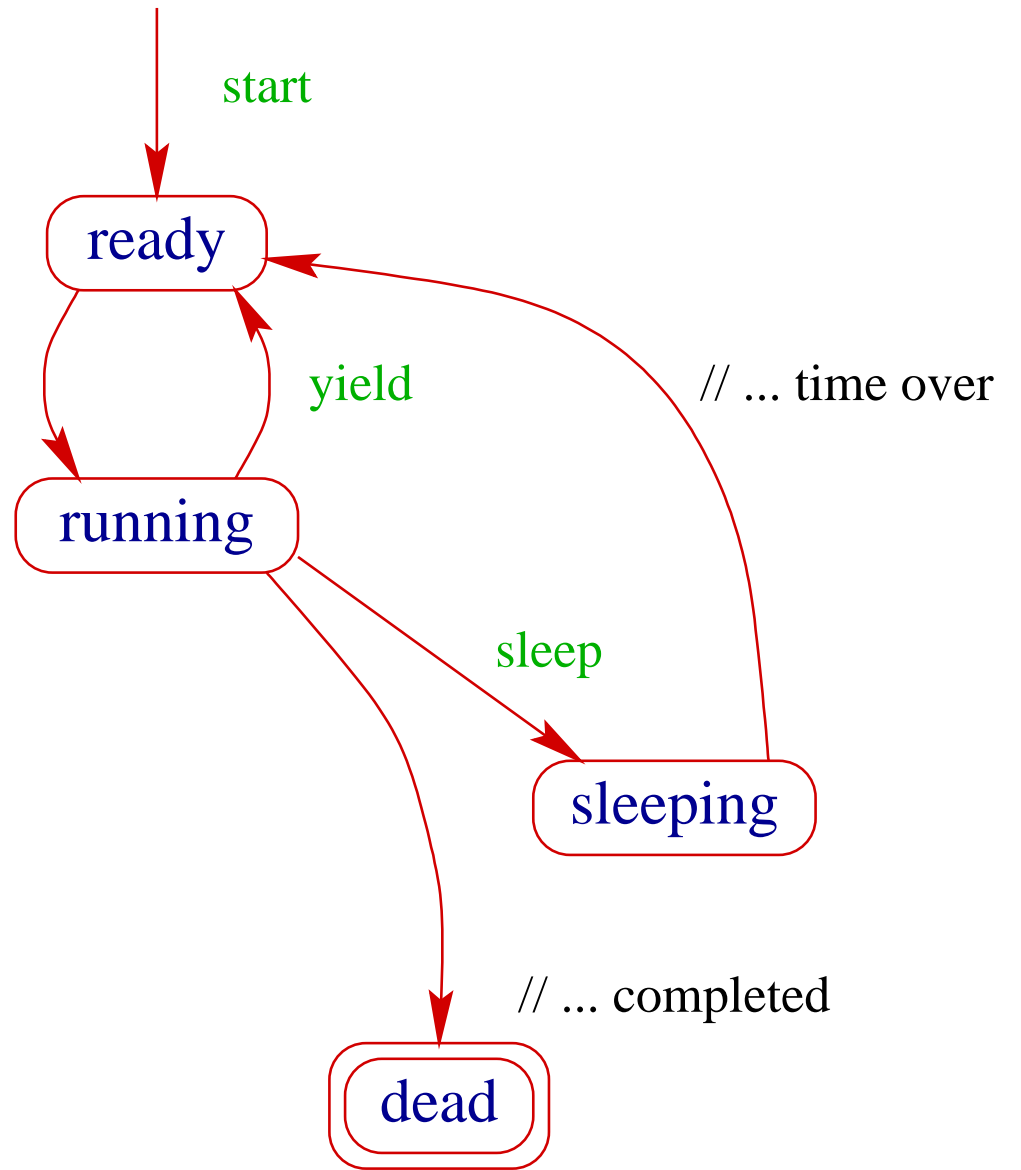
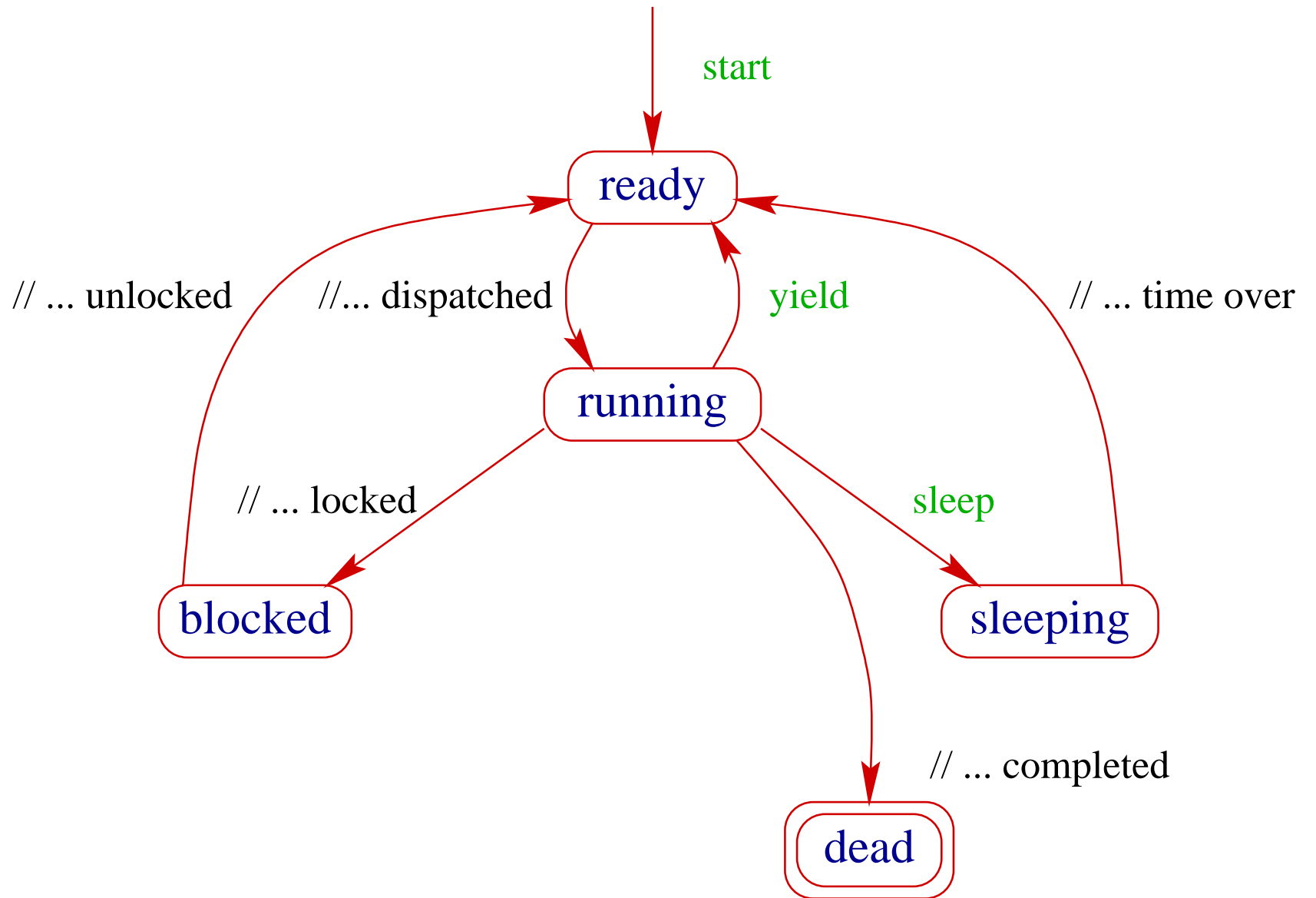


## Idee:

- Inkrementieren der Variable  $x$  sollte ein **atomarer Schritt** sein, d.h. nicht von parallel laufenden Threads unterbrochen werden können.
- Mithilfe des Schlüsselworts `synchronized` kennzeichnen wir Objekt-Methoden einer Klasse  $L$  als ununterbrechbar.
- Für jedes Objekt `obj` der Klasse  $L$  kann zu jedem Zeitpunkt nur ein Aufruf `obj.synchMeth(...)` einer `synchronized`-Methode `synchMeth()` ausgeführt werden. Die Ausführung einer solchen Methode nennt man **kritischen Abschnitt** (“critical section”) für die gemeinsame Resource `obj`.
- Wollen mehrere Threads gleichzeitig in ihren kritischen Abschnitt für das Objekt `obj` eintreten, werden alle bis auf einen **blockiert**.





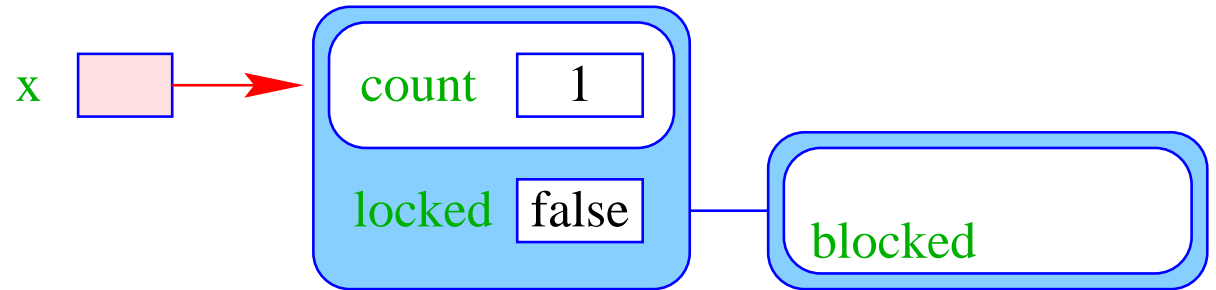
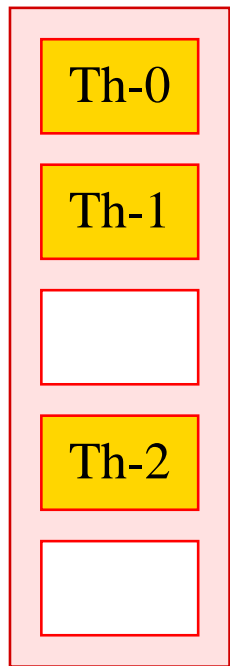
- Jedes Objekt `obj` mit `synchronized`-Methoden verfügt über:
  1. über ein boolesches Flag `boolean locked;` sowie
  2. über eine Warteschlange `ThreadQueue blockedThreads.`
- Vor Betreten seines kritischen Abschnitts führt ein Thread **(implizit)** die **atomare** Operation `obj.lock()` aus:

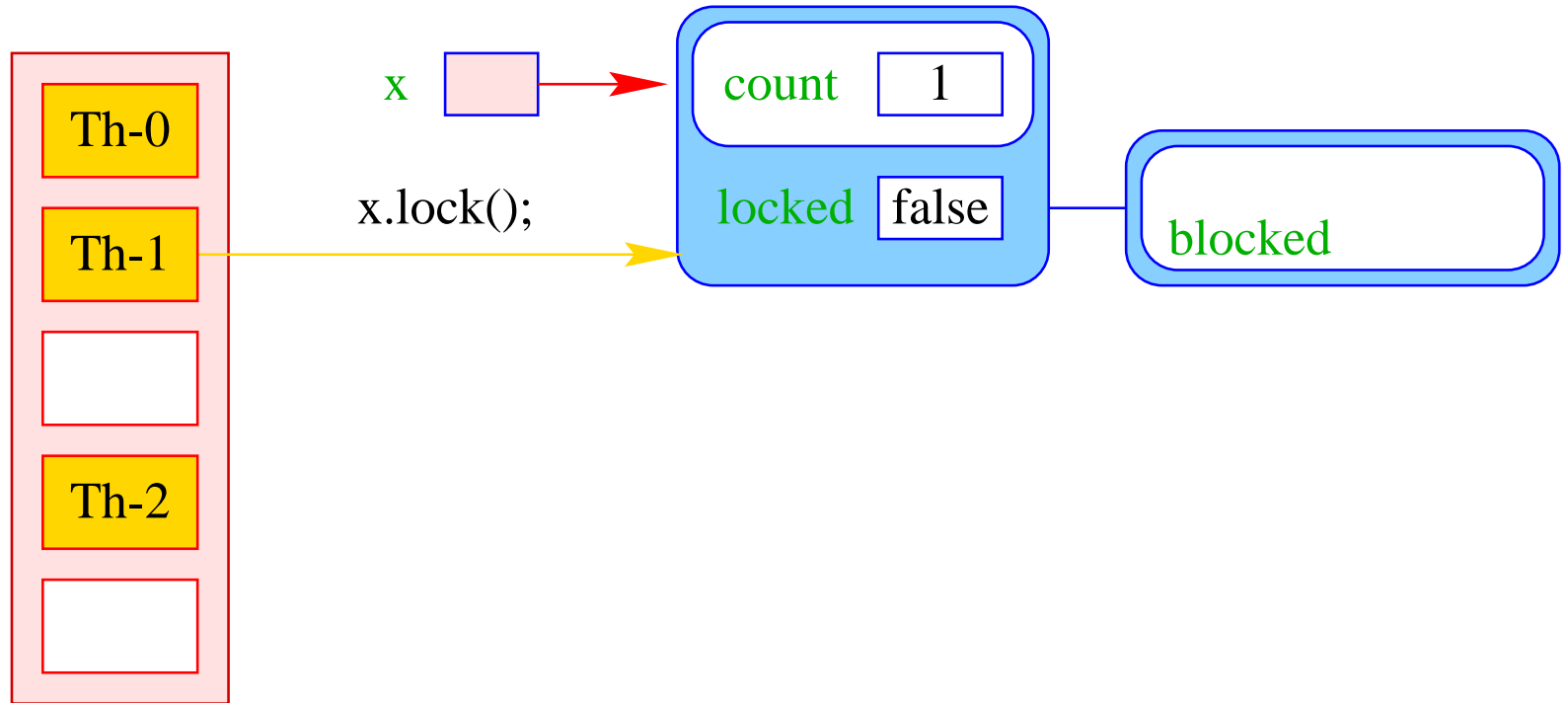
```
private void lock() {  
    if (!locked) locked = true; // betrete krit. Abschnitt  
    else { // Lock bereits vergeben  
        Thread t = Thread.currentThread();  
        blockedThreads.enqueue(t);  
        t.state = blocked; // blockiere  
    }  
} // end of lock()
```

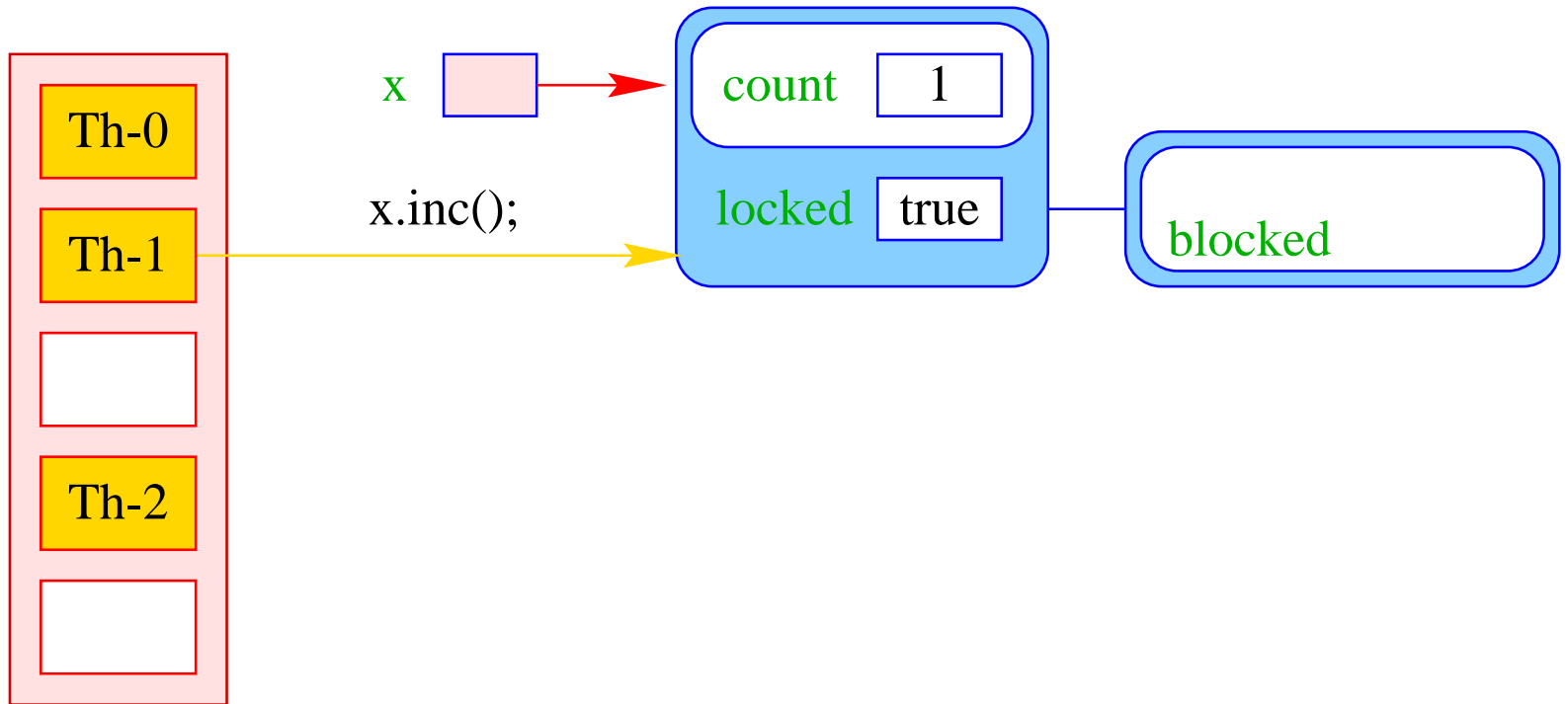
- Verlässt ein Thread seinen kritischen Abschnitt für **obj** (evt. auch mittels einer Exception :-), führt er (implizit) die atomare Operation **obj.unlock()** aus:

```
private void unlock() {
    if (blockedThreads.empty())
        locked = false; // Lock frei geben
    else {                // Lock weiterreichen
        Thread t = blockedThreads.dequeue();
        t.state = ready;
    }
} // end of unlock()
```

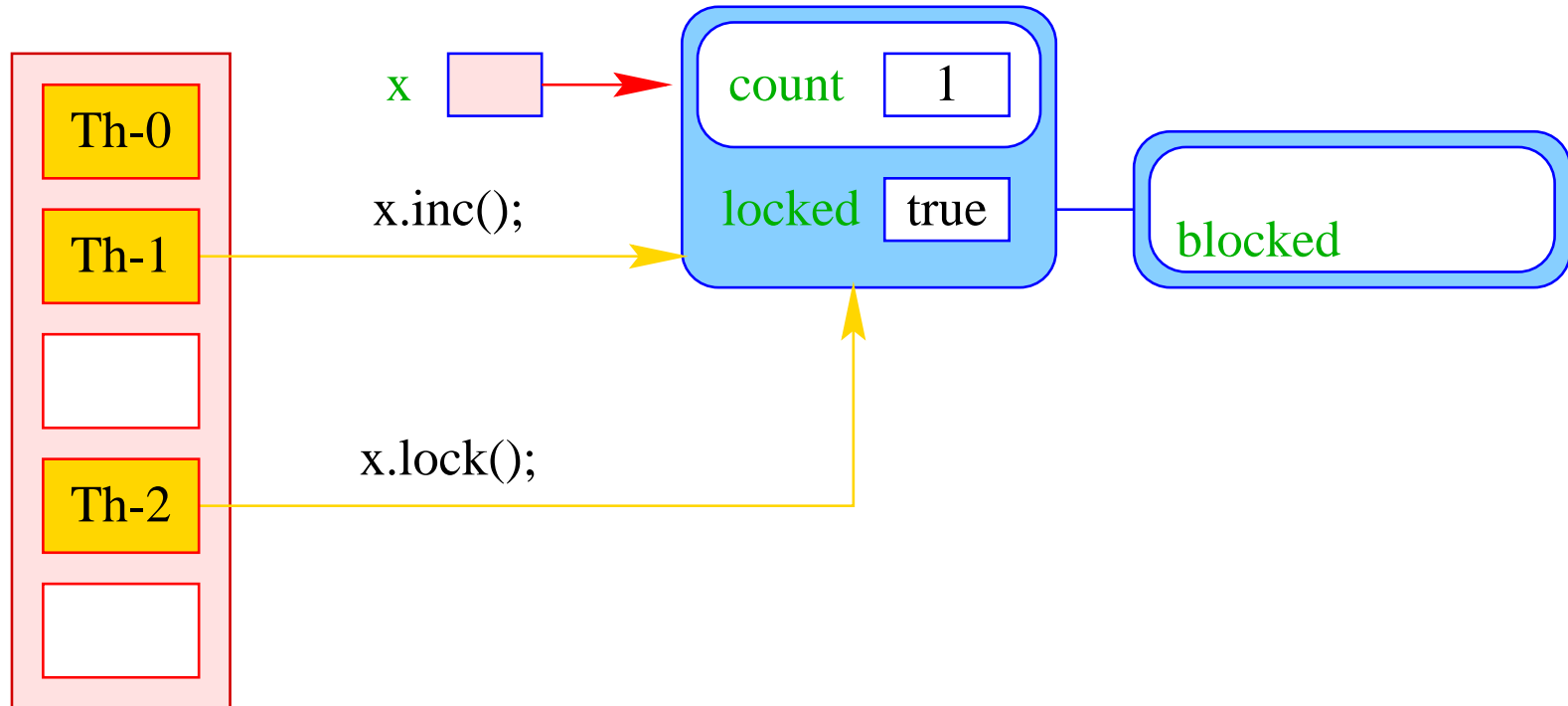
- Dieses Konzept nennt man **Monitor**.

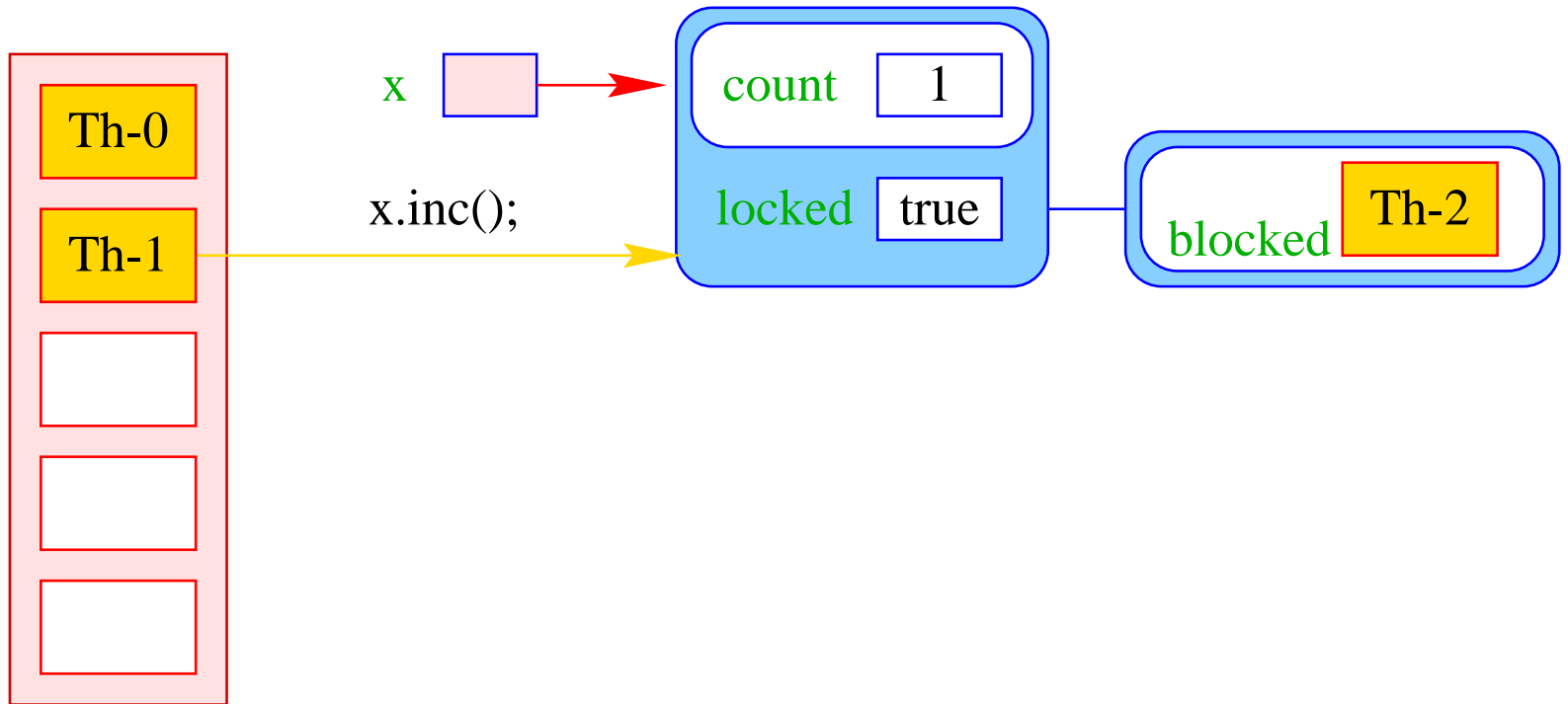


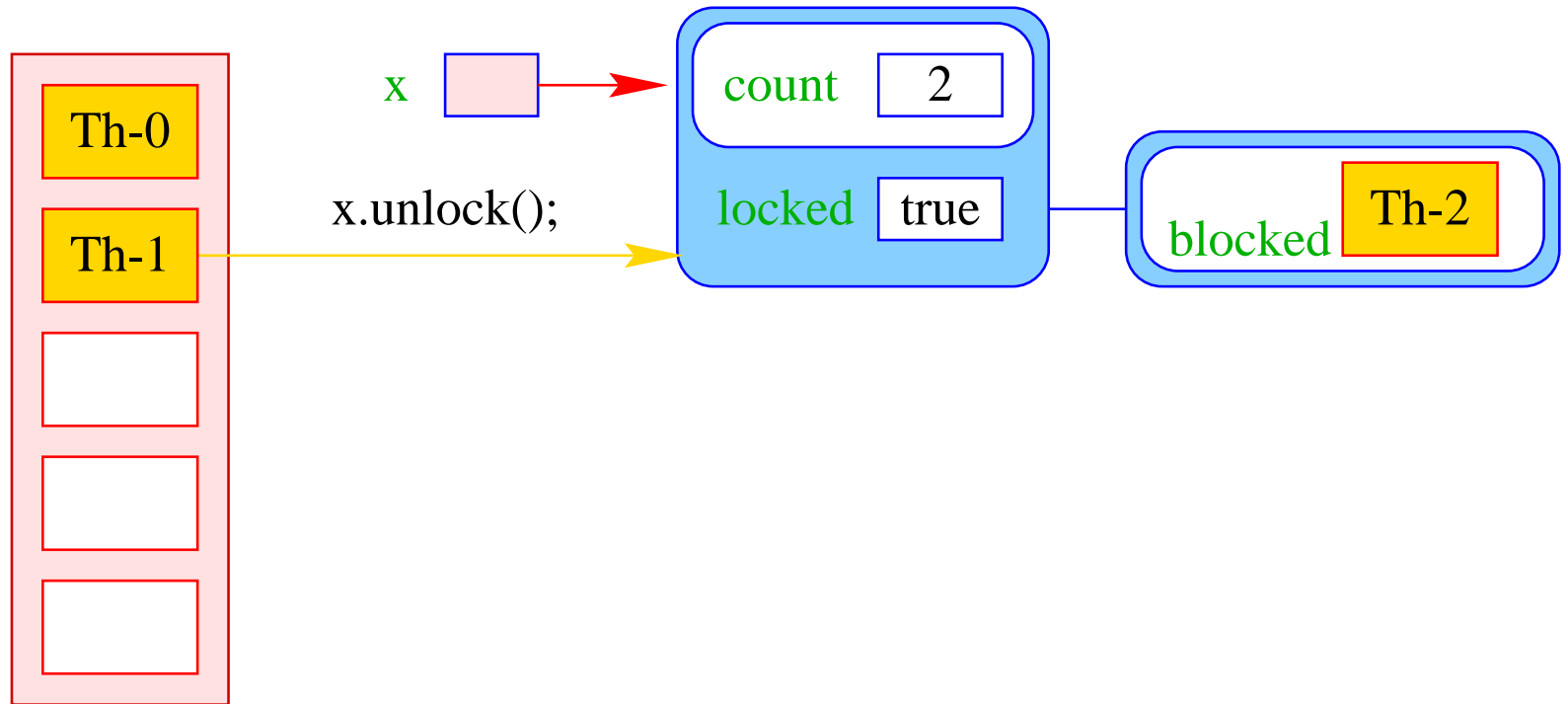


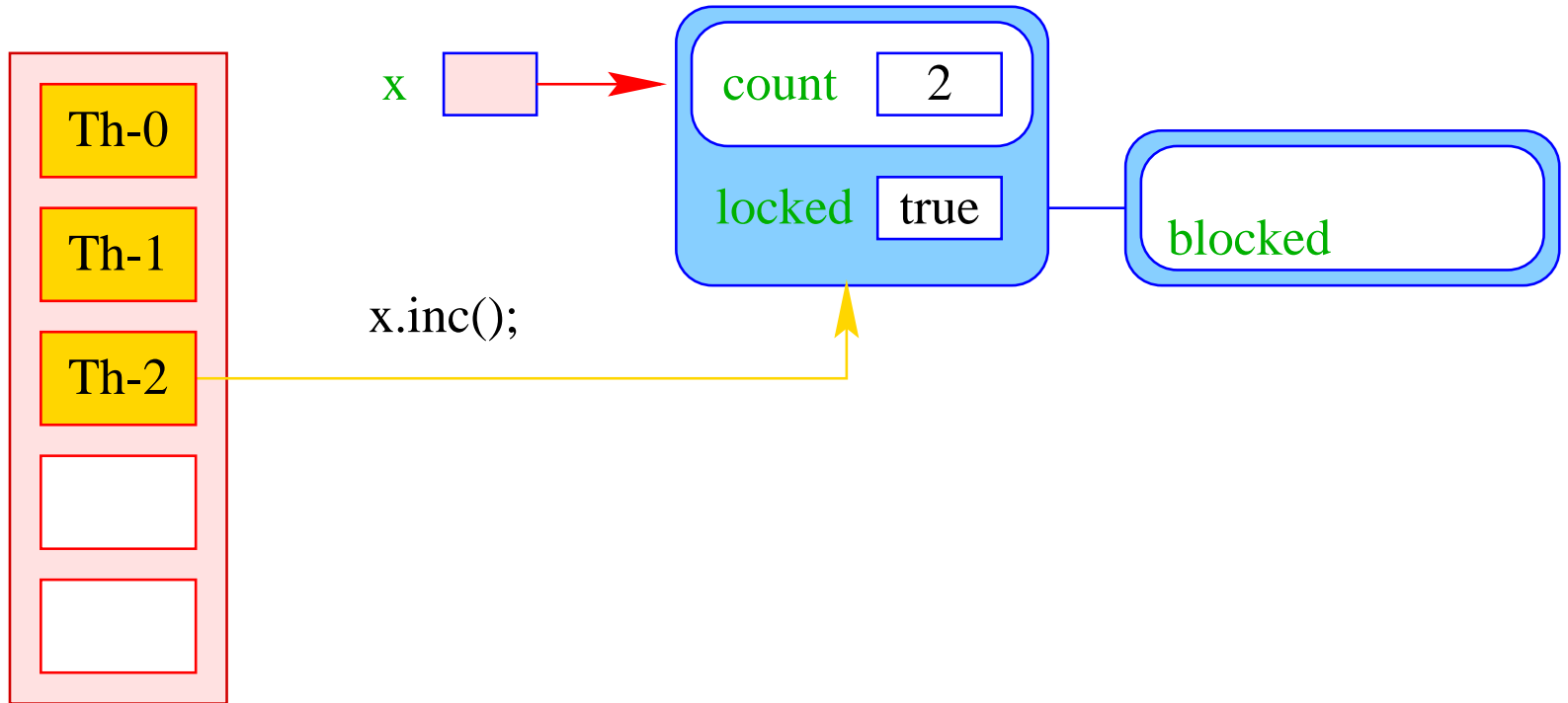


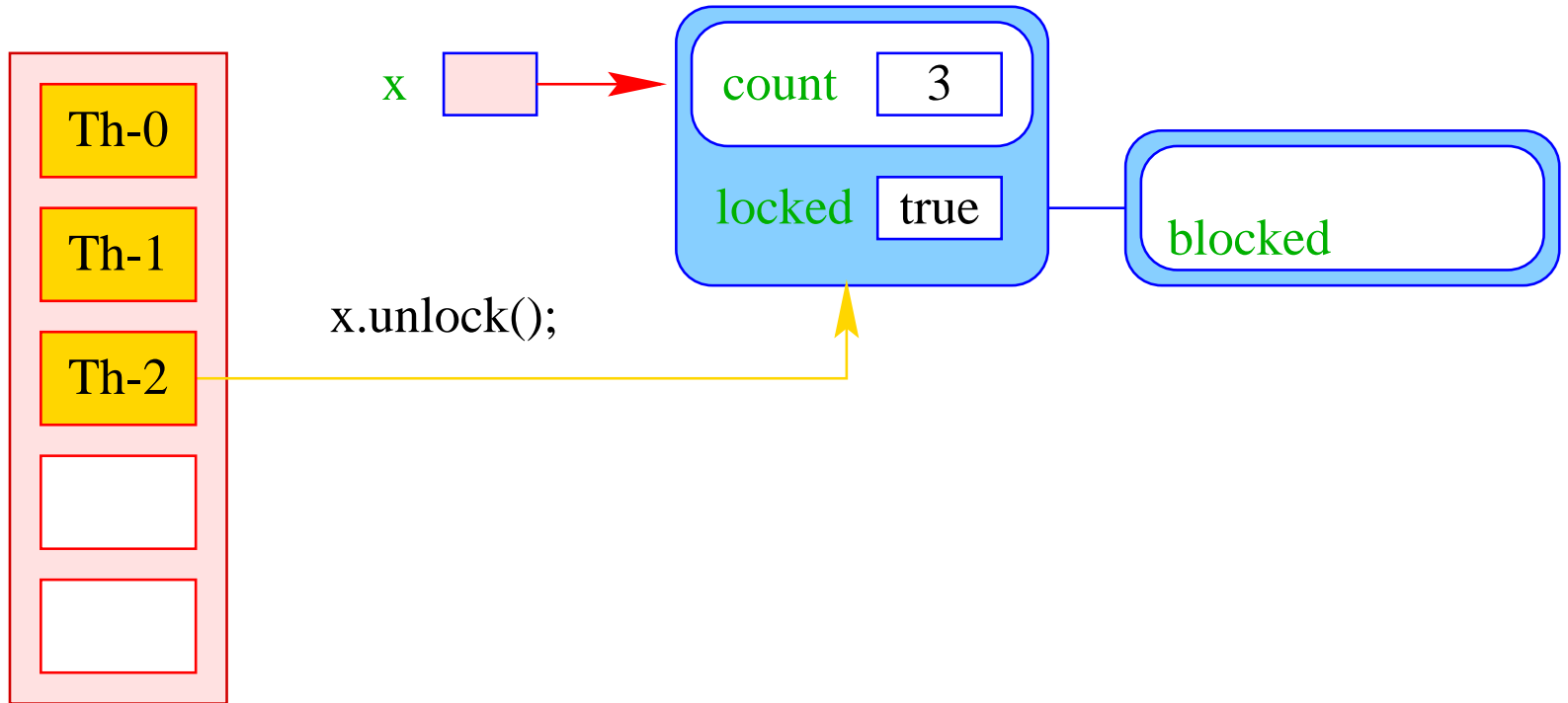


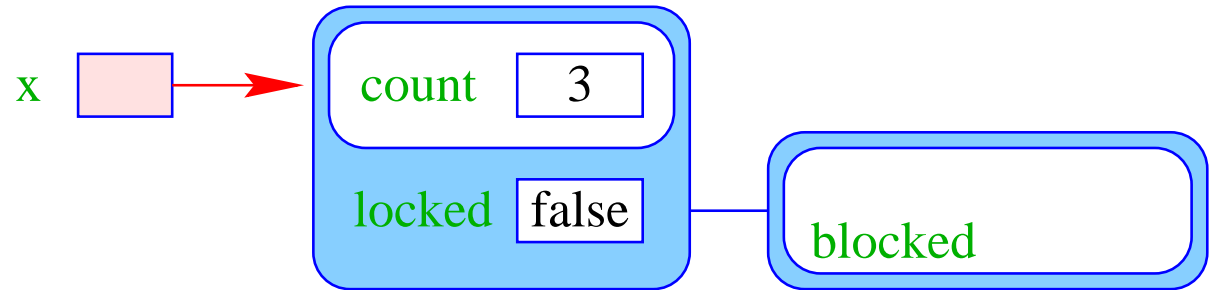
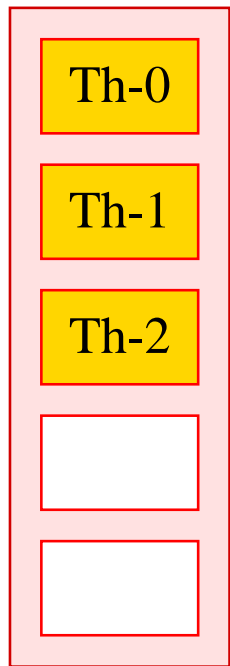












```

public class Count {
    private int x = 0;
    public synchronized void inc() {
        String s = Thread.currentThread().getName();
        int y = x;    System.out.println(s+ " read "+y);
        x = y+1;     System.out.println(s+ " wrote "+(y+1));
    }
} // end of class Count

public class IncSync implements Runnable {
    private static Count x = new Count();
    public void run() { x.inc(); }
    public static void main(String[] args) {
        (new Thread(new IncSynch())).start();
        (new Thread(new IncSynch())).start();
        (new Thread(new IncSynch())).start();
    }
} // end of class IncSync

```

... liefert:

```
> java IncSync
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-1 wrote 2
Thread-2 read 2
Thread-2 wrote 3
```

## Achtung:

- Die Operationen `lock()` und `unlock()` erfolgen nur, wenn der Thread nicht bereits **vorher** das Lock des Objekts erworben hat.
- Ein Thread, der das Lock eines Objekts **obj** besitzt, kann **weitere** Methoden für **obj** aufrufen, ohne sich selbst zu blockieren **:-)**



- Um das zu garantieren, legt ein Thread für jedes Objekt `obj`, dessen Lock er nicht besitzt, aber erwerben will, einen neuen Zähler an:

```
int countLock[obj] = 0;
```

- Bei jedem Aufruf einer `synchronized`-Methode `m(...)` für `obj` wird der Zähler inkrementiert, für jedes Verlassen (auch mittels Exceptions :-)) dekrementiert:

```
if (0 == countLock[obj]++) lock();
```

```
Ausführung von obj.m(...)
```

```
if (--countLock[obj] == 0) unlock();
```

- `lock()` und `unlock()` werden nur ausgeführt, wenn

```
(countLock[obj] == 0)
```

## Andere Gründe für Blockierung:

- Warten auf Beendigung einer IO-Operation;
- `public final void join() throws InterruptedException`  
(eine Objekt-Methode der Klasse Thread) wartet auf die Beendigung eines anderen Threads...

... ein Beispiel:

```
public class Join implements Runnable {
    private static int count = 0;
    private int n = count++;
    private static Thread[] task = new Thread[3];
    public void run() {
        try {
            if (n>0) {
                task[n-1].join();
                System.out.println("Thread-"+n+" joined Thread-"+(n-1));
            }
        } catch (InterruptedException e) {
            System.err.println(e.toString());
        }
    }
    ...
}
```

```
...
public static void main(String[] args) {
    for(int i=0; i<3; i++)
        task[i] = new Thread(new Join());
    for(int i=0; i<3; i++)
        task[i].start();
}
} // end of class Join
```

... liefert:

```
> java Join
Thread-1 joined Thread-0
Thread-2 joined Thread-1
```

## Beachte:

- Threads, die auf Beendigung eines anderen Threads warten, gehen in einen Zustand `joining` über.
- Threads, die auf Beendigung einer IO-Operation warten, gehen in einen Zustand `joiningIO` über.
- Diese Zustände ähneln dem Zustand `blocked` für wechselseitigen Ausschluss von kritischen Abschnitten. Insbesondere gibt es
  - ... für jeden Thread `t` eine Schlange `ThreadQueue joiningThreads`;
  - ... analoge Warteschlangen für verschiedene IO-Operationen.

## Beachte:

- Threads, die auf Beendigung eines anderen Threads warten, gehen in einen Zustand `joining` über.
- Threads, die auf Beendigung einer IO-Operation warten, gehen in einen Zustand `joiningIO` über.
- Diese Zustände ähneln dem Zustand `blocked` für wechselseitigen Ausschluss von kritischen Abschnitten. Insbesondere gibt es
  - ... für jeden Thread `t` eine Schlange `ThreadQueue joiningThreads`;
  - ... analoge Warteschlangen für verschiedene IO-Operationen.

**Spaßeshalber** betrachten wir noch eine kleine Variation des letzten Programms:

```

public class CW implements Runnable {
    private static int count = 0;
    private int n = count++;
    private static Thread[] task = new Thread[3];
    public void run() {
        try { task[(n+1)%3].join(); }
        catch (InterruptedException e) {
            System.err.println(e.toString());
        }
    }
    public static void main(String[] args) {
        for(int i=0; i<3; i++)
            task[i] = new Thread(new CW());
        for(int i=0; i<3; i++) task[i].start();
    }
} // end of class CW

```

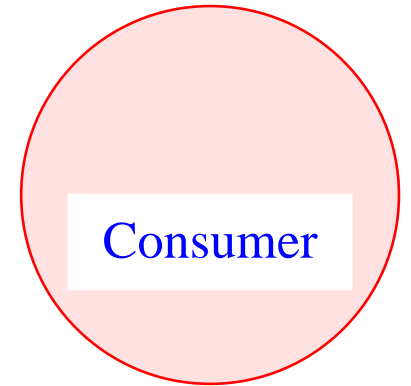
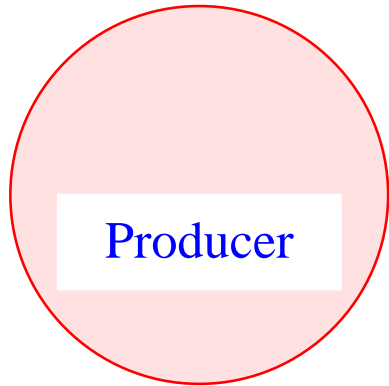
- Jeder Thread geht in einen Wartezustand (hier: **joining**) über und wartet auf einen anderen Thread.
- Dieses Phänomen heißt auch **Circular Wait** oder **Deadlock** – eine unangenehme Situation, die man in seinen Programmen tunlichst vermeiden sollte :-)

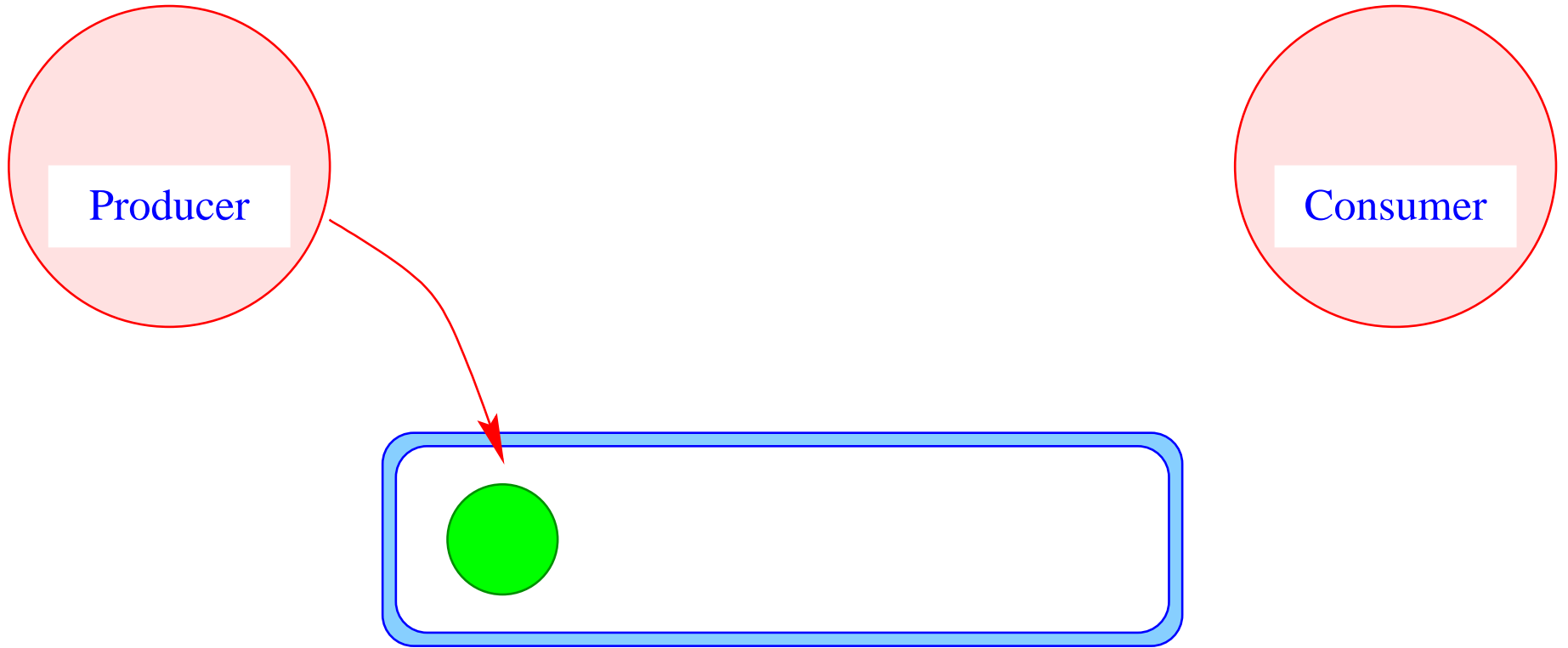


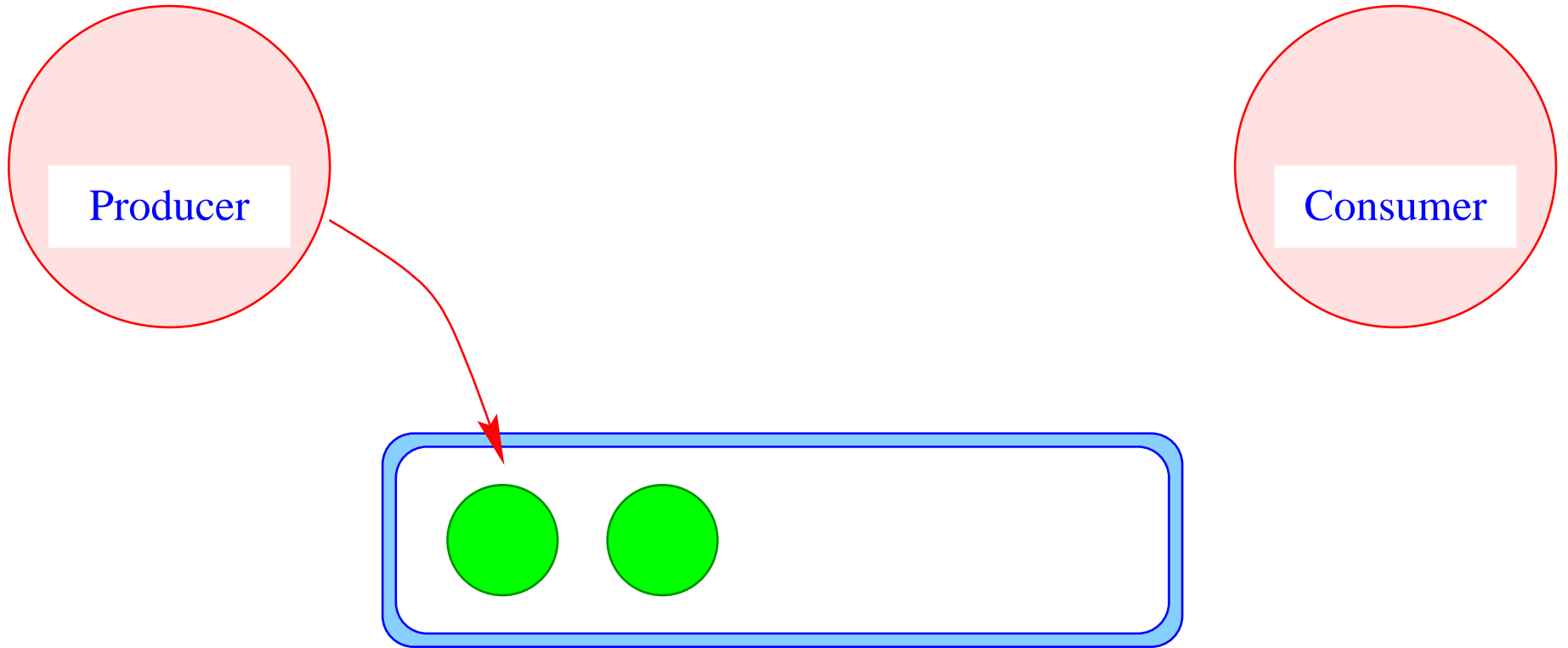
## 21.2 Semaphore und das Producer-Consumer-Problem

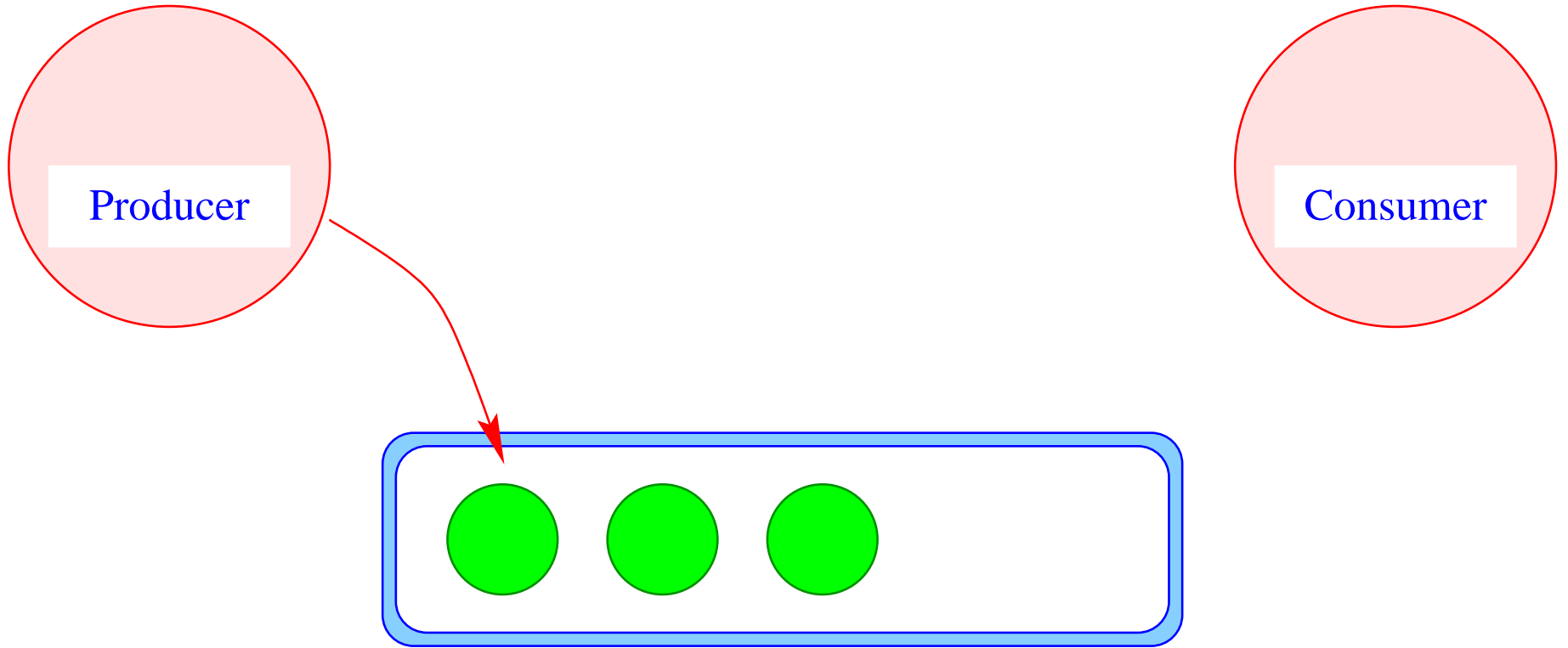
### Aufgabe:

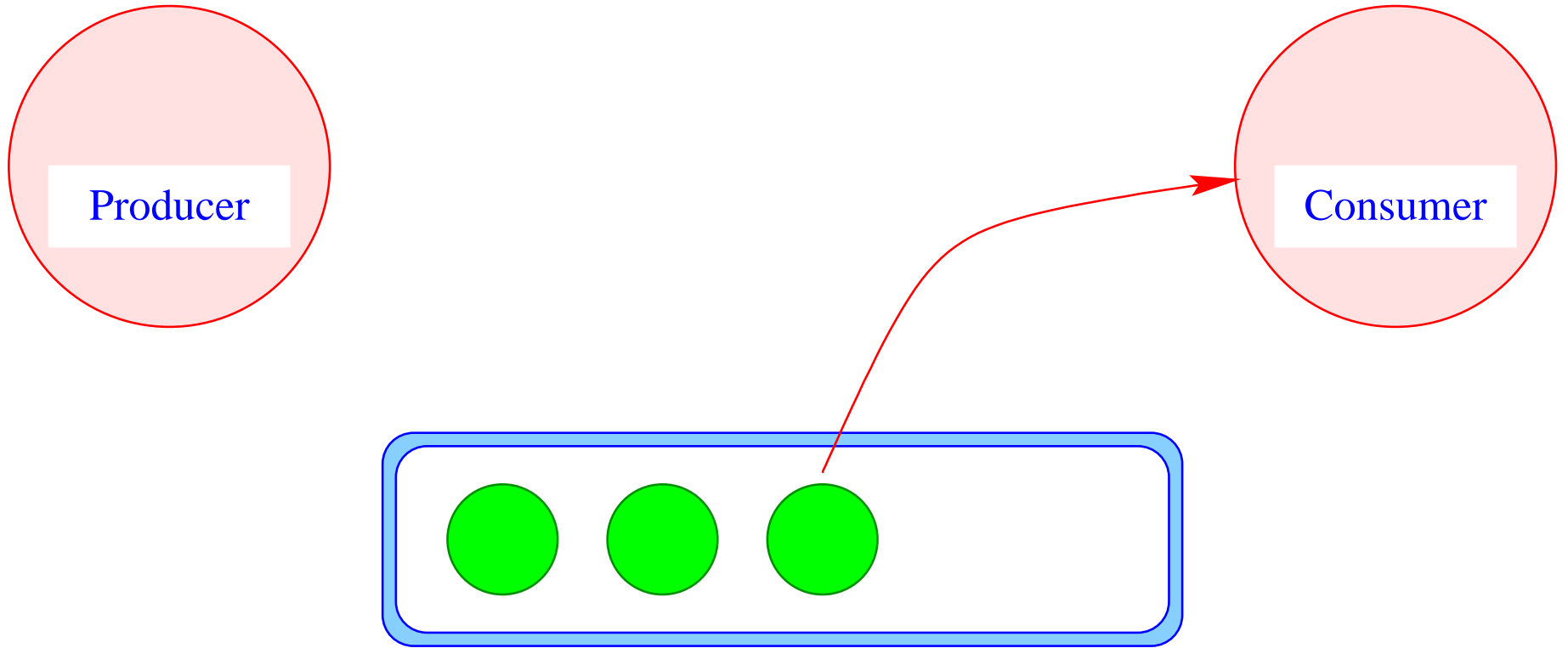
- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (**Producer**).
- Der andere konsumiert sie (**Consumer**).
- Zur Übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

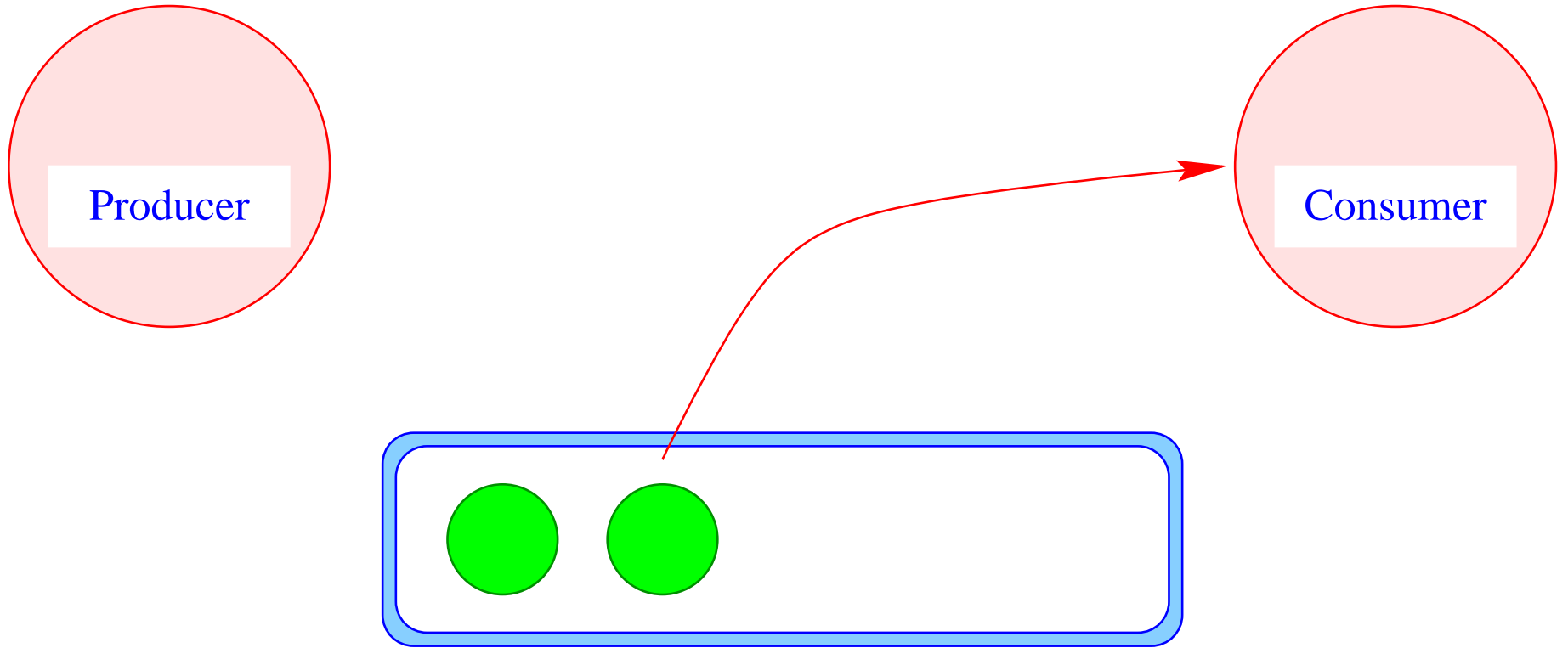












## 1. Idee:

- 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 ...

