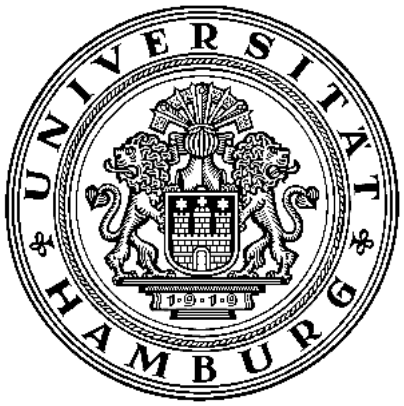

Universität Hamburg
FB Informatik

Diplomarbeit

Entwurf und Implementierung einer portablen multiprozessorfähigen virtuellen Maschine für eine persistente, objektorientierte Programmiersprache

Juli 1998

Marc Weikard
Gänseblümchenweg 18
21614 Buxtehude

Betreuer:
Prof. Dr. Florian Matthes
Dr. Martin Lehmann

Typographie

In dieser Arbeit werden unterschiedliche Schriftarten verwendet, um die Lesbarkeit zu verbessern. Die Bedeutung der einzelnen Schriftarten ist in der folgenden Tabelle wiedergegeben:

Schriftart	**Bedeutung**	**Beispiel**
Emphasized	englischer Begriff oder Definition	*Software Engineering*
Sans Serif	Firmen- oder Produktname	Tycoon-2 yacc
`Typewriter`	TL-2-Ausdrücke oder Klassennamen	`cv.timedWait(10.asLong, mx)` `CompiledMethod`

Inhaltsverzeichnis

1. Einleitung **1**

1.1 Zielsetzung 2

1.2 Vorgehen 3

1.3 Gliederung der Arbeit 3

2. Anforderungen **5**

2.1 Spracheigenschaften 5

2.2 Systemeigenschaften 8

2.2.1 Persistenz 9

2.2.2 Multithreading 9

2.2.3 Offenheit 10

2.2.4 Portabilität 10

2.2.5 Kompatibilität mit dem Prototypen 10

2.2.6 Performanz 11

3. Entwurf **13**

3.1 Systemarchitektur 13

3.1.1 Applikationsschicht 13

3.1.2 Virtuelle Maschine und Objektspeicher 14

3.1.3 Betriebssystem 16

3.2 Objektspeicher 16

3.2.1 Aufbau 16

3.2.2 Globale Verwaltungsinformationen 17

3.2.3 Speicherseiten 18

3.2.4 Objekte 19
3.2.5 Klassendeskriptoren 21
3.2.6 C-Strukturobjekte 22
3.2.7 Speicherbereinigung 24
3.2.8 Schwache Referenzen 25
3.2.9 Persistenz 26
3.3 Virtuelle Maschine 26
3.3.1 Interpreter 26
3.3.2 Vordefinierte Klassen 33
3.3.3 Methoden 37
3.3.4 C-Aufrufschnittstelle 44
3.3.5 Funktionen höherer Ordnung 44
3.3.6 Ausnahmebehandlung 46

4. Implementierung **49**
4.1 Objektspeicher 49
4.1.1 Objekte 49
4.1.2 Objektallokation 50
4.1.3 Seitenallokation 51
4.1.4 Speicherbereinigung 53
4.1.5 Persistenz 58
4.2 Virtuelle Maschine 60
4.2.1 Zusammenspiel von TL-2 und virtueller Maschine 60
4.2.2 Methodensuche 62
4.2.3 Methodencache 65
4.2.4 C-Aufrufschnittstelle 67
4.2.5 Ausnahmebehandlung 70

5. Multithreading und Multiprozessorunterstützung **73**
5.1 Entwurf 73
5.1.1 Basisklassen 74
5.1.2 Externe Ressourcen 79
5.2 Implementierung 86

5.2.1 Synchronisation der virtuellen Maschine 87
5.2.2 Zweistufiger Methodencache 90
5.2.3 Blockierende externe Methoden 92
5.2.4 Erzeugen und Beenden von Threads 96
5.2.5 Start der virtuellen Maschine 99

6. Zusammenfassung **103**
6.1 Laufzeitverhalten 103
6.2 Bewertung und Ausblick 107

A. Beispiel: *Boss-Worker* **109**

B. Virtuelle Maschine: Befehlssatz und Datenstrukturen **119**
B.1 Registers 119
B.2 Stackframes 119
B.3 Bytecodes 120
B.4 Structures 125
B.4.1 Boxed Values 125
B.4.2 Method Objects 125
B.4.3 Exceptions 127
B.4.4 Special Objects 127
B.4.5 Synchronization Objects 129

Literaturverzeichnis **131**

1. Einleitung

Heutige, moderne Informationssysteme sind von drei wesentlichen Merkmalen gekennzeichnet: Ihrem Einsatz in heterogenen Netzwerkumgebungen, der Langlebigkeit ihrer Daten und der Fähigkeit, mehrere Klienten parallel zu bedienen. Ihre Architektur basiert, als Konsequenz des ersten Merkmals, auf einer verteilten Ausführung nach dem *Client-Server* Modell. Lassen sich Daten zwischen den verschiedenen, in einem solchen System verbundenen Rechnern noch mit Hilfe genormter Übertragungsprotokolle gemeinsam nutzen, so scheitert dies bei spezifisch für eine Plattform generiertem Programmcode an unterschiedlichen Betriebssystemen und Prozessorarchitekturen.

Eine Lösung dieser Problematik, die eine Verteilung und gemeinsame Nutzung von Programmen auch in heterogenen Netzen gestattet, liegt in der Erzeugung von Programmcode für eine virtuellen Maschine. Die virtuelle Maschine abstrahiert dabei von der zugrundeliegenden Rechnerarchitektur durch die Definition einer einheitlichen Daten- und Coderepräsentation und eines plattformunabhängigen Ausführungsmodells. Ein Programm interpretiert den Code und die Datenstrukturen und emuliert damit die virtuelle Maschine gegenüber der jeweiligen Zielplattform.

Als weitere Anforderung einer *Client-Server* Architektur müssen ein oder wenige Server in der Lage sein, mehrere Aufgaben zur gleichen Zeit zu bearbeiten. Ein Mittel zur Realisierung dieser Eigenschaft sind Threads, nebenläufige unabhängige Ausführungskontexte innerhalb eines Prozesses.

Die Datenbestände in Informationssystemen, z.B. Kundendaten oder Produktinformationen, zeichnen sich durch ihre Unabhängigkeit von den auf ihnen operierenden Applikationen aus, d.h. ihre Lebensdauer ist nicht auf einzelne Transaktionen beschränkt. Integrierte, persistente Programmiersysteme [DCBM89; MSS96] unterstützen diese Langlebigkeit durch die Verbindung von Programmiersprachen mit Datenbanktechnologie und schaffen so eine einheitliche Persistenzabstraktion für Code und Daten, im Gegensatz zu traditionellen Datenbanken, die eine Trennung von Code und Daten implizieren.

Tycoon-2, der kommerzielle Nachfolger des am Arbeitsbereich DBIS entstandenen Tycoon-Systems [Mat93], stellt mit seiner rein objektorientierten Programmiersprache TL-2 und seiner integrierten Programmierumgebung ein leistungsfähiges Werkzeug für die Entwicklung datenintensiver Softwaresysteme dar. Basierend auf einer virtuellen Maschine mit einem persistenten Objektspeicher verbindet Tycoon-2 orthogonale Persistenz und Plattformun-

abhängigkeit von Daten, Code und Threads.

1.1 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung einer auf die Hochsprache TL-2 abgestimmten virtuellen Maschine für das Tycoon-2 System. Zusammen mit dem parallel durchgeführten und in [Wie97] beschriebenen *Bootstrap* der Programmierumgebung soll der bisher existierende, auf dem alten Tycoon-System basierende Prototyp durch ein eigenständiges System abgelöst werden.

An das Tycoon-2 System werden eine Reihe von Anforderungen gestellt, die von der virtuellen Maschine zu erfüllen sind. Für diese gilt es Lösungen zu erarbeiten und zu implementieren. Notwendige Eigenschaften des Endproduktes und zu ihrer Realisierung gewählte Alternativen sind hierbei:

Persistenz: Alle Objekte des Systems sind langlebig, d.h. ihre Lebensdauer ist unabhängig von den eingesetzten Applikationen. Dieses Konzept unterstützt unmittelbar die Langlebigkeit von Daten in Informationssystemen und entbindet den Programmierer von der Pflicht, relevante Daten explizit zu sichern. Daten, Code und Threads werden in einem persistenten Objektspeichersystem gehalten und uniform behandelt.

Multithreading: Mehrere Threads, auch leichtgewichtige Prozesse (*lightweight process*) genannt, können nebenläufig innerhalb des Kontextes eines Prozesses ausgeführt werden. Die nebenläufige, auf Mehrprozessorarchitekturen auch parallele Ausführung von Threads ermöglicht eine effizientere Implementierung *Client-Server* basierter Anwendungen bei gleichzeitiger Steigerung von Durchsatz und Performanz. Hierzu bedarf es des Einsatzes betriebssystemeigener Threads.

Offenheit: Eine Anbindung externer Diensterbringer erweitert das Anwendungsspektrum. Über externe Bibliotheken und eine C-Aufrufschnittstelle kann zusätzliche Funktionalität integriert werden, ohne Eingriffe an der virtuellen Maschine vorzunehmen.

Portabilität: Die virtuelle Maschine kann ohne bzw. mit nur geringer Modifikation der Quelltexte auf anderen Rechnerarchitekturen übersetzt und ausgeführt werden. Die Kosten für die Entwicklung und Portierung der virtuellen Maschine werden durch eine konsequente Einhaltung von Standards bei der eingesetzten Programmiersprache (ANSI C) und der Betriebssystemschnittstelle (POSIX) minimiert. Eine hohe Portabilität erschließt den erstellten Applikationen eine breite Rechnerbasis.

Kompatibilität: Die auf dem existierenden Prototypen entwickelten Applikationen können ohne Änderung der Quelltexte auf das neue System übernommen werden. Die virtuelle Maschine implementiert hierfür die gleiche Grundfunktionalität wie der Prototyp.

Performanz: Eine im Vergleich zum Prototypen höhere Ausführungsgeschwindigkeit der virtuellen Maschine ist für einen Einsatz von Tycoon-2 in kommerziellen Projekten erforderlich. Die virtuelle Maschine wird in ihrem Befehlssatz und Ausführungsmodell an die objektorientierte Hochsprache TL-2 angepaßt.

Die Leistungsfähigkeit des Systems und die Herausforderung an den Entwurf und die Implementierung der virtuellen Maschine ergeben sich aus der Kombination der einzelnen Eigenschaften. So sollen z.B. die bereitgestellten Threads einerseits auf Betriebssystemthreads basieren, dabei aber gleichzeitig den Anforderungen an die Persistenz und Portabilität genügen.

1.2 Vorgehen

Ausgehend vom existierenden Prototypen ist die der Programmierumgebung bereitzustellende Funktionalität zu analysieren. Dabei muß auch die Sprache TL-2 auf ihre Konzepte und die sich hieraus ergebenden Rahmenbedingungen für den Entwurf der virtuellen Maschine untersucht werden. Anforderungen, die in diesem Zusammenhang für einen Einsatz im kommerziellen Umfeld wichtig sind, wie z.B. hohe Performanz und Portabilität, müssen zusätzlich berücksichtigt werden.

Von den gewonnenen Erkenntnissen ausgehend, sind die einzelnen Systemkomponenten zu identifizieren, und aus diesen das Kernsystem zu entwerfen und zu implementieren. Schwerpunkte bilden die Definition der virtuellen Maschine mit ihrer Schnittstelle zur Programmierumgebung sowie der Entwurf des Objektspeichersubsystems.

Aufbauend auf diesem Kernsystem sind das Multithreading und die persistenten Threads auf Basis betriebssystemeigener Threads zu realisieren.

In einer abschließenden Bewertung ist zu überprüfen, ob das entstandene System die eingangs gestellten Anforderungen erfüllt. Zusätzlich ist ein Vergleich mit dem Prototypen und anderen im gleichen Umfeld eingesetzten Programmiersprachen bzw. -systemen hinsichtlich ihrer Performanz und Eigenschaften erforderlich.

1.3 Gliederung der Arbeit

In Kapitel 2 werden die Anforderungen an die virtuelle Maschine analysiert: die gegenüber der Programmierumgebung und der Hochsprache TL-2 zu erbringende Funktionalität sowie weitere durch die Systemeigenschaften bedingte Vorgaben.

Die Systemarchitektur und der Entwurf der virtuellen Maschine werden in Kapitel 3 vorgestellt. Der Schwerpunkt liegt auf den beiden Hauptkomponenten, dem Objektspeicher und dem Bytecodeinterpreter.

Kapitel 4 beschreibt die Implementierung ausgewählter Systemteile. Die Schwerpunkte liegen auf den Allokationsmechanismen und der Speicherbereinigung des Objektspeichers sowie der Methodensuche der virtuellen Maschine.

Der Entwurf und die Implementierung des Multithreadings und der persistenten Threads werden separat in Kapitel 5 erläutert.

Eine Zusammenfassung und Bewertung des entstandenen Systems und ein Ausblick auf die weitere Entwicklung in Kapitel 6 beenden diese Arbeit.

2. Anforderungen

In diesem Kapitel werden die Anforderungen an die virtuelle Maschine beschrieben, auf denen in den weiteren Kapiteln aufgebaut wird. Die virtuelle Maschine definiert einerseits die Schnittstelle zu einer abstrakten Zielmaschine und ermöglicht so die vollständige Plattformunabhängigkeit der gesamten Programmierumgebung und der entwickelten Applikationen. Auf der anderen Seite wird der Begriff der virtuellen Maschine auch synonym für das implementierte Programm benutzt, das die Spezifikation durch eine Emulation erfüllt und so die Mittlerrolle zwischen Programmierumgebung und zugrundeliegender Rechnerarchitektur einnimmt.

Die Anforderungen werden im wesentlichen durch die Kernkonzepte der Hochsprache TL-2 (Abschnitt 2.1) und die Systemeigenschaften von Tycoon-2 (Abschnitt 2.2) bestimmt. Naturgemäß kann hier keine vollständige Einführung in die Sprache gegeben werden; ausführliche Informationen finden sich in [Wah98; Wie97].

2.1 Spracheigenschaften

Die Hochsprache TL-2 folgt dem Paradigma der reinen, klassenbasierten Objektorientierung. Das zugrundeliegende Objektmodell sieht Objekte als abstrakte Datentypen an, die sowohl Zustand als auch Verhalten kapseln. Objekte und zwischen ihnen ausgetauschte Nachrichten bilden die einzige Grundlage der Programmierung. Abbildung 2.1 illustriert das hinter TL-2 stehende Konzept.

TL-2 basiert auf einer Referenzsemantik, die auf der eindeutigen Identität jedes Objekts beruht. Jedes Objekt ist Exemplar einer Klasse, die die einzelnen Exemplarvariablen und das Verhalten beim Empfang einer Nachricht definiert. Eine Nachricht besteht aus einem Selektor und weiteren Argumenten. Die Klasse des Empfängers bildet einen Selektor auf eine Methode ab. Für die Methodensuche ist allein die Klasse des Empfängers ausschlaggebend. Beim Empfang einer Nachricht wird die zum Selektor gehörende Methode mit einem Verweis auf das Empfängerobjekt und den Argumenten evaluiert und ihr Ergebnis an den Aufrufer zurückgegeben.

Auch die Abfrage und die Manipulation des Objektzustandes erfolgen über Nachrichten. Jede Exemplarvariable besitzt hierfür eine implizite Lese- und Schreibmethode.

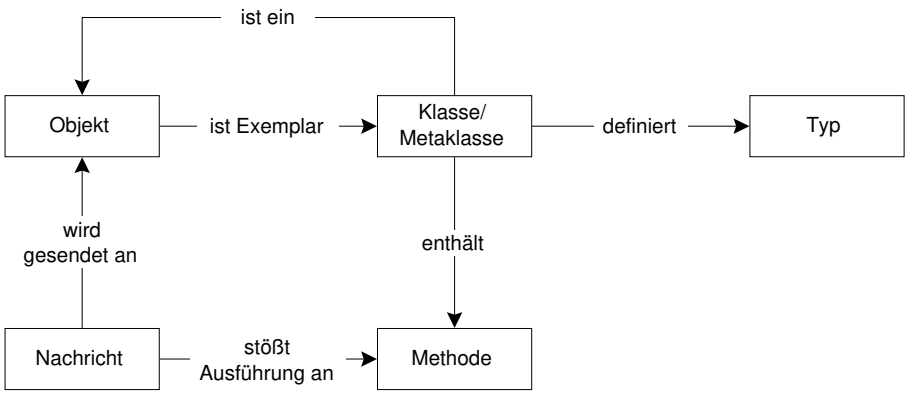

Abbildung 2.1: Konzeption der Sprache TL-2

Methoden können sowohl öffentlich als auch privat deklariert werden. Bei privaten Methoden ist der Zugriff nur vom gleichen Objekt aus möglich, d.h. das Subjekt selbst (*self*) ist der einzige zulässige Empfänger eines privaten Methodenaufrufs.

Das folgende Beispiel veranschaulicht den Aufbau einer TL-2 Klasse.

```
class Counter
super Object                    ;;  Superklasse
metaclass SimpleConcreteClass ;;  Metaklasse

public methods

increment() :Void       ;;  öffentliche Methode
{
  _count := _count + 1  ;;  Exemplarvariable erhöhen
}

private

_count :Int             ;;  private Exemplarvariable
```

Die Definition einer Klasse beginnt mit dem Schlüsselwort `class` gefolgt vom Klassennamen, im Beispiel `Counter`. Weitere, optionale Angaben beinhalten eine oder mehrere Vaterklassen und eine Metaklasse. Öffentliche und private Definitionen werden mit den Schlüsselworten `public` bzw. `private` eingeleitet, bei Methoden mit dem zusätzlichen Schlüsselwort `methods`. Die Beispielklasse besitzt eine öffentliche Methode (`increment`) und eine private Exemplarvariable (`_count`).

Als Folge der reinen Objektorientierung existieren keine primitiven Datentypen; auch einfache Typen wie Ganzzahlen und Zeichenketten sind gleichberechtigte Objekte erster Klasse. Alle Berechnungen werden über das Senden von Nachrichten an Objekte ausgedrückt:

- Einfache Kontrollstrukturen, wie Bedingungen (*if*), Schleifen (*for, while*) und die Ausnahmebehandlung sind nicht Bestandteil der Sprache, sondern durch Rekursion und Funktionen höherer Ordnung implementiert.

  Die beiden folgenden Methoden zeigen die Definition der `while` bzw. `until` Schleife.

```
while(cond :Fun0(Bool), statement :Fun0(Void))
{
  cond[] ? {statement[], while(cond, statement)}
}

until(cond :Fun0(Bool), statement :Fun0(Void))
{
  statement[], !cond[] ? {until(cond, statement)}
}
```

  Die Bedingung und der Schleifenkörper bestehen jeweils aus einer Funktion höherer Ordnung, die an die Methoden übergeben werden. Im Fall der `while` Schleife wird die Bedingung evaluiert ("[ ]") und, wenn diese erfüllt ist, der Schleifenkörper ausgeführt. Im Anschluß wird eine weitere Iteration durch einen rekursiven Aufruf der Methode gestartet.

- Funktionen höherer Ordnung mit statischem Sichtbarkeitsbereich sind Objekte erster Klasse, die eine Evaluationsnachricht verstehen (vgl. im vorigen Beispiel).

- Alle primitiven Operationen wie die Ganzzahlarithmetik, der Zugriff auf Exemplarvariablen und die Feldindizierung werden ebenfalls über Nachrichten abgewickelt.

```
class Int
super Integer(Int)
metaclass IntClass

public methods

...

"+"(x :Int) :Int
"-"(x :Int) :Int
"*"(x :Int) :Int
"/"(x :Int) :Int

...
```

  Das Beispiel zeigt die Definition der vier Grundrechenarten in der Klasse `Int`. Die Methoden besitzen keinen TL-2 Rumpf, die nötige Funktionalität muß von der virtuellen Maschine bereitgestellt werden.

- Auch Klassen sind wiederum Exemplare einer Metaklasse und können somit Nachrichten empfangen, z.B. zur Erzeugung und Initialisierung ihrer Exemplare.

- Ausnahmen erlauben es, zur Laufzeit auftretende Fehlersituationen abzufangen und gezielt zu bearbeiten.

```
verboseOpen(path :String) :File
{
  try({
    File.openRead(path)
  },
  fun(e:Exception) {
    errorLog.error("Could not open file: " + e.printString),
    nil
  })
}
```

  Die in der Klasse `Objekt` definierte Methode `try` erwartet, wie im Beispiel zu sehen, zwei Funktionen höherer Ordnung als Argumente. Die erste Funktion wird von der Methode evaluiert. Tritt hierbei eine Ausnahme auf, so wird die zur Ausnahmebehandlung übergebene zweite Funktion ausgeführt. Dieser Mechanismus muß von der virtuellen Maschine unterstützt werden..

TL-2 verwendet für die Mehrfachvererbung die aus Loops [BS81; BCH+96] bekannte Form der Klassenpräzedenzliste (*class precedence list*). Die Linearisierung des Klassengraphen, bei der jede Vorfahrenklasse hinter den von ihr abgeleiteten Klassen eingeordnet wird, ermöglicht es, Methodenimplementierungen durch sequentielles Durchsuchen der Klassenpräzedenzliste zu finden. Bei einem normalen Methodenaufruf wird vom Beginn der Liste an gesucht, beim Aufruf einer Methode einer Vorfahrenklasse (*super send*) ab der letzten Fundstelle. Die Mehrfachvererbung ist für die virtuelle Maschine aufgrund der Linearisierung der Klassenhierarchie in der Klassenpräzedenzliste transparent. Ein Fehlschlag der Suche, wenn also keine zum Selektor passende Methode gefunden werden konnte, stellt eine Ausnahmesituation dar.

Das Typsystem von TL-2 mit struktureller Subtypisierung und begrenztem parametrischem Polymorphismus (*f-bounded parametric polymorphism*) [Ern98] ist auf den statischen Typprüfer der Programmierumgebung beschränkt. Der TL-2 Übersetzer erzeugt auch ohne Typprüfung lauffähigen Code. Es ist nicht die Aufgabe der virtuelle Maschine, dynamische Überprüfungen zur Laufzeit durchzuführen oder zwischen öffentlichen und privaten Methoden zu unterscheiden.

2.2 Systemeigenschaften

Zwei wesentliche Systemeigenschaften kennzeichnen Tycoon-2: die orthogonale Persistenz und die Unterstützung nebenläufiger Threads. Weitere Anforderungen, die beachtet werden müssen, sind seine Portabilität, Kompatibilität und Erweiterbarkeit.

2.2.1 Persistenz

Informationssysteme, d.h. Systeme zur Unterstützung und Koordination von kooperativer Arbeit, zeichnen sich durch ihre Langlebigkeit aus. Die Persistenz erstreckt sich bei ihnen nicht nur auf den Datenbestand selbst, sondern auch auf Programme und langlebige Aktivitäten, z.B. Geschäftsprozesse. Eine integrierte, persistente Programmierumgebung für die Entwicklung von Informationssystemen muß diese Anforderungen durch die Bereitstellung eines universellen Objektspeicherssystems für Daten, Code und Aktivitäten erfüllen.

Wie die Erfahrungen in [MMS97] gezeigt haben, ist eine automatische Speicherbereinigung (*garbage collection*) aufgrund der Komplexität der Applikationen und des hohen Bestands an Massendaten ein wichtiger Faktor für die Stabilität des Systems. Eine automatische Deallokation entbindet den Programmierer von der expliziten Verwaltung allozierter Objekte und verhindert so Speicherlecks (*memory leaks*) und Zugriffe auf bereits gelöschte Objekte (*dangling references*).

2.2.2 Multithreading

Client-Server Architekturen erfordern auf *Server*-Seite die parallele Ausführung mehrerer Aktiväten unter Ausnutzung der vorhandenen Systemressourcen. Die Vorteile, die sich nach [NM97] aus der Nutzung von Threads ergeben, sind:

Höherer Durchsatz: Eine Applikation kann mit mehreren nebenläufigen oder parallelen Threads mehr Arbeit pro Zeiteinheit verrichten. Beispiele hierfür sind die parallele Ausführung von Threads auf Mehrprozessorsystemen oder die Nutzung synchroner Ein/Ausgabe Operationen ohne Blockierung der gesamten Applikation.

Bessere Reaktionszeiten: Durch die Erzeugung eines eigenen Thread für jede von der Applikation durchgeführte Aufgabe kann unmittelbar auf Benutzeraktionen reagiert und die Interaktion verbessert werden.

Einsparung von Ressourcen: Alle Threads eines Prozesses teilen sich einen gemeinsamen Adressraum. Sie benutzen gemeinsam globale Datenstrukturen und Systemressourcen.

Schnellere Operationen: Der Kontext eines Thread enthält im Gegensatz zu einem Prozeß nur wenige Systemressourcen. Als Folge dieser Leichtgewichtigkeit ist die Verwaltung von Threads – Konstruktion, Destruktion und Kontextwechsel – im Vergleich zum Prozeßmanagement mit weniger Aufwand verbunden.

Natürlicher Programmierstil: Threads ermöglichen es einem Programmierer, in einfacher Weise Nebenläufigkeit und Parallelität in Applikationen auszudrücken.

2.2.3 Offenheit

Ein offenes System verfügt über zwei wichtige Merkmale: hohe Skalierbarkeit und Erweiterbarkeit. Kernpunkte für ihre Realisierung sind die Integration neuer Funktionalität, ohne Änderungen am System selbst vornehmen zu müssen, und die Anbindung externer Dienstleister.

Externe Dienste bieten ihre Leistungen über klar definierte Schnittstellen an, i.d.R. durch statische oder dynamische Bibliotheken mit einem C-API (*Application Program Interface*); ein Weg, der auch für die Erweiterung der Systemfunktionalität genutzt werden kann. Eine generische Schnittstelle für externe Funktionen stellt somit die Schlüsselkomponente für die Offenheit des Systems dar.

2.2.4 Portabilität

Eine hohe Portabilität senkt nicht nur die Entwicklungskosten und fördert die Verbreitung, für datenintensive Anwendungen tritt ein weiterer Aspekt in den Vordergrund: ihre Langlebigkeit ist direkt an die Langlebigkeit des Implementationssystems gekoppelt. So wird die Portabilität der virtuellen Maschine bei der fortschreitenden Entwicklung von Hardware und Betriebssystemen zu einem entscheidenden Faktor für die Anpassungsfähigkeit des Gesamtsystems.

Die Erfüllung dieser Anforderung setzt eine konsequente Einhaltung der durch ANSI C und POSIX.1 definierten Standards bei der gewählten Implementationssprache bzw. der Schnittstelle zu den Betriebssystemdiensten voraus.

2.2.5 Kompatibilität mit dem Prototypen

Tycoon-2 ist ein reflektives System. Die gesamte Programmierumgebung, einschließlich des TL-2 Compilers und der Klassenverwaltung, ist in der Hochsprache selbst implementiert und allen anderen Applikationen direkt zugänglich. Die Entwicklung der virtuellen Maschine stellt daher nur die eine Hälfte der für die Realisierung eines eigenständigen Tycoon-2 Systems notwendigen Arbeit dar. Die andere Hälfte, die Gegenstand der Arbeit von [Wie97] ist, besteht aus der Implementierung der Entwicklungsumgebung für TL-2 in TL-2. Ausgangspunkt ist eine auf dem Tycoon-System in der Sprache TL vorliegende prototypische Implementierung der Programmierumgebung. Der Übergang auf die in dieser Arbeit beschriebene Zielmaschine erfolgt mittels eines *Bootstrap*.

Die neue Programmierumgebung ist in der Lage, sich selbst zu übersetzen und ein neues Tycoon-2 System zu generieren. Diese Fähigkeit ist nach der Lösung vom Prototypen aufgrund der engen Kopplung mit der neuen virtuellen Maschine zwingend notwendig, falls kritische von der virtuellen Maschine benutzte Systemklassen geändert werden. Gleichzeitig ist dieser komplexe Vorgang ein geeigneter Test für die Korrektheit und Stabilität sowohl

der Programmierumgebung und der Bibliotheken der implementierten Sprache als auch der virtuellen Maschine.

Als wesentliche Anforderung des *Bootstrap* stellt sich die Notwendigkeit dar, TL-2 Quelltexte unverändert in das neue System zu übernehmen. Die virtuelle Maschine muß hierzu die bereits vom Prototypen bereitgestellte Basisfunktionalität implementieren. Hierzu zählen u.a.:

- Allokation von Objekten
- Zugriff auf Exemplarvariablen
- Feldindizierungen
- Arithmetische und logische Operationen auf Ganzzahlen

Auch die interaktive Entwicklungsumgebung (*toplevel*) muß unterstützt werden. So werden z.B. neue Klassen dynamisch zur Laufzeit des Systems übersetzt und eingebunden. Objekte, die systemweit zur Verfügung stehen müssen, werden in einem global zugreifbaren *Pool* abgelegt.

2.2.6 Performanz

Die Basis des Tycoon-2 Prototypen ist das Tycoon-System. Dessen virtuelle Maschine weist zwei Punkte auf, die sich negativ auf die Performanz des Prototypen auswirken. Zum einen ist der Interpreter der virtuellen Maschine für die Programmiersprache TL mit ihren imperativen und funktionalen Konstrukten optimiert. Der Prototyp integriert nur zwei zusätzliche Befehle für das Senden von Nachrichten. Der zweite Punkt ist die Anbindung an den Objektspeicher, die über eine abstrakte Aufrufschnittstelle erfolgt.

Eine insbesondere für den Einsatz in kommerziellen Informationssystemen wichtige Steigerung der Performanz setzt an drei Punkten an:

- Die virtuelle Maschine wird auf die reine Objektorientierung von TL-2 optimiert, u.a. durch eingebaute Methoden und spezielle Nachrichtenbefehle mit festen Selektoren (vgl. [GR83]).
- Die Kommunikation und der Datenaustausch zwischen der TL-2 Programmierumgebung und der virtuellen Maschine erfolgt über gemeinsam genutzte Datenstrukturen.
- Der Objektspeicher wird in die virtuelle Maschine integriert. Objektverweise werden direkt durch Adressen repräsentiert und die Schnittstelle beim Zugriff auf Objektdaten entfällt.

3. Entwurf

In diesem Kapitel wird der Entwurf der virtuellen Maschine vorgestellt. Er ist gekennzeichnet durch eine starke Verzahnung der einzelnen Komponenten sowie die enge Bindung an und Abstimmung auf TL-2. Dies äußert sich in einer breiten Schnittstelle zur Programmierumgebung zur Vermeidung von Engpässen, die sich negativ auf die Performanz auswirken können (*performance bottlenecks*). Ziele sind u.a. die gemeinsame Nutzung von Datenstrukturen zwischen der Programmierumgebung und der virtuellen Maschine und eine direkte Umsetzung von Speicheradressen in Objektidentifikatoren innerhalb der Maschine. Wichtigster Leitsatz ist die Entwicklung eines Kernsystems hoher Effizienz.

Nach einem Überblick über die Systemarchitektur (Abschnitt 3.1) werden der Entwurf des Objektspeichers (Abschnitt 3.2) und des Interpreters (Abschnitt 3.3) vorgestellt.

3.1 Systemarchitektur

Die Architektur von Tycoon-2 ist in Abbildung 3.1 illustriert. Sie besteht aus drei Schichten, über die im folgenden ein kurzer Überblick gegeben wird.

3.1.1 Applikationsschicht

Die oberste Schicht des Systems stellen die TL-2 Applikationen dar. Hierzu gehören neben den entwickelten Anwendungsprogrammen auch die komplette in TL-2 implementierte Entwicklungsumgebung mit:

- der Klassenverwaltung,
- dem TL-2 Compiler,
- dem statischen Typprüfer und
- dem interaktivem *Toplevel*.

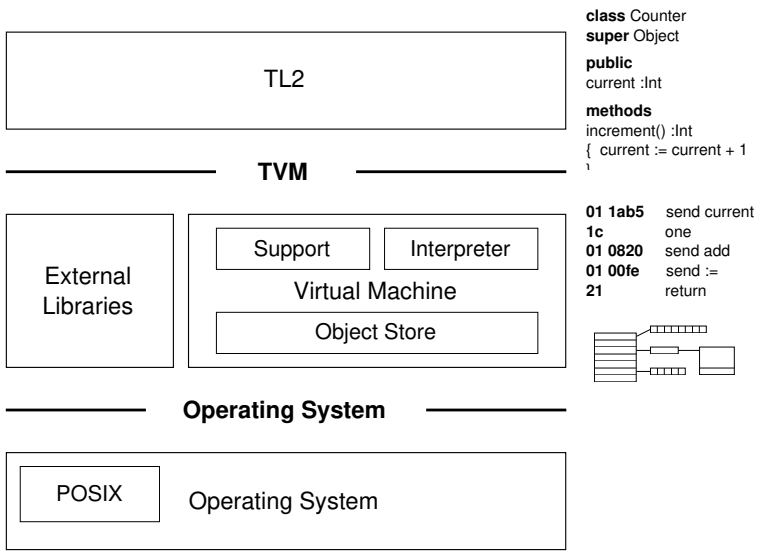

Abbildung 3.1: Tycoon-2 Systemarchitektur

3.1.2 Virtuelle Maschine und Objektspeicher

Die virtuelle Maschine nimmt die Mittlerfunktion zwischen der Applikationsebene und dem Betriebssystem ein. Ihre wichtigsten Komponenten sind der Interpreter, der die definierte virtuelle Zielmaschine emuliert, und der Objektspeicher. Anders als im Prototypen ist der Objektspeicher ein fest integrierter Bestandteil der virtuellen Maschine und keine separate, austauschbare Komponente [MMS97]. Die Laufzeitunterstützung sorgt u.a. für das Multithreading sowie die Bindung externer Bibliotheken und den Aufruf der von ihnen exportierten C-Funktionen.

Der Objektspeicher

Der Objektspeicher realisiert die Persistenz des Tycoon-2 Systems. Zu seinen Aufgaben gehören neben der Allokation beliebiger Objekttypen auch die Sicherung und Wiederherstellung von Objektspeicherzuständen sowie die Durchführung einer automatischen Speicherbereinigung.

In Verbindung mit mehreren Threads und der C-Aufrufschnittstelle muß die Speicherbereinigung in der Lage sein, die Integrität von außerhalb des Objektspeichers referenzierten Objekten auch während der Speicherfreigabe zu gewährleisten. Der in [Bar88] beschriebene *Garbage Collector* verfügt über die hiefür nötigen Mechanismen und wird daher als Grundlage für den weiteren Entwurf eingesetzt.

Der Interpreter

Da eine Registermaschine bei einer rein interpretierten Ausführung keine Vorteile gegenüber einer einfachen Kellermaschine besitzt [Pak96], wird nach dem Vorbild von Smalltalk 80 [GR83] und des Tycoon-Systems [Wei96] eine Kellerarchitektur eingesetzt. Ihr Befehlssatz weist folgende Klassen von Instruktionen auf:

- Nachrichtenbefehle für den Aufruf von an Objekte gebundene Methodenimplementierungen, zur Realisierung des hochsprachlichen Kernkonzeptes.
- Zugriffsfunktionen für das Lesen und Schreiben der Speicherzellen des Kellers.
- Operationen zur Unterstützung von Funktionen höherer Ordnung.
- Instruktionen zur Implementierung von Kontrollstrukturen. Diese Befehlsklasse ist aufgrund der reinen Objektorientierung von TL-2 optional [GM95a], für Optimierungen während der Codeerzeugung jedoch wünschenswert.

Die weitere Basisfunktionalität wird, da alle Berechnungen über das Senden von Nachrichten ausgedrückt werden, in Form von in den Interpreter eingebauten Methoden bereitgestellt (vgl. [GR83]). Hierzu zählen:

- Arithmetische und logische Operationen auf 32 Bit breiten Ganzzahlen.
- Allokation von Objekten.
- Feldindizierungen.
- Zugriff auf Exemplarvariablen.
- Auslösen von Ausnahmen.
- Thread- und Synchronisationsprimitive.

Alle eingebauten Methoden führen wenn möglich Laufzeitüberprüfungen durch, da eine statische Typüberprüfung des generierten Codes nicht vorausgesetzt werden kann. Beispiele sind u.a. die Überprüfung der Klasse übergebener Argumente oder der Bereichsgrenzen bei der Indizierung von Feldern.

Externe Bibliotheken

Die ebenfalls in der Schicht der virtuellen Maschine angesiedelten externen Bibliotheken enthalten zusätzliche Funktionalität, die nicht ständig benötigt wird. Durch ihre Auslagerung kann die virtuelle Maschine auf ein Kernsystem reduziert werden. Ein Beispiel ist die

RTS-Bibliothek (*runtime support*), die Arithmetik für 64 Bit breite Ganz- und Gleitkommazahlen und *Wrapper* für verschiedene Betriebssystemfunktionen beinhaltet. Auch die von externen Diensten bereitgestellten Bibliotheken wie SQL-Gateways gehören in diese Kategorie. Der Zugang zu diesen Bibliotheken ist von TL-2 aus nur über die C-Schnittstelle der virtuellen Maschine möglich.

3.1.3 Betriebssystem

Die grundlegenden Dienste werden durch das Betriebssystem bereitgestellt. Von der virtuellen Maschine werden u.a. benötigt:

- Ein/Ausgabeoperationen (`stdio.h`).
- Threads (`pthread.h`).
- Signalbehandlung (`signal.h`).
- Dynamische Bibliotheken (`dl.h`).
- Speichermanagement (`memory.h`).
- Kontextmanipulation (`setjmp.h`).

Die Anbindung erfolgt über POSIX konforme C-Schnittstellendateien und Standardbibliotheken.

3.2 Objektspeicher

Die unterste Schicht der in dieser Arbeit vorgestellten virtuellen Maschine bildet der Objektspeicher. Er ermöglicht über seine Schnittstelle die Allokation und den Zugriff auf Objekte, er verfügt über eine automatische Speicherbereinigung und ist zudem in der Lage, ein persistentes Abbild auf einem nichtflüchtigen Speichermedium zu sichern und wiederherzustellen.

3.2.1 Aufbau

Der Objektspeicher besteht aus einem oder mehreren, nicht notwendigerweise aufeinanderfolgenden Hauptspeicherblöcken, die in einzelne Speicherseiten fester Größe unterteilt sind. Jede Speicherseite wird über einen eigenen Seitendeskriptor, der ihren Zustand beschreibt, verwaltet. Diese Organisation wird durch die grundlegende Arbeitsweise der Speicherbereinigung, die auf dem in [Bar88] beschriebenen Algorithmus basiert, vorgegeben. Das Modell des Objektspeichers ist in Abbildung 3.2 dargestellt.

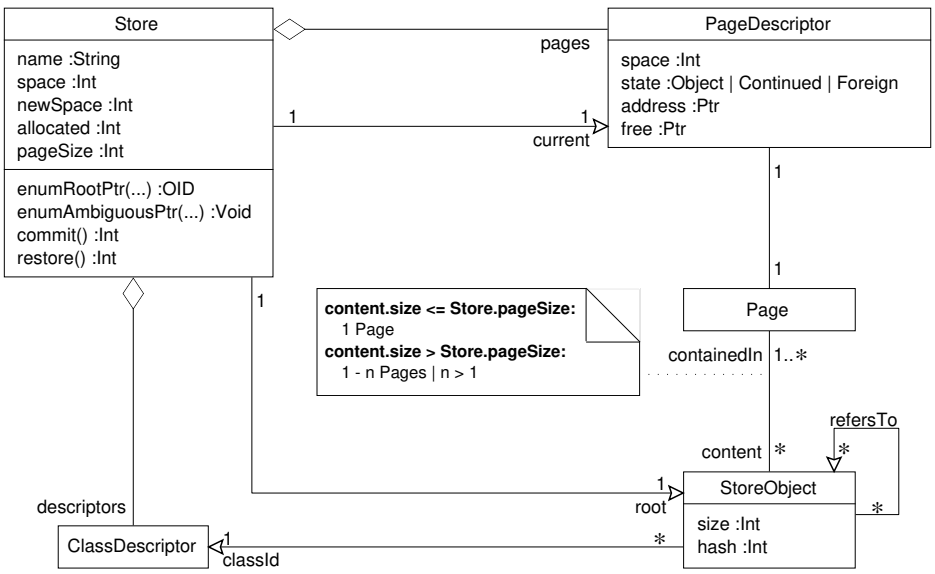

Abbildung 3.2: Modell des Objektspeichers

Neben der Speicherbereinigung (Abschnitt 3.2.7) integriert der Entwurf zahlreiche weitere Leistungen, u.a. die dynamische Nachforderung von Hauptspeicher und die Verwaltung großer Objekte (Abschnitt 3.2.3), die Unterstützung verschiedener Objektarten einschließlich C-Strukturen und schwacher Referenzen (Abschnitte 3.2.5 und 3.2.8) sowie die persistente Sicherung des Objektspeicherzustands (Abschnitt 3.2.9). Situationen, die bei einem Mangel an Ressourcen auftreten, werden in den Abschnitten zur Speicherbereinigung und Persistenz behandelt.

3.2.2 Globale Verwaltungsinformationen

Der Zustand des Objektspeichers wird von einer globalen Datenstruktur (`Store`) beschrieben:

`name`: Dateiname des mit dem Objektspeicher assoziierten persistenten Abbildes.

`root`: Wurzelobjekt des persistenten Objektgraphen.

`allocated`: Anzahl der belegten Speicherseiten.

`pageSize`: Größe einer Speicherseite.

`space, newSpace`: Identifikatoren, über die jede Speicherseite einem logischen Bereich zugeordnet werden kann (siehe Abschnitte 3.2.3 und 3.2.7). Im normalen Betrieb besitzen beide denselben Wert und kennzeichnen eine Seite als frei oder belegt. Während der

Speicherbereinigung unterscheiden sich ihre Werte und ordnen belegte Seiten einem Teilbereich des Objektspeichers zu (*from space* oder *to space*).

`pages`: Tabelle der Seitendeskriptoren (siehe Abschnitt 3.2.3).

`descriptors`: Tabelle der Klassendeskriptoren.

`current`: Aktuelle Speicherseite für die Allokation neuer Objekte.

`enumRootPtr()`: Enumerationsfunktion für zusätzliche Wurzelobjekte.

`enumAmbiguousPtr()`: Enumerationsfunktion für Speicherseiten.

`commit()`: Funktion zur persistenten Sicherung des Objektspeicherzustands.

`restore()`: Wiederherstellung des Objektspeicherzustands aus einem persistenten Abbild.

3.2.3 Speicherseiten

Der vom Objektspeicher allozierte Hauptspeicher ist in Speicherseiten gleicher Größe unterteilt, von denen jede über einen eigenen Deskriptor zur Zustandsbeschreibung verfügt (Abbildung 3.3). Die Seitengröße ist unabhängig von der Speicherverwaltung des Betriebssystems oder der Hardware. Ein Seitendeskriptor verwaltet folgende Informationen:

`state`: Typ der Speicherseite: `Object`, `Continued` oder `Foreign`.

`address`: Hauptspeicheradresse der Seite.

`free`: Freizeiger innerhalb der Seite.

`space`: Ordnet Speicherseiten einem der zwei Teilbereiche des Objektspeichers (*from space* oder *to space*) zu, wenn der Wert dem globalen Identifikator entspricht (siehe Abschnitt 3.2.7). Alle anderen Werte kennzeichnen eine Seite als frei, sofern ihr Typ nicht `Foreign` ist.

Objekte, die die Seitengröße überschreiten, werden in aufeinanderfolgenden Seiten alloziert. Die erste dieser Speicherseiten wird, wie alle gewöhnlich allozierten Seiten, mit dem Typ `Object` initialisiert. Alle folgenden Seiten erhalten den Typ `Continued`, was sie als Fortsetzung der ersten Seite markiert. In einer so allozierten Sequenz von Seiten befindet sich immer nur ein Objekt.

Der Objektspeicher benutzt keinen statisch zugewiesenen Hauptspeicher, sondern kann bei Bedarf zusätzliche Speicherblöcke vom Betriebssystem nachfordern. Da neu allozierte Speicherbereiche sich nicht direkt an den bestehenden Speicher anschließen müssen, kann es zur Fragmentierung kommen. Die nicht unter Kontrolle des Objektspeichers stehenden Seiten werden in ihren Deskriptoren als `Foreign` markiert.

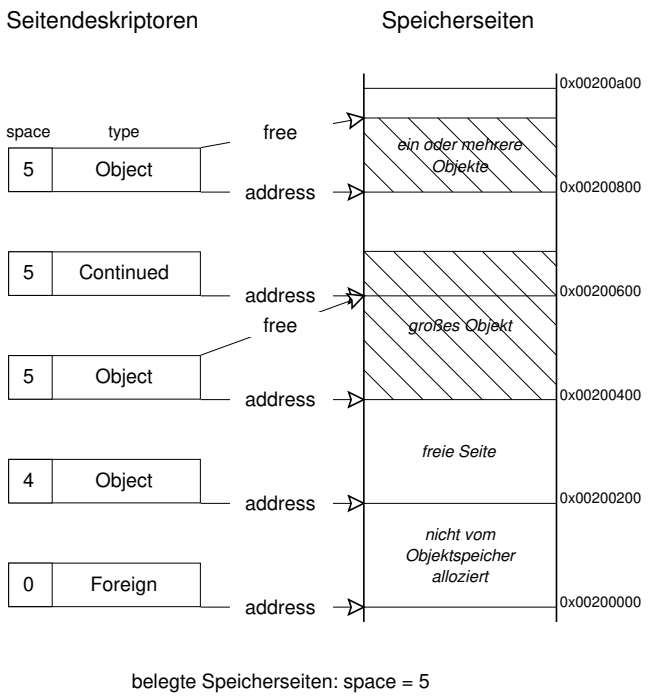

Abbildung 3.3: Seitendeskriptoren des Objektspeichers

3.2.4 Objekte

Jedes im Objektspeicher gehaltene Objekte besitzt außer seinem Datenbereich zusätzliche Verwaltungsinformationen:

`size`: Die Länge des Datenbereichs in Byte.

`hash`: Hashwert, der in TL-2 für den Aufbau von Objektsammlungen (*Collections*) benutzt wird.

`classId`: Klassennummer des Objekts. Über diese Nummer ist jedes Objekt einem Klassendeskriptor (Abschnitt 3.2.5) und mittels der im Wurzelobjekt verankerten Klassentabelle einem Klassenobjekt (Abschnitt 3.3.2) zugeordnet.

Der Zugriff auf die Verwaltungsinformationen und die Objektallokation erfolgt über die in Tabelle 3.1 angegebene Aufrufschnittstelle. Die Größe von Objekten wird statisch bei der Allokation angegeben und ist nicht dynamisch änderbar. Der Datenbereich neu angelegter Objekte wird mit 0 initialisiert.

Neue Klasse registrieren
ClassId newClassId(Object wLayout, CType * pszDescriptor);
Neues Objekt allozieren
OID newArray(ClassId classId, Word nElements);
OID newByteArray(ClassId classId, Word nElements);
OID newShortArray(ClassId classId, Word nElements);
OID newIntArray(ClassId classId, Word nElements);
OID newLongArray(ClassId classId, Word nElements);
OID newStack(ClassId classId, Word nElements);
OID newThread(ClassId classId);
OID newStruct(ClassId classId);
OID newWeakRef(ClassId classId);
OID newCopy(OID pObjekt);
Verwaltungsinformationen eines Objekts erfragen und setzen
Word size(OID pObjekt);
ClassId classId(OID pObjekt);
Word hash(OID pObjekt);
void setHash(OID pObjekt, Word wHash);
Wurzelobjekt erfragen und setzen
OID root(void);
void setRoot(OID pRoot);
Schwache Referenzen initialisieren und finalisieren
WeakRef * initWeakRef(WeakRef * pWeakRef);
WeakRef * scheduleWeakRefs(void);
void setFinalizer(Finalizer finalizer);
Enumeratorfunktionen setzen
void setEnumRootPtr(EnumRootPtr enumRootPtr);
void setEnumAmbiguousRootPtr(EnumRootPtr enumRootPtr);
Speicherbereinigung auslösen
void gc(Bool fCompact);
Objektspeicherzustand sichern und restaurieren
Int commit(void);
Int restore(String szName);

Tabelle 3.1: Aufrufschnittstelle des Objektspeichers

Objekte können 64 Bit breite Werte enthalten. Daher ist es aufgrund der Ausrichtung der Speicherzugriffe existierender Prozessorarchitekturen notwendig, den Datenbereich eines Objekts ebenfalls an 64 Bit Grenzen auszurichten. Je nach Länge der Daten können hierdurch bis zu 7 Bytes zwischen zwei Objekten ungenutzt bleiben. Der Datenbereich selbst wird in Abschnitt 3.2.5 behandelt.

Jedes Objekt besitzt einen eindeutigen Identifikator, die `OID` (*object identifier*), über den die Referenzierung und Bindung an andere Objekte erfolgt: die Hauptspeicheradresse seines Datenbereichs. Die `OID` eines Objekts und somit auch bestehende Objektverweise können

sich zur Laufzeit durch die Arbeit der Speicherbereinigung ändern. Es wird vorausgesetzt, daß Adressen und somit die `OID`s auf den jeweiligen Zielmaschinen eine Breite von 32 Bit aufweisen. Hierdurch wird gleichzeitig die maximale Größe des Objektspeichers auf 4 Gigabyte beschränkt. Der Zugriff und die Manipulation der Objektdaten erfolgt direkt durch lesende bzw. schreibende indizierte Hauptspeicherzugriffe auf den Datenbereich. Es existiert kein Schutzmechanismus, der fehlerhafte Zugriffe, z.B. das Überschreiben eines folgenden Objekts, unterbindet.

Eine Besonderheit stellen `OID`s dar, die ein Objekt direkt kodieren. Die Speicherausrichtung der Objekte erlaubt es, die letzten beiden Bits einer `OID`, die im Fall eines Verweises immer auf 0 gesetzt sind, zur Markierung (*tagging*) zu benutzen. Das Markierungsschema ist in Abbildung 3.4 illustriert. Ein spezielles Objekt stellt der ungültige Verweis auf die Speicheradresse 0 dar, der als `nil`-Objekt interpretiert wird. Zusätzlich zu Objektverweisen werden vorzeichenbehaftete Ganzzahlen (siehe auch Abschnitt 3.3.2) mit einer Breite von 30 Bit repräsentiert.

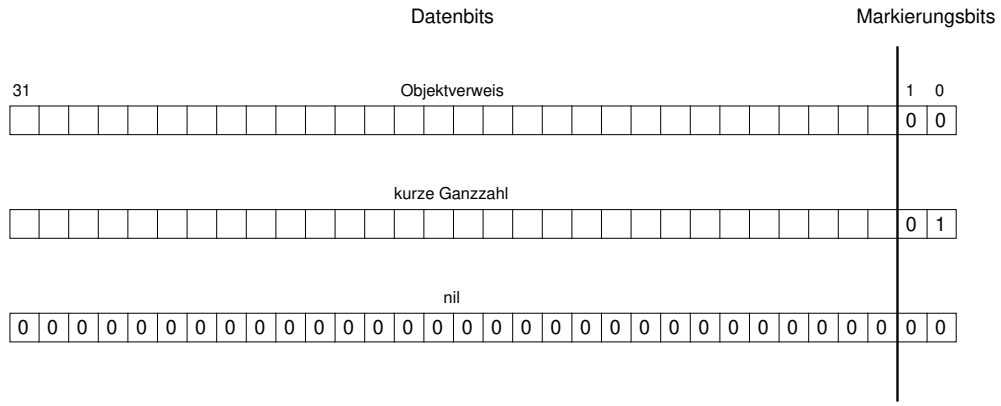

Abbildung 3.4: Markierungsschema für `OID`s

3.2.5 Klassendeskriptoren

Analog zu der vom Interpreter benutzten Klassentabelle (siehe Abschnitt 3.3.2) verwaltet der Objektspeicher eine Deskriptortabelle, die die Struktur des Datenbereichs von Objekten einer Klasse beschreibt (Abbildung 3.5). Die Tabelle wird von der Speicherbereinigung für die Suche nach Objektverweisen benötigt. Insgesammt werden 9 Layoutarten vom Objektspeicher unterschieden:

- Feld von Objektverweisen (`OID`)
- Feld von Bytes (8 Bit)

- Feld von kurzen Ganzzahlen (16 Bit)
- Feld von Ganzzahlen (32 Bit)
- Feld von langen Ganzzahlen (64 Bit)
- Leichtgewichtiger Prozeß (*Thread*)
- Laufzeitstapel des Interpreters (*Stack*)
- Schwache Referenz
- C-Strukturobjekt

Objekte der Typen *Thread* und *Stack* bedürfen einer speziellen Behandlung während der Speicherfreigabe, da sie im Gegensatz zu allen anderen Objekten direkte Verweise „in“ Objekte enthalten dürfen (vgl. Abschnitt 3.3.1).

Neue Klassen müssen beim Objektspeicher unter Angabe einer entsprechenden Beschreibung registriert werden. Dieser erstellt einen neuen Deskriptor und vergibt eine neue Klassennummer.

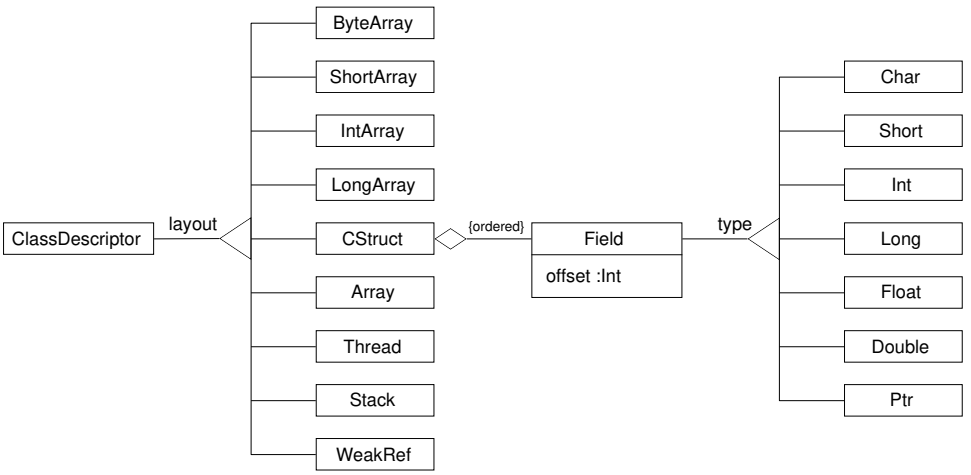

Abbildung 3.5: Klassendeskriptoren des Objektspeichers

3.2.6 C-Strukturobjekte

Objekte dieses Typs entsprechen flachen C-Strukturen fester Größe. C-Strukturobjekte werden in die C-Aufrufschnittstelle und für den Datenaustausch zwischen der TL-2 Programmierumgebung und der virtuellen Maschine eingesetzt (vgl. Abschnitte 3.3.4 und 4.2.1). Der

Klassendeskriptor eines C-Strukturobjekts verfügt über zusätzliche Informationen: Eine Beschreibung der einzelnen Datenfelder und ihres Offset ab Strukturbeginn.

Im einzelnen werden unterstützt:

Ganzzahlige Konstanten (jeweils mit und ohne Vorzeichenbehaftung):

- Zeichenkonstanten (*char*)
- 16 Bit Ganzzahlen (*short*)
- 32 Bit Ganzzahlen (*int*, *long*)[1]
- 64 Bit Ganzzahlen (*long long*)

Gleitkommazahlen:

- einfacher Genauigkeit (*float*, 32 Bit)
- doppelter Genauigkeit (*double*, 64 Bit)

Zeiger:

- Objektverweis (*void* $*$, 32 Bit)

Die Beschreibungen werden von der Speicherbereinigung bei der Suche nach Objektverweisen und dem Interpreter für die Konvertierung von C und TL-2 Datentypen benötigt (vgl. Abschnitt 3.3.3). Weiterhin ermöglichen sie die automatische Anpassung der Strukturelemente bei der Migration eines gesicherten Objektspeicherzustandes auf eine andere Rechnerarchitektur.

Zeiger müssen auf Objekte im Objektspeicher verweisen oder den Wert 0 (*NULL pointer*) enthalten. Zeiger auf externe Objekte werden vom Objektspeicher nicht unterstützt, sie können jedoch durch 32 Bit Ganzzahlen repräsentiert werden.

C-Strukturobjekte implizieren ein hohes Maß an Plattformabhängigkeit. Die Datentypen unterscheiden sich je nach Prozessorarchitektur in ihrer Größe, der Ausrichtung im Speicher (*alignment*) und der Bytefolge (*litte endian* oder *big endian*). In Abbildung 3.6 ist als Beispiel das Speicherobjekts mit zugehörigem Deskriptor für folgende C-Struktur dargestellt:

```
struct dirent {
  int        inode;
  long long  offset;
  short      size;
  char      *name;
}
```

[1] Die Implementierung setzt eine Breite von 32 Bit der Datentypen *int*, *long* und *void* $*$ voraus.

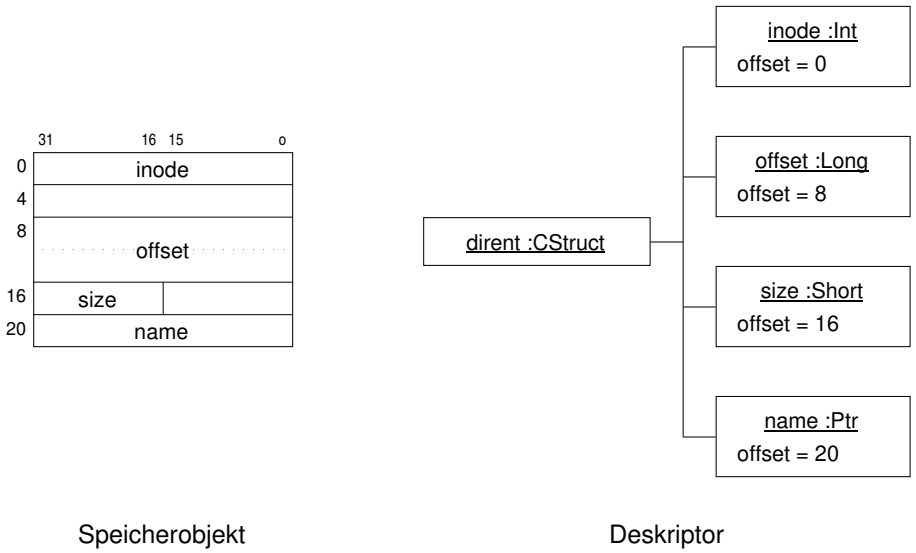

Abbildung 3.6: C-Strukturobjekt auf der SPARC-Architektur

3.2.7 Speicherbereinigung

Der Objektspeicher verfügt über eine automatische Speicherbereinigung, für die eine an die Erfordernisse des Systems angepaßte Variante des in [Bar88] beschriebenen Algorithmus zur Anwendung kommt. Die Speicherfreigabe kann implizit von jeder Objektallokation ausgelöst werden und ist mit Ausnahme der weiter unten beschriebenen Enumeratorfunktionen für die höheren Ebenen des Laufzeitsystems transparent. Die Speicherfreigabe ist nicht reentrant und kann nicht nebenläufig ausgeführt werden, was besondere Synchronisationsmaßnahmen in einer nebenläufigen Umgebung erfordert (siehe Abschnitt 5.2.1).

Ausgangspunkt der persistent im Objektspeicher gehaltenen Objekte ist das Wurzelobjekt. Alle von ihm transitiv erreichbaren Objekte bilden den Objektgraphen. Das Wurzelobjekt wird vom Objektspeicher verwaltet und kann über die Aufrufschnittstelle erfragt und gesetzt werden.

Während der Speicherbereinigung wird der Objektspeicher ähnlich der traditionellen *stop and copy garbage collection* in zwei Bereiche unterteilt: Die Seiten der alten Objekte (*from space*) und den Bereich der Objektkopien (*to space*). Die Zuordnung der belegten Speicherseiten zu einem dieser Bereiche erfolgt über die beiden globalen Identifikatoren `space` bzw. `newSpace`. Der Objektgraph wird ausgehend vom Wurzelobjekt vom alten in den neuen Teilbereich kopiert. Um die transitive Hülle zu erfassen, werden alle kopierten Objekte nach weiteren Referenzen durchsucht. Neben der Möglichkeit, ein Objekt zu kopieren, können auch komplette Speicherseiten mit all ihren Objekten durch Promotion, d.h. Änderung des Bereichsidentifikators in ihrem Deskriptor, in den neuen Bereich übernommen werden. Am

Ende der Speicherbereinigung werden die Seiten des alten Teilbereichs freigegeben.

Zusätzlich zur ausgezeichneten Wurzel können über zwei Enumeratorfunktionen, die mit Beginn der Speicherfreigabe aufgerufen werden, zusätzliche Objekte als persistente Wurzeln markiert werden:

1. Eine Enumerationsfunktion über Objektverweise. Die referenzierten Objekte werden kopiert, womit sich ihre `OID` ändert. Gleichzeitig wird die neue `OID` zurückgegeben, so daß die höheren Systemebenen bestehende Verweise anpassen können.

2. Eine Enumerationsfunktion über Hauptspeicheradressen. Sollte eine dieser Adressen in eine Seite des Objektspeichers verweisen, so wird diese Seite, bei einem großen Objekt zusätzlich alle zugehörigen Seiten, gesperrt und komplett in den neuen Speicherbereich (*new space*) übernommen, d.h. alle in ihr befindlichen Objekte werden nicht kopiert und ihre `OID` somit nicht verändert. Über diesen Mechanismus kann die Integrität externer Referenzen auf Objekte, wie sie z.B. beim Aufruf externer Bibliotheken entstehen, auch in einer Umgebung mit mehreren aktiven Threads gewährleistet werden.

Kann die Speicherbereinigung nicht genügend Seiten freigeben, ist es erforderlich, den Objektspeicher zu vergrößern, um die weitere Allokation von Objekten zu gewährleisten. Hierzu wird zusätzlicher Hauptspeicher vom Betriebssystem alloziert. Scheitert die Speicheranforderung aufgrund eines Mangels an virtuellem Speicher, wird der Tycoon-2 Prozeß mit einer Fehlermeldung beendet. Die Speicherbereinigung selbst kann nicht aufgrund von zu wenig Hauptspeicher fehlschlagen, wenn sie bei einer Seitenbelegung bis maximal 50% ausgelöst wird.

3.2.8 Schwache Referenzen

Nicht mehr referenzierte Objekte werden von der Speicherbereinigung automatisch freigegeben. In einigen Fällen wird jedoch eine zusätzliche Kontrollmöglichkeit benötigt, etwa bei externen Ressourcen, die zusätzlich Betriebsmittel oder Speicherbereiche freigeben müssen. Die hierfür nötigen Mechanismen werden von sogenannten schwachen Referenzen (*weak references*) bereitgestellt.

Schwache Referenzen sind spezielle Objekte, die einen Objektverweis etablieren, ohne das referenzierte Objekt vor der Speicherbereinigung zu schützen. Jede schwache Referenz verfügt über eine assoziierte TL-2 Funktion, die ausgeführt wird, nachdem die Speicherbereinigung die Unerreichbarkeit des Zielobjektes festgestellt hat. Als erreichbar gelten hierbei alle Objekte, zu denen ein Pfad vom Wurzelobjekt oder Objekten der Enumeratorfunktionen existiert, dessen vorletzter Knoten keine schwache Referenz ist.[2] Die Zielobjekte werden erst nach Beendigung der Funktion freigegeben, allerdings ist es auch möglich, ihre Erreichbarkeit innerhalb der Funktion wiederherzustellen. Verweisen mehrere schwache Referenzen auf

[2] Dies schließt einen Zyklus innerhalb schwach referenzierter Objekte ein. Alle diese Objekte gelten als erreichbar und werden nicht freigegeben.

ein Objekt, so werden alle assoziierten Funktionen in einer nicht spezifizierten Reihenfolge ausgeführt.

3.2.9 Persistenz

Um die Persistenz von Tycoon-2 zu gewährleisten, muß der Zustand des im Hauptspeicher gehaltenen Objektspeichers auf einem nicht flüchtigen Medium abgelegt werden. Die Aufrufschnittstelle des Objektspeicher stellt hierzu zwei atomare Operationen zur Verfügung, die es erlauben, ein Speicherabbild im Dateisystem des jeweiligen Zielrechners zu sichern und wieder einzulesen:

`commit()`: Speichert ein Abbild des Objektspeichers in eine Datei. Ist nicht genügend Speicher auf dem Dateisystem vorhanden, wird der Vorgang abgebrochen und eine Ausnahme ausgelöst. Das System arbeitet auch bei einem fehlgeschlagenen `Commit` weiter. Die TL-2 Schicht fängt die Ausnahme ab und kann je nach Applikation darauf reagieren, z.B. mit einer Fehlermeldung auf der Console oder dem Senden einer *E-Mail*. Ein bereits bestehendes Objektspeicherabbild wird in einem solchen Fall nicht verändert.

`restore()`: Restauriert einen gesicherten Objektspeicherzustand. Diese Operation wird nur einmal beim Neustart des Systems ausgeführt. Ist nicht genügend Hauptspeicher vorhanden, um den gesicherten Zustand wiederherzustellen, wird die Systeminitialisierung mit einer Fehlermeldung abgebrochen und der Tycoon-2 Prozeß wird beendet.

Ein gesicherter Objektspeicherzustand ist nicht an einen Rechner gebunden, sondern kann auf jeder Plattform, die von Tycoon-2 unterstützt wird, wiederhergestellt werden. Weitere Angaben finden sich in Abschnitt 4.1.5.

3.3 Virtuelle Maschine

Die Spezifikation der virtuellen Maschine definiert die Schnittstelle für die TL-2 Programmierumgebung. Sie beinhaltet den Befehlssatz des Interpreters, gemeinsam genutzte Datenstrukturen und elementare Methoden.

3.3.1 Interpreter

Die virtuelle Maschine basiert auf einem Ausführungsmodell mit einem Laufzeitstapel, minimalem Registersatz und stapelorientierten Instruktionen. Es existieren u.a. Opcodes zum Versenden von Nachrichten an Objekte und Sprungbefehle zur Realisierung von Kontrollstrukturen. Das stapelorientierte Modell vereinfacht sowohl die Implementierung eines Interpreters als auch des TL-2 Compilers. Abbildung 3.7 illustriert die definierten Datenstrukturen.

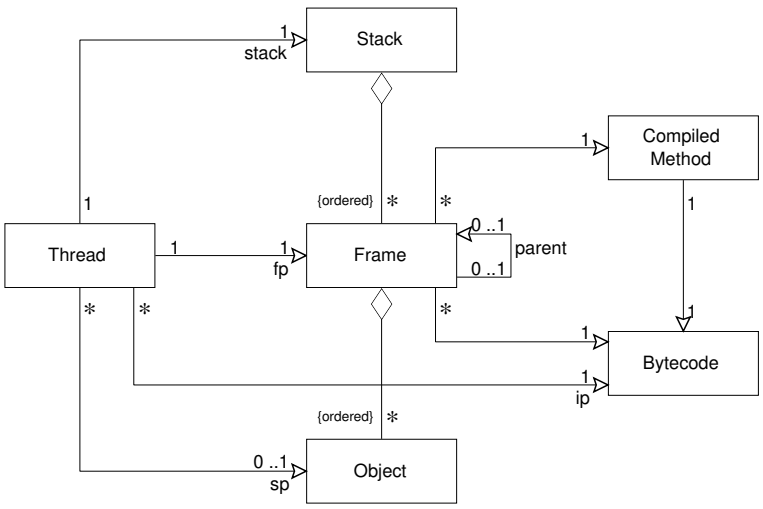

Abbildung 3.7: Datenstrukturen des Interpreters

Registersatz

Der Registersatz der virtuellen Maschine besteht aus 3 Registern:

IP: Befehlszähler. Er verweist auf den Bytecode, der vom Interpreter ausgeführt wird.

SP: Stapelzeiger. Er adressiert das oberste Element des Stacks.

FP: Rahmenzeiger. Er verweist in den Laufzeitstapel, auf den Sicherungsrahmen der aktuellen Methode bzw. Funktion höherer Ordnung.

Alle Register enthalten direkte Verweise in Objekte.

Bytecode

Alle Opcodes werden als Byte (8 Bit) kodiert, evtl. gefolgt einem 8 oder 16 Bit breiten Operanden. 16 Bit breite Operanden werden als *little endian*, d.h. das niederwertige Byte zuerst, kodiert. Opcodes können Objektverweise als Argumente auf dem Laufzeitstapel erwarten und als Resultat einen Objektverweis auf dem Laufzeitstapel ablegen.

Die Opcodes sind in 4 Klassen unterteilt:

- Stapelmanipulation
- Sprungbefehle

Opcode	Argument[1]	Beschreibung
Nachrichtenbefehle		
send	<id>	Nachricht mit der Selektornummer `id` senden
send0	<id>	wie `send`, jedoch mit fest codierter Argumentzahl
send1	<id>	"
send2	<id>	"
send3	<id>	"
send4	<id>	"
send5	<id>	"
sendSuper	<id>	wie `send`, jedoch wird in den Vorfahrenklassen gesucht
sendTail	<id>	wie `send`, jedoch endrekursiv
Spezielle Nachrichtenbefehle mit festem Selektor		
sendAdd	<id>	zwei markierte Ganzzahlen addieren
sendSub	<id>	zwei markierte Ganzzahlen subtrahieren
sendLessOrEqual	<id>	zwei markierte Ganzzahlen vergleichen ($\leq$)
sendEqual	<id>	zwei `OID`s auf Identität vergleichen
sendNotEqual	<id>	Test, ob zwei `OID`s ungleich sind
sendFun0Apply	<id>	Evaluationsnachricht für `HOF` ohne Argumente
sendFun1Apply	<id>	Evaluationsnachricht für `HOF` mit einem Argument
sendIsNil	<id>	Empfänger mit `nil` vergleichen
sendIsNotNil	<id>	Test, ob Empfänger ungleich `nil` ist
Ladeoperationen des Stapelspeichers		
global	<n>	globale Variable aus Closure laden
literal8	<n>	Literal aus Literalvektor laden
literal16	<nn>	wie `literal8`, jedoch mit 16 Bit breitem Argument
arg	<n>	Argument laden
local	<n>	lokale oder temporäre Variable laden
char	<n>	Zeichenkonstante laden
byte	<i>	vorzeichenbehaftete 8 Bit Zahl laden
short	<ii>	vorzeichenbehaftete 16 Bit Zahl laden
true		boolschen Wert `true` laden
false		boolschen Wert `false` laden
nil		`nil` laden
minusOne		markierte Ganzzahl -1 laden
zero		markierte Ganzzahl 0 laden
one		markierte Ganzzahl 1 laden
two		markierte Ganzzahl 2 laden
Speicheroperationen des Stapelspeichers		
storeLocal	<n>	oberstes Element in lokale Variable schreiben

[1] `id`: 16 Bit Selektornummer, `n`: 8 Bit Index, `nn`: 16 Bit Index, `i`: 8 Bit Int, `ii`: 16 Bit Int

Tabelle 3.2: Bytecodes der virtuellen Maschine (1)

Opcode	Argument[1]	Beschreibung
Stapelmanipulation		
drop	<n>	`n` Elemente vom Stapel entfernen
adjust	<n>	`n` Elemente unter dem obersten vom Stapel entfernen
pop		oberstes Element vom Stapel entfernen
Zugriff auf veränderliche Bindungen		
cellNew		neue Zelle für veränderliche Bindungen allozieren
cellLoad		Wert aus Zelle laden
cellStore		Wert in Zelle schreiben
Closure erzeugen		
closure	<n>	Closure allozieren und initialisieren
Kontrollflußbefehle		
return		aus Methodenaufruf zurückkehren
ifTrue8	<o>	Sprung, wenn das oberste Stapelelement `true` ist
ifTrue16	<oo>	wie `ifTrue8`, jedoch mit 16 Bit Offset
ifFalse8	<o>	Sprung, wenn das oberste Stapelelement `false` ist
ifFalse16	<oo>	wie `ifFalse8`, jedoch mit 16 Bit Offset
jump8	<o>	unbedingter Sprung mit 8 Bit Offset
jump16	<oo>	unbedingter Sprung mit 16 Bit Offset
Synchronisation		
sync		Synchronisationswunsch überprüfen
Maschineninterner Bytecode — wird nicht vom TL-2 Compiler generiert		
terminateThread		aktuellen Thread sofort terminieren

[1] `n`: 8 Bit Zahl, `o`: 8 Bit Offset, `oo`: 16 Bit Offset

Tabelle 3.3: Bytecodes der virtuellen Maschine (2)

- Methodenaufruf
- Erzeugung von Funktionsabschlüssen

Die Tabellen 3.2 und 3.3 zeigen eine Übersicht aller Opcodes, eine detaillierte Auflistung findet sich im Anhang.

Bei einem `send`-Opcode wird der Methodenselektor und die Anzahl der auf dem Laufzeitstack übergebenen Argumente in einem 16 Bit breiten Argument, der Selektornummer, kodiert. Diese Nummer dient als Index in die im Wurzelobjekt verankerte Selektortabelle (siehe Abschnitt 3.3.2). Sie erleichtert und beschleunigt die Implementierung eines Nachrichten-Caches (siehe Abschnitt 4.2.3) und vermeidet die explizite Angabe eines Selektorobjekts im Literalvektor. Die Argumente eines Methodenaufrufs werden von links nach rechts auf dem Laufzeitstapel abgelegt, d.h. das Empfängerobjekt zuerst.

Als Optimierung kodieren einige Nachrichtenbefehle ihre Argumentzahl im Bytecode. Weiterhin existieren spezielle Opcodes, die besonders häufige Nachrichten direkt, ohne einen

Methodenaufruf, realisieren (*special sends*). Es handelt sich hierbei um arithmetische Operationen auf markierten Ganzzahlen (siehe auch Abschnitt 3.2.4) und elementare Vergleichsfunktionen. Falls die Typen der von ihnen erwarteten Argumente nicht der Spezifikation entsprechen, wird die in der Selektornummer kodierte Nachricht gesandt. Wenn der TL-2 Compiler diese Optimierungsmöglichkeit nutzt, können die entsprechenden Methoden auf der TL-2 Seite nicht mehr redefiniert werden.

Einige Opcodes existieren in zwei Formen, mit 8 bzw. 16 Bit breitem Argument, z.B. alle Sprungbefehle. Da der Großteil der Methoden kurz ist (im Durchschnitt 35 Byte) wird der Code kompakter und kann gleichzeitig schneller dekodiert werden.

Literale, wie z. B. Zeichenketten und Gleitkommazahlen, werden, um die Anzahl der Objektallokationen zur Laufzeit zu vermindern, nicht direkt in den Bytecode eingebettet. Sie werden vom TL-2 Compiler bereits zur Übersetzungszeit alloziert und im Literalvektor des Methodenobjekts abgelegt. Dort sind über den `literal`-Opcode zugreifbar. Für besonders häufig benötigte Literale wie `true`, `false`, `nil`, Zeichenkonstanten und kleine Ganzzahlen existieren spezielle Kurzformen, vergleichbar mit den entsprechenden Opcodes in Smalltalk80 [GR83].

Laufzeitstapel

Der Laufzeitstapel ist, mit Ausnahme der Sicherungsrahmen, ein linearer Bereich von markierten Speicherworten (`OID`). Auf diesem Stapel werden Argumente und Resultate von Byteopcodes, Methodenargumente, lokale Variablen und, beim Methodenaufruf, Sicherungsrahmen abgelegt. Die Adressierung des obersten Speicherwortes erfolgt durch den Stapelzeiger, die des aktuellen Sicherungsrahmens durch den Rahmenzeiger. Der Stapel wächst im Speicher von der höchsten Speicheradresse zur niedrigsten.

Der zu einer Methode gehörende Stapelbereich gliedert sich in drei Bereiche (siehe Abbildung 3.8):

Lokaler Datenbereich: Auf dem lokalen Datenbereich werden lokale Variablen, Operanden und Resultate der Bytecodes abgelegt.

Argumentbereich: Der Argumentbereich beinhaltet Empfängerobjekt und Argumente eines Methodenaufrufs. Er ist gleichzeitig Teil des lokalen Datenbereichs der aufrufenden Methode.

Sicherungsrahmen: Sicherungsrahmen halten zusammen mit dem Argumentbereich den Kontext (*environment*) eines Methodenaufrufs fest. Im Gegensatz zu den beiden anderen Datenbereichen sind sie nur dem Laufzeitsystem zugänglich, d.h. es existieren keine Bytecodes oder eingebaute Funktionen, die den Zugriff auf den Methodenkontext erlauben.[3]

[3] Diese Einschränkung muß im Zuge der Implementierung einer vollständigen Entwicklungsumgebung mit integriertem Debugger geändert werden.

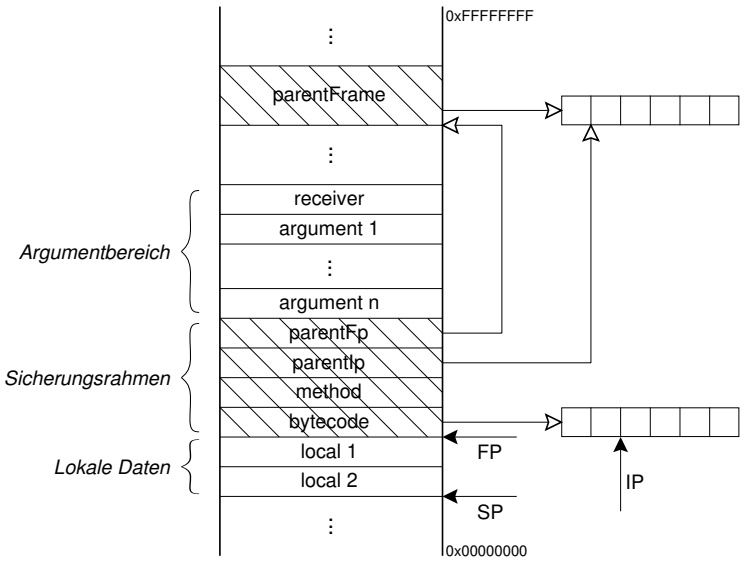

Abbildung 3.8: Laufzeistapel des Interpreters

Sicherungsrahmen

Sicherungsrahmen werden bei Aktivierung einer Methode aufgebaut und enthalten den für den Interpreter spezifischen Kontext eines Methodenaufrufs. Ein Sicherungsrahmen besteht im einzelnen aus:

`parentIp`: Sicherung des Instruktionszählers der aufrufenden Methode.

`parentFp`: Verweis auf den Kontext der aufrufenden Methode.

`method`: Verweis auf das Methodenobjekt der ausgeführten Methode (vgl. Abschnitt 3.3.3).

`bytecode`: Verweis auf den ausgeführten Bytecode. Dieser Eintrag ist auch über das Methodenobjekt erreichbar und wird nur für die Speicherbereinigung im Sicherungsrahmen abgelegt.

Die Felder `parentIp` und `parentFp` stellen eine Kopie der Registerinhalte des Instruktions- und des Rahmenzeigers zum Zeitpunkt des Methodenaufrufs dar. Mit Beendigung einer Methode (`return`-Bytecode) werden die jeweiligen Registerinhalte aus diesen Feldern restauriert und die Abarbeitung der aufrufenden Methode fortgesetzt.

Sicherungsrahmen enthalten direkte Zeiger in Objekte (`parentIP`, `parentFP`) und müssen von der Speicherbereinigung separat behandelt werden. Laufzeitstapel wie auch Threadobjekte besitzen aus diesem Grund einen eigenen Layouttyp (siehe Abschnitt 3.2.5).

Klasse	Layout	Klasse	Layout
Basisklassen			
WeakRef	WeakRef	MutableString	ByteArray
ByteArray	ByteArray	Symbol	ByteArray
ShortArray	ShortArray	Bool	CStruct
IntArray	IntArray	True	CStruct
LongArray	LongArray	False	CStruct
Thread	Thread	Nil	Array
TVMStack	Stack	Char	CStruct
Array	Array	Int	CStruct
MutableArray	Array	Long	CStruct
String	ByteArray	Real	CStruct
Spezielle Objekte			
EmptyList	Array	DLL	Array
List	Array	Root	Array
Class	Array	Tycoon	Array
MethodDictionary	Array	Object	Array
Synchronisationsobjekte			
Mutex	CStruct	BroadcastingCondition	CStruct
Condition	CStruct		
Methodenobjekte			
Method	CStruct	PoolAccessMethod	CStruct
CompiledMethod	CStruct	PoolUpdateMethod	CStruct
BuiltinMethod	CStruct	CSlotAccessMethod	CStruct
ExternalMethod	CStruct	CSlotUpdateMethod	CStruct
SlotMethod	CStruct	CompiledFun	CStruct
SlotAccessMethod	CStruct	DeferredMethod	CStruct
SlotUpdateMethod	CStruct	UnimplementedMethod	CStruct
PoolMethod	CStruct		
Closures			
Function	Array	Fun5	Array
Fun0	Array	Fun6	Array
Fun1	Array	Fun7	Array
Fun2	Array	Fun8	Array
Fun3	Array	Fun9	Array
Fun4	Array	Fun10	Array
Ausnahmen der virtuellen Maschine			
TypeError	Array	IndexOutOfBounds	Array
DoesNotUnderstand	Array	BlockingCallInterrupt	Array
MustBeBoolean	Array	ThreadCancelled	Array
DLLOpenError	Array	CommitError	Array
DLLCallError	Array	WrongSignature	Array
DivisionByZero	Array		

Tabelle 3.4: Vordefinierte Klassen

3.3.2 Vordefinierte Klassen

Die virtuelle Maschine operiert auf einem kleinen Satz vordefinierter Klassen, die dem Datenaustausch von Programmierumgebung zum Laufzeitsystem dienen. Eine vollständige Übersicht bietet Tabelle 3.4. Neben elementaren Datentypen, wie Ganz- und Gleitkommazahlen, Zeichenketten, Wahrheitswerten und verschiedenen linearen Feldern, finden sich hier auch die Klassen von Methoden- und Funktionsobjekten sowie Ausnahmeobjekten. Aus Sicht der virtuellen Maschine stellen die gemeinsam genutzten Objekte elementare C-Datentypen oder C-Strukturobjekte dar.

Auf gemeinsam genutzte Datenstrukturen greift die Maschine nur lesend zu, Initialisierung und Manipulation werden in TL-2 implementiert. Eine Ausnahme hiervon stellen nur Exemplare der Klassen `Thread`, `WeakRef`, `Stack` und der vom Laufzeitsystem erzeugten Ausnahmen dar, die der direkten Kontrolle der Maschine unterliegen.

Ausgewählte Objekte von besonderer Bedeutung werden in den folgenden Abschnitten näher betrachtet, wobei nur Exemplarvariablen mit Relevanz für die virtuelle Maschine berücksichtigt werden (für weitergehende Informationen siehe [Wie97]).

Das Wurzelobjekt

Ausgangspunkt des persistenten Objektgraphen ist das Wurzelobjekt. Es stellt einen Vektor von Objektverweisen dar, über den alle für die virtuelle Maschine und die übergeordnete TL-2 Schicht relevanten Datenstrukturen erreichbar sind. Das Wurzelobjekt ist das einzige Objekt, dessen Objektverweis direkt über die Objektspeicherschnittstelle erfragbar ist.

Abbildung 3.9 zeigt die Darstellung des Wurzelobjekts reduziert auf die von der virtuellen Maschine benötigten Datenstrukturen. Im einzelnen sind dies:

`threads`: Liste aller aktiven Threads.

`classTable`: Tabelle aller im System definierten Klassenobjekte. Der Zugriff erfolgt indiziert mit der Klassennummer.

`selectorTable`: Tabelle der Methodenselektoren. Selektoren sind konstante Zeichenketten, von denen jeweils nur ein Exemplar im System existiert, so daß ihr Objektverweis eindeutig ist. Der Zugriff erfolgt indiziert mit der Selektornummer.

`argTable`: Zur Selektortabelle analoge Tabelle, mit der von einem Selektor erwarteten Argumentzahl.[4]

`true, false`: Die boolschen Wahrheitswerte.

`charTable`: Tabelle aller Zeichenkonstanten.

Die aufgeführten Datenstrukturen werden in den folgenden Abschnitten näher beschrieben.

[4] Dies erlaubt der Interpreterseite den Zugriff auf das Empfängerobjekt und die Prüfung auf die korrekte Argumentzahl; der Typ der einzelnen Argumente ist hierbei nicht von Relevanz.

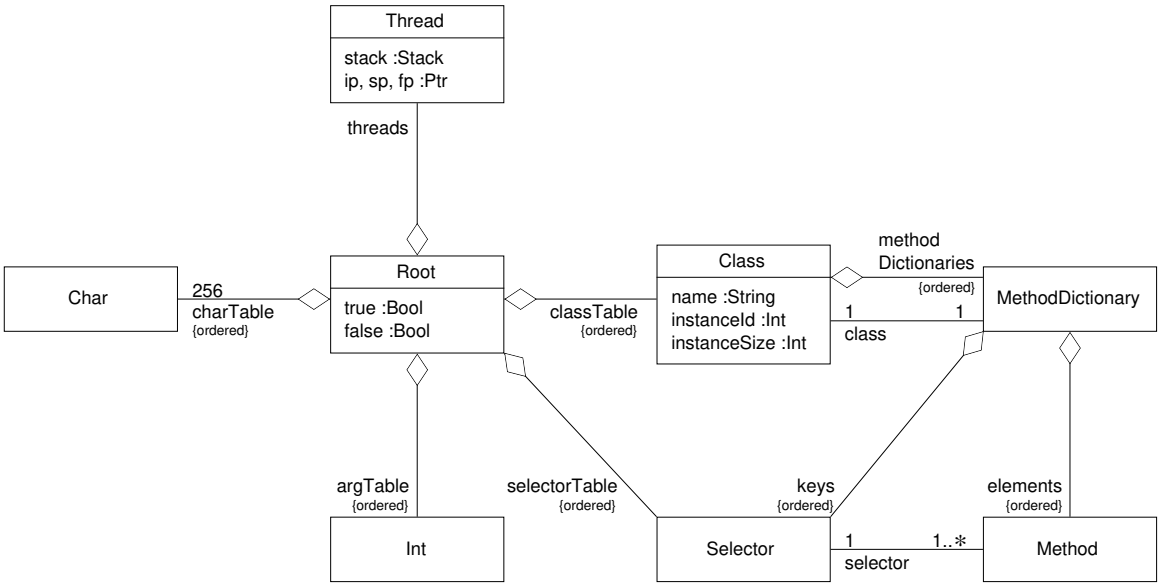

Abbildung 3.9: Das Wurzelobjekt

Thread

Threadobjekte beschreiben einen leichtgewichtigen TL-2 Prozeß. Jeder Thread besitzt einen eigenen Laufzeitstapel (`stack`) und Registersatz (`sp, fp, ip`). Eine detailliertere Betrachtung findet in Kapitel 5 statt.

Klassentabelle

Wie in Abschnitt 3.2.4 beschrieben, besitzt jedes Objekt eine Klassennummer, die es gegenüber der virtuellen Maschine als Exemplar einer Klasse identifiziert. Die Zuordnung von Klassennummer zu Klassenobjekt erfolgt über die Klassentabelle des Wurzelobjekts.

Der Vorteil einer fortlaufenden Nummer im Vergleich zu einem Objektverweis liegt einerseits in der Speicherersparnis und zum anderen in der Möglichkeit, innerhalb der Maschine mit einem kleinen Satz vordefinierter Konstanten für die Standardklassen zu arbeiten. Auf TL-2 Seite steht dem der erhöhte Aufwand einer expliziten Verwaltung der Klassentabelle entgegen, die sich nicht auf die Speicherbereinigung des Objektspeichers stützen kann.

Selektortabelle

Die virtuelle Maschine erkennt Methoden anhand der Selektornummer, dem Argument der verschiedenen `send`-Bytecodes, die als Index in die Selektortabelle dient. Die Selektortabelle

verzeichnet alle dem System bekannten Methodennamen, die Argumenttabelle die einem Selektor zugehörige Argumentzahl.

Das System differenziert bei der Methodensuche zwischen Methoden mit gleichem Namen, d.h. die Selektortabelle verweist auf dasselbe Symbol, jedoch anderer Argumentzahl (siehe Abschnitt 4.2.2). Sie besitzen daher verschiedene Selektornummern.

Klassenobjekte

Klassenobjekte beschreiben alle Exemplare einer Klasse und sind mittels elementarer Methoden für ihre Erzeugung zuständig. Sie enthalten Informationen über Super- und Metaklasse, private und öffentliche Zustandsvariablen sowie die Klassenpräzedenzliste (*class precedence list*). Für die virtuelle Maschine von Bedeutung sind:

`name:` Klassenname

`instanceId:` Klassennummer, die beim Erzeugen neuer Exemplare vergeben wird.

`instanceSize:` Größe der erzeugten Exemplare, abhängig von der Art des erzeugten Objektes, z.B. Byte bei Bytefeldern, Anzahl der Exemplarvariablen bei Standardklassen. Im Fall von C-Strukturobjekten verwendet der Objektspeicher seine interne Deskriptortabelle (siehe Abschnitt 3.2.6) und dieses Feld ist ohne Bedeutung.

`methodDictionaries:` Liste von Methodenverzeichnissen, die entsprechend der Klassenpräzedenzliste vom TL-2 Compiler berechnet wird. Diese Liste wird beim Empfang einer Nachricht linear nach der ersten zum Selektor passenden Methode durchsucht.

Methodenverzeichnisse

Methodenverzeichnisse sind Hashtabellen, die eine Abbildung von einem Selektor zu einem Methodenobjekt definieren. Sowohl auf auf TL-2 Seite wie in der virtuellen Maschine muß zum Zugriff derselbe Algorithmus zum Einsatz kommen, in diesem Fall das *hashing with linear probing*. Für den Aufbau der Tabelle wird der Hashwert der Selektorobjekte benutzt (siehe Abschnitt 3.2.4).

`keys:` Tabelle der Selektoren

`elements:` Tabelle der zugehörigen Methodenobjekte

`class:` Verweis auf das Klassenobjekt, zu dem das Methodenverzeichnis gehört

Spezielle Werte

Von den Boolschen Wahrheitswerten `true` und `false` existiert jeweils nur ein Exemplar, das im Wurzelobjekt registriert ist. Basisoperationen auf Wahrheitswerten, wie Objektidentität und -vergleich, reduzieren sich dadurch auf einen Vergleich der Objektverweise mit den entsprechenden Einträgen im Wurzelobjekt.

Exemplare der Klasse `nil` werden, wie in Abschnitt 3.2.4 beschrieben, als Objektverweis auf die Speicheradresse 0 dargestellt. Hierdurch entspricht der Wert `nil` in Verbindung mit der C-Schnittstelle der in [KR90] definierten Repräsentation des NULL-Zeigers.

Ganzzahlen

Für vorzeichenbehaftete 32 Bit Ganzzahlen (Klasse `Int`) gibt es abhängig von ihrem Wert zwei unterschiedliche Repräsentationen:

- die Kurzform als markierter Objektverweis (*tagged integer*), wenn sich ihr Wert in 30 Bit darstellen läßt
- als eigenständiges 32 Bit großes Objekt, auf das per Indirektion über eine `OID` zugegriffen wird

Die Forderung, diese Zweiteilung gegenüber dem TL-2 System transparent zu halten, hat maßgebliche Auswirkungen auf die Implementierung des Interpreters. Alle Operationen (eingebaute Methoden etc.) auf Ganzzahlen müssen beide Formen verarbeiten. Konkret bedeutet dies, daß jede dieser Operationen ihre Ganzahlargumente auf den vorliegenden Typ prüfen und ihren Wert extrahieren muß. Alle Berechnungen finden mit 32 Bit Breite statt. Wird eine Ganzzahl als Ergebnis geliefert, muß diese, abhängig von ihrem Wert, markiert werden oder es muß ein neues Objekt erzeugt werden.

Applikation	Operation	Ergebnis		Anteil
		markiert	Objekt	Objekte
Parser Generator	+	9271580	16233	0.18 %
	−	4124157	3167	0.08 %
	*	1851518	16233	0.88 %
Type Checker	+	2592552	108540	4.19 %
	−	686829	11697	1.70 %
	*	601759	108542	18.04 %

Tabelle 3.5: Anteil der Objektallokationen bei 32 Bit Ganzzahlarithmetik

Dieser erhöhte Aufwand und damit größere Zeitbedarf zur Ausführung der Primtive, im Vergleich zu einer Beschränkung auf markierte Ganzzahlen wie im Prototyp, wird durch den

vollständig zur Verfügung stehenden Wertebereich aufgewogen. Der Vorteil einer markierten Repräsentation in Bezug auf Ganzzahlarithmetik, die Vermeidung einer Objektallokation bei der Bereitstellung des Resultats, bleibt durch den Wertebereich von 30 Bit bei einem Großteil aller Operationen erhalten (siehe Tabelle 3.5). Die in Abschnitt 3.3.1 beschriebenen speziellen Nachrichtenbefehle für markierte Ganzzahlen unterstützen dieses Vorgehen.

Zeichenkonstanten

Zeichenkonstanten besitzen im Tycoon-System eine Breite von 8 Bit, d.h. von der Klasse Zeichenkonstante (`Char`) existieren nur 256 verschiedene, nicht veränderliche Exemplare. Alle Exemplare dieser Klasse sind daher, wie auch in Smalltalk80, in einer im Wurzelobjekt registrierten Zeichentabelle zusammengefaßt. Dieses auch als *flyweight pattern* bekannte Entwurfsmuster [GHJV95] bietet drei wesentliche Vorteile:

- Der Speicherverbrauch für Zeichenkonstanten wird minimiert.
- Es müssen zur Laufzeit keine weiteren Exemplare erzeugt werden.
- Jede Zeichenkonstante besitzt eine eindeutige Identität.

3.3.3 Methoden

Die virtuelle Maschine kennt verschiedene Arten von Methoden, die beim Senden einer Nachricht an ein Objekt zur Ausführung kommen können. In Abbildung 3.10 ist die TL-2 Klassenhierarchie der Methoden dargestellt, wobei nur Exemplarvariablen, die von der Maschine zugegriffen werden, aufgeführt sind. Sie kann in zwei Teilbereiche gegliedert werden:

- Methoden zur Ausführung von Bytecode (`CompiledMethod`, `CompiledFun`) und Maschinencode (`BuiltinMethod`, `ExternalMethod`)
- Methoden zum Zugriff und zur Änderung von Exemplarvariablen (`PoolMethod` und `SlotMethod`). Sie sind, wie die eingebauten Methoden auch, direkt in den Interpreter integriert.

Method

Die abstrakte Klasse `Method` ist die Wurzel des Klassenbaums und definiert die allen Methodenobjekten gemeinsamen Exemplarvariablen:

`selector`: Name der Methode.

`args`: Anzahl der erwarteten Argumente.

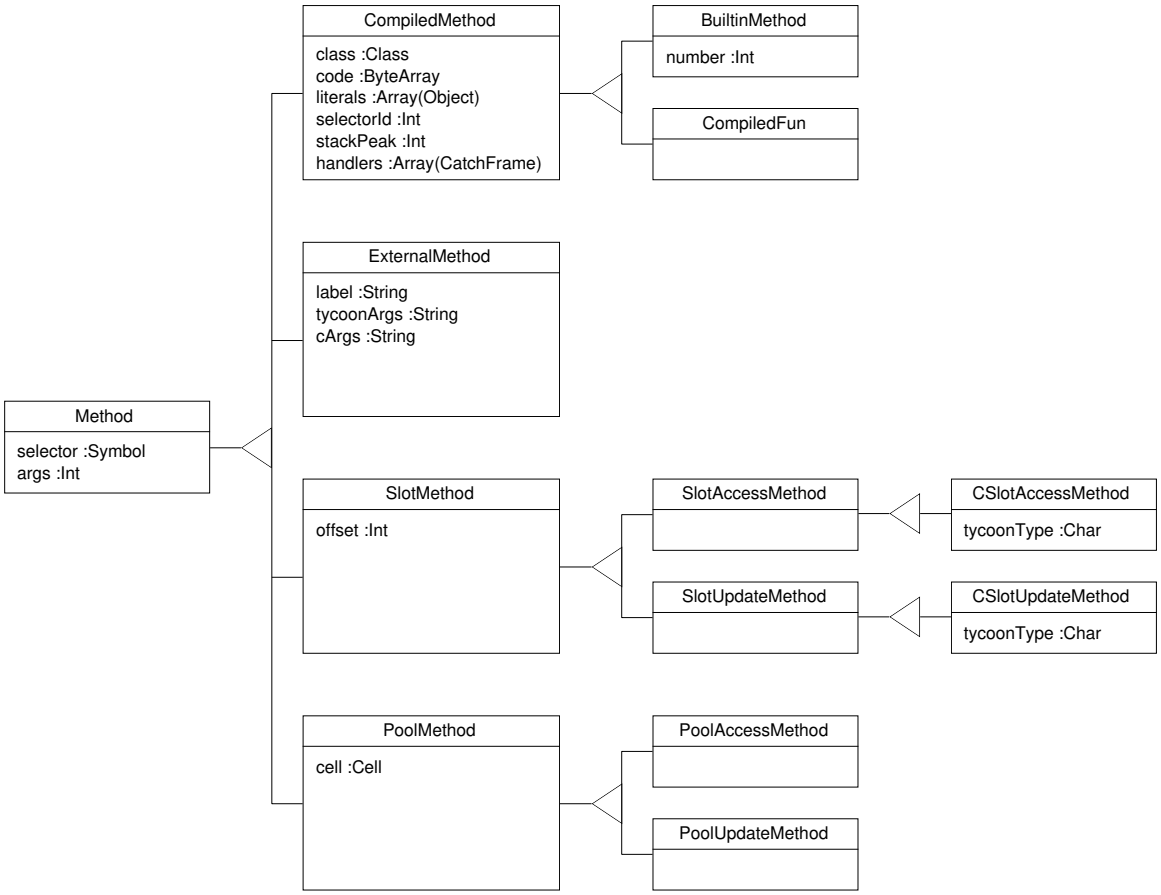

Abbildung 3.10: Klassenhierarchie der Methodenobjekte

CompiledMethod

Aus allen in TL-2 geschriebenen Methoden generiert der Compiler der Programmierumgebung Exemplare der Klasse `CompiledMethod`. Die von der Maschine zusätzlich benötigten Exemplarvariablen sind:

`class`: Verweis auf das Klassenobjekt, in dem die Methode definiert ist. Diese Exemplarvariable wird für die Methodensuche des `sendSuper`-Opcodes benötigt.

`code`: Bytecode der Methode.

`literals`: Vektor aller nicht im Bytecode kodierten Literale, z.B. Zeichenketten.

`selectorId`: Selektornummer.

`stackPeak`: Maximal von der Methode benötigter Platz auf dem Laufzeitstapel. Dieser vom TL-2 Übersetzer errechnete Wert versetzt die Maschine in die Lage, einen möglichen Stapelüberlauf bereits vor Ausführung der Methode zu erkennen und darauf zu reagieren. Hierdurch werden zeitaufwendige Prüfungen zur Laufzeit eingespart.

`handlers`: Tabelle der innerhalb der Methode definierten Ausnahmebehandlungen (siehe Abschnitt 3.3.6).

CompiledFun

`CompiledFun`-Objekte repräsentieren den statischen Teil einer Funktion höherer Ordnung (siehe Abschnitt 3.3.5).

BuiltinMethod

Exemplare der Klasse `BuiltinMethod` beschreiben elementare Methoden, die direkt als Maschinencode in den Interpreter der virtuellen Maschine integriert sind (Tabellen 3.6, 3.7 und 3.8). Sie implementieren u.a. grundlegende arithmetische und logische Operationen, ermöglichen es der Programmierumgebung, die virtuelle Maschine über Ereignisse zu informieren, und stellen die Schnittstelle zum Objektspeicher und der Laufzeitumgebung her. Jede elementare Methode wird über einen Identifikator (`number`) angesprochen.

Elementare Methoden können optional über eine TL-2 Implementierung verfügen, die bei Fehlersituationen aufgerufen wird. Dies ermöglicht es, flexibel mit den Mitteln der Hochsprache auf z.B. unerwartete Empfängerobjekte, zu reagieren.

ExternalMethod

Externe Methoden ermöglichen den Aufruf von C-Funktionen aus statisch oder dynamisch gebundenen Bibliotheken und stellen so die Verbindung des Systems mit externen Diensterbringern, wie z.B. relationalen Datenbanken, her. Die Modalitäten eines externen Funktionsaufrufs, wie die durchzuführende Datenkonvertierung, werden von der C-Aufrufschnittstelle (siehe Abschnitt 3.3.4) definiert. Empfänger externer Methodenaufrufe sind `DLL`-Objekte.

`label`: Name der externen Funktion.

`tycoonArgs`: Signatur der TL-2 Methode.

`cArgs`: Signatur der aufzurufenden C-Funktion.

`AbstractArrayClass`: Feld von `OID`s allozieren	
_new1(size :Int) :Instance	
`AbstractStringClass`: Zeichenkette allozieren	
_new1(size :Int) :Instance	
`Array`: Zugriffsmethoden für Felder von `OID`s	
"[]"(i :Int) :E	_set(i :Int, e :E) :E
size :Int	_replace(...) :Void
`BlockingExternalMethodCaller`: Synchronisation für blockierende externe Resourcen	
__suspendBlockingCall(...) :Bool	__resumeBlockingCall(...) :Void
`BroadcastingCondition`: Ereignisvariablen signalisieren und initialisieren	
signal :Void	__init :Bool
`ByteArray`: Zugriffsmethoden für Feldern von 8 Bit Werten	
"[]"(i :Int) :Int	size :Int
"[]:="(i :Int, e :Int) :Int	
`ByteArrayClass`: Feld von 8 Bit Werten allozieren	
_new1(size :Int) :ByteArray	
`Char`: Zeichenkonstante in Ganzzahl wandeln	
asInt :Int	
`ConcreteClass`: Neues Exemplar einer konkreten Klasse allozieren	
_new :Instance	
`Condition`: Elementare Operationen und Initialisierung von Ereignisvariablen	
wait(mx :Mutex) :Void	__init :Bool
timedWait(mx :Mutex, abstime :Long) :Bool	__finalize :Void
signal :Void	
`CStructClass`: Neues Exemplar einer C-Strukturklasse allozieren	
_new :Instance	
`DLL`: Externe Bibliotheken öffnen und schließen	
__open(path :String) :Int	__close(handle :Int) :Void
`Exception`: TL-2 Ausnahmen auslösen	
_raise :Nil	
`Finalizer`: Schwach referenzierte Objekte vom Objektspeicher erfragen	
__fetchWeaks :WeakRef(Object)	
`Fun0`: Evaluationsnachricht für Funktionen höherer Ordnung ohne Argumente	
"[]"() :T	
`Fun1`: Evaluationsnachricht für Funktionen höherer Ordnung mit einem Argument	
"[]"(arg1 :T1) :T	
`Fun2`: Evaluationsnachricht für Funktionen höherer Ordnung mit zwei Argumenten	
"[]"(arg1 :T1, arg2 :T2) :T	
`Fun3-Fun10`: Analog zu `Fun0-Fun2`	

Tabelle 3.6: Eingebaute Methoden (1)

`Int`: Elementare Arithmetik für 32 Bit Ganzzahlen	
"+"(x :Int) :Int	">="(x :Int) :Bool
"−"(x :Int) :Int	"|"(x :Int) :Int
"∗"(x :Int) :Int	"&"(x :Int) :Int
"/"(x :Int) :Int	"xor"(x :Int) :Int
"%"(x :Int) :Int	"not" :Int
"<"(x :Int) :Bool	">>"(nBits :Int) :Int
"<="(x :Int) :Bool	"<<"(nBits :Int) :Int
">"(x :Int) :Bool	asChar :Char

`IntArray`: Zugriffsmethoden für Felder von 32 Bit Werten	
"[]"(i :Int) :Int	size :Int
"[]:="(i :Int, e :Int) :Int	

`IntArrayClass`: Feld von 32 Bit Werten allozieren	
_new1(size :Int) :IntArray	

`LongArray`: Zugriffsmethoden für Felder von 64 Bit Werten	
"[]"(i :Int) :Long	size :Int
"[]:="(i :Int, e :Long) :Long	

`LongArrayClass`: Feld von 64 Bit Werten allozieren	
_new1(size :Int) :LongArray	

`MutableArray`: Zuweisung auf veränderliche Felder von `OID`s	
"[]:="(i :Int, e :E) :E	

`MutableString`: Zuweisung auf veränderliche Zeichenketten	
"[]:="(i :Int, ch :Char) :Char	

`Mutex`: Elementare Operationen und Initialisierung von Mutex Objekten	
lock :Void	__init :Bool
tryLock :Bool	__finalize :Void
unlock :Void	

`Object`: Allgemeine Zugriffsmethoden auf Objekte und diverse Basisoperationen	
"=="(x :Object) :Bool	"nil" :Nil
"class" :Class	_typeCast(x :Object, T <:Object) :T
__basicSize :Int	_hash :Int
__basicAt(i :Int) :Object	_setHash(hash :Int) :Int
__basicAtPut(i :Int, value :Object) :Object	_perform(...) :Nil
"true" :Bool	__doesNotUnderstand(...) :Nil
"false" :Bool	shallowCopy :Self

`Root`: Neue Klassen registrieren und Methodencache leeren	
newClassId(...) :Int	_flushClass(c :Class) :Void
_flushAll :Void	_flushSingle(c :Class, s :Selector) :Void

`ShortArray`: Zugriffsmethoden für Felder von 16 Bit Werten	
"[]"(i :Int) :Int	size :Int
"[]:="(i :Int, e :Int) :Int	

Tabelle 3.7: Eingebaute Methoden (2)

`ShortArrayClass`: Feld von 16 Bit Werten allozieren	
_new1(size :Int) :ShortArray	
`String`: Zugriffsmethoden für Zeichenketten	
"[]"(i :Int) :Char	_set(k :Int, e :Char) :Char
size :Int builtin	_replace(...) :Void
`Thread`: Initialisierung von Threads	
__init :Bool	_eintr :Void
`ThreadClass`: Thread allozieren und aktuelles Threadobjekt erfragen	
this :Thread(Void)	_new :Thread(T)
`Tycoon`: Sicherung des Systemzustandes, Zugriff auf Kommandozeilenargumente etc.	
errno :Int	_commit :Int
backTrace :Void	builtinArgv :Array(String)
_rollback :Nil	_platformCode :Int
`WeakRef`: Schwache Referenz etablieren	
__init :Self	
`WeakRefClass`: Schwache Referenz allozieren	
_new :WeakRef(T)	

Tabelle 3.8: Eingebaute Methoden (3)

SlotMethod

Der Zugriff auf Exemplarvariablen erfolgt im Tycoon-System über Nachrichten. Der TL-2 Compiler erzeugt hierzu ein Methodenpaar, bestehend aus einer `SlotAccessMethod` und einer `SlotUpdateMethod`, die das Lesen bzw. Schreiben einer Exemplarvariablen realisieren. Die Position der angesprochenen Exemplarvariable innerhalb des Objektes wird durch die Exemplarvariable `offset` bestimmt.

CSlotMethod

Objekte, die in ihrem Aufbau C-Strukturen darstellen, bedürfen spezieller Zugriffsmethoden, die eine Konvertierung der Datenwerte durchführen und dabei die Plattformabhängigkeit dieser Objekte berücksichtigen. Zusätzlich zu einer `SlotMethod` wird folgende Information benötigt:

`tycoonType`: Übergebener TL-2 Datentyp beim schreibenden Zugriff, bzw. erwarteter Datentyp beim Lesen. Der korrespondierende C-Datentyp wird aus den Deskriptortabellen des Objektspeichers ermittelt.

In Abbildung 3.11 sind die zulässigen Kombinationen von TL-2- und C-Datentypen dargestellt. Zu beachten ist, daß die Konvertierung dem *typecast* von C entspricht:

- Datentypen gleicher Größe: der Datenwert wird nicht verändert
- kleiner auf großen Typ: der Wert wird vorzeichenrichtig erweitert
- großer zu kleinerem Typ: höherwertige Bytes werden abgeschnitten

Bei gleich großen Datentypen können Seiteneffekte auftreten, wenn vorzeichenbehaftete in vorzeichenlose Werte und umgekehrt konvertiert werden. So wird z.B. bei der Umwandlung von einem `Int`-Objekt in eine vorzeichenlose Ganzzahl (`unsigned int`) aus einer negativen eine große positive Zahl.

Bisher fehlt in der Syntax der Sprache TL-2 die Möglichkeit, Klassen zu deklarieren, deren Exemplare C-Datenstrukturen sind. Für die vordefinierten Klassen dieses Typs erzeugt der TL-2 Compiler Zugriffsmethoden mit einer Standardkonvertierung, die in der Tabelle markiert ist.

C \ TL-2	Bool	Char	Int	Long	Real	Object
char (s/u)	*AU*	*AU* (d)	*AU*	*AU*		
short (s/u)	*AU*	*AU*	*AU*	*AU*		
int (s/u)	*AU* (d)	*AU*	*AU* (d)	*AU*		
long (s/u)	*AU*	*AU*	*AU*	*AU*		
long long	*AU*	*AU*	*AU*	*AU* (d)		
float					*AU*	
double					*AU* (d)	
void * (OID)						*AU* (d)

A: Access (C ➪ TL-2) *U*: Update (TL-2 ➪ C) (d): default

Abbildung 3.11: Konvertierungen von C/TL-2 Datentypen für C-Strukturobjekte

PoolMethod

Die Tycoon-Programmierumgebung verwaltet ein Verzeichnis systemweit zugreifbarer globaler Variablen (*pool*). Jede dieser definierten Variablen wird analog zu den Exemplarvariablen über ein Methodenpaar für den lesenden und schreibenden Zugriff angesprochen. Um die Sichtbarkeit aller Änderungen zu gewährleisten, verweisen die Methoden nur indirekt auf die Objekte.

Indirekte Verweise werden über Exemplare der Klasse `Cell` hergestellt, deren einzige Exemplarvariable das jeweilige Objekt referenziert. Das Paar aus `Access`- und `Update`-Methode liest diese Exemplarvariable bzw. überschreibt sie.

3.3.4 C-Aufrufschnittstelle

Die Aufrufschnittstelle für externe Funktionen bietet eine einfache, generische Möglichkeit, statische oder dynamische Bibliotheken an das Laufzeitsystem zu binden und deren Funktionalität der Tycoon-2 Programmierumgebung zugänglich zu machen. Neben dem Öffnen und Schließen der Bibliotheken und der Funktionssuche ist sie für die Konvertierung der Argumente und des Funktionsergebnisses verantwortlich. Details der Implementierung werden in Abschnitt 4.2.4 beschrieben.

In Abbildung 3.12 sind die von der C-Schnittstelle automatisch durchführbaren Konvertierungen dargestellt. Sind darüber hinausgehende Konvertierungen nötig, z.B. durch die Rückgabe strukturierter Werte oder Zeiger, so müssen die entsprechenden externen Methoden auf C-Seite in sogenannten *wrapper functions* gekapselt werden. Zur Art der Konvertierung gilt das unter Abschnitt 3.3.3 geschriebene.

C \ TL-2	Bool	Char	Int	Long	Real	(Mutable)String	Object
char (s/u)	*AR*	*AR*	*AR*	*AR*			
short (s/u)	*AR*	*AR*	*AR*	*AR*			
int (s/u)	*AR*	*AR*	*AR*	*AR*			
long (s/u)	*AR*	*AR*	*AR*	*AR*			
long long	*AR*	*AR*	*AR*	*AR*			
float					*AR*		
double					*AR*		
char *						*AR*	
void * (OID)							*A*
void			*R* (Funktionsergebnis wird ignoriert)				

A: Argument (TL-2 ➪ C) *R*: Rückgabewert (C ➪ TL-2)

Abbildung 3.12: Konvertierungen von C/TL-2 Datentypen der C-Aufrufschnittstelle

3.3.5 Funktionen höherer Ordnung

In der Programmiersprache TL-2 sind Funktionen Objekte erster Klasse, d.h. es ist möglich, Funktionsausdrücke überall dort zu verwenden, wo TL-2 Ausdrücke erlaubt sind. Funktionen können auf Variablen ihres statischen Sichtbarkeitsbereichs zugreifen. Da sie jedoch eine uneingeschränkte Lebensdauer besitzen, können sie ihren Kontext verlassen. Sie bestehen, wie in Abbildung 3.13 dargestellt, aus diesem Grund aus zwei Komponenten, einem Funktionsobjekt und einem Funktionskörper.

Der Funktionskörper (`CompiledFun`) ist der statische Teil und wird bereits zur Übersetzungszeit vom Compiler generiert. Das Funktionsobjekt (*closure*) wird dynamisch zur Laufzeit erzeugt. Es fixiert den Funktionskörper zusammen mit eingegangenen Bindungen an freie Variable. Da änderbare Bindungen von allen betroffenen Funktionen und allen ihren Aufrufen gemeinsam benutzt werden, müssen diese, wie im Fall der globalen Variablen (siehe Abschnitt 3.3.3), indirekt referenziert werden. Funktionsobjekte verstehen eine spezielle Aufruf-Nachricht, die den Funktionskörper in der um die Argumente erweiterten eingefangenen Umgebung aufruft.

Zur Erzeugung von Funktionen höherer Ordung existieren vier spezielle Bytecodes:

`closure`: Erzeugt ein Funktionsobjekt.

`cellNew`, `cellLoad`, `cellStore`: Erzeugen und Manipulieren indirekte Verweise auf änderbare freie Variable mittels `Cell`-Objekten.

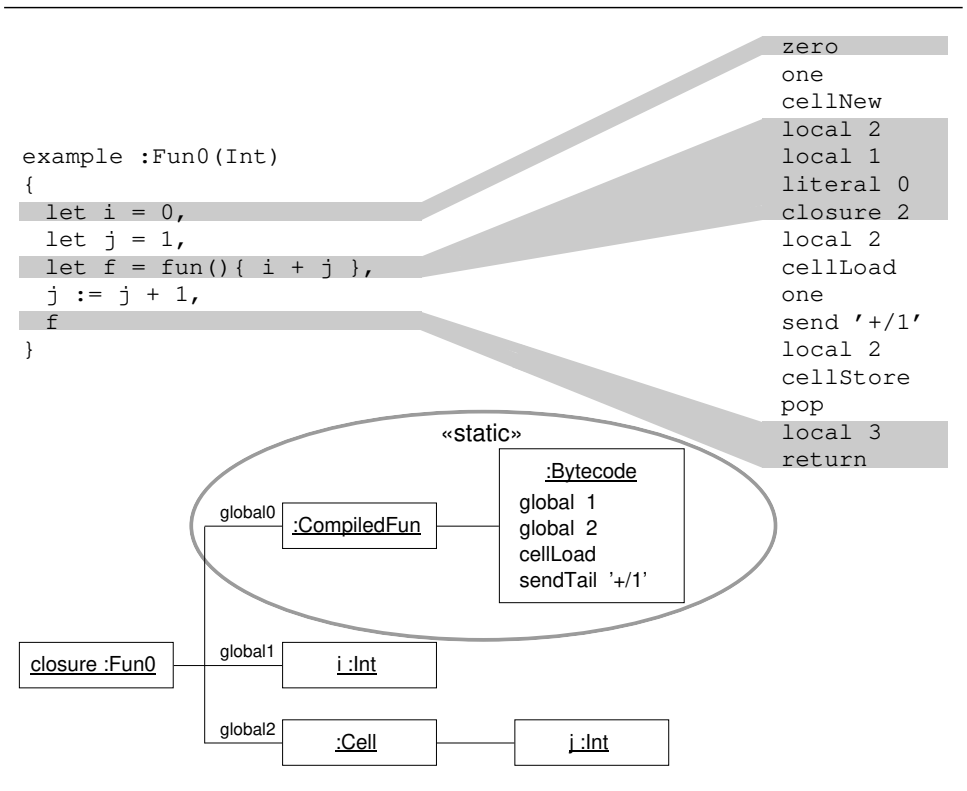

Abbildung 3.13: Funktion höherer Ordnung

3.3.6 Ausnahmebehandlung

Bei der Ausführung von Code können Ausnahmezustände auftreten, sei es programmgesteuert durch Senden des entsprechenden Primitivs an ein Ausnahmeobjekt oder durch Verletzung maschineninterner Vorgaben. TL-2 erlaubt mittels des `try`-Konstrukts das kontrollierte Abfangen und Verarbeiten solcher Ausnahmesituationen (vgl. Abschnitt 4.2.5). Abbildung 3.14 veranschaulicht dies anhand eines Beispiels.

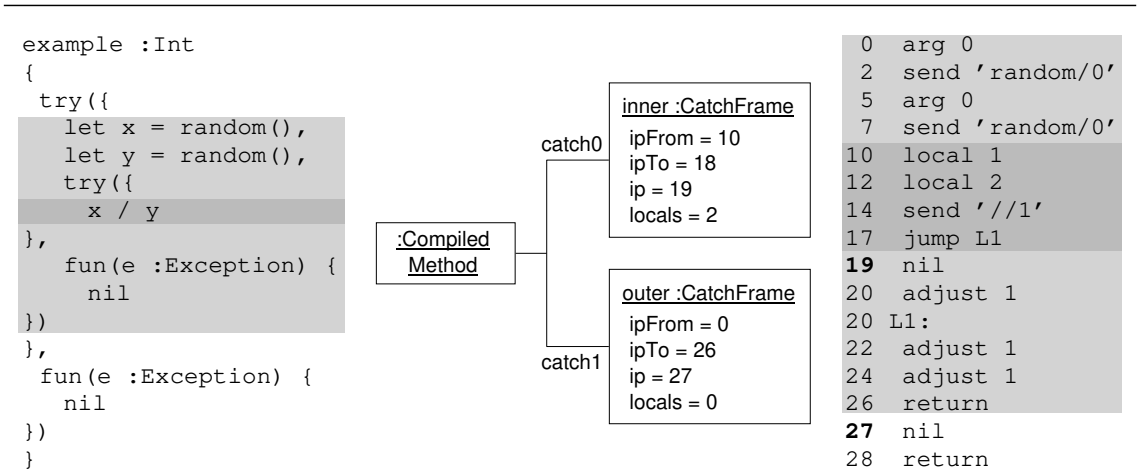

Abbildung 3.14: Beispiel geschachtelter Ausnahmebehandlungen

Methoden, für die eine oder mehrere Ausnahmebehandlungen existieren, besitzen eine Tabelle sogenannter `CatchFrames`. Jeder dieser Frames beschreibt einen Bytecodebereich, für den eine Ausnahmebehandlung definiert ist, wohin im Fall einer Ausnahme verzweigt wird und wie viele lokale Variablen mit Eintritt in die entsprechende Routine auf dem Laufzeitstapel liegen:

`ipFrom, ipTo`: Bytecode Start- und Endadresse, innerhalb der Ausnahmen abgefangen werden.

`ip`: Bytecode Startadresse der Ausnahmebehandlung.

`locals`: Anzahl Worte auf dem Laufzeitstapel.

Die Frames sind entsprechend ihrer Bytecodeposition geordnet, bei geschachtelten Ausnahmen erfolgt die Sortierung so, daß innen liegende Ausnahmen vor den äußeren eingereiht werden. Dies ermöglicht es, durch eine lineare Suche die passende Ausnahmebehandlung zu finden.

Ein zusätzlicher Aufwand zur Laufzeit entsteht bei dem gewählten Verfahren nur durch die beim Auftreten einer Ausnahme notwendige Suche nach einer passenden Ausnahmebehandlung. Hier liegt der Vorteil im Vergleich zum Prototypen, der einen eigenen Stapelspeicher für Ausnahmen verwendet. Mit jedem Betreten und Verlassen eines geschützten Bereichs muß der Prototyp einen Ausnahmekontext auf den Stapelspeicher legen bzw. wieder entfernen.

4. Implementierung

In diesem Kapitel werden ausgewählte Schwerpunkte der Implementierung betrachtet, insbesondere die Objektallokation und Speicherbereinigung des Objektspeichers (Abschnitte 4.1.2 und 4.1.4) sowie die Methodensuche der virtuellen Maschine (Abschnitt 4.2.2).

4.1 Objektspeicher

4.1.1 Objekte

Die Metainformationen eines Objekts werden, wie in Abbildung 4.1 dargestellt, in einem Objektkopf vor dem Datenbereich gehalten. Für die Implementierung der Speicherbereinigung werden zusätzlich zu den in Abschnitt 3.2.4 definierten Felder weitere Markierungen benötigt:

`size/forward`: Die Objektgröße oder, wenn das Objekt von der Speicherbereinigung kopiert wurde, ein Verweis auf die Objektkopie.

`classId`: Die 12 Bit breite Klassennummer.

`hash`: Der 14 Bit breite Hashwert.

`f`: Markierung, ob das Feld `size/forward` als Objektgröße oder Objektverweis zu interpretieren ist.

`w`: Markierung für schwach referenzierte Objekte.

Die Anzahl verschiedener Klassen im System wird durch die 12 Bit breite Klassennummer auf 4096 begrenzt. In der Praxis stellt dieser Wert keine Einschränkung dar; das aktuelle System mit allen Applikationen verfügt über weniger als 1000 Klassen. Zudem kann die Maximalzahl durch die Nutzung der noch freien Bits erhöht werden.

Der Hashwert eines Objektes wird bei der Allokation aus der initialen `OID` gewonnen. Dieses Verfahren erfordert keinen Aufwand zur Laufzeit und hat sich bezüglich der Gleichverteilung aller Hashwerte als ausreichend erwiesen.

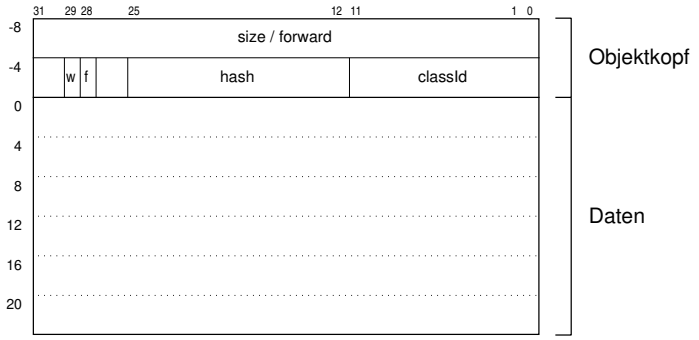

Abbildung 4.1: Aufbau eines Objekts

4.1.2 Objektallokation

Neue Objekte werden in der aktuellen Speicherseite des Objektspeichers, die in den globalen Verwaltungsinformationen vermerkt ist, alloziert. Die verschiedenen `_new`-Funktionen für die einzelnen Layouttypen (siehe Abschnitt 3.2.5) aus der Aufrufschnittstelle basieren dabei alle auf einer generischen Routine, die auch von der Speicherbereinigung bei der Erzeugung von Objektkopien benutzt wird. Ihr Aktivitätsdiagramm ist in Abbildung 4.2 dargestellt.

Die von der Allokationsroutine erwarteten Argumente sind die Klassennummer des neuen Objekts sowie die Größe seines Datenbereichs. Im ersten Schritt wird der insgesamt erforderliche Speicherplatz berechnet, bestehend aus Objektkopf und Datenbereich, erweitert bis zur nächsten Langwortgrenze, um die Speicherausrichtung eventuell folgender Allokationen zu gewährleisten (siehe Abschnitt 3.2.4). Abhängig von der errechneten Größe können zwei Situationen auftreten:

1. In der aktuellen Seite ist genügend freier Speicher vorhanden: Der Freizeiger der Speicherseite wird um den berechneten Betrag weitergesetzt und das neue Objekt initialisiert. Der Datenbereich wird gelöscht und der Objektkopf mit den übergebenen Argumenten – Klassennummer und Objektgröße – sowie dem Hashwert angelegt. In der Praxis hat sich ein Hashwert aus Bit 4 – 17 der neuen `OID` hinsichtlich einer gleichmäßigen Verteilung aller Hashwerte als ausreichend erwiesen.

2. Die aktuelle Seite muß gewechselt werden: Die Seitenallokation (siehe Abschnitt 4.1.3) initialisiert die für das neue Objekt nötige Anzahl freier, aufeinanderfolgender Speicherseiten und setzt die aktuelle Speicherseite auf die erste Seite des belegten Bereichs. Anschließend wird wie im ersten Fall der Freizeiger weitergesetzt, bei einem großen Objekt explizit auf das Seitenende, so daß keine weitere Allokation aus den Speicherseiten dieses Objekts erfolgen kann, und das Objekt wird initialisiert. Die bei einem großen Objekt auf die erste Seite folgenden Seiten vom Typ `Continued` (siehe Abschnitt 3.2.3) gelten automatisch als vollständig belegt und zur ersten Seite gehörig.

Als Resultat liefert die Allokationsroutine die `OID` des neuen Objekts.

Von der Objektallokation verworfene Speicherseiten kommen unabhängig von ihrer Belegung für weitere Allokationen nicht mehr in Betracht. Dieses aus Gründen der Effizienz gewählte Vorgehen bedingt, daß ein Teil des Speicherplatzes ungenutzt bleibt. Bei der im System gewählten Seitengröße von 512 Byte und einer durchschnittlichen Objektgröße von 27 Byte konnte im Mittel eine Füllung der Speicherseiten von 85 – 90 Prozent beobachtet werden.

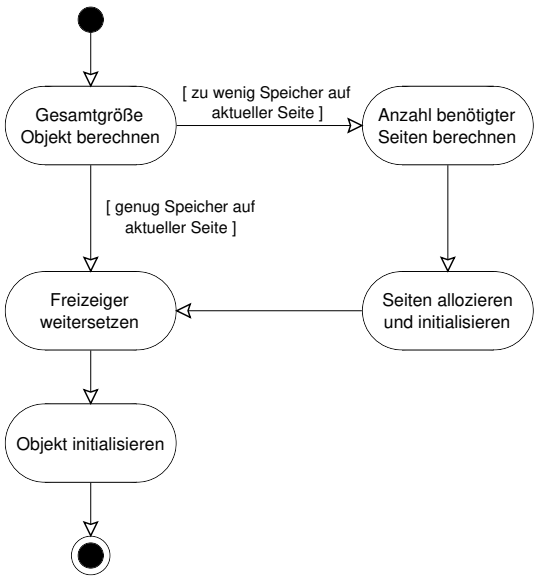

Abbildung 4.2: Objektallokation

4.1.3 Seitenallokation

Aufgrund der seitenorientierten Verwaltung des Objektspeichers ist die Allokation neuer Objekte mit der Suche nach freien, aufeinanderfolgenden Speicherseiten und deren Initialisierung verbunden. Diese Funktionalität wird von einer Routine (siehe Abbildung 4.3) bereitgestellt, die bei Speichermangel darüberhinaus für die Auslösung der Speicherbereinigung verantwortlich ist.

Ausgehend von der aktuellen Seite wird der Objektspeicher linear durchsucht. Die Allokation einer Sequenz von freien Seiten wird hierbei durch die Fragmentierung des Objektspeichers, d.h. einer Zersplitterung der freien Seiten in kleine Teilbereiche (*cluster*), erschwert. Hervorgerufen wird diese Fragmentierung durch belegte Seiten, die große Objekte enthalten oder Objekte, auf die externe Referenzen existieren und die daher nicht von der Speicherbereinigung freigegeben werden dürfen (siehe Abschnitt 4.1.4), sowie Lücken innerhalb des Objektspeicherbereichs durch fremde Seiten (`Foreign`). Der Vorgang terminiert, sobald die

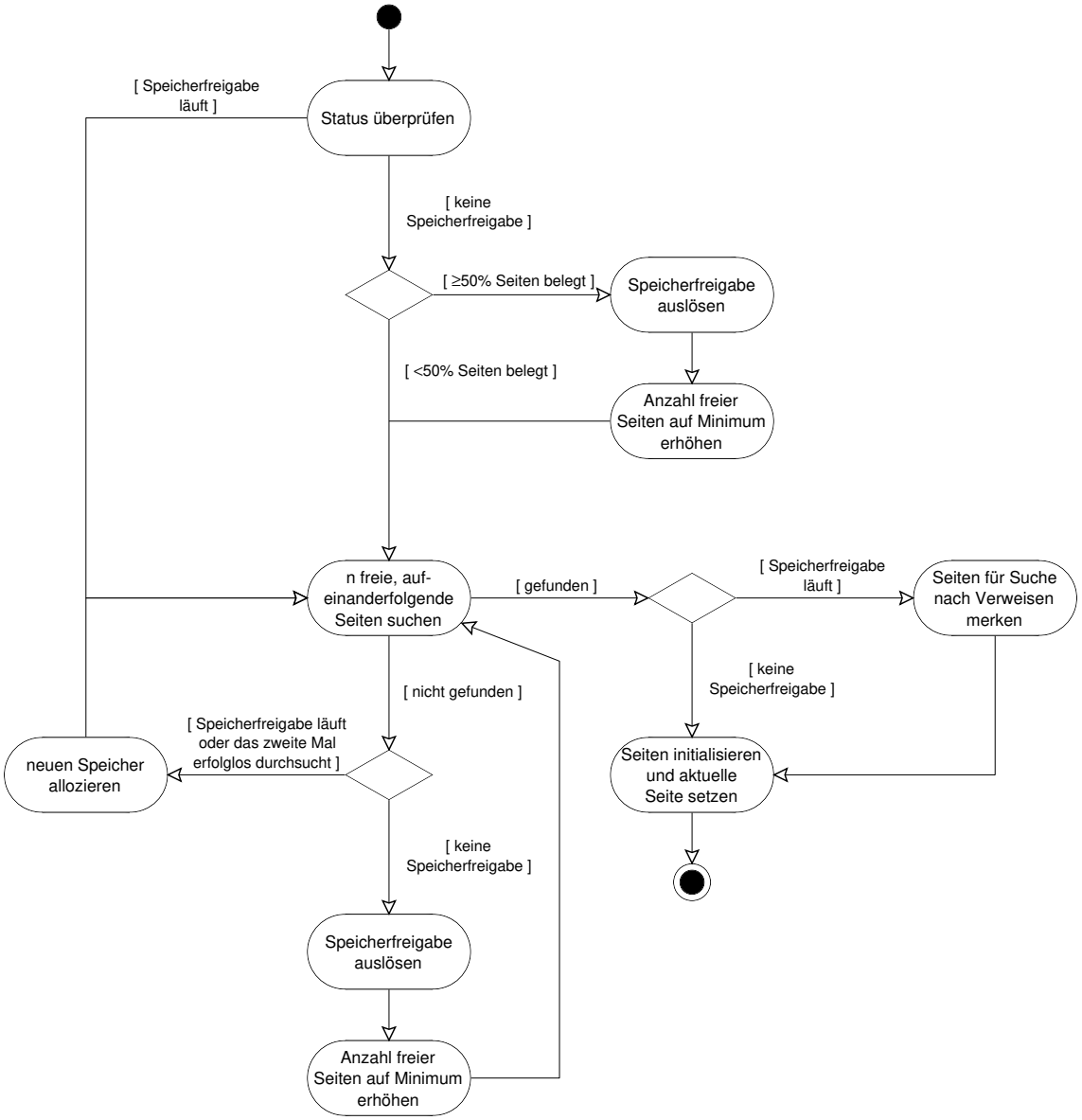

Abbildung 4.3: Seitenallokation

gewünschte Zahl freier Speicherseiten gefunden oder der gesamte Objektspeicher einmal erfolglos durchsucht wurde.

Die gefundenen Seiten werden wie folgt initialisiert: Die erste Speicherseite erhält den Typ `Object` und ihr Freizeiger wird auf den Seitenanfang gesetzt. Gleichzeitig wird sie zur aktuellen Speicherseite für die folgende Objektallokation und bei laufender Speicherbereinigung

für die Suche nach weiteren Objektverweisen vermerkt. Alle im Fall eines großen Objekts folgenden Seiten werden als `Continued` markiert. Der Bereichsidentifikator (`age`) aller Speicherseiten wird auf den Wert der globalen Zustandsvariable `newAge` gesetzt. So werden alle während der Speicherbereinigung allozierten Seiten dem neuen Speicherbereich der Objektkopien zugeordnet (siehe Abschnitte 3.2.3 und 4.1.4).

Die Speicherbereinigung kann durch zwei Bedingungen aktiviert werden:

1. Wenn bei Eintritt in die Funktion bereits die Hälfte oder mehr Seiten belegt sind. Diese obere Schranke für die maximale Objektspeicherbelegung ergibt sich aus der Arbeitsweise der Speicherbereinigung, alle referenzierten Objekte zu kopieren. Sie benötigt im ungünstigsten Fall (*worst case*), wenn kein Objekt freigegeben werden kann, den doppelten Speicherplatz.
2. Wenn alle Speicherseiten einmal erfolglos durchsucht wurden. Diese Situation kann auftreten, wenn insgesamt nicht genügend freie Seiten vorhanden sind oder aufgrund einer zu großen Fragmentierung des Objektspeichers durch fremde (`Foreign`) oder bereits belegte Seiten kein freier, linearer Bereich ausreichender Größe gefunden werden kann.

Voraussetzung hierzu ist, daß die Speicherbereinigung nicht bereits läuft, um rekursive Aufrufe zu verhindern. Der Arbeitsmodus des Objektspeichers wird in einer globalen Zustandsvariablen angezeigt. Die explizite Manipulation dieser Variablen über die Aufrufschnittstelle des Objektspeichers (`_inhibitGc, _allowGc`) erlaubt es, die Speicherbereinigung während kritischer Programmphasen, z.B. der Systeminitialisierung, zu unterbinden.

Nach erfolgter Speicherbereinigung wird der verfügbare freie Speicherplatz, wenn nicht genügend Seiten freigegeben wurden, auf einen einstellbaren Betrag erweitert (4 MB in der aktuellen Implementierung). Auf diese Weise ist sichergestellt, daß die Systemperformanz nicht unnötig durch eine zu hohe Frequenz von Speicherbereinigungen gesenkt wird.

Ist die Suche nach freien Seiten trotz Speicherbereinigung nicht erfolgreich oder kommt es während der Speicherbereinigung zu einem Mangel an freien Seiten, ein Zustand, zu dem eine zu geringe Seitenfüllung oder starke Fragmentierung führen kann, muß der Objektspeicher vergrößert werden. Über das Betriebssystem wird ein der nötigen Seitenzahl entsprechender und, um die Effizienz zu erhöhen, mindestens 1 MB großer, linearer Speicherbereich alloziert und initialisiert.

4.1.4 Speicherbereinigung

Eine wesentliche Gemeinsamkeit des Großteils aller allozierten Objekte ist ihre geringe Lebensdauer. Die Speicherbereinigung sorgt für die automatische Freigabe des von nicht mehr benötigten, d.h. unreferenzierten, Objekten reklamierten Speicherplatzes. Ihre prinzipielle Arbeitsweise ist in Abschnitt 3.2.7 beschrieben, im folgenden werden das Zusammenspiel mit den höheren Systemebenen und die Handhabung schwacher Referenzen eingehender betrachtet.

Zusammenspiel mit der virtuellen Maschine

In Abbildung 4.4 ist die Kommunikation des Objektspeichers mit der virtuellen Maschine während der Speicherbereinigung beispielhaft dargestellt. Dabei werden zwei externe Referenzen und ein zusätzliches Objekt enumeriert.

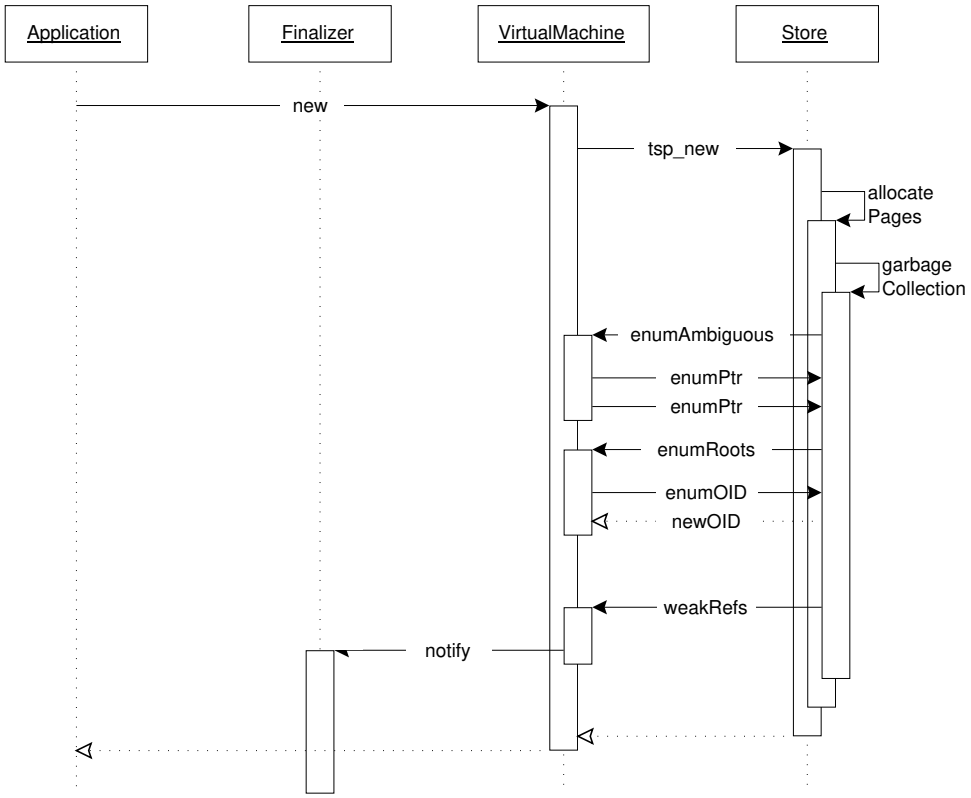

Abbildung 4.4: Zusammenspiel von Objektspeicher und virtueller Maschine

Ausgangspunkt der Speicherbereinigung ist eine aufgrund von Speichermangel fehlgeschlagene Objektallokation (vgl. Abschnitt 4.1.3). Einzige Ausnahme ist die `Commit`-Operation, die im Vorfeld versucht, den Objektspeicher zu verdichten. Es werden dabei folgende Phasen durchlaufen:

1. Die Aktivierung der Speicherbereinigung wird in einer globalen Zustandsvariablen vermerkt und der Identifikator für den Speicherbereich der Objektkopien (`newSpace`) weitergesetzt. Ab jetzt werden alle von der Seitenallokation belegten Seiten dem neuen Teilbereich zugeordnet.

2. Die registrierte Enumeratorfunktion für Hauptspeicheradressen wird aufgerufen. Für

jede von der virtuellen Maschine aufgezählte Adresse, die auf eine belegte Objektspeicherseite verweist, wird die entsprechende Seite direkt in den neuen Bereich übernommen. Ist die Seite Bestandteil eines großen Objekts, werden alle weiteren zum Objekt gehörenden Seiten ebenfalls übernommen.

3. Das Wurzelobjekt wird kopiert.

4. Die Enumeratorfunktion für Objekte wird aufgerufen. Alle so enumerierten Objekte werden kopiert.

5. Die schwachen Referenzen werden kopiert. Hierzu verwaltet der Objektspeicher alle schwachen Referenzen separat in einer doppelt verketteten Liste.

6. Alle Objekte in den Speicherseiten des neuen Teilbereichs werden nach weiteren Objektverweisen durchsucht. Dabei werden die jeweils referenzierten Objekte, falls noch nicht geschehen, kopiert und die Verweise angepaßt (siehe Abbildung 4.5). Der neue Bereich wächst so während der Suche dynamisch an. Von der Seitenallokation neu belegte Seiten werden in einer FIFO (*first in first out*) Warteschlange für den weiteren Suchvorgang vermerkt.

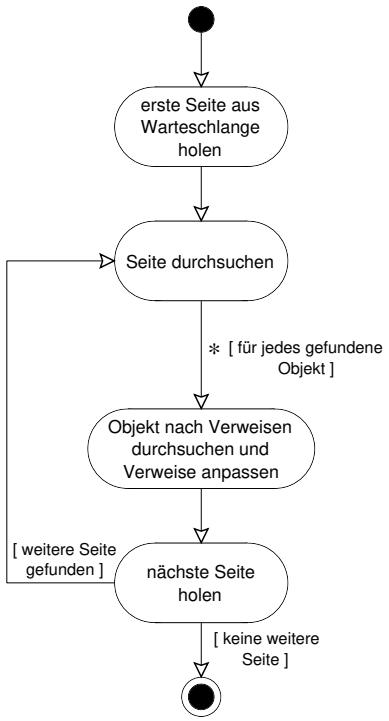

Abbildung 4.5: Suche nach Objektverweisen

7. Schwach referenzierte Objekte, auf die keine weiteren Verweise existieren, können nach erfolgter Kopie des Objektgraphen anhand einer Markierung im Objektkopf (siehe Abschnitt 4.1.1) erkannt werden. Die schwachen Referenzen, die auf diese Objekte verweisen, werden aus der internen Liste des Objektspeichers in eine neue Liste umgesetzt, die über die virtuelle Maschine an das TL-2 System zur Finalisierung, d.h. dem Aufruf der assoziierten benutzerdefinierten Funktion, weitergereicht wird. Durch diese Übergabe wird die Liste mit ihrer transitiven Hülle Teil des Objektgraphen. Die in ihr verzeichneten schwachen Referenzen und die von diesen aus erreichbaren Objekte werden erst in einer folgenden Speicherbereinigung freigegeben, wenn die zuständige Instanz, der *Finalizer*, den letzten Verweis auf diese Objekte, den Listeneintrag, mit Beendigung der assoziierten TL-2 Funktion löscht und die Erreichbarkeit innerhalb der assoziierten Funktion nicht wiederhergestellt wird.

8. Zum Abschluß wird die globale Zustandsvariable zurückgesetzt und der Identifikator für den alten Teilbereich weitergesetzt (`space = newSpace`). Hiermit werden alle alten Speicherseiten freigegeben.

Nach Beendigung der Speicherbereinigung erfolgt die Allokation neuer Objekte ab der ersten freien Speicherseite. In Verbindung mit einem kleinen persistenten Objektgraphen und vielen temporären Objekten erhöht dies die Lokalität der Zugriffe, da so ein Teil der Speicherseiten ausschießlich als Reserve vorgehalten wird. Insbesondere ermöglicht dies der virtuellen Speicherverwaltung des Betriebssystems, diese Seiten dauerhaft auszulagern.

Kopieren von Objekten

Das Kopieren des persistenten Objektgraphen aus dem alten in den neuen Teilbereich ist gleichzeitig mit der Anpassung aller Objektverweise (`OID`s) auf die neuen Hauptspeicheradressen und der Markierung schwach referenzierter Objekte verbunden. Jeder während der Suchphase der Speicherbereinigung gefundene Verweis wird einer Routine übergeben, die diese Aufgaben übernimmt. Dabei wird unterschieden, ob es sich um einen regulären Verweis handelt oder ob die `OID` das Objekt nur schwach referenziert.

Wie in Abbildung 4.6 ersichtlich müssen drei Fälle bei Übergabe eines Objektverweises unterschieden werden:

1. Die Speicherseite des Objekts befindet sich im neuen Teilbereich. Es ändert sich nichts.

2. Das Objekt wurde bereits kopiert. Im Objektkopf des alten Objektes findet sich ein Verweis auf die Kopie.

3. Das Objekt liegt noch im alten Teilbereich. Bei einem großen Objekt, das seine Speicherseiten exklusiv belegt, ist das Kopieren unnötig. Alle zugehörigen Seiten werden direkt in den neuen Bereich überführt. Ein kleines Objekt wird kopiert, indem über die

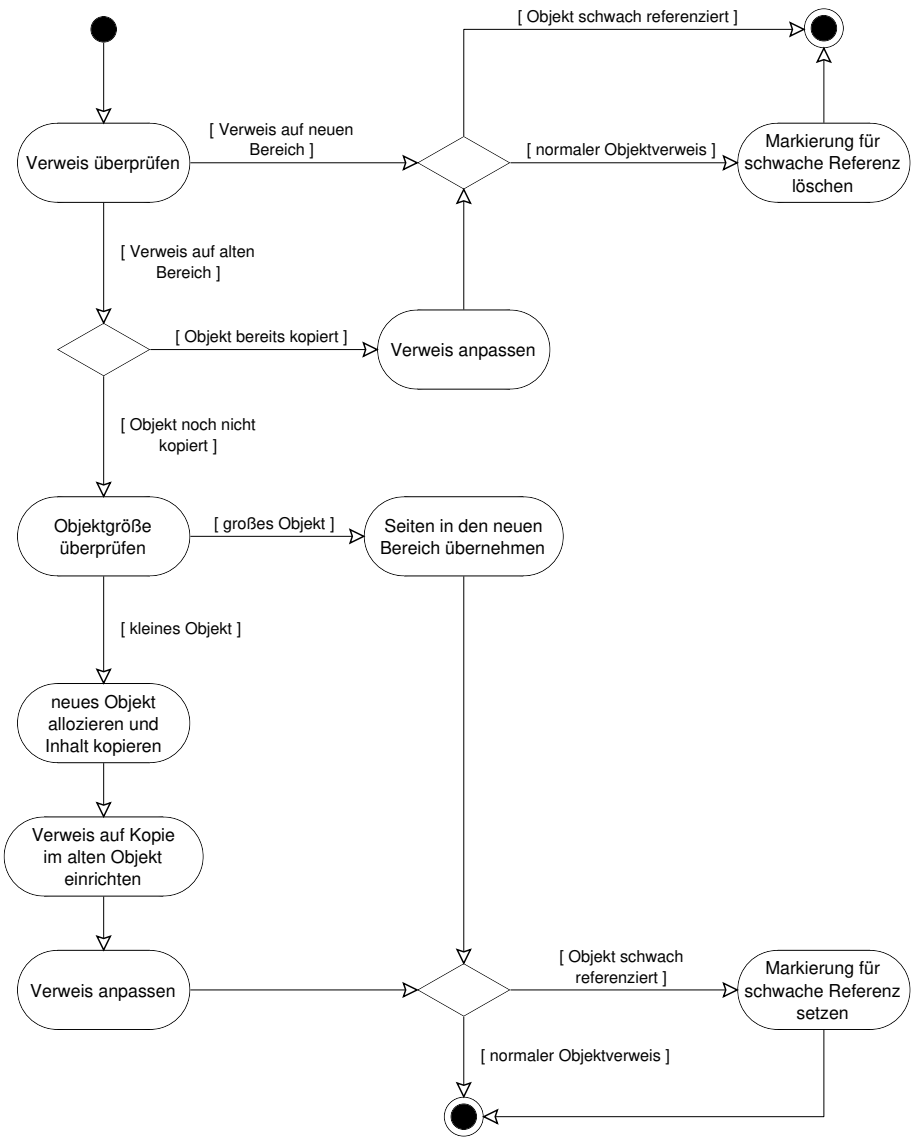

Abbildung 4.6: Kopieren von Objekten

Objektallokation ein gleichartiges Objekt im neuen Bereich angelegt und sein Datenbereich sowie die relevanten Informationen des Objektkopfs, Größe, Klassennummer und Hashwert, übernommen werden. Im Objektkopf des alten Objekts wird ein Verweis auf die Objektkopie hinterlassen, so daß bei allen weiteren Aufrufen mit derselben `OID` Fall 2 eintritt.

Rückgabewert ist die unveränderte `OID`, wenn sich das Objekt durch Promotion seiner Spei-

cherseite bereits im neuen Bereich befindet, oder der gültige Verweis auf die Objektkopie.

Schwach referenzierte Objekte, auf die keine weiteren Verweise existieren, werden anhand eines Markierungsbits im Objektkopf identifiziert. Dieses Bit wird genau dann gesetzt, wenn ein Verweis aus einer schwachen Referenz heraus existiert und sich das entsprechende Objekt noch im alten Teilbereich befindet. Normale Verweise löschen die Markierung. In der Konsequenz bleibt das Bit also gesetzt, wenn das Objekt nur schwach referenziert wird.

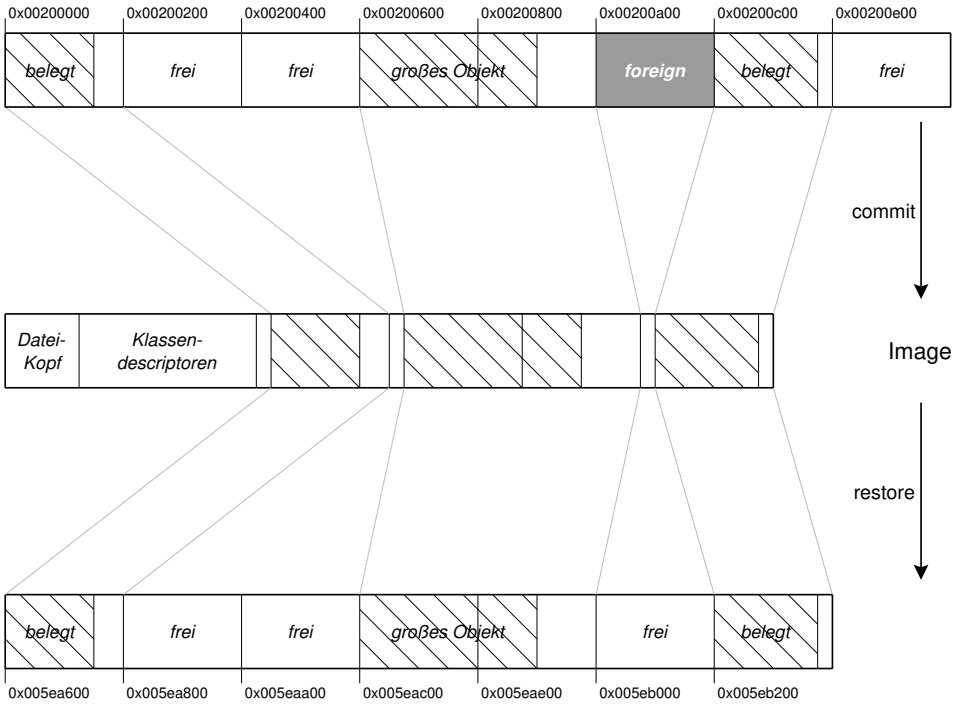

Abbildung 4.7: Sicherung und Wiederherstellung des Objektspeicherzustands

4.1.5 Persistenz

Die Persistenz des Tycoon-2 Systems wird durch die beiden atomar ausgeführten Methoden `commit` und `restore` realisiert. Abbildung 4.7 illustriert den Vorgang des Sicherns und anschließenden Restaurierens des Objektspeicherzustands:

`commit()`: Es werden alle belegten Seiten mit ihren Deskriptoren sowie weitere Verwaltungsinformationen des Objektspeichers in eine Datei geschrieben.

`restore()`: Die Verwaltungsinformationen und Speicherseiten werden restauriert. Der Seitenspeicher zwischen der ersten und letzten gesicherten Seite wird wieder in den Ursprungszustand versetzt, d.h. die Lücken wiederhergestellt, indem ein genügend großer

Dateikopf		
b0 - b15	hostname	Rechnerarchitektur
w0	magic number	Identifikator
w1	store version	Versionsnummer des Objektspeichers
w2	first page	Nummer der ersten gesicherten Seite
w3	last page	Nummer der letzten gesicherten Seite
w4	used pages	Anzahl gesicherter Seiten
w5	class descriptors	Anzahl gesicherter Klassendeskriptoren (n)
w6	pageblocks	Anzahl folgender Seitenblöcke (m)
w7	root object	Verweis auf das Wurzelobjekt
w8	weak references	Verweis auf erste schwache Referenz
w9 - w15	reserved	

n Klassendeskriptoren (ab w1 nur für C-Strukturen)		
w0	object layout	Layouttyp des Deskriptors
w1	elements	Anzahl der Strukturelemente
w2	size	Größe der C-Struktur
w3-w...	offsets	Offsets ab Strukturbeginn
b0-b...	types	Beschreibung der Strukturelemente

m Seitenblöcke (ab b512 nur für große Objekte)		
w0	number	Seitennummer
w1	pages	Anzahl folgender Speicherseiten (Seiten für große Objekte werden in einem Stück geschrieben)
w2	filled	Füllung der ersten Seite
w3	reserved	
b0 - b511	page contents	Speicherseite
b512 - b...	page contents	weitere Speicherseiten bei großem Objekt

Tabelle 4.1: Dateiformat eines Objektspeicherabbildes

linearer Speicherbereich, evtl. zuzüglich der für die Speicherbereinigung nötigen Reserve, vom Betriebssystem angefordert wird.

Vor dem Sichern wird eine Speicherbereinigung, die auch große Objekte kopiert, durchgeführt, um die Dateigröße zu minimieren und die belegten Seiten zu verdichten, d.h. die Fragmentierung zu verringern.

Alle Daten werden im Format der aktuellen Rechnerarchitektur geschrieben; die Datei besteht ausschließlich aus 8 und 32 Bit Worten. Die `Restore`-Operation ist, da die Speicher-

zuteilung dynamisch durch das Betriebssystem erfolgt, für die Anpassung der Objektverweise und bei der Migration auf eine andere Plattform zusätzlich für die Konvertierung der Daten verantwortlich. Hintergrund dieser Entscheidung ist die Tatsache, daß `Commit`-Operationen während der Laufzeit regelmäßig, eventuell in kurzen Zeitabständen, erfolgen und eine möglichst hohe Performanz aufweisen sollen, der Objektspeicherzustand jedoch nur einmal beim Systemstart restauriert wird.

Besondere Aufmerksamkeit verdient die Konvertierung von C-Strukturobjekten, z.B. beim Wechsel von der SPARC zur Intel x86 Architektur, da die einzelnen Datenfelder neu im Speicher ausgerichtet werden müssen. Der Objektspeicher kennt hierzu die Repräsentation der benutzten Datentypen (*size*, *byteorder*, *alignment*) auf allen zum Einsatz kommenden Plattformen. Zudem muß er, da sich C-Strukturen auf verschiedenen Architekturen in ihrer Größe unterscheiden können und die Objektgröße nicht dynamisch änderbar ist, bereits beim Anlegen eines solchen Objekts den maximal nötigen Speicherplatz reservieren.

Die Dateistruktur ist in Tabelle 4.1 dargestellt. Sie gliedert sich in drei Teile:

1. Der Dateikopf. Er enthält die globalen Verwaltungsinformationen und dient der Versionskontrolle.
2. Die Klassendeskriptoren.
3. Die belegten Seiten mit ihren Deskriptoren. Im Fall eines großen Objekts werden die von ihm belegten Seiten in einem Stück mit nur einem Deskriptor geschrieben.

4.2 Virtuelle Maschine

4.2.1 Zusammenspiel von TL-2 und virtueller Maschine

Bereits im Entwurf wird die enge Bindung von Objektspeicher, virtueller Maschine und der TL-2 Programmierumgebung hervorgehoben. Von besonderer Bedeutung sind hierbei die gemeinsam genutzten Datenstrukturen, von denen ein großer Teil durch C-Strukturobjekte repräsentiert ist (siehe Abschnitt 3.2.6). Dies erlaubt es dem in C implementierten Laufzeitsystem, direkt auf die einzelnen Datenfelder zuzugreifen; auf der anderen Seite stellen diese Strukturen vollwertige TL-2 Objekte dar.

Abbildung 4.8 illustriert anhand der beiden Klassen `Method` und `CompiledMethod` die beiden unterschiedlichen Sichten. Die Gegenüberstellung mit der TL-2 Definition verdeutlicht folgendes:

- Die C-Struktur ahmt die Vererbung der TL-2 Klasse nach.
- Die Sicht der virtuellen Maschine beschränkt sich auf die Exemplarvariablen, d.h. sie kann selbständig keine Methoden aufrufen, obwohl der Interpreter für die Ausführung von Methoden zuständig ist.

TL-2 Klassen und C-Strukturen müssen von Hand synchronisiert werden. Insbesondere erfordern Änderungen an den Exemplarvariablen bzw. Strukturelementen einen neuen *Bootstrap* der TL-2 Programmierumgebung und die erneute Übersetzung der virtuelle Maschine.

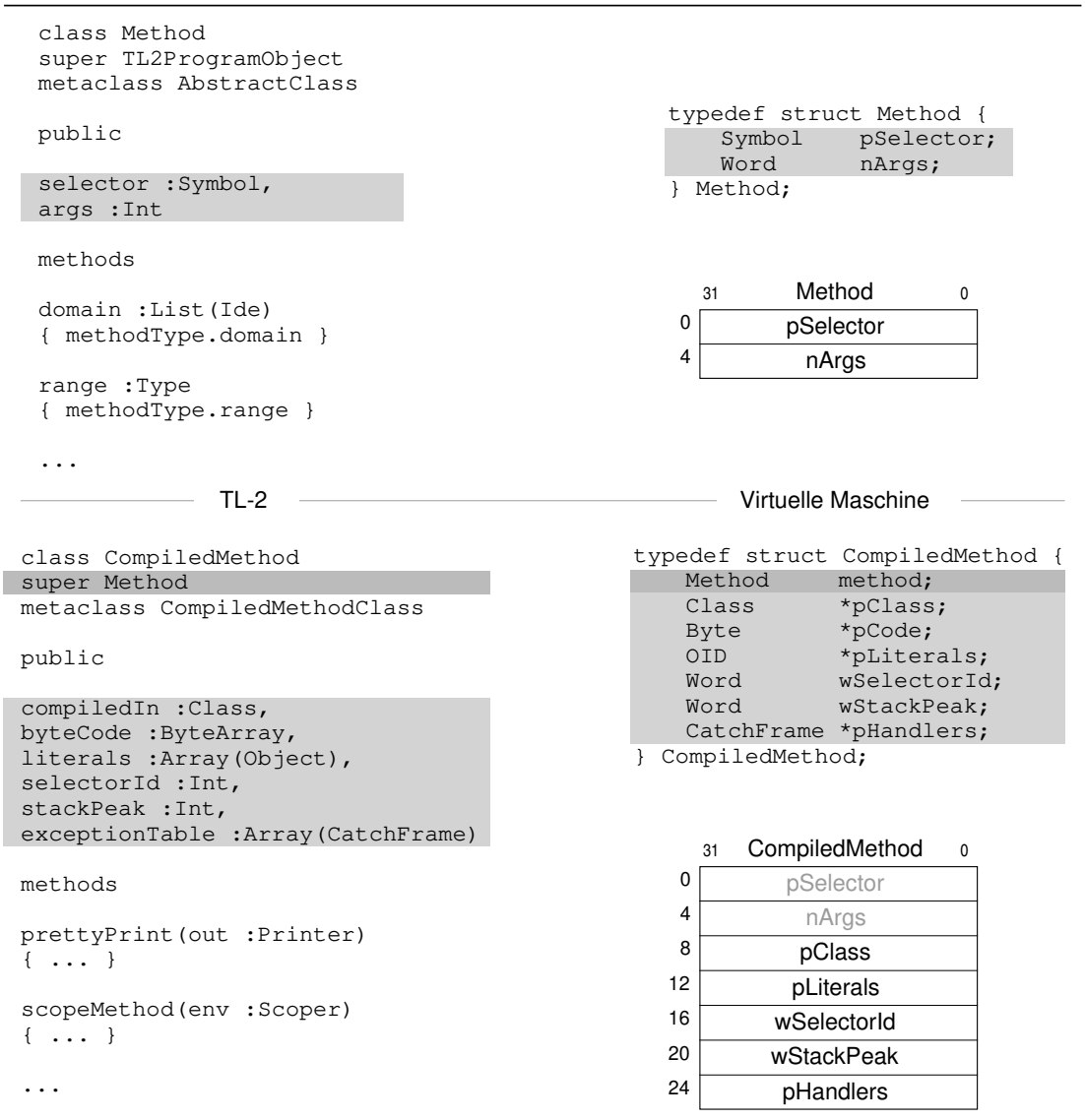

Abbildung 4.8: Gemeinsame Nutzung von Objekten

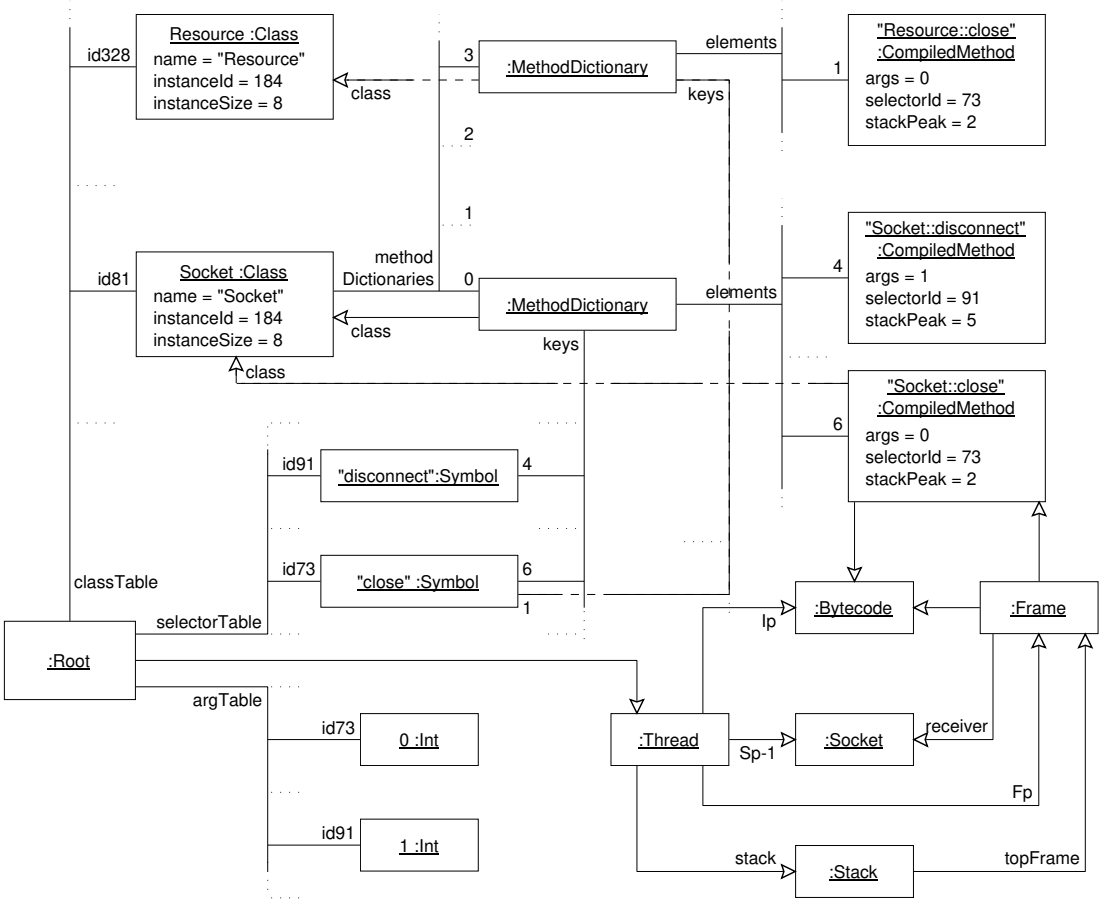

Abbildung 4.9: Datenstrukturen des Interpreters

4.2.2 Methodensuche

Das Versenden von Nachrichten ist die zentrale Operation des Tycoon-2 Systems. Die zur Laufzeit durchgeführte Methodensuche bindet die gesendeten Nachrichten dynamisch an die gefundenen Implementierungen. Dabei müssen hinsichtlich der Methodensuche zwei Varianten unterschieden werden:

`send`: Die Methodensuche beginnt am Anfang der Klassenpräzedenzliste des Empfängerobjekts (siehe Abschnitt 2.1).

`sendSuper`: Da geerbte Methoden redefiniert werden können, erlaubt der `sendSuper`-Bytecode die Suche nach Methodenimplementierungen in Vorgängerklassen. Die Methodensuche beginnt in der Klassenpräzedenzliste des Empfängerobjekts hinter der Klasse, in der sich die Implementierung der aktuell ausgeführten Methode befindet.

Im folgenden wird der interne Ablauf beim Versenden einer Nachricht erläutert. Einen besonderen Schwerpunkt bilden die Zusammenhänge zwischen den einzelnen Datenstrukturen der virtuellen Maschine.

Datenstrukturen

Die Übersicht in Abbildung 4.9 zeigt beispielhaft die vom Wurzelobjekt aus erreichbaren Datenstrukturen. Abgesehen vom Laufzeitstapel und Threadobjekt werden alle Datenstrukturen vom TL-2 System erzeugt und vom Bytecode-Interpreter nur gelesen. Insbesondere ist das übergeordnete Programmiersystem (der TL-2 Compiler und die Klassenverwaltung) für die Korrektheit der Daten, z.B. die Linearisierung der Methodenverzeichnisse entsprechend der Klassenpräzedenzliste und die eindeutige Vergabe von Selektornummern etc., verantwortlich.

send

Der Ausgangspunkt eines Methodenaufrufs stellt sich wie folgt dar (siehe Abbildung 4.10): Der Instruktionszähler verweist auf den `send`-Bytecode, ihm folgt die 16 Bit breite Selektornummer. Auf dem Laufzeitstapel befinden sich, das Empfängerobjekt zuerst, die von den vorangegangenen Bytecodes abgelegten Argumente.

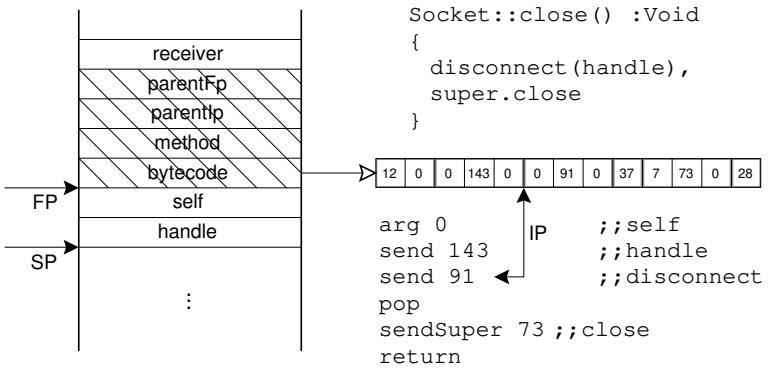

Abbildung 4.10: Beispielszenario vor einem Methodenaufruf

Abbildung 4.11 verdeutlicht anhand der Pfeile, wie sich der Interpreter in den weiteren Schritten durch die Datenstrukturen bewegt. Direkt zugänglich sind die Maschinenregister und das Wurzelobjekt:

1. Die Selektornummer wird aus dem Bytecode geladen.

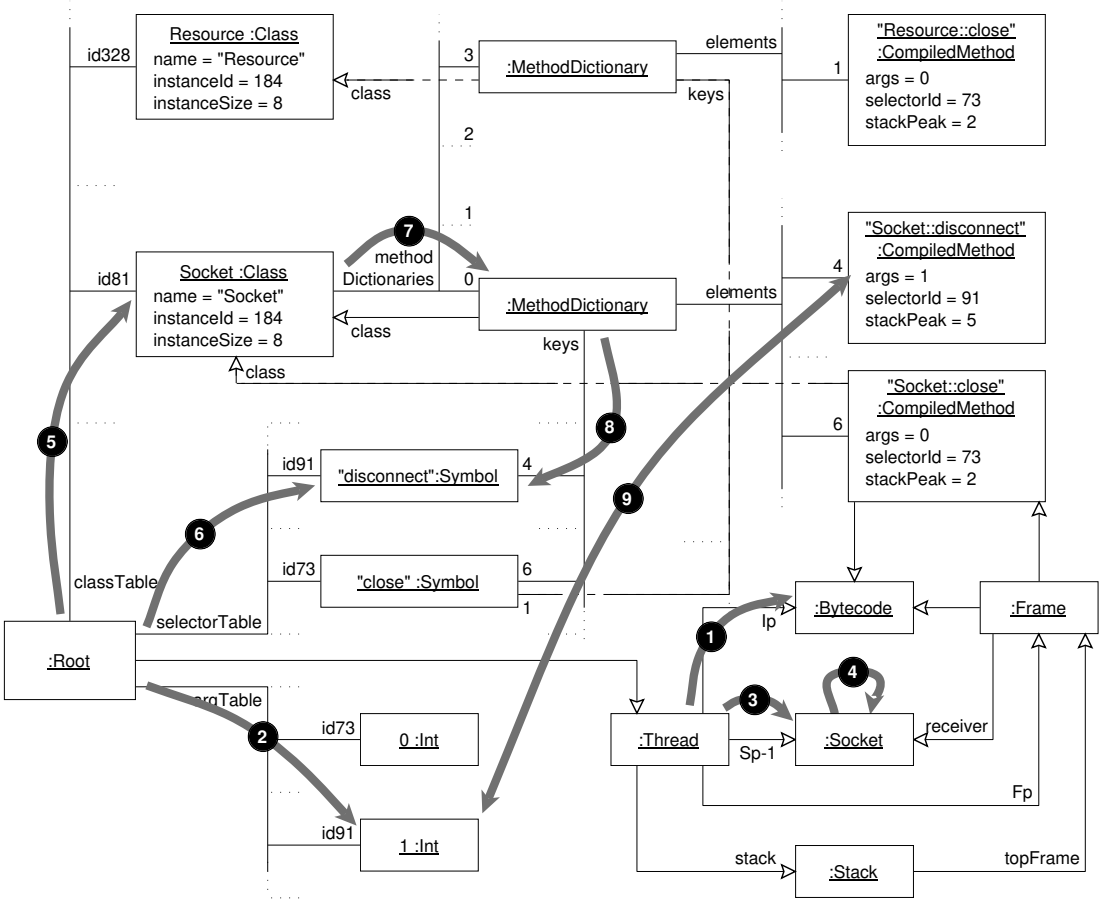

Abbildung 4.11: Methodensuche des Interpreters

2. Die Anzahl der Argumente auf dem Laufzeitstapel wird anhand der Selektornummer aus der Argumenttabelle der Wurzelobjekts ermittelt. Dieser Schritt entfällt für `send`-Bytecodes, die ihre Argumentzahl fest kodieren.

3. Das Empfängerobjekt wird vom Laufzeitstapel geholt.

4. Die Klassennummer des Empfängers wird über die Aufrufschnittstelle des Objektspeichers aus dem Objektkopf gelesen. Bei markierten Ganzzahlen oder `nil` wird eine vorgegebene Nummer benutzt.

5. Mit der Klassennummer wird mittels der Klassentabelle das Klassenobjekt des Empfängers geladen.

6. Der zu suchende Selektor wird aus der Selektortabelle gelesen.

7. Die Liste der im Klassenobjekt registrierten Methodenverzeichnisse wird linear nach einer passenden Methodenimplementierung durchsucht. Wird kein Methodenobjekt gefunden, das einen identischen Selektor (8.) und gleiche Argumentzahl (9.) besitzt, wird eine Ausnahme (`DoesNotUnderstand` bzw. `WrongSignature`) ausgelöst.

Ist die Methodensuche erfolgreich, wird das gefundene Methodenobjekt auf seine Klasse überprüft: Bei Methoden vom Typ `CompiledMethod` wird ein neuer Sicherungsrahmen aufgebaut und der Methodenrumpf, d.h. der Bytecode, ausgeführt. Alle anderen Methodenarten werden direkt von Routinen des Interpreter bearbeitet.

sendSuper

Die Suche nach Methodenimplementierungen in Vorgängerklassen unterscheidet sich in ihrem Vorgehen nur in den letzten Schritten vom `send`-Aufruf (siehe Abbildung 4.12):

1. – 6. wie oben.
7. Über den Sicherungsrahmen auf dem Laufzeitstapel wird das Klassenobjekt, in dem die aktuelle Methode definiert wurde, geladen.
8. In der Liste der Methodenverzeichnisse werden alle Einträge einschließlich des Verzeichnisses, in dem die aktive Methode registriert ist, übersprungen. Dies geschieht durch einen Vergleich des in Schritt 7 ermittelten Klassenobjekts mit dem vom jeweiligen Methodenverzeichnis referenzierten Klassenobjekt.
9. Die restlichen Methodenverzeichnisse werden wie im Fall eines `send`-Opcodes nach einer passenden Methodenimplementierung durchsucht.

4.2.3 Methodencache

Die dynamische Methodensuche ist in einem rein objektorientierten System wie Tycoon-2 der größte Engpaß für den Durchsatz (*performance bottleneck*) des Interpreters. Als eine Möglichkeit zu seiner Beseitigung bietet es sich an, einmal gefundene Methodenobjekte zwischenzuspeichern (*caching*). Wiederholte Nachrichten mit demselben Selektor an Exemplare derselben Klasse finden die zugehörigen Methodenobjekte bereits im Cache vor und die Methodenverzeichnisse müssen nicht erneut durchsucht werden.

Der Interpreter setzt für die beiden unterschiedlichen Varianten der Methodensuche zwei getrennte Methodencaches ein. Die Caches sind Hashtabellen und enthalten Verweise auf Methodenobjekte. Die Schlüssel setzen sich aus der Klassennummer des Empfängerobjekts und der Selektornummer der versandten Nachricht zusammen, beim `sendSuper` kommt als zusätzlicher eindeutiger Identifikator für den Beginn der Methodensuche die Exemplarvariable `InstanceClassId` aus dem Klassenobjekt, in dem die aktuelle Methode definiert ist,

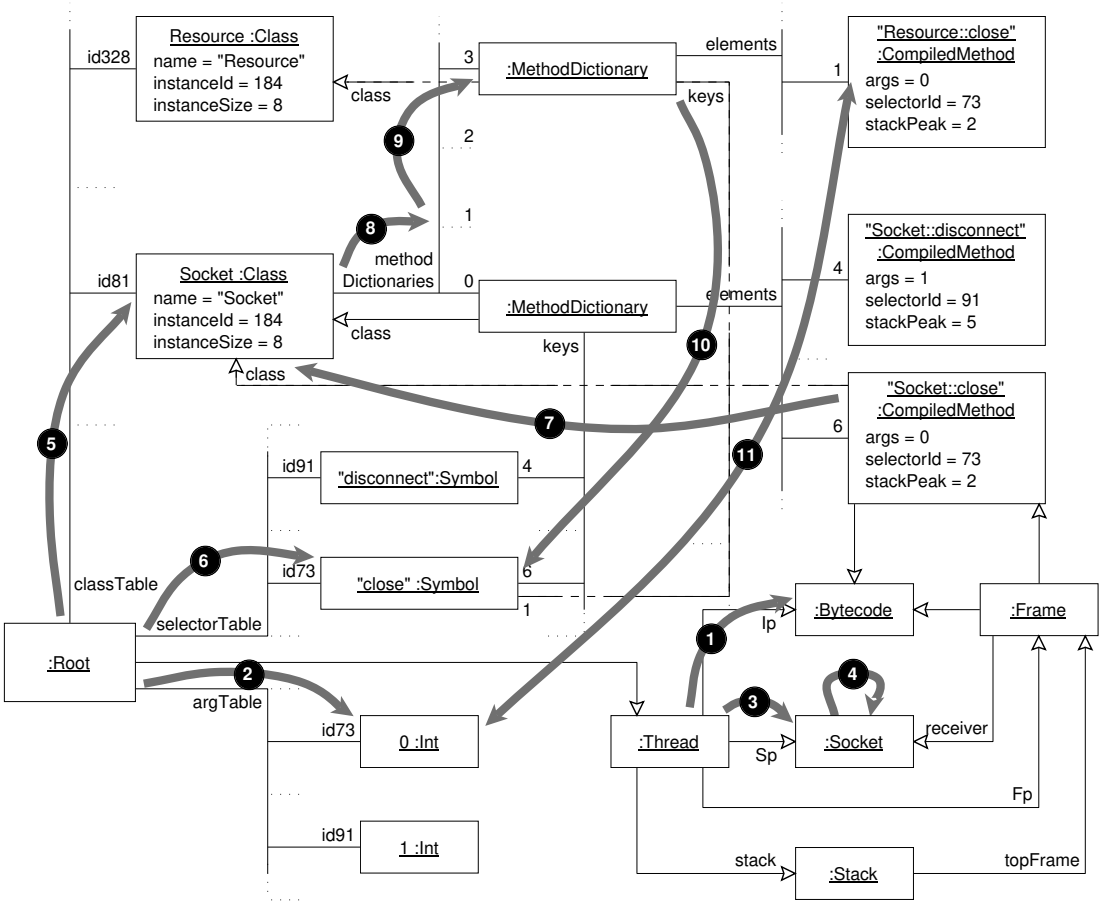

Abbildung 4.12: Suche nach Methodenimplementierungen in Vorgängerklassen

hinzu. Der Hashwert für den Tabellenzugriff wird aus diesen Komponenten durch bitweises Verschieben und bitweise exklusive Oder-Verknüpfungen gebildet.

Werden Änderungen an Klassen und Methoden vorgenommen, so müssen die Caches geleert werden. Die TL-2 Programmierumgebung teilt dies der virtuellen Maschine durch die elementaren Methoden `_flushAll`, `_flushClass` und `_flushSingle` mit.[1] Da die Caches Datenstrukturen außerhalb des Objektspeichers sind, werden mit Aktivierung der Speicherbereinigung alle in ihnen eingetragenen Verweise auf Methodenobjekte über die Enumerationsfunktion für Objektverweise (siehe Abschnitt 3.2.7) angepaßt.

Als Erweiterung des Algorithmus auf Seite 63 werden zwei zusätzliche Schritte zwischen den Schritten 4 und 5 eingefügt:

[1] Die selektiven Varianten sind noch nicht vollständig implementiert; die Caches werden bisher immer vollständig gelöscht.

a. Der Hashwert wird berechnet und im Cache nachgesehen.

b. Bei einem Treffer wird das Methodenobjekt aus dem Cache geladen und ausgeführt.

Ein erfolgloser Zugriff resultiert in einer normalen Methodensuche, an deren Ende der Cache mit den gefundenen Daten gefüllt wird. Es wird eine verdrängende Strategie eingesetzt, d.h. alte Einträge werden überschrieben. Die aktuelle Implementierung benutzt einen Cache für `send`-Bytecodes mit 2^{13} Einträgen; der Cache für `sendSuper`-Bytecodes ist aufgrund ihrer geringen Häufigkeit um den Faktor vier kleiner dimensioniert. Trotz der einfachen Arbeitsweise wird hiermit eine Trefferrate von über 98% erzielt. Im Vergleich zu einem System ohne Methodencache ergibt sich eine durchschnittliche Steigerung der Systemperformanz um den Faktor 3.

4.2.4 C-Aufrufschnittstelle

Das Tycoon-2 System ist durch eine hohe Interoperabilität mit externen Diensterbringern gekennzeichnet, wie bei der Nutzung elementarer Betriebssystemdienste (Dateisystem, Netzwerkdienste), aber auch der Bindung an kommerzielle Systeme, z.B. relationale Datenbanken (Adabas, Oracle) oder betriebliche Informationssysteme (SAP R/3). Allen gemeinsam ist die Art, wie die Dienstleistungen angeboten werden: als statische oder dynamische Bibliotheken, die in TL-2 Programme eingebunden werden müssen und ihre Funktionalität über eine C-Schnittstelle zur Verfügung stellen.

Dynamische Bibliotheken

Externe Bibliotheken, sogennannte `DLL`s (*dynamic link libraries*), werden durch TL-2 Objekte repräsentiert. Die `DLL`-Klasse definiert folgende Exemplarvariablen und Basismethoden:

Exemplarvariablen:

`handle`: Identifikator, der vom Betriebssystem bei erfolgreichem Öffnen der Bibliothek vergeben und von allen weiteren Betriebssystemfunktionen benötigt wird.

`path`: Zur Laufzeit ermittelter Dateiname der Bibliothek.

Methoden:

`open()`: Bibliothek zur Benutzung öffnen.

`close()`: Bibliothek schließen.

`resolve()`: Gültigen Bibliotheksnamen zur Laufzeit ermitteln. Diese Methode wird vor dem Öffnen einer Bibliothek implizit aufgerufen, da je nach System unterschiedliche Dateikennungen verwendet werden, z.B. `.so` unter Solaris und Linux, `.sl` unter HPUX.

Der Aufruf von Funktionen wird über externe Methoden in den jeweiligen `DLL`-Subklassen und die C-Schnittstelle der virtuellen Maschine realisiert (siehe Abschnitte 3.3.3 und 3.3.4).

Abbildung 4.13 zeigt als Beispiel die Gegenüberstellung eines Auszug aus der C-Prototypendatei bzw. der TL-2 Bibliotheksklasse für die Datenbankschnittstelle OCI (*Oracle Call Interface*). Zeiger auf externe Datenstrukturen werden auf TL-2 Seite als 32 Bit breite Ganzzahlen (`Int`) dargestellt.

```
class OCI
super DLL
metaclass SimpleConcreteClass(OCI)

public
...

methods
...

(* Terminating a Connection *)

ologof (lda:Int) :Int extern

(* Preparing Requests *)

oopen(cursor:Int, lda:Int, dbn:String, dbnl:Int, arsize:Int,
        uid:String, uidl:Int) :Int extern
oparse(cursor:Int, sqlstm:String, sqllen:Int, defflg:Int,
        lngflg:Int) :Int extern

(* Submitting Requests *)

oexec(cda:Int) :Int extern

(* Terminating a Statement *)

ocan(cda:Int) :Int extern
oclose(cda:Int) :Int extern
...
```

```
/*
 * Declare the OCI functions.
 * Prototype information is included.
 * Use this header for ANSI C compilers.
 */

#ifndef OCIAPR
#define OCIAPR

#include <oratypes.h>
#include <ocidfn.h>

...

sword  ologof(struct cda_def *lda);

sword  oopen(struct cda_def *cursor, struct cda_def *lda,
                text *dbn, sword dbnl, sword arsize,
                text *uid, sword uidl);
sword  oparse(struct cda_def *cursor, text *sqlstm,
                sb4 sqllen, sword defflg, ub4 lngflg);

sword  oexec(struct cda_def *cursor);

sword  ocan(struct cda_def *cursor);
sword  oclose(struct cda_def *cursor);
...

#endif /* OCIAPR */
```

Abbildung 4.13: Gegenüberstellung von C-Prototypen und TL-2 Bibliotheksklasse

Der wesentliche Vorteil dynamischer Bibliotheken besteht in einer z.T. deutlichen Speichereinsparnis, da von ihnen nur jeweils eine Instanz im System existiert, die von allen Prozessen gemeinsam benutzt wird. Die zu ihrer Einbindung nötigen Betriebssystemfunktionen sind auf Systemen, die das ELF-Format unterstützen (u.a. Solaris und Linux):

`void * dlopen(const char * pathname, int mode)`: Dynamische Bibliothek laden und öffnen.

`int dlclose(void * handle)`: Bibliothek schließen.

`void * dlsym(void * handle, const char * name)`: Über den Funktionsnamen, der in der Symboltabelle der Bibliothek verzeichnet ist, die Funktionsadresse ermitteln.

Andere Systeme besitzen äquivalente Funktionalität, die sich nur in der bereitgestellten Schnittstelle unterscheidet. Fehler, die vom Betriebssystem bei der Nutzung signalisiert werden, erzeugen Ausnahmen des Laufzeitsystems.

Die TL-2 Seite unterscheidet nicht zwischen statischen und dynamischen Bibliotheken. Für statische Bibliotheken wird der `DLL`-Mechanismus vom Laufzeitsystem nachgebildet. Alle statischen Bibliotheken und die von ihnen bereitgestellten Funktionen werden in eine Tabelle eingetragen, die zusammen mit der virtuellen Maschine übersetzt wird. Die Bibliotheken werden statisch an das ausführbare Programm gebunden, jede Änderung ist daher an die Generierung einer neuen Maschine gekoppelt. Dieses Vorgehen ist prinzipiell nicht auf statische Bibliotheken beschränkt. Auch dynamische Bibliotheken können so eingebunden werden, um bereits zur Übersetzungszeit Fehler, z.B. fehlende Funktionen, zu erkennen. Zusätzlich müssen die so eingesetzten Bibliotheken nicht erst über das Betriebssystem geöffnet werden. Dies wird für zwei essentiell wichtige Bibliotheken, `ANSI-C` und `RTS` (*runtime support*), ausgenutzt, die bereits in der Initialisierungsphase des TL-2 Systems benötigt werden und deren Namensauflösung (`resolve`) wechselseitige Abhängigkeiten erzeugen würde.

Parameterübergabe

Die C-Schnittstelle führt eine automatische Typkonvertierung zwischen den TL-2 Methoden- und den C-Funktionsargumenten durch (vgl. Abschnitt 3.3.4). Auf allen Plattformen ergibt sich dabei die Schwierigkeit, die Argumente korrekt an die jeweilige Funktion zu übergeben und das gültige Resultat zu erhalten. Naturgemäß kann die Konvertierung aufgrund der großen Zahl an Kombinationen bei den Argumenttypen und unterschiedlichen Funktionssignaturen nicht durch statische, spezialisierte Routinen übernommen werden. Es existieren jedoch Aufrufkonventionen, die einen generalisierten Ansatz zulassen. Dies wird am Beispiel der Sparc-Architektur unter dem Betriebssystem Solaris erläutert. Die folgenden Angaben stammen aus [Sun95].

Alle Argumente, mit Ausnahme der Datentypen *long long* (64 Bit Ganzzahl) und *double* (64 Bit Gleitkommazahl), werden als 32 Bit Werte übergeben. Die Übergabe der ersten sechs 32 Bit Werte erfolgt in den Registern %o0 bis %o5, 64 Bit breite Werte zählen dabei als zwei 32 Bit Werte und belegen zwei aufeinanderfolgende Register. Weitere Argumente werden auf den Stack geschrieben.

Funktionsergebnisse werden in folgenden Registern zurückgegeben:

- %o0 bei 32 Bit Ganzahlen
- %o0 und %o1 bei 64 Bit Ganzzahlen
- %f0 bei 32 Bit Gleitkommazahlen

- %f0 und %f1 bei 64 Bit Gleitkommazahlen

Die Argumente werden nach erfolgter Konvertierung typkorrekt auf 32 bzw. 64 Bit erweitert und unter Beachtung der Registergrenze, da 64 Bit Werte nicht teilweise im letzten Register und auf dem Stack übergeben werden dürfen, in einen linearen Speicherbereich geschrieben. Um das korrekte Resultat zu erhalten, existiert für jeden möglichen Ergebnistyp ein eigener Funktionsaufruf, der den kompletten Speicherbereich als Argumentliste erhält. Überzählige, uninitialisierte Speicherworte stören hierbei nicht, da die Argumente in umgekehrter Reihenfolge von rechts nach links auf den Stack geschrieben werden und die aufgerufene Funktion diese nicht sieht, da die aufrufende Funktion für die Bereinigung des Stacks verantwortlich ist.

Die anderen Plattformen, für die bereits Portierungen vorliegen, folgen diesem Schema. Dabei müssen u.U. architekturspezifische Besonderheiten beachtet werden. So ist auf PARISC-Prozessoren die Übergabe von 64 Bit Werten nur in bestimmten Registerpaaren zulässig, zudem müssen dabei die beiden 32 Bit Hälften vertauscht werden.

Die Grenzen dieser generischen Lösung sind erreicht, sobald auf einer Plattform die Argumente abhängig von ihrem Typ (Ganz- oder Fließkommazahl, 32 oder 64 Bit Breite) in verschiedenen Registern erwartet werden. Eine Alternative besteht in diesem Fall darin, die Konvertierungs- und Aufruffunktionen für eine externe Methode bei Bedarf zur Laufzeit in Maschinencode zu generieren (*just in time*).

4.2.5 Ausnahmebehandlung

Mit der Unterstützung von Ausnahmen stellt das Laufzeitsystem der TL-2 Umgebung einen effizienten Mechanismus zur Verfügung, Fehler kontrolliert abzufangen. Grundlage bilden die in Abschnitt 3.3.6 beschriebenen `CatchFrames`, die einer TL-2 Methode eine beliebige Anzahl von Ausnahmebehandlungen zuordnen.

Tritt eine Ausnahme auf, wird der Laufzeitstapel der virtuellen Maschine beginnend mit der gerade aktiven Methode nach der ersten passenden Ausnahmebehandlung durchsucht. Weist eine Methode keine entsprechende Behandlung auf, so wird sie vom Stapel entfernt und mit der aufrufenden Methode fortgefahren; der Laufzeitstapel wird also schrittweise abgebaut. Sobald eine Behandlung gefunden wurde, werden die Maschinenregister entsprechend des gefundenen `CatchFrame`s gesetzt und die Programmausführung im Interpreter fortgesetzt. Kann eine Ausnahme nicht abgefangen werden, so wird der entsprechende TL-2 Thread beendet.

Neben den benutzerdefinierten Ausnahmen, die über eine elementare Methode ausgelöst werden, verfügt die virtuelle Maschine über eigene Ausnahmen. Mit diesen signalisiert sie interne Fehlerzustände und leitet die Bearbeitung an die Hochsprache weiter. Ein Problem ergibt sich dabei aus der Tatsache, daß diese Ausnahmen von verschiedenen Moduln, z.B. der C-Schnittstelle oder dem dynamischen Linker, ausgelöst werden können, zu ihrer Behandlung aber nach Manipulation des Kontexts der virtuellen Maschine wieder in

den Interpreter zurückgekehrt werden muß. Erschwerend kommt hinzu, daß der Interpreter auf lokalen Maschinenregistern arbeitet und nicht rekursiv aufgerufen werden darf, da ansonsten überzählige Speicherworte auf dem Prozessorstapel liegenbleiben, der so mit jeder Ausnahme unkontrolliert wächst, und den korrekten Programmfluß gefährden.

Die Variante, den Kontext nur innerhalb des Interpreters zu ändern und aus aufgerufenen Moduln mit Fehlercodes zurückzukehren, erweist sich als zu umständlich. Eine einfachere Lösung besteht darin, neben dem Kontext der virtuellen Maschine, dem Thread- und dem Stackobjekt, auch den Kontext des Prozesses bzw. Threads zu manipulieren und nach jeder Ausnahme den Interpreter komplett neu zu starten. Hierfür bieten sich die Betriebssystemfunktionen `setjmp` und `longjmp` an:

`setjmp` sichert den Ausführungskontext eines Programms bzw. Threads, d.h. die Prozessorregister, in eine Datenstruktur.

`longjmp` restauriert den gesicherten Kontext und fährt mit der Ausführung so fort, als ob der Aufruf des vorangegangenen `setjmp` gerade erfolgt wäre. Ein Argument, das an `longjmp` übergeben und als Funktionsergebnis des `setjmp` Aufrufs durchgereicht wird, ermöglicht die Unterscheidung, ob ein Kontext gesichert oder wiederhergestellt wurde.

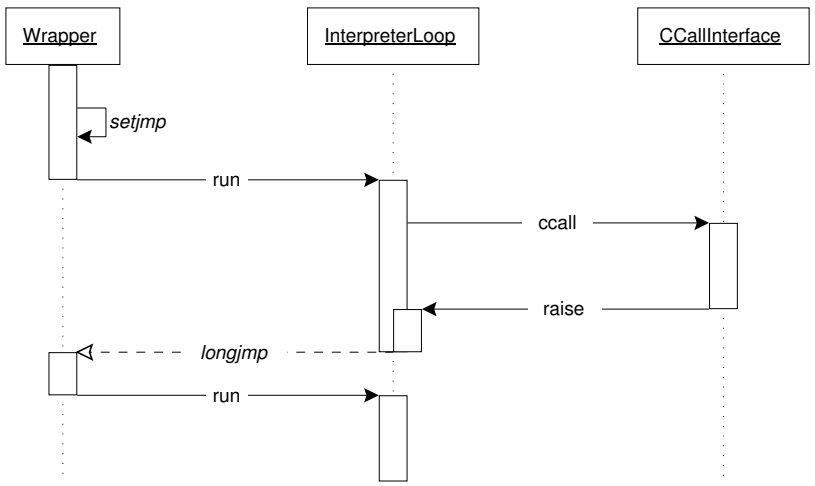

Abbildung 4.14: Ausnahmebehandlung mit `setjmp` und `longjmp`

Abbildung 4.14 demonstriert das Vorgehen anhand eines Beispiels. Die Interpreterschleife wird von einer Funktion gekapselt (`run`), die am Ende der Systeminitialisierung genau vor dem Start des Interpreters (`runint`) den Prozeßkontext mittels `setjmp` einfängt. Im weiteren Verlauf verzweigt der Interpreter in die C-Aufrufschnittstelle (`ccall`), die, z.B. aufgrund einer unzulässigen Argumentkonvertierung, eine Ausnahme auslöst. Die Suche nach einer

passenden Ausnahmebehandlung und Änderung des Thread- und Stackobjekts geschieht in einer separaten Routine (`exception`), die zum Abschluß den gesicherten Prozeßkontext wiederherstellt. Der Interpreter wird neu gestartet und beginnt mit der Abarbeitung der gefundenen Ausnahmebehandlung.

5. Multithreading und Multiprozessorunterstützung

Für den Einsatz von Tycoon-2 im kommerziellen Bereich, beispielsweise als Server im World Wide Web, muß das System mehrere Threads innerhalb eines Prozesses unterstützen. Hierzu bedarf es einer effizienten Lösung, die u.a. ein Blockieren des Gesamtsystems bei synchronen I/O-Operationen verhindert und die auch in der Lage ist, mehrere Prozessoren zur Steigerung der Performanz zu verwenden. Anders als im Tycoon-System, das einen eigenen kooperativen Ansatz mittels Coroutinen verwirklicht [Pie96], ist die Nutzung von Betriebssystemdiensten erforderlich. Um trotzdem portabel zu sein, stützt sich die virtuelle Maschine auf den POSIX.1c Standard[1], der auch als Pthread [IEE96] bekannt ist. Dieser Teil des POSIX-Gesamtstandards umfaßt ausschließlich Threads und ihr API, läßt jedoch die Implementierung bezüglich einiger Details offen, so daß auch bei seiner Verwendung z.B. keine Mehrprozessorunterstützung garantiert werden kann [NBF96; NM97].

Dieses Kapitel behandelt den Entwurf der nötigen Basisfunktionalität (Abschnitt 5.1.1) und das auf dieser aufbauende Management für externe TL-2 Ressourcen (Abschnitt 5.1.2). Besondere Beachtung finden dabei die Probleme, die sich aus der Erweiterung der in den vorigen Kapiteln vorgestellten virtuellen Maschine um Threads ergeben, z.B. die Synchronisierung des nicht reentranten Objektspeichers (Abschnitt 5.2.1). Weiterhin wird die Realisierung des dritten Aspekts der orthogonalen Persistenz[2] von Tycoon-2 betrachtet, die Sicherung aller Ausführungskontexte und ihre Wiederherstellung beim Hochfahren des Systems (Abschnitt 5.2.5).

5.1 Entwurf

Das Tycoon-2 System besitzt kein Transaktionskonzept für den Zugriff und die Manipulation gemeinsam genutzter Objekte. Auf Ebene der virtuellen Maschine können Lese- und Schreiboperationen von Kontextwechseln unterbrochen werden. In einer Mehrprozessorumgebung

[1] Institute of Electrical and Electronic Engineers (IEEE) Portable Operating System Interface (POSIX) 1003.1.

[2] Die orthogonale Persistenz des Systems erstreckt sich auf Daten, Code und Threads.

existiert keine fest definierte Reihenfolge, in der parallele Hauptspeicherzugriffe erfolgen, so daß für eine koordinierte Nutzung gemeinsamer Daten geeignete Synchronisiationsmechanismen zur Verfügung gestellt werden müssen.

5.1.1 Basisklassen

Nebenläufigkeit und parallele Ausführung mehrerer Threads sowie die notwendigen Synchronisationsmechanismen werden von vier Klassen bereitgestellt: `Thread`, `Mutex`, `Condition` und `BroadcastingCondition`. Ihre elementaren Methoden werden direkt auf Funktionen aus dem API des Pthread-Standards und damit auf die entsprechenden Betriebssystemobjekte abgebildet. Die virtuelle Maschine stellt keine zusätzlichen *Wrapper*-Funktionen bereit, die TL-2 Klassen verhalten sich somit genauso wie die Betriebssystemobjekte. Alle weitergehenden, höheren Sychronisationskonzepte, wie z.B. Semaphore oder Monitore, können mit Hilfe dieser Klassen in TL-2 aufgebaut werden.

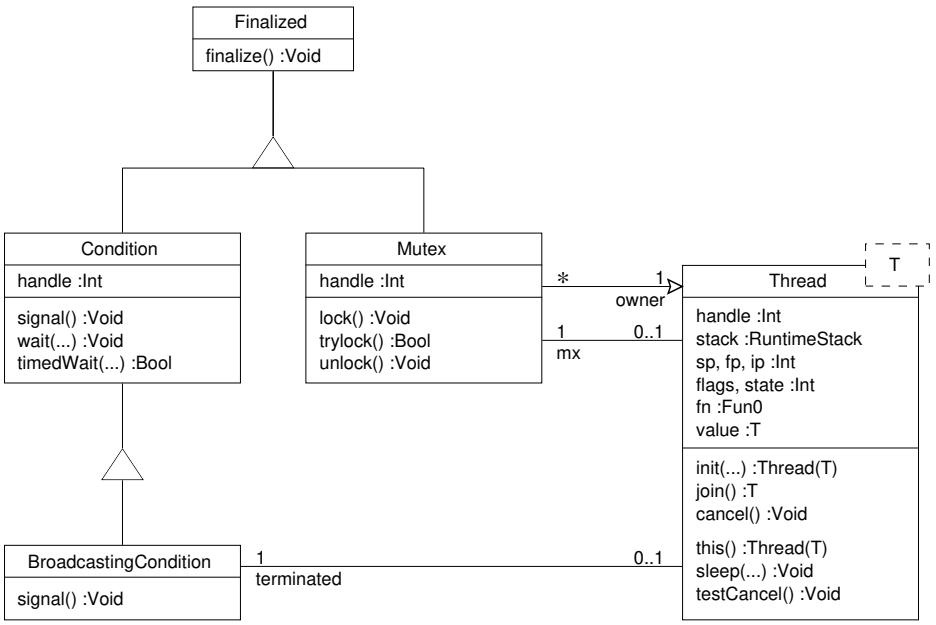

Abbildung 5.1: Basisklassen

Im Klassendiagramm in Abbildung 5.1 sind `Mutex` und `Condition` Subklassen der abstrakten Klasse `Finalized`. Bei der Allokation eines Exemplars der Subklassen etabliert diese automatisch eine schwache Referenz auf das neu angelegte Objekt. Die assoziierte, benutzerdefinierte TL-2 Funktion `finalize()` stellt sicher, daß bei Freigabe dieser Objekte durch die Speicherbereinigung (siehe Abschnitte 3.2.8 und 4.1.4) auch die Deallokation belegter Betriebsmittel erfolgt. Von einem Thread allozierte Betriebssystemressourcen werden unmittelbar vor seiner Beendigung von der virtuellen Maschine freigegeben (siehe Abschnitt 5.2.4).

In den weiteren Abschnitten werden die einzelnen Klassen näher erläutert.

Mutex

Ein Mutex (*mutual exclusion*) erlaubt mehreren Threads den synchronisierten Zugriff auf gemeinsam genutzte Ressourcen. Jeder Mutex kann zwei verschiedene Zustände einnehmen: von einem Thread erworben (*locked*) oder frei (*unlocked*). Ein Thread kann einen Mutex nur erwerben, wenn dieser frei ist, andernfalls blockiert er so lange, bis der Besitzer des Mutex diesen freigibt. Warten mehrere Threads auf einen Mutex, so wird genau einer von ihnen den Mutex nach dessen Freigabe erwerben und mit seiner Programmausführung fortfahren. Versucht ein Thread, einen Mutex ein zweites Mal zu erwerben, ohne ihn vorher freigegeben zu haben, oder versucht er einen Mutex freizugeben, den er nicht selbst erworben hat, führt dies in einen undefinierten Zustand.

Die Klasse `Mutex` definiert folgende Exemplarvariablen und Methoden:

`handle`: Der Identifikator des zugehörigen Betriebssystemobjekts.

`owner`: Der Thread, der den Mutex erworden hat oder `nil`, wenn der Mutex frei ist.

`lock()`: Ein Thread erwirbt mit dieser Methode den Mutex. Er blockiert dabei solange, wie der Mutex von einem anderen Thread gehalten wird.

`trylock()`: Ein Thread versucht mit dieser Methode, den Mutex zu erwerben. Wird dieser bereits von einem anderen Thread gehalten, kehrt die Methode mit dem Resultat `false` zurück, anderenfalls konnte der Mutex erfolgreich erworben werden.

`unlock()`: Der Mutex wird freigegeben.

Die Methoden `lock` und `unlock` kapseln einen kritischen Programmabschnitt, der nur von einem Thread betreten werden kann, und serialisieren somit die Ausführung von Threads:

```
mutex.lock
modify shared data  ;;  kritischer Abschnitt
mutex.unlock
```

Besondere Vorsicht ist erforderlich, wenn der von `lock` und `unlock` gekapselte Code Ausnahmen auslösen kann, die nicht von einer entsprechenden Ausnahmebehandlung abgefangen werden. Dies kann dazu führen, daß der Mutex nicht freigegeben wird und beim erneuten Versuch, den kritischen Abschnitt zu betreten, eine Verklemmung auftritt.

Ein Mutex kann auch als binärer Semaphor betrachtet werden, wenn davon abgesehen wird, daß Semaphore keinen Besitzer haben.

Condition und BroadcastingCondition

Ereignisvariablen (*condition variables*) sind ein Synchronisationsmechanismus für die Koordination von Aktionen zwischen Threads. Mit ihrer Hilfe können ein oder mehrere Threads auf den Eintritt eines bestimmten Ereignisses warten, das von einem anderen Thread signalisiert wird. Die Exemplarvariablen und Methoden einer Ereignisvariablen sind:

`handle`: Der Identifikator des zugehörigen Betriebssystemobjekts.

`signal()`: Sendet ein Signal an die Ereignisvariable. Bei einer `BroadcastingCondition` werden alle wartenden Threads geweckt, bei einer `Condition` nur einer.

`wait(mx :Mutex)`: Ein Thread wartet mit dieser Methode auf den Eintritt eines Ereignisses.

`timedWait(mx :Mutex, abstime :Long)`: Im Gegensatz zur Methode `wait` wartet ein Thread bei dieser Methode nur so lange auf den Eintritt eines Ereignisses, bis die Systemzeit den Wert von `abstime` erreicht. Bei dem Resultat `false` wurde die Systemzeit überschritten, im anderen Fall ein Ereignis signalisiert.

An der Programmierung mit Ereignisvariablen sind drei Komponenten beteiligt: die Ereignisvariable selbst, ein assoziierter Mutex und eine Bedingung. Anhand der Bedingung kann ein Thread überprüfen, ob ein Ereignis bereits eingetreten ist oder er warten muß. Der assoziierte Mutex schützt die Bedingung vor nebenläufigen Zugriffen:

```
mutex.lock
while(!predicate)          ;;  Bedingung überprüfen
  condition.wait(mutex)
do work
mutex.unlock
```

Die Auswertung der Bedingung in einer Schleife ist essentiell, da ein Thread dem Pthread-Standard zufolge auch unvorhergesehen aus dem `wait` oder `timedWait` zurückkehren kann, ohne daß ein Ereignis signalisiert wurde und die Bedingung erfüllt wird (*spurious wakeups*). Bei der Programmierung in TL-2 muß dies berücksichtigt werden. Eine besondere Bedeutung kommt diesem Verhalten beim Neustart des Systems zu (siehe Abschnitt 5.2.5).

Der assoziierte Mutex wird, sobald ein Thread in den Wartezustand geht, automatisch und atomar freigegeben. Wird der Ereignisvariablen ein Signal gesendet, so wird erst der Mutex wieder erworben, bevor der Thread aus dem Wartezustand zurückkehrt. Dies erlaubt es einem anderen Thread in der Zwischenzeit, den Mutex zu erwerben, um seinerseits ebenfalls auf das Ereignis zu warten oder die Bedingung zu erfüllen und den Eintritt des Ereignisses zu signalisieren:

```
mutex.lock
predicate := true  ;;  Bedingung setzen
condition.signal   ;;  Ereignis signalisieren
mutex.unlock
```

Der Entwurf modelliert Ereignisvariablen getrennt nach Ereignisvariablen, die nur einen Thread aufwecken (*signal*) oder alle (*broadcast*). Diese Trennung ist nach dem Pthread-Standard, der nur eine Art von Ereignisvariablen mit beiden Varianten definiert, unnötig. Sie erfolgte vielmehr in Hinblick auf eine geplante Portierung des System auf Windows NT, das nur den POSIX.1-Standard ohne Pthread unterstützt. Die den Ereignisvariablen äquivalenten, betriebssystemeigenen NT-Objekte unterscheiden zwischen einem *signal* und *broadcast*.

Thread

Ein Thread, auch *lightweight process* genannt, stellt einen unabhängigen Ausführungskontext innerhalb eines Prozesses dar. Jeder Thread besitzt einen eigenen Registersatz und einen Laufzeitstapel, alle weiteren Ressourcen werden mit dem Prozeß geteilt. Ihre Konstruktion, Destruktion und Kontextwechsel (*scheduling*) sind daher im Vergleich zu einem Prozeß mit geringeren Kosten verbunden.

Im Tycoon-2 System werden Threads auf Betriebssystemthreads abgebildet. Der unabhängige Kontext besteht aus einer eigenen Instanz des Bytecode-Interpreters, der eine Funktion höherer Ordnung ausführt.

Ein Thread verfügt über folgende Exemplarvariablen und Methoden:

`handle`: Der Indentifikator des Betriebssystemthreads.

`stack`: Der Laufzeitstapel der virtuellen Maschine.

`sp, fp, ip`: Die Maschinenregister.

`flags`: Der interne, von der virtuellen Maschine verwaltete Zustand des Threads.

`state`: Der TL-2 Zustand des Threads.

`fn`: Die ausgeführte Funktion höherer Ordnung.

`value`: Das Funktionsergebnis oder, wenn der Thread noch nicht beendet ist, `nil`.

`mx`: Der TL-2 Zustand wird durch einen Mutex geschützt.

`terminated`: Eine Ereignisvariable, die mit Beendigung des Thread signalisiert wird. Andere Threads können somit auf seine Terminierung warten, wie in Abschnitt 5.2.4 bei der Implementierung der `join`-Methode beschrieben.

`init(inner :Fun0)`: Initialisiert ein neu alloziertes Threadobjekt und startet die als Argument übergebene Funktion höherer Ordnung.

`join()`: Andere Threads können mit dieser Methode auf die Beendigung des Threads warten und sein Funktionsergebnis abfragen.

`cancel()`: Signalisiert dem Thread, daß er sich beenden soll.

`this()`: Liefert das TL-2 Objekt des aktuellen Threads.

`sleep(seconds :Long)`: Der aktuelle Thread wartet für die angegebene Dauer.

`testCancel()`: Der aktuelle Thread überprüft, ob er sich beenden soll.

Die virtuelle Maschine unterscheidet zwischen dem TL-2 öffentlich zugänglichen und dem internen Zustand eines Threads. Ein Thread kann nach außen die Zustände `RUNNING`, `CANCEL-REQUESTED` und `TERMINATED` einnehmen, d.h. der Thread läuft, er läuft, soll aber beendet werden, bzw. er wurde beendet. Der interne Zustand ergibt sich aus einer Kombination folgender Fälle:

`BLOCKED`: Der Thread ist blockiert, z.B. durch den Versuch, einen Mutex zu erwerben.

`CCALL`: Es wird ein externer Funktionsaufruf ausgeführt.

`WAITING`: Der Thread wartet auf ein Ereignis.

`INTERRUPT`: Der Thread ist in einer externen Funktion blockiert, während ein `Commit` durchgeführt wird (siehe Abschnitt 5.2.3).

Der Fall `INTERRUPT` ist nur für die Implementierung von Bedeutung, er wird hier jedoch bereits der Vollständigkeit halber angegeben. Die verschiedenen Zustände eines Threads während seines Lebenszyklus sind in Abbildung 5.2 dargestellt.

Ein Thread kann nicht explizit von einem anderen Thread beendet werden (*cancellation*). Die einzige Möglichkeit besteht darin, einen Thread über die Methode `cancel` von einem Terminierungswunsch zu benachrichtigen. Der angesprochene Thread kann dies durch die Methode `testCancel`, die bei einem anliegenden Wunsch eine Ausnahme auslöst, überprüfen. Der Sinn dieses Vorgehens liegt darin begründet, undefinierte oder unsichere Systemzustände, wie die dauerhafte Blockierung externer Ressourcen, die daraus resultieren, daß die virtuelle Maschine keine Informationen über höhere Synchronisationsmechanismen besitzt, zu vermeiden. Der TL-2 Programmierer ist auf dieser Ebene der einzige, der bestimmen kann, wann es sicher ist, einen Thread abzubrechen, bzw. der einzige, der vorher einen konsistenten Systemzustand herstellen kann.

Obwohl der Pthread-Standard verschiedene Strategien für Kontextwechsel definiert (*scheduling policies*), ist es aufgrund unterschiedlicher Implementierungen bzw. Restriktionen nicht möglich, die Reihenfolge, in der Threads ausgeführt werden, zu bestimmen [NBF96]. Dies und die Tatsache, daß die virtuelle Maschine keine Abhängigkeiten zwischen Threads erkennen kann, muß bei der Implementierung, z.B. bei der Initialisierung des Systems, berücksichtigt werden (siehe Abschnitt 5.2.5).

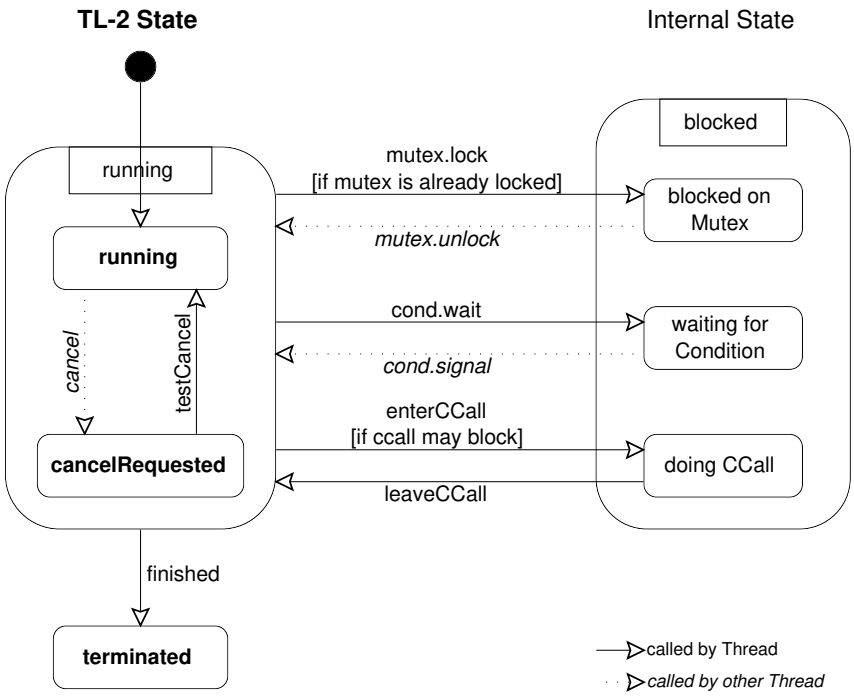

Abbildung 5.2: Zustandsdiagramm eines Thread

5.1.2 Externe Ressourcen

Die Semantik des `Commit` in Tycoon-2 ist es, den Systemzustand zu einem beliebigen Zeitpunkt einzufrieren, um ihn später wiederherzustellen. Die Programmausführung wird bei einem Neustart dabei an der Stelle fortgesetzt, an der sie von der `Commit`-Operation des Objektspeichers unterbrochen wurde. Externe Ressourcen werfen in diesem Zusammenhang eine besondere Problematik auf, da sie außerhalb des persistenten Systemzustands stehen und nicht der direkten Kontrolle des Tycoon-2 Systems unterliegen. Um die für das Tycoon-2 System definierte Persistenz auch auf externe Ressourcen anzuwenden, muß der sichtbare, konsistente Zustand einer Ressource zum Zeitpunkt des `Commit` mit gesichert werden. Bei einem Neustart wird dieser Zustand wiederhergestellt, bevor weitere Operationen auf der Ressource möglich sind.

Das im weiteren Verlauf vorgestellte transaktionsorientierte Konzept erweitert die Persistenzabstraktion auf externe Ressourcen. Es basiert auf den in Abschnitt 5.1.1 beschriebenen Basisklassen, wobei die für die Synchronisation und Sicherung der Konsistenz notwendigen Mechanismen für den Benutzer transparent sind. Das Diagramm in Abbildung 5.3 gibt einen Überblick über die beteiligten Klassen.

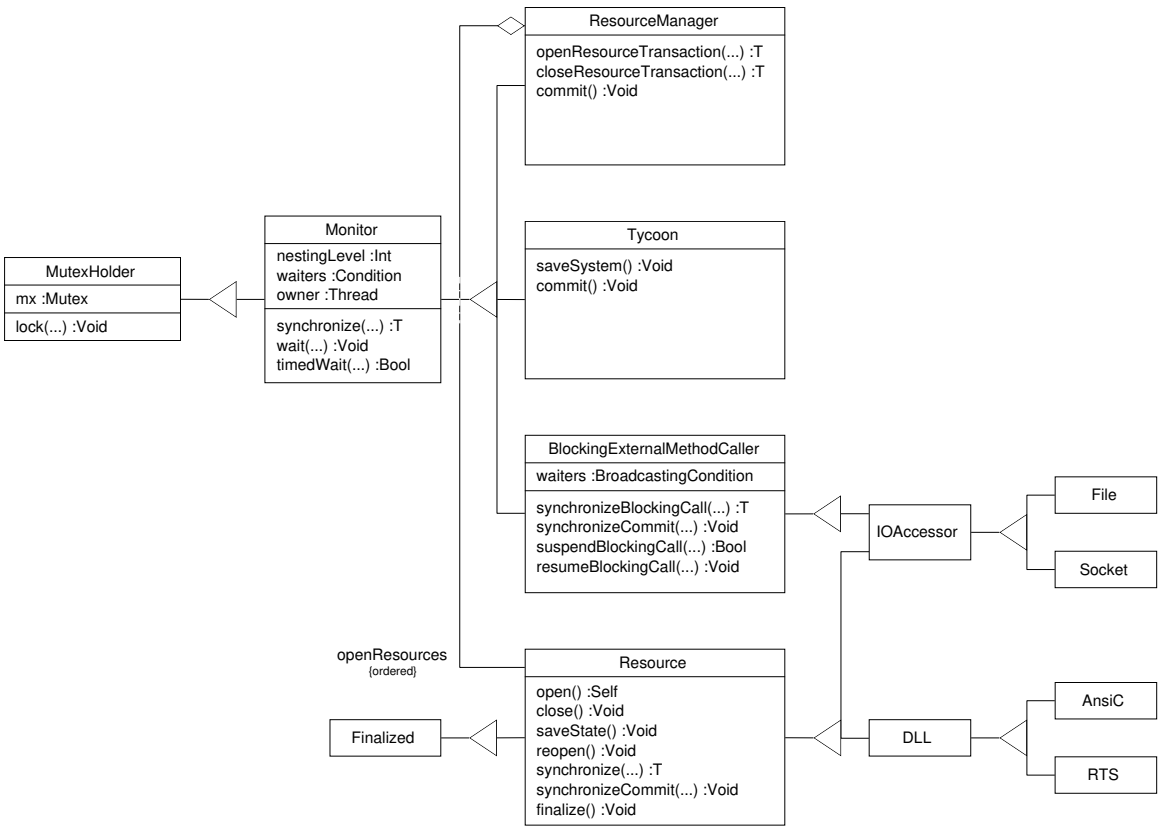

Abbildung 5.3: Klassendiagramm für externe Ressourcen

MutexHolder

Die Methode `lock` führt eine Folge geschützter Anweisungen aus. Im Gegensatz zur expliziten Programmierung mit einem Mutex garantiert sie, daß der kritische Abschnitt auch bei einer Ausnahme korrekt verlassen wird, d.h. der assoziierte Mutex wird vorher freigegeben. Ausnahmen werden an den Aufrufer weitergereicht.

Monitor

Ein Monitor synchronisiert analog zu einem Mutex den Zugriff auf gemeinsam von mehreren Threads genutzte Daten. Im Unterschied zu einem Mutex ist er jedoch reentrant, so daß derselbe Thread ihn mehrmals hintereinander erwerben kann. Die kritischen Abschnitte werden von der Methode `synchronize` geschützt. Sie garantiert wie die Methode `lock` der Klasse `MutexHolder` das Abfangen von Ausnahmen und das korrekte Verlassen des kritischen Abschnitts.

Jeder Monitor besitzt einen internen Zustand, der angibt, von welchem Thread der Monitor erworben wurde (`owner`) und wie oft dieser einen kritischen Abschnitt betreten hat (`nestingLevel`). Ist der Monitor frei, so haben diese Exemplarvariablen den Wert `nil` bzw. 0. Der interne Zustand des Monitors wird von einem Mutex geschützt, der nur beim Betreten und Verlassen des kritischen Abschnitts erworben wird, bei der Ausführung der geschützten Anweisungen jedoch frei ist.

Versucht ein Thread, einen bereits von einem anderen Thread erworbenen Monitor zu betreten, so blockiert er, indem er auf die Ereignisvariable `waiters` wartet, die bei Freigabe des Monitors signalisiert wird.

BlockingExternalMethodCaller

Die Klasse `BlockingExternalMethodCaller` erweitert einen Monitor um die Funktionalität, blockierende externe Methodenaufrufe zu synchronisieren (`synchronizeBlockingCall`) und den Zustand einer hierfür erworbenen Ressource auch während eines `Commit` korrekt zu sichern (`synchronizeCommit`). Die genauen Details werden in Abschnitt 5.2.3 dargelegt.

Tycoon

Die Klasse `Tycoon` stellt verschiedene, systemweit verfügbare Dienste bereit. Einer dieser Dienste ist die Sicherung eines persistenten Systemabbildes (`saveSystem`). Die elementare Methode `commit` stellt dabei die Schnittstelle zur virtuellen Maschine dar.

ResourceManager

Zentrale Instanz für die Verwaltung externer Ressourcen ist der Ressourcenmanager, von dem nur ein Exemplar im System existiert. Er führt Buch über alle geöffneten Ressourcen, hält auf diese jedoch nur schwache Referenzen, damit die Speicherbereinigung nicht mehr erreichbare, aber noch offene Ressourcen finden und finalisieren kann (siehe Abschnitt 3.2.8). Das Finalisieren schließt die Ressource und entfernt sie aus dem Ressourcenmanagement. Der Ressourcenmanager verfügt über drei Methoden:

`openResourceTransaction()`: Registriert eine Ressource beim Manager und öffnet diese.

`closeResourceTransaction()`: Schließt eine Ressource und entfernt sie aus dem Ressourcenmanager.

`commit()`: Bereitet alle registrierten Ressourcen auf ein `Commit` des Objektspeichers vor.

Alle Methoden sind synchronisiert.

Resource

`Resource` ist die abstrakte Superklasse aller konkreten externen Ressourcen. Sie definiert die grundlegenden Methoden zum Öffnen und Schließen (`open`, `close`) und zur Zustandssicherung bzw. -wiederherstellung bei einem `Commit` (`saveState`, `reopen`, `synchronizeCommit`). Ihre Synchronisationsmethode `synchronize` besitzt keine eigene Funktionalität. Diese wird erst durch Mehrfachvererbung von einer geeigneten Klasse, die `synchronize` ersetzt, z.B. `Monitor`, realisiert und ermöglicht den exklusiven Erwerb einer Ressource für eine Transaktion.

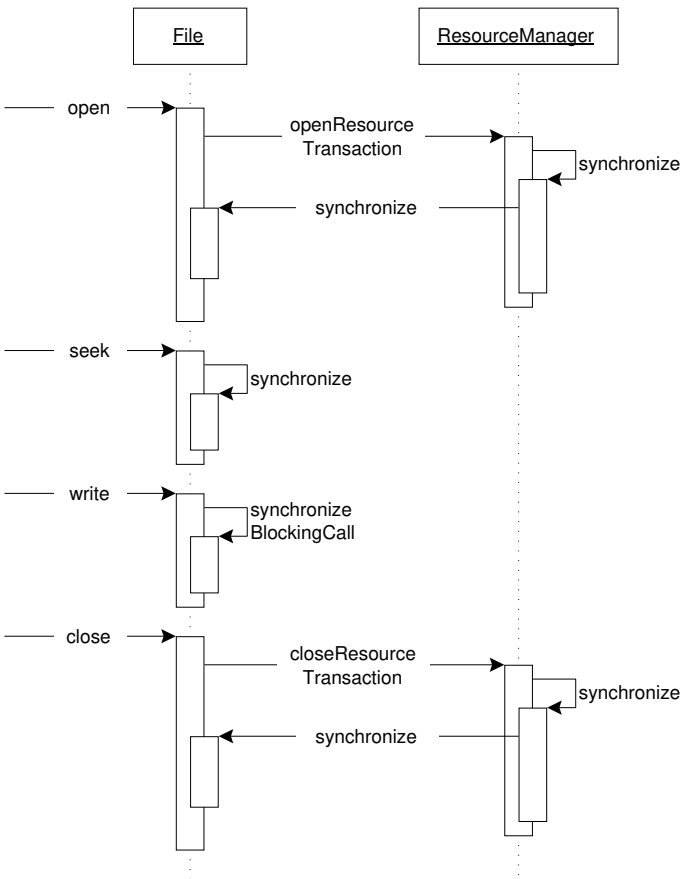

Abbildung 5.4: Sequenzdiagramm einer Dateioperation

Der Lebenszyklus einer konkreten Ressource ist in Abbildung 5.4 anhand einer Datei beispielhaft illustriert. Zu erkennen ist:

- Ressourcen werden im Zuge des Öffnens und Schließens automatisch beim Ressourcenmanager registriert bzw. wieder entfernt. Das Protokoll, welches sie einhalten müssen,

garantiert, daß sich der Ressourcenmanager vor der Ressource und der eigentlichen Operation synchronisiert. Diese Eigenschaft ist wichtig für den korrekten Ablauf eines `Commit`.

- Alle Operationen, die den Zustand einer Ressource ändern und mit Seiteneffekten verbunden sind, werden, um die Konsistenz des Ressource zu gewährleisten, atomar in einer Transaktion ausgeführt. Die Anweisungen der entsprechenden Methoden werden hierzu von einem `synchronize` oder, wenn der Thread dabei blockieren kann, `synchronizeBlocking` gekapselt. Eine Ausnahme hiervon stellen nur die Methoden `saveState` und `reopen` dar, die nur während des `Commit` aufgerufen werden, zu einem Zeitpunkt, da der verantwortliche Thread bereits die direkte Kontrolle über die Ressource erlangt hat.

Commit

Der Ablauf eines `Commit` auf TL-2 Ebene gliedert sich in drei Phasen:

1. Der Thread, der das `Commit` durchführt, erwirbt alle offenen Ressourcen und sichert deren Zustand.
2. Die virtuelle Maschine erzeugt ein persistentes Objektspeicherabbild.
3. Die Ressourcen werden wieder freigegeben. Bei einem Neustart des Systems wird ihr gesicherter Zustand vorher restauriert.

Die Sequenzdiagramme in den Abbildungen 5.5 und 5.6 illustrieren den Vorgang exemplarisch anhand zweier beteiligter Ressourcen.

Die Sicherung des Systemzustands wird durch eine `saveSystem` Nachricht an das `Tycoon` Objekt ausgelöst. Dieses delegiert das Sichern der Zustände aller offenen Ressourcen an den Ressourcenmanager.

Die Ressourcen werden in der umgekehrten Reihenfolge ihrer Registrierung auf das anstehende `Commit` des Objektspeichers vorbereitet. Der Ressourcenmanager sendet hierzu der zuletzt geöffneten Ressource die Nachricht `synchronizeCommit`. Die Ressource synchronisiert sich daraufhin selbst durch Aufruf der `synchronize` Methode.[3] Sobald die Kontrolle über die Ressource erlangt wurde, wird deren Zustand gesichert (`saveState`). Im Anschluß veranlaßt die Ressource die Synchronisation der vor ihr geöffneten Ressource. Auf diese Weise werden alle offenen Ressourcen während des gesamten `Commit` für andere Threads gesperrt. Ist dieser Vorgang beendet und alle Ressourcen synchronisiert, wird das `Commit` auf Ebene der virtuellen Maschine ausgeführt.

Die elementare Methode `commit` der Klasse `Tycoon` sichert das persistente Objektspeicherabbild (siehe auch Abschnitt 5.2.1). Anhand des von ihr gelieferten Rückgabewertes kann

[3] Die besondere Behandlung blockierender externer Methoden erfolgt in Abschnitt 5.2.3.

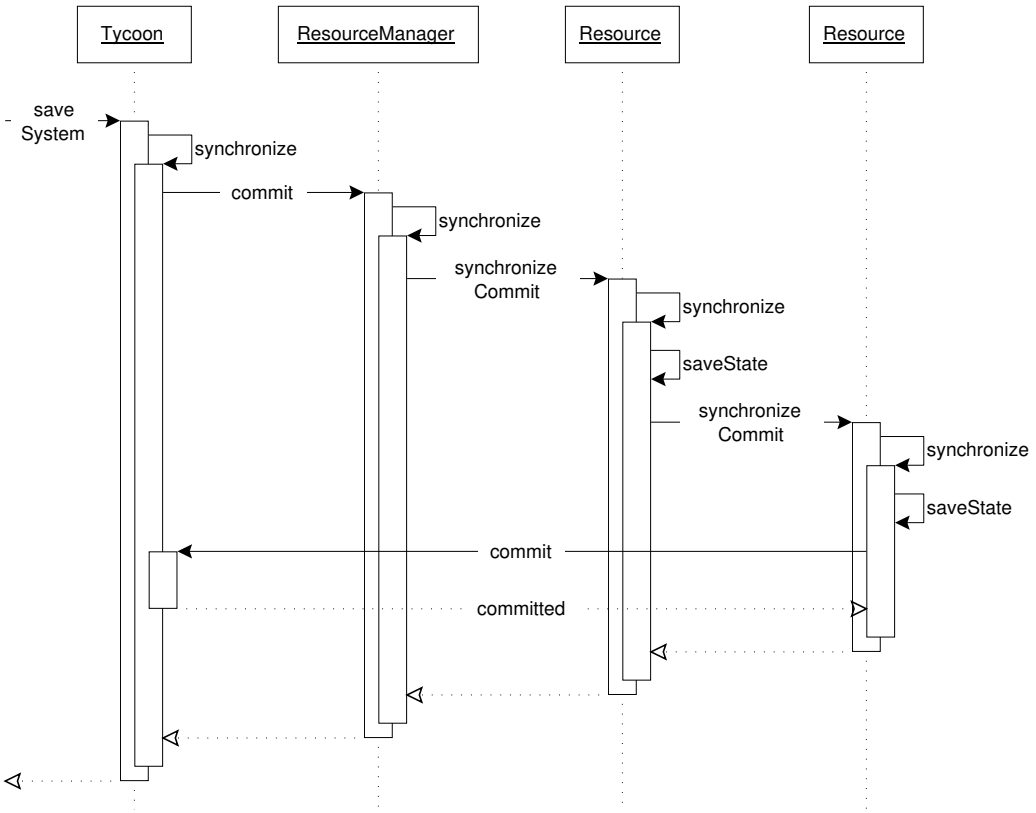

Abbildung 5.5: Ablauf eines `Commit`

erkannt werden, ob die Programmausführung fortgesetzt wird oder das System neu gestartet wird:

- Bei Fortsetzung des Programmausführung werden alle Ressourcen freigegeben, womit andere Threads sie wieder erwerben können.
- Bei einem Neustart versucht jede Ressource in der Reihenfolge, in der sie geöffnet wurden, ihren gesicherten Zustand zu restaurieren (`reopen`). Anschließend wird die Ressource freigegeben. Ein Umstand, mit dem bei der Migration des Objektspeicherabbildes auf einen anderen Rechner oder aufgrund der unbestimmten Zeitspanne zwischen `Commit` und Neustart gerechnet werden muß, sind Fehler, die während des Öffnens einer Ressource abgefangen werden. Die entsprechende Ressource steht dem System dann nicht mehr zur Verfügung und ihre Benutzung löst Ausnahmen aus.

Der Thread, der das `Commit` durchführt, hält während des gesamten Vorgangs die Kontrolle über alle offenen Ressourcen. So ist sichergestellt, daß kein anderer Thread, wenn

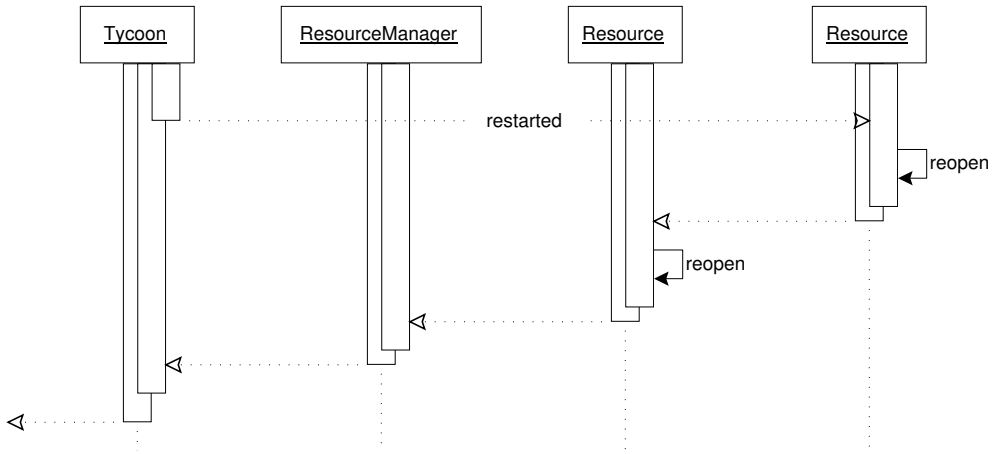

Abbildung 5.6: Ablauf eines `Restart`

der Systemzustand gesichert wird, eine dieser Ressourcen erwerben und in einen möglicherweise inkonsistenten Zustand bringen kann. Außerdem können, da die entsprechenden Methoden des Ressourcenmanagers synchronisiert sind, keine weiteren Ressourcen geöffnet oder geschlossen werden.[4] In diesem Zusammenhang ist wichtig, daß das Protokoll für die Interaktion von Ressourcen und Ressourcenmanager eingehalten wird. Der Versuch, eine Ressource während eines `Commit` zu schließen, könnte sonst, wenn sich die Ressource vor dem Ressourcenmanager synchronisiert, eine Verklemmung verursachen. Dieser Fall ist in Abbildung 5.7 darstellt.

Folgende Annahmen müssen für eine korrekte Funktionsweise eingehalten werden:

- Das *Scheduling* der Synchronisationsmechanismen ist fair, d.h. der Thread, der den Systemzustand sichert, muß eine Ressource nach endlicher Zeit erwerben.
- Es existieren keine wechselseitigen Abhängigkeiten zwischen Ressourcen. Es ist für den Ressourcenmanager nicht zu bestimmen, welche Abhängigkeiten existieren und, falls dies überhaupt möglich ist, in welcher Reihenfolge die Ressourcen geöffnet werden müssen. Die Auswirkungen auf den Ressourcezustand sind in einem solchen Fall unbestimmt. Bei synchronisierten Ressourcen kann es zudem durch eine unterschiedliche Reihenfolge bei ihrem Erwerb zu Verklemmungen kommen, analog zu dem in Abbildung 5.7 dargestellten Beispiel.

Lassen sich die Abhängigkeiten serialisieren, so muß diese Reihenfolge auch beim `Commit` eingehalten werden, d.h. die Ressourcen müssen in der entsprechenden Reihenfolge beim

[4] Über diesen Mechanismus wird auch der evtl. aktive *Finalizer* (siehe Abschnitt 4.1.4) bis zur Beendigung des `Commit` blockiert.

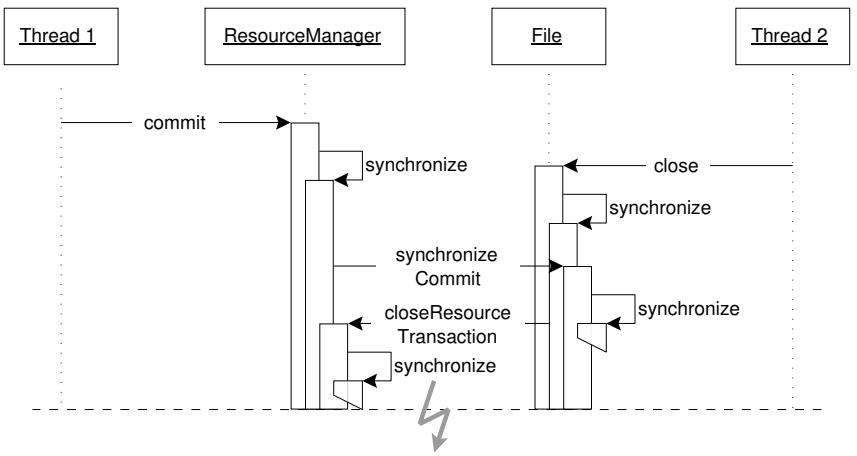

Abbildung 5.7: Verklemmung bei Verletzung des Protokolls zwischen Ressource und Ressourcenmanager

Ressourcenmanager registriert werden. Ein Beispiel hierfür sind die dynamischen Bibliotheken (`DLL`): Der Entwurf sieht vor, dynamische Bibliotheken nicht zu synchronisieren. Dies hat den Vorteil, daß die von ihnen zur Verfügung gestellten Methoden nebenläufig und parallel von mehreren Threads eingesetzt werden können. Es bedingt jedoch zwei Voraussetzungen, um zu verhindern, daß bei einem Neustart externe Methodenaufrufe getätigt werden, bevor die entsprechende Bibliothek geöffnet wurde, was in einem Laufzeitfehler resultieren würde. Zum einen müssen alle Klassen, die externe Bibliotheken benutzen, synchronisiert und von der Ressourcenverwaltung erfaßt werden, zum anderen muß eine Bibliothek immer vor den auf ihr aufbauenden Ressourcen geöffnet werden.

Bei Verwendung externer Datenbestände, angefangen von einzelnen Dateien bis hin zu kompletten Datenbanken, beschränkt sich der Zustand des Tycoon-2 Systems nicht mehr allein auf den Objektspeicher und kann somit bei einem `Commit` nicht vollständig eingefroren werden. Bei einem Neustart des Systems können sich die externe Ressourcen also im Vergleich zum Zeitpunkt des `Commit` geändert haben und der gesicherte Zustand kann nicht mehr restauriert werden. Die Lösung dieses Problems ist jedoch nicht Aufgabe der virtuellen Maschine, sondern kann nur auf Ebene der Applikationsprogrammierung erfolgen.

5.2 Implementierung

Die Implementierung nebenläufiger, persistenter Threads umfaßt zwei Kerngebiete: Die Synchronisationsmechanismen innerhalb der virtuellen Maschine (Abschnitte 5.2.1 und 5.2.2) sowie das Erzeugen und Beenden von Threads (Abschnitt 5.2.4) und den Neustart des Systems (Abschnitt 5.2.5). Die für die Unterstützung blockierender externer Methodenaufrufe

bereitgestellte Funktionalität wird in Abschnitt 5.2.3 vorgestellt.

5.2.1 Synchronisation der virtuellen Maschine

Die Nebenläufigkeit der vom System bereitgestellten Threads muß nicht nur bei der Programmierung in TL-2 berücksichtigt werden, auch die virtuelle Maschine muß bei mehreren, parallel arbeitenden Instanzen des Bytecodeinterpreters synchronisiert werden. Schwerpunkte bilden hierbei der Schutz globaler Datenstrukturen und nicht reentranter Systemteile sowie die Zusicherung definierter Synchronisationspunkte, zu denen das Gesamtsystem einen konsistenten Zustand einnimmt.

Globale Datenstrukturen

Die Synchronisation globaler Daten erfordert nur einen geringen Aufwand, da die nötige Funktionalität bereits durch die Pthreads in Form von Mutexen bereitgestellt wird. Bevor diese jedoch implementiert wird, muß geklärt werden, welche der betroffenen Daten eines Zugriffsschutzes bedürfen:

- Globale Daten, die einmal statisch initialisiert und anschließend von den verschiedenen Threads nur noch gelesen werden, müssen nicht synchronisiert werden, wenn garantiert werden kann, daß ihre Initialisierung vor dem ersten Zugriff erfolgt. Hierunter fallen z.B. beim Programmstart in der Kommandozeile übergebene Argumente.
- Datenstrukturen, auf die mehrere Threads lesend und schreibend zugreifen, müssen immer geschützt werden. Ein Beispiel hierfür ist der Methodencache, der in Abschnitt 5.2.2 ausführlich behandelt wird.

Hinzu kommen Daten, von denen jeder Thread zwingend eine eigene Kopie benötigt. Für die Verwaltung dieser für jeden Thread lokalen Daten (*thread local data*) stellt das Pthread-API eigene Funktionen bereit: `pthread_key_create` generiert einen neuen Identifikator, dem jeder Thread mittels `pthread_setspecific` und `pthread_getspecific` einen eigenen Wert zuweisen bzw. auslesen kann. Auf diese Weise erhält jeder Betriebssystemthread einen Verweis auf sein Threadobjekt im Objektspeicher (siehe Abschnitt 5.1.1) und einen eigenen Kontextpuffer für die Behandlung von Ausnahmen (siehe Abschnitt 4.2.5).

Synchronisationspunkte

Der Objektspeicher nimmt bei der Synchronisation der virtuellen Maschine eine Sonderstellung ein. Als einziges Subsystem weist er nicht reentrante Komponenten, wie z.B. den Objektallokator, auf, so daß ein Teil seiner Funktionen nicht nebenläufig ausgeführt werden kann. Als weitere Schwierigkeit erweist sich die Tatsache, daß zwei seiner Operationen noch weitergehende Anforderungen stellen:

- Während der Durchführung der Speicherbereinigung muß der Zugriff auf den Objektspeicher gesperrt werden. Dies schließt alle Operationen auf Objekten ein, damit nicht mit evtl. ungültigen Daten gearbeitet wird. Da Objekte direkt und nicht über die Aufrufschnittstelle des Objektspeichers gelesen und geschrieben werden, ist es notwendig, alle Threads für die Dauer der Speicherbereinigung anzuhalten.

- Das erfolgreiche Sichern und Restaurieren eines Objektspeicherabbildes setzt voraus, daß alle Threads zum Zeitpunkt des `Commit` auf Ebene des Objektspeichers einen konsistenten Zustand einnehmen.

Zur Lösung dieser Problematik verfügt die virtuelle Maschine über einen Mechanismus, der es einem Thread ermöglicht, sich mit allen anderen im System aktiven Threads zu synchronisieren und einen kritischen Vorgang wie die Speicherbereinigung auszuführen, während alle anderen Threads in einem fest definierten, konsistenten Zustand warten (Synchronisationspunkt). Der Zustand eines Threads gilt hierbei als konsistent, wenn alle vom Bytecodeinterpreter lokal gehaltenen Maschinenregister mit den Einträgen im entsprechenden Threadobjekt abgeglichen wurden. Insbesondere muß berücksichtigt werden, daß Threads blockieren können und damit nicht mehr in der Lage sind, aktiv auf einen Synchronisationswunsch zu reagieren.

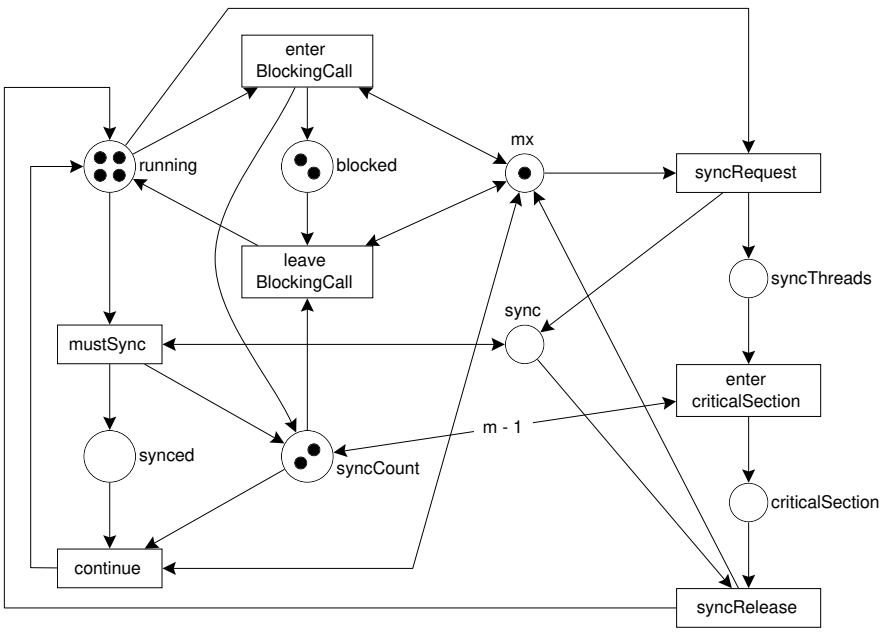

Abbildung 5.8: Synchronisation auf Maschinenebene mit $m = 6$ Threads

Das Stellen/Transitions-Netz [JV87] in Abbildung 5.8 illustriert das eingesetzte Verfahren. Marken in den Stellen `mx`, `sync` und `syncCount` erfüllen Synchronisationsaufgaben, alle anderen Marken repräsentieren Threads. Der Ausgangszustand zeigt ein System mit 6 Threads ($m = 6$), von denen 4 laufen (`running`) und 2 blockiert[5] (`blocked`) sind. Ein globaler Mutex (`mx`) steuert die Synchronisierung.

Threads in den Stellen `synced` und `blocked` befinden sich in einem synchronisierten, konsistenten Zustand. Nimmt ein Thread einen dieser beiden Zustände ein, wird gleichzeitig eine Marke in die Stelle `syncCount`, die die Anzahl der synchronisierten Threads zählt, hinzugefügt, beim Verlassen wird sie wieder abgezogen.

Der Übergang vom aktiven in den blockierten Zustand und zurück durch die Transitionen `enterBlocking` und `leaveBlocking` wird vom globalen Mutex als Nebenbedingung geschützt. Kein Thread kann somit, falls ein Synchronisationswunsch anliegt, blockieren oder, wenn er während der Ausführung der kritischen Operation aus dem blockierenden Aufruf zurückkehren sollte, einfach weiterlaufen. Ein blockierter Thread ist immer synchronisiert.

Ein Thread, der einen kritischen Vorgang ausführen will, erwirbt den Mutex dauerhaft, d.h. zieht die Marke aus der Stelle `mx` ab, und signalisiert allen anderen Threads seinen Synchronisationswunsch durch eine Marke in der Stelle `sync`, womit gleichzeitig die Nebenbedingung der Transition `mustSync` aktiviert wird. Laufende Threads können nur noch die Transition `mustSync` schalten, womit sie sich selbst synchronisieren, danach halten sie genauso wie alle blockierten Threads, da keine der Ausgangstransitionen aufgrund der nicht erfüllten Nebenbedingung aktiviert ist. Sobald die Synchronisierung abgeschlossen ist befinden sich 5 Marken in der Stelle `syncCount` und der in `syncThreads` wartende Thread kann in den kritischen Abschnitt eintreten (`enterCriticalSection`). Bei seinem Verlassen wird der Synchronisationswunsch zurückgezogen und der Mutex wieder freigegeben. Alle angehaltenen oder blockierten Threads können nun weiterlaufen.

Im dargestellten Netz ist es denkbar, daß eine weitere Synchronisationsaufforderung eintrifft, bevor die Transition `continue` für alle Marken in `synced` geschaltet hat. In der Praxis ist es durch diesen Mechanismus jedoch nicht möglich, Threads dauerhaft festzuhalten, da die so geschützten Operationen nur in sehr großen Zeitabständen auftreten.

Im abgebildeten Netz lassen sich folgende Invarianten erkennen:

$$m(mx) + m(sync) = 1 \tag{5.1}$$

$$m(running) + m(blocked) + m(synced) + m(syncThreads) + m(criticalSection) = m \tag{5.2}$$

$$m(blocked) + m(synced) = m(syncCount) \tag{5.3}$$

Hieraus läßt sich, ohne formalen Beweis, folgern:

[5] Zu den blockierten Threads zählen alle unter Abschnitt 5.1.1 angegebenen Varianten.

- Zwischen dem zweimaligen Schalten der Transition `syncRequest` muß `syncRelease` genau einmal geschaltet haben (5.1). Es kann also nur ein Thread die Synchronisation des Systems veranlassen und den kritischer Abschnitt betreten.

$$m(syncThreads) + m(criticalSection) \leq 1 \tag{5.4}$$

- Damit die Transition `enterCriticalSection` schalten kann müssen eine Marke in `syncThreads` und $m - 1$ Marken in `syncCount` liegen. Aus (5.4), (5.3) und (5.2) folgt damit, daß keine Marke in `running` liegt. Beim Eintritt eines Threads in einen kritischen Abschnitt ist somit garantiert, daß alle anderen Threads synchronisiert sind.

Bei einem Synchronisationswunsch ist es einem laufenden Thread im vorliegenden ST-Netz nur möglich, sich selbst zu synchronisieren. In der konkreten Implementierung wird dieses Verhalten nachgebildet, indem jeder Thread an bestimmten festen Punkten seiner Programmausführung überprüft, ob er sich synchronisieren muß. Nur so kann verhindert werden, daß ein laufender Thread nicht auf eine Synchronisationsaufforderung reagiert. Die folgende Zusammenstellung gibt einen Überblick aller definierten Synchronisationspunkte, an denen ein Thread angehalten werden kann und einen konsistenten Zustand herstellen muß:

- Eintritt in eine eingebaute oder externe Methode, die blockieren kann.
- Erzeugung oder Beendigung eines Threads, da sich hierdurch die Gesamtzahl der aktiven Threads verändern.
- Aufruf einer Funktion, die selbst eine Synchronisation auslösen kann, wie bei den nicht reentranten Allokationsroutinen des Objektspeichers. Threads könnten sonst, beim Versuch, eine dieser Funktionen auszuführen, blockieren, falls bereits ein anderer Thread einen Synchronisationswunsch signalisiert hat.
- Aufbau eines Sicherungsrahmens auf dem Laufzeitstapel. Hierdurch wird sichergestellt, daß die Überprüfung regelmäßig erfolgt, falls ein Thread über einen längeren Zeitraum hinweg keine der vorher genannten Methoden oder Funktionen aufruft.
- Ausführung des `sync` Bytecodes. Der Bytecode wird vom TL-2 Compiler am Beginn eines Schleifenkörpers generiert, um die Synchronisation bei lang andauernden Iterationen ohne zusätzliche Synchronisationspunkte zu gewährleisten.

5.2.2 Zweistufiger Methodencache

Wie alle globalen Datenstrukturen ist auch der in Abschnitt 4.2.3 beschriebene Methodencache von der Erweiterung der virtuellen Maschine um Multithreading-Fähigkeiten betroffen. Als Problem erweisen sich hierbei die einzelnen Cacheeinträge, die nicht atomar gelesen

und geschrieben werden können. Bei mehreren Threads können somit Cachezugriffe unterbrochen oder parallel ausgeführt werden, einzelne Einträge werden inkonsistent mit nicht vorhersehbaren Folgen für den weiteren Programmablauf.

Zwei Lösungsmöglichkeiten, die in Betracht kommen, erfordern nur geringe Modifikationen:

1. Der globale Methodencache bleibt bestehen, wird jedoch durch einen Mutex geschützt. Dieser Ansatz verursacht durch den Synchronisatationsaufwand der Pthread-Primitive selbst bei nur einem aktiven Thread einen Performanzverlust von bis zu 30%.
2. Jeder Thread bekommt einen eigenen Methodencache zugewiesen. Die Zugriffe müssen nicht synchronisiert werden, da jeder Thread auf lokalen Daten arbeitet. Der für jeden Thread zusätzlich benötigte Speicher beträgt, um die Effizienz der Caches beizubehalten, 160KB. Zudem kann das Ergebnis einer bereits einmal erfolgreich durchgeführten Methodensuche nicht mehr von einem anderen Thread genutzt werden.

Beide Verfahren weisen Nachteile auf, die nicht akzeptabel sind. Auch die Möglichkeit, den globalen Cache beizubehalten und den Synchronisationsaufwand durch eine spezielle Reihenfolge beim Lesen und Schreiben der Datenworte eines Cacheeintrags zu minimieren, wie es in [NM97] aufgezeigt wird, scheidet aus. Der Grund hierfür ist in der Architektur moderner Multiprozessorsysteme zu suchen, die, um einen höheren Speicherdurchsatz zu erzielen, keine sequentielle Konsistenz ihrer Speicherzugriffe garantieren (*weak memory ordering*). So kann in einem solchen System ein Prozessor Werte in die Speicherzelle A und anschließend in B schreiben, ein anderer Prozessor, der zuerst aus B und dann aus A liest, sieht aber eventuell den neuen Wert nur in Speicherzelle B und den alten in A. Der Pthread-Standard schreibt daher für eine Reihe seiner Funktionen die Synchronisation des Speichers vor, damit alle Threads auf die gleichen Daten zugreifen (*memory barriers*).

Als Alternative kommt eine Kombination aus beiden Lösungsmöglichkeiten zum Einsatz. Der Methodencache ist in zwei Stufen organisiert:

- Einträge aus dem ersten Cache werden nur gelesen (*read only*), er erfordert daher keine Synchronisation. Schlägt eine Anfrage auf diesen Cache fehl, so wird sie an die zweite Stufe weitergereicht.
- Der zweite Cache ist durch einen Mutex vor parallelen Zugriffen geschützt. Er wird bei einer erfolglosen Anfrage mit dem Ergebnis der Methodensuche aktualisiert.

Die Anzahl der Fehlgriffe im ersten Cache wird protokolliert. Übersteigt sie eine vorgegebene Schwelle, 10000 im aktuellen System, werden alle aktiven Threads mit den im vorigen Abschnitt beschriebenen Synchronisationsmechanismen angehalten und die erste Stufe mit den Daten des zweiten Caches aufgefüllt. Durch dieses Vorgehen kann eine hohe Trefferquote der ersten Cache-Stufe sichergestellt werden, so daß keine Verschlechterung der Performanz beobachtet werden konnte.

Als Erweiterung dieser Lösung ist die Umstellung der ersten Cache-Stufe auf effiziente statische Selektortabellen, wie sie in [Dri93] vorgestellt werden, denkbar.

5.2.3 Blockierende externe Methoden

Wie in Abschnitt 5.1.2 erläutert, werden beim `Commit` alle offenen Ressourcen in der ersten Phase vom `Commit`-Thread erworben und in einen konsistenten Zustand gebracht. Dieses Vorgehen ist nur erfolgreich, wenn keiner der anderen Threads zu diesem Zeitpunkt einen externen Methodenaufruf auf einer Ressource tätigt, der blockiert. Der `Commit`-Thread muß sonst, bevor er die Ressource erwerben kann, eine unbestimmte Zeit auf die Beendigung dieses Aufrufs warten. Abhängig von der aufgerufenen Funktionen könnte das `Commit` somit dauerhaft blockiert werden, wenn von einem Terminal Eingaben gelesen werden beispielsweise so lange, bis ein Benutzer Daten eingibt.

Auf Betriebssystemebene existieren zwar für eine Vielzahl von Funktionen nicht blockierende Varianten, z.B. asynchrone Ein/Ausgabeoperationen, diese können jedoch nicht effektiv genutzt werden, da die mit ihnen verbundene Signalbehandlung in einer Umgebung mit mehreren Threads (siehe auch [NBF96]) und der darüber hinaus nicht zu vernachlässigenden Trennung von TL-2 Code und virtueller Maschine sehr problembehaftet ist. Aktives Polling scheidet aufgrund seiner negativen Auswirkungen auf die Performanz ebenfalls aus.

Da einem Thread die Kontrolle über eine Ressource nicht entzogen werden kann, muß die laufende Transaktion auf der Ressource unterbrochen oder blockiert werden, so daß der `Commit`-Thread ihren Zustand sichern kann. Dies ist insofern problematisch, als die Transaktionen in der Regel Seiteneffekte auf die Ressource selbst und den Objektspeicher haben. Die hier demonstrierte Lösung setzt daher voraus, daß eine unterbrochene Transaktion erneut gestartet werden kann, ohne daß sich hierdurch das Ergebnis verändert.

Ressourcen, die einen Thread blockieren können, erben von der Klasse `ExternalBlockingMethodCaller`, die spezielle Synchronisationsmethoden für den externen Methodenaufruf und das `Commit` bereitstellt (Abschnitt 5.1.2). Der nachstehende TL-2 Programmcode ist repräsentativ für einen externen blockierenden Methodenaufruf:

```
read(buffer :ByteArray, n :Int) :Int
{
  synchronizeBlockingCall({
    let rts = Tycoon.rts,
    let handle = _handle,
    let rmax = Int.max(n, buffer.size),
    let r = rts.tystdio_read(handle, buffer, rmax),
    (errno != 0) ?
      {raise IOError.new(errno)},
    r
  })
}
```

Drei Teilbereiche können unterschieden werden:

1. Der Prolog liest Betriebssystemhandles und Bibliotheksobjekte und initialisiert weitere Funktionsargumente.

2. Die externe Methode(`tystdio_read`) ruft die C-Funktion auf. An dieser Position kann eine Blockierung auftreten.

3. Im Epilog werden Fehlersituationen abgefangen und evtl. neue Betriebssystemhandles übernommen.

Die Möglichkeit, mehrere externe Aufrufe innerhalb einer Transaktion auszuführen, ändert nichts an der prinzipiellen Funktionsweise und wird im weiteren Verlauf nicht weiter beachtet.

Der Synchronisationsmechanismus für blockierende externe Methoden wird im folgenden beschrieben. Grundlage ist das nach [Jan95] modellierte gefärbte Petrinetz in Abbildung 5.9. Die definierten Farben und Variablen sind:

Farben:

$$
\begin{aligned}
T &= \{p, q\} \\
O &= T \cup \{nil\} \\
MS &= \{(s, r) | s \in O, r \in Bool\} \\
S &= \{ccall, interrupt\} \\
TS &= 2^S
\end{aligned}
$$

Variablen:

$$
\begin{aligned}
x &: T \\
b &: Bool \\
u &: TS
\end{aligned}
$$

Zwei Threads (`p` und `q`) sind nebenläufig aktiv, `p` führt Transaktionen auf einer geöffneten Ressource aus, während `q` die Ressource für ein `Commit` synchronisieren muß. Die Stelle `MonitorState` repräsentiert den Zustand des die Ressource schützenden Monitors, d.h. seinen Besitzer und ein Prioritätssignal, `ThreadState` den internen Zustand von Thread `p`.

Eine von der Methode `synchronizeBlockingCall` geschützte Transaktion erwirbt zuerst den Monitor (`acquireMonitor`) und damit die Ressource, wobei sie ein vorhandenes Prioritätssignal beachtet, und führt ihren Prolog aus, bevor sie die eventuell blockierende externe Methode betritt. Nach der Rückkehr aus der externen Methode und Ausführung des Epilogs wird der Monitor wieder freigegeben (`freeMonitor`). Der eigentliche Funktionsaufruf (`ccall`) wird in der C-Schnittstelle der virtuellen Maschine von einem Ein- und Austrittsprotokoll gekapselt[6] (`enterCCall` und `leaveCCall`), das, solange der Threadzustand eine Unterbrechung signalisiert (`interrupt`), beim Versuch die Funktion aufzurufen eine Ausnahme auslöst bzw. beim Verlassen der Funktion blockiert. Wird eine Ausnahme ausgelöst, so wird die Ressource automatisch freigegeben und die Transaktion neu gestartet.

[6] Nur blockierende C-Funktionen werden von diesem Protokoll gekapselt; sie müssen hierzu der virtuellen Maschine bekannt gemacht werden. Da die Syntax von TL-2 bisher keine diesbezügliche Kennzeichnung externer Methoden vorsieht, sind alle betroffenen Funktionsnamen statisch in der Maschine festgelegt.

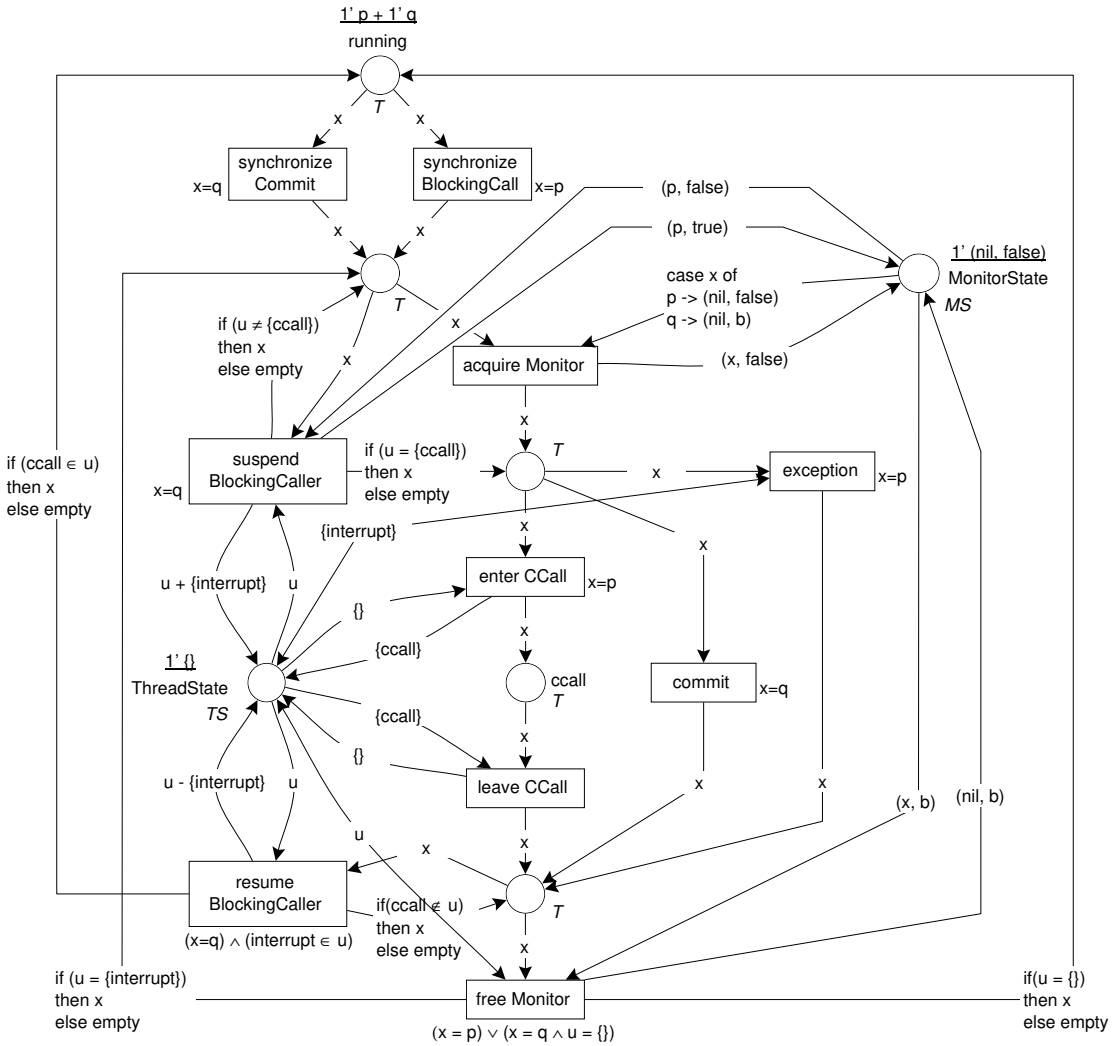

Abbildung 5.9: Synchronisation blockierender externer Methoden beim `Commit`

Wenn Thread `q` die Ressource synchronisiert (`synchronizeCommit`) können mehrere Situationen auftreten; dabei ist es jedoch ausgeschlossen, daß die Ressource von ihm bereits erworben wurde:

- Die Ressource ist frei: Die Ressource wird regulär erworben und das `Commit` fortgesetzt. Ist das `Commit` abgeschlossen wird die Ressource wieder freigegeben.
- Die Ressource wurde von einem anderen Thread erworben. Dem Besitzer der Ressource wird eine Unterbrechung und, wenn dieser sich nicht innnerhalb der aufgerufenen C-Funktion befindet, der Ressource mit der Methode `suspendBlockingCaller` ein

Prioritätswunsch signalisiert. Diese Aktion wird von der virtuellen Maschine atomar durchgeführt. Je nachdem, wo sich der andere Thread zu diesem Zeitpunkt gerade befindet, ergeben sich drei Möglichkeiten:

1. Der Thread befindet sich im Prolog auf dem Weg in den externen Methodenaufruf. Das Eintrittsprotokoll löst eine Ausnahme aus, womit die Ressource freigeben wird. Durch die Prioritätsvorgabe ist der `Commit`-Thread nun als einziger in der Lage die Ressource zu erwerben und das `Commit` fortzuführen.
2. Der Thread befindet sich in der externen C-Funktion. Durch das Austrittsprotokoll wird der Thread so lange blockiert, bis das `Commit` der Ressource beendet wurde. Der Zustand der Ressource wird gesichert, ohne daß diese zuvor erworben wurde. Dies geschieht in der Annahme, daß die C-Funktion selbst den Zustand nicht ändert. Bei einem Neustart des Systems sendet sich der Besitzer der Ressource als erstes eine Ausnahme (siehe Abschnitt 5.2.5).
3. Der Thread ist dabei die Transaktion zu verlassen. Nach Freigabe der Ressource garantiert die Prioritätsvorgabe des `Commit`-Threads, daß dieser die Ressource als nächstes erwerben kann.

Sobald das `Commit` auf der Ressource abgeschlossen ist, wird die Unterbrechungsanforderung zurückgenommen und die Ressource gegebenenfalls freigegeben.

Es ist zu beachten, daß blockierende Transaktionen nicht geschachtelt werden dürfen; in den Fällen 1 und 3 kann es sonst zu einer Verklemmung des Mechanismus kommen. Ebenso dürfen die Methoden `saveState` und `reopen`, die für das Sichern und Wiederherstellen des Ressourcezustandes verantwortlich sind, wie in Abschnitt 5.1.2 bereits angedeutet, nicht synchronisiert werden, um eine Verklemmung für den Fall, daß der `Commit`-Thread eine Ressource nicht erwerben kann, auszuschließen.

Ein wichtiger Punkt bei der Entscheidung, den Bereich für die Auslösung einer Ausnahme bei der Unterbrechung einer Transaktion so eng zu fassen, liegt darin begründet, den TL-2 Programmcode vor und nach der externen Methode ohne Unterbrechung durch ein `Commit` auszuführen. Auf diese Weise können die Konsistenz einer Ressource gefährdende Seiteneffekte auf den C-Funktionsaufruf selbst eingegrenzt werden.

Wird ein persistenter Thread, der zum Zeitpunkt des `Commit` innerhalb einer externen Funktion blockiert war, neu gestartet, so wird automatisch eine Ausnahme ausgelöst und die Transaktion neu gestartet. Aus der Sicht der virtuellen Maschine stellt dies die einzig sinnvolle Reaktion auf die vorgefundene Situation dar. Durch Seiteneffekte können hierbei allerdings Fehler auftreten, wenn etwa beim ersten Mal ein inkonsistenter Zustand hinterlassen wurde oder eine Transaktion ein zweites Mal ausgeführt wird. Diese Problematik ist jedoch nicht Gegenstand dieser Arbeit.

5.2.4 Erzeugen und Beenden von Threads

Die im Entwurf (siehe Abschnitt 5.1.1) definierte Funktionalität für Threads sieht eine konsequente Arbeitsteilung von virtueller Maschine und TL-2 Code vor. Ziel ist es, die Schnittstelle zur virtuellen Maschine auf wenige notwendige elementare Methoden, die z.B. auf das Pthread-API abgebildet werden müssen, zu beschränken und die noch fehlende Funktionalität mit den bereits zur Verfügung stehenden Klassen in TL-2 zu realisieren. Dies erleichtert die Wartung und Erweiterbarkeit und reduziert gleichzeitig die Komplexität der virtuellen Maschine. Deutlich wird dieses Vorgehen bei der Konstruktion und Destruktion von Threads sowie an der Trennung von internem und TL-2 Zustand.

Der erste Schritt bei der Erzeugung eines Threads wird in TL-2 durchgeführt und besteht darin, ein neues Threadobjekt zu allozieren und dessen TL-2 Datenstrukturen in der Klasse `Thread` zu initialisieren:

```
init(inner :Fun0) :Thread
{
  mx := Mutex.new,
  terminated := BroadcastingCondition.new,
  fn := fun() {          ;;  create wrapper function
    let val = try({
      inner[]            ;;  execute user fun
    },
    fun(e :Exception) {
      nil                ;;  return nil in case of unhandled exception
    }),
    mx.lock,
    value := val,
    state := TERMINATED,
    terminated.signal,
    mx.unlock
  },
  let success = __init, ;;  builtin: start OS-thread
  assert success,
  self
}
```

Die initialisierten Datenstrukturen bestehen aus einem Mutex, der den TL-2 Zustand schützt, einer Ereignisvariablen, über die andere Threads auf die Terminierung des neuen Threads warten können, sowie die vom Interpreter auszuführende Funktion höherer Ordnung. Die Funktion kapselt den vom Programmierer als Argument übergebenen Programmcode und erfüllt drei wesentliche Aufgaben:

- Sie fängt Ausnahmen, die nicht innerhalb des benutzerdefinierten Codes behandelt werden, ab und sorgt so für eine korrekte Beendigung des Threads,
- setzt das zurückgelieferte Resultat und

- benachrichtigt alle auf die Terminierung wartenden Threads.

Das Threadobjekt wird im Anschluß an die virtuelle Maschine übergeben. Diese legt einen neuen Laufzeitstapel an, initialisiert die restlichen internen Datenbereiche – die Maschinenregister und den internen Zustand – und registriert das Threadobjekt in der Threadliste des Wurzelobjektes. Nun wird ein neuer Betriebssystemthread gestartet, der, nachdem er seine lokalen Daten gesetzt hat (siehe Abschnitt 5.2.1), in einer neuen Instanz des Bytecodeinterpreters mit der Ausführung des TL-2 Codes beginnt.

Da eine beliebige Funktion zur Ausführung kommen kann, muß es für die Maschine eine Möglichkeit geben, die Beendigung eines TL-2 Threads zu erkennen, um den Betriebssystemthread zu terminieren.[7] Auf den Laufzeitstapel wird deshalb vor den Sicherungsrahmen der initial zu startenden Funktion höherer Ordnung ein spezieller Sicherungsrahmen (*barrier frame*) geschrieben, der einzig auf eine Folge von reservierten Bytecodes verweist (`terminateThread`, siehe Abbildung 5.10). Beendet sich die gestartete Funktionen, d.h. sie kehrt mit einem `return` Bytecode zurück, so wird ihr Sicherungsrahmen abgebaut und der Bytecode des Barriererahmens ausgeführt. Der Bytecode entfernt das Threadobjekt aus der Threadliste des Wurzelobjektes und terminiert den Betriebssystemthread über das Pthread API (`pthread_exit`). Wird bei der Suche nach einer Ausnahmebehandlung der Barriererahmen gefunden, so wird der Thread sofort beendet.[8] In diesem Fall setzt die virtuelle Maschine den TL-2 Zustand und benachrichtigt eventuell wartende Threads.

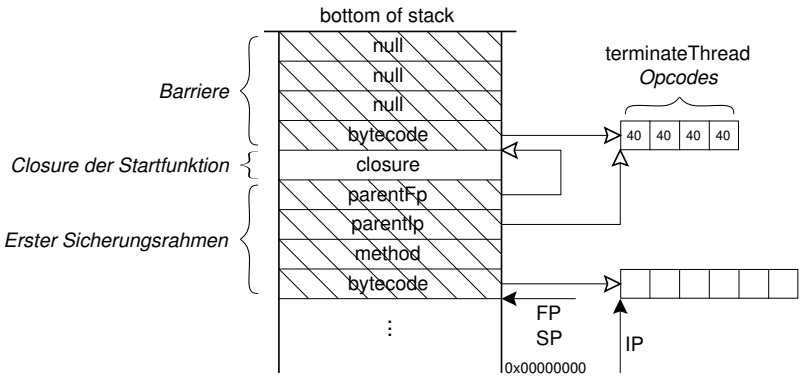

Abbildung 5.10: Barriererahmen der virtuellen Maschine

Auf dem Quellcode von Seite 96 baut die Methode `join` auf, mit der das Ergebnis eines Threads abgefragt wird. Sie ist vollständig in TL-2 implementiert:

[7] Der Interpreter selbst arbeitet in einer Endlosschleife.

[8] Der Barriererahmen wird nur gefunden, wenn der die übergebene Funktion kapselnde TL-2 Code eine Ausnahme auslöst.

```
join() :Object
{
  mx.lock,
  while({state != TERMINATED}, {
    terminated.wait(mx)
  }),
  mx.unlock,
  value
}
```

Das von einem Thread gelieferte Resultat ist somit solange zugänglich, wie das Threadobjekt erreichbar ist. Darüber hinaus kann das Betriebssystem alle mit dem Thread verbundenen Ressourcen sofort bei dessen Beendigung wieder freigeben[9], was bei einem Rückgriff auf die entsprechende Pthread Funktion (`pthread_join`) nicht möglich ist.

Der folgende Programmcode realisiert das Senden bzw. Überprüfen eines Terminierungswunsches, der aus den in Abschnitt 5.1.1 geschilderten Gründen der Kooperation zweier Threads bedarf:

```
cancel() :Void
{
  mx.lock,
  state := (state | CANCELREQUEST),
  _eintr,      ;;  interrupt blocked external call
  mx.unlock
}

testCancel() :Void
{
  assert (self == Thread.this),
  mx.lock,
  (state & CANCELREQUEST) != 0 ? {
    state := (state & (CANCELREQUEST.not)),
    mx.unlock,
    ThreadCancelled.new.raise  ;;  raise exception
  }),
  mx.unlock
}

_eintr() :Void builtin
```

Um sicherzustellen, daß auch ein in einem externen Methodenaufruf blockierter Thread auf einen Terminierungswunsch reagieren kann, existiert die elementare Methode `_eintr`. Sie sendet mit der Pthread Funktion `pthread_kill` ein benutzerdefiniertes Signal an den blockierten Thread. Dabei wird das Verhalten von Unix Betriebssystemen ausgenutzt, blokkierte Systemaufrufe bei einem Signal zu unterbrechen und mit einem Fehlercode zurückzukehren.

[9] Die Betriebssystemthreads werden nach ihrer Erzeugung *detached*.

5.2.5 Start der virtuellen Maschine

Der Start eines Tycoon-2 Prozesses stellt einen zu einem vorigen Zeitpunkt persistent gesicherten Systemzustand wieder her. Die Hauptaufgabe der virtuellen Maschine liegt hierbei darin, alle TL-2 Threads an dem Punkt, an dem sie beim `Commit` unterbrochen wurden, neu aufzusetzen. Die einzelnen Phasen, die während eines Neustarts der virtuellen Maschine durchlaufen werden, sind in Abbildung 5.11 dargestellt. Das gefärbte Petrinetz definiert folgende Farben und Variablen:

Farben:

$$
\begin{aligned}
M &= \{m\} \\
TS &= \{blocked, ccall, waiting\} \\
T &= 2^T S \\
C &= N
\end{aligned}
$$

Variablen:

$$
\begin{aligned}
mt &: M \\
t &: T \\
x, y &: C
\end{aligned}
$$

Der `Tycoon-2` Prozess wird vom Betriebssystem mit einem aktiven Thread (*main thread*) gestartet. Dieser initialisiert, angefangen mit dem Objektspeicher und der Wiederherstellung des persistenten Objektspeicherabbildes, nacheinander alle weiteren Systemmodule. Den Abschluß bildet der Start der `TL-2` Threads, der nun näher betrachtet wird.

Bevor neue Threads erzeugt werden können, muß der Hauptthread die Synchronisationsprimitive, die von der virtuellen Maschine bereitgestellt werden – also alle Exemplare der Klassen `Mutex`, `Condition` und `BroadcastingCondition` – initialisieren, d.h. die äquivalenten Betriebssystemobjekte allozieren und deren Indentifikatoren (*handles*) in den Exemplaren registrieren. Da beim Anlegen eines solchen `TL-2` Objekts auch stets eine schwache Referenz erzeugt wird (siehe Abschnitt 5.1.1) muß hierzu nur die interne Liste von schwachen Referenzen des Objektspeichers nach entsprechenden Verweisen durchsucht werden. Anschließend wird für jeden in der Threadliste des Wurzelobjekts verzeichneten TL-2 Thread ein neuer Betriebssystemthread gestartet. Sind diese Aufgaben erfüllt, wird der Hauptthread beendet.

Jeder neue Thread führt, vor seinem Sprung in den Bytecodeinterpreter und der Fortsetzung seiner `TL-2` Programmausführung, eine eigene Initialisierung durch:

- Er setzt seine lokalen Daten (siehe Abschnitt 5.2.1) – den Kontextpuffer für seine `setjmp` Aufrufe (siehe Abschnitt 4.2.5) und einen Verweis auf sein TL-2 Threadobjekt.
- Er erwirbt alle Mutexe, die er zum Zeitpunkt des `Commit` besessen hat und gleicht auf diese Weise die Zustände von TL-2 Mutexobjekten und den darunterliegenden Betriebssystemobjekten ab. Da ein Mutex einen impliziten Besitzer hat, ist es notwendig, daß jeder Thread dies für seine eigenen Mutexe selber durchführt. Wie der

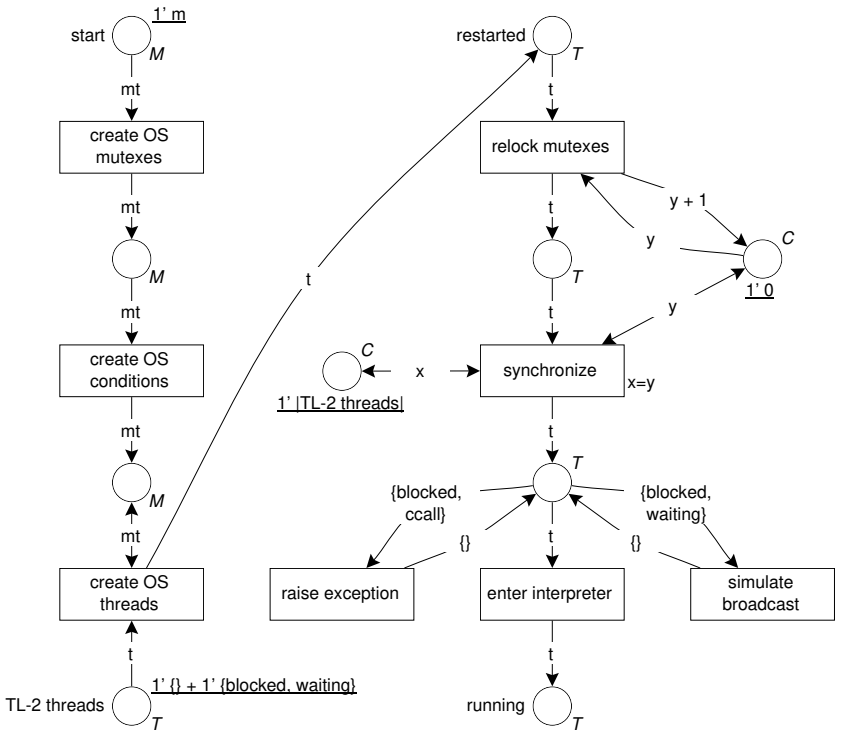

Abbildung 5.11: Neustart der virtuellen Maschine

Hauptthread zuvor bei der Initialisierung muß er die Liste von schwachen Referenzen nach Mutexobjekten durchsuchen, in denen sein TL-2 Threadobjekt als Besitzer eingetragen ist.

Alle Threads müssen vor der weiteren Ausführung ihre Initialisierung abgeschlossen haben. Dies ist notwendig, um beim Erwerb der Mutexe Konflikte auszuschließen. Zu diesen könnte es sonst aufgrund der nicht vorhersehbaren Kontextwechsel kommen, wenn ein Thread bereits seine Ausführung im Bytecodeinterpreter fortsetzt, während ein anderer seine Initialisierung noch nicht abgeschlossen hat.

Bis auf zwei Ausnahmen, die im internen Zustandsvektor eines Threads angezeigt werden und die eine spezielle Behandlung benötigen, springt jeder Thread nach erfolgter Initialisierung und Synchronisation in den Bytecodeinterpreter. Dort wird der Bytecode, der zum Zeitpunkt des `Commit` ausgeführt wurde wiederholt.[10] Die beiden besonders zu behandelnden Fälle sind:

[10] Die eingebaute Methode `commit` manipuliert explizit den Laufzeitstapel und das Instruktionsregister, so daß ein `Commit` von einem Neustart unterschieden werden kann bzw. beim Neustart der auf die `Commit`-Nachricht folgende Bytecode ausgeführt wird.

- Der Thread wartet auf eine Ereignisvariable. Aufgrund der nicht spezifizierten Strategie für die Kontextwechsel darf der Thread das Warten auf die Ereignisvariable nicht einfach fortsetzen, da sonst, wie in den Abbildungen 5.12 und 5.13 illustriert, Signale verlorengehen können und die Synchronisation gestört wird. Statt dessen wird ein *Broadcast* auf die Ereignisvariable simuliert, d.h. der Thread versucht den mit der Ereignisvariablen assoziierten Mutex zu erwerben und fährt danach in seiner Ausführung mit dem folgenden Bytecode fort. Dieses Verhalten ist konform zum Pthread-Standard, der eine unvorhergesehene Rückkehr (*spurious wakeups*) für Ereignisvariablen definiert, und bei Einhaltung der in Abschnitt 5.1.1 beschriebenen Programmierkonvention ohne weitere Auswirkungen.
- Der Thread ist in einem externen Methodenaufruf blockiert. Der Thread wartet, bis die mit dem Methodenaufruf involvierte externe Ressource wiederhergestellt ist, was der `Commit`-Thread über die Rücknahme der Unterbrechungsanforderung (siehe Abschnitt 5.2.3) signalisiert, und löst eine Ausnahme aus. Die Ausnahmebehandlung wird in der Folge vom Bytecodeinterpreter bearbeitet und die unterbrochene Transaktion wiederholt.

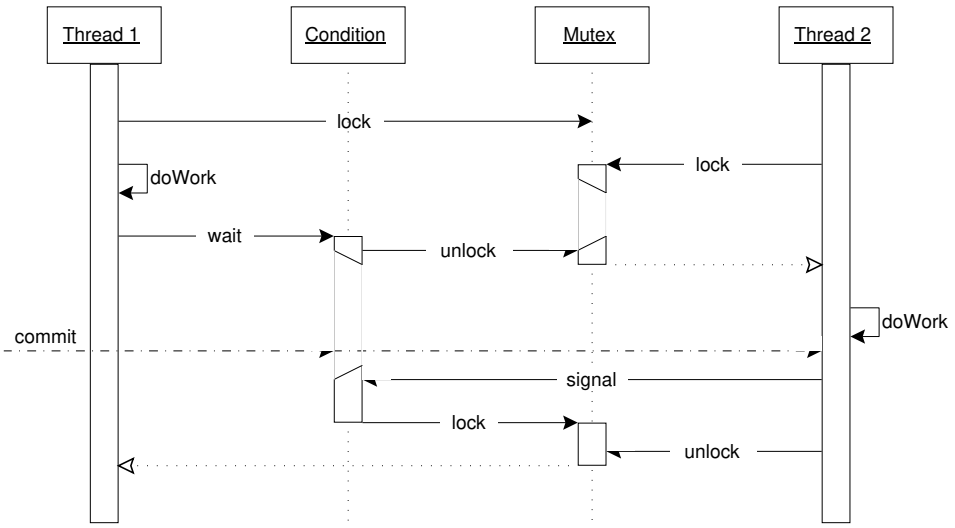

Abbildung 5.12: Unterbrechung der Synchronisation durch ein `Commit`

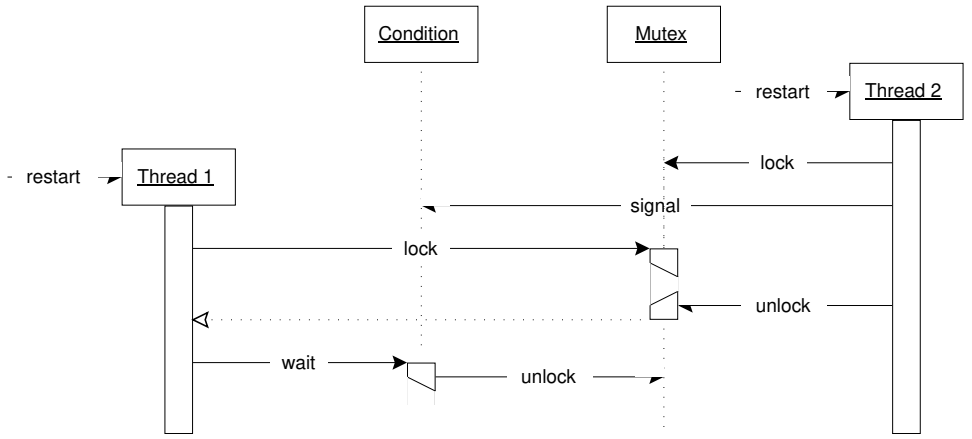

Abbildung 5.13: Verlust eines Signals beim Neustart des Systems

6. Zusammenfassung

Nachdem in den vorangegengenen Kapiteln Entwurf und Implementierung der virtuellen Maschine vorgestellt wurden, erfolgt in den folgenden Abschnitten eine kurze Betrachtung des Laufzeitverhaltens im Vergeich mit dem Prototypen und anderen Programmiersprachen. Eine Bewertung und ein Ausblick auf weitere Entwicklungen beenden diese Arbeit.

6.1 Laufzeitverhalten

Für den Vergleich von Tycoon-2 mit dem auf dem Tycoon-System implementierten Prototypen wurden, wie bereits in anderen Arbeiten [Kir94; Pak96], zwei Standardtestprogramme verwendet, zum einen ein Auszug aus der *Standford Suite* und der *Richards' Benchmark*.

Richards' ist ein ursprünglich in Smalltalk entwickelter Benchmark mittlerer Größe, der das Szenario von Kontextwechseln innerhalb eines Betriebssystemkerns simuliert. Die *Stanford Suite* ist eine Sammlung einfacher Programme, die auf imperativen und funktionalen Sprachkonstrukten, wie z.B. rekursiven Funktionen, Iteration und Feldbearbeitung, basieren. Fünf Programme aus der Sammlung wurden für die Messungen benutzt:

Quicksort: Sortierung eines Feldes von 10000 Ganzzahlen mit dem *Quicksort*-Algorithmus.

Bubblesort: Sortierung eines Feldes von 1000 Ganzzahlen mit dem *Bubblesort*-Algorithmus.

Permutations: Berechnung aller Permutationen über ein Ganzzahlenfeld mit 10 Einträgen.

Queens: 50 maliges Lösen des 8 Damen Problems.

Puzzle: Iterative Feldbearbeitung.

Alle Messungen wurden auf einer Sun Ultra-1 durchgeführt, wobei auf eine geringe Systemauslastung geachtet wurde, um die Ergebnisse nicht zu verfälschen. Beide virtuelle Maschinen wurden mit dem GNU C-Compiler bei maximaler Optimierung übersetzt. Die Ermittlung der jeweils vom Prozeß zur Ausführung einer Aufgabe verbrauchten CPU-Zeit erfolgte mit den systeminternen Zeitgebern über die Standardbibliothek des Betriebssystems.

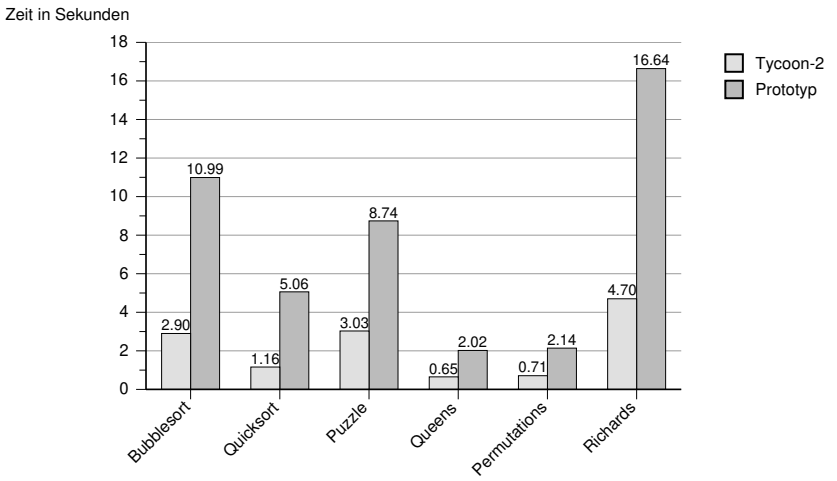

Abbildung 6.1: Laufzeitvergleich zwischen dem Prototypen und Tycoon-2

Die Laufzeiten der verschiedenen Messungen auf den beiden virtuellen Maschinen sind in Abbildung 6.1 gegenübergestellt. Aufgeführt ist jeweils der mittlere Wert einer Reihe von fünf Messungen.

Wie an den Ergebnissen zu erkennen ist, weist das Tycoon-2 System eine durchschnittliche Steigerung der Performanz um den Faktor 3–4 aus. Hauptverantwortlich hierfür sind:

- die wesentlich effizientere Schnittstelle zum Objektspeichersystem,
- die Abstimmung und Optimierung des Interpreters auf die Hochsprache TL-2 und
- die enge Bindung an die Programmierumgebung durch die direkte, gemeinsame Verwendung von Datenstrukturen.

Der überwiegend imperative Charakter der eingesetzten Testprogramme hat keinen Einfluß auf den Vergleich. Da die Programmierumgebungen beider Systeme für die selbe Hochsprache entwickelt wurden und keinerlei Optimierungen bezüglich des generierten Codes vornehmen, ist nur die Effizienz der jeweiligen virtuellen Maschine für die Messergebnisse verantwortlich.

Die in den Messungen sichtbare Geschwindigkeitssteigerung konnte auch bei der Transition der bereits auf dem Prototypen entwickelten Applikationen auf das neue System beobachtet werden.

Für eine realistische Einschätzung der Systemperformanz von Tycoon-2 ist die Betrachtung anderer objektorientierten Programmiersprachen bzw. -systeme unerläßlich. Für einen Vergleich wurden auf der einen Seite Java [GM95b] als ebenfalls plattformunabhängige, auf

einer virtuellen Maschine basierende objektorientierte Programmierspache, auf der anderen Seite C++ als Sprache mit einem plattformspezifischen Übersetzer ausgewählt. Entwicklungsumbegungen waren der Sun Workshop mit seinem C++-Compiler bzw. das Sun JDK in den Versionen 1.0.2 und 1.1.6. Die ältere Version des JDK weist einen in C implementierten Interpreter auf und ist der Version 1.1.6, in der wesentliche Teile der Interpreterschleife in Assembler neu geschrieben wurden, für den Vergleich mit Tycoon-2 vorzuziehen. Auf der anderen Seite zeigt die hierdurch erzielte Leistungssteigerung noch vorhandenes Optimierungspotiential für die virtuelle Maschine von Tycoon-2 auf.

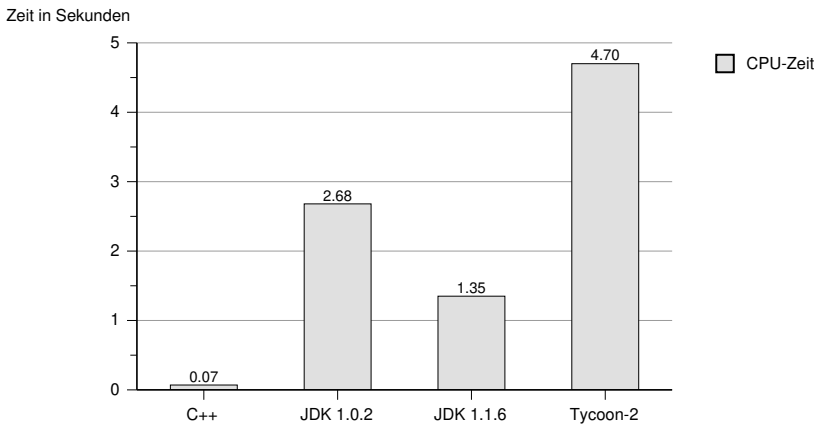

Abbildung 6.2: Ergebnisse des *Richards'* Benchmark

Der erste Vergleich erfolgte mit dem bereits aus den vorigen Messungen bekannten *Richards' Benchmark* auf der selben Testumgebung. Die Resultate in Abbildung 6.2 weisen für Java einen Performanzvorsprung um den Faktor 1,75 bzw. 3,5 gegenüber Tycoon-2 auf. Die C++ Variante zeigt sich bezüglich ihrer Geschwindigkeit den drei virtuellen Maschinen weit überlegen (Faktor 20 bis 70).

Das zweite Testprogramm simuliert mit seinem *Boss-Worker* Modell [NM97] eines der Haupteinsatzgebiete von Tycoon-2: Ein System mit einem oder mehreren Produzenten und mehreren nebenläufigen Konsumenten, wie es beispielsweise in *Client-Server* Applikationen auftritt. Die Kommunikation zwischen *Boss* und *Worker* Threads erfolgt über eine synchronisierte Warteschlange. Um die Vorteile eines Mehrprozessorsystems deutlich zu machen, wurden die Messungen auf einem unter geringer Last laufenden Sun Ultra-2 Server mit 2 Prozessoren vorgenommen.

Die Abbildungen 6.3 und 6.4 zeigen die Ergebnisse bei einem Simulationslauf mit jeweils einem Produzenten und Konsumenten bzw. mit einem Produzenten und zwei Konsumenten. Angegeben ist neben der CPU-Zeit, die sich aus der Addition der auf jeder CPU verbrauchten Rechenzeit ergibt und die im wesentlichen durch die Konsumenten bestimmt wird, auch die vergangene Realzeit. Unterschiede bei den CPU-Zeiten zwischen beiden Läufen liegen

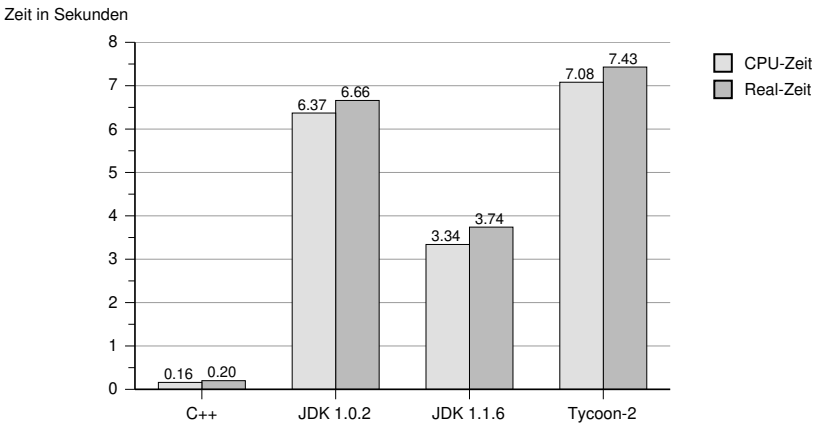

Abbildung 6.3: Ergebnisse des *Boss-Worker* Benchmarks mit 1 Produzenten und 1 Konsumenten

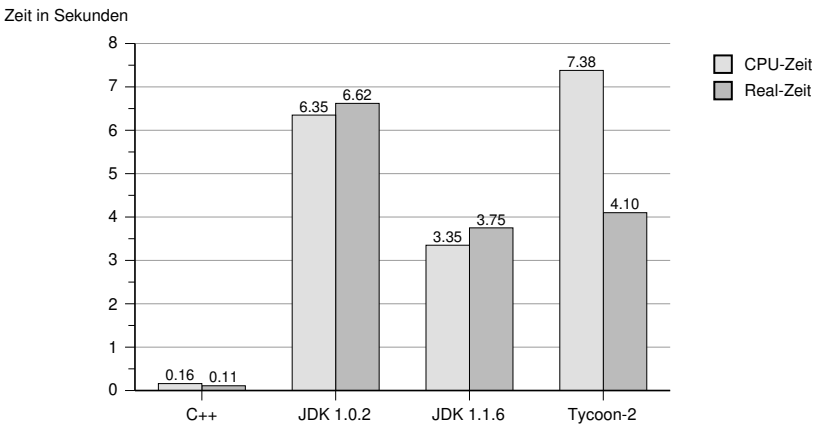

Abbildung 6.4: Ergebnisse des *Boss-Worker* Benchmarks mit 1 Produzenten und 2 Konsumenten

im Bereich der auch bei den anderen Messungen aufgetretenen Schwankungen. Auffällig ist, daß die Geschwindigkeitsunterschiede im Vergleich zum vorigen Test etwas geringer ausfallen, was auf die notwendigen Synchronisationsmechanismen zurückzuführen ist. Tycoon-2 und die C++-Lösung, deren *Multithreading* auf den Pthreads basiert, ziehen Vorteile aus dem zusätzlichen Prozessor.[1]

[1] Sun offeriert eine Erweiterung des JDK 1.1 um native Threads, die jedoch nur unter Solaris 2.6 arbeitet. Diese Betriebssystemversion stand auf keinem Mehrprozessorrechner zur Verfügung.

Wie die Ergebnisse der beiden Benchmarks zeigen, bleibt die Performanz von Tycoon-2 hinter der von Java zurück. Verantwortlich hierfür sind:

- Die reine Objektorientierung von TL-2, da sich der dynamische Methodenaufruf als wesentlicher *bottleneck* des Systems erweist [Pak96; USCH92].
- Die implizite orthogonale Persistenz aller Objekte, die einen höheren Verwaltungs- und Synchronisationsaufwand erfordert.

Gerade diese beiden Punkte, die sich für die Performanz als negativ herausstellen, sind jedoch zugleich die wesentlichen Stärken von Tycoon-2. Dem erhöhten Aufwand zur Laufzeit steht eine höhere Effizienz und Geschwindigkeit bei der Applikationsentwicklung entgegen, womit sich das System besonders für den schnellen Entwurf und die Implementierung von Prototypen eignet. Die reine Objektorientierung erweist sich hierbei gegenüber einer hybriden Sprache durch die saubere Definition des Sprachkerns und den kleinen Satz von Primitiven konzeptionell überlegen [Pak96]. Die orthogonale Persistenz, wie sie TL-2 bietet, kann in dieser Form mit keiner der beiden Vergleichssprachen realisiert werden. Die Sicherung persistenter Daten muß in diesen Sprachen explizit programmiert werden und über das Dateisystem oder Datenbanken erfolgen, womit ihr Geschwindigkeitsvorsprung gegenüber Tycoon-2 erheblich sinkt [Kir94]. Darüber hinaus fehlt ihnen die Persistenz dynamischer Abläufe (Threads) und somit ein mächtiges Modellierungsmittel für neue Paradigmen der Informationsverarbeitung, wie z.B. mobile Agenten [Mat96] oder Workflow Automaten.

6.2 Bewertung und Ausblick

Tycoon-2 ist eine persistente, objektorientierte Programmierumgebung für die Entwicklung von Informationssystemen in heterogenen Rechnernetzen. Basis des Systems ist die in dieser Arbeit vorgestellte virtuelle Maschine mit integriertem Objektspeicher, welche von der zugrundeliegenden Rechnerarchitektur abstrahiert und den höheren Systemebenen eine einheitliche Daten- und Coderepräsentation bietet. Hierdurch ermöglicht sie die Plattformunabhängigkeit der in der Hochsprache TL-2 entwickelten Applikationen.

Wesentliche Merkmale der virtuellen Maschine sind:

- Orthogonale Persistenz von Daten, Code und Threads
- Multithreading
- Persistente Threads auf Basis betriebssystemeigener Threads
- Unterstützung persistenter externer Ressourcen

	Java		Tycoon-2	
Programm		5828		40884
Bibliotheken	libjava	704100	libtm2	133680
			librts	24124
Gesamt		709928		198688

Tabelle 6.1: Größenvergleich der virtuellen Maschinen von Java und Tycoon-2

Die virtuelle Maschine verfügt über einen kleinen, effizienten Kern (siehe Tabelle 6.1) und weist durch die Implementierung in ANSI C und die Einhaltung der von POSIX definierten Standards eine hohe Portabilität auf. Ihre leistungsfähige C-Aufrufschnittstelle öffnet sie externen Diensterbringern und sichert ihre einfache Erweiterbarkeit.

Die Kombination von reiner Objektorientierung, orthogonaler Persistenz und Multithreading erschließt Tycoon-2 spezielle Anwendungsfelder. Hierzu zählen die schnelle Entwicklung von Prototypen (*rapid prototyping*) und der Einsatz als Informationsserver in verteilten Anwendungen.

Es existieren Implementierungen für die Betriebssysteme Solaris 2.x, HP-UX 10.x und 11.x, sowie Linux. Eine Portierung auf Windows NT ist Gegenstand einer laufenden Diplomarbeit.

Auf Tycoon-2 entwickelte Applikationen werden als kommerzielle Informationsserver im WWW beim Springer-Verlag und der Deutschen Presse-Agentur (dpa) eingesetzt. An der Technischen Universität Hamburg-Harburg ist Tycoon-2 Grundlage und Gegenstand weiterer Studien- und Diplomarbeiten, so z.B. im Bereich der Agenten und *Business Conversations* [Weg98] und der komponentenbasierten Programmiersprachen [Wie]. Schwerpunkte bei der Weiterentwicklung liegen in den Bereichen Objektserialisierung und Migration von Objekten zur Laufzeit.

A. Beispiel: *Boss-Worker*

Das folgende Beispiel, ein Auszug der wichtigsten Klassen aus dem *Boss-Worker* Benchmark in Kapitel 6, zeigt die Übersetzung von TL-2 Methoden in den Bytecode der virtuellen Maschine. Aufgeführt sind die einzelnen Klassen für:

- die Produzenten (`Boss` und `BossClass`),
- die Konsumenten (`Worker` und `WorkerClass`) sowie
- die synchronisierte Warteschlange (`WorkQueue`).

Die Routinen zur Ablaufsteuerung sind aufgrund ihrer Größe hier nicht aufgelistet.

Produzenten und Konsumenten werden über die `new`-Methoden in ihren Metaklassen (`BossClass` und `WorkerClass`) erzeugt. Dabei wird ihnen eine vorher erzeugte synchronisierte Warteschlange (`WorkQueue`) bekanntgemacht, über die der Datenaustausch von den Produzenten zu den Konsumenten erfolgt. Für Produzenten wird bei der Erzeugung zusätzlich die Anzahl der zu generierenden Aufträge (`items`) festgelegt.

Gestartet wird er Benchmark durch den Aufruf der `run`-Methoden von Produzenten und Konsumenten, wobei jeweils ein neuer Thread erzeugt wird. Die Produzenten generieren nun die eingestellte Anzahl von Arbeitsaufträgen (`WorkItem`) und legen diese in die Warteschlange (`addWork`). Die Konsumenten lesen die Aufträge (`getWork`) und evaluieren sie.

Die zu verrichtende Arbeit wird als Funktion höherer Ordnung in den Arbeitsaufträgen (`WorkItem`) registriert und kann von den Konsumenten durch die Methode `eval` ausgeführt werden.

Die Warteschlange ist über einen von der Klasse `MutexHolder` geerbten Mutex und die Methode `lock` sowie eine zusätzliche Ereignisvariable (`_barrier`) synchronisiert. Die Warteschlange verfügt über keine Kapazitätsgrenze, sie wächst dynamisch mit der Anzahl der Arbeitsaufträge, d.h. Produzenten werden nicht blockiert. Konsumenten warten bei leerer Warteschlange auf die Ereignisvariable `_barrier`, die bei Hinzufügen eines neuen Arbeitsauftrags signalisiert wird.

Boss

TL-2 Quellcode

```
 1: class Boss
    super Object
    metaclass BossClass

 5: public methods

    run :Thread(Void)
    {
      Thread.new({
10:     for(1, _items, fun(i :Int) {
          let w = WorkItem.new1(fun() {
            let n = Work.new,
            n.doWork,
            i
15:       }),
          _workQueue.addWork(w),
          _workList := List.cons(w, _workList)
        })
      })
20: }

    private

    _workQueue :WorkQueue(WorkItem(Int)),
25: _workList  :List(WorkItem(Int)),
    _items :Int

    methods

30: _init :Self
    {
      _workList := List.new,
      self
    }
```

Bytecode

CompiledMethod: run :Thread(Void)

```
Selector Id: 818 ("run/0")
Stack Peak: 3
Literals: 0: CompiledFun []
.             (* Startfunktion für Thread: Zeile 10-19 *)
.             Selector Id: 6 ("[]/0")
.             Stack Peak: 5
.             Free Variables: 1: "self"
.             Literals: 0: CompiledFun [](i :Int)
.             .             (* Schleifenrumpf: Zeile 11-18 *)
.             .             Selector Id: 7 ("[]/1")
.             .             Stack Peak: 5
.             .             Free Variables: 1: "self"
.             .             Literals: 0: CompiledFun []
```

```
.             .             .             (* Funktion für Worker: Zeile 12-15 *)
.             .             .             Selector Id: 6 ("[]/0")
.             .             .             Stack Peak: 2
.             .             .             Free Variables: 1: "i"
.             .             .                             2: "self"
.             .             .             Literals:
.             .             .             Code:
.             .             .               global 2    ;; self
.             .             .               send0 "Work/0"
.             .             .               send0 "new/0"
.             .             .               local 1
.             .             .               send0 "doWork/0"
.             .             .               pop
.             .             .               global 1    ;; i
.             .             .               return
.             .           Code:
.             .             global 1  ;; self
.             .             send0 "WorkItem/0"
.             .             global 1  ;; self
.             .             arg 0     ;; i
.             .             literal 0
.             .             closure 2
.             .             send1 "new1/1"
.             .             global 1  ;; self
.             .             send0 "_workQueue/0"
.             .             local 1
.             .             send1 "addWork/1"
.             .             pop
.             .             global 1  ;; self
.             .             global 1  ;; self
.             .             send0 "List/0"
.             .             local 1
.             .             global 1  ;; self
.             .             send0 "_workList/0"
.             .             send2 "cons/2"
.             .             sendTail "_workList:=/1"
.           Code:
.             global 1        ;; self
.             one
.             global 1        ;; self
.             send0 "_items/0"
.             global 1        ;; self
.             literal 0
.             closure 1
.             sendTail "for/3"
Code:
  arg 0 ;; self
  send0 "Thread/0"
  arg 0 ;; self
  literal 0
  closure 1
  sendTail "new/1"
```

CompiledMethod: _init :Self

```
Selector Id: 38 ("_init/0")
Stack Peak: 2
```

```
Literals:
Code:
  arg 0 ;; self
  arg 0 ;; self
  send0 "List/0"
  send0 "new/0"
  send1 "_workList:=/1"
  pop
  arg 0 ;; self
  return
```

BossClass

TL-2 Quellcode

```
 1: class BossClass
    super ConcreteClass(Boss)
    metaclass MetaClass

 5: public methods

    new2(queue :WorkQueue(WorkItem(Int)), items :Int) :Boss
    {
      let m = _new,
10:   m._workQueue := queue,
      m._items := items,
      m._init
    }
```

Bytecode

CompiledMethod: new2(queue :WorkQueue(WorkItem(Int)), items :Int) :Boss

```
Selector Id: 745 ("new2/2")
Stack Peak: 3
Literals:
Code:
  arg 2 ;; self
  send0 "_new/0"
  local 1
  arg 1 ;; queue
  send1 "_workQueue:=/1"
  pop
  local 1
  arg 0 ;; items
  send1 "_items:=/1"
  pop
  local 1
  sendTail "_init/0"
```

Worker

TL-2 Quellcode

```
 1: class Worker
    super Object
    metaclass WorkerClass

 5: public methods

    run :Thread(Void)
    {
      Thread.new({
10:     let loop = true,
        while({loop}, {
          let w = _workQueue.getWork,
          w == nil
          ? { loop := false }
15:       : { w.eval }
        })
      })
    }

20: private

    _workQueue :WorkQueue(WorkItem(Int))
```

Bytecode

CompiledMethod: run :Thread(Void)

```
Selector Id: 818 ("run/0")
Stack Peak: 3
Literals: 0: CompiledFun []
.             (* Startfunktion für Thread: Zeile 10-16 *)
.             Selector Id: 6 ("[]/0")
.             Stack Peak: 4
.             Free Variables: 1: "self"
.             Literals:
.             Code:
.               true
.               jump L1
.             L4:
.               sync
.               global 1        ;; self
.               send0 "_workQueue/0"
.               send0 "getWork/0"
.               local 2
.               nil
.               sendEqual "==/1"
.               ifFalse L2
.               false
.               storeLocal 1
.               pop
.               jump L3
.             L2:
```

```
.               local 2
.               send0 "eval/0"
.               pop
.             L3:
.               pop
.             L1:
.               local 1
.               ifTrue L4
.               nil
.               return
Code:
  arg 0 ;; self
  send0 "Thread/0"
  arg 0 ;; self
  literal 0
  closure 1
  sendTail "new/1"
```

WorkerClass

TL-2 Quellcode

```
 1: class WorkerClass
    super ConcreteClass(Worker)
    metaclass MetaClass

 5: public methods

    new1(queue :WorkQueue(WorkItem(Int))) :Worker
    {
      let m = _new,
10:   m._workQueue := queue,
      m
    }
```

Bytecode

CompiledMethod: new1(queue :WorkQueue(WorkItem(Int))) :Worker

```
Selector Id: 53 ("new1/1")
Stack Peak: 3
Literals:
Code:
  arg 1 ;; self
  send0 "_new/0"
  local 1
  arg 0 ;; queue
  send1 "_workQueue:=/1"
  pop
  local 1
  return
```

WorkQueue

TL-2 Quellcode

```
 1: class WorkQueue(T <: Object)
    super MutexHolder
    metaclass SimpleConcreteClass(WorkQueue(T))

 5: public methods

    addWork(work :T) :Void
    {
      lock({
10:     _queue.addLast(work),
        _slaveWaiting > 0
        ? { _barrier.signal }
      })
    }
15:
    getWork() :T
    {
      lock({
        while({_queue.isEmpty && {!_finished}}, {
20:        _slaveWaiting := _slaveWaiting + 1,
           _barrier.wait(_mx),
           _slaveWaiting := _slaveWaiting - 1
        }),
        _queue.isEmpty && {_finished}
25:     ? { nil }
        : { _queue.removeFirst }
      })
    }

30: setFinished :Void
    {
      lock({
        _finished := true,
        _barrier.signal
35:   })
    }

    private

40: _queue :Queue(T),
    _slaveWaiting :Int,

    _barrier :BroadcastingCondition,
    _finished :Bool
45:
    methods

    _init() :Self
    {
50:   super._init,
      _queue := Queue.new1(100),
      _slaveWaiting := 0,
      _barrier := BroadcastingCondition.new,
```

```
      _finished := false,
55:   self
    }
```

Bytecode

CompiledMethod: addWork(work :T) :Void

```
Selector Id: 3699 ("addWork/1")
Stack Peak: 4
Literals: 0: CompiledFun []
.             (* Geschützte Funktion: Zeile 10-13 *)
.             Selector Id: 6 ("[]/0")
.             Stack Peak: 2
.             Free Variables: 1: "work"
.                             2: "self"
.             Literals:
.             Code:
.               global 2        ;; self
.               send0 "_queue/0"
.               global 1        ;; work
.               send1 "addLast/1"
.               pop
.               global 2        ;; self
.               send0 "_slaveWaiting/0"
.               zero
.               send1 ">/1"
.               ifFalse L1
.               global 2        ;; self
.               send0 "_barrier/0"
.               sendTail "signal/0"
.             L1:
.               nil
.               return
Code:
  arg 1 ;; self
  arg 1 ;; self
  arg 0 ;; work
  literal 0
  closure 2
  sendTail "lock/1"
```

CompiledMethod: getWork :T

```
Selector Id: 3687 ("getWork/0")
Stack Peak: 3
Literals: 0: CompiledFun []
.             (* Geschützte Funktion: Zeile 19-27 *)
.             Selector Id: 6 ("[]/0")
.             Stack Peak: 3
.             Free Variables: 1: "self"
.             Literals:
.             Code:
.               jump L1
.             L3:
.               sync
```

```
.               global 1        ;; self
.               global 1        ;; self
.               send0 "_slaveWaiting/0"
.               one
.               sendAdd "+/1"
.               send1 "_slaveWaiting:=/1"
.               pop
.               global 1        ;; self
.               send0 "_barrier/0"
.               global 1        ;; self
.               send0 "_mx/0"
.               send1 "wait/1"
.               pop
.               global 1        ;; self
.               global 1        ;; self
.               send0 "_slaveWaiting/0"
.               one
.               sendSub "-/1"
.               send1 "_slaveWaiting:=/1"
.               pop
.             L1:
.               global 1        ;; self
.               send0 "_queue/0"
.               send0 "isEmpty/0"
.               ifFalse L2
.               global 1        ;; self
.               send0 "_finished/0"
.               ifFalse L3
.             L2:
.               global 1        ;; self
.               send0 "_queue/0"
.               send0 "isEmpty/0"
.               ifFalse L4
.               global 1        ;; self
.               send0 "_finished/0"
.               ifFalse L4
.               nil
.               return
.             L4:
.               global 1        ;; self
.               send0 "_queue/0"
.               sendTail "removeFirst/0"
Code:
  arg 0 ;; self
  arg 0 ;; self
  literal 0
  closure 1
  sendTail "lock/1"
```

CompiledMethod: setFinished :Void

```
Selector Id: 3698 ("setFinished/0")
Stack Peak: 3
Literals: 0: CompiledFun []
.             (* Geschützte Funktion: Zeile 33-35 *)
.             Selector Id: 6 ("[]/0")
.             Stack Peak: 2
```

```
.             Free Variables: 1: "self"
.             Literals:
.             Code:
.               global 1        ;; self
.               true
.               send1 "_finished:=/1"
.               pop
.               global 1        ;; self
.               send0 "_barrier/0"
.               sendTail "signal/0"
Code:
  arg 0 ;; self
  arg 0 ;; self
  literal 0
  closure 1
  sendTail "lock/1"
```

CompiledMethod: _init :Self

```
Selector Id: 38 ("_init/0")
Stack Peak: 3
Literals:
Code:
  arg 0 ;; self
  sendSuper "_init/0"
  pop
  arg 0 ;; self
  arg 0 ;; self
  send0 "Queue/0"
  byte 100
  send1 "new1/1"
  send1 "_queue:=/1"
  pop
  arg 0 ;; self
  zero
  send1 "_slaveWaiting:=/1"
  pop
  arg 0 ;; self
  arg 0 ;; self
  send0 "BroadcastingCondition/0"
  send0 "new/0"
  send1 "_barrier:=/1"
  pop
  arg 0 ;; self
  false
  send1 "_finished:=/1"
  pop
  arg 0 ;; self
  return
```

B. Virtuelle Maschine: Befehlssatz und Datenstrukturen

B.1 Registers

ip instruction pointer

fp frame pointer

sp stack top pointer

B.2 Stackframes

The stack grows downwards (towards zero).

```
[high address]
    [previous frames]
    ...
    arg0 (= self/receiver)
    arg1
    ...
    argn
    saved ip                first control word
    saved fp
    receiver
    CompiledMethod          last control word
fp: byteCode
    local1
    ...
sp: localn
    [first free word]
```

The bytecode is independent of the number of control words since there are special instructions to access local variables.

B.3 Bytecodes

Instructions are encoded as one byte "opcode" followed by argument bytes:

<**selector-id**> 16 bit encoding selector

<**n8**> 8 bit integer

<**n16**> 16 bit integer

<**offset8**> 8 bit offset relative to current instruction

<**offset16**> 16 bit offset relative to current instruction

16 bit arguments are encoded in little endian.

---

send <**selector-id**>

The argument <selector-id> encodes the selector name and number of arguments. Arguments are expected to be pushed from left to right, receiver first. Performs a method lookup on the receiver (argument lowest on the stack):

- If no method is found, a `DoesNotUnderstand` exception is raised.
- If a `CompiledMethod` is found, the stack peak is examined, extending the stack if necessary, and a new stack frame is created and initialized. Execution continues at the beginning of the method's bytecode.
- If a `BuiltinMethod` is found, the builtin code for the method's number gets a chance to handle the call. If it can't, the optional `TL-2` code is executed as in a `CompiledMethod`.
- `SlotAccessMethods` push their receiver's nth slot on the stack.
- `SlotUpdateMethods` set their receiver's nth slot to their argument, and return the argument.
- `CStructAccessMethods` and `CStructUpdateMethods` are similar, but consider the architecture specific offset.

---

send0 <**selector-id**>

send1 <**selector-id**>

send2 <**selector-id**>

send3 <**selector-id**>

send4 <**selector-id**>

send5 <**selector-id**>

Like `send` with fixed number of arguments.

sendSuper <selector-id>

Same as `send`, except that method lookup skips the method dictionaries of the receiver's CPL up to and including the method dictionary which points to the `CompiledMethods` class, i.e. the class in which the current `CompiledMethod` object has been defined in.

---

sendTail <selector-id>

Discards the current frame and arguments, leaving the n topmost stack elements where the arguments in this stack frame were, and sends <selector-id>. Similar effect like `send-tail <selector-id>` followed by a `return`, but avoids harmful stack growth.

---

sendAdd, sendSub, sendLessOrEqual, sendEqual, sendNotEqual, sendFun0Apply,

sendFun1Apply, sendIsNil, sendIsNotNil

Several instructions are allocated as sends with fixed selector. This makes the code more compact (one byte instead of three). In addition, the interpreter knows about builtins (like `Int::"+"`), and in these cases can skip the method lookup.

---

global <n8>

Push the nth global variable. The current `CompiledMethod` object must be a `CompiledFun` created by `closure` <nb_globals>, where $1 \leq n \leq$ nb_globals.

Pseudocode:

```
*(--sp) = fp->receiver[n]
```

---

literal8 <n8>

literal16 <n16>

Push the nth element of the literal array (zero based).

---

arg <n8>

Push one of the nb_args method arguments, where $0 \leq n <$ nb_args. `self` is argument number (nb_args - 1), last argument is 0.

Pseudocode:

```
*(--sp) = fp[nb_controlWords + n]
```

local <n8>

Push the nth local variable, where 1 ≤ n ≤ nb_locals.
Pseudocode:

```
*(--sp) = fp[-n]
```

---

char <n8>

Push `OID` for nth character object.
Pseudocode:

```
*(--sp) = root->charTable[n]
```

---

byte <n8>

Push signed byte extended to tagged int.

---

short <n16>

Push signed short extended to tagged int.

---

true, false, nil, minusOne, zero, one, two

Push frequently used constants.

---

storeLocal <n8>

Store (do not pop) the topmost stack element in the nth local variable, where 1 ≤ n ≤ nb_locals.
Pseudocode:

```
fp[-n] = *sp
```

---

drop <n8>

Pop n elements off the stack.
Pseudocode:

```
sp += n
```

adjust <n8>

Pop n elements under the top element off the stack.

Pseudocope:

```
v = *sp
sp += n
*sp = v
```

---

pop

Pop one element off the stack.

Pseudocode:

```
sp++
```

---

cellNew

Create a new cell for mutable bindings (`Array(OID)` with 1 element), value is on top of stack.

Pseudocode:

```
*sp = newCell(*sp)
```

---

cellLoad

Load value from allocated cell, cell `OID` is on top of stack.

Pseudocode:

```
*sp = (*sp)[0]
```

---

cellStore

Store value into cell, cell `OID` followed by value are on top of stack.

Pseudocode:

```
(*sp)[0] = *(sp + 1)
sp++
```

---

closure <n8>

Create a closure. Stack contents before the instruction:

global_n
...
global_1
sp: `CompiledFun`

These are replaced by the newly created function object.

return

Pops the current frame and arguments from the stack, and leaves the topmost stack element where the arguments were. The number of arguments is accessible through the `CompiledMethod` object.

Pseudocode:

```
retval = *sp
sp = ((Word*)(fp + 1)) + fp->compiledCode->nArgs
ip = fp->parent.ip
fp = fp->parent.fp
*sp = retval
```

---

ifTrue <offset8>

ifTrue <offset16>

If the topmost stack element is not a boolean, the exception `MustBeBoolean` is raised. Otherwise, the element is popped and the instruction pointer is adjusted by offset, a one (or two, for a long jump version) byte two's complement value, if the element is true.

Pseudocode:

```
val = *(sp++),
if(val == true)
  ip += offset
else if(val != false)
  raiseException
```

---

ifFalse <offset8>

ifFalse <offset16>

See `ifTrue`.

---

jump <offset8>

jump <offset16>

Unconditional jump.

Pseudocode:

```
ip += offset
```

---

sync

Check for synchronization requests.

B.4 Structures

B.4.1 Boxed Values

```
typedef struct tyc_Bool {
  Bool value;
} tyc_Bool;

typedef struct tyc_Char {
  Char value;
} tyc_Char;

typedef struct tyc_Int {
  Int value;
} tyc_Int;

typedef struct tyc_Long {
  Long value;
} tyc_Long;

typedef struct tyc_Real {
  Real value;
} tyc_Real;
```

B.4.2 Method Objects

```
typedef struct tyc_Method {
  tsp_OID sourcePos;
  tyc_Symbol pSelector;             /* selector */
  tsp_OID methodType;
  char * pszDocumentation;
  tsp_OID pPrecondition;
  tsp_OID pPostcondition;
  Bool fIsPrivate;
  Word nArgs;                       /* number of arguments */
  void * pNativeCode;               /* for jit compiler */
} tyc_Method;

typedef struct tyc_CompiledMethod {
  tyc_Method method;
  tsp_OID body;
  Word wDebugMode;
  tyc_Class * pClass;
  Byte * pbCode;
  tyc_Array * pLiterals;
  Word idSelector;
  Word cwStackPeak;                 /* stack peak */
  tyc_CatchFrame * asHandlerTable;  /* may be null (no handlers) */
} tyc_CompiledMethod;

typedef struct tyc_BuiltinMethod {
  tyc_CompiledMethod compiledMethod;
  Word iNumber;
  void * pNativeBody;
} tyc_BuiltinMethod;
```

```
typedef struct tyc_CompiledFun {
  tyc_CompiledMethod compiledMethod;
  tsp_OID freeValueIdes;
  void * pNativeBody;
} tyc_CompiledFun;

typedef struct tyc_ExternalMethod {
  tyc_Method method;
  char * pszLanguage;
  char * pszLabel;
  char * pszTycoonArgs;
  char * pszCArgs;
  Bool   fBlocking;                /* mark blocking CCalls */
  void * pFunction;                /* fun address (Int) */
} tyc_ExternalMethod;

typedef struct tyc_slotMethod {
  tyc_Method method;
  Word iOffset;
} tyc_SlotMethod;

typedef struct tyc_SlotAccessMethod {
  tyc_SlotMethod slotMethod;
} tyc_SlotAccessMethod;

typedef struct tyc_SlotUpdateMethod {
  tyc_SlotMethod slotMethod;
  tyc_Symbol slotName;
} tyc_SlotUpdateMethod;

typedef struct tyc_PoolMethod {
  tyc_Method method;
  tsp_OID * pCell;
} tyc_PoolMethod;

typedef struct tyc_PoolAccessMethod {
  tyc_PoolMethod poolMethod;
} tyc_PoolAccessMethod;

typedef struct tyc_PoolUpdateMethod {
  tyc_PoolMethod poolMethod;
} tyc_PoolUpdateMethod;

typedef struct tyc_CSlotAccessMethod {
  tyc_SlotAccessMethod slotAccessMethod;
  char cTycoonType;
} tyc_CSlotAccessMethod;

typedef struct tyc_CSlotUpdateMethod {
  tyc_SlotUpdateMethod slotUpdateMethod;
  char cTycoonType;
} tyc_CSlotUpdateMethod;

typedef struct tyc_Fun {            /* only one struct for all FunNs */
  tyc_CompiledFun * pCompiledFun;
  tsp_OID awGlobals[1];             /* actually 0..n */
} tyc_Fun;
```

B.4.3 Exceptions

```
typedef struct tyc_TypeError {
  tsp_OID pObject;
  tyc_Class * pClass;
} tyc_TypeError;


typedef struct tyc_DoesNotUnderstand {
  tsp_OID pReceiver;
  tyc_Symbol pSelector;
} tyc_DoesNotUnderstand;


typedef struct tyc_MustBeBoolean {
  tsp_OID pObject;
} tyc_MustBeBoolean;


typedef struct tyc_DLLOpenError {
  tsp_OID pDLL;
} tyc_DLLOpenError;


typedef struct tyc_DLLCallError {
  tsp_OID pDLL;
  char * pszEntry;
} tyc_DLLCallError;


typedef struct tyc_DivisionByZero {
  tsp_OID pObject;
} tyc_DivisionByZero;


typedef struct tyc_IndexOutOfBounds {
  tsp_OID pObject;
  tsp_OID pIndex;
} tyc_IndexOutOfBounds;


typedef struct tyc_WrongSignature {
  tsp_OID pMethod;                /* method object found */
  tsp_OID pReceiver;              /* message receiver */
  tsp_OID pArgs;                  /* argument list received */
} tyc_WrongSignature;
```

B.4.4 Special Objects

```
typedef struct tyc_DLL {
  tsp_WeakRef * weakRef;            /* from Resource */
  tsp_OID hDLL;                     /* must be an Int */
  char * pszPath;
} tyc_DLL;


typedef struct tyc_List {
  tsp_OID pHead;
  struct tyc_List * pTail;
} tyc_List;
```

```
typedef struct tyc_Root {
  tyc_Thread * pThread;         /* first thread in linked list */
  tyc_Class ** apClassTable;    /* mapping class id <-> tyc_Class object */
  tyc_Symbol * apSelectorTable; /* mapping selector id <-> tyc_Symbol object */
  Byte       * apArgTable;      /* mapping selector id <-> # of arguments */
  tyc_Bool * pTrue;
  tyc_Bool * pFalse;
  tyc_Nil  * pNil;              /* NULL */
  tyc_Char ** apCharTable;      /* mapping ascii <-> tyc_Char object */
  tsp_OID symbolTable;
  tsp_OID selectorTableSize;
  tsp_OID classTableSize;
  tyc_Finalizer * pFinalizer;   /* weak reference finalizer object */
  tsp_OID storeDescriptors;
  tyc_Bool * pActive;
} tyc_Root;

typedef struct tyc_Finalizer {
  tyc_Mutex * pMutex;
  tyc_Condition * pCondition;
  tsp_WeakRef * pWeaks;
  tyc_Thread * pThread;
  tsp_OID running;
  tsp_OID idle;
} tyc_Finalizer;

typedef struct tyc_Class {
  tsp_OID pos;
  tyc_Long * sourceTime;
  tyc_Long * fingerPrint;
  char * pszName;                   /* class name */
  tsp_OID domain;
  tsp_OID supers;
  char * pszDocumentation;
  tsp_OID selfTypeSig;
  tsp_OID metaClassDeclaration;
  tsp_OID publicSlots;
  tsp_OID privateSlots;
  tsp_OID _methodDictionary;
  tsp_OID wInstanceId;              /* instance id (tagged tyc_ClassId) */
  tsp_OID wInstanceSize;            /* instance size in slots (tagged Int) */
  tsp_OID cpl;
  tyc_List * pMethodDictionaries;   /* list of method dictionaries */
  tsp_OID cplTypes;
  tsp_OID typeIde;
  tsp_OID dependencies;
  tsp_OID classManager;
  tsp_OID _slotMap;
  tsp_OID _slotMethods;
} tyc_Class;

typedef struct tyc_MethodDictionary {
  tsp_OID _elementCount;           /* tagged Int */
  tyc_Symbol * apszKeys;
  struct tyc_Method ** apElements;
  tyc_Class * pClass;
} tyc_MethodDictionary;
```

```
typedef struct tyc_CatchFrame { /* ip relative to beginning of bytecode */
  short ipFrom;                 /* start of exception frame */
  short ipTo;                   /* end of exception frame */
  short ip;                     /* start of handler */
  short cwLocals;               /* locals (words) on stack */
  void * pNativeCode;           /* handler entry points for jit compiler */
} tyc_CatchFrame;               /* internal representation as ShortArray */
```

B.4.5 Synchronization Objects

```
typedef struct tyc_Mutex {
  void * hOSMutex;
  tsp_OID pOwner;
} tyc_Mutex;


typedef struct tyc_Condition {
  void * hOSCondition;
} tyc_Condition;


typedef struct tyc_Thread {
  void * hOSThread;
  tyc_Stack pStack;
  tyc_Stack sp;
  tyc_StackFrame * fp;
  Byte * ip;
  tyc_Stack pStackLimit;          /* set to pStack + tmthread_MAX_STACKPEAK */
  /* flags used by the virtual machine */
  Word wFlags;
  /* abstract state */
  Word flags;
  /* link fields */
  struct tyc_Thread * pNext, * pPrev;
  /* startup closure */
  tyc_Fun * pFun;
  /* tycoon2 fields */
  tsp_OID fun;                    /* startup function */
  tsp_OID value;                  /* return value */
  tyc_Mutex * mutex;              /* mutex syncronizing state */
  tyc_Condition * terminated;     /* condition variable */
  String name;                    /* thread name */
} tyc_Thread;
```

Literaturverzeichnis

[Bar88] J.F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots.* Technical report, Digital Equipment Corporation, Februar 1988.

[BCH+96] K. Barrett, B. Cassels, P. Haahr, D.A. Moon, K. Playford and Withington. *A Monotonic Superclass Linearization for Dylan.* In: *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, San Jose, California*, 1996, S. 69–82.

[BS81] D.G. Bobrow and M. Steifik. *The Loops Manual.* Technical Report LB-VLSI-81-13, Knowledge Systems Area, Xerox Palo Alto Research Center, 1981.

[DCBM89] A. Dearle, R. Connor, F. Brown and R. Morisson. *Napier88 – A Database Programming Language?* In: *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, Juni 1989.

[Dri93] K. Driesen. *Selector Table Indexing & Sparse Arrays.* In: *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Washington DC*, 1993.

[Ern98] M. Ernst. *Typprüfung in einer polymorphen objektorientierten Programmiersprache.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson and J. Vlissades. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GM95a] A. Gawecki and F. Matthes. *TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification.* FIDE Technical Report Series FIDE/95/135, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1995.

[GM95b] J. Gosling and H. McGilton. *The Java Language Environment – A Whitepaper.* Technical report, Sun Microsystems, Oktober 1995.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[IEE96] IEEE. *IEEE/ANSI Std 1003.1, Information Technology – Portable Operating System Interface (POSIX) – Part1: System Application: Program Interface (API)*, 1996.

[Jan95] M. Jantzen. *Petrinetze.* Vorlesungsskript, Fachbereich Informatik, Universität Hamburg, 1995.

[JV87] E. Jessen and R. Valk. *Rechensysteme: Grundlagen der Modellbildung.* Springer-Verlag, 1987.

[Kir94] P. Kiradjiev. *Dynamische Optimierung in CPS-orientierten Zwischensprachen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Dezember 1994.

[KR90] B.W. Kernighan and D.M. Ritchie. *Programmieren in C: ANSI C.* Hanser, 1990.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993.

[Mat96] B. Mathiske. *Mobilität in persistenten Objektsystemen.* Dissertation, Fachbereich Informatik, Universität Hamburg, Oktober 1996.

[MMS97] F. Matthes, R. Müller and J.W. Schmidt. *Towards a Unified Model of Untyped Object Stores: Experience with the Tycoon Store Protocol.* In: M.P. Atkinson (Hrsg.). *Fully Integrated Data Environments.* Springer-Verlag, 1997.

[MSS96] F. Matthes, G. Schröder and J.W. Schmidt. *Tycoon: A Scalable and Interoperable Persistent System Environment.* Technical report, Hamburg University, 1996.

[NBF96] B. Nichols, D. Buttlar and J. Proulx Farrell. *Pthreads Programming.* O'Reilly, 1996.

[NM97] S.J. Norton and M.D.Dipasquale. *Thread Time: Multithreaded Programming Guide.* Prentice Hall, 1997.

[Pak96] M. Pakendorf. *Optimierung der persistenten objektorientierten Programmiersprache TooL.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Juni 1996.

[Pie96] A. Piellusch. *Synchronisation langlebiger Aktivitäten in persistenten Objektsystemen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Juni 1996.

[Sun95] SunSoft. *C 4.0 User's Guide*, November 1995.

[USCH92] D. Ungar, R.B. Smith, C. Chambers and U. Hölzle. *Object, Message, and Performance: How They Coexist in Self.* IEEE Computer (Special Issue on Inheritance & Classification), Jg. 25, Oktober 1992, Nr. 10, S. 53–65.

[Wah98] J. Wahlen. *Entwurf einer objektorientierten Sprache mit statischer Typisierung unter Beachtung kommerzieller Anforderungen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Juni 1998.

[Weg98] H. Wegner. *Objektorientierter Entwurf und Realisierung eines Agentensystems für kooperative Internet-Informationssysteme.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Mai 1998.

[Wei96] M. Weikard. *Dynamische Maschinencodegenerierung für das Tycoon-System.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, Oktober 1996.

[Wie] A. Wienberg. *Komponenten in einer objektorientierten Programmiersprache: Modell, sprachliche Umsetzung und Visualisierung.* Diplomarbeit.

[Wie97] A. Wienberg. *Bootstrap einer persistenten objektorientierten Programmierumgebung.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, August 1997.

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, den 28. Juli 1998

(Marc Weikard)