

Softwareentwicklung 4

Threading Einführung

Dominik Dolezal

Höhere Lehranstalt für Informationstechnologie

7. November 2016

Wiederholung

Geteilter Speicher

Konkurrierende Zugriffe

Thread Synchronisation

Threading

Das Betriebssystem simuliert „Multitasking“

- ▶ Auf einem Prozessor kann nur 1 Prozess gleichzeitig ausgeführt werden
- ▶ Das Betriebssystem (genauer: der Scheduler) wechselt sehr schnell den jeweiligen Prozess, der gerade am Prozessor ausgeführt wird
- ▶ Prozesse haben einen eigenen Adressraum (Speicherbereich) und arbeiten unabhängig voneinander (Ausnahme: shared memory)
- ▶ Natürlich können bei Mehrkernsystemen Prozesse tatsächlich parallel ausgeführt werden

Oft möchte oder muss eine einzige Anwendung mehrere Aufgaben gleichzeitig (parallel) abarbeiten, z.B.

- ▶ ein Server, der mehrere Clients bedient
- ▶ ein Sortieralgorithmus, der parallelisiert werden kann (z.B. Mergesort)
- ▶ ein Programm, welches eine „blockierende“ Methode aufruft und auf ein Ergebnis wartet (z.B. Webrequest, Betriebssystem-Funktionen)

Wir haben bisher ausschließlich sequentielle Programme geschrieben (d.h. nicht-parallelisiert).

Welche Möglichkeiten gibt es nun, Programme zu parallelisieren?

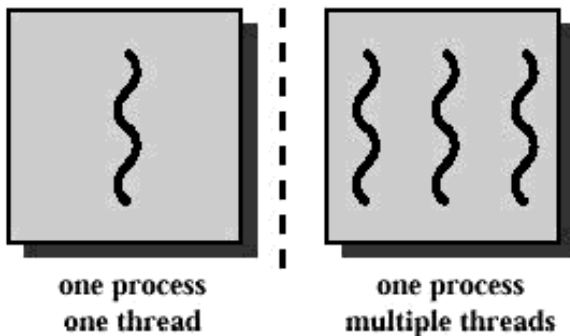
1. **Möglichkeit:** Mehrere *Prozesse* verwenden

- ▶ Prozesse besitzen einen eigenen Adressraum, d.h. einen eigenen Speicherbereich im Hauptspeicher
- ▶ Damit Prozesse miteinander kommunizieren können, benötigt es Interprozess-Kommunikation (IPC), welche relativ aufwendig ist
- ▶ Der bereits erwähnte ständige Wechsel zwischen unterschiedlichen Prozessen durch den Scheduler (Kontextwechsel oder *context switch*) ist sehr teuer
- ▶ Eigene Prozesse sind daher sehr „schwergewichtig“
- ▶ Beispiel: „Fork-Server“



2. **Möglichkeit:** Mehrere *Threads* verwenden

- ▶ Ein Prozess besteht immer aus mind. 1 Thread („Ausführungsstrang“, „Faden“) und ein Thread gehört immer zu genau 1 Prozess
- ▶ Unsere Programme haben bisher immer genau 1 Thread verwendet („main thread“)
- ▶ Threads im selben Prozess teilen sich den Adressraum und können einfacher miteinander synchronisiert werden, einfacher gestartet und einfacher zerstört werden
- ▶ Der Kontextwechsel ist weitaus günstiger als der von Prozessen, da der Adressraum nicht getauscht werden muss – es ist nicht einmal das Betriebssystem involviert
- ▶ Threads sind daher wie „leichtgewichtige“ Prozesse

2. Möglichkeit: Mehrere *Threads* verwenden



- Beispiele: Java Virtual Machine, moderne Spiele, moderne Server, GUI-Toolkits, ...

>	 E8_Threading.exe	33,4%
>	 E8_Threading.exe	66,6%

Der Einsatz von Threads ermöglicht also das parallele Abarbeiten von Aufgaben, um Folgendes zu erreichen:

- ▶ Schnellere Abarbeitung von gut parallelisierbaren Aufgaben durch Einsatz mehrerer Prozessoren
- ▶ Blockierende Aufrufe (z.B. I/O-Operationen) halten nicht mehr den gesamten Prozess auf
- ▶ Logische Trennung von Aufgaben, die voneinander unabhängig sind

Wie erstelle ich einen Thread unter Python?

- ▶ Es gibt zwei grundlegende Module für Multi-Threading in Python:
Das Modul `thread` (veraltet) und das Modul `threading`
- ▶ Beispiel (auf [GitHub](#) verfügbar):

```
import threading

class EndlosschleifenThread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        while True:
            pass
```

```
t1 = EndlosschleifenThread()  
t2 = EndlosschleifenThread()  
  
t1.start()  
t2.start()  
  
t1.join()  
t2.join()
```

```
import threading
```

```
class EndlosschleifenThread(threading.Thread):  
    def __init__(self):  
        threading.Thread.__init__(self)
```

- ▶ Threads erben von `threading.Thread`
- ▶ Im Konstruktor muss der Basisklassen-Konstruktor aufgerufen werden
- ▶ Parameter für den Thread werden ebenfalls im Konstruktor übergeben (und als Instanzvariablen über `self.<variablenname>` gespeichert)

```
class EndlosschleifenThread(threading.Thread):  
    def __init__(self):  
        threading.Thread.__init__(self)  
    def run(self):  
        while True:  
            pass
```

```
t = EndlosschleifenThread()
```

```
t.start()
```

- ▶ Die run-Methode wird ausgeführt, sobald der Thread über start() gestartet wird
- ▶ Achtung: run() **nie** direkt aufrufen – **nur** indirekt über start()!

```
t = EndlosschleifenThread()
```

```
t.start()
```

```
t.join()
```

- ▶ In manchen Sprachen wird der ganze Prozess terminiert, sobald der main-Thread beendet ist und alle anderen Threads werden „abgewürgt“, bevor sie ihre Arbeit verrichten konnten
- ▶ Daher wartet oft der main-Thread auf die Beendigung von den anderen Threads
- ▶ Mit der Methode `t.join()` kann auf die Terminierung von Thread `t` gewartet werden
- ▶ Hierbei handelt es sich also um einen blockierenden Methodenaufruf – der main-Thread wird solange aufgehalten, bis der zweite Thread terminiert wurde

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

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

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

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)

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

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?**

```
...  
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

- ▶ Threads teilen sich folgenden Speicher:
 - ▶ Globale und statische Variablen sowie Klassenvariablen
 - ▶ Alle Objekte (sofern Referenz vorhanden)
- ▶ Threads teilen sich folgenden Speicher **nicht**:
 - ▶ Lokale Variablen (in Funktionen/Methoden)
 - ▶ Threadlokale Variablen (`threading.local`)
- ▶ Durch geteilten Speicher entstehen *race conditions*
 - ▶ ... Wenn das Ergebnis von der Reihenfolge der Ausführung der Threads abhängt
- ▶ Um solche Effekte zu vermeiden, werden Threads *synchronisiert*, z.B. durch
 - ▶ Locks (bzw. Mutexe)
 - ▶ Events und Bedingungsvariablen
 - ▶ Queues
 - ▶ Atomare Variablen