Stefan Waidele
Ensisheimer Straße 2
79395 Neuenburg am Rhein
Stefan@Waidele.info

AKAD University
Immatrikulationsnummer: 102 81 71

Assignment im Modul JAV02

Erstellung einer Multithread—Anwendung zur Untersuchung von Erzeuger–Verbraucher–Szenarien

Betreuer: Prof. Dr. Franz–Karl Schmatzer

14. Februar 2015

AKAD University

Inhaltsverzeichnis

Abbildungsverzeichnis **iii**

Tabellenverzeichnis **iv**

Abkürzungsverzeichnis **v**

1 Einleitung **1**
1.1 Begründung der Problemstellung 1
1.2 Ziele dieser Arbeit 1
1.3 Abgrenzungen 2

2 Erzeuger und Verbraucher **3**
2.1 Erzeuger–Verbraucher—Muster 3
2.2 Zwischenspeicher 3
2.3 Kritische Abschnitte 4
2.4 Beachtenswerte Zustände 4
2.5 Freiwilliges und unfreiwilliges Warten 4
2.6 Realisierung mit Semaphoren und Mutex 4

3 Genutzte Sprachmerkmale von JAVA **5**
3.1 Generische Klassen 5
3.2 Iteratoren 5
3.3 Threads 5
3.4 Realisierung mit LinkedBlockingQueue 5
3.5 Zufall 5

4 Implementierung **6**
4.1 Klasse: Akteur 6
4.2 Klasse: Erzeuger 6
4.3 Klasse: Verbraucher 6
4.4 Klasse: Logger 6
4.5 Sonstige Programmmerkmale 6

5 Laufzeitbetrachtungen **7**
5.1 Erzeuger schneller als Verbraucher 7
5.2 Erzeuger langsamer als Verbraucher 7
5.3 Ezeuger und Verbraucher gleich schnell 7
5.4 Reduzierung auf einen Erzeuger und einen Verbraucher 7

6 Fazit & Ausblick **8**
6.1 Fazit 8
6.2 Ausblick 8

A Quelltext der Anwendung **9**

B Ergebnisse verschiedenener Programmläufe **9**

Literatur– und Quellenverzeichnis **vi**

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

FIFO	First In, First Out
LIFO	Last In, First Out
Mutex	Mutually Exclusive
E	Erzeuger
V	Verbraucher
Q	Queue
P	Produkt

1 Einleitung

1.1 Begründung der Problemstellung

Erzeuger–Verbraucher–Konstellationen sind sowohl in der Wirtschaft als auch in der Informationsverarbeitung sehr häufig zu beobachten und prägen viele alltäglichen Vorgänge. Aufgrund der Vielzahl der zu betrachtenden Parameter, die neben der Anzahl der Erzeuger und Verbraucher auch die jeweilige Produktions- bzw. Verbrauchsgeschwindigkeit und die Lagerkapazität für fertige Erzeugnisse einschließt handelt es sich hierbei trotz ihrer Alltäglichkeit um komplexe Systeme, welche mich rein oberflächlichen Untersuchungen nicht komplett erfasst werden können.

Des Weiteren sind Erzeuger–Verbraucher–Szenarien auch in der Informatik anzutreffen. Dies gilt sowohl für die Systemebene[1] als auch auf der Anwendungsebene[2]

1.2 Ziele dieser Arbeit

Ziel dieser Arbeit ist es, eine Erzeuger–Verbraucher Anwendung in Java zu entwickeln. Anhand dieser Anwendung sollen die typischen Problemstellungen nebenläufiger Anwendungen diskutiert, sowie das Laufzeitverhalten der erstellten Anwendung beobachtet und erörtert werden.

Die zu erstellende Anwendung soll als konkretes Beispiel eine Pizzeria simulieren, die laufend Pizzas herstellt und für Kunden zur Abholung bereit hält. Mehrere Verbraucher entnehmen in zufälligen Intervallen Pizzas aus diesem Vorrat. Hierbei werden sowohl der Erzeuger als auch jeder Verbraucher in einem jeweils eigenen Threads simuliert. Es ist darauf zu achten, dass kein Kunde eine Pizza aus einem leeren Vorratsspeicher entnehmen kann. Ebenfalls kann der Pizzabäcker keine

[1] z.B. Logging oder Interrupts
[2] z.B. die Darstellung von Messdaten oder der Abruf von Netzwerkresourcen

weitere Pizza in einen bereits vollen Vorratsspeicher hinzufügen. Sowohl beim Einstellen als bei der Abholung soll zur Dokumentation des Laufzeitverhaltens Erzeuger (E) bzw. Verbraucher (V) sowie die in der Warteschlange bzw. Queue (Q) befindliche Anzahl der Pizzen bzw. Produkte (P) ausgegeben werden.

Zunächst werden hierzu in den Kapitel *2 Erzeuger und Verbraucher* und *3 Genutzte Sprachmerkmale von JAVA* durch Literaturrecherche die Grundlagen der Problemstellung sowie die notwendigen Werkzeuge der Programmiersprache herausgearbeitet. Anschließend wird in Kapitel *4 Implementierung* das Simulationsprogramm erstellt. Die unterschiedlichen Simulationsläufe liefern die Daten für die Untersuchung des Laufzeitverhaltens in Kapitel *5 Laufzeitbetrachtungen.*

1.3 Abgrenzungen

2 Erzeuger und Verbraucher

2.1 Erzeuger–Verbraucher—Muster

Bei Erzeuger–Verbraucher–Konstellationen treten Produzenten und Konsumenten miteinander über eine Warteschlange bzw. einen Buffer miteinander in Kontakt.[3]

Hierbei ist es unerheblich, was hierbei erzeugt bzw. verbraucht wird. Es kann sich um reale Güter handeln, die von Kunden nachgefragt werden, oder auch um Informationen, die Abgerufen werden. Ebenfalls ist es unerheblich, ob der Erzeuger tatsächlich den Bedarf des Verbrauchers befriedigt, oder ob Anfragen erzeugt werden, die dann vom Verbraucher abgearbeitet bzw. gelöst werden.[4] Auch können die Anzahl der Erzeuger und die der Verbraucher beliebig gewählt werden oder sogar schwanken. Allen Situationen ist jedoch gemeinsam, dass das Erzeugnis nur einem Verbraucher zur Verfügung steht und nur ein mal konsumiert werden kann.

Durch das einfügen eines Zwischenspeichers stellen Erzeuger–Verbraucher–Konstellationen einen Mechanismus dar, durch den eine beliebige Anzahl von Produzenten und Verbraucher miteinander kommunizieren können. Es handelt es sich somit um ein Entuwrfsmuster, welches in der Programmierung von nebenläufigen Anwendungen.[5]

2.2 Zwischenspeicher

Bei der Zwischenspeicher (engl. Buffer) können verschiedene Prinzipien zum Einsatz kommen. Bei Last In, First Out (LIFO) werden diejenigen Elemente zuerst entnommen, die als letztes hinzugefügt wurden. Dieses Verfahren kommt bei der Implementierung mithilfe eines Kellerspeichers (engl. Stack) zum Tragen.

[3] vgl. Tanenbaum & Woodhull (2005), S. 76f
[4] Ein Beispiel hierfür ist z.B. ein Trouble–Ticket–System: Hier werden vom Nutzer Supportanfragen „erzeugt“, die dann von den Supportmitarbeitern „verbraucht” werden.
[5] vgl. Hoffmann & Lienhart (2008), S. 111

Das First In, First Out (FIFO) Prinzip, bei dem das zuerst hinzugefügte Element auch als erstes wieder entnommen wird kann durch eine Warteschlange (engl. Queue) oder mithilfe eines Ringbuffers implementiert werden.[6]

Bei der im Rahmen dieser Arbeit wird eine FIFO–Warteschlange genutzt.[7]

2.3 Kritische Abschnitte

Bei FIFO *müssten* gleichzeitiges Lesen und Schreiben möglich sein.

Erklärung von Mutex und Semaphoren.[8]

2.4 Beachtenswerte Zustände

Schreiben bei voller Queue, Lesen bei leerer Queue

2.5 Freiwilliges und unfreiwilliges Warten

2.6 Realisierung mit Semaphoren und Mutex

[6] vgl. Sedgewick & Wayne (2011), Abschnitt 1.3.3.8 und 1.3.3.9
[7] vgl. Oracle (Hrsg.) (1993)
[8] vgl. Silberschatz et al. (2012)

3 Genutzte Sprachmerkmale von JAVA

3.1 Generische Klassen

3.2 Iteratoren

3.3 Threads

3.4 Realisierung mit LinkedBlockingQueue

3.5 Zufall

4 Implementierung

4.1 Klasse: Akteur

4.2 Klasse: Erzeuger

4.3 Klasse: Verbraucher

4.4 Klasse: Logger

4.5 Sonstige Programmmerkmale

5 Laufzeitbetrachtungen

5.1 Erzeuger schneller als Verbraucher

5.2 Erzeuger langsamer als Verbraucher

5.3 Ezeuger und Verbraucher gleich schnell

5.4 Reduzierung auf einen Erzeuger und einen Verbraucher

Facade–Designpattern: Die n Verbraucher mit zufälligen Wartezeiten verhalten sich wie 1 Verbraucher mit kürzeren, aber ebenfalls zufälligen Wartezeiten. Dadurch ist die Anzahl der Verbraucher und analog dazu auch die der Erzeuger für die Betrachtung irrelevant.

6 Fazit & Ausblick

6.1 Fazit

6.2 Ausblick

Ausbau der Anwendung, um verschiedene Szenarien simulieren zu können, z.B. variables Arbeitstempo der Erzeuger (höhere Belastung von Maschinen oder Menschen), zusätzlichen Erzeugern (Wie lange dauert das Anfahren?), Hinterlegung von Kosten für Lagerhaltung, Wartezeiten bei Erzeuger oder Verbraucher, etc. Bis hin zu hinterlegter Kalkulation, das den Endpreis des Produktes ermittelt und somit steuernd auf die Nachfrage einwirkt. Hierdurch könnten Aussagen zu Stoßzeiten (Gastronomie, Einzelhandel, Energie, Webserver, ...) getroffen werden.

Für eine solche feingliedrige Simulation ist eine eigene Implementierung der Queue oder evt. sogar des Threading denkbar, um an beliebigen Stellen den Zustand des Systems abfegen und analysieren zu können.

A Quelltext der Anwendung

B Ergebnisse verschiedenener Programmläufe

Literatur– und Quellenverzeichnis

Hoffmann, Simon & Lienhart, Rainer (2008): *OpenMP: Eine Einführung in die parallele Programmierung mit C/C++*. Springer-Verlag.

Oracle (Hrsg.) (1993): *Java Documentation: Class LinkedBlockingQueue<E>*. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html`, abgerufen am 14.02.2015.

Sedgewick, Robert & Wayne, Kevin (2011): *Algorithms*. 4th edition, video enhanced Aufl. Addison-Wesley Professional.

Silberschatz, Abraham, Galvin, Peter B. & Gagne, Greg (2012): *Operating System Concepts, 9th Edition*. E–Book. John Wiley & Sons.

Tanenbaum, Andrew S. & Woodhull, Albert S. (2005): *Operating Systems Design and Implementation*. 3rd Aufl. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Eidesstattliche Erklärung

Ich versichere, dass ich das beiliegende Assignment selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet habe.

(Datum, Ort) (Unterschrift)