Nebenläufige Programme mit Python

PyCon DE 2012

Stefan Schwarzer, SSchwarzer.com
info@sschwarzer.com

Leipzig, Deutschland, 2012-10-30

Überblick

- Wann ist Nebenläufigkeit sinnvoll?
- Grundproblem: Konkurrierende Zugriffe
- Lösungsansatz: mehrere Threads in einem Prozess
- Lösungsansatz: mehrere Prozesse
- Absicherung mit Lock
- Absicherung mit Queue
- Zusammenfassung

Wann ist Nebenläufigkeit sinnvoll?

- CPU-intensive Aufgaben – wenn man sie auf mehrere Prozessorkerne verteilen kann
- Ein-/Ausgabe – Während Daten gesendet und empfangen werden, können andere Programmteile weiterlaufen.
- Bedienbarkeit – Während ein GUI-Programm eine Hintergrundaufgabe ausführt, soll es für den Nutzer bedienbar bleiben.

Begriffe

- Nebenläufigkeit / Concurrency:
  mehrere Ausführungsstränge – aber nicht unbedingt gleichzeitige Ausführung (siehe folgender Punkt)
- Parallelism:
  gleichzeitige Ausführung von Code
  (beispielsweise auf verschiedenen CPU-Kernen)
- Atomare Operation / Atomic Operation:
  ein Vorgang, der nicht von einem anderen Thread oder Prozess unterbrochen werden kann
- Race Condition:
  Threads oder Prozesse kommen sich in die Quere.

Grundproblem

- Konkurrierender Zugriff auf Ressourcen muss abgesichert werden.
- Konkurrierend ist alles, was den Zustand einer Ressource ändert.
  - Unterscheidung in lesende und schreibende Zugriffe ist mitunter irreführend.
  - Beispiel: Lesender Zugriff auf eine Datei ändert den Dateizeiger.
- Ressourcen sind zum Beispiel:
  - Einzelwerte und Datenstrukturen im Speicher
  - Dateien
  - Sockets
  - Bildschirm/Fenster

Ansätze für Nebenläufigkeit in Python

Multithreading

- Nebenläufigkeit innerhalb eines Prozesses
- Modul `threading` (in der Standardbibliothek)
- Ausführungsstränge können auf Daten im Speicher zugreifen.
- Bei CPython kommt das Global Interpreter Lock (GIL) zum Tragen.
- Das GIL verhindert die parallele Ausführung von Python-Code.
  Es wird bei I/O-Operationen freigegeben.
  Auch C-Erweiterungen können das GIL freigeben.
- Das GIL begrenzt also nur bei CPU-begrenzten Abläufen;
  bei I/O-begrenzten Abläufen ist es eher unproblematisch.

Ansätze für Nebenläufigkeit in Python

Multiprocessing

- Nebenläufigkeit zwischen verschiedenen Prozessen
- Modul `multiprocessing` (in der Standardbibliothek)
- Datenaustausch zwischen Prozessen über Nachrichten oder über Shared Memory
- Bei Austausch von Nachrichten müssen diese serialisiert werden (in Python meist mit dem `pickle`-Modul). Serialisierung ist Zusatzaufwand.
- Vorteil bei Multiprocessing: keine Einschränkungen der gleichzeitigen Ausführung, auch nicht bei CPU-Begrenzung

Als Mischform sind natürlich auch mehrere Prozesse mit einem oder mehreren Threads möglich.

Multithreading-Beispiel

Code ohne Absicherung gleichzeitiger Zugriffe

```
import threading, time

counter = 0
def count_without_lock():
    global counter
    for i in xrange(10):
        time.sleep(0.001)  # Make race condition more likely.
        counter += 1

threads = []
for i in xrange(100):
    thread = threading.Thread(target=count_without_lock)
    thread.start()  # Start thread. Don't confuse with `run`.
    threads.append(thread)
for thread in threads:
    thread.join()   # Wait until thread is finished.
print "Total: %d" % counter
```

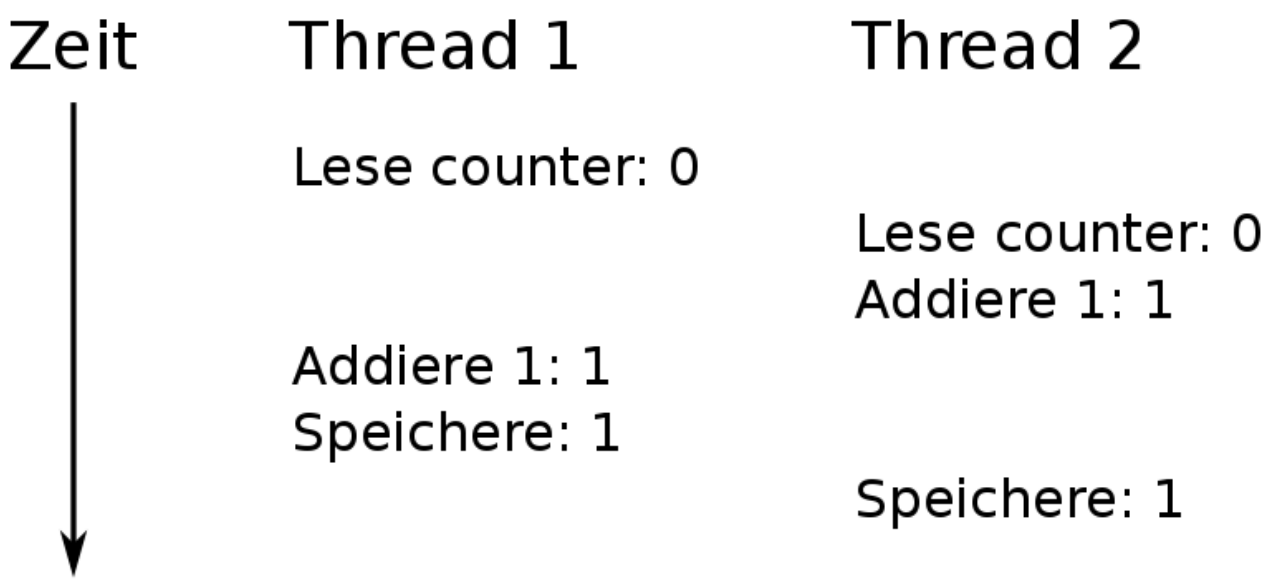
Multithreading-Beispiel

Ausgabe ohne Absicherung gleichzeitiger Zugriffe

```
$ python race_condition.py
Total: 976
$ python race_condition.py
Total: 970
$ python race_condition.py
Total: 981
$ python race_condition.py
Total: 943
$ python race_condition.py
Total: 996
$ python race_condition.py
Total: 884
```

Multithreading-Beispiel

Erklärung – Race Condition bei gleichzeitigen Zugriffen



Thread 2 liest noch den früheren Wert von `counter`, da Thread 1 den Wert noch nicht gespeichert hat.

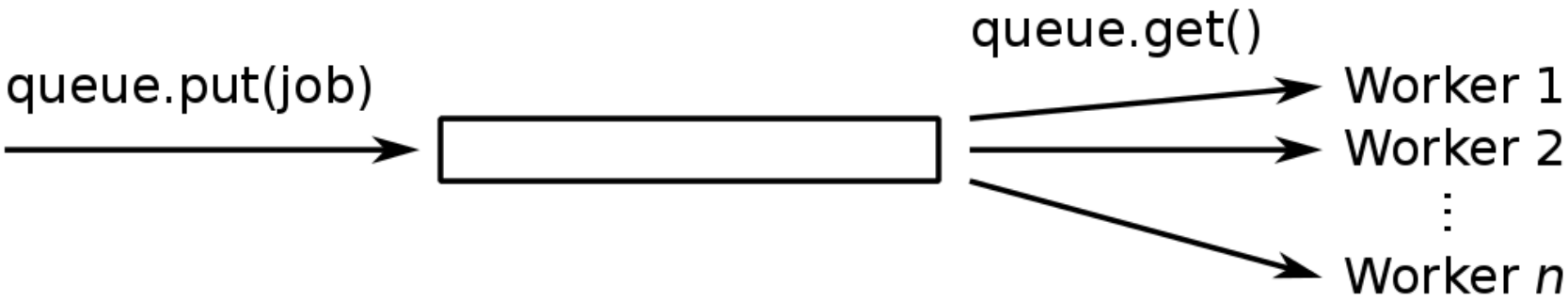
Multithreading-Beispiel

Code mit Absicherung gleichzeitiger Zugriffe

```
import threading, time

counter = 0
lock = threading.Lock()

def count_with_lock():
    global counter
    for i in xrange(10):
        time.sleep(0.001)
        with lock:
            counter += 1

threads = []
for i in xrange(100):
    thread = threading.Thread(target=count_with_lock)
    thread.start()
    threads.append(thread)
...
```

Multithreading mit Queue

Schema für das folgende Beispiel



Prinzip: `put` und `get` sind atomare Operationen.

Multithreading mit Queue

Vorspann

```
#! /usr/bin/env python2

import logging, Queue, random, threading, time

logging.basicConfig(level=logging.INFO, format="%(message)s")
logger = logging.getLogger("queue_example")

WORKER_COUNT = 10
JOB_COUNT = 100
STOP_JOB = object()
job_queue = Queue.Queue()


class Job(object):

    def __init__(self, number):
        self.number = number
```

Multithreading mit Queue

Worker-Thread

```
class Worker(threading.Thread):

    def run(self):
        while True:
            job = job_queue.get(block=True)
            if job is STOP_JOB:
                break
            self.do_job(job)

    def do_job(self, job):
        # Wait between 0 and 0.01 seconds.
        time.sleep(random.random() / 100.0)
        # Atomic output
        logger.info("Job number %d" % job.number)
```

Multithreading mit Queue

Erzeugen und Verarbeiten von Jobs

```
def main():
    workers = []
    # Start workers.
    for i in xrange(WORKER_COUNT):
        worker = Worker()
        worker.start()
        workers.append(worker)
    # Assign jobs to workers.
    for i in xrange(JOB_COUNT):
        job_queue.put(Job(i))
    # Schedule stopping of workers.
    for i in xrange(WORKER_COUNT):
        job_queue.put(STOP_JOB)
    # Wait for workers to finish.
    for worker in workers:
        worker.join()

main()
```

Multithreading mit Queue

Ausgabe

```
Job number 0
Job number 5
Job number 7
Job number 10
Job number 2
Job number 1
Job number 6
...
Job number 93
Job number 99
Job number 91
Job number 95
Job number 92
Job number 98
Job number 97
```

Zusammenfassung

- Nebenläufige Programme haben ihre Tücken, auch wenn nur ein Prozessor dabei genutzt wird.
- Konkurrierende Zugriffe müssen abgesichert werden
- ... unabhängig davon, ob das Programm ohne Absicherung richtig zu laufen scheint oder nicht!
- Atomar aussehende Anweisungen sind es nicht unbedingt.
- Faustregeln:
  - Verwende `threading` bei I/O-begrenzten Aufgaben.
  - Verwende `multiprocessing` bei CPU-begrenzten Aufgaben.
- Konkurrierende Zugriffe können zum Beispiel mit Locks oder Queues abgesichert werden.
- Dieser Vortrag konnte das Thema Nebenläufigkeit leider nur anreißen.

Links

- Module `threading`, `Queue`, `multiprocessing`
  http://docs.python.org/library/threading.html
  http://docs.python.org/library/queue.html
  http://docs.python.org/library/multiprocessing.html
- Dr. Dobb's Parallel Computing
  http://www.drdobbs.com/parallel (Übersichts-Seite)
  http://www.drdobbs.com/212903586 (Einführung)
- High-Level Concurrency (sehr empfehlenswert)
  http://www.russel.org.uk/Presentations/
  bcsEdinburgh_2011-09-14_JeepersGPars.pdf
- Design-Empfehlungen
  http://stackoverflow.com/questions/1190206/
- Active Object Pattern
  http://www.drdobbs.com/225700095

Danke für die Aufmerksamkeit! :-)

Fragen?

Anmerkungen?

Diskussion?

sschwarzer@sschwarzer.com

http://sschwarzer.com