HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG (HTWG)
Fakultät Informatik
Rechner- und Kommunikationsnetze
Prof. Dr. Dirk Staehle

Vorlesung Rechnernetze

Laborübung zum Einstieg in Python

Simulation eines Supermarkts

Prof. Dr. Dirk Staehle

Die Abgabe erfolgt durch Hochladen der bearbeiteten Word-Datei in Moodle.

Bearbeitung in Zweier-Teams

Team-Mitglied 1:

Team-Mitglied 2:

1. Einleitung

Diese Aufgabe dient dazu, die Programmiersprache Python und insbesondere die Programmierung mit Threads in Python kennenzulernen, die später bei der Programmierung von verteilten Anwendungen mit TCP und UDP Sockets benötigt wird.

Als Beispiel wird hier die Simulation eines Supermarktes gewählt, in dem Kunden in bestimmten Abständen ankommen, um einzukaufen. Während des Einkaufs durchlaufen die Kunden bestimmte Stationen wie den Bäcker, die Wurstttheke, die Käsetheke oder die Kasse. An diesen Stationen müssen die KundInnen eventuell warten, bis sie bedient werden. Zwischen den Stationen kaufen die KundInnen im Supermarkt für eine bestimmte Zeit ein, ohne sich gegenseitig zu stören.

Die Simulation dient zusätzlich der Veranschaulichung von Netzwerken, in denen eine KundIn einem Paket entspricht und jede Station einem Netzknoten, an dem ein Paket warten muss, bis es übertragen wird. Das Wechseln zwischen den Stationen entspricht der Zeit, die ein Paket benötigt, um eine Leitung von einem Netzknoten zum nächsten zu durchlaufen.

Die Simulation soll in zwei Varianten implementiert werden: einer ereignisorientierten Simulation und einer Realzeitsimulation.

2. Simulationsmodell

Der Supermarkt besteht aus einer Reihe von Stationen, die von KundInnen in einer bestimmten Reihenfolge durchlaufen werden. An jeder Station arbeitet eine Service-Kraft, die die Anfragen einer KundIn in einer bestimmten Zeit bedient. Die Zeit, die sich eine KundIn an einer Station X befindet, ist also die Anzahl N_x der Anfragen der KundIn multipliziert mit der Zeit S_x, die die Service-Kraft benötigt, um die Anfrage zu erfüllen.

Eine KundIn, die 3 Brezeln und 7 Brötchen an der Bäckerei kauft, benötigt z.B. 100 Sekunden, wenn die BäckereifachverkäuferIn 10 Sekunden pro Gebäckstück benötigt. Komplexere und realitätsnähere Modelle, in dem der Aufenthalt sich aus Anzahl unterschiedlicher Gebäcksorten, Anzahl Gebäckstücke pro Gebäcksorte plus Bezahlvorgang ergibt, sind später sehr einfach in die Simulation zu integrieren, wenn der Kern der Simulation steht.

Im Supermarkt kaufen mehrere KundInnen gleichzeitig ein und es kann vorkommen, dass eine KundIn an einer Station warten muss. Die KundIn reiht sich dann in die Warteschlange ein, lässt die Station aus oder bricht den Einkauf ab, wenn die Warteschlange zu lang ist.

Jede KundIn betritt den Supermarkt zu einer bestimmt Zeit. Dann durchläuft die KundIn Stationen des Supermarkts in einer individuellen Reihenfolge und benötigt eine bestimmte Zeit, um von einer Station zur nächsten zu wechseln und währenddessen im Supermarkt einzukaufen. Diese Zeit ist unabhängig davon, was die anderen KundInnen im Supermarkt machen.

Der Aufenthalt wird also festgelegt durch den Beginn des Einkaufs sowie einer Liste der Stationen. Für jede Station X wird festgelegt,

1. die Zeit T_x, die die KundIn benötigt, um vom Beginn des Einkaufs bzw. von der letzten Station Y bis zu der Station X zu gelangen

2. ab welcher Warteschlangenlänge W_x die KundIn die Station X auslässt

3. die Anzahl der Einkäufe N_x, die die KundIn an der Station X tätigt

2.1. Beispiel: Supermarkt mit vier Stationen

- Bäcker mit 10 Sekunden pro Gebäckstück
- Wurstttheke mit 30 Sekunden pro Wurstsorte
- Käsetheke mit 60 Sekunden pro Käsesorte
- Kasse mit 5 Sekunden pro Ware

Zwei Typen von KundInnen:

- Typ 1 (vollständiger Einkauf):
  - Bäcker ($T_{Bäcker} = 10s, W_{Bäcker} = 10, N_{Bäcker} = 10$)
  - Wurstttheke ($T_{Wurst} = 30s, W_{Wurst} = 10, N_{Wurst} = 5$)
  - Käsetheke ($T_{Käse} = 45s, W_{Käse} = 5, N_{Käse} = 3$)
  - Kasse ($T_{Kasse} = 60s, W_{Kasse} = 20, N_{Kasse} = 30$)
- Typ 2 (Leberkäs-Semmel):
  - Wurstttheke ($T_{Wurst} = 30s, W_{Wurst} = 5, N_{Wurst} = 2$)
  - Kasse ($T_{Kasse} = 30s, W_{Kasse} = 20, N_{Kasse} = 3$)
  - Bäcker ($T_{Bäcker} = 20s, W_{Bäcker} = 20, N_{Bäcker} = 3$)

Die erste KundIn vom Typ 1 beginnt bei Simulationsbeginn zum Zeitpunkt 0s den Einkauf und alle 200s treffen weitere KundInnen vom Typ 1 ein. Die erste KundIn vom Typ 2 beginnt 1s nach Simulationsbeginn den Einkauf und alle 60s treffen weitere KundInnen vom Typ 2 ein. Ziel der Simulation ist zu bestimmen, welche KundInnen den Einkauf vollständig durchführen, wie lange ein vollständiger Einkauf inklusive Wartezeiten dauert und welche Stationen aufgrund der langen Warteschlange gemieden werden.

2.2. Lastanalyse

- Bäcker:
  - alle 200s ein Kunde von Typ 1 mit 10*10s=100s Bearbeitungsdauer
  - alle 60s eine Kunde vomTyp 2 mit 3*10s=30s Bearbeitungsdauer
  - Last durch Kunden vom Typ 1: 100s Arbeitszeit alle 200s=100s /200s = 0,5 (50%)
  - Last durch Kunden vom Typ 2: 30s Arbeitszeit alle 60s =30s / 60s = 0,5 (50%)
  - Gesamtlast: 50% (Typ 1) + 50% (Typ 2) = 100% (Bäcker ist voll ausgelastet)
- Wurstttheke:
  - Last durch Kunden vom Typ 1: 150s Arbeitszeit alle 200s = 150s /200s = 0,75 (75%)
  - Last durch Kunden vom Typ 2: 60s Arbeitszeit alle 60s =60s / 60s = 1,0 (100%)
  - Gesamtlast: 75% (Typ 1) + 100% (Typ 2) = 175% (Wurstttheke ist überlastet)

- Käsetheke:
    - Last durch Kunden vom Typ 1: 180s Arbeitszeit alle 200s=180s /200s = 0,9 (90%)
    - Gesamtlast: 90% (Typ 1) + 0% (Typ 2)=90% (Wurstttheke ist stark ausgelastet)
- Kasse: 150s/200s + 15s / 60s = 1 (zu 100% ausgelastet)
    - Last durch Kunden vom Typ 1: 150s Arbeitszeit alle 200s=150s /200s = 0,75 (75%)
    - Last durch Kunden vom Typ 2: 15s Arbeitszeit alle 60s = 15s / 60s = 0,25 (25%)
    - Gesamtlast: 75% (Typ 1) + 25% (Typ 2)=100% (Kasse ist voll ausgelastet)

3. Ereignisorientierte Simulation

Bei einer ereignisorientierten Simulation wird das System, d.h. der Supermarkt und die KundInnen, nur betrachtet, wenn ein Ereignis eintritt, das den Systemzustand verändert. Der Systemzustand ist im Beispiel des Supermarkts die Anzahl und der Typ der KundInnen, die an den verschiedenen Stationen warten. Dementsprechend sind die Ereignisse der Beginn eines Einkaufs, das Eintreffen einer KundIn an einer Station und das Verlassen einer Station durch eine KundIn. Der Beginn und das Ende eines Bedienvorgangs tritt auch immer zusammen mit einem der beiden vorgenannten Ereignisse ein.

Die Steuerung des zeitlichen Ablaufs einer ereignisorientierten Simulation erfolgt über eine Ereignisliste, in der alle Ereignisse verwaltet und in zeitlich aufsteigender Reihenfolge abgearbeitet werden. Die Ereignisse werden also in der Ereignisliste nach aufsteigenden Ereigniszeitpunkten sortiert, und das vorderste Ereignis in der Liste findet als nächstes statt.

Ein Ereignis kann bewirken, dass sich der Systemzustand ändert oder neue Ereignisse erzeugt und in die Ereignisliste einsortiert werden. Zudem erfolgt bei der Abarbeitung eines Ereignisses die Erfassung der Statistiken zur Bestimmung des Simulationssergebnisses. Die Simulationszeit entspricht immer dem Zeitpunkt des gerade abgearbeiteten Ereignisses und springt also von Ereignis zu Ereignis.

3.1. Beispiel

Im Supermarktbeispiel haben wir folgende Ereignisse, für die auch die resultierenden Aktionen aufgeführt sind:

- Beginn des Einkaufs
    - Ereignis Ankunft an der ersten Station erzeugen
    - nächstes Ereignis Beginn des Einkaufs für den gleichen KundInnen-Typ erzeugen
- Ankunft an einer Station
    - anhand der Warteschlangenlänge überprüfen, ob an der Station eingekauft wird
    - wenn eingekauft wird, entweder einreihen in die Warteschlange (Systemzustand ändern) oder im Falle einer direkten Bedienung das Ereignis Verlassen der Station erzeugen
    - wenn nicht eingekauft wird, direkt das Ereignis Ankunft an der nächsten Station erzeugen
- Verlassen einer Station
    - Ereignis Ankunft an der nächsten Station erzeugen

- wenn sich weitere KundInnnen in der Warteschlange befinden, erste KundIn aus der Warteschlange nehmen und Ereignis Verlassen der Station für die nächste KundIn erzeugen

Die Initialisierung der Simulation erfolgt, indem ein „Beginn des Einkaufs“-Ereignis pro KundInnen-Typ erzeugt wird.

In Moodle finden Sie eine Excel-Datei für eine Beispiel-Simulation mit den oben angegebenen Parametern.

3.2. Implementierung

Der Kern der Implementierung der ereignisoprientierten Simulation ist die Ereignisliste. In Python bietet sich hier die Verwendung einer Heap-Queue aus dem Modul heapq.py an. In einer Heap-Queue bestehen Einträge aus Tupeln, die in der Reihenfolge ihrer Einträge sortiert werden. Z.B.

(1,2,3,4,5)
(1,2,4,1,1)
(2,1,1,1,1)
…

Ein Ereignis wird als 5er-Tupel realisiert, das aus dem Ereigniszeitpunkt, der Ereignispriorität, der Ereignisnummer, der Ereignisfunktion und optional den Ereignisargumenten besteht. Die Ereignispriorität dient dazu, die Abarbeitungsreihenfolge bei gleichzeitigen Ereignissen unterschiedlichen Typs festzulegen. Wenn also das Verlassen einer Station eine höhere Priorität (einen niedrigeren Prioritätswert) hat als die Ankunft an einer Station, dann wird bei Gleichzeitigkeit zunächst die Warteschlange verringert und erst dann geprüft, ob die neue KundIn die Station meidet oder nicht. Die Ereignisnummer wird global hochgezählt und sorgt für eine eindeutige Reihenfolge, falls zwei Ereignisse derselben Priorität zum gleichen Zeitpunkt stattfinden. Die Ereignisfunktion wird mit den Ereignisargumenten bei der Abarbeitung des Ereignisses aufgerufen. Die Klasse Ev ist in der Skelett-Implementierung bereits vorgegeben.

Die Ereignisliste soll als Klasse implementiert werden, die als Klassenvariablen die Heap-Queue und die Simulationszeit besitzt. Zudem definiert die Klasse die Methoden event=pop(), push(event) und start(), wobei hier event nicht zwangsweise eine Klasse bedeutet, sondern auch als Tupel aus Zeit, Priorität und Ereignisfunktion realisiert werden kann. In der Methode pop() wird die Ereignisfunktion aufgerufen, die im entsprechenden Event hinterlegt ist.

Die Simulation wird gestartet, in dem die Methode start() aufgerufen wird. Die Methode start() besteht aus einer Schleife, die solange Ereignisse aus der Ereignisliste nimmt und deren Ereignisfunktion aufruft, bis die Ereignisliste leer ist.

Neben der Ereignisliste sind die Klassen Station und KundIn zu implementieren.

Die Klasse KundIn wird über eine Liste aus 3er-Tupeln spezifiziert, die die noch zu besuchenden Stationen beschreibt. Hinweis: Verwenden Sie den Befehl `L2=list(L1)` im Konstruktor, um eine Liste zu kopieren bzw. eine neue Liste `L2` mit den Elementen der Liste `L1` zu erzeugen. In der Klasse werden Instanz-Methoden für die Arbarbeitungsfunktionen der Ereignisse Beginn des Einkaufs, Ankunft an einer Station und Verlassen einer Station bereitgestellt.

Die Klasse Station wird über die Abarbeitungdauer (pro Gebäckstück, pro Wurstsorte, etc.) definiert. Die Klasse bietet den KundInnen die Schnittstelle queue(), über die sich die KundIn an der Station anstellen kann und entweder direkt bedient wird oder warten muss. Die Warteschlange kann über eine einfache

Liste realisiert werden. Das Ende des Bedienvorgangs wird über ein Ereignis realisiert, für das eine Abarbeitungsmethode implementiert werden muss.

4. Realzeitsimulation

Die Realzeitsimulation läuft mit realer Geschwindigkeit. Wenn also die Wartezeit an der Wursttheke zwei Minuten beträgt, so vergehen auch in der Simulation zwei Minuten. Eine Realzeitsimulation wird über Threads realisiert und dient unter anderem dazu, dass Sie den Umgang mit Threads, speziell des Moduls `threading`, erlernen.

In der Realzeitsimulation implementieren Sie alle Stationen und KundInnen als Threads, indem Sie die Klassen Station und KundIn von der Klasse Thread ableiten. Die Station-Threads werden zu Beginn der Simulation gestartet und warten, bis Sie von einem KundIn-Thread aufgeweckt werden. Die Zeit für die Bedienung einer KundIn wird simuliert, in dem sich der Station-Thread für die Bediendauer schlafen legt. Das wird über die `sleep`-Funktion aus dem Modul `time` realisiert. Danach wird die bediente KundIn aufgeweckt. Wenn keine weiteren KundInnen in der Warteschlange sind, wartet die Station auf die Ankunft des nächsten Kunden.

Die KundIn-Threads werden zum Beginn des Einkaufs gestartet. Das Einkaufen im Supermarkt zwischen den Stationen wird simuliert, in dem der Thread schlafen gelegt wird (`time.sleep`). Wenn eine KundIn an einer Station eintrifft, wird die KundIn in die Warteschlange eingereiht und der wartende Stations-Thread aktiviert.

Das Aufwecken von wartenden Threads wird über `Events` oder `Conditions` aus dem Modul `threading` realisiert. In der Simulation ist die Verwendung von `Events` ausreichend.

Am Anfang der Simulation wird beispielsweise für jede Station ein Ankunfts-Event `CustomerWaitingEv` erzeugt. Die Station wartet darauf, dass das Ereignis eintrifft `CustomerWaitingEv.wait()`. Wenn eine KundIn bei der Station eintrifft, wird sie in die Warteschlange eingereiht und das Ankunfts-Ereignis wird mit `CustomerWaitingEv.set()` ausgelöst. Die KundIn wartet darauf, dass der Bedienvorgang zu Ende ist. Dazu wird von der Station ein Bedienende-Event `servEv` erzeugt und in der Warteschlange zusammen mit der Anzahl Einkaufsvorgänge bei der Station abgespeichert. Der KundIn-Thread legt sich dann mit einem `servEv.wait()` schlafen, bis er vom Stations-Thread mit einem `servEv.set()` wieder aufgeweckt wird.

Wenn eine Station eine KundIn fertig bedient hat und keine weiteren KundInnen mehr warten, legt sich der Thread wieder schlafen und wartet aus das Eintreffen des nächsten Kunden. Dazu wird das Ankunfts-Ereignis wieder zurückgesetzt mit `arrEv.clear()`.

Beim Verwalten der Warteschlange und beim Aktivieren/Deaktivieren des Ankunfts-Ereignisses muss auf Race-Conditions geachtet werden, d.h. diese Operationen sollten in einem Block ohne Unterbrechung durch einen anderen Thread ablaufen. Dazu gibt es im Modul `threading` die Klasse `Lock`.

Die KundInnen-Threads können in einem Start-Thread pro KundInnen-Typ gestartet werden, der sich nach dem Starten eines KundInnen-Threads bis zum nächsten Einkaufsbeginn schlafen legt.

Das Ende der Simulation ist erreicht, wenn alle Start-Threads und alle KundInnen-Threads beendet sind. Das können Sie entweder über eine Thread-Verwaltung oder über einen KundIn-Zähler als Klassen-Valiable der Klasse KundIn realisieren.

5. Simulationsergebnisse

Das Ziel der Simulationen ist, zu bestimmen

- wann der/die letzte KundIn bedient wurde
- wie viele KundInnen vollständig bedient wurden
- wie lange ein vollständiger Einkauf dauert
- pro Station, welcher Prozentsatz der KundInnen die Station auslässt

Starten Sie dazu die Simulation mit der obigen Beispielskonfiguration und generieren Sie für 30 Minuten neue Kunden. Bei der Realzeitsimulation empfiehlt es sich, die Zeiten um einen bestimmten Faktor zu verkürzen sonst läuft die Simulation tatsächlich mehr als 30 Minuten. Insbesondere während der Debug-Phase ist das sehr langwierig.

Die Erfassung der Simulationsergebnisse können Sie recht einfach über Klassenvariablen in der KundInnen-Klasse realisieren. Für die Erfassung der Ergebnisse pro Station sind Dictionaries geeignet. Generell empfiehlt es sich, den Stationen und KundInnen Namen zu geben und Log-Daten auszugeben, um den Simulationsverlauf analysieren zu können. Hier sollten Sie Nachrichten erzeugen wie

5.2s: Ankunft KundIn A4 (vierte KundIn von Typ 4) an Station Bäcker

Eventuell ist auch eine Ausgabe der Ereignisliste oder die Anzeige der ausgeführten und hinzugefügten Ereignisse hilfreich.