0.1 Parallele Prozesse allgemein

Neben den grundlegenden inhaltlichen Herausforderungen, die das Programmieren mit sich bringt, ist die gleichzeitige Verwendung von **Ressourcen** bei der Arbeit im Team noch eine zusätzliche Schwierigkeit. Das beginnt beispielsweise schon damit, dass in der Regel nicht zwei Entwickler an der gleichen Datei arbeiten können, wenn sie sich einfach irgendwo in einem Ordner befindet.
Die Beschäftigung mit solchen Problemen ist dabei so wichtig (und hat auch technische Auswirkungen), dass sich eine eigene Theorie darum aufgebaut hat. Für das Abtiur müssen die Grundlagen dieser Theorie ebenfalls verstanden werden. Für das konkrete Projekt finden sich Anmerkungen - z.B. zur Versionskontrolle und der effektiven Zusammenarbeit mit verschiedenen Tools - wie immer im Kapitel „Das Projekt“. Hier soll es um die theoretischen Hintergründe und Fachbegriffe gehen.
Als Ausgangslage stellen wir uns zwei Prozesse vor, die in irgendeiner Form in **Konkurrenz** zueinander stehen. Beispielsweise zwei Programme, die beide auf bestimmte Daten im Arbeitsspeicher zugreifen wollen.
Die Probleme liegen hier auf der Hand, wird der Bereich des Arbeitsspeichers von einem Program bearbeitet, während das andere ebenfalls bereits darauf zugegriffen hat, so entstehen Inkonsistenzen und Fehler.
Erinnerung: Das Problem ist bereits aus der Mittelstufe von Datenbanken bekannt, dort wurden Anomalien definiert. Im Folgenden ein Auszug aus einem Arbeitsblatt (der Text bezieht sich auf eine Tabelle Bestellung, in der Informationen zu Kunden und gekauften Artikeln vorhanden sind):
„... Es gibt jedoch noch weitere Probleme (auch Anomalien genannt), die durch Redundanzen entstehen, z.B. die UPDATE-Anomalie.
Wird einer oder mehrere Einträge einer Tabelle geändert, so spricht man von einem update (da der zugehörige SQL-Befehl ebenfalls UPDATE heißt!). Offensichtlich ist z.B. eine Namensänderung eines Kunden - beispielsweise durch Heirat - hier sehr problematisch, da jede einzelne seiner Bestellungen geändert werden muss, ansonsten ist die Datenbank inkonsistent. Das bedeutet, dass sie Widersprüche enthält.

Möchte man etwas aus einer Tabelle löschen benutzt man den SQL-Befehl DELETE. Möchte ein Kunde aus dem System gelöscht werden, so muss auch hier jede einzelne seiner Bestellungen gelöscht werden, um seine Daten vollständig zu entfernen. Dabei kann leicht etwas vergessen werden. Zusätzlich besteht die Gefahr auch die Informationen über einen Artikel vollständig zu löschen, wenn die Bestellung die einzige zu diesem Artikel war. Im zweiten Fall spricht man auch von der DELETE-Anomalie.

Angenommen, man möchte einen neuen Artikel zum Sortiment hinzufügen, so ist dies mit obigem Tabellenschema nicht ohne weiteres möglich. Zu jedem Artikel muss eine Bestellung existieren!

Auf den ersten Blick scheint es möglich, bei allen Werten einfach Null, also „nichts“ einzutragen, die bisher nicht bekannt sind. Dies wird jedoch durch den Primärschlüssel verhindert, der hier aus KdNr, ArtNr und BDatum besteht. Es muss also eine fiktive Bestellung zwangsweise angelegt werden, um den Artikel aufnehmen zu können, es kommt zu einer INSERT-Anomalie, also zu einer Anomalie bzw. Inkonsistenz beim Einfügen neuer Daten. ...“

Die Lösung dieser Probleme bestand darin, die Tabellen der Datenbank geschickt zu basteln, sodass diese Anomalien möglichst nicht mehr vorkommen. Dennoch bleiben Probleme, es könnten z.B. zwei Benutzer gleichzeitig versuchen einen Datenbankeintrag zu ändern, das ist die Stelle an der wir jetzt ansetzen - ein gutes Design reicht nicht mehr aus.
Statt miteinander zu konkurrieren, sollen die Programme miteinander **kooperieren**, d.h. ihre Aktionen bewust aufeinander abstimmen - in der Praxis kann dies natürlich nicht in der Interaktion mit jedem anderen Program passieren, deswegen übernimmt die Aufgabe der Koordinatione der Kooperation in der Regel eine übergeordnete Instanz, z.B. das Betriebssystem oder bei den erwähnten Datenbanken das Datenbankmanagementsystem (DBMS).
Zur Lösung des Problems wird das sogenannte **Semaphor-Prinzip** verwendet. Ein **Semaphor** war ursprünglich ein mechanisches Eisenbahnsignal, das anzeigt, ob ein Zug einen Gleisabschnitt belegt. Folgt man dieser Logik, so ist ein Semaphor im Wesentlichen eine Datenstruktur, die verwaltet, ob auf eine bestimmte Ressource gerade zugegriffen wird oder nicht.

Abbildung 1: Quelle: Wikipedia

Im einfachsten Fall besteht sie einfach aus einem Wahrheitswert oder einer Ganzzahl, die den Zustand der Ressource beschreibt, einer Methode „Reservieren()“, die effektiv den Zugriff für alle anderen sperrt und „Freigeben()“, die die entsprechende Ressource wieder entsperrt.
Der Semaphor kann auch erweitert werden, sodass er als Warteschlange fungiert, z.B. könnte man den obigen Zähler erweitern und speichern, wie viele Prozesse gerade „warten“. **Beispielbereiche für die Anwendung des Semaphor-Prinzips**

1. **DBMS**
2. **Druckerwarteschlangen**
3. **Dateizugriff**
4. **Verwaltung von Threads**: Auch bei Software an sich möchte man Parallelsierung nutzen, da es häufig Fälle gibt, bei denen ähnliche Dinge „gleichzeitig“ ablaufen sollen bzw. berechnet werden, dazu werden **Threads** verwendet.

Der letzte Punkt bringt uns direkt zum nächsten Thema:

0.2 Parallele Prozesse in Java

In den letzten zwei Jahrzehnten sind die Anforderungen an Prozessoren immens gewachsen, da immer größere Probleme mit Hilfe von Computern gelöst werden sollen, einige Beispiele:

1. Klimasimulationen
2. Strömungsmechanik
3. Modellierung allgemein
4. Animierte Filme
5. Computerspiele
6. Web-Suche (das Internet ist riesig! :)

Obwohl der Film schon etwas älter ist, sind die Zahlen bei Baymax beeindruckend:

Quelle: disney.fandom

Der Film wurde auf einem Cluster von 4.600 Rechnern mit insgesamt 55.000 Kernen erstellt mit einer kumulierten Rechenzeit von 200 Millionen Stunden. Das entspricht einer Rechenzeit von etwa 83 Stunden pro Frame!
Ohne Parallelisierung - d.h. ein einziger Kern rendert den kompletten Film - wäre solch ein Mammutprojekt nicht denkbar.
Hinweis: Tatsächlich wäre eine reine Parallelisierung immer noch nicht ausreichend, um die gigantischen Anforderungen des Rendering abzufangen, die größten Verbesserungen kommen durch die Verwendung intelligenter Algorithmen, die Parallelisierung schadet aber natürlich nicht.

An dieser Stelle könnte man einen langen Exkurs einfügen, wie wichtig Parallelität in den meisten Bereichen der Technik inzwischen ist, dafür haben wir hier aber keine Zeit.

Umsetzung in Java
Die Umsetzung in Java wird uns noch in der zwölften Klasse ausführlich beschäftigen, hier ein kleiner Vorgeschmack:

```
public class Threading{
    public static void main(String[] args) {
        // Wir erzeugen die Threads und geben ihnen eine Aufgabe
        Thread t1 = new Thread(new Task());
        Thread t2 = new Thread(new Task());

        // Wir starten die Threads.
        t1.start();
        t2.start();

        // Wir warten darauf, dass beide Threads fertig sind.
        try {
            t1.join();
            t2.joint();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Task finished");
    }
    }

    class Task implements Runnable {
    @Override
```

```
    public void run() {
        // Irgendeine Aufgabe wird ausgeführt.
        System.out.println("Task executed by " + Thread.currentThread().getName());
    }
}
```

Die Verwendung von Threads ist in Java grundsätzlich nicht schwer, da bereits eine vorgebaute Klasse **Thread** existiert, die dem entsprechenden Betriebssystem mitteilt, dass möglichst ein zweiter Kern genutzt werden soll. Natürlich ergibt sich hier aber ein analoges Problem zum vorigen Kapitel, wenn die Aufgabe, die jeder Thread ausführt auf gemeinsame Ressourcen zugreifen muss, so kann es zu den bereits besprochenen Problemen kommen. Wie Java diese Probleme löst ist Stoff der 12. Klasse.