1. Idee:

- Wir definieren eine Klasse `Buffer2`, die (im wesentlichen) aus einem Feld der richtigen Größe, sowie zwei Verweisen `int first, last` zum Einfügen und Entfernen verfügt:

```
public class Buffer2 {
    private int cap, free, first, last;
    private Data[] a;
    public Buffer2(int n) {
        free = cap = n; first = last = 0;
        a = new Data[n];
    }
    ...
```

- Einfügen und Entnehmen sollen synchrone Operationen sein ...

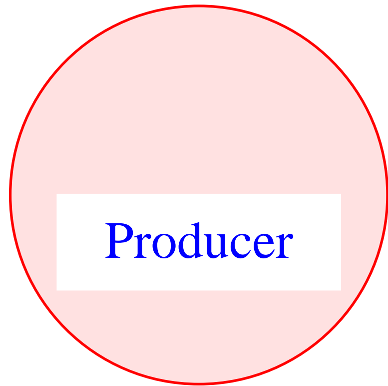
Probleme:

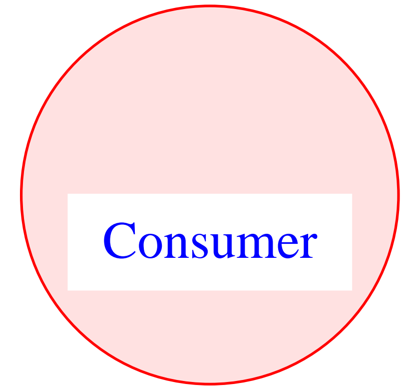
- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

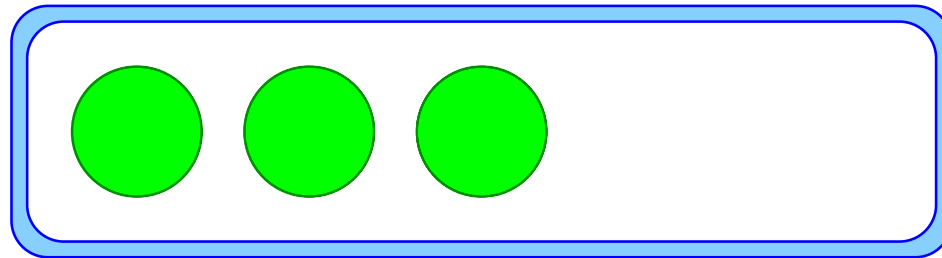
Probleme:

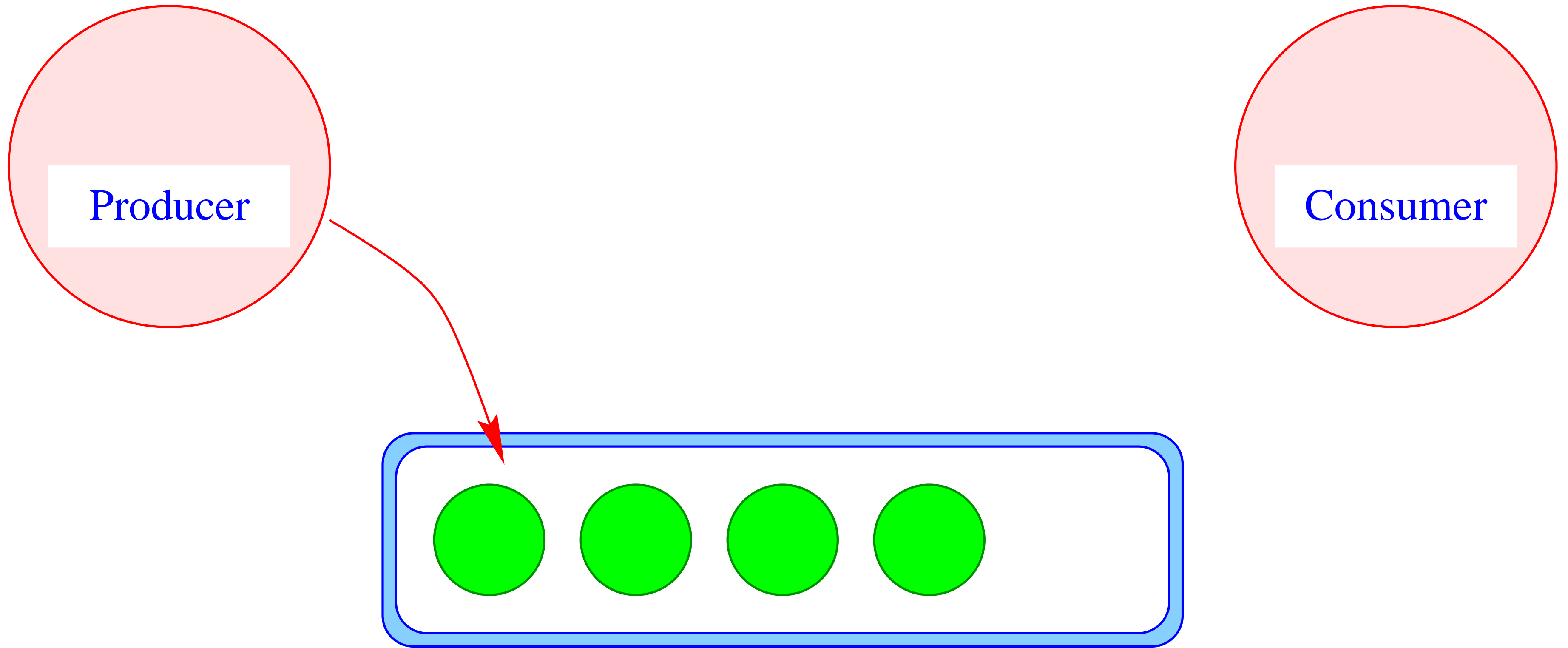
- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

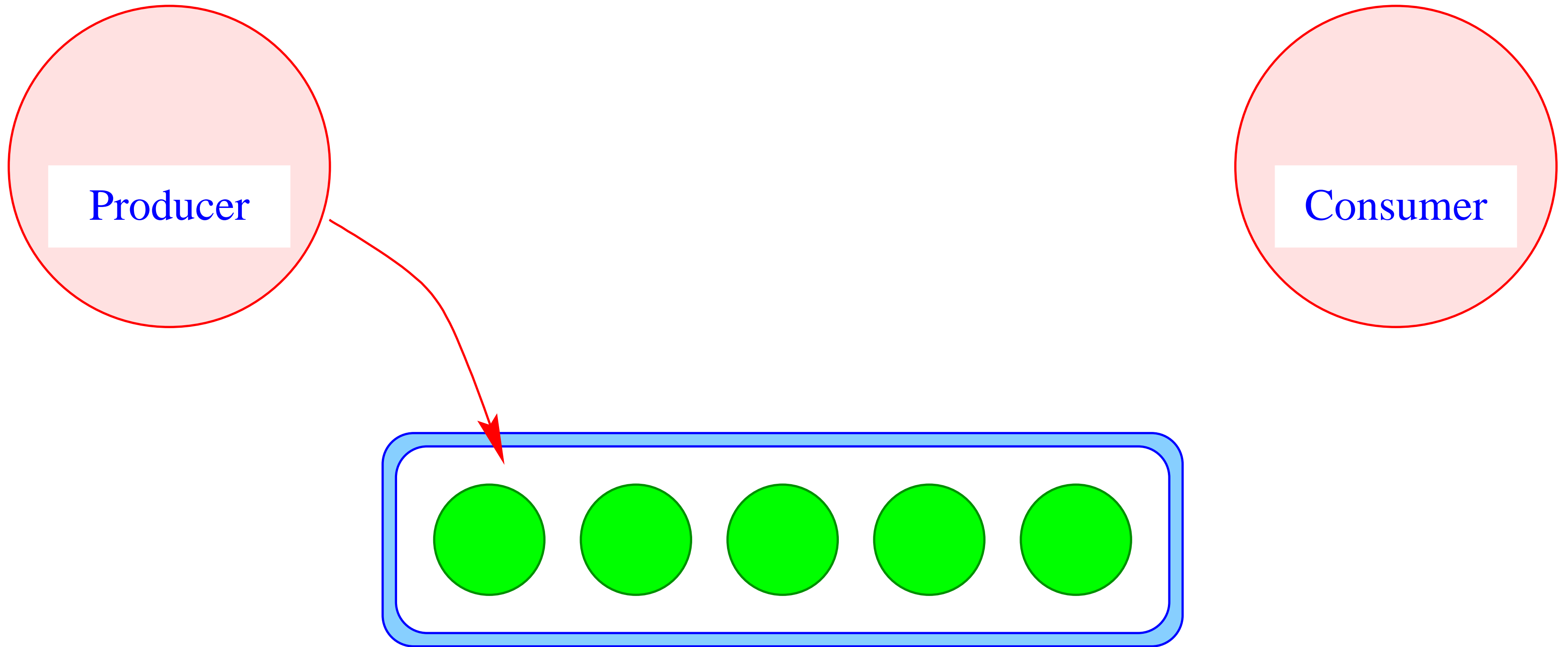
Java's Lösungsvorschlag: Warten ...

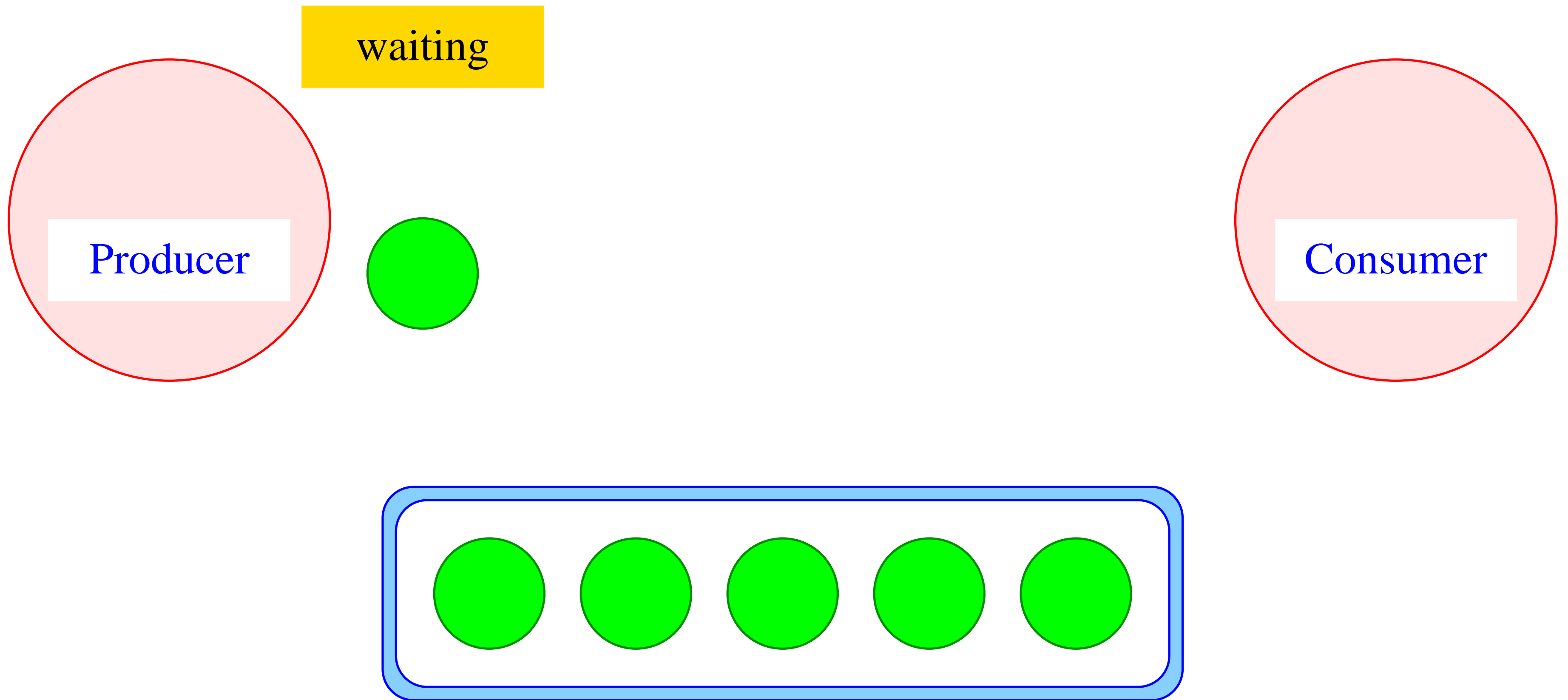

Producer

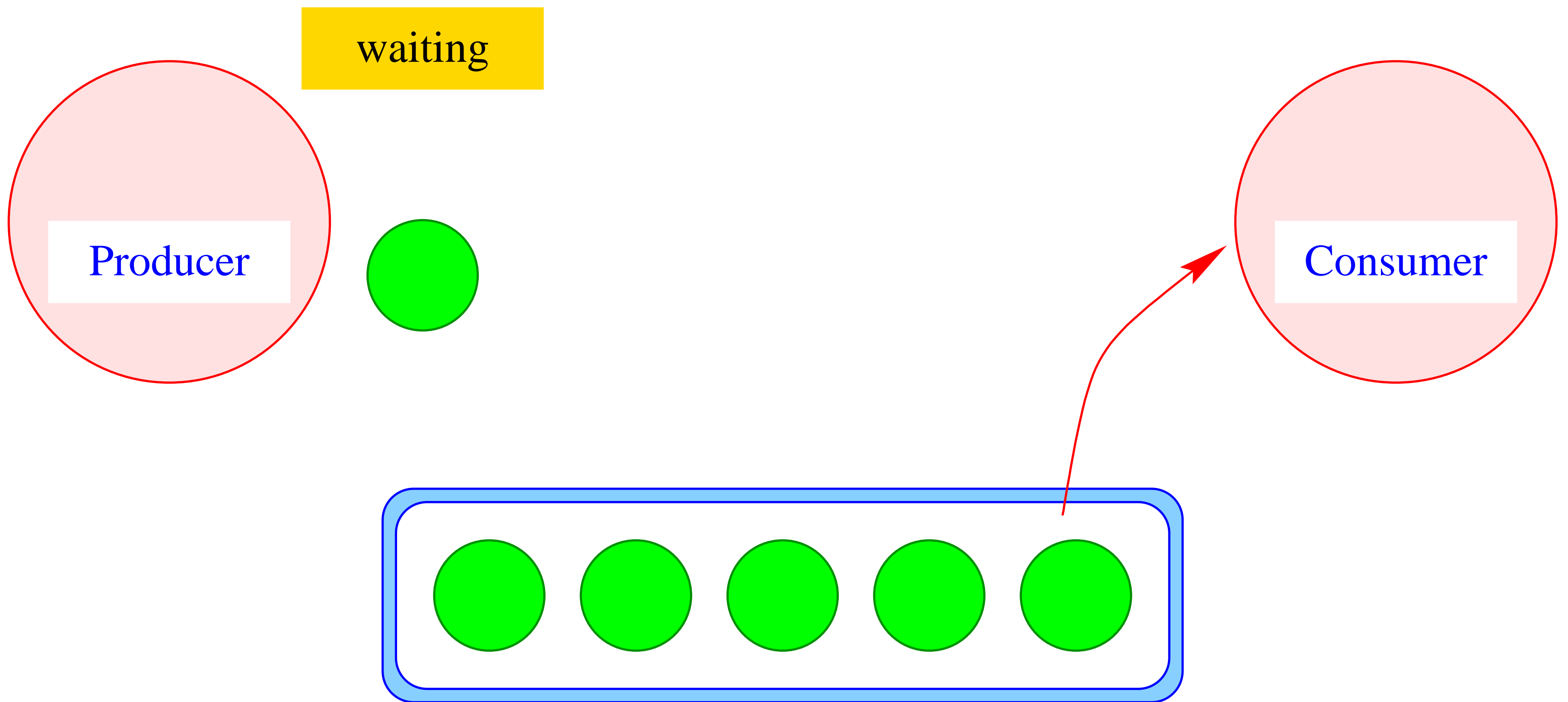

Consumer

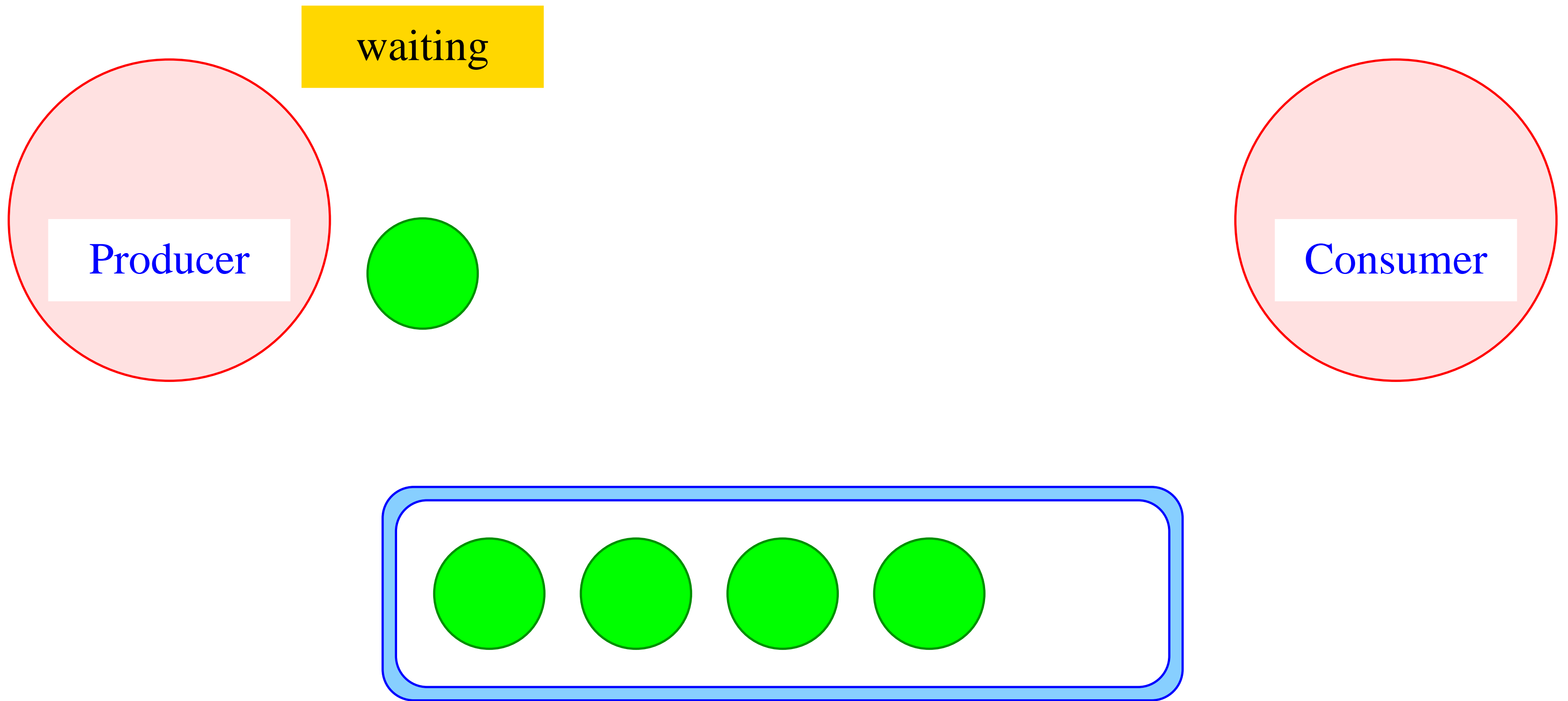


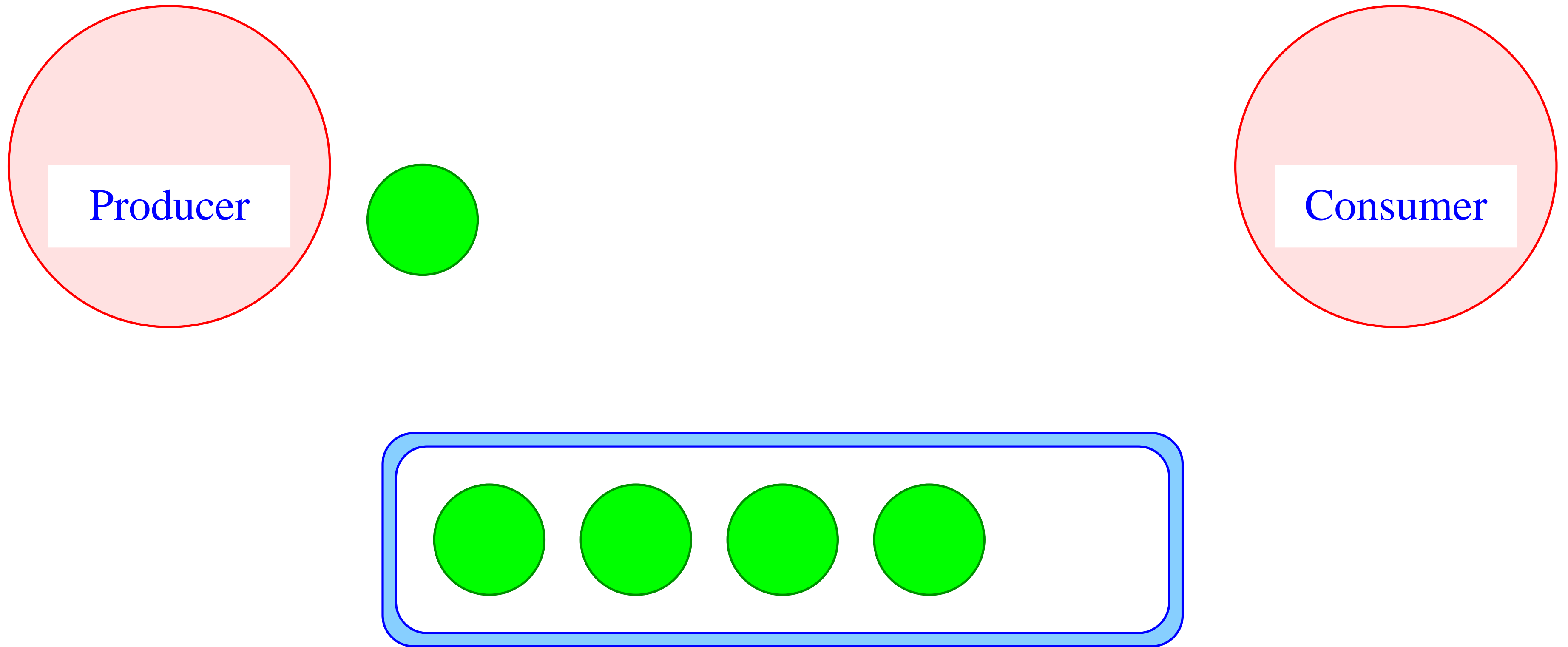

Producer
Consumer

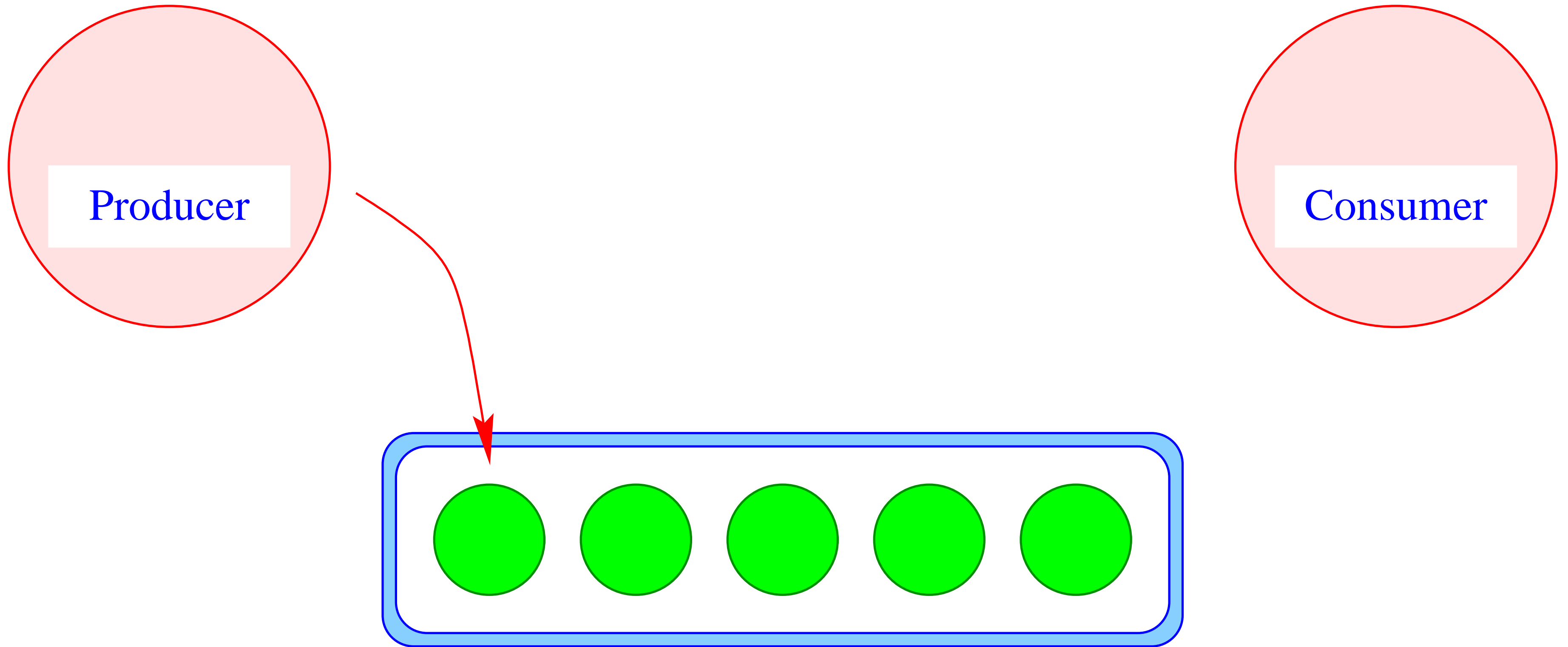

Producer
Consumer


waiting
Producer
Consumer


waiting
Producer
Consumer


waiting
Producer
Consumer


Producer
Consumer

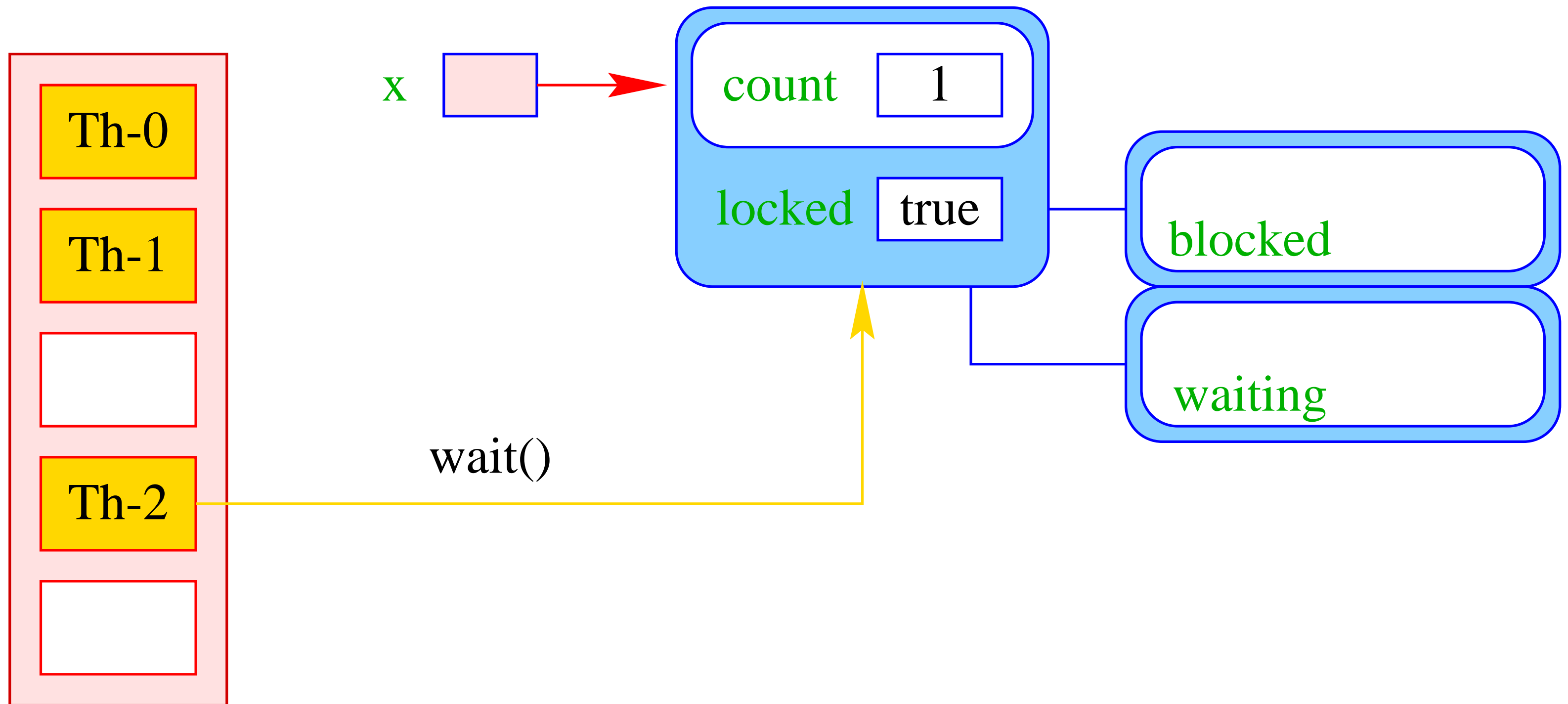

Producer
Consumer

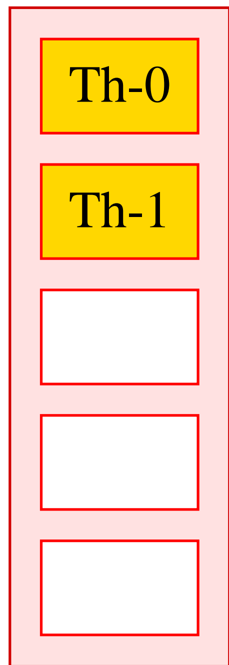
- Jedes Objekt (mit `synchronized`-Methoden) verfügt über eine weitere Schlange `ThreadQueue waitingThreads` am Objekt wartender Threads sowie die Objekt-Methoden:

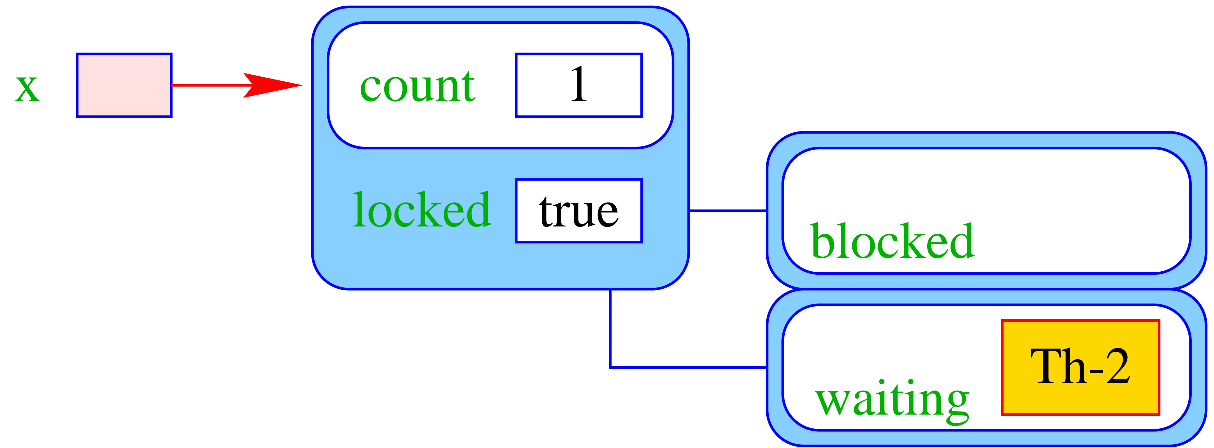
```
public final void wait() throws InterruptedException;

public final void notify();

public final void notifyAll();
```

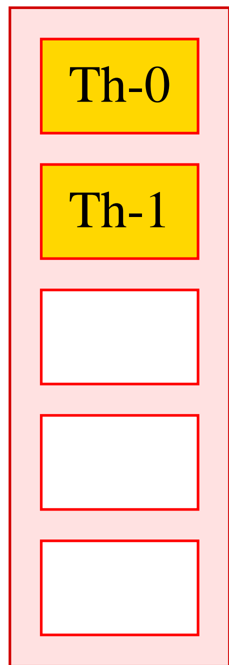
- Diese Methoden dürfen nur für Objekte aufgerufen werden, über deren Lock der Thread verfügt !!!
- Ausführen von `wait();` setzt den Zustand des Threads auf waiting, reiht ihn in eine geeignete Warteschlange ein, und gibt das aktuelle Lock frei:

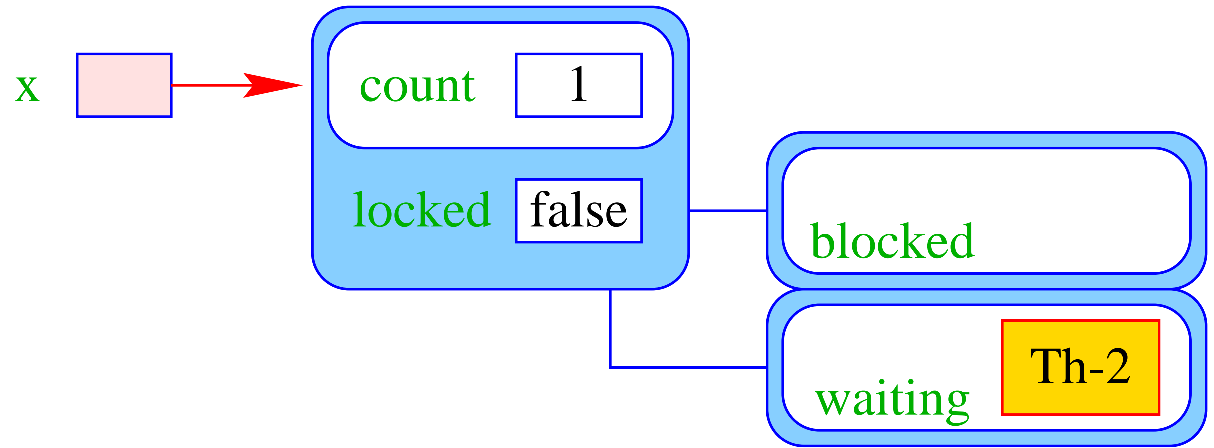
```
public void wait() throws InterruptedException {
    Thread t = Thread.currentThread();
    t.state = waiting;
    waitingThreads.enqueue(t);
    unlock();
}
```


Th-0
Th-1
Th-2
x
count
1
locked
true
blocked
waiting
wait()


Th-0
Th-1


x
count
1
locked
true
blocked
waiting
Th-2


Th-0
Th-1


x
count
1
locked
false
blocked
waiting
Th-2

- Ausführen von `notify();` weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand ready ...

```
public void notify() {
    if (!waitingThreads.isEmpty()) {
        Thread t = waitingThreads.dequeue();
        t.state = ready;
    }
}
```

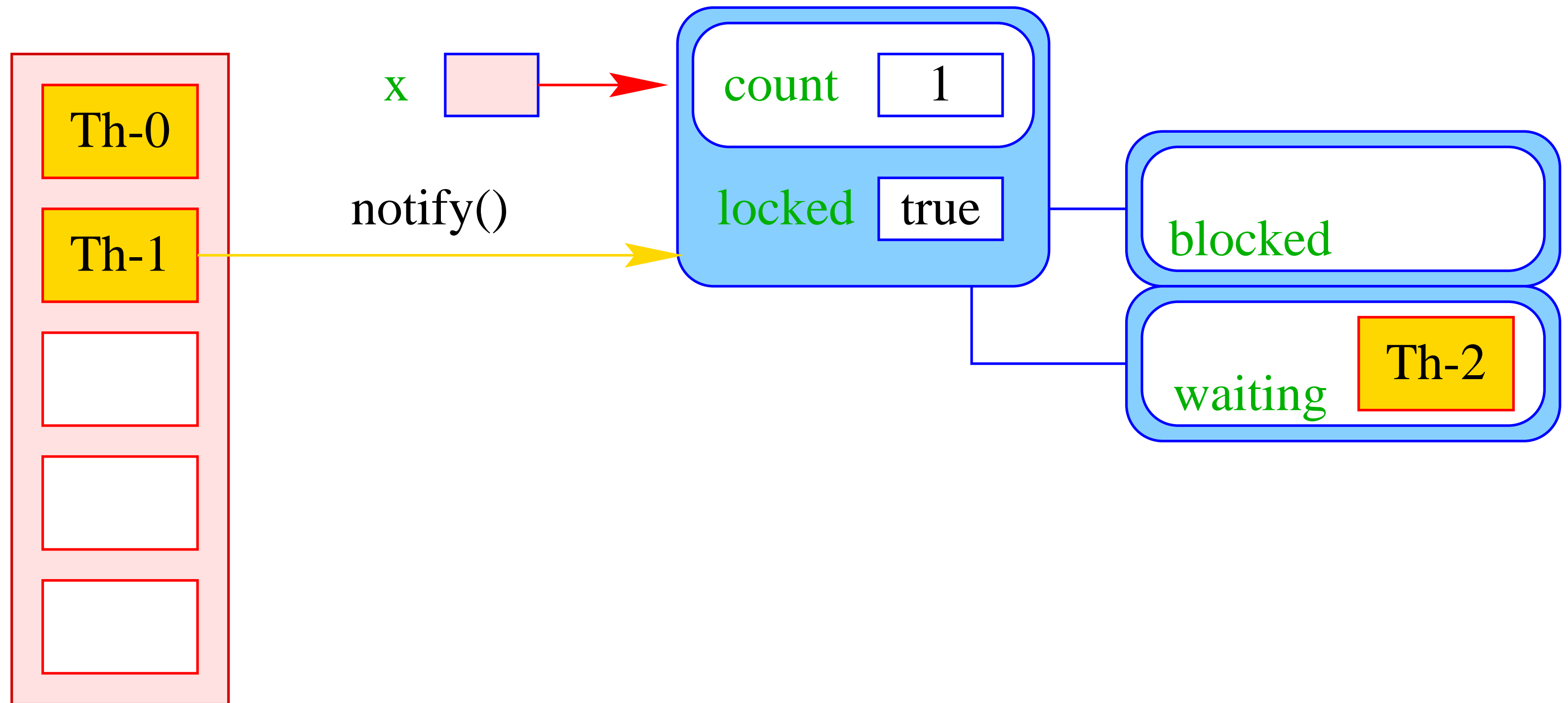
- ... mit der Auflage, erneut das Lock zu erwerben, d.h. als erste Operation hinter dem `wait();` ein `lock()` auszuführen.

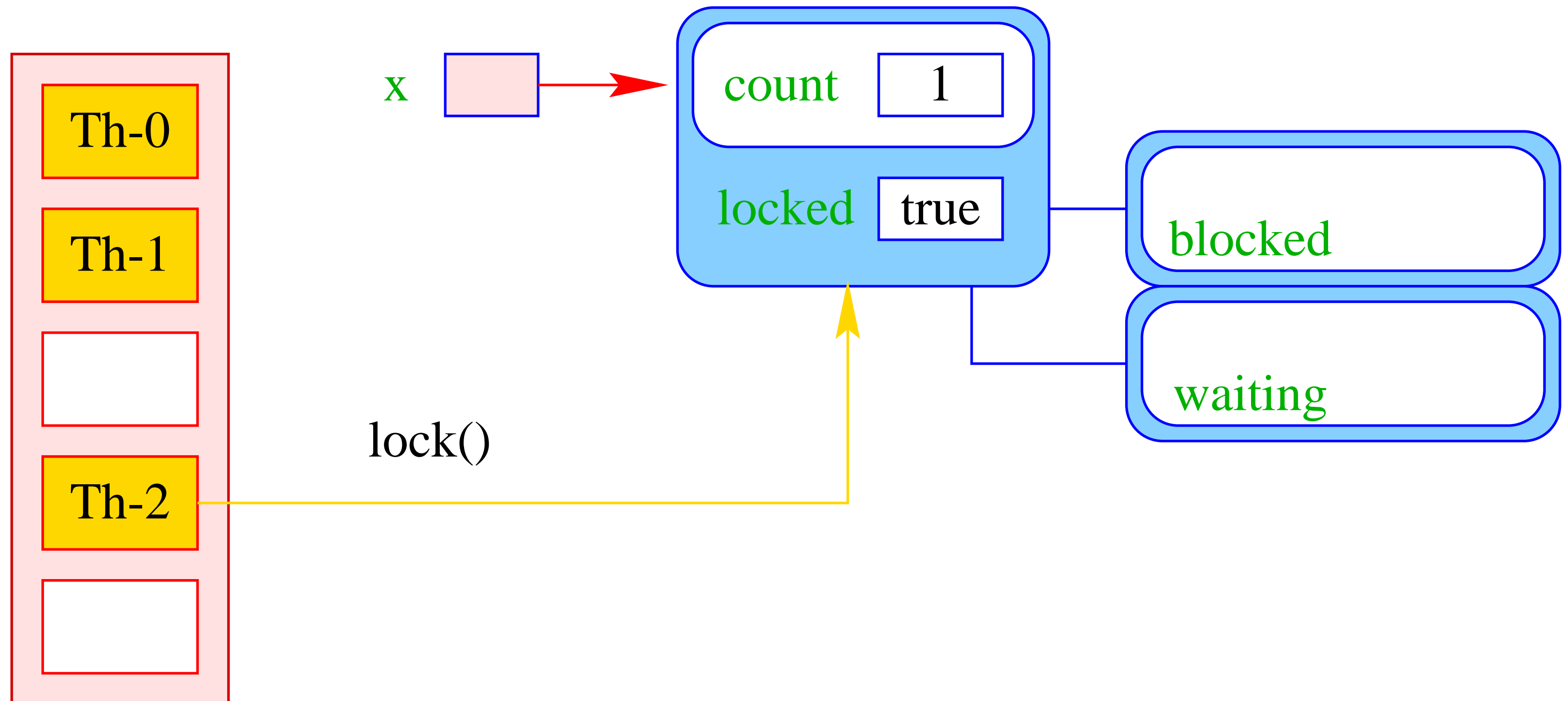
- Ausführen von `notify();` weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand ready ...

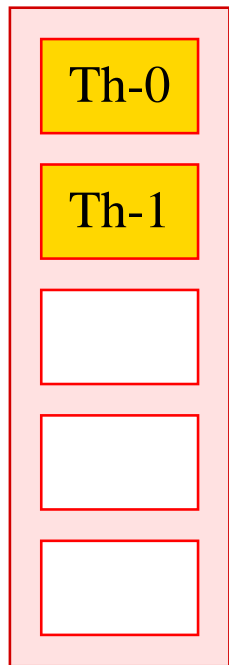
```
public void notify() {
    if (!waitingThreads.isEmpty()) {
        Thread t = waitingThreads.dequeue();
        t.state = ready;
    }
}
```

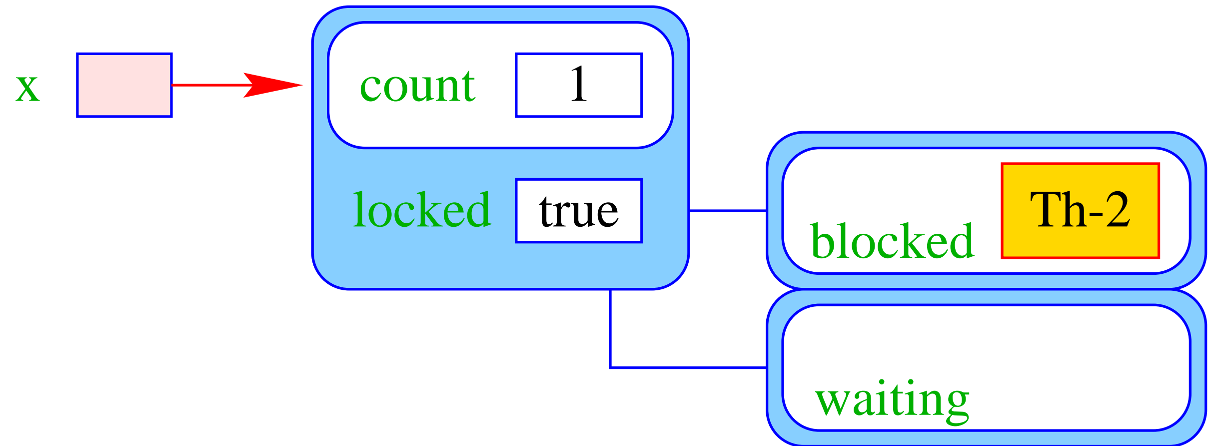
- ... mit der Auflage, erneut das Lock zu erwerben, d.h. als erste Operation hinter dem `wait();` ein `lock()` auszuführen.
- `notifyAll();` weckt alle wartenden Threads auf:

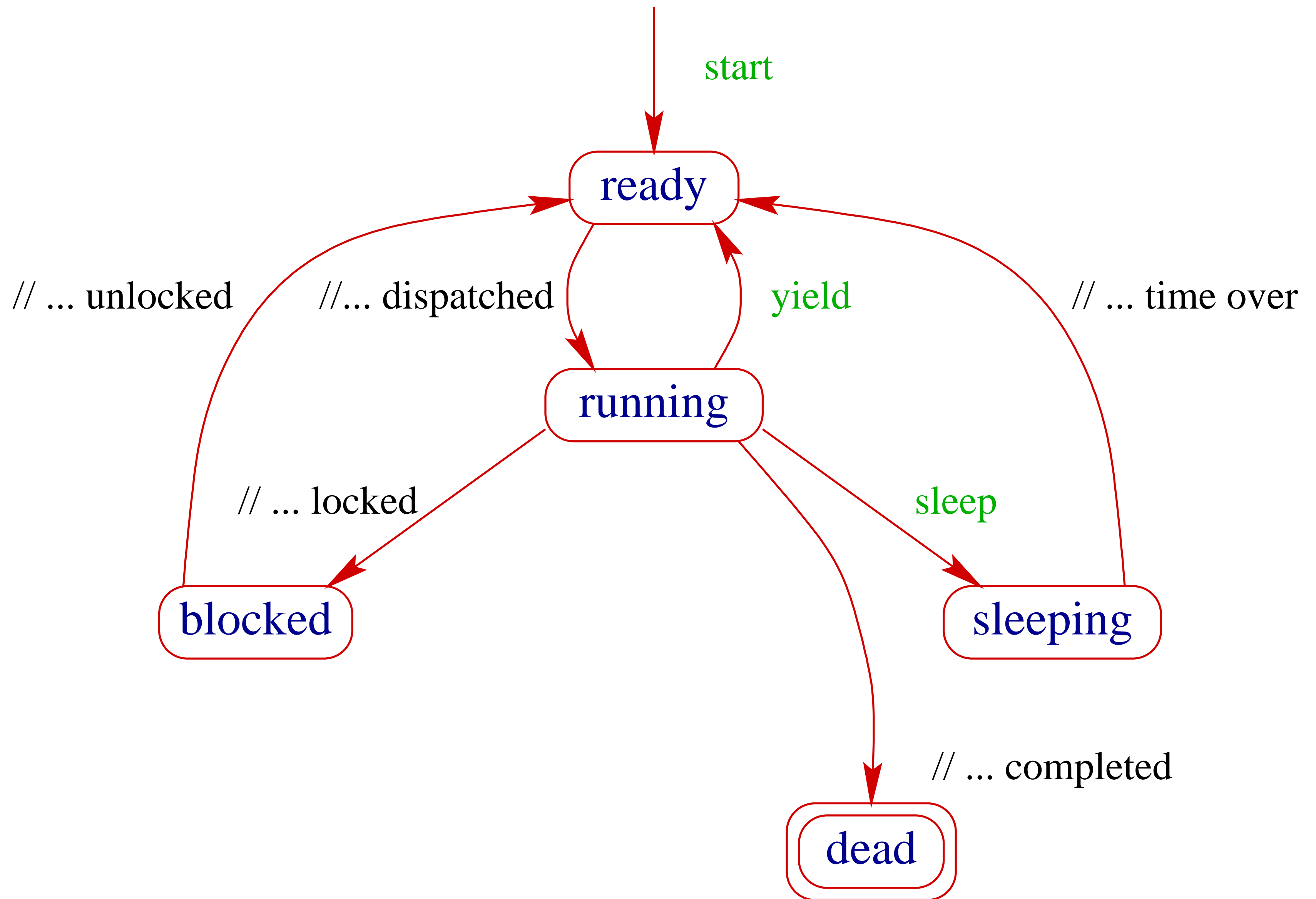
```
public void notifyAll() {
    while (!waitingThreads.isEmpty()) notify();
}
```

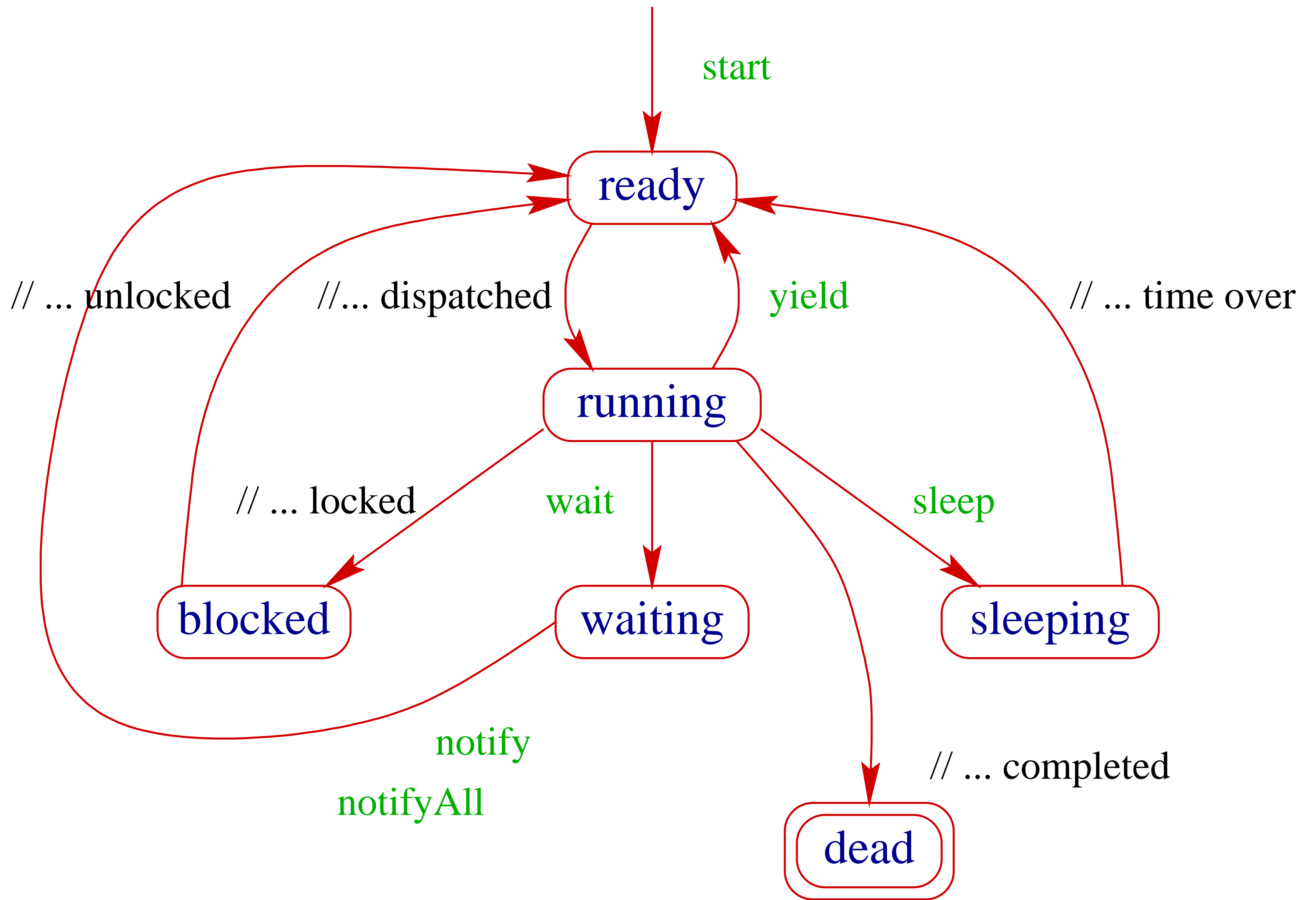

Th-0
Th-1
x
notify()
count
1
locked
true
blocked
waiting
Th-2


Th-0
Th-1
Th-2
x
count
1
locked
true
blocked
waiting
lock()


Th-0
Th-1


x
count
1
locked
true
blocked
Th-2
waiting


start
ready
// ... unlocked
//... dispatched
yield
// ... time over
running
// ... locked
sleep
blocked
sleeping
// ... completed
dead


start
ready
// ... unlocked
//... dispatched
yield
// ... time over
running
// ... locked
wait
sleep
blocked
waiting
sleeping
notify
notifyAll
// ... completed
dead

Anwendung:

```
  ...
  public synchronized void produce(Data d) throws InterruptedException {
     if (free==0) wait(); free--;
     a[last] = d;
     last = (last+1)%cap;
     notify();
  }
  public synchronized Data consume() throws InterruptedException {
     if (free==cap) wait(); free++;
     Data result = a[first];
     first = (first+1)%cap;
     notify(); return result;
  }
} // end of class Buffer2
```

- Ist der Puffer voll, d.h. keine Zelle frei, legt sich der Producer schlafen.
- Ist der Puffer leer, d.h. alle Zellen frei, legt sich der Consumer schlafen.
- Gibt es für einen Puffer genau einen Producer und einen Consumer, weckt das `notify()` des Consumers (wenn überhaupt, dann) stets den Producer ...

  ... und umgekehrt.
- Was aber, wenn es mehrere Producers gibt? Oder mehrere Consumers ???

2. Idee: Wiederholung der Tests

- Teste nach dem Aufwecken erneut, ob Zellen frei sind.
- Wecke nicht einen, sondern alle wartenden Threads auf ...

```
...
public synchronized void produce(Data d)
                              throws InterruptedException {
   while (free==0) wait(); free--;
   a[last] = d;
   last = (last+1)%cap;
   notifyAll();
}
...
```

```
  ...
  public synchronized Data consume() throws InterruptedException {
     while (free==cap) wait();
     free++;
     Data result = a[first];
     first = (first+1)%cap;
     notifyAll();
     return result;
  }
} // end of class Buffer2
```

- Wenn ein Platz im Puffer frei wird, werden sämtliche Threads aufgeweckt – obwohl evt. nur einer der Producer bzw. nur einer der Consumer aktiv werden kann :-(

3. Idee: Semaphore

- Producers und Consumers warten in verschiedenen Schlangen.
- Die Producers warten darauf, dass `free > 0` ist.
- Die Consumers warten darauf, dass `cap-free > 0` ist.

3. Idee: Semaphore

- Producers und Consumers warten in verschiedenen Schlangen.
- Die Producers warten darauf, dass `free > 0` ist.
- Die Consumers warten darauf, dass `cap-free > 0` ist.

```
public class Sema { private int x;
   public Sema(int n) { x = n; }
   public synchronized void up() {
      x++; if (x<=0) notify();
   }
   public synchronized void down() throws InterruptedException {
      x--; if (x<0) wait();
   }
}  // end of class Sema
```

- Ein Semaphor enthält eine private `int`-Objekt-Variable und bietet die `synchronized`-Methoden `up()` und `down()` an.
- `up()` erhöht die Variable, `down()` erniedrigt sie.
- Ist die Variable positiv, gibt sie die Anzahl der verfügbaren Resourcen an.

  Ist sie negativ, zählt sie die Anzahl der wartenden Threads.
- Eine `up()`-Operation weckt genau einen wartenden Thread auf.

Anwendung (1. Versuch :-)

```
public class Buffer {
    private int cap, first, last;
    private Sema free, occupied;
    private Data[] a;
    public Buffer(int n) {
        cap = n; first = last = 0;
        a = new Data[n];
        free = new Sema(n);
        occupied = new Sema(0);
    }
    ...
```

```
  ...
  public synchronized void produce(Data d) throws InterruptedException {
    free.down();
    a[last] = d;
    last = (last+1)%cap;
    occupied.up();
  }
  public synchronized Data consume() throws InterruptedException {
    occupied.down();
    Data result = a[first];
    first = (first+1)%cap;
    free.up();
    return result;
  }
}  // end of faulty class Buffer
```

- Gut gemeint – aber leider fehlerhaft ...
- Jeder Producer benötigt zwei Locks gleichzeitig, um zu produzieren:

  1. dasjenige für den Puffer;
  2. dasjenige für einen Semaphor.

- Muss er für den Semaphor ein `wait()` ausführen, gibt er das Lock für den Semaphor wieder zurück ... nicht aber dasjenige für den Puffer !!!
- Die Folge ist, dass niemand mehr eine Puffer-Operation ausführen kann, insbesondere auch kein `up()` mehr für den Semaphor $\Longrightarrow$ Deadlock

- Gut gemeint – aber leider fehlerhaft ...
- Jeder Producer benötigt zwei Locks gleichzeitig, um zu produzieren:
  1. dasjenige für den Puffer;
  2. dasjenige für einen Semaphor.
- Muss er für den Semaphor ein `wait()` ausführen, gibt er das Lock für den Semaphor wieder zurück ... nicht aber dasjenige für den Puffer !!!
- Die Folge ist, dass niemand mehr eine Puffer-Operation ausführen kann, insbesondere auch kein `up()` mehr für den Semaphor $\Longrightarrow$ Deadlock

Anwendung (2. Versuch :-) Entkopplung der Locks

```
   ...
   public void produce(Data d) throws InterruptedException {
      free.down();
      synchronized (this) {
         a[last] = d; last = (last+1)%cap;
      }
      occupied.up();
   }
   public Data consume() throws InterruptedException {
      occupied.down();
      synchronized (this) {
         Data result = a[first]; first = (first+1)%cap;
      }
      free.up(); return result;
   }
}  // end of corrected class Buffer
```