

# Softwareentwicklung 4

## Threading Einführung

Dominik Dolezal

Höhere Lehranstalt für Informationstechnologie

14. November 2016

Wiederholung

Events

Bedingungsvariablen

Queues

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)

- ▶ Gemeinsam nutzbarer Speicher erleichtert zwar einerseits die Kommunikation zwischen Threads
- ▶ Der gleichzeitige Zugriff bringt aber auch Gefahren mit sich
- ▶ Einfaches Beispiel: Was gibt das Programm aus?

---

```
import threading

class SimpleCounter(threading.Thread):
    # Globaler Zaehler
    counter = 0
    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        for i in range(1000):
            curValue = SimpleCounter.counter
            print("Current Value:" + str(curValue))
            SimpleCounter.counter = curValue + 1
```

---

```
# 10 Instanzen der Thread-Klasse erstellen
threads = []
for i in range(0, 10):
    thread = SimpleCounter()
    threads += [thread]
    thread.start()

# Auf die Kind-Threads warten
for x in threads:
    x.join()

# Counter ausgeben
print(SimpleCounter.counter)
```

```
...  
Current Value:1012  
Current Value:1013  
Current Value:1014  
Current Value:1015  
Current Value:1016  
Current Value:1017  
1018
```

- ▶ Wider Erwarten ist das Ergebnis nicht 10.000
- ▶ Das Ergebnis schwankt im Bereich 1.000-10.000 – **Warum?**
- ▶ Die Threads werden zwischen dem Lese- und Schreibvorgang unterbrochen, sodass sie gegenseitig ihre Änderungen überschrieben!

## Thread 1

```
# counter (global): 10
curValue = SimpleCounter.counter
# curValue (lokal): 10

SimpleCounter.counter = curValue + 1
# counter (global): 11
```

## Thread 2

```
# counter (global): 10
curValue = SimpleCounter.counter
# curValue (lokal): 10

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!

- ▶ 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



Es gibt mehrere Möglichkeiten, Threads sicher zu gestalten:

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

```
class SimpleCounter(threading.Thread):
    counter = 0
    # Globale Lock erzeugen
    lock = threading.Lock()
    def __init__(self):
        threading.Thread.__init__(self)
    def run(self):
        for i in range(1000):
            # Lock sperren (falls frei), ansonsten warten
            with SimpleCounter.lock:
                # --- Beginn kritischer Abschnitt ---
                curValue = SimpleCounter.counter
                print("Current Value:" + str(curValue))
                SimpleCounter.counter = curValue + 1
                # --- Ende kritischer Abschnitt ---
```

```
# 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 ---
```

- ▶ Mutex steht für **mutual exclusion** Objekt und ist wie eine „Sperre“ vorstellbar
- ▶ **with** ist ein Schlüsselwort, welches die Lock akquiriert und beim Verlassen des Blocks automatisch wieder freigibt (auch im Fehlerfall)
- ▶ Andere Threads werden blockiert, bis Lock wieder freigegeben ist
- ▶ Alternative: `lock.acquire()` und `lock.release()` manuell in einem try-finally-Statement aufrufen

```
# 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

- ▶ 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

---

```
class SimpleWorker(threading.Thread):
    def __init__(self, event, threadnumber):
        threading.Thread.__init__(self)
        self.event = event
        self.threadnumber = threadnumber
    def run(self):
        global word
        # Auf das Event warten (blockiert)
        self.event.wait()
        temp = []
        for i in range(len(word)):
            temp += [chr(ord(word[i])+self.threadnumber)]
        print(''.join(temp))
```

---

```
event = threading.Event()
# 10 Instanzen der Thread-Klasse erstellen
threads = []
for i in range(0, 10):
    thread = SimpleWorker(event, i)
    threads += [thread]
    thread.start()

word = input("Wie lautet das Wort?")

# Event auslösen - Threads werden aufgeweckt
event.set()

# Auf die Kind-Threads warten
for x in threads:
    x.join()
```

---

Wie lautet das Wort?Hallo

Thread 8:Pittw

Thread 3:Kdoor

Thread 4:Lepps

Thread 7:Ohssv

Thread 9:Qjuux

Thread 0:Hallo

Thread 2:Jcnnq

Thread 6:Ngrru

Thread 5:Mfqqt

Thread 1:Ibmmp

---

- ▶ Erst nach der Eingabe starten die Threads mit der Verarbeitung
- ▶ Alle werden gleichzeitig aufgeweckt
- ▶ `clear()` setzt Event wieder zurück (und es kann wieder darauf gewartet werden)



- ▶ 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

```
class Producer(threading.Thread):
    def __init__(self, numbers, condition):
        threading.Thread.__init__(self)
        self.numbers = numbers
        self.condition = condition
    def run(self):
        number = 0
        while True:
            with self.condition:
                print("Producer sperrt condition")
                number = number + 1
                print("Producer erzeugt zahl %d" % number)
                self.numbers.append(number)
                print("Producer gibt condition wieder
                      frei")
                self.condition.notify()
            time.sleep(0.01)
```

```
class Consumer(threading.Thread):
    def __init__(self, numbers, condition):
        threading.Thread.__init__(self)
        self.numbers = numbers
        self.condition = condition
    def run(self):
        while True:
            with self.condition:
                print("Consumer sperrt condition")
                while True:
                    if self.numbers:
                        print("Zahl: %d" % self.numbers.pop())
                    else:
                        break
                print("Consumer gibt condition wieder frei")
            self.condition.wait()
```

```
if __name__ == '__main__':  
    numbers = []  
    condition = threading.Condition()  
    t1 = Producer(numbers, condition)  
    t2 = Consumer(numbers, condition)  
    t1.start()  
    t2.start()  
    t1.join()  
    t2.join()
```

- ▶ Die Abfrage `if __name__ == '__main__':` stellt sicher, dass es sich um den Main-Thread handelt und das Skript direkt ausgeführt wird
- ▶ Programm könnte ja auch importiert werden
- ▶ Dadurch wird vermieden, dass diese Befehle ausgeführt werden, wenn das Modul von einem anderen Skript importiert wird

## 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**)

## 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

## 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

# Bedingungsvariablen: wait()

- ▶ Es muss immer zuerst die Lock gesperrt werden, bevor wait() aufgerufen wird
- ▶ wait() legt den Thread schlafen und gibt die Lock gleichzeitig wieder frei
- ▶ Sobald der Thread aufgeweckt wird, sperrt wait() wieder die Lock
- ▶ Die **while**-Schleifen sind deshalb wichtig, weil sich die Bedingung nach dem wait() geändert haben kann oder der Thread „unabsichtlich“ geweckt wurde (spurious wakeup)

## 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()
```



# Bedingungsvariablen: notify()

- ▶ Der Producer kann über `notify()` einen einzelnen Thread oder über `notifyAll()` alle wartenden Threads wecken
- ▶ `notify()` und `notifyAll()` geben jedoch *nicht* automatisch die Lock frei
- ▶ Die Lock wird in diesem Beispiel freigegeben, sobald der Thread das **with**-Statement verlässt
- ▶ **Achtung:** Niemals `sleep()` aufrufen, bevor eine Lock freigegeben wurde!

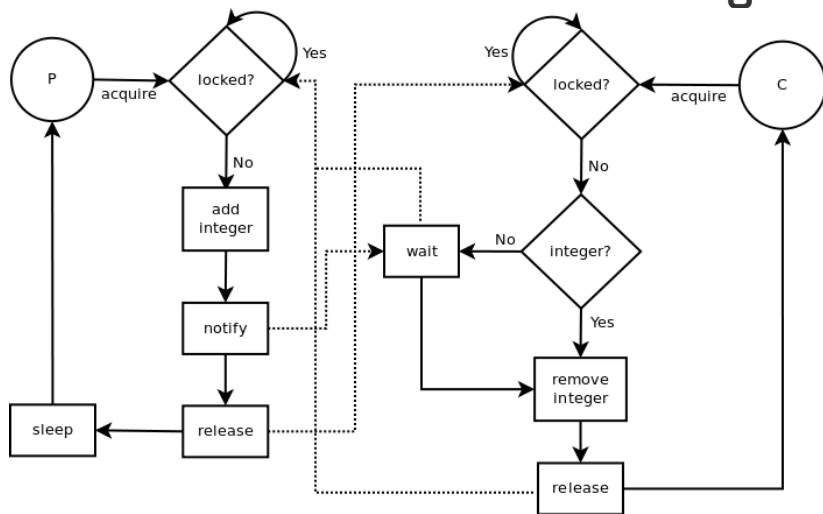
## Producer

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

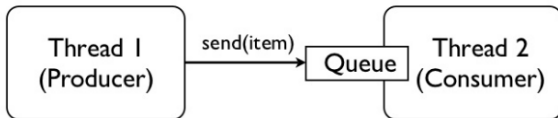
## Consumer

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

# Bedingungsvariablen



- ▶ 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



```
class Producer(threading.Thread):  
    def __init__(self, queue):  
        threading.Thread.__init__(self)  
        self.queue = queue  
  
    def run(self):  
        number = 0  
        while True:  
            number = number + 1  
            self.queue.put(number)  
            self.queue.join()
```

- ▶ Der Erzeuger sendet über `put()` die erzeugte Zahl an die Queue
- ▶ Über `join` kann optional darauf gewartet werden, bis der Verbraucher mit der Verarbeitung fertig ist

```
class Consumer(threading.Thread):  
    def __init__(self, queue):  
        threading.Thread.__init__(self)  
        self.queue = queue  
  
    def run(self):  
        while True:  
            number = self.queue.get()  
            print("Consumer empfangt Zahl: %d" %  
                  number)  
            self.queue.task_done()
```

- ▶ Der Verbraucher empfängt über `get()` die erzeugte Zahl. Falls die Queue gerade leer ist, blockiert `get()` so lange, bis eine Zahl an die Queue gesendet wurde
- ▶ Über `task_done()` wird das Signal gesendet, dass er mit der Verarbeitung fertig ist

---

```
if __name__ == '__main__':  
    queue = queue.Queue()  
    t1 = Producer(queue)  
    t2 = Consumer(queue)  
    t1.start()  
    t2.start()  
    t1.join()  
    t2.join()
```

---

## Producer

```
while True:
```

```
    number = number + 1
```

```
    self.queue.put(number)
```

```
    self.queue.join()
```

## Consumer

```
while True:
```

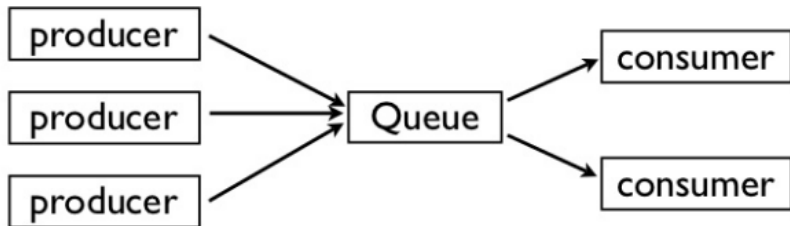
```
    number = self.queue.get()
```

```
    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!

- ▶ Es kann auch mehrere Erzeuger und Verbraucher geben – man sollte es jedoch so einfach wie möglich halten!





- ▶ **Events** dienen dazu, um über den Eintritt eines Ereignisses zu benachrichtigen
  - ▶ Sie erzeugen jedoch keinen kritischen Abschnitt, da keine Lock verwendet wird!
- ▶ **Bedingungsvariablen** sind höhere Konzepte und signalisieren ebenfalls das Eintreten einer Bedingung
  - ▶ Sie besitzen jedoch eine eingebaute Lock und spannen daher ebenso wieder einen kritischen Bereich auf
- ▶ **Queues** sind eine schlanke Lösung für das Erzeuger-Verbraucher-Problem und werden in der Praxis in Python oft eingesetzt, um Threads miteinander zu synchronisieren
  - ▶ Es ist keine Lock notwendig, da ausschließlich über die Queue kommuniziert wird
  - ▶ Die Queue selbst ist threadsicher