Fortgeschrittene Programmiertechniken

G. Köhler

Version:
Erzeugung: 17. Juni 2025
RCSfile: skript.tex,v
Revision: 1.18

Date: 2025/06/17 18:55:52

Kapitel 1

Literatur

Literatur mit verschiedenen Schwerpunkten:

1. Rainer Oechsle: *Parallele und verteilte Anwendungen in Java*, 3. Auflage, Hanser Verlag (2011)

   Parallelität + Java.

2. Dietrich Boles: *Parallele Programmierung spielend gelernt mit dem Java-Hamster-Modell - Programmierung mit Java-Threads*, Vieweg+Teubner-Verlag (2008), ISBN 978-3-8351-0229-3

   Online verfügbar unter:
   `http://www.java-hamster-modell.de/eBooks/hamster3.pdf`

3. Danny Poo, Derek Kiong, Swarnalatha Ashok: *Object–Oriented Programming and Java*, 2nd ed., Springer (2008)

   Einführungsbuch in Java. Die Grundlegenden Konzepte wie Threads werden vorgestellt.

4. Peter Pacheco: *An Introduction to Parallel Programming*, Elsevier (2011), ISBN: 978-0-12-374260-5

   Paralleles Programmieren mit MPI, Pthreads und OpenMP

5. W. Richard Stevens: *UNIX Network Programming*, Prentice Hall (1990) bzw. neuere Auflagen.

   Sehr bekanntes Buch im UNIX–Feld, das auch Synchronisationsmechanismen und Interprozesskommunikation erklärt.

6. *http://java.sun.com/docs/books/tutorial/essential/concurrency/*

   Online–Tutorial zu Parallelität + Java.

Kapitel 2

Programmier–Vorschläge

Aufgabe 1: Schreiben Sie ein Java–Programm, das fünf Threads erzeugt und startet.

Der **Haupt–Thread** gibt zu jedem Thread folgende Infos aus: seinen Namen (`getName()`) und seine ID (`getId()`). Außerdem seinen Zustand *lebendig* oder *tot*, der mit `isAlive()` abgefragt werden kann.

Diese Ausgaben sollen jeweils *vor* und *nach* dem Aufruf der `start()`–Methode erzeugt werden.

Außerdem wird mittels `sleep()` unmittelbar nach dem Aufruf von `start()` eine künstliche Verzögerung eingeführt. Variieren Sie diese Zeit!

Jeder der erzeugten **Threads** gibt seinen Namen aus und verfügt ebenfalls über eine künstliche Verzögerung. Experimentieren Sie auch mit dieser Zeit.

Wie verhält sich Ihr Programm in Abhängigkeit von den Verzögerungszeiten?

Aufgabe 2: Schreiben Sie eine Java–Klasse, die einen Thread erzeugt und startet. Sie soll außerdem mindestens folgende Elemente besitzen:

- *Eine* auf 0 initialisierte Zählvariable `int val`.
- Eine Methode `increment()`, welche den Wert von `val` um Eins erhöht.
- Eine Methode `sum_up()`, welche den Wert von `val` mit Hilfe von `increment()` insgesamt $N = 2\,000\,000$–mal um Eins erhöht.

Sowohl der Haupt–Thread als auch der erzeugte Thread rufen `sum_up()` auf, und zwar:

1. Erst ruft der Haupt–Thread `sum_up()`, dann startet er den Thread.
2. Umgekehrte Reihenfolge (probieren Sie beide aus!).

Daran anschließend folgt eine Wartezeit mittels `sleep()`, die Sie wieder variieren können.

Zum Abschluß wird der Wert von `val` ausgegeben. Können Sie alle Ausgaben erklären?

Aufgabe 3: Unter *moodle* finden Sie das Beispiel `Bsp_01.java`, das zwei Threads mittels einer Warteschleife und eines Flags zu synchronisieren versucht.

Versuchen Sie, das Programm zu verstehen und in Ihrer Java–Umgebung ablaufen zu lassen. Geht's? Wenn ja: Pech gehabt, denn Ihr Test hat einen Fehler in der Implementierung übersehen! Geht's nicht: Woran könnte das liegen?

Aufgabe 4: Nochmal Aufgabe 2, diese Variante:

Schreiben Sie eine Java–Klasse, die einen Thread erzeugt und startet. Sie soll außerdem mindestens folgende Elemente besitzen:

- *Eine* auf 0 initialisierte Zählvariable `int val`.
- Eine Methode `increment()`, welche den Wert von `val` um Eins erhöht.
- Eine Methode `sum_up()`, welche den Wert von `val` mit Hilfe von `increment()` insgesamt $N = 2\,000\,000$–mal um Eins erhöht.

Sowohl der Haupt–Thread als auch der erzeugte Thread rufen `sum_up()` auf, und nun zwar genau in dieser Reihenfolge:

- Der Haupt–Thread startet den anderen Thread, dann ruft er `sum_up()`.
- Das Ende der beiden Threads mittels `join()` synchronisieren.
- Zum Abschluß wird der Wert von `val` ausgegeben.

Wenn Sie die Ausgabe erklären können, haben Sie etwas Wesentliches über mögliche Probleme bei nebenläufigen Programmen verstanden!

Aufgabe 5: N Mechaniker sitzen um einen runden Tisch herum. Jeweils zwischen zweien von Ihnen liegt ein Schraubenschlüssel auf dem Tisch. Jeder von Ihnen repariert einen Motor.

Manchmal (die Zeitpunkte zufällig → z. B. `Math.random` benutzen!) benötigt einer der Mechaniker die beiden Schraubenschlüssel. Er verfährt dann folgendermaßen:

- Er nimmt zuerst den links von ihm liegenden Schraubenschlüssel auf. Wird dieser gerade von seinem links sitzenden Kollegen verwendet, wartet der Mechaniker, bis der Schraubenschlüssel wieder verfügbar ist.
- Dann nimmt er in analoger Weise den rechts von ihm liegenden Schraubenschlüssel (oder wartet auf ihn).
- Er verwendet die beiden Schlüssel eine gewisse Zeit (feste oder zufällige Zeit → rumspielen!). Dann legt er beide wieder neben sich ab und arbeitet ohne sie weiter, bis er wieder beide benötigt.

Realisieren Sie jeden Mechaniker als eigenen Thread. Testen Sie ihr Programm z. B. für $N = 5$ mittels debug–Ausgaben. Anhand dieser soll genau nachvollziehbar sein, wer gerade was tut. Gibt es Schwächen in der Implementierung?

Aufgabe 6: Fügen Sie in ihre Programme `join()`–Aufrufe ein.

Aufgabe 7: Die zwei nächsten Vorschläge für Interessierte: Testen Sie die Aufrufe der `interrupt()`–Familie.

Aufgabe 8: Die beispielsweise in `java.util.Timer` definierten Java–Timer starten nach einer angegebenen Zeit eine Aufgabe in einem übergebenen Thread (wie kann man sich so etwas selber schreiben?).

Die in der Vorlesung besprochenen Timer hingegen informieren einen laufenden Arbeiter–Thread nach einer gewissen Zeit mittels eines Interrupts. Vollziehen Sie nochmals die Funktionsweise nach, z. B.: Klasse `MyTimer` übernimmt im Konstruktor eine Wartezeit und einen Arbeiter–Thread, der ausgeführt wird und dem nach der Wartezeit Beendigung signalisiert wird ...

Aufgabe 9: Arbeiten Sie `volatile` in Ihre Programme ein, falls nötig.

Aufgabe 10:

- Gelingt es Ihnen, mittels *busy loops* und *flags* eine Lösung zu finden, die die beiden Threads in Aufgabe 4 passend synchronisiert? Dabei soll die *critical section* der Erhöhung von `val` um Eins geschützt werden.
- Vergleichen Sie Ihre Lösung mit der Methode von GVKVIJFE oder UVBBVI[1]. Versuchen Sie, die genannten Algorithmen zu verstehen.
- Implementierung und Testen der Algorithmen in Java.
- Implementieren Sie den GVKVIJFE in C/C++ und testen Sie ihn an der Aufgabe 4. Was geschieht? Falls Sie nicht das erwartete Ergebnis erhalten: Woran könnte das liegen?
- `atomic`–Variablen mal anschauen. Wie ginge es damit?

Aufgabe 11: Konzipieren Sie ein Programm, das ein Parkhaus simuliert:

- Das Parkhaus hat nur eine maximale Anzahl an Stellplätzen.
- Ein Auto, das in ein volles Parkhaus einfahren will, muß solange warten, bis wieder ein Platz frei geworden ist.
- Ein Platz wird frei, indem ein Auto das Parkhaus verläßt.
- Jedes Auto ist ein eigener Thread.

[1] Caesar–verschlüsselt, damit Sie nicht sofort nachschauen :)

- Ein passendes Test–Programm läßt jedes Auto eine zufällige Zeit umherfahren, bevor es ins Parkhaus will. Steht es im Parkhaus, dann ebenso für einen zufälligen Zeitraum.

Aufgabe 12: In Java existiert mit `java.util.concurrent.Semaphore` eine Implementierung eines zählenden Semaphors: bitte mal anschauen! Eine entsprechende Mutex–Implementierung fehlt, ist aber in Java quasi „eingebaut“ (s. Vorlesung).

Aufgabe 13: Um in Aufgabe 4 die problematische Erhöhung der Zählvariablen `val` korrekt durchzuführen, würde ein einzelner Mutex genügen. Aber mit Semaphoren geht's ebenfalls einfach. Nämlich wie?

Aufgabe 14: Programmieren Sie eine einfache Variante des *Erzeuger–Verbraucher*–Modells:

- Ein Thread (*Schreiber* oder *Erzeuger*) produziert Daten und schreibt sie in einen gemeinsam genutzten Puffer.
- Ein zweiter Thread (*Leser* oder *Verbraucher*) entnimmt dem Puffer ein Zeichen und verarbeitet es weiter.
- Ist der Puffer voll, so muß der Erzeuger warten, bis der Verbraucher ein Zeichen entnommen hat und somit wieder Platz ist.
- Ist der Puffer leer, so muß der Verbraucher warten, bis der Erzeuger ein neuerliches Zeichen in den Puffer geschrieben hat.
- Implementieren Sie die einfachste Variante, nämlich daß der Puffer lediglich Platz für ein einziges Zeichen bietet.
- Was müßte geändert werden, wenn der Puffer Platz für mehr als ein Zeichen hat?

Aufgabe 15: Implementieren Sie die in der Vorlesung nur angerissene „unfaire“ Lösung für die Parkhaussimulation aus Aufgabe 11 nochmal korrekt und vollständig. Benutzen Sie Semaphore. Was fehlt für eine „faire“ Lösung?

Aufgabe 16: Überlegen Sie bei den vergangenen und bei zukünftigen Aufgaben, ob und wie sie wahlweise mittels `synchronized` bzw. mit Semaphoren realisiert werden können.

Aufgabe 17: Reicht `synchronized` bereits aus, um damit die Klasse `Semaphore` zu implementieren?

Aufgabe 18: Benutzen Sie Semaphore, um eine *Barriere* zu implementieren: Das ist eine Stelle im Code, die alle N Threads eines Programms durchlaufen müssen. Dabei blockieren die ersten $N-1$ Threads solange, bis auch der letzte Thread die Stelle erreicht hat. Erst dann laufen alle weiter.

Aufgabe 19: Versuchen Sie nach und nach, den *merge–sort*–Algorithmus zu parallelisieren. Kontrollieren Sie auch, ob sich die Parallelisierung von der Laufzeit her lohnt, verglichen mit einer Ein–Thread–Implementierung.

Aufgabe 20: Mit den Semaphoren ist es nun möglich, die Aufgabe 5 der fünf Mechaniker korrekt zu lösen. Tun Sie es. Welche Daten müssen durch die Semaphore geschützt werden? Würden Mutexe ausreichen? Falls ja, tun Sie es so.

Aufgabe 21: Programmieren Sie eine Lösung für das *8–Damen–Problem* (oder gleich allgemein für beliebige n Damen)

- Zuerst seriell mit einem Thread.
- Überlegen Sie auch Optimierungsmöglichkeiten (z. B. Position der ersten Dame).
- Wie parallelisieren Sie Ihre Lösung?
- Testen Sie das Laufzeitverhalten Ihrer Lösungen.

Aufgabe 22: Können Sie bereits ein „unfaires“ Parkhaus mittels `wait()` und `notify()` implementieren? Mit anderen Worten: Benutzen Sie *condition variables*, um eine `Semaphore`–Klasse zu implementieren.

Aufgabe 23: Das Erzeuger–Verbraucher–Problem (Aufgabe 14) mit `wait()` und `notify()`.

Aufgabe 24: Implementieren Sie die *Barriere* aus Aufgabe 18, nun aber mit `wait()` und `notifyAll()`.

Aufgabe 25: Realisieren Sie ein “faires“ Parkhaus. Tipp dazu: Platznummern vergeben + `notifyAll()`.

Aufgabe 26: Allgemein: `wait()` etc. benützen, um bisherige Aufgaben zu implementieren.

Aufgabe 27: Programmieren Sie einen *Sudoku*–Löser.

- Zuerst seriell mit einem Thread.
- Wie parallelisieren Sie Ihre Lösung?
- Testen Sie das Laufzeitverhalten Ihrer Lösungen.

Aufgabe 28: Spielen Sie `https://deadlockempire.github.io` durch ;-)