Softwareentwicklung 4

Threading Einführung

Dominik Dolezal

Höhere Lehranstalt für Informationstechnologie

14. November 2016

Inhalt

tgm
Die Schule der Technik

Geteilter Speicher

tgm Die Schule der Technik

Wir haben gesagt, dass Threads einen gemeinsamen Adressraum haben
Konkret heißt das für uns:

- Sie teilen sich den dynamischen Speicherbereich (Heap)
  - D.h. **alle Objekte** sind für alle Threads zugänglich, wenn sie eine Referenz besitzen
  - In Java: von `new` bis zum Wegräumen durch die Garbage Collection
  - In Python: Ebenfalls Garbage Collection
- Sie teilen sich globale und statische Variablen (bzw. Klassenvariablen)
- Sie besitzen einen eigenen Stack, d.h. lokale Variablen (in Funktionen / Methoden) werden nicht geteilt
- Sie besitzen eigene Kopien von (globalen) Variablen, die mit `threading.local` erstellt wurden (Python) bzw. `ThreadLocal` (Java)

Konkurrierende Zugriffe

tgm Die Schule der Technik

Thread 1	Thread 2
`# counter (global): 10`	
`curValue = SimpleCounter.counter`	
`# curValue (lokal): 10`	
	`# counter (global): 10`
	`curValue = SimpleCounter.counter`
	`# curValue (lokal): 10`
`SimpleCounter.counter = curValue + 1`	
`# counter (global): 11`	
	`SimpleCounter.counter = curValue + 1`
	`# counter (global): 11`

Obwohl zweimal addiert wurde, ist `counter` nur um 1 erhöht worden! Die zweite Addition hat einen veralteten Wert gelesen, weshalb das Ergebnis der ersten Addition überschrieben wurde!

Konkurrierende Zugriffe

tgm
Die Schule der Technik

- Der finale Wert ist unvorhersagbar und hängt also von der Reihenfolge ab, in welcher die Operationen ausgeführt werden
- Diese ungewollten Effekte nennt man auch *race conditions*
- Achtung: Der Effekt kann auch auftreten, wenn z.B. `counter=counter+1` oder `counter++` verwendet wird
  - Obwohl es sich um nur eine Zeile handelt, sind die Befehle trotzdem nicht atomar
  - `counter++` besteht nach wie vor aus einer Leseoperation, einer Additionsoperation und einem Schreibzugriff
- Bei Mehrkernsystemen verstärkt sich dieser Effekt klarerweise

Thread Synchronisation

tgm
Die Schule der Technik

Es gibt mehrere Möglichkeiten, Threads sicher zu gestalten:

- Locks (bzw. Mutexe)
- Events und Bedingungsvariablen
- Queues
- Atomare Variablen (in Standard-Python nicht vorhanden)

Lock bzw. Mutex

tgm
Die Schule der Technik

```
# Lock sperren (falls frei), ansonsten
# warten, bis sie frei ist
with SimpleCounter.lock:
  # --- Beginn kritischer Abschnitt ---
  curValue = SimpleCounter.counter
  print("Current Value:" + str(curValue))
  SimpleCounter.counter = curValue + 1
  # --- Ende kritischer Abschnitt ---
```

- Es wird ein sogenannter **kritischer Bereich** definiert
- Im kritischen Bereich kann sich nur 1 Thread gleichzeitig befinden
- Dort werden jene Operationen durchgeführt, die nicht unterbrochen werden dürfen

Events

tgm
Die Schule der Technik

- Sind in Python Objekte der Klasse `threading.Event`
- Werden verwendet, um andere Threads über das Eintreten eines Ereignisses zu benachrichtigen
- Zwei wichtige Methoden: `wait()` und `set()`
  - `wait()` lässt den aktuellen Thread auf das Eintreten des Events warten
  - `set()` löst das Event aus – alle wartenden Threads werden (gleichzeitig) aufgeweckt
- Anwendungsfälle: Warten auf Initialisierung, Eingaben, Zwischenergebnisse, Benachrichtigungen, Barrieren

Bedingungsvariablen

tgm
Die Schule der Technik

- Englisch: Condition Variable
- Kombination aus Event und Lock
  - Ein Event weckt alle Threads, die auf dieses Event warten, auf und alle beginnen anschließend gleichzeitig zu arbeiten
  - Eine Lock ist nicht für Benachrichtigungen geeignet, sondern sperrt kritische Abschnitte
  - Bedingungsvariablen haben eingebaute Locking- und Benachrichtigungssystem
- Bedingungsvariablen sind „Higher Level“-Konzepte, während Locks und Events eher „Low Level“ sind
- Anwendungsfälle: Erzeuger-Verbraucher-Muster (Englisch: Consumer-Producer-Pattern), Nachrichtenaustausch

Bedingungsvariablen

Producer

```
while True:
  with self.condition:
    number = number + 1
    self.numbers.append(number)
    self.condition.notify()
  time.sleep(0.01)
```

Consumer

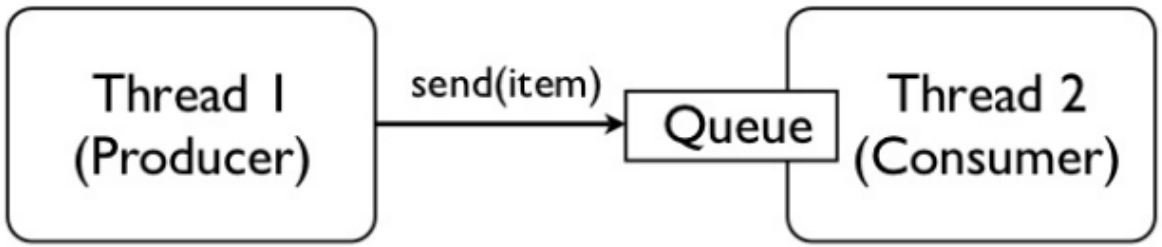
```
while True:
  with self.condition:
    while True:
      if self.numbers:
        print(self.numbers.pop())
      else:
        break
      self.condition.wait()
```

- Zuerst wird immer die Lock der Bedingungsvariable gesperrt (**with**)
- Anschließend kann der Consumer über `wait()` auf den Eintritt der Bedingung warten: In dieser Zeile legt er sich schlafen
- Über `notify()` weckt der Producer den wartenden Thread auf und benachrichtigt ihn somit darüber, dass in der geteilten Datenstruktur eine neue Zahl liegt

Queues

tgm
Die Schule der Technik

- Das Erzeuger-Verbraucher-Muster wird so oft benötigt, dass es in Python eine direkte Implementierung über Queues erhielt
- Anstatt sich Daten zu teilen, werden Nachrichten verschickt



Queues: main-Skript

tgm
Die Schule der Technik

Producer

```
while True:
    number = number + 1
    self.queue.put(number)
    self.queue.join()
```

Consumer

```
while True:
    number = self.queue.get()
    print("Consumer: %d" % number)
    self.queue.task_done()
```

- Es wird hier keine Lock benötigt!
- Alle Methoden von Queue sind threadsicher
- Solange man nur die Basisfunktionalitäten einer Queue verwendet, muss keine zusätzliche Thread-Synchronisation eingebaut werden!

Probleme der Nebenläufigkeit

tgm
Die Schule der Technik

- Wir haben schon von *race conditions* gehört
- Diese können durch die vorgestellten Methoden der Thread Synchronisation verhindert werden
- Dadurch können allerdings bei unbedachter Implementierung wieder neue Probleme entstehen
- Hierfür wollen wir unser Zähler-Beispiel erweitern

Probleme der Nebenläufigkeit

tgm
Die Schule der Technik

```
class Counter(object):
    def __init__(self):
        self.counter = 0
    def inc(self):
        self.counter += 1
        return self.counter
```

Eine einfache Zählerklasse, welche eine Instanzvariable erhöhen und zurückliefern kann

Probleme der Nebenläufigkeit

tgm Die Schule der Technik

```
class SimpleCounterThread1(threading.Thread):
    def __init__(self, counter1, counter2, lock1, lock2):
        threading.Thread.__init__(self)
        self.counter1 = counter1
        self.counter2 = counter2
        self.lock1 = lock1
        self.lock2 = lock2

    def run(self):
        for i in range(1000):
            with self.lock1:
                print("Counter1:" + str(self.counter1.inc()))
                with self.lock2:
                    print("Counter2:"+str(self.counter2.inc()))
```

Ein einfacher Thread, welcher zuerst `lock1` sperrt und `counter1` erhöht und anschließend `lock2` sperrt und `counter2` erhöht

Probleme der Nebenläufigkeit

tgm
Die Schule der Technik

```
class SimpleCounterThread2(threading.Thread):
    def __init__(self, counter1, counter2, lock1, lock2):
        threading.Thread.__init__(self)
        self.counter1 = counter1
        self.counter2 = counter2
        self.lock1 = lock1
        self.lock2 = lock2

    def run(self):
        for i in range(1000):
            with self.lock2:
                print("Counter2:" + str(self.counter2.inc()))
                with self.lock1:
                    print("Counter1:"+str(self.counter1.inc()))
```

Noch ein einfacher Thread, allerdings erhöht dieser zuerst counter2 und dann erst counter1

tgm
Die Schule der Technik

Sieht hier jemand ein Problem?

Probleme der Nebenläufigkeit

tgm
Die Schule der Technik

```
Counter1:1
Counter2:1
Counter1:2
Counter2:2
Counter1:3
Counter2:3
Counter1:4
Counter2:4
Counter1:5
Counter2:5
```

- Das Programm bleibt an beliebigen Zeitpunkten einfach stehen
- Alle Threads sind blockiert und der gesamte Prozess terminiert nicht mehr
- Was passiert hier?

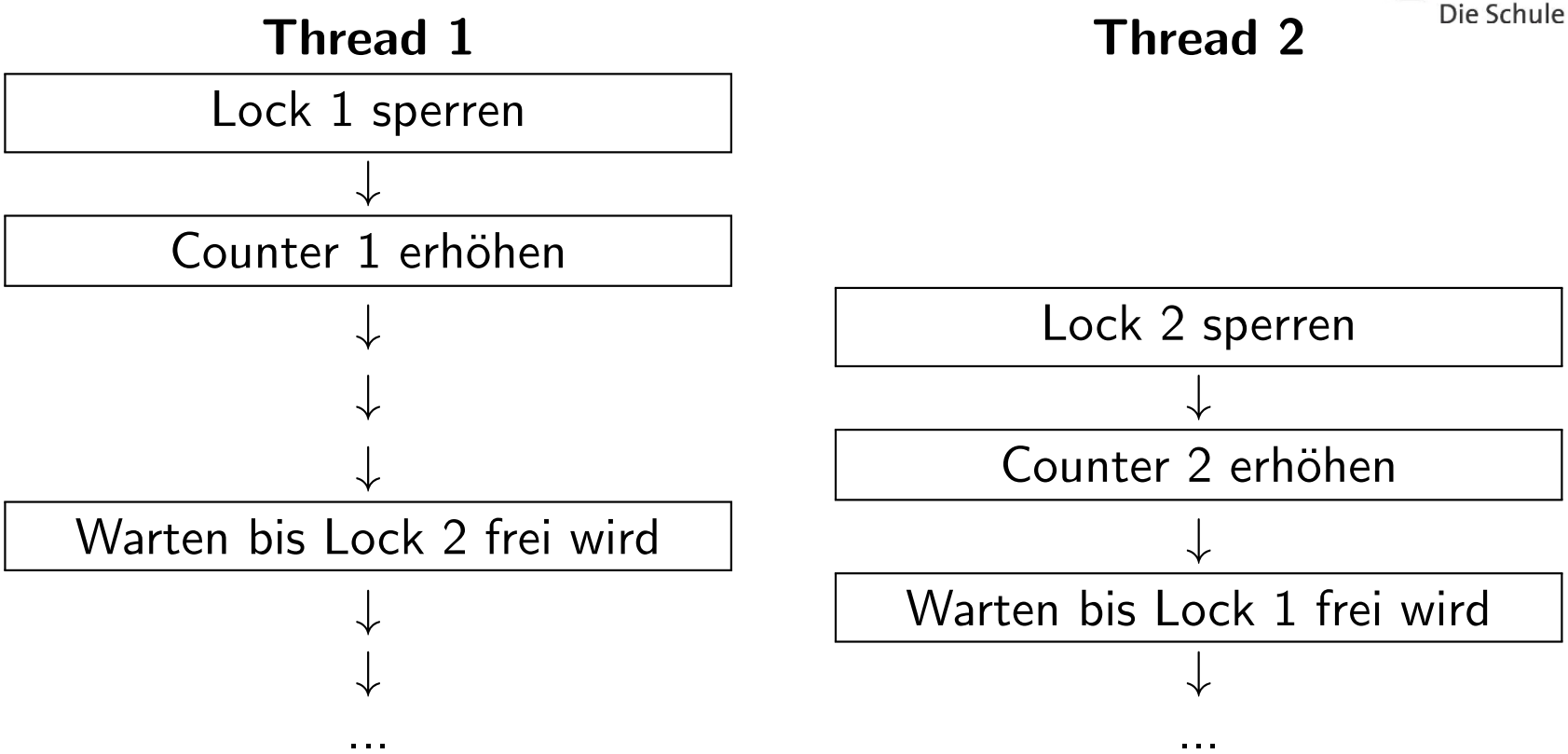
Probleme der Nebenläufigkeit

tgm
Die Schule der Technik

Thread 1	Thread 2
`with self.lock1:`	
`  self.counter1.inc()`	
	`with self.lock2:`
	`  self.counter2.inc()`
`  with self.lock2:`	
`  # Lock2 bereits gesperrt => warten`	
	`  with self.lock1:`
	`  # Lock1 bereits gesperrt`

Probleme der Nebenläufigkeit

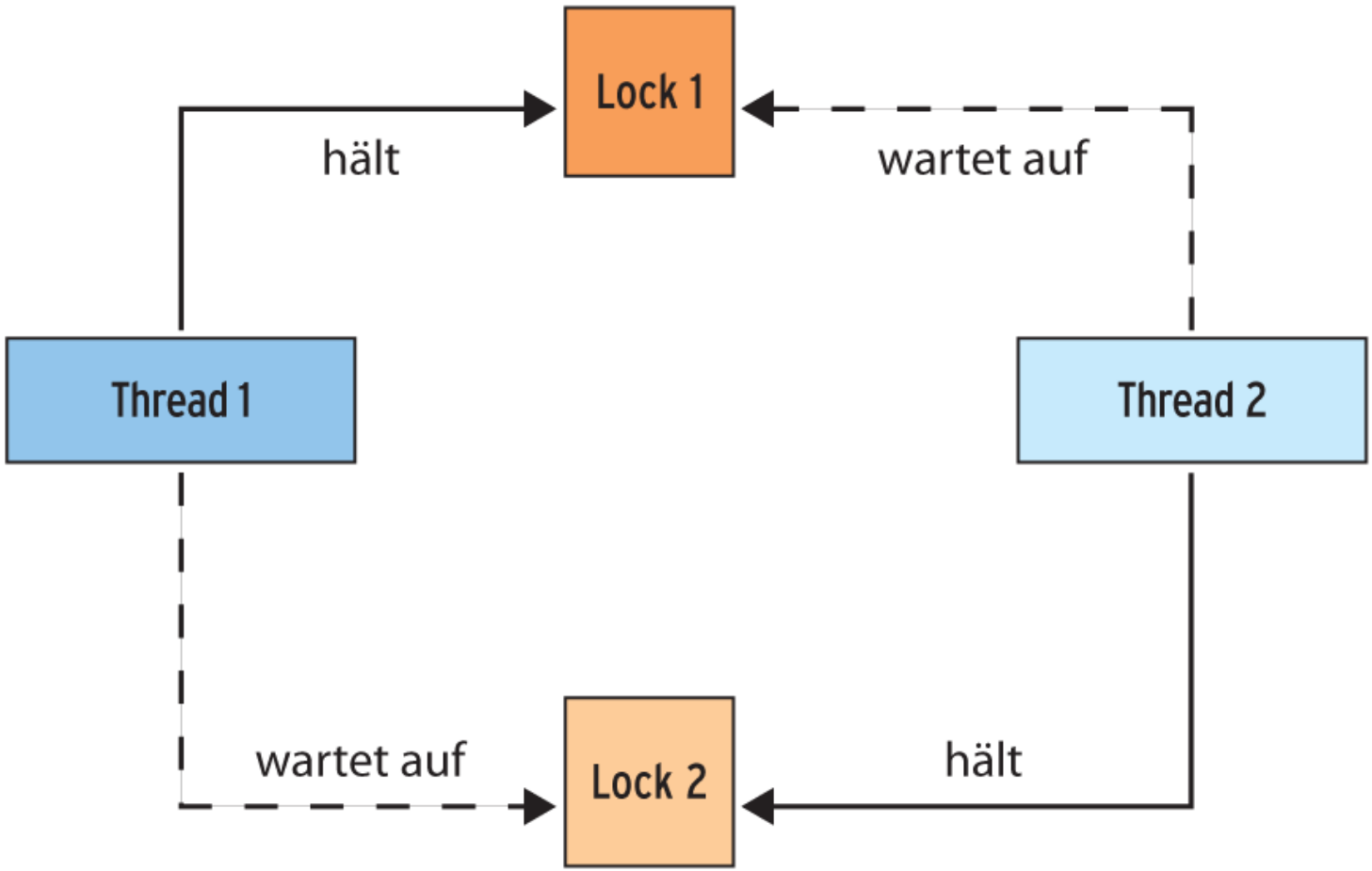
tgm Die Schule der Technik



Thread 1 hat somit den exklusiven Zugriff auf Lock 1 und Thread 2 hat den exklusiven Zugriff auf Lock 2, sie warten bis in alle Ewigkeit aufeinander.

Deadlock

- Diesen Effekt nennt man *Deadlock*



Deadlock-Bedingungen

tgm
Die Schule der Technik

Deadlocks treten auf, wenn *alle* vier Bedingungen zutreffen:

- *Mutual Exclusion*: Exklusiver Zugriff auf Ressourcen ist möglich
- *Hold and Wait*: Ressourcen können blockiert werden, während der Prozess (Thread) selbst auf eine andere Ressource wartet
- *No Preemption*: Zugewiesene Ressourcen können einem Prozess nicht mehr weggenommen werden
- *Circular Wait*: Es kommt zu einer zirkulären geschlossenen Kette von Prozessen, die aufeinander warten

Deadlock-Strategien

tgm
Die Schule der Technik

Es gibt verschiedene Strategien gegen Deadlocks:

- Deadlock *Prevention* (Verhütung): Man lässt Deadlocks durch Verhinderung einer der vier Bedingungen gar nicht erst entstehen, z.B. durch das Design unseres Programms
- Deadlock *Avoidance* (Vermeidung): Man versucht rechtzeitig zu erkennen, dass ein Deadlock entstehen könnte, und versucht darauf zu reagieren (sehen wir später)
- Deadlock *Detection* (Erkennung): Zyklische Beziehungen werden erkannt und aufgelöst (z.B. Terminierung eines Prozesses / Threads)
- *Ostrich* Algorithmus (Vogelstrauß):

Deadlock-Strategien

tgm
Die Schule der Technik

Es gibt verschiedene Strategien gegen Deadlocks:

- Deadlock *Prevention* (Verhütung): Man lässt Deadlocks durch Verhinderung einer der vier Bedingungen gar nicht erst entstehen, z.B. durch das Design unseres Programms
- Deadlock *Avoidance* (Vermeidung): Man versucht rechtzeitig zu erkennen, dass ein Deadlock entstehen könnte, und versucht darauf zu reagieren (sehen wir später)
- Deadlock *Detection* (Erkennung): Zyklische Beziehungen werden erkannt und aufgelöst (z.B. Terminierung eines Prozesses / Threads)
- *Ostrich* Algorithmus (Vogelstrauß): Deadlocks ignorieren und passieren lassen

Deadlock Avoidance

tgm Die Schule der Technik

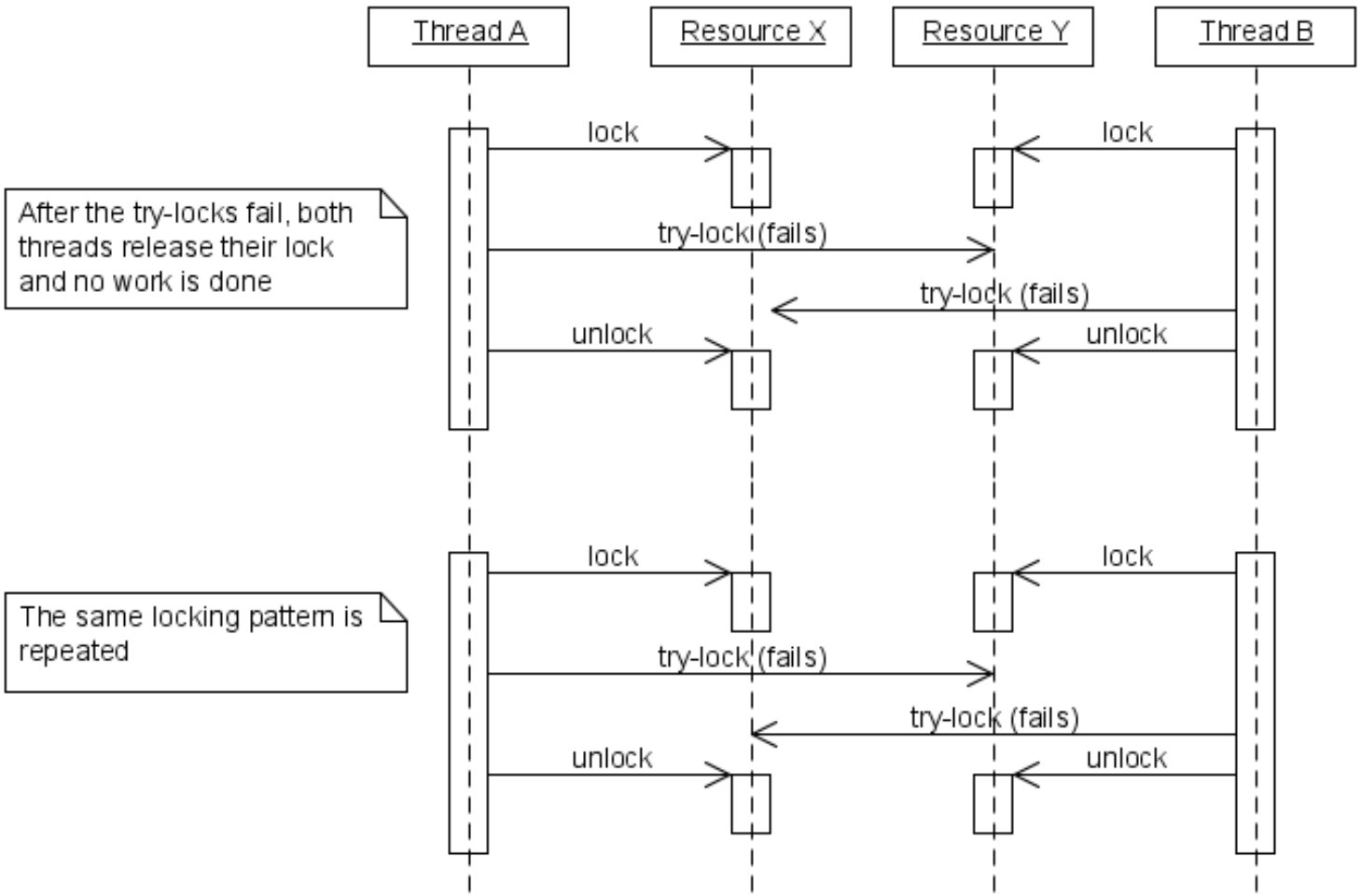
- Deadlock *Avoidance* (Vermeidung): Man versucht rechtzeitig zu erkennen, dass ein Deadlock entstehen könnte, und versucht darauf zu reagieren
- In Python kann `Lock.acquire` beispielsweise **`False`** als Parameter übergeben werden
- `Lock.acquire` liefert einen Wahrheitswert zurück und ist *kein* blockierender Aufruf
- Theoretisch könnte man also folgendes Verfahren wählen:
  1. Versuche Lock 1 zu sperren, falls nicht möglich: X Sekunden warten und wieder probieren
  2. Versuche Lock 2 zu sperren, falls nicht möglich: Lock 1 freigeben und zurück zu Schritt 1
  3. Versuche Lock 3 zu sperren, falls nicht möglich: Lock 1 und 2 freigeben und zurück zu Schritt 1
  4. ...

Livelock

tgm
Die Schule der Technik

- Achtung: Bei einer unsauberen Implementierung kann man mit einer Deadlock-Avoidance-Strategie schnell in einen *Livelock* laufen
- Ein Livelock ist dem Deadlock sehr ähnlich
- Im Unterschied zum Deadlock verharren mehrere Prozesse / Threads jedoch nicht im selben Zustand (z.B. Schlafend), sondern wechseln permanent ihren Zustand
- Bsp.:
  - Zwei Threads wollen dieselben zwei Ressourcen beanspruchen
  - Thread 1 beansprucht Lock 2, Thread 2 beansprucht Lock 1
  - Sie wollen die jeweils andere Ressource beanspruchen
  - Sie erkennen die Deadlock-Gefahr und geben beide ihre Ressourcen frei
  - Beide warten dieselbe Zeit und beginnen von Neuem

Livelock


Thread A
Resource X
Resource Y
Thread B
lock
lock
After the try-locks fail, both threads release their lock and no work is done
try-lock (fails)
try-lock (fails)
unlock
unlock
lock
lock
The same locking pattern is repeated
try-lock (fails)
try-lock (fails)
unlock
unlock

Livelock

tgm
Die Schule der Technik

Beispiel aus der Realität: Am Gehsteig kommt dir eine Person entgegen und ihr wollt wiederholt in dieselbe Richtung ausweichen und es entsteht ein peinlicher Tanz („Sidewalk shuffle“).

Deadlock Prevention

tgm
Die Schule der Technik

- Deadlock *Prevention* (Verhütung): Man lässt Deadlocks durch Verhinderung einer der vier Bedingungen gar nicht erst entstehen
- Es kann eine Klasse geschrieben werden, welche mehrere Locks auf einmal sperrt, wie z.B. in C++ die Methode `lock(mutex1, mutex1, ...)`
- Siehe z.B. http://dabeaz.blogspot.co.at/2009/11/python-thread-deadlock-avoidance_20.html
- Oder: Locks vermeiden, so gut es geht, und stattdessen z.B. Queues verwenden

Deadlock Detection

tgm
Die Schule der Technik

- Es wird versucht, zu bestimmen, ob ein Deadlock vorliegt und diesen ggf. aufzulösen (z.B. durch „killen“ eines Threads)
- Hier kommt üblicherweise die Zyklensuche zum Einsatz
- Dadurch werden zyklische Wartebedingungen erkannt und aufgelöst
- Relativ aufwändig
- Kommt bei manchen Datenbankmanagementsystemen zum Einsatz

Starvation

tgm
Die Schule der Technik

- Ein weiterer problematischer Effekt ist *Starvation*
- Man spricht von Starvation, wenn ein Thread keinen oder kaum Fortschritt machen kann, weil die Ressource(n) ständig von einem anderen Thread blockiert werden
- Dieser Effekt verstärkt sich, je länger sich ein anderer Thread im kritischen Bereich aufhält
- Threads, die sehr lange im kritischen Bereich verweilen und somit die anderen Threads lange warten lassen, nennt man auch „greedy“

Zusammenfassung

tgm
Die Schule der Technik

- Durch den Einsatz von Threads können zusätzliche problematische Effekte entstehen:
  - Race Conditions
  - Deadlocks
  - Livelocks
  - Starvation
- Die meisten sind natürlich auch beim Einsatz mehrerer Prozesse möglich
- Für das Auftreten von Deadlocks gibt es vier Voraussetzungen
- Es gibt verschiedene Strategien, Deadlocks zu behandeln