Fernstudium
INFORMATIK
im Hochschulverbund

Trier University
of Applied Sciences

HOCHSCHULE TRIER

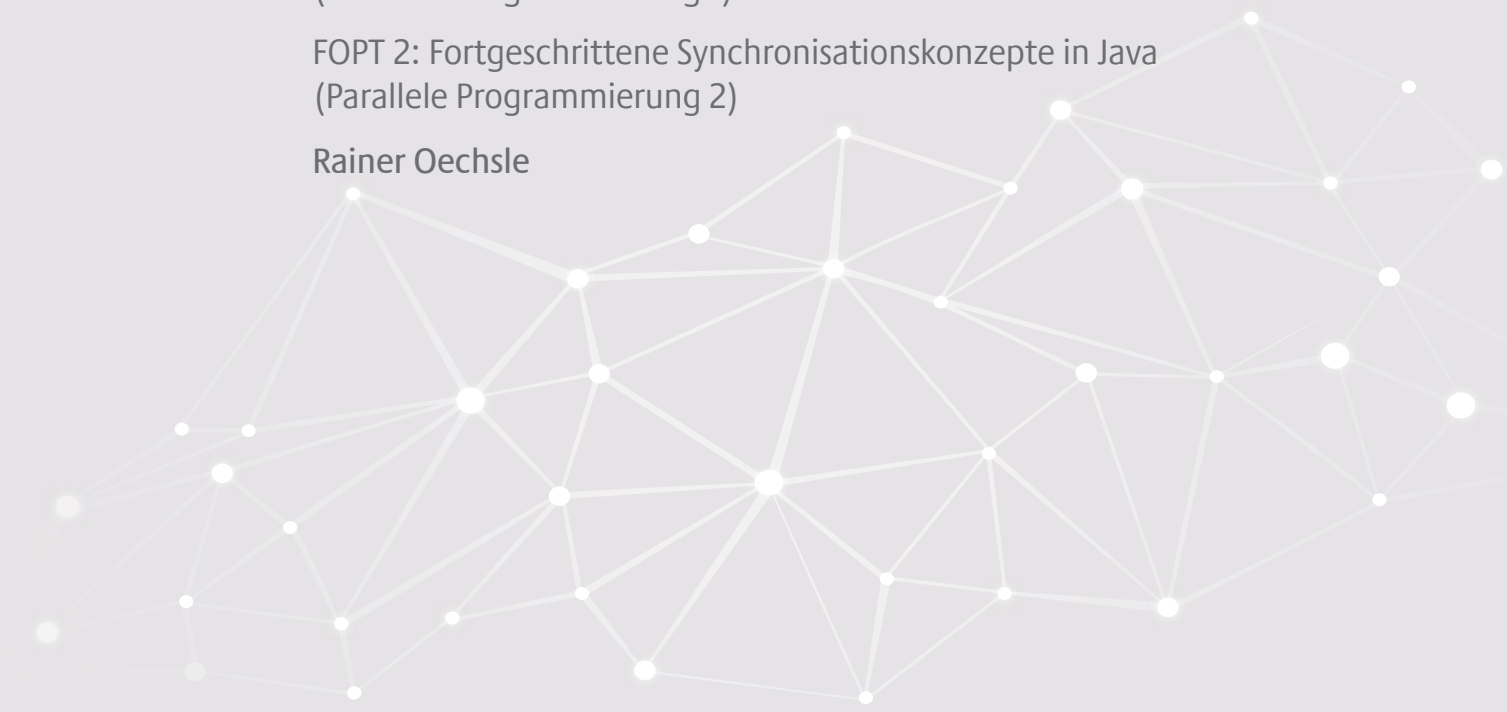
FOPT 1/2

Fortgeschrittene Programmiertechniken

FOPT 1: Grundlegende Synchronisationskonzepte in Java
(Parallele Programmierung 1)

FOPT 2: Fortgeschrittene Synchronisationskonzepte in Java
(Parallele Programmierung 2)

Rainer Oechsle



M.C.Sc. / Zertifikat

Fortgeschrittene Programmiertechniken

Herausgeber:
Prof. Dr. Rainer Oechsle
Hochschule Trier

Grundlegende Synchronisationskonzepte in Java
Fortgeschrittene Synchronisationskonzepte in Java

Autor:
Prof. Dr. Rainer Oechsle
Hochschule Trier, Fachbereich Informatik

6. Auflage

Herausgeber: Fernstudium Informatik

Anschrift: Hochschule Trier
Fachbereich Informatik • Fernstudium Informatik
Postfach 18 26 • 54208 Trier
Telefon: (06 51) 81 03-77 0

Vertrieb: zfh – Zentrum für Fernstudien im Hochschulverbund

Anschrift: Konrad-Zuse-Str. 1 • 56075 Koblenz
Telefon: (02 61) 9 15 38-0

D1.22

Einleitung

Dieses Modul mit dem Namen „Fortgeschrittene Programmiertechniken“ vertieft und erweitert objektorientierte Programmierkenntnisse in Java vor allem in drei Bereichen: parallele Programmierung mit Java Threads, grafische Benutzeroberflächen mit JavaFX und verteilte Anwendungen mit Sockets, RMI, Servlets und Java Server Faces. Alle drei Bereiche stellen wesentliche und zentrale Lehrinhalte im Rahmen einer Informatik-Ausbildung dar und haben eine hohe Relevanz für die berufliche Praxis.

Im ersten Teil über parallele Programmierung lernen Sie, welche grundsätzlichen Probleme der Einsatz von Parallelität birgt und wie diese Probleme beherrscht werden können. Das Thema gewinnt für die berufliche Praxis durch die zunehmende Verbreitung von Multi-Core-Prozessoren stark an Relevanz. Im zweiten Teil über die Programmierung grafischer Benutzeroberflächen lernen Sie nicht nur, wie man Programme mit einer grafischen Benutzeroberfläche entwickelt, sondern vor allem auch, wie Architektur- bzw. Entwurfsmuster zielgerichtet und Gewinn bringend eingesetzt werden können. Sie werden dadurch vom „Programmieren im Kleinen“ zum „Programmieren im etwas Größeren“ geführt und lernen insbesondere, wie man Programme strukturiert, was in der Praxis von großer Bedeutung ist. Im dritten Teil geht es um die Entwicklung verteilter Anwendungen, und zwar zum einen um eigenständige Anwendungen, die über Sockets oder Java RMI kommunizieren, und zum anderen um webbasierte Anwendungen, die auf Servlets und Java Server Faces basieren. Im Zeitalter des Internets sind die meisten Anwendungen heute in der Regel verteilt; die praktische Relevanz dieses Teils muss daher nicht weiter betont werden.

Die drei Themengebiete sind aber nicht nur isoliert jeweils für sich genommen bedeutsam, sie hängen in vielfältiger Weise miteinander zusammen: Bei der Entwicklung von Programmen mit grafischer Benutzeroberfläche ist der Einsatz von Parallelität notwendig, falls von den Benutzerinnen und Benutzern des Programms länger andauernde Aktionen angestoßen werden. Bei verteilten Anwendungen besitzt die Client-Seite in der Regel eine grafische Benutzeroberfläche, während auf der Server-Seite Parallelität im Spiel ist. Außerdem kann das Entwurfsmuster MVP, das typischerweise in Anwendungen mit einer grafischen Benutzeroberfläche Verwendung findet, auch sehr gut zur Strukturierung verteilter Anwendungen herangezogen werden.

Die Kurseinheiten des Moduls „Fortgeschrittene Programmiertechniken“ sind als Begleittexte zum Lehrbuch „Parallele und verteilte Anwendungen in Java“ konzipiert. FOPT 1 / FOPT 2 und FOPT 5 / FOPT 6 leiten dabei lediglich zum Studium der relevanten Lehrbuch-Kapitel an. Die beiden Kurseinheiten FOPT 3 und FOPT 4 liefern zusätzlich wichtige Grundlagen, die über den Stoff des Lehrbuchs deutlich hinausgehen.

In der ersten Kurseinheit „Grundlegende Synchronisationskonzepte in Java“ sollen Sie folgende Lernziele erreichen:

- Sie können in einem Programm mehrere parallele Threads erzeugen und starten. Sie wissen auch, wie Sie diese Threads auf gemeinsamen Daten arbeiten lassen können.
- Sie kennen die Probleme, die es beim parallelen Zugriff auf gemeinsame Daten gibt, können Beispiele hierfür angeben und können in konkreten Fällen entscheiden, ob eine Synchronisation nötig ist und wie diese durchzuführen ist.
- Sie verstehen die Problematik, die es beim Abbruch von Threads gibt, und Sie sind in der Lage, Threads so zu programmieren, dass sie „sanft“ abgebrochen werden können.
- Sie kennen die Wirkung der Java-Synchronisationsprimitive wait, notify und notifyAll genau. Außerdem sind Sie in der Lage, für verschiedene Situationen Wartebedingungen und deren Auflösung mittels dieser Synchronisationsprimitive zu programmieren.

In der zweiten Kurseinheit „Fortgeschrittene Synchronisationskonzepte in Java“ lernen Sie vor allem, die Grundlagen, die Sie in der ersten Kurseinheit kennen gelernt haben, auf unterschiedliche Situationen anzuwenden. Im Einzelnen werden die folgenden Lernziele verfolgt:

- Sie verstehen die Wirkungsweise des „klassischen“ Synchronisationskonzepts Semaphor und seiner Varianten wie additive Semaphore und Semaphorgruppen. Sie können diese Konzepte selbst implementieren und in konkreten Situationen anwenden.
- Sie wissen, dass die aus UNIX bzw. Linux bekannten Interprozesskommunikationskonzepte Message Queues und Pipes nach dem Erzeuger-Verbraucher-Schema gestaltet sind, und Sie können die wesentlichen Unterschiede zwischen diesen Konzepten erläutern.
- Sie wissen, was man unter dem Philosophen-Problem und dem Leser-Schreiber-Problem versteht, und sind in der Lage, diese Probleme mit unterschiedlichen Synchronisationskonzepten zu lösen.
- Sie haben einen Überblick über die Concurrent-Klassenbibliothek von Java. Sie kennen die Bedeutung und Arbeitsweise von Thread-Pools, Locks und Conditions, den Atomic-Klassen und verschiedenen Synchronisationsklassen wie CountDownLatches, CyclicBarriers sowie Exchangers, und Sie sind in der Lage, diese in Ihren eigenen Anwendungen einzusetzen.
- Schließlich ist Ihnen geläufig, wie Verklemmungen entstehen können, welche Bedingungen für das Entstehen von Verklemmungen erfüllt sein müssen und welche Maßnahmen gegen Verklemmungen getroffen werden können.

Inhaltsverzeichnis

FOPT 1:
Grundlegende Synchronisations-konzepte in Java
(Parallele Programmierung 1)

1 Einleitung

Zunächst geht es darum, dass Sie eine erste Vorstellung von Begriffen wie Parallelität, Nebenläufigkeit, Programm, Prozess und Thread bekommen, die für das Modul „Fortgeschrittene Programmiertechniken“ von zentraler Bedeutung sind.

Lesen Sie dazu das Kapitel 1 des Lehrbuchs.

Zusammenfassung

Das echte oder scheinbar gleichzeitige Ablaufen von Vorgängen auf einem Rechner wird als Parallelität bzw. Nebenläufigkeit bezeichnet. Sind mehrere Rechner im Spiel, so spricht man von Verteilung. Die Unterscheidung zwischen Parallelität bzw. Nebenläufigkeit einerseits und Verteilung andererseits kann in Einzelfällen schwierig sein.
Programme sind Java-Dateien bzw. in übersetzter Form Class-Dateien. Ein Prozess bildet einen eigenen Adressraum, der von den Adressräumen anderer Prozesse isoliert ist. Threads sind Aktivitätsträger. Jeder Prozess hat mindestens einen Thread. Umgekehrt ist ein Thread in unveränderbarer Weise nur einem Prozess zugeordnet. Die Threads, die zum selben Prozess gehören, teilen sich den Adressraum und können daher auf gemeinsamen Daten arbeiten.

Übungsaufgaben

1.1 Was ist der Unterschied zwischen Parallelität und Nebenläufigkeit?

1.2 Stellen Sie in einer Tabelle die Informatikbegriffe Prozess, Thread usw. den Entsprechungen aus dem alltäglichen Leben gegenüber!

2 Erzeugung und Start von Java-Threads

Um mit Threads arbeiten zu können, müssen Sie zuerst lernen, wie diese erzeugt und gestartet werden.

Lesen Sie den Beginn des Kapitels 2 sowie Abschnitt 2.1 des Lehrbuchs.

Zusammenfassung

Wenn ein Java-Programm gestartet wird, so führt der Main-Thread die Methode main der im Java-Kommando angegebenen Klasse aus. Der Programmcode weiterer Threads muss sich in parameterlosen Methoden mit dem Namen run und dem Rückgabetyp void befinden. Eine solche Run-Methode muss sich entweder in einer aus der Klasse Thread abgeleiteten Klasse oder in einer Klasse, welche die Schnittstelle Runnable implementiert, befinden. Da Runnable eine funktionale Schnittstelle ist, kann auch ein Lambda-Ausdruck verwendet werden, um die Runnable-Schnittstelle zu implementieren. Mit dem Erzeugen eines Objekts der Klasse, in der sich die Methode run befindet, ist noch kein Thread im Sinne einer selbstständig ablaufenden Aktivität vorhanden. Dies erfolgt erst durch die Anwendung der Methode start auf ein Thread-Objekt. Wenn die Ausführung der Run-Methode zu Ende ist, dann ist auch die Thread-Aktivität zu Ende, auch wenn das entsprechende Objekt noch vorhanden ist.
Threads können im Konstruktor oder durch Aufruf der Methode setName Namen zugewiesen werden, die durch getName wieder gelesen werden können.
Mit der statischen Methode sleep der Klasse Thread kann der Ablauf eines Threads für eine in Millisekunden anzugebende Zeit angehalten werden. Die Methode sleep sollte allerdings nicht zur Steuerung einer bestimmten Reihenfolge der Aktivitäten in unterschiedlichen Threads verwendet werden.

Übungsaufgaben

2.1 Warum gibt es die Möglichkeit, dass sich die Run-Methode auch in einer Klasse befinden kann, die nicht aus Thread abgeleitet ist, sondern die die Schnittstelle Runnable implementiert?

2.2 Ändern Sie das Beispielprogramm aus Listing 2.3 des Lehrbuchs so ab, dass sich die Run-Methode nicht mehr in einer aus Thread abgeleiteten Klasse befindet, sondern in einer Runnable implementierenden Klasse!

2.3 Ändern Sie das Beispielprogramm aus Listing 2.3 des Lehrbuchs so ab, dass Sie Lambda-Ausdrücke für die Runnable-Objekte verwenden!

2.4 In Abschnitt 2.1.4 des Lehrbuchs werden einige Beispiele gegeben für die Anzahl der möglichen Abläufe bei n Threads, wobei jeder Thread a Aktionen ausführt. Mit welcher Formel können diese Zahlen berechnet werden?

3 Probleme beim Zugriff auf gemeinsam genutzte Objekte

Threads, die zu demselben Prozess gehören, arbeiten in einem gemeinsamen Adressraum. Sie können daher auf dieselben Objekte zugreifen. Dies kann allerdings problematisch sein.

Lesen Sie Abschnitt 2.2 des Lehrbuchs.

Zusammenfassung

Beim parallelen Zugriff auf gemeinsame Daten kann es zu Problemen kommen, die beim sequenziellen Zugriff nicht möglich sind. Der erste Versuch, das Problem dadurch zu lösen, dass der Zugriff in einer einzigen Java-Anweisung codiert wird, scheitert, weil eine Java-Anweisung vom Compiler in mehrere Anweisungen auf Byte-Code-Ebene aufgeteilt wird. Außerdem ist der Zugriff in einer einzigen Java-Anweisung im Allgemeinen nicht möglich. Der zweite Versuch, Sperrflags einzuführen, ist ebenfalls nicht von Erfolg gekrönt, da die Lösung ineffizient ist und – schlimmer noch – den ausschließlichen Zugriff durch höchstens einen Thread zu einem Zeitpunkt nicht garantieren kann.

Übungsaufgaben

3.1 In dieser Aufgabe geht es um eine Klasse, die einen Zähler des Typs int als Attribut besitzt. Die Klasse soll die Methoden increment und decrement haben, die den Zähler um eins erhöhen bzw. erniedrigen. Der Zähler soll allerdings nicht kleiner als 0 und nicht größer als 10 werden können. Das heißt: Ist der Zähler 0, soll der Aufruf von decrement ohne Wirkung bleiben. Ist der Zähler 10, soll der Aufruf von increment ohne Wirkung bleiben. Kann bei paralleler Nutzung eines Objekts dieser Klasse garantiert werden, dass der Zähler zwischen 0 und 10 bleibt?

3.2 a) Warum gibt es das Problem der gemeinsam benutzten Daten in den Beispielen des Abschnitts 2.1 des Lehrbuchs (Listing 2.3, 2.4 und 2.5) nicht? Mit anderen Worten: Warum gibt es kein Problem mit der Laufvariablen i?

b) Ändern Sie die Beispielklasse Loop2 aus Listing 2.4 so, dass statt der Ableitung aus Thread den Thread-Objekten ein Runnable-Objekt übergeben wird! Setzen Sie den Namen der Threads beim Erzeugen der Thread-Objekte (d.h. verwenden Sie den Thread-Konstruktor mit den Argumenten des Typs Runnable und String)! Da die Klasse, die die Methode run enthält, jetzt nicht mehr aus Thread abgeleitet ist, ist der Aufruf der Methode getName nicht mehr möglich. Ersetzen Sie den Aufruf von getName durch Thread.currentThread().getName()! Realisieren Sie folgende Varianten:

b1) Jeder Thread soll sein eigenes Runnable-Objekt erhalten.

b2) Alle Threads benutzen dasselbe Runnable-Objekt.

Bei welchen Varianten gibt es ein Problem durch gemeinsam benutzte Daten?

c) Ändern Sie die Original-Beispielklasse Loop2 so, dass i statt einer lokalen Variable nun ein Attribut der Klasse ist! Gibt es ein Problem durch gemeinsam benutzte Daten? Welche Variante (Original im Vergleich zu der so geänderten Version) ist besser bzw. „schöner“?

d) Kombinieren Sie nun die Änderungen aus den Teilaufgaben b und c so, dass Sie wieder mit Runnable arbeiten, zusätzlich aber i statt einer lokalen Variable als Attribut der Runnable implementierenden Klasse definieren! Betrachten Sie wieder die beiden Varianten aus Teilaufgabe b (jeder Thread hat eigenes Runnable-Objekt bzw. alle Threads benutzen dasselbe Runnable-Objekt)! Gibt es ein Problem durch gemeinsam benutzte Daten?

4 Synchronized und volatile

In diesem Abschnitt werden Sie sehen, wie einfach sich das im vorigen Abschnitt dargestellte Problem lösen lässt.

Lesen Sie Abschnitt 2.3 des Lehrbuchs.

Zusammenfassung

Mit synchronized kann ein Objekt in korrekter und effizienter Weise gesperrt werden, so dass bei gleichzeitiger Nutzung eines Objekts durch mehrere Threads garantiert ist, dass ein Objekt immer nur von höchstens einem Thread zu einem Zeitpunkt benutzt wird. Es gibt zwei Varianten von synchronized: sowohl Methoden als auch Anweisungsblöcke können mit synchronized versehen sein. Bei Synchronized-Blöcken muss man das Objekt, das gesperrt werden soll, explizit angeben, während sich bei Synchronized-Methoden, die nicht static sind, die Sperre immer auf this bezieht. Wird eine Static-Methode mit synchronized gekennzeichnet, so wird damit die Klasse gesperrt. Dies hat allerdings keinen Effekt auf die Objekte dieser Klasse. Beim Zugriff auf gemeinsame Daten von mehreren Threads aus, wobei mindestens von einem Thread die Daten verändert werden, müssen alle Zugriffe durch Synchronisationsmechanismen abgesichert werden. Dies gilt nicht nur für die schreibenden, sondern auch für die lesenden Zugriffe. Gerade dieser letzte Punkt wird leicht vergessen und sollte daher besondere Beachtung finden.

Übungsaufgaben

4.1 Demonstrieren Sie die Wirkung von synchronized mit Hilfe eines Beispielprogramms! Gehen Sie dazu wie folgt vor: Schreiben Sie eine Klasse mit einer Methode, die durch Aufruf von Thread.sleep eine längere Ausführungszeit hat! Geben Sie in dieser vor und nach dem Sleep-Aufruf jeweils eine Zeile auf die Console aus! Erzeugen Sie zwei Threads, die auf demselben Objekt der obigen Klasse die Methode, welche den Sleep-Aufruf enthält, aufrufen! Führen Sie das Programm aus, wenn die Methode, welche den Sleep-Aufruf enthält, synchronized ist! Wiederholen Sie die Ausführung ohne synchronized! Welchen

Unterschied können Sie beobachten? Was passiert, wenn die beiden Threads unterschiedliche Objekte benutzen?

4.2 Zeigen Sie, dass synchronized wesentlich effizienter arbeitet als die selbst gebastelte Sperre aus Listing 2.8 des Lehrbuchs (die ja im Übrigen nicht korrekt gearbeitet hat)! Verlängern Sie künstlich zu diesem Zweck im Bankbeispiel den Aufruf von transferMoney durch Aufruf der Methode Thread.sleep! Synchronisieren Sie die Methode transferMoney dann einmal korrekt mit synchronized und einmal mit der selbst gebastelten Sperre aus Listing 2.8! Beobachten Sie mit Hilfe des Task-Managers die CPU-Auslastung Ihres Rechners für beide Fälle!

4.3 Die letzte Variante des Programms von Aufgabe 3.2 (Teilaufgabe d, alle Threads benutzen dasselbe Runnable-Objekt) hat eine ziemlich verwirrende Ausgabe ergeben, bei der nicht alle Zahlen zwischen 1 und 1000 jeweils genau ein einziges Mal zu sehen sind, so dass die Anzahl der Ausgabezeilen auch nicht 1000 sein muss, sondern mehr oder weniger sein kann. Dies liegt daran, dass gemeinsame Daten benutzt wurden ohne Synchronisation. Ergänzen Sie die letzte Variante des Programms mit einer korrekten Synchronisation für die gemeinsam genutzte Variable i der Typs int!

5 Ende von Java-Threads

In diesem Abschnitt geht es darum, wie Threads beendet werden können, wie man das Ende eines Threads erkennt, und wie man auf das Ende eines Threads wartet.

Lesen Sie Abschnitt 2.4 des Lehrbuchs.

Zusammenfassung

Ob ein Thread noch läuft, kann durch Aufruf der Methode isAlive – angewendet auf das entsprechende Thread-Objekt – herausgefunden werden. Allerdings sollte man auf das Ende eines Threads nicht durch wiederholte Ausführung von isAlive warten, sondern mit Hilfe der Methode join. So wird wesentlich weniger Rechenzeit vebraucht.
Wenn man eine rechenintensive Aufgabe auf mehrere Threads aufteilt, erzielt man bei einem Rechner mit nur einem Prozessor keinen Laufzeitgewinn. Eine Aufteilung ist allerdings aus folgenden Gründen in vielen Fällen sinnvoll: Falls dasselbe Programm später auf einer anderen Hardware mit mehreren Prozessoren ausgeführt wird, dann bringt die Parallelisierung eine Reduktion der Laufzeit. Außerdem entstehen in vielen Fällen auch bei einem System mit nur einem Prozessor Vorteile, falls Wartezeiten in den Threads entstehen, denn diese können von anderen Threads ausgenützt werden. Schließlich dient eine Aufteilung einer Aufgabe auf mehrere Threads häufig auch einfach nur der Strukturierung jenseits aller Betrachtungen der Laufzeit.
Soll ein Thread von außen abgebrochen werden können, so sollte diese Möglichkeit im Programmcode des entsprechenden Threads vorgesehen werden. Es sollte zum Abbruch ein entsprechendes Flag verwendet werden, das vom abzubrechenden Thread „in günstigen Augenblicken“ abgefragt wird und das den Thread veranlasst, aus seiner Run-Methode zurückzukehren, falls das Flag gesetzt ist. Als Flag kann man ein eigenes Attribut des Typs boolean verwenden. Alternativ kann man dafür das in den Threads bereits vorhandene Interrupt-Flag einsetzen, das mit der Methode isInterrupted abgefragt und mit der Methode interrupt gesetzt werden kann. Die zweite Variante mit einem Einsatz des Interrupt-Flags hat den Vorteil, dass durch den Aufruf der Methode interrupt blockierende Methodenaufrufe wie sleep und join unterbrochen werden.

Übungsaufgaben

5.1 Probieren Sie aus, was im Lehrbuch über eine Änderung von Listing 2.17 gesagt wird! Das heißt: Schachteln Sie in der Methode run aus Listing 2.17 nicht die While-Schleife in den Try-Block, sondern packen Sie nur den Aufruf von Thread.sleep innerhalb der While-Schleife in einen Try-Catch-Block! Probieren Sie dann aus, dass damit das Programm nicht mehr abgebrochen werden kann!

5.2 Warum muss im Programm aus Listing 2.20 (asynchrone Beauftragung mit Rückruf) synchronized verwendet werden, während dies im Programm aus Listing 2.14 (asynchrone Beauftragung ohne Rückruf) nicht nötig ist?

5.3 Realisieren Sie als Beispiel für die asynchrone Beauftragung mit Rekursion das Zählen der True-Werte in einem großen booleschen Feld! Verwenden Sie dazu die Klasse TaskNodeExecutor!

a) Als Ergebnis sollen nicht nur die True-Werte zurückgegeben werden, sondern auch die Anzahl der Knoten des Baums sowie die Baumtiefe.

b) In einer Variante soll das Ergebnis statt der Anzahl der Knoten des Baums nun lediglich die Anzahl der Blätter enthalten.

6 Wait und notify

Es gibt Situationen, in denen innerhalb einer Synchronized-Methode gewartet werden muss. Wenn dies falsch gemacht wird, führt es zu Problemen. In diesem Abschnitt lernen Sie, wie man richtig innerhalb von synchronized wartet, und wie wartende Threads von anderen Threads aus der Wartesituation befreit werden können.

Lesen Sie Abschnitt 2.5 des Lehrbuchs.

Zusammenfassung

Mit Hilfe der Methode wait kann ein Thread warten, bis er mit notify von einem anderen Thread geweckt wird. Die Methode wait muss auf ein Objekt angewendet werden, das durch synchronized vom wait aufrufenden Thread gesperrt ist. Der Thread wartet an dem Objekt, auf das die Wait-Methode angewendet wird. Wichtig ist, dass dabei die Sperre freigegeben wird. Ein Thread, der auf einem Objekt notify aufruft, muss ebenfalls das betreffende Objekt mit synchronized gesperrt haben. Würde bei wait die Sperre nicht freigegeben, so könnte kein anderer Thread notify aufrufen, da das Objekt immer noch von dem wartenden Thread gesperrt ist. Die Methode notify weckt irgendeinen der an dem Objekt wartenden Threads. Der geweckte Thread kann nicht unmittelbar weiterlaufen, da der notify aufrufende Thread noch die Sperre hält. Erst wenn dieser Thread die Sperre freigibt, kann der geweckte Thread die Sperre setzen. Es muss aber nicht so sein, dass der geweckte Thread als nächster das Objekt sperrt. Genauso gut könnte der Thread, der notify aufgerufen hat und die Sperre freigegeben hat, das Objekt unmittelbar danach erneut sperren. Oder es könnte ein dritter Thread die Sperre auf das Objekt vor dem geweckten Thread setzen.

Übungsaufgaben

6.1 Betrachten Sie nochmals die Klasse ParkingGarage aus Listing 2.26 im Lehrbuch! Begründen Sie, ohne im Lehrbuch zu lesen, warum while in der Methode enter nicht durch if ersetzt werden darf!

6.2 In Abschnitt 2.5.4 des Lehrbuchs wird diskutiert, dass man bei der Betrachtung der Ausgabe des Parkhausprogramms den Eindruck bekommen kann, dass sich mehr Autos im Parkhaus befinden würden als Plätze vorhanden sind. Im Lehrbuch wird ein Versuch angegeben, das Problem mit einem Lock-Objekt zu lösen, der allerdings nicht korrekt ist. Wir variieren die Lösung anschließend in der Weise, dass in dem Synchronized-Block, der das Ein- bzw. Ausfahren mit der Ausgabe klammert, nicht ein Lock-Objekt gesperrt wird, sondern das Parkhaus garage selbst. Das Programm sieht dann so aus:

```
class Car extends Thread
{
    private ParkingGarage garage;

    public Car(String name, ParkingGarage garage)
    {
        ...
    }

    public void run()
    {
        while(true)
        {
            ... //sleep wie zuvor
            synchronized(garage)
            {
                garage.enter();
                System.out.println(getName()
                                   + ": eingefahren");
            }
            ... //sleep wie zuvor
            synchronized(garage)
            {
                garage.leave();
                System.out.println(getName()
                                   + ": ausgefahren");
            }
        }
    }
}
```

Begründen Sie mit eigenen Worten, warum diese Lösung funktioniert!

6.3 Die in der vorigen Aufgabe angegebene Problematik der irreführenden Ausgaben ließe sich auch dadurch lösen, dass man die Ausgaben in die Methoden enter und leave einbaut. Auch diese Lösungsvariante wird im Lehrbuch angesprochen. Schildern Sie in eigenen Worten, welche Nachteile diese Lösung besitzt!

7 NotifyAll

Es gibt Situationen, bei denen das Wecken eines Threads mit notify nicht ausreicht. In diesen Fällen sollte notifyAll verwendet werden.

Lesen Sie Abschnitt 2.6 des Lehrbuchs.

Zusammenfassung

Mit der Methode notifyAll werden alle Threads geweckt, die an dem Objekt, auf das notifyAll angewendet wird, warten. Die Verwendung von notify ist nicht ausreichend, wenn es

1. mehrere Threads gibt, die auf unterschiedliche Bedingungen warten (in diesem Fall könnte mit notify ein „falscher Thread“ geweckt werden, der nicht weiterlaufen kann, während ein anderer Thread, der weiterlaufen könnte, nicht geweckt wurde), oder wenn es
2. mehrere Threads gibt, die weiterlaufen können.

Trifft mindestens eine der Bedingungen zu, so muss notifyAll verwendet werden.

Übungsaufgaben

7.1 Gehen Sie von der korrekten Lösung des Erzeuger-Verbraucher-Problems aus Listing 2.30 des Lehrbuchs aus! Nehmen Sie an, es gäbe wie in Listing 2.29 drei Erzeuger, die jeweils 100 Werte in den Puffer stellen! Anders als in Listing 2.29 gäbe es aber nur einen einzigen Verbraucher, der 300 Werte aus dem Puffer entnimmt und diese ausgibt. Könnte dann der im Lehrbuch beschriebene Effekt immer noch eintreten, dass das Auslesen von 98 vor dem Auslesen von 97 gemeldet wird?

7.2 In Listing 2.31 des Lehrbuchs wird eine Möglichkeit für eine Implementierung eines fairen Parkhauses vorgestellt. Alternativ kann man ein faires Parkhaus auch so implementieren, dass man alle Threads immer in eine Warteschlange einreiht und nur denjenigen Thread in das Parkhaus einfahren lässt, der sich ganz vorne in der Schlange befindet. Implementieren Sie ein faires Parkhaus, das auf dieser Idee basiert! Um in den Parkhaus-Methoden eine Referenz auf den Thread zu bekommen, der gerade läuft, soll die statische Methode currentThread der Klasse

Thread verwendet werden. Sie müssen für die Lösung dieser Aufgabe wissen, wie die Methode verwendet wird und was sie bewirkt. Die Methode wird im Lehrbuch erst in Abschnitt 2.7 vorgestellt. Zusätzlich oder alternativ zum Lehrbuch können Sie auch die Java-Dokumentation nutzen.

8 Verschiedenes

In den folgenden Abschnitten von Kapitel 2 des Lehrbuchs geht es um weitere Aspekte wie Prioritäten von Threads, Thread-Gruppen und Thread-lokale Daten, die deutlich weniger wichtig sind als die bisher behandelten Themen. Am wichtigsten dürfte der Abschnitt über Vordergrund- und Hintergrund-Threads sein.

Lesen Sie den Abschnitt 2.9. Bei Interesse können Sie auch noch die Abschnitte 2.7, 2.8, 2.10 und 2.11 des Lehrbuchs lesen, wobei die dort vermittelten Inhalte nicht Teil des Pflichtlehrstoffs sind. Die Zusammenfassung des Kapitels 2 in Abschnitt 2.12 sollten Sie dann aber wieder zur Wiederholung und Festigung des Lehrstoffs lesen.

FOPT 2:
Fortgeschrittene Synchronisationskonzepte in Java
(Parallele Programmierung 2)

9 Semaphore

Semaphore sind ein „klassisches“ Synchronisationskonzept, das es schon deutlich länger gibt als die Programmiersprache Java. In diesem Abschnitt geht es um verschiedene Varianten von Semaphoren, die in Java programmiert werden.

Lesen Sie den Beginn des Kapitels 3 und Abschnitt 3.1 des Lehrbuchs.

Zusammenfassung

Semaphore dienen zur Synchronisation von parallelen Threads. Mit Hilfe der Methoden p und v (manchmal auch down und up oder acquire und release) kann der Wert eines Zählers um 1 erniedrigt bzw. erhöht werden. Der Wert des Zählers darf allerdings nie negativ werden. Soll der Wert des Zählers erniedrigt werden, wenn er bereits 0 ist, dann wird der aufrufende Thread so lange blockiert, bis der Zähler größer als 0 ist, um ihn dann zu erniedrigen. Semaphore dienen zum gegenseitigen Ausschluss oder zur Herstellung einer vorgegebenen Ausführungsreihenfolge. Eine Variante von Semaphoren sind additive Semaphore, bei denen der Wert des Zählers „auf einen Schlag“ um mehr als 1 erniedrigt und erhöht werden kann. Eine weitere Variante sind Semaphorgruppen, bei denen der Zählerwert mehrerer Semaphore „auf einen Schlag“ erniedrigt und erhöht werden kann.

Übungsaufgaben

9.1 Eine weitere Variante von Semaphoren sind binäre Semaphore, bei denen der Wert des Zählers immer nur 0 oder 1 ist. Soll der Zähler bereits hochgezählt werden, wenn er schon 1 ist, dann hat die Ausführung der Methode v keine Wirkung. Die Methode p ist wie gehabt. Man kann den Zähler in diesem Fall durch einen Wert des Typs boolean ersetzen: false für 1 und true für 0. Das Attribut kann in diesem Fall als ein Sperrflag interpretiert werden. Die Methode p kann lock heißen und dient zum Setzen der Sperre, die Methode v kann unlock heißen und dient zur Freigabe der Sperre. Geben Sie eine Implementierung eines solchen binären Semaphors an, wobei die Klasse Lock heißen soll!

9.2 Für das Problem zur Herstellung einer vorgegebenen Ausführungsreihenfolge in Abschnitt 3.1.3 wurde zur Lösung ein systematischer Ansatz verwendet, bei dem im konkreten Beispiel 6 Semaphore benötigt wurden (s. Listing 3.3). Gibt es auch eine Lösung mit weniger Semaphoren?

9.3 Angenommen, Sie haben zwei Threads. Ein Thread gibt fortlaufend „ich bin der eine“ aus, der andere ebenfalls fortlaufend „ich bin der andere“. Sorgen Sie durch Synchronisation über Semaphore dafür, dass die Ausgaben immer abwechselnd (alternierend) erfolgen! Das heißt, kein Thread soll zwei Mal direkt hintereinander seine Meldung ausgeben.

9.4 Erweitern Sie die Klasse Semaphore um mehrere Varianten der Methoden p:

a) Eine Methode, die über einen Interrupt unterbrechbar ist.

b) Zwei Methoden, bei denen eine maximale Wartezeit angegeben werden kann. Die Methoden sollen durch einen Rückgabewert des Typs boolean anzeigen, ob sie erfolgreich waren, das heißt, ob der Wert erniedrigt wurde (Rückgabewert true) oder ob die angegebene Zeit abgelaufen ist (Rückgabewert false). Die beiden Methoden sollen sich dadurch unterscheiden, dass eine der Methoden unterbrechbar ist und die andere nicht.

9.5 Erweitern Sie die Klasse Semaphore so, dass abgefragt werden kann, welche Threads in der Methode p warten!

10 Message Queues und Pipes

Message Queues und Pipes sind Kommunikationsmechanismen aus dem Betriebssystem UNIX bzw. Linux. In den folgenden zwei Abschnitten werden sie in Java nachprogrammiert.

Lesen Sie die Abschnitte 3.2 und 3.3 des Lehrbuchs.

Zusammenfassung

Message Queues und Pipes sind Lösungen für das Erzeuger-Verbraucher-Problem mit mehr als einem Pufferplatz. In Message Queues abgelegte Bytefolgen bleiben als Einheit erhalten; beim Empfangen wird folglich eine Bytefolge zurückgeliefert, die zuvor in genau diesem Umfang gesendet wurde. Hingegen werden die Bytes mehrerer Sendeoperationen in einer Pipe so hintereinander abgelegt, dass man die Grenzen zwischen den Bytefolgen unterschiedlicher Sendeoperationen nicht mehr erkennen kann. Beim Empfangen wird eine Folge von Bytes zurückgeliefert, die nur ein Teil einer gesendeten Bytefolge sein oder aus mehreren Bytefolgen bestehen kann.

Übungsaufgaben

10.1 Wie kann man das Verhalten eines Semaphors mit einer Message Queue bzw. einer Pipe nachahmen?

10.2 Wie kann man das Erzeuger-Verbraucher-Problem mit Hilfe von Semaphoren, aber ohne weitere Nutzung von synchronized, wait, notify und notifyAll realisieren?

10.3 Schreiben Sie ein Programm, in dem die Klasse MessageQueue wie folgt verwendet wird: Zwei Threads sollen in einer Endlosschleife (oder sehr oft) über zwei MessageQueues Nachrichten austauschen, wobei eine MessageQueue für das Senden der Nachrichten von Thread 1 zu Thread 2 und die andere MessageQueue für die umgekehrte Richtung benutzt wird. Thread 1 soll dabei immer abwechselnd auf MessageQueue 1 senden und von MessageQueue 2 empfangen, während Thread 2 abwechselnd zuerst von MessageQueue1 empfangen und dann auf MessageQueue 2 senden soll.

11 Weitere Synchronisationsprobleme und deren Lösung

Das Philosophen-Problem und das Leser-Schreiber-Problem sind zwei Beispiele für „klassische“ Synchronisationsprobleme, an denen man demonstrieren kann, wie gut man mit unterschiedlichen Synchronisationsmechanismen diese Probleme lösen kann. Zum Abschluss werden als eine Art Zusammenfassung der unterschiedlichen Programmbeispiele der Kapitel 2 und 3 des Lehrbuchs Schablonen für Synchronized-Methoden präsentiert sowie auf das Thema Konsistenz eingegangen.

Lesen Sie die Abschnitte 3.4, 3.5 und 3.6 des Lehrbuchs.

Zusammenfassung

Das Philosophen-Problem ist ein Problem, bei dem der Ablauf von zyklisch angeordneten Threads so synchronisiert werden muss, dass keine zwei benachbarten Threads eine kritische Tätigkeit (im Beispiel Essen) gleichzeitig ausführen können, aber nicht benachbarte Threads die kritische Tätigkeit nach Möglichkeit sehr wohl gleichzeitig ausführen können. Für dieses Problem wurden Lösungen mit synchronized-wait-notify-notifyAll, einfachen Semaphoren und Semaphorgruppen vorgestellt.
Beim Leser-Schreiber-Problem darf eine Tätigkeit (im Beispiel Lesen) von mehreren Threads gleichzeitig ausgeführt werden, eine andere Tätigkeit (im Beispiel Schreiben) aber nur von höchstens einem Thread zu einem Zeitpunkt. Das Problem kann mit synchronized-wait-notify-notifyAll oder additiven Semaphoren gelöst werden. Die eine Lösung bevorzugte die Schreiber, während die andere Lösung die Leser bevorzugte.
Als Zusammenfassung wurden Synchronized-Methoden in 6 Kategorien klassifiziert und in Form von Schablonen angegeben:

- eine lesende Methode ohne wait,
- eine lesende Methode mit wait,
- eine schreibende Methode ohne wait und ohne notify-notifyAll,
- eine schreibende Methode mit wait, aber ohne notify-notifyAll,
- eine schreibende Methode ohne wait, aber mit notify-notifyAll,
- und eine schreibende Methode mit wait und mit notify-notifyAll.

Übungsaufgaben

11.1 Im Lehrbuch wurden zwei Lösungen für das Leser-Schreiber-Problem angegeben: eine mit einer Bevorzugung der Schreiber und eine mit einer Bevorzugung der Leser. Geben Sie eine faire Lösung des Leser-Schreiber-Problems an! Verwenden Sie dazu synchronized-wait-notify-notifyAll!

11.2 Finden Sie für die folgenden Synchronized-Methoden die Entsprechungen bei den Schablonen-Methoden m1 bis m6: p und v aus Listing 3.1, put und get aus Listing 3.6 sowie takeFork und putFork aus Listing 3.9!

12 Die Concurrent-Klassenbibliothek

Vor der Java-Version 5 gab es zur Unterstützung der Parallelität nur die in der Kurseinheit „Grundlegende Synchronisationskonzepte in Java“ vorgestellten Klassen und Konzepte. Seit der Java-Version 5 wird Java mit einer mächtigen Klassenbibliothek zur Unterstützung der Parallelität ausgeliefert, die z.B. die Klasse Semaphore schon enthält. Im Folgenden erhalten Sie einen Überblick über die wichtigsten Klassen in dieser so genannten Concurrent-Klassenbibliothek. Dabei werden die Klassen, die erst in späteren Java-Versionen dazugekommen sind, hier (noch) nicht behandelt.

Lesen Sie Abschnitt 3.7 des Lehrbuchs.

Zusammenfassung

Die Concurrent-Klassenbibliothek lässt sich in fünf Themenbereiche unterteilen:

- Mit Hilfe der Schnittstellen Executor und ExecutorService kann man Aufträge erteilen, die asynchron ausgeführt werden. Die Schnittstelle ExecutorService enthält auch Methoden, mit denen man das Ergebnis der Aufträge über ein Future-Objekt (später) abholen kann. Eine wichtige Klasse ist die Klasse ThreadPool, die die Schnittstelle ExecutorService implementiert. Die Verwendung eines ThreadPools kann es ersparen, selbst Threads in seinem Programm erzeugen und starten zu müssen.
- Locks und Conditions sind eine Alternative für synchronized, wait, notify und notifyAll. Lock ist eine Klasse mit den Methoden lock und unlock, wobei ein Aufruf von lock dem Betreten eines Synchronized-Blocks und unlock dem Verlassen des Synchronized-Blocks entspricht. Conditions sind Objekte, die man sich von einem Lock-Objekt geben lassen kann und die mit diesem Lock assoziiert sind. Auf Conditions kann man mit await warten. Wie bei wait wird der Thread dabei nicht nur blockiert, sondern es wird auch die Sperre auf den mit der Condition assoziierten Lock freigegeben. Der Thread wartet an dem Condition-Objekt, auf das er die Methode await aufgerufen hat. Mit den Methoden signal und signalAll der Klasse Condition kann man irgendeinen bzw. alle an dem Condition-Objekt wartenden Threads wecken, die wie bei notify bzw. notifyAll erst wieder die Sperre setzen müssen, bevor sie weiterlaufen können. Da man sich zu einem Lock beliebig viele Condition-Objekte geben lassen

kann, kann das Aufwecken von Threads gezielter erfolgen als mit notify bzw. notifyAll.

- Wie die Wrapper-Klassen Boolean, Integer, Long usw. legen die Atomic-Klassen eine Objekthülle um Variable des Typs boolean, int, long usw. Im Gegensatz zu den Wrapper-Klassen kann man mit den Atomic-Klassen auch Änderungen an den Variablen vornehmen, wobei der Zugriff thread-safe (d.h. für die parallele Nutzung durch mehrere Threads geeignet) ist. Mit den Atomic-Klassen lässt sich eine Synchronisation ohne Sperren (lock-free) realisieren, die in speziellen Situationen (zum Beispiel, wenn die Zugriffe auf die gemeinsam genutzten Daten nur kurze Zeit dauern) einem Zugriff mit Sperren überlegen ist.
- Zu den Synchronisationsklassen zählen die Klassen Semaphore (die Methoden p und v heißen dort acquire und release), CountDownLatch (ein Zähler wird heruntergezählt und Threads warten, bis der Zähler 0 wird), CyclicBarrier (N Threads können damit gegenseitig aufeinander warten) und Exchanger (zwei Threads tauschen zwei Objekte aus).
- Queue-Klassen bieten einen synchronisierten Datenaustausch zwischen Threads nach dem Erzeuger-Verbraucher-Prinzip ähnlich wie die Klasse MessageQueue an.

Übungsaufgaben

12.1 Geben Sie eine eigene Implementierung von ReentrantLock mit den Methoden lock und unlock an! Mit lock wird eine Sperre gesetzt und mit unlock wird diese Sperre wieder freigegeben. Wird lock aufgerufen bei gesetzter Sperre, so muss gewartet werden, bis die Sperre freigegeben wird. Falls aber derjenige Thread, der die Sperre gesetzt hat, lock erneut aufruft, so soll dieser nicht warten (das ist die Bedeutung von Reentrant). Die Sperre soll bei wiederholten Aufrufen von lock für andere Threads dann wieder zu setzen sein, falls der Thread, der die Sperre hält, entsprechend oft unlock aufgerufen hat. Um Conditions müssen Sie sich bei dieser Aufgabe nicht kümmern.

12.2 Realisieren Sie einen additiven Semaphor mit Locks und Conditions!

12.3 Schreiben Sie ein kleines Demoprogramm für die Klasse CountDownLatch! Starten Sie dazu einige Threads, die auf einen CountDownLatch mit await warten und danach eine Zeile auf die Console ausgeben!

Zählen Sie dann den CountDownLatch mit countDown in einem anderen Thread stufenweise herunter!

12.4 Programmieren Sie die Klasse CountDownLatch in einer eigenen Klasse nach! Beachten Sie dabei, dass die Methode await eine InterruptedException werfen kann, die vom Aufruf von wait nach außen gegeben werden kann! Das heißt, dass Sie diese Exception nicht in der Methode await fangen müssen.

13 Das Fork-Join-Framework

In der Java-Version 7 wurde das Konzept von Thread-Pools für baumartige Berechnungen erweitert. Der entscheidende Punkt ist, dass beim Fork-Join-Framework ein Thread des Pools, der einen neuen Auftrag erteilt und auf dessen Ende wartet, während des Wartens nicht für andere Aufträge blockiert ist. Deshalb kann mit dem Fork-Join-Framework auch ein großer Auftrag, der baumartig berechnet wird und einen Baum mit vielen Knoten generiert, mit wenigen Threads bearbeitet werden, was mit einem normalen Thread-Pool nicht möglich ist.
In Java 8 wurden zwei weitere Erweiterungen eingeführt, die mit Parallelität zu tun haben. Das ist zum einen das Data-Streaming-Framework, mit dem ein Datenstrom wie in einem Fließband in unterschiedlichen Stationen verarbeitet wird, wobei optional eine parallele Verarbeitung möglich ist. Zum anderen gibt es seit Java 8 die Completable Futures, mit denen ebenfalls Verarbeitungsschritte verkettet werden können. Diese beiden Themen werden im Rahmen des Moduls „Fortgeschrittene Programmiertechniken“ nicht behandelt.

Lesen Sie Abschnitt 3.8 des Lehrbuchs. Gerne können Sie optional auch die Abschnitte 3.9 und 3.10 mehr oder weniger genau lesen. Die Themen Data-Streaming-Framework und Completable Futures sind kein Pflichtlehrstoff.

Zusammenfassung

Das Fork-Join-Framework erweitert das Konzept von Thread-Pools für baumartige Berechnungen. In der Regel leitet man aus den Klassen RecursiveAction oder RecursiveTask eine eigene Klasse ab. Ein Objekt dieser Klasse stellt einen Knoten des Berechnungsbaums dar. In einer überschriebenen Methode wird geprüft, ob der Auftrag klein genug ist, um direkt bearbeitet zu werden. Falls dies nicht der Fall ist, werden weitere Objekte der aus RecursiveAction oder RecursiveTask abgeleiteten Klasse erzeugt und durch Aufruf der Methode fork dem Thread-Pool zur Bearbeitung übergeben. Durch Anwendung der Methode join auf diese Objekte kann man auf das Ende der Bearbeitung warten und die Teilergebnisse zu einem Gesamtergebnis kombinieren.

Übungsaufgaben

13.1 Realisieren Sie mit Hilfe des Fork-Join-Frameworks die Berechnung der Fibonacci-Zahlen!

13.2 Realisieren Sie mit Hilfe des Fork-Join-Frameworks das Sortieren eines Feldes von Zahlen mit Hilfe des Quicksort-Algorithmus!

14 Verklemmungen

Synchronisation kann auch negative Auswirkungen haben. Es kann durch eine falsche Synchronisation nämlich dazu kommen, dass Threads in einer Kette unendlich lange aufeinander warten. Man spricht in diesem Zusammenhang von Verklemmungen. Im Folgenden erfahren Sie, wie es dazu kommen kann und was man dagegen tun kann.

Lesen Sie die Abschnitte 3.11 und 3.12 des Lehrbuchs sowie die Zusammenfassung in Abschnitt 3.13.

Zusammenfassung

Eine Verklemmung ist eingetreten, wenn es einen Zyklus von Threads gibt, so dass ein Thread auf ein Betriebsmittel wartet, welches der im Zyklus folgende Thread besitzt. Betriebsmittel können z.B. Objekte sein, deren Zugriff mit synchronized, Semaphoren oder Locks synchronisiert wird. Voraussetzung für Verklemmungen sind neben dem zyklischen Warten, dass die Betriebsmittel nur unter gegenseitigem Ausschluss benutzbar sind, dass die Betriebsmittel einem Thread nicht entzogen werden können, und dass Threads bereits Betriebsmittel besitzen und weitere anfordern.
Zur Vermeidung von Verklemmungen wurden folgende Gegenmaßnahmen vorgestellt: alle Betriebsmittel werden von einem Thread „auf einen Schlag“ angefordert, die Anforderung der Betriebsmittel erfolgt immer in einer bestimmten Reihenfolge, an die sich alle Threads halten, oder es wird bei jeder Anforderung von Betriebsmitteln eine Bedarfsanalyse mit Hilfe des Bankier-Algorithmus durchgeführt.

Übungsaufgaben

14.1 Im Folgenden wird modelliert, wie mehrere Studierende für das Anfertigen einer Seminararbeit die Bücher „FOPT1“, „FOPT2“ und „FOPT3“ benötigen, wobei diese Bücher als Semaphore modelliert werden. Vom ersten benötigten Buch „FOPT1“ sind 4 Exemplare in der Bibliothek vorhanden, vom zweiten Buch „FOPT2“ 2 und vom dritten Buch „FOPT3“ 5.

a) Ergänzen Sie die Klasse Studi!

```
class Studi extends Thread
{
    private Semaphore buch1, buch2, buch3;

    public Studi(Semaphore b1, Semaphore b2, Semaphore b3)
    {
        buch1 = b1;
        buch2 = b2;
        buch3 = b3;
    }

    public void run()
    {
        //1. Buch, dann 2. Buch, dann 3. Buch belegen
        //Seminararbeit erstellen
          (nicht näher interessant)
        //alle Bücher wieder zurücklegen
    }
}
```

b) Ergänzen Sie das folgende Hauptprogramm, in dem die Bücher und die Studis erzeugt werden!

```
public class Seminarbetrieb
{
    public static void main(String[] args)
    {
        Semaphore buch1, buch2, buch3;
        //Bücher erzeugen, 20 Studis erzeugen und starten
        for(int i = 0; i < 20; i++)
        {
            Studi s = ...
        }
    }
}
```

c) Kann das Programm eine Verklemmung hervorrufen?

14.2 Betrachten Sie das Verklemmungsbeispiel aus Listing 3.32 des Lehrbuchs! Wie kann die Verklemmungsgefahr verhindert werden?

Literaturverzeichnis

s. Kapitel „Literatur“ im Lehrbuch

Lösungen zu den Aufgaben

Aufgabe 1.1

Wir werden die beiden Begriffe nicht unterscheiden, sondern als Synonyme verwenden. Andere verstehen unter Parallelität echte Parallelität, während Nebenläufigkeit mit Pseudoparallelität gleichgesetzt wird. Dass Parallelität und Nebenläufigkeit nicht unterschieden werden, ist dadurch gerechtfertigt, dass unsere Betrachtungsebene auf der Ebene der Threads den Unterschied zwischen echter Parallelität und Pseudoparallelität ohnehin verdeckt.

Aufgabe 1.2

Programm	Kochbuch
Prozess	Küche
Thread	Koch
Rechner	Haus
Betriebssystem	Hausmeister
verteiltes System	mehrere Häuser, Dorf

Aufgabe 2.1

Falls man die Run-Methode in einer Klasse ansiedeln will, die bereits aus einer anderen Klasse abgeleitet wird, so kann diese Klasse nicht auch noch aus der Klasse Thread abgeleitet werden, da es in Java keine Mehrfachvererbung gibt. Eine Klasse, die aber aus einer anderen Klasse abgeleitet ist und eventuell noch eine oder mehrere Schnittstellen implementiert, kann immer noch eine weitere Schnittstelle implementieren.

Aufgabe 2.2

```
public class LoopRunnable implements Runnable
{
    private String myName;

    public LoopRunnable(String name)
    {
        myName = name;
    }

    public void run()
    {
        for(int i = 1; i <= 1000; i++)
```

```
            {
                System.out.println(myName + " (" + i + ")");
            }
        }

        public static void main(String[] args)
        {
            LoopRunnable runnable1 = new LoopRunnable(
                                                    "Thread 1");
            LoopRunnable runnable2 = new LoopRunnable(
                                                    "Thread 2");
            LoopRunnable runnable3 = new LoopRunnable(
                                                    "Thread 3");
            Thread t1 = new Thread(runnable1);
            Thread t2 = new Thread(runnable2);
            Thread t3 = new Thread(runnable3);
            t1.start();
            t2.start();
            t3.start();
        }
    }
```

Aufgabe 2.3

```
public class LoopRunnableLambda
{
    private static void execute(String myName)
    {
        for(int i = 1; i <= 1000; i++)
        {
            System.out.println(myName + " (" + i + ")");
        }
    }

    public static void main(String[] args)
    {
        new Thread(()->execute("Thread 1")).start();
        new Thread(()->execute("Thread 2")).start();
        new Thread(()->execute("Thread 3")).start();
    }
}
```

Aufgabe 2.4

Bei n Threads und a Aktionen pro Thread werden insgesamt n * a Aktionen durchgeführt. Dafür gibt es (n * a)! mögliche Reihenfolgen. Die a Aktionen innerhalb eines Threads laufen allerdings in einer festen Reihenfolge ab und können nicht vertauscht werden. Bei den (n * a)! Möglichkeiten hat man also pro Thread a!-mal zu viele Reihenfolgen berechnet. Bei n Threads sind das dann n * a!-mal zu viele, die man

durch eine Division wieder „herausrechnen“ muss. Insgesamt ergibt sich deshalb für die Anzahl der möglichen Abläufe: (n * a)! / (n * a!).
Etwas allgemeiner: Wenn ein Thread i ai Aktionen durchführt, ist die Anzahl der möglichen Abläufe (a1 + a2 + … + an)! / (a1! * a2! * … * an!).

Aufgabe 3.1

```
public class Counter
{
    private int counter;

    public void increment()
    {
        if(counter < 10)
        {
            counter++;
        }
    }

    public void decrement()
    {
        if(counter > 0)
        {
            counter--;
        }
    }
}
```

Angenommen, der Zähler hat zu einem bestimmten Zeitpunkt den Wert 1. Nun wollen zwei Threads gleichzeitig die Methode decrement auf ein gemeinsam benutztes Objekt der Klasse Counter anwenden. Der erste Thread führt die Prüfung der Bedingung in der If-Anweisung durch und kommt zu dem Ergebnis, dass die Bedingung counter > 0 erfüllt ist. Bevor er aber den Zähler herunterzählen kann, wird auf den zweiten Thread umgeschaltet, der die Bedingung in der Methode decrement ebenfalls als erfüllt ansieht und den Zähler um eins auf den aktuellen Stand 0 erniedrigt. Wird nun wieder auf den ersten Thread geschaltet, der die Bedingung in der If-Anweisung bereits erfolgreich geprüft hat, so wird auch dieser Thread anschließend den Zählerwert erniedrigen, so dass der Zähler nun den Wert -1 hat. Dieser Zustand verletzt die Vorgabe, dass der Zählerstand nicht negativ werden soll.
Mit einem ähnlichen Beispiel kann man zeigen, dass es möglich ist, dass der Zählerstand auf 11 ansteigen kann, wenn man mit einem Zählerstand von 9 startet und zwei Threads annimmt, die gleichzeitig die Methode increment ausführen wollen.

Wird der gedankliche Ablauf mit mehr als zwei Threads durchgeführt, so können sich auch Zählerstände von -2, -3, -4 usw. bzw. 12, 13, 14 usw. ergeben.

Aufgabe 3.2

a) Hier gibt es kein Problem, da i eine lokale Variable ist. Von jedem Thread wird die Methode run aufgerufen. Deshalb gibt es ein Exemplar von i für jeden Aufruf und daher für jeden Thread. Also gibt es keine gemeinsame Benutzung von Daten und somit keine Probleme.

b1)

```
public class LoopRunnable implements Runnable
{
    public void run()
    {
        for(int i = 1; i <= 1000; i++)
        {
            System.out.println(Thread.currentThread().
                               getName() +
                               " (" + i + ")");
        }
    }

    public static void main(String[] args)
    {
        LoopRunnable runnable1 = new LoopRunnable();
        LoopRunnable runnable2 = new LoopRunnable();
        LoopRunnable runnable3 = new LoopRunnable();
        Thread t1 = new Thread(runnable1, "Thread 1");
        Thread t2 = new Thread(runnable2, "Thread 2");
        Thread t3 = new Thread(runnable3, "Thread 3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

b2)

```
public class LoopRunnable implements Runnable
{
    public void run() // wie zuvor in b1
    ...

    public static void main(String[] args)
    {
        LoopRunnable runnable = new LoopRunnable();
        Thread t1 = new Thread(runnable, "Thread 1");
        Thread t2 = new Thread(runnable, "Thread 2");
        Thread t3 = new Thread(runnable, "Thread 3");
```

```
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Auch in diesem Fall gibt es keine gemeinsam benutzten Daten und somit keine Probleme (Begründung wie in Teilaufgabe a).

c)

```
public class Loop extends Thread
{
    private int i;

    public Loop(String name)
    {
        super(name);
    }

    public void run()
    {
        for(i = 1; i <= 1000; i++)
        {
            System.out.println(getName() + " (" + i + ")");
        }
    }

    public static void main(String[] args)
    {
        Loop t1 = new Loop("Thread 1");
        Loop t2 = new Loop("Thread 2");
        Loop t3 = new Loop("Thread 3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Auch in diesem Fall gibt es keine gemeinsame Benutzung von Daten, aber aus einem anderen Grund: Da für jeden Thread ein eigenes Objekt des Typs Loop erzeugt wird, arbeitet jeder Thread mit seinem eigenen Attribut i und somit auch in diesem Fall mit je einem eigenen Exemplar.

d) Der Programmcode entsteht durch Mischen der Lösungen aus den Teilaufgaben b und c. Wenn jeder Thread sein eigenes Runnable-Objekt hat, dann gibt es wieder keine gemeinsam benutzten Daten. Wenn aber alle Threads dasselbe Runnable-Objekt benutzen, dann gibt es in der Tat nur ein einziges Exemplar von i. In diesem Fall ist die Ausgabe ziemlich wenig vorhersagbar und vielleicht ein wenig erstaunlich. Ein Lauf des Programms, den ich auf meinem Computer beobachtet habe, endete beispielsweise mit den folgenden Ausgabezeilen:

```
...
```

```
Thread 3 (998)
Thread 3 (999)
Thread 3 (1000)
Thread 2 (931)
Thread 1 (1002)
```

Aufgabe 4.1

```
class SchlafObjekt
{
    public void schlafen(int sekunden)
    // Alternative: public synchronized void schlafen(...)
    {
        System.out.println("Beginn des Schlafens");
        try
        {
            Thread.sleep(sekunden * 1000);
        }
        catch(InterruptedException e)
        {
        }
        System.out.println("Ende des Schlafens");
    }
}

public class Schlaefer extends Thread
{
    private SchlafObjekt meinSo;

    public Schlaefer(SchlafObjekt so)
    {
        meinSo = so;
    }

    public void run()
    {
        meinSo.schlafen(1);
    }

    public static void main(String[] args)
    {
        SchlafObjekt so = new SchlafObjekt();
        Schlaefer t1 = new Schlaefer(so);
        Schlaefer t2 = new Schlaefer(so);
        t1.start();
        t2.start();
    }
}
```

Die Ausgabe dieses Programms ist wie folgt:

```
Beginn des Schlafens
Beginn des Schlafens
```

```
Ende des Schlafens
Ende des Schlafens
```

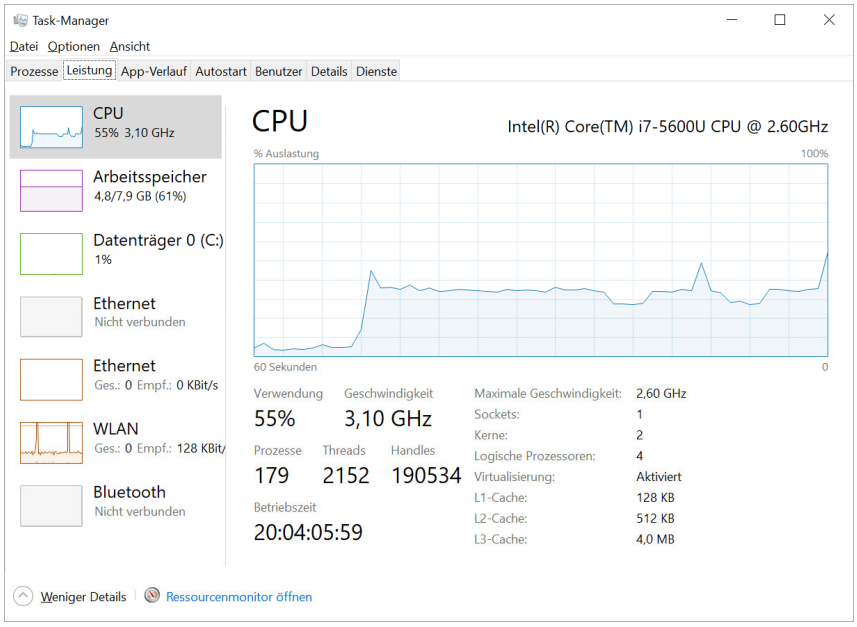
Man sieht, dass einer der Threads die Methode schlafen betritt. Bevor dieser Thread die Methode verlässt, betritt auch der andere Thread die Methode (dies sieht man an den ersten beiden Zeilen). Wird die Methode schlafen allerdings mit synchronized markiert, sieht die Ausgabe so aus:

```
Beginn des Schlafens
Ende des Schlafens
Beginn des Schlafens
Ende des Schlafens
```

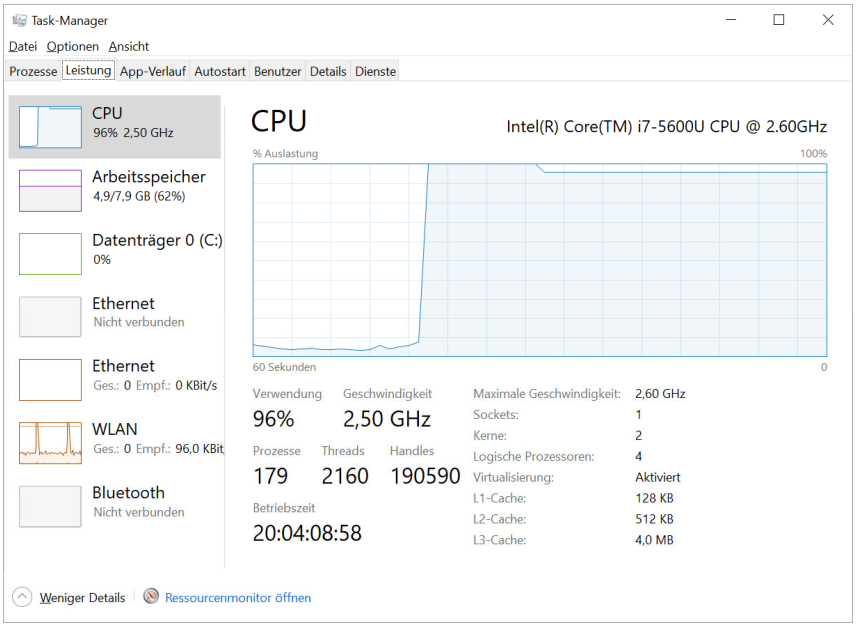
Man kann deutlich erkennen, dass erst einer der Threads die Methode schlafen verlassen muss, bevor sie der andere Thread betreten kann. Werden statt des einen SchlafObjekts zwei unterschiedliche Schlaf-Objekte verwendet (für jeden der beiden Threads ein anderes), dann ergibt sich die erste Ausgabe, unabhängig davon, ob synchronized verwendet wird oder nicht. Das zeigt, dass synchronized nur auf das betreffende Objekt wirkt.

Aufgabe 4.2

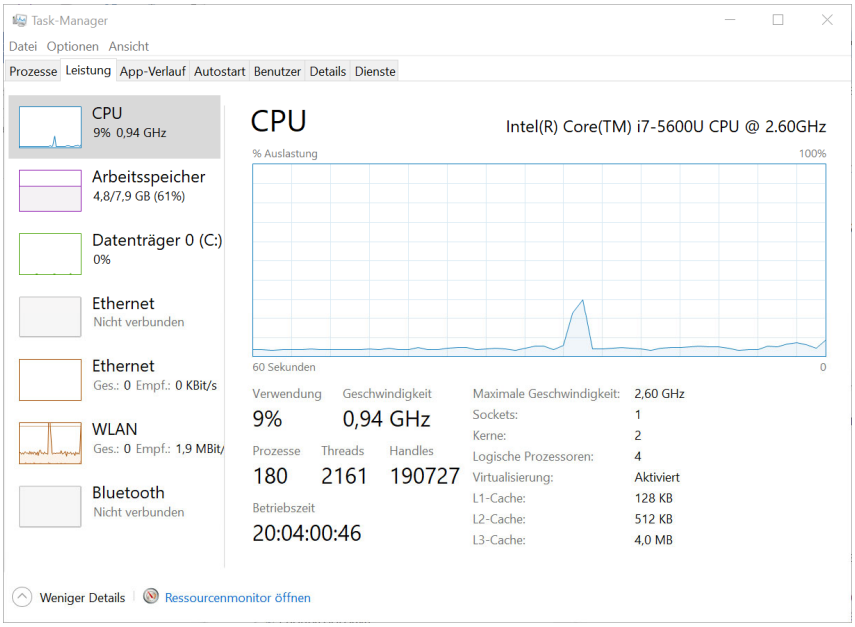
Die folgende Aufnahme des Task-Managers zeigt die CPU-Auslastung während der Ausführung des Programms mit aktivem Warten bei zwei Threads:



Nach dem Starten des Programms geht die Auslastung auf ca. 30 bis 40%. Dies liegt nicht an dem Aufruf von Thread.sleep, da beim Schlafen eines Threads der Prozessor nicht belastet wird. Allerdings verursacht der andere Thread, der ständig das Sperrflag abfragt, diese Last. Bei einem Mehrprozessor- bzw. Mehrkern-System ist die Auslastung dann 100%, wenn alle Prozessoren bzw. Kerne voll ausgelastet sind. Deshalb ist in diesem Fall die Auslastung nicht 100%, weil nur ein Thread eine hohe Last erzeugt und somit nur ein Prozessor bzw. Kern ausgelastet wird. Werden dagegen mehrere Threads gestartet, dann fragen mehrere Threads ständig das Sperrflag ab. In diesem Fall erhält man eine Auslastung von nahezu 100%.



Wird das Sperrflag entfernt und dafür der Zugriff in korrekter Weise mit synchronized geregelt, ist bei Ausführung des Programms die CPU-Auslastung deutlich geringer. Sie ist sogar so gering, dass man die Ausführung des Programms anhand der CPU-Auslastung nicht einmal bemerkt. Lediglich der Start des Programms ist auf dem Diagramm an der „Haifischflosse“ erkennbar:



Aufgabe 4.3

Die Aufgabenstellung hat lediglich eine korrekte Synchronisation des Exemplars von i verlangt, nicht aber, dass die Ausgaben von i synchronisiert sind und somit i in aufsteigender Reihenfolge ausgegeben wird. Ein Programm, das lediglich die Anforderung nach einer korrekten Synchronisation von i erfüllt, könnte so aussehen:

```
public class LoopRunnable implements Runnable
{
    private int i;

    private synchronized int incrementAndGet()
    {
       i++;
       return i;
    }

    public void run()
    {
        int j;
        while((j = incrementAndGet()) <= 1000)
        {
            System.out.println(Thread.currentThread().
                               getName() +
                               " (" + j + ")");
        }
    }

    public static void main(String[] args)
    {
        LoopRunnable runnable = new LoopRunnable();
```

```
        Thread t1 = new Thread(runnable, "Thread 1");
        Thread t2 = new Thread(runnable, "Thread 2");
        Thread t3 = new Thread(runnable, "Thread 3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Dieses Programm garantiert, dass exakt 1000 Zeilen ausgegeben werden, in denen die Zahlen zwischen 1 und 1000 jeweils genau ein Mal vorkommen. Die Reihenfolge ist aber nicht garantiert. Ein Beispiellauf auf meinem Rechner sah so aus (mittlerer Teil durch Punkte ersetzt):

```
Thread 1 (1)
Thread 1 (4)
Thread 3 (3)
Thread 3 (6)
Thread 2 (2)
Thread 2 (8)
Thread 2 (9)
Thread 2 (10)
Thread 3 (7)
Thread 1 (5)
...
Thread 3 (998)
Thread 3 (999)
Thread 3 (1000)
Thread 2 (982)
Thread 1 (989)
```

Aufgabe 5.1

Zum Ausprobieren.

Aufgabe 5.2

In Listing 2.14 werden die Ergebnisse eingesammelt, nachdem alle Threads zu Ende gelaufen sind. Eine gleichzeitige, gemeinsame Benutzung von Daten ist ausgeschlossen. In Listing 2.20 dagegen melden die parallel ausgeführten Threads ihr Ergebnis einem gemeinsam benutzten Objekt. Da mehrere Threads gleichzeitig fertig werden können, ist eine gleichzeitige, gemeinsame Benutzung von Daten möglich. Der Zugriff muss daher synchronisiert werden. Wird dies unterlassen, so könnte die Meldung des Ergebnisses eines Threads verloren gehen.

Aufgabe 5.3

a)

```
class BooleanCountingResult
{
    private int result;
    private int numberOfTasks;
    private int depth;

    public BooleanCountingResult(int result,
                                 int numberOfTasks,
                                 int depth)
    {
        this.result = result;
        this.numberOfTasks = numberOfTasks;
        this.depth = depth;
    }

    public BooleanCountingResult(int result)
    {
        this(result, 1, 1);
    }

    public int getResult()
    {
        return result;
    }

    public int getNumberOfTasks()
    {
        return numberOfTasks;
    }

    public int getDepth()
    {
        return depth;
    }
}

class BooleanCountingTask
      implements Task<BooleanCountingResult>
{
    private static final int MINLENGTH = 100;

    private boolean[] array;
    private int start;
    private int end;
    private int splitFactor;

    public BooleanCountingTask(boolean[] array, int start,
                               int end, int splitFactor)
    {
        this.array = array;
        this.start = start;
        this.end = end;
```

```
    this.splitFactor = splitFactor;
}

public BooleanCountingTask(boolean[] array,
                           int splitFactor)
{
    this(array, 0, array.length-1, splitFactor);
}

public BooleanCountingTask(boolean[] array)
{
    this(array, 2);
}

public boolean isDivisible()
{
    return end-start+1 > MINLENGTH;
}

public List<Task<BooleanCountingResult>> split()
{
    int tempStart = start;
    int tempEnd;
    int howMany = (end-start+1) / splitFactor;
    int numberOfTasks = splitFactor;
    if(howMany < 1)
    {
        howMany = 1;
        numberOfTasks = end-start+1;
    }
    List<Task<BooleanCountingResult>> tasks =
                                    new ArrayList<>();
    for(int i = 0; i < numberOfTasks; i++)
    {
        if(i < numberOfTasks-1)
        {
            tempEnd = tempStart + howMany - 1;
        }
        else
        {
            tempEnd = end;
        }
        tasks.add(new BooleanCountingTask(array,
                    tempStart, tempEnd, splitFactor));
        tempStart = tempEnd + 1;
    }
    return tasks;
}

public BooleanCountingResult execute()
{
    int result = 0;
    for(int i = start; i <= end; i++)
    {
        if(array[i])
        {
```

```
                result++;
            }
        }
        return new BooleanCountingResult(result);
    }

    public BooleanCountingResult
           combine(List<BooleanCountingResult> results)
    {
        int r = 0;
        int numberOfTasks = 0;
        int depth = 0;
        for(BooleanCountingResult result: results)
        {
            r += result.getResult();
            numberOfTasks += result.getNumberOfTasks();
            if(depth < result.getDepth())
            {
                depth = result.getDepth();
            }
        }
        return new BooleanCountingResult(r,
                                         numberOfTasks+1,
                                         depth+1);
    }
}

public class TaskTreeBooleanCounting
{
    public static void main(String[] args)
    {
        boolean[] array = new boolean[100000];
        for(int i = 0; i < array.length; i++)
        {
            if(i % 2 == 0)
            {
                array[i] = true;
            }
        }
        BooleanCountingResult result = TaskNodeExecutor.
            executeAll(new BooleanCountingTask(array, 5));
        if(result.getResult() != array.length/2)
        {
            System.out.print(">>>>>ERROR>>>>>");
        }
        System.out.println("SearchResult: " +
                           result.getResult() +
                           " (threads: " +
                           result.getNumberOfTasks() +
                           ", depth: " +
                           result.getDepth() + ")");

    }
}
```

b) Wenn nur die Blätter gezählt werden sollen, muss lediglich die letzte Anweisung der Methode combine der Klasse BooleanCountingTask

```
return new BooleanCountingResult(r, numberOfTasks+1,
                                 depth+1);
```

wie folgt geändert werden:

```
return new BooleanCountingResult(r, numberOfTasks,
                                 depth+1);
```

Aufgabe 6.1

If ist nicht korrekt, da man nicht sicher sein kann, dass der Thread, der geweckt wurde, auch tatsächlich als nächster die Sperre setzt. Das heißt, es kann nicht garantiert werden, dass zwischen dem Erhöhen von places durch einen ausfahrenden Thread und dem Weiterlaufen des geweckten Threads kein anderer Thread gelaufen ist, der die Sperre auf das Parkhaus gesetzt und places bereits wieder auf 0 erniedrigt hat.

Aufgabe 6.2

Wenn in der Methode enter die Methode wait aufgerufen wird, dann ist das Objekt garage an dieser Stelle zweifach gesperrt. Diese zweifache Sperrung wird aufgehoben, so dass ein anderer Thread den Synchronized-Block zum Verlassen des Parkhauses betreten kann. Wenn der geweckte Thread dann aufgeweckt wird und die Sperre wieder setzen kann, dann sperrt er das Objekt wieder zweifach. Nach dem Verlassen von enter wird dann der Sperrenzähler um eins heruntergezählt und nach dem Verlassen des Synchronized-Blocks wird die Sperre endgültig freigegeben. Nach dem erfolgreichen Ein- und Ausfahren bleibt das Parkhaus somit gesperrt, bis eine Meldung ausgegeben wurde. Erst nach der Ausgabe wird das Parkhaus freigegeben. Ein anderer Thread kann also erst nach der Ausgabe auf das Parkhaus zugreifen und auch erst danach seine Meldung bzgl. des Ein- oder Ausfahrens ausgeben.

Aufgabe 6.3

Dies ist eine Lösung, die zwar korrekt funktioniert, von der ich aber abraten würde (es sei denn, zu Testzwecken). Es könnte zum Beispiel sein, dass Sie die Parkhausklasse in anderen Anwendungen wiederverwenden wollen, in denen diese Ausgaben gar nicht passen. So könnte es etwa sein, dass in einer größeren Simulation überhaupt keine Ausgaben erscheinenen sollen, oder dass die Ausgaben in einer ganz anderen

Sprache erfolgen sollen, oder dass die Ausgaben nicht auf der Console, sondern in eine Datei oder auf eine grafische Benutzeroberfläche geschrieben werden sollen. Auch ist es denkbar, dass die Klasse in einem komplett anderen Kontext eingesetzt wird, die mit Autos gar nichts zu tun hat (die Parkhausklasse kommt in FOPT 2 als Klasse Semaphore in einem wesentlich allgemeineren Kontext wieder).
Übrigens: Dass man in den Methoden enter und leave die Methode getName zum Erfragen des Thread-Namens nicht aufrufen kann, da die Parkhausklasse nicht aus Thread abgeleitet ist, ist kein Nachteil, den man hier anführen könnte, denn man könnte den Thread-Namen des ausführenden Threads über Thread.currentThread().getName() ermitteln (s. Aufgabe 3.2).

Aufgabe 7.1

Wenn es nur einen Verbraucher gibt, dann kann der beschriebene Effekt nicht eintreten.

Aufgabe 7.2

```
public class ParkingGarageFair
{
    private int places;
    private ArrayList<Thread> waitingThreads;

    public ParkingGarageFair(int places)
    {
        this.places = places;
        this.waitingThreads = new ArrayList<Thread>();
    }

    public synchronized void enter() // einfahren
    {
        Thread me = Thread.currentThread();
        waitingThreads.add(me);
        while(waitingThreads.get(0) != me || places == 0)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        places--;
        waitingThreads.remove(0);
        notifyAll(); // wichtig !!!!
    }
```

```
    public synchronized void verlassen() // ausfahren
    {
        places++;
        notifyAll(); // wichtig !!!!
    }
}
```

Aufgabe 9.1

```
public class Lock
{
    private boolean locked;

    public Lock(boolean init)
    {
        locked = init;
    }

    public Lock()
    {
        this(false);
    }

    public synchronized void lock()
    {
        while(locked)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        locked = true;
    }

    public synchronized void unlock()
    {
        locked = false;
        notify();
    }
}
```

Die Methode unlock kann noch etwas verbessert werden: Nur, wenn das Sperrflag gesetzt ist, hat die Methode unlock überhaupt eine Wirkung. Deshalb muss nur in diesem Fall locked verändert und notify aufgerufen werden:

```
    public synchronized void unlock()
    {
        if(locked)
```

```
        {
            locked = false;
            notify();
        }
    }
```

Aufgabe 9.2

Man kann mit nur zwei Semaphoren auskommen: einen zwischen T1 einerseits und T2, T3 und T4 andererseits, und einen zwischen T2, T3 und T4 einerseits und T5 andererseits. In T1 wird nach Ausführung von a1 dreimal v auf dem ersten Semaphor ausgeführt. T2, T3 und T4 führen dann alle p je einmal auf dem ersten Semaphor aus, führen ihre Aktion durch (a2, a3 bzw. a4) und führen dann je einmal v auf dem zweiten Semaphor aus. T5 muss vor der Ausführung von a5 dreimal p auf dem zweiten Semaphor ausführen. Beide Semaphore werden mit 0 initialisiert.

Aufgabe 9.3

```
public class AlternatingThread extends Thread
{
    private Semaphore sem1;
    private Semaphore sem2;
    private String output;

    public AlternatingThread(Semaphore sem1,
                             Semaphore sem2,
                             String output)
    {
        this.sem1 = sem1;
        this.sem2 = sem2;
        this.output = output;
        start();
    }

    public void run()
    {
        while(true)
        {
            sem1.p();
            System.out.println(output);
            sem2.v();
        }
    }

    public static void main(String[] args)
    {
        Semaphore sem1 = new Semaphore(1);
        Semaphore sem2 = new Semaphore(0);
        new AlternatingThread(sem1, sem2,
                              "ich bin der eine");
```

```
        new AlternatingThread(sem2, sem1,
                              "ich bin der andere");
    }
}
```

Aufgabe 9.4

```
public class ExtendedSemaphore
{
    private int value;

    public ExtendedSemaphore(int init)
    {
        if(init < 0)
        {
            init = 0;
        }
        value = init;
    }

    public synchronized void p()
    {
        while(value == 0)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        value--;
    }

    public synchronized void pInterruptibly()
           throws InterruptedException
    {
        while(value == 0)
        {
            wait();
        }
        value--;
    }

    private synchronized boolean tryP()
    {
        if(value > 0)
        {
            value--;
            return true;
        }
        else
        {
            return false;
```

```
        }
    }

    public synchronized boolean tryP(long timeout)
    {
        if(timeout < 0)
        {
            throw new IllegalArgumentException();
        }
        long endTime = System.currentTimeMillis()+timeout;
        while(value == 0 &&
              System.currentTimeMillis() < endTime)
        {
            try
            {
                wait(endTime-System.currentTimeMillis());
            }
            catch(InterruptedException e)
            {
            }
        }
        return tryP();
    }

    public synchronized boolean tryPInterruptibly(long
                                                  timeout)
           throws InterruptedException
    {
        if(timeout < 0)
        {
            throw new IllegalArgumentException();
        }
        long endTime = System.currentTimeMillis()+timeout;
        while(value == 0 &&
              System.currentTimeMillis() < endTime)
        {
            wait(endTime - System.currentTimeMillis());
        }
        return tryP();
    }

    public synchronized void v()
    {
        value++;
        //notify reicht trotz unterschiedlicher
        //Wartebedingungen
        notify();
    }
}
```

Aufgabe 9.5

```
import java.util.*;

public class ExtendedSemaphore
{
    private int value;
    private LinkedList<Thread> waitingThreads;

    public ExtendedSemaphore(int init)
    {
        if(init < 0)
        {
            init = 0;
        }
        value = init;
        waitingThreads = new LinkedList<Thread>();
    }

    private void addCurrentThread()
    {
        waitingThreads.add(Thread.currentThread());
    }

    private void removeCurrentThread()
    {
        waitingThreads.remove(Thread.currentThread());
    }

    public synchronized void p()
    {
        addCurrentThread();
        while(value == 0)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        removeCurrentThread();
        value--;
    }

    public synchronized void pInterruptibly()
           throws InterruptedException
    {
        addCurrentThread();
        try
        {
            while(value == 0)
            {
                wait();
            }
```

```
            value--;
        }
        finally
        {
            removeCurrentThread();
        }
    }

    public synchronized void v()
    {
        value++;
        notify();
    }

    public synchronized int getNumberOfWaitingThreads()
    {
        return waitingThreads.size();
    }

    public synchronized List<Thread> getWaitingThreads()
    {
        //nicht waitingThreads zurückgeben, sondern Kopie
        return new LinkedList<Thread>(waitingThreads);
    }
}
```

Aufgabe 10.1

Mit Hilfe einer Message Queue: Das Senden einer beliebigen (eventuell leeren) Nachricht entspricht der Methode v, das Empfangen der Methode p.
Mit Hilfe einer Pipe: Das Senden eines einzigen Bytes entspricht der Methode v, das Empfangen eines Bytes der Methode p.

Aufgabe 10.2

Man benötigt drei Semaphore:

- einen Semaphor namens mutexSemaphore (Mutex steht für MUTual EXclusion), der – wie der Name sagt – für den gegenseitigen Ausschluss benutzt wird und das fehlende Synchronized ersetzt;
- einen Semaphor namens itemsSemaphore, dessen Zähler zählt, wie viele Elemente sich momentan im Puffer befinden und abgeholt werden können ohne zu blockieren;
- einen Semaphor namens placesSemaphore, dessen Zähler zählt, wie viele Plätze im Puffer frei sind und wie viele Daten abgelegt werden können ohne zu blockieren.

Vor dem Ablegen von Daten im Puffer wird auf placesSemaphore die Methode p angewendet. Wenn kein Platz zum Ablegen frei ist, wird der

aufrufende Thread blockiert. Nach dem Ablegen wird auf itemsSemaphore die Methode v ausgeführt, um zu signalisieren, dass ein Element mehr im Puffer ist. Das Abholen eines Elements aus dem Puffer erfolgt analog, wobei die Rollen der beiden Semaphore vertauscht sind. Mit Hilfe des Semaphors mutexSemaphore wird abgesichert, dass nicht mehr als ein Thread zu einem Zeitpunkt auf den Daten arbeitet: Achtung: Der Aufruf von p auf placesSemaphore bzw. itemsSemaphore muss in jedem Fall vor dem Aufruf von p auf mutexSemaphore erfolgen, da es sonst bei vollem bzw. leerem Puffer dazu führt, dass der kritische Abschnitt vom aufrufenden Thread nie mehr verlassen wird. Der Programmcode sieht wie folgt aus:

```
public class BufferSem
{
    private int head;
    private int tail;
    private int[] data;
    private Semaphore mutexSemaphore;
    private Semaphore itemsSemaphore;
    private Semaphore placesSemaphore;

    public BufferSem(int n)
    {
        head = 0;
        tail = 0;
        data = new int[n];
        mutexSemaphore = new Semaphore(1);
        itemsSemaphore = new Semaphore(0);
        placesSemaphore = new Semaphore(n);
    }

    public void put(int x)
    {
        placesSemaphore.p();
        mutexSemaphore.p();
        data[tail++] = x;
        if(tail == data.length)
        {
            tail = 0;
        }
        mutexSemaphore.v();
        itemsSemaphore.v();
    }

    public int get()
    {
        itemsSemaphore.p();
        mutexSemaphore.p();
        int ergebnis = data[head++];
        if(head == data.length)
        {
            head = 0;
```

```
        }
        mutexSemaphore.v();
        placesSemaphore.v();
        return ergebnis;
    }
}
```

Aufgabe 10.3

```
public class MsgQueueExample
{
    public static void main(String[] argv)
    {
        MessageQueue fromClientToServer =
                                    new MessageQueue(2);
        MessageQueue fromServerToClient =
                                    new MessageQueue(2);
        new MsgQueueClient(fromClientToServer,
                           fromServerToClient);
        new MsgQueueServer(fromClientToServer,
                           fromServerToClient);
    }
}

class MsgQueueClient extends Thread
{
    private MessageQueue fromClientToServer;
    private MessageQueue fromServerToClient;

    public MsgQueueClient(MessageQueue fromClientToServer,
                          MessageQueue fromServerToClient)
    {
        super("Client");
        this.fromClientToServer = fromClientToServer;
        this.fromServerToClient = fromServerToClient;
        start();
    }

    public void run()
    {
        while(true)
        {
            byte[] request = new byte[10];
            fromClientToServer.send(request);
            fromServerToClient.receive();
            try
            {
                sleep((int) (Math.random() * 4000));
            }
            catch(InterruptedException e)
            {
            }
        }
    }
```

```
}

class MsgQueueServer extends Thread
{
    private MessageQueue fromClientToServer;
    private MessageQueue fromServerToClient;

    public MsgQueueServer(MessageQueue fromClientToServer,
                          MessageQueue fromServerToClient)
    {
        super("Server");
        this.fromClientToServer = fromClientToServer;
        this.fromServerToClient = fromServerToClient;
        start();
    }

    public void run()
    {
        while(true)
        {
            byte[] request = fromClientToServer.receive();
            try
            {
                sleep((int) (Math.random() * 4000));
            }
            catch(InterruptedException e)
            {
            }
            byte[] response = request.clone();
            fromServerToClient.send(response);
        }
    }
}
```

Aufgabe 11.1

In Anlehnung an die Implementierung des fairen Parkhauses (s. Listing 2.31 im Lehrbuch):

```
abstract class AccessControlFair
{
    private int activeReaders = 0;
    private int activeWriters = 0;
    private int nextWaitingNumber = 0;
    private int nextEntryNumber = 0;

    protected abstract Object reallyRead();

    protected abstract void reallyWrite(Object obj);

    public Object read()
    {
        beforeRead();
        Object obj = reallyRead();
```

```
    afterRead();
    return obj;
}

public void write(Object obj)
{
    beforeWrite();
    reallyWrite(obj);
    afterWrite();
}

private synchronized void beforeRead()
{
    int myNumber = nextWaitingNumber++;
    while(myNumber != nextEntryNumber ||
          activeWriters > 0)
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    activeReaders++;
    nextEntryNumber++;
    notifyAll(); // wichtig !!!!
}

private synchronized void afterRead()
{
    activeReaders--;
    notifyAll();
}

private synchronized void beforeWrite()
{
    int myNumber = nextWaitingNumber++;
    while(myNumber != nextEntryNumber ||
          activeWriters > 0 ||
          activeReaders > 0)
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    activeWriters++;
    nextEntryNumber++;
}

private synchronized void afterWrite()
```

```
    {
        activeWriters--;
        notifyAll();
    }
}
```

Dass nach dem Lesen und nach dem Schreiben wartende Threads geweckt werden müssen, leuchtet sicher unmittelbar ein (deshalb der Aufruf von notifyAll in afterRead und afterWrite). Eventuell ist auf den ersten Blick aber nicht so klar, warum auch am Ende von beforeRead der Aufruf von notifyAll nötig ist. Dazu ein kleines Beispiel unter der Annahme, dass der Aufruf von notifyAll am Ende von beforeRead nicht da wäre:
Angenommen, zwei Leser-Threads müssen warten, weil ein Schreiber momentan aktiv ist. Am Ende seiner Schreibphase ruft der Schreiber die Methode afterWrite auf, in der mit notifyAll die beiden Leser geweckt werden. Nehmen wir nun an, dass zufällig der Thread mit der höheren Wartenummer zum Laufen kommt. Da er noch nicht an der Reihe ist, wartet er wieder. Nun ist der Thread mit der richtigen Wartenummer an der Reihe: Er kann die Methode beforeRead verlassen und mit dem Lesen beginnen. Nun könnte auch der zweite Leser-Thread mit dem Lesen beginnen, was aber nicht geschieht, da dieser Thread im Wartezustand ist. Weckt aber der erste Leser mit notifyAll den zweiten Leser auf, dann ist der eben beschriebene Ablauf nicht möglich.

Aufgabe 11.2

p	m4
v	m5
put	m6
get	m6
takeFork	m4
putFork	m5

Aufgabe 12.1

```
public class MyReentrantLock
{
    private Thread lockingThread;
    private int lockingCounter;

    public synchronized void lock()
    {
        while(lockingCounter > 0 &&
              Thread.currentThread() != lockingThread)
        {
```

```
                try
                {
                    wait();
                }
                catch(InterruptedException e)
                {
                }
            }
            lockingThread = Thread.currentThread();
            lockingCounter++;
        }

        public synchronized void unlock()
        {
            if(Thread.currentThread() != lockingThread)
            {
                throw new IllegalAccessError();
            }
            lockingCounter--;
            if(lockingCounter == 0)
            {
                lockingThread = null;
                notify();
            }
        }
    }
```

Aufgabe 12.2

```
    public class AdditiveSemaphoreLock
    {
        private int value;
        private ReentrantLock lock;
        private Condition condition;

        public AdditiveSemaphoreLock(int init)
        {
            if(init < 0)
            {
                init = 0;
            }
            this.value = init;
            lock = new ReentrantLock();
            condition = lock.newCondition();
        }

        public void p(int x)
        {
            if(x <= 0)
            {
                return;
            }

            lock.lock();
            try
```

```
        {
            while(value - x < 0)
            {
                condition.awaitUninterruptibly();
            }
            value -= x;
        }
        finally
        {
            lock.unlock();
        }
    }

    public void v(int x)
    {
        if(x <= 0)
        {
            return;
        }

        lock.lock();
        try
        {
            value += x;
            condition.signalAll();
        }
        finally
        {
            lock.unlock();
        }
    }
}
```

Aufgabe 12.3

```
import java.util.concurrent.CountDownLatch;

class DownCounter extends Thread
{
    private CountDownLatch latch;

    public DownCounter(String name,
                       CountDownLatch latch)
    {
        super(name);
        this.latch = latch;
    }

    public void run()
    {
        try
        {
            latch.await();
        }
```

```
            catch(InterruptedException e)
            {
            }
            System.out.println(getName()
                               + ": Warten beendet");
        }
    }

    public class CountDownDemo
    {
        public static void main(String[] args)
        {
            CountDownLatch latch = new CountDownLatch(5);
            for(int i = 1; i <= 5; i++)
            {
                DownCounter dc = new DownCounter("DownCounter"
                                                 + i, latch);
                dc.start();
            }
            for(int i = 1; i <= 5; i++)
            {
                try
                {
                    Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                }
                System.out.println("Zähle Latch herunter...");
                latch.countDown();
            }
        }
    }
```

Aufgabe 12.4

```
    public class MyCountDownLatch
    {
        private int counter;

        public MyCountDownLatch(int initValue)
        {
            if(initValue <= 0)
            {
                throw new IllegalArgumentException("...");
            }
            counter = initValue;
        }

        public synchronized void countDown()
        {
            counter--;
            notifyAll();
        }
```

```
    public synchronized void await()
           throws InterruptedException
    {
        while(counter > 0)
        {
            wait();
        }
    }
}
```

Die Methode countDown kann noch etwas verfeinert werden: zum einen muss der Zähler nur heruntergezählt werden, wenn er größer als 0 ist, zum anderen müssen wartende Threads nur benachrichtigt werden, wenn der Zähler auf 0 gegangen ist.

```
    public synchronized void countDown()
    {
        if(counter > 0)
        {
            counter--;
            if(counter == 0)
            {
                notifyAll();
            }
        }
    }
```

Aufgabe 13.1

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class Fibonacci extends RecursiveTask<Integer>
{
    private int n;

    public Fibonacci(int n)
    {
        if(n < 0)
        {
            throw new IllegalArgumentException("negativer " +
                                                "Parameter");
        }
        this.n = n;
    }

    @Override
    protected Integer compute()
    {
        if (n <= 1)
        {
            return n;
        }
```

```
        Fibonacci firstTask = new Fibonacci(n-1);
        Fibonacci secondTask = new Fibonacci(n-2);
        firstTask.fork();
        secondTask.fork();
        return firstTask.join() + secondTask.join();
    }

    public static void main(String[] args)
    {
        ForkJoinPool pool = new ForkJoinPool();
        for (int i=0; i <= 40; i++)
        {

            Fibonacci task = new Fibonacci(i);
            int result = pool.invoke(task);

            System.out.println("fib(" + i + ") = " + result);
        }

    }
}
```

Es ist von ganz großer Wichtigkeit, diese Implementierung wie folgt zu kommentieren: Das Prinzip der Rekursion kann an diesem Beispiel zur Berechnung der Fibonacci-Zahlen zwar gut erläutert werden, es ist aber in höchstem Maße ineffizient.

Aufgabe 13.2

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class QuickSort<E extends Comparable<E>>
       extends RecursiveAction
{
    private static final int SIZE = 100;

    private List<E> data;
    private int left;
    private int right;

    public QuickSort(List<E> data)
    {
        this.data=data;
        this.left = 0;
        this.right = data.size() - 1;
    }

    public QuickSort(List<E> data, int left, int right)
    {
        this.data = data;
        this.left = left;
```

```
    this.right = right;
}

@Override
protected void compute()
{
    if (left < right)
    {
        int pivotIndex = left + ((right - left)/2);
        pivotIndex = partition(pivotIndex);


        QuickSort<E> leftTask = new QuickSort<>(data, left,
                                                pivotIndex-1);
        QuickSort<E> rightTask = new QuickSort<>(data,
                                                 pivotIndex+1,
                                                 right);
        leftTask.fork();
        rightTask.fork();
        leftTask.join();
        rightTask.join();

        //oder einfacher:
        /*
        invokeAll(new QuickSort<>(data, left,
                                  pivotIndex-1),
                  new QuickSort<>(data, pivotIndex+1,
                                  right));
        */
    }
}

private int partition(int pivotIndex)
{
    E pivotValue = data.get(pivotIndex);

    swap(pivotIndex, right);

    int storeIndex = left;
    for (int i=left; i<right; i++)
    {
        if(data.get(i).compareTo(pivotValue) < 0)
        {
            swap(i, storeIndex);
            storeIndex++;
        }
    }

    swap(storeIndex, right);

    return storeIndex;
}

private void swap(int i, int j)
{
    if (i != j)
```

```
        {
            E iValue = data.get(i);

            data.set(i, data.get(j));
            data.set(j, iValue);
        }
    }

    public static void main(String[] args)
    {
        List<Integer> myList = new ArrayList<Integer>(SIZE);

        for (int i=0; i<SIZE; i++)
        {
            int value = (int) (Math.random() * 1000);
            myList.add(value);
        }

        System.out.println(myList);

        QuickSort<Integer> quickSort = new QuickSort<>(myList);
        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(quickSort);

        System.out.println(myList);
    }
}
```

Aufgabe 14.1

a)

```
 class Studi extends Thread
 {
     private Semaphore buch1, buch2, buch3;

     public Studi(Semaphore b1, Semaphore b2, Semaphore b3)
     {
         buch1 = b1;
         buch2 = b2;
         buch3 = b3;
     }

     public void run()
     {
         //1. Buch, dann 2. Buch, dann 3. Buch belegen
         buch1.p();
         buch2.p();
         buch3.p();

         //Seminararbeit erstellen
           (nicht näher interessant)
         ...

         //alle Bücher wieder zurücklegen
```

```
            buch1.v();
            buch2.v();
            buch3.v();
        }
    }
```

b)

```
public class Seminarbetrieb
{
    public static void main(String[] args)
    {
        Semaphore buch1, buch2, buch3;

        //Bücher erzeugen, 20 Studis erzeugen und starten
        buch1 = new Semaphore(4);
        buch2 = new Semaphore(2);
        buch3 = new Semaphore(5);

        for(int i = 0; i < 20; i++)
        {
            Studi s = new Studi(buch1, buch2, buch3);
            s.start();
        }
    }
}
```

c)
Eine Verklemmung ist ausgeschlossen, da alle Threads die Betriebsmittel in derselben Reihenfolge anfordern.

Aufgabe 14.2

```
    public void transferMoney(int fromAccountNumber,
                              int toAccountNumber,
                              float amount)
    {
        int lowerAccountNumber, higherAccountNumber;
        if(fromAccountNumber < toAccountNumber)
        {
            lowerAccountNumber = fromAccountNumber;
            higherAccountNumber = toAccountNumber;
        }
        else
        {
            lowerAccountNumber = toAccountNumber;
            higherAccountNumber = fromAccountNumber;
        }

        synchronized(account[lowerAccountNumber])
        {
            synchronized(account[higherAccountNumber])
            {
```

```
                account[fromAccountNumber].debitOrCredit(
                                                    -amount);
                account[toAccountNumber].debitOrCredit(
                                                     amount);
            }
        }
    }
```

Glossar

Message Queue

Ein nach dem Erzeuger-Verbraucher-Prinzip synchronisiertes Konzept zum Senden und Empfangen von Nachrichten. Im Gegensatz zu Pipes bleiben in Message Queues abgelegte Bytefolgen als Einheit erhalten; das heißt, dass beim Empfangen eine Bytefolge zurückgeliefert wird, die zuvor in genau diesem Umfang in einer Sendeoperation gesendet wurde.

Pipe

Ein nach dem Erzeuger-Verbraucher-Prinzip synchronisiertes Konzept zum Senden und Empfangen von Nachrichten. Im Gegensatz zu einer Message Queue werden die Bytes mehrerer Sendeoperationen in einer Pipe so hintereinander abgelegt, dass man die Grenzen zwischen den Bytefolgen unterschiedlicher Sendeoperationen nicht mehr erkennen kann. Beim Empfangen wird eine Folge von Bytes zurückgeliefert, die nur ein Teil einer gesendeten Bytefolge sein oder aus mehreren gesendeten Bytefolgen bestehen kann.

Prozess

Beim Starten eines Programms wird vom Betriebssystem ein neuer Prozess erzeugt, der mindestens eine Aktivität (d.h. einen Thread enthält). Der Prozess bildet einen eigenen Adressraum. Alle Threads können in der Regel nur Daten innerhalb dieses Adressraums ansprechen. Somit werden die Daten von Prozessen oder des Betriebssystems vor anderen Prozessen geschützt.

Semaphor (engl. Semaphore)

Semaphore sind ein „klassisches" Synchronisationskonzept. Mit Hilfe der Methoden p und v (manchmal auch down und up oder acquire und release) kann der Wert eines Zählers um 1 erniedrigt bzw. erhöht werden. Der Wert des Zählers darf allerdings nie negativ werden. Soll der Wert des Zählers erniedrigt werden, wenn er bereits 0 ist, dann wird der aufrufende Thread so lange blockiert, bis der Zähler größer als 0 ist, um ihn dann zu erniedrigen.

Thread

Ein Thread ist ein Aktivitätsträger. Jeder Prozess hat mindestens einen Thread. Umgekehrt ist ein Thread in unveränderbarer Weise nur einem Prozess zugeordnet. Die Threads, die zum selben Prozess gehören, teilen sich den Adressraum und können daher auf gemeinsamen Daten arbeiten.

Verklemmung

Eine Verklemmung ist eingetreten, wenn es einen Zyklus von Threads gibt, so dass ein Thread auf ein Betriebsmittel wartet, welches der im Zyklus folgende Thread besitzt. Betriebsmittel können z.B. Objekte sein, deren Zugriff mit synchronized, Semaphoren oder Locks synchronisiert wird. Voraussetzung für Verklemmungen sind neben dem zyklischen Warten, dass die Betriebsmittel nur unter gegenseitigem Ausschluss benutzbar sind, dass die Betriebsmittel einem Thread nicht entzogen werden können, und dass Threads bereits Betriebsmittel besitzen und weitere anfordern.

Stichwortverzeichnis

s. Kapitel „Index“ im Lehrbuch

Zentrum für Fernstudien
im Hochschulverbund

Eine Einrichtung der Bundesländer
Rheinland-Pfalz | Hessen | Saarland