Grundlagen Softwareentwicklung

Nebenläufigkeit

Einführung und Anwendung in Python

Markus Reichl

14. März 2017

Inhaltsverzeichnis

1 Einführung

Die Nebenläufigkeit oder Parallelität (englisch *concurrency*) beschreibt die **gleichzeitige Abfertigung mehrerer Anweisungen**.

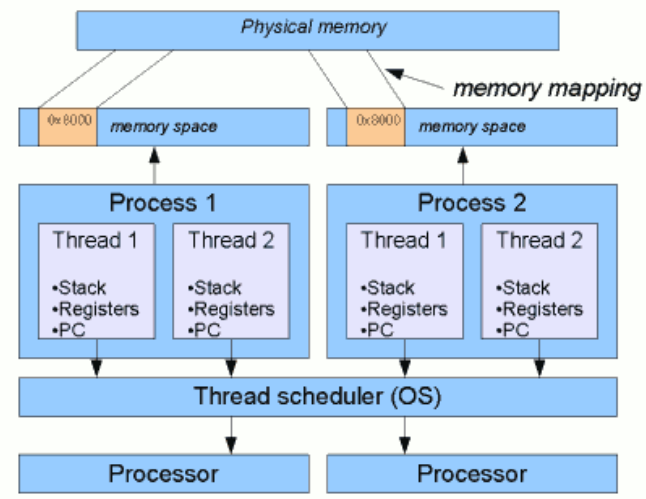
Eine reale Nebenläufigkeit ist gegeben, wenn diese Anweisungen physisch voneinander getrennt sind. In der Softwareentwicklung wird jedoch meist nur eine scheinbare Parallelität verwendet, diese bezeichnet man als **Multitasking**.

1.1 Anwendungsbereiche

Parallelität wird vor allem dort benötigt wo Aufgaben unabhängig voneinander abgearbeitet werden sollen. Darunter fallen zum Beispiel: *Benutzereingaben*, *Server*, *Sortieralgorithmen*, ...

2 Multitasking

Beim Multitasking werden die verschiedenen Prozesse in so kurzen Abständen aktiviert, dass der Eindruck der Gleichzeitigkeit entsteht. Die Verwaltung dieser Prozesse übernimmt im Betriebssystem der Scheduler.



Multitasking am Prozessor
http://www.javamex.com/tutorials/threads/how_threads_work.shtml

Threads Alle Threads innerhalb eines Prozesses teilen sich denselben Adressraum, wodurch sie im Gegensatz zu diesen leichter gestartet, verwaltet und auch zerstört werden können.

Prozesse Prozesse besitzen einen eigenen Adressraum im Hauptspeicher und bestehen aus mindestens einem Thread[1]. Im Gegensatz zum Threading ist Multiprocessing zwar performanter, jedoch ist auch die Kommunikation zwischen den Prozessen aufwendiger.

[1] In Python ist dieser als `'__main__'` gekennzeichnet

2.1 Multithreading

Thread anhand einer Funktion

```
import threading

threading.Thread(target=print, args=('Hello World!',)).start()
```

Die kürzeste Methode einen neuen Thread zu starten geht über die Klasse `threading.Thread`. Diese übernimmt eine Funktion (`target=print`) als Objekt, sowie ihre Parameter (`args=('Hello World!',)`) als Argumente. Nach der Instanziierung des Objekts wird die `run` Methode mittels `start()` aufgerufen und der Thread gestartet.

Eine Funktion als Thread zu instanziieren ist in den meisten Sprachen der Standard, bei komplexeren Anweisungen wird diese Variante jedoch schnell unübersichtlich. Besonders in Python hat sich hier eine andere Variante durchgesetzt, bei welcher direkt von `threading.Thread` geerbt wird.

Thread anhand einer Klasse

```
1  import threading
2
3  class MyThread(threading.Thread):
4    def __init__(self):
5      super().__init__()
6
7    def run(self):
8      print('Hello World!')
9
10 thread = MyThread()
11 thread.start()
```

Die eigene Klasse erbt von `threading.Thread` und ruft im eigenen Konstruktor den des Elternteils auf. Die zuvor aufgerufene `run` Methode wird mit der eigenen Funktionalität überschrieben, welche parallelisiert werden soll. Anstatt nun ein Objekt der Stammklasse zu instanziieren wird die eigene Klasse verwendet und über `start()` gestartet.

Terminierung

Sobald ein Thread die `run` Methode verlässt wird dieser automatisch beendet. Häufig sind jedoch andere Anweisungen vom Ergebnis dieses Threads abhängig und sollen auf diese warten.

Join Die `join()` Methode der Klasse `threading.Thread` stellt einen blockierenden Aufruf dar. Dieser wird erst beendet sobald der zugehörige Thread geschlossen wird.

```
13 thread.join()
14
15 print('Bye!')
```

Daemons Während einige Threads den Ablauf direkt beeinflussen sind andere nur im Hintergrund tätig. Diese sollten die Abfertigung des Programms nicht behindern und müssen daher geschlossen werden.
Zu diesem Zweck werden in Python sogenannte *daemonic Threads* verwendet.

```
1  import threading
2
3  def loop():
4    while True:
5      print('Hi there')
6
7  thread = threading.Thread(target=loop)
8  # Mark this thread as daemonic
9  thread.daemon = True
10 thread.start()
```

Daemonic Threads werden beendet sobald alle anderen Threads[1] geschlossen wurden und müssen vor dem Starten mit Hilfe ihres `daemon` Attributes markiert werden.

[1] Inklusive dem `'__main__'` Thread

2.2 Multiprocessing

In Python unterscheidet sich der Aufbau von Threads und Prozessen nur geringfügig, die Grundstruktur bleibt also gleich mit dem einzigen Unterschied, dass `multiprocessing.Process` anstatt von `threading.Thread` verwendet wird.

Prozess anhand einer Funktion

```
import multiprocessing

multiprocessing.Process(target=print, args=('Hello World!',)).start()
```

Prozess anhand einer Klasse

```
1  import multiprocessing
2
3  class MyProcess(multiprocessing.Process):
4    def __init__(self):
5      super().__init__()
6
7    def run(self):
8      print('Hello World!')
9
10 process = MyProcess()
11 process.start()
```

Terminierung

Die Terminierung erfolgt syntaktisch wie gehabt und ändert sich von der Funktion her nur geringfügig.

Join

```
13 process.join()
```

Daemons

```
7  process = multiprocessing.Process(target=loop)
8  # Mark this process as daemonic
9  process.daemon = True
10 process.start()
```

Prozesse welche als *daemonic* markiert wurden, werden beendet sobald alle anderen Prozesse innerhalb des Programms geschlossen sind.