**Exercise 120.** Consider the simple lock-free queue for a single enqueuer and a single dequeuer, described earlier in Chapter 3. The queue is presented in Fig. 10.20.

```
class TwoThreadLockFreeQueue<T> {
  int head = 0, tail = 0;
  T[] items;

  public TwoThreadLockFreeQueue(int capacity) {
    head = 0; tail = 0;
    items = (T[]) new Object[capacity];
  }

  public void enq(T x) {
    while (tail - head == items.length) {};
    items[tail % items.length] = x;
    tail++;
  }

  public Object deq() {
    while (tail - head == 0) {};
    Object x = items[head % items.length];
    head++;
    return x;
  }
}
```

Figure 10.20 A Lock-free FIFO queue with blocking semantics for a single enqueuer and single dequeuer. The queue is implemented in an array. Initially the head and tail fields are equal and the queue is empty. If the head and tail differ by capacity, then the queue is full. The enq() method reads the head field, and if the queue is full, it repeatedly checks the head until the queue is no longer full. It the stores the object in the array, and increments the tail field. The deq() method works in a symmetric way.

This queue is blocking, that is, removing an item from an empty queue or inserting an item to a full one causes the threads to block (spin). The surprising thing about this queue is that it requires only loads and stores and not a more powerful read–modify–write synchronization operation. Does it however require the use of a memory barrier? If not, explain, and if so, where in the code is such a barrier needed and why?

É necessário o uso de um bloqueio nas operações de inclusão e remoção de ítens. Threads diferentes podem, por exemplo, passar pela verificação, alterar ou obter o mesmo ítem e incrementar os valores de *head* e *tail*, de forma que posições sejam ignoradas em execuções posteriores. Também é possível sobrescrever ítens, caso a verificação seja ultrapassada por várias threads próximo ao limite inferior ou superior. Várias inconsistências podem ser geradas por conta da ausência total de bloqueios nesse método.

**Exercise 121.** Design a bounded lock-based queue implementation using an array instead of a linked list.

1. Allow parallelism by using two separate locks for head and tail.

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class ArrayBoundedQueue<T> implements Pool<T> {
   private ReentrantLock enqLock, deqLock;
   private Condition notEmptyCondition, notFullCondition;
   private T[] values;
   private int head, tail, capacity;

   public ArrayBoundedQueue(int capacity) {
      this.capacity = capacity;
      this.head = 0;
      this.tail = 0;
      this.values = (T[]) new Object[capacity];

      this.enqLock = new ReentrantLock();
      this.deqLock = new ReentrantLock();

      this.notFullCondition = enqLock.newCondition();
      this.notEmptyCondition = deqLock.newCondition();
   }

   @Override
   public void put(T item) {
      boolean mustWakeDequeuers = false;
      this.enqLock.lock();
      try {
         while (this.getSize() == this.capacity) {
            try {
               this.notFullCondition.await();
            } catch (InterruptedException e1) {
               // ops... something was supposed to be handled here
            }
         }
         this.values[this.head % this.capacity] = item;
         this.head++;
         if (this.getSize() == 0) {
```

```
            mustWakeDequeuers = true;
        }
    } finally {
        enqLock.unlock();
    }

    if (mustWakeDequeuers) {
        this.deqLock.lock();
        try {
            this.notEmptyCondition.signalAll();
        } finally {
            this.deqLock.unlock();
        }
    }
}

@Override
public T get() {
    T result;
    boolean mustWakeEnqueuers = false;
    this.deqLock.lock();
    try {
        while (this.getSize() == 0) {
            try {
                this.notEmptyCondition.await();
            } catch (InterruptedException e) {
                // ops... something was supposed to be handled here
            }
        }

        result = this.values[this.tail % this.capacity];
        this.tail++;
        if (this.getSize() == this.capacity) {
            mustWakeEnqueuers = true;
        }
    } finally {
        this.deqLock.unlock();
    }

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

```
        return result;
    }

    private int getSize() {
        return this.head - this.tail;
    }
}
```

2. Try to transform your algorithm to be lock-free. Where do you run into difficulty?

A dificuldade maior é em fazer com que posições incorretas sejam acessadas por threads diferentes. Tal problema é solucionado utilizando as verificações com operações atômicas através de um AtomicReference.

```
import java.util.concurrent.atomic.AtomicReference;

public class LockFreeArrayBoundedQueue<T> implements Pool<T> {
    private AtomicReference<T>[] values;
    private int head, tail;

    public LockFreeArrayBoundedQueue(int capacity) {
        this.head = 0;
        this.tail = 0;

        // create an array of AtomicReference to be used on item storage
        this.values = new AtomicReference[capacity];
        for (int i = 0; i < this.values.length; i++) {
            this.values[i] = new AtomicReference<T>();
        }
    }

    @Override
    public void put(T item) {
        if (item == null) {
            throw new RuntimeException("An item can not be null.");
        }

        // the item will only be stored if the desired position is null (no one stored on it before)
        while (!this.values[this.head % this.values.length].compareAndSet(null, item)) {

        }

        this.head++;
    }
```

```
    @Override
    public T get() {
        T result = null;

        // a null result on this operation means that the current thread is attempting to access an
        // empty array or a position that was already removed by another thread
        while (result == null) {
            result = this.values[this.tail % this.values.length].getAndSet(null);
        }

        this.tail++;
        return result;
    }
}
```

**Exercise 122.** Consider the unbounded lock-based queue's deq() method in Fig. 10.8. Is it necessary to hold the lock when checking that the queue is not empty? Explain.

Sim, pois mais de uma thread pode passar pela verificação quando os dados estão próximos ao limite de acabar. Com isso, alguma thread pode remover o último ítem e o EmptyException será trocado por um NullPointerException na execução do "result = head.next.value".

**Exercise 123.** In Dante's Inferno, he describes a visit to Hell. In a very recently discovered chapter, he encounters five people sitting at a table with a pot of stew in the middle. Although each one holds a spoon that reaches the pot, each spoon's handle is much longer than each person's arm, so no one can feed him- or herself. They are famished and desperate.
Dante then suggests "why do not you feed one another?"
The rest of the chapter is lost.

1. Write an algorithm to allow these unfortunates to feed one another. Two or more people may not feed the same person at the same time. Your algorithm must be, well, starvation-free.

```
import java.util.Random;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Person extends Thread {
    private String name;
    private Person personToFeed;
    private Random random;

    // this lock will be used to not allow a person to eat and feed at the same time
    private Lock actionLock;
```

```
public Person(String name) {
    this.name = name;
    this.random = new Random();
    this.actionLock = new ReentrantLock();
}

public boolean eat() {
    if (this.actionLock.tryLock()) {
        try {
            try {
                // sometimes it spend a little more time to eat... but it will be more than one
                // second and less than three seconds
                sleep((this.random.nextInt(2) * 1000) + 1000);
            } catch (InterruptedException e) {
                // do nothing
            }

            return true;
        } finally {
            this.actionLock.unlock();
        }
    }

    return false;
}

public void feed() {
    if (this.actionLock.tryLock()) {
        try {
            if (this.personToFeed.eat()) {
                System.out.println(this.personToFeed.name + " was fed by " + this.name);
            }
        } finally {
            this.actionLock.unlock();
        }
    }
}

public void registerPersonToFeed(Person personToFeed) {
    this.personToFeed = personToFeed;
}

@Override
public void run() {
    super.run();
```

```
        while (true) {
            this.feed();
        }
    }
}
```

```
import java.util.LinkedList;
import java.util.List;

public class Table {

    private static final int PEOPLE_ON_TABLE = 5;

    private List<Person> people;

    public Table() {
        this.people = new LinkedList<Person>();

        // each one seat on table
        for (int i = 0; i < PEOPLE_ON_TABLE; i++) {
            this.people.add(new Person("Person" + (i + 1)));
        }

        // everyone pick the next one on table to feed and start eating and feeding process
        for (int i = 0; i < PEOPLE_ON_TABLE; i++) {
            Person person = this.people.get(i);
            int personToFeedIndex = (i + 1) % PEOPLE_ON_TABLE;
            person.registerPersonToFeed(this.people.get(personToFeedIndex));
            person.start();
        }
    }

    public static void main(String[] args) {
        new Table();
    }
}
```

2. Discuss the advantages and disadvantages of your algorithm. Is it centralized, decentralized, high or low in contention, deterministic or randomized?

O método é determinístico, fazendo com que cada pessoa alimente o próximo na mesa. Um problema do algorítimo é o fato de que caso a pessoa a ser alimentada também esteja alimentando a primeira não consegue alimentá-la. O uso de locks impede que uma pessoa alimente e seja alimentado ao mesmo tempo.

**Exercise 124.** Consider the linearization points of the enq() and deq() methods of the lock-free queue:

1. Can we choose the point at which the returned value is read from a node as the linearization point of a successful deq()?

Sim. O ponto de linearização (linearization) acontece quando o método tem efeito (takes effect), ou seja o seu resultado é visível por todas as outras threads, neste caso, é quando ele completa com sucesso o compareAndSet na linha 38, ou senão quando lança a exception na linha 33.

2. Can we choose the linearization point of the enq() method to be the point at which the tail field is updated, possibly by other threads (consider if it is within the enq()'s execution interval)? Argue your case.

Sim, o ponto de linearização (linearization) acontece quando é chamado compareAndSet para redirecionar o tail para o novo nó inserido, no momento em que a chamada do método tem efeito.