Vorlesung Angewandte Softwareentwicklung

Übungsblatt 6: Nebenläufigkeit

Hariolf Betz, Michael Stuber, Karl Dubies

Inhalte

- **Threads in Python**
- **Synchronisationstechniken**

1 Produzent & Verbraucher

In dieser Aufgabe schreiben wir Funktionen, um das Füllen und Leeren eines Lagers zu simulieren.

Ein *Lager* (realisiert als *Liste*) enthält *Produkte* (realisiert als ganzzahlige *Zufallszahlen*), die von mehreren Verbrauchern immer wieder entnommen werden. Gleichzeitig wird geprüft, ob noch Produkte im Lager sind. Falls das Lager leer ist, werden neue Produkte (d.h. Zufallszahlen) in das Lager (Liste) gelegt.

1. Erstellen Sie eine neue Python-Datei. Importieren Sie das Modul `time`. Erzeugen Sie testhalber eine UNIX-Timestamp mit `time.time()`.
2. Erstellen Sie eine neue Funktion `pruefe_lager(lager)`. Für den Parameter `lager` soll eine *Liste* übergeben werden. Die Funktion soll das folgende tun:
   (a) Überprüfen, ob die Liste `lager` leer ist[1].
   (b) *Falls ja*: zehn zufällige Elemente hinzufügen (mit einer Schleife).
   (c) *In jedem Fall:* 100 Millisekunden schlafen.

   Die Funktion sollte außerdem via Konsolenausgabe darüber informieren, wenn das Lager aufgefüllt wird.

   – **Tipp:** Sie kennen mittlerweile mehrere Arten, Zufallszahlen zu erzeugen.
3. Testen Sie Ihre Funktion mit dem folgenden *Hauptprogramm*:

```
lager = []

tic = time.time()
pruefe_lager(lager)
toc = time.time()

print(lager)
print('Zeit: %.3f Sekunden' % (toc - tic))
```

   – **Tipp:** Die Funktion sollte eine Liste mit zehn zufälligen Elementen ausgeben. Es sollten etwa 0.1 Sekunden vergangen sein.
4. Ändern Sie `pruefe_lager(lager)` so ab, dass die in 1.2 genannten Schritte in einer *Endlosschleife*[2] durchgeführt werden (s. Abbildung 1). (Die Funktion wird also *niemals* verlassen.)
5. Testen Sie die Funktion abermals.

   – **Tipp:** Da Ihre Funktion *endlos* läuft, müssen Sie sie natürlich mit dem **Debugger** testen.
6. Erstellen Sie nun eine Funktion `verbrauche(lager, id)`. Für den Parameter `lager` soll eine *Liste* und für den Parameter `id` eine Identifikationsnummer übergeben werden. (Der Wert von `id` kann ein String oder ein `int`-Wert sein.) Die Funktion soll das folgende tun:
   (a) Überprüfen, ob die Liste `lager` leer ist.
   (b) Falls *nein*:
      i. Das *größte* Element[3] im Lager in einer Variable `max_elem` zwischenspeichern,
      ii. den Wert von `max_elem` und den Wert von `id` *ausgeben* und dann
      iii. `max_elem` aus der Liste *entfernen*.
   (c) *In jedem Fall:* 1 Millisekunde *schlafen*.
7. Testen Sie Ihre Funktion mit dem folgenden *Hauptprogramm*:

```
lager = [4, 7, 42, 11]
verbrauche(lager, 'A')
print(lager)
```

   – **Tipp:** Die Ausgabe sollte so oder ähnlich lauten:

```
[A] 42
[4, 7, 11]
```

8. Versehen Sie auch die Funktion `verbrauche` mit einer Endlosschleife (vgl. Abbildung 2). Testen Sie Ihre Methode.

   – **Tipp:** Nun sollten alle Elemente der Liste *"verbraucht"* werden.

[1] =falls die Länge Null beträgt

[2] `while True:`

[3] `max(...)`

2 Threading-Probleme

In dieser Aufgabe lassen wir die Funktionen aus der vorigen Aufgabe nebenläufig laufen, stoßen dabei auf eine Race Condition *und lösen das Problem mit einem* Lock Token.

1. Passen Sie nun das Hauptprogramm folgendermaßen an:
   (a) Legen Sie ein leeres `lager` an.
   (b) Starten Sie einen `pruefe_lager`-Thread, der das Lager regelmäßig überprüft.
   (c) Starten Sie *drei* `verbrauche`-Threads mit unterschiedlichen `id`s, die Elemente des Lagers *"verbrauchen"*.

   – **Tipp:** Denken Sie daran, dass ein 1-elementiges Tupel, welches das `lager` enthält, folgendermaßen aussieht: `(lager, )`.
2. Starten Sie das Programm und beobachten Sie die Ausgabe eine Weile. Was passiert? Welches Problem tritt auf?

   – **Tipp:** Beachten Sie: Das Problem kann nur deshalb auftreten, weil *mehrere* Threads gleichzeitig auf der Liste `lager` arbeiten.
3. Beheben Sie das Threading-Problem mit einem `Lock`-Objekt. Testen Sie anschließend, ob Ihre Maßnahme funktioniert.

   – **Tipp:** Überlegen Sie genau, wo die *kritischen Bereiche* liegen. Wählen Sie die synchronisierten[4] Bereiche *so klein wie möglich* aber *so groß wie nötig*!

   – **Tipp:** Wenn Sie **innerhalb** eines *synchronisierten* Abschnitts eine Endlosschleife programmieren, wird das `Lock` *niemals* freigegeben (vgl. Abbildung 3). *Das ist natürlich ein Fehler!*

   – **Tipp:** Wenn eine `sleep`-Anweisung **innerhalb** eines *synchronisierten* Abschnitts steht, werden andere Threads ausgehungert. *Auch das ist ein Fehler!*

   Die Verbraucher sollten *im Wechsel* auf das Lager zugreifen:

```
[A] 97
[C] 83
[A] 73
[B] 59 ...
```

Bei dieser Aufgabe ist eine Abgabe erforderlich! Hierzu müssen Sie lediglich den Inhalt Ihrer gerade geöffneten `.py`-Datei in das *Textabgabefeld* im Moodlekurs copy/pasten. (Exportieren, zippen, expliziter Dateiupload etc. sind *nicht* erforderlich.)

[4] = per *Lock* geschützten

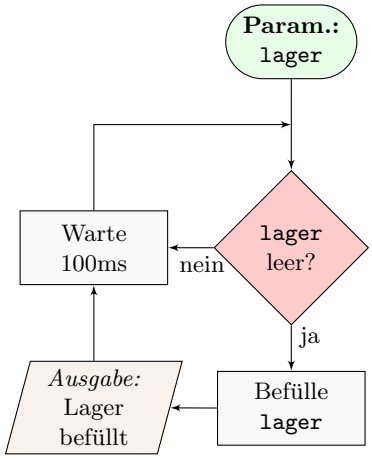

Abbildung 1: `pruefe_lager` mit Endlosschleife

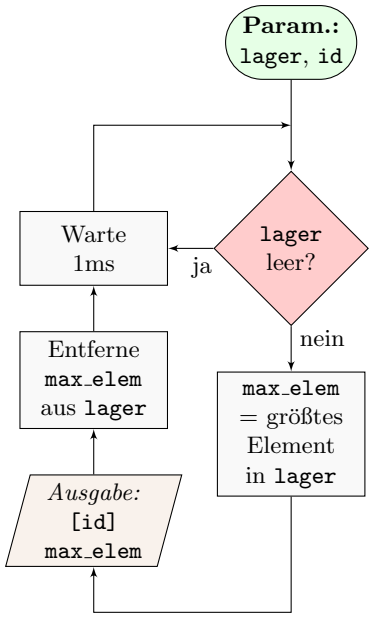

Abbildung 2: `verbrauche` mit Endlosschleife

```
1 with my_lock:
2     while True:
3         # tu irgendwas
```

Abbildung 3: Lock-Token, das niemals wieder freigegeben wird *(Programmierfehler!)*