

# Softwareentwicklung 4

## Threading Einführung

Dominik Dolezal

Höhere Lehranstalt für Informationstechnologie

10. Oktober 2016

Einführung Threading

Threading in Python

# Threading

Was ist der Unterschied zwischen den beiden Anwendungen?



E8\_Threading.exe



33,4%



E8\_Threading.exe



66,6%

Was ist der Unterschied zwischen den beiden Anwendungen?

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

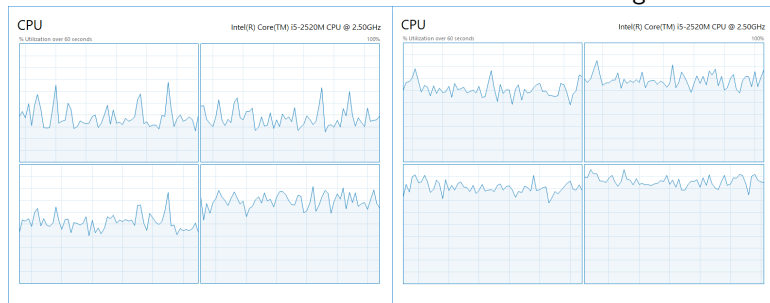
- ▶ Beide Anwendungen befinden sich in einer simplen Endlosschleife

Was ist der Unterschied zwischen den beiden Anwendungen?

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

- ▶ Beide Anwendungen befinden sich in einer simplen Endlosschleife
- ▶ Aber nur eine läuft auf mehr als einem Prozessor!

Was ist der Unterschied zwischen den beiden Anwendungen?



Läuft auf 1 Prozessor

Läuft auf 2 Prozessoren

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?

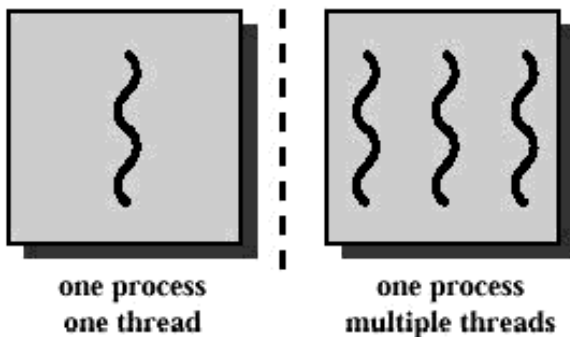
## 1. **Möglichkeit:** Mehrere *Prozesse* verwenden

- ▶ 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, ...

Was ist der Unterschied zwischen den beiden Anwendungen?

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

- ▶ Beide Anwendungen befinden sich in einer simplen Endlosschleife
- ▶ Die zweite Anwendung verwendet jedoch zwei Threads und kann daher auf zwei Prozessoren gleichzeitig ausgeführt werden!
- ▶ Beide Threads befinden sich dort in einer Endlosschleife

>	 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

- ▶ Mithilfe von Threads können Aufgaben parallel ausgeführt werden
  - ▶ Parallelisierung von Algorithmen (z.B. Mergesort)
  - ▶ Blockierende Aufrufe (z.B. Benutzereingaben)
  - ▶ Logische Trennung von Aufgaben
- ▶ Im Gegensatz zu Prozessen sind Threads „leichtgewichtig“ (gemeinsamer Speicher)
  - ▶ Sie können leichter gestartet werden
  - ▶ Sie können leichter untereinander synchronisiert werden
  - ▶ Sie können leichter terminiert werden

- ▶ In Python erbt man von `threading.Thread`
  - ▶ Basisklassen-Konstruktor aufrufen  
`threading.Thread.__init__(self)`
  - ▶ Aufgaben in der `run()`-Methode ausführen
  - ▶ Thread erstellen, über `start()` die Ausführung beginnen und mit `join()` auf das Ende der `run()`-Methode warten

---

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

```
t = EndlosschleifenThread()
t.start()
t.join()
```