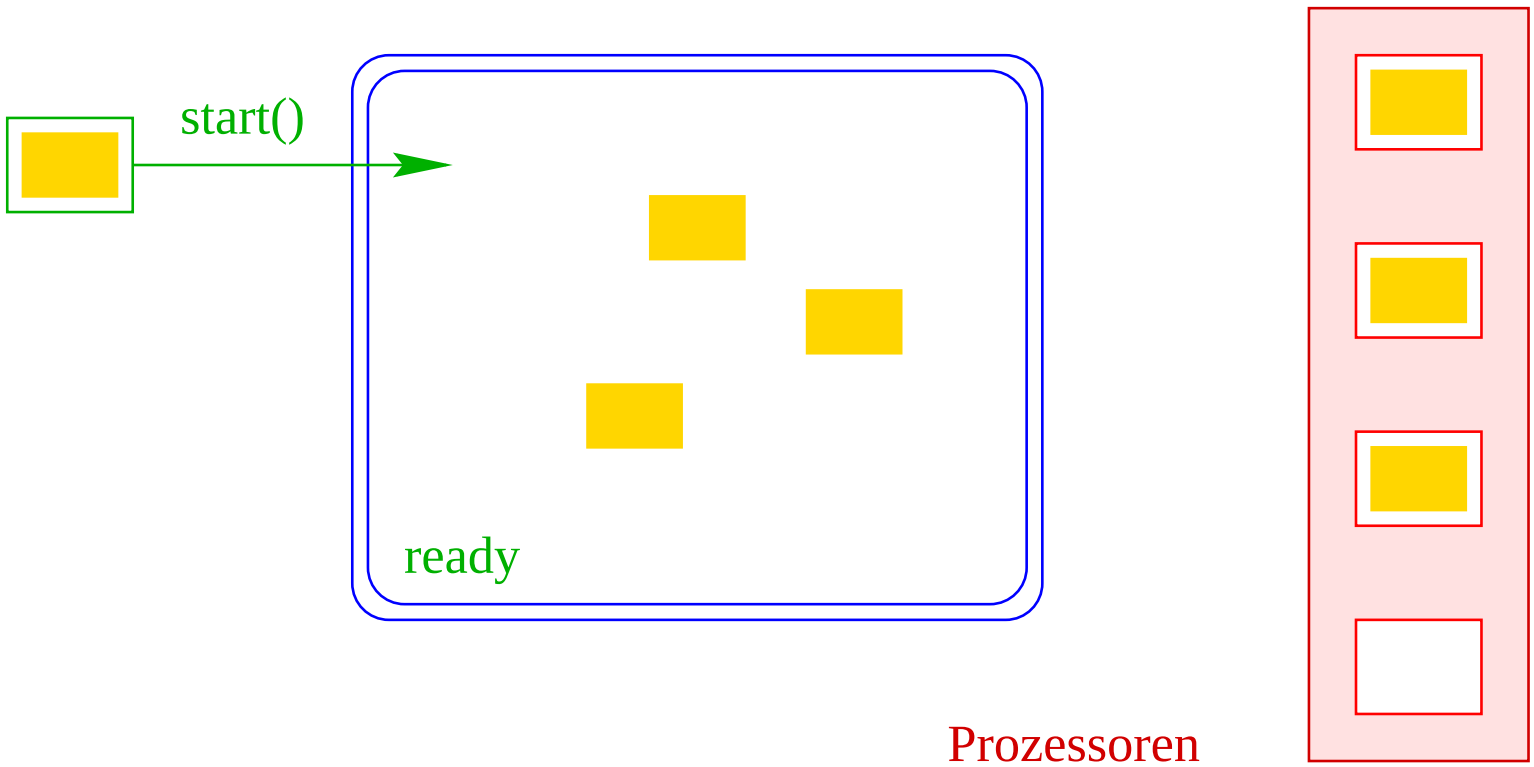
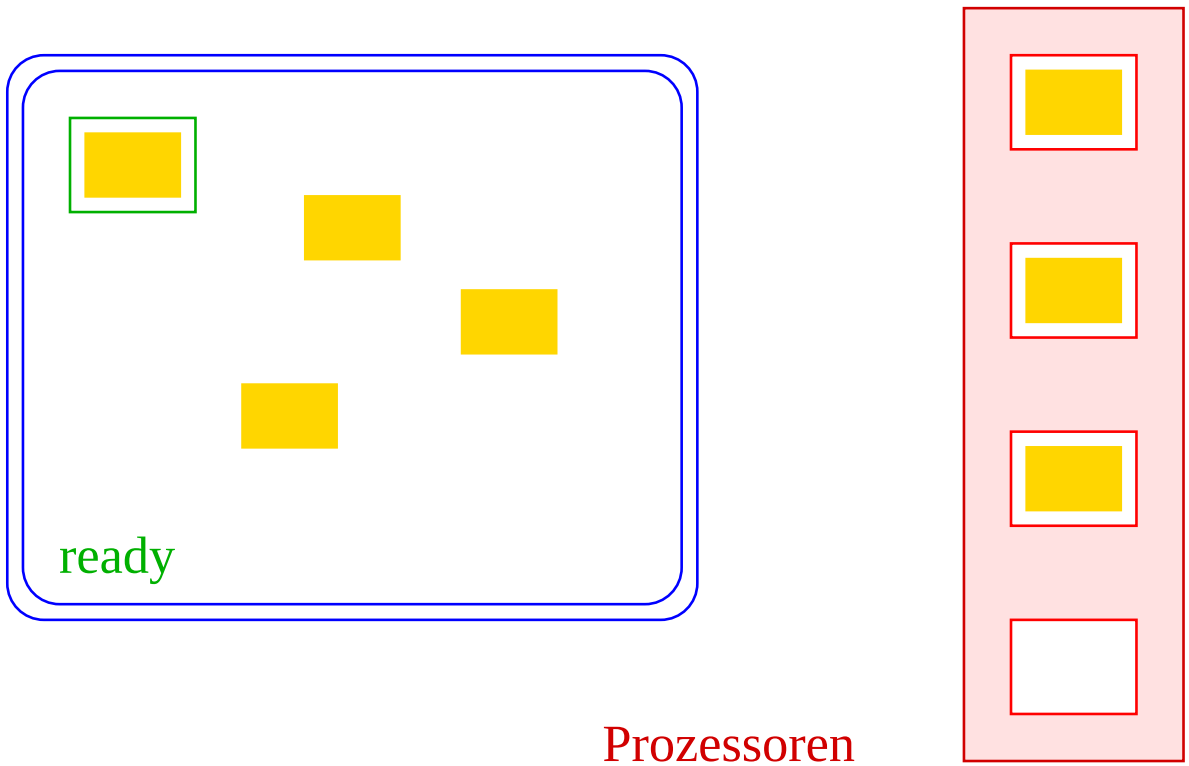
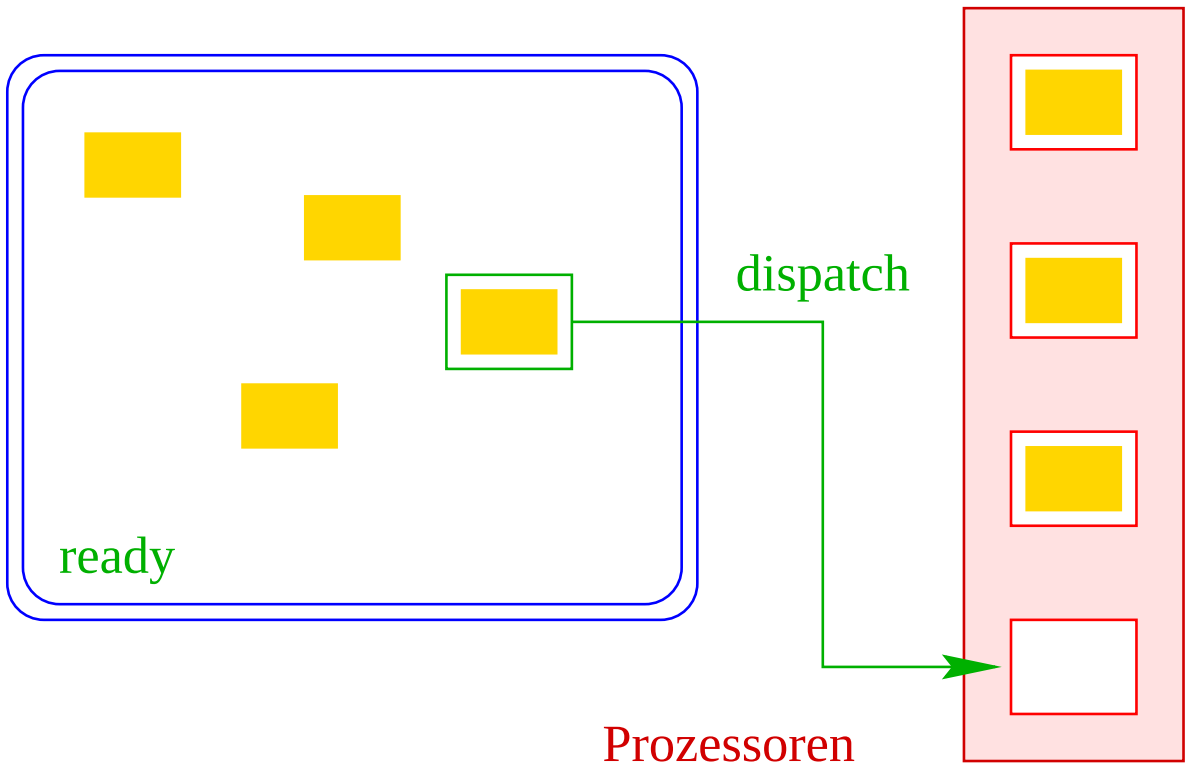
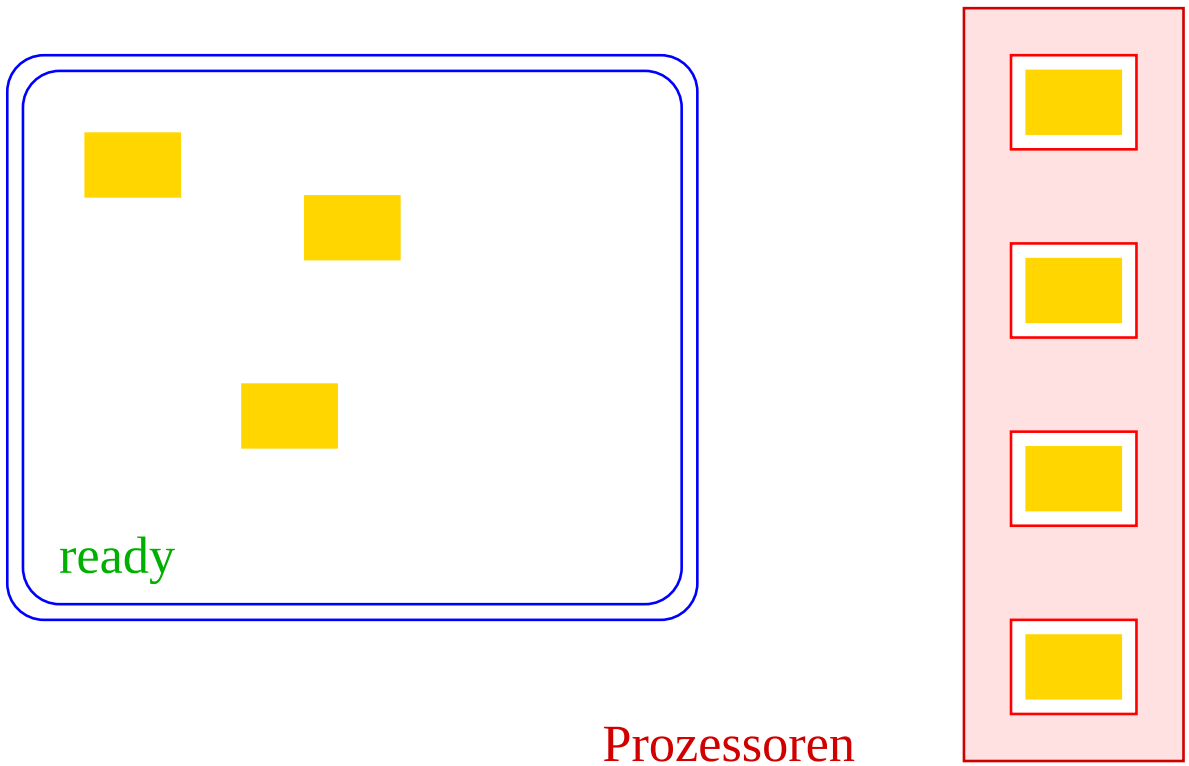
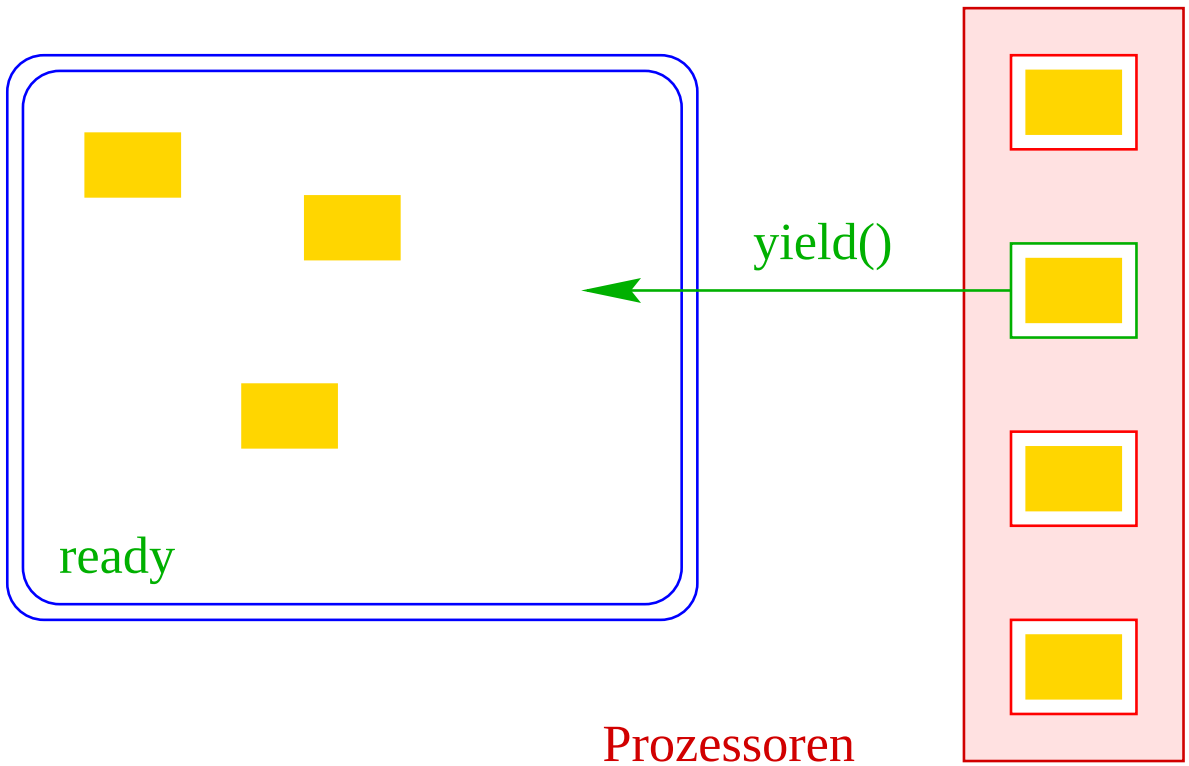
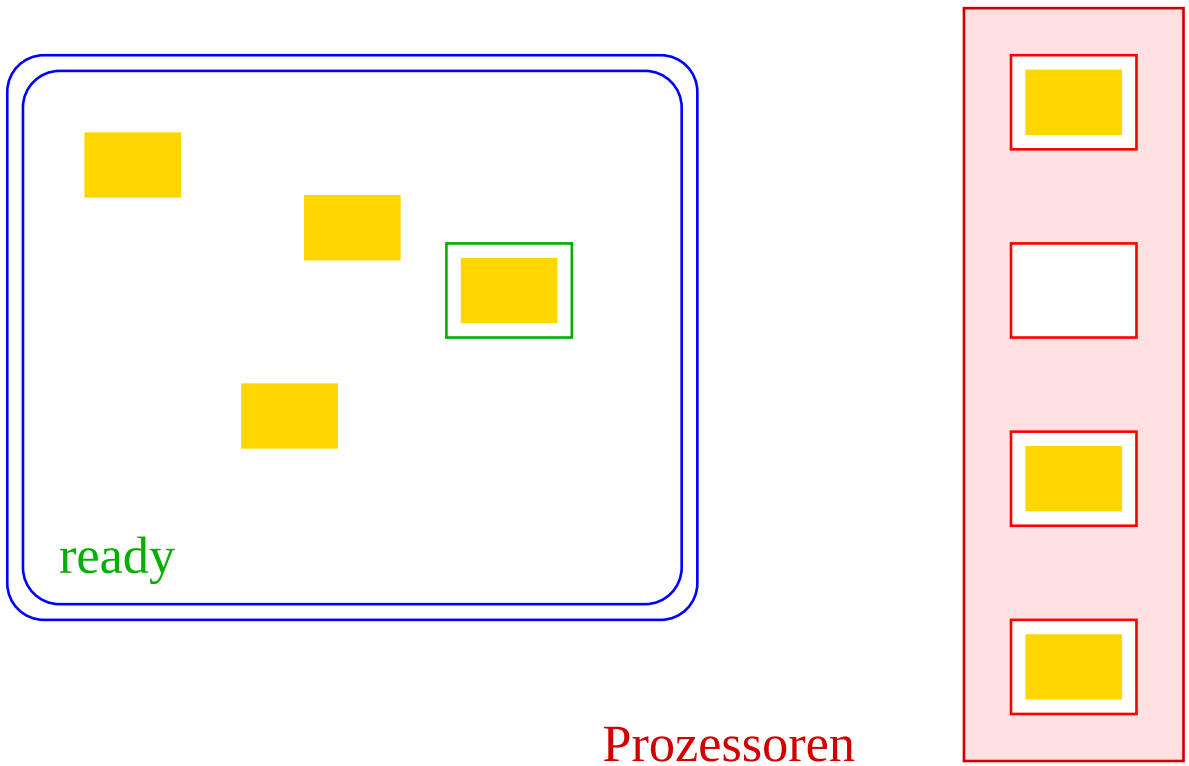

start()
ready
Prozessoren

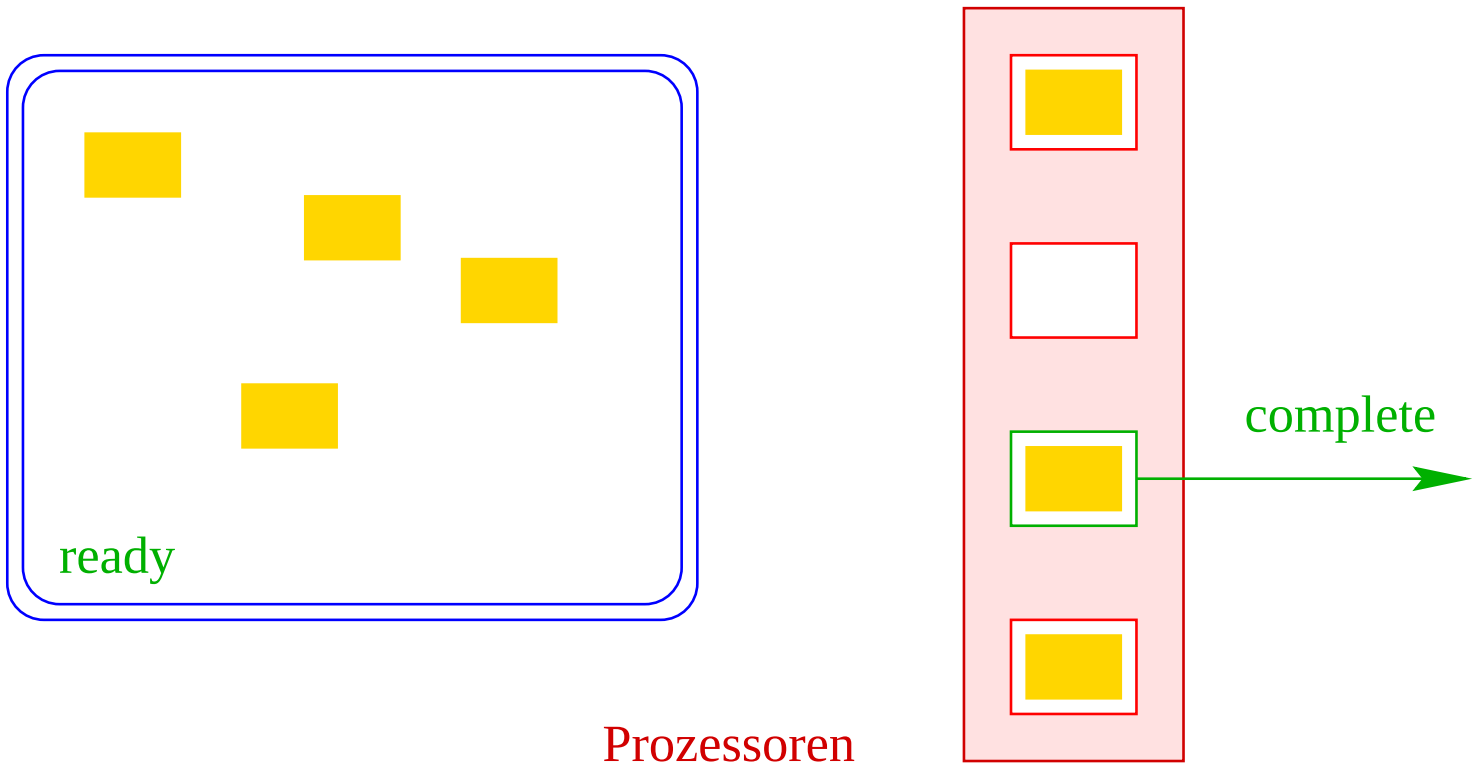

ready
Prozessoren

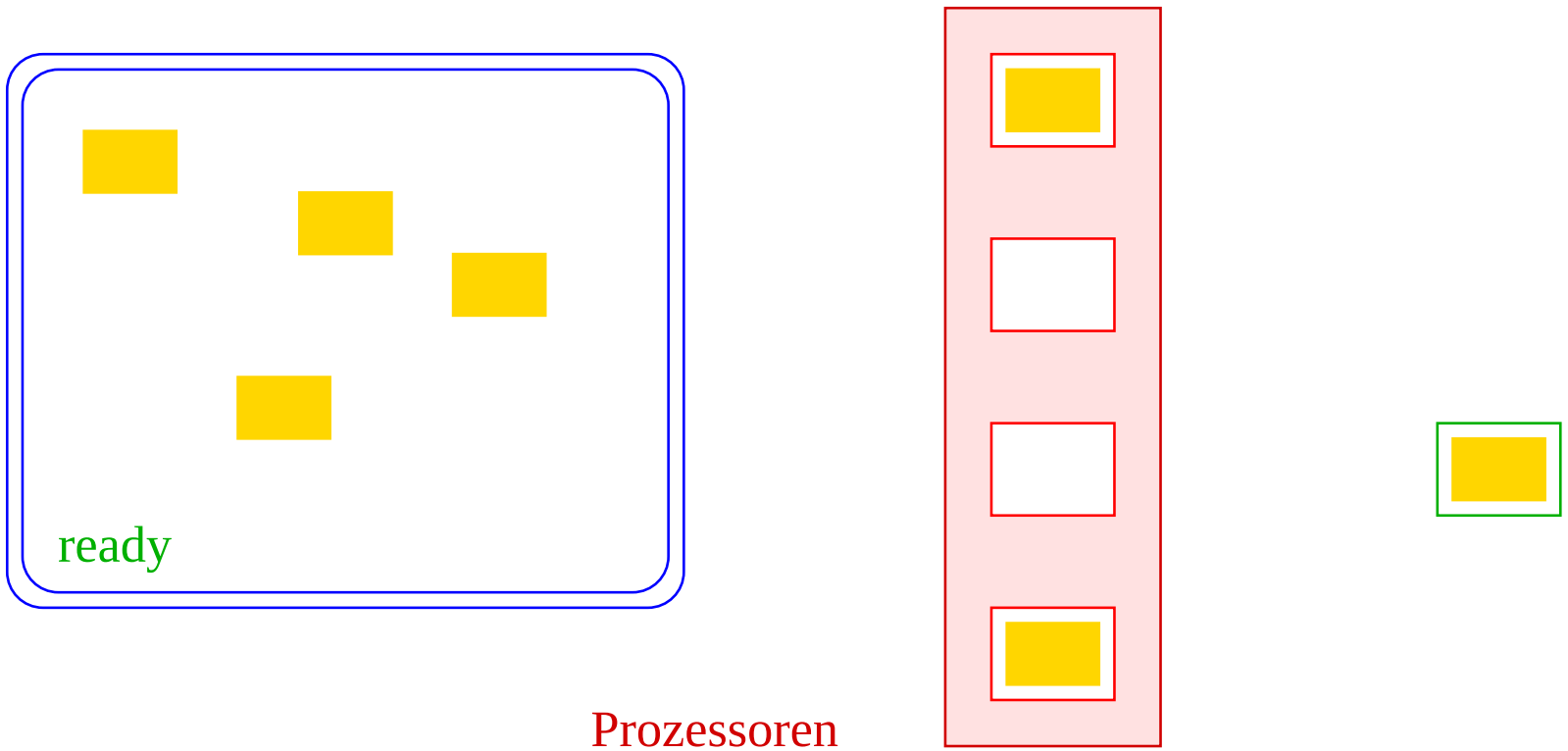

dispatch
ready
Prozessoren


ready
Prozessoren


yield()
ready
Prozessoren


ready
Prozessoren


ready
complete
Prozessoren


ready
Prozessoren

19.1 Monitore

- Damit Threads sinnvoll miteinander kooperieren können, müssen sie miteinander Daten austauschen.
- Zugriff mehrerer Threads auf eine gemeinsame Variable ist problematisch, weil nicht feststeht, in welcher Reihenfolge die Threads auf die Variable zugreifen.
- Ein Hilfsmittel, um geordnete Zugriffe zu garantieren, sind Monitore.

... ein Beispiel:

```
public class Inc implements Runnable {
   private static int x = 0;
   private static void pause(int t) {
      try {
         Thread.sleep((int) (Math.random()*t*1000));
      } catch (InterruptedException e) {
         System.err.println(e.toString());
      }
   }
   public void run() {
      String s = Thread.currentThread().getName();
      pause(3); int y = x;
      System.out.println(s+ " read "+y);
      pause(4); x = y+1;
      System.out.println(s+ " wrote "+(y+1));
   }
```

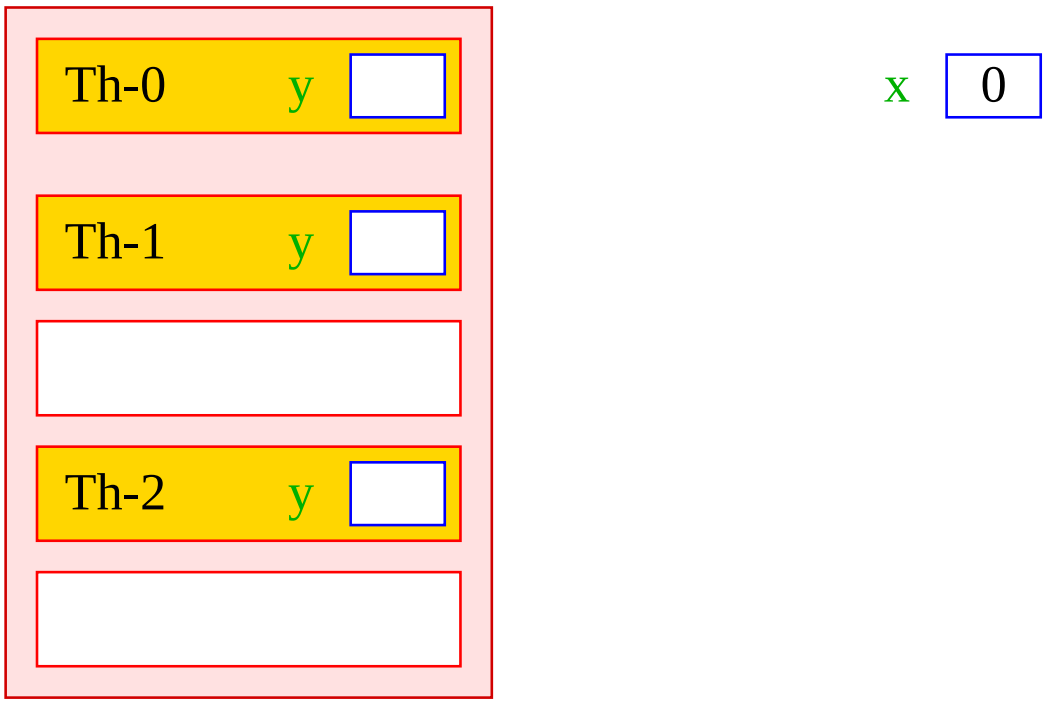
```
    ...
    public static void main(String[] args) {
       (new Thread(new Inc())).start();
       pause(2);
       (new Thread(new Inc())).start();
       pause(2);
       (new Thread(new Inc())).start();
    }
  }  // end of class Inc
```

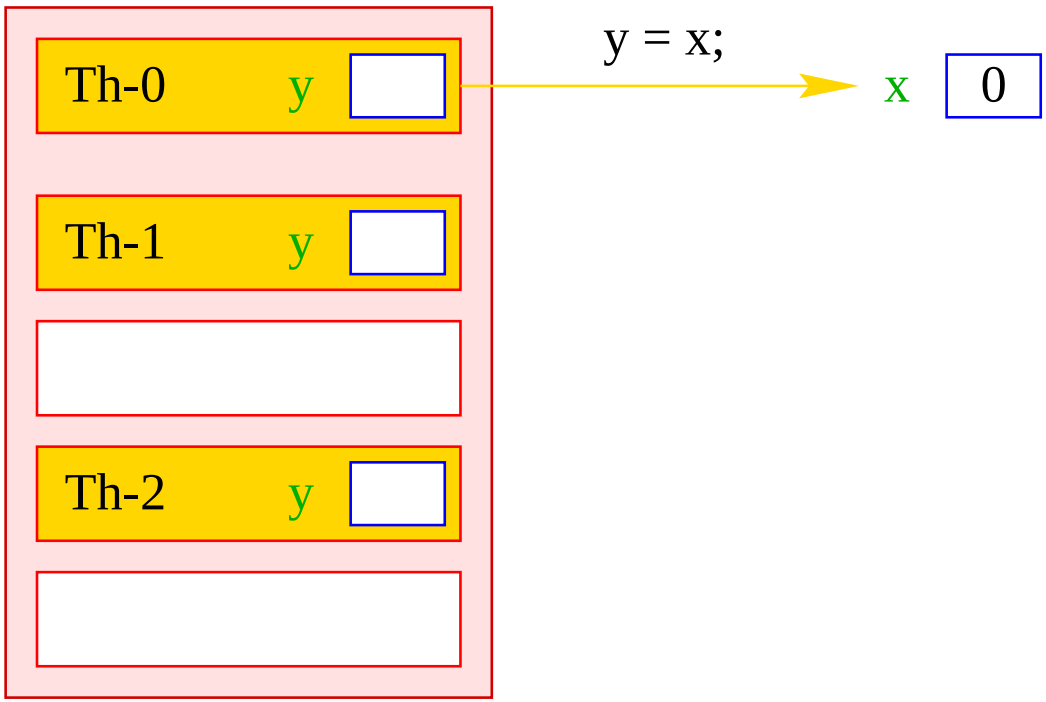
- `public static Thread currentThread();` liefert (eine Referenz auf) das ausführende Thread-Objekt.
- `public final String getName();` liefert den Namen des Thread-Objekts.
- Das Programm legt für drei Objekte der Klasse `Inc` Threads an.
- Die Methode `run()` inkrementiert die Klassen-Variable `x`.

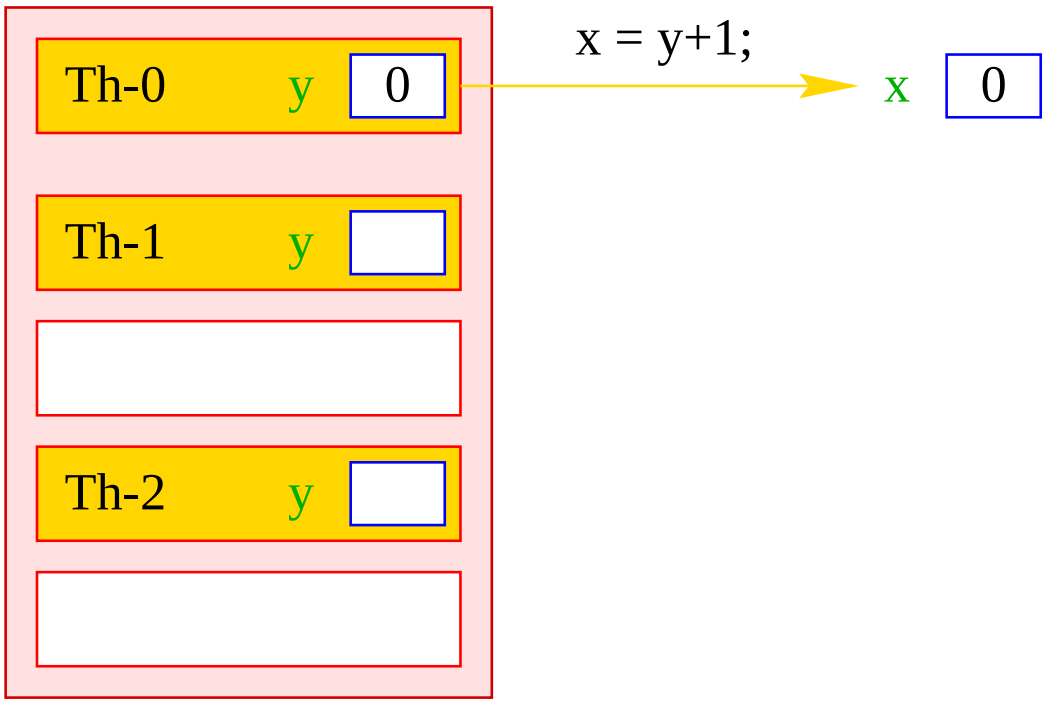
Die Ausführung liefert z.B.:

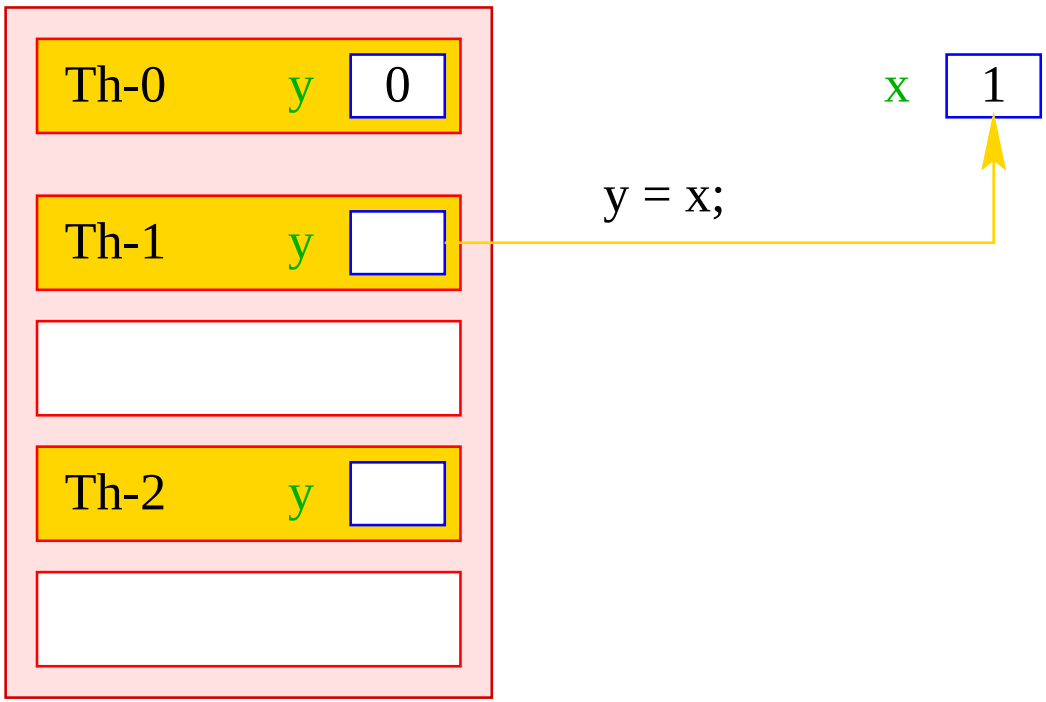
```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

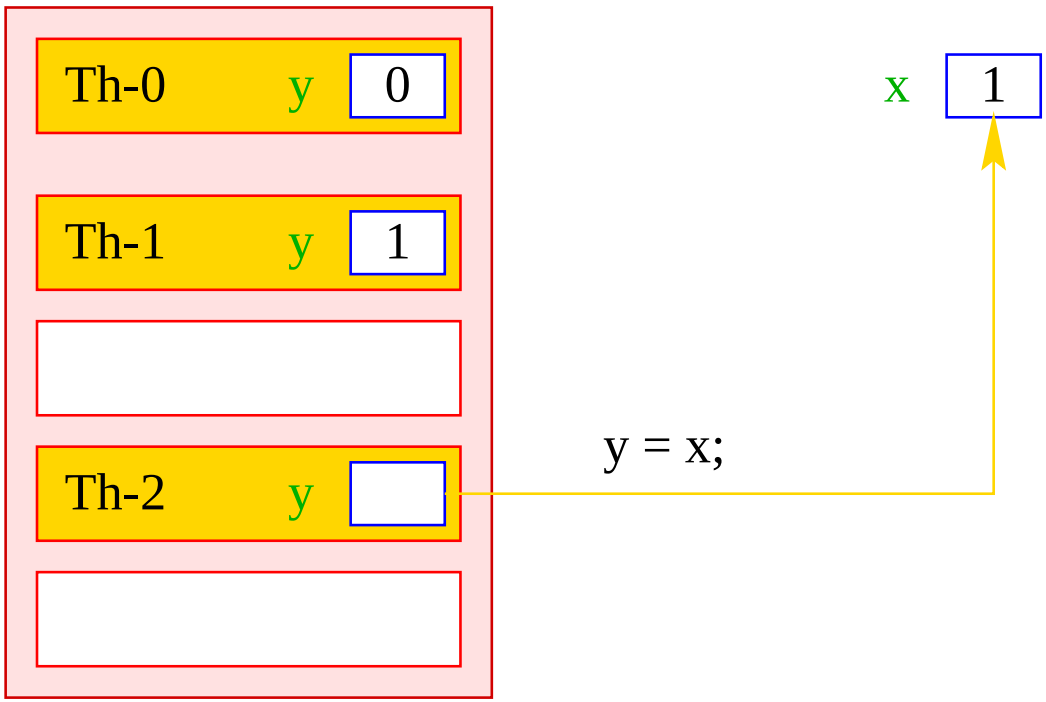
Der Grund:

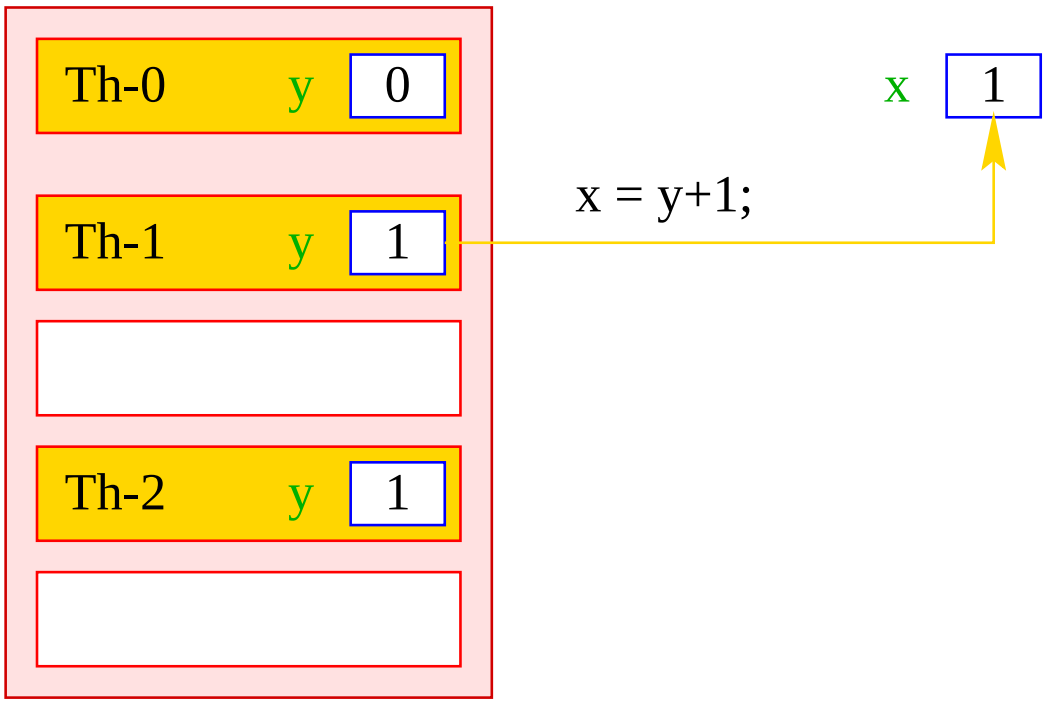

Th-0
y
Th-1
y
Th-2
y
x
0

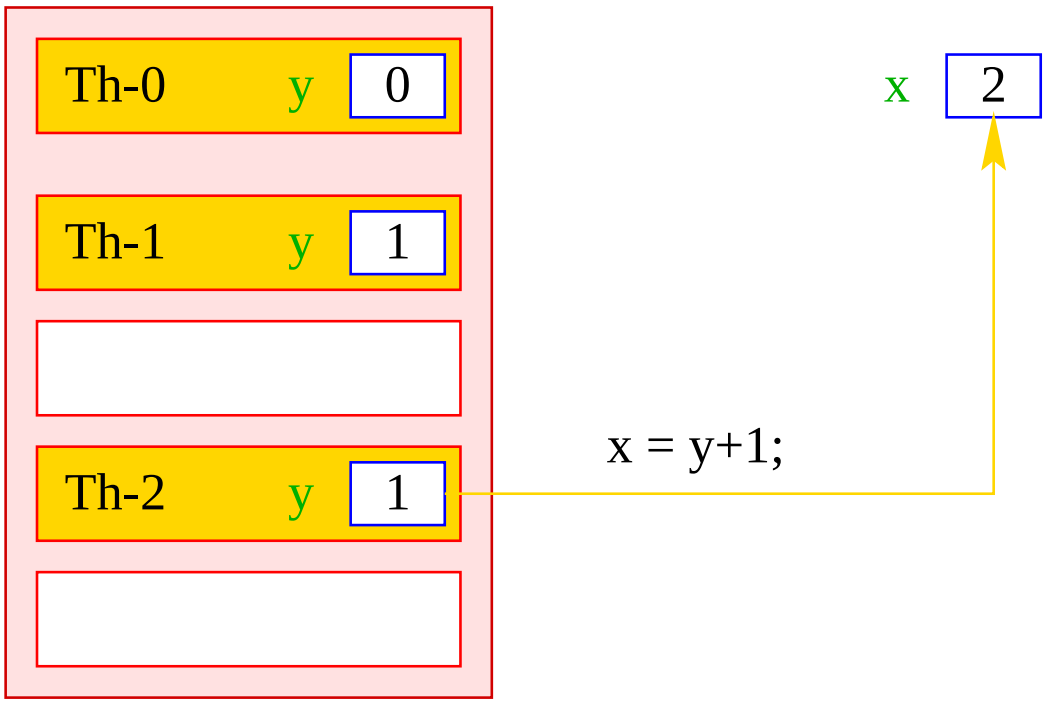

Th-0
y
y = x;
x
0
Th-1
y
Th-2
y

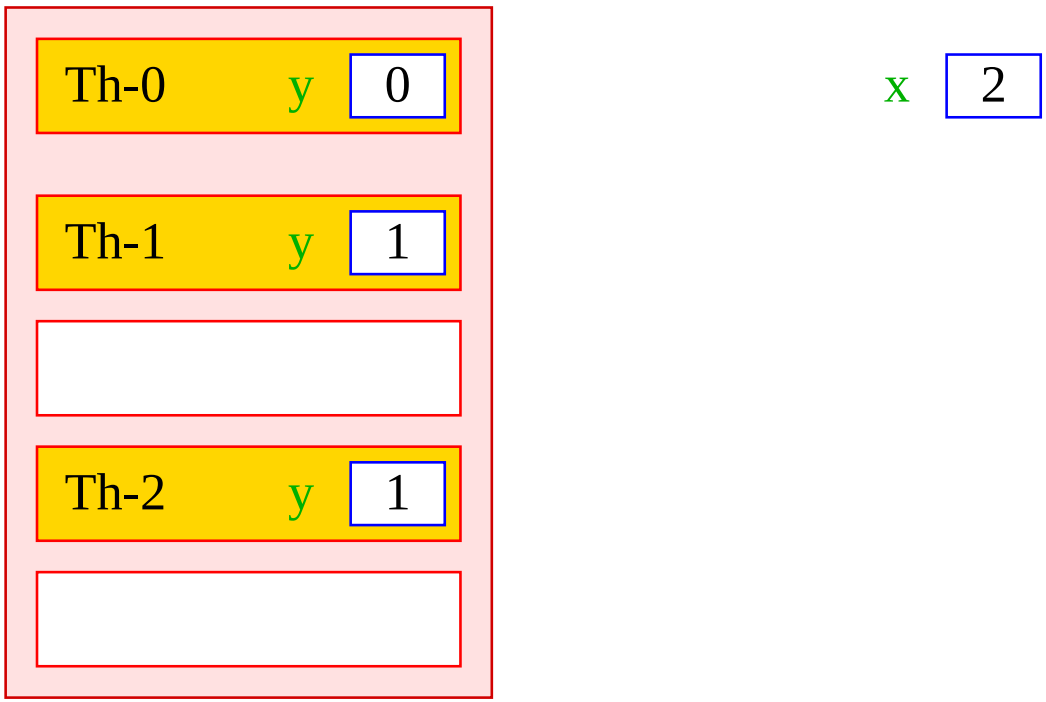

Th-0
y
0
x = y+1;
x
0
Th-1
y
Th-2
y


Th-0
y
0
x
1
y = x;
Th-1
y
Th-2
y


Th-0
y
0
Th-1
y
1
Th-2
y
x
1
y = x;

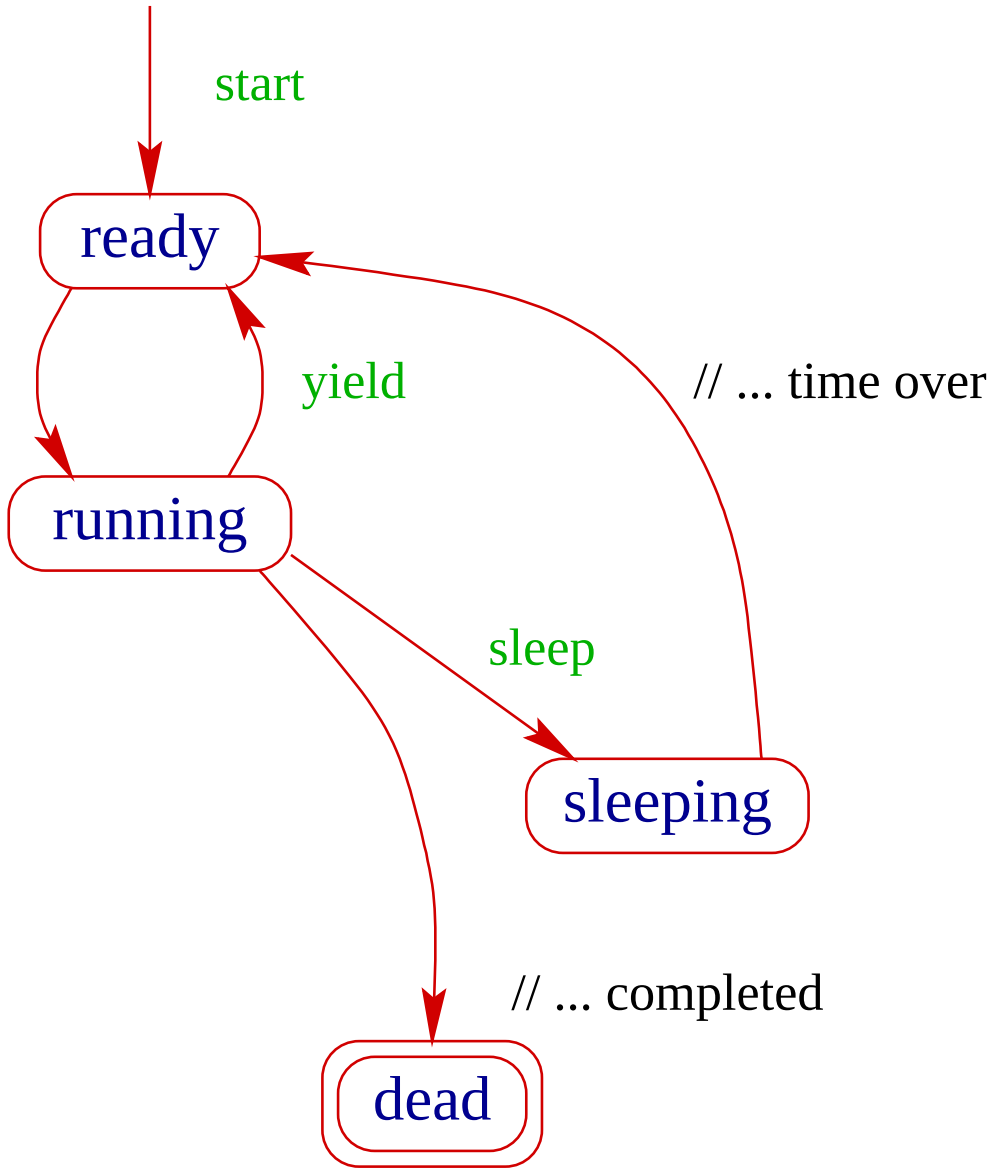

Th-0
y
0
Th-1
y
1
Th-2
y
1
x = y+1;
x
1

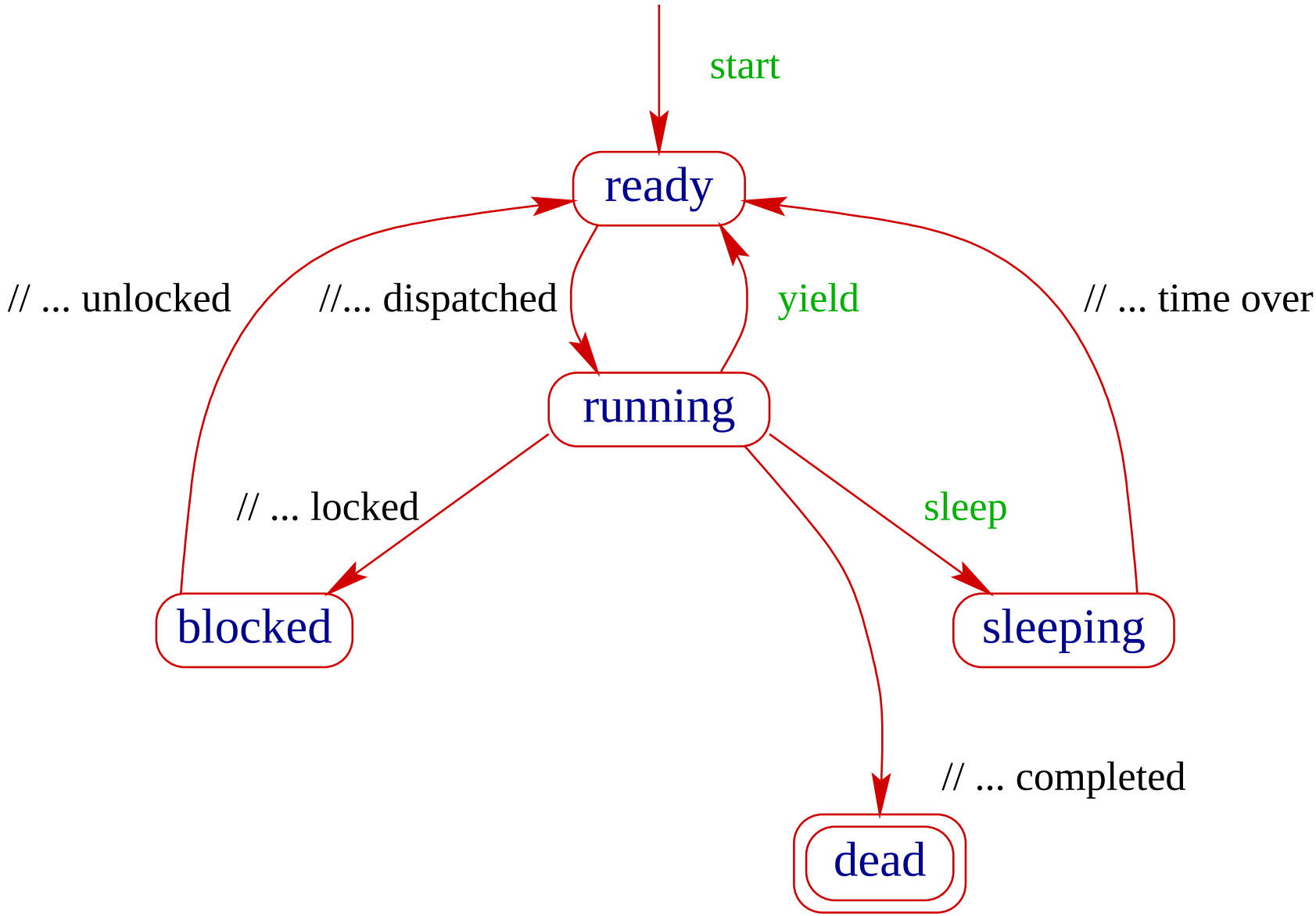

Th-0
y
0
Th-1
y
1
Th-2
y
1
x = y+1;
x
2


Th-0
y
0
Th-1
y
1
Th-2
y
1
x
2

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.


start
ready
yield
// ... time over
running
sleep
sleeping
// ... completed
dead


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

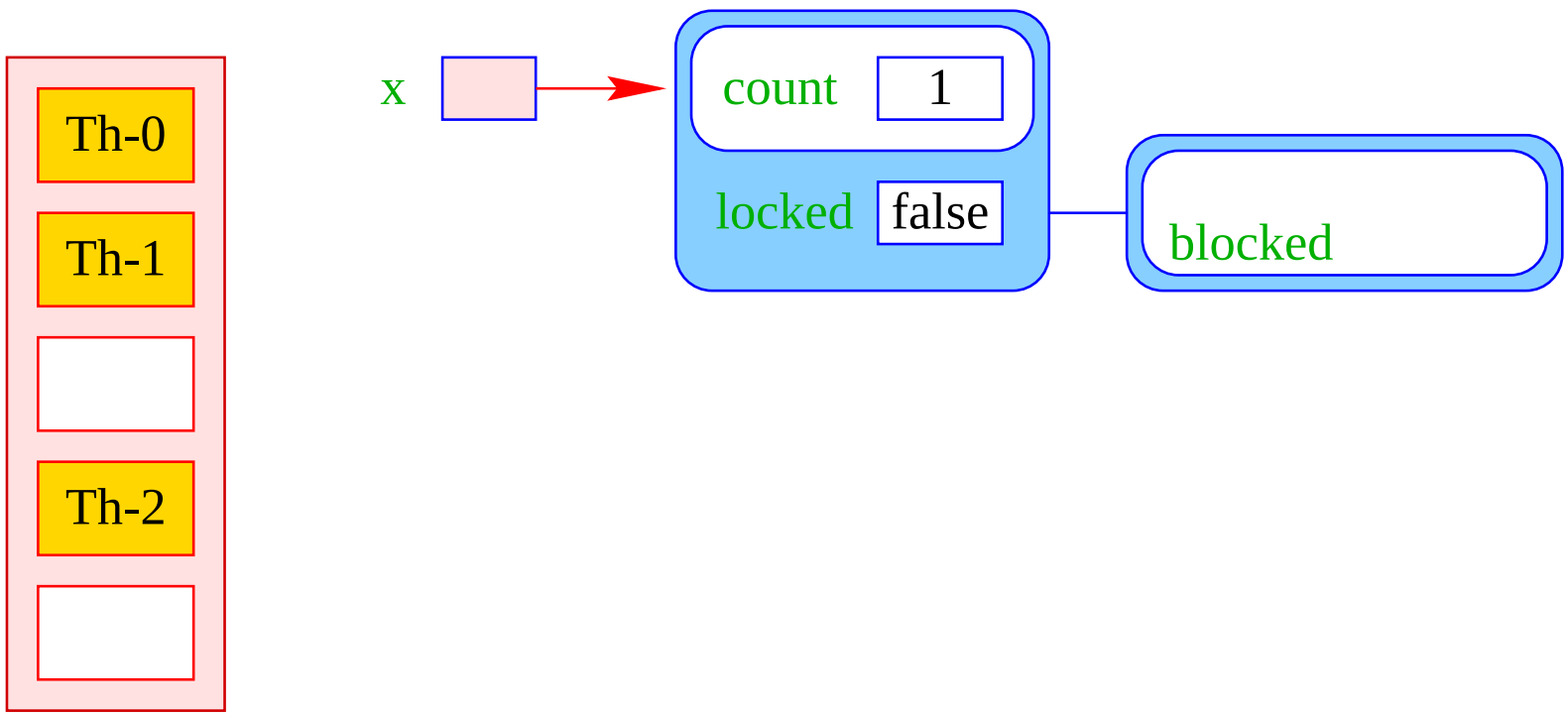
- Jedes Objekt obj mit `synchronized`-Methoden verfügt über:
  1. über ein boolesches Flag `boolean locked;` sowie
  2. über eine Warteschlange `ThreadQueue blockedThreads`.
- Vor Betreten seines kritischen Abschnitts führt ein Thread (implizit) die atomare Operation obj`.lock()` aus:

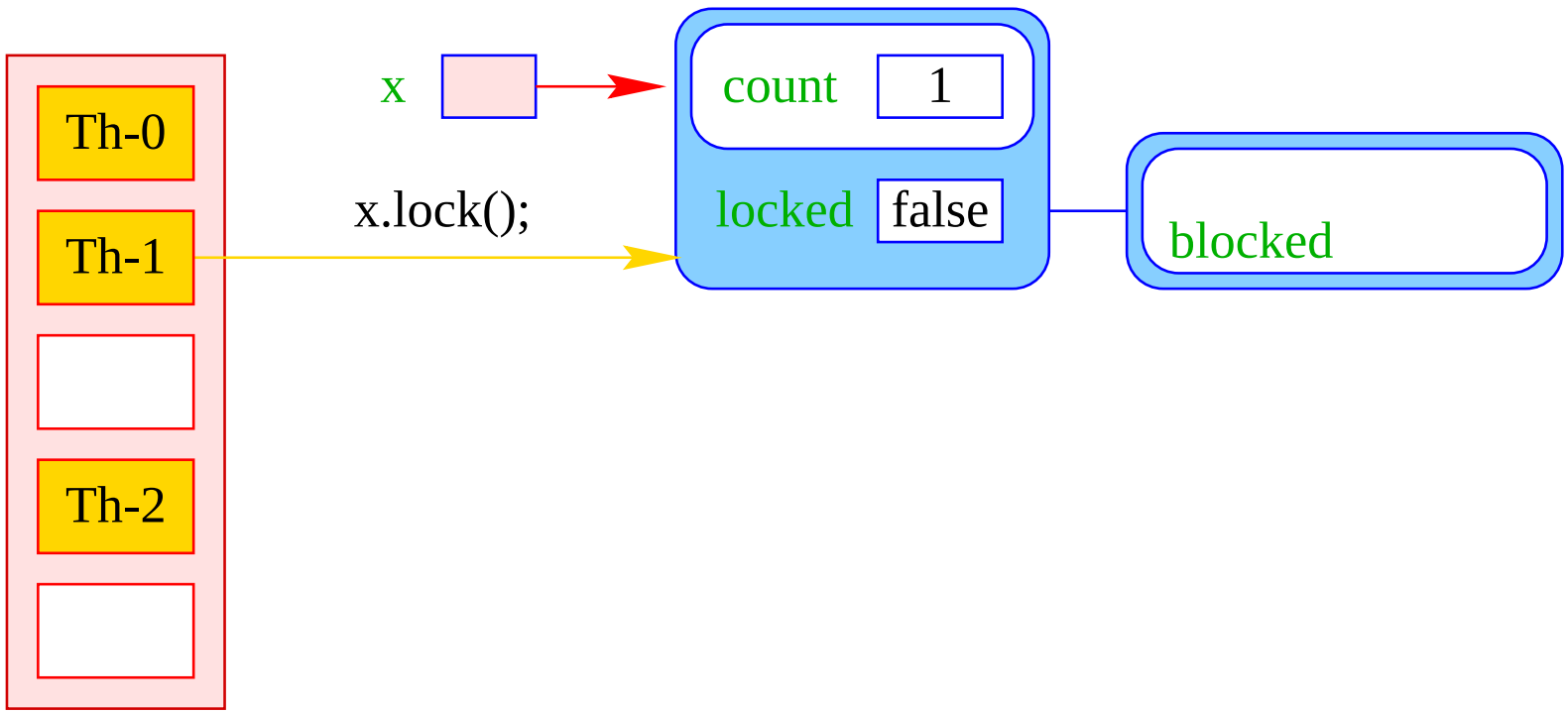
```
private void lock() {
   if (!locked) locked = true; // betrete krit. Abschnitt
   else {                      // Lock bereits vergeben
      Thread t = Thread.currentThread();
      blockedThreads.enqueue(t);
      t.state = blocked;       // blockiere
   }
}  // end of lock()
```

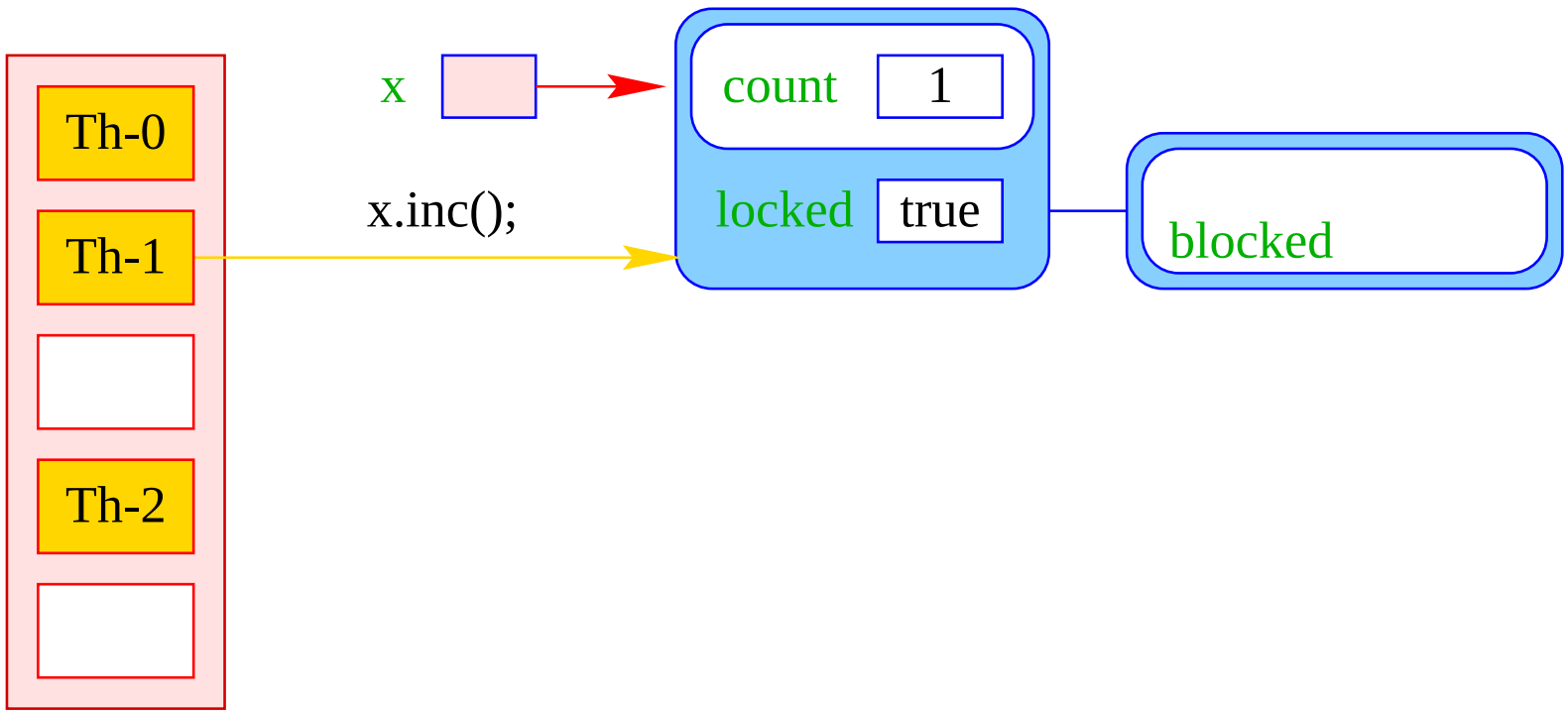
- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception :-), führt er (implizit) die atomare Operation obj.`unlock()` aus:

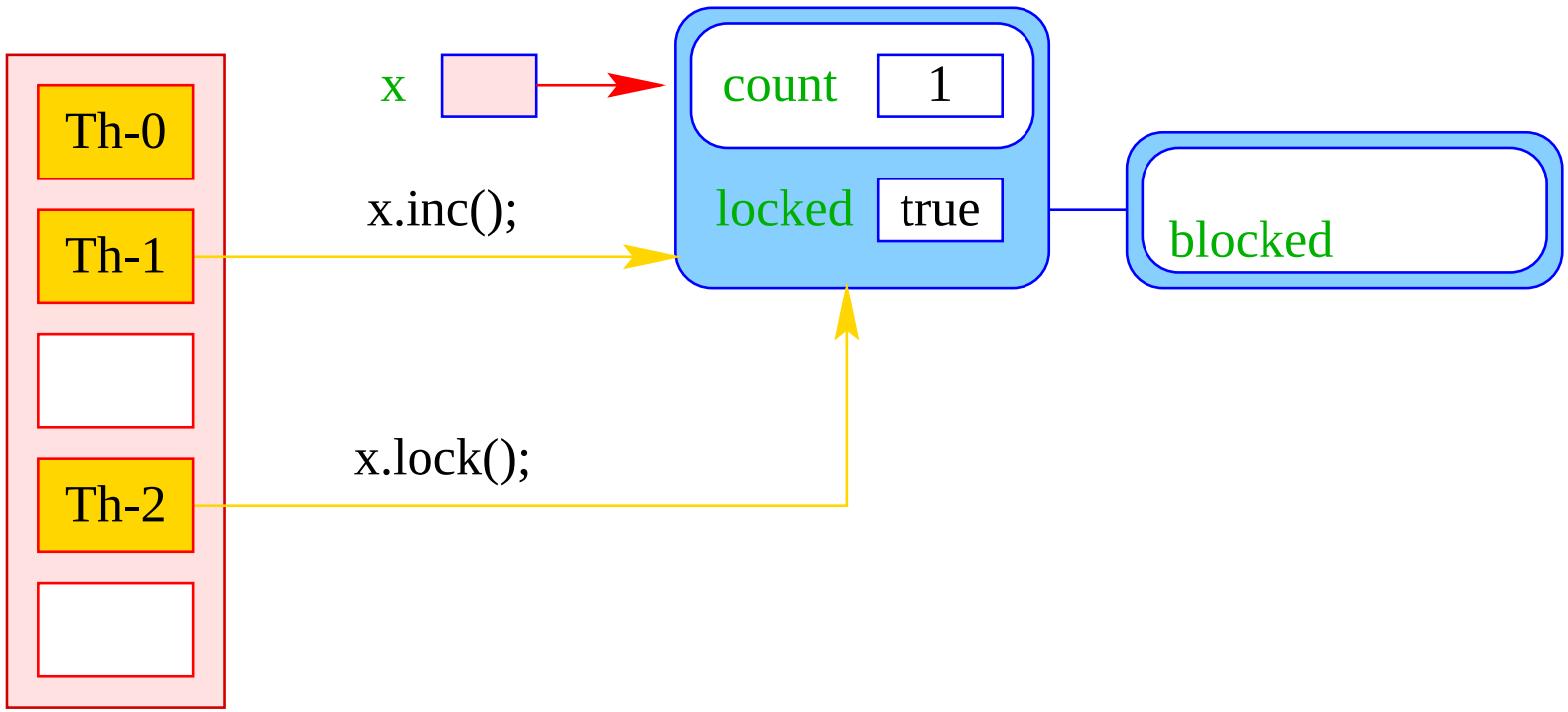
```
private void unlock() {
   if (blockedThreads.empty())
      locked = false; // Lock frei geben
   else {             // Lock weiterreichen
      Thread t = blockedThreads.dequeue();
      t.state = ready;
   }
}  // end of unlock()
```

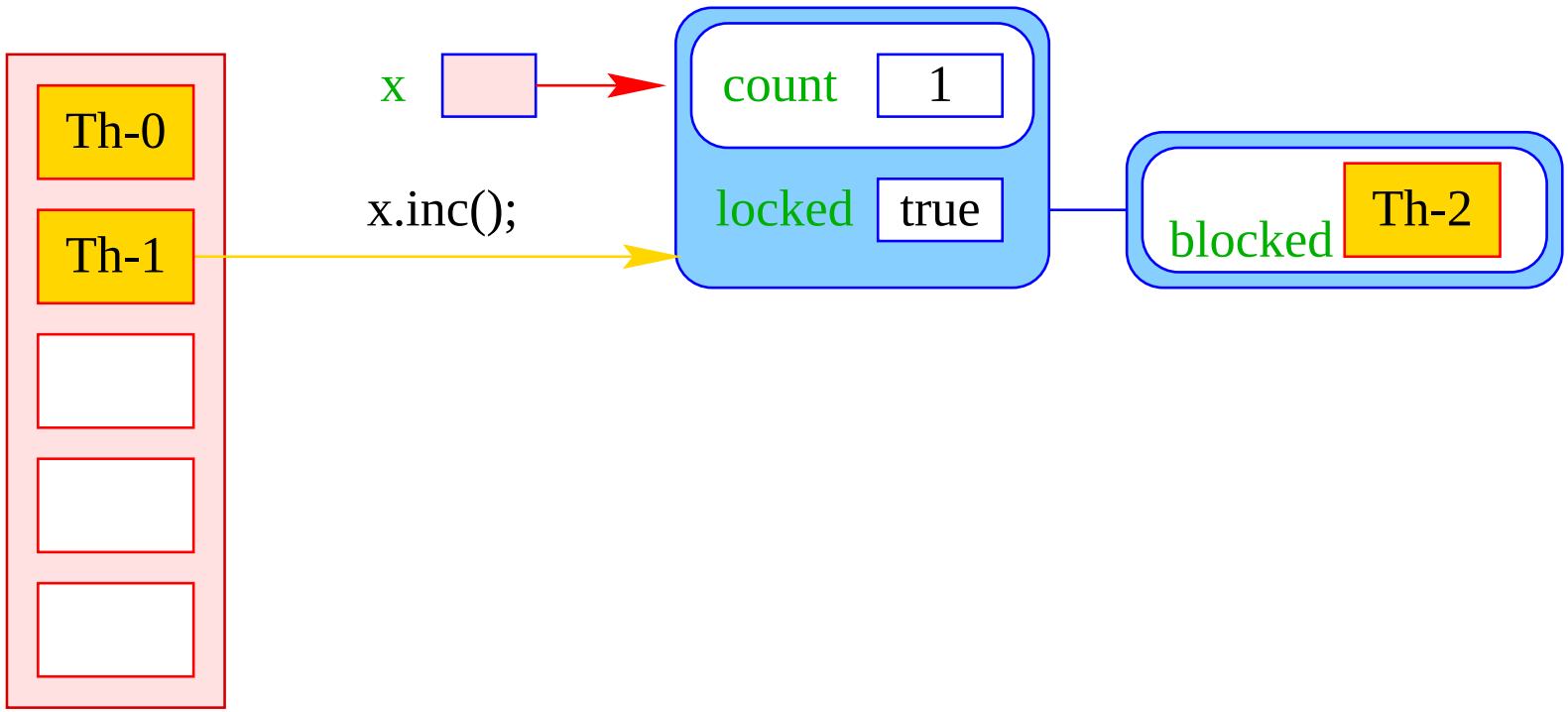
- Dieses Konzept nennt man Monitor.

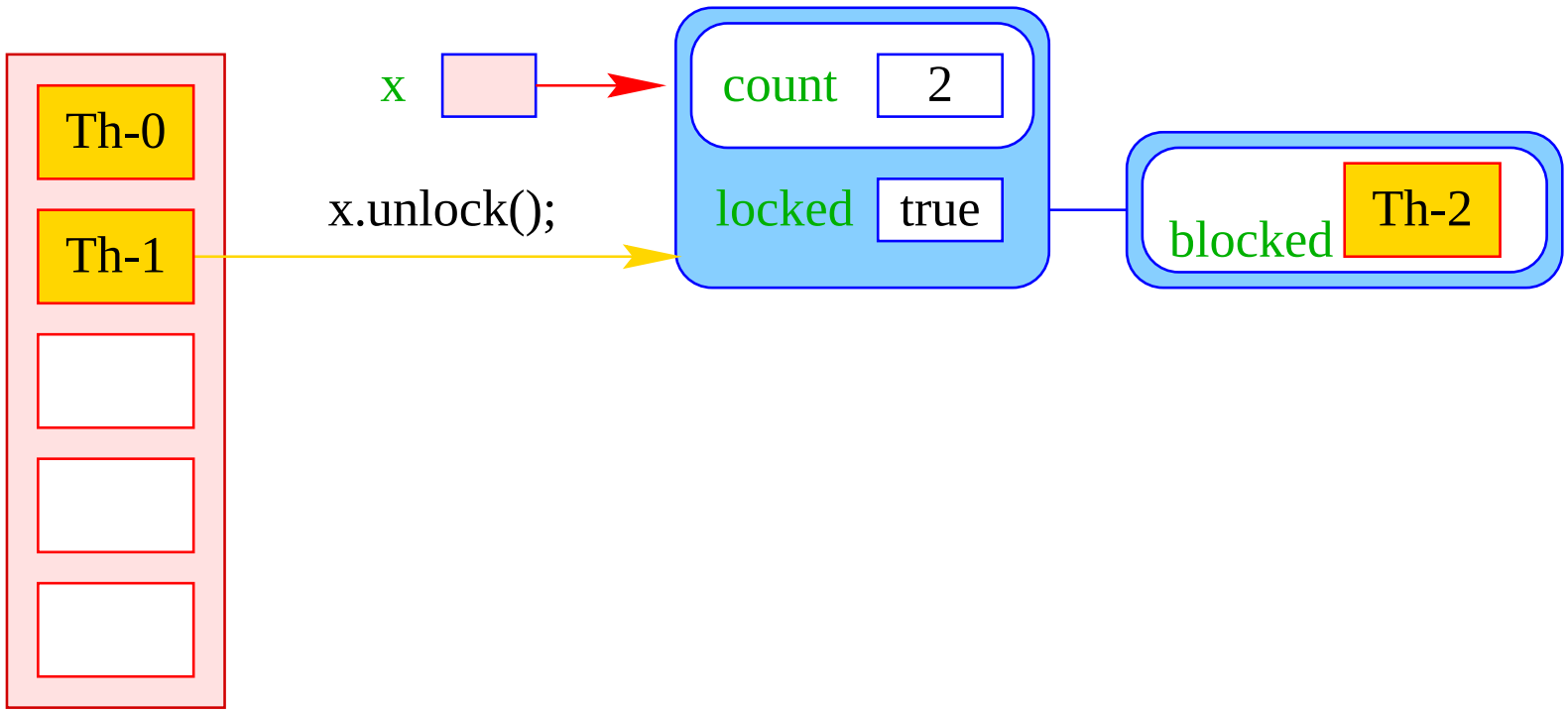

Th-0
Th-1
Th-2
x
count
1
locked
false
blocked

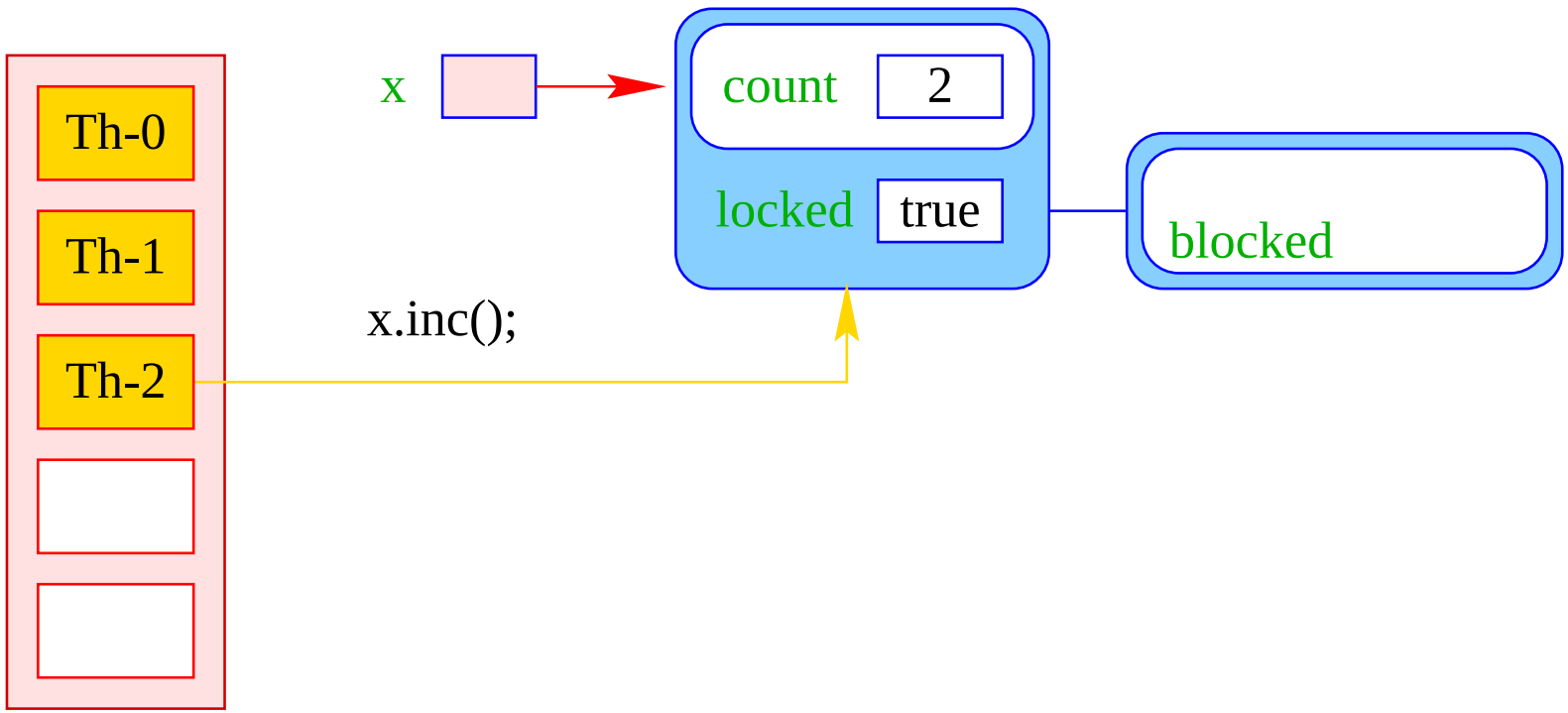

Th-0
Th-1
Th-2
x
x.lock();
count
1
locked
false
blocked

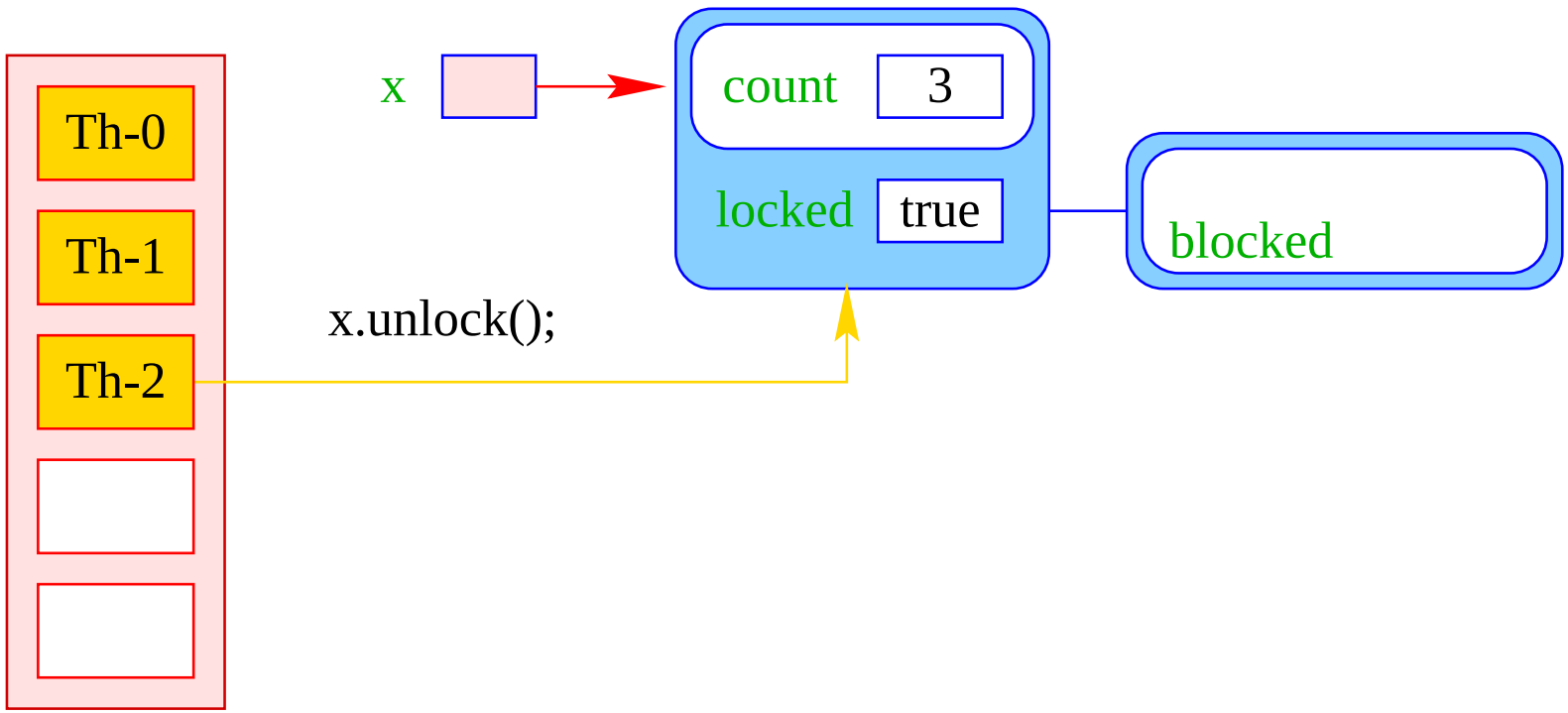

Th-0
Th-1
Th-2
x
x.inc();
count
1
locked
true
blocked

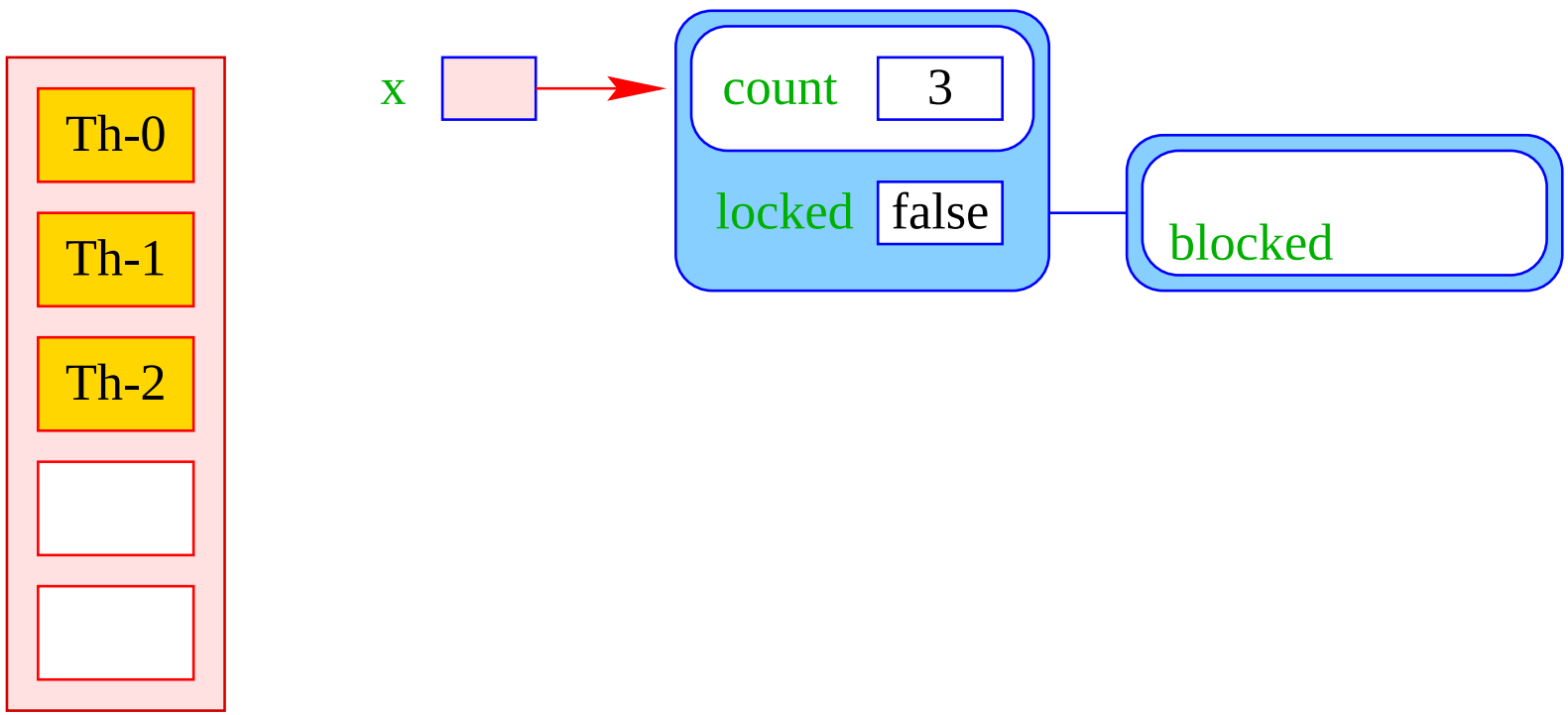

Th-0
Th-1
Th-2
x
x.inc();
x.lock();
count
1
locked
true
blocked


Th-0
Th-1
x
x.inc();
count
1
locked
true
blocked
Th-2


Th-0
Th-1
x
x.unlock();
count
2
locked
true
blocked
Th-2


Th-0
Th-1
Th-2
x
count
2
locked
true
blocked
x.inc();


Th-0
Th-1
Th-2
x
count
3
locked
true
blocked
x.unlock();


Th-0
Th-1
Th-2
x
count
3
locked
false
blocked

```
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
```