Synchronisation langlebiger Aktivitäten in persistenten Objektsystemen

Diplomarbeit

von

Andreas Piellusch

Betreuer:

Prof. Dr. J. W. Schmidt

Dr. M. Lehmann

Fachbereich Informatik

Universität Hamburg

Juni 1996

Ich danke Bernd Mathiske für die vielen interessanten und hilfreichen Diskussionen über die Themen dieser Arbeit sowie die vielfältige Unterstützung und Betreuung während ihrer Durchführung. Ebenso danke ich Gerald Schröder für die schriftliche und mündliche Betreuung, die mir gerade in der Endphase dieser Arbeit eine große Unterstützung gewesen ist. Prof. Dr. Schmidt möchte ich für die Unterstützung, das Interesse am Thema dieser Arbeit und für die guten Bedingungen am Arbeitsbereich DBIS danken. Dr. Lehmann danke ich dafür, daß er mich auf einige Unstimmigkeiten der schriftlichen Version meiner Arbeit aufmerksam gemacht hat und für seine Ausführungen über viele Teilbereiche des Arbeitsthemas. Dr. Florian Matthes danke ich für seine Hilfe bei der Festlegung des Aufbaus und des Zeitplans dieser Arbeit.

Für Martina

Inhaltsverzeichnis

1 Einführung und Überblick **1**

1.1 Zielsetzung 3

1.2 Aufbau der Arbeit 4

2 Tycoon **7**

2.1 Der Aufbau des Tycoon-Systems 8

2.2 Die Programmiersprache TL 9

2.3 Module und Schnittstellen 13

2.4 Speicherprotokoll und persistenter Objektspeicher 15

2.5 Langlebige Aktivitäten 17

3 Synchronisation nebenläufiger Prozesse **19**

3.1 Nebenläufigkeit in Rechensystemen 19

3.2 Synchronisationsmechanismen 21

3.3 Basismethoden 22

3.3.1 Ununterbrechbarer Speicherzugriff 22

3.3.2 Unterbrechungssperren 23

3.3.3 Atomare Prozessorinstruktionen 24

3.3.4 Zusammenfassung 27

3.4 Erweiterte Methoden 27

3.4.1 LOCK und UNLOCK 27

3.4.2 Semaphore 28

3.4.3 Zählen von Ereignissen 31

3.4.4 Zusammenfassung 33

3.5 Synchronisation in Programmiersprachen 33

3.5.1 Kritische Regionen 34

3.5.2 Monitore 35

3.5.3 Kommunikationskanäle und Rendezvous 38
3.5.4 Weitere Mechanismen 40
3.5.5 Zusammenfassung 41
3.6 Synchronisation in persistenten Objektsystemen 42

4 Synchronisationsmechanismen für persistente Objektsysteme 43
4.1 Anforderungen und Auswahl 43
4.1.1 Eigenschaften des Tycoon-Systems 44
4.1.2 Anforderungen an Synchronisationsmechanismen 44
4.1.3 Aufbau der Mechanismen 45
4.2 Bibliotheken für Synchronisationsmechanismen 48
4.2.1 Semaphore 49
4.2.2 Mutexe und Bedingungsvariable 51
4.3 Implementation im Tycoon-System 56
4.3.1 Das Laufzeitsystem 56
4.3.2 Interner Aufbau der Synchronisationsmechanismen 60
4.3.3 Tycoon-Threads und Standards 64
4.4 Realisierung weiterer Synchronisationsmechanismen 67
4.4.1 Kommunikationskanäle 67
4.4.2 Rendezvous 70
4.4.3 Syntaxerweiterungen 72

5 Synchronisation migrierender Threads 77
5.1 Implementation der Thread–Mobilität 77
5.1.1 Entfernte Funktionsaufrufe 78
5.1.2 Threadmigration 80
5.2 Die Minimierung transitiver referentieller Abhängigkeiten 84
5.3 Zustandswechsel migrierender Threads 86

6 Zusammenfassung und Ausblick 89
6.1 Erfahrungen mit dem Tycoon-System 90
6.2 Der Stand der Implementierung 90
6.3 Resümee und Ausblick 91

A Ausgewählte Modulschnittstellen **93**

A.1 Das Modul Mutex 93

A.2 Das Modul Condition 95

A.3 Das Modul Rendezvous 96

A.4 Das Modul Thread 97

B Syntaxerweiterungen **101**

B.1 Grammatik für Rendezvous 101

Literaturverzeichnis **103**

Abbildungsverzeichnis

2.1 Schematischer Aufbau des Tycoon-Systems 8
2.2 Threadmigration 18

3.1 Isolation vs. Kooperation 20
3.2 Hierarchie der Synchronisationsmethoden 21

4.1 Synchronisationsmethoden und Laufzeitsystem 46
4.2 Unterbindung eines Koroutinenwechsels 47
4.3 Aufbau und Zuordnung der Synchronisationsmethoden 48
4.4 Sperrung kritischer Abschnitte 53
4.5 Synchronisation an einer Barriere 54
4.6 Schematischer Aufbau des Laufzeitsystems 57
4.7 Anordnung der aktiven Koroutinen 58
4.8 Zustandsübergänge der Tycoon-Threads 60

5.1 Migration mehrerer Threads 83
5.2 Übertragung eines Objekts durch Tiefenkopie 84
5.3 Rebinden ubiquitärer Objekte 86
5.4 Zustandsübergänge unter Berücksichtigung von Migration 87

Kapitel 1

Einführung und Überblick

Eine allgemeine Definition des Begriffs *Prozeß* liefert die DIN 66200. Demnach bezeichnet man als einen *Prozeß* [DIN 81]:

> "Die Umformung und/oder den Transport von Materie, Energie und Informationen".

Für die Informatik lautet eine umgangssprachliche, enger gefaßte Definition [Engesser 88]:

> Ein Prozeß ist der "Vorgang einer algorithmisch ablaufenden Informationsbearbeitung".

Hoare definiert einen Prozeß als das Verhaltensmuster eines Objekts über einer Menge von Ereignissen, dem Alphabet des Prozesses [Hoare 85]:

> "...to use the word *process* to stand for the behaviour pattern of an object, insofar as it can be described in terms of the limited set of events selected as its alphabet".

Unter einem sequentiellen Prozeß versteht man dann, bezogen auf ein Rechnersystem, die sequentielle Ausführung eines gegebenen Programms.

Als nebenläufige Prozesse werden Vorgänge oder Aktivitäten bezeichnet, die unabhängig voneinander ablaufen können. Für den Ablauf nebenläufiger Vorgänge ist es zunächst unbedeutend, in welcher zeitlichen Reihenfolge dies geschieht. Zum Beispiel können zwei gegebene, nebenläufige Vorgänge A und B in beliebiger Folge nacheinander, mit zeitlicher Überschneidung oder gleichzeitig ausgeführt werden. *Nebenläufigkeit* ist somit auch ein allgemeinerer Begriff als der Begriff der *Parallelität*, der den gleichzeitigen Ablauf von Vorgängen beschreibt, und beinhaltet auch den Begriff der *Sequentialität*, der den aufeinanderfolgenden Ablauf von Vorgängen definiert.

Die in der realen Welt ablaufenden Vorgänge sind im allgemeinen zueinander nebenläufig. Sequentielle Vorgänge ohne nebenläufige Handlungen stellen dagegen eher Sonderfälle dar. Für Rechnersysteme galt diese Aussage hingegen lange Zeit nicht. Die ersten Rechnersysteme der vierziger und frühen fünfziger Jahre erlaubten beispielsweise weder die parallele, beziehungsweise nebenläufige Ausführung von Programmen, noch die nebenläufige Benutzung

von Prozessoreinheit und Ein- und Ausgabegeräten [Stallings 92]. Gänzlich anders stellt sich die heutige Situation in Rechensystemen dar. Viele Systeme erlauben eine nebenläufige Ausführung von Programmen. Nebenläufigkeit kann in Rechensystemen in unterschiedlicher Ausprägung auftreten. Unterschiede in der Implementation von Nebenläufigkeit bestehen in folgenden Punkten:

- Nebenläufige Prozesse dienen zur gleichzeitigen oder überlappenden Lösung voneinander unabhängiger Aufgaben. Die Prozesse arbeiten in Isolation voneinander, es besteht keine Notwendigkeit der Kommunikation untereinander. Konflikte, die bei der gemeinsamen Nutzung von Ressourcen des Rechensystems auftreten können, werden von einer übergeordneten Instanz, beispielsweise einem Betriebssystem, aufgelöst.

- Nebenläufige Prozesse arbeiten gemeinsam und kooperativ an der Lösung von Aufgaben. Diese Art der Nebenläufigkeit erfordert eine Abstimmung der beteiligten Prozesse. Prozesse, die in einem gemeinsamen Adreßraum ablaufen, kommunizieren über die Benutzung gemeinsamer Datenstrukturen miteinander. Prozesse, deren Adreßräume voneinander getrennt sind, tauschen Nachrichten über gesonderte Kommunikationskanäle aus und beeinflussen sich somit gegenseitig.

Ein Beispiel für nebenläufige Vorgänge mit einer Trennung von Adreßräumen stellen Betriebssystemprozesse dar. Diese erwünschte, weitestgehende Trennung und Isolierung der Adreßräume sichert die voneinander ungestörte Ausführung von Prozessen. Eine Überlagerung von Teilen von Adreßräumen ist nur durch explizite Deklaration von gemeinsam benutzten Speicherbereichen möglich. Eine andere Art der Nebenläufigkeit existiert in der in modernen Betriebssystemen und Programmiersprachen möglichen, prozeßinternen Nebenläufigkeit. In einem Betriebssystemprozeß können gleichzeitig mehrere unterschiedliche Ausführungspfade desselben Programms existieren. Derartige Ausführungspfade innerhalb eines solchen Programms werden in der englischsprachigen und oft auch in der deutschsprachigen Literatur im allgemeinen als *Threads* bezeichnet[1]. Im Gegensatz zu isolierten Betriebssystemprozessen befinden sich daher die zu einem Prozeß gehörenden Threads im Adreßraum des Prozesses und besitzen gleichberechtigte Zugriffsmöglichkeiten auf alle prozeßinternen Datenstrukturen.

Eine Synchronisation nebenläufiger Aktivitäten, also von Prozessen und Threads, wird notwendig, wenn diese Aktivitäten miteinander kommunizieren. So ist es beispielsweise bei der kooperativen Bearbeitung von Datenbeständen erforderlich, daß ein Prozeß eine bestimmte Handlung beendet hat, bevor ein anderer Prozeß die Bearbeitung weiterführen kann. Synchronisation bedeutet daher die kontrollierte Einschränkung von Nebenläufigkeit, die Abstimmung oder Sequentialisierung von Handlungen.

Seit Beginn der Entwicklung von Rechensystemen und auch heute steht die Erforschung geeigneter Mechanismen zur Synchronisation nebenläufiger Vorgänge im Zentrum vieler wissenschaftlicher Arbeiten. Die Synchronisation nebenläufiger Vorgänge, also gerade die kontrollierte Einschränkung von Nebenläufigkeit, wurde jedoch vor allem im Kontext von Betriebssystemen und Datenbanken betrachtet. Mit den seit einigen Jahren auch für die Anwendungsentwicklung verfügbaren, programmiersprachlichen Konstrukten zur Parallelisie-

[1] Von einigen Autoren werden auch Bezeichnungen wie die wörtliche Übersetzung von *Thread* als *Ausführungsfaden* und von *multithreaded application* als *gefädelte Anwendung* gewählt.

rung von Softwaresystemen erlangten auch die hierfür notwendigen Methoden zur Synchronisation entsprechende Bedeutung. Beispiele für Programmierkonstrukte, die nebenläufige Aktivitäten und deren Synchronisation unterstützen, sind in vielen neueren Programmiersprachen wie beispielsweise Modula-3 [Cardelli et al. 88], Obliq [Cardelli 94] und Java [Campione, Walrath 96] zu finden.

Neben dem allgemeinen Einsatz von Methoden zur Programmierung nebenläufiger Aktivitäten sind auch auf den Gebieten der Programmiersprachen und Datenbanken bedeutende Fortschritte erzielt worden. Eine besondere Klasse neuerer Programmiersysteme stellen persistente Objektsysteme dar [Matthes 93]. Diese Systeme bieten ihren Benutzern unter anderem die transparente und langlebige Speicherung von Programmobjekten unter dem Einsatz fortschrittlicher Eigenschaften der zugrundeliegenden Programmiersprachen. Hierzu gehören Eigenschaften wie Polymorphismus, Objektorientierung und Funktionen höherer Ordnung. Ein Beispiel für derartige Systeme ist das am Arbeitsbereich Datenbanken und Informationssysteme (DBIS) der Universität Hamburg entwickelte persistente Objektsystem Tycoon[2]. Neben den angeführten und weiteren Eigenschaften bietet dieses System eine Integration von Techniken zur Realisierung nebenläufiger Aktivitäten in einer persistenten Umgebung in Form persistenter Threads an [Matthes, Schmidt 94]. Diese persistenten Threads sind darüber hinaus in der Lage, ihren Adreßraum zu verlassen und in andere, entfernte Adreßräume zu migrieren [Mathiske et al. 95]. Die Verwendung migrierender und persistenter Threads gestattet dem Benutzer daher die Modellierung langlebiger, mobiler Aktivitäten, die

- kooperativ gemeinsame Aufgaben bearbeiten,
- in andere Adreßräume migrieren können sowie
- innerhalb eines gemeinsamen Adreßraums über die Benutzung gemeinsamer Objekte miteinander kommunizieren können.

Eine Kommunikation von Threads über gemeinsame Speicherinhalte muß jedoch mittels geeigneter Synchronisationsverfahren aufeinander abgestimmt werden. Dies ist bis heute, wie später erläutert wird, nur sehr eingeschränkt möglich. Methoden zur Synchronisation kooperativer, nebenläufiger und mobiler Aktivitäten im Rahmen persistenter Objektsysteme stehen im Mittelpunkt dieser Arbeit.

1.1 Zielsetzung

Das Ziel dieser Arbeit ist zunächst die Untersuchung und Bereitstellung von grundlegenden Methoden für die Synchronisation nebenläufiger Aktivitäten im Umfeld polymorpher und persistenter Objektsysteme. Das am Arbeitsbereich DBIS der Universität Hamburg entwickelte, polymorphe persistente Objektsystem Tycoon mit der dazugehörigen Programmiersprache TL[3] bietet mit seinen persistenten Threads die praktische Arbeitsgrundlage für das zu behandelnde Thema.

[2] Tycoon ist die Abkürzung für: Typed communicating objects in open environments.
[3] Tycoon Language.

Ausgehend vom allgemeinen Problem der Synchronisation von zueinander nebenläufig ablaufenden, sequentiellen Aktivitäten, werden zunächst bekannte und wohlverstandene klassische Synchronisationsmechanismen auf ihre Eignung und Einsatzmöglichkeit in der vorliegenden Umgebung untersucht.

Die im ersten Teil untersuchten und ausgewählten Synchronisationsmechanismen sollen für das Tycoon-System in geeigneter Weise implementiert werden. Hierbei sind im Gegensatz zu den in herkömmlichen Programmiersystemen herrschenden Bedingungen jedoch eine Anzahl von zusätzlichen Eigenschaften des Tycoon-Systems zu berücksichtigen. Zum einen ist dies die durch Tycoon bereitgestellte, transparente Persistenzeigenschaft aller TL-Sprachobjekte. Synchronisationmechanismen für persistente Objektsysteme müssen daher ebenfalls die Eigenschaft der Persistenz besitzen.

Weiterhin sind grundlegende Teile des vorliegenden Tycoon-Systems in der standardisierten Programmiersprache ANSI-C unter besonderer Berücksichtigung von Portabilitätsaspekten implementiert. Auch Synchronisationsmethoden für ein derartiges System müssen leicht auf weitere Systeme portierbar sein.

Die ausgewählten, grundlegenden Synchronisationsmechanismen ermöglichen bereits die Behandlung aller in Frage kommenden Synchronisationsprobleme, sind jedoch für die Entwicklung von komplexen Anwendungen auf einem ungeeignet niedrigen Abstraktionsniveau angesiedelt und können daher Ursache vieler, bei der Entwicklung von nebenläufigen Systemen möglicher, subtiler Fehler sein. Aus diesem Grund werden, aufbauend auf den Basismechanismen zur Synchronisation, eine Reihe von höheren Methoden zur Synchronisation untersucht und implementiert. Weiterhin werden Syntaxerweiterungen für die Sprache TL zur Realisierung sicher zu benutzender Synchronisationsverfahren verwendet.

Eine Behandlung von Synchronisationsproblemen ist ohne die Untersuchung der Aspekte verteilter Systeme unvollständig. Wie in der Einleitung bereits erwähnt, ist ein aktueller Gegenstand der Forschung am DBIS Arbeitsbereich die Realisierung verteilter, persistenter Systeme durch mobile Tycoon-Threads. Die Fähigkeit von Tycoon-Threads, über Systemgrenzen zu migrieren und ihre Aktivität in einem anderen Adreßbereich fortzusetzen, wirft für die Synchronisation eine Reihe von interessanten Fragestellungen auf. Insbesondere werden in der vorliegenden Arbeit mögliche Zustandswechsel der an der Synchronisation direkt und indirekt beteiligten Threads bei Migrationsvorgängen berücksichtigt. So ist zum Beispiel die Migration eines eine aktuelle Synchronisationssperre haltenden Threads auf ein anderes System in geeigneter Art zu behandeln, um die Lebendigkeit und Verklemmungsfreiheit der verbleibenden Aktivitäten zu gewährleisten. Weiterhin sollen die Synchronisationsmethoden ebenfalls die Fähigkeit zur Migration besitzen. Diese Eigenschaft erscheint zumindest für Situationen wünschenswert, in denen der Zugriff auf ebenfalls mobile Objekte des Tycoon-Systems synchronisiert werden muß. Die Ergebnisse der Untersuchung dieser Fragestellungungen fließen in die konkrete Implementation der Synchronisationsmechanismen ein.

1.2 Aufbau der Arbeit

Der Hauptteil der Arbeit beginnt mit einer Einführung in das Tycoon-System (Kapitel 2). Hier werden alle für diese Arbeit relevanten Aspekte des Tycoon-Systems vorgestellt; ins-

besondere wird auf persistente Threads und Thread-Migration eingegangen.

Im anschließenden Abschnitt "Synchronisation nebenläufiger Prozesse" (Kapitel 3) werden bekannte Mechanismen zur Synchronisation im Detail beschrieben. Zu diesem Zweck beginnt das Kapitel mit einer Erörterung wichtiger Aspekte der Nebenläufigkeit kooperativer und voneinander isolierter Aktivitäten. Nach einer Diskussion der Grundvoraussetzungen zur Synchronisation werden aufeinander aufbauende Synchronisationsmechanismen beschrieben.

Das Hauptaugenmerk der folgenden Kapitel ist auf die Auswahl und Implementation der im vorhergehenden Abschnitt vorgestellten Synchronisationsprimitive im Umfeld persistenter Umgebungen gerichtet (Kapitel 4). Nach der Diskussion der für Synchronisationsmechanismen relevanten Eigenschaften des Tycoon-Laufzeitsystems werden die zu implementierenden Synchronisationsprimitive im Detail in ihrer Syntax und Semantik beschrieben. Daran anschließend erfolgt die Beschreibung der konkreten Implementation der Mechanismen. Die in den vorangegangenen Abschnitten von Kapitel 4 implementierten Methoden bilden die Basis für die Entwicklung generischer und hochsprachlicher Synchronisationskonzepte (Abschnitt 4.4) sowie deren Integration in das Tycoon-System.

Die Fähigkeit von Tycoon-Threads zur Migration wirkt sich auch auf die Untersuchung und Implementation der Synchronisationsmechanismen aus. In Kapitel 5 werden Techniken zur Behandlung der damit verbundenen Probleme erörtert.

Eine Zusammenfassung der wichtigsten Ergebnisse und Erfahrungen (Kapitel 6) bildet den Abschluß der Arbeit.

Kapitel 2

Tycoon

Im Rahmen der vorliegenden Arbeit werden Synchronisationsmechanismen für langlebige, mobile Aktivitäten in persistenten Objektsystemen untersucht. Diese Arbeit wird im Rahmen des am Arbeitsbereich DBIS der Universität Hamburg laufenden Tycoon-Projekts durchgeführt. Das in diesem Projekt entwickelte Tycoon-System bildet dabei die technologische und implementatorische Basis dieser Arbeit. Im Tycoon-Projekt wird eine Verbesserung der Qualität persistenter Objektsysteme angestrebt. Dieses Ziel wird erreicht durch die:

> "spezielle sprachliche und architekturelle Unterstützung für die vollständig transparente Verwaltung langlebiger Daten- und Programmobjekte, die mengenorientierte Verarbeitung (benutzerdefinierter) Kollektionen, die Implementierung generischer Bibliotheken und die typsichere Anbindung externer Systemsoftware (z.B. SQL Datenbanken oder Fenstersysteme)" [Matthes 93].

Im diesem Kapitel werden der Aufbau und die Eigenschaften der für diese Arbeit relevanten Teile des Tycoon-Systems beschrieben. Eine Übersicht über die Organisation der funktionalen Schichten des Tycoon-Systems wird in Abschnitt 2.1 gegeben. Daran anschließend folgt eine Beschreibung der persistenten, polymorphen Programmiersprache des Systems (Abschnitt 2.2). Als ein Ergebnis dieser Arbeit werden für das Tycoon-System eine Reihe von Modulen zur Thread-Synchronisation implementiert. Der Aufbau des zugrundeliegenden Modulsystems wird in Abschnitt 2.3 angesprochen. Eine weitere zentrale Systemkomponente bilden das Speicherprotokoll TSP[1] und die über dieses Protokoll angesprochenen Objektspeicher (Abschnitt 2.4). Im letzten Abschnitt dieses Kapitels folgt eine Darstellung der durch das Tycoon-System gegebenen Möglichkeiten zur Formulierung langlebiger Aktivitäten (Abschnitt 2.5).

Spezielle, weitere Eigenschaften des Tycoon-Systems werden im nachfolgenden Kapitel intensiver behandelt. Eine ausführliche Diskussion des Tycoon-Laufzeitsystems erfolgt in Abschnitt 4.3.1, eine Erläuterung der Übertragung von TL-Objekten auf andere Systeme findet sich in Kapitel 5.

[1] Tycoon Store Protocol.

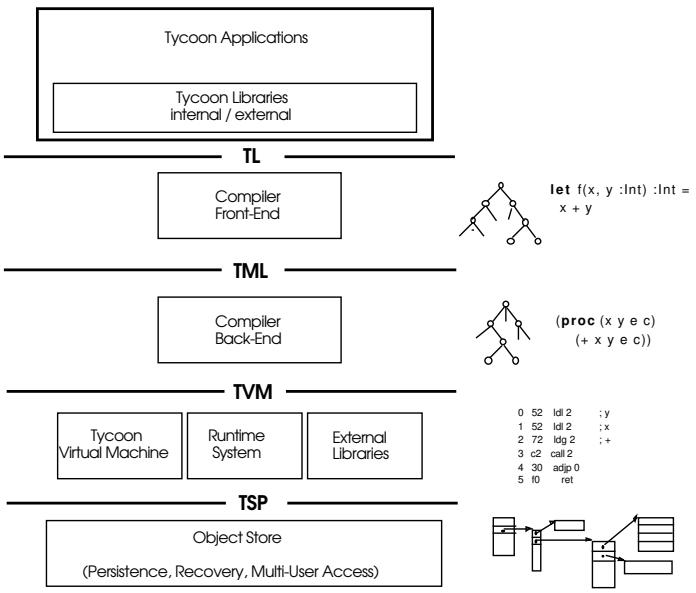

Abbildung 2.1: Schematischer Aufbau des Tycoon-Systems

2.1 Der Aufbau des Tycoon-Systems

Der Aufbau des Tycoon-Systems gliedert sich in mehrere spezialisierte Schichten. Die in Abbildung 2.1 sichtbaren, aufeinander aufbauenden Schichten des Tycoon-Systems bestehen dabei aus den Komponenten:

TL (Tycoon Language): Die Programmiersprache TL stellt die Schnittstelle des Systems zu den Programmierern dar. In dieser Sprache sind alle Bibliotheken und Anwendungen des Tycoon-Systems formuliert. Das Compiler-Front-End des Systems wandelt Ausdrücke der Sprache TL in eine Zwischenrepräsentation um. Diese besteht aus mit Typinformationen versehenen, abstrakten Syntaxbäumen. Diese Zwischenrepräsentation wird an das Compiler-Back-End weitergereicht.

TML (Tycoon Machine Language): TML ist die vom Compiler-Back-End erzeugte, maschinenunabhängige Zwischenrepräsentation [Gawecki, Matthes 94]. Auf dieser untypisierten Zwischensprache können verschiedene maschinenunabhängige Optimierungen durchgeführt werden. Die Ausgabe des Compiler-Back-End besteht aus vom Tycoon-Bytecodeinterpreter (TVM) direkt ausführbaren Code, dessen Befehlssatz an den der Transputer-Mikroprozessoren angelehnt ist [INMO87 87].

TVM (Tycoon Virtual Machine): Der vom Compiler-Back-End erzeugte Bytecode wird von der TVM direkt ausgeführt. Die TVM bedient sich zur Ausführung des Bytecodes der Dienste des Tycoon-Laufzeitsystems, das vom Betriebssystem unabhängige Funktionalität zur Verfügung stellt (vgl. Abschnitt 4.3.1).

TSP (Tycoon Store Protocol): Die gesamte Kommunikation der TVM und des Laufzeitsystems mit dem Objektspeicher findet über das Speicherprotokoll TSP statt. Alle

in der Programmiersprache TL erzeugten Daten, wie atomare Werte, Programmcode und Threads, werden in einem Objektspeicher persistent abgelegt. Der Objektspeicher dient weiterhin zur persistenten Speicherung verschiedener Datenstrukturen des Tycoon-Laufzeitsystems. Das Protokoll TSP gestattet die Anbindung verschiedener Objektspeicher über eine einheitliche Softwareschnittstelle.

Ein Ziel bei der Konzeption des Tycoon-Systems ist die Austauschbarkeit der beschriebenen, verschiedenen Komponenten. So existieren Konfigurationen des Systems, in denen verschiedene Compilerkomponenten und TSP-Implementationen benutzt werden.

2.2 Die Programmiersprache TL

Die Sprache TL [Matthes, Schmidt 92] ist eine polymorphe, funktionale Programmiersprache mit Funktionen höherer Ordnung und wird im Tycoon-System zur Entwicklung von Anwendungsprogrammen sowie zur Datenmodellierung benutzt. Weiterhin wird diese Sprache als Systemprogrammiersprache eingesetzt; der weitaus größte Teil des Tycoon-Systems ist in TL programmiert. In diesem Abschnitt wird eine kurze Einführung in die für diese Arbeit wesentlichen Elemente der Programmiersprache TL gegeben. Für eine vollständige Beschreibung, insbesondere zu den Gebieten Polymorphie und Subtypbeziehungen, wird auf [Matthes 93] verwiesen.

Auswertungsreihenfolge und Wertkonvertierung: Für Ausdrücke gilt in TL eine strikte Auswertungsreihenfolge: Ausdrücke werden von "links nach rechts" bearbeitet. Diese Reihenfolge gilt auch für alle arithmetischen Ausdrücke; Vorrangregeln wie in anderen Programmiersprachen existieren nicht. Die Auswertung des ersten Ausdrucks des folgenden Beispiels[2] ergibt den Wert 14 und nicht, wie in Sprachen mit arithmetischen Vorrangregeln, den Wert 11. Die Auswertung eines Ausdrucks kann durch Klammerung geändert werden.

$3 + 4 * 2$
$\Rightarrow 14$ *:Int*
$3 + \{4 * 2\}$
$\Rightarrow 11$ *:Int*

Eine automatische Wertekonvertierung und eine Überladung von Bezeichnern, wie sie zum Beispiel in den Programmiersprachen C++ und Ada möglich ist [Ellis, Stroustrup 90; Ichbiah, others 83], wird von TL nicht zur Verfügung gestellt. Beispielsweise existiert keine automatische Typkonvertierung von ganzzahligen Werte in Fließkommawerte.

Werte- und Typbindungen: TL verfügt über die folgenden vordefinierten Datentypen: Bool, Integer, Real, Char und String. Weitere Datentypen können mit Mitteln der Programmiersprache definiert werden. Beispiele für Werte der genannten Basistypen sind:

[2] Durch $\Rightarrow$ werden in den Beispielen die Ausgaben des Tycoon-Systems gekennzeichnet.

```
true false                      (* Die beiden möglichen Werte des Typs Bool *)
4712                            (* Ein ganzzahliger Wert des Typs Int *)
2.71828                         (* Eine Fließkommazahl des Typs Real *)
"Große Worte, kleine Wirkung"   (* Eine Zeichenkette des Typs String *)
```

In der Programmiersprache TL sind statische, veränderliche und dynamische Wertbindungen möglich. Eine statische Bindung wird zum Beispiel durch den Ausdruck

```
let x = 3.14
```

definiert. Nach der Auswertung dieses Ausdrucks ist der Bezeichner x statisch an den Wert 3.14 gebunden.

Eine veränderliche Bindung einer Variable an einen Wert kann durch einen Ausdruck der folgenden Form erreicht werden:

```
let var x = 10.5
```

Der Wert der Variablen x kann dann durch eine Zuweisung geändert werden:

```
x := 3.33
```

Die Sichtbarkeit einer TL-Variablen kann durch die Schlüsselworte **begin** und **end** eingeschränkt werden. Im folgenden Beispiel wird zunächst der Wert 815 an den Bezeichner a gebunden. Innerhalb des gekennzeichneten Blocks wird eine neue, lokale Bindung an den Bezeichner a vorgenommen. Die vorgenommene Bindung verschattet die vorangegangene Bindung. Nach Verlassen des Blocks sind die lokalen Bindungen an a und b nicht mehr sichtbar. Der Wert der Variablen c entspricht nach der Bindung dem Wert der globalen Bindung von a:

```
let a = 815
begin
  let a = 3
  let b = 7
end
let c = a    (* Der Bezeichner c wird an den Wert 815 gebunden *)
```

Jedes Tycoon-Objekt ist von einem bestimmten, unveränderlichen Typ. Der Typ einer Variablen kann bei Bindung an einen Wert durch den Compiler abgeleitet oder als Zusicherung direkt angegeben werden:

```
let s = "Ein String"
let s :String = "Ein String"
```

Im ersten Beispiel wird der Typ der Variablen a durch den angegebenen Wert vom Compiler abgeleitet. Hier ist jeder Wert jeden Typs zulässig. Im zweiten Beispiel wird die Angabe

eines Werts, der nicht vom Typ *String* ist, einen Typfehler hervorrufen, da der gewünschte Typ der Variablen *a* durch die Typangabe *:String* zugesichert wurde.

Die dynamische Bindung einer Variablen an einen Wert wird in TL durch die Parameter von Funktionen dargestellt. Funktionsdefinitionen werden durch das vorangestellte Schlüsselwort **fun** eingeleitet:

```
let inc = fun(n :Int) :Int begin n + 1 end (* Funktionsdefinition *)
inc(1)  (* Funktionsaufruf *)
⇒ 2 :Int
```

Durch den Aufruf einer Funktion werden die aktuellen Parameter dynamisch an die formalen Parameter der Funktion gebunden. Im Rumpf einer Funktion kann auf alle Variablen des globalen Sichtbarkeitsbereichs, lokale Variablen der Funktion sowie die aktuellen Werte der Formalparameter zugegriffen werden. Die Angabe des Typs eines Funktionsparameters ist immer notwendig, während der Typ des Ergebniswerts wiederum vom Compiler abgeleitet werden kann. Eine verkürzte Schreibweise für obige Funktionsdefinition ist:

```
let inc(n :Int) = n + 1
```

Typbindungen werden in TL durch Voranstellung des Schlüsselwortes **Let** angegeben. Im folgenden Beispiel werden die Typen *T* und *Person* definiert sowie Werte an Bezeichner dieser Typen gebunden:

```
Let T = Tuple x :Real end          (* Typbindung *)
let t :T = tuple let x = 3.14 end  (* Wertbindung *)

Let Person = Tuple        (* Typbindung *)
  name :String
  age :Int
end
let p :Person = tuple     (* Wertbindung *)
 let name = "Meier"
 let age = 44
end
```

Durch die im vorangegangenen Beispiel benutzten Schlüsselwörter **Tuple** und **tuple** werden jeweils geordnete Mengen von Bindungen formuliert.

Polymorphismus: In TL werden Subtyppolymorphismus sowie parametrischer Polymorphismus unterstützt. Für die genaue Beschreibung von Typbeziehungen und Typisierungsregeln in TL sei hier auf [Matthes 93] verwiesen. An dieser Stelle sollen lediglich einige Beispiele dieser Arten von Polymorphismus folgen.

Die Angabe einer Signatur der Form *x :T1* stellt eine partielle Typspezifikation dar, die besagt, daß *x* mindestens *T1* erfüllt. Ein an den Bezeichner *x* gebundener Wert kann aber auch eine genauere Spezifikation *T2* erfüllen. Eine partielle Ordnung auf Typen wird in TL durch die Notation *T2 <: T1* beschrieben. Diese besagt, daß der Typ *T2* eine präzisere

Spezifikation des Typs *T1* darstellt. Der Typ *T1* wird auch als Supertyp des Typs *T2*, der Typ *T2* als Subtyp des Typs *T1* bezeichnet. Der Supertyp aller nichtparametrisierten Typen in TL ist der Typ **Ok**.

Die folgenden Typen *Person* und *Student* stehen durch ihre Struktur in der Subtypbeziehung *Student* <: *Person*. Die Funktion *getAge* ist eine polymorphe Funktion, die als Argumente Werte des Typs *Person* sowie aller Subtypen des Typs Person, also beispielsweise Werte des Typs *Student*, akzeptiert.

```
Let Person = Tuple
  age :Int
  name :String
end
Let Student = Tuple
  age :Int
  name :String
  semestercount :Int
end

let getAge(P <:Person  p :P) = p.age
```

Weiterhin wird in TL die Parametrisierung von Typen unterstützt. Das folgende Beispiel zeigt das Prinzip der Definition eines parametrisierten Typs *Queue* sowie einer polymorphen Funktion *newQueue*, die die Erzeugung beliebig parametrisierter Elemente des Typs *Queue* gestattet:

```
Let Queue(R <:Ok) <:Ok = Tuple ... end
let newQueue(R <:Ok) :Queue(R) = begin ... end

let intQueue = newQueue(:Int)
let personQueue = newQueue(:Person)
```

Funktionsparameter und Funktionen höherer Ordnung: Funktionsdefinitionen liefern in TL Werte erster Klasse. Dies bedeutet unter anderem, daß Funktionen wie andere TL Werte gespeichert und als Argumente weiterer Funktionen benutzt werden können:

```
let sort(T <:Ok  a :Array(T)  greaterEqual(x,y :T) :Bool) :Ok =
  begin ... end

let a = array 3 8 40 1 0 7 ... end
let greaterEqual(x,y :Int) = x >= y

sort(a greater)
```

In diesem Beispiel erwartet die Funktion *sort* als Argument ein indizierbares Feld (Array) mit Elementen eines beliebigen, aber einheitlichen Typs sowie eine auf dem Typ der Feldelemente definierte Vergleichsfunktion *greaterEqual*. In der Funktion *sort* kann dadurch ein typunabhängiger Sortieralgorithmus formuliert werden.

Das folgende Beispiel zeigt die Definition der polymorphen Funktion *makeFun*, die eine weitere Funktion *f* als Argument erwartet und als Ergebnis eine anonyme Funktion liefert, die *f* zweimal auf ihr Argument anwendet:

```
let makeFun(T<:Ok  f(x :T):T)(x :T) = f(f(x))

let f1 = makeFun(fun(x :Int) x + 1)
let f2 = makeFun(fun(x :Real) real.mul(x x))
f1(3);
⇒ 5 :Int
f2(2.0);
⇒ 16.0000 :Real
```

Kontrollstrukturen: Die Sprache TL bietet weiterhin für Fallunterscheidung, Iteration, unbedingte und bedingte Wiederholungen sowie zur Ausnahmebehandlung die folgenden Kontrollstrukturen an:

```
if x > 7 then ... else ... end     (* Fallunterscheidung *)
for i = 0 upto 10 do ... end       (* Iteration *)
loop ... end                       (* Unbedingte Wiederholung *)
while x < 1000 do ... end          (* Bedingte Wiederholung *)
```

```
try      (* Ausnahmebehandlung *)
 ...     (* Auszuführender Block *)
when Ausnahme then
 ...     (* Auszuführender Block für eine bestimmte Ausnahme *)
else
 ...     (* Auszuführender Block für andere Ausnahmen *)
end
```

2.3 Module und Schnittstellen

Module dienen im Tycoon-System der Organisation wiederverwendbarer Programmteile. Ein Modul wird durch seine Schnittstelle und einen Implementationsteil beschrieben. In der Schnittstelle zu einem Modul werden die Signaturen der vom Modul exportierten Typen und Werte spezifiziert. Ein Beispiel für die Schnittstelle eines Moduls *mutex* ist die folgende Definition:

```
interface Mutex
import thread
export
  T <: Ok
  new() :T
  lock(m :T) :Ok
```

```
  unlock(m :T) :Ok
  getLocks(t :thread.T(Ok)) :Array(T)
end;
```

Eine Schnittstelle zu einem Modul wird durch das Schlüsselwort **interface** eingeleitet. Bezeichner aus anderen Modulen, die in der Schnittstellendefinition benötigt werden, können über eine Importliste (**import**) importiert werden. Nach dem Schlüsselwort **export** folgen die vom Modul exportierten Bezeichner. Diese stehen dann zur Verwendung in Programmen und anderen Modulen zur Verfügung. Um die durch ein Modul exportierten Typen, Funktionen und Werte außerhalb des Moduls nutzen zu können, ist die Voranstellung des Modulbezeichners notwendig. Zum Beispiel kann die Funktion *lock* aus der obigen Schnittstellendefinition durch *mutex.lock* aufgerufen werden, wenn der Implementationsteil des Moduls den Namen *mutex* hat.

Die Kodierung der in einer Schnittstellendefinition festgelegten Objekte erfolgt in der Modulimplementation:

```
module mutex
import thread
export
Let T = Tuple ... end
let  new() :T = ...
let  lock(m :T) :Ok = ...
let  unlock(m :T) :Ok = ...
let  getLocks(t :thread.T(Ok)) :Array(T) = ...
end;
```

Wiederum werden durch **import** und **export** die Import- und Exportlisten der Modulimplementation festgelegt. Die Implementationsmodule enthalten mindestens die in einer entsprechenden Schnittstellendefinition aufgeführten Komponenten.

Modulschnittstellen werden im Tycoon-System durch Tupel-Typen repräsentiert und bilden wie Funktionen und andere Typen Werte erster Klasse. Modulschnittstellen können daher wie alle anderen Typen des Tycoon-Systems manipuliert werden. Werte eines solchen Tupel-Typen werden durch den Import eines zugehörigen Implementationsmoduls gebildet.

Im Gegensatz zu anderen Programmiersprachen, die über ein Modulsystem verfügen, können im Tycoon-System zu einer Schnittstelle mehrere Implementationsmodule vorhanden sein. Die konkret benutzte Implementation zu einer Modulschnittstelle wird im Tycoon-System über Bibliotheksdefinitionen spezifiziert. In einer solchen Spezifikation werden Modulschnittstellen den Modulimplementationen zugeordnet:

```
library threadenv
import
  thread
with
  interface
    Mutex
  module
```

```
    mutex :Mutex
  interface
    Rendezvous
  module
    rendezvous :Rendezvous
end;
```

In diesem Beispiel werden den Schnittstellen *Mutex* und *Rendezvous* die Implementationsmodule *mutex* und *rendezvous* zugeordnet. Der größte Teil der in einem Tycoon-System vorhandenen Programme ist in Modulen abgelegt. Module verwandter Funktionalität werden durch Bibliotheksdefinitionen gruppiert. Beispielsweise stehen in der Bibliothek *bulkenv* eine Reihe von Modulen zur Verarbeitung von Massendaten zur Verfügung. Die in dieser Arbeit implementierten Funktionen zur Synchronisation werden in der Bibliothek *threadenv* zusammengefaßt.

2.4 Speicherprotokoll und persistenter Objektspeicher

Alle mittels der Programmiersprache TL erzeugten Objekte und Bindungen sind persistent. Diese Eigenschaft von TL-Objekten ist für Benutzer des Systems transparent. Erreicht wird die Persistenz von TL-Objekten durch eine automatische Speicherung der Objekte in einem Objektspeicher, dessen Zustand auf einem nichtflüchtigen Medium stabilisiert werden kann. Hierbei ist die Persistenz eines Objekts durch die Erreichbarkeit dieses Objekts von dem bei der Erzeugung eines Objektspeichers angelegten Wurzelobjekts definiert (siehe unten).

Die Kommunikation des Tycoon-Systems mit einem Objektspeicher wird über das Speicherprotokoll TSP abgewickelt, das ein, im Sinne des Typsystems der Programmiersprache TL, untypisiertes Speicherprotokoll auf einer Abstraktionsebene unterhalb von TL-Objekten darstellt [Matthes et al. 95]. TL-Objekte werden auf der Ebene des TSP als atomare oder zusammengesetzte Werte der Basistypen des TSP verwaltet.

Im einzelnen stellt das TSP dem Tycoon-Laufzeitsystem die folgende Funktionalität zur Verfügung:

Uniformen Zugriff auf verschiedene Objektspeicher: Für unterschiedliche Objektspeicher ist jeweils eine Implementation des TSP für den Objektspeicher, ein TSP-Adapter, zu entwickeln, der die Funktionalität des TSP auf Funktionalität des Objektspeichers abbildet. Jeder TSP-Adapter stellt seinen Klienten einen einheitlichen Satz von TSP-Funktionsaufrufen zur Verfügung. TSP-Adapter existieren für eine Reihe verschiedener Objektspeicher:

Tymem: Ein portabler Einbenutzer-Objektspeicher, der vollständig im Hauptspeicher des Systems gehalten wird. Die Persistenz des Objektspeichers wird durch die Standarddateidienste des zugrundeliegenden Betriebssystems erreicht.

Tysin: Ein portabler Einbenutzer-Objektspeicher. Dieser Objektspeicher hält nur einen geringen Teil der genutzten Daten im Hauptspeicher des Systems.

Tyobject: Ein TSP-Adapter für den kommerziellen Mehrbenutzer-Objektspeicher ObjectStore [Lamb et al. 92].

Programmierschnittstelle: Die Programmierschnittstelle des TSP beinhaltet eine Reihe von Funktionsaufrufen:

- ▷ Erzeugung, Modifikation und Löschen von Objektspeichern.
- ▷ Erzeugung, Modifikation von Objektspeicherwerten.
- ▷ Abfragefunktionen, die Auskunft über Eigenschaften des zugrundeliegenden Objektspeichers geben. Hierzu gehören beispielsweise Eigenschaften wie die Verfügbarkeit von 2-Phase-Commits und Sperrmöglichkeiten für einzelne Speicherobjekte.
- ▷ Stabilisierung des Objektspeichers. Hierbei werden alle durchgeführten Änderungen an Objekten auf einem nichtflüchtigen Medium dauerhaft abgelegt (*Commit*).
- ▷ Wiederherstellung des Zustands der letzten Stabilisierung. Alle seit der letzten Objektspeicherstabilisierung durchgeführten Änderungen werden wieder rückgängig gemacht (*Rollback*).
- ▷ Export und Import von TL-Objekten in einem vom Objektspeicher unabhängigen Format. Hierdurch ist ein Objektaustausch zwischen verschiedenen Objektspeicherimplementationen gewährleistet.

Eine vollständige Beschreibung der TSP-Programmierschnittstelle findet sich in [Matthes et al. 95].

Automatische Freispeicherverwaltung: Objekte der Sprache TL können durch Benutzerintervention nicht explizit zerstört werden. Statt dessen wird bei Bedarf, also zum Beispiel bei Speichermangel oder durch Steuerung seitens des Benutzers, durch den Objektspeicher eine Speicherbereinigung (Garbage Collection) durchgeführt. Hierbei werden alle nicht länger erreichbaren Objekte wieder dem freien Objektspeicherbereich zugeordnet. Die Erreichbarkeit eines Objekts ist hierbei wie folgt definiert:

1. Das Wurzelobjekt des Objektspeichers ist erreichbar. Dieses Objekt wird bei der Erzeugung eines Objektspeichers angelegt.
2. Alle durch eine vom Benutzer definierte Aufzählungsfunktion angegebenen Objekte sind erreichbar.[3]
3. Alle transitiv von dem Wurzelobjekt erreichbaren Objekte sind erreichbar.

[3] Hierunter fallen unter anderem Datenstrukturen des Laufzeitsystems, die durch TSP-Aufrufe erzeugt worden, aber nicht vom Wurzelobjekt erreichbar sind.

Transaktionsverwaltung: Der Zustand eines Objektspeichers kann durch Aufruf von TSP-Funktionen auf einem nichtflüchtigen Medium stabilisiert werden. Alle seit der letzten Stabilisierung geänderten oder erzeugten TL-Objekte werden dabei dauerhaft abgelegt. Eine Stabilisierung des Objektspeichers entspricht dabei einer abgeschlossenen Transaktion. Mittels Funktionalität des TSP kann eine solche Transaktion nicht rückgängig gemacht werden. Während der weiteren Arbeit mit dem System ist jedoch der Zustand des Objektspeichers, der bei der letzten Objektspeicherstabilisierung vorlag, jederzeit wieder erreichbar.

Erweiterbarkeit: Das TSP wird durch verschiedene Erweiterungen ergänzt. Derartige Erweiterungen bedienen sich der Funktionalität einer vorliegenden TSP-Implementation und stellen wiederum eine TSP-konforme Programmierschnittstelle zur Verfügung. Die Funktionalität des TSP kann daher durch "Stapelung" von TSP-konformen, zusätzlichen Protokollschichten erweitert werden. Beispielsweise existieren Erweiterungen des TSP, die eine Definition von mehreren Sicherungspunkten ermöglichen, zu denen der Objektspeicherzustand im Verlauf einer Sitzung zurückgesetzt werden kann [Kornacker 95]. Zusätzliche Erweiterungen wie Dienste zur Zugriffskontrolle [Rudloff et al. 95] stehen ebenfalls zur Verfügung.

2.5 Langlebige Aktivitäten

Langlebige Aktivitäten werden im Tycoon-System durch persistente Threads realisiert. Tycoon-Threads sind Objekte "erster Klasse", sie können gespeichert werden, auf Identität verglichen werden und als Funktionsargumente und Rückgabewerte fungieren.

Tycoon-Threads werden durch Aufruf der Funktionen *new* und *fork* des Standardmoduls *thread* erzeugt.[4] Ein Tycoon-Thread führt, nebenläufig zu weiteren Threads, eine Funktion aus, die bei der Erzeugung des Threads angegeben wird. Das folgende Beispiel zeigt die Definition einer Funktion *func* und die Erzeugung eines Threads *t*, der die Funktion *func* ausführt:

```
let func(self :thread.T(Int)) :Int =
  begin
    ...
    thread.kill(self)
    ...
  end

let t = thread.fork(func)
```

Eine Funktion, die als Thread ausgeführt werden soll, weist stets einen Parameter *self* auf, dessen Typ entsprechend dem Rückgabewert der Funktion parametrisiert werden muß. Der Parameter *self* wird einer Funktion bei der Erzeugung eines Threads für diese Funktion übergeben. Dadurch ist es möglich, daß ein Thread Operationen auf sich selbst durchführt. Beispielsweise beendet sich der Thread *t* im obigen Programmfragment durch Aufruf der Funktion *kill*.

[4] Die vollständige Schnittstellendefinition des Moduls *thread* befindet sich in Anhang A.4.

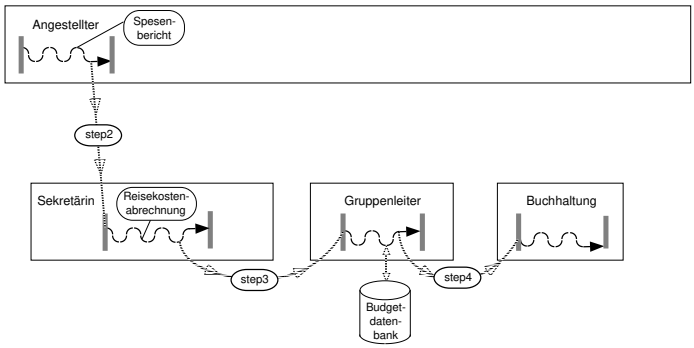

Abbildung 2.2: Threadmigration

Der Rückgabewert eines Threads ist der Rückgabewert der durch den Thread ausgeführten Funktion. Der Typ R des Rückgabewerts ergibt sich aus dem für den Parameter *self* angegebenen, parametrisierten Typ *thread.T(R)*. Durch den folgenden Aufruf kann der Rückgabewert des obigen Threads t abgerufen werden:

let *result :Int = thread.join(t)*

Durch diesen Aufruf wird der aufrufende Thread bis zur Beendigung des Threads t blockiert. Nach Beendigung des Threads t werden dessen Rückgabewert an den Bezeichner *result* gebunden und die Ausführung des aufrufenden Threads fortgesetzt.

Tycoon-Threads sind wie alle Objekte des Tycoon-Systems persistent. So ist auch der Ausführungszustand eines Threads persistent. Beispielsweise werden aktive Threads, also Threads, die sich in Ausführung befinden, nach einem Neustart des Systems automatisch wieder aktiviert.

Eine besondere Eigenschaft des Tycoon-Systems ist die Migration von Tycoon-Objekten in entfernte Tycoon-Systeme beziehungsweise Objektspeicher [Mathiske 96]. Diese in Abschnitt 5 genauer beschriebene Eigenschaft gilt auch für Tycoon-Threads. Diese können einen Objektspeicher verlassen und ihre Tätigkeit in einem anderen Objektspeicher fortsetzen. Die Abbildung 2.2 zeigt die Migration eines Threads aus dem System *Angestellter* über die Systeme *Sekretärin* und *Gruppenleiter* in das System *Buchhaltung*. In jedem der Systeme werden durch den Thread gewisse Aufgaben erfüllt, bevor dieser Thread in ein weiteres System migriert.

Kapitel 3

Synchronisation nebenläufiger Prozesse

Zunächst wird im folgenden Abschnitt eine Übersicht über die verschiedenen Ziele der nebenläufigen Ausführung von Programmen gegeben. Die dieser Arbeit zugrundeliegende Art der Nebenläufigkeit, die der kooperativen Nebenläufigkeit, erfordert die Bereitstellung von Mechanismen, die einen sicheren Zugriff auf Systemressourcen gewährleisten. Dieser Zugriff auf Systemressourcen, unter denen hier sowohl physikalische Geräte wie auch Programmobjekte zu verstehen sind, muß in geeigneter Weise synchronisiert werden.

Da Beschreibung und Analyse nebenläufiger Vorgänge seit mehreren Jahrzehnten Bestandteil verschiedener Teildisziplinen der Informatik sind, steht eine große Auswahl von Methoden zur Synchronisation zur Verfügung. Eine Beschreibung und Analyse der wesentlichen Methoden befindet sich in Abschnitt 3.2. Ein Überblick über die in verschiedenen persistenten Programmiersystemen eingesetzten Synchronisationsmethoden gibt Abschnitt 3.6.

3.1 Nebenläufigkeit in Rechensystemen

Nebenläufigkeit in Rechensystemen kann ein ganzes Spektrum von Modellen umfassen (Abbildung 3.1). An einem Ende dieses Spektrums findet man die transaktionale Sichtweise, in der einzelne Prozesse weitestgehend voneinander abgeschirmt sind. Diese Art der Nebenläufigkeit findet sich in den klassischen Datenbanksystemen, in denen die Auswirkungen von Nebenläufigkeit beim Zugriff auf Datenbestände vom System verborgen werden. Die dem Programmierer zur Absicherung eines nebenläufigen Zugriffs zur Verfügung stehenden Sprachmittel beschränken sich auf das explizite Sperren von Datenbeständen zum Zweck der Bearbeitung sowie auf die Formulierung von Transaktionen, bei denen die notwendige Serialisierung von konkurrierenden Datenzugriffen vollständig in der Verantwortung des Datenbanksystems liegt. Weiterhin besteht die Möglichkeit der Rücknahme von Transaktionen, um im Fehlerfall einen zurückliegenden, konsistenten Systemzustand erreichen zu können. Ein solches transaktionales System wird durch die ACID Eigenschaften der Atomarität, Konsistenz, Isolation und Dauerhaftigkeit spezifiziert [Gray, Reuter 93]. Zusammenfassend muß ein transaktionales System über die folgenden Mechanismen verfügen:

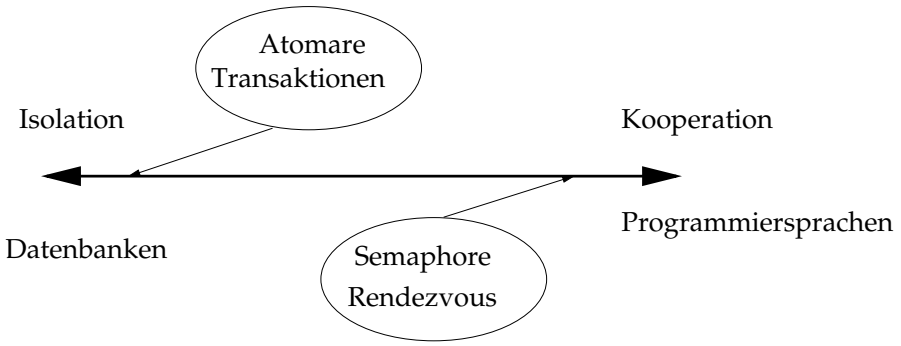

Abbildung 3.1: Isolation vs. Kooperation

- Mechanismen, die die Isolation von Prozessen voneinander sicherstellen. Dazu gehören das Sperren von Datenbeständen, die automatische Serialisierung von Datenzugriffen sowie Methoden des optimistischen konkurrierenden Zugriffs auf Datenbestände.
- Mechanismen, die die Atomarität der Transaktionen sicherstellen. Hierunter ist die automatische Rücknahme der durch fehlerhafte oder abgebrochene Prozesse verursachten Änderungen zu verstehen. Die Auswirkungen einer Transaktion können von anderen Prozessen nur bei erfolgreichem Anschluß dieser Transaktion festgestellt werden.
- Mechanismen, die die Persistenz der durchgeführten Änderungen sicherstellen. Fehlerhafte Systemzustände führen lediglich zu einer Rücknahme von Modifikationen noch nicht abgeschlossener Transaktionen.

Die auf der rechten Seite von Abbildung 3.1 zu findenden Systeme stellen die Synchronisation von Zugriffen auf gemeinsam benutzte Datenbestände in die alleinige Verantwortung des Programmierers. Zu kooperativen Systemen gehören insbesondere Programmiersprachen, die die Formulierung nebenläufiger Aktivitäten innerhalb eines Programms gestatten. Um die innerhalb eines Programms definierten, nebenläufigen Aktivitäten von den Betriebssystemprozessen abzugrenzen, wird im allgemeinen der Begriff *Thread* verwendet. Threads besitzen, da sie voneinander verschiedene, nebenläufige Ausführungpfade desselben Programms darstellen, einen vollständigen Zugriff auf alle vorhandenen Programmobjekte. Im Unterschied zu transaktionalen Systemen steht hier die Kooperation dieser Aktivitäten im Vordergrund. Verschiedene Threads arbeiten gemeinsam an einem vorliegenden Problem und kommunizieren miteinander. Zur Abstimmung der bei dieser Kooperation und Kommunikation entstehenden Konflikte beim Zugriff auf gemeinsam zu nutzende Programmobjekte stehen in allen solchen Systemen verschiedene Synchronisationsmechanismen zur Verfügung.

Bezüglich der transaktionalen Eigenschaften des Tycoon-Systems sind folgende Besonderheiten zu beachten [Kornacker 95]:

- Das Tycoon-System ist ein Einbenutzersystem. Es existiert keine Multiprozeßeigenschaft des Systems. In einem gegebenen Tycoon-System wird ein Programm immer

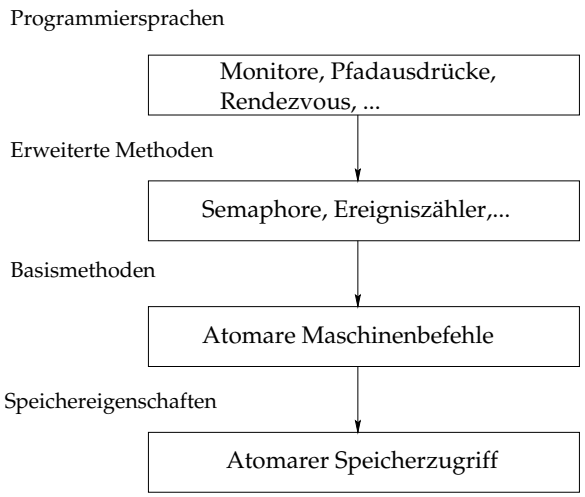

Abbildung 3.2: Hierarchie der Synchronisationsmethoden

auch nur durch einen Betriebssystemprozeß ausgeführt.

- Programme können durch mehrere Threads ausgeführt werden. Diese besitzen einen vollständigen Zugriff auf alle Programmobjekte. Eine durch einen Thread durchgeführte Transaktion beeinflußt alle weiteren vorhandenen Threads.

- Transaktionen können durch persistente Sicherungspunkte weiter strukturiert werden. Im Verlauf einer Transaktion können beliebig viele Sicherungspunkte gesetzt werden, zu denen das System später zurückkehren kann. Das Ende einer Transaktion bewirkt auch die Aufhebung dieser Sicherungspunkte.

Eine Differenzierung von Prozessen und Threads wird in den folgenden Abschnitten fallengelassen, da für die Betrachtung von Synchronisationsproblemen lediglich der Sachverhalt des gemeinsamen Zugriffs auf einzelne Ressourcen relevant ist. Welche Granularität die daran beteiligten, aktiven Objekte haben, steht dabei im Hintergrund.

3.2 Synchronisationsmechanismen

Ziel dieses Abschnitts ist die Vorstellung bekannter Verfahren zur Synchronisation von Prozessen. Aus den vorgestellten Verfahren wird im folgenden Kapitel eine Auswahl von für persistente Objektsysteme geeigneten Verfahren getroffen.

Bei näherer Betrachtung der verfügbaren Literatur zum Thema Synchronisation zeigt es sich, daß die Verfahren zweckmäßigerweise in aufeinander aufbauende Gruppen unterteilt werden können. Einen ersten Überblick über den hierarchischen Aufbau von Synchronisationsmethoden für Systeme mit gemeinsamen Speicher gibt Abbildung 3.2.

3.3 Basismethoden

Die in diesem Abschnitt vorgestellten Verfahren basieren auf speziellen Eigenschaften der Rechnersysteme. Im einzelnen sind dieses Verfahren, die

- ▷ auf bestimmten Eigenschaften des verwendeten Speichersystems aufbauen,
- ▷ eine bestimmte Konfiguration des verwendeten Rechnersystems voraussetzen und
- ▷ spezielle Fähigkeiten des verwendeten Prozessors oder zusätzlichen Schaltungsaufwand erfordern.

3.3.1 Ununterbrechbarer Speicherzugriff

Eine grundlegende Voraussetzung für die Synchronisation von Prozessen ist die Verfügbarkeit eines ununterbrechbaren Zugriffs auf Speicherzellen des Rechnersystems während Schreib- und Leseoperationen. Dieses Verhalten wird durch den Aufbau des Speichersystems sichergestellt [Lagemann 87]. Die Ununterbrechbarkeit eines Speicherzugriffs ermöglicht bereits die Bereitstellung von Algorithmen, die den gegenseitigen Ausschluß von konkurrierenden Prozessen beim Zugriff auf gemeinsam zu nutzende Daten gewährleisten. Bis zu den Arbeiten von Dekker und Dijkstra nahm man an, daß dieses Problem nicht durch einfachen Speicherausschluß zu lösen ist [Dijkstra 68]. Der Algorithmus von Dekker ist hier wegen seiner Bedeutung noch einmal in einer vereinfachten Form nach Peterson [Jessen, Valk 87, Seite 134–135] in der Programmiersprache TL angegeben.

```
let var c1 = 0 and var c2 = 0 and var turn = 1

let funcA(self :thread.T(Ok)) =
  begin
    (* Eingangsprotokoll *)
    c1 := 1
    turn := 2
    while {c2 == 1} and {turn == 2} do end
    (* kritischer Abschnitt *)
    c1 := 0
  end

let funcB((self :thread.T(Ok)) =
  begin
    (* Eingangsprotokoll *)
    c2 := 1
    turn := 1
    while {c1 == 1} and {turn == 1} do end
    (* kritischer Abschnitt *)
    c2 := 0
  end
```

```
(* Erzeugung der Threads *)
let tA = thread.fork(funcA) and tB = thread.fork(funcB)
```

Jeder Thread, der versucht in den kritischen Abschnitt zu gelangen muß zunächst das Eingangsprotokoll durchlaufen: Die Variable *c1* wird auf 1 gesetzt, wenn der Thread *tA* in den kritischen Abschnitt eintreten möchte; ebenso wird *c2* auf 1 gesetzt, wenn Thread *tB* in den kritischen Abschnitt eintreten möchte. Die Variable *turn* dient der Verhinderung von Verklemmungen, in ihr wird der nächste Thread vermerkt, der in den kritischen Abschnitt eintreten darf. Versuchen beide Threads gleichzeitig das Eingangsprotokoll auszuführen, so ermöglicht derjenige Thread, der die Variable *turn* als letztes ändert, dem jeweils anderen Thread den Eintritt in den kritischen Abschnitt.

Eine Analyse der Methode von Dekker [Jessen, Valk 87] zeigt jedoch, daß der von ihm entwickelte Algorithmus verschiedene, gravierende Nachteile aufweist:

- Alle Prozesse, die an dem zu lösenden Ausschlußproblem beteiligt sind, müssen von vornherein bekannt sein.
- Der Algorithmus benutzt die Technik des aktiven Wartens, das heißt, ein Prozeß verbraucht auch im Wartezustand Rechenzeit.
- Der Algorithmus ist relativ kompliziert und die Rechenzeit des Algorithmus steigt bei der Beteiligung von mehr als zwei Prozessen stark an.

Trotz dieser Nachteile erbringt der Algorithmus von Dekker den Nachweis, daß der gegenseitige Speicherausschluß, der durch den Aufbau des Speichersystems gegeben ist, die Lösung von Synchronisationsproblemen zuläßt.

3.3.2 Unterbrechungssperren

Eine einfache Möglichkeit zur Synchronisation stellt das Verbot von Unterbrechungen laufender Prozesse dar. Unterbrechungen werden in Rechnersystemen auf vielfältige Art benutzt. Beispiele hierfür sind Ein- und Ausgabeanforderungen durch externe Geräte oder zyklische Unterbrechungen durch externe Zeitgeber. Typischerweise werden in Betriebssystemen, die die nebenläufige Abarbeitung von Programmen erlauben, Prozeßumschaltungen als Folge abgelaufener Rechenzeitkontingente durchgeführt. Kritische Abschnitte können dann dadurch realisiert werden, daß für die Dauer eines solchen Abschnitts jede Unterbrechung untersagt wird. Diese Methode weist jedoch eine Reihe von Problemen auf:

- Sperren für Unterbrechungsanforderungen werden für kritische Abschnitte im Betriebssystem selbst benötigt und sind im allgemeinen auch nur durch privilegierte Maschineninstruktionen realisierbar, die auf der Ebene von Benutzerprozessen nicht zugelassen sind.
- Die gesamte Funktionsweise des Systems kann durch lang andauernde oder nicht wieder aufgehobene Sperren von Unterbrechungsanforderungen negativ beeinflußt werden.
- Unterbrechungssperren sind für Mehrprozessorsysteme nicht ohne weiteres realisierbar.

3.3.3 Atomare Prozessorinstruktionen

Wie in Abschnitt 3.3.1 besprochen, bietet der ununterbrechbare Speicherzugriff bereits die Möglichkeit, Synchronisationsprobleme zu behandeln. Um die programmiertechnische Behandlung von Synchronisationsproblemen zu vereinfachen, bieten viele Rechnerprozessoren spezielle Instruktionen zur Synchronisation an. Diese zeichnen sich im wesentlichen durch drei Eigenschaften gegenüber anderen Prozessorbefehlen aus:

- Es handelt sich um sogenannte Read-Modify-Write-Instruktionen [Hilf, Nausch 84]. Dies bedeutet, daß die Instruktionen den adressierten Speicherbereich lesen, den gelesenen Wert modifizieren und den modifizierten Wert zurück in den adressierten Speicherbereich übertragen.
- Die Abarbeitung einer solchen Instruktion, also die Modifikation der angesprochenen Speicherzelle, ist nicht unterbrechbar. Unterbrechungsanforderungen werden bis zum Ende der Abarbeitung verzögert.
- Auch Zugriffe durch diese Instruktionen auf den gemeinsamen Speicher in Multiprozessorumgebungen sich nicht unterbrechbar.

Beispiele für solche Befehle stellen die Befehle *TEST-AND-SET*, *EXCHANGE* und *COMPARE-AND-SWAP* dar. Die Wirkung dieser Befehle wird im folgenden besprochen.

TEST-AND-SET: Dieser Befehl entspricht in seiner Wirkung dem Auslesen einer Speicherzelle und dem Setzen eines prozessorinternen Zustandsbits in Abhängigkeit des Werts der Speicherzelle. Anschließend wird der Wert der angesprochenen Speicherzelle auf einen definierten Wert gesetzt. Dieser Wert kann bei einigen Prozessoren als Befehlsoperand übergeben werden. TEST-AND-SET ist als Befehl *TAS* zum Beispiel auf Motorola 680xx/880xx-Prozessoren [Hilf, Nausch 84] implementiert. Das folgende Programmfragment verdeutlicht die Wirkung des Befehls *TAS*:

```
let testAndSet(var value :Bool) :Bool =
  begin
    let local = value
    value := true
    local
  end
```

Eine Anwendung dieses Befehls zur Realisierung des gegenseitigen Ausschlusses kann dann wie folgt realisiert werden:

```
let var busy = false

let funcA() =
  begin
    while testAndSet(busy) do end
    (* kritischer Abschnitt A *)
    busy := false
  end

let funcB() =
  begin
    while testAndSet(busy) do end
    (* kritischer Abschnitt B *)
    busy := false
  end
```

Es ist anzumerken, daß der Einsatz von TEST-AND-SET Operationen zu einem unfairen Verhalten bezüglich der Möglichkeit des Eintritts beteiligter Prozesse in kritische Abschnitte führen kann. Faires Verhalten liegt dann vor, wenn garaniert ist, daß ein Ereignis, das stattfinden kann auch stattfindet. Hierbei wird jedoch keine Aussage darüber getroffen, wann dies der Fall ist (vgl. [Jessen, Valk 87]). Im vorigen Beispiel besteht dagegen die Möglichkeit, daß ein Prozeß nie in den kritischen Abschnitt eintreten kann, wenn andere Prozesse ständig früher die Bedingung für den Eintritt vorfinden ($busy == false$).

EXCHANGE: Dieser Befehl stellt eine Verallgemeinerung der TEST-AND-SET Instruktion dar. Die Inhalte zweier Speicherzellen werden atomar ausgetauscht:

```
let exchange(var x :Int  var y :Int) =
  begin
    let var temp = x
    x := y
    y := temp
  end
```

Kritische Abschnitte können dann wie folgt implementiert werden:

```
let var busy = 1

let funcA() =
  begin
    let var localA = 0
    loop
      exchange(busy localA)
      if localA == 1 then exit
    end
    (* kritischer Abschnitt *)
    exchange(busy localA)
  end

let funcB() =
  begin
    let var localB = 0
    loop
      exchange(busy localB)
      if localB == 1 then exit
    end
    (* kritischer Abschnitt *)
    exchange(busy localB)
  end
```

EXCHANGE ist für Intel-x86-Prozessoren als Befehl *XCHG* implementiert [Crawford, Gelsinger 87]. Für Sparc-Prozessoren ist die EXCHANGE-Instruktion als Befehl *SWAP* neben

einer ganzen Reihe anderer atomarer Instruktionen wie *LDSTUB, LDSTUBA* und *SWAPA* implementiert [Catanzaro 91].[1]

COMPARE-AND-SWAP: Eine weitere, unteilbare Maschineninstruktion stellt COMPARE-AND-SWAP dar. Im Unterschied zur einfachen EXCHANGE-Anweisung ist bei dieser Anweisung ein Austausch der adressierten Speicherzellen abhängig von deren Inhalt. Folgendes Beispiel verdeutlicht die Wirkung von COMPARE-AND-SWAP [Gray, Reuter 93]:

```
let compareAndSwap(var cell, old, new :word.T) :Bool =
  begin
    if cell == old then
      cell := new
      true
    else
      old := cell
      false
    end
  end
```

Die COMPARE-AND-SWAP-Anweisung bietet im Gegensatz zu den bisher besprochenen Instruktionen die Möglichkeit, kritische Operationen auf Maschinenworten zu programmieren, ohne auf zusätzliche Hilfsvariablen für die Synchronisation des kritischen Abschnitts zurückgreifen zu müssen.

Weitere maschinennahe Instruktionen: Neben den bisher besprochenen, atomaren Instruktionen bieten verschiedene Prozessoren eine noch weitergehende Unterstützung zur Lösung von Synchronisationsproblemen an. Beispiele hierfür sind atomare Additionsbefehle und Befehle zur direkten Manipulation von Warteschlangen auf VAX/11 Maschinen (ADAWI und INSQHI/REMQHI) [DEC 80].

Einen allgemeineren Mechanismus zur Nachbildung verschiedener, atomarer Instruktionen beschreibt Moore für den Mikroprozessor PPC601 [Moore 93]. Durch die Maschineninstruktion LOAD-AND-RESERVE wird eine Speicherzelle gelesen und als reserviert gekennzeichnet. Eine Schreiboperation auf diese Speicherzelle wird in internen Prozessorregistern vermerkt. Die atomare Operation STORE-CONDITIONAL wertet den Inhalt dieser Register aus und führt eine Schreiboperation nur dann durch, wenn der angesprochene Wert zwischenzeitlich nicht geändert wurde. Bei erfolgreicher Schreiboperation wird die Reservierung der Speicherstelle zurückgenommen. Diese Instruktion kann mit zusätzlichen Verzweigungsoperationen kombiniert werden, um Sequenzen von Operationen zu programmieren, die die oben beschriebenen, atomaren Anweisungen nachbilden.

[1] Bei den genannten Befehlen wird der Inhalt eines Prozessorregisters mit dem Inhalt einer Speicherzelle vertauscht.

3.3.4 Zusammenfassung

Zur Lösung von Synchronisationsproblemen bietet sich eine ganze Reihe von Möglichkeiten auf niedrigem Abstraktionsniveau an. Dekkers Algorithmus erbringt den Nachweis, daß bereits die Eigenschaft des nichtunterbrechbaren Schreib- und Lesezugriffs auf den Hauptspeicher des Rechnersystems zur Synchronisation von nebenläufigen Prozessen ausreichend ist. Diese Lösung eignet sich sowohl für Einprozessorrechner wie auch für Multiprozessormaschinen [Dijkstra 71]. Eine spezielle Lösung für Einprozessorsysteme stellt das Verbot von Unterbrechungen dar, das jedoch abhängig vom zur Verfügung stehenden Betriebsystem nicht immer realisierbar ist. Wie aus dem folgenden Abschnitt ersichtlich wird, eignen sich die angesprochenen Maschineninstruktionen für die Implementation von Synchronisationsmethoden auf höherer Ebene. Ihr direkter Einsatz für Synchronisationsprobleme ist jedoch mit einer Reihe von Nachteilen verbunden. Zum einen kann durch das aktive Warten Rechenzeit gebunden werden, zum anderen besteht die Möglichkeit, daß ein Prozeß nie in einen kritischen Abschnitt eintreten kann: Andere Prozesse können ständig vor diesem Prozeß die erfüllte Bedingung für einen Eintritt in den kritischen Abschnitt vorfinden. Die einzelnen Instruktionen sind darüber hinaus abhängig von der verwendeten Prozessortechnik.

3.4 Erweiterte Methoden

Die Nachteile der bereits angesprochenen, maschinennahen Synchronisationsanweisungen führen zur Entwicklung weiterer Synchronisationsmechanismen. Die im folgenden besprochenen Mechanismen sind weiterhin auf einem eher niedersprachlichen Niveau angesiedelt, abstrahieren aber schon von den direkten, maschinenabhängigen, atomaren Instruktionen. Nach der nahezu direkten Umsetzung atomarer Maschinenbefehle in erweiterte Konstrukte werden Verfahren entwickelt, die den Nachteil des aktiven Wartens auf Ereignisse vermeiden. Dies wird in den meisten der nachfolgend beschriebenen Verfahren durch eine Kombination von maschinennahen Befehlen sowie der Benutzung von Warteoperationen des eigentlichen Betriebssystems erreicht. Maschinennahe Befehle werden bei diesen Verfahren nur noch zum Schutz von kritischen Abschnitten im Betriebssystem selbst verwendet, ihre Anwendung bleibt den Benutzerprozessen verborgen.

3.4.1 LOCK und UNLOCK

Eines der ersten Synchronisationsverfahren, das allerdings noch den Nachteil des aktiven Wartens aufweist, stellte Denning 1971 vor [Denning 71]. Bei diesem Verfahren werden den kritischen Abschnitten sogenannte Sperrbits zugeordnet, die durch die atomaren Instruktionen LOCK und UNLOCK gesetzt bzw. zurückgesetzt werden. Möchte ein Prozeß in einen kritischen Abschnitt eintreten, wird dies durch die Operation LOCK angemeldet. Ist der Abschnitt bereits durch einen anderen Prozeß gesperrt, muß der Prozeß aktiv auf die Freigabe des Abschnitts durch die Operation UNLOCK warten. Diese Operationen können unmittelbar durch einen atomaren Maschinenbefehl wie zum Beispiel TEST-AND-SET implementiert werden.

(Ein freies Lockbit ist mit false zu initialisieren *)*

```
let lock(var l :Bool) =
  begin
    while testAndSet(l) do end
  end

let unlock(var l :Bool) =
  begin
    l := false
  end
```

3.4.2 Semaphore

Einen wesentlichen Fortschritt in der Entwicklung von Methoden zur Synchronisation erzielte Dijkstra mit der Einführung des Konzepts der Semaphore im Jahr 1965 [Dijkstra 68]. In diesem Artikel wird ein Verfahren für die kooperative Zusammenarbeit von Prozessen dargestellt. Zwei oder mehrere Prozesse kommunizieren hierbei über den Austausch von Signalen so, daß die Prozesse sich gegenseitig über die Möglichkeit der Fortführung einer Aktivität informieren. Dieser Signalaustausch erfolgt mittels spezieller Variablen, über Semaphore.

Ein Semaphor entspricht dabei einer ganzzahligen Variablen *sem*, auf der zwei atomare Operationen P und V definiert sind. In Dijkstras Artikel von 1968 werden diese Operationen wie folgt definiert [Dijkstra 68, Seite 67–68]:

> "The V-Operation is an operation with one argument, which must be the identifikation of a semaphore. Its function is to increase the value of its argument by 1; this increase is to be regarded as an indivisible operation."
>
> " The P-Operation is an operation with one argument, which must be the identifikation of a semaphore. Its function is to decrease the value of its argument by 1 as soon as the resulting value would be non-negative. The completion of the P-operation—i.e. the decision that this is the appropriate moment to effectuate the decrease and the subsequent decrease itself—is to be regarded as an indivisible operation."

Trifft die Bedingung $sem > 0$ während einer P-Operation nicht zu, so muß der ausführende Prozeß so lange warten, bis diese Bedingung erfüllt ist. Semaphore, die mit dem Wert 1 initialisiert sind, werden binäre Semaphore genannt.

Die Änderungen des Werts von *sem* in beiden Operationen werden atomar durchgeführt. Versuchen zwei Prozesse gleichzeitig, eine P-Operation auf einem binären Semaphor durchzuführen, und ist das Semaphor frei, so wird einer der beiden Prozesse so lange verzögert, bis *sem* durch eine V-Operation wieder einen Wert $sem > 0$ annimmt. Der Wert von *sem* wird sodann in der P-Operation um den Wert 1 vermindert, und der wartende Prozeß kann in den kritischen Abschnitt eintreten. Analog wird während einer V-Operation der Wert des Semaphors *sem* atomar um 1 erhöht. Warten mehrere Prozesse auf die Freigabe des Semaphors, so ist die Reihenfolge der Zuteilung des Semaphors bei Freigabe (der V-Operation) unspezifiziert. Das folgende Programmfragment verdeutlicht die Anwendung der Semaphoroperationen:

```
let var sem = 1

let activity(n :Int) :Ok =
  begin
    while true do begin
      P(sem)
      (* kritischer Abschnitt von Aktivität n *)
      V(sem)
      (* restliche Tätigkeiten von Aktivität n *)
    end
  end

(* Erzeugung von N Threads, die im wechselseitigen Ausschluß in ihren
   kritischen Abschnitt eintreten *)
for i = 1 upto N do
  thread.fork(fun(self :thread.T(Ok)) activity(n))
end
```

Die beschriebenen, binären Semaphore gestatten genau einem Prozeß die Anwesenheit in einem durch ein Semaphor geschützten Abschnitt. Zur Lösung umfangreicherer Probleme eignen sich allgemeine Semaphore. Sie unterscheiden sich von der ursprünglichen Definition in dem Sachverhalt, daß der Initialwert des Semaphors einen beliebigen Wert annehmen kann. Ein Beispiel für die Anwendung des Semaphorkonzepts zur Implementation einer kooperativen Zusammenarbeit mehrerer Prozesse ist das Erzeuger-Verbraucher-Problem [Stallings 92] .

Eine Implementation der allgemeinen Semaphore mittels der atomaren Maschineninstruktion TEST-AND-SET gibt z.B. Stallings an [Stallings 92]. Auch diese Lösung verwendet die Technik des aktiven Wartens. Jedoch werden durch den Einsatz von TEST-AND-SET nur sehr kurze Programmabschnitte, in denen die internen Datenstrukturen eines Semaphors manipuliert werden, geschützt. Durch die Verwendung der TEST-AND-SET-Instruktion ist diese Implementation auch für Mehrprozessorsysteme mit gemeinsamem Speicher geeignet. Eine Implementation mit Unterbrechungssperren für die Dauer der Manipulation des Semaphors findet sich in der gleichen Quelle. Implementationen unter Verwendung anderer atomarer Maschineninstruktionen wie z.B. COMPARE-AND-SWAP geben Gray und Reuter an [Gray, Reuter 93]. Zusammenfassend ergeben sich folgende Eigenschaften des Semaphorkonzepts:

- Allgemeines Konzept zur Lösung von Synchronisationsproblemen und zur Implementation kooperativer Aktivitäten.
- Aktives Warten wird auf den kurzen Abschnitt der Manipulation der eigentlichen Semaphorvariablen beschränkt.
- Semaphore verhalten sich fair, wenn zur Implementation der wartenden Prozesse Warteschlangenoperationen verwendet werden.

Zahlreiche Autoren weisen aber darüber hinaus auf mögliche Probleme bei der Verwendung von Semaphoren hin:

- ▷ Mit Semaphoren realisierte Programme sind oft nicht leicht zu lesen und zu warten, die Anzahl der Semaphoroperationen ist oft groß und die Semaphoroperationen sind unstrukturiert über den Programmtext verteilt [André et al. 85].
- ▷ Es ist oft sehr schwierig, für ein gegebenes Synchronisationsproblem eine Lösung anzugeben, die gleichzeitig korrekt und effizient ist [André et al. 85].

Diese Probleme führten zur Entwicklung von Synchronisationsmethoden die auf einem höherem Abstraktionsniveau als Semaphore angesiedelt sind. Diese werden in Abschnitt 3.5 besprochen.

3.4.2.1 Erweiterte Semaphoroperationen

Zur Lösung von Problemen, deren Bearbeitung mit allgemeinen Semaphoren sehr schwierig ist oder für die eine Lösung bisher gar nicht angegeben werden konnte, werden zahlreiche Erweiterungen vorgeschlagen. Einige Varianten sind:

Semaphor-Arrays: Diese unterscheiden sich von den allgemeinen Semaphoren lediglich dadurch, daß Arrayelemente als Argumente für Semaphoroperationen zugelassen sind.

Gleichzeitige Semaphoroperationen Bei den gleichzeitigen Semaphoroperationen sind gleichzeitige P- und V-Operationen auf mehreren Semaphoren zugelassen:

$$P(sem_0, sem_1, \ldots, sem_n)$$
$$V(sem_0, sem_1, \ldots, sem_n)$$

Ein Prozeß kann bei einer multiplen P-Operation erst dann fortfahren, wenn alle dabei angesprochenen Semaphore sem_i durch entsprechende V-Operationen wieder freigegeben werden. Gleichzeitige Semaphoroperationen stehen z.B. als Betriebssystemaufruf in allen UNIX-Implementationen zur Verfügung [USO 90].

Leser-Schreiber-Semaphoren: Gray und Reuter geben eine Erweiterung der Semaphoroperationen für das in Datenbanken oft auftretende Leser-Schreiber-Problem an [Gray, Reuter 93]. Bei diesem Synchronisationsproblem tritt der Fall ein, daß ein Datum häufig gleichzeitig von vielen Prozessen gelesen wird, aber nur selten in seinem Wert verändert wird. Ein exklusives Sperren des mit dem Zugriff verbundenen, kritischen Abschnitts mittels binärer Semaphore führt demnach zu folgenden Ineffizienzen:

- ▷ Es ist nur einem Prozeß erlaubt, gleichzeitig im kritischen Abschnitt aktiv zu sein. Das gleichzeitige Lesen eines Datums ist somit nicht möglich.
- ▷ Semaphoroperationen (P-Operation) auf belegten Semaphoren führen zur Suspendierung der aufrufenden Prozesse. Dies bedeutet, daß lesende Prozesse grundlos verzögert werden.

Die Erweiterung der allgemeinen Semaphore zu Leser-Schreiber-Semaphoren wird durch die Einführung zusätzlicher Modi bei den Semaphoroperationen erreicht:

- ▷ Semaphore können exklusiv angefordert werden. Dies entspricht der bisherigen P-Operation. Kein anderer Prozeß kann in den kritischen Abschnitt eintreten.
- ▷ Semaphore können als Lese-Semaphore angefordert werden. Weitere Anforderungen als Lese-Semaphor können sofort durchgeführt werden. Prozesse, die einen schreibenden Zugriff auf ein Datum im kritischen Abschnitt benötigen, müssen warten, bis alle Lesevorgänge beendet sind und das Semaphor wieder freigegeben ist.

3.4.3 Zählen von Ereignissen

Einen Synchronisationsmechanismus, der auf dem Zählen von durch Prozesse ausgelösten Ereignissen basiert, stellen die von Reed und Kanodia beschriebenen *Eventcounts* und *Sequencer* dar [Reed, Kanodia 79]. Im Unterschied zu einer Kommunikation zwischen Prozessen über gemeinsam genutzte, z.B. durch Semaphore geschützte Variablen, findet bei diesem Mechanismus der Austausch von Ereignissen direkt über sogenannte Ereigniszähler (*Eventcounts*) statt. Der Fortschritt eines Prozesses bei einer Berechnung wird über diese Ereigniszähler angezeigt. Ein Ereigniszähler ist eine mit Null initialisierte, ganzzahlige Variable, auf der die folgenden drei Operationen definiert sind[2]:

$$
\begin{array}{lcl}
Advance(E) & : & // \; E := E + 1 \; // \\
x := Read(E) & : & x := E \\
Await(E, v) & : & \text{Der aufrufende Prozeß wird blockiert, bis } E \geq v \text{ gilt.}
\end{array}
$$

Die Operation *Advance* erhöht den Wert des Ereigniszählers E atomar um den Wert 1. Zur Beobachtung des derzeitigen Werts von E dienen die Operationen *Read* und *Await*. Während über *Read* lediglich der aktuelle Wert des Ereigniszählers ausgelesen wird, ist es mit der Operation *Await* möglich, auf einen bestimmten Wert der Variablen E zu warten. Zur Steuerung der Reihenfolge der Abarbeitung von Ereignissen dienen Folgezähler (*Sequencer*). Ein Folgezähler ist eine ganzzahlige Variable, auf der die Operation

$$x := Ticket(S) \quad : \quad // \; x := S \; ; \; S := S + 1 \; //$$

definiert ist. Diese Operation liefert den alten Wert der Variablen S und inkrementiert daraufhin diese Variable. Die gesamte Anweisungfolge läuft atomar ab. Durch eine Ticket-Anweisung wird dem aufrufenden Prozeß ein garantiert eindeutiger Wert zugewiesen, der benutzt werden kann, um den geordneten Eintritt von Prozessen in kritische Abschnitte zu gewährleisten. Die Simulation der Semaphoroperationen verdeutlicht dies [Freisleben 86]:

```
Let Sem <:Ok= Tuple
  var init   :Int
  var event  :Int
  var ticket :Int
```

[2] Die Notation // ... // kennzeichnet die atomare Ausführung einer Anweisungsfolge.

```
end

(* Initialisierung eines binären Semaphors *)
let sem :Sem = tuple let var init = 1 and var event = 0 and var ticket = 0 end

let wait(sem :Sem) :Ok =
  begin
    let t = Ticket(sem.ticket)
    Await(sem.event   t - sem.init + 1)
  end

let signal(sem :Sem) :Ok =
  begin
    Advance(sem.event)
  end
```

Die in der Definition des Semaphors enthaltene Variable *init* gibt dessen Initialisierungswert an. Für binäre Semaphore ist hier der Wert 1 zu wählen. Ein die Prozedur *wait* aufrufender Prozeß erhält durch das atomare Ticket (*sem.ticket*) einen eindeutigen Wert zugewiesen. Hierdurch wird die Reihenfolge der Zuteilung des Semaphors an die die Prozedur *wait* aufrufenden Prozesse festgelegt. Der Ausdruck $sem.event - (t - sem.init + 1)$ beschreibt die Anzahl der Prozesse, die weiterhin die Prozedur *wait* aufgerufen können, ohne durch die Operation *Await* blockiert zu werden. Die in der Prozedur *signal* aufgerufene Operation *Advance* erhöht die Anzahl der Freigabeereignisse atomar um 1 und gibt damit das Semaphor frei.

Reed und Kanodia merken an, daß Ereignis- und Folgezähler gegenüber Semaphoroperationen als grundlegendere Operationen anzusehen sind:

> "Eventcounts and sequencers can be viewed as primitives at a lower level than semaphores. We can build semaphores out of eventcounts and sequenceres, and in addition, can build some more powerful operations on these semaphores" [Reed, Kanodia 79, Seite 119].

Dieses belegen Reed und Kanodia durch den Einsatz ihrer Mechanismen zur Implementation von Leser-Schreiber-Sperren sowie zur Implementation von gleichzeitigen Operationen auf mehreren (simulierten) Semaphoren [Reed, Kanodia 79] (vgl. Abschnitt 3.4.2.1).

Die Autoren weisen außerdem darauf hin, daß durch die monoton anwachsenden Werte der Ereignis- und Folgezähler Schwierigkeiten bei der Implementation der Mechanismen auftreten. Dies gilt insbesondere auf Systemen mit kleinen Wortgrößen, in denen die Zähler durch mehrere Maschinenworte realisiert sind, deren Werte nicht mehr durch atomare Schreiboperationen geändert werden können. Als Lösung geben Reed und Kanodia einen Algorithmus zur Implementation ihrer Mechanismen an, der lediglich die atomare Manipulation einzelner Bits von Maschinenworten voraussetzt [Reed, Kanodia 79].

3.4.4 Zusammenfassung

Dijkstras Semaphoroperationen stellen einen allgemeinen Mechanismus zur Synchronisation und Kommunikation nebenläufiger Aktivitäten in Systemen mit gemeinsamem Speicher dar. Es existieren zahlreiche Erweiterungen dieses Konzepts, die jedoch lediglich dazu dienen, die Behandlung spezieller Problemfälle zu vereinfachen. Zahlreiche Autoren weisen auf die oftmals schwer zu verstehenden Lösungen von Synchronisationsproblemen durch Semaphoroperationen hin. Weiterhin kann die Trennung der Anforderungs- und Freigabeoperationen leicht zur Ursache von Programmierfehlern werden. Die weiteren, in diesem Kapitel betrachteten Synchronisationmechanismen ermöglichen ebenfalls die einfachere Lösung mancher Synchronisationsprobleme, können jedoch zusätzliche Probleme bei ihrer Implementation verursachen. Semaphore und die verwandten Synchronisationsverfahren sind daher eher als maschinenunabhängige, elementare Verfahren zu betrachten, die Bausteine für strukturierte Mechanismen zur Synchronisation darstellen.

3.5 Synchronisation in Programmiersprachen

Die direkt in Programmiersprachen eingebundenen Synchronisationsmethoden versuchen den wesentlichen Nachteil der erweiterten Synchronisationsmethoden zu vermeiden: fehlende Strukturierungsmöglichkeiten durch den direkten Einsatz von Semaphoren und äquivalenten, erweiterten Methoden, die zu Fehleranfälligkeit, erschwerter Wartbarkeit und mangelnder Durchsichtigkeit der mit diesen Mechanismen erstellten Programmen führen. Hochsprachliche Synchronisationsmethoden dienen dazu,

- den strukturierten und sicheren nebenläufigen Zugriff von Prozessen auf gemeinsam genutzte Ressourcen zu ermöglichen,
- Synchronisationsvorgänge durch strukturierte Anweisungen zu lokalisieren sowie
- vom Modell der direkten Speichersynchronisation über gemeinsam genutzte Variablen zu abstrahieren und durch Konzepte wie z.B. den Nachrichtenaustausch über Kommunikationskanäle und die explizite Formulierung von Synchronisationsbedingungen zu erweitern.

Die hochsprachlichen Synchronisationsmechanismen gliedern sich demzufolge in:

- Verfahren, die wie die erweiterten Methoden auf der direkten Manipulation von gemeinsamen Variablen beruhen, jedoch strukturierte Konstrukte zur Synchronisation benutzen.
- Verfahren, die Synchronisation gänzlich ohne gemeinsame Variablen erreichen und statt dessen über spezielle Kanäle miteinander kommunizieren.
- Methoden, die die Formulierung von Synchronisationbedingungen vom eigentlichen Programmtext trennen und die Generierung des eigentlichen, nebenläufigen Programms in die Verantwortung von generischen Werkzeugen, wie z.B. spezielle Compiler oder Programmgeneratoren, legen.
- Verfahren, die eine Mischung der genannten Methoden benutzen.

3.5.1 Kritische Regionen

Kritische Regionen und bedingte kritische Regionen wurden 1972 von Hoare [Hoare 72] und Brinch Hansen [Brinch Hansen 72] eingeführt. Diese Konstrukte gehörten zu den ersten Synchronisationsmechanismen, die eine strukturierte Notation für die nebenläufige Benutzung von Ressourcen zur Verfügung stellten. Bei diesen Konstrukten werden die von Prozessen gemeinsam zu nutzenden Ressourcen bei ihrer Deklaration durch Schlüsselwörter gekennzeichnet. Der eigentliche Zugriff auf diese Ressourcen kann dann später nur innerhalb gekennzeichneter Programmabschnitte, den eigentlichen kritischen Regionen, erfolgen. Die Vorteile dieser Notation, die Kennzeichnung nebenläufig genutzter Objekte, liegen in den durch einen Compiler automatisch eingefügten Synchronisationsoperationen und der Überprüfung der korrekten Benutzung gemeinsamer Ressourcen. Eine mögliche Notation einfacher kritischer Regionen in TL-Syntax lautet wie folgt:

```
(* Deklaration *)
let shared person :Person= tuple
  let var name = "Hoare"
  let var age  = 30
  let var spec = ...
end

(* Benutzung *)
region person do
  person.spec  = f(person.age)
end
```

Die Benutzung der gemeinsam genutzten Variablen *person* ist nur innerhalb einer mit *region ... end* gekennzeichneten kritischen Region zulässig. Die Implementation kritischer Regionen kann durch das Einfügen von Semaphoranweisungen durch den Compiler erreicht werden. Aus

region *person* **do** *S* **end**

wird für jede kritische Region mit demselben Bezeichner *person* die Anweisungsfolge

$P(sem_{person})$
S
$V(sem_{person})$

erzeugt. Die einfachen kritischen Regionen stellen die atomare Modifikation bzw. Benutzung von Programmressourcen sicher. Um eine Kommunikation zwischen Prozessen mittels kritischer Regionen zu ermöglichen, wurde eine Erweiterung kritischer Regionen, bedingte kritische Regionen von Hoare und Brinch Hansen vorgeschlagen. Diese Erweiterung besteht aus der Möglichkeit, die Ausführung einer kritischen Region mit einer Bedingung zu verknüpfen. Erst wenn diese Bedingung erfüllt ist, wird die Ausführung eines Prozesses innerhalb der kritischen Region fortgesetzt. Bei Nichterfüllung der Bedingung wird die kritische Region freigegeben, andere Prozesse können dann in diese eintreten. Nach Verlassen der kritischen Region wird die Bedingung erneut geprüft und entsprechend dem Ergebnis der Prüfung fortgefahren:

```
(* Deklaration siehe letztes  Programmbeispiel *)

(* Benutzung mit Bedingung *)
region person when person.age >= 18 do
  person.spec  := ...
end
```

Des weiteren besteht die Möglichkeit, innerhalb einer kritischen Region auf das Eintreten eines Ereignisses zu warten:

```
(* Deklaration siehe oben *)

(* Benutzung mit Warten innerhalb der kritischen. Region *)
region person do
  ...
  await(person.spec == default)
  ...
end
```

Solange die Bedingung innerhalb der kritischen Region nicht erfüllt ist, werden der darin ausführende Prozeß verzögert und die kritische Region freigegeben. Sobald die Bedingung erfüllt ist und sich kein anderer Prozeß innerhalb der kritischen Region befindet, wird die Ausführung des wartenden Prozesses fortgesetzt. Die Implementation bedingter kritischer Regionen ist insofern problematisch, als daß die Bedingungen für den Eintritt in eine kritische Region bzw. das Fortsetzen eines Prozesses nach einer **await**-Anweisung nach jedem Verlassen der kritischen Region durch einen anderen Prozeß erneut geprüft werden müssen. Dieses stellt, verglichen mit der Programmierung desselben Kommunikationsproblems durch Semaphore, einen nicht unerheblichen zusätzlichen Aufwand dar [Hoare 85]. Brinch Hansen bezeichnet diese Form des Wartens auf Ereignisse als kontrolliertes aktives Warten ("controlled form of busy waiting") [Brinch Hansen 72].

3.5.2 Monitore

Die Zusammenfassung komplexer Datenstrukturen mit den auf ihnen operierenden Funktionen sind in verschiedenen Ausprägungen in vielen Programmiersprachen vorhanden. Das Modulkonzept des Tycoon-Systems sowie die Klassenkonzepte von Programmiersprachen wie z.B. Ada, C++, Modula-3, Smalltalk oder der auf der Basis des Tycoon-Systems entwickelten, objektorientierten Sprache TooL sind Beispiele hierfür. Gemeinsam ist diesen Konzepten unter anderem die Kontrolle des Zugriffs auf die eigentlichen Datenobjekte durch Abkapselung der verschiedenen Datenstrukturen voneinander. Der Zugriff auf Datenobjekte ist nur durch die zu den Datenobjekten selbst gehörenden, bei ihrer Deklaration angegebenen Zugriffsfunktionen möglich. Eine Entsprechung dieses Konzepts bezüglich der Synchronisationsanforderungen liegt bei den sogenannten Monitoren vor [Hoare 74]. In der Datenstruktur eines Monitors sind die spezifizierten, zum Monitor gehörenden Komponenten zusätzlich vor konkurrierendem Zugriff durch nebenläufige Prozesse geschützt. Es ist zu einem Zeitpunkt lediglich einer Zugriffsfunktion gestattet, im Monitor aktiv zu sein. Andere

Prozesse, die ebenfalls Zugriffsfunktionen des Monitors aufrufen wollen, werden verzögert, bis der derzeitige Prozeß den Monitor verlassen hat. Zusätzlich zu diesen Einschränkungen besteht, analog zu den kritischen Regionen, die Möglichkeit, während der Ausführung einer Zugriffsfunktion auf die Erfüllung angegebener Ereignisse zu warten. Hierdurch wird bis zum Eintreten dieser Ereignisse die Ausführungskontrolle an andere Prozesse abgegeben. Ein Beispiel für die Deklaration und Benutzung eines Monitors ist die Simulation eines binären Semaphors [Hoare 85]. Die hier verwendete Notation stellt bereits eine geringfügige Erweiterung des ursprünglichen Monitorkonzepts dar (siehe unten):

```
(* Deklaration *)
monitor Singleresource;
var free :Boolean;
procedure *acquire;
  when free do free := false;      (* wait(free) *)
procedure *release;
  begin  free := true; end;        (* signal(free) *)
begin
  free := true;
  ...
end

(* Benutzung *)
instance semaphore :Singleresource;
semaphore.acquire;
...
semaphore.release;
```

Der Aufruf der Monitorprozedur *acquire* im obigen Beispiel bewirkt die Verzögerung des aufrufenden Prozesses, falls die Auswertung der Variablen *free* nicht den Wert *true* ergibt. Eine anderer wartender Prozeß kann nun in den Monitor eintreten. Die Rolle der Variable *free* entspricht der einer von Hoare für das Warten innerhalb von Monitoren vorgeschlagenen Bedingungsvariablen (Condition variable). Auf einer Bedingungsvariablen *cond* sind die folgenden drei Operationen definiert:

wait(cond)	:	Der aufrufende Prozeß gibt die kritische Region frei und wartet auf das Eintreffen der Bedingung *cond*.
signal(cond)	:	Der aufrufende Prozeß signalisiert das Eintreffen der Bedingung *cond*. Ein auf diese Bedingung wartender Prozeß kann nach Verlassen der kritischen Region durch den *signal(cond)* aufrufenden Prozeß weitergeführt werden.
queue(cond)	:	Sind auf die Bedingung *cond* wartende Prozesse vorhanden, so ergibt diese Funktion den boolschen Wert *true*.

Bedingungsvariablen ermöglichen also die Anwesenheit mehrerer Prozesse in einem Monitor. Bis auf einen ausführenden Prozeß sind allerdings alle anderen Prozesse durch das Warten auf den Eintritt einer Bedingung blockiert.

Eine auf Semaphoren basierende Implementation von Monitoren gibt Hoare an [Hoare 74]. Eine vereinfachte Implementation des Monitorkonzepts findet sich in der objektorientierten Programmiersprache Modula-3 [Nelson 91]. Da Modula-3 bereits über alle Mechanismen zur Definition von Modulen verfügt, wurden zur Realisierung von Monitoren lediglich Bedingungsvariablen sowie eine Form der binären Semaphore, sogenannte Mutexe, hinzugefügt. Zur eigentlichen Sprache gehört hier lediglich das geordnete Akquirieren und Freigeben der Mutexe durch das Sprachelement *LOCK*. Die Verwendung dieses Schlüsselwortes stellt die ausschließliche Benutzung eines Mutex durch den aktuell im Programmabschnitt ablaufenden Prozeß sicher. Eine anderweitige Manipulation, wie beispielsweise die explizite Freigabe eines Mutex durch einen anderen Prozeß, ist nicht ohne weiteres möglich. Als Beispiel für die Benutzung von Mutexen und Bedingungsvariablen ist folgendes Programmfragment in der Programmiersprache Modula-3 angegeben:

```
(* Deklaration *)
VAR m := NEW(MUTEX);
VAR c := NEW(Thread.Condition);

(* Prozeß A: evtl. Warten auf Eintritt der Bedingung B1 *)
BEGIN
LOCK(m) DO
  WHILE NOT(B1) DO
    Thread.Wait(m, c);
    S1;        (* Bearbeitung geschützter Ressourcen *)
  END;
END;

(* Prozeß B: Erfüllung der Bedingung c und ihrer Signalisierung *)
BEGIN
LOCK(m) DO
   S2;                 (* Erfüllung der Bedingung B1 *)
   Thread.Signal(c);   (* Signalisieren der Bedingung B1 *)
  END;
END;
```

Die Benutzung von Bedingungsvariablen ist nur im Zusammenhang mit Mutexen zum Schutz von kritischen Abschnitten sinnvoll. Die Bedingungsvariablen ermöglichen, analog zum Monitorkonzept, die Aufhebung des durch einen Mutex erzwungenen, gegenseitigen Ausschlusses von Prozessen. Durch Aufruf der Funktion *Thread.Wait(m,c)* im obigen Beispiel werden der Mutex m freigegeben und der aufrufende Prozeß suspendiert (Prozeß A). Nach Erfüllung der Bedingung B_1 (Prozeß B) wird dieser Sachverhalt durch Aufruf der Funktion *Thread.signal(c)* signalisiert. Dieser Aufruf führt zur erneuten Aktivierung von Prozeß A. Nach Freigabe des Mutex m durch Prozeß B wird dieser Mutex von Prozeß A erneut akquiriert. Wurde keinem anderen Prozeß dieser Mutex zugeteilt, kann Prozeß A in seinem kritischen Abschnitt fortfahren. Bei der Modula-3–Implementation der Bedingungsvariablen ist zu beachten, daß im Gegensatz zu denen bei Hoare beschriebenen Monitoren die Erfüllung einer Bedingung beim Signalisieren derselben durch *Thread.Signal* nicht garantiert ist. Andere Prozesse können bereits vor dem auf die Bedingung wartenden Prozeß

im kritischen Abschnitt (nach Akquirierung des den Abschnitt schützenden Mutex) aktiv geworden sein [Birell et al. 87]. Aus diesem Grund ist der erneute Test der Bedingung (B_1, s.o.) nach Verlassen der Funktion *Thread.Wait* notwendig.

Monitore sind in verschiedenen Ausprägungen unter anderem in den Sprachen Concurrent Pascal, Pascal-Plus, Modula-2 sowie, wie bereits besprochen, in Modula-3 implementiert [Brinch Hansen 75] [Welsh, Bustard 79] [Wirth 85] [Nelson 91].

3.5.3 Kommunikationskanäle und Rendezvous

Bei den bisher besprochenen Synchronisationsmechanismen kommunizieren mehrere nebenläufige Prozesse über die durch Synchronisationsmechanismen kontrollierte Modifikation von Variablen. Diese Art der Kommunikation und Synchronisation setzt das Vorhandensein eines gemeinsamen Speichers voraus. Hoare präsentierte 1978 ein allgemeines mathematisches Modell zur Beschreibung kommunizierender, konkurrierender Prozesse, das unter anderem von Randbedingungen wie dem Vorhandensein eines gemeinsamen Speichers abstrahiert [Hoare 78]. CSP[3]

- leistet die Formalisierung des Begriffs des *sequentiellen Prozesses* sowie die Einführung des Prozeßkonzepts als mathematische Abstraktion der Interaktion zwischen einem System und dessen Umgebung;
- erlaubt die Beschreibung und programmtechnische Behandlung nichtdeterministischer Vorgänge;
- gibt eine mathematische Beschreibung der Kommunikation von Prozessen als Spezialfall von Prozeßinteraktion.

Die Programmiersprache CSP nutzt die von Dijkstra eingeführten Sprachmittel der parallelen Komposition, den sogenannten bewachten Befehlen (Guarded Commands), sowie der nichtdeterministischen Auswahl zwischen Alternativen [Dijkstra 75]. Die parallele Komposition von Prozessen beschreibt die Zusammenfassung nebenläufiger Prozesse zu einem Prozeß, dessen Verhalten sich aus dem Verhalten der einzelnen Unterprozesse ergibt. Die Notation

$$X :: [\, P \parallel Q \parallel R \,]$$

bezeichnet einen Prozeß X, der sich entsprechend dem nebenläufigen Verhalten der Unterprozesse P, Q und R verhält.

Ein bewachter Befehl besteht aus einem boolschen Ausdruck, dem Wächter (Guard), sowie einer Liste von Programmanweisungen. Diese Liste wird nur dann ausgeführt, wenn die Auswertung des Wächters den boolschen Wert *true* ergibt. Mehrere bewachte Anweisungen können zu einer Liste kombiniert werden. Ergibt die Auswertung mehrerer Wächter den boolschen Wert *true*, so wird die Entscheidung, welche der den Wächtern zugeordneten Anweisungsfolgen bearbeitet wird, nichtdeterministisch getroffen. Hoare benutzte für dieses Konstrukt die folgende Notation:

[3] Communicating Sequential Processes.

$$X :: [\ g1 \rightarrow P \ \square\ g2 \rightarrow Q \ \square\ g3 \rightarrow R\ ]$$

Diese Anweisungsfolge beschreibt einen Prozeß X, der sich wie die Prozesse P, Q oder R verhält, wenn einer der Wächter *g1*, *g2* oder *g3* zum boolschen Wert *true* evaluiert. Ist dies bei mehr als einem Wächter der Fall, so wird die Auswahl zwischen den auf die Wächter folgenden Prozesse nichtdeterministisch durchgeführt. Ergibt keine Auswertung eines Wächters den Wert *true*, so kann der gesamte Prozeß X nicht weiter ausgeführt werden.

Zur Kommunikation zwischen Prozessen dienen Ein- und Ausgabeanweisungen, die jeweils einen Wert an einen bezeichneten Prozeß übermitteln bzw. einen Wert von einem Prozeß empfangen. Der Nachrichtenaustausch findet erst dann statt, wenn beide beteiligten Prozesse eine passende Kommunikationsanforderung absetzen. Diese Form des Nachrichtenaustausches wird auch als *Rendezvous* bezeichnet. Die Anweisungsfolgen

$$X :: *[\ a?x \rightarrow P \ \square\ b?x \rightarrow Q\ ] \qquad Y :: *[\ c!y \ldots\ ]$$

beschreiben einen Prozeß X, der Werte von den Prozessen a oder b liest und sich dann wie die Prozesse P und Q verhält. Der Prozeß Y gibt dagegen einen Wert an den Prozeß c aus. In diesem Beispiel werden Ein- und Ausgabeanweisungen als Wächter benutzt. Diese Anweisungen ergeben jeweils den boolschen Wert *false*, wenn die durch sie adressierten, schreibenden und lesenden Prozesse beendet sind. Das Zeichen $*$ beschreibt die wiederholte Ausführung der jeweiligen Anweisungsfolgen. Im ursprünglichen Entwurf der Sprache CSP war eine Kommunikation nur durch direkte Aufzählung der beteiligten Prozesse möglich. Dies hat den Nachteil, daß alle Prozesse von vornherein bekannt sein müssen. In einer späteren, erweiterten Fassung von CSP werden benannte Kommunikationskanäle eingeführt, die eine indirekte Adressierung von Prozessen gestatten [Hoare 85]. In diesem Fall entsprechen die obigen Variablen a, b und c solchen Kommunikationkanälen.

Eine Synchronisation nebenläufiger Prozesse wird in CSP allein durch die Kommunikation über benannte Kanäle erreicht. Ein gemeinsamer Speicher, dessen nebenläufige Benutzung durch andere Synchronisationsmechanismen, wie z.B. Semaphore, abgesichert wird, ist nicht notwendig. Hierdurch sind diese Mechanismen für verteilte Systeme geeignet, die im allgemeinen über keinen gemeinsamen Speicher verfügen.

Bewachte Befehle mit nichtdeterministischer Auswahl und das Konzept des Nachrichtenaustausches über Rendezvous wurden in zahlreiche weitere Sprachen übernommen. Hier ist im speziellen die Sprache Occam zu nennen, die eine sehr direkte Umsetzung der Konzepte von CSP darstellt [Steinmetz 88]. Diese Sprache ist zur Programmierung spezieller Mikroprozessoren, den sogenannten Transputern, entwickelt worden, die bereits eine maschinelle Unterstützung zur Realisierung nebenläufiger Prozesse und die Kommunikation über physikalische Kanäle bieten [Inmos 85]. Das Konzept der Rendezvous, zusammen mit der nichtdeterministischen Auswahl, finden in den *select*-Anweisungen der Sprachen Synchronous C++ [Abel 93] und Ada ihre Entsprechung [Burns et al. 87]. Darüber hinaus stellt Ada weitere Synchronisationmechanismen wie Monitore und Semaphore zur Verfügung. Rendezvous sowie nichtdeterministische Auswahl sind ebenfalls in der experimentellen Sprache Orca zu finden [Bal 95]. Implementationstechniken zur Realisierung verteilter Rendezvous werden detailliert von Atkinson beschrieben [Atkinson et al. 88].

3.5.4 Weitere Mechanismen

In den bisher besprochenen Verfahren zur Synchronisation werden die eigentlichen Synchronisationsanweisungen teilweise explizit benutzt, wie zum Beispiel die *LOCK*-Anweisung in Modula-3 oder das *AWAIT* der Pascal-Plus Monitore, aber auch implizit von Compilern oder Laufzeitsystemen erzeugt. Der automatische, gegenseitige Ausschluß bei kritischen Regionen, Monitoren oder die Synchronisation durch die Kommunikationsmechanismen von CSP sind Beispiele hierfür. Gemeinsam ist diesen Verfahren die Verteilung von Anweisungen für die Synchronisation über den gesamten Programmtext. Ein alternativer Ansatz hierzu ist die Spezifikation von Synchronisationsbedingungen, die von den eigentlichen Programmanweisungen getrennt sind. Die Realisierung der konkreten Synchronisationsanweisungen wird dann vollständig durch Compiler bzw. Laufzeitsysteme übernommen.

Pfadausdrücke: Die von Campell und Habermann vorgeschlagenen Pfadausdrücke beschreiben Synchronisationsanweisungen und Ausführungsreihenfolgen in einer regulären Ausdrücken ähnlichen Form. Synchronisationsbedingungen werden getrennt vom sonstigen Programmtext formuliert [Habermann 75]. Das (sequentielle) Verhalten der Programme wird durch Prozedur- und Typendeklarationen beschrieben, die eine mit Monitoren verwandte Struktur aufweisen. Im einzelnen können mit Pfadausdrücken folgende Arten der Prozedurausführung spezifiziert werden:

path *P1* **end**	:	Die Prozedur *P1* kann wiederholt ausgeführt werden. Es ist jedoch zu einem bestimmten Zeitpunkt nur einem Prozeß möglich, innerhalb von *P1* aktiv zu sein. Diese entspricht dem Verhalten einer kritischen Region oder eines Monitors.
{*P1*}	:	Die Prozedur *P1* kann nebenläufig durch mehrere Prozesse ausgeführt werden. Es können jedoch nur dann weitere Prozesse *P1* ausführen, solange ein anderer Prozeß noch in *P1* aktiv ist.
P1 ; *P2* ; *P3*	:	Diese Notation beschreibt die sequentielle Ausführungsreihenfolge der genannten Prozeduren *P1*, *P2* und *P3*. Die Prozeduren können durch verschiedene Prozesse ausgeführt werden, die Reihenfolge der Ausführung findet jedoch wie angegeben statt. Beispielsweise wird ein Prozeß, der mit der Ausführung der Prozedur *P3* beginnen möchte verzögert, bis die Prozeduren *P1* und *P2* bearbeitet wurden.
P1, *P2*, *P3*	:	Bei dieser Selektionsanweisung kann genau eine der Prozeduren ausgeführt werden. Die Auswahl, welche der Prozeduren ausgeführt wird, findet nichtdeterministisch statt.

Der Name einer Prozedur darf in einem Pfadausdruck nur einmal erscheinen, sonst können die Ausdrücke beliebig verschachtelt sein. Das folgende Beispiel zeigt die Synchronisation von Leser- Schreibervorgängen durch Pfadausdrücke [Freisleben 86, Seite 122–123].

path *requestwrite* ***end;***
path {*read*} , *write* ***end;***

```
procedure read:
begin
  (* Lesevorgang *)
end;

procedure write:
begin
  (* Schreibvorgang *)
end;

procedure requestwrite:
begin write end;
```

Ein Schreibvorgang wird durch Ausführung der Prozedur *requestwrite* eingeleitet, weitere Schreibvorgänge werden bis zur Beendigung der Prozedur *write* verzögert. Ebenfalls werden gleichzeitige Leseversuche bis zur Beendigung der Prozedur *write* verzögert. Da die Prozedur *write* vor der Prozedur *requestwrite* beendet wird, erhalten wartende Leser Vorrang vor wartenden Schreibern. Hat ein Prozeß mit dem Lesevorgang begonnen, können weitere Prozesse Lesevorgänge einleiten.

Ein Beispiel für einen komplexen Pfadausdruck ist:

```
path P1, (P2; P3; {P4; P5}), P6 end;
```

Hierdurch wird die wiederholte Ausführung der notierten Prozeduren beschrieben. Zunächst kann entweder *P1* oder *P2* oder *P6* ausgeführt werden. Bei der Auswahl von *P2* wird zunächst *P2*, dann *P3* sequentiell bearbeitet. Die folgende Ausführung des Ausdrucks {*P4*; *P5*} kann von mehreren Prozessen parallel durchgeführt werden. Jedoch muß ein Prozeß, der *P5* aufruft, warten, bis die Bearbeitung von *P4* beendet ist.

Pfadausdrücke erinnern durch die Art ihrer Notation und Kombinierbarkeit an reguläre Ausdrücke. Wie bei diesen ist es nicht möglich, z.B. Informationen über den Zustand der durch einen Pfadausdruck beschriebenen Abläufe zu erhalten. So ist es in einem Pfadausdruck nicht möglich, die Anzahl der auf die Ausführung einer Prozedur wartenden Prozesse zu erhalten bzw. zu benutzen. Synchronisationsprobleme, die diese Art von Informationen benötigen, lassen sich nur durch Verwendung weiterer Synchronisationsmechanismen oder durch Erweiterungen der Pfadausdrücke lösen [Freisleben 86]. Eine Anwendung von Pfadausdrücken für die Kommunikation von Prozessen in verteilten Rechnersystemen finden sich bei McLaughry [McLaughry 95]. Eine Implementation von Pfadausdrücken mittels Semaphoren gibt Habermann an [Habermann 75].

3.5.5 Zusammenfassung

In allen Sprachen, die die nebenläufige Ausführung von Programmabschnitten erlauben, finden sich verschiedenartige Mechanismen zur Prozeßsynchronisation. Wesentliche Unterschiede bestehen in der Möglichkeit bzw. Notwendigkeit, Synchronisationsanweisungen explizit zu formulieren oder diese Arbeit einem Werkzeug, wie z.B. einem Compiler, zu überlassen. Unterschiede bestehen weiterhin in der Effizienz möglicher Implementationen. Pfadausdrücke unterliegen bezüglich der Menge der lösbaren Probleme den gleichen Einschränkungen wie die regulären Ausdrücke.

3.6 Synchronisation in persistenten Objektsystemen

Für Synchronisationsmechanismen in persistenten Objektsystemen sind eine Reihe verschiedener Ansätze vorhanden. Für alle gilt die Feststellung, daß die in herkömmlichen, relationalen Datenbanksystemen verfolgte Strategie zur Herstellung von Datenkonsistenz, der Verwendung atomarer Transaktionen nach dem ACID-Modell, für typische Aufgabenstellungen persistenter Objektsysteme nur ungenügend geeignet ist. Im Gegensatz zu solchen Systemen steht in persistenten Objektsystemen die Formulierung von Operationen auf sehr komplexen Objektstrukturen mittels der vorhandenen Sprachmittel im Vordergrund. Ein weiterer Faktor ist das Vorhandensein lang andauernder, kooperativer Aktivitäten, zu denen die in herkömmlichen Datenbanksystemen vorgesehenen, für sehr kurz andauernde Transaktionen ausgelegten Mechanismen, nicht passen.

Im einzelnen gliedern sich die in persistenten Objektsystemen verwirklichten Mechanismen zur Sicherstellung von Datenkonsistenz in Methoden zur Spezifikation erweiterter, kooperativer Transaktionsmechanismen und Mechanismen, die, wie die in dieser Arbeit betrachteten Mechanismen, eine explizite, nichttransaktionale Kontrolle über nebenläufige Zugriffe auf Datenobjekte gestatten.

Zu den erweiterten, transaktionalen Ansätzen zählen Methoden, die die strikte Forderung nach Isolation nach dem ACID-Modell lockern und Zugriff auf in laufenden Transaktionen benutzte Objekte gestatten [Gray, Reuter 93]. Ein Beispiel hierfür sind die sogenannten Transaktionsgruppen [Munro 93]. In solchen Transaktionsgruppen werden Transaktionen in einem Baum angeordnet, dessen Elemente aus Knoten und Blättern bestehen. Die zu einem Knoten gehörenden Blätter stellen hierbei kooperative Transaktionen dar, die gegenseitig auf in ihnen bearbeitete Objekte zugreifen können. Transaktionsgruppen werden durch Spezifikation zu Pfadausdrücken ähnlichen Mustern spezifiziert, die die gewünschte Reihenfolge von Transaktionen beschreiben [Habermann 75]. Weiterhin werden durch derartige Muster auch Operationen beschrieben, die sich gegenseitig ausschließen. Im System ist dadurch eine Menge kooperativer Transaktionen vorhanden. Die Rücknahme einer einzelnen Transaktion betrifft nur einen möglicherweise sehr kleinen Teil der durchgeführten Änderungen und nicht das gesamte System. Eine Implementation von Transaktionsmustern liegt in der Sprache DPS-Algol vor, einer Erweiterung der persistenten Sprache PS-Algol [Munro 93] [Atkinson et al. 83].

Eine Erweiterung von PS-Algol um Sprachkonstrukte zur Formulierung nebenläufiger Threads ist in der Sprache DPS-Algol verwirklicht [Krablin 87]. In dieser Sprache stehen durch Threads implementierte, kooperative Aktivitäten im Vordergrund. Threads synchronisieren Zugriffe auf Datenobjekte mittels bedingter kritischer Regionen (siehe Abschnitt 3.5.1). Transaktionale Vorgänge im System werden durch kritische Regionen und Funktionen höherer Ordnung verwirklicht.

Ein System, das wie das Tycoon-System über persistente Threads verfügt, ist das Napier88-System der Universität St. Andrews [Kirby et al. 95]. Nebenläufige Zugriffe auf Datenobjekte können in Napier88 durch Semaphore koordiniert werden [Kirby et al. 96]. Weiterhin werden Mittel zur Spezifikation der Sichtbarkeit von Transaktionen nach dem CACS-Modell[4] bereitgestellt [Stemple, Morisson 92].

[4] Communicating Actions Control System.

Kapitel 4

Synchronisationsmechanismen für persistente Objektsysteme

Wie aus dem vorangegangenen Kapitel ersichtlich ist, existieren die verschiedensten Verfahren für die Synchronisation nebenläufiger Aktivitäten. In diesem Kapitel wird die Auswahl und Implementation grundlegender Verfahren für das persistente Objektsystem Tycoon beschrieben. Im folgenden Abschnitt werden die für Synchronisationsmechanismen relevanten besonderen Eigenschaften des Tycoon-Systems erörtert und die Anforderungen an die zu implementierenden Mechanismen festgelegt. Die Spezifikation der Verfahren sowie Beispiele für ihre Benutzung ist Thema von Abschnitt 4.2. Der Abschnitt 4.3 beschreibt die wichtigsten Aspekte der konkreten Implementation der Verfahren im Tycoon-System. Von der Einordnung der vorgestellten Verfahren in existierende Standards für die Programmierung nebenläufiger Aktivitäten handelt Abschnitt 4.3.3. Im Abschnitt 4.4 werden höhere Synchronisationsmechanismen auf der Basis der in den vorangegangenen Abschnitten behandelten Mechanismen implementiert.

4.1 Anforderungen und Auswahl

Die im Kapitel 3 vorgestellten Synchronisationsmechanismen zeigen, wie groß das Spektrum der zur Verfügung stehenden Alternativen zur Behandlung von Synchronisationsproblemen ist. Um zu einer Entscheidung über die zu implementierenden Synchronisationverfahren zu gelangen, ist daher zunächst die Erörterung folgender Punkte notwendig:

1. Die für Synchronisationsmechanismen relevanten Eigenschaften des Tycoon-Systems sind zu untersuchen. Hierzu gehören die Auswirkungen des hierarchischen Aufbaus des Systems auf Synchronisationsprobleme sowie Eigenschaften der einzelnen Komponenten Laufzeitsystem, Speicherprotokoll, Objektspeicher und der Sprache TL.

2. Es sind die für die Synchronisation in persistenten Objektsystemen wünschenswerten Eigenschaften der Synchronisationsmechanismen festzulegen.

3. Es ist zu entscheiden, auf welcher der verschiedenen Ebenen der Synchronisationsmechanismen die zu implementierenden Mechanismen angesiedelt sein sollen.

4.1.1 Eigenschaften des Tycoon-Systems

Für das zur Verfügung stehende persistente Objektsystem Tycoon sind die folgenden, für Synchronisationsmechanismen wichtigen Eigenschaften in Betracht zu ziehen:

- Das Tycoon-Laufzeitsystem ist als virtuelle Maschine implementiert (Abschnitt 4.3.1), die ein Einprozessorsystem simuliert [Mathiske 96]. Diese virtuelle Maschine steht für eine Vielzahl von Betriebssystemen zur Verfügung.
- Betriebssystemfunktionalität, wie Hauptspeicherverwaltung und Ein- und Ausgabefunktionen, wird durch das Laufzeitsystem gekapselt und unabhängig vom eigentlichen Betriebssystem zur Verfügung gestellt. Ein Programmierer findet auf allen eingesetzten Betriebssystemen eine einheitliche Programmierschnittstelle vor.
- Tycoon-Threads, die nebenläufige Aktivitäten ausführen, sind als Koroutinen implementiert, die zeitscheibengesteuert die Ausführungskontrolle an weitere Koroutinen abgeben. Es existiert demzufolge keine zentrale Instanz im Laufzeitsystem, die präemptiv die Zuteilung von Rechenzeit an Threads verwaltet. Erst auf der Ebene der Programmiersprache TL erscheinen diese Koroutinen als voneinander unabhängige Threads. Eine in einigen Betriebssystemen bereits vorhandene Unterstützung von Threads [Sun 93] wird aus Gründen der Portabilität nicht benutzt.
- Zugriffe auf TL-Objekte erfolgen über die Objektspeicherschnittstelle TSP, die von dem konkret eingesetzten Objektspeicher abstrahiert und bereits einen atomaren Zugriff auf nichtzusammengesetzte TL-Objekte garantiert. Dieses Verhalten entspricht der Eigenschaft des unteilbaren Speicherzugriffs in herkömmlichen, nicht persistenten Systemen. Laufzeitsystem, TSP und Objektspeicher stellen voneinander logisch und unter Umständen auch implementationstechnisch getrennte Schichten des Tycoon-Systems dar (siehe Abbildung 2.1).
- Die Programmiersprache TL bietet durch Eigenschaften wie Funktionen höherer Ordnung und Polymorphismus Möglichkeiten, über die viele der in Abschnitt 3.5 vorgestellten Programmiersprachen nicht verfügen. Ein weiterer wesentlicher Gesichtspunkt ist die Möglichkeit, Einfluß auf die Syntax der Programmiersprache TL durch Verwendung von Syntaxerweiterungen nehmen zu können [Schröder 94].
- Tycoon-Objekte sind mobil und können zwischen verschiedenen Objektspeichern migrieren [Mathiske 96].

4.1.2 Anforderungen an Synchronisationsmechanismen

Unter Berücksichtigung der oben genannten Punkten werden nachfolgend für die zu implementierenden Synchronisationsmechanismen folgende Anforderungen festgelegt:

- Da das Tycoon-Laufzeitsystem auf einer Vielzahl von sehr unterschiedlichen Betriebssystemen implementiert ist, sollen auch die Synchronisationsmechanismen mit angemessenem Aufwand auf weitere Systeme portierbar sein. Es dürfen daher möglichst wenige funktionale Abhängigkeiten vom vorliegenden Betriebssystem vorhanden sein.

- ▷ Die in Kapitel 3 und Abschnitt 3.5 vorgestellten Mechanismen bieten verschiedene Ansätze zur Behandlung von Synchronisationsproblemen an. Wesentliche Unterschiede ergeben sich in dem Ausmaß der Unterstützung einer sicheren, programmtechnischen Lösung von Synchronisationsproblemen. Zur Realisierung dieser Mechanismen ist immer mindestens ein grundlegendes Verfahren aus Abschnitt 3.4 notwendig, das seinerseits wieder auf einer der in Abschnitt 3.3 beschriebenen Basismethoden aufbauen muß. Um eine Weiterentwicklung und Erweiterbarkeit von Synchronisationsmechanismen zu gewährleisten, ist es sinnvoll, diesen hierarchischen Aufbau ebenfalls im Tycoon-System zur Verfügung zu stellen. Dieser Ansatz stellt eine größtmögliche Flexibilität bei der Lösung von Synchronisationsproblemen sicher.

- ▷ Es sollten verschiedene Programmierstile, wie die Synchronisation mittels Nachrichtenaustausch über Kommunikationskanäle und über gemeinsam benutzte Variablen, möglich sein.

- ▷ Abhängig vom vorliegenden Synchronisationsproblem, werden Synchronisationsmechanismen sehr oft von den beteiligten Threads aufgerufen. Bei der Implementation der Mechanismen muß daher besonders auf eine hohe Leistungsfähigkeit geachtet werden.

- ▷ Mobilität ist eine orthogonale Eigenschaft der Tycoon-Objekte und soll daher auch für Synchronisationsprimitive und erweiterte Verfahren gelten. Es ist beispielsweise denkbar, daß mehrere TL-Threads, die sich untereinander über bestimmte Mechanismen synchronisieren, gemeinsam in einen entfernten Objektspeicher migrieren. Die Synchronisationsbeziehungen zwischen diesen Threads müssen auch nach der Migration erhalten bleiben.

- ▷ Die Mechanismen sollen keine Einschränkungen bezüglich der Menge der lösbaren Synchronisationsprobleme aufweisen.

4.1.3 Aufbau der Mechanismen

Die durch Syntaxerweiterungen gegebene Möglichkeit, die Sprache TL nachträglich zu modifizieren, gestattet es, sich bei der Auswahl der Synchronisationsmechanismen zunächst auf einen leicht implementierbaren Satz von Synchronisationsprimitiven zu beschränken. Wie bereits in Abschnitt 3.5 besprochen, können die dort vorgestellten Synchronisationsmethoden auf eine Implementation durch Semaphore zurückgeführt werden. Die universellen Eigenschaften des Semaphorkonzepts sowie seine leichte Implementierbarkeit lassen Semaphore auch für persistente Objektsysteme als geeignete Kandidaten für einen grundlegenden Synchronisationsmechanismus erscheinen. Für Semaphore gelten zwar die in Abschnitt 3.4 besprochenen Nachteile wie Unübersichtlichkeit und Fehleranfälligkeit, sie sollten jedoch wegen ihrer Allgemeinheit und Mächtigkeit dem Programmierer zur Verfügung gestellt werden. Für die leichte Programmierbarkeit spezieller Anforderungen wie die gleichzeitige Manipulation von Semaphoren erscheint es darüber hinaus sinnvoll, erweiterte Semaphoroperationen, wie P- und V-Operationen auf mehreren Semaphoren, zu implementieren.

Zur Programmierung von Monitoren bietet es sich an, das Konzept und den Aufbau der Mutexe und Bedingungsvariablen (Condition variables) der Sprache Modula-3 zu überneh-

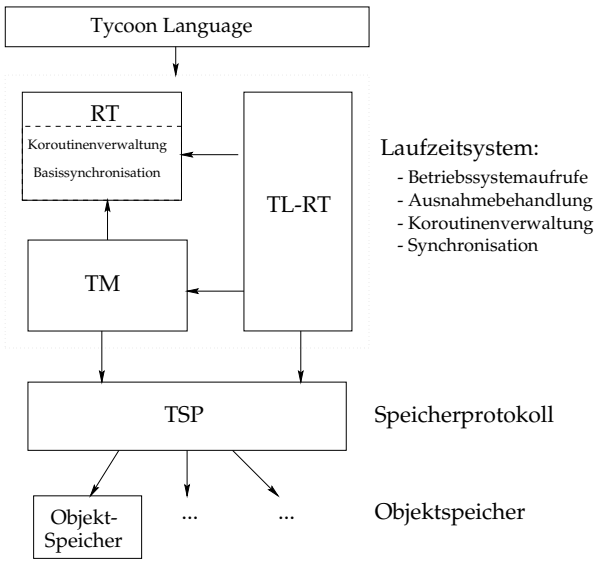

Abbildung 4.1: Synchronisationsmethoden und Laufzeitsystem

men [Horning et al. 93]. Diese Mechanismen entsprechen im wesentlichen dem Monitorkonzept von Hoare (Abschnitt 3.5.2). Mutexe sind leicht durch eine geringfügige Modifikation der Semaphore, durch die Registrierung des zugeordneten Threads, der aktuell das Mutex hält, zu implementieren.

Für alle weiteren, in Abschnitt 3.4 genannten Verfahren wie Rendezvous und kritische Regionen werden Erweiterungen der Sprache TL vorgesehen. Beispielhafte Syntaxerweiterungen für Rendezvous werden in Abschnitt 4.4 behandelt.

4.1.3.1 Einordnung in die Systemarchitektur

Eine Voraussetzung für jede Art von Synchronisation ist die Sicherstellung eines unteilbaren Zugriffs auf gemeinsam genutzte Objekte. Diese Voraussetzung wird in Mehrprozessorsystemen durch Verwendung atomarer Maschineninstruktionen, in Einprozessorsystemen durch Unterbrechungssperren erfüllt. Im Tycoon-System erfolgt jeder Zugriff auf persistente Objekte über das Speicherprotokoll TSP, das unter anderem den unteilbaren Zugriff auf Objekte unterstützt. Diese im transaktionalen Sinn zu verstehende Eigenschaft des Speicherprotokolls ist eng mit den Eigenschaften des jeweils konkret eingesetzten Objektspeichers verknüpft, der eine derartige Funktionalität bereits unterstützten muß. Eine Erweiterung des TSP zur generischen Unterstützung von Transaktionen mit interner Parallelität wäre ein Ansatzpunkt für die Unterstützung kooperativer, nebenläufiger Aktivitäten. Die Implementation nebenläufiger Aktivitäten ist jedoch unabhängig von Speicherprotokoll und Objektspeicher im Laufzeitsystem angesiedelt. Zur generischen Unterstützung von Synchronisation unter Beibehaltung des vorhandenen, geschichteten Systemaufbaus ist es erforderlich, die Basisfunktionalität für Synchronisationsmethoden ebenfalls im Laufzeitsystem anzusiedeln.

Abbildung 4.1 zeigt die Einordnung der Synchronisationsfunktionalität in den Aufbau des

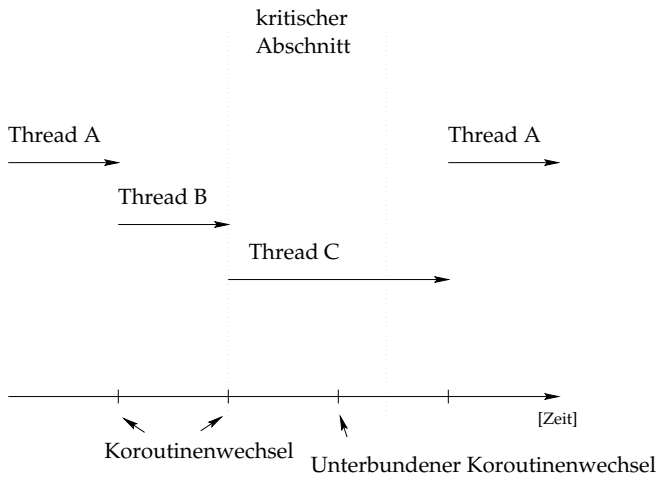

Abbildung 4.2: Unterbindung eines Koroutinenwechsels

Tycoon-Laufzeitsystems. Ohne die Erläuterung des Systemaufbaus aus Abschnitt 4.3.1 vorwegzunehmen, ist der genannten Abbildung die Isolation der Basisfunktionalität zur Synchronisation auf das Modul *RT*, in dem die Koroutinenverwaltung des Tycoon-Systems angesiedelt ist, zu entnehmen.

4.1.3.2 Basismethoden

Es stellt sich die Frage, welcher Mechanismus der Basismethoden aus Abschnitt 3.3 zur Implementation weitergehender Synchronisationsmethoden benutzt werden soll. Wie bereits angemerkt, ist das Tycoon-Laufzeitsystem in seiner jetzigen Form als von Synchronisationsmethoden des Betriebssystems unabhängige, virtuelle Maschine konzipiert. Auch auf Mehrprozessormaschinen wird eine Instanz des Tycoon-Systems stets als ein einzelner Benutzerprozeß ausgeführt. Nebenläufigkeit wird vollständig vom Laufzeitsystem selbst kontrolliert und greift auf keine der in Mehrprozessorbetriebssystemen vorhandenen Systemaufrufe zur Kontrolle nebenläufiger Aktivitäten zurück. Diese, die Portabilität des Tycoon-Systems garantierende Eigenschaft, läßt die Verwendung betriebssystemspezifischer Mechanismen oder den Einsatz atomarer Maschineninstruktionen als nicht zweckmäßig erscheinen. Es ist daher sinnvoll, für Basissynchronisationsmethoden den Aufbau des Laufzeitsystems als virtuelle Maschine auszunutzen und direkt in die interne Koroutinenverwaltung einzugreifen. Atomare Aktionen von Tycoon-Threads, die intern im Laufzeitsystem als Koroutinen realisiert sind, können durch eine Verhinderung eines Koroutinenwechsels sichergestellt werden. Ein solcher Mechanismus entspricht den in Abschnitt 3.3 vorgestellten Unterbrechungssperren. Abbildung 4.2 verdeutlicht die Unterbindung eines Koroutinenwechsels während eines kritischen Abschnitts. Dieser Mechanismus sollte nur zur Sicherung kurzer, kritischer Abschnitte in den darauf aufbauenden Synchronisationsmechanismen benutzt werden, um die Nebenläufigkeit innerhalb des Systems nicht unnötig einzuschränken.

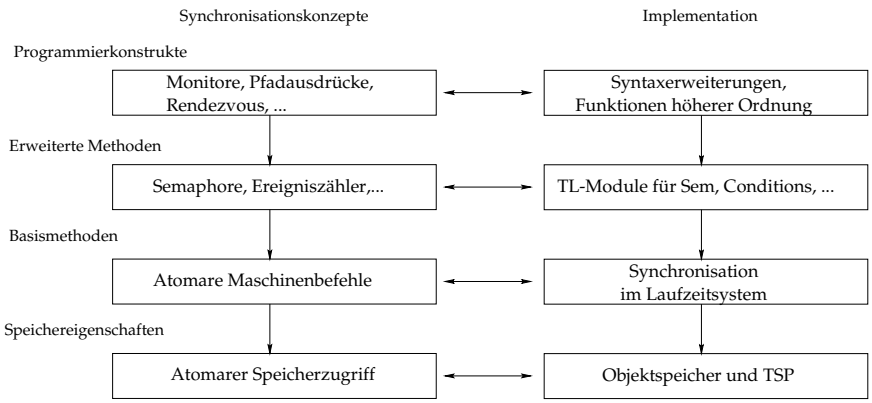

Abbildung 4.3: Aufbau und Zuordnung der Synchronisationsmethoden

4.1.3.3 Zusammenfassung

Die Unterteilung des Tycoon-Systems in Laufzeitsystem und Speicherprotokoll TSP sowie die Konzeption des Laufzeitsystems als virtuelle Einprozessormaschine ermöglichen es, Synchronisationsmechanismen unter Verwendung der Verhinderung von Koroutinenwechseln zu entwickeln. Der Aufbau der Synchronisationsmethoden gestaltet sich wie folgt:

- ▷ Bereitstellung eines Mechanismus, der in kritischen Abschnitten eine Zuteilung von Rechenzeit an weitere Koroutinen unterbindet.
- ▷ Implementation von Semaphoren und Mutexen sowie von Bedingungsvariablen als eigenständige Module. Diese Module werden mittels der Programmiersprache TL implementiert.
- ▷ Implementation höherer Synchronisationsmechanismen unter Benutzung von Syntaxerweiterungen und Funktionen höherer Ordnung.

Abbildung 4.3 gibt einen Überblick über den Aufbau der Synchronisationsmechanismen und die Einordnung in die Hierarchie der Synchronisationsmethoden (vgl. Abschnitt 3.2).

4.2 Bibliotheken für Synchronisationsmechanismen

Der Aufbau der Synchronisationsprimitive gliedert sich analog zu der in Abbildung 4.3 gezeigten Darstellung. Aufbauend auf dem im Systemmodul *thread* (siehe Anhang A.4) implementierten Mechanismus *thread.atomic* zur zeitweisen Sperrung von Koroutinenwechseln, werden in diesem Abschnitt Tycoon-Module für Semaphore, Mutexe und Bedingungsvariablen vorgestellt. Es werden die implementierten Funktionsaufrufe der einzelnen Module sowie einige Bespiele für ihre Anwendung beschrieben. Die Beschreibung der Implementation der Funktion *thread.atomic* folgt in Abschnitt 4.3.1.

4.2.1 Semaphore

Im Modul *semaphore* sind allgemeine Semaphore mit beliebigem Anfangswert für die internen Zähler eines Semaphors implementiert. Vom Modul werden der abstrakte Datentyp *semaphore.T* sowie die für diesen Datentyp definierten Funktionen exportiert. Ein Element des Typs *semaphore.T* wird mittels der Operation

new(count :Int) :T

instantiiert. Hierbei gibt der Wert der ganzzahligen Variablen *count* die Anzahl der Threads an, die ein Semaphor akquirieren können, ohne daß der Versuch der Akquisition zu einer Blockierung des aufrufenden Threads führt. Diese Funktionalität kann beispielsweise zur Verwaltung von Ressourcen benutzt werden, die einen gleichzeitigen Zugriff von *count*-Threads erlauben.

Für den Zugriff auf eine Instanz des Datentyps *semaphore.T* werden vom Modul die folgenden Funktionen exportiert:

wait(sem :T) :**Ok**

post(sem :T) :**Ok**

Die Funktionen *wait* und *post* entsprechen den in Abschnitt 3.4.2 beschriebenen *P*- und *V*-Operationen. Der Versuch, die Operation *wait* auf einen Semaphor anzuwenden, führt nach $count + 1$ Aufrufen dieser Funktion zur Suspendierung des aufrufenden Threads, wenn vorher keine Erhöhung des internen Zählers des Semaphors mittels der Funktion *post* erfolgte. Mit der Funktion *post* wird der interne Zähler des Semaphors um den Wert 1 erhöht und einer der auf das Semaphor wartenden Threads aktiviert. Auf einen Semaphor wartende Threads werden in einer Warteschlange verwaltet, um eine faire Zuteilung des Semaphors zu gewährleisten. Es ist zu beachten, daß die Funktionen *wait* und *post* für ein gegebenes Semaphor *sem* von verschiedenen Threads aufgerufen werden können. Hiermit läßt sich zusätzlich zum Schutz von kritischen Abschnitten ein Austausch von Signalen realisieren. Das folgende Beispiel beschreibt den Schutz eines kritischen Abschnitts sowie einen Signalaustausch:

```
(* Verwaltung des Zugriffs auf einen Ringpuffer des Typs :Int und der Länge N *)
let N = ...   (* Länge des Ringpuffers *)
let buffer = arrayOp.new(:Int N 0)
let protect = semaphore.new(1)
let full    = semaphore.new(0)
let empty   = semaphore.new(N)
let var inpointer = 0  and  var outpointer = 0
```

```
(* Ein Element schreiben *)
let putElement(element :Int) :Ok =
  begin
    semaphore.wait(empty)
    semaphore.wait(protect)
    buffer[inpointer] := element
    inpointer := {inpointer + 1} % N
    semaphore.post(protect)
    semaphore.post(full)
  end
```

```
(* Ein Element lesen *)
let getElement() :Int =
  begin
    semaphore.wait(full)
    semaphore.wait(protect)
    let element = buffer[outpointer]
    outpointer:= {outpointer + 1} % N
    semaphore.post(protect)
    semaphore.post(empty)
    element
  end
```

Im obigen Beispiel dient das Semaphor *protect* dem Schutz der Integrität des Ringpuffers *buffer* und der Variablen *inpointer* sowie *outpointer*. Diese Variablen geben die aktuelle Position für Schreib- und Lesevorgänge im Puffer an. Die Initialisierung des Semaphors *protect* mit dem Wert 1 garantiert, daß höchstens ein Thread innerhalb der Funktionen *getElement* und *putElement* aktiv ist. Demgegenüber dienen die Semaphore *full* und *empty* der Signalisierung des Zustandes des Ringpuffers. Ist die maximale Länge des Puffers erreicht, wurde auf das Semaphor *empty* N–mal zugegriffen, und Threads, die die Funktion *putElement* aufrufen, werden verzögert, bis durch einen Aufruf der Funktion *getElement* ein Element entnommen wird. Ein leerer Puffer führt zu einer Suspendierung eines *getElement* aufrufenden Threads durch das Semaphor *full*. Hierdurch wird ein Lesen aus einem leeren Puffer verhindert.

Die weiteren, im Modul *semaphore* enthaltenen Funktionen sind:

tryWait(sem :T) :Bool

waitMultipleOr(semarray :Array(T)) :T

*waitMultipleAnd(semarray :Array(T)) :***Ok**

*postMultiple(semarray :Array(T)) :***Ok**

Mittels der Funktion *tryWait* wird versucht, einen Semaphor *sem* zu akquirieren. Gelingt dies, ist der interne Zähler des Semaphors *sem* um den Wert 1 vermindert, und die Funktion gibt den boolschen Wert *true* zurück, andernfalls ist das Ergebnis der Funktion *false*. Diese Funktion führt in keinem Fall zu einer Suspendierung des aufrufenden Threads und kann zum versuchsweisen Zugriff auf kritische Abschnitte benutzt werden. Gelingt die Akquisition des Semaphors nicht, können beispielsweise andere Funktionen ausgeführt werden.

Die Funktionen *waitMultipleOr* und *waitMultipleAnd* dienen dem gleichzeitigen Zugriff auf eine Menge von Semaphoren. Die Funktion *waitMultipleOr* gibt ein freies Semaphor aus der in dem Array *semarray* angeführten Menge von Semaphoren zurück. Sind alle Semaphore belegt (alle Semaphore besitzen einen internen Zählerwert von ≤ 0), wird der aufrufende Thread verzögert. Die Freigabe eines oder mehrerer Semaphore führt zur erneuten Aktivierung des Threads, wobei im Falle mehrerer freier Semaphore eine zufällige Auswahl getroffen wird. Das beim Aufruf dieser Funktion manipulierte Semaphor wird als Wert von

waitMultipleOr zurückgegeben. Mit der Funktion *waitMultipleAnd* wird die gesamte Menge der in *semarray* angegebenen Semaphore manipuliert. Diese Funktion kehrt zurück, wenn alle Semaphore belegt werden konnten. Dadurch ist es beispielsweise einfach möglich, eine Menge von Betriebsmitteln oder kritischen Abschnitten gleichzeitig zu belegen. Es ist darauf hinzuweisen, daß die Operationen *waitMultipleOr* und *waitMultipleAnd* auch mittels der elementaren Semaphorfunktionen *wait* und *post* direkt programmiert werden können [Dijkstra 71]. Darüber hinaus muß bei der Operation *waitMultipleAnd* auf die Reihenfolge der im Array *semarray* aufgeführten Semaphore geachtet werden. Alle Threads, die eine *waitMultipleAnd*-Operation auf der gleichen (Teil)-Menge von Semaphoren ausführen, müssen die verwendeten Semaphore in gleicher Reihenfolge angeben, da sonst eine Verklemmung entstehen kann [Dijkstra 71].

4.2.2 Mutexe und Bedingungsvariable

Die Module *mutex* und *condition* vervollständigen die für das Tycoon-System implementierten Basissynchronisationsmechanismen. Diese Module dienen ebenfalls dem Schutz von kritischen Abschnitten sowie dem Signalisieren von Bedingungen. Mutexe bieten eine den binären Semaphoren ähnelnde Funktionalität, weisen aber im Gegensatz zu diesen einige Einschränkungen bezüglich ihrer Verwendung auf. Ein Mutex ist ein einfacher Mechanismus zur Sperrung von kritischen Abschnitten. Ein Mutex befindet sich immer in einem von zwei möglichen Zuständen, es ist entweder gesperrt oder frei. Ein gesperrtes Mutex kann nur von dem Thread, der das Mutex gesperrt hat, wieder freigegeben werden. Diese Eigenschaft verhindert die versehentliche Freigabe eines kritischen Abschnitts durch von der beabsichtigten Programmlogik eigentlich nicht berechtigte Threads, verhindert aber eine Kommunikation zwischen Threads analog zu dem im vorangegangenen Abschnitt beschriebenen Beispiel des Ringpuffers. Weiterhin ist es bei der Realisierung kritischer Abschnitte durch Mutexe nur durch zusätzliche Mechanismen möglich, daß sich mehrere Threads in einem solchen Abschnitt befinden. Ein Mutex stellt die einfachste Möglichkeit dar, kritische Abschnitte zu realisieren. Das Modul *mutex* ist wie folgt aufgebaut: Ein Element des Typs *mutex.T* wird mittels der Operationen

new() :T

newLocked():T

erzeugt. Die Funktion *new* gibt eine freies, nicht gesperrtes Mutex zurück, ein Aufruf der Funktion *newLocked* liefert ein bereits gesperrtes Mutex.

Die Zugriffsfunktionen für ein Element des Typs *mutex.T* bestehen aus den Aufrufen

lock(mutex :T) :**Ok**

unlock(mutex :T) :**Ok**

tryLock(mutex :T) :Bool

Der Aufruf der Funktion *lock* führt zu einer Suspendierung des aufrufenden Threads, falls das Mutex schon gesperrt wurde. Die Funktion *unlock* gibt ein gesperrtes Mutex wieder

frei und führt zu einer Aktivierung eines eventuell wartenden Threads. Wird die Funktion *unlock* nicht von dem Thread aufgerufen, der das Mutex gesperrt hat, wird der Ausnahmefehler *mutex.error* generiert. Auch im Modul *mutex* werden wartende Threads in einer Warteschlange verwaltet, um eine faire Zuteilung der Mutexe zu gewährleisten. Ein Aufruf der Funktion *tryLock* führt zur Sperrung des angegebenen (freien) Mutex, aber bei bereits gesperrtem Mutex nicht zur Suspendierung des aufrufenden Threads. Bei einem erfolgreichen Sperren des Mutex wird der boolsche Wert *true* zurückgegeben, andernfalls ergibt die Auswertung von *tryLock* den Wert *false*.

Zur Lösung von Programmierproblemen, die die Freigabe eines kritischen Abschnitts bis zu Eintritt eines spezifizierten Ereignisses erfordern, sind die Funktionen im Modul *condition* vorgesehen. Diese Funktionalität entspricht der in Abschnitt 3.5.2 beschriebenen Funktionalität der Modula-3–Monitore und wird in der folgenden Beschreibung der einzelnen Funktionsaufrufe näher erläutert.

new() :T

Die Funktion *new* erzeugt eine neue Bedingungsvariable des Typs *condition.T*. Auf einer solchen Variablen sind die folgenden Funktionen implementiert:

wait(m :mutex.T cond :T) **:Ok**

timedWait(m: mutex.T cond :T timeout :time.T) **:Ok**

signal(cond :T) **:Ok**

broadcast(cond :T) **:Ok**

Die Benutzung einer gegebenen Bedingungsvariablen erfolgt immer zusammen mit dem der Bedingungsvariablen logisch zugeordneten Mutex. Zur Ausführung der Operation *wait(mutex condition)* ist es erforderlich, daß der aufrufende Thread das Mutex *mutex* bereits gesperrt hat. Der Aufruf dieser Funktion entsperrt das Mutex und blockiert den Thread auf der der Bedingungsvariablen zugeordneten Warteschlange. Dies führt zur Freigabe des durch das Mutex geschützten kritischen Abschnitts, ein weiterer Thread kann nach Akquisition des Mutex in den kritischen Abschnitt eintreten.

Die Angabe eines maximalen Wartezeitraums bis zum Eintreten einer Bedingung ermöglicht der Aufruf der Funktion *timedWait*. Wird die Bedingung nicht bis zum Erreichen des angegebenen Zeitpunkts signalisiert, kehrt die Funktion *timedWait* nach erneutem Sperren des verwendeten Mutex zurück.

Die Erfüllung der mit einer Bedingungsvariablen verknüpften Bedingung wird durch die Funktion *signal* angezeigt. Der Aufruf dieser Funktion bewirkt eine Deblockierung eines auf die jeweilige Bedingungsvariablen wartenden Threads. Sind keine wartenden Threads vorhanden, erfolgt keine weitere Aktion. Die Funktion *broadcast* startet alle auf eine Bedingungsvariable wartenden Threads. Die Abbildungen 4.4 sowie 4.5 illustrieren die beiden folgenden Programmbeispiele zur Verwendung von Mutexen und Bedingungsvariablen.

Die Sperrung eines kritischen Abschnitts unter Verwendung von Mutexen zeigt folgendes Programmfragment (vgl. Abbildung 4.4):

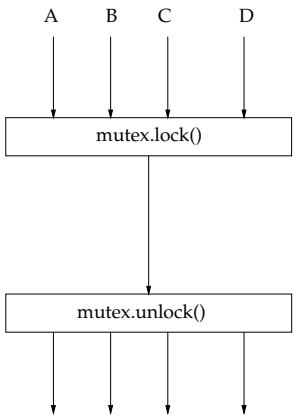

Abbildung 4.4: Sperrung kritischer Abschnitte

```
let lock = mutex.new()

mutex.lock(lock)
 (* kritischer Abschnitt *)
mutex.unlock(lock)
```

Das folgende, etwas anspruchsvollere Beispiel zeigt das Warten mehrerer Threads aufeinander (vgl. Abbildung 4.5):

```
let lock = mutex.new()
let cond = condition.new()
let var barrierCount = 0

let barrier(n :Int) =
 begin
  mutex.lock(lock)
    barrierCount := barrierCount + 1
    if barrierCount == n then
      condition.broadcast(cond)
      barrierCount := 0
    else
      while barrierCount < n do
        condition.wait(lock  cond)
      end
    end
   mutex.unlock(lock)
 end

let doA() = ...
let doB() = ...
```

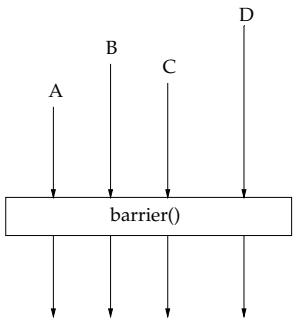

Abbildung 4.5: Synchronisation an einer Barriere

```
let funcA(n :Int  f :Fun():Ok) =
 begin
   ... f() ...
   barrier(n)
   ...
 end

let threadA = thread.fork(fun(self :thread.T(Ok)) funcA(2 doA))
let threadB = thread.fork(fun(self :thread.T(Ok)) funcA(2 doB))
```

Jeder der N Threads ruft zu einem bestimmten Zeitpunkt die im obigen Programmfragment angegebene Funktion *barrier* mit dem Argument N auf. Der Aufruf dieser Funktion bewirkt eine Blockierung aller *barrier* aufrufenden $N-1$ Threads. Alle Threads warten an der Barriere *barrier* aufeinander, bis beim N–ten Thread schließlich die Ausführung der Funktion *broadcast* zur Wiederaktivierung aller auf die Bedingungsvariable *cond* wartenden Threads führt (Funktion *barrier*).

Ein analoges Beispiel zeigt das Warten eines Threads auf N weitere Threads:

```
let lock  = mutex.new()
let cond  = condition.new()
let var N = 0

(* Initialisierung der gemeinsam benutzten Variablen N *)
let initWait(n :Int) =
  begin
    mutex.lock(lock)
      N  :=  n
    mutex.unlock(lock)
  end

(* Auf das Eintreffen von N Threads warten *)
let waitForAll() =
  begin
```

```
    mutex.lock(lock)
      while N > 0 do
        condition.wait(lock cond)
      end
    mutex.unlock(lock)
  end

(* Alle wartenden Threads wieder aktivieren *)
let barrier() =
  begin
    mutex.lock(lock)
      N := N - 1
      if N == 0 then
        condition.broadcast(cond)
      else
        while N > 0 do
          condition.wait(lock cond)
        end
      end
    mutex.unlock(lock)
  end
```

Von einem Thread A werden N weitere Threads gestartet. Vor dem Starten weiterer Threads initialisiert Thread A den gemeinsam benutzten Zähler N durch die Funktion *initWait*. Thread A wartet daraufhin durch Aufruf der Funktion *waitForAll* auf die Erfüllung der Bedingung *cond* durch alle durch ihn initiierten Threads. Die durch Thread A gestarteten, untergeordneten Threads warten durch Aufruf der Funktion *barrier* aufeinander. Der N–te, *barrier* aufrufende Thread aktiviert alle durch den Thread A gestarteten Threads, sowie den Thread A selbst.

Ein Beispiel für die Verwendung der Funktion *timedWait* zeigt der folgende Programmausschnitt. Ein Thread, dessen maximaler, akzeptabler Zeitraum bis zum Eintreten des Ereignisses *cond* drei Sekunden betragen soll, initialisiert die Variable *timeout* und ruft die Funktion *timedWait* mit diesem Parameter auf. Zuvor wird in einer Programmschleife die mit der Bedingungsvariable *cond* verknüpfte Bedingung geprüft. Ist diese erfüllt, wird mit der Ausführung weiterer Aktionen fortgefahren und der Schleifenkörper verlassen. Andernfalls wird die Überschreitung des gewählten Zeitpunkts getestet. Im Fall einer Überschreitung des angegebenen Zeitpunkts wird eine Ausnahmebedingung erzeugt:

```
let m = mutex.new() and cond = condition.new()

mutex.lock(m)
  (* Das Timeoutintervall beträgt drei Sekunden *)
  let timeout = time.add(time.now() time.create(0 0 3 0))
  loop
    if < Bedingung erfüllt > then
      (* Weitere Aktionen des Programms *)
      exit
```

```
    end
    if time.greaterEqual(time.now()  timeout) then
      raise condition.timeout end
    end
    condition.timedWait(m  cond  timeout)
  end
mutex.unlock(m)
```

Die angeführten Beispiele zeigen die korrekte Verwendung des Condition-Mechanismus, da, wie in Abschnitt 3.5.2 bereits angemerkt, die Erfüllung einer Bedingung nach Wiederaktivierung eines Threads durch die Funktion *signal* nicht gewährleistet ist. Beispielsweise kann jeder Thread, der das Mutex *lock* im obigen Programmfragment akquiriert, den durch das Mutex geschützten Wert ändern. Dieses macht ein erneutes Testen der mit einer Bedingungsvariablen verknüpften Bedingung notwendig.

4.3 Implementation im Tycoon-System

Zur Implementation atomarer Aktionen im Tycoon-System wird in Abschnitt 4.1.3.2 die Benutzung eines Mechanismus zur Verhinderung von Koroutinenwechsel vorgeschlagen. Um einen Ansatz für die Implementation eines derartigen Mechanismus zu finden, ist es notwendig, das im Tycoon-System realisierte Laufzeitsystem näher zu betrachten. Der Aufbau und die Arbeitsweise des Laufzeitsystems werden daher im folgenden Abschnitt erläutert. Daran anschließend werden die zur Sicherstellung des Basismechanismus notwendigen Eingriffe in das Laufzeitsystem beschrieben. Die Bereitstellung eines Timeout-Mechanismus für den im Modul *condition* vorhandenen Funktionsaufruf *timedWait* erfordert ebenfalls unterstützende Maßnahmen im Laufzeitsystem. Zur Modifikation des Laufzeitsystems ist die Benutzung der für die Implementation des Laufzeitsystems verwendeten Programmiersprache C notwendig. Die für die Erweiterung des Laufzeitsystems verwendeten Datenstrukturen müssen wie Objekte der Programmiersprache TL die Eigenschaft der Persistenz besitzen. Dies wird durch die im nächsten Abschnitt beschriebenen Maßnahmen erreicht. Zur Implementation der auf dem Basismechanismus aufbauenden Module *semaphore*, *mutex* und *condition* wird ausschließlich die Programmiersprache TL verwendet. Dieser Einsatz von TL sichert automatisch die Persistenz aller verwendeten Programmobjekte.

Der folgende Abschnitt gibt einen Überblick über das im Tycoon-System implementierte Laufzeitsystems.

4.3.1 Das Laufzeitsystem

Das Tycoon-Laufzeitsystem ist fast vollständig[1] in der standardisierten Programmiersprache Ansi-C geschrieben. Dieser Sachverhalt sowie der modulare Aufbau des Laufzeitsystems gewährleisten die Portierbarkeit auf weitere Betriebssysteme. Das Tycoon-Laufzeitsystem gliedert sich in die folgenden drei Komponenten [Mathiske 96]:

[1] Zur Realisierung eines Koroutinenwechsels für das Betriebssystem SunOS 4 war der Einsatz einer Maschinenanweisung zur Simulation der Funktion *setjump* erforderlich [Mathiske 96].

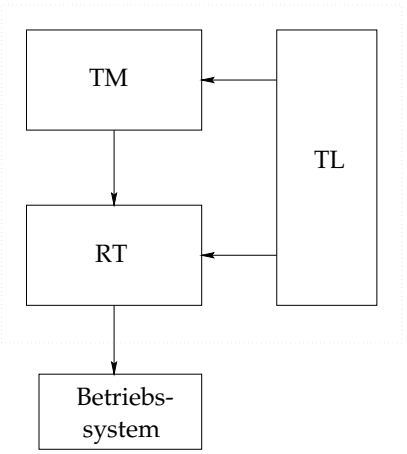

Abbildung 4.6: Schematischer Aufbau des Laufzeitsystems

RT ("RunTime"-Support): Mit dieser Komponente wird den weiteren Teilen des Laufzeitsystems eine genormte Schnittstelle für betriebssystemspezifische Operationen zur Verfügung gestellt. Diese Operationen umfassen unter anderem die Dateiverwaltung und sonstige Ein- und Ausgabefunktionen sowie Zugriffsfunktionen zur Speicherverwaltung des Betriebssystems. Weiterhin ist in dieser Komponente die gesamte Koroutinenverwaltung implementiert. Ein weiterer wichtiger Teil des *RT* ist die Unterkomponente *rtsession*. In diesem Modul werden flüchtige Variablen des Laufzeitsystems registriert, die eine Sitzung im Tycoonsystem überdauern sollen. Diese registrierten Variablen werden vor einer Stabilisierung des Objektspeichers in Objektspeicherstrukturen übertragen und sind somit persistent. Bei einem Neustart des Systems oder der Rücknahme einer Transaktion werden diese persistenten Variablen in die entsprechenden C-Strukturen zurückübertragen und stehen zur weiteren Benutzung durch das Laufzeitsystem zur Verfügung.

TM ("Tycoon Machine"): Die Tycoon Maschine stellt den Hauptteil des Laufzeitsystems dar. Hier ist der Bytecodeinterpreter, der den vom Tycoon-Übersetzer erzeugten Bytecode direkt abarbeitet, angesiedelt. Weitere Funktionen dieser Komponente sind die Transaktionsverwaltung sowie die Verwaltung persistenter Sicherungspunkte.

TL ("Tycoon Language"-Extensions): Dieser Teil des Laufzeitsystems enthält eine Reihe von Funktionen, die ihrerseits von verschiedenen Tycoon–Modulen benutzt werden. Es handelt sich hierbei um Hilfsfunktionen, deren Aufgaben auf der Ebene der Programmiersprache TL nicht oder nur mit großem Aufwand realisiert werden können. Hierunter sind Funktionen zur Threadsteuerung, typunsichere Operationen auf TL–Objekten, Hilfsfunktionen für C–Callbacks und Funktionen zur Symbolverwaltung für dynamisches Linken von TL-Objekten zu finden.

In Abbildung 4.6 sind die Abhängigkeiten der einzelnen Laufzeitsystemkomponenten dargestellt. Die Komponente *TL* benutzt Funktionen aus den Komponenten *TM* und *RT*. In der Komponente *TM* werden unter anderem die von *RT* bereitgestellten Funktionen zur

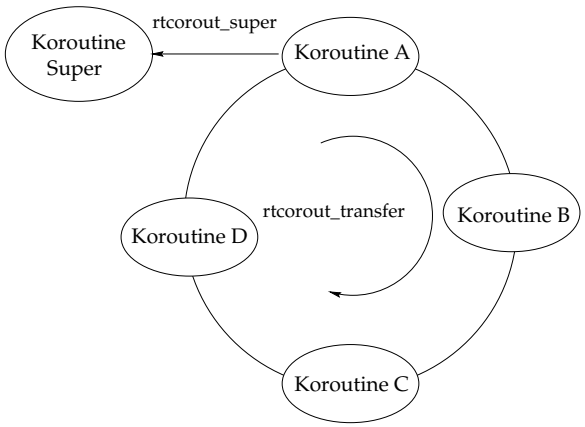

Abbildung 4.7: Anordnung der aktiven Koroutinen

Koroutinenumschaltung benutzt. Nur die Komponente *RT* greift auf Funktionen des unterliegenden Betriebssystems zurück.

Threads werden im Tycoon-System durch Koroutinen simuliert. Die grundlegende Verwaltung der Koroutinen ist in Abbbildung 4.7 illustriert. Die sich im Tycoon-System befindenden, aktiven Koroutinen sind in einer doppelt verketteten Ringstruktur angeordnet. Die Umschaltung zwischen diesen Koroutinen findet zyklisch und abhängig von einem Zeitgeber während der Abarbeitung des Bytecodes in der Komponente *TM* statt. Zu diesem Zweck wird durch eine Zeitgeberroutine die Tycoon-Systemvariable *rtcorout_fSwitch* gesetzt. An ausgesuchten Punkten des Tycoon-Bytecodeinterpreters wird diese Systemvariable ausgewertet. Abhängig vom ihrem Wert findet dann durch Aufruf der Transferfunktion *rtcorout_transfer* die Umschaltung zur nächsten Koroutine des Rings statt. Dieses nicht präemptive Verhalten stellt die Ununterbrechbarkeit der Tycoon–Bytecodebefehle durch die Auswahl geeigneter Umschaltpunkte sicher.

In einem Tycoon–System sind immer mindestens zwei Threads aktiv: Dies sind der sogenannte Superthread und der Hauptthread der Tycoon-Sitzung. Bestimmte Funktionen des Laufzeitsystems können nur durch den Superthread ausgeführt werden. Hierzu gehören alle Funktionen, die mit der Transaktionsverwaltung gekoppelt sind. Während der Stabilisierung des Objektspeichers können beispielsweise keine Threads außer dem angesprochenen Superthread aktiv sein, da der derzeitige Zustand dieser Threads im Objektspeicher durch die Stabilisierung persistent gemacht wird. Der Superthread ist der einzige nichtpersistente Thread im Tycoon-System. Bei jedem Start des Systems wird ein neuer Superthread erzeugt, der alle zuvor aktiven Threads wieder aktiviert.

Im einzelnen wird die Koroutinenverwaltung in der Komponente *RT* prinzipiell mittels folgender Funktionen verwirklicht. Weitere, die Koroutinenverwaltung betreffende Hilfsfunktionen werden detailliert in [Breilmann 95] beschrieben.

rtcorout_new: Es wird eine neue Koroutine unter Verwendung der C-Bibliotheksfunktion *setjmp* erzeugt [USO 90]. Zu diesem Zweck wird durch eine weitere Funktion des *RT* ein Speicherbereich für die Prozessorregister und den Stackbereich der Koroutine

bereitgestellt. Zu diesen Prozessorregistern zählt ein Zeiger, der die aktuelle Position des auszuführenden Maschinencodes angibt. Dieser Zeiger wird dann durch die Adresse der Funktion *tmexec_runGC*, in der der eigentliche Bytekodeinterpreter kodiert ist, ersetzt. Durch diese Vorgehensweise wird eine Stackumgebung erzeugt, die bei einer nachfolgenden Aktivierung der Koroutine zu der Ausführung einer weiteren Instanz des Bytekodeinterpreters führt.

rtcorout_transfer: Diese Funktion führt zu einer Umschaltung zu einer weiteren Koroutine. Mittels der C-Bibliotheksfunktion *setjmp* werden die für die Ausführung relevanten Prozessorregister der aktuellen Koroutine gesichert. Die Funktion *longjmp* restauriert die durch ihre Parameter gegebene Prozessorumgebung und setzt die weitere Programmausführung gemäß dieser Umgebung fort.

timerHandler: Diese Routine dient zum zyklischen Aktivieren der Koroutinenumschaltung.

Die Schnittstelle zur Abbildung der persistenten Strukturen von Tycoon-Threads auf Koroutinen wird durch die folgenden Funktionen gewährleistet:

tmthread_newGC: Es wird eine neue Objektidentifikation *(OID)* für einen neu anzulegenden, persistenten Thread vom Objektspeicher angefordert. Diese Anforderung wird durch den Aufruf der entsprechenden Funktionen des Speicherprotokolls TSP erfüllt. Weiterhin werden durch zusätzliche Aufrufe des TSP persistente Speicherbereiche für den Stackbereich des Threads erzeugt.

tmthread_runGC: Ein neu erzeugter oder bereits vorhandener Thread wird für die Ausführung durch die *TM* vorbereitet. Dies umfaßt die Erzeugung einer neuen Koroutine durch Aufruf der Funktion *rtcorout_new* sowie die Initialisierung der flüchtigen Strukturen des Koroutinenrings mit den aus der persistenten Threadstruktur entnommenen Daten für Stackbereich und *TM*-Maschinenregister.

tmthread_stopGC: Diese Routine wird bei einer Suspendierung oder Beendigung eines Threads abgearbeitet. Die aktuellen Werte der Register der *TM* werden wieder in die persistente Datenstruktur des Threads übertragen. Der der aktiven Koroutine zugeordnete, nichtpersistente Speicherbereich für die Ringstruktur der Koroutine wird abgebaut und an das Betriebssystem zurückgegeben.

Die Ausführungszustände und möglichen Zustandsübergänge der Tycoon-Threads sind aus Abbildung 4.8 ersichtlich. Übergänge zwischen den einzelnen Zuständen werden mittels der entsprechenden Funktionen aus dem Modul *thread* (siehe Anhang A.4) programmiert oder ergeben sich während der Ausführung eines Threads. Ein neu erzeugter Thread befindet sich im Zustand *suspended* (Funktion *thread.new*) oder im Zustand *running* (*thread.fork*). Beendete Threads befinden sich entweder im Zustand *terminated* oder, als Folge nicht abgefangener Ausnahmebedingungen, im Zustand *exception*. In den Zustand *blocking* gelangt ein Thread durch Aufruf der Funktion *thread.join*. Diese Funktion führt zur Blockierung des aufrufenden Threads, bis der als Parameter der Funktion *thread.join* angegebene Thread seine Ausführung beendet. Nur Threads, die sich im Zustand *running* befinden, sind als Elemente von Datenstrukturen des Laufzeitsystems erreichbar. Alle Threads, die sich in

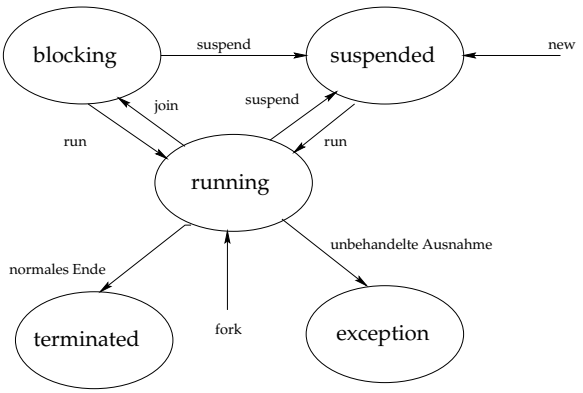

Abbildung 4.8: Zustandsübergänge der Tycoon-Threads

einem der anderen Zustände befinden, sind nur über die *OID*, die eine Referenz zur persistenten Datenstruktur der Threads im Objektspeicher darstellt, erreichbar. Es existiert keine Struktur, in der nicht aktive Threads verzeichnet sind.

4.3.2 Interner Aufbau der Synchronisationsmechanismen

Zur Sicherstellung der atomaren Ausführung einer Befehlssequenz wird in den internen Mechanismus des Laufzeitsystems zur Koroutinenumschaltung eingegriffen. Zur Implementation eines Timeoutmechanismus für die Funktion *timedWait* des Moduls *Condition* sind ebenfalls Erweiterungen des Laufzeitsystems erforderlich. Alle weiteren Mechanismen zur Synchronisation sind vollständig in der Sprache TL verfaßt. In den folgenden drei Unterabschnitten sind Details der jeweiligen Implementation beschrieben.

4.3.2.1 Atomare Aktionen

Um eine Umschaltung zur nächsten Koroutine nach Ablauf des vorgegebenen Zeitintervalls zu verhindern, wurde eine weitere Variable des C-Laufzeitsystems, *tmthread_nCritical*, eingeführt. Zur Einleitung und Beendigung eines ununterbrechbaren Abschnitts während der Ausführung eines Threads sind zwei Funktionen des Laufzeitsystems vorgesehen:

tlthread_beginCritical: Der Wert der Variablen *tmthread_nCritical* wird inkrementiert.

tlthread_endCritical: Der Wert der Variablen *tmthread_nCritical* wird dekrementiert.

Der Wert von *tmthread_nCritical* wird in den im Bytecodeinterpreter vorhandenen Umschaltpunkten für Koroutinen ausgewertet. Liegt ein Wert ungleich Null vor, findet keine Umschaltung zur nächsten Koroutine statt. Durch Inkrementieren und Dekrementieren der Variablen *tmthread_nCritical* sind geschachtelte, atomare Aktionen eines Threads möglich. Die Anwesenheit eines Threads in einem ununterbrechbaren Abschnitt muß dauerhaft festgehalten werden, da während der Ausführung alle Operationen des Systems ausgeführt werden können. Dazu gehören auch alle Operationen zur Stabilisierung des Objektspeichers.

Diese Forderung wird durch Registrierung von *tmthread_nCritical* als Sitzungsvariable, die unmittelbar vor der Objektspeicherstabilisierung in eine persistente Struktur übertragen wird, erfüllt.

Die Schnittstelle zur Benutzung der Funktionen *tlthread_beginCritical* und *tlthread_endCritical* ist im Modul *thread* als Funktionsaufruf *thread.atomic* implementiert. Das folgende Beispiel zeigt die Benutzung dieser Funktion:

```
let var shared = 0

(* Funktion, die atomar auszuführen ist *)
let incAtomic(var x :Int) =
  begin
    x := x + 1
    x
  end

(* Atomares Inkrementieren eines Werts *)
let result = thread.atomic(fun() incAtomic(shared))
```

Die Funktion *thread.atomic* erwartet als Parameter eine parameterlose Funktion. Diese Funktion wird durch den oben beschriebenen Mechanismus ununterbrechbar ausgeführt. Um beliebige Funktionen mittels *thread.atomic* ausführen zu können, bietet es sich an, wie im obigen Beispiel auf Funktionen höherer Ordnung zurückzugreifen.

Die zur Sicherstellung atomarer Abschnitte verwendete Funktion *thread.atomic* wird wie folgt implementiert:

```
let atomic(R <:Ok  action() :R) :R =
  begin
    beginCritical()
    try
      let result = action()
      endCritical()
      result
    else
      endCritical()
      reraise
    end
  end
```

Die Funktionen *beginCritical* sowie *endCritical* beinhalten lediglich direkte Aufrufe des Laufzeitsystems zum Inkrementieren und Dekrementieren der Variablen *tmthread_nCritical*. Die als Argument übergebene Funktion *action* wird dann im Schutz dieser Aufrufe atomar ausgewertet. Zu beachten ist darüber hinaus die Propagierung eventuell auftretender Ausnahmebedingungen bei der Ausführung der Funktion *action* durch Klammerung der Auswertungssequenz mit *try* und *reraise*.

Die vorliegende Implementation von *thread.atomic* erfordert weitere Vorkehrungen im Laufzeitsystem. Diese beinhalten unter anderem die Aufhebung der Umschaltsperren bei zeitweiser oder endgültiger Beendigung eines Threads, hervorgerufen durch Ausnahmebedingungen oder expliziten Aufruf der Funktionen *suspend* und *kill*.

4.3.2.2 Timeouts

Um die Implementation von zeitabhängigen Wartefunktionen zur Threadsteuerung zu ermöglichen, ist die Erweiterung des Laufzeitsystems um einen Timeoutmechanismus erforderlich. Ein derartiger Mechanismus wird unter anderem für die im Modul *condition* enthaltene Funktion *timedWait* benötigt. Die genannte Funktionalität wird durch Programmierung einer zeitgesteuerten Warteschlangenverwaltung auf der Ebene *RT* des Laufzeitsystems ermöglicht.

Threads, die für ein gegebenes Zeitintervall verzögert werden sollen, werden in einer speziellen Warteschlange eingetragen. Die Ordnung in der Warteschlange ist durch Registrierung der zeitlichen Differenz zum nächsten Warteschlangeneintrag gewährleistet. Ist beispielsweise die Verzögerung eines Threads *T1* um $7.4sec$ und die eines weiteren Threads *T2* um $9.4sec$ notwendig, so werden für diese Threads in der Warteschlange die Zeiten $\Delta T1 = 7.4sec$ und $\Delta T2 = 2.0sec$ als Zeitintervalle bis zur Aktivierung festgehalten. Durch diese Anordnung ist für die Aktivierung lediglich die Betrachtung des ersten Warteschlangeneintrags notwendig. Einträge werden bei Vorhandensein aktiver Threads in der Zeitgeberroutine zur Koroutinenumschaltung überprüft. Im Fall des Ablaufs des Zeitintervalls des ersten Warteschlangeneintrags wird der Thread durch die Funktion *thread.run* des *RT* erneut gestartet. Bei Abwesenheit aktiver Threads wird mittels eines betriebssysteminternen, zeitabhängigen Unterbrechungsmechanismus die exakte Wartezeit bis zur notwendigen Aktivierung des ersten Warteschlangeneintrags programmiert. Diese Vorgehensweise ist notwendig, um eine Portierbarkeit dieses Mechanismus zu gewährleisten, da in vielen Betriebssystemen nur eine begrenzte Anzahl von Zeitgebern je Betriebssystemprozeß verfügbar ist.

Die Einführung zeitabhängiger Warteschlangen führt zum Auftreten asynchroner Vorgänge im Laufzeitsystem. Das bei Ablauf einer Wartezeit notwendige, nebenläufige Einfügen von Threads in den Ring der aktiven Threads wird daher durch Sperrmechanismen während des Zugriffs auf interne Strukturen des Laufzeitsystems synchronisiert.

Die Implementation des obigen Mechanismus erfolgt in der Programmiersprache ANSI-C. Um die Persistenz der dabei verwendeten Datenstrukturen zu erreichen, wird die Zeitgeberwarteschlange analog zu dem in Abschnitt 4.3.1 genannten Verfahren vor der Objektspeicherstabilisierung in persistente Datenstrukturen übertragen. Da in der oben genannten Warteschlange durch Speicherrückgewinnung änderbare Objektidentifikatoren in flüchtigen C-Datenstrukturen gespeichert sind, ist die Implementation einer Enumeratorfunktion notwendig [Matthes et al. 95].

4.3.2.3 Die Module zur Synchronisation

Durch den im Abschnitt 4.3.2.1 beschriebenen Mechanismus *atomic* zur Verwirklichung ununterbrechbarer Anweisungsfolgen ist es möglich, die Synchronisationsmechanismen für Semaphore, Mutexe und Bedingungsvariablen vollständig in der Sprache TL zu verfassen.

Dadurch besitzen alle vorgestellten Synchronisationsmechanismen automatisch die Eigenschaft der Persistenz. Die Implementation ist bei allen Mechanismen prinzipiell ähnlich aufgebaut. Ein Element des Typs *sync.T*, wobei hier *sync* als Synonym für alle angesprochenen Mechanismen steht, ist als TL-Tupel der Form

```
Let T <:Ok = Tuple
    ...                                  (* Zähler für Statistiken *)
    ...                                  (* Semaphorzähler *)
    var owner     :thread.T(Ok)          (* 'Halter' des Elements *)
    waitqueue     :queue.T(thread.T(Ok)) (* Liste der wartenden Threads *)
end
```

aufgebaut. Für jedes Element des Typs *sync.T* wird im Falle der Akquisition der Halter der Synchronisationsvariable in der veränderlichen Variable *owner* festgehalten. Diese Variable wird derzeit bei Freigabeoperationen nur für Synchronisationsvariablen des Typs *mutex.T* ausgewertet, um die Berechtigung für diese Operation festzustellen. In der Variablen *waitqueue* ist die Liste aller auf die Synchronisationsvariable wartenden Threads festgehalten. Alle Modifikationen, die an einem Element des Typs *sync.T* vorgenommen werden, sind durch die Operation *thread.atomic* geschützt und werden atomar durchgeführt. Als Beispiele sei hier der prinzipielle Ablauf der Operationen des Wartens und Freigebens eines Semaphors angeführt. Die Funktionen *semaphore.wait* sowie *semaphore.post* können unter dem Schutz der Operation *thread.atomic* wie folgt implementiert werden:

```
(* Warten auf ein Semaphor *)
let wait(sem :T) =
 begin
  let self = thread.self()
  thread.atomic(fun() begin
  sem.count := sem.count - 1
   if sem.count < 0 then
    queue.push(sem.waitqueue self)
      thread.suspend(self)
   end
  end)
end
```

```
(* Freigabe eines Semaphors *)
let post(sem :T) :Ok =
  begin
   thread.atomic(fun() begin
    sem.count := sem.count + 1
    if sem.count ≤ 0 then
     thread.run(queue.pop(sem.waitqueue))
    end
   end)
  end
```

Die in der Funktion *wait* erforderliche Freigabe der durch *thread.atomic* erzwungenen Umschaltsperre für Koroutinen ist durch die Implementation der Funktion *thread.suspend* sichergestellt (Abschnitt 4.3.2.1). Es ist zu beachten, daß der in der Funktion *post* vorhandene Aufruf *thread.run* zu einem Aufbau der für das Laufzeitsystem erforderlichen, flüchtigen Datenstruktur sowie des zu aktivierenden Threads führt. Weiterhin wird der zu startende Thread in die Ringstruktur der aktiven Threads eingefügt. Die Aktivierung des Threads findet nach Beendigung der durch *thread.atomic* bedingten Umschaltsperre im Rahmen der normalen Rechenzeitzuteilung statt.

4.3.3 Tycoon-Threads und Standards

Für die Programmierung nebenläufiger Aktivitäten existieren eine Reihe weitverbreiteter Programmierschnittstellen. Die für das Tycoon-System implementierten Synchronisationsmechanismen wurden auch in Hinblick auf derartige Standards entworfen. Im folgenden wird die Einordnung der Funktionen zur Threadsteuerung und Synchronisation in diese Standards beschrieben.

Als Ausgangspunkt werden die für das Unix-Betriebssystem Solaris spezifizierte Threadbibliothek [Sun 93] sowie die Spezifikation nach Posix für eine standardisierte Multithreadschnittstelle herangezogen [POSIX 90]. Der Standard nach Posix deckt die folgenden Punkte zur Programmierung von Threads ab:

Threadeigenschaften: Hierunter fällt eine Reihe von Funktionen zum Setzen und Abfragen threadspezifischer Parameter wie Stackgröße und Prioritätssteuerung. Weiterhin sind Methoden des Threadschedulings wie Zeitscheibensteuerung und präemptives Scheduling einstellbar.

Threadsteuerung: Hierunter fallen Funktionen zur Erzeugung und Beseitigung von Threads, Funktionen zur Kontrollübermittlung wie *join* und *exit* sowie Aufrufe zum Vergleich von Threads.

Eigenschaften der Synchronisationsmechanismen: Mit den Funktionen dieser Gruppe sind Prioritätsparameter (Übernahme der Prioritätsparameter der die Synchronisationsmechanismen benutzenden Threads) und die Speicherabbildung von Synchronisationsmechanismen (zum Prozeß lokaler Speicher oder von mehreren Prozessen gemeinsam benutzter Speicher) steuerbar.

Benutzung der Synchronisationsmechanismen: In dieser Gruppe sind Funktionen zum Zugriff auf Synchronisationsmechanismen wie Mutexe und Bedingungsvariablen spezifiziert.

Der verabschiedete Posix-Standard beinhaltet eine minimale, aber vollständige Menge von Synchronisations- und Threadsteuerungsmechanismen. Als Mechanismen zur Synchronisation von Threads sind lediglich Mutexe und Bedingungsvariablen spezifiziert. Die Mechanismen zur Threadsteuerung bilden die in vielen Betriebssystemen vorhandenen Steuerungsmechanismen für die Rechenzeiteinteilung von Threads und Systemprozessen ab. Ein Beispiel hierfür ist die Möglichkeit, Prozesse und Threads als Echtzeitprozesse mit garantiertem Rechenzeitkontingent und berechenbaren Antwortzeiten bei der Reaktion auf Ereignisse ablaufen zu lassen [Deitel 90].

Die für die genannten Unix-Betriebssysteme spezifizierten Mechanismen sind eng an den Posixstandard angelehnt. Darüber hinaus werden zusätzliche Synchronisationsmechanismen wie Semaphore und Leser- und Schreibersperren angeboten.

Grundsätzlich muß für eine Implementation von Funktionen des genannten Standards bereits eine Entsprechung von Funktionalitäten im Zielsystem vorhanden sein. Die in dieser Arbeit realisierten Synchronisationsmechanismen stellen zusammen mit den vorhandenen Aufrufen zur Threadsteuerung eine im Rahmen des Tycoonsystems sinnvolle Untermenge des Posixstandards dar. Nicht vom Posixstandard übernommene Funktionalitäten wie

Prioritätssteuerung und Interprozeßkommunikation sind im Tycoon-System nicht vorhanden. Eine Übersicht über äquivalente Funktionen des Posixstandards Posix.1c/D10 [POSIX 90] und den Tycoonmodulen *thread*, *mutex* und *condition* gibt die folgende Tabelle:

Posix-Threads	Tycoon-Threads	Bemerkungen
Funktionen zur Threadsteuerung		
pthread_create	thread.fork	auch thread.new
pthread_self	thread.self	
pthread_equal	pid1 == pid2	Vergleich mittels Vergleichsfunktionen der Sprache TL bei Threads mit gleicher Signatur
pthread_exit	thread.kill	
pthread_join	thread.join	
Funktionen zur Mutexsteuerung		
pthread_mutex_init	mutex.new	
pthread_mutex_destroy	–	nicht notwendig[2]
pthread_mutex_lock	mutex.lock	
pthread_mutex_trylock	mutex.tryLock	
Funktionen für Bedingungsvariablen		
pthread_cond_init	condition.new	
pthread_cond_destroy	–	nicht notwendig[2]
pthread_cond_signal	condition.signal	
pthread_cond_broadcast	condition.broadcast	
pthread_cond_wait	condition.wait	
pthread_cond_timedwait	condition.timedWait	

[2]Nicht mehr referenzierbare Objekte werden durch Speicherrückgewinnung entfernt.

4.4 Realisierung weiterer Synchronisationsmechanismen

In den vorangegangenen Abschnitten werden der Aufbau und die Implementation der Module *semaphore* und *mutex* sowie die zur Realisierung von Monitoren benutzten Funktionen im Modul *condition* beschrieben. In diesem Abschnitt wird unter Benutzung der in den genannten Modulen verwirklichten Synchronisationsprimitive eine Reihe weiterer, in der Hierarchie der Synchronisationsmethoden höher angesiedelten Synchronisationsverfahren beschrieben (vgl. Abbildung 4.3). Die in den folgenden Beispielen implementierten Verfahren dienen der Synchronisation durch die Übertragung von Nachrichten über Kommunikationskanäle ohne Benutzung weiterer gemeinsamer Variablen.

Es sind verschiedene Möglichkeiten der Synchronisation durch die Übertragung von Nachrichten denkbar. Folgende Unterschiede sind im Verhalten beim Senden und Empfangen der Nachrichten möglich:

Blockierendes Senden und Empfangen: Sowohl Sender als auch Empfänger werden blockiert, bis die Nachricht übertragen wurde. Dieses entspricht den in Abschnitt 3.5.3 besprochenen Eigenschaften der Kommunikationskanäle von CSP und der Rendezvous. Der Empfang und das Senden einer Nachricht sind eng gekoppelt.

Nichtblockierendes Senden, blockierendes Empfangen: Der sendende Prozeß arbeitet sofort weiter, und gesendete Nachrichten werden bis zum Lesen der Nachricht zwischengespeichert. Ein Empfänger wird demgegenüber blockiert, bis eine Nachricht eingetroffen ist. Diese Methode der Implementation erlaubt es einem sendenden Prozeß, sehr schnell eine Anzahl von Empfangsprozessen mit Nachrichten zu versorgen. Es besteht jedoch die Gefahr der übermäßigen Beschlagnahme von Systemressourcen, falls die Aufnahmekapazität eines Kommunikationskanals nicht beschränkt wird.

Nichtblockierendes Senden und Empfangen: Empfangene und sendende Prozesse werden in keinem Fall blockiert. Die Methode weist den Nachteil des aktiven Wartens auf der Seite des Empfängers auf.

In den folgenden Abschnitten werden beispielhaft Kommunikationsmechanismen für zwei der angeführten Möglichkeiten implementiert: In Abschnitt 4.4.1 wird die Implementation von Funktionen zur direkten Kommunikation von Threads über Kommunikationskanäle zur Übermittlung von Nachrichten beschrieben (nichtblockierendes Senden, blockierendes Empfangen), daran anschließend die Implementation eines den Ada-Rendezvous ähnlichen Mechanismus (Abschnitt 4.4.2). Der Einsatz von Syntaxerweiterungen (Abschnitt 4.4.3) erlaubt eine sichere und einfachere Benutzung dieser Synchronisationsmechanismen.

4.4.1 Kommunikationskanäle

Kommunikationskanäle dienen dem Austausch von Nachrichten zwischen Threads, ohne daß eine Benutzung expliziter Synchronisationsverfahren, wie Semaphore oder Mutexe, notwendig wird. Die für eine derartige Kommunikation notwendigen Funktionen sind im Modul *event* realisiert. Der Nachrichtenaustausch über Funktionen des Moduls *event* erlaubt ein

nichtblockierendes Senden von Nachrichten. Falls keine Nachrichten vorliegen, führt demgegenüber der Versuch, eine Nachricht zu empfangen, zur Blockierung eines empfangsbereiten Threads.

Der Nachrichtenaustausch erfolgt über typisierte Kommunikationsendpunkte. Vom Modul *event* werden der parametrisierte, abstrakte Datentyp *event.T(R)* sowie die auf diesem Datentyp definierten Funktionen exportiert. Ein Element des Typs *event.T(R)* wird mittels der Operation

*new(R <:***Ok***) :T(R)*

erzeugt. Diese Funktion erwartet als Parameter den Typ R der zu übermittelnden Nachrichten. Eine Nachricht des Typs R wird mittels der Funktion

*put(R <:***Ok** *ev :T(R) value :R) :***Ok**

übertragen. Die Parameter dieser Funktionen bestehen wiederum aus dem Typ der Nachricht sowie aus der Angabe der Ereignisvariablen *ev* und dem Wert des zu übertragenden Ereignisses *value*. Der Empfang von Nachrichten erfolgt mit der Funktion

*get(R <:***Ok** *ev :T(R)) :R.*

Falls keine ungelesenen Nachrichten vorliegen, wird ein diese Funktion aufrufender Thread bis zum Eintreffen einer Nachricht blockiert.

Das gleichzeitige Warten auf mehrere Nachrichten erfolgt unter Verwendung der Funktion

*select(alt :Array(SelectT)) :***Ok***.*

Die Benutzung dieser Funktion erfordert die Übergabe eines Feldes von Elementen des Typs *event.SelectT*. Die direkte Übergabe mehrerer Elemente des Typs *event.T(R)* ist nicht möglich, da alle Elemente eines Feldes vom gleichen Typ sein müssen. Um dennoch die Verwendung von Ereignisvariablen unterschiedlichen Typs in der Funktionen *select* zu ermöglichen, werden die Typen dieser Elemente unter Verwendung der Operation

*makeEntry(R<:***Ok** *ev :T(R) func :***Fun***(:R) :***Ok***) :SelectT*

verborgen. Die Funktion *makeEntry* erwartet die Angabe einer Ereignisvariablen *ev* sowie die Angabe einer Funktion *func*. Die Funktion *func* dient der eigentlichen Übermittlung des Werts der eingetroffenen Nachricht. Sie wird während der Nachrichtenübermittlung mit dem Wert der Nachricht aufgerufen. Diese Vorgehensweise ist notwendig, da nach Verbergen des Nachrichtentyps durch die Funktion *makeEntry* der Typ der Nachricht nicht mehr unmittelbar zur Verfügung steht.

Die Implementation des Moduls *event* erfolgt mittels der im Modul *semaphore* vorhandenen Funktionen. Jedem Element des Typs *event.T(R)* sind zwei Semaphore sowie eine Warteschlange zur Aufnahme der Werte zugeordnet. Ein binäres Semaphor *lock*, das beim Erzeugen einer Ereignisvariable mit dem Wert 1 initialisiert wird, dient dem gegenseitigen Ausschluß von Prozessen beim Zugriff auf die in der Ereignisvariablen vorhandenen

internen Strukturen. Werte werden beim Aufruf der Funktion *event.put* in der internen Warteschlange gespeichert. Das Vorhandensein eines Werts wird mittels eines weiteren, im Typ *event.T(R)* verwendeten, allgemeinen Semaphors *isval* angezeigt. Dieses Semaphor wird beim Erzeugen einer Ereignisvariablen mit dem Wert 0 initialisiert und nach dem Schreiben in die Warteschlange durch Aufruf der *semaphore.signal*–Operation inkrementiert. Ein Aufruf der Funktion *event.get* führt beim Nichtvorhandensein von Werten zu einer Blockierung lesender Threads durch Ausführung der *semaphore.wait*–Operation auf dem Semaphor *isval*. Die Funktion *select* wird mittels der Funktion *semaphore.waitMultipleOr* implementiert, die ein gleichzeitiges Warten auf mehrere Semaphore ermöglicht. Das folgende Beispiel zeigt die Erzeugung zweier Kommunikationsendpunkte und deren Benutzung:

```
Let Person = Tuple name :String  age :Int end
let e1 = event.new(:Person)
let e2 = event.new(:Ok)

let funcA(self :thread.T(Ok)) =
  begin
    let processPerson(x :Person) = begin ... end
    let entry1 = event.makeEntry(e1  processPerson)
    let entry2 = event.makeEntry(e2  fun(x :Ok) ok)

    ...
    loop
      ...
      event.select(array entry1  entry2 end)
      ...
    end
  end

let funcB(self :thread.T(Ok)) =
  begin
    loop
      ...
      let value = tuple ... end
      event.put(e1 value)
      ...
      event.put(e2 ok)
    end
  end

let threadA = thread.fork(funcA)  and  threadB = thread.fork(funcB)
```

In diesem Beispiel kommunizieren *threadA* sowie *threadB* über den Austausch von Nachrichten der Typen *Person* und **Ok**. Der Austausch von Nachrichten des Typs **Ok** dient lediglich der Signalisierung eines Ereignisses, der tatsächliche Wert der Nachricht, hier immer der konstante Wert **ok**, wird ignoriert.

4.4.2 Rendezvous

Im Modul *rendezvous* sind Kommunikationsmechanismen für blockierendes Senden und Empfangen von Nachrichten implementiert: Der Versuch, eine Nachricht zu senden, führt zur Blockierung des Senders, bis ein Empfänger versucht, eine Nachricht zu empfangen. Der Versuch, eine Nachricht zu empfangen, blockiert den Empfänger, bis ein Sender eine Nachrichtenübermittlung anstößt.

Die Implementation von Rendezvous ähnelt der des im vorigen Abschnitt eingeführten Moduls *event*. Ein Rendezvous entspricht einem Kommunikationkanal mit blockierendem Senden und Empfangen. Im Gegensatz zu den in CSP und Ada verwirklichten Rendezvous werden Prozesse nicht direkt, sondern indirekt über Rendezvousvariablen adressiert.

Das Modul *rendezvous* besteht aus den folgenden Funktionen: Eine neue Rendezvousvariable wird durch die Funktion

*new(R <:***Ok***) :T(R)*

erzeugt, die als Parameter den Typ R der zu übermittelnden Nachrichten erwartet.

Ein Rendezvous wird mittels der Funktionen

*put(R <:***Ok** *rend :T(R) value :R) :***Ok**

und

*get(R <:***Ok** *rend :T(R)) :R*

eingeleitet. Hierbei wird der in der Funktion *put* angegebene Wert *value* einem die Funktion *get* aufrufenden Empfänger übermittelt. Der Austausch einer Nachricht findet erst statt, wenn sowohl Empfänger als auch Sender für einen Nachrichtenaustausch über die als Argument für *put* und *get* angegebene Rendezvousvariable *rend* bereit sind.

Für das Warten auf mehrere Alternativen für ein Rendezvous steht die Funktion *select* zur Verfügung. Für die Benutzung dieser Funktion ist es, analog zur Funktion *event.select*, notwendig, die Typen der zu benutzenden Rendezvous-Elemente durch die Funktion

*makeEntry(R<:***Ok** *rend :T(R) func :***Fun***(:R) :***Ok** *guard :***Fun***() :Bool) :SelectT*

zu verbergen. Die Argumente der Funktion *select* bestehen aus einer Variable des Typs *rendezvous.T(R)* sowie einer Funktion *func*, die als Argument den empfangenen Wert erhält. Weiterhin ist die Angabe einer Funktion *guard* notwendig, die der Auswahl möglicher Alternativen für ein Rendezvous über den Aufruf von *select* dient.

Die Funktion *select* erwartet als Argumente ein Feld von Elementen des Typs *rendezvous.SelectT* sowie die Angabe einer (optionalen) Funktion *elseCase*[3]:

*select(alt :Array(SelectT) elseCase :optional.T(***Fun***():***Ok***)) :***Ok***.*

[3]Optionale Werte können durch Funktionen des Moduls *optional* realisiert werden [Matthes 93].

Die Implementation der Funktion *select* lehnt sich an das gleichnamige Konstrukt der Programmiersprache Ada an [Burns et al. 87]. Jedoch verfügt das in Ada vorhandene Konstrukt noch über weitergehende Möglichkeiten, wie beispielsweise die Angabe von Zeitschranken, innerhalb derer ein Rendezvous eingeleitet werden muß.

Die der Funktion *select* übergebenen Argumente werden wie folgt ausgewertet:

- Zuerst werden alle Wächteranweisungen der im Feld *alt* übergebenen Rendezvousvariablen ausgewertet. Rendezvousvariablen, deren Wächter den boolschen Wert *true* ergeben, werden als offen registriert. Damit stehen sie für ein mögliches Rendezvous zur Verfügung. Wächter, deren Auswertung *false* ergibt, werden als geschlossen bezeichnet und für ein Rendezvous nicht berücksichtigt.
- Existieren keine offenen Wächter und existiert keine Funktion *elseCase*, so wird der Aufruf von *select* mit einer Ausnahme abgebrochen.
- Aus der Menge der Alternativen mit offenen Wächtern werden diejenigen ausgewählt, für die bereits ein Kommunikationspartner für ein Rendezvous vorhanden ist. Aus diesen zur Verfügung stehenden Rendezvouspartnern wird ein Partner ausgewählt und das Rendezvous vollzogen. Die Auswahl eines Rendezvouspartners wird über einen Zufallszahlengenerator gesteuert.
- Existieren keine Kommunikationspartner für ein Rendezvous und wurde eine Funktion *elseCase* übergeben, so wird diese ausgeführt.
- Existieren keine offenen Wächter und wurde eine Funktion *elseCase* übergeben, so wird diese ausgeführt.
- Existieren keine potentiellen Rendezvouspartner und wurde keine Funktion *elseCase* übergeben, wird der die Funktion *select* aufrufende Thread bis zum Eintreffen eines Rendezvouspartners blockiert.

Das folgende Beispiel zeigt die Implementation eines einfachen Puffers für Datenelemente des Typs *Int*. Der Puffer besteht aus einem eigenständigen Thread, der die Entnahme und das Hinzufügen von Datenelementen über Rendezvous gestattet. In der Funktion *select* wird auf Rendezvous für das Hinzufügen von Datenelementen *(append)* sowie für die Entnahme von Datenelementen *(take)* gewartet. Als Erzeuger und Verbraucher von Pufferelementen fungieren zwei weitere Threads *producer* und *consumer*, die über die Funktionen *put* und *get* Rendezvous mit dem Puffer-Thread einleiten:

```
(* Erzeugen der Rendezvous-Datenstrukturen *)
let append = rendezvous.new(:Int)
let take   = rendezvous.new(:Tuple var x :Int end)

(* Definition des Datenpuffers *)
let buffer(size :Int) = begin
   let var b = arrayOp.new(size 0)   (* Pufferarray *)
   let var inptr  = 0                (* Index für nächsten Schreibzugriff *)
   let var outptr = 0                (* Index für nächsten Lesezugriff *)
   let var n = 0                     (* Anzahl der gültigen Elemente *)
```

```
(* Anfügen eines Elementes *)
 let appendFun(x :Int) =
  begin
   b[inptr] := x
   n := n + 1
   inptr := {inptr + 1} % size
  end
```

```
(* Entfernen eines Elementes *)
 let takeFun(x :Tuple var x :Int end) =
  begin
   x.x := b[outptr]
   n := n - 1
   outptr:= {outptr + 1} % size
  end
```

```
 (* Einträge für den select-Aufruf erstellen *)
 let append = rendezvous.makeEntry(append  appendFun  fun() n < size)
 let take = rendezvous.makeEntry(take  takeFun  fun() n > 0 )
 let elseCase = optional.nil(:Fun():Ok)

 (* Akzeptieren von Rendezvous *)
 loop
  rendezvous.select(array append take end  elseCase)
 end
end (* buffer *)
```

```
(* Rendezvousaufrufe *)
let producer(self :thread.T(Ok)) =
 begin
  loop
   let val = ...
   rendezvous.put(append val)
  end
 end
```

```
let consumer(self :thread.T(Ok)) =
 begin
  let val = tuple let var x = 0 end
  loop
   rendezvous.put(take val)
   ...
  end
 end
```

```
let bufferThread = thread.fork(fun(self :thread.T(Ok)) buffer(100))
let consumerThread = thread.fork(consumer)
let producerThread = thread.fork(producer)
```

4.4.3 Syntaxerweiterungen

Die Syntax der Programmiersprache TL ist erweiterbar. Hierbei ist es möglich, schon bestehende Syntaxregeln um neue Alternativen zu erweitern und neue Regeln zur Sprache hinzuzufügen [Schröder 94]. Eine Erweiterung der Sprache TL ist dabei dynamisch, das heißt zur Laufzeit des Tycoon-Systems möglich. Durch Angabe eines speziellen Konstruktors **grammar** und der Aufzählung zusätzlicher Regeln wird der Parser des Systems dynamisch erweitert.

Der Mechanismus zur Definition von Syntaxerweiterungen wird in diesem Abschnitt zur Erweiterung von Synchronisationsmechanismen benutzt. Die beispielhaft eingeführten Erweiterungen dienen der einfacheren und sicheren Benutzung von Synchronisationsmechanismen.

Als erstes Beispiel für eine Syntaxerweiterung dient der in Abschnitt 3.5.2 vorgestellte Mechanismus der Sprache Modula-3 für den Zugriff auf ein Mutex. Durch die Klammerung eines kritischen Abschnitts durch die Schlüsselwörter *LOCK(mutex)* und *END* können in Modula-3 durch ein Mutex geschützte Programmabschnitte definiert werden. Der Vorteil dieser Notation liegt in der durch den Übersetzter sichergestellten Freigabe des Mutex am Blockende. Die Definition der entsprechenden Syntaxerweiterung für die Programmiersprache TL lautet wie folgt:

```
grammar
  value3:Value | ==
      p="lock" v=value "do" b=bnds "end" =>
      value<< |
        begin
          mutex.lock(v)
          try
            let result = begin b end
            mutex.unlock(v)
            result
          else
            mutex.unlock(v)
            reraise
          end
        end | >>
end
```

Es wird ein neues Schlüsselwort **lock** eingeführt. Der auf **lock** folgende Wert wird als Argument für die Funktionen *mutex.lock* und *mutex.unlock* benutzt. Während der Typüberprüfungsphase des Übersetzungsvorgangs wird die Kompatibilität des **lock** folgenden Werts mit den Signaturen der Zugriffsfunktionen *mutex.lock* und *mutex.unlock* überprüft. Die dem Schlüsselwort **do** folgenden Programmanweisungen werden nach Akquirierung des Mutex durchgeführt. Anschließend erfolgt die Freigabe des Mutex. Ausnahmebedingungen, die während der Auswertung des geschützten Abschnitts auftreten, werden abgefangen und nach Freigabe des Mutex an den aufrufenden Thread weitergegeben.

Das neue Schlüsselwort **lock** kann dann, wie im nächsten Beispiel gezeigt, für den Zugriff auf ein Mutex benutzt werden.

```
let m = mutex.new()

lock m do
 (* kritischer Abschnitt *)
end
```

Ein weiteres Beispiel für eine Syntaxerweiterung dient der einfachen Benutzung der Funktion *rendezvous.select*. Die in Anhang B.1 angegebene Syntaxerweiterung bietet gegenüber der direkten Benutzung der Funktion *rendezvous.select* die folgenden Vorteile:

- Durch die unmittelbare Angabe der im Falle eines Rendezvous auszuführenden Programmteile steigt die Übersichtlichkeit des Programms.

- Die Funktion *rendezvous.select* erwartet als Argumente ein Feld, dessen Elemente die Alternativen für ein Rendezvous angeben. Elemente dieses Feldes sind durch Aufruf der Funktion *rendezvous.makeEntry* zu bilden. Durch die Syntaxerweiterung wird dieser Vorgang automatisiert.
- Die Angabe einer Wächteranweisung ist optional. Wird keine Wächteranweisung angegeben, ist die entsprechende Alternative immer offen.

Das folgende Beispiel illustriert die Verwendung des neuen Select-Konstrukts. Das Programmbeispiel entspricht dem in Abschnitt 4.4.2 angegebenen Puffer für ganze Zahlen:

```
(* Erzeugen der Rendezvous-Datenstrukturen *)
let append = rendezvous.new(:Int)
let take   = rendezvous.new(:Tuple var x :Int end)

(* Definition des Datenpuffers *)
let buffer(size :Int) = begin
   let var b = arrayOp.new(size 0)    (* Pufferarray *)
   let var inptr  = 0                 (* Index für nächsten Schreibzugriff *)
   let var outptr = 0                 (* Index für nächsten Lesezugriff *)
   let var n = 0                      (* Anzahl der gültigen Elemente *)

  loop
    select
    when n < size ⇒
      accept append(x :Int) do
        (* – Anfügen eines Elementes *)
        b[inptr] := x
        n := n + 1
        inptr := {inptr + 1} % size
      end
    or when n > 0 ⇒
      accept take(x :Tuple var x :Int end) do
       (* – Entfernen eines Elementes *)
        x.x := b[outptr]
        n := n - 1
        outptr:= {outptr + 1} % size
      end (* accept *)
    end (* select *)
  end (* loop *)
end (* buffer *)
```

```
(* Rendezvousaufrufe *)
let producer(s, n :Int) =
  begin
    for i=s upto n do
      let val=...
      rendezvous.put(append i)
    end
  end
```

```
let consumer() =
  begin
    let val = tuple let var x = 0 end
    loop
      rendezvous.put(take val)
      ...
    end
  end
```

Die angegebenen, exemplarischen Syntaxerweiterungen ermöglichen eine einfache und sichere Benutzung der Funktionen der Module *mutex* und *rendezvous*. Weitere Syntaxerweiterungen, beispielsweise zur Einführung von Pfadausdrücken (vgl. Abschnitt 3.5.4) sind denkbar. Dieser Mechanismus kann durch bekannte Algorithmen unter Verwendung von Semaphoren implementiert werden [Habermann 75].

Kapitel 5

Synchronisation migrierender Threads

Das Tycoon-System verfügt über Mechanismen, beliebige persistente Objekte in entfernte Objektspeicher zu verlagern [Mathiske 96]. Diese Mechanismen sind daher auch auf Tycoon-Threads anwendbar. Thread-Mobilität bedeutet, daß ein Thread den Adreßraum, in dem der Thread abläuft, verlassen und in einen entfernten Adreßraum (Objektspeicher) migrieren kann. In diesem wird die Ausführung des Threads fortgesetzt. Weitere Migrationsvorgänge, zum Beispiel die Rückkehr in den ursprünglichen Adreßraum, können folgen.

Die potentielle Mobilität von Tycoon-Threads ist bei der Implementation der Synchronisationsverfahren zu berücksichtigen. In diesem Kapitel wird zunächst eine Übersicht über Funktionsweise und Implementation von Thread- und Objektmobilität gegeben (Abschnitt 5.1). Ein generelles, bei der Migration von Tycoon-Objekten auftretendes Problem stellen transitive referentielle Abhängigkeiten zwischen Tycoon-Objekten dar. In Abschnitt 5.2 werden eine Übersicht über den Zusammenhang zwischen Migration, referentiellen Abhängigkeiten und Synchronisationsmechanismen gegeben sowie die Auswirkungen von Thread-Migration auf die Implementation der Synchronisationsmechanismen beschrieben.

Wie nachfolgend beschrieben wird, kann ein Thread eigenständig migrieren oder passiv von einem weiteren Thread in einen entfernten Objektspeicher übertragen werden. Threads, die in diesem Zusammenhang an Synchronisationsvorgängen beteiligt sind, können den Zustand weiterer, ebenfalls an diesen Synchronisationvorgängen beteiligter Threads und Synchronisationsvariablen verändern. Die dadurch erforderlichen Maßnahmen, die eine korrekte Funktionsweise der Synchronisationsmechanismen bei Migration sicherstellen, werden im Abschnitt 5.3 behandelt.

5.1 Implementation der Thread–Mobilität

Um eine Mobilität von Tycoon-Objekten zu erreichen, werden typsichere, entfernte Funktionsaufrufe[1] benutzt [Göllnitz 96]. Prinzipiell vollzieht sich die Übertragung eines Tycoon-Objekts durch dessen Übergabe als Argument an eine entfernte Funktion:

[1] Remote Procedure Calls (RPC).

1. Argumentwerte werden mittels Netzwerkfunktionalität zu einem entfernten System übertragen.

2. Die entfernte Funktion wird mit den erhaltenen Argumenten ausgeführt.

3. Der Rückgabewert der entfernten Funktion wird an das aufrufende System zurückgesendet.

Um eine entfernte Funktion aufrufen zu können, ist zunächst eine Bindung an einen entfernten Dienst notwendig. Durch eine derartige Bindung wird einem Dienstnehmer ein ausführbares Tycoon-Objekt zur Verfügung gestellt, das eine Anbindung an eine entfernte Funktion leistet. Wie eine derartige Bindung an einen entfernten Dienst realisiert wird, beschreibt der folgende Abschnitt.

5.1.1 Entfernte Funktionsaufrufe

Ein Tycoon-System, das weiteren Tycoon-Systemen Dienste als entfernte Funktionsaufrufe zur Verfügung stellt, registriert diese Dienste durch Aufruf der Funktion *register* des Moduls *server*. Diese Registrierung findet bei einem im Tycoon-System implementierten Auskunftsdienst für entfernte Dienste statt [Göllnitz 96]:

```
register(server
        :ServiceType
        :Attributes
        attributFunction
        searchFunction
        rpc)
```

Die Funktion *register* erwartet als Argument ein Server-Objekt *server*, das die späteren Dienstanfragen bearbeitet. Der Typ des zur Verfügung gestellten, entfernten Dienstes wird durch den Datentyp *:ServiceType* repräsentiert. Das Argument *Attributes* dient der Angabe des Datentyps der weiteren Attribute des Dienstes, die Funktion *attributFunction* auf der Seite des Diensterbringers der dynamischen Feststellung von Attributen. Beispielsweise kann eine derartige Funktion bei einer Anfrage zeitabhängige Attribute berechnen. Der Rückgabewert dieser Funktion wird für die Auswahl des gewünschten Dienstes benutzt (siehe *bind*). Eine auf den definierten Dienst abgestimmte Suchfunktion wird durch die Funktion *searchFunction* spezifiziert. Diese Funktion kann auf der Seite des Dienstnehmers benutzt werden, um einen Dienst nach einem Verbindungsabbruch wieder aufzufinden. Durch den Wert *rpc* wird die Funktionalität angegeben, die einem entfernten Dienstnehmer zur Verfügung gestellt wird. Die in diesem Wert kodierten Funktionen werden bei einem entfernten Aufruf durch den Server-Thread *server* ausgeführt.

Dienstnehmer erhalten durch Aufrufe der Funktion *bind* des Moduls *client* Zugriff auf entfernte Dienste:

```
let rpc = client.bind(:ServiceType  :Attributes  predicate)
```

Wiederum wird der Typ des benötigten Dienstes durch das Argument *:ServiceType* angegeben. Der gewünschte Datentyp der Dienstattribute wird durch Angabe des Arguments *:Attributes* spezifiziert. Eine Prädikatsfunktion, durch die ein Dienst auf Seite des Diensterbringers ermittelt wird, enthält das Argument *predicate*. Dieser Funktion wird während der Dienstauswahl der Rückgabewert der zuvor besprochenen Funktion *attributeFunktion* übergeben. Das folgende Beispiel beschreibt die Registrierung eines Dienstes zur Berechnung eines ganzzahligen Werts auf der Seite des Diensterbringers und dessen Benutzung seitens des Dienstnehmers:

```
(* Spezifikation des Dienstes *)
Let ServiceType = Tuple compute(x :Int) :Int end
Let Attributes = String
let attributeFunction() :Attributes = "computeservice"
let rpc :ServiceType = tuple
  let compute(x :Int) = begin ... end
end

(* Registrierung des Dienstes *)
let newServer = server.new()
server.register(newServer :ServiceType  :Attributes  attributeFunction
                server.searchFun  rpc)
```

Auf der Seite des Diensterbringers steht nach Aufruf der Funktion *register* ein Dienst *compute* für entfernte Aufrufe zur Verfügung. Auf der Seite des Dienstnehmers kann jetzt die Bindung an diesen Dienst erfolgen:

```
(* Spezifikation des Dienstes *)
Let ServiceType = Tuple compute(x :Int) :Int end
Let Attributes = String
let predicate(s :Attributes) :Bool = string.equal(s  "computeservice")

(* Bindung an den entfernten Dienst *)
let computeservice = client.bind(:ServiceType  :Attributes  predicate)

(* Benutzung des Dienstes *)
let result = computeservice.compute(77)
```

Durch Aufruf der Funktion *bind* wird ein passender Dienst über den Auskunftsdienst ermittelt. Diese Auswahl findet über den Typ des Dienstes (*Attributes*) und das angegebene Prädikat (*predicate*) statt. Zunächst werden aus der Menge der auf Diensterbringerseite verfügbaren Dienste diejenigen ausgewählt, deren Typ dem über *bind* angegebenen Typ entspricht. Dann wird für jeden dieser Dienste die dazugehörige Attributfunktion zur Ermittlung der Dienstattribute ausgewertet. Mit dem Resultatwert einer solchen Auswertung wird die von *bind* übergebenen Prädikatsfunktion *predicate* aufgerufen. Ergibt eine solche Auswertung den boolschen Wert *true*, so wird dieser Dienst an den Dienstnehmer übermittelt.

Auf die Realisierung der konkreten Übertragung der Objektdaten wird an dieser Stelle nicht tiefer eingegangen. Dort, wo Details der dafür zuständigen Mechanismen notwendig sind, folgen weitere Erläuterungen.

5.1.2 Threadmigration

Die Migration von Tycoon-Threads bedient sich der im vorherigen Abschnitt beschriebenen, entfernten Funktionsaufrufe. Auf der Senderseite wird eine entfernte Funktion der Empfängerseite aufgerufen, die als Parameter den zu übertragenen Thread erhält. Auf der Empfängerseite wird dieser Thread entgegengenommen und erneut gestartet. Die Migration von Threads wird wie folgt implementiert [Mathiske 96]:

Die Deklaration des Typs der zu übertragenden Datenobjekte und des Typs der entfernten Funktion zur Datenübertragung findet im Modul *gate* statt, das von allen an Migrationsvorgängen beteiligten Tycoon-Systemen importiert werden muß:

```
Let Parameter(Data <:Ok) = Tuple
  var data :optional.T(Data)
  agent :thread.T(Ok)
end
```

Ein Element des Typs *Parameter(Data)* wird auf das entfernte System übertragen. Es enthält den zu übertragenden Thread *agent* sowie ein Datenelement *data*, das die Anbindung des übertragenen Threads an den Adreßraum des Zielrechners ermöglicht.

```
Let T(Data <:Ok) <:Ok = Tuple
  transfer(param :Parameter(Data)) :Ok
end
```

Der parametrisierte Typ *T(Data)* des entfernten Funktionsaufrufs zur Thread-Übertragung ist der Servicetyp des entfernten Dienstes (vgl. Abschnitt 5.1.1).

Für den Migrationsvorgang steht die Funktion *migrateTo* zur Verfügung. In dieser Funktion werden durch *thread.launch* ein Hilfsthread erzeugt und der zu migrierende Thread suspendiert. Dieser Hilfsthread überträgt durch Aufruf der durch *site* gegebenen, entfernten Funktion *transfer* seinen Erzeugerthread auf das Zielsystem [Mathiske 96]:

```
let migrateTo(Data <:Ok  site :T(Data)) :Data =
  begin
    let param :Parameter(Data) = tuple
      let var data = optional.nil(:Data)
      let agent = thread.self()
    end
    thread.launch(fun(self :thread.T(Ok))
      begin
        site.transfer(param)        (* Aufruf der entfernten Funktion *)
        thread.kill(param.agent)
      end)
    (* Dieser Teil wird bereits auf dem entfernten System ausgeführt *)
```

```
    optional.value(param.data)
  end
```

Zu beachten ist die Beendigung des *migrateTo* aufrufenden Threads auf dem Ursprungssystem durch den Aufruf von *thread.kill*. Dies führt zur Beendigung des Threads im Sendersystem und zum Abbau der dem Thread zugeordneten, flüchtigen Datenstrukturen des Laufzeitsystems.

Der Aufruf der Funktion *optional.value(param.data)* gibt einen Wert des entfernten Systems an den übertragenen Thread zurück. Dadurch wird die Öffnung des Adreßraums des entfernten Systems für den übertragenen Thread erreicht. Der Wert *param.data* wird durch die im folgenden erläuterte Technik des dynamischen Bindens von Tycoon-Objekten ermittelt:

Zunächst lautet die Implementation der auf dem Zielsystem ausgeführten, entfernten Funktion *transfer* wie folgt:

```
let data() = begin (* Rückgabe eines Werts des Zielsystems *) end
let rpc = tuple
  let transfer(param :Parameter(Data)) =
    begin
      param.data := optional.new(data())
      thread.run(param.agent)
    end
  end
```

Die Funktion *transfer* startet den vom Sendersystem übertragenen Thread erneut. Der durch die Auswertung der Funktion *data()* erhaltene Wert des Zielsystems wird von der Funktion *migrateTo* als Ergebniswert zurückgegeben. Da diese Auswertung der Funktion *data()* auf dem Zielsystem stattfindet und den Wert eines Objekts des Zielsystems zurückgibt, erhält der übertragene Thread Zugriff auf Objekte des Zielsystems: Der Adreßbereich des übertragenen Threads ist um Objekte des Zielsystems erweitert worden.

Die konkrete Migration eines Threads vollzieht sich dann in zwei Schritten. Zuerst wird die Zieladresse des entfernten Systems festgelegt. Dann erfolgt der eigentliche Migrationsvorgang durch Aufruf der Funktion *migrateTo*:

```
let site = client.bind(...)
let data = migrateTo(site)
```

Auf der Seite des Dienstnehmers, des Senders des Threads, wird zunächst durch Aufruf der Funktion *bind* die Zieladresse für die Migration durch Binden an einen Dienst des Zielrechners festgelegt. Die Funktionalität dieses Dienstes besteht, wie beschrieben, in der Entgegennahme und dem Starten des übertragenen, passiven (suspendierten) Thread-Objekts.

Es ist darauf hinzuweisen, daß die Implementation von Thread-Migration lediglich die direkte Übertragung eines Threads unterstützt. Befinden sich weitere Threads im transitiven oder direkten Sichtbarkeitsbereich des migrierenden Threads (siehe Abschnitt 5.2), so liegt die Verantwortung für die korrekte Übertragung dieser Threads bei dem Programmierer des zu migrierenden Threads. Insbesondere ist ein Benutzer der Mechanismen zur

Thread-Migration für die Suspendierung referenzierter Threads zum Abbau flüchtiger Datenstrukturen des Laufzeitsystems vor der Übertragung und für die erneute Aktivierung dieser Threads auf dem Zielsystem selbst verantwortlich. Das Beispiel in Abbildung 5.1 (Seite 83) zeigt, wie eine Migration mehrerer, zum migrierenden Thread lokaler Threads, zu implementieren ist:

Alle Threads, die mit auf das entfernte System migrieren sollen, sind von dem die Migration durchführenden Thread vor der Übertragung zu suspendieren. Die Aktivierung des migrierenden Hauptthreads (*agent*) wird auf der Empfängerseite durch die Funktion *transfer* gewährleistet. Alle weiteren Threads sind durch die Funktion *agent* auf der Empfängerseite erneut zu aktivieren. Ein Beenden der lokalen Threads durch *thread.kill* ist nicht notwendig, da diese aufgrund ihrer lokalen Sichtbarkeit nicht außerhalb von *agent* referenzierbar sind.

```
(* Implementation der Threadübertragung *)
let agent(site :ServiceType) =
  begin
    (* Deklarationen *)
    let funcA(self :thread.T(Ok)) =
      begin
        ...   (* Arbeitsteil der Funktion funcA *)
      end

    let funcB(self :thread.T(Ok)) =
      begin
        ...   (* Arbeitsteil der Funktion funcB *)
      end

    (* Starten lokaler Threads *)
    let threadA = thread.fork(funcA)
    let threadB = thread.fork(funcB)

    (* Weitere Kodeabschnitte von agent *)
    ...
    (* Suspendierung der lokalen Threads vor der Migration *)
    thread.suspend(threadA)
    thread.suspend(threadB)

    (* Durchführung der Migration *)
    let data = migrateTo(site)

   (* Die weiteren Abschnitte werden bereits auf dem entfernten System ausgeführt *)
    thread.run(threadA)    (* Neuaktivierung der lokalen Threads *)
    thread.run(threadB)

    (* Weitere Kodeabschnitte *)
  end

(* Festlegung der Zieladresse *)
let site = client.bind(...)

(* Starten des Threads für die Funktion agent *)
thread.fork(fun(self :thread.T(Ok)) agent(site))
```

Abbildung 5.1: Migration mehrerer Threads

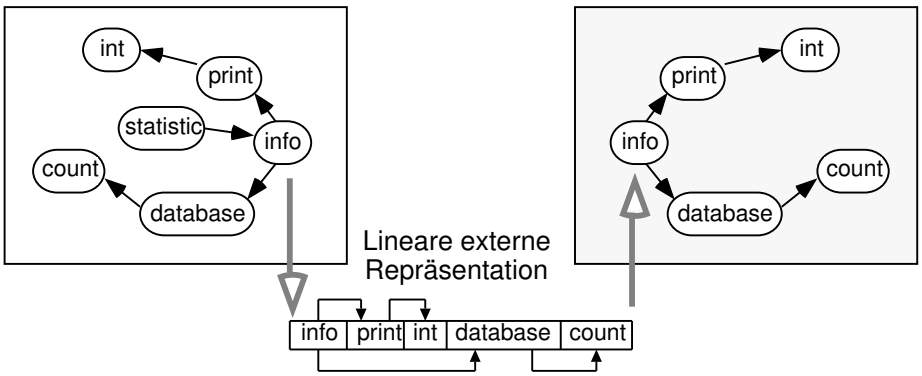

Abbildung 5.2: Übertragung eines Objekts durch Tiefenkopie

5.2 Die Minimierung transitiver referentieller Abhängigkeiten

Ein wichtiger Begriff im Zusammenhang mit der Mobilität von Tycoon-Objekten ist der mit diesen Objekten verbundene, transitive Sichtbarkeitsbereich. Hierunter ist die Menge der durch ein Objekt transitiv erreichbaren Objekte zu verstehen. Ein Tycoon-Objekt hat im allgemeinen Referenzen auf weitere Tycoon-Objekte. Im linken Teil der Abbildung 5.2 sind Referenzen zwischen einigen Tycoon-Objekten verzeichnet. Beispielsweise besteht die Menge der durch das Objekt *info* transitiv erreichbaren Objekte aus *database*, *count*, *print* und *int*. Das Objekt *statistic* gehört dagegen nicht zur Menge der von *info* erreichbaren Objekte. Da bei einem Migrationsvorgang die referentielle Integrität zwischen Tycoon-Objekten erhalten bleiben muß, werden auch alle von dem zu übertragenen Objekt referenzierten Objekte übertragen. Dieser Kopiervorgang (*Tiefenkopie, deep copy*) überträgt alle durch eine Objekt transitiv erreichbaren, weiteren Objekte in den entfernten Objektspeicher. Ein Nachteil dieses Verfahrens besteht in dem Umfang der zu übertragenen Datenmengen. Werden zum Beispiel durch einen zu übertragenen Thread eine umfangreiche Datenbank und mehrere Module des Tycoon-System referenziert, müssen diese Objekte ebenfalls übertragen werden. Dies führt zu erheblichen Datenmengen, die den Nutzen von Migrationsvorgängen erheblich reduzieren. Im ungünstigsten Fall kann ein Migrationsvorgang zur Übertragung des gesamten Objektspeicherinhalts führen.

Die Implementation der Objektübertragung und Thread-Migration führt zur Mitübertragung aller durch ein Objekt transitiv erreichbarer Objekte. Insbesondere werden auch alle transitiv erreichbaren Threads übertragen. Wie in Abschnitt 5.1 angemerkt, werden diese Threads durch die Funktion zur Thread-Migration (*migrateTo*) nicht berücksichtigt. Der Benutzer dieser Funktion ist für die Verwaltung dieser Threads verantwortlich.

Durch Synchronisationsmechanismen ist eine weitere Verschärfung der mit der Tiefenkopie verbundenen Probleme möglich. Threads, die untereinander keine Referenzen besitzen, können durch Synchronisationsmechanismen Referenzen aufeinander erhalten. Dieses Problem tritt immer dann auf, wenn Threads, die Referenzen auf eine Variable eines Synchronisationsmechanismus besitzen, einen Migrationsvorgang einleiten. Ein einfaches Beispiel

macht die Problematik deutlich:

```
let m = mutex.new()
let funcA(self :thread.T(Ok)) = begin
  mutex.lock(m)
  ...
  mutex.unlock(m)
  let site = ...
  let data = migrateTo(site)
end

let funcB(self :thread.T(Ok)) =
  begin
    mutex.lock(m)
   ...
    mutex.unlock(m)
  end

let threadA = thread.fork(funcA)
let threadB = thread.fork(funcB)
```

Auf Synchronisationsmechanismen wartende Threads werden in einer Warteschlange innerhalb der Synchronisationsvariablen gespeichert. Ein Migrationsvorgang führt dann zur Mitübertragung dieser Threads. Im obigen Beispiel führt daher der Migrationsvorgang von *threadA* zur Mitübertragung von *threadB* und aller von *threadB* transitiv erreichbaren Objekte, wenn *threadB* sich zum Zeitpunkt der Migration auf der Warteschlange des Mutex m befindet. Derartige Referenzen können unter anderem auch durch Benutzung von Tycoon-Modulen entstehen, deren interne Datenstrukturen durch Synchronisationsmechanismen abgesichert sind. Ein Verfahren zur Minimierung dieser transitiven Referenzen wird im folgenden beschrieben.

Um eine Übertragung von Tycoon-Objekten in entfernte Objektspeicher ohne die Mitübertragung großer Mengen transitiv referenzierter Objekte zu ermöglichen, bietet das Tycoon-System ein Verfahren zur Minimierung derartiger Referenzen an. Ein wesentlicher Gesichtspunkt bei diesen Verfahren besteht in der Beobachtung, daß typische Tycoon-Systeme zueinander ähnlich aufgebaut sind. So existiert eine große Menge von Modulen, deren Existenz in jedem Tycoon-System vorausgesetzt werden kann. Derartige Module werden als allgegenwärtig (*ubiquitär*) bezeichnet. Das Tycoon-Modul *atomic* gestattet die Kennzeichnung derartiger ubiquitärer Module [Pour 96]. Diese Kennzeichnung ist jedoch nicht auf Module beschränkt, sondern ermöglicht die Deklaration beliebiger Tycoon-Objekte als ubiquitäre Ressourcen. Die konkrete Übertragung von Objekten zwischen Tycoon-Systemen ist so implementiert, daß ubiquitäre Ressourcen nicht mitübertragen werden. Statt dessen wird der Objektgraph eines zu übertragenen Objekts, der die Menge aller durch ein Tycoon-Objekt transitiv erreichbaren Objekte enthält, bei als ubiquitär gekennzeichneten Objekten abgeschnitten und durch eine symbolische Referenz ersetzt. Nach der Übertragung wird dieser Objektgraph durch einen Bindevorgang [Pour 96] wieder vervollständigt, symbolische Referenzen ubiquitärer Objekte werden durch die entsprechenden Objektgraphen auf dem Empfängersystem ersetzt. Abbildung 5.3 zeigt eine derartige Übertragung:

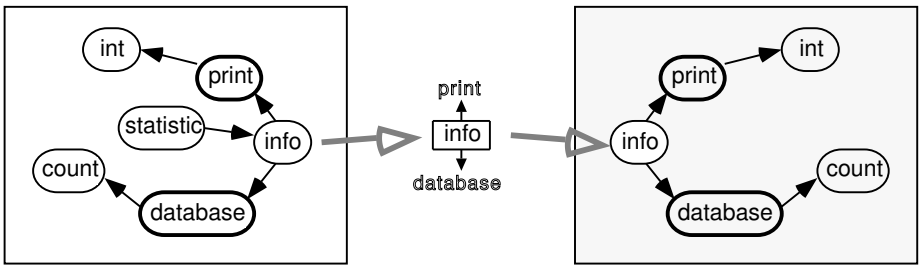

Abbildung 5.3: Rebinden ubiquitärer Objekte

Das Tycoon-Objekt *info* ist einen anderen Objektspeicher zu übertragen. Die Objekte *print* und *database* sind als ubiquitär gekennzeichnet. Dadurch ist lediglich die Übertragung des Objekts *info* notwendig. Nach der Übertragung wird durch dynamisches Rebinden der Objektgraph von *info* wieder vervollständigt.

Eine unerwünschte, indirekte Übertragung von Threads durch in ubiquitären Objekten benutzte Synchronisationsmechanismen ist aus den oben genannten Gründen ausgeschlossen. Es ist jedoch anzumerken, daß durch die Migration von Threads, die mit ubiquitären Objekten in Synchronisationsbeziehungen stehen, weitere Probleme durch Deaktivierung und Neuaktivierung dieser Threads entstehen können (siehe folgenden Abschnitt).

Bei der Programmierung migrierender Threads ist immer zu beachten, daß zusätzliche transitive Referenzen durch Synchronisationsvorgänge mit nicht ubiquitären Objekten enstehen können.

5.3 Zustandswechsel migrierender Threads

Migrierende Tycoon-Threads sind für die Migration weiterer Threads aus ihrem Sichtbarkeitsbereich verantwortlich. Aktive Threads müssen von dem den Migrationsvorgang vollziehenden Thread vor der Migration deaktiviert werden. Nach dem Migrationsvorgang sind derartige Threads erneut zu aktivieren. Hierbei ist zu beachten, daß eine einfache Suspendierung und erneute Aktivierung aller zu migrierenden Threads nicht ausreichend ist. Alle von einem Thread während des Migrationsvorgangs verwalteten Threads können sich zum Zeitpunkt der Übertragung in allen Threadzuständen befinden (vgl. Abbildung 4.8, Seite 60). Der Zustand eines Threads wird jedoch bei Migrationsvorgängen geändert:

- Insbesondere werden Threads auch aufgrund von Migrationsvorgängen suspendiert und beendet. Die Beendigung zu übertragender Threads auf dem Ursprungssystem kann daher zu Verklemmungen führen, wenn diese Threads Synchronisationsvariablen akquiriert haben, auf die weitere Threads warten. Diese Situation kann beispielsweise entstehen, wenn zu migrierende Threads Synchronisationsmechanismen ubiquitärer Ressourcen halten.
- Bei einem Migrationsvorgang wird der aktiv migrierende Thread auf der Empfängerseite neu gestartet. Alle weiteren, migrierenden Threads müssen nach der Migration

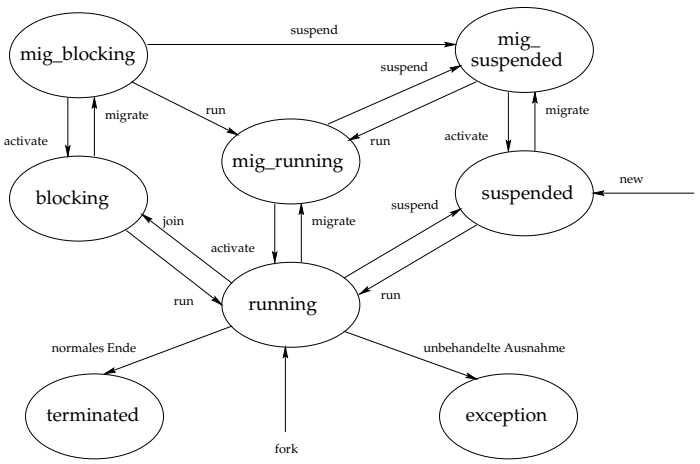

Abbildung 5.4: Zustandsübergänge unter Berücksichtung von Migration

auf das entfernte System von dem aktiv migrierenden Thread erneut gestartet werden. Dieses erneute Aktivieren muß von einer Aktivierung nach Zuteilung einer Synchronisationsvariablen unterschieden werden. Erfolgt diese Unterscheidung nicht, so werden blockierte Synchronisationsmechanismen durch diese inkorrekte Aktivierung wartender Threads fälschlicherweise als verfügbar betrachtet.

Die Probleme, die sich aus der Beendigung von Threads ergeben, die Synchronisationsmechanismen ubiquitärer Ressourcen halten, kann derzeit nur durch eine explizite Berücksichtigung bei der Programmierung von Thread-Migration gelöst werden. Es ist zu verhindern, daß sich ein zu migrierender Thread in einem internen, kritischen Abschnitt eines ubiquitären Objekts befindet, da nach einem Migrationsvorgang eine neue, automatische Akquisition des kritischen Abschnitts nicht gewährleistet werden kann.

Eine mögliche Lösung für obiges Problem ist die Einführung von Netzwerkreferenzen für Tycoon-Objekte. Bei einem Migrationsvorgang werden Referenzen auf immobile (ubiquitäre) Objekte durch Netzwerkreferenzen auf dieses Objekt ersetzt. Derartige Referenzen ermöglichen einen transparenten Zugriff auf nicht migrationsfähige Objekte aus entfernten Objektspeichern. Diese Technik wird beispielsweise in der Programmiersprache Obliq eingesetzt [Cardelli 94].

Der Zustand eines Threads muß nach Abschluß eines Migrationsvorgangs dem Zustand im ursprünglichen System entsprechen. Um dies zu erreichen ist die Einführung zusätzlicher Hilfszustände für Threads notwendig. Die in der Abbildung 4.8 angeführten Zustände werden um die Zustände *mig_blocking*, *mig_running* und *mig_suspended* erweitert. Abbildung 5.4 zeigt die neuen Zustandsübergänge für Threads im Tycoon-System. Das Modul *thread* wird um zusätzliche Funktionen zur Steuerung migrierender Threads erweitert. Im einzelnen werden die Funktionen

*migrate(t :thread.T(**Ok**)) :**Ok***

*activate(t :thread.T(**Ok**)) :**Ok***

neu eingeführt. Die Funktion *migrate* überführt den angegebenen Thread abhängig von seinem ursprünglichen Zustand in einen neuen Zustand. Falls der Thread vor Ausführung von *migrate* im Zustand *running* ist, wird der Thread deaktiviert und in den Zustand *mig_running* überführt. Dieser entspricht effektiv dem alten Zustand *suspended*, alle flüchtigen Datenstrukturen des Laufzeitsystems sind abgebaut. Jedoch führen nachfolgende Aufrufe der Funktion *suspend* zur Überführung in den Zustand *mig_suspended*. Analog wirken sich Aufrufe der in Abbildung 5.4 angegebenen Funktionen auf die weiteren, neuen Zustände aus. In keinem Fall wird ein Thread erneut aktiviert, mit der Ausnahme der Aktivierung mittels *activate* durch den die Migration veranlassenden Hauptthread. Dadurch bleibt der im Sendersystem vorhandene Zustand eines Threads auch nach Migration erhalten.

Kapitel 6

Zusammenfassung und Ausblick

Die vorliegende Arbeit beschreibt Synchronisationsmechanismen für nebenläufige Aktivitäten im Umfeld persistenter Objektsysteme. Für das dieser Arbeit zugrundeliegende, persistente Objektsystem Tycoon werden portable, persistente und migrationsfähige Basismechanismen zur Synchronisation implementiert. Das Tycoon-System stellt seinen Benutzern mit der Programmiersprache TL sehr leistungsfähige Mechanismen wie Polymorphismus, Funktionen höherer Ordnung sowie die Möglichkeit zur Erweiterung der Basissprache zur Verfügung. Diese Sprachmittel gestatten es, bei der Auswahl von Synchronisationsmethoden Aspekte wie sichere Benutzbarkeit und Verständlichkeit von Synchronisationsmechanismen zunächst in den Hintergrund zu stellen. Vielmehr erlauben die genannten Systemeigenschaften die Bereitstellung einer Hierarchie von Synchronisationsmethoden, deren untere Ebenen auf Schwerpunkte wie leichte Implementierbarkeit und Erweiterbarkeit ausgerichtet sind. Sicherheit und leichte Verständlichkeit bei der Benutzung werden durch den Einsatz von Funktionen höherer Ordnung und Syntaxerweiterungen erreicht. Den Benutzern des Systems stehen damit eine Reihe von flexiblen und erweiterbaren Synchronisationsmechanismen zur Verfügung.

Die Synchronisationsmechanismen werden auf der Grundlage eines portablen Mechanismus für den atomaren Ablauf von TL-Funktionen aufgebaut. Die Portabilität dieses Mechanismus erlaubt die unmittelbare Benutzung von Synchronisationsmechanismen auf allen für das Tycoon-System verfügbaren Betriebssystemplattformen. Der Aufwand für die Sicherstellung der Persistenz des atomaren Grundmechanismus ist auf wenige Eingriffe in das Tycoon-Laufzeitsystem beschränkt.

Der alleinige Einsatz der Programmiersprache TL zur Implementation aufbauender Mechanismen sichert die Persistenz dieser Mechanismen. Synchronisationsmechanismen im Tycoon-System verhalten sich wie alle anderen im System vorhandenen, persistenten Objekte.

Der geschichtete Aufbau der Synchronisationsmechanismen sichert weiterhin deren potentielle Mobilität, da in jedem Tycoon-System, unabhängig vom zugrundeliegenden Betriebssystem, der gleiche Grundmechanismus zur Ermöglichung atomarer Vorgänge vorhanden ist. Die Festlegung der Mobilitätseigenschaften von Synchronisationmechanismen erfolgt durch den für alle TL-Objekte verfügbaren Mechanismus der Registrierung als ubiquitäre Ressource. Besonderheiten, die bezüglich der Synchronisationsmechanismen bei der Migration von Aktivitäten vorliegen, wurden in der Implementation berücksichtigt.

Der Aufbau der Mechanismen orientiert sich weitgehend an vorhandenen Standards zur Threadsynchronisation in Programmiersprachen und Betriebssystemen. Dadurch sind diese Mechanismen für schon mit derartigen Standards vertraute Benutzer besonders einfach zu verwenden.

6.1 Erfahrungen mit dem Tycoon-System

Das Tycoon-System bietet durch seinen klar verständlichen, geschichteten Aufbau eine komfortable Arbeitsgrundlage für die Implementation portabler, persistenter und mobiler Synchronisationsmechanismen in persistenten Objektsystemen. Die mächtigen, zur Verfügung stehenden Sprachmittel, das hierarchische Modulkonzept und das Vorhandensein von Mechanismen zur Manipulation grundlegender Spracheigenschaften erleichtern darüber hinaus die Schaffung generischer, erweiterbarer Synchronisationsmechanismen. Die praktische Implementation der Mechanismen stieß jedoch auf eine Reihe von Problemen, die eng mit dem derzeitigen, experimentellen Stand des Laufzeitsystems verknüpft sind. Durch das dieser Arbeit zugrundeliegende Thema wurde zum ersten Mal massiv mit nebenläufigen Threads sehr großer Anzahl gearbeitet, die zudem interagieren und miteinander kommunizieren. Dadurch war es möglich, noch im Laufzeitsystem vorhandene Fehler und Probleme zu identifizieren. Diese teilweise sehr subtilen Probleme, deren wahre Ursachen durch die vorhandene Nebenläufigkeit oft schwer zu lokalisieren waren, nahmen in der Implementationsphase einen nicht unerheblichen Teil der zur Verfügung stehenden Zeit in Anspruch. So zeigten sich verschiedene Probleme sowohl in der Verwaltung der Koroutinen als auch in der Interaktion zwischen Laufzeitsystem und Speicherprotokoll TSP. Die zuvor mit der Portierung der ersten und der derzeitigen zweiten Version des Tycoon-Systems auf ein Unix-Betriebssystem gemachten Erfahrungen erwiesen sich jedoch bei der Beseitigung dieser Probleme als großer Vorteil. Zusammenfassend ergibt sich daher, zusätzlich zur Verfügbarkeit persistenter, migrationsfähiger Synchronisationsmechanismen, eine erhöhte Stabilität und Fehlerfreiheit des gesamten Tycoon-Systems für die Benutzung nebenläufiger Aktivitäten.

6.2 Der Stand der Implementierung

Die Implementation der in Kapitel 4 besprochenen Mechanismen ist weitgehend abgeschlossen und ausgetestet. Die mit der Synchronisation nebenläufiger Aktivitäten einhergehenden Probleme, wie beispielsweise Verklemmungen, bedürfen noch einer weiteren Behandlung. Jedoch wurden bei der Implementation der Mechanismen bereits Vorkehrungen für die Lösung derartiger Probleme getroffen. So ist bereits ein Zugriff auf die durch einen Thread gehaltenen Synchronisationsobjekte im Laufzeitsystem vorgesehen. Diese Informationen bilden zusammen mit der Liste der auf ein Synchronisationsobjekt wartenden Threads die Grundlage zur Implementation bekannter Algorithmen zur Erkennung, Vermeidung und Auflösung von Verklemmungen.

Eine der umfangreichsten, noch zu behandelnden Aufgaben im Zusammenhang mit nebenläufigen Threads und deren Synchronisation bilden die bereits in großem Umfang vorhandenen, generischen Modulbibliotheken des Tycoon-Systems. Hier ist es erforderlich, jedes

Modul auf die Eignung durch eine nebenläufige Benutzung zu untersuchen und gegebenenfalls unter Benutzung der implementierten Synchronisationsmechanismen zu modifizieren. Von besonderem Interesse sind hier die vorhandenen Bibliotheken zur Verwaltung von Massendaten, deren sichere, nebenläufige Benutzung zu gewährleisten ist.

6.3 Resümee und Ausblick

Im Rahmen der Architektur des derzeitigen Tycoon-Systems ist mit den implementierten Mechanismen zur Synchronisation eine adäquate Lösung von Synchronisationsproblemen möglich.

Jedoch ist die Verwirklichung von Synchronisationsmechanismen für persistente Objektsysteme mit der vorliegenden Arbeit keinesfalls abgeschlossen. Gravierende Probleme treten bei der Behandlung migrierender Threads im Zusammenhang mit Synchronisationsmechanismen auf. Ein Teil der Verantwortung für die korrekte Verwendung von Thread-Migration und Synchronisationsmechanismen liegt in den Händen des Benutzers. Jedoch werden für diesen Zweck durch die in Kapitel 5 vorgestellten Maßnahmen Hilfsmittel zur Verfügung gestellt. Da die dort vorgestellten Effekte nicht nur durch die Verwendung von Synchronisationsmechanismen, sondern immer im Zusammenhang mit der Speicherung von Threads auftreten, ist hier noch weiterer Forschungsbedarf gegeben.

Ein weiteres, interessantes offenes Thema stellt die Ausnutzung von der in vielen Rechensystemen vorhandenen Mehrprozessorfähigkeit durch das Tycoon-System dar. Eine derartige Erweiterung scheint jedoch erhebliche Auswirkungen auf Teile des Tycoon-Systems zu haben:

- Zur Ausnutzung der Fähigkeiten zur echten Parallelität von Mehrprozessorsystemen ist die Benutzung von Threadmechanismen notwendig, die schon von diesen Systemen bereitgestellt werden. Dies eröffnet die Frage nach der Implementation persistenter Threads auf Basis betriebssystemspezifischer Threads.

- Nebenläufige Threads auf Mehrprozessorsystemen lassen die Verwendung von Objektspeichern, die ebenfalls eine interne nebenläufige Verarbeitung unterstützen, notwendig erscheinen. Auch muß durch die im TSP vorhandenen Mechanismen gegebenenfalls eine Verzögerung von Threads veranlaßt werden können. Ein solches Verhalten erfordert die Verwendung von spezifischen Methoden des Laufzeitsystems und steht im Widerspruch zum bisherigen Systemaufbau.

- Die Verwendung betriebssysteminterner Synchronisationsmechanismen muß, zumindest teilweise, auf einer Ebene unterhalb der Programmiersprache TL erfolgen. Verwendete, flüchtige Datenstrukturen müssen persistent gemacht werden und bei der automatischen Speicherrückgewinnung berücksichtigt werden. So muß angeforderter, flüchtiger Systemspeicher bei der Beseitigung zugeordneter, nicht mehr referenzierbarer, persistenter Objekte freigegeben werden. Ein solches Verhalten setzt das Vorhandensein von schwachen Objektreferenzen im vorhandenen Objektspeicher voraus [Horning et al. 93] und kann die Verwendung beliebiger Objektspeicher, die nicht über derartige Mechanismen verfügen, erschweren.

- Jede Verwendung systemspezifischer Mechanismen vervielfacht den Aufwand für eine Portierung des Tycoon-System auf weitere Betriebssysteme.

Einen weiteren, noch offenen Aspekt stellt die Verbindung der im Tycoon-System vorhandenen Transaktionsmechanismen mit nebenläufigen Aktivitäten dar. Transaktionen betreffen stets die Gesamtheit aller beteiligten Tycoon-Objekte. Eine feinere Granularität von Transaktionen, die das Vorhandensein nebenläufiger Aktivitäten berücksichtigt, erscheint wünschenswert. Alle im Zusammenhang mit der Verwendung von Mehrprozessorsystemen und nebenläufigen Transaktionen gemachten Überlegungen sind keinesfalls vollständig. Sie verdeutlichen vielmehr den nach wie vor vorhandenen Bedarf an weiteren Forschungsarbeiten.

Anhang A

Ausgewählte Modulschnittstellen

A.1 Das Modul Mutex

interface *Mutex*

import *thread*

export

error, deadlock :***Exception end***

T <:***Ok***

State <: ***Ok***

lockedMutex, gotMutex :*State*

new():T
(Create a new unlocked mutex *)*

newLocked():T
(Create a new locked mutex *)*

lock(mutex :T) :***Ok***
(Lock mutex *)*

unlock(mutex :T) :***Ok***
(Unlock mutex, release blocked threads *)*

tryLock(mutex :T) :*State*
(Asynchronous polling of mutex m. Aquires m if m is unlocked.*
Does not block if lock is held (returns locked)
*Returns ok if locking m was successfull *)*

```
    print(mutex :T) :thread.T(Ok)
    (* Print out some statistics *)

end
```

A.2 Das Modul Condition

```
interface Condition

import thread mutex

export

  error, deadlock :Exception end

  T <: Ok

  new():T
  (* Create a new condition *)

  wait(m :mutex.T  cond :T) :Ok
  (* Calling thread must have mutex locked. Atomically unlock
     mutex and wait on cond. Then relock mutex *)

  signal(cond :T) :Ok
  (* Unblock one thread waiting on cond *)

  broadcast(cond :T) :Ok
  (* Unblock all threads waiting on cond *)

end;
```

A.3 Das Modul Rendezvous

```
interface Rendezvous

export

  error, deadlock :Exception end


  T(R <:Ok) <:Ok


  SelectT  <: Ok


  new(R <:Ok  x :R) :T(R)
  (* Create a new rendevous *)


  get(R <:Ok  r :T(R)) :R
  (* Block thread self on r until another thread calls put(r) *)


  put(R <:Ok  r :T(R)  value :R) :Ok
  (* Block thread self on r until another thread calls get(r) *)

  makeEntry(R<:Ok  rend :T(R)  func :Fun(:R) :Ok  guard :Fun() :Bool) :SelectT
  (* Create new entry for select.
     Function func(:R) will be called with value given by put(rend value)
  *)

  select(alt :Array(SelectT)  elseCase :optional.T(Fun():Ok)) :Ok
  (* Select rendezvous from array alt or execute elseCase function (if any) *)

end;
```

A.4 Das Modul Thread

```
interface Thread
  Author : Bernd Mathiske
export

  error :Exception end

  abortion :Exception end

  T(R <:Ok) <:Ok
  (* A value 'thread' of type 'T(R)' is a thread that
     computes a function with the result type 'R'. *)

  State <: Ok

  runningState, blockingState, suspendedState,
  terminatedState, exceptionState :State

  state(R <:Ok  thread :T(R)) :State
  (* Return the current evaluation state of 'thread'. *)

  new(R <:Ok  f(self :T(R)) :R) :T(R)
  (* Create a new "suspended" thread that is bound to evaluate 'f'.
     A handle to the new thread is provided both to the parent
     and the child: it is returned by 'new' and passed as
     an argument ('self') to 'f'. *)

  fork(R <:Ok  f(self :T(R)) :R) :T(R)
  (* Create a new "running" thread that evaluates 'f'.
     A handle to the new thread is provided both to the parent
     and the child: it is returned by 'fork' and passed as
     an argument ('self') to 'f'. *)

  launch(R <:Ok  f(self :T(R)) :R) :Ok
  (* Like 'suspend(thread.self())' and 'fork(f)' in parallel,
     but avoiding that the intermediate result of 'fork', a thread,
     be referentially reachable by the current thread. *)

  duplicate(R <:Ok  thread :T(R)) :T(R)
  (* Return a shallow copy of 'thread'. *)

  join(R <:Ok  thread :T(R)) :R
  (* If 'thread' is the current thread then raise 'error'.
     "Block" the current thread until 'thread' has terminated,
     Iff the final state of 'thread' is 'stateTerminated'
```

```
   then resume execution of the current thread
   by returning the result of 'thread',
   else raise 'error'. *)

joinCatch(R <:Ok  thread :T(R)) :R
(* If 'thread' is the current thread then raise 'error'.
   "Block" the current thread until 'thread' has terminated,
   then 'catch(thread)'.
   Iff the final state of 'thread' is 'stateTerminated'
   then resume execution of the current thread
   by returning the result of 'thread'. *)

suspend(R <:Ok  thread :T(R)) :Ok
(* Iff 'state(thread)' is one of 'runningState' and
   'blockingState' then suspend it, i.e. stop its evaluation
   and set its state to 'suspendedState'.
   Iff 'state(thread) == suspendedState' then do nothing.
   Else raise 'error'. *)

run(R <:Ok  thread :T(R)) :Ok
(* Iff 'state(thread)' is one of 'runningState' and
   'blockingState' then do nothing.
   Iff 'state(thread) == suspendedState' then set its state to
   'runningState' and let it "run", i.e. perform its evaluation.
   Else raise 'error'. *)


(**** Exception Handling: ****)
catch(R <:Ok  thread :T(R)) :Ok
(* Iff 'state(thread) == exceptionState' then raise the identical
   exception that has terminated 'thread' in the current thread,
   else do nothing. *)

throw(R <:Ok  thread :T(R)  raiser() :Ok) :Ok
(* Iff 'thread' is already finished then raise error
   in the current thread.
   Else evaluate 'raiser' in the current thread.
   Iff an exception evades from 'raiser' then pass
   it over to 'thread', leaving the current thread unharmed.
   Iff there is no exception from 'raiser' then
   do not bother 'thread'. *)

setThrowHandler(R <:Ok  thread :T(R)
              handler(self :T(R)  catch() :Ok) :Ok) :Ok
(* Default: 'let handler(...) = catch()'. *)
```

```
(**** Premature termination: ****)
abort(R <:Ok  thread :T(R)) :Ok
(* 'throw(thread  fun() raise abortion'. *)

kill(R <:Ok  thread :T(R)) :Ok
(* Terminate 'thread' immediately and unconditionally -
   in 'exceptionState' with exception 'abortion'. *)


(**** Queries: ****)
main() :T(Int)
(* Return the thread that has been created by the runtime system
   in order to execute a function with a signature like 'Main.main'. *)

self() :T(Ok)
(* Return the current thread that calls this function,
   taking complete loss of the thread's return type into account. *)

running() :Array(T(Ok))
(* Return a vector of all currently running threads,
   neglecting their specific return types. *)


(**** Synchronization: ****)
atomic(R <:Ok  action() :R) :R
(* Execute a "critical section" as an atomic 'action'. *)

sleep(timeout :Real) :Ok
(* Supend self() for 'timeout' seconds *)

(**** Migration: ****)
migrate(thread :T(Ok)) :Ok
(* Prepare thread for migration. Iff 'state(thread) == runningState'
   then suspend thread'. *)

activate(thread :T(Ok)) :Ok
(* Change thread state.  Iff 'state(thread)==mig_running'
   then restart thread. *)

end;
```

Anhang B

Syntaxerweiterungen

B.1 Grammatik für Rendezvous

```
grammar

selectG:Value ===
 "select"
 selectCase=selectCaseG
 selectCases=selectCasesG
 elseCase=optSelectElseCaseG
 "end"
 allSelects=mkBndsCons(selectCase selectCases)
 => value << | rendezvous.select(array allSelects end  elseCase) | >>

selectCasesG:Bnds ===
 "or" selectCase=selectCaseG selectCases=selectCasesG
  => mkBndsCons(selectCase selectCases)
| => mkBndsNil()

selectCaseG:Binding ===
 condition=optSelectCaseConditionG
 swAlternative=swAlternativeG(condition)
 => mkBndAnonValue(swAlternative)

optSelectElseCaseG:Value ===
 "else" b=bnds => value << | optional.new(fun() :Ok  begin b end) | >>
|              => value << | optional.nil(:Fun() :Ok) | >>

optSelectCaseConditionG:Value ===
  "when" condition=value "=>" => value << | fun() condition | >>
|                             => value << | fun() true | >>

swAlternativeG(condition:Value):Value ===
```

```
 "accept"
 name=identifier
 formals=optSWFormalsG
 block=optSWBlockG(formals)
  => value << | rendezvous.makeEntry(name block condition) | >>

optSWFormalsG:Sigs ===
  "(" s=sigs ")" => s
|                 => mkSigsNil()

optSWBlockG(formals:Sigs):Value ===
  "do" b=bnds "end" => value << | fun(formals) begin b end | >>
|                   => value << | fun(formals) ok | >>


value3:Value | == selectValue=selectG => selectValue

end;
```

Literaturverzeichnis

Abel 93: Abel, D. „An introduction to concurrent programming with Synchronous C++". Technical report, Laboratorie de Téléinformatique Ecole Polytechnique Fédérale de Lausane (EPFL), May 1993.

André et al. 85: André, F. Herman, D., and Verjus, J.-P. *Synchronisation of parallel Programs.* NORTH OXFORD ACADEMIC Publishing Company Limited, 1985.

Atkinson et al. 83: Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., and Morrison, R. „PS-Algol, A Language for Persistent Programming". In: *10th Australian National Computer Conference*, Seite 70–79, 1983.

Atkinson et al. 88: Atkinson, C., Moreton, T., and Natali, A. *Ada for Distributed Systems.* Cambridge University Press, 1988.

Bal 95: Bal, H.E. „Comparing data synchronisation in Ada 9X and Orca". *ACM Ada Letters*, 15(1):50–63, January 1995.

Birell et al. 87: Birell, A.D., Guttag, J.V., Horning, J.J., and Levin, R. „Synchronisation Primitives for a Multiprozessor: A Formal Specification". Technical Report 20, Digital Equipment Corporation, Systems Research Center, August 1987.

Breilmann 95: Breilmann, M. „Studienarbeit: Portierung des Tycoon-Sytems auf das Macintosh Betriebssystem". Master's thesis, Fachbereich Informatik, Universität Hamburg, Germany, November 1995.

Brinch Hansen 72: Brinch Hansen, P. „A Comparsion of Two Synchronizing Concepts". *Acta Informatica*, 1(3):238–250, February 1972.

Brinch Hansen 75: Brinch Hansen, P. „The Programming Language Concurrent Pascal". *IEEE Transactions on Software Engineering*, SE-1(2):199–207, June 1975.

Burns et al. 87: Burns, A., Lister, A.M., and Wellings, A.J. *A Review of Ada Tasking*, Band 262, *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

Campione, Walrath 96: Campione, M. and Walrath, K. *The Java Tutorial.* Sun Microsystems, Inc., 1996.

Cardelli et al. 88: Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. „Modula-3 Report". Technical Report ORC-1, Olivetti Research Center, 2882 Sand Hill Road, Menlo Park, California, 1988.

Cardelli 94: Cardelli, L. „Obliq: a language with distributed scope“. Technical report, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, November 1994.

Catanzaro 91: Catanzaro, B.J. *The SPARC Technical Papers.* Sun Microsystems, Inc., 1991.

Crawford, Gelsinger 87: Crawford, H.C. and Gelsinger, P.P. *Programming the 80386.* Sybex Inc., 1987.

DEC 80: DEC. *VAX Hardware Handbook.* Digital Equipment Corporation, 1980.

Deitel 90: Deitel, H.M. *Operating Systems.* Addison-Wesley Publishing Company, second edition, 1990.

Denning 71: Denning, P.J. „Third Generation Computer Systems“. *ACM Computing Surveys*, 3(4):175–216, December 1971.

Dijkstra 68: Dijkstra, E.W. „Cooperating Sequential Processes“. *Programming Languages*, 1968.

Dijkstra 71: Dijkstra, E.W. „Hierachical Ordering of Sequential Processes“. *Acta Informatica*, 1(2), October 1971.

Dijkstra 75: Dijkstra, E.W. „Guarded Commands, Nondeterminacy and Formal Derivation of Programs“. *Comm. ACM*, 18, 1975.

DIN 81: DIN. *DIN66200, Informationsverarbeitung I,5.* Beuth-Verlag, Berlin, 1981.

Ellis, Stroustrup 90: Ellis, M.A. and Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

Engesser 88: Engesser, H. *Duden "Informatik".* Dudenverlag, Mannheim, 1988.

Freisleben 86: Freisleben, B. *Mechanismen zur Synchronisation paralleler Prozesse*, Band 133, *Informatik Fachberichte.* Springer-Verlag, 1986.

Gawecki, Matthes 94: Gawecki, A. and Matthes, F. „The Tycoon Machine Language TML: An Optimizable Persistent Program Representation“. FIDE Technical Report FIDE/94/100, Fachbereich Informatik, Universität Hamburg, Germany, August 1994.

Göllnitz 96: Göllnitz, M. „Studienarbeit: Polymorphe, Persistente Client/Server–Programmierung mit dynamischer, hierarchischer Adreßauflösung“. Master's thesis, Fachbereich Informatik, Universität Hamburg, Germany, April 1996.

Gray, Reuter 93: Gray, J. and Reuter, A. *Transaction Processing – Concepts and Techniques.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, 1993.

Habermann 75: Habermann, A. N. „Path Expressions“. Technical report, Computer Science Department, Carnegie-Mellon University, June 1975.

Hilf, Nausch 84: Hilf, W. and Nausch, A. *M68000 Familie, Band I, Grundlagen und Architektur.* te-wi Verlag GmbH, München, 1984.

Hoare 72: Hoare, C.A.R. *Operating System Techniques*, Band 9, Kapitel Towards a theory of parallel programming, Seite 61–71. Academic Press, London, April 1972.

Hoare 74: Hoare, C.A.R. „Monitors, an Operating System Structuring Concept". *Comm. ACM*, 17:549–557, 1974.

Hoare 78: Hoare, C.A.R. „Communicating Sequential Processes". *Comm. ACM*, 21, 1978.

Hoare 85: Hoare, C.A.R. *Communicationg Sequential Processes.* Series in Computer Science. Prentice Hall International Ltd, Hertfordshire, 1985.

Horning et al. 93: Horning, J., Kalsow, B., McJones, P., and Nelson, G. „Some useful Modula-3 Interfaces". Technical Report 113, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, December 1993.

Ichbiah, others 83: Ichbiah et al. „The Programming Language Ada: Reference Manual". Technical Report MIL-STD-1815A-1983, ANSI, 1983.

INMO87 87: INMOS, INMOS GmbH, 8057 Eching. *The transputer instruction set — a compiler writer's guide*, November 1987.

Inmos 85: Inmos. *IMS T414 Reference Manual.* Inmos Ltd., 1985.

Jessen, Valk 87: Jessen, E. and Valk, R. *Rechensysteme: Grundlagen d. Modellbildung.* Studienreihe Informatik. Springer-Verlag, 1987.

Kirby et al. 95: Kirby, G, Brown, F., Connor, R., Cutts, Q., Dearle, A., Vivienne, D., Morrison, R., and Munro, D. *Napier88 Reference Manual Release 2.0.* University of St. Andrews, Department of Computing Science, 2.0 edition, 1995.

Kirby et al. 96: Kirby, G, Brown, F., Connor, R., Cutts, Q., Dearle, A., Vivienne, D., Morrison, R., and Munro, D. *Napier88 Standard Library Reference Manual Release 2.2.1.* University of St. Andrews, Department of Computing Science, 2.2.1 edition, March 1996.

Kornacker 95: Kornacker, M. „Persistente Sicherungspunkte für langlebige Aktivitäten in offenen Umgebungen". Master's thesis, Fachbereich Informatik, Universität Hamburg, Germany, August 1995.

Krablin 87: Krablin, G.L. „Building Flexible Multilevel Transactions in a Distributed Pesistent Environment". In: *2nd International Workshop on Persistent Object Systems*, Appin, August 1987.

Lagemann 87: Lagemann, K. *Rechnerstrukturen: Verhaltensbeschreibung und Entwurfsebenen.* Springer-Verlag, Berlin, 1987.

Lamb et al. 92: Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. „The ObjectStore Database System". *Communications of the ACM*, 34(10):50–64, 1992.

Mathiske et al. 95: Mathiske, B., Matthes, F., and Schmidt, J.W. „On Migrating Threads". Technical report, Fachbereich Informatik, Universität Hamburg, Germany, January 1995.

Mathiske 96: Mathiske, B. *Mobilität in persistenten Objektsystemen.* Dissertation, Fachbereich Informatik, Universität Hamburg, Germany, May 1996.

Matthes et al. 95: Matthes, F., Müller, R., and Schmidt, J.W. „Towards a Unified Model of Untyped Object Stores: Experience with the Tycoon Store Protocoll". Technical report, Fachbereich Informatik, Universität Hamburg, Germany, March 1995.

Matthes, Schmidt 92: Matthes, F. and Schmidt, J.W. „Definition of the Tycoon Language TL – A Preliminary Report". Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

Matthes, Schmidt 94: Matthes, F. and Schmidt, J.W. „Persistent Threads". In: *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, Seite 403–414, Santiago, Chile, September 1994. (An extended version of this text appeared as [MaSc94b]).

Matthes 93: Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993.

McLaughry 95: McLaughry, S. „TS+ + : Communication Specification Using Path Expressions in the Tuple Space Programming Model". Master's thesis, WILLIAMS COLLEGE, Williamstown, Massachusetts, November 1995.

Moore 93: Moore, C.R. „PowerPC 601 Microprocessor". In: *Proceedings COMPCON 93*, Seite 109–116, San Francisco, CA, February 1993. IEEE.

Munro 93: Munro, D.S. *On the Integration of Concurrency, Distribution and Persistence.* PhD thesis, University of St. Andrews, Department of Computing Science, 1993.

Nelson 91: Nelson, G., Hrsg. *Systems programming with Modula-3.* Series in innovative technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

POSIX 90: „Portable Operating System Interface for Computer Environments (POSIX)". Federal information processing standards publication NBS-FIPS-PUB-151-1, National Bureau of Standards, 1990.

Pour 96: Pour, N.V. „Studienarbeit: Flexible Bindungstechniken für ubiquitäre Ressourcen in verteilten Anwendungen". Master's thesis, Fachbereich Informatik, Universität Hamburg, Germany, February 1996.

Reed, Kanodia 79: Reed, D.P. and Kanodia, R.K. „Synchronisation with Eventcounts and Sequenzers". *Communications of the ACM*, 22(2):115–123, February 1979.

Rudloff et al. 95: Rudloff, A., Matthes, F., and Schmidt, J.W. „Security as an Add-On Quality in Persistent Object Systems". In: *Second International East/West Database Workshop, Klagenfurt, Austria*, Workshops in Computing, Seite 90–108. Springer-Verlag, 1995. (Also appeared as TR FIDE/95/138).

Schröder 94: Schröder, Gerald. „Diplomarbeit: Syntaktische Erweiterbarkeit von Programmiersprachen". Master's thesis, Fachbereich Informatik, Universität Hamburg, Germany, February 1994.

Stallings 92: Stallings, W. *Operating Systems.* Macmillian Publishing Company, New York, 1992.

Steinmetz 88: Steinmetz, R. *OCCAM 2. Die Programmiersprache für parallele Verarbeitung.* Dr. Alfred Hüthig Verlag GmbH, 1988.

Stemple, Morisson 92: Stemple, D. and Morisson, R. „Specifing Flexible Concurrency Control Schemes: An Abstrakt Operational Approach". In: *15th Australian National Computer Conference*, Seite 873–891, Hobart, Tasmania, 1992.

Sun 93: Sun. *SunOS5.3 Guide to Multithread Programming.* Sun Microsystems, Inc., 1993.

USO 90: USO, Unix Software Operation. *UNIX SYSTEM V/386 Release 4, Programmer's Reference Manual.* Prentice-Hall, Inc., 1990.

Welsh, Bustard 79: Welsh, J. and Bustard, J.W. „Pascal-Plus - Another Language for Modular Multiprogramming". *Software-Practice and Experience*, 9(10):947–957, 1979.

Wirth 85: Wirth, N. *Programmieren in Modula-2.* Springer-Verlag, 1985.

Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Hamburg, den 18. Juni 1996

Andreas Piellusch