Nebenläufige Programme mit Python

Chemnitzer Linuxtage 2013

Stefan Schwarzer, SSchwarzer.com
info@sschwarzer.com

Chemnitz, Deutschland, 2013-03-17

Überblick

- Wann ist Nebenläufigkeit sinnvoll?
- Grundproblem: Konkurrierende Zugriffe
- Implementierung von Nebenläufigkeit
  - Threads
  - Prozesse
  - Event Loop
- Umgang mit konkurrierenden Zugriffen
  - Locks
  - Queues
- Deadlocks
- Umgang mit konkurrierenden Zugriffen – High-Level-Ansätze
  - Active Objects
  - Process Networks
- Software Transactional Memory (STM)

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 durch zwei oder mehr Ausführungsstränge muss abgesichert werden, um Race Conditions zu verhindern.
- 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.

Ansätze für Nebenläufigkeit in Python

Event Loop

- Schleife („Main Loop“) registriert Ereignisse (Beispiele: Mausklick, eintreffende Netzwerk-Daten).
- Je nach Ereignis wird ein „Handler“ dafür aufgerufen, der die Verarbeitung des Ereignisses übernimmt.
- Die Kontrolle geht (spätestens) danach an die Hauptschleife zurück.
- Programmfluss möglicherweise schwer zu überblicken, wenn die Handler voneinander abhängen.
- Eine Event Loop dient der Steuerung von Nebenläufigkeit, nicht von Parallelität (Ausnutzung mehrerer CPU-Kerne). Zusätzliche Handhabung von Threads oder Prozessen wird nicht vereinfacht.

Ansätze für Nebenläufigkeit in Python

Event Loop – Typische Anwendungen

- GUI-Frameworks (PyGTK, PyQt, wxPython, . . .)
- Netzwerk (Gevent, Twisted, . . .)
- Hybride (netzwerkfähige GUI-Anwendungen)

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
```

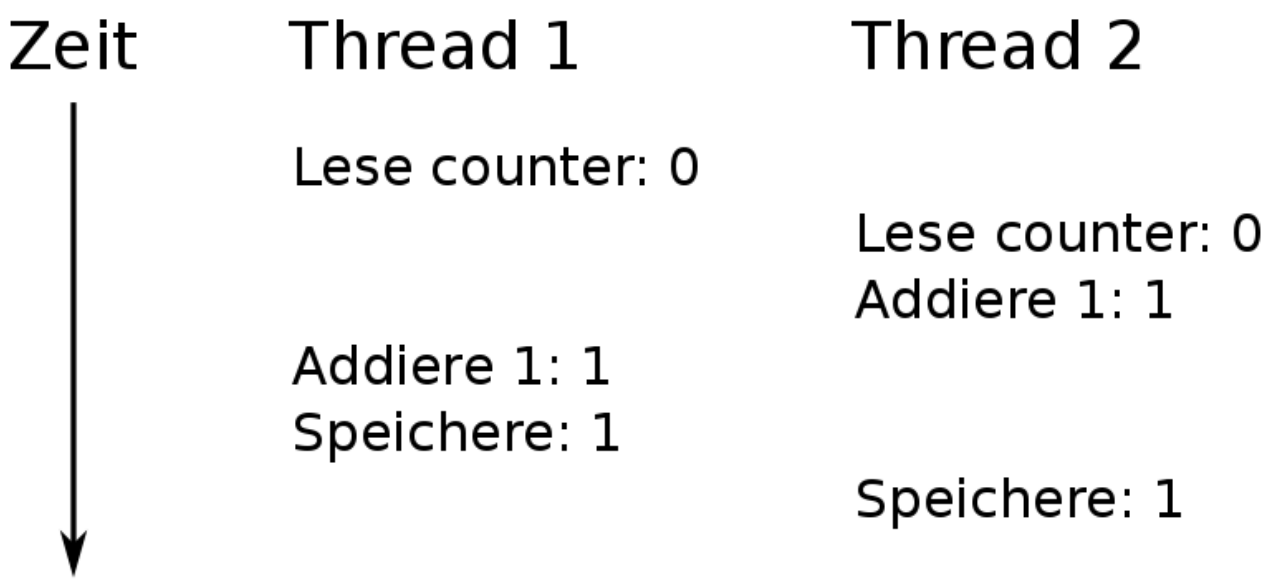
Multithreading-Beispiel

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.

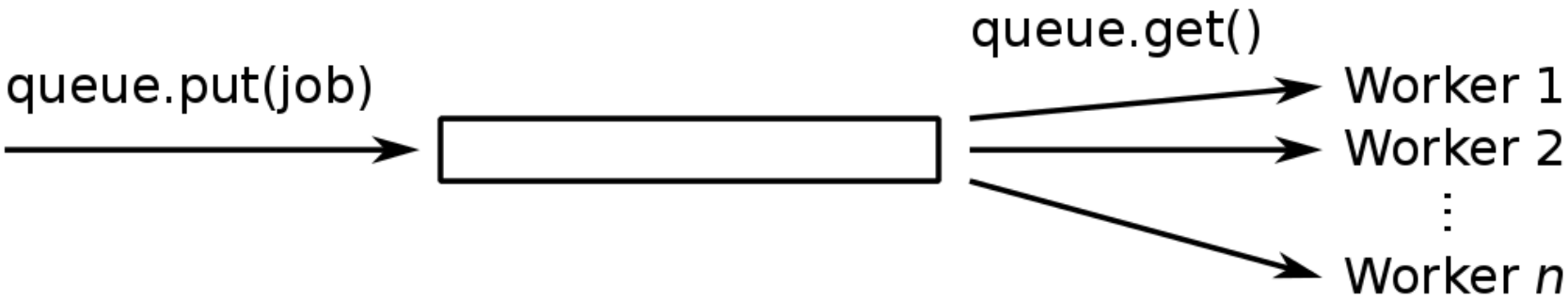
Multithreading-Beispiel

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

```
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
```

Deadlock

Prinzip

- Ein Deadlock entsteht, wenn sich Ausführungsstränge gegenseitig Ressourcen vorenthalten.
- Ausführungsstränge in diesem Sinn können Threads oder Prozesse sein.

Deadlock

Beispiel

- Zwei Threads lesen aus einer Eingabedatei und schreiben in eine Ausgabedatei.
- Dabei sind beide Dateien mit je einem Lock gesichert.

Deadlock

Beispiel – Code

```
def thread1():
    with input_lock:
        with output_lock:
            input_line = input_fobj.readline()
            # Process input ...
            output_fobj.write(output_line)


def thread2():
    with output_lock:
        with input_lock:
            input_line = input_fobj.readline()
            # Process input ...
            output_fobj.write(output_line)
```

Deadlock

Beispiel – Möglicher Programmablauf

- Thread 1 reserviert das Eingabe-Lock.
- Thread 2 reserviert das Ausgabe-Lock.
- Thread 1 versucht, auch das Ausgabe-Lock zu bekommen, aber dies wird von Thread 2 gehalten.
- Thread 2 versucht, auch das Eingabe-Lock zu bekommen, aber dies wird von Thread 1 gehalten.
- Beide Threads blockieren sich gegenseitig $\rightarrow$ Deadlock!
- Ein Deadlock ist umso wahrscheinlicher, je länger die beteiligten Locks von den Threads gehalten werden.

Locks vs. Queues

- Solange ein Lock gehalten wird, kann kein anderer Thread den gesicherten Code-Abschnitt ausführen.
- Threads müssen also aufeinander warten.
- Keine Parallelisierung des gesicherten Code-Abschnitts
- Gefahr von Deadlocks (umso wahrscheinlicher, je länger die beteiligten Locks gehalten werden)
- Queues reduzieren diese Probleme, weil das implizite Lock (beziehungsweise Locks bei getrennt gesicherten „Enden“) nur kurzzeitig gehalten wird.
- Die Verwendung von Queues erleichtert es, die Funktionsweise des nebenläufigen Codes zu verstehen (später mehr dazu).

Active Object Pattern

- Prinzip: Locks, Queues oder andere Synchronisierungs-Mechanismen sind nicht Bestandteil der API eines Objekts.
- Synchronisierung, soweit nötig, ist in High-Level-Methoden versteckt.

Active Object Pattern

Beispiel – Konstruktor

```
import Queue
import threading


STOP = object()


class Worker(object):

    def __init__(self):
        self._in_queue = Queue.Queue()
        self._out_queue = Queue.Queue()
        self._worker_thread = threading.Thread(
                                target=self._work)
        self._worker_thread.start()
```

Active Object Pattern

Beispiel – Öffentliche Methoden

```
def process(self, work_item):
    # The ‘work_item‘ _must not_ be modified by other
    # threads. Best pass in an immutable objects or
    # copy the work item as deeply _as necessary_.
    self._in_queue.put(work_item)

def next_result(self):
    return self._out_queue.get(block=True)

def stop(self):
    self._in_queue.put(STOP)
    # If you want to stop synchronously, use ‘join‘.
    self._worker_thread.join()
```

Active Object Pattern

Beispiel – Interne Methoden

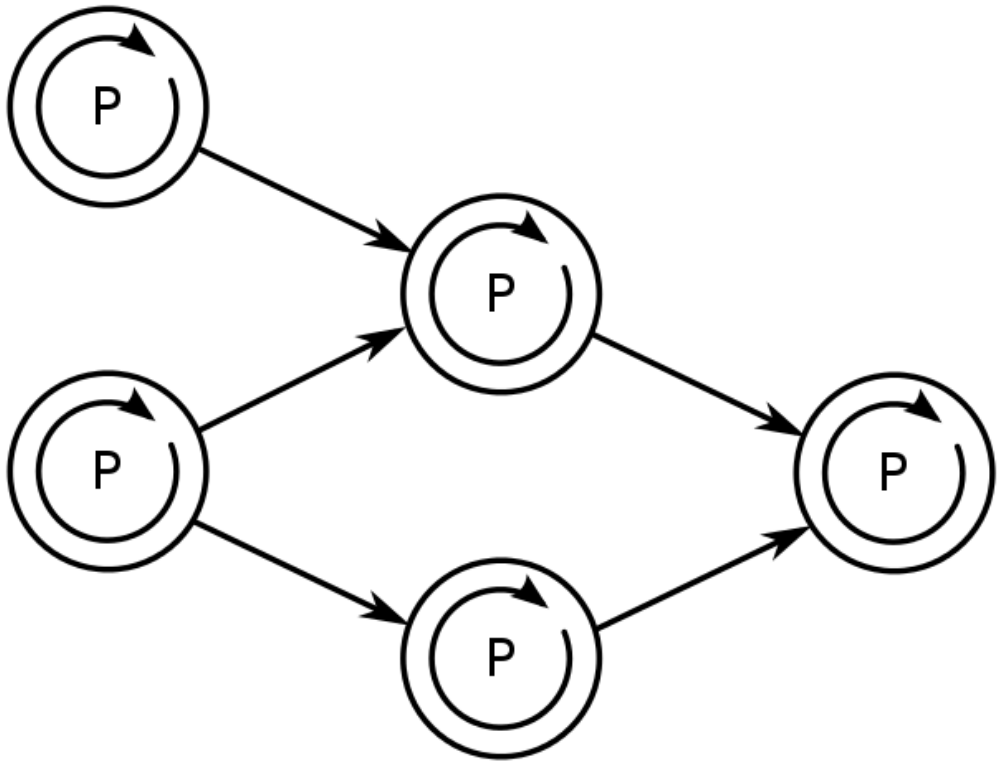
```
    def _work(self):
        while True:
            work_item = self._in_queue.get(block=True)
            if work_item is STOP:
                break
            result = self._result(work_item)
            self._out_queue.put(result)

    def _result(self, work_item):
        raise NotImplementedError("define in subclass")


class Adder(Worker):

    def _result(self, work_item):
        return work_item + 1000
```

Active Object Pattern

Beispiel – Verwendung

```
def main():
    ITEM_COUNT = 100
    adder = Adder()
    for i in xrange(ITEM_COUNT):
        # Doesn't block
        adder.process(i)
    # Do other things.
    # ...
    # Collect results.
    for i in xrange(ITEM_COUNT):
        # May block
        print adder.next_result()
    # May block
    adder.stop()


main()
```

Process Networks



Process Networks

- „Prozesse“ erhalten Eingabedaten und/oder erzeugen Ausgabedaten.
- „Prozesse“ können beispielsweise auch Python-Threads sein.
- Datenübertragung zwischen Prozessen durch Nachrichten-Übertragung (Message Passing)
- Kein gemeinsamer Zustand
- Keine Race Conditions
- Varianten
  - Concurrent Sequential Processes (CSP)
  - Dataflow
  - Flow-Based Programming
  - ...
- Python-Bibliotheken: PyCSP, PyF, DAGPype, ...

Software Transactional Memory (STM)

- Prinzip: Transaktionen wie bei Datenbanken
- Transaktion wird vollständig ausgeführt oder gar nicht
- Programmierer muss festlegen, welcher Code zu einer Transaktion gehört
- Wird eine Race Condition entdeckt, muss die betreffende Transaktion wiederholt werden.
- Einschränkungen für Transaktionen mit Nebenwirkungen (Interaktion „mit der Außenwelt“)
- Unterstützung für STM in PyPy in Entwicklung

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:
  - `threading` für I/O-begrenzte Aufgaben
  - `multiprocessing` für CPU-begrenzte Aufgaben
  - Event Loop für GUI
- Deadlocks entstehen, wenn sich Ausführungsstränge gegenseitig Ressourcen vorenthalten.
- Konkurrierende Zugriffe können beispielsweise durch Locks, Queues, Active Objects oder Process Networks gehandhabt werden.

Danke für die Aufmerksamkeit! :-)

Fragen?

Anmerkungen?

Diskussion?

sschwarzer@sschwarzer.com

http://sschwarzer.com

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