

Netzwerk - Programmierung

Threads

Jan Krüger

jkrueger@cebitec.uni-bielefeld.de

Alexander Sczyrba

asczyrba@cebitec.uni-bielefeld.de

Übersicht

- Probleme mit `fork`
- Threads
- Python `_thread` und `threading` API
- Locks

Probleme mit `fork`

- `fork` ist teuer :
 - Speicher wird für den Kindsprozess kopiert
 - Alle Deskriptoren werden dupliziert
 - ...
- Interprocess Communication (IPC) zur Kommunikation zwischen Parent und Child nötig
 - Übergabe für den `fork` einfach
 - Rückgabe von Informationen (Child zum Parent) schwieriger

Thread

- Auch : *leightweigh process* oder *LWP*
- Jeder Thread läuft unabhängig von anderen Threads in einem gemeinsamen Prozess
- Erzeugen von einem Thread ist 10-100x schneller als das Erzeugen eines Prozesses
- Alle Threads eines Prozesses teilen den gleichen globalen Speicher
- Vereinfacht Austausch von Daten
- Aber : *Synchronisation* nötig

Threads(2)

- Threads eines Prozesses teilen u.a.
 - Globale Variablen
 - Geöffnete Dateien (z.B. Datei Deskriptoren)
 - Signal Handlers
- Jeder Thread hat u.a. eigene(n)
 - Thread ID
 - Register (inkl. Program counter und stack pointer)
 - Stack (für lokale Variablen und Rücksprungadressen)

Aufgabe

- Der Befehl top bzw. htop zeigt Informationen über die zur Zeit laufenden Prozesse an
 - <http://htop.sourceforge.net>
 - starte [h]top und beobachte, welche Prozesse Threads benutzen
 - **Achtung!** Evtl. muss die Ansicht angepasst werden
 - beschränke [h]top auf Deine eigene Prozesse
 - starte den ServiceServer

Erzeugen von Threads in Python

Es existieren zwei Module in Python3 für threaded computing:

`_thread:`

- Low level API
- Threads sind Funktionen
- Veraltet (deprecated) in Python 3

`threading:`

- High level API
- Beliebig einfach erweiterbar (Threads sind Objekte)
- Sammeln in Threadpools und Queues relativ einfach

threading-Modul

```
import time
from threading import Thread

def sleeper(i):
    print("thread %d sleeps for 5 seconds" % i)
    time.sleep(5)
    print("thread %d woke up" % i)

for i in range(10):
    t = Thread(target=sleeper, args=(i,))
    t.start()
```


Warten auf Threadbeendigung

Etwas fehlt noch: `t.join(timeout=None)`

- Aufrufender/threadender Thread blockiert, bis gestarteter Thread terminiert ist
- Rückgabewert immer **None**
- **RuntimeError** bei Deadlock
- Kehrt bei Ablauf des **Timeouts** zurück
- Thread muss dann mit `t.is_alive()` getestet werden, ob der Thread noch am Leben ist
- Kann erneut `gejoint` werden

Aufgaben

- Schreibe ein Programm, das einen Thread erzeugt
- der Thread soll die globale Variable `count` 500 mal inkrementieren
- Gib jeweils den aktuellen Wert der Variable `count` im Thread aus
- Benutze an geeigneten Stellen `sleep`, um das Verhalten Deines Programms mit `[h]top` zu verfolgen
- Gib die Variable nochmal aus, nachdem der Thread joined wurde

Race Conditions

```
from threading import Thread

threads, count = [], 0

def counter():
    global count
    while count < 1000000:
        count+=1

for i in range(1000):
    t = Thread(target=counter)
    threads.append(t)

for t in threads:
    t.start()

for t in threads:
    t.join()
```

Aufgabe

- Ändere Dein Programm so ab, dass die Funktion `counter()` mit `while` arbeitet. Die Bedingung soll sein: `count < 1.000.000`
- Starte 1.000 Threads mit `counter()` als Target direkt nacheinander, um die `count` Variable bis 1.000.000 zu inkrementieren
- Was fällt Dir auf?

Synchronisation in Python mittels Locks, Re-entrant Locks und Semaphoren, Events...

- Python unterstützt bekannte Synchronisationsmechanismen wie Locks, RLocks und Semaphoren
- Verwendet Lock-Funktionalität aus **threading**!!!

```
lock = threading.Lock()
```

```
lock.acquire()
```

```
...
```

```
lock.release()
```

Aufgabe

- Ändere Dein Counter-Programm so ab, dass der kritische Codebereich gesperrt wird. Bekommst Du jetzt das erwartete Ergebnis?