Informatik 2

Wintersemester 2009/10

Helmut Seidl

Institut für Informatik
TU München

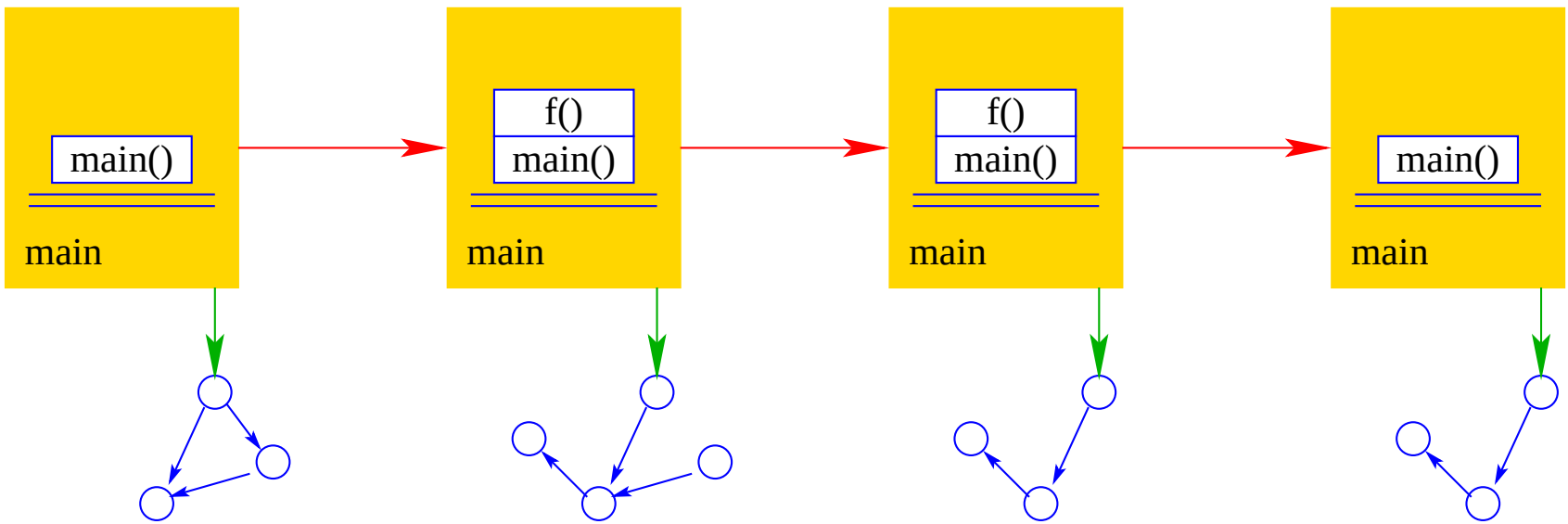
0 Allgemeines

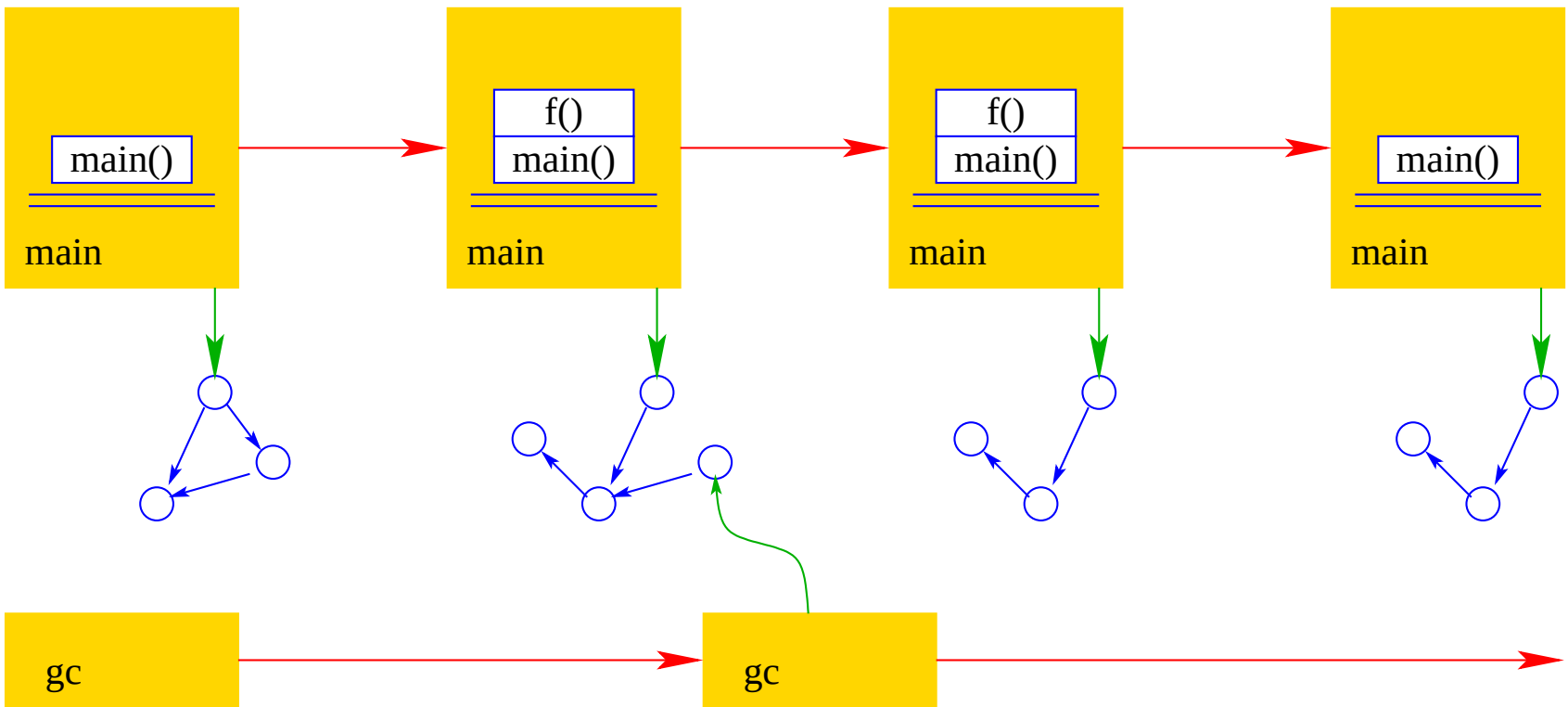
Inhalt dieser Vorlesung:

- Nebenläufigkeit in Java;
- Funktionales Programmieren mit OCaml :-)

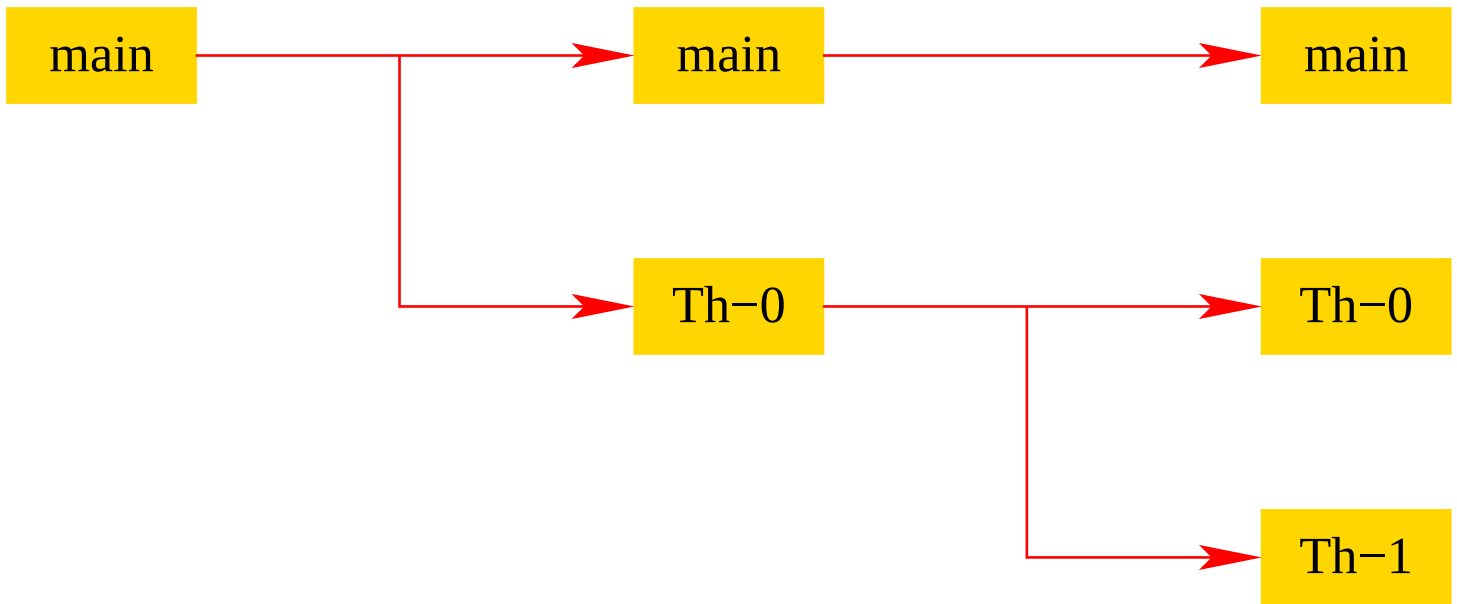
1 Threads

- Die Ausführung eines Java-Programms besteht in Wahrheit nicht aus einem, sondern mehreren parallel laufenden Threads.
- Ein Thread ist ein sequentieller Ausführungs-Strang.
- Der Aufruf eines Programms startet einen Thread `main`, der die Methode `main()` des Programms ausführt.
- Ein weiterer Thread, den das Laufzeitsystem parallel startet, ist die Garbage Collection.
- Die Garbage Collection soll mittlerweile nicht mehr erreichbare Objekte beseitigen und den von ihnen belegten Speicherplatz der weiteren Programm-Ausführung zur Verfügung stellen.


main()
main
f()
main()
main
f()
main()
main
main()
main


main()
main
f()
main()
main
f()
main()
main
main()
main
gc
gc

- Mehrere Threads sind auch nützlich, um
  - ... mehrere Eingabe-Quellen zu überwachen (z.B. Mouse-Klicks und Tastatur-Eingaben) ↑Graphik;
  - ... während der Blockierung einer Aufgabe etwas anderes Sinnvolles erledigen zu können;
  - ... die Rechenkraft mehrerer Prozessoren auszunutzen.
- Neue Threads können deshalb vom Programm selbst erzeugt und gestartet werden.
- Dazu stellt Java die Klasse `Thread` und das Interface `Runnable` bereit.


main
main
main
Th−0
Th−0
Th−1

Beispiel:

```
public class MyThread extends Thread {
   public void hello(String s) {
      System.out.println(s);
   }
   public void run() {
      hello("I'm running ...");
   }  // end of run()
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();
      System.out.println("Thread has been started ...");
   }  // end of main()
}     // end of class MyThread
```

- Neue Threads werden für Objekte aus (Unter-) Klassen der Klasse `Thread` angelegt.
- Jede Unterklasse von `Thread` sollte die Objekt-Methode `public void run();` implementieren.
- Ist `t` ein `Thread`-Objekt, dann bewirkt der Aufruf `t.start();` das folgende:

  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objekt-Methode `run()` für `t` wird angestoßen;
  3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

Beispiel:

```
public class MyRunnable implements Runnable {
   public void hello(String s) {
      System.out.println(s);
   }
   public void run() {
      hello("I'm running ...");
   }  // end of run()
   public static void main(String[] args) {
      Thread t = new Thread(new MyRunnable());
      t.start();
      System.out.println("Thread has been started ...");
   }  // end of main()
}     // end of class MyRunnable
```

- Auch das Interface `Runnable` verlangt die Implementierung einer Objekt-Methode `public void run();`
- `public Thread(Runnable` obj`);` legt für ein `Runnable`-Objekt obj ein `Thread`-Objekt an.
- Ist `t` das `Thread`-Objekt für das `Runnable` obj, dann bewirkt der Aufruf `t.start();` das folgende:
  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objekt-Methode `run()` für obj wird angestoßen;
  3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

Mögliche Ausführungen:

```
Thread has been started ...
I'm running ...
```

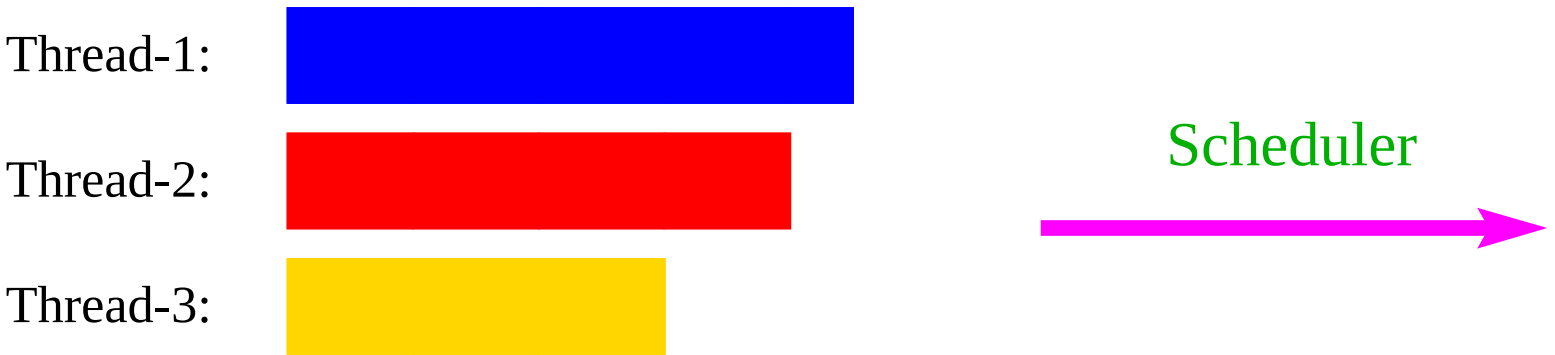
... oder:

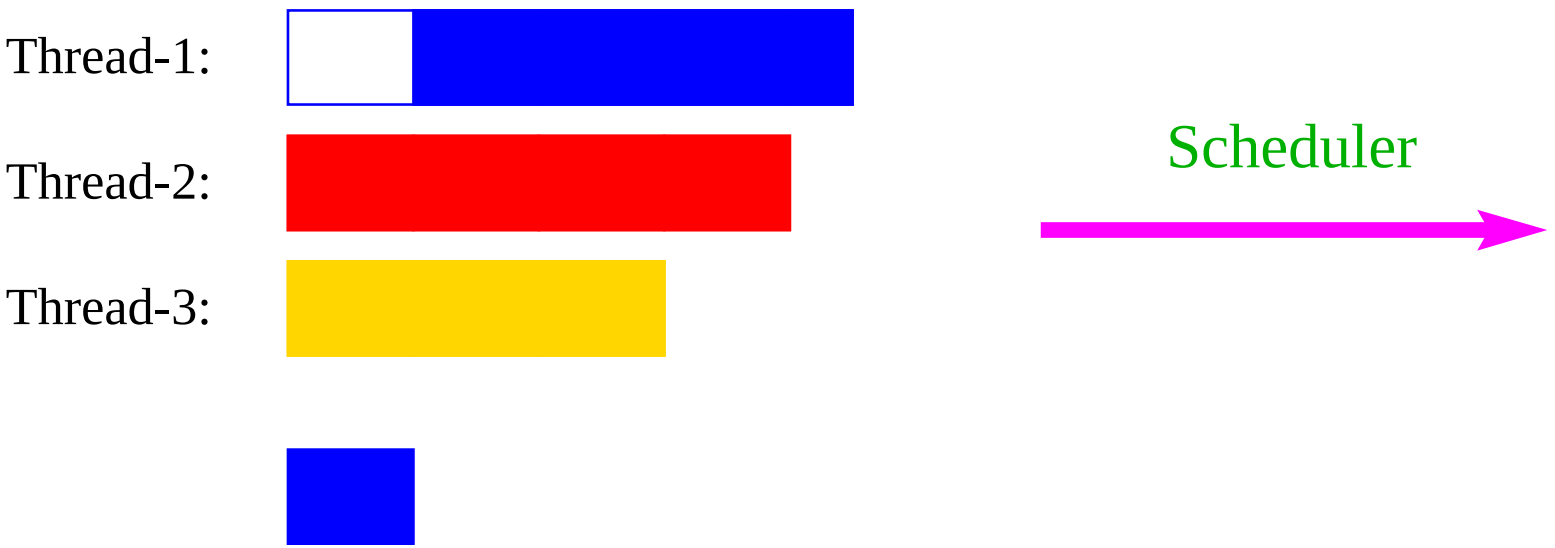
```
I'm running ...
Thread has been started ...
```

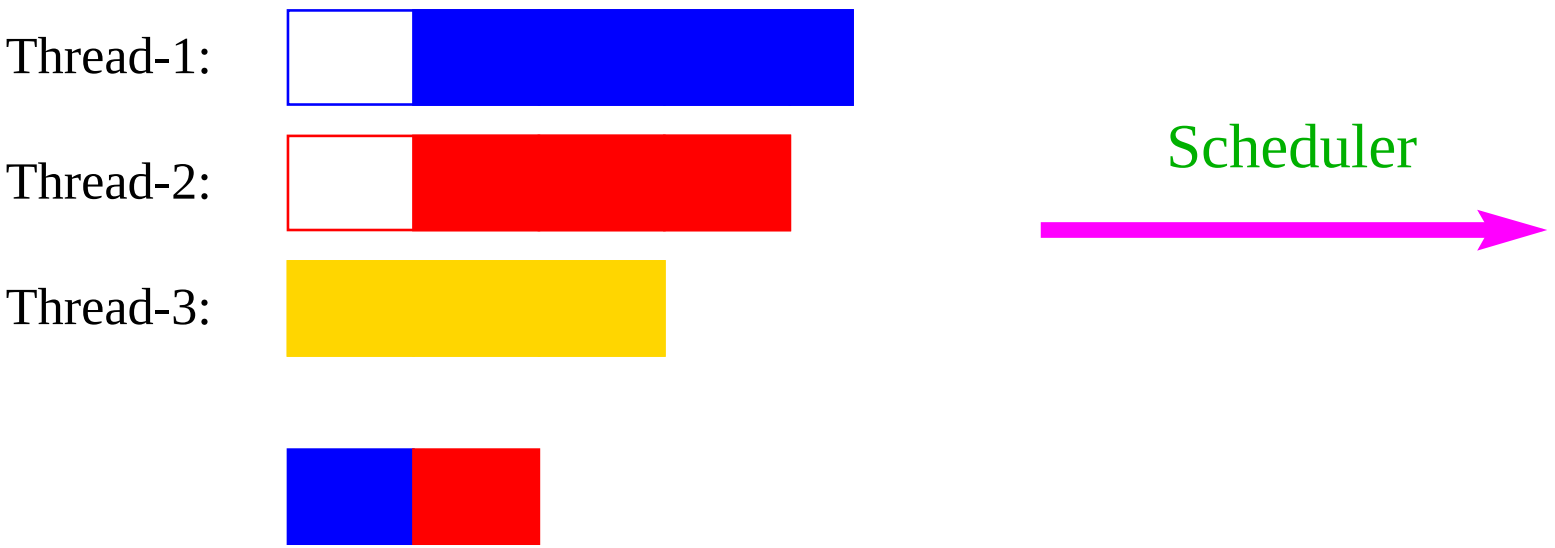
- Ein Thread kann nur eine Operation ausführen, wenn ihm ein Prozessor (CPU) zur Ausführung zugeteilt worden ist.
- Im Allgemeinen gibt es mehr Threads als CPUs.
- Der Scheduler verwaltet die verfügbaren CPUs und teilt sie den Threads zu.
- Bei verschiedenen Programm-Läufen kann diese Zuteilung verschieden aussehen!!!
- Es gibt verschiedene Politiken, nach denen sich Scheduler richten können ↑Betriebssysteme.

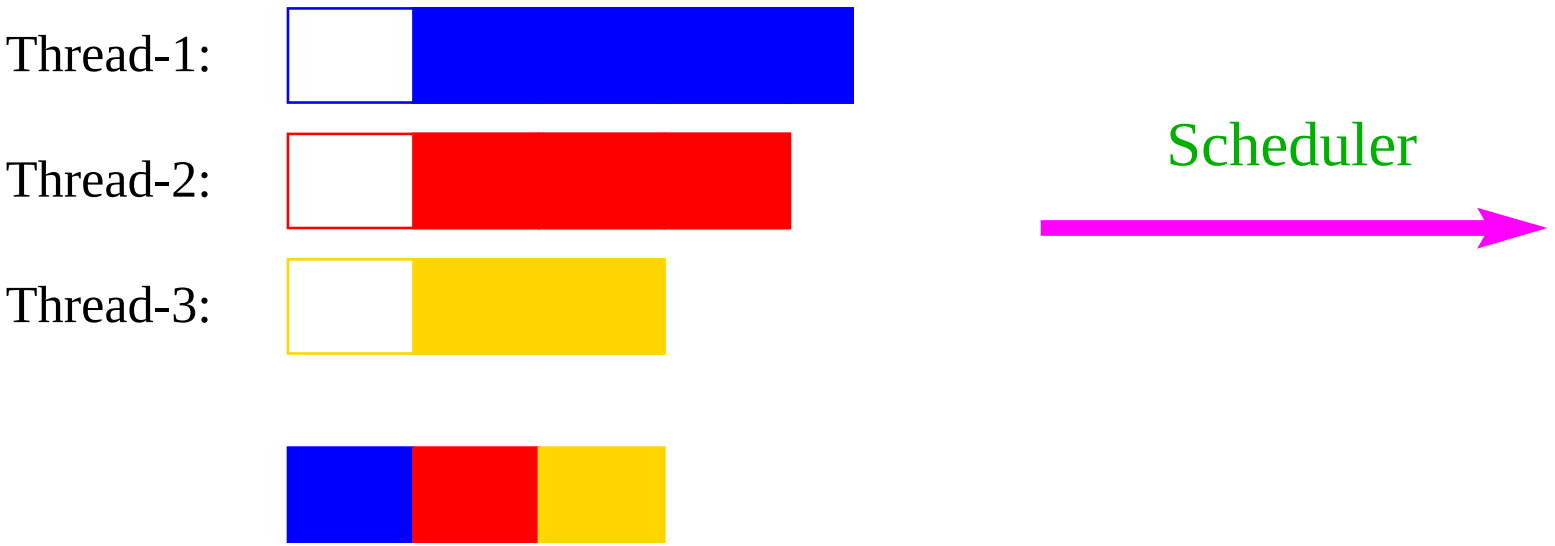
1. Zeitscheiben-Verfahren:

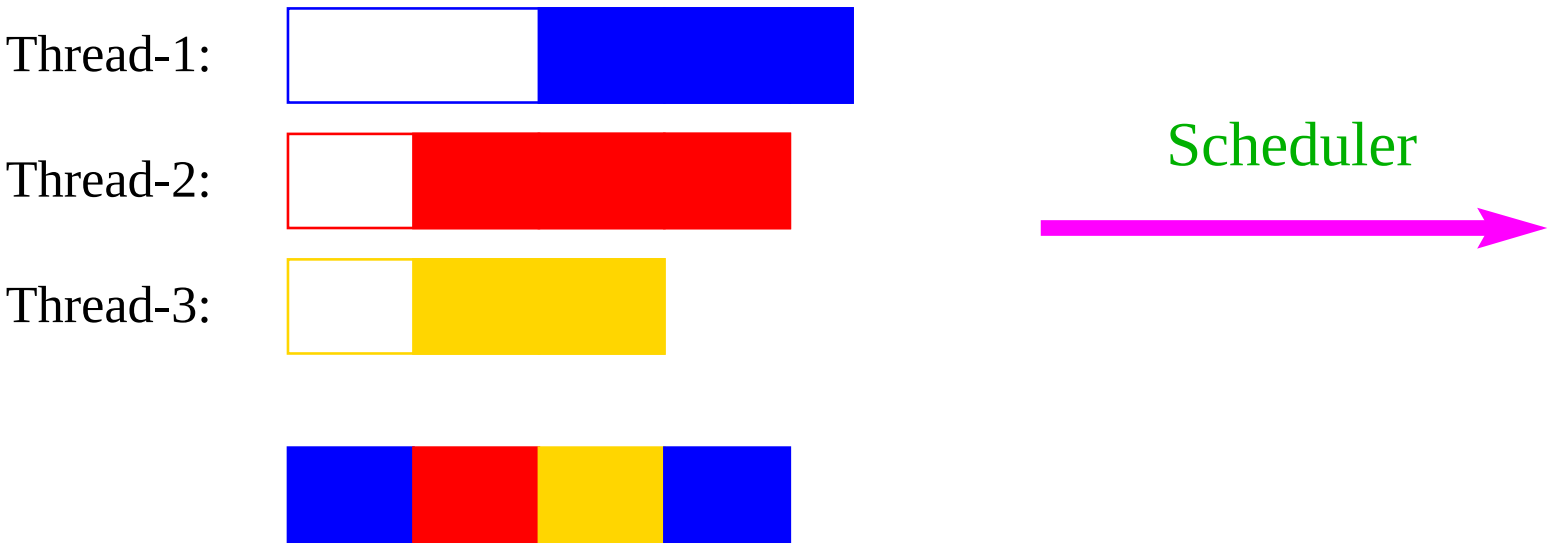
- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

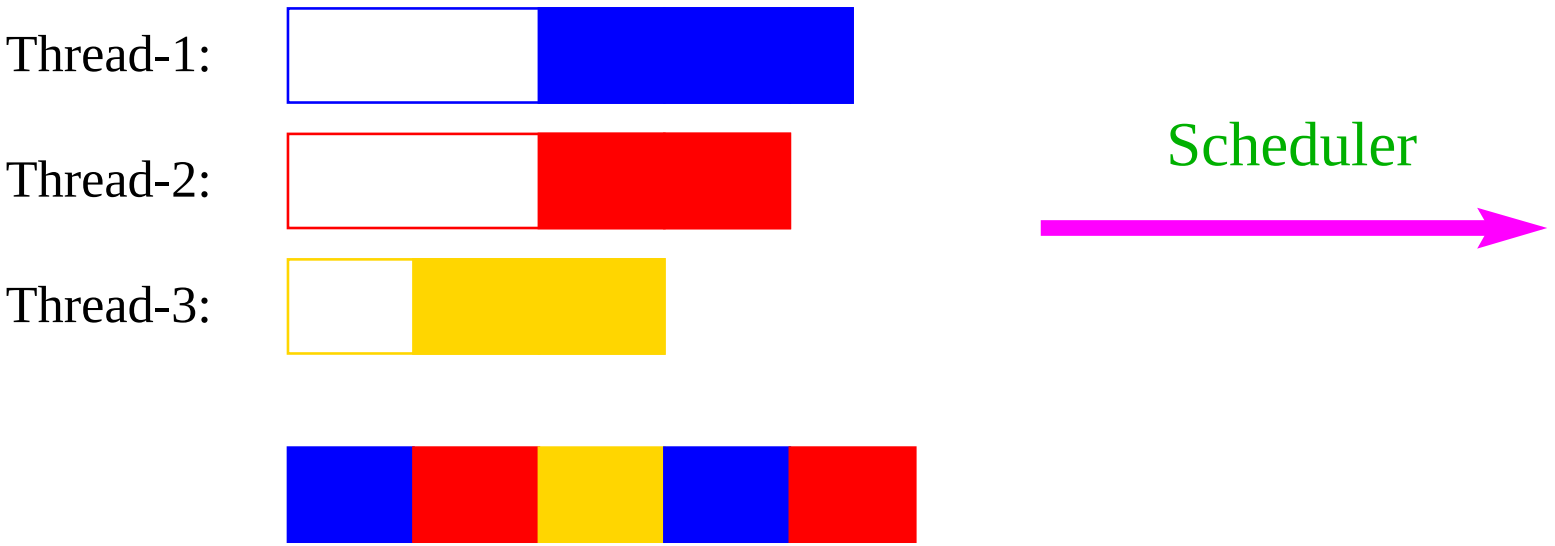

Thread-1:
Thread-2:
Thread-3:
Scheduler

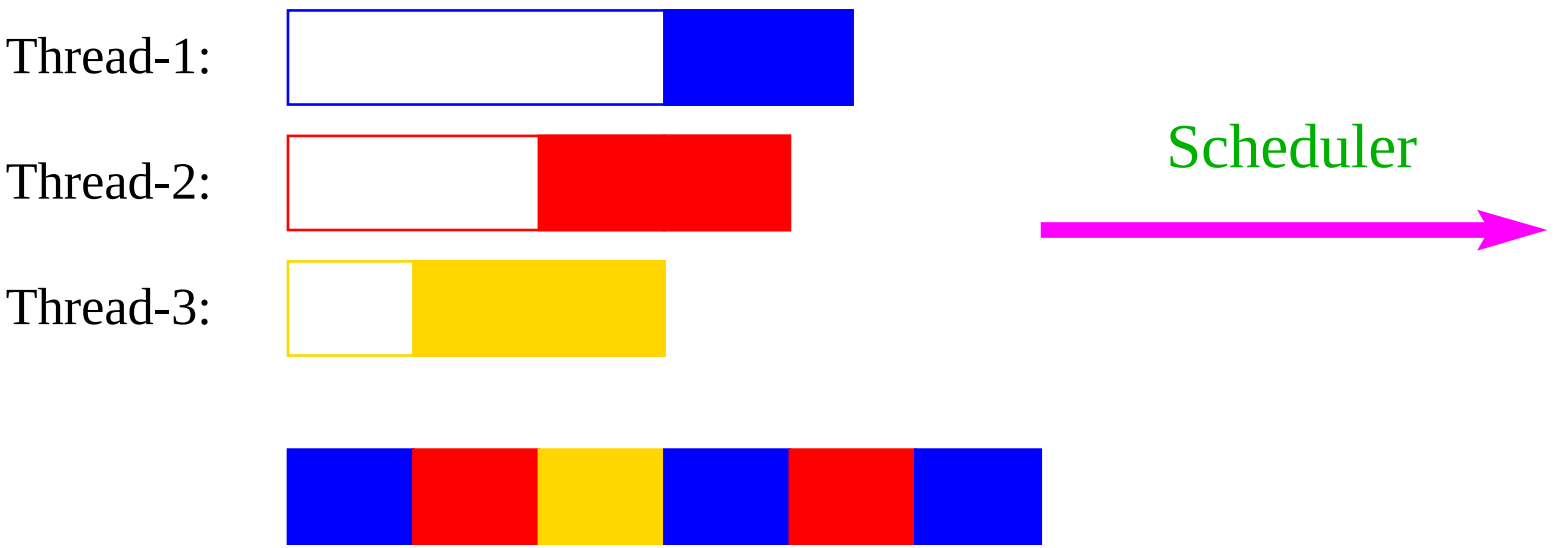

Thread-1:
Thread-2:
Thread-3:
Scheduler

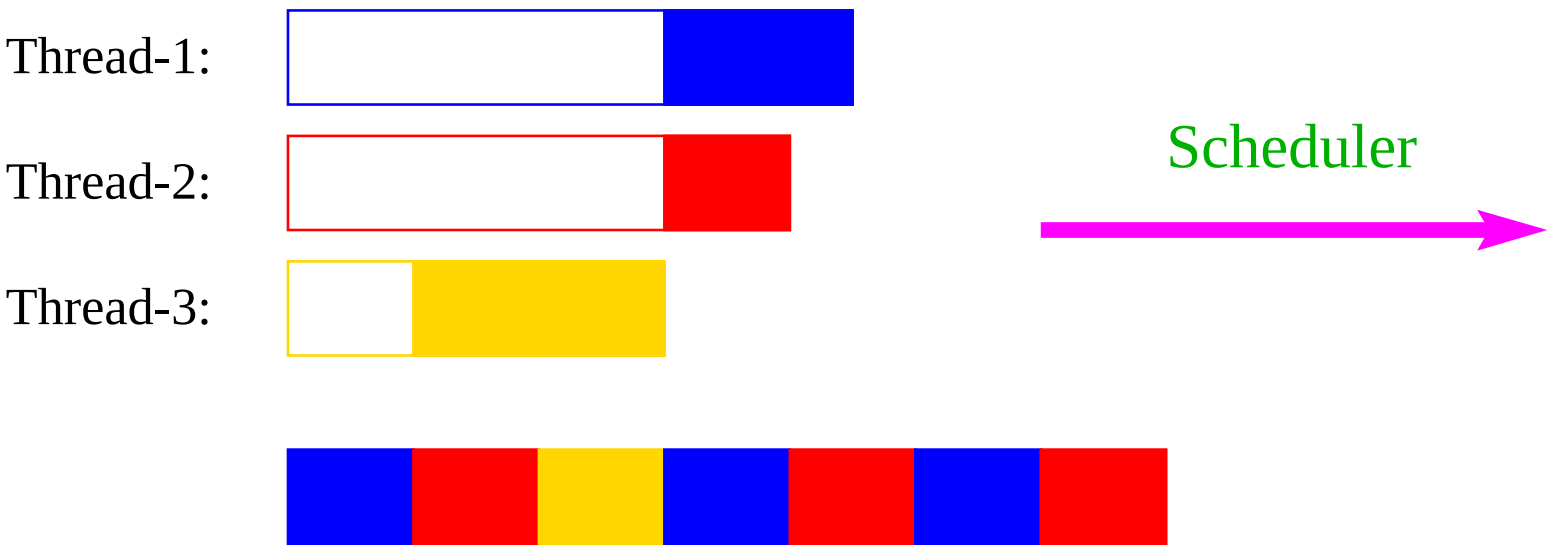

Thread-1:
Thread-2:
Thread-3:
Scheduler

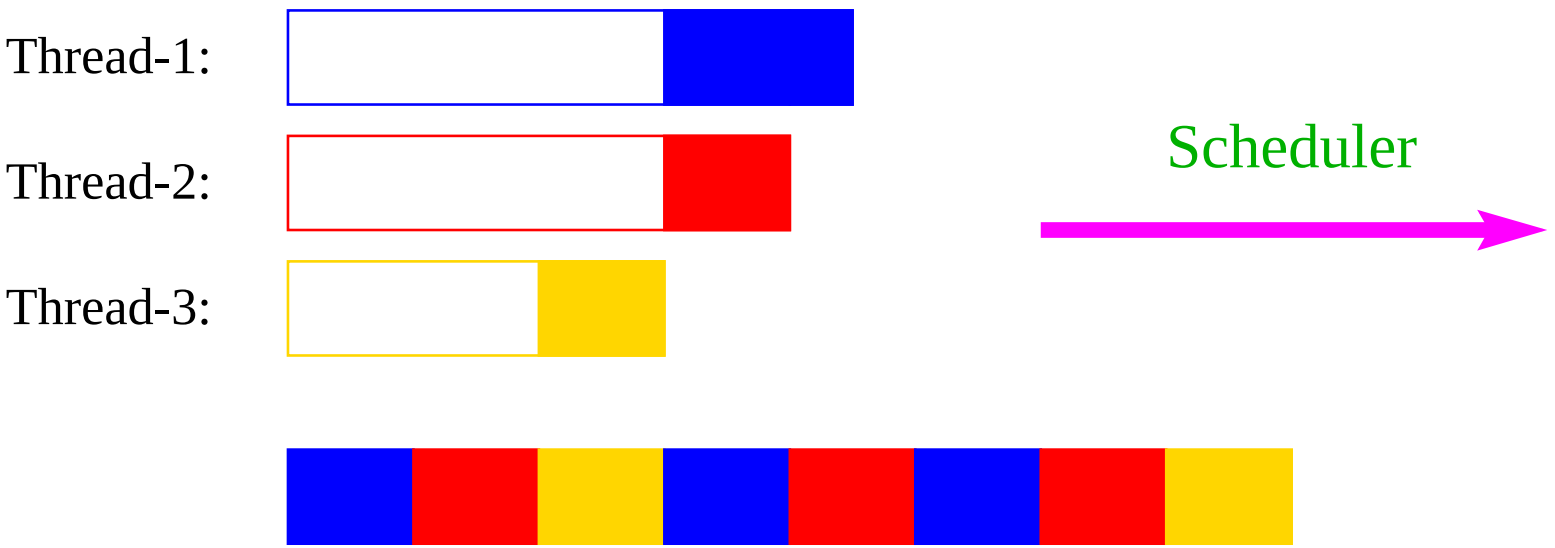

Thread-1:
Thread-2:
Thread-3:
Scheduler

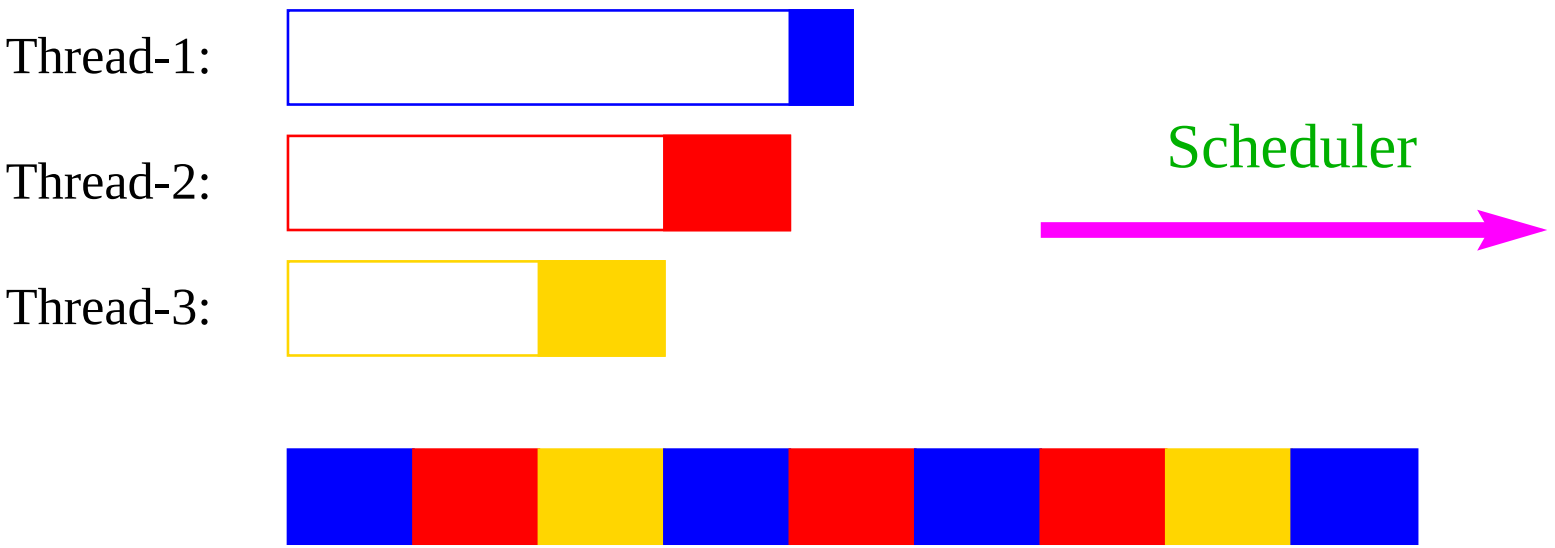

Thread-1:
Thread-2:
Thread-3:
Scheduler

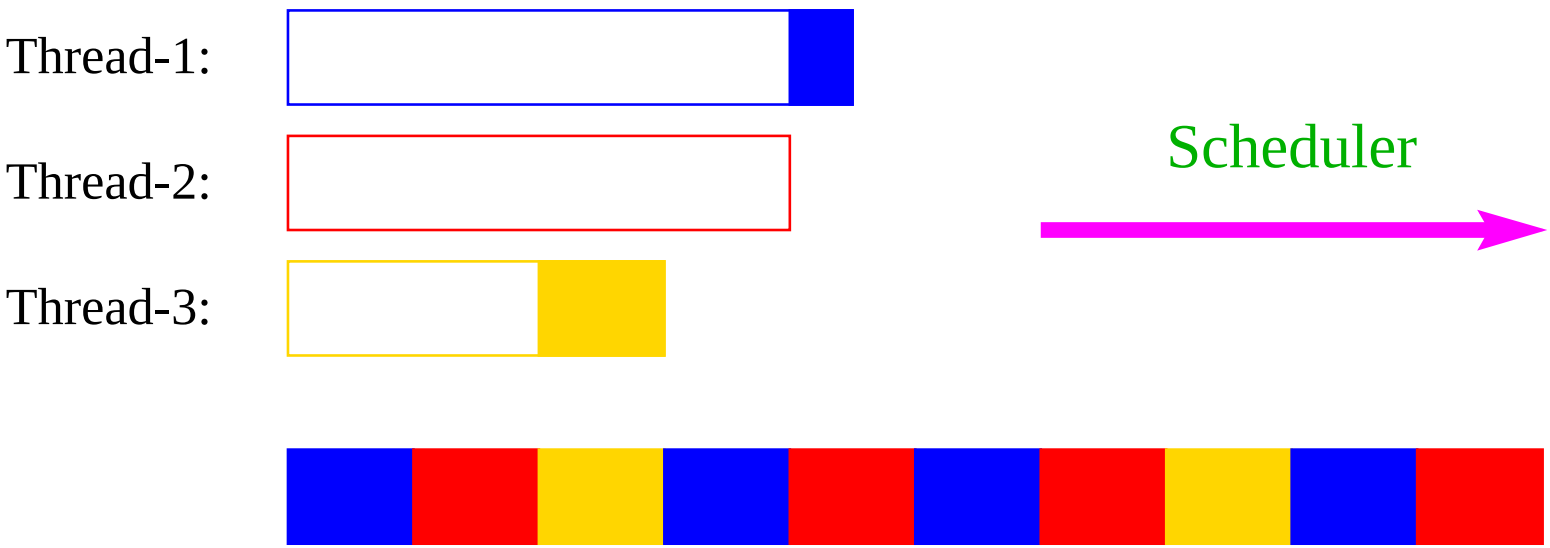

Thread-1:
Thread-2:
Thread-3:
Scheduler

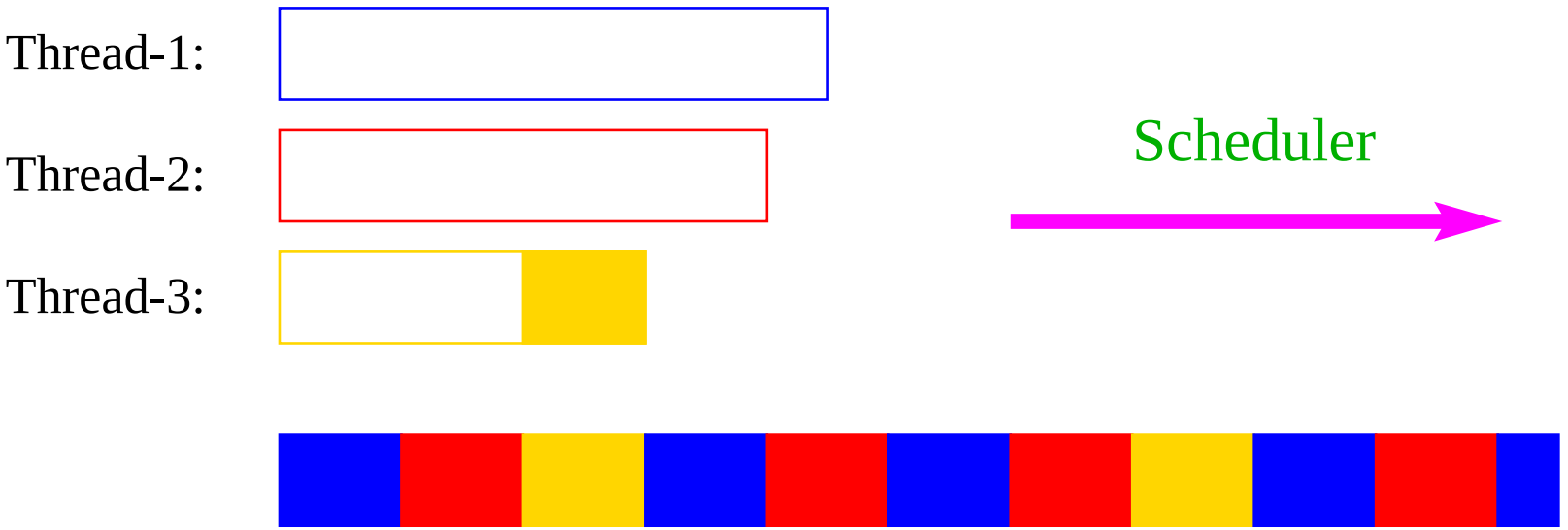

Thread-1:
Thread-2:
Thread-3:
Scheduler

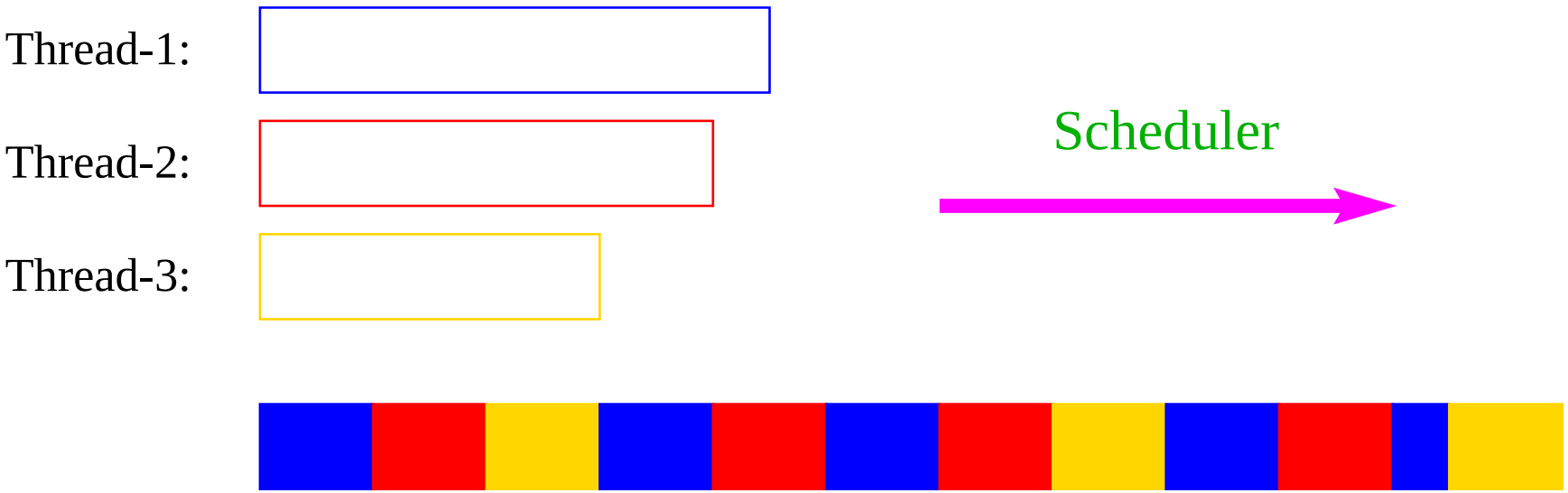

Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler

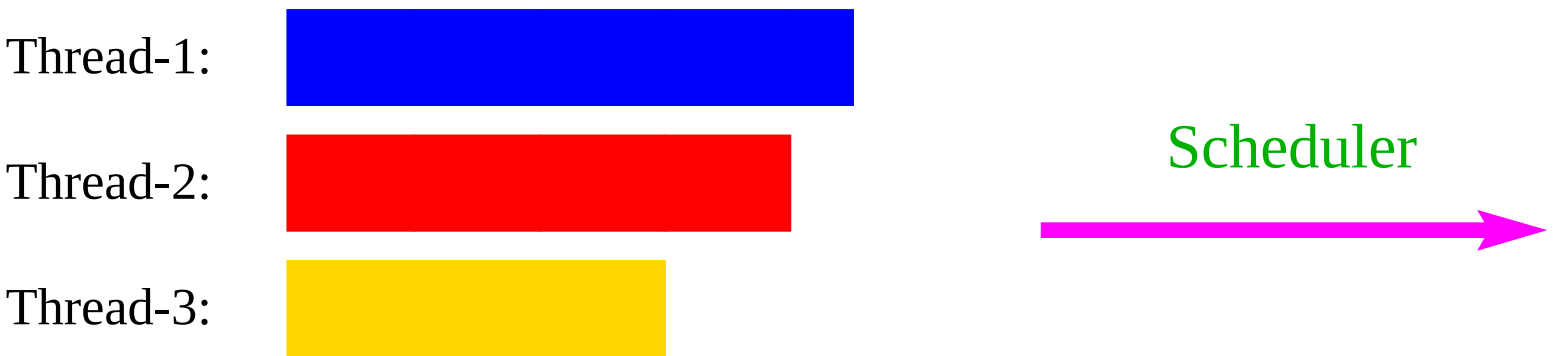

Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler

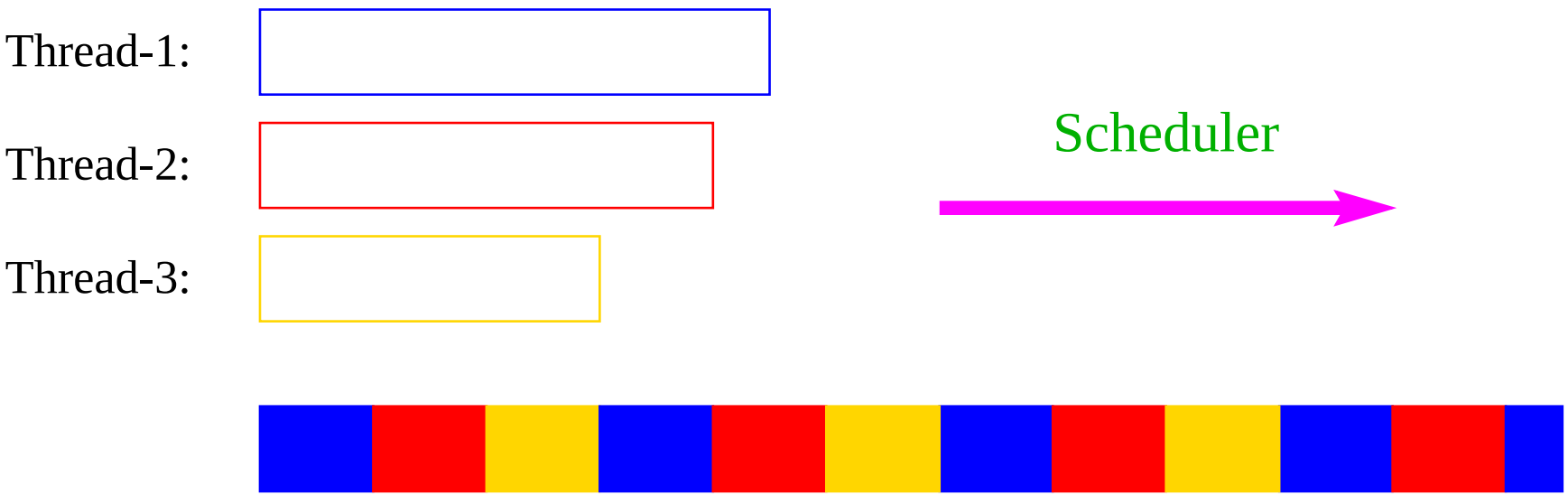
Achtung:

Eine andere Programm-Ausführung mag dagegen liefern:



Achtung:

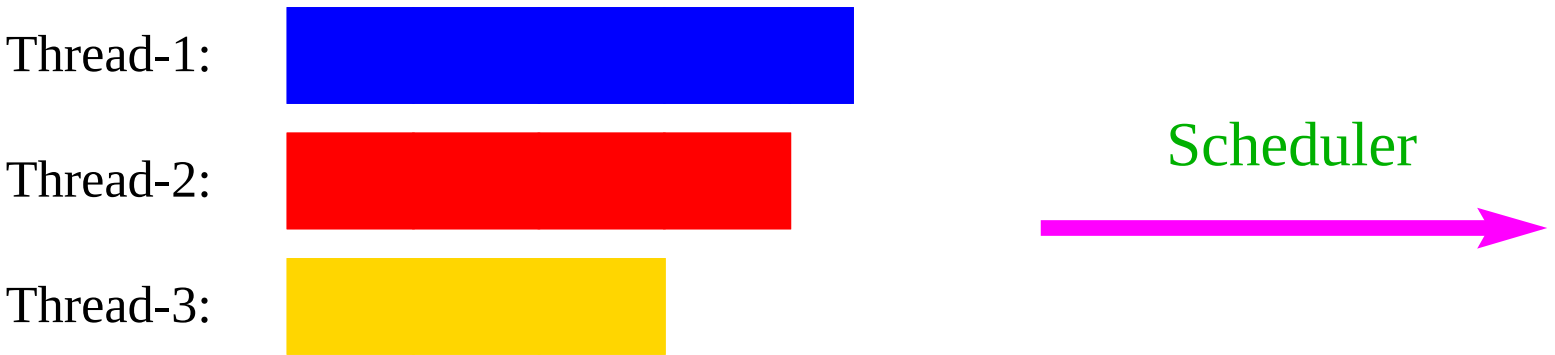
Eine andere Programm-Ausführung mag dagegen liefern:

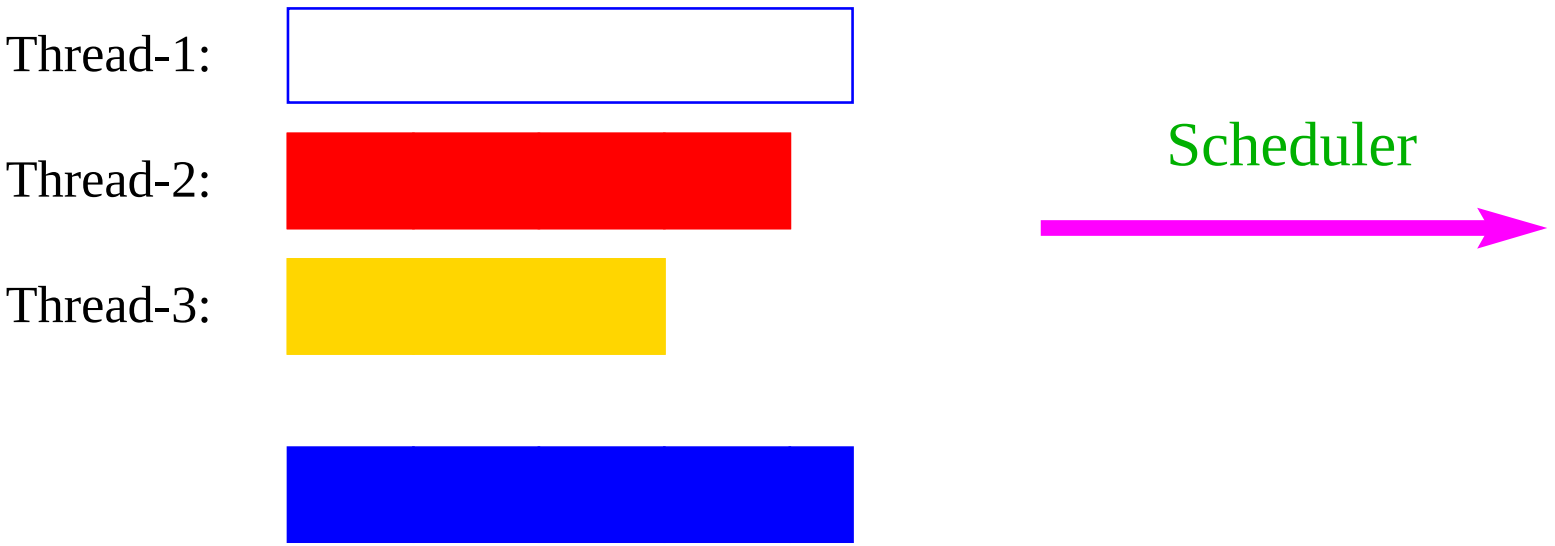


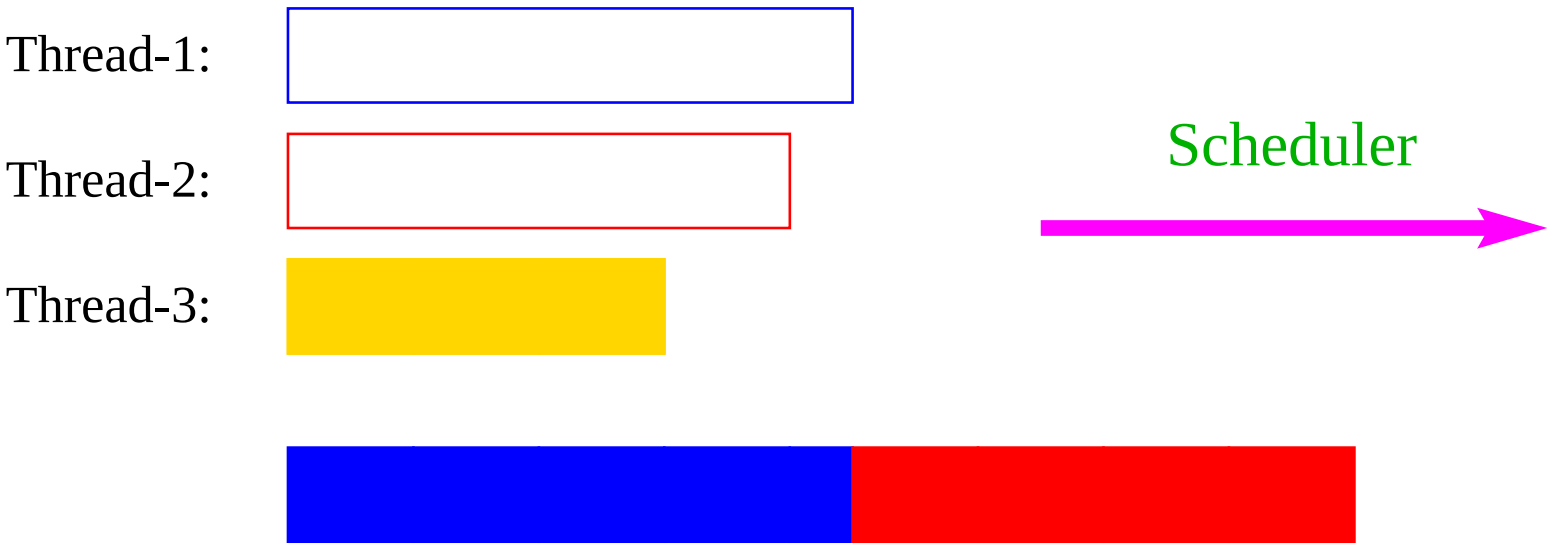
- Ein Zeitscheiben-Scheduler versucht, jeden Thread fair zu behandeln, d.h. ab und zu Rechenzeit zuzuordnen – egal, welche Threads sonst noch Rechenzeit beanspruchen.
- Kein Thread hat jedoch Anspruch auf einen bestimmten Time-Slice.
- Für den Programmierer sieht es so aus, als ob sämtliche Threads "echt" parallel ausgeführt werden, d.h. jeder über eine eigene CPU verfügt :-)

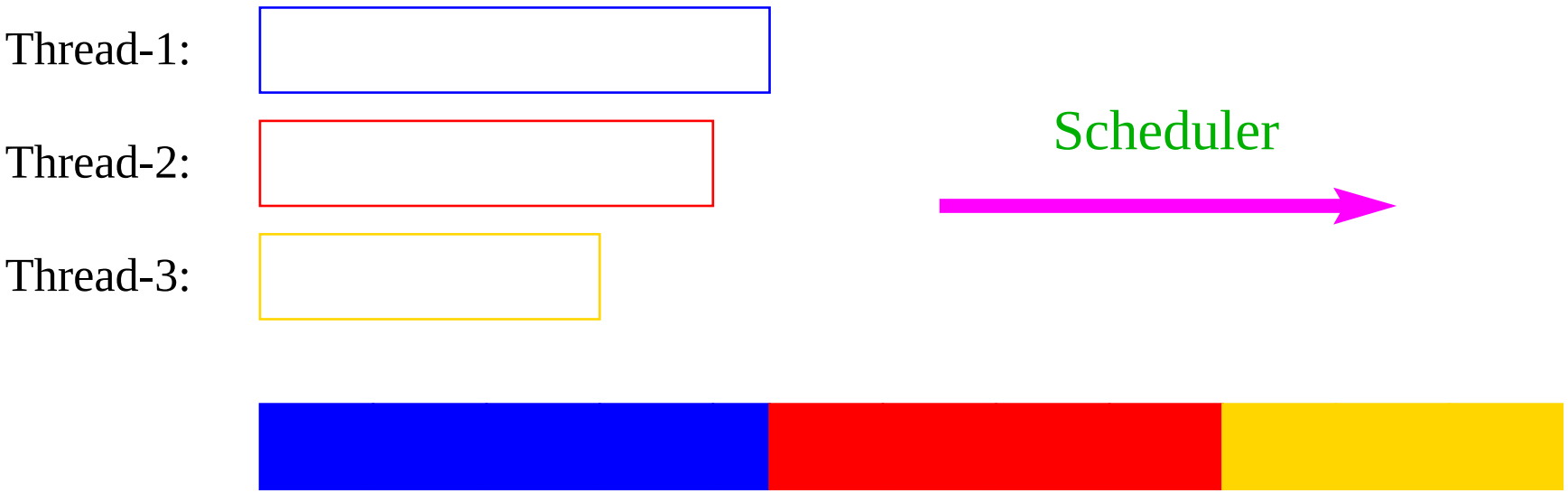
2. Naives Verfahren:

- Erhält ein Thread eine CPU, darf er laufen, so lange er will ...
- Gibt er die CPU wieder frei, darf ein anderer Thread arbeiten ...


Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler

Beispiel:

```
public class Start extends Thread {
public void run() {
   System.out.println("I'm running ...");
   while(true) ;
}
public static void main(String[] args) {
   (new Start()).start();
   (new Start()).start();
   (new Start()).start();
   System.out.println("main is running ...");
   while(true) ;
}
}  // end of class Start
```

... liefert als Ausgabe (bei naivem Scheduling und einer CPU) :

```
main is running ...
```

... liefert als Ausgabe (bei naivem Scheduling und einer CPU) :

```
main is running ...
```

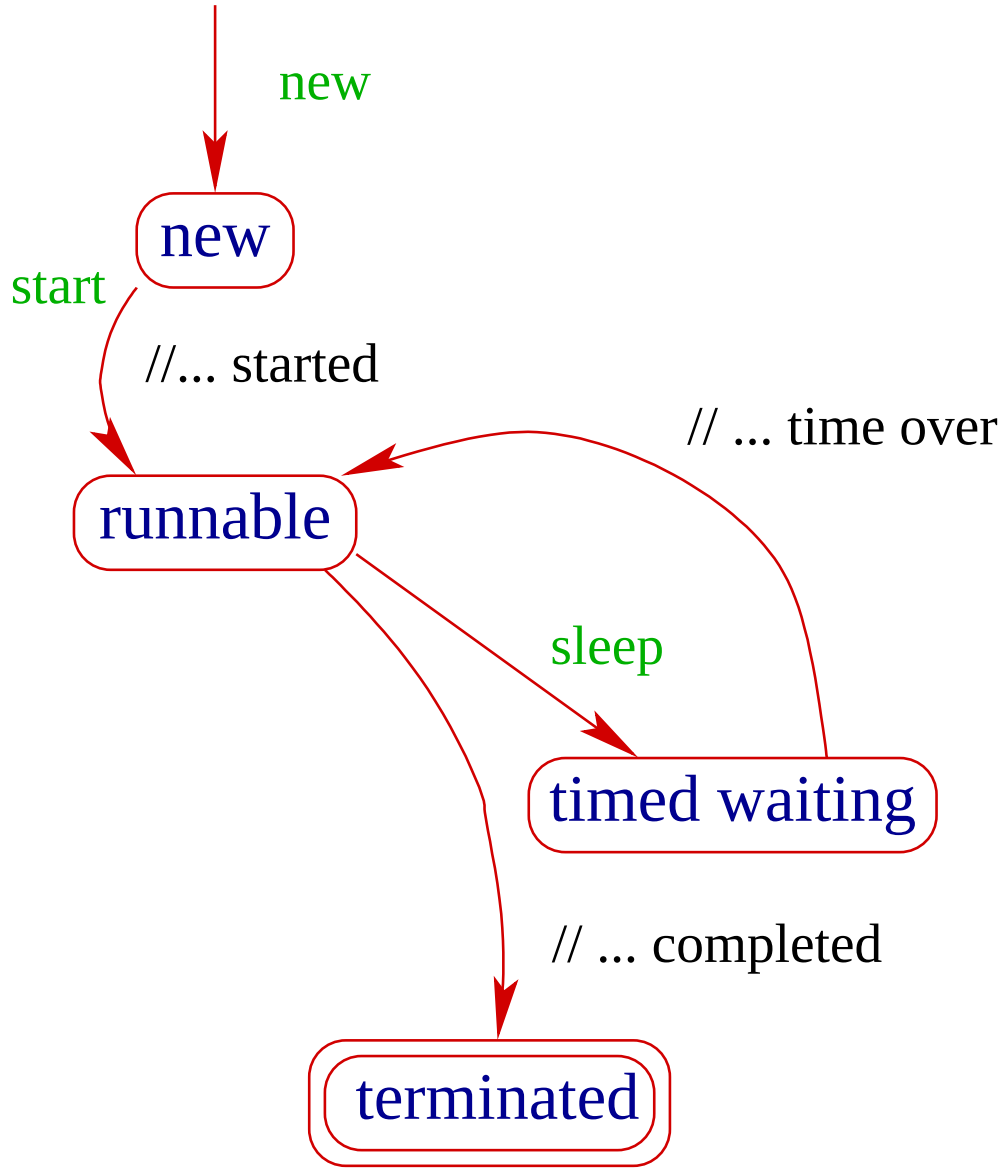
- Weil `main` nie fertig wird, erhalten die anderen Threads keine Chance, sie verhungern.
- Faires Scheduling mit einem Zeitscheiben-Verfahren würde z.B. liefern:

```
I'm running ...
main is running ...
I'm running ...
I'm running ...
```

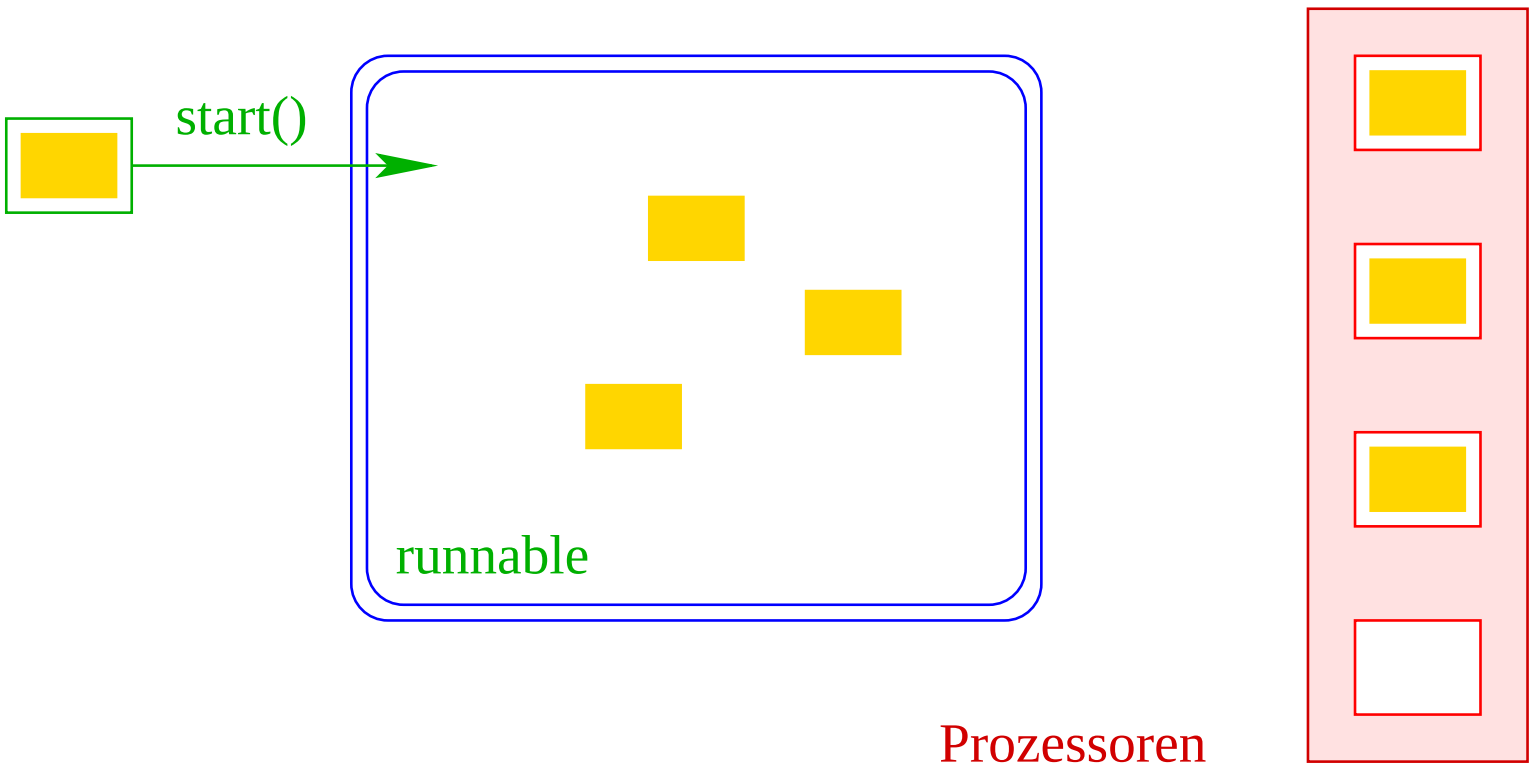
- Java legt nicht fest, wie intelligent der Scheduler ist.
- Die aktuelle Implementierung unterstützt faires Scheduling :-)
- Programme sollten aber für jeden Scheduler das gleiche Verhalten zeigen. Das heißt:
- ... Threads, die aktuell nichts sinnvolles zu tun haben, z.B. weil sie auf Verstreichen der Zeit oder besseres Wetter warten, sollten stets ihre CPU anderen Threads zur Verfügung stellen.
- ... Selbst wenn Threads etwas Vernünftiges tun, sollten sie ab und zu andere Threads laufen lassen.

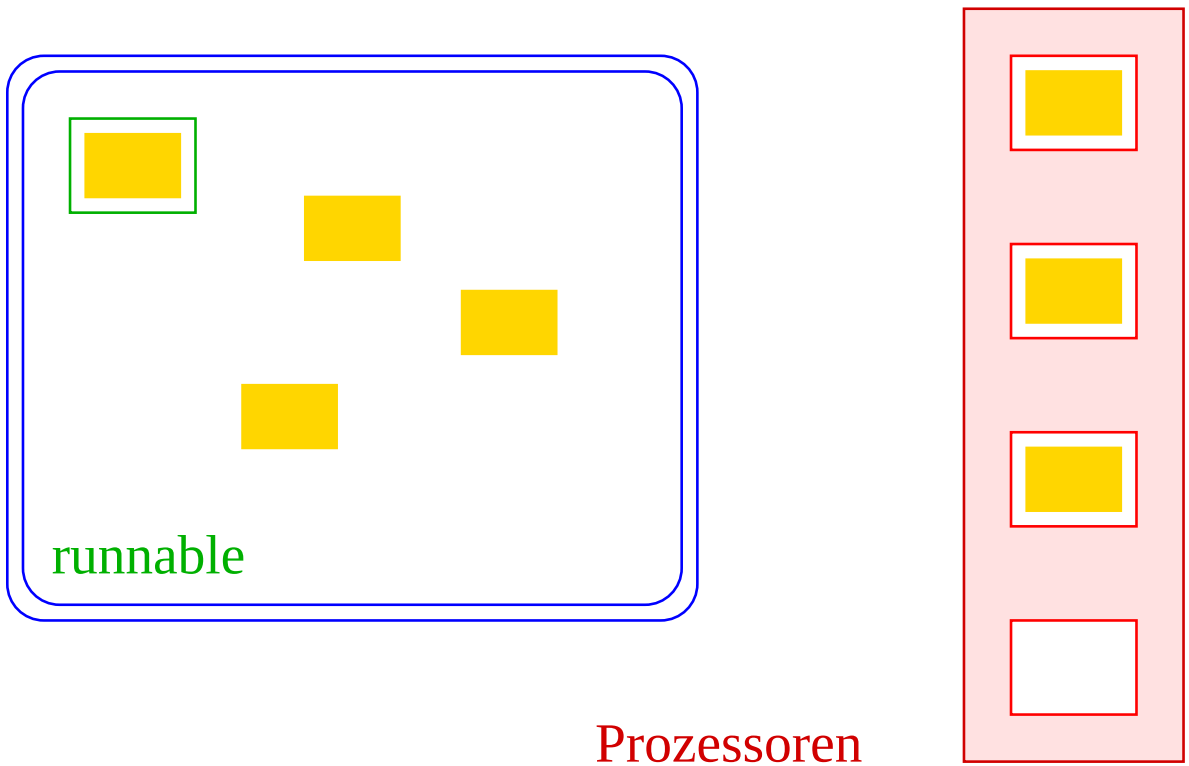
  (Achtung: Wechsel des Threads ist teuer!!!)
- Dazu verfügt jeder Thread über einen Zustand, der bei der Vergabe von Rechenzeit beücksichtigt wird.

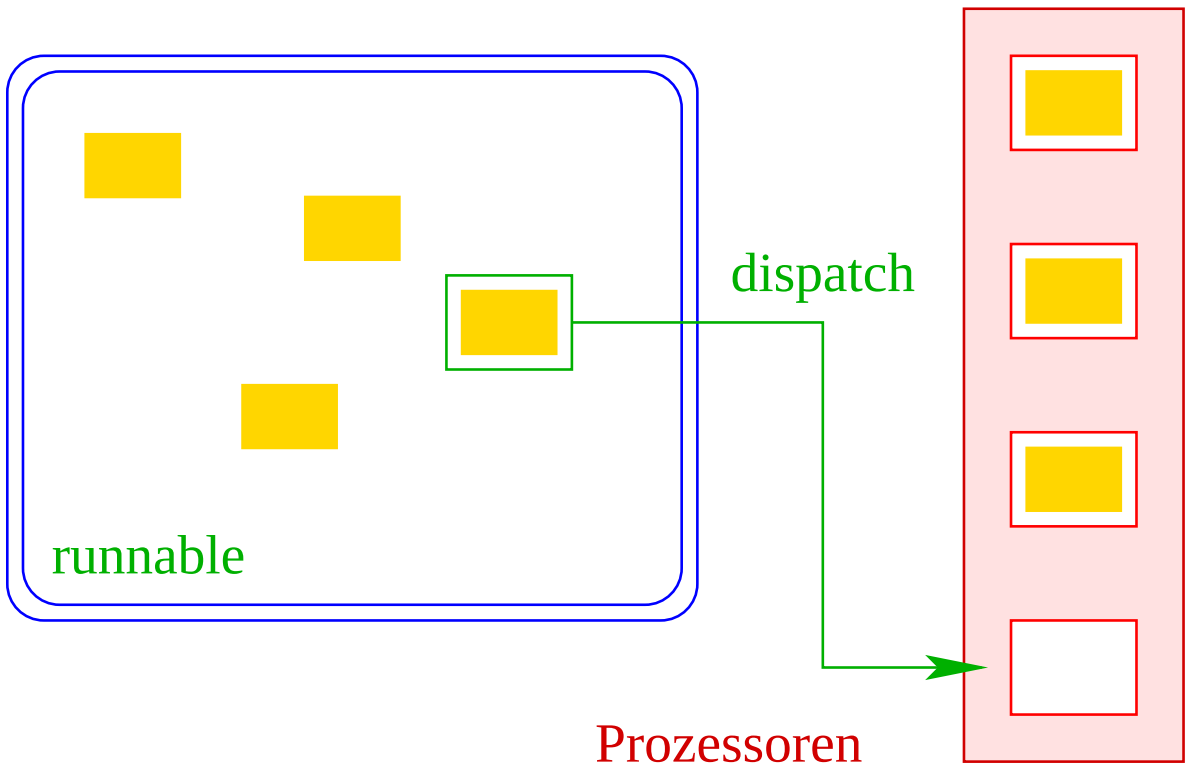
Einige Thread-Zustände:

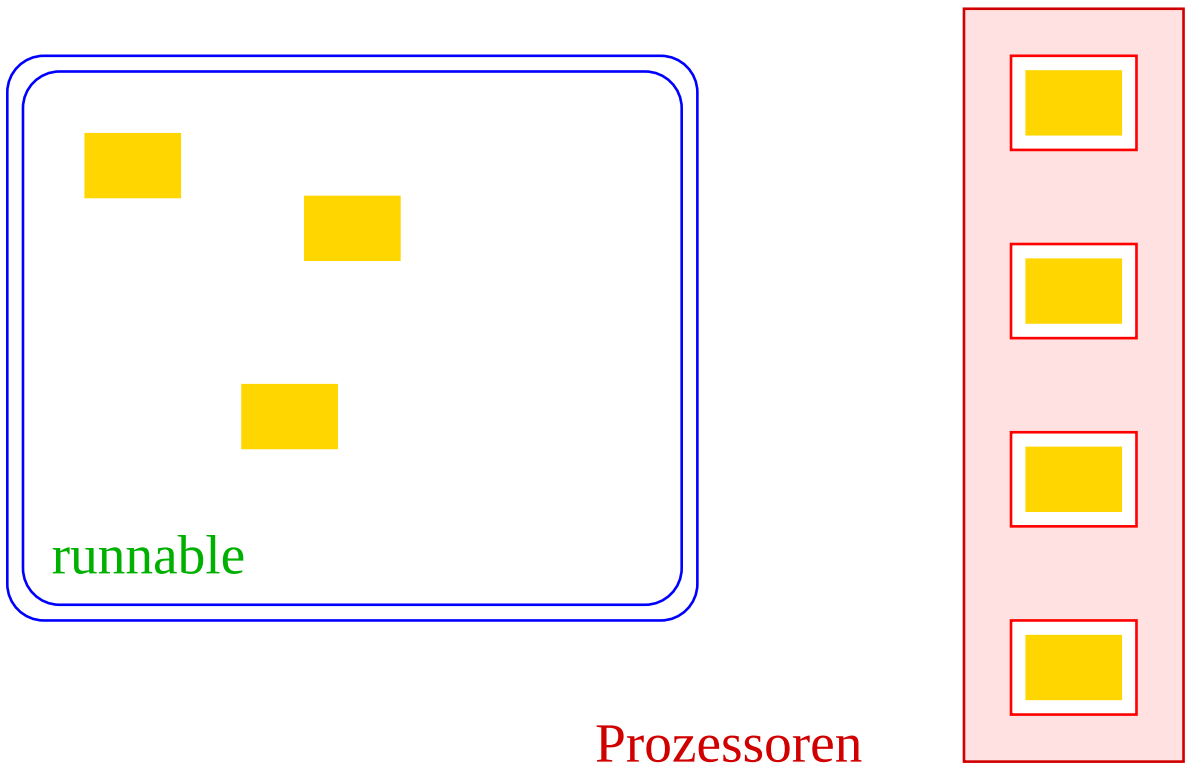


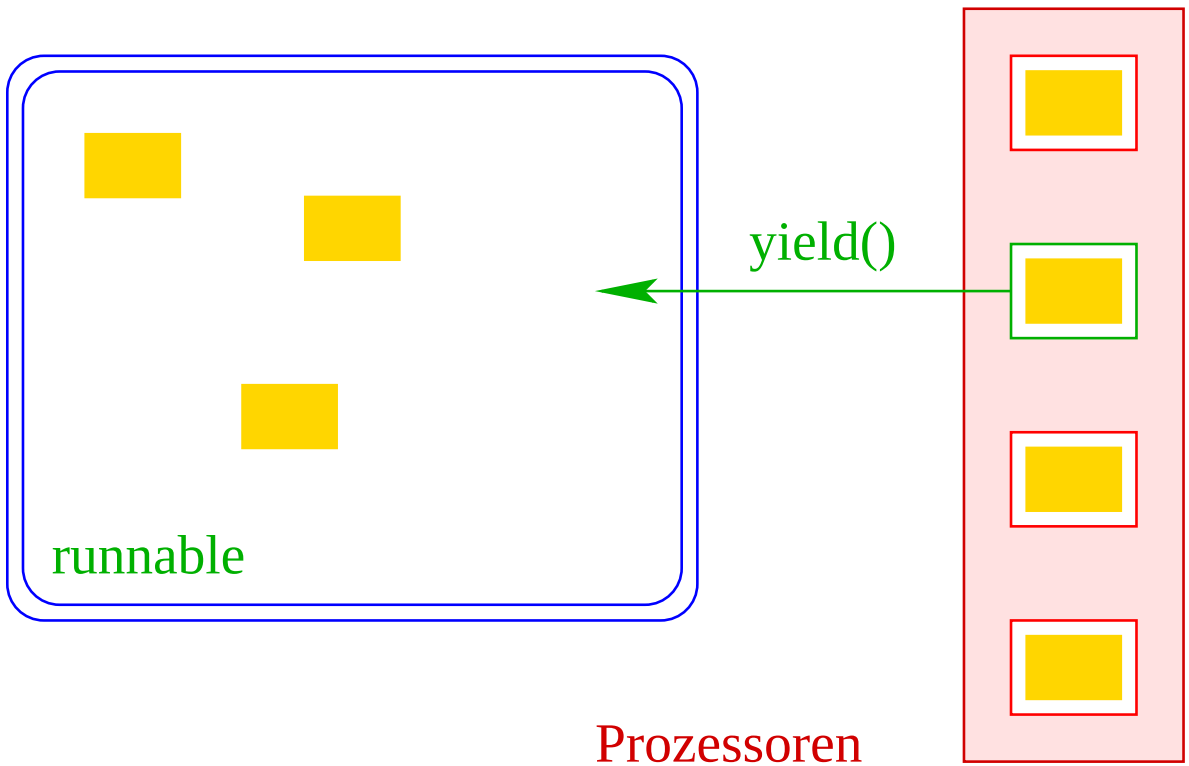
- Anlegen eines Threadobjekts setzt den Zustand auf new.
- `public void start();` legt einen neuen Thread an, setzt den Zustand auf runnable und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand runnable sind, Prozessoren zu ("dispatching").
- `public static void yield();` unterbricht die aktuelle Thread-Ausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand timed waiting wechselt.

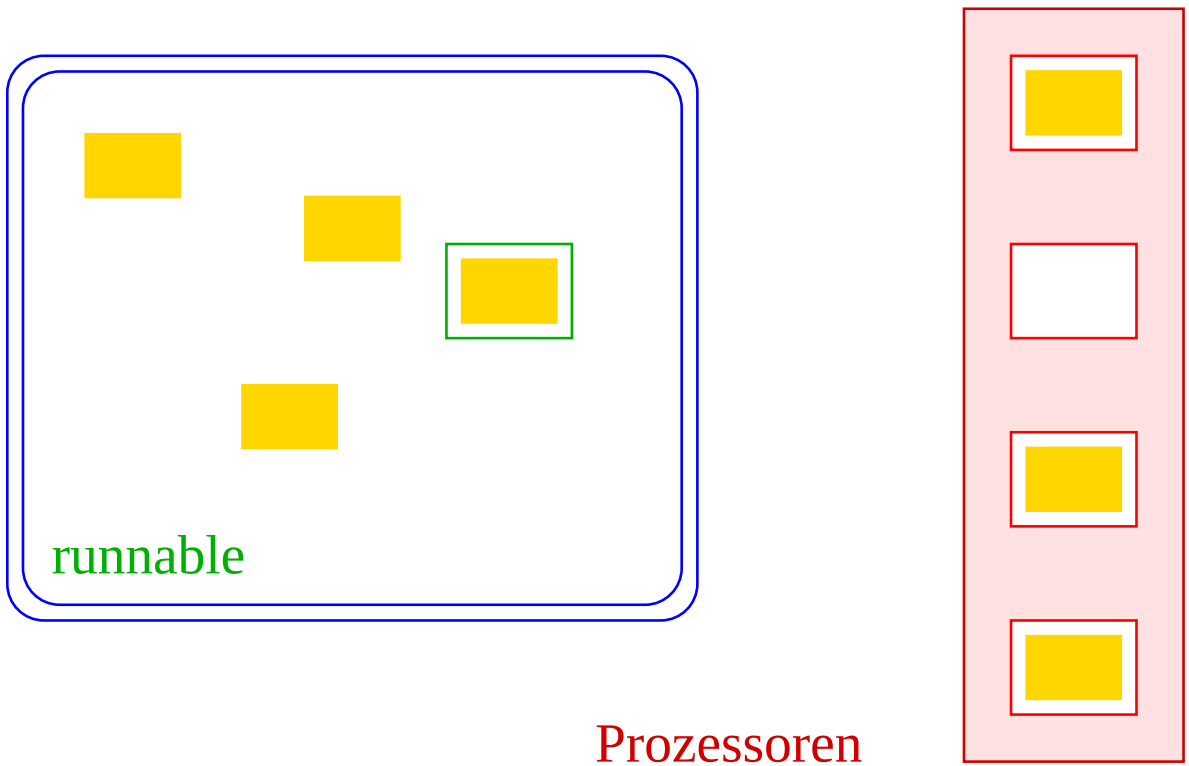

start()
runnable
Prozessoren

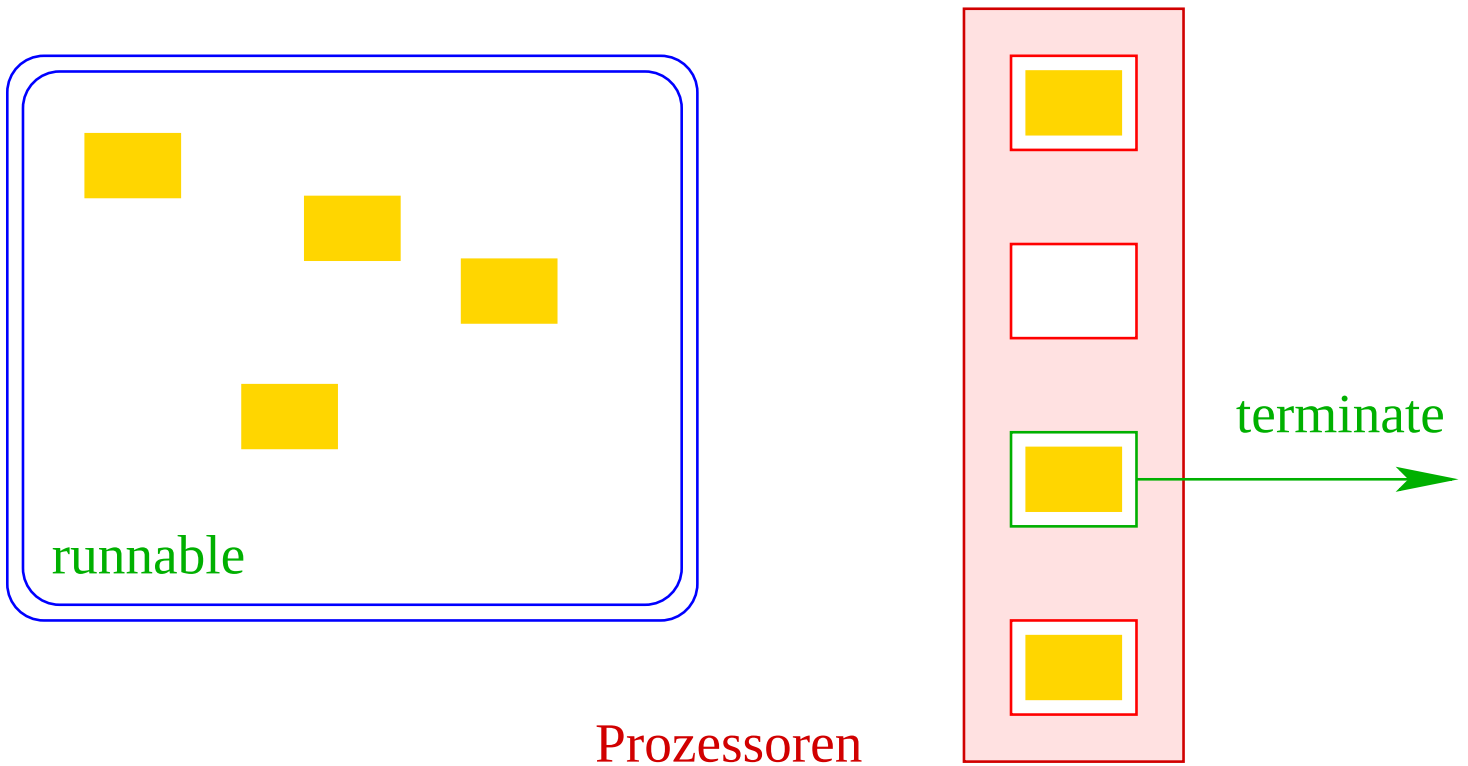

runnable
Prozessoren

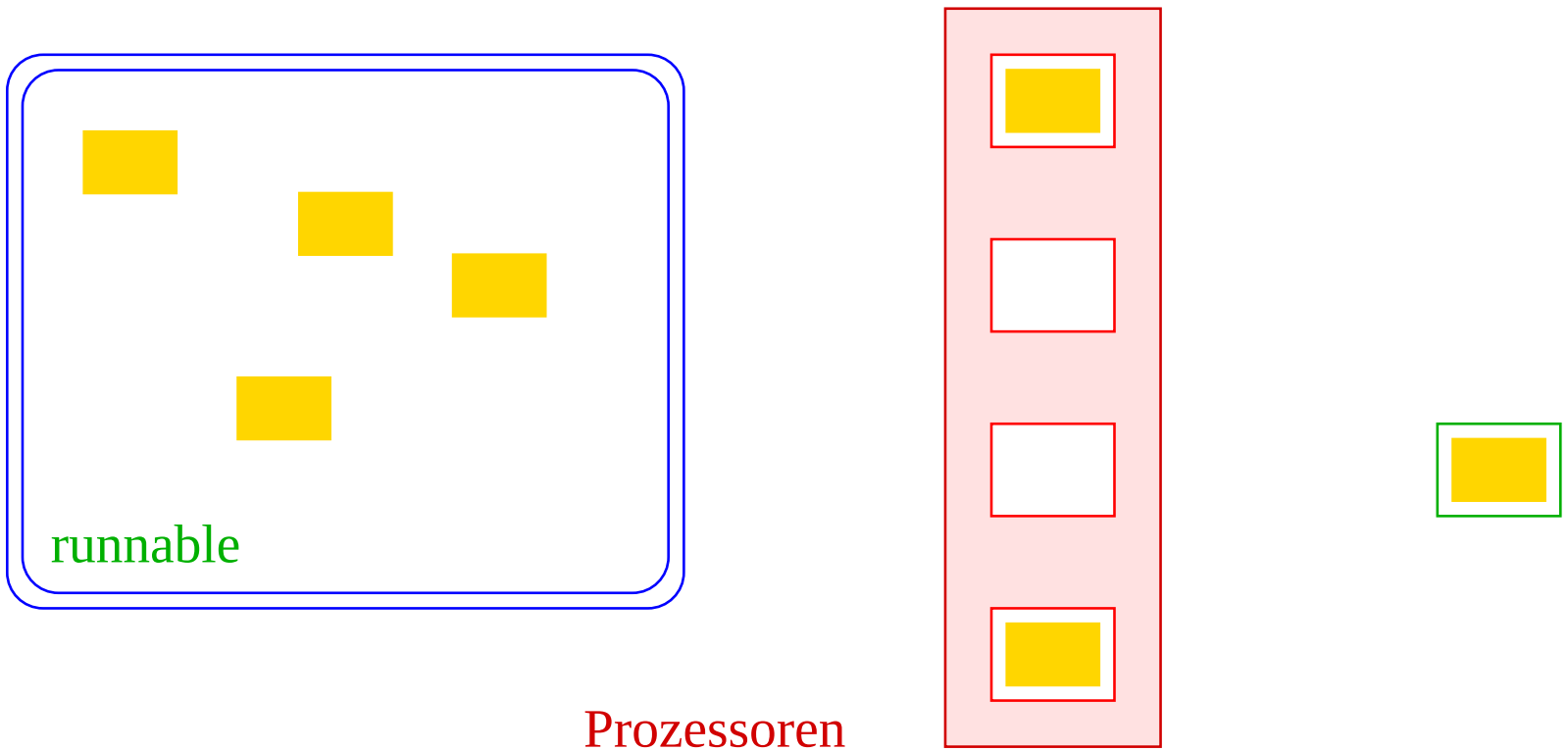

dispatch
runnable
Prozessoren


runnable
Prozessoren


yield()
runnable
Prozessoren


runnable
Prozessoren


runnable
terminate
Prozessoren


runnable
Prozessoren

1.1 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));
   }
```

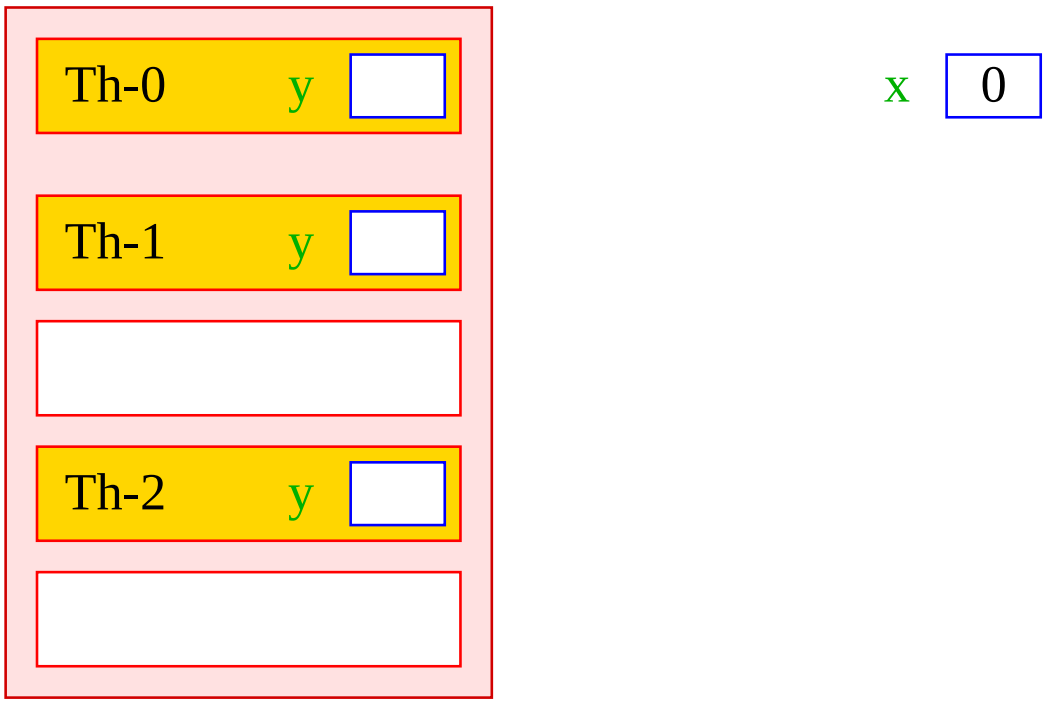
```
    ...
    public static void main(String[] args) {
       (new Thread(new Inc())).start();
       pause(2);
       (new Thread(new Inc())).start();
       pause(2);
       (new Thread(new Inc())).start();
    }
} // end of class Inc
```

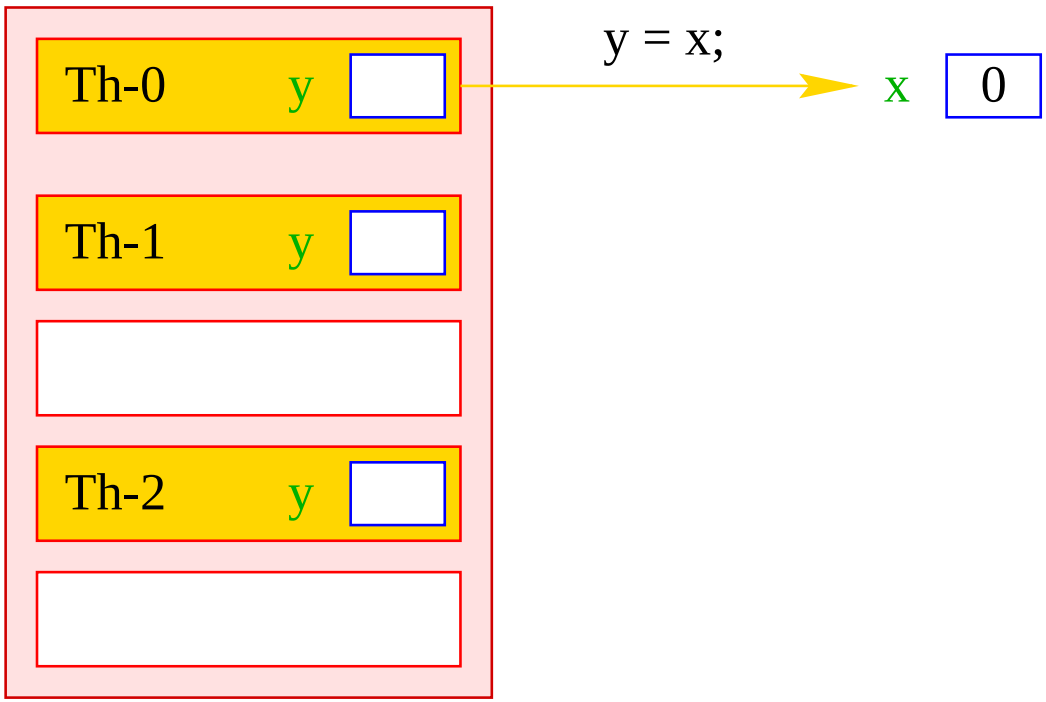
- `public static Thread currentThread();` liefert (eine Referenz auf) das ausführende Thread-Objekt.
- `public final String getName();` liefert den Namen des Thread-Objekts.
- Das Programm legt für drei Objekte der Klasse `Inc` Threads an.
- Die Methode `run()` inkrementiert die Klassen-Variable `x`.

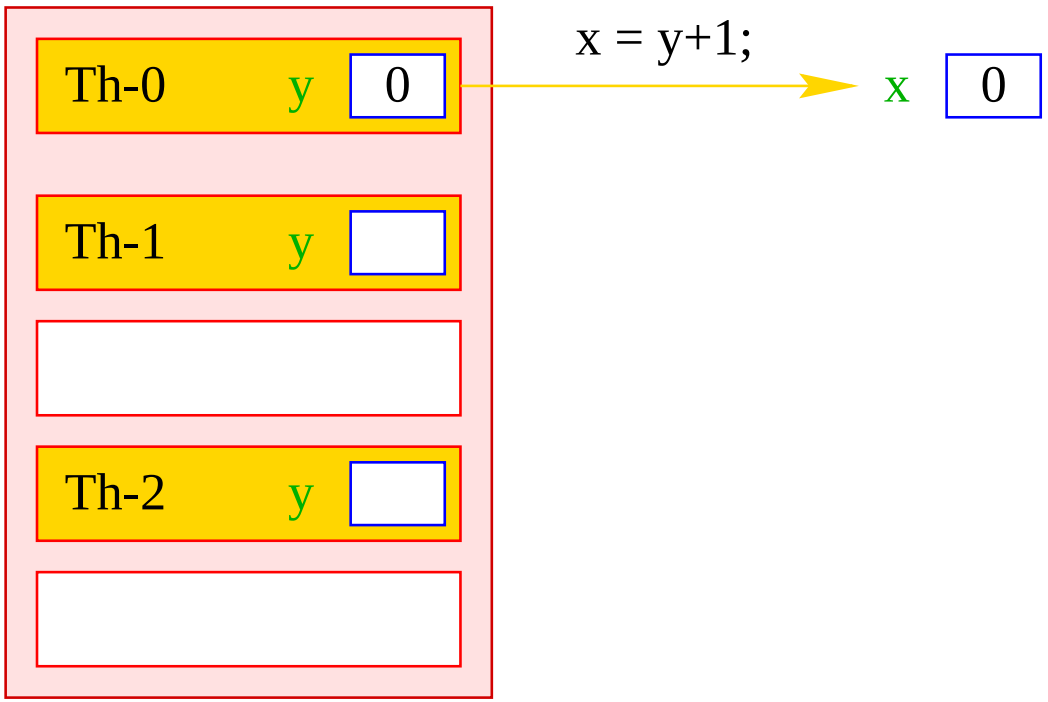
Die Ausführung liefert z.B.:

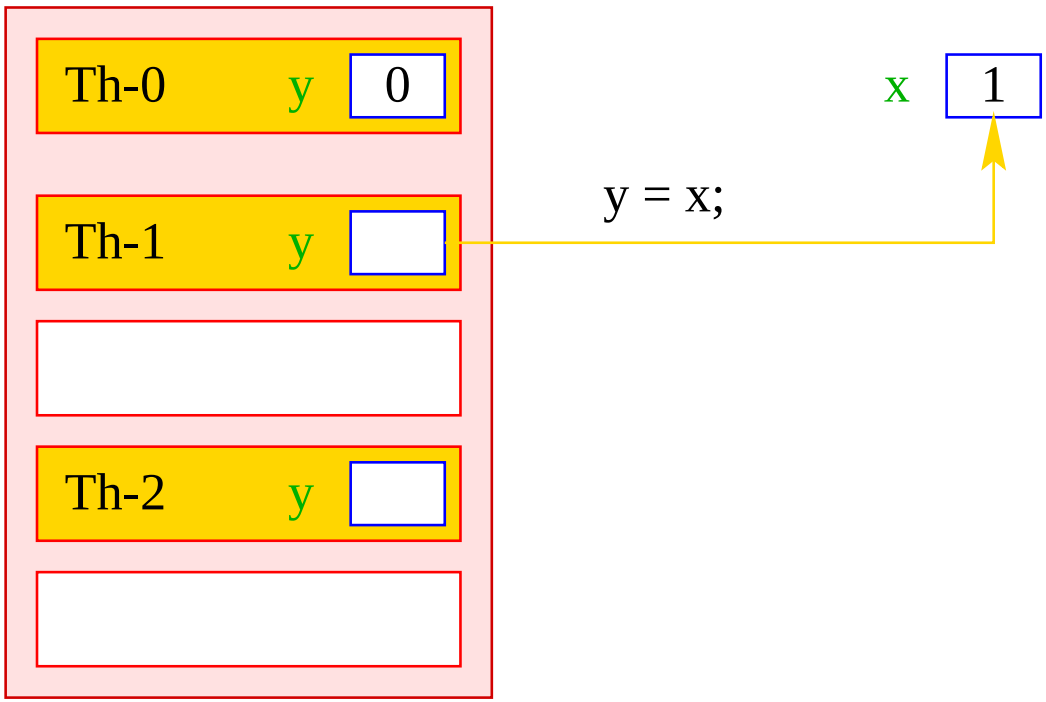
```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

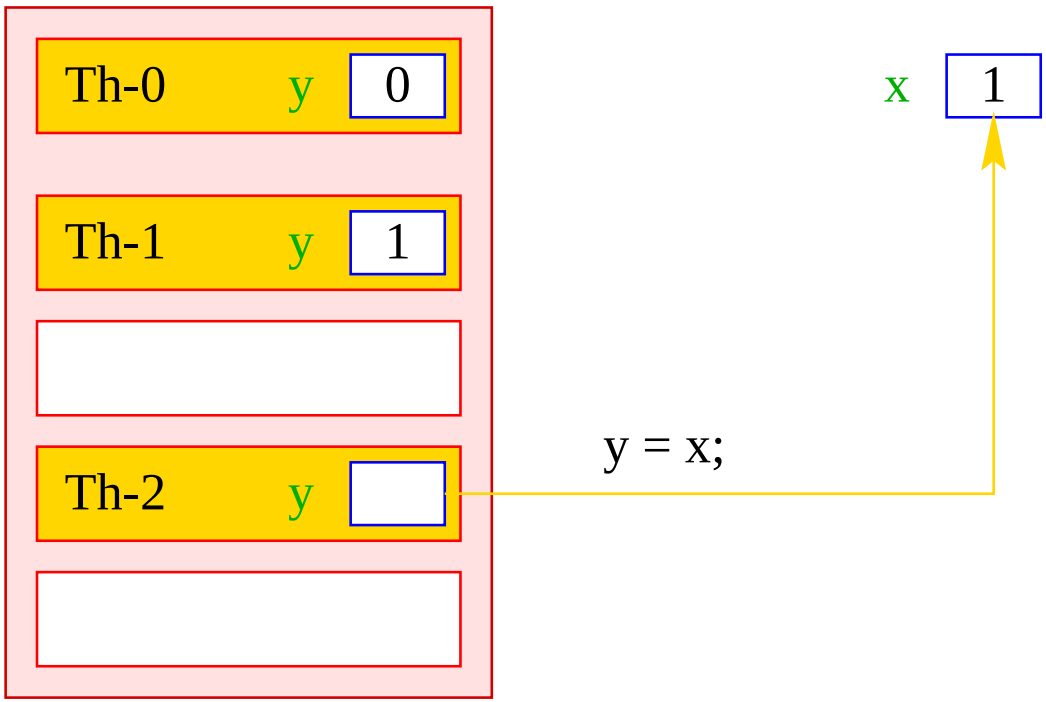
Der Grund:

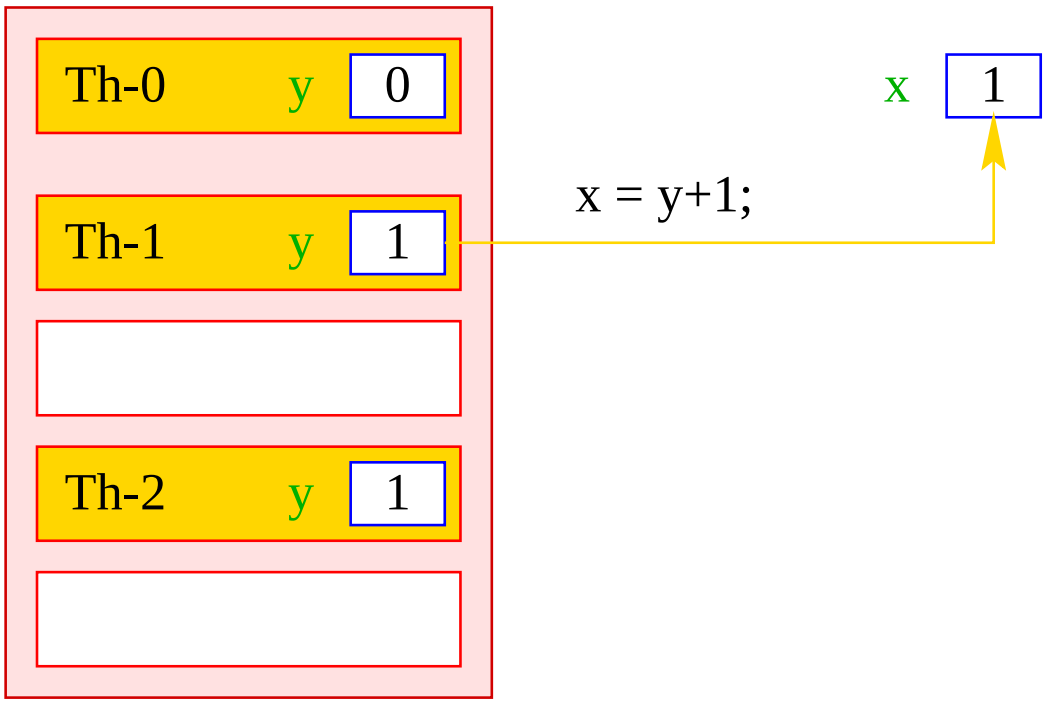

Th-0
y
Th-1
y
Th-2
y
x
0

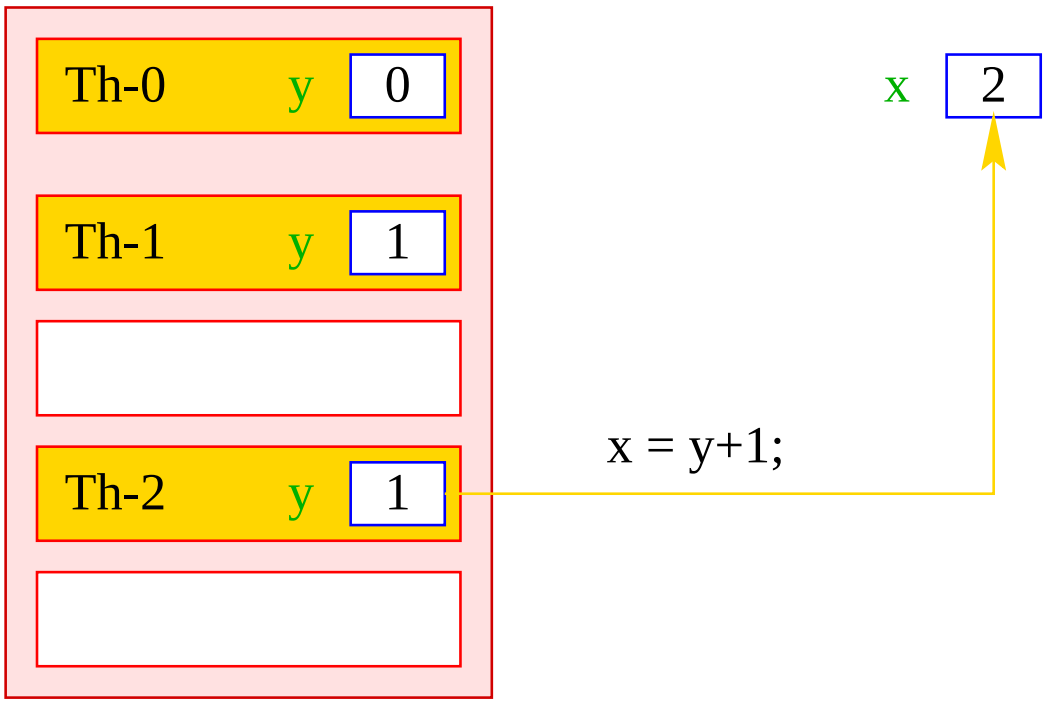

Th-0
y
y = x;
x
0
Th-1
y
Th-2
y

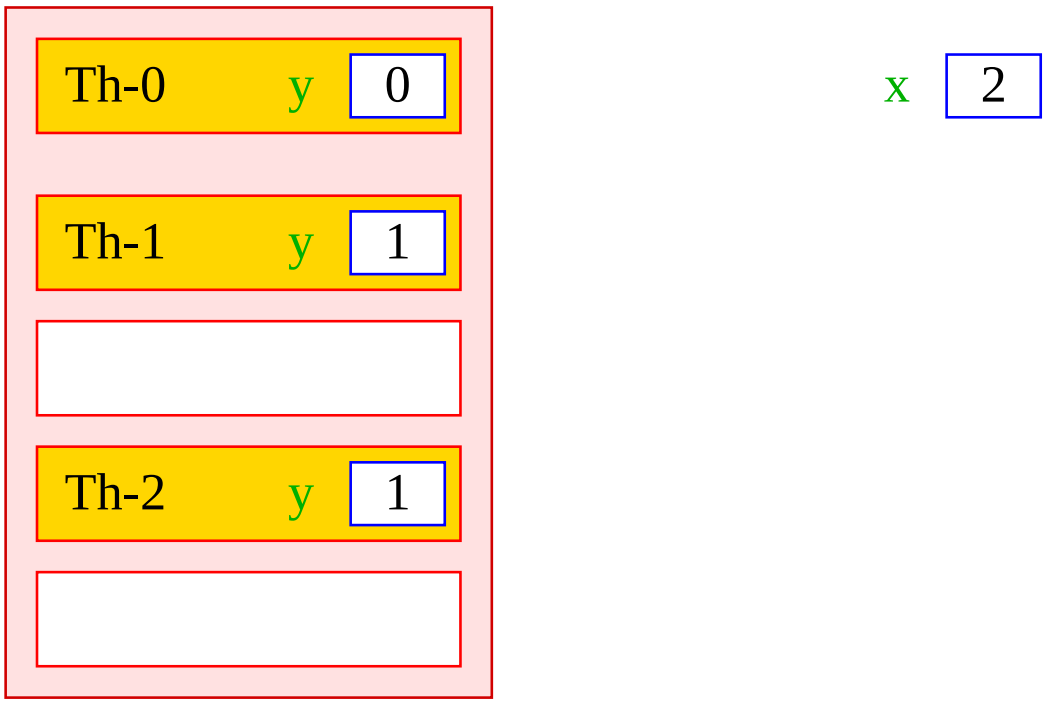

Th-0
y
0
x = y+1;
x
0
Th-1
y
Th-2
y


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


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

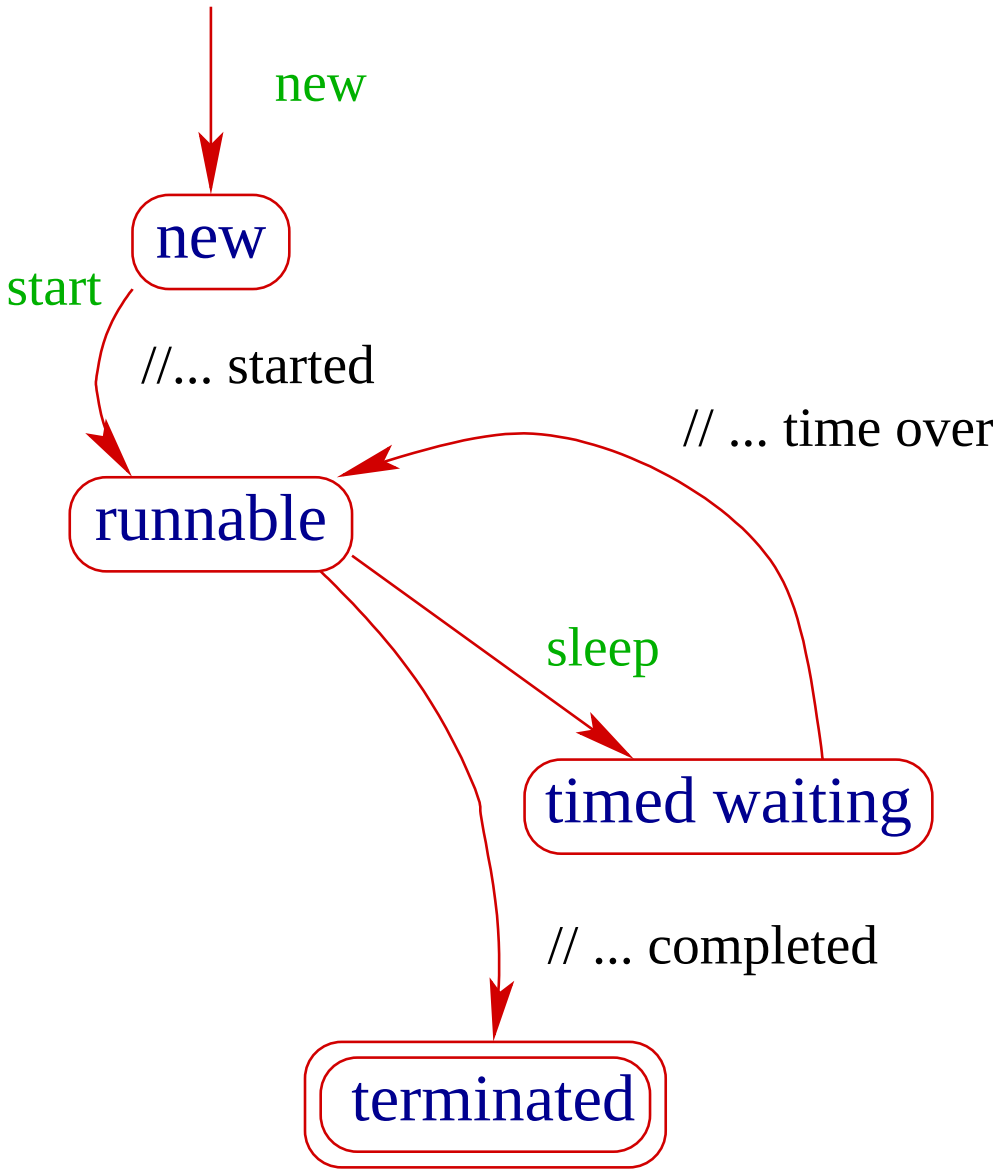

Th-0
y
0
Th-1
y
1
Th-2
y
1
x = y+1;
x
1

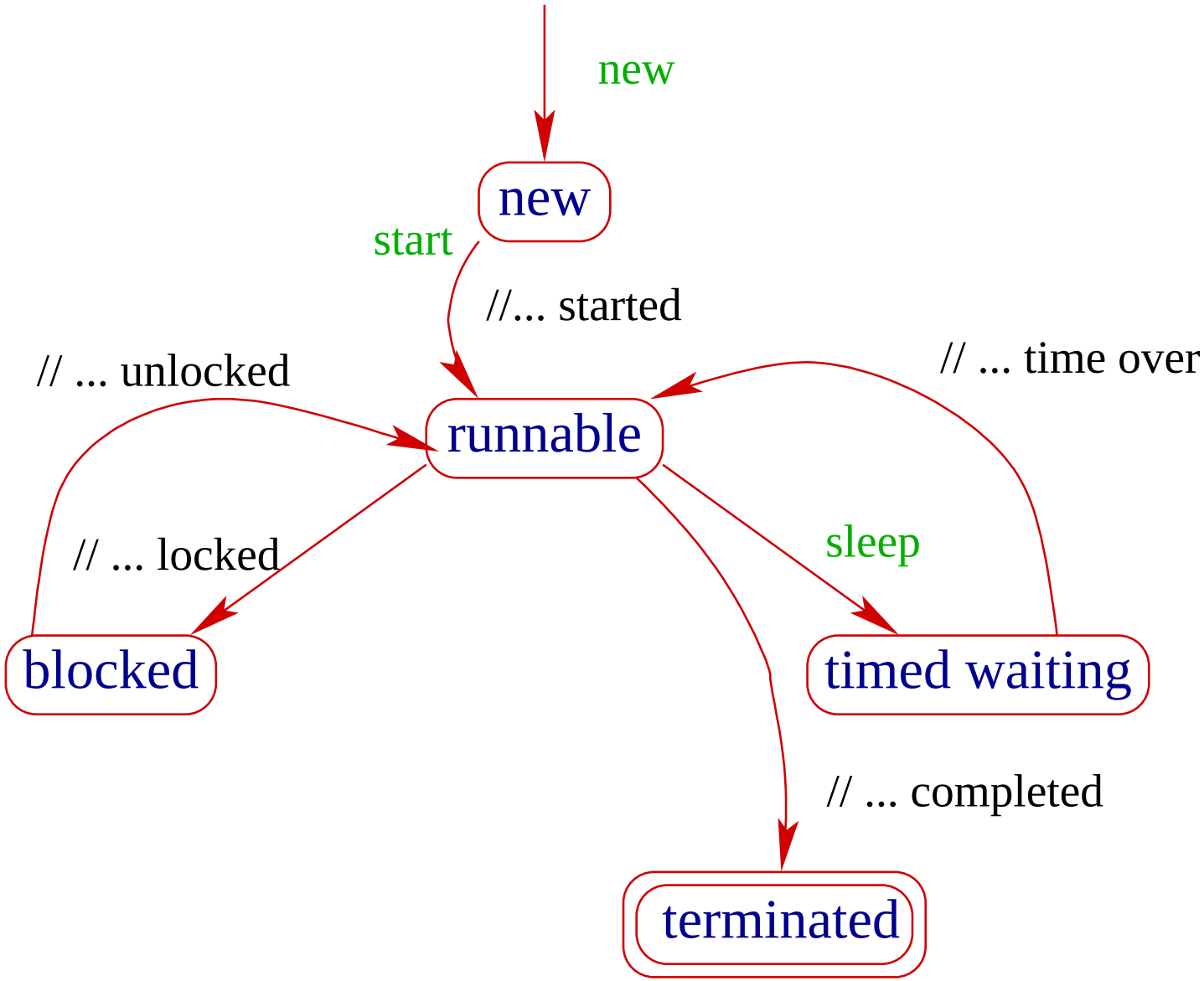

Th-0
y
0
Th-1
y
1
Th-2
y
1
x = y+1;
x
2


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.


new
new
start
//... started
// ... time over
runnable
sleep
timed waiting
// ... completed
terminated


new
new
start
//... started
// ... unlocked
// ... time over
runnable
// ... locked
sleep
blocked
timed waiting
// ... completed
terminated