Fraunhofer Institut
Software- und
Systemtechnik

Modellieren und Programmieren von nebenläufigen Prozessen

Beispiele in Java

Stefan Jaksch

Bericht 67/03
September 2003

Zusammenfassung

Seit dem historischen Besuch von mehreren, immer hungrigen Philosophen in einem Lokal mit zu wenig Eßbesteck sind nebenläufige Prozesse das Schicksal der Programmierer [4]. Wer nebenläufige Prozesse kennt, der weiß, daß jene eine Modellierung mögen und sich über Synchronisation freuen. Die gegenseitige Blockade hingegen ist das Todesurteil für die beteiligten Prozesse - und meist auch der Alptraum des Verursachers oder desjenigen, der die intern nebenläufig arbeitenden Komponenten verwendet.

Nach etlichen durchlebten Nächten mit Alpträumen versucht dieser Bericht, den Mythos über die nebenläufigen Prozesse zu erklären.

Im Mittelteil wird stellvertretend auf einige gut geeignete Modellierungswerkzeuge eingegangen. Zuvor findet ein kleiner Exkurs in die Theorie der Prozesse und deren Beschreibung und Synchronisation statt.

Am Ende schließen sich praktische Beispiele für Prozesse in der Programmiersprache Java an. Darin werden die wichtigsten Mechanismen zum Umgang mit den Prozessen vorgestellt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Bis auf den sogenannten Urlader, der *Gott* unter den Prozessen, werden alle vom Urlader gestarteten Prozesse in einer Ausführungsumgebung, die auch den Scheduler enthält, gestartet. Der Scheduler kann beispielsweise Bestandteil eines Betriebssystems (z.B. Unix, Windows und Echtzeitbetriebssysteme) oder einer speziellen Ausführungsumgebung (z.B. Java Virtual Machine (JVM)) sein. Die Aufgabe des Schedulers ist die Zuteilung der begrenzt vorhandenen Betriebsmittel, wie z.B. den Prozessor (Rechenzeit) oder Ein- und Ausgabegeräte, an die darum konkurrierenden Prozesse. Dafür nutzen sie unter Umständen verschiedene Strategien.

Die Tatsache, daß ein Algorithmus ein Ergebnis mit mehreren nebenläufigen Aktivitäten berechnet, beschert dem Algorithmus insbesondere zwei nicht sehr nützliche Eigenschaften. Nebenläufigkeit bedingt ohne besondere Maßnahmen Indeterminiertheit und Indeterminismus. Ersteres bezeichnet die Eigenschaft eines Algorithmus, bei mehreren Aufrufen mit den gleichen Parametern nicht dasselbe Ergebnis zu liefern, letzteres bezeichnet die zufällige Reihenfolge der nebenläufig ausgeführten Aktivitäten, die auch eine Ursache für die Indeterminiertheit ist.

Somit ist es sogar möglich, daß ein Wechsel des Schedulers eine dieser beiden Eigenschaften zur Folge hat, da die Scheduler verschiedene Strategien zur Auswahl des nächstes auszuführenden Prozesses anwenden. Indeterminiertheit und Indeterminismus können durch die Synchronisation der Kooperation und Konkurrenz der nebenläufigen Prozesse eliminiert werden.

Dieser Bericht gibt einen kurzen theoretischen Einblick in die Welt der Prozesse und einen Überblick über die verschiedenen Möglichkeiten zur Synchronisation derselben. Daran schließt sich eine Vorstellung von zwei Modellierungsmöglichkeiten für nebenläufige Prozesse an. Zum Schluß erläutert der Bericht die Erzeugung, Synchronisation und Terminierung von Prozessen in der Programmiersprache Java an Hand von Beispielen der häufig benötigten Prozesse.

Kapitel 2

Theoretische Grundlagen

Nebenläufige Prozesse treten auf verschiedenen Abstraktionsniveaus zutage. Von der Ebene der groben Beschreibung der Gesamtfunktionalität über die schrittweise Verfeinerung bis zur Modellierung eines Programm-Moduls sind Prozesse vorhanden. Zunächst geht dieses Kapitel auf die genaue Begriffsdefinition und Möglichkeiten zur Beschreibung von Prozessen ein. Im Anschluß daran wird die Verwaltung und Synchronisation von Prozessen skizziert.

2.1 Prozesse

Die Gesamtfunktionalität eines Programms wird durch viele Einzelschritte erbracht, die durch bestimmte Anweisungen und Eingaben gesteuert werden. Jeder dieser Schritte wird als **Aktivität** (*activity*) bezeichnet. Aktivitäten können aus sequentiellen oder nebenläufigen Teilaktivitäten bestehen. Es ist eine Frage der Abstraktion, wie man Aktivitäten voneinander abgrenzt. Die Abgrenzung kann man auf der Ebene von Assembler-Befehlen oder auf der Ebene von Anweisungen in einer höheren Programmiersprache ziehen. Für die weiteren Betrachtungen nutzen wir vorwiegend das letztere Abstraktionsniveau.

Das Programm besteht demnach aus einer Menge $\mathcal{M}$ von Aktivitäten. Die Aktivitäten müssen nun strukturiert werden, so daß ihre Abfolge verdeutlicht wird. Dazu wird der **Prozeß** (*process*) eingeführt. Er definiert eine sequentielle Folge von Aktivitäten, durch die eine definierte Aufgabe bearbeitet wird. Enthält ein Programm nebenläufige Anweisungen, hat seine Ausführung die Durchführung mehrerer Prozesse zur Folge, denn nebenläufige Aktivitäten sind zwangsweise verschiedenen Prozessen zugeordnet. Man nennt diese Prozesse, Aktivitäten und Anweisungen zueinander nebenläufig.

So, wie sich eine Aktivität gemäß der obigen Überlegungen aus mehreren nebenläufigen Teilaktivitäten zusammensetzen kann, so kann ein Prozeß, der eine solche Aktivität enthält, seinerseits aus nebenläufigen Prozessen bestehen.

Ein Prozeß entsteht durch die Ausführung von Anweisungen auf einem **Prozessor** (*processor*). Vielfach wird der Prozessor auch als Rechner bezeichnet. Prozesse sind in erster Linie virtuelle Objekte zur Strukturierung des Programms. Nicht jeder virtuelle Prozeß muß einen realen Prozeß bilden, das heißt, nicht für jeden Prozeß muß ein Prozessor vorhanden sein. Es reicht aus, sich für jeden Prozeß einen virtuellen Prozessor zu denken. Die Zuordnung von virtuellem zu realem Prozessor ist die Sache der Ausführungsumgebung, die den Programmcode ausführt. Darauf wird später eingegangen.

2.2 Prozeßbeschreibung

Für die Beschreibung nebenläufiger Prozesse in Programmsystemen gibt es verschiedene Sprachmittel. Diese Sprachmittel entwickelten sich aus der Idee der Aufspaltung des Kontrollflusses in einem (sequentiellen) Programm. Sie äußern sich demzufolge größtenteils in Anweisungen, die zur Erzeugung von Prozessen führen. Demgegenüber steht der jüngere Ansatz, Prozesse als eigenständige Programmelemente zu deklarieren. Dieser Abschnitt geht kurz auf beide Arten von Sprachmitteln ein.

2.2.1 Ablauforientierte Prozeßbeschreibung

Auf einem Prozessor existiert initial ein Prozeß, der aus den Anweisungen des Urladers, die der Prozessor automatisch nach seinem Start ausführt, resultiert. Weitere Prozesse entstehen dadurch, daß ein existierender Prozeß Aktivitäten durchführt, mit denen neue Prozesse ins Leben gerufen werden. Damit entstehen verwandtschaftliche Beziehungen zwischen Elternprozessen (*parent processes*) und Kindprozessen (*child processes*).

Nach dem Start eines Kindprozesses durch seinen Elternprozeß gibt es verschiedene Möglichkeiten, mit dem alten Prozeß fortzufahren. Die einzelnen Arten der Prozeßerzeugung unterscheidet man nach dem Verhältnis, das sie zwischen Eltern- und Kindprozeß schaffen. Folgende Möglichkeiten gibt es:

- Mit dem Aufruf des Kindprozesses terminiert der Elternprozeß. Die Prozesse lösen sich ab.
- Beim Aufruf des Kindprozesses wird der Elternprozeß in einen Wartezustand versetzt. Es gibt zwei Alternativen für das Fortführen des Elternprozesses: Entweder er wird von einem beliebig anderen Prozeß wieder aktiviert, oder er wird genau dann fortgesetzt, wenn seine Kindprozesse beendet sind.
- Eltern- und Kindprozeß arbeiten nach dem Start des Kindprozesses nebenläufig weiter.

Die ablauforientierte Prozeßbeschreibung verwendet zum Erzeugen und Warten auf Prozesse die Konstrukte "`forc`" und "`join`". Im Gegensatz dazu unterscheidet die Prozeßdeklaration mehrere Phasen.

2.2.2 Prozeßdeklaration

Eine Prozeßdeklaration bezeichnet ein Programmstück, dessen Ausführung zur Bildung eines nebenläufigen Prozesses führt. Bei der ablauforientierten Prozeßbeschreibung konnte man davon ausgehen, daß ein Prozeß durch die Ausführung einer Anweisung wie zum Beispiel `forc` begonnen wird. Bei der Prozeßdeklaration muß jedoch genauer unterschieden werden: Durch eine Prozeßdeklaration erfolgt zunächst lediglich eine Prozeßanmeldung (*process creation*). Sie macht den Prozeß als Objekt bekannt. An die Anmeldung können sich später der Prozeßstart (*process start*), das Prozeßende (*process termination*) und schließlich die Prozeßabmeldung (*process deletion*) anschließen.

Durch Prozeßdeklarationen wird neben der Prozeßstruktur eines Programmsystems auch die Zuordnung der von den Prozessen verwendeten Daten deutlicher als bei der ablauforientierten Prozeßbeschreibung. Für lokale Variablen (*local variables*), die nur ein Prozeß benutzen kann, sind nämlich keine Synchronisationen notwendig, um die Gefahr der Indeterminiertheit bei gemeinsamen Variablen (*shared variables*) zu vermeiden. Gemeinsame Variablen werden in einem Gültigkeitsbereich vereinbart, der mehrere Prozeßdeklarationen umschließt. Im Gegensatz dazu werden lokale Variablen in einer Prozeßdeklaration vereinbart, die selbst keine weiteren Prozeßdeklarationen enthält.

2.3 Prozeßabwicklung

Der vorangegangene Abschnitt zeigte, mit welchen programmiersprachlichen Mitteln Prozesse beschrieben werden können. Dieser Abschnitt beschäftigt sich nun mit der Verwaltung und Ausführung der Prozesse durch die Ausführungsumgebung.

Als bekannteste Ausführungsumgebung für Prozesse dürften die Betriebssysteme bekannt sein. Dennoch existieren einige spezielle Ausführungsumgebungen für bestimmte Programmcodes. So zum Beispiel existiert für die Programmiersprache Java die Java Virtual Machine (JVM), die den vom Java-Compiler erzeugten Byte-Code ausführt und eine eigene interne Prozeßabwicklung bereitstellt.

Eine Prozeßverwaltung hat die Aufgabe, Prozessen in geordneter Weise einen Prozessor zur Verfügung zu stellen. Um ausgeführt werden zu können, benötigt ein Prozeß nicht nur einen Prozessor, sondern unter Umständen auch Ein-

und Ausgabegeräte und den Zugriff auf den Hauptspeicher.

Alle Elemente einer Rechenanlage, die von einem Prozeß benutzt werden können, nennt man **Betriebsmittel** (*resources*). Ein Prozeß kann erst dann arbeiten, wenn er über alle notwendigen Betriebsmittel verfügt. All das, worauf ein Prozeß warten kann, kann ein Betriebsmittel darstellen. Auch Eingaben, die Prozesse von ihren Benutzern benötigen, sind in diesem Sinne ebenfalls Betriebsmittel.

Für das Erlangen von Betriebsmitteln kann man verschiedene Verfahren wählen: Entweder gibt es im System spezielle Prozesse, die als Zuteiler (*dispatcher*) für die Vergabe von Betriebsmitteln sorgen und einem anderen Prozeß Betriebsmittel implizit oder auf Antrag hin zuweisen, oder jeder Prozeß kann sich ein Betriebsmittel durch Abwicklung eines dazu vorgesehenen Algorithmus selbst beschaffen.

Für die Vergabe des Betriebsmittels Prozessor (CPU) sind zwei grundsätzliche Vorgehensweisen möglich. Die Vergabe übernimmt ein Disponent (*scheduler*), der einen Prozeß aus der Menge der ausführbaren Prozesse auswählt und ihm den Prozessor zuteilt. Das Verfahren der Zuteilung heißt **Multitasking**, von dem es zwei Ausprägungen gibt:

- **Cooperative Multitasking**
  Bei dieser Arbeitsweise des Disponenten wird jedem Prozeß der Prozessor solange zugeteilt, bis er ihn freiwillig wieder abgibt. Der Disponent wird einen arbeitenden Prozeß nicht selbständig unterbrechen. Dieses Verfahren nennt man kooperativ, da sich die Prozesse so untereinander koordinieren müssen, daß sich keine Verklemmungen ergeben. Sie können dadurch entstehen, daß ein Prozeß den Prozessor nicht mehr zurück gibt.

- **Preemptive Multitasking**
  Der Disponent entscheidet in diesem Verfahren, wie lange ein ausführbarer Prozeß die CPU nutzen darf, bevor ein anderer Prozeß diesen Prozessor erhält. Dem Disponenten ist es somit möglich, den laufenden Prozeß zu unterbrechen, um einen anderen Prozeß auszuführen. Ein Kriterium für die Zuteilung der CPU ist einfacherhalber die Nutzungszeit, das darauf basierende Verfahren wird auch als *time sharing* oder *time slicing* bezeichnet.

Die Ausführung von nebenläufigen Aktivitäten ist neben den während der Programmierung eingebauten Maßnahmen auch von der Art und Weise der Betriebsmittelzuweisung und des Multitasking abhängig. Aus diesem Grund sollte man bei der Programmierung von nebenläufigen Aktivitäten darauf achten, daß sie sich unabhängig von den Eigenschaften der Ausführungsumgebung gleich verhalten.

Bei den bei der Programmierung von nebenläufigen Prozessen eingebauten Maßnahmen handelt es sich vorrangig um die Synchronisation der Kooperation und Konkurrenz der Prozesse. Die verschiedenen Möglichkeiten werden im nächsten Abschnitt erläutert.

2.4 Synchronisation

Wie in den ersten Abschnitten gezeigt wurde, müssen sich nebenläufige Prozesse hin und wieder synchronisieren, damit sie ihre jeweiligen Arbeitsziele erreichen. Durch die Synchronisation wird die Unabhängigkeit der Abfolge von Aktivitäten verschiedener Prozesse eingeschränkt. Bestimmte Aktivitäten eines Prozesses werden mit bestimmten Aktivitäten anderer Prozesse zeitlich geordnet, indem der Beginn ihrer Ausführung verzögert wird.

Zur Synchronisation ist unter anderem die nachfolgend vorgestellte Auswahl von Verfahren und Konstrukten nutzbar. Die Nutzbarkeit eines Verfahrens ist in erster Linie von dessen Unterstützung durch den jeweils benutzten Compiler oder Interpreter abhängig.

2.4.1 Schloßvariablen

Mit Schloßvariablen läßt sich der Zugang zu kritischen Abschnitten synchronisieren. Kritische Abschnitte sind zum Beispiel Modifikationen an gemeinsam benutzten Variablen durch verschiedene Prozesse. Um den Zugang zu einem kritischen Abschnitt zu regeln, versieht man ihn mit einem Schloß, das von Prozessen

aufgesperrt und verriegelt werden kann. Programmtechnisch wird ein solches Schloß durch eine **Schloßvariable** (*locking variable*) realisiert. Eine Schloßvariable ist eine abstrakte Datenstruktur, auf die alle Prozesse zugreifen müssen, die kritische Abschnitte betreten wollen. Die Implementierungen ihrer Zugriffsprozeduren bezeichnen wir als **Schloßalgorithmen** (*lock algorithm*).

Beim Betreten eines kritischen Abschnitts geht ein Prozeß folgendermaßen vor: Der Prozeß wartet so lange, bis das zugehörige Schloß offen ist, dann betritt er den Abschnitt und verschließt das Schloß von innen, so daß kein anderer Prozeß folgen kann. Diese Operation nennt man *lock* (verschließen). Hat er den kritischen Abschnitt beendet, schließt der Prozeß das Schloß wieder auf. Diese Operation ist unter der Bezeichnung *unlock* bekannt.

Wie man *lock* und *unlock* implementieren kann, wird in mehreren Variationen in [3] beschrieben. Es gibt die Möglichkeit, Schloßalgorithmen mit unteilbaren Operationen[1] oder ohne solche speziellen Operatoren zu realisieren.

Das Warten eines Prozesses zum Betreten eines kritischen Bereichs kann durch die folgende Schleife

while not *Bedingung* repeat

–nichts

end repeat

implementiert werden. Wartet ein Prozeß auf das Eintreten dieser Bedingung, so geschieht dies mit aktiver Nutzung des Prozessors, indem immer wieder die Werte von Variablen abgefragt werden. Diesen Vorgang bezeichnet man als **aktives Warten** (*busy waiting*). Die Schloßalgorithmen würden mit cooperative Multitasking nicht ausführbar sein.

Eine weitere Unzulänglichkeit ist die Forderung, daß Prozesse nicht im kritischen Abschnitt terminieren dürfen, da ansonsten die Schloßvariable nicht mehr entsperrt werden könnte. Hierfür ist eine Kontrolle durch die Ausführungsumgebung unerläßlich. Ebenso muß der Übersetzer prüfen, ob in einem Programm für jede *lock*-Operation auch ein entsprechendes *unlock* notiert wurde. Sollte die Paarung nämlich einmal nicht stimmen, würde die Synchronisation der Prozesse durcheinander geraten.

Auf Grund dieser gravierenden Nachteile und potentiellen Gefahren sollten Schloßvariablen ohne Beschränkung der Allgemeinheit nicht zur Synchronisation verwendet werden.

2.4.2 Semaphore

Die im vorigen Abschnitt beschriebenen Synchronisationsprozeduren sind auf Einprozessorsystemen nicht sinnvoll verwendbar, da ein Prozeß aktiv auf den Eintritt in einen kritischen Bereich warten muß. Vielmehr sollten diejenigen Prozesse, die auf das Eintreten einer Bedingung warten müssen, von der Nutzung des Prozessors auch ausgeschlossen werden. Mit dem hier vorgestellten Verfahren ist dies möglich.

Semaphore (*semaphores*) sind der Bedeutung des Wortes nach Signale, die vor der Einführung des Telegraphendienstes vor allem in der Seefahrt zur optischen Übermittlung von Signalen über weite Entfernungen verwendet wurden. Dijkstra hat den Begriff Semaphor gewählt, um ein Hilfsmittel zur Synchronisation nebenläufiger Prozesse zu bezeichnen, mit dem das aktive Warten eines Prozesses vermieden wird [3]. Im Synchronisationsfall wird ein Prozeß blockiert und in eine Warteschlange eingeordnet. Bei Eintritt des Ereignisses, auf das er wartet, wird er aus der Warteschlange entfernt.

Ein Semaphor ist ein abstrakter Datentyp, dessen Objekte aus einem Zähler und einer Warteschlange bestehen. Er definiert zwei Operationen, die traditionell mit P und V bezeichnet werden. Diese Kürzel stammen aus dem Holländischen und stehen für *passeeren* (passieren) und *vrijgeven* (freigeben, im Deutschen auch "verlassen", um die Mnemonik[2] nachzubilden).

[1] Unteilbare Operationen werden manchmal auch als "atomar" bezeichnet. Sie sind spezielle Maschineninstruktionen, die nicht unterbrechbar sind. So liest zum Beispiel der Befehl `test_and_set (a.b)` den Wert der Boolschen Variablen `a`, kopiert ihn nach `b` und setzt `a` auf `true` in einer unteilbaren Aktivität.

[2] Eselsbrücke

Die Operationen arbeiten folgendermaßen: Mit jedem *P*-Aufruf wird der Wert des Semaphorzählers um eins erniedrigt. Erreicht er dabei einen negativen Wert, wird der die Operation ausführende Prozeß in die Warteschlange des Semaphors eingeordnet und blockiert. Mit dem *V*-Aufruf wird der Wert des Semaphorzählers um eins erhöht. Sollte der Zahlenwert dann 0 oder negativ sein, wird der erste Prozeß aus der Warteschlange des Semaphors herausgeholt und in die Warteschlange der ausführungsbereiten Prozesse eingereiht.

Die Operation *P* dient zum Warten auf das Eintreffen einer Bedingung. Das Eintreten einer Bedingung wird dargestellt durch einen positiven Wert des Semaphorzählers. Mit der *V*-Operation wird das Eintreten einer Bedingung angezeigt, indem der Zählerwert erhöht wird. Mit Semaphoren lassen sich die mehrseitige und die einseitige Synchronisation implementieren.

Im Falle der mehrseitigen Synchronisation gibt der Initialwert des Semaphorzählers die Anzahl der Prozesse an, die maximal zu einer Zeit im kritischen Abschnitt arbeiten können. *P* und *V* umschließen den kritischen Abschnitt genau wie *lock* und *unlock* im Abschnitt 2.4.1. Auch hier existieren bestimmte Ein- und Austrittsprotokolle, zu jeder *P*-Operation muß der zugehörige *V*-Aufruf im Programm enthalten sein.

```
s : semaphore(1);
P1 : process          P2 : process
   ...                   ...
    s→P                   s→P
      critical              critical
    s→V                   s→V
   ...                   ...
end process           end process
```

Darf ein kritischer Abschnitt nur exklusiv betreten werden, hat der Semaphorzähler den Initialwert 1. Auf diese Weise verwendete Semaphore nennt man hin und wieder auch **ausschließende Semaphore** (*excluding semaphores*).

Semaphore lassen sich auch zur einseitigen Synchronisation verwenden. Jeder Synchronisationsbedingung wird dann ein Semaphor zugeordnet. Bei einer einseitigen Synchronisation stehen einander zugeordnete Semaphoroperationen in verschiedenen Prozessen. Mit einer *P*-Operation wartet ein Prozeß darauf, daß ihm ein anderer Prozeß mit einer *V*-Operation das Eintreffen einer Bedingung anzeigt. Wenn diese Bedingung zu Beginn noch nicht gilt, hat das Semaphor den Initialwert 0.

```
s : semaphore(0);
P1 : process          P2 : process
   ...                   ...
    s→V                   s→P
   ...                   ...
end process           end process
```

Solange das erwartete Ereignis nicht eingetreten ist, wird der Prozeß blockiert. Er kann also nicht durch aktives Warten unnütz den Prozessor belegen. Man bezeichnet den so angewandten Semaphor auch als **signalisierenden Semaphor** (*signaling semaphores*), um sie von der mehrseitigen Synchronisation zu unterscheiden.

Da die Semaphoroperationen, also alle Aktionen, die mit der Blockierung und der Verwaltung der Warteschlange in Zusammenhang stehen, den Zustand von Prozessen verändern, handelt es sich bei ihnen um Operationen der Prozeßverwaltung. Diese ist Bestandteil der Ausführungsumgebung und wurde in Abschnitt 2.3 beschrieben. Oft sind in Ausführungsumgebungen statt dem allgemeinen Semaphor, dessen Zählervariable beliebige Werte annehmen kann, nur **binäre Semaphore** (*binary semaphores*) implementiert. Man benötigt dann zur Darstellung des Zählers nämlich nur ein Bit.

Für die Verwendung von Semaphoren zur Lösung verschiedener Probleme, wie zum Beispiel von Produzenten und Konsumenten, Lesern und Schreibern gibt es in [9] Beispiele für deren Implementierung. Ferner wird dort auf zwei Erweiterungen von Semaphoren eingegangen, und zwar auf die additive Semaphore und auf Mehrfach-Semaphoroperationen. Die Sprache PEARL stellt eine eingeschränkte Variante der additiven Semaphore in Form von Riegeln (*bolts*) bereit.

Zudem lassen sich verschiedene, in der Literatur formulierte Probleme mit Semaphoren lösen. Für das weitere Studium seien hier stellvertretend die **speisenden Philosophen** (*dining philosophers problem*), der **schlafende Barbier** (*sleeping barbers problem*) in [3] und das **Zigarettenraucher-Problem** (*cigarette smokers problem*) in [12] genannt.

2.4.3 Monitore

Als einen Nachteil der bedingten kritischen Abschnitte, die in den vorangegangenen Beispielen beschrieben wurden, kann man betrachten, daß kritische Abschnitte mit ihren Synchronisationsanweisungen über das gesamte Programm verstreut sind. Man kann davon ausgehen, daß kritische Abschnitte einer Klasse stets logisch zusammenhängende gemeinsame Variablen mehrerer Prozesse schützen. Statt Synchronisationsmaßnahmen über mehrere Prozesse zu verteilen, liegt es nahe, sie in einer abstrakten Datenstruktur zusammenzufassen und so bei jedem Zugriff automatisch den gegenseitigen Ausschluß zu garantieren. Dies ist das in groben Zügen geschilderte Konzept der Monitore.

Monitore (*monitors*) sind das am weitesten verbreitete auf Semaphoren basierende Synchronisationsmittel für den Zugriff auf gemeinsame Variablen. Ein Monitor ist eine abstrakte Datenstruktur mit impliziten Synchronisationseigenschaften, die ihre Verwendung in einem nebenläufigen System erlauben: Den Benutzern eines Monitors bleibt nicht nur die Implementierung der Zugriffsfunktionen auf gemeinsame Daten, sondern auch die Realisierung des gegenseitigen Ausschlusses für diese Funktionen verborgen.

Alle Zugriffsfunktionen werden im gegenseitigen Ausschluß durchgeführt. Die im Monitor vereinbarten Variablen können auf Grund dieses gegenseitigen Ausschlusses nie nebenläufig benutzt werden; sie stellen exklusive Betriebsmittel für Prozesse dar. Durch den gegenseitigen Ausschluß wird eine mehrseitige Synchronisation realisiert.

Zum Monitor gehört eine Warteschlange, in die alle Prozesse, die Monitoroperationen aufrufen, eingereiht werden, sollte der Monitor gerade mit der Ausführung einer Operation belegt sein. Prozesse werden in der Reihenfolge ihrer Eintragung in die Warteschlange abgearbeitet.

Die einseitige Synchronisation innerhalb des Monitors kann auf verschiedene Arten geregelt werden. Durchgesetzt hat sich die Verwendung von **Ereignisvariablen** (*event variables*), für die zwei Operationen *signal* und *wait* definiert sind. Durch *signal* wird das Eintreten eines Ereignisses oder einer Bedingung angezeigt. Durch *wait* wird auf ein durch *signal* anzuzeigendes Ereignis gewartet.

Ein Prozeß, der *wait* auf einer Ereignisvariablen e ausführt, wird so lange blockiert, bis ein anderer Prozeß *signal* auf e ausführt. Der blockierte Prozeß wird hierbei in eine der Ereignisvariablen e zugeordneten Warteschlange eingereiht.

Signale werden nicht gespeichert: Wartet gerade kein Prozeß auf ein Signal, so bleibt der Aufruf von *signal* ohne Wirkung. Der Grund dafür liegt auf der Hand: Bis ein anderer Prozeß ein ausgelöstes Signal abnimmt, kann die von diesem Signal angezeigte Bedingung schon lange nicht mehr gelten. Würde ein wartender Prozeß im Monitor bleiben, könnte ein signalisierender Prozeß den Monitor nicht betreten. Um diese Verklemmung auszuschließen, wird der Monitor von einem Prozeß automatisch freigegeben, sobald er durch eine *wait*-Operation blockiert wird.

Monitore bilden lediglich eine syntaktische Abstraktion von Semaphoren und können daher mit Hilfe von Semaphoren implementiert werden. Einige Programmiersprachen bieten auch schon eine Reihe von Schlüsselwörtern an, mit denen sich Monitore realisieren lassen. Beispiele in der Programmiersprache Java sind in Abschnitt 4 zu finden.

Neben den hier vorgestellten Synchronisationsverfahren wird in [9] noch auf den letzten Schritt der Abstraktion von Semaphoren, den **Pfadausdrücken** (*path expressions*) eingegangen. Bei den Monitoren wurden die Operationen zur Synchronisation vor dem Benutzern versteckt, synchronisiert wird nur im Monitor: Die mehrseitige Synchronisation erfolgt

durch den gegenseitigen Ausschluß der Monitorprozeduren und die einseitige Synchronisation wird durch Ereignisvariablen an verschiedenen Stellen vorgenommen. Die konsequente Fortführung des Monitoransatzes besteht darin, auch die einseitige Synchronisation aus den Zugriffsoperationen herauszuziehen. Der Vorteil besteht darin, daß sämtliche Anweisungen zur Synchronisation übersichtlich an einer Stelle zusammengefaßt sind.

2.5 Verklemmungen

Durch Synchronisationsmaßnahmen werden Prozesse verzögert. Irgendwann soll der Grund für die Verzögerung wegfallen, und die Prozesse sollen weiterarbeiten können. Leider können Prozesse selbst verhindern, daß sie fortgesetzt werden. Das folgende Beispiel verdeutlicht dies:

Zwei Schreiber wollen gleichzeitig eine Notiz aufschreiben. Ihnen stehen allerdings nur ein Blatt und ein Bleistift zur Verfügung. Ein Schreiber ist schneller und sichert sich den Stift, der andere sichert sich schonmal das Blatt Papier. Beide sind nicht bereit, die Utensilien wieder zurückzulegen und warten unendlich auf das jeweils andere notwendige Utensil.

Eine solche Situation, in der Prozesse auf ein Ereignis warten, das nicht mehr eintreten kann, wird als **Verklemmung** (*deadlock*) bezeichnet.

2.5.1 Ursachen

Im obigen Beispiel entstand die Verklemmung aus einem Streit um Betriebsmittel. Da ein Betriebsmittel in Abschnitt 2.3 als all jenes definiert wurde, auf das ein Prozeß warten kann, ist auch ein kritischer Abschnitt ein Betriebsmittel.

Es läßt sich zeigen, daß Verklemmungen beim Streit um Betriebsmittel nur auftreten können, wenn die folgenden vier notwendigen und hinreichenden Bedingungen erfüllt sind [2].

1. Die umstrittenen Betriebsmittel sind nur exklusiv nutzbar.

2. Die umstrittenen Betriebsmittel können nicht entzogen werden.

3. Die Prozesse belegen die schon zugewiesenen Betriebsmittel auch dann, wenn sie auf die Zuweisung weiterer Betriebsmittel warten.

4. Es gibt eine zyklische Kette von Prozessen, von denen jeder mindestens ein Betriebsmittel besitzt, das der nächste Prozeß in der Kette benötigt.

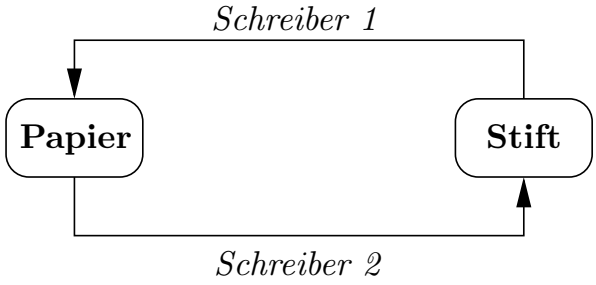


Abbildung 2.1: Verklemmung der beiden *Schreiber*-Prozesse, die gegenseitig auf Papier und Stift warten.

Die letzte Bedingung 4 läßt sich anschaulich in Form eines **Betriebsmittelgraphen** (*resource graph*) darstellen, wie ihn Abbildung 2.1 für das Beispiel des obigen Schreibvorgangs zeigt. Ein Knoten stellt hierbei ein Betriebsmittel und eine gerichtete Kante einen Prozeß dar, der ein Betriebsmittel besitzt (Beginn der Kante) und ein anderes anfordert (Ende der Kante).

2.5.2 Vermeidung

Dem Verklemmungsproblem kann man auf drei Arten begegnen:

1. Man kann versuchen, die Verklemmung zu vermeiden, indem man darauf achtet, daß immer mindestens eine der obigen vier Bedingungen nicht erfüllt ist.

2. Man kann versuchen, die Verklemmung zu vermeiden, indem man die zukünftigen Betriebsmittelanforderungen der Prozesse analysiert und Zustände verbietet, die zu Verklemmungen führen können.

3. Man kann versuchen, eine Verklemmung festzustellen und sie, sollte sie eingetreten sein, dann beseitigen.

Die einzelnen Punkte werden in den folgenden Abschnitten genauer betrachtet.

Punkt 1: Regeln Werden für die Vergabe von Betriebsmitteln Regeln festgelegt, die dafür sorgen, daß mindestens eine der genannten Bedingungen für eine Verklemmung nicht erfüllt ist, kann keine Verklemmung auftreten. Am Beispiel der Schreibenden lassen sich solche Regeln finden:

- Man erlaubt, daß mehrere Schreiber ein Schreibutensil gemeinsam benutzen. Dies funktioniert nicht immer, da es Betriebsmittel gibt, die prinzipiell nicht gemeinsam benutzbar sind.
- Man legt fest, daß alle Schreibutensilien zurückgelegt werden, sobald ein bereits reserviertes Utensil angefordert wird. Bei Prozessen ist diese Regel nicht anwendbar, wenn die mögliche Abgabe des Betriebsmittels dem Ziel des Prozesses entgegensteht: Ein Drucker sollte nicht entzogen werden, wenn erst die Hälfte des Textes ausgedruckt wurde.
- Man definiert eine Regel, daß bei einem Aufnahmevorgang stets alle zum Schreiben notwendigen Utensilien ausgeliehen werden müssen. Weitere Anforderungen vor der Rückgabe der Utensilien sind nicht zulässig.
- Man definiert eine Ordnung der Betriebsmittel. Fortan darf das Papier nur noch vor dem Stift aufgenommen werden. Die Ordnung der Betriebsmittel könnte das Problem ergeben, daß Betriebsmittel nicht mehr in der vom Prozeßablauf her natürlichen Reihenfolge angefordert werden dürfen.

Nicht für jedes Verklemmungsproblem lassen sich solche Regeln finden. Ferner muß man für jedes Problem neue Betrachtungen zur Verklemmungsvermeidung anstellen. Aus diesen Gründen benötigt man allgemeinere Algorithmen, wie zum Beispiel die im Anschluß vorgestellte Bedarfsanalyse.

Punkt 2: Bedarfsanalyse Prinzipiell wünscht man sich zur Verklemmungsvermeidung einen Algorithmus, der bei der Anforderung von Betriebsmitteln feststellt, ob ihre Zuteilung ohne die Gefahr von Verklemmungen möglich ist. Ein solcher Algorithmus ist durchaus nicht einfach, da eine Verklemmung nicht unmittelbar nach der Zuteilung auftreten muß. Abbildung 2.2 zeigt dies für zwei Prozesse und zwei Betriebsmittel.

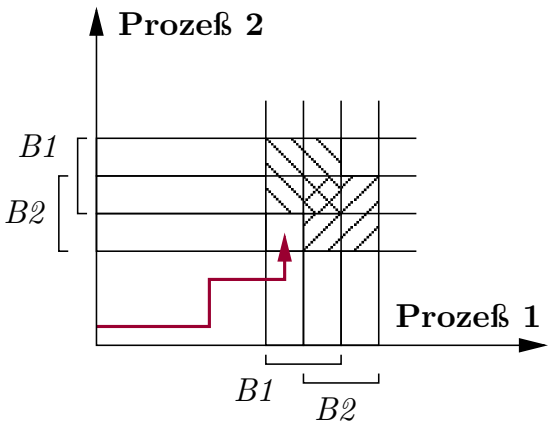


Abbildung 2.2: Bewegung von Prozessen auf einen Verklemmungszustand hin. Die schraffierten Flächen markieren die Verklemmungszustände.

In Abbildung 2.2 ist der Ablauf von Prozeß `P1` nach rechts und der Ablauf von Prozeß `P2` nach oben aufgetragen. Die Ausführung von `P1` bei gleichzeitiger Blockade von `P2` wird durch eine waagerechte Linie und entsprechend die Ausführung von `P2` bei gleichzeitiger Blockade von `P1` durch eine senkrechte Linie dargestellt. Prozeß `P1` benötigt zuerst Betriebsmittel `B1` und dann `B2`, wobei für kurze Zeit beide benötigt werden. Entsprechend benötigte Prozeß `P2` zuerst Betriebsmittel `B2`, dann `B1` und für kurze Zeit ebenfalls beide. Solche Anforderung kann laut den angegebenen Bedingungen zu einer Verklemmung führen.

Eine Verklemmung zeigt sich in Abbildung 2.2 graphisch, indem die schraffierten Bereiche betreten werden. Die Prozesse geraten in eine zwangsweise zur Verklemmung führende Situation, wenn sie ihren Weg weder nach rechts (`P1`) noch nach oben (`P2`) fortsetzen können, ohne in einen schraffierten Bereich zu gelangen.

Bevor die beiden Prozesse jedoch ihre jeweiligen zur Verklemmung führenden Anforderungen stellen, können sie in einem unvermeidlich

zur Verklemmung führenden Zustand noch ungehindert arbeiten.

Hier zeigt sich die Schwierigkeit der Verklemmungserkennung: Man muß vorausschauend prüfen. Dazu sind allerdings Kenntnisse über die künftigen Anforderungen von Prozessen notwendig.

Ein Algorithmus, mit dem es möglich ist, Verklemmungen zu vermeiden, geht auf Dijkstra [3] zurück. Der Algorithmus geht vereinfachend davon aus, daß Prozesse nur eine Art von Betriebsmitteln benötigen. Weiter wird vorausgesetzt, daß die maximalen Betriebsmittelanforderungen jedes Prozesses im voraus bekannt sind. Die anschauliche Darstellung dieses Verfahrens ist in Form des **Bankiersalgorithmus** (*banker's algorithm*) bekannt.

Bei der Anwendung des Algorithmus in der Praxis muß berücksichtigt werden, daß Prozesse gewöhnlicherweise Betriebsmittel unterschiedlicher Art verwenden. Problematischer als die Einschränkung auf eine Betriebsmittelart ist beim Bankiersalgorithmus die Voraussetzung, daß der Maximalbedarf eines Prozesses von vornherein bekannt sein muß. Da dieser Maximalbedarf bei der Vergabe Berücksichtigung findet, wird immer der ungünstigste Fall angenommen, der praktisch vielleicht gar nicht auftritt.

Zudem besitzt der Algorithmus einen quadratischen Aufwand ($\mathcal{O}(n^2)$), der von der Anzahl der Prozesse abhängt. Diesen Zeitaufwand wird man zumindest in Echtzeitsystemen oft nicht investieren können.

Punkt 3: Erkennung Eine in der Praxis häufig angewandte Strategie besteht darin, Verklemmungen in Kauf zu nehmen, sie zu erkennen und danach zu beseitigen. Ein Indiz für eine Verklemmung kann zum Beispiel sein, daß ein angefordertes Betriebsmittel nach einer gewissen Zeit noch nicht zugewiesen wurde. Mit einer Zeitüberwachung von Betriebsmittelanforderungen kann man dafür sorgen, daß nach einer gewissen Zeit ein Algorithmus gestartet wird, der untersucht, ob wirklich eine Verklemmung eingetreten ist.

Das Erkennen einer Verklemmung ist der erste Schritt zu ihrer Beseitigung und läßt sich algorithmisch auf die Suche eines geschlossenen Kreises im Betriebsmittelgraphen zurückführen, wie er schon in Abbildung 2.1 gezeigt wurde.

Zur Beseitigung einer Verklemmung muß wenigstens einer der an der Verklemmung beteiligten Prozesse abgebrochen werden, damit ihm das Betriebsmittel entzogen werden kann. Der Abbruch von Prozessen ist eine ziemlich radikale Maßnahme und bedeutet unter Umständen eine erhebliche Vergeudung von Rechenzeit. Aus diesem Grund ist es sinnvoll, die beteiligten Prozesse nur auf bestimmte vorher festgelegte **Rücksetzpunkte** (*checkpoints*) zurückzusetzen, von wo aus sie fortgesetzt werden. Diese Technik wird häufig in Datenbanken [11] eingesetzt.

2.6 Zusammenfassung

Prozesse sind die Grundlage für die meisten Abläufe in einer Rechnerarchitektur und von darauf ausgeführten Programmen. Dieser Abschnitt erklärte die Definition, Verwaltung und Synchronisationsverfahren für Prozesse. Diese Informationen sind hilfreich, wenn es darum geht, selbst nebenläufige Programme zu schreiben. Für die vorangehende Konzeptionsphase gibt es verschiedene Modellierungswerkzeuge. Auf speziell für nebenläufige Prozesse abgestimmte Werkzeuge geht der nächste Abschnitt ein.

Kapitel 3

Modellierung nebenläufiger Prozesse

Die Programmierung von nebenläufigen Prozessen ist eine Entwurfsentscheidung, die verschiedene Ursachen hat - auch solche, auf die man als Programmierer nicht immer Einfluß hat. Einerseits können in den Phasen des Software-Entwicklungsprozesses, zum Beispiel bei der Definition der Funktionalität oder der Verfeinerung von Komponenten, nebenläufige Prozesse identifiziert werden. Andererseits kann der Fall eintreten, daß die Entscheidung für die Benutzung einer Zusatzfunktionalität nebenläufige Prozesse einbringt, so daß man den eigenen Programmcode dazu vorbereiten muß. Letzteres ist der Fall, wenn der eigene Java-Programmcode zum Beispiel als RMI[1]-Server agiert.

Welche Ursache die Entscheidung für die Verwendung von nebenläufigen Prozessen auch hat - ein besonderes Augenmerk sollte auf der Konzeption und Dokumentation des entstehenden Programms liegen. Die Dokumente erleichtern die Fehlersuche und Wartung erheblich.

Zur Modellierung von nebenläufigen Prozessen existieren neben dem altbewährten Mittel Papier und Bleistift auch computergestützte Werkzeuge. Zwei dieser Werkzeuge, ein Petri-Netz-Editor mit Netz-Debugger und ein Tool zum Erstellen und Testen von Prozessen, die SDL[2]-konform beschrieben und damit eher datenflußorientiert sind, werden in diesem Abschnitt vorgestellt.

[1] Remote Method Invocation (RMI): Der RMI-Server muß die Funktionsaufrufe der RMI-Clients bearbeiten. In der Regel wird für jeden aufrufenden Client ein eigener Prozeß gestartet.

[2] Service Description Language

3.1 Petri-Netze

Nebenläufige Systeme und deren Synchronisation lassen sich anschaulich mit Petri-Netzen (*Petri nets*) modellieren. Seit ihrer ersten Veröffentlichung [13] wurden zahlreiche Varianten dieser Netze entwickelt.

Prinzipiell besitzen Petri-Netze eine wohldefinierte mathematische Struktur, die in verschiedener Weise interpretiert werden kann. So können bei der Modellierung nebenläufiger Systeme durch Petri-Netze formal Netzeigenschaften nachgewiesen werden, die sich dann als Systemeigenschaften interpretieren lassen. Verschiedene Varianten von Netzen erlauben hierbei unterschiedliche Aussagen über Systemeigenschaften. Stellvertretend seien hier kurz die Varianten Kanal-Instanz-Netze, Prädikats-Transitions-Netze und Petri-Netze mit unterscheidbaren Marken erwähnt [9].

Zunächst wird auf die Elemente und einige wichtige Eigenschaften von Petri-Netzen eingegangen. Am Ende wird das Tool zur Erstellung und Simulation von Netzen anhand von zwei Beispielnetzen erläutert.

3.1.1 Elemente und Eigenschaften

Petri-Netze bilden einen gerichteten Graphen mit zwei disjunkten Mengen von Knoten. Die Knoten werden üblicherweise als **Stellen** $\mathcal{S}$ (*places*) und **Transitionen** $\mathcal{T}$ (*transitions*) bezeichnet. Graphisch werden Stellen als Kreise und Transitionen als Rechtecke (manchmal auch als Striche) dargestellt. Die gerichteten Kanten führen jeweils von einem Element der einen zu einem Element der anderen Knotenmenge. Die Menge der Kanten des Graphen

nennt man **Flußrelation** $\mathcal{F}$ (*flow relation*). Die Struktur eines Petri-Netzes läßt sich leicht formal beschreiben: Ein Tripel $(\mathcal{S}, \mathcal{T}, \mathcal{F})$ heißt Petri-Netz, wenn folgende Bedingungen gelten:

1. $\mathcal{S} \cap \mathcal{T} = \emptyset$
   Die Mengen der Stellen und Transitionen sind disjunkt.

2. $\mathcal{S} \cup \mathcal{T} \neq \emptyset$
   Die Knotenmenge enthält mindestens eine Stelle oder Transition.

3. $\mathcal{F} \subseteq (\mathcal{S} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{S})$
   Die Kantenmenge enthält nur Elemente, die Knoten verschiedener Mengen (Stellen oder Transitionen) miteinander verbinden.

Mit diesen Grundlagen ist es bereits möglich, ein einfaches Petri-Netz zu erstellen. Eine mögliche Interpretation stellen die **Kanal-Instanz-Netze** (*channel agency nets*) dar. Eine Stelle wird als **Kanal** interpretiert. Unter einem solchen Kanal kann man sich eine passive Systemkomponente vorstellen, in der Informationen oder Material abgelegt werden können, zum Beispiel einen Puffer, ein Lager oder einen Schreibtisch. Eine Transition wird als **Instanz** interpretiert. Instanzen repräsentieren aktive Komponenten, die Informationen oder Material verarbeiten, zum Beispiel einen Prozeß als Produzenten, einen Roboter oder Sachbearbeiter. Instanzen kommunizieren über Kanäle, so daß zum Beispiel Informations- oder Materialflüsse in einem System beschrieben werden können.

Insbesondere bei der Analyse komplexer Systeme, die automatisiert werden sollen, sind solche Netze nützlich. Betrachten wir dies am Beispiel einer einfachen Materialverwaltung, wie sie in Abbildung 3.1 dargestellt ist.

Die Bestellungen kommen zur Bestellaufnahme. Durch die Bestellaufnahme werden Lieferaufträge an die Auslieferung erteilt. Diese bedient sich aus einem Lager und liefert die Waren aus. Um ständig Produkte auf Lager zu haben, muß bei der Bestellung eines Produkts ein neuer Produktionsauftrag erteilt werden. Das Produkt, das ein Kunde bekommt, muß jedoch nicht das sein, welches auf Grund seines Auftrags neu produziert wird - im Lager können ja noch ausreichend Waren vorhanden sein.

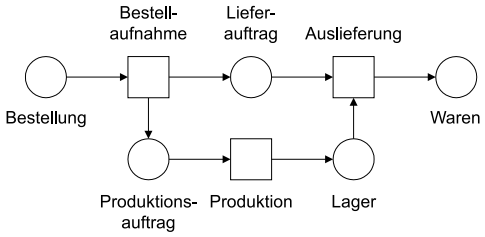


Abbildung 3.1: Darstellung einer Materialverwaltung mit einem Kanal-Instanz-Netz

Die Modellierung der Materialverwaltung ist noch recht grob. Man kann sich vorstellen, daß hierfür eine genauere Darstellung möglich ist, die natürlich zu einer größeren Anzahl von Knoten und Kanten im Netz führt. Bei realen Systemen erhält man rasch riesige Netze, die nicht mehr überschaubar sind. Dies ist der Grund für die Einführung einer hierarchischen Verfeinerung von Knoten.

Die Verfeinerung von Knoten erhält man durch strukturerhaltende Abbildungen, denn nur so können beim schrittweisen Entwurf des Petri-Netzes formale Fehler vermieden werden. Die strukturerhaltende Abbildung wird als **Netzmorphismus** (*net morphism*) bezeichnet. Ein Beispiel für die verfeinerte Instanz "Auslieferung" der Materialverwaltung aus Abbildung 3.1 wird in Abbildung 3.2 gezeigt.

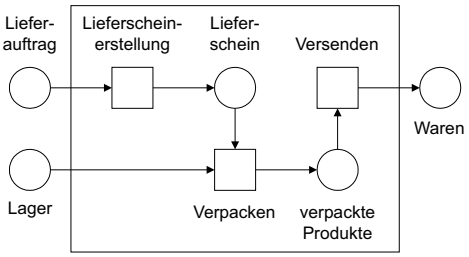


Abbildung 3.2: Materialverwaltung - Verfeinerung der Auslieferung

Um nun neben den eher statischen Beziehungen zwischen Stellen und Transitionen auch die dynamischen Eigenschaften eines Systems zu modellieren, führt man eine veränderliche **Markierung** (*marking*) der Stellen ein. Dabei bestimmen die **Schaltregeln** (*firing rules*), wie eine Markierung in eine Nachfolgemarkierung übergeht. Verschiedene Knoteninterpreta-

tionen, Markierungsvorschriften und Schaltregeln führen zu einer Vielzahl möglicher Netzinterpretationen, von denen hier eine Zusammenfassung der wichtigsten Elemente und Regeln wiedergegeben werden soll. Mit den im folgenden vorgestellten Netzelementen lassen sich eine Vielzahl von Systemen modellieren und mit geeigneten Tools auch verifizieren.

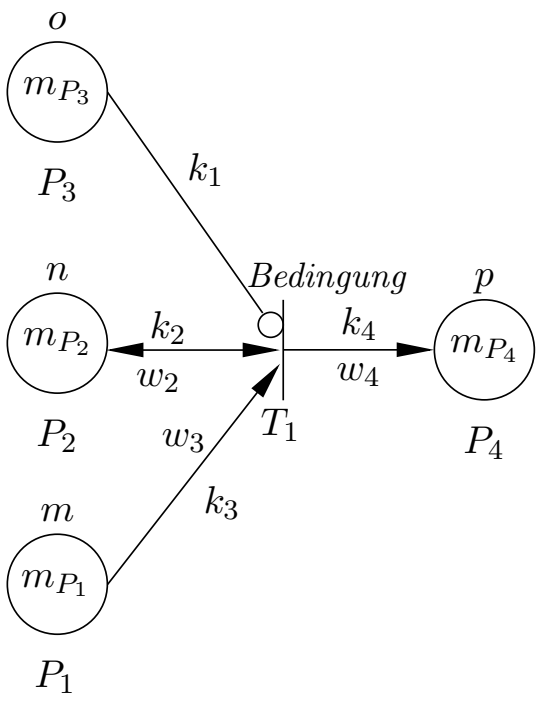


Abbildung 3.3: Wichtige Netzelemente im Petri-Netz zur Modellierung der dynamischen Systemeigenschaften

Das in Abbildung 3.3 gezeigte Petri-Netz enthält alle benötigten Elemente. Diese sind im einzelnen die Stellen P_1, P_2, P_3 und P_4, die auch als Plätze bezeichnet werden, die Transition T_1 und die Kanten k_1, k_2, k_3 und k_4.

Wenden wir uns zunächst den Plätzen zu. Die Plätze des Petri-Netzes können mit Marken gefüllt sein. Dabei geben die natürlichen Zahlen m, n, o und p die maximale Aufnahmekapazität (C) des Platzes an. An einem Platz können keine Marken über der Kapazitätsgrenze abgelegt werden. Fehlt in den folgenden Beispielen die Angabe der Aufnahmekapazität, so hat der Platz die Aufnahmekapazität von 1. In einigen Publikationen trifft man ab und zu auf die Aussage, daß bei fehlender Angabe die Kapazität eines Platzes unendlich ist [9]. In Hinblick auf die automatische Simulation und Umsetzung des Netzes in Software könnte diese Annahme ein Hindernis und auch eine mögliche Fehlerquelle sein. Die Anzahl der Marken eines Platzes $m_{P_1}, \ldots, m_{P_4}$ werden entweder mit der entsprechenden Anzahl von schwarz eingefärbten Kreisen oder mit dem entsprechenden Zahlenwert innerhalb des Platzes dargestellt.

Die Plätze im Petri-Netz sind durch die Kanten mit den Transitionen verbunden. In den Transitionen findet der eigentliche Schaltvorgang statt. Die Zustandswechsel des Netzes werden durch das Schalten von Transitionen vorgenommen.

Damit eine Transition schalten kann, müssen die Vorbedingung, die Nachbedingung und die Schaltbedingung, die an der Transition zum Beispiel in Form eines boolschen Ausdrucks notiert ist, erfüllt sein. Die Vorbedingung prüft alle zur Transition hinführenden Kanten (k_1, k_2, k_3). Die drei Kanten in Abbildung 3.3 haben dabei verschiedene Eigenschaften, was durch deren unterschiedliche Syntax zum Ausdruck gebracht wird:

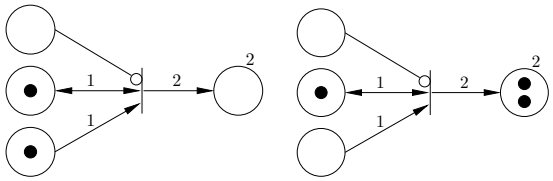
- **Inhibitorkante** k_1: Die Inhibitorkante erlaubt genau dann das Schalten der Transition, wenn der mit ihr verbundene Platz keine Marke enthält ($P_3 : m_{P_3} = 0$).

- **Testkante** k_2: Die Testkante erlaubt genau dann das Schalten, wenn der mit ihr verbundene Platz die geforderte Anzahl von Marken (w_2) enthält, wenn also $P_2 : m_{P_2} = w_2$ gilt. Beim Schaltvorgang verändert sich die Anzahl der Marken in P_2 nicht.

- **Flußkante** k_3: Die Flußkante erlaubt genau dann das Schalten, wenn die Anzahl der Marken im mit ihr verbundenen Platz mindestens gleich der geforderten Anzahl (w_3) ist, wenn also $P_1 : m_{P_1} \geq w_3$ gilt. Beim Schalten werden w_3 Marken vom Platz entfernt.

Die Nachbedingung prüft alle von der zu untersuchenden Transition T_1 wegführenden Kanten (k_4). Die Nachbedingung ist erfüllt, wenn für jede Kante k und dem mit ihr verbundenen

Platz $P : w_k \leq C - m_P$ gilt. Beim Schalten werden dann w_k Marken in den entsprechenden Plätzen abgelegt, wie es in Abbildung 3.4 dargestellt ist.

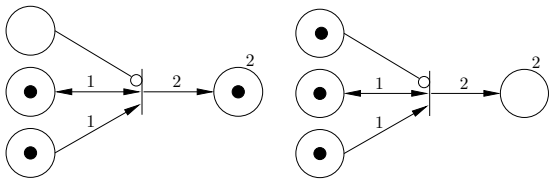


(a) Ausgangszustand des Petri-Netzes mit Marken in den Plätzen

(b) Zustand des Netzes nach Ausführen des Schaltvorgangs

Abbildung 3.4: Der Schaltvorgang eines Petri-Netzes stellt einen Zustandswechsel des modellierten Systems dar.

Beim in der Abbildung 3.5 gezeigten Ausgangszustand kann die Transition nicht schalten, da beim linken Petrinetz die Nachbedingung nicht erfüllt ist. Der Platz kann nur zwei Marken aufnehmen. Da er aber schon eine Marke enthält, würde die Kapazitätsgrenze nach dem Schalten überschritten werden. Beim rechten Petrinetz ist die Vorbedingung der Inhibitorkante nicht erfüllt, so daß auch hier die Transition nicht schaltfähig ist.



(a) Nachbedingung der Transition ist nicht erfüllt

(b) Nicht erfüllte Vorbedingung verhindert das Schalten der Transition

Abbildung 3.5: Transition kann jeweils infolge der nicht erfüllten Nachbedingung und Vorbedingung nicht schalten.

Die hier vorgestellten Elemente ermöglichen die Erstellung von Petri-Netzen. Eine wichtige Eigenschaft ist die **Lebendigkeit** des Netzes. Diese Eigenschaft läßt sich induktiv beschreiben: Ein Petri-Netz $(\mathcal{S}, \mathcal{T}, \mathcal{F})$ mit der Markierung M ist lebendig, wenn es mindestens eine Transition $t \in \mathcal{T}$ gibt, die schalten kann, und wenn für jede solche Transition das Netz mit der entsprechenden Folgemarkierung wieder lebendig ist. Lebendige Netze stellen also sicher, daß es weder zu einem Mangel noch zu einem Überfluß an Marken kommt.

Ein System, das für einen Endlosbetrieb ausgelegt sein soll, kann nur durch ein lebendiges Petri-Netz beschrieben werden. Fordert man für den Entwurf von Prozessen die Lebendigkeit eines Petri-Netzes, so weist das Fehlen dieser Eigenschaft auf eine Verklemmung (*Deadlock*) durch gegenseitiges Blockieren von Ressourcen hin.

3.1.2 Modellierung und Simulation mit Peneca Chromos

Zur Erstellung, Verwaltung und Simulation von Petri-Netzen wurde an der Technischen Universität Ilmenau das Programm "Peneca Chromos" [5] entwickelt.

Die Funktionalität dieses Programms hinsichtlich der hier kurz beschriebenen Netzelemente geht weit darüber hinaus. Neben der Möglichkeit zur Definition von Unternetzen können Teile des Netzes auch eingefärbt werden. Während der Simulation können die Schaltbedingungen der Transitionen angepaßt und dynamisch geändert werden.

Zwei Beispielnetze sind die im folgenden näher vorgestellten klassischen Probleme der speisenden Philosophen und des schlafenden Barbiers[3].

Speisende Philosophen In der Abbildung 3.6 ist das Petri-Netz für das Problem der speisenden Philosophen dargestellt. So wurde es in das Programm Peneca Chromos eingespeist und simuliert. Die Lebendigkeit des entstandenen Netzes läßt darauf schließen, daß keine Verklemmung entsteht.

Die Lebendigkeit des Modells ist zwar eine Bestätigung für den richtig gewählten Ansatz zur Abbildung des Prozesses mit Hilfe des Petri-Netzes, jedoch keine Garantie für die Lebendigkeit des ausgeführten Programmcodes, der dieses Modell in eine Folge von Prozessorbefehlen abbildet.

[3] Falls kein geeignetes Simulationsprogramm zur Verfügung steht, sind für diese Beispiele und auch in ähnlich gelagerten Fällen M3-Muttern als "bewegbare" Marken hilfreich.

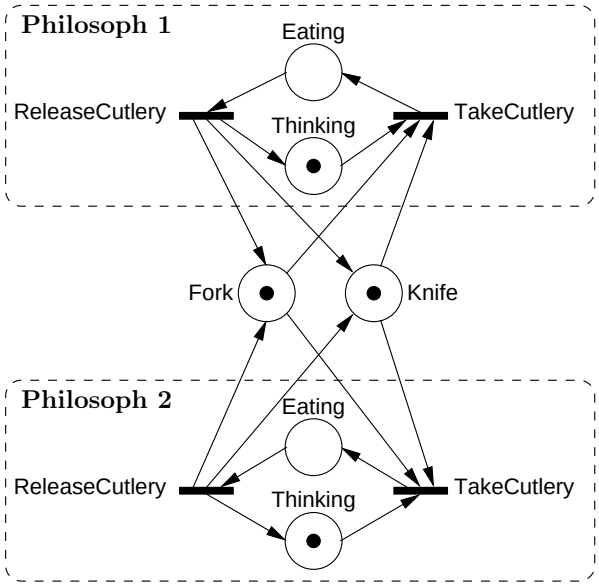


Abbildung 3.6: Petri-Netz für zwei mit Messer (*Knife*) und Gabel (*Forc*) abwechselnd speisende Philosophen.

Für kritische Abschnitte im Petri-Netz, die man an bestimmten Mustern erkennen kann, müssen im Programmcode spezielle Vorkehrungen getroffen werden, um das gleiche Verhalten wie das Petri-Netz zu erzwingen.

Die Transition "*TakeCutlery*" modelliert den Vorgang der Aufnahme des Bestecks. Dieser Vorgang ist eines der Muster, die besondere Maßnahmen erfordern, da hier mehrere kritische begrenzte Ressourcen beteiligt sind. Schaltet diese Transition, so werden die Marken der Plätze "*Knife*" und "*Fork*" **gleichzeitig** entnommen. Bei der Ausführung des Programmcodes durch den Prozessor gibt es jedoch kein "**gleichzeitig**". Der Merksatz Nr. 1 lautet: Irgendwann wird alles **serialisiert**. Ferner besteht die Möglichkeit, daß Ausführungsumgebungen zum Zwecke der Optimierung Befehle nicht unbedingt in der Reihenfolge ausführen, wie sie im Programm-Code geordnet sind.

Die Implementierung der speisenden Philosophen muß demnach dafür sorgen, daß die Aufnahme des Eßbestecks so gestaltet wird, daß keine Verklemmungen entstehen.

Schlafender Barbier Das Problem des schlafenden Barbiers ist ähnlich mit der Abarbeitung einer Warteschlange mit einer endlichen Anzahl von Warteplätzen. Das Geschäft des Barbiers hat einen Warteraum mit n Plätzen. Falls der Barbier keine Kunden hat, schläft er auf seinem Stuhl.

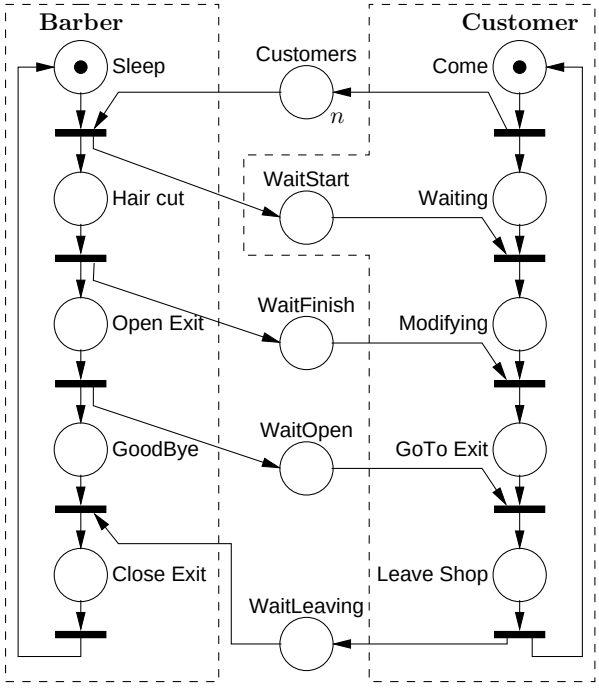


Abbildung 3.7: Petri-Netz für den (nicht immer) schlafenden Barbier (von Sevilla). Der Platz "*Customers*" simuliert n Warteplätze im Warteraum des Barbiers.

Betritt ein Kunde den Raum und der Barbier schläft, so weckt er den Barbier. Andernfalls nimmt er im Warteraum Platz, sofern dort noch Plätze vorhanden sind. Falls nicht, kommt er später wieder.

Der Barbier schneidet dem Kunden die Haare, öffnet dem Kunden persönlich die Ausgangstür und schließt sie danach wieder. Im Anschluß wartet er auf den nächsten Kunden oder holt den nächsten Kunden aus dem Wartezimmer.

Der Barbier und ein Kunde sind im Petri-Netz in Abbildung 3.7 gezeigt. Es können mehrere Kunden einbezogen werden. Die Plätze zwischen dem Barbier und Kunden sind zur Synchronisation der Vorgänge notwendig. Der Platz "*Customers*" simuliert n Warteplätze im Warteraum des Barbiers.

Die darunterliegenden Plätze synchronisieren die Handlungen des Barbiers mit dem aktuell bedienten Kunden. Bis auf den Platz "*Wait-*

Start", der dem gerade bedienten Kunden signalisiert, daß er an der Reihe ist, sind alle Plätze global einem Barbier zugeordnet, da er nur einen Kunden gleichzeitig bedienen kann.

Neben der hier vorgestellten Anwendung zur Erstellung und Simulation von Petri-Netzen existiert eine Vielfalt anderer, teilweise frei verfügbarer Programme. Auch ohne solche Hilfsmittel besticht die Einfachheit und Intuition von Petri-Netzen.

3.2 Specification and Description Language (SDL)

Bei der *Specification and Description Language* (SDL) handelt es sich um eine objektorientierte formale Beschreibungssprache, welche die Konzeption, die Entwicklung und das Testen von Systemen unterstützt. Erstmalig definiert wurde sie von der CCITT[4]. Vorrangig wird es zur Entwicklung von einzelnen miteinander kommunizierenden Schichten (Layern) von Protokoll-Stacks eingesetzt [1].

Beschreibungssprachen wie SDL erfüllen die folgenden Anforderungen:

- aus bekannten Konzepten bestehend,
- eindeutig, klar und präzise,
- eine Basis zur Analyse der Beschreibung bildend,
- stellen eine Ausgangsbasis zur Validierung der Implementation gegen die Spezifikation dar,
- Ermöglichung der Konsistenzprüfung der Spezifikation,
- Erzeugung von Applikationen ohne traditionelle Implementierungsphase.

Mit SDL ist die Beschreibung der Struktur, des Verhaltens und der Daten von verteilten Echtzeitkommunikationssystemen möglich. Auf Grund der mathematischen Definition werden Mehrdeutigkeiten eliminiert und Systemintegrität gewährleistet. Die Bedeutung jedes Symbols und Konzepts ist eindeutig definiert.

Diese Eigenschaften prädestinieren die Verwendung von SDL zur Beschreibung von großen komplexen Echtzeitsystemen.

3.2.1 Theoretisches Modell

Das zugrundeliegende Modell eines SDL-Systems enthält endliche Automaten[5], die parallel arbeiten. Die Automaten sind voneinander unabhängig und kommunizieren mit diskreten Signalen.

Ein SDL-System enthält die folgenden Komponenten, die nachfolgend erläutert werden:

- Struktur: System-, Block-, Prozeß- und Prozedur-Hierarchie
- Kommunikation: Signale mit optionalen Parametern und Kanälen (Routen)
- Verhalten: Prozessbeschreibungen
- Daten: Abstrakte Datentypen (ADT)
- Vererbung: Beschreibung von Relationen und Spezialisierungen

Struktur Die Struktur des SDL-Systems erlaubt die Nutzung von vier hierarchischen Ebenen. In den jeweils übergeordneten Ebenen können mehrere Instanzen der untergeordneten Ebene enthalten sein. Zum Beispiel kann ein Prozeß mehrere Prozeduren enthalten. Die in SDL angebotenen Ebenen sind: Systeme, Blöcke, Prozesse und Prozeduren.

Die Aufteilung eines Systems in verschiedene Ebenen nennt man Partitionierung. Einige Ziele der Partitionierung sind zum Beispiel:

- Verstecken von Informationen (Details möglichst in untere Ebenen verschieben)
- Abbildung vorhandener Strukturen und funktionalen Gruppierungen
- Erzeugung von handhabbaren Modulen (beschränkte Größe und Komplexität)

[4] **C**omité **C**onsultatif **I**nternational **T**éléphonique et **T**élégraphique als Untergruppe der ITU (**I**nternational **T**elecommunication **U**nion)

[5] *finite state machines* (FSM)

- Modellierung von bestehender Hard- und Software
- Wiederverwendung von bereits existierenden Beschreibungen

Die statische Struktur eines Systems wird in SDL durch diese Hierarchie definiert. Zwischen den Elementen bestehen Schnittstellen. Ein Block wird zum Beispiel nach dem Black-Box-Modell behandelt.

Darüber hinaus besteht die Möglichkeit, eine dynamische Struktur zu definieren. Dies wird mit Hilfe der Konzepte der Prozesse und Signalrouten realisiert.

Kommunikation SDL verwendet keine globalen Variablen zur Kommunikation zwischen einzelnen Systemteilen, sondern bietet zwei Verfahren zur Kommunikation an: Asynchrone Signale mit optionalen Signalparametern und synchrone Funktionsaufrufe. Beide Verfahren können Parameter zum Datenaustausch und zur Synchronisation zwischen SDL-Prozessen, mit dem SDL-System und seiner Umgebung, zum Beispiel externen Anwendungen oder anderen SDL-Systemen, übertragen.

SDL stellt darüber hinaus verschiedene Zeitgeber zur Verfügung, die einfach über Signale zu integrieren sind. Dies erleichtert die Modellierung von Timeout-Prozessen zur Überwachung.

Die Vergabe von Prioritäten für Signale und Prozesse ist in SDL nicht möglich. Dieses Problem wird in die Implementierungsphase verlagert, da dies spezialisierte Funktionen übernehmen.

Verhalten Das dynamische Verhalten eines SDL-Systems ist in den Prozessen beschrieben. Prozesse können beim Start des Systems oder während der Laufzeit erzeugt und beendet werden. Von einem Prozeß kann mehr als eine Instanz existieren. Jede Instanz hat einen eindeutigen Identifikator. Der Identifikator ermöglicht die selektive Übertragung von Signalen zu bestimmten Instanzen eines Prozesses. Für eine Echtzeitbeschreibungssprache ist dies besonders wichtig.

Daten SDL ermöglicht die Definition und Beschreibung von Datentypen mittels abstrakten Datentypen (ADT) sowie ASN.1[6] [6]. Die Nutzung von ASN.1 erlaubt den zwischensprachlichen Austausch der Datentypen und die Wiederverwendung existierender Datenstrukturen.

Vererbung Das Konzept zur Vererbung als Bestandteil der Objektorientierung in SDL basiert auf der Typ-Deklaration. Bei der Spezialisierung können die Eigenschaften des Supertyps vererbt und vom Subtyp überschrieben werden.

3.2.2 Syntax

Für SDL existiert eine graphische Syntax (SDL/GR), deren wichtigste Symbole in den Abbildungen 3.8 und 3.9 dargestellt sind.

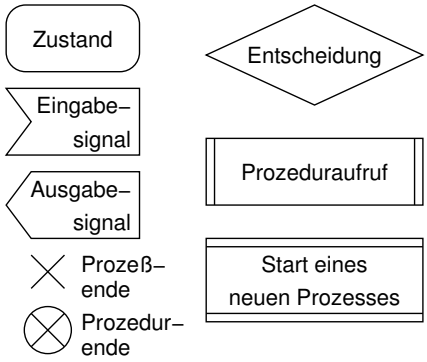


Abbildung 3.8: Übersicht einiger Symbole in SDL

Systeme, Blöcke und Prozesse kommunizieren über Kanäle, über die definierte Nachrichten ausgetauscht werden. Jeder Prozeß und jede Prozedur besitzt ein Startsymbol, bei dem die Ausführung beginnt, sowie ein Endsymbol, nach welchem der Prozeß oder die Prozedur beendet ist.

In einem Zustand wird ein Zustandsübergang durch ein Eingabsignal ausgelöst. Der Zustand kann Tasks, Entscheidungen, Prozeduraufrufe, Prozeßerzeugungen mit Prozeßstart oder Signalausgaben enthalten, bevor der Automat in den nächsten Zustand übergeht.

Ein mit SDL beschriebenes verteiltes Diskurssystem ist in [8], und ein ebenfalls mit SDL

[6] **A**bstract **S**yntax **N**otation

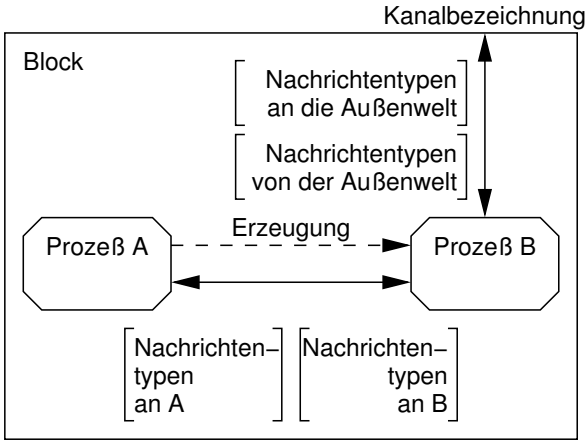


Abbildung 3.9: Übersicht einiger Symbole in einem SDL-Block

spezifizierter verteilter Algorithmus für einen Web-Proxy mit Cooperative Garbage Collection ist in [10] zu finden.

3.3 Zusammenfassung

Petri-Netze und SDL sind zwei von vielen Möglichkeiten zur Modellierung nebenläufiger Prozesse. Die Kriterien zur Auswahl eines Modells oder Werkzeugs zur Verwendung in einem Projekt sind ebenso vielfältig. Ein wichtiges Kriterium ist die Frage: Ist das Modell soweit formalisiert und in sich geschlossen, daß damit der angestrebte Detaillierungsgrad des Systems erreicht werden kann? Darüber hinaus sollten die existierenden Computer-Anwendungen zur Unterstützung des Modells sorgfältig ausgewählt und getestet werden.

Kapitel 4

Thread-Programmierung in Java

Die nebenläufige Programmierung in Java wird durch die in die Sprache eingebetteten Konstrukte wesentlich erleichtert. Es existieren Objekte mit verschiedenen Methoden speziell für diese Aufgaben. So wird in Java ein Monitorkonzept zur Synchronisation angeboten, wie es in Abschnitt 2.4.3 vorgestellt wurde.

Die Ausführungsumgebung, auch Laufzeitumgebung genannt, für Java-Code ist die *Java Virtual Machine* (JVM). Sie verwaltet und führt Prozesse aus, die *Threads* genannt werden. Zur Behandlung von Threads und zur Synchronisation des Datenaustauschs zwischen Threads garantiert die JVM die Einhaltung bestimmter Regeln. Diese Regeln stellen die Richtlinien für die Entwicklung des Java-Codes dar.

Andererseits bedeutet dies, daß diese Regeln auch verbindlich für die jeweilige Implementierung der Ausführungsumgebung, die *Java Runtime Environment* (JRE) genannt wird, sein müssen. Kleine Abweichungen und andere Fehlerquellen sind möglicherweise die Ursache für den Indeterminismus des Programms.

Bevor am Ende dieses Abschnitts auf einige praktische Beispiele eingegangen wird, folgen zunächst zwei Exkurse zur Behandlung von Threads, also dem Weg von ihrer Erzeugung bis zum Ende, und zur Synchronisation von gemeinsam benutzten Daten.

4.1 Threads in Java

Die Ausführung eines Programms beginnt in der Regel mit der Ausführung der `main`-Methode der Hauptklasse durch die Ausführungsumgebung. Zu diesem Zeitpunkt wird zunächst der Vater-Prozeß (*Main Thread*) der Anwendung gestartet.

Neben dem *Main Thread* können auch je nach Anforderungen des Programms weitere Threads von der Ausführungsumgebung selbst erzeugt werden. Zum Beispiel benötigt eine Java Swing-Anwendung einen *Event Dispatch Thread*, der alle Interaktionen des Nutzers mit Maus und Tastatur bearbeitet, und zum Betrieb eines RMI-Servers werden verschiedene Threads zur Anbindung des RMI-Servers gestartet.

Die vom Programmierer nicht beeinflußbaren Entscheidungen der Ausführungsumgebungen hinsichtlich der notwendigen Prozesse setzen einen gegen Unterbrechungen geschützten Quelltext beziehungsweise Programmcode voraus. Diese Eigenschaft des Codes bezeichnet man auch als reentrant oder *thread-safe*.

Im weiteren Verlauf können im Programm explizit Anweisungen enthalten sein, die neue Threads erzeugen.

An Hand einer Eigenschaft lassen sich zwei Kategorien von Threads unterscheiden: Threads und Dämon-Threads. Diese Eigenschaft beeinflußt das Verhalten der Laufzeitumgebung beim Beenden des Programms.

Die Laufzeitumgebung wartet nach dem Empfang des Signals zum Beenden auf das natürliche Ende aller gestarteten Threads, die keine Dämonen sind. Die Threads können in diesem Fall ungestört weiterarbeiten, was nicht immer beabsichtigt ist. Im Gegensatz dazu werden Dämon-Threads von der Laufzeitumgebung sofort terminiert.

Es besteht die Möglichkeit, den Thread mittels der Methode `void setDaemon(boolean on)` als Dämon-Thread zu markieren, oder

auch rückwirkend wieder als normalen Thread bekanntzumachen. Mit der Methode `boolean isDaemon()` kann getestet werden, ob der Thread ein Dämon ist oder nicht.

Treten während der Abarbeitung eines Threads Ausnahmen (*Exceptions*) auf, die nicht behandelt werden, so führen diese zur Terminierung des Threads.

4.2 Lebenszyklus eines Threads

In Java existieren zwei Möglichkeiten, einen Thread zu erzeugen. Zum einen kann ein von der Klasse `Thread` spezialisiertes Objekt erzeugt werden. Zum anderen kann das nebenläufig auszuführende Objekt die Schnittstelle `Runnable` implementieren. In beiden Fällen muß die `run`-Methode des spezialisierten Objekts überschrieben werden. Die zweite Vorgehensweise hat den Vorteil, daß das Objekt weiterhin in einer Vererbungslinie eingesetzt werden kann. Die Mehrfachvererbung wird in Java nämlich nicht unterstützt.

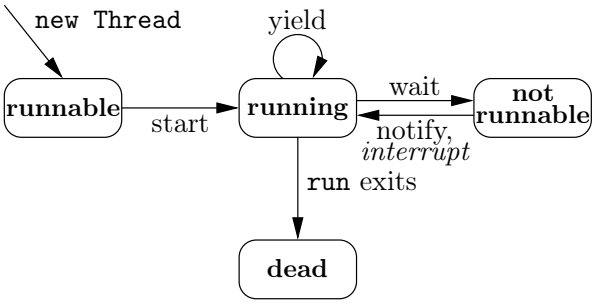


Abbildung 4.1: Lebenszyklus eines Java-Threads mit den ”erlaubten” Methoden (die in neueren Java-Versionen als *deprecated* markierten Methoden wurden weggelassen) als Auslöser für Zustandsübergänge.

Nach der Erzeugung des Threads mit dem Aufruf des Konstruktors mit `new` ist der Thread im Zustand *Runnable*, das heißt, er ist ausführbar, aber noch nicht gestartet. Wie in Abbildung 4.1 gezeigt wird, gelangt der Thread durch den Aufruf der `start`-Methode in den Zustand *Running*.

In den Zustand *Dead* gelangt der Thread nur, wenn die `run`-Methode verlassen wird. Dies geschieht entweder durch das normale Beenden der Methode oder durch das Verlassen der Methode nach einer vorher nicht abgefangenen Ausnahme.

Aus dem Zustand *Running* kann der Thread durch Aufruf von `wait` gebracht werden. Der Thread befindet sich nun im Zustand *Not Runnable*, den er entweder durch ein zugehöriges `notify`-Signal oder durch den Aufruf der `interrupt`-Methode verlassen kann.

Die `interrupt`-Methode muß nicht unbedingt zum Verlassen der `run`-Methode führen, da die durch den Aufruf ausgelöste `InterruptedException` abgefangen werden könnte. Das wünschenswerte Verhalten beim Empfang einer solchen Ausnahme ist der Wechsel in den Zustand *Dead*. Die Ausnahme kann zwar abgefangen werden, sollte dann aber weitergegeben werden.

Ein Thread mit der im Programmausdruck 4.1 aufgeführten `run`-Methode könnte niemals beendet werden, da die InterruptedException innerhalb der `while`-Schleife abgefangen wird.

```
public void run() {
  while(true) {
    if (isInterrupted()) break;
    try {
      Thread.wait();
    } catch (InterruptedException e) {
      System.out.println("Terminiere nie!");
    }
  }
}
```

Programmausdruck 4.1: Thread terminiert nie

Zum Beenden des Threads von außerhalb ist ein eigener implementierter Mechanismus die geeignetste Variante. Die in früheren Versionen der Laufzeitumgebung angebotenen Funktionen zum externen Beenden eines Threads werden nicht mehr angeboten. Abgesehen davon gehört es zum guten Programmierstil, alle Threads am Ende der Anwendung mit einem deterministischen Verfahren zu terminieren. Eine Möglichkeit dazu wird im Programmausdruck 4.2 gezeigt.

```
public void run() {
  try {
    while(true) {
      if (isInterrupted()) break;
      Thread.sleep(500);
    }
  } catch (InterruptedException e) {
    System.out.println("Terminated");
  }
}
```

Programmausdruck 4.2: Thread wird korrekt terminiert

Der Thread kann mit dem Aufruf der `interrupt`-Methode beendet werden. Der Thread befindet sich zum Zeitpunkt des Aufrufs von `interrupt()` höchstwahrscheinlich in der `sleep`-Methode. Diese Methode wirft dann die `InterruptedException`. Andernfalls wird die `while`-Schleife verlassen, da die Bedingung `isInterrupted()` erfüllt ist.

Die Behandlung der Ausnahme `InterruptedException` setzt das Flag zum Beenden eines Threads wieder zurück. Deswegen terminiert der Thread in Programmausdruck 4.1 nicht. Zur Abhilfe muß innerhalb des `catch`-Blocks die `interrupt`-Methode nochmals aufgerufen werden, damit im weiteren Verlauf die Bedingung `isInterrupted()` erfüllt ist, und der Thread terminiert.

Neben den Funktionen, die Einfluß auf den Zustand des Threads ausüben, gibt es noch weitere Funktionen, von denen die `join`-Methode an dieser Stelle herausgegriffen wird. Der Aufruf der `join`-Methode kehrt erst zurück, wenn der Thread die `run`-Methode verlassen hat. Optional kann diese Methode mit einem Parameter aufgerufen werden, in dem die maximale Wartezeit angegeben ist. Dann kehrt sie auch zurück, wenn die im Parameter angegebene Zeit verstrichen ist. Ein Beispiel ist in Programmausdruck 4.3 angegeben.

Auf mehrere Threads kann entweder durch aufeinanderfolgende Aufrufe der `join`-Methode oder durch Zusammenfassen der Threads in einer Thread-Gruppe und dem Aufruf der `join`-Methode der Thread-Gruppe gewartet werden.

Beim erstgenannten Weg ist es unerheblich,

```
class JoinTheThread {
  static class JoinerThread extends Thread {
    public int result;
    public void run() {
      result = 1;
    }
  }
  public static void main(String args[]) throws
      Exception {
    JoinerThread t = new JoinerThread();
    t.start();
    t.join();
    System.out.println(t.result);
  }
}
```

Programmausdruck 4.3: Thread, auf den gewartet wird (aus [15])

welcher Thread am schnellsten fertig ist. Denn erst wenn der langsamste Thread beendet ist, kann weitergearbeitet werden. Die Reihenfolge des Aufrufs der `join`-Methoden ist in diesem Fall unerheblich.

4.3 Synchronisation von Objekten

Die von mehreren Threads einer Anwendung ausgeführten Aktionen müssen miteinander synchronisiert werden, um einen korrekten beziehungsweise den gewünschten Ablauf sicherzustellen.

Die Ziele der Synchronisation sind in zwei Hauptkategorien aufteilbar: Ein Ziel ist die Realisierung eines komplexen, von Bedingungen gesteuerten Ablaufs von Prozessen. Ein Beispiel hierfür sind der Barber- und Customer-Prozeß des schlafenden Barbiers in Abbildung 3.7. Zwischen den beiden Prozessen sind mehrere Synchronisationspunkte definiert.

Die zweite Kategorie hat als vorrangiges Ziel die Sicherung der Exklusivität bestimmter Aktionsfolgen oder Datenstrukturen. In den zu schützenden Bereichen darf sich jeweils nur ein Prozeß aufhalten. So ist zum Beispiel bei den speisenden Philosophen das Aufnehmen des Eßbestecks durch einen Philosophen eine exklusive Operation (siehe Abbildung 3.6).

Für die Synchronisation stellt die Program-

miersprache Java bestimmte Hilfsmittel zur Verfügung. Mit ihnen lassen sich auch bedingungsgesteuerte Prozeßabläufe modellieren und auch andere Mechanismen zur Synchronisation, wie sie im Abschnitt 2.4 vorgestellt wurden, realisieren.

4.3.1 Schlüsselwörter und Funktionen

Zur Koordination nebenläufiger Aktivitäten stehen in Java die in folgender Aufzählung aufgeführten Schlüsselwörter und Funktionen zur Verfügung. Auf ihre Bedeutung wird kurz in diesem Abschnitt und ausführlich in den folgenden Abschnitten eingegangen.

`volatile` Die mit diesem Schlüsselwort deklarierten Variablen werden von der Laufzeitumgebung im Hinblick auf den Zugriff gesondert behandelt. Näheres ist im Abschnitt 4.3.3 zu finden.

`synchronized` Die markierten Methoden oder umschlossenen Blöcke werden mit Hilfe des Monitors synchronisiert (siehe Abschnitt 4.3.2).

`Object.wait()` Das Objekt wartet auf die Freigabe des Monitors, also auf ein durch die `notify`-Methode oder die `notifyAll`-Methode ausgelöstes Signal (siehe Abschnitt 4.3.3).

`Object.notify()` Der Aufruf signalisiert einem mit der `wait`-Methode auf den Monitor wartenden Thread, daß sich der Monitorstatus vielleicht ändern wird. Der Thread wird aus der Liste aller auf diesen Monitor wartenden Threads nach einem durch die Ausführungsumgebung festgelegten Verfahren ausgewählt (siehe Abschnitt 4.3.3).

`Object.notifyAll()` Verhält sich vom Prinzip her wie die `notify`-Methode. Dieser Aufruf leitet jedoch das Signal nicht nur an einen, sondern an alle auf diesen Monitor wartenden Threads weiter (siehe Abschnitt 4.3.3).

4.3.2 Monitore

Jedem Java-Objekt[1] ist ein Monitor zugeordnet. Die in der im Abschnitt 2.4.3 vorgestellten Theorie mit *signal* und *wait* bezeichneten Methoden werden in Java auf die `notify(All)`-Methode und `wait`-Methode der Klasse `Object` abgebildet.

Die Methoden zum Warten und Signalisieren können nur aufgerufen werden, wenn der aufrufende Thread den Monitor der jeweiligen Objektinstanz besitzt. Andernfalls wird von den Methoden die Ausnahme `IllegalMonitorStateException` ausgelöst.

Den Monitor eines Objekts besitzt ein Thread genau in den folgenden Fällen:

1. Bei der Ausführung einer mit `synchronized` markierten Methode des Objekts
2. Bei der Ausführung eines mit `synchronized(this) {}` umschlossenen Blocks, der diese Objektinstanz `this` synchronisiert
3. Bei der Ausführung einer mit `synchronized` und `static` markierten Methode von Objekten vom Typ `Class`

In allen anderen Fällen als diesen genannten besitzt der Thread nicht den Monitor des Objekts. Dies bedeutet, daß sich ungeschützte Bereiche eines Objekts nicht mit Monitoren kontrollieren lassen. Deswegen ist es wichtig, bei der Arbeit mit Monitoren jede Stelle im Code zu finden, die synchronisiert werden muß.

Einen Spezialfall stellen die Konstruktoren von Objekten dar. Mit ihnen werden neue Instanzen von Objekten erzeugt. Der Konstruktor kann nicht synchronisiert werden. Hier hilft die Verschiebung der zur Initialisierung der neuen Objektinstanz notwendigen Aktionen in eine objekteigene Methode, die dann synchronisiert

[1] Genauer steht "*Java-Objekt*" für eine Instanz eines Java-Objekts. Jede Instanz besitzt einen eigenen Monitor. Eine Ausnahme bilden `static`-Methoden, da sie nur einmal von der Ausführungsumgebung instantiiert werden.

und als statisch deklariert wird. Eine synchronisierte Methode allein reicht nicht aus, da der Monitor nur für diese Instanz gilt. Eine statische Methode hingegen wird dem Monitor der Objektklasse zugeordnet.

Als Beispiel wird die Erzeugung von weitgehend eindeutigen Bezeichnern (*Identifier*) an Hand eines falschen und richtigen Beispiels gezeigt. Die einzelnen Identifikatoren (Objektinstanzen) besitzen als Unterscheidungsmerkmale (Attribute) den Zeitpunkt ihrer Erzeugung und einen Integer-Wert.

Der Konstruktor ruft die mit `synchronized` geschützte `generateID()`-Methode auf, um die internen Variablen der Instanz zu füllen (siehe Programmausdruck 4.4).

Um den Bezeichner mit einer hohen Wahrscheinlichkeit eindeutig werden zu lassen, wird zum einen der Zeitpunkt der Erzeugung auf Millisekundengenauigkeit berücksichtigt. Falls zu diesem Zeitpunkt schon eine andere Instanz erzeugt wurde, wird zum anderen ein objektinterner Zähler um 1 erhöht und der aktuelle Wert der gerade erzeugten Instanz zugewiesen. Falls der Zeitpunkt nicht mehr derselbe ist, wird dieser Zähler wieder auf den Wert Null zurückgesetzt, wie der Programmausdruck 4.5 zeigt.

Die neuen Instanzen sollen auf Grund der Erstellungszeit und des Zählers eindeutig sein. In einem in Programmausdruck 4.6 gezeigten Test werden dazu verschiedene Bezeichner von zwei Threads erzeugt.

Mit der Anweisung `yield` gibt jeder Thread nach der Erzeugung eines Bezeichners die Kontrolle an den anderen Thread ab.

Dennoch sind die so erzeugten Objektinstanzen nicht voneinander verschieden. In die `generateID()`-Methode wurde ein kurzes `sleep`-Statement eingebaut. Dies steht stellvertretend dafür, daß genau an dieser Stelle der aktuelle Prozeß, zum Beispiel der Erzeuger-Thread *runner1* die CPU an den anderen Thread *runner2* abgibt. Und zwar genau zwischen der Erhöhung des globalen Zählers und der Zuweisung dieses Wertes zur lokalen Variable der Instanz. Dadurch besteht die Möglichkeit, daß mehrere Bezeichner denselben Zählerstand zugewiesen bekommen, da der zweite Thread den Zähler erhöht, während der erste Thread das Ende der `sleep`-Methode abwartet und dann den neuen Zählerstand seiner Instanz zuweist.

Die Ausgaben des Testprogramms bestätigen die Vermutung. Der hinter dem Punkt ausgewiesene Stand des Zählers enthält beim ersten von Thread `T1` erzeugten Identifikator den Wert `0`. In derselben Millisekunde erhöht Thread `T2` den globalen Zähler um 1, um dann zu pausieren. Nun erhöht Thread `T1` wieder den Zähler und pausiert ebenfalls. Danach wacht Thread `T2` wieder auf und weist den aktuellen Zähler-

```
public Identifier() {
  super();
  generateID();
}
```

Programmausdruck 4.4: Konstruktor des Bezeichners

```
private static int globalCounter= 0;
private static long prevCreationTime= 0;
private int counter= 0;
private long creationTime= 0;
public synchronized void generateID() {
  Date date= new Date();
  long tmpTime= date.getTime();
  if (tmpTime == prevCreationTime) {
    globalCounter++;
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    counter= globalCounter;
    creationTime= prevCreationTime;
  } else {
    globalCounter= 0;
    counter= globalCounter;
    creationTime= tmpTime;
    prevCreationTime= creationTime;
  }
}
public String toString() {
  return creationTime + "." + counter;
}
```

Programmausdruck 4.5: `generateID()`-Methode der Klasse `Identifier`

```
public static void main(String[] args) {
  Thread runner1= new Thread() {
    public void run() {
      for (int i= 0; i < 200; i++) {
        System.out.println("T1:␣" + new
            Identifier());
        yield();
      }
    }
  };
  Thread runner2= new Thread() {
    public void run() {
      for (int i= 0; i < 200; i++) {
        System.out.println("T2:␣" + new
            Identifier());
        yield();
      }
    }
  };
  runner1.start();
  runner2.start();
  try {
    runner1.join();
    runner2.join();
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

Programmausdruck 4.6: Test zur Erzeugung von Instanzen der Klasse `Identifier`

stand 2 dem neuen Identifikator zu. Thread `T1` tut danach dasselbe, so daß zwei gleiche Bezeichner erzeugt wurden:

```
T1: 1060068954562.0
T2: 1060068954562.2
T1: 1060068954562.2
T2: 1060068954671.0
T1: 1060068954671.2
T2: 1060068954671.2
T1: 1060068954765.0
...
```

Die synchronisierte `generateID()`-Methode reserviert nur den Monitor auf Objektebene, also pro Instanz des Objekts. Der globale Zähler `globalCounter` ist jedoch durch das Schlüsselwort `static` im gesamten Namensraum der Klasse, dem alle Instanzen angehören, sichtbar.

Die Lösung besteht darin, eine ebenfalls statische und synchronisierte Methode einzuführen, die eine neue Instanz des Bezeichners zurückliefert (Programmausdruck 4.7). Der Konstruktor ist dann nicht mehr öffentlich verfügbar.

```
public static synchronized Identifier get() {
  return new Identifier();
}
```

Programmausdruck 4.7: Durch den Monitor geschützte Erzeugung von Instanzen der Klasse `Identifier`

Die synchronisierte statische Methode nutzt den Monitor auf der instanzenübergreifenden `Class`-Ebene, und schützt somit den globalen statischen Zähler.

Mit dieser Modifikation enthält die Menge der erzeugten Identifikatoren paarweise unterschiedliche Elemente, wie ein Ausschnitt der Ausgaben des Testprogramms zeigen:

```
T2: 1060068755640.0
T2: 1060068755734.0
T1: 1060068755640.1
T1: 1060068755843.0
T2: 1060068755734.1
...
```

Durch das Beispiel wird deutlich, daß die Synchronisation von Threads häufig mit einer Synchronisation des Zugriffs auf gemeinsam benutzte Daten eng zusammenhängt.

4.3.3 Gemeinsam genutzte Daten

Während der Ausführung von Threads greifen diese auf Variablen zu. Aus der Sicht des Programmierers stellen Variablen referenzierte Speicherabschnitte dar, die zum Ablegen von Datenstrukturen dienen. Analog zur Speicherhierarchie der Princeton-Rechnerarchitektur, bei der sich zwischen CPU und Hauptspeicher noch weitere Speicher (Caches) befinden, definiert die Java-Ausführungsumgebung ebenfalls ein hierarchisches Speichermodell.

Das Speichermodell der Java-Ausführungsumgebung ist in Abbildung 4.2 schematisch dargestellt. Wie im Modell zu sehen ist, werden jedem erzeugten Thread zwei Zwischenspeicher zugeordnet, und zwar ein *Cache* und ein *Buffer*. Die Zwischenspeicher

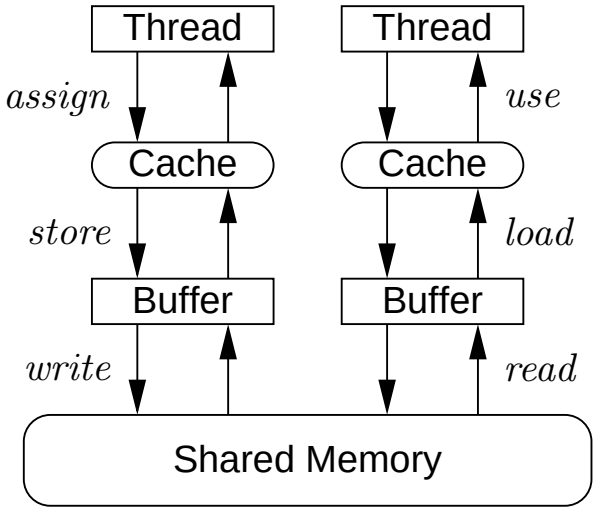


Abbildung 4.2: Speichermodell der Java-Ausführungsumgebung

verschiedener Threads sind nicht miteinander verbunden. Vielmehr können die Threads nur über den "globalen" Speicher des Ausführungsumgebungsprozesses, in der Abbildung als *Shared Memory* bezeichnet, gemeinsame Daten austauschen und benutzen.

Damit ein Thread eine Variable lesen kann, muß er die Operationen *read*, *load* und *use* in dieser Reihenfolge ausführen. Zum Ändern einer Variable muß er die Operationen *assign*, *store* und *write* in dieser Reihenfolge ausführen.

Der Befehlsumfang des Hauptspeichers umfaßt neben den Befehlen *read* und *write* auch die Befehle *lock* und *unlock*. Sie dienen zum Sperren und Freigeben von Variablen und werden von den Monitoren der Objekte gesteuert.

Für die Ausführung der hier genannten Befehle gelten die folgenden Regeln:

- Die von einem Thread ausgeführten Aktionen (Kette von diesen Befehlen) sind total geordnet, das heißt, eine Aktion folgt der anderen.
- Für die im Hauptspeicher auf den einzelnen Variablen ausgeführten Aktionen gilt dies ebenfalls, auch wenn sie vorher mit einem *lock*-Befehl exklusiv gesperrt wurden.
- Es ist nicht erlaubt, ein- und dieselbe Aktion direkt hintereinander auszuführen (z.B. muß auf ein *read* immer ein *write* folgen, und umgekehrt).

Mit diesen Regeln ist aber nicht sichergestellt, daß das Laden und Schreiben einer Variable eine unteilbare Aktion ist. Insbesondere Variablen des Typs **double** und **long** erfordern zwei Lade- und Schreibzyklen, da sie mit 64 Bit kodiert sind, aber die Verarbeitungsbefehle momentan nur eine Breite von 32 Bit aufweisen. Zwischen diesen zwei Zyklen ist jedoch eine Unterbrechung des Threads möglich, so daß sich bei gleichzeitiger Nutzung dieser Variable irrsinnige Belegungen ergeben könnten.

Speziell für diese Datentypen wurde der Modifikator **volatile** eingeführt. Alle Zugriffsoperationen werden auf dieser so gekennzeichneten Variable unteilbar ausgeführt. Neben der Atomarität der Zugriffsoperationen verbietet der Modifikator die Zwischenspeicherung des aktuellen Wertes der Variable. Nach jeder neuen Zuweisung muß der Wert der Variable im Hauptspeicher aktualisiert werden.

Dies reduziert zum einen die Optimierungsmöglichkeiten der Ausführungsumgebung [15] und zum anderen die Geschwindigkeit des Programms.

Greifen nun mehrere Threads schreibend auf eine Variable zu, so muß diese besonders geschützt werden, um die korrekte Funktionsweise sicherzustellen. Am besten geschieht dies unter Verwendung von Monitoren.

Abarbeitung von Aufträgen einer Liste Der folgende Programmausdruck 4.8 zeigt eine Datenstruktur mit Zugriffsmethoden zum Bearbeiten von Elementen einer Liste. Das Einfügen und Entfernen von Aufträgen geschieht synchronisiert.

Alle schreibenden und lesenden Funktionen dieser Objektinstanz sind synchronisiert. Die `retrieveObject()`-Methode wartet bei leerer Liste solange, bis mit der `queueObject`-Methode ein Objekt eingefügt wurde. Enthält die Liste beim Aufruf der `retrieveObject()`-Methode Elemente, so wird das erste Element der Liste zurückgeliefert. Dabei ist es wichtig, die `InterruptedException` an den aufrufenden Thread weiterzuleiten.

```
public class FIFOQueue implements Queue {
  private LinkedList objects =
   new LinkedList();
  private int currObjectsCount = 0;
  public synchronized void queueObject(
       Object o) {
    objects.add(o);
    if (++currObjectsCount == 1) {
      notify();
    }
  }
  public synchronized Object retrieveObject()
       throws InterruptedException {
    while (currObjectsCount == 0) {
      wait();
    }
    --currObjectsCount;
    return objects.removeFirst();
  }
  public synchronized boolean peek() {
    return currObjectsCount > 0;
  }
}
```

Programmausdruck 4.8: Datenstruktur mit einer Liste

Die so gekapselte Listendatenstruktur kann problemlos in verschiedenen Versionen der Ausführungsumgebungen eingesetzt werden, da die korrekte Synchronisation nicht mehr von den zugesicherten Eigenschaften der im `java.util`-Package enthaltenen Datenstruktur abhängig sind.

Mit dieser synchronisierten Auftragsliste können mehrere asynchron lesende und schreibende Threads arbeiten. Der hier vorgestellte `FIFOQueueWriter` in Programmausdruck 4.9 füllt die Auftragsliste mit nummerierten Aufträgen. Die Aufträge werden vom `FIFOQueueReader` im Programmausdruck 4.10 aus der Liste ausgelesen.

Beim schreibenden Thread ist auf die Abbruchbedingung der `while`-Schleife Wert zu legen. Die Auftragsliste stellt nämlich nicht sicher, daß der Schreibende und Lesende synchronisiert werden. Der Schreibende kann also unabhängig von der Anzahl der Aufträge neue Aufträge einfügen. Die Ausführungsumgebung muß sicherstellen, daß neben dem Schreibenden auch der Lesende die CPU erhält. Neben dem Zuteilungsproblem kann unter Umständen der Speicher mit der immer größer werdenden Liste belegt werden, so daß der Ausführungsumgebung nicht mehr genügend freien Speicher zur Verfügung steht.

Dies könnte dadurch gelöst werden, daß die Liste auf eine maximale Anzahl von Elementen begrenzt wird. Ist diese Obergrenze erreicht, müssen auch die Schreiber warten, bis die lesenden Threads einige Elemente aus der Liste entnommen haben.

Wird beim `FIFOQueueWriter` die `interrupt`-Methode aufgerufen, dann wird die Abarbeitung der `while`-Schleife abgebrochen, da die `Thread.interrupted()`-Methode wahr wird und somit der Thread terminiert.

```
public class FIFOQueueWriter
 implements Runnable {
  private FIFOQueue queue= null;
  public FIFOQueueWriter(FIFOQueue queue) {
    super();
    this.queue= queue;
  }
  public void run() {
    int counter= 0;
    while (!Thread.interrupted()) {
      counter++;
      queue.queueObject("Job " + counter);
      System.out.println("Writer: Job " +
           counter);
    }
    System.out.println("Writer: Term...");
  }
}
```

Programmausdruck 4.9: In die Liste schreibender Prozeß

Der lesende Thread beendet sich erst, wenn alle Aufträge aus der Liste abgearbeitet sind. Dazu ist die Variable `shouldTerminate` notwendig. In ihr wird vermerkt, daß der Thread ein Interrupt-Signal empfangen hat und nur noch solange weitere Aufträge aus der Liste abarbeitet, bis die Liste das erste Mal seit dem Empfang des Interrupt-Signals leer ist.

Die hier vorgestellten lesenden und schreibenden Threads verwendet das im Programmausdruck 4.11 gezeigte Testprogramm. Nach der Erzeugung der Auftragsliste und den bei-

```
public class FIFOQueueReader
 implements Runnable {
  private FIFOQueue queue= null;
  private boolean shouldTerminate=false;
  public FIFOQueueReader(FIFOQueue queue) {
    super();
    this.queue= queue;
  }
  public void run() {
    while (!shouldTerminate || queue.peek()) {
      try {
        System.out.println("Reader:␣" + queue.
            retrieveObjectInterruptible());
      } catch (InterruptedException e) {
        if (queue.peek()) {
          break;
        } else {
          System.out.println("Reader:␣Elemente␣
              enthalten");
          shouldTerminate=true;
        }
      }
    }
    System.out.println("Reader:␣Term...");
  }
}
```

Programmausdruck 4.10: Lesender Prozeß, der die `FIFOQueue` verwendet

```
public static void main(String[] args) {
  FIFOQueue queue= new FIFOQueue();
  FIFOQueueReader reader= new
      FIFOQueueReader(queue);
  FIFOQueueWriter writer= new
      FIFOQueueWriter(queue);
  ThreadGroup group= new ThreadGroup("
      QueueWorker");
  Thread threadWriter= new Thread(group,
      writer, "Writer");
  Thread threadReader= new Thread(group,
      reader, "Reader");
  threadReader.start();
  threadWriter.start();
  try {
    Thread.sleep(50);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  group.interrupt();
  try {
    threadWriter.join();
    threadReader.join();
  } catch (InterruptedException e1) {
    e1.printStackTrace();
  }
}
```

Programmausdruck 4.11: Testprogramm für die `FIFOQueue`

den Threads werden diese gestartet und wenige Zeit später unterbrochen.

Dazu werden beide Threads in eine Gruppe (`ThreadGroup`) eingefügt. Die Gruppe ist unter anderem für die Weiterleitung des Interrupt-Signals an alle enthaltenen Threads verantwortlich. Somit genügt hier der Aufruf `group.interrupt()`. Nachdem die Threads der Gruppe die `run`-Methode verließen, terminiert die Anwendung.

Semaphore in Java

Die im vorangegangenen Abschnitt vorgestellten Monitore sind zur Synchronisation von einzelnen Objekten geeignet. Sind jedoch an den zu schützenden Transaktionen sehr viele Objekte beteiligt, kann die Synchronisation mittels Monitoren schnell unübersichtlich werden.

Mit Hilfe von Monitoren lassen sich auch andere Mechanismen zur Koordination von Prozessen bilden. So zum Beispiel auch die im Abschnitt 2.4.2 vorgestellten Semaphore. Mit ihnen lassen sich bestimmte, frei definierbare Programmblöcke synchronisieren.

Die einfachste Form eines Semaphors ist das binäre Semaphor. Es kennt nur zwei Belegungsvarianten, nämlich "*Frei*" und "*Belegt*". Die im Programmausdruck 4.12 angegebene Umsetzung in Java benutzt Monitore zur Synchronisation dieses Semaphors.

Die `lock()`-Methode kehrt erst zurück, wenn das Semaphor frei ist. Andernfalls wird der aufrufende Prozeß in die Warteschlange des Objektmonitors eingereiht. Mit der Rückkehr dieser Methode hat der Prozeß den Semaphor verschlossen und kann somit als einziger die zu schützende Transaktion durchführen.

Ist die Transaktion erledigt, kann das Semaphor mit der `release()`-Methode wieder aufgeschlossen werden.

Dabei ist besonderes Augenmerk auf die Art

```
public class BinarySemaphore {
  private boolean free = true;
  public BinarySemaphore() {
    super();
  }
  public synchronized void lock()
   throws InterruptedException {
    while (false == free) {
      wait();
    }
    free = false;
  }
  public synchronized void release() {
    free = true;
    notifyAll();
  }
}
```

Programmausdruck 4.12: Binäres, mit Monitoren synchronisiertes Semaphor

und Weise der Entsperrung zu legen. Nach dem Ende der Transaktion muß das Semaphor auf jeden Fall - also auch bei auftretenden Ausnahmen - wieder aufgeschlossen werden. Ein möglicher ausführender Programmtext ist in Programmausdruck 4.13 gezeigt.

```
semaphore.lock();
try {
  transaction.doThis()
} catch (Throwable e) {
  throw e;
} finally {
  semaphore.release();
}
```

Programmausdruck 4.13: Sperren und Entsperren eines Semaphors

4.4 Realisierung der "Speisenden Philosophen"

Das im Abschnitt 3.1.2 vorgestellte Problem der speisenden Philosophen wird im folgenden implementiert und die beteiligten Java-Klassen werden diskutiert.

Die speisenden Philosophen sitzen gemeinsam an einem Tisch. Zwischen zwei Philosophen liegt nur ein Eßbesteck. Während des Essens versuchen die Philosophen, die Eßbestecke rechts und links von ihnen zu bekommen, um damit zu essen. Nach einer Weile legt der speisende Philosoph das Besteck ab, um im Anschluß daran über ein Problem nachzudenken, bis er wieder hungrig wird.

Die Abbildung in das Modell identifiziert die Eßbestecke als exklusiv nutzbare Betriebsmittel und die Philosophen als eigenständige Prozesse. Sie versuchen konkurrierend, die Eßbestecke zu benutzen.

Die Abbildung des Problems auf Programm-Code soll möglichst verklemmungsfrei sein. Allerdings ist die Erfüllung dieser Anforderung bei diesem Problem nicht trivial, wie später noch erläutert werden wird.

Eßbesteck Beim Eßbesteck handelt es sich um das exklusiv zu benutzende Betriebsmittel. Es kann schon in Benutzung sein oder ist noch frei. Ein Philosoph muß unter Umständen auf sein Eßbesteck warten.

Dies legt nahe, zur Darstellung eine binäre Semaphore zu verwenden. Die Klasse `BinarySemaphore` aus Abschnitt 4.3.3 wird noch um einige Informationen angereichert, wie der Programmausdruck 4.14 zeigt.

```
public class Cutlery extends BinarySemaphore
      {
  private String name= null;
  private int number= 0;
  public Cutlery(String name, int number) {
    this.name= name;
    this.number= number;
  }
  public String toString() {
    return name + number;
  }
}
```

Programmausdruck 4.14: Eßbesteck der Philosophen

Mit der `lock`-Methode läßt sich das Besteck reservieren, und mit der `release`-Methode wieder freigeben.

Hungrige Philosophen Die Philosophen sind nach einer Denkpause hungrig und möchten beide Eßbestecke zu ihren Seiten aufneh-

men. Beim Streit um diese Betriebsmittel kann der Fall einer Verklemmung (*Deadlock*) auftreten. Zunächst sei der Programmausdruck 4.15 gezeigt, der das Verhalten eines Philosophen abbildet.

```
public class Philosoph
 implements Runnable {
  private Random random= new Random();
  private Cutlery rightCutlery= null;
  private Cutlery leftCutlery= null;
  private String name= null;
  public Philosoph(String name, Cutlery
       rightCutlery, Cutlery leftCutlery) {
    this.rightCutlery= rightCutlery;
    this.leftCutlery= leftCutlery;
    this.name= name;
  }
  public void run() {
    try {
      while (!Thread.interrupted()) {
        printMsg("THINKING");
        Thread.sleep(100);
        printMsg("Try␣Eating␣with:␣" +
             printCutlery());
        if (random.nextBoolean()) {
          rightCutlery.lock();
          leftCutlery.lock();
        } else {
          leftCutlery.lock();
          rightCutlery.lock();
        }
        printMsg("EATING␣with:␣" +
             printCutlery());
        Thread.sleep(100);
        leftCutlery.release();
        rightCutlery.release();
      }
    } catch (InterruptedException e) {
      printMsg("Interrupted!");
    }
    printMsg("Finished");
  }
  private void printMsg(String msg) {
    System.out.println(name + ":␣" + msg);
  }
  private String printCutlery() {
    return "R:␣" + rightCutlery + "␣L:␣" +
         leftCutlery;
  }
  public String toString() {
    return name + ":␣" + printCutlery();
  }
}
```

Programmausdruck 4.15: Klasse `Philosoph`

Jeder Philosoph wird mit einem Namen und jeweils einem rechten und linken Eßbesteck initialisiert. Während des Eßvorgangs wechseln sich die Eß- und Denkphasen gegenseitig ab.

Nach der Denkphase versucht der Philosoph an das Eßbesteck heranzukommen. Um hier Verklemmungen zu vermeiden, wird durch einen zufällig erzeugten Wahrheitswert, der entweder `0` oder `1` sein kann, die Reihenfolge der Eßbesteckaufnahme vertauscht.

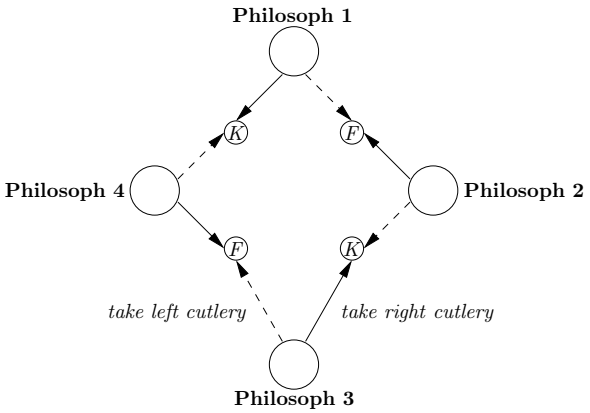


Abbildung 4.3: Verklemmung bei den speisenden Philosophen

Geschieht dies nicht, entsteht eine Verklemmung, wie sie in Abbildung 4.3 gezeigt ist: Jeder Philosoph versucht, das rechts von ihm liegende Eßbesteck vor dem links von ihm liegenden zu reservieren. Reservieren nun alle Philosophen das rechte Eßbesteck, warten alle unendlich auf das linke Eßbesteck, das ja das rechte des Nachbarn ist.

Dieser Determinismus bei der Besteckaufnahme hat demnach eine Verklemmung zur Folge. Die Reihenfolge der Eßbesteckaufnahme geschieht hier zufällig ausgewählt, also indeterministisch. Somit kann gezielter Indeterminismus als ein Mittel zur Vermeidung von Verklemmungen angesehen werden.

Simulation am Eßtisch Für einen Test der präsentierten Klassen `Cutlery` und `Philosoph` übernimmt der Programmausdruck 4.16 die Erzeugung der notwendigen Objekte des Beispielprogramms. Das Programm erzeugt nach seinem Start Ausschriften in der Standardausgabe.

```
public class Table {
  private static Thread[] getPhilosopherThreads
      (
    Philosoph[] philosophers,
    ThreadGroup group) {
    Thread[] threads=
     new Thread[philosophers.length];
    for (int i= 0; i < philosophers.length; i++) {
      threads[i]= new Thread(group,
       philosophers[i]);
    }
    return threads;
  }
  public static void simulatePhilosophers(
    int halfCountPhilosophers) {
    int countPhilosophers=
     2 * halfCountPhilosophers;
    // Bestecke
    Cutlery[] cutlery=
     new Cutlery[countPhilosophers];
    for (int i= 0; i < cutlery.length; i++) {
      if (i % 2 == 0) {
        cutlery[i]= new Cutlery("F", i);
      } else {
        cutlery[i]= new Cutlery("K", i);
      }
    }
    // Philosophen
    Philosoph[] philosophers=
     new Philosoph[countPhilosophers];
    for (int i= 0; i < philosophers.length; i++) {
      philosophers[i]= new Philosoph(
       "P " + i,
       cutlery[i],
       cutlery[(i + 1) % countPhilosophers]);
    }
    // Threads erzeugen
    ThreadGroup group=
     new ThreadGroup("Philosophers");
    Thread[] threads=
     getPhilosopherThreads(philosophers, group);
    // Starten
    for (int i= 0; i < threads.length; i++) {
      threads[i].start();
    }
    // Warten
    try {
      Thread.sleep(500);
    } catch (InterruptedException e) {
      System.err.println("Interrupted! " + e);
    }
    // Beenden
    group.interrupt();
    for (int i= 0; i < threads.length; i++) {
      try {
        threads[i].join();
      } catch (InterruptedException e1) {
        System.err.println(
         "Join on Thread " + i +
         " interrupted!" + e1);
      }
    }
    System.out.println("Meal Finished");
  }
  public static void main(String[] args) {
   simulatePhilosophers(10);
  }
}
```

Programmausdruck 4.16: Simulation eines Mahls mit Philosophen

Die `simulatePhilosophers`-Methode erwartet als Argument einen Wert, der dann verdoppelt wird. Die Idee dahinter ist die Tatsache, daß durch eine gerade Anzahl von Philosophen jeder Philosoph jeweils ein Messer und eine Gabel neben sich liegen hat. Abgesehen davon müssen ohnehin die Hälfte der beteiligten Philosophen Linkshänder sein, und jeweils ein Linkshänder muß einen Rechtshänder als Tischnachbarn haben. Soviel zu den Voraussetzungen der Simulation.

Begonnen wird mit der Erstellung der Eßbestecke. Der Buchstabe "`F`" symbolisiert eine Gabel (*Forc*), und "`K`" steht für das Messer (*Knife*).

Die Philosophen werden in der nachfolgenden `for`-Schleife erzeugt. Jeder Philosoph erhält ein rechtes und linkes Eßbesteck. Nachfolgend wird eine Thread-Gruppe mit den erzeugten Philosophen-Threads erstellt. Die Simulation kann gestartet werden.

Zu Beginn des Mahls werden alle Philosophen-Threads gestartet. Nun beginnt das vom Denken unterbrochene Essen mit dem knappen Besteck. Die Philosophen speisen abwechselnd mit dem Besteck.

Nach einer Weile wird dem Wirt das bunte Treiben zuviel. Er räumt ab, bringt die Rechnung und wartet, bis alle Philosophen gegangen sind.

Das Beenden der Philosophen-Threads geschieht durch den Aufruf der `group.interrupt()`-Methode. Der Main-Thread wartet danach auf das Ende der

Threads, um danach das Programm zu terminieren.

Die Ausgabe des Simulationsprogramms sieht mit vier speisenden Philosophen so aus:

```
P 0: THINKING
P 1: THINKING
P 2: THINKING
P 3: THINKING
P 0: Try Eating with: R: F0 L: K1
P 0: EATING with: R: F0 L: K1
P 1: Try Eating with: R: K1 L: F2
P 2: Try Eating with: R: F2 L: K3
P 2: EATING  with: R: F2 L: K3
P 3: Try Eating with: R: K3 L: F0
P 0: THINKING
P 3: EATING with: R: K3 L: F0
P 1: EATING with: R: K1 L: F2
P 2: THINKING
P 0: Try Eating with: R: F0 L: K1
P 3: THINKING
P 1: THINKING
P 0: EATING with: R: F0 L: K1
P 2: Try Eating with: R: F2 L: K3
P 2: EATING with: R: F2 L: K3
P 3: Try Eating with: R: K3 L: F0
P 1: Try Eating with: R: K1 L: F2
P 0: THINKING
P 3: EATING with: R: K3 L: F0
P 1: EATING with: R: K1 L: F2
P 2: THINKING
P 0: Interrupted!
P 0: Finished
P 1: Interrupted!
P 1: Finished
P 2: Interrupted!
P 2: Finished
P 3: Interrupted!
P 3: Finished
Meal Finished
```

4.5 Abschließende Bemerkungen

Die Programmiersprache Java wird durch einen Interpreter ausgeführt. Der durch den Java-Compiler erzeugte Bytecode wird von der Ausführungsumgebung, der Java Virtual Machine (JVM) interpretiert und ausgeführt. Ein weiterer Bestandteil der Ausführungsumgebung ist der Hotspot-Compiler, der Optimierungen vornimmt. Als Ergebnis dieser Optimierungen wird der Programmablauf beschleunigt.

Das Speichermodell der Java-Ausführungsumgebung ist in der Sprachspezifikation [7] beschrieben. Bisher existiert kein formalisiertes Speichermodell. Dieser Umstand wird in einigen Veröffentlichungen erkannt [16] und wurde auch schon als Vorschlag dem Java-Entwicklungskonsortium in Form eines *Java Specification Request* (JSR) unterbreitet [14].

Java-Bytecode kann auf sehr vielen Betriebssystemen und Endgeräten ausgeführt werden. Für jede dieser Plattformen ist eine eigene JVM erforderlich. Sie hat unter anderem die Aufgabe, die spezifizierten Eigenschaften des Speichermodells in Java mit dem Speichermodells des jeweiligen Zielprozessors und des Betriebssystems in Einklang zu bringen. Dieses Vorhaben scheitert in einigen Punkten aus verschiedenen Gründen. Zum Beispiel sind die verwendeten Bibliotheken nicht ablaufinvariant (*reentrant*) ausgelegt oder der Befehlssatz des Prozessors reicht nicht aus.

Weiterhin entstehen durch eine Neuordnung der Ausführungsreihenfolge der im Java-Bytecode enthaltenen Befehle bis jetzt nicht lösbare Probleme.

Eines dieser Probleme befaßt sich mit der Parallelisierung der Erzeugung einer einzigen Objektinstanz. Nebenläufig versuchen Prozesse, durch den Aufruf einer Methode eine Instanz eines Objekts zu erzeugen. Dieses Objekt wird auch als *Singleton* bezeichnet. Der Name dieser Problems lautet "**Doppelt überprüfte Blockierung**" (*double-checked locking*). Der Autor von [14] zeigt in einem Essay, daß momentan keine Lösung für dieses Problem möglich ist.

Andere, nicht interpreterbasierte Sprachen, wie zum Beispiel die Programmiersprache C, weisen nicht diesen Mangel auf. Der C-Compiler erzeugt sofort dem jeweiligen Prozessor verständlichen Code und nutzt die Funktionen der Betriebssystembibliotheken direkt. Im Gegensatz zu Java kann man im Programmcode Assembler-Anweisungen mit Prozessordirektiven angeben, um zum Beispiel eine explizite Speicherbarriere anzufordern (`asm('memoryBarrier')`).

Um in eigenen Programmen die hier geschilderten Grenzsituationen zu vermeiden, sollten bei nebenläufig agierenden Threads keine Wettlaufsituationen (*race conditions*) beim Lesen und Schreiben von Variablen auftreten. Dies ist dadurch zu erreichen, daß alle konkurrierenden Zugriffe auf Variablen synchronisiert werden.

Kapitel 5

Zusammenfassung

Die für die Programmierung notwendigen Grundkenntnisse im nebenläufigen Programmieren wurden in ihren Ansätzen im ersten Teil des Berichts dargestellt (Kapitel 2). Dazu zählten eben der Darstellung von ausgewählten Möglichkeiten zur Prozeßbeschreibung auch eine Übersicht über bekannte Verfahren zur Synchronisation. Auf die in der Praxis häufig auftretenden Verklemmungen von Prozessen wird ebenfalls eingegangen.

Im Mittelteil (Kapitel 3) schließt sich die Vorstellung von zwei Modellen zur Abbildung und Simulation von nebenläufigen Prozessen an. Dies war zum einen der Petri-Netz-Simulator und ein Werkzeug zum Simulieren von Systemen, die mit SDL beschrieben sind.

In der Praxis ist es jedoch schon allein aus Zeit- und Kostengründen nicht möglich, jeden kleinen Prozeß in einem größeren Programm vor der Realisierung zunächst zu modellieren und simulieren. Vielmehr sollten einige Fallstudien an ausgewählten Szenarien durchgeführt werden, um das allgemeine Verständnis für Betriebsmittel und konkurrierende Prozesse zu erhöhen. Ein derart geschultes Auge erkennt bei der Realisierung einen Großteil der zur Synchronisation notwendigen Strukturen und deren Einsatzpunkte im Quelltext.

Das abschließende Kapitel 4 zeigt, welche theoretischen Ansätze zur Synchronisation von Prozessen und des Zugriffs auf gemeinsam benutzte Daten in der Programmiersprache Java umgesetzt wurden. Dazu wird das Speichermodell von Java erläutert und kritisch betrachtet.

Zur Illustration werden einige häufig benötigten Beispielprogramme angegeben und analysiert. Das Problem der speisenden Philosophen wurde ebenfalls in Java realisiert, und zwar mit einer etwas ungewöhnlichen Lösung zum Vermeiden einer möglichen Verklemmung.

Literaturverzeichnis

[1] Sophia Antipolis. *Methods for Testing and Specification (MTS); Overview of validation techniques for European Telecommunication Standards (ETSs) containing SDL*. Techn. Ber., ETSI, 1995.

[2] E.G. Coffman, M.J. Elphick und A. Shoshani. *System Deadlocks*. In *ACM Computing Surveys*, Bd. 3. 1971.

[3] E.W. Dijkstra. *Cooperating Sequential Processes*. In F. Genuys, Hg., *Programming Languages*. Academic Press, New York, 1968.

[4] E.W. Dijkstra. *Hierarchial Ordering of Sequential Processes*. In C.A.R. Hoare und R.H. Perrott, Hg., *Operating Systems Techniques*. Academic Press, New York, 1971.

[5] Berd Däne. *Petri-Netz-Simulator Peneca Chromos*. Technische Universität Ilmenau. URL www.theoinf.tu-ilmenau.de/. URI `ra1/skripte/pn_chr/`.

[6] *Abstract Syntax Notation Number One (ASN.1)*. France Telecom. URL asn1.elibel.tm.fr/en/.

[7] James Gosling, Bill Joy und Guy Steele. *The Java Language Specification*. GOTOP Information Inc., 1996.

[8] Gregor Gärtner. *Replikationskonzept für ein verteiltes Diskurssystem im Internet*. Diplomarbeit, Technische Universität Ilmenau, April 2001.

[9] Ralf Guido Herrtwich und Günter Hommel. *Nebenläufige Programme*. Springer Verlag, 1994.

[10] Stefan Jaksch. *Cooperative Garbage Collection in einer großen Web-Proxy-Installation*. Diplomarbeit, Technische Universität Ilmenau, September 1999.

[11] E. Knapp. *Deadlock Detection in Distributed Databases*. In *ACM Computing Surveys*, Bd. 19. 1987.

[12] S.S. Patil. *Limitations and Capabilites of Dijkstras Semaphore Primitives for Coordination Among Processes*. Memo 57, Computational Structures Group, Cambridge, 1971.

[13] Carl Adam Petri. *Kommunikation mit Automaten*. Diplomarbeit, Schriften des Institutes für instrumentelle Mathematik, 1962.

[14] William Pugh. *Fixing the Java Memory Model*. In *Java Grande*, S. 89–98. 1999. URL citeseer.nj.nec.com/. URI `/pugh99fixing.html`.

[15] Christian Ullenboom. *Java ist auch eine Insel - Programmieren für die Java 2-Plattform in der Version 1.4*. Galileo Press, 3. Aufl., Mai 2003.

[16] Yue Yang, Ganesh Gopalakrishnan und Gary Lindstrom. *Analyzing the CRF Java Memory Model*. School of Computing, University of Utah.