Deadlocksanalyse unter der Betrachtung des Beispiels der Speisenden Philosophen von Dijkstra.

Städtisches Gymnasium Mittweida

Komplexe Leistung

largevorgelegt von

Lily Wagner

Klasse 10c

Fachlehrer: Herr Schmidt

30.09.2020

Inhaltsverzeichnis

1 Einleitung **3**

2 Grundlagen **5**

2.1 Erklärende Begriffe 5

2.2 Begriff Verklemmungen 6

3 Die speisenden Philosophen **8**

3.1 Vorstellung des Modells 8

3.2 Darstellung des Problems 9

3.3 Lösungsansätze 12

4 Programmierung **14**

4.1 Erste Lösung mit Problemdarstellung 14

4.2 Lösungen 17

5 Fazit **19**

Literatur **20**

Materialienverzeichnis **21**

1 Einleitung

Stellen Sie sich vor, Sie fahren auf einer Straße in ihrem Auto und gelangen an eine Kreuzung ohne Ampeln und Verkehrsschilder. Sie stehen mit vier weiteren Autos an dieser Kreuzung. In diesen Situationen gilt die Regel „rechts vor links“. Das bedeutet, dass kein Auto fahren darf, solange nicht das Auto rechts von ihm von der Stelle gefahren ist. Bestenfalls fahren zwei Autos gleichzeitig los und stoppen wieder, um einen Aufprall zu verhindern, das ist eine Frage der Koordination. Abstrahiert man dieses Problem in die IT und ersetzt die Autos durch Prozesse, nennt man dies eine Verklemmung. Die folgenden Seiten werden konkret das Problem der speisenden Philosophen genauer beleuchten. Dieses Problem wurde als erstes 1965 von Djikstra formuliert und von dem selben auch gelöst. Seitem ist es ein beliebtes Beispiel zur Visualisierung bzw. Illustration von Verklemmungen und Deadlocks. Es ist eines der Worst-Case-Szenarios um Probleme in Algorithmen zu zeigen, in diesem Fall ist das Problem somit ein Deadlock.

Ein Beispiel eines Deadlocks gab es im Jahr 2019. Microsoft hatte ein Sammelupdate für Windows 10 herausgebracht, welches ein paar Probleme lösen sollte. Allerdings hatte sich bei Benutzern, die Windows 10 Version 1903 benutzten und das Update installierten, ein Prozess von Cortana verklemmt. Dabei kam es erst zu einer dauerhaften CPU-Auslastung von ca 40 Prozent, da das Problem dann auftrat, wenn eine Anfrage an die Suchmaschine Bing von einer Methode wie einer Registry-Anfrage unterbunden wurde. Das Ergebnis dieses Deadlocks war, dass die Suchmaschine keine Suchergebnisse angezeigt hat.(vgl. Kolokythas 2020)

Es gibt unzählige Lösungen des Philosophen-Problems bis heute. Zu Anfang werden

Grundlagen von Prozessen, Programmen sowie des Schedulings vorgestellt. Das darauf folgende Kapitel wird eine Verklemmung definieren, sowie die Bedingungen für eine solche erklären. Lösungsansätze werden vorgestellt, sowie Möglichkeiten gezeigt, wie eine Verklemmung visualisiert werden kann. Lösungsansätze einer Verklemungssituation sind die Themen des letzten Kapitels.

2 Grundlagen

2.1 Erklärende Begriffe

Ein Programm ist die Beschreibung eines mechanischen Rechenverfahrens, sodass sie der Computer speichern und ausführen kann (vgl. Rechenberg 2000, S. 14). Vergleichbar ist dies mit einem Rezept. Wenn das Programm ausgeführt wird, nennt man es dann einen Prozess(vgl. Tanenbaum 2016, S.71).Das Programm löst sozusagen einen Prozess aus. In der Analogie entspricht dies dem Koch, welcher das Rezept ausführt. Dabei braucht ein Prozess auch "Zutaten", um das Programm auszuführen, diese entsprechen den Betriebsmitteln. Das entspricht in der Realität zum Beispiel Arbeitsspeicher oder Dateien, Ressourcen, die beschrieben werden können. Es gibt wiederverwendbare Betriebsmittel oder konsumierbare Betriebsmittel. Die wiederverwendbaren Betriebsmittel werden von Prozessen belegt und nach deren Ausführung zur nächsten Verwendung wieder freigegeben. Konsumierbare Betriebsmittel werden im laufenden System erzeugt (produziert) und anschließend zerstört (konsumiert).

Mit der Benutzeroberfläche des Bertriebssystems, die Schnittstelle zwischen Hard- und Software, lässt sich beobachten, dass die Programme auf den Kernen des Prozessors sehr schnell wechseln. In wenigen Milisekunden wechselt die CPU also zwischen den Programmen umher sodass zu einem Zeitpunkt nur ein Programm läuft.So können in einer Sekunde mehrere Programme laufen und es sieht es für unser Auge so aus, als würde alles gleichzeitig ausgeführt werden können. Damit das funktioniert, wird ein Scheduler benötigt, welcher die Prozesse in eine Reihenfolge bringt und entscheidet welcher Prozess wann

rechnen darf (vgl. Tanenbaum 2016, S.199).

Es kann allerdings auch passieren, dass ein Scheduler selbst einen verklemmten Prozess weiterrechnen lässt, was allerdings zu keinem Ergebnis führt. Wenn nun zwei Prozesse bereit zum Rechnen sind, um das gleiche Betriebsmittel konkurrieren und das Endergebniss der zwei Prozesse davon abhängt, welcher von den Prozessen als erstes rechnen darf, wird schnell der kritische Abschnitt erreicht. Der kritische Abschnitt umfasst „die Teile des Programms, in denen auf gemeinsam genutzter Speicher zugegriffen wird" (Tanenbaum 2016, S. 164). Wenn man diesen kritischen Abschnitt betritt, kommt es zu sogenannten Race Conditions. *Race Conditions* sind Situationen, „in denen zwei oder mehr Prozesse einen gemeinsamen Speicher lesen oder beschreiben und das Endergebnis davon abhängt, welcher wann genau läuft" (Tanenbaum 2016, S. 166 f.). Diese müssen vermieden werden wenn man in einer Situation ist, wo zwei Prozesse die gleiche Datei beschrieben werden sollen und es davon abhängt wann welcher Prozess rechnen darf, zum Beispiel wenn ein Dokument bedruckt werden soll. Dies funktioniert nicht wenn zwei Prozesse das Dokument beschreiben und dann kein inhaltlicher Zusammenhang mehr besteht. Verhindern kann man dies mit wechselseitigem Ausschluss. Es kommt also gar nicht dazu, dass zwei Prozesse das gleiche Betriebsmittel verwenden.

2.2 Begriff Verklemmungen

„Eine Verklemmung (Deadlock)bezeichnet einen Zustand, in dem die beteiligten Prozesse wechselseitig auf den Eintritt von Bedingungen warten, die nur durch andere Prozesse aus dieser Gruppe selbst hergestellt werden können." (J. Nehmer 2001, S.248). Diese wird durch Synchronisationsfehler erzeugt. Demnach macht keiner der Prozesse einen Fortschritt.

Es benötigt insgesamt vier Voraussetzungen damit letztendlich eine Verklemmung entsteht. Die erste ist der wechselseitige Ausschluss der Prozesse miteinander(engl. mutual exclusion). Dadurch ist ein Betriebsmittel unteilbar und exclusiv nutzbar. So werden zwar

Race Conditions vermieden aber es besteht nun die Vorraussetzung, dass zwei Prozesse nicht zur selben Zeit das gleiche Betriebsmittel verwenden können. Die Zweite ist das Nachfordern von den Betriebsmitteln ohne ein anderes loszulassen (engl. hold and wait). Die konkurrierenden Prozesse können nur schrittweise die Betriebsmittel belegen. Die dritte Bedingung besteht aus dem Fakt, dass einem Prozess die Betriebsmittel nicht entzogen werden können und nicht rückforderbar sind (engl. no preemption). Damit überhaupt eine Verklemmung vorliegen kann, müssen alle diese drei Bedingungen und eine Weitere eintreten. Diese vierte Bedingung heißt auf Englisch „circular wait“ und wird mit *zirkuläres Warten* übersetzt. Deshalb muss es eine geschlossene Kette an wartenden Prozessen geben. Um zu einem Ergebnis zu kommen benötigt Prozess A das, was Prozess B erst herstellen muss und Prozess B, das was Prozess A erst noch herstellen muss. Also kommt es zu einem wechselseitigen kreisförmigen(zirkulären) Warten(vgl. Baun 2017, S. 195f.).

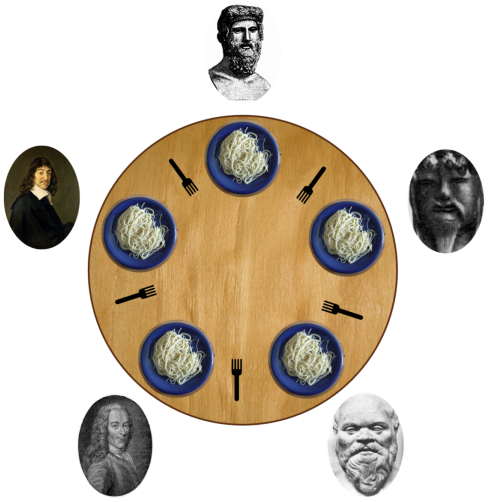
Neben dem Deadlock, bei dem der Prozesszustand bei `BLOCKED` ist, gibt es auch noch den Livelock. Bei diesem ist der Prozesszustand bei `RUNNING`, dabei führt die CPU den Prozess gar nicht aus da dieser mit einem anderen verklemmt ist. Der Livelock ist also wesentlich schwerer zu erkennen als der Deadlock (vgl. Tanenbaum 2016, S. 561 f.).

3 Die speisenden Philosophen

3.1 Vorstellung des Modells

Das 1965 von Dijkstra formulierte Philosophenproblem lässt sich wie folgt darstellen: Fünf Philosophen sitzen an einem runden Tisch. Jeder der Philosophen hat einen Teller mit Spagetti vor sich und weil die Nudeln so schlupfrig sind, benötigt man zwei Gabeln um sie zu essen. So liegt zwischen jedem Teller eine Gabel. Zusammengefasst gibt es nun fünf Teller mit den Spagetti und fünf Gabeln. Nun hat jeder Philosoph eine bestimmte Reihenfolge wie sie diese Tätigkeiten ausführen. Diese ist fest und unaustauschbar. Erst denken die Philosophen und weil Denken so hungrig macht, essen sie danach und heben erst die linke und dann die rechte Gabel auf. Dies kann auch lange gut gehen. Aber was passiet wenn alle Philosophen gleichzeitig hungrig sind und gleichzeitig nach der, von ihnen aus, linken Gabel greifen? Wenn jeder Philosoph nach seiner linken Gabel greift, hat keiner seine rechte Gabel. Also werden die Philosophen verhungern.(vgl. Tanenbaum 2016, S.220)

Abbildung 3.1: Modell (Wikipedia 2020)



Nun werden die Philosophen durch Prozesse ersetzt und die Gabeln durch Betriebsmittel und es kristalisiert sich eine Verklemmung heraus denn alle vier Bedingungen sind erfüllt. *Mutual exclusion*, da die Philosophen nicht zur selben Zeit mit der gleichen Gabel essen können. *Hold and wait*, da die Philosophen immer zuerst an sich denken und die Gabel keinem anderen überlassen. *No preemption*, die Philosophen sind gebildete Leute mit Manieren. Sie reißen einem anderem keine Gabel aus der Hand. Weil die Philosophen an einem runden Tisch sitzen und jeder auf die rechte Gabel wartet wird auch das *Zirkuläre Warten* erfüllt.

3.2 Darstellung des Problems

Um das Problem ausfindig zu machen, gibt es verschiedene Möglichkeiten eine Verklemmung grafisch darzustellen. Um das Problem besser auflösen zu können, muss es erst sichtbar gemacht werden.

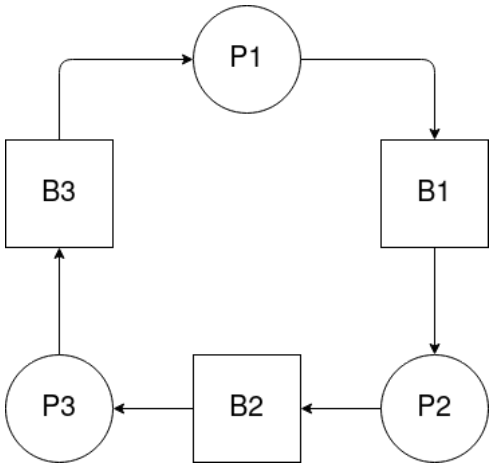
Vorgestellt wird zuerst der Betriebsmittelbelegungsgraph. Bei diesem werden die Prozesse mit Kreis und die Betriebsmittel mit Quadrat dargestellt. Diese werden ebenfalls als Knoten bezeichnet. Die Belegung und Anforderung auf Betriebsmittel wird mit Pfeilen,

auch Kanten genannt, dargestellt. Kommt dabei der Pfeil von dem Betriebsmittel zu dem Prozess, so bedeutet das, dass der Prozess dieses schon im Besitz hat. Verläuft der Pfeil von Prozess zu Betriebsmittel so stellt der Prozess die Anforderung an das Betriebsmittel. Der Grundaufbau wird in Abbildung 3.2 gezeigt.

Abbildung 3.2: Normaler Betriebsmittelbelegungsgraph

Bildet sich dabei eine zirkuläre Wartebeziehung so ist eine Verklemmung entstanden (vgl. Baun 2017, S. 197). In Abbildung 3.3 wird ein verklemmter Betriebsmittelbelegungsgraph dargestellt. P1 besitzt Betriebsmittel B3 und benötigt B1. P2 hat Betriebsmittel B1 im Besitz und stellt die Anforderung an B2, welches schon von P3 belegt ist. P3 belegt B3 und fordert B1 an. Damit schließt sich der Kreis und es entsteht eine Verklemmung, da die vorhandenen Prozesse in einer zirkulären Wartebeziehung stehen.

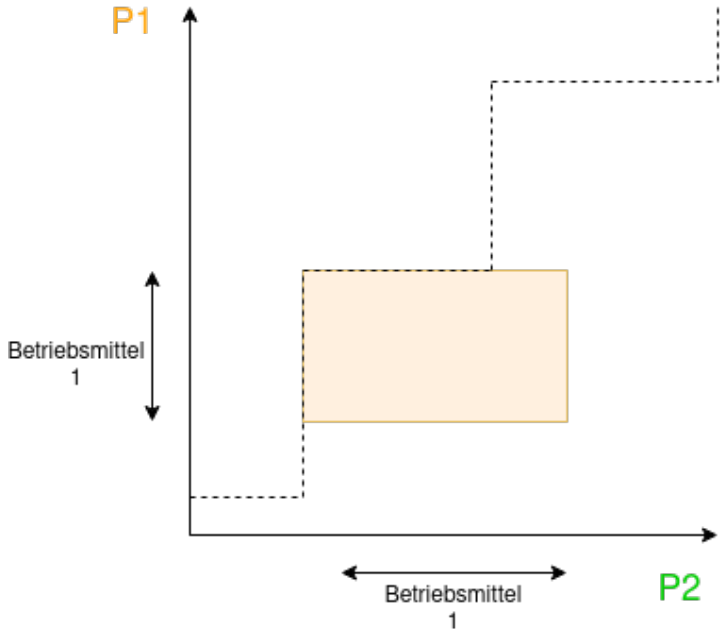
Abbildung 3.3: Bei diesem Beispiel liegt eine Verklemmung vor



Eine andere Möglichkeit, eine Verklemmung darzustellen, ist das Prozess-Ablaufdiagramm. Bei letzterem werden die Prozesse auf der x- bzw. y-Achse dargestellt und deren Prozessfortschritt als Linie. Verläuft die Linie in y-Richtung, so rechnet P1, verläuft sie in x-Richtung, rechnet P2. Die in dem Diagramm dargestellten “Ecken” symbolisieren die

Änderung des Rechenvorgangs von P1 auf P2. Auf jeden der Achsen sind außerdem die angeforderten Betriebsmittel gekennzeichnet. Das orangene Feld kennzeichnet die Spanne wann ein Betriebsmittel belegt wird. In diesen Bereich können die Prozesse nicht hinein. In Abbildung 3.4 wird ein normales Prozessablaufdiagramm ohne eine Verklemmung dargestellt.

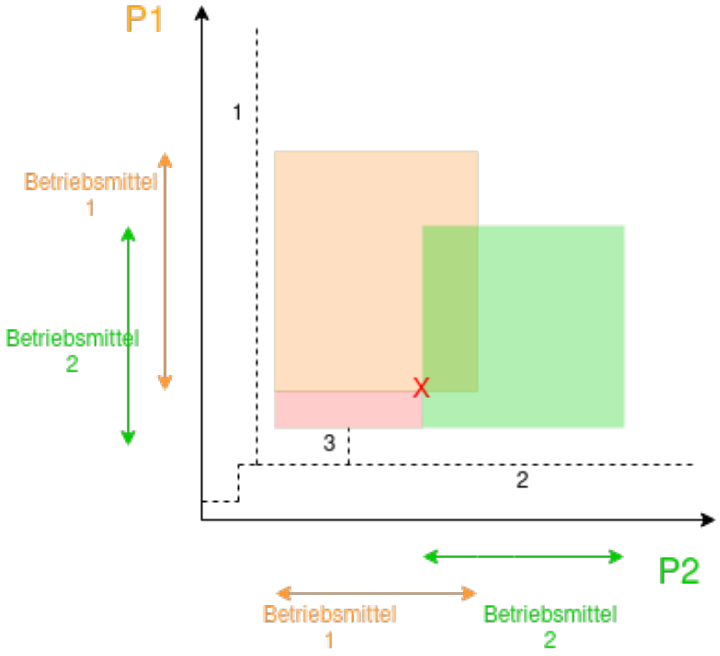
Abbildung 3.4: Möglicher Verlauf bei einem Prozessablaufdiagramm ohne Verklemmung



Nun wird ein zweites Betriebsmittel dazugenommen. Wie gewohnt rechnen die Prozesse jedoch kann bei dem Beispiel 3.5 eine Verklemmung entstehen. Die Zahlen 1-3 stellen mögliche Abläufe der Prozesse dar. Bei Ablauf 1 und 3 entsteht keine Verklemmung denn diese laufen jeweils links(1) und unterhalb(3) der gekennzeichneten Bereiche. Sie können ungehindert rechnen, denn sie belegen die Betriebsmittel, aber lassen sie anschließend wieder frei. Ablauf 3 jedoch gerät in den kritischen Abschnitt bis sich dann schließlich eine Verklemmung erkennen lässt. Verläuft die Linie des 3. Ablaufs weiter vertikal, so entsteht eine Verklemmung denn Betriebsmittel 1 ist bereits von Prozess 1 belegt. Rechnet Prozess 1 weiter, so entsteht eine Verklemmung, denn Betriebsmittel 2 ist bereits von Prozess 2 bereits belegt ist. Zusammengefasst, betritt die Linie 3 den roten Bereich, so ist eine

Verklemmung unvermeidbar. Eine Verklemmung ist bei einem solchen Diagramm nur möglich, wenn sich die Bereiche (in dieser Abbildung orange und grün gekennzeichnet) der Betriebsmittel so überschneiden, dass die Prozesse beide Betriebsmittel zur gleichen Zeit nutzen würden.

Abbildung 3.5: Beispiel eines Prozessablaufdiagrammes mit Verklemmung



3.3 Lösungsansätze

Eine der möglichen Lösungen besteht daraus, das Problem zu ignorieren und einfach nichts zu tun. Das ist wahrlich eine sehr naive Lösung, die dennoch häufig benutzt wird. Eine andere Lösung besteht darin, eine vermittelnde Instanz zu erschaffen, welcher die Regeln bzw. den Ablauf für die Tätigkeiten der Philosophen auf den Tisch legt. Die Reihenfolge der Tätigkeiten der Philosophen wird genau festgelegt. Bei dieser Lösung kann bei fünf Gabeln nur ein Philosoph essen.(vgl. Tanenbaum 2016, S.220)

Die sinnvollste Lösung für ein solches Problem ist folgende: Die Philosophen müssen fragen, ob die linke Gabel frei ist. Ist die linke Gabel nicht frei, so denkt er weiter. Ist die

linke Gabel frei, so fragt er nach der rechten Gabel. Ist diese nicht frei, legt er die linke Gabel wieder hin und denkt weiter. Erst wenn auch die rechte Gabel frei ist kann der Philosoph essen. Diese Lösung ermöglicht laut Tanenbaum „ein Maximum an Parallelität für eine beliebige Anzahl von Philosophen."(Tanenbaum 2016, S.222)

Diese Lösung kann mithilfe von Sperren (engl. Lock) erreicht werden. Eine Sperre kann nur die Werte 1 oder 0 annehmen. Gilt der Wert der Sperre als 1, so ist diese belegt. Ist sie 0, dann nicht. Damit ein Prozess eine Sperre besetzen kann, so muss der Wert der Sperre 0 sein(vgl. Mandl 2020, S.148).

4 Programmierung

4.1 Erste Lösung mit Problemdarstellung

In diesem Kapitel soll die mögliche Programmierung des Modells der speisenden Philosophen mithilfe der Programmiersprache Python3 vorgestellt werden.

In dem Programm werden die Bibliotheken „Multiprocessing" sowie „random" und „time" importiert. Die Klasse „Philosophen" erbt von der Klasse „Process" alle Funktionen und erhält zusätzlich die, die in der Klasse„Philosophen" definiert werden. Mit der Klasse „Process" kann ein Subprozess initiiert werden. Da sich die Philosophen über die gleichen Merkmale definieren können sie in einer Klasse zusammengefasst werden. Dem Konstruktor werden die Parameter `name` sowie `leftFork` und `rightFork` übergeben. Aus letzteren werden die Objekte leftFork und rightFork erzeugt. Diese stehen für die rechte sowie linke Gabel.

Listing 4.1: Klasse Philosophen

```
1
2 from multiprocessing import Process, current_process, RLock
3 import time
4 import random
5
6
7 class Philosophen(Process):
8   def __init__(self, name, leftFork, rightFork):
```

```
9       print("{} hat sich an den Tisch gesetzt".format(name))
10      Process.__init__(self, name=name)
11      self.leftFork = leftFork
12      self.rightFork = rightFork
```

Die von der Klasse „Process" geerbte `un()} -Funktion wu`de in das Programm eingefügt um die Handlungsfolge der Tätigkeiten zu bestimmen. In der Zeit, in der der Philosoph denkt, schläft der Prozess(Programmzeile 18) repräsentativ. Für den angegebenen Zeitraum wird die Ausführung des Programms angehalten. Ist diese Zeitspanne vorbei so geht der Programmcode weiter. In diesem Beispiel wird der besprochene Zeitraum zufällig mithilfe der `random` Methode zwischen einer und fünf Sekunden ausgewählt (*"time-Time access and conversions"* 2020). Wenn der Philosoph fertig mit denken ist, fordert er die linke Gable an. Danach soll er versuchen die rechte Gabel zu bekommen. Hat er beide Gabeln so kann er essen. Ist er damit fertig, so kann er erst die rechte und dann die linke Gabel wieder freigeben.

Listing 4.2: erste Lösung

```
1
2   def run(self):
3     print("{} hat angefangen zu denken"
4     .format(current_process().name))
5     #Philosoph x hat mit denken begonnen
6     while True:
7       time.sleep(random.randint(1, 5))
8       print("{} ist fertig mit denken"
9       .format(current_process().name))
10      self.leftFork.acquire()
11      #philosoph x hat die linke Gabel
12      time.sleep(random.randint(1, 5))
13      try:
```

```
14          print("{} hat die linke Gabel bekommen"
15          .format(current_process().name))
16
17          self.rightFork.acquire()
18          #Philosoph x hat die rechte Gabel
19          try:
20            print("{} hat beide Gabeln, isst gerade".format(current_proce
21
22          finally:
23            self.rightFork.release()
24            print("{} hat die rechte Gabel freigegeben"
25            .format(current_process().name))
26            #Philosoph ist fertg mit essen und gibt die rechte Gable wied
27
28        finally:
29          self.leftFork.release()
30          print("{} hat die linke Gabel freigegeben"
31          .format(current_process().name))
32          #Philosoph x hat gibt die linkte Gabel frei
```

Nun der Lösungsansatz hierbei ist, dass die Gabeln durch einen RLock repräsentiert werden. Das heißt die Prozesse besetzen immer Sperren. Um die Sperren zu besetzen, ruft der Prozess die `aquire()`-Funktion auf. Um sie wieder freizugeben ruft er die `release()`-Funktion auf. Diese „Lösung" ist jedoch nicht deadlockfrei. Wollen zwei Prozesse die gleiche Sperre besetzen so verklemmt das Programm sofort.

4.2 Lösungen

Um diesem Problem vorzubeugen kann man der `aquire()`-Funktion über den Parameter `blocking` ein True oder ein False mitgeben. Setzt man blocking auf True, so geht der Programmcode nicht weiter bis die Funktion ausgeführt ist, in diesem Fall wenn der Lock besetzt ist. Das heißt, will ein Prozess den Lock besetzen, so muss er immer wieder versuchen, diesen zu besetzen bis dieser wieder freigegeben ist.

Listing 4.3: Endlösung

```
1
2   def run(self):
3     print("{} hat mit denken begonnen"
4     .format(current_process().name))
5     #Philosoph x hat mit denken begonnen
6     while True:
7       time.sleep(random.randint(1, 5))
8       print("{} ist fertig mit denken"
9       .format(current_process().name))
10      #Philosoph x ist bereit zum Essen
11
12      self.leftFork.acquire(True)
13      #prozess x will den Lock belegen und muss es so lange versuchen b
14      time.sleep(random.randint(1, 5))
15      try:
16        print("{} hat die linke Gabel"
17        .format(current_process().name))
18
19        locked = self.rightFork.acquire(False)
20        #Prozess x versucht nur einmal den lock zu besetzen
21        if locked:
```

```
22            print("{} hat beide Gabeln, isst gerade"
23            .format(current_process().name))
24             #Prozess x hat beide Locks besetzt
25            self.rightFork.release()
26            print("{} hat die rechte Gabel freigegeben"
27            .format(current_process().name))
28            self.leftFork.release()
29            print("{} hat die linke Gabel freigegeben"
30            .format(current_process().name))
31            #wenn Zustand = locked ist(die Philosophen gegessen haben), w
32          else:
33            self.leftFork.release()
34            #wenn Zustand != locked, dann Lock "leftFork" frei geben(weil
35            print("{} hat die linke Gabel freigegeben"
36            .format(current_process().name))
```

Der Prozess wird immer wieder versuchen die linke Gabel zu belegen, bis diese freigegeben ist. Dann soll er versuchen die rechte Gabel zu erlangen. Da dieser RLock auf False gesetzt ist, wird er das nur einmal versuchen. Zusammengefasst gibt es blockierende Funktionen, bei ihnen versucht der Prozess solange den Lock zu besetzen bis dieser frei ist und es gibt nichtblockierende Funktionen, welche nur einmal versuchen den Lock zu ergattern (vgl. J. Nehmer 2001, S.164). Geprüft werden kann dies an der Variable „locked". Scheitert dieser Versuch, so ist der Prozess nicht auf „locked" gesetzt und die linke Gabel wird wieder freigeben. Ist der Zustand des RLock's auf locked, so ist der Versuch gelungen. Der Philosoph hat nun beide Gabeln, kann essen, dann die rechte und zuletzt auch die linke Gabel wieder freigeben. Nun kann der nächste Prozess wieder versuchen den Lock zu ergattern.

5 Fazit

Literatur

"time- Time access and conversions" (2020). Letzter Stand 29.12.2020, 11:01. URL: `https://docs.python.org/3/library/time.html` (besucht am 29. 12. 2020).

Baun, C. (2017). *Betriebssysteme kompakt*. Berlin: Springer-Verlag.

J. Nehmer, P. Sturm (2001). *Grundlagen moderner Bertriebssysteme*. Heidelberg: dpunkt.verlag.

Kolokythas, Panagiotis (2020). *"Windows 10: Update-Bug sorgt für hohe CPU-Auslastung"*. Letzter Stand 22.11.2020, 15:02. URL: `https://www.pcwelt.de/news/Windows-10-Update-Bug-sorgt-fuer-hohe-CPU-Auslastung-10657717.html` (besucht am 22. 11. 2020).

Mandl, P. (2020). *Grundkurs Betriebssysteme*. Wiesbaden: Springer-Vieweg.

Rechenberg, P. (2000). *Was ist Informatik*. München, Deutschland und Wien, Österreich: Hanser.

Tanenbaum, A. (2016). *Moderne Betriebssysteme*. 4. Aufl. Hallbergmoos: Pearson.

Wikipedia (2020). *Aufbau des Philosophenproblems*. Letzter Stand 31.12.2020, 13:21. URL: `https://de.wikipedia.org/wiki/Philosophenproblem` (besucht am 31. 12. 2020).

Materialienverzeichnis

3.1 Modell (Wikipedia 2020) 9

3.2 Normaler Betriebsmittelbelegungsgraph 10

3.3 Bei diesem Beispiel liegt eine Verklemmung vor 10

3.4 Möglicher Verlauf bei einem Prozessablaufdiagramm ohne Verklemmung 11

3.5 Beispiel eines Prozessablaufdiagrammes mit Verklemmung 12

Tabellenverzeichnis