Übung 3: Semaphoren

Aufgabe 1: Faire Semaphoren
Ein Semaphor wird verwendet um eine beschränkte Anzahl von Ressourcen zu kontrollieren. Ein Semaphor verwaltet einen Zähler, welcher mit der Methode *acquire* dekrementiert und mit der Methode *release* inkrementiert werden kann. Ist der Zähler des Semaphors 0 wenn *acquire* aufgerufen wird, so wartet dieser Aufruf, bis der Zähler wieder positiv ist. Der Zähler des Semaphors kann mit dem Konstruktor initialisiert und mit der Methode *available* abgefragt werden.

Implementieren Sie Ihren Semaphor so, dass first-in first-out garantiert wird, d.h. bei einem `release` muss jener Thread weiterfahren können, welcher am längsten auf die Erlaubnis wartet. Man nennt so eine Semaphor-Implementierung *fair.* Diese Fairness-Bedingung müssen Sie ausprogrammieren, denn Java bietet Ihnen mit `wait/notify` diesbezüglich KEINE Garantien.

Im Folgenden ist ein Gerüst für die Klasse SemaphoreImpl gegeben. Sie finden dieses Gerüst auch im Wochenprojekt 03_Condition_Synchronization.zip im Package `as.semaphore`. In der Klasse SemaphoreTest finden Sie noch einige Tests, die Ihre Implementierung überprüfen. Wie Sie aber wissen, können solche Tests nur Fehler aufzeigen, jedoch keine Korrektheit garantieren. Dies gilt insbesondere für die Fairness-Bedingung.

```
public final class SemaphoreImpl implements Semaphore {
    private int value;

    public Semaphore(int initial) {
        if (initial < 0) throw new IllegalArgumentException();
        value = initial;
    }

    public int available() { … }
    public void acquire() { … }
    public void release() { … }
}
```

Bemerkung:
Die Methode `acquire` deklariert keine `InterruptedException`, d.h. das acquire ist nicht unterbrechbar.

Aufgabe 2: Queue
In der Vorlesung haben Sie eine Implementierung der Queue gesehen. Die Queue kann auch einfach mit Semaphoren implementiert werden. Die Methode `dequeue` sieht wie folgt aus (`used` und `free` sind vom Typ Semaphor, wobei `free` mit dem Wert N initialisiert wird und `used` mit 0).

```
public Object dequeue() {
   used.acquire();
   Object result;
   synchronized(this) {
      result = buf[head];
      buf[head] = null;
      head = (head + 1) % SIZE;
   }
   free.release();
   return result;
}
```

```
public void enqueue(Object x) {
   free.acquire();

   synchronized(this) {

      buf[tail] = x;
       tail = (tail + 1) % SIZE;
   }
   used.release();

}
```

Diese Implementierung ist jedoch bezüglich Synchronisierung zu restriktiv, Threads, die eigentlich ohne Probleme parallel arbeiten könnten werden gegenseitig ausgeschlossen. Erklären Sie wo in dieser Implementierung zu stark synchronisiert wird und korrigieren Sie dieses Problem.