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

19. Februar 2015

AKAD University

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

FIFO	First In, First Out
LIFO	Last In, First Out
Mutex	Mutually Exclusive Lock
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

Die zur Implementierung der Threadsynchronisation genutzten Konstrukte Mutually Exclusive Lock (Mutex) und Semaphore sowie die als Warteschlange eingesetzte LinkedBlockingQueue werden nicht grundlegend erklärt sondern nur in dem Ausmaß Beschrieben, wie es für diese Arbeit notwendig ist. Ausführlichere Erklärungen finden sich in den angegebenen Quellen.

Die in Abschnitt *2.4 Freiwilliges und unfreiwilliges Warten* dargestellte getrennte Erfassung von freiwilligen und unfreiwilligen Wartezeiten wird in dieser Arbeit nicht durchgeführt, da die Bewertung dieser unterschiedlichen Zeiten stark vom konkreten Einsatzgebiet abhängt.

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. Jeder der Akteure handelt hierbei unabhängig von den anderen und quasi gleichzeitig.[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

[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

zuerst entnommen, die als letztes hinzugefügt wurden. Dieses Verfahren kommt bei der Implementierung mithilfe eines Kellerspeichers (engl. Stack) zum Tragen. 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 (engl. Queue) genutzt.[7]

2.3 Kritische Abschnitte

Damit Verbraucher nicht auf eine leere Queue zugreift, muss zunächst geprüft werden, ob ein Element zur Verfügung steht. Da Prüfung und Zugriff jedoch getrennte Operationen darstellen, besteht die Möglichkeit, dass ein weiterer Verbraucher das letzte Element zwischen diesen beiden Vorgängen entnimmt.[8] Um solche konkurrierende Zugriffe zu Verhindern, muss verhindert werden, dass sich mehrere Verbraucher gleichzeitig in diesem kritischen Abschnitt befinden.

Eine solche Zugangskontrolle lässt im sich Programm mithilfe von Mutex' bzw. Semaphoren realisieren. Beiden Konzepten ist gemein, dass ein Thread vor dem Eintritt in einen kritischen Abschnitt dessen Verfügbarkeit prüft und gegebenenfalls solange wartet, bis diese eintritt. Anschließend wird vermerkt, dass sich ein Thread im kritischen Abschnitt befindet. Beim Verlassen wird diese Markierung wieder entfernt. Bei einem Mutex wird die Zugangskontrolle über einen Wahrheitswert geregelt, weshalb hier maximal ein Thread den kritischen Bereich betreten kann.[9] Bei Semaphoren kommt ein Zähler zum Einsatz, über den die Anzahl der gleichzeitig erlaubten Threads gesteuert werden kann.[10] Hierbei ist sicher zu stellen, dass zwischen der Verfügbarkeitsprüfung und der Sperre für andere Threads kein

[6] vgl. Sedgewick & Wayne (2011), Abschnitt 1.3.3.8 und 1.3.3.9
[7] vgl. Oracle (Hrsg.) (1993b)
[8] Bei mehreren Erzeugern tritt das Problem analog bei fast voller Warteschlange auf.
[9] vgl. Silberschatz et al. (2012), Abschnitt 5.5
[10] vgl. Silberschatz et al. (2012), Abschnitt 5.6

anderer Thread diesen kritischen Abschnitt betreten kann.[11] stattfindet. Code, der kritische Abschnitte korrekt behandelt wird auch als threadsicher bezeichnet.

Hierbei ist zu beachten, dass sich Verbrauchertheads und Erzeugerthreads nicht gegenseitig blockieren. Bei FIFO Warteschlangen dürfen diese Operationen auch quasi gleichzeitig ablaufen. Hier können Verbraucher das älteste Element der Warteschlange entnehmen, während ein Erzeuger ein neues Element hinzufügt. Es bestehen somit zwei getrennte Kritische Abschnitte[12], die unabhängig voneinander überwacht werden können.

Im Falle einer Realisierung mit Hilfe einer LIFO Warteschlange besteht ein gemeinsamer kritischer Abschnitt. Erzeuger und Verbraucher können hier nicht gleichzeitig auf die Warteschlange zugreifen. Hierbei ist sicher zu stellen, dass die Akteure bei leerer bzw. voller Warteschlange den kritischen Abschnitt wieder verlassen, um den Programmfluss nicht dauerhaft in einem sogenannten „Deadlock“ zu blockieren.[13]

2.4 Freiwilliges und unfreiwilliges Warten

Wie im Abschnitt *4.1 Klasse: Akteur* noch näher beschrieben wird, besteht sowohl der Vorgang des Erzeugens als auch der des Verbrauchens im wesentlichen aus drei Phasen: Die Zeit, die für die Herstellung bzw. für den Konsum eines Elements benötigt wird[14], welche von außen als freiwillige Wartezeit wahrgenommen wird; die Wartezeit, die gegebenenfalls beim Warten auf den kritischen Abschnitt anfällt und somit als unfreiwillig anzusehen ist, sowie dem eigentlichen Zugriff auf die Warteschlange. Bei Untersuchungen konkreter Erzeuger–Verbraucher–Systeme können diese Zeiten entsprechend gemessen und unterschiedlich bewertet werden.

[11] Diese Operation muss nicht atomar im strengen Sinne sein. Kontextwechsel sind durchaus zu erlauben, jedoch nicht zu Threads, die um die betreffende Resource konkurrieren.
[12] „Prüfung ob Element vorhanden und Entnahme“ sowie „Prüfung ob Platz vorhanden und Einstellen”
[13] Ein blockierender Verbraucher bei leerer Queue würde hierbei auch einen Erzeuger blockieren, der das erwartete Element somit nicht liefern kann.
[14] Dies schließt die Phase der „Sättigung“ mit ein.

3 Genutzte Sprachmerkmale von Java

3.1 Collections, generische Klassen, Iteratoren

Die zu erstellende Anwendung benötigt Klassen, in denen mehrere Objekte[15] gespeichert werden können. Die Java–Klassenbibliothek stellt hierzu verschiedene Containerklassen zur Verfügung, welche von der Klasse `Collection` abstammen[16] und Objekte beliebigen Typs aufnehmen.[17]

Um dennoch eine Prüfung der Typen zu ermöglichen, wurde in Java ab der Version 5.0 das Konzept der generischen Programmierung (auch „parametrisierte Typen“) eingeführt. Hierbei kann bei der Instanziierung angegeben werden, welcher Klasse die in der Collection zu speichernden Objekte angehören werden. So wird durch die Deklaration `ArrayList <Verbraucher> v = new ArrayList<Verbraucher>();` ein Behälter vom Typ `ArrayList` erzeugt, der Elemente vom Typ `Verbraucher` speichert.[18]

Iteratoren stellen eine Möglichkeit dar, auf die in einer Collection gespeicherten Elemente zuzugreifen. Hierbei ist keine Kenntnis der tatsächlichen Stelle, an der das jeweilige Element gespeichert ist notwendig. Sind alle Elemente abgearbeitet, liefert die Methode `hasNext()` als Ergebnis `FALSE`.[19] In der zu erstellenden Anwendung wird ein Iterator implizit in Form einer erweiterten `for`–Schleife genutzt.

3.2 Threads

Die im Abschnitt *2.1 Erzeuger–Verbraucher—Muster* beschriebene Unabhängigkeit und Gleichzeitigkeit der Akteure lässt sich in Java mit Hilfe von Threads implementieren. Hierbei handelt es sich um Anweisungsstränge die innerhalb

[15] Hierbei handelt es sich um die Verbraucherthread–Objekte sowie um die Elemente der Warteschlange
[16] Ausnahme: `java.util.Map`, vgl. Heinisch et al. (2011), Seite 688
[17] vgl. Oracle (Hrsg.) (1993a)
[18] vgl. Heinisch et al. (2011), Seite 622ff
[19] vgl. Heinisch et al. (2011), Seite 692

eines Prozesses nebenläufig abgearbeitet werden.[20] Hierzu ist eine Ableitung der Klasse `Thread` zu erstellen, welche die Methode `run()` implementiert. Bei der Instanziierung der Objekte werden dann die entsprechenden Threads erzeugt, welche anschließend durch den Aufruf der Methode `start()` gestartet werden.[21]

3.3 Threadsicherheit

Die Java–Klassenbibliothek bietet keine Semaphoren zur Sychronisierung von Threads an. Ein entsprechender Mechanismus müsste somit in einer eigenen Klasse implementiert werden. Hierbei ist besonders zu beachten, dass wie in Abschnitt *2.3 Kritische Abschnitte* beschrieben die Verfügbarkeitsprüfung und die Sperre der anderen Threads in einer quasi–atomaren Operation erfolgt. Dies kann mit Hilfe des Schlüsselwortes `synchronized` sichergestellt werden, welches eine Mutex um den damit markierten Block bzw. um die damit ausgezeichnete Methode legt.[22] Alternativ können über die Klasse `ReentrantLock` von Codeblöcken unabhängige Sperren angefordert werden.[23] Die Überprüfung der Queue auf vorhandene Elemente bzw. Platz zum Einfügen neuer Elemente muss hierbei genau so wie der eigentliche Zugriff sind hierbei ebenfalls zu kodieren.

Die generische Klasse `LinkedBlockingQueue<E>` implementiert alle im vorhergehenden Absatz genannten Aufgaben. Hierbei handelt es sich um eine FIFO–Queue, die threadsicheren Zugriff auf die Elemente ermöglicht. Beim Hinzufügen durch die Methode `put()` wird gegebenenfalls gewartet, bis Platz vorhanden ist, beim Entnehmen durch `take` wird gegebenenfalls gewartet, bis die Queue nicht mehr leer ist.[24] Intern verwendet die Klasse `ReentrantLock` zur Vermeidung von konkurrierenden Zugriffen[25]

[20] vgl. Heinisch et al. (2011), Seite 748f
[21] vgl. Heinisch et al. (2011), Seite 752f
[22] vgl. Heinisch et al. (2011), Seite 767
[23] vgl. Oracle (Hrsg.) (1993c)
[24] vgl. Oracle (Hrsg.) (1993b)
[25] vgl. Lea, Zeile 377–402

3.4 Zufällige Wartezeiten

Pseudo–Zufallszahlen werden in Java mithilfe von Objekten der Klasse Random generiert. Die Methode `nextInt(n)` liefert eine zufällige Ganzzahl r für die gilt $0 \leq r < n$. Durch die Addition von 1 kann hierdurch eine Wartezeit von mindestens einer Zeiteinheit erzeugt werden.

Gewartet wird innerhalb eines Thread mit der Methode `sleep()`, welche Wartezeit in Millisekunden als Argument erwartet. Daher kann die Zeiteinheit sehr klein gewählt werden, was einen schnellen Lauf der Simulation ermöglicht. Oder durch einen entsprechenden Faktor kann die Zeiteinheit größer gewählt werden, damit der Verlauf der Simulation besser live beobachtet werden kann.

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

Heinisch, Cornelia, Müller-Hoffmann, Frank & Goll, Joachim (2011): *Java als erste Programmiersprache*. 6. Aufl. Vieweg + Teubner Verlag, Wiesbaden.

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

Lea, Doug (): *OpenJDK Sourcecode: java.util.concurrent.LinkedBlockingQueue*. `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/LinkedBlockingQueue.java#LinkedBlockingQueue.take%28%29`, abgerufen am 18.02.2015.

Oracle (Hrsg.) (1993a): *Java Documentation: Class Collection<E>*. `http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html`, abgerufen am 18.02.2015.

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

Oracle (Hrsg.) (1993c): *Java Documentation: Class ReentrantLock*. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html`, abgerufen am 18.02.2015.

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)