guarantee mutually exclusive access via the flag principle using the mark field of the *owner* flag and the tables locks array, *acquire()* first acquires its locks and then reads the mark field, while *resize()* first sets mark and then reads the locks during the *quiesce()* call. This ordering ensures that any thread that acquires the locks after *quiesce()* has completed will see that the set is in the processes of being resized, and will back off until the resizing is complete.

```
  protected void quiesce() {
    for (ReentrantLock lock : locks) {
      while (lock.isLocked()) {}  // spin
    }
  }
```

```
  public void acquire(T x) {
    boolean[] mark = {true};
    Thread me = Thread.currentThread();
    Thread who;
    while (true) {
      do { // wait until not resizing
        who = owner.get(mark);
      } while (mark[0] && who != me);
      ReentrantLock[] oldLocks = this.locks;
      int myBucket = Math.abs(x.hashCode() % oldLocks.
         length);
      ReentrantLock oldLock = oldLocks[myBucket];
      oldLock.lock();  // acquire lock
      who = owner.get(mark);
      if ((!mark[0] || who == me) && this.locks ==
         oldLocks) { // recheck
        return;
      } else {  //  unlock & try again
```

```
            oldLock.unlock();
        }
    }
}
```

8.2.3 Experiment Description

The test creates 8 threads that perform *enq()* and *deq()* operations in sequential and parallel. The dequeue values are revised and if a value is missing a fail assertion is thrown.

8.2.4 Observations and Interpretations

When executing the test the test fails on the parallel part due to a duplicate value when executing the parallel both test.

```
Assertion error: duplicate pop Exception in thread "
    Thread-13" junit.framework.AssertionFailedError:
    DeqThread: duplicate pop
```

8.2.5 Proposed changes to fix the problem

By adding terminal output through System.out in the *remove()* method or Debugging the file the execution works, thus becoming a problem of observation, since the fail assertion only happens when its not observed, this happens with the TCuckooHash and CuckooHas tests.

```
compile-test:
.sequential add, contains, and remove
.parallel add
.parallel remove
.parallel both

Time: 0.081

OK (4 tests)
```

8.2.6 Proposed solution

It is not clear why the observation would affect the execution, thus it is possible that the issue is a propagation of the values in the memory which are forced to be executed constantly through the debug implementation. Also worth mentioning that several hashing test suffer from this same situation and adding volatile didn't help as in other tests.

Chapter 9

Skiplists and Balanced Search

9.1 Double Ended Queue

9.1.1 Particular Case

In this experiment each thread keeps a pool of tasks waiting to be executed in the form of a double-ended queue (DEQueue), providing *pushBottom()*, *popBottom()*, and *popTop()* methods.

9.1.2 Solution

According to the theory when a thread creates a new task, it calls *pushBottom()* to push that task onto its *DEQueue*. When a thread needs a task to work on, it calls *popBottom()* to remove a task from its own *DEQueue*. If the thread discovers its queue is empty, then it becomes a thief, it chooses a victim thread at random, and calls that threads DEQueues *popTop()* method to steal a task for itself.

```
public Runnable popTop() {
  int[] stamp = new int[1];
  int oldTop = top.get(stamp), newTop = oldTop + 1;
  int oldStamp = stamp[0], newStamp = oldStamp + 1;
  if (bottom <= oldTop) // empty
    return null;
  Runnable r= tasks[oldTop];
  if (top.compareAndSet(oldTop, newTop, oldStamp,
     newStamp))
    return r;
  return null;
}
```

9.1.3 Experiment Description

The test creates 16 threads, eight that need to be coordinate in order to *pushBottom()* and *popBottom()* an array of values. All threads have to cooperate to add and remove elements from the queue and eight *Dummy()* threads will be stealing execution from the running ones. After each thread execution, if everything works according to the test the map will have all values set to true. If that is not the case, a duplicate or missing fail will be raised.

9.1.4 Observations and Interpretations

The tests executed as expected and no errors where found.

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 1.162 sec

------------- Standard Output ---------------
sequential pushBottom and popBottom
concurrent pushBottom and popBottom
---------------- ---------------------- ----------------

test-single:
BUILD SUCCESSFUL (total time: 2 seconds)
```

9.2 Parallel Matrix Multiplication

9.2.1 Particular Case

In this experiment, the particular case we are trying to study is matrix parallel multiplication by devising methods to add and multiply matrices.

9.2.2 Solution

To implement a parallel multiplication we create a Matrix class that provides *set()* and *get()* methods to access matrix elements, along with a constant-time *split()* method that splits an n-by-n matrix into four $(n/2)$-by-$(n/2)$ submatrices.

```
double get(int row, int col) {
  return data[row+rowDisplace][col+colDisplace];
}

void set(int row, int col, double value) {
  data[row+rowDisplace][col+colDisplace] = value;
}

Matrix[][] split() {
  Matrix[][] result = new Matrix[2][2];
  int newDim = dim / 2;
  result[0][0] = new Matrix(data, rowDisplace,
     colDisplace, newDim);
  result[0][1] = new Matrix(data, rowDisplace,
     colDisplace + newDim, newDim);
  result[1][0] = new Matrix(data, rowDisplace +
     newDim, colDisplace, newDim);
  result[1][1] = new Matrix(data, rowDisplace +
     newDim, colDisplace + newDim, newDim);
  return result;
}
```

For simplicity, we consider matrices whose dimension n is a power of 2. Any such matrix can be decomposed into four submatrices, thus their sums can be done in parallel.

9.2.3 Experiment Description

The test creates two matrices with a dimension of four which will be multiplied in parallel, the resulting matrix will be verified and a fail assertion will be thrown is the result is incorrect.

9.2.4 Observations and Interpretations

When compiling the code the Matrix class had to be changed from private to public and the offset and dimension of the Matrix must be added to the test class as shown below.

```
    Matrix aa = new Matrix(a, 0, 0, 4);
    Matrix bb = new Matrix(b, 0, 0, 4);
```

After making this changes the codes compiles and everything works fine.

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 0.369 sec

------------- Standard Output ---------------
run
------------- ---------------- ---------------

test-single:
BUILD SUCCESSFUL (total time: 0 seconds)
```

9.2.5 Proposed changes to fix the problem

The MatrixTask class needs to have the visibility of the inner Matrix class change to public in order for the test to view it.

```
  public static class Matrix {
    int dim;
    double[][] data;
    int rowDisplace;
    int colDisplace;
    ---
  }
```

9.3 Unbounded Double Ended Queue

9.3.1 Particular Case

In this experiment we implement an unbounded double queue based on the previous double queue implemented.

9.3.2 Solution

According to the theory the *UnboundedDEQueue* class dynamically resizes itself as needed. We implement the *UnboundedDEQueue* in a cyclic array, with top and bottom fields as in the *BoundedDEQueue* (except indexed modulo the array's capacity). As before, if bottom is less than or equal to top, the *UnboundedDEQueue* is empty. The difference resides in the *CircularArray()* class which provides *get()* and *put()* methods that add and remove tasks, and a *resize()* method that allocates a new circular array and copies the old array's contents into the new array.

```
class CircularArray {
  private int logCapacity;
  private Runnable[] currentTasks;
  CircularArray(int logCapacity) {
    this.logCapacity = logCapacity;
    currentTasks = new Runnable[1 << logCapacity];
  }
  int capacity() {
    return 1 << this.logCapacity;
  }
  Runnable get(int i) {
    return this.currentTasks[i % capacity()];
  }
  void put(int i, Runnable task) {
    this.currentTasks[i % capacity()] = task;
  }
  CircularArray resize(int bottom, int top) {
    CircularArray newTasks =
        new CircularArray(this.logCapacity+1);
    for (int i = top; i < bottom; i++) {
      newTasks.put(i, this.get(i));
```

```
        }
        return newTasks;
    }
  }
```

9.3.3 Experiment Description

The test creates 16 threads, eight that need to be coordinate in order to *pushBottom()* and *popBottom()* an array of values. All threads have to cooperate to add and remove elements from the queue and eight *Dummy()* threads will be stealing execution from the running ones. After each thread execution, if everything works according to the test the map will have all values set to true. If that is not the case, a duplicate or missing fail will be raised.

9.3.4 Observations and Interpretations

The test needs some changes in order to compile, once this changes are implemented the test executes as expected.

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 1.4 sec

------------- Standard Output ---------------
sequential pushBottom and popBottom
concurrent pushBottom and popBottom
---------- ---------------- ----------------

test-single:
BUILD SUCCESSFUL (total time: 1 second)
```

9.3.5 Proposed changes to fix the problem

The UDEQueueTest class needs to have replaced the calls to non defined reset by new instance in order to compile.

```
  //instance.reset();
  instance = new UDEQueue(THREADS);
```

Chapter 10

Priority Queues

10.1 Sequential Heap

10.1.1 Particular Case

In this experiment we implement a linearisable priority queue that supports priorities from an unbounded range. It uses fine-grained locking for synchronization.

10.1.2 Solution

According to the theory a sequential heap implementation represent a binary heap by using an array of nodes, where the trees root is array entry 1, and the right and left children of array entry i are entries $2i$ and $(2i)+1$, respectively. Each node has an item and a score field. To add an item, the *add()* method sets child to the index of the first empty array slot.

```
public void add(T item, int priority) {
  int child = next++;
  heap[child].init(item, priority);
  while (child > ROOT) {
    int parent = child / 2;
    int oldChild = child;
    if (heap[child].priority < heap[parent].priority)
        {
      swap(child, parent);
      child = parent;
    } else {
      return;
    }
  }
}
```

To remove and return the highest-priority item, the *removeMin()* method records the roots item, which is the highest-priority item in the tree.

```
public T removeMin() {
  int bottom = --next;
  T item = heap[ROOT].item;
  swap(ROOT, bottom);
  if (bottom == ROOT) {
```

```
      return item;
    }
    int child = 0;
    int parent = ROOT;
    while (parent < heap.length / 2) {
      int left = parent * 2; int right = (parent * 2) +
          1;
      if (left >= next) {
        break;
      } else if (right >= next || heap[left].priority <
          heap[right].priority) {
        child = left;
      } else {
        child = right;
      }
      // If child higher priority than parent swap then
          else stop
      if (heap[child].priority < heap[parent].priority)
          {
        swap(parent, child); // Swap item, key, and tag
            of heap[i] and heap[child]
        parent = child;
      } else {
        break;
      }
    }
    return item;
  }
```

10.1.3 Experiment Description

The test adds 512 elements to the heap, does a check of the elements and removes the values with min priority. We verify the priorities of the removed elements and assert a fail if we have non ascending priorities.

10.1.4 Observations and Interpretations

The test execute as expected and no problems are thrown.

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 0.413 sec

------------- Standard Output ---------------
testSequential
OK.
------------- ---------------- ---------------

test-single:
BUILD SUCCESSFUL (total time: 1 second)
```

10.2 Skiplist-Based Unbounded Priority Queue

10.2.1 Particular Case

In this experiment we implement a skip-list priority queue which in contrast to the *FineGrainedHeap* priority queue algorithm, that the underlying heap structure requires complex, coordinated rebalancing. We requires no rebalancing.

10.2.2 Solution

According to the theory the *PrioritySkipList* class sorts items by priority instead of by hash value, ensuring that high-priority items (the ones we want to remove first) appear at the front of the list. Removing the item with highest priority is done lazily. A node is logically removed by marking it as removed, and is later physically removed by unlinking it from the list.

```
public T removeMin() {
    Node<T> node = skiplist.findAndMarkMin();

    if (node != null) {
      skiplist.remove(node);
      return node.item;
    } else{
      return null;
    }
}
```

The *add()* and *remove()* calls take skiplist nodes instead of items as arguments and results. These methods are straightforward adaptations of the corresponding *LockFreeSkipList* methods. This class nodes differ from *LockFreeSkipList* nodes in two fields, an integer score field, and an *AtomicBoolean* marked field used for logical deletion from the priority queue (not from the skiplist).

```
boolean add(Node node) {
  int bottomLevel = 0;
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL +
     1];
```

```
    Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL +
       1];
    while (true) {
      boolean found = find(node, preds, succs);
      if (found) { // if found it's not marked
        return false;
      } else {
        for (int level = bottomLevel; level <= node.
           topLevel; level++) {
          Node<T> succ = succs[level];
          node.next[level].set(succ, false);
        }
        // try to splice in new node in bottomLevel
           going up
        Node<T> pred = preds[bottomLevel];
        Node<T> succ = succs[bottomLevel];
        node.next[bottomLevel].set(succ, false);
        if (!pred.next[bottomLevel].compareAndSet(succ,
            node, false, false)) {// lin point
          continue; // retry from start
        }
        // splice in remaining levels going up
        for (int level = bottomLevel+1; level <= node.
           topLevel; level++) {
          while (true) {
            pred = preds[level];
            succ = succs[level];
            if (pred.next[level].compareAndSet(succ,
               node, false, false))
              break;
            find(node, preds, succs); // find new preds
                and succs
          }
        }
        return true;
      }
    }
  }
```

```
boolean remove(Node<T> node) {
  int bottomLevel = 0;
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL +
     1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL +
     1];
  Node<T> succ;
  while (true) {
    boolean found = find(node, preds, succs);
    if (!found) {
      return false;
    } else {
      // proceed to mark all levels
      // some levels could stil be unthreaded by
         concurrent add() while being marked
      // other find()s could be modifying node's
         pointers concurrently
      for (int level = node.topLevel; level >=
         bottomLevel+1; level--) {
        boolean[] marked = {false};
        succ = node.next[level].get(marked);
        while (!marked[0]) { // until I succeed in
           marking
          node.next[level].attemptMark(succ, true);
          succ = node.next[level].get(marked);
        }
      }
      // proceed to remove from bottom level
      boolean[] marked = {false};
      succ = node.next[bottomLevel].get(marked);
      while (true) { // until someone succeeded in
         marking
        boolean iMarkedIt = node.next[bottomLevel].
           compareAndSet(succ, succ, false, true);
        succ = succs[bottomLevel].next[bottomLevel].
           get(marked);
        if (iMarkedIt) {
```

```
            // run find to remove links of the
               logically removed node
            find(node, preds, succs);
            return true;
          } else if (marked[0]) return false; //
             someone else removed node
          // else only succ changed so repeat
        }
      }
    }
  }
```

10.2.3 Experiment Description

The test creates 8 threads, eight that need to be coordinate in order to *add()* and *removeMin()* an array of values. All threads have to cooperate to add and remove elements from the queue and a validation of the priorities is done to validate non-ascending priorities. If that is not the case, a fail will be raised.

10.2.4 Observations and Interpretations

The test needs some changes in order to execute and avoid starvation.

```
------------- Standard Output -------------
add
OK.
testParallelAdd
OK.
testParallelBoth
```

10.2.5 Proposed changes to fix the problem

The *findAndMarkMin()* method must choose the coresponding element that will be removed in order to mark it and later on the *remove()* method we verify the marked node and remove it, but the atomicity of this operations are not guarantee, thus when a node is marked it can be marked by other

thread causing the remove process to hand, since the internal logic tries to remove the node and has no escape sequence when the mark is affected.

```
public T removeMin() {
  synchronized(this) {
    Node<T> node = skiplist.findAndMarkMin();

    if (node != null) {
      skiplist.remove(node);
      return node.item;
    } else{
      return null;
    }
  }
}
```

By doing so, the execution finishes as expected and no errors are found.

```
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time
   elapsed: 0.358 sec

------------- Standard Output ---------------
add
OK.
testParallelAdd
OK.
testParallelBoth
OK.
------------- ---------------- ---------------

test-single:
BUILD SUCCESSFUL (total time: 0 seconds)
```