Universität Hamburg
Fachbereich Informatik
Vogt-Kölln-Straße 30
D-22527 Hamburg

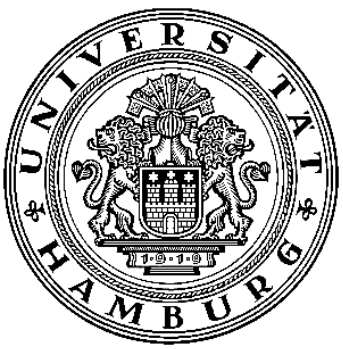
Betreuer:
Prof. Dr. Florian Matthes
Dr. Martin Lehmann

DIPLOMARBEIT

Portierung einer multiprozessorfähigen virtuellen Maschine für eine persistente, objektorientierte Programmiersprache

André Willomat
Lönsring 4b
D-21217 Seevetal

25. Januar 2000



Typographie

In dieser Arbeit wurde eine einheitliche Typographie verwendet, um die Lesbarkeit zu verbessern. Die Bedeutung der einzelnen Schriftarten ist aus der folgenden Tabelle ersichtlich:

Schriftart	**Bedeutung**	**Beispiel**
Emphasized	Wichtiger oder englischer Begriff	*Operating system*
Bold Face	Definition	**Persistenz**
Sans Serif	Firmen- oder Produktname	MIT, Tycoon-2
`Typewriter`	Ausdrücke oder Anweisungen einer Programmiersprache	`tosThread.create` `CreateFile`

Inhaltsverzeichnis

1. Einleitung **1**
- 1.1. Qualitätsmerkmale von Software . . . 1
- 1.2. Von Tycoon nach Tycoon-2 . . . 2
- 1.3. Ziel der Arbeit . . . 4
- 1.4. Gliederung der Arbeit . . . 5

2. Konzept für eine portable Architektur **7**
- 2.1. Portable Systeme . . . 7
  - 2.1.1. Portable Systembibliotheken . . . 7
  - 2.1.2. Virtuelle Maschinen . . . 10
  - 2.1.3. Statische und dynamische Portabilität . . . 14
- 2.2. Entwurf einer portablen Systemarchitektur für Tycoon-2 . . . 15
  - 2.2.1. Aktuelle Systemarchitektur . . . 16
  - 2.2.2. Portable Zielarchitektur . . . 18

3. Übersetzungssteuerung **21**
- 3.1. Softwareentwicklung komplexer Systeme . . . 21
- 3.2. Übersetzungssteuerung mit make . . . 22
- 3.3. Übersetzungssteuerung mit Jam/MR . . . 24
- 3.4. Eine portable Übersetzungssteuerung für Tycoon-2 . . . 27
- 3.5. Konfiguration . . . 31

4. TycoonOS – Eine portable Schnittstelle zum Betriebssystem **33**
- 4.1. Grundbegriffe . . . 33
- 4.2. Architektur von Windows NT . . . 35
- 4.3. Implementierungsaspekte . . . 38
  - 4.3.1. Portable Systembibliotheken für UNIX und Windows NT . . . 38
  - 4.3.2. Anforderungen an das Betriebssystem . . . 40
- 4.4. Schnittstellenübersicht . . . 41
- 4.5. Basisdienste . . . 42
  - 4.5.1. Basisdefinitionen . . . 42
  - 4.5.2. Arithmetikfunktionen . . . 43
  - 4.5.3. Systemnahe Basisdienste . . . 44
  - 4.5.4. Fehlerbehandlung und Ablaufverfolgung . . . 46

4.6. Prozesse und Threads . . . 49
4.6.1. Grundlegende Threadfunktionen . . . 49
4.6.2. Prioritätsebenen und Scheduling . . . 52
4.6.3. Dynamischer threadspezifischer Speicher . . . 55
4.6.4. Synchronisation . . . 57
4.6.5. Prozesse . . . 65
4.6.6. Dynamische Bibliotheken . . . 66
4.7. Dateien und Verzeichnisse . . . 68
4.7.1. Dateifunktionen . . . 69
4.7.2. Verzeichnisfunktionen . . . 71
4.8. Netzwerkdienste und weitere Werkzeuge . . . 73
4.8.1. Netzwerkdienste . . . 73
4.8.2. Datums- und Zeitroutinen . . . 74
4.8.3. Datensignatur . . . 75
4.8.4. Zugriffsrechte und Sicherheitsdienste . . . 76
4.9. Gesamtübersicht . . . 78

5. Konsequenzen auf hochsprachlicher Ebene 79
5.1. Systemdienste . . . 79
5.1.1. Threads und Threadsynchronisation . . . 79
5.1.2. Dateisystemklassen . . . 80
5.1.3. Dynamische Bibliotheken . . . 82
5.1.4. Netzwerkdienste . . . 84
5.2. Ressourcen-Management und mobile Objektspeicher . . . 85

6. Bewertung und Ausblick 91
6.1. Ausblick . . . 92

A. Die Schnittstellen von TycoonOS 97
A.1. Basisdefinitionen . . . 97
A.1.1. tos.h . . . 97
A.1.2. tosDBG.h . . . 97
A.1.3. tosTYPES.h . . . 99
A.2. Arithmetikfunktionen . . . 100
A.2.1. tosInt32.h . . . 100
A.2.2. tosLong.h . . . 101
A.2.3. tosReal.h . . . 101
A.3. Fehlerbehandlung und Ablaufverfolgung . . . 102
A.3.1. tosError.h . . . 102
A.3.2. tosLog.h . . . 103
A.4. Systemmodule . . . 104
A.4.1. tosMemory.h . . . 104
A.4.2. tosString.h . . . 105

A.4.3. tosSystem.h . . . 105
A.5. Threads und Threadsynchronisation . . . 107
A.5.1. tosThread.h . . . 107
A.5.2. tosTLS.h . . . 109
A.5.3. tosMutex.h . . . 109
A.5.4. tosCondition.h . . . 110
A.6. Dateisystemfunktionen . . . 112
A.6.1. tosFile.h . . . 112
A.6.2. tosFileinfo.h . . . 113
A.6.3. tosFilemode.h . . . 114
A.6.4. tosFilename.h . . . 114
A.6.5. tosFilepos.h . . . 116
A.6.6. tosStdio.h . . . 116
A.7. Verzeichnisfunktionen . . . 117
A.7.1. tosDirectory.h . . . 117
A.8. Prozesse und dynamische Bibliotheken . . . 119
A.8.1. tosProcess.h . . . 119
A.8.2. tosDll.h . . . 119
A.9. Netzwerkdienste . . . 121
A.9.1. tosNet.h . . . 121
A.9.2. tosSocket.h . . . 121
A.10. Verschiedene Werkzeuge . . . 123
A.10.1. tosDate.h . . . 123
A.10.2. tosMd5.h . . . 124
A.10.3. tosSecurity.h . . . 125

Zusammenfassung

Um den Anforderungen der immer schnelleren Innovationszyklen in der Softwareentwicklung und der damit einhergehenden Forderung nach immer leistungsfähigeren programmiersprachlichen Abstraktionen gerecht zu werden, entstand an der Universität Hamburg in langjähriger Entwicklungsarbeit die persistente Programmierumgebung Tycoon (*Typed Communicating Objects in Open Environments*) und später, in seiner kommerziellen Weiterentwicklung, das Tycoon-2-System.

Zentraler Bestandteil von Tycoon-2 ist TL-2, eine objektorientierte, polymorphe Programmiersprache höherer Ordnung, deren übersetzte Klassen als Bytecode auf einer virtuellen Maschine ausgeführt werden. Der Grundgedanke ist hierbei, vergleichbar mit Java, durch einen maschinenunabhängigen Bytecode eine globale Übertragbarkeit des Objektcodes zu erreichen. Ein Nachteil in der Systemarchitektur von Tycoon-2 ist bisher die enge Bindung an das POSIX-Threadmodell und der damit schwierigen Übertragbarkeit auf die weit verbreiteten Betriebssysteme von Microsoft.

Das Ziel dieser Diplomarbeit ist die Portierung des Tycoon-2-Systems unter Berücksichtigung seiner multiprozessorfähigen, stark threadbasierten Architektur von UNIX nach Windows NT. Dabei soll die Standardschnittstelle von Windows, die Win32 API (*Application Programming Interface*) benutzt werden, so daß Tycoon-2 auch auf Windows 95 bzw. Windows 98 zur Verfügung steht. Der Schwerpunkt liegt dabei in der Umsetzung der bisher verwendeten POSIX-Threads auf das Threadmodell der Win32 API. Insgesamt soll aber auch die Portabilität von Tycoon-2 im allgemeinen erhöht werden, um die zukünftige Übertragung auf andere Betriebssysteme zu erleichtern.

1. Einleitung

Betrachtet man die aktuelle Entwicklung umfangreicher Softwaresysteme wie Betriebssysteme, Bürokommunikationssysteme (*Office*-Systeme) oder graphische Datenverarbeitungssysteme (CAD, Photo- oder Bildverarbeitungssysteme), so ist eine starke Zunahme an Komplexität und Funktionsvielfalt in diesen Systemen feststellbar. Angesichts der immer kürzeren Produktzyklen einerseits und der enormen Entwicklungskosten dieser Produkte andererseits leidet die Stabilität und damit die zugesicherte Funktionalität dieser Systeme. Beispielsweise stellte Lewis in [Lewis et al. 98] fest, daß Windows NT 5.0 bzw. Windows 2000 bei ca. 27 Millionen Programmzeilen voraussichtlich etwa 6 Millionen Fehler enthalten wird. Eine Reduzierung dieser Fehlerquote auf das Niveau von Windows 95 (inoffiziell 5000 Fehler) würde eine Investition von ca. 200 Millionen US-Dollar in entsprechende Testläufe erfordern.

Von dem Problem mangelnder Zuverlässigkeit bei großen Softwaresystemen scheinen in letzter Zeit auch immer mehr Basisfunktionen betroffen zu sein. Phänomene, wie das sporadische Verschlucken von Fußzeilen bei Microsoft Word oder die fehlerhafte Euro-Umstellung bei Finanzprogrammen wie Money 99 bzw. Quicken 2000 werden zunehmend kritisch in der Fachpresse diskutiert [Loviscach et al. 99; Cube 98; Hollich 99]. In diesem Problemfeld stellt sich die Frage, wie eine hinreichende **Softwarequalität** der Produkte sichergestellt werden kann, und wie korrekt die strategischen IT-Entscheidungen sind, die die Software-Konzerne im Rahmen ihrer Produktentwicklungen treffen müssen. Hier kann die Informatik mit dem umfangreichen Gebiet der **Software-Technik** einige Antworten geben.

1.1. Qualitätsmerkmale von Software

Schon Mitte der siebziger Jahre wurden verschiedene Faktoren der Softwarequalität identifiziert, die bis heute Gültigkeit haben [McCall et al. 77], wie beispielsweise die *Benutzbarkeit*, *Effizienz*, *Korrektheit* und *Wartungsfreundlichkeit* von Softwaresystemen. Heute gibt es umfangreiche ISO- bzw. DIN-Normen sowohl für den Herstellungsprozeß von Software (dem *prozeßorientierten Qualitätsmanagement* im ISO 9000-Normenwerk [ISO 96; ISO 99]), wie auch für deren Qualitätsmerkmale (dem *produktorientierten Qualitätsmanagement* in der ISO 9126 [ISO/IEC 93]).

Nach der ISO 9126 wird **Softwarequalität** als die *"Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf die Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen"* definiert. Als wesentliche Merkmale für die Qualität von Software (**Softwarequalitätsmerkmale**) werden dabei die folgenden genannt:

Funktionalität: Alle Funktionen müssen festgelegte Eigenschaften erfüllen, wobei unter an-

derem folgende Anforderungen erfüllt werden müssen: *Korrektheit* der erzielten Ergebnisse und Wirkungen, *Interoperabilität* in der Zusammenwirkung mit den vorgegebenen Systemumgebungen und *Sicherheit* gegen den unberechtigten Zugriff auf Programme und Daten.

Zuverlässigkeit: Die Fähigkeit der Software, ihre Leistungsfähigkeit unter den festgelegten Bedingungen verfügbar zu halten, wobei neben einer gewissen *Reife* der Software auch *Fehlertoleranz* und *Wiederherstellbarkeit* im Versagensfall gefragt sind.

Benutzbarkeit: Die für den Anwender nötigen Aufwände, um mit der Software umgehen zu können. Diese Aufwände werden durch die Faktoren *Verständlichkeit*, *Erlernbarkeit* und *Bedienbarkeit* maßgeblich bestimmt.

Effizienz: Das Verhältnis zwischen Zeit- und Verbrauchsverhalten der Software und dem Umfang der eingesetzten Betriebsmittel.

Änderbarkeit: Die Aufwände für Korrekturen, Verbesserungen und Anpassungen der Software in der Erwartung, daß die verschiedenen Zyklen im Softwareentwicklungsprozeß mehrfach durchlaufen werden. Im wesentlichen betrifft dies die *Analysierbarkeit* der änderungsbedürftigen Teile, deren *Modifizierbarkeit* und die *Prüfbarkeit* der geänderten Software.

Portabilität: Die Eignung der Software, von einer Systemumgebung in eine andere **übertragbar** zu sein. Hier sind die *Anpaßbarkeit* an verschiedene Hardware- und Softwareumgebungen, die *Installierbarkeit* unter diesen Umgebungen, die *Konformität* zu bestimmten Normen oder Vereinbarungen und die *Austauschbarkeit* gegenüber vergleichbarer Software die wesentlichen Merkmale.

Für eine weitere Vertiefung dieser Thematik wird auf [Balzert 98] verwiesen, diese Arbeit konzentriert sich auf das Qualitätsmerkmal der Übertragbarkeit. Dabei soll konkret die Portabilität eines komplexen Softwaresystems, das an der Universität Hamburg und später kommerziell weiterentwickelte Tycoon-2-System, verbessert werden. Die dabei auftretenden Probleme, wie z.B. die Einführung einer systemunabhängigen Übersetzungssteuerung oder die Kapselung aller systemspezifischen Komponenten in einer separaten Schicht sind eher allgemeiner Natur, so daß auch andere Portierungsprojekte einen Nutzen aus dieser Arbeit ziehen können.

1.2. Von Tycoon nach Tycoon-2

Ein wichtiges Ziel in der Entwicklung der persistenten Programmierumgebung Tycoon (*Typed Communicating Objects in Open Environments*) [Matthes 93] an der Universität Hamburg war es, den immer schnelleren Innovationszyklen in der Softwareentwicklung standzuhalten. Die schon in [Willomat 97] aufgeführte Abbildung 1.1 verdeutlicht diesen Prozeß: Die Entwicklung neuer Anwendungssysteme führt zu dem Bedarf an neuen Abstraktionen, und um

die Komplexität in den weiteren Entwicklungszyklen beherrschbar zu machen, müssen diese Abstraktionen möglichst nahtlos wieder in die Entwicklungssysteme einfließen.

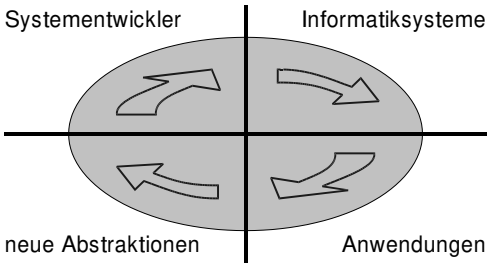


Abbildung 1.1.: Innovationszyklen heutiger Informationssysteme

Diese Fähigkeit wurde in Tycoon durch die leistungsfähigen programmiersprachlichen Abstraktionen der Sprache TL, wozu die Polymorphie oder die Funktionen höherer Ordnung gehören, sowie weiterer herausragender Systemeigenschaften ermöglicht, z.B. der orthogonalen Persistenz durch das einheitliche Objektspeichersystem TSP. Darauf aufbauend konnten neue, komplexere Abstraktionen entwickelt werden, wie beispielsweise **persistente, migrierende Threads** [Mathiske et al. 95; Mathiske 96].

In der folgenden kommerziellen Weiterentwicklung von Tycoon zum Tycoon-2-System [Wienberg 97; Weikard 98] sollten diese Fähigkeiten als Wettbewerbsvorteil am Markt genutzt werden. Insbesondere ein *rein objektorientierter* Ansatz (einheitliches, nachrichtenbasiertes Berechnungsparadigma, Mehrfachvererbung durch Klassenpräzedenzliste, Klassen sind Objekte von Metaklassen etc.) sollte in Tycoon-2 die nahtlose Integration neuer Abstraktionen ermöglichen. Ein aktuelles Beispiel für die Entwicklung und Integration solcher neuen Abstraktionen in Tycoon-2 stellen die **Business Conversations** als Abstraktion zur Modellierung von Geschäftsprozessen dar [Matthes 97; Wegner 98].

Zunächst wurde Tycoon-2 auf den UNIX-Betriebssystemen Solaris und HP-UX sowie in Ansätzen auch auf Linux realisiert. In dieser Entwicklung wurden in [Weikard 98] die eingangs erläuterten Softwarequalitätsmerkmale weitgehend berücksichtigt, vor allem die Übertragbarkeit der virtuellen Maschine durch Einhaltung verschiedener Standards (ANSI C und POSIX) sowie der Effizienz (eine der Schwächen von Tycoon-1) durch Einführung einer multiprozessorfähigen Architektur. Durch die dabei entstandene enge Bindung an die POSIX-Threads bleibt die Übertragbarkeit auf die Betriebssysteme Windows 98 bzw. Windows NT allerdings problematisch, da Microsoft solche offenen Standards nicht unterstützt. So existiert für Windows NT lediglich ein zum POSIX Standard Level 1 kompatibles Subsystem, welches keine Funktionalität für Threads anbietet – dies würde eine Implementierung von POSIX Level 4 erfordern und eine solche Weiterentwicklung ist derzeit fraglich (siehe Abschnitt 4.3.1).

Diese Lücke zu den Microsoft-Betriebssystemen zu schließen und dabei die Übertragbarkeit des gesamten Tycoon-2-Systems zu verbessern, ist das Ziel dieser Diplomarbeit.

1.3. Ziel der Arbeit

Die Aufgabe dieser Diplomarbeit besteht in der Übertragung der virtuellen Maschine und der Kernbibliotheken des Tycoon-2-Systems von UNIX nach Windows NT, unter Benutzung der Win32 API (*Application Programming Interface*). Dabei sind folgende Rahmenbedingungen einzuhalten:

- Die **Übertragbarkeit** des Tycoon-2-Systems soll insgesamt verbessert werden. Von der Portierung sind also nicht nur die virtuelle Maschine, sondern auch die Kernbibliotheken des Systems und die gesamte Softwareentwicklungsumgebung betroffen.

- Das Tycoon-2-System soll auch weiterhin seine multiprozessorfähige Architektur behalten, die auf einer effizienten Implementierung nebenläufig ausführbarer Kontrollflüsse (*Threads*) basiert. Dabei sollen die Threadmodelle der verschiedenen Betriebssysteme unmittelbar ausgenutzt werden (**Native thread support**).

- Die besonderen Spracheigenschaften von Tycoon-2 müssen berücksichtigt werden, wie beispielsweise die **orthogonale Persistenz**. Ein persistenter Objektspeicher soll architekturneutral, also direkt auf andere Systeme übertragbar sein (Migration, **orthogonale Mobilität**).

- Eine langfristige **Koexistenz** und **Wartung** der verschiedenen, betriebssystemabhängigen Bestandteile soll auch in der zukünftigen Weiterentwicklung von Tycoon-2 sichergestellt sein. Dies soll durch die Kapselung dieser Bestandteile (Prinzip der Lokalität und der Modularisierung) und nicht zuletzt auch durch eine gute **Dokumentation** erreicht werden.

- Die systemunabhängigen Basisdienste für die virtuelle Maschine, beispielsweise für Threads, Dateien oder Datums- und Zeitfunktionen, sollen auch auf TL-2-Sprachebene (der eigentlichen Programmiersprache von Tycoon-2) zur Verfügung gestellt werden. Die Schnittstelle zu diesen Systemdiensten wird im folgenden **TycoonOS** (*Tycoon Operating System*) genannt, sie löst das alte, noch aus der Tycoon-1-Zeit stammende RTS (*RunTime System*) ab.

Für eine Durchführung der Übertragung des Tycoon-2-Systems unter den hier geforderten Rahmenbedingungen ergeben sich damit folgende zentrale Aufgaben für diese Arbeit:

1. Die Identifikation und Übertragung eines portablen Architekturkonzeptes auf die virtuelle Maschine des Tycoon-2-Systems.

2. Eine Verbesserung der Übertragbarkeit der Programmierumgebung von Tycoon-2, einschließlich der Übersetzungssteuerung und deren Konfiguration.

3. Die Übertragung und Übersetzung der virtuellen Maschine des Tycoon-2-Systems auf Windows NT als Kernpunkt des praktischen Teils dieser Arbeit.

4. Weitere Anpassungen und Erweiterungen der TL-2-Standardbibliotheken, die durch Einführung der neuen, portablen Architektur notwendig sind.

Aus diesen Aufgaben läßt sich unmittelbar folgende Gliederung für diese Arbeit ableiten.

1.4. Gliederung der Arbeit

In Kapitel 2 werden verschiedene Architekturkonzepte portabler Systeme untersucht. Dabei steht die Betrachtung *virtueller Maschinen* im Umfeld von Programmiersprachen im Vordergrund. Als Konklusio wird ein Architekturkonzept für ein portables Tycoon-2-System hergeleitet und dem aktuellen Ist-Zustand des Systems gegenübergestellt.

Für die praktische Umsetzung dieser Systemarchitektur wird in Kapitel 3 zunächst eine betriebssystemunabhängige Übersetzungssteuerung und deren Konfiguration vorgestellt. Eine Schnittstelle für portable Betriebssystemdienste als Basis für das Tycoon-2-System wird unter Berücksichtigung verschiedener Implementierungsaspekte in Kapitel 4 beschrieben. Dabei steht eine Gegenüberstellung der unterschiedlichen Threadmodelle von Windows NT und UNIX im Mittelpunkt.

Schließlich werden in Kapitel 5 die Auswirkungen der im Rahmen dieser Arbeit eingeführten portablen Architektur von Tycoon-2 auf die TL-2-Standardbibliotheken untersucht. Als Zielvorstellung soll erreicht werden, daß ein persistenter Zustand des Objektspeichers über Betriebssystemgrenzen hinweg austauschbar ist.

2. Konzept für eine portable Architektur

Im folgenden Kapitel werden verschiedene portable Softwaresysteme aus der Praxis untersucht, um die gemeinsamen Architekturkonzepte dieser Systeme kennenzulernen. Neben dem Konzept der Schichtung mittels statischer, *portabler Systembibliotheken* wird das Konzept der *virtuellen Maschine* näher betrachtet. Im Mittelpunkt steht die Frage, welche **Entwurfsmuster** sich für portable Softwaresysteme identifizieren lassen, um diese dann im praktischen Teil dieser Arbeit anzuwenden.

2.1. Portable Systeme

Ein **portables Softwaresystem** ist ein Softwaresystem mit der Fähigkeit, von einer Systemumgebung in eine andere übertragbar zu sein. Diese **Übertragbarkeit** ist beispielsweise nach ISO 9126 [ISO/IEC 93] ein wichtiges Qualitätsmerkmal für Software.

Unternehmen, die *nicht-übertragbare* Softwaresysteme entwickeln, müssen sich bei der Frage nach dem zu unterstützenden Betriebssystem von Anfang an festlegen – ein Wechsel ist später selten noch möglich. Was bei einer solchen strategischen IT-Entscheidung erschwerend hinzukommt, ist der aktuelle Konkurrenzkampf zwischen populären Betriebssystemen, wie beispielsweise zwischen Windows NT und Linux in der nicht ganz unumstrittenen Studie von Mindcraft [Mindcraft 99].

Dagegen haben es vor allem kleine und mittelständische Unternehmen in der Vergangenheit geschafft, sich mit portablen Softwaresystemen einen Wettbewerbsvorteil zu verschaffen. Beispiele hierfür sind Star Division mit der Bürosoftware StarOffice oder Netscape mit ihrer Kommunikationszentrale für das Internet, dem Netscape Communicator. Beide Softwaresysteme benutzen als Grundlage für ihre portable Architektur das Konzept der *portablen Systembibliothek*, welches im folgenden genauer betrachten werden soll.

2.1.1. Portable Systembibliotheken

Eine **portable Systembibliothek** ist eine Bibliothek, die auf unterschiedlichen Betriebssystemen implementiert ist, und über eine auf allen Systemen gleiche, statische Schnittstelle Zugriffe auf dieses Betriebssystem kapselt. Das jeweilige Betriebssystem wird in diesem Zusammenhang auch als **Wirtsbetriebssystem** (*Host operating system*) bezeichnet. Im folgenden werden einige Beispiele für portable Systembibliotheken vorgestellt.

StarView: Star Division setzte die portable Bibliothek StarView [Busch et al. 93] ein, um eine möglichst weite Verbreitung ihrer Bürosoftware StarOffice über die am Markt relevan-

ten Betriebssysteme zu erreichen. Dabei wurde StarView zeitweilig als eigenständiges Produkt vertrieben, um die Entwicklung portabler Softwaresysteme zu fördern – so fand diese Bibliothek auch als Basis für eine portable, graphische Benutzerschnittstelle Einzug in das Tycoon-1-System [Geisler 95; Hachmann 96]. StarView ist eine objektorientierte Klassenbibliothek und stellt neben Basisfunktionen für Dateien, Verzeichnisse und Datums-/Zeit-Routinen vor allem Funktionen für die graphische Benutzeroberfläche zur Verfügung, wie für Fenster, Dialoge oder Zeichensätze. Dagegen fehlen Funktionen zur Unterstützung von *Multithreading*, was sich vor allem in den frühen Versionen von StarOffice in Form eines mangelhaften Laufzeitverhalten bemerkbar machte [Weber, Hüskes 96].

Netscape portable runtime (NSPR): Die NSPR war zunächst ein fester Bestandteil des Netscape Communicators. Sie wurde erst nach der Offenlegung der Quelltexte durch Netscape aus dem Umfeld des Communicators herausgelöst und ist jetzt als eigenständiges, frei erhältliches Softwarepaket verfügbar. Die NSPR bietet vor allem betriebssystemnahe Dienste an, wie z.B. für Threads und Threadsynchronisation, Dateien, grundlegende Netzwerkfunktionen, den Zugriff auf dynamische Bibliotheken und Datums- bzw. Zeit-Routinen. Vor allem die Unterstützung von Threads und deren Synchronisation ist sehr ausgereift, sie basiert auf dem Konzept der **Monitore** nach [Hoare 74]. Abbildung 2.1 verdeutlicht am Beispiel der NSPR, wie eine betriebssystemübergreifende, modulare Softwareentwicklung einzelner Anwendungskomponenten mit Hilfe einer portablen Systembibliothek möglich ist.

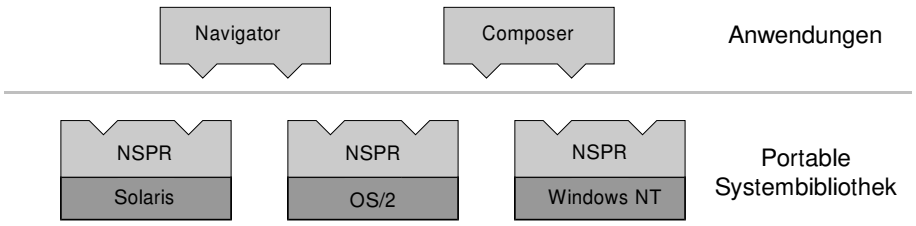


Abbildung 2.1.: Unabhängige Entwicklung von Anwendungsbausteinen durch die portable Systembibliothek NSPR

IBM Open Class Library: Während die bisher betrachteten portablen Bibliotheken im Umfeld konkreter Anwendungssysteme entstanden sind, ist die Open Class Library von IBM [IBM 95; Tsuji et al. 96] von Anfang an als Bibliothek für offene Systemdienste auf verschiedenen Betriebssystemplattformen entwickelt worden. Sie ist ähnlich wie StarView eine objektorientierte Klassenbibliothek, jedoch deutlich umfangreicher. So umfaßt sie zahlreiche Multimedia-Dienste und unterstützt ein offenes, CORBA-basiertes Objektmodell (SOM/DSOM). Daneben gibt es aber auch Basisdienste für Threads und einem elementaren aber systemunabhängigen Synchronisationsmodell, basierend auf Semaphoren.

Diese Beispiele zeigen, daß sich das Prinzip der **Schichtung** mit Hilfe einer einheitlichen Schnittstelle zu einer portablen Systembibliothek als ein wesentliches Architekturkonzept identifizieren läßt. Die Vorteile dieses Architekturkonzeptes sind offensichtlich:

- Neue Anwendungen können auf Basis der portablen Bibliothek entwickelt werden, so daß deren Übertragbarkeit ohne zusätzliche Entwicklungskosten erreicht werden kann.
- Die Einführung einer Zwischenschicht ermöglicht die Bildung neuer Abstraktionen, über die betriebssystemspezifische Details verborgen werden können.
- Die Portierung der Anwendungssoftware auf ein neues Betriebssystem konzentriert sich auf die Übertragung der portablen Systembibliothek: Ist diese Bibliothek erst einmal auf dem neuen System implementiert, so kann die Übertragung der darauf aufbauenden Software fast schon mit einem einfachen Übersetzungsvorgang abgeschlossen werden. Dies kann durch umfangreiche Testprogramme, die schon in einem frühen Entwicklungsstadium die korrekte Funktion der Bibliothek auf der neuen Systemplattform verifizieren, zusätzlich unterstützt werden.

Insbesondere der letzte Punkt spricht für eine Anwendung dieses Architekturkonzeptes für die Portierung des Tycoon-2-Systems nach Windows NT. Neben den hier betrachteten Vorteilen gibt es allerdings auch einige Probleme, die bei der Auswahl bzw. Entwicklung portabler Bibliotheken zu beachten sind. So sollte die Unabhängigkeit vom Betriebssystem nicht durch eine neue Abhängigkeit vom Bibliothekshersteller erkauft werden – die mit der Benutzung einer portablen Systembibliothek verbundenen Lizenzbedingungen sollten daher genau bezüglich solcher Abhängigkeiten geprüft werden.

Ein weiteres zentrales Problem bei der Entwicklung portabler Bibliotheken besteht darin, ob in der Schnittstelle zu der Bibliothek auch Funktionen angeboten werden sollten, die auf einem der unterstützten Betriebssysteme nicht zur Verfügung stehen. Zur Lösung dieses Problems gibt es zwei Ansätze:

- Beim **Schnittmengenansatz** wird nur die gemeinsame Menge der Funktionalität aller Betriebssysteme unterstützt. Damit besteht die Gefahr, daß sich mit einem wachsenden Feld der unterstützten Betriebssysteme die Leistungsfähigkeit dieser Bibliothek *ausdünnen* kann.
- Dagegen wird beim **Obermengenansatz** die fehlende Funktionalität in der Bibliothek nachgebildet, was man oft bei Bibliotheken für graphische Benutzerschnittstellen (wie StarView) antreffen kann [Müßig 94].

Die Nachbildung betriebssystemnaher Funktionen beim Obermengenansatz kann allerdings durch grundlegende Unterschiede in der Betriebssystemarchitektur sehr aufwendig sein, eine Anwendung dieser Lösung ist daher nicht immer einfach möglich. Beispielsweise sind die UNIX-spezifischen *Signale*, die im WEB-Server des Tycoon-2-Systems verwendet werden, nicht ohne weiteres auf die Windows-Betriebssysteme übertragbar. Noch problematischer ist

die Einbindung betriebssystemtypischer Dienste, wie für Multimedia oder spezifische Datenbankanbindungen, z.B. über die *Data Access Objects* Schnittstelle (DAO) unter Windows [Hamilton, Williams 96].

In der Regel kann und soll daher eine portable Systembibliothek nicht das komplette Betriebssystem ersetzen, so daß es immer noch **systemabhängige Bestandteile** in den Anwendungsprogrammen geben kann. In Tycoon-2 kann man diesen Problemen flexibel mit der Möglichkeit begegnen, systemspezifische externe Dienste je nach Bedarf einzubinden. Ein gut durchdachtes **Ressourcen-Management** sollte dann allerdings dafür sorgen, daß es auf Systemen, auf denen diese Dienste nicht zur Verfügung stehen, zu einem wohldefinierten Verhalten kommt (siehe Kapitel 5.2).

Die Übertragbarkeit des bisher betrachteten Architekturkonzepts bezieht sich auf die *Quelltexte* der Programme, durch Nutzung einer portablen Schnittstelle enthalten diese nur noch wenige systemabhängige Bestandteile. Eine andere Form der Übertragbarkeit ist in den letzten Jahren im Zusammenhang mit dem Internet und der Entwicklung von Java populär geworden und findet auch im Tycoon-2-System Anwendung: Die unmittelbare Übertragbarkeit und Ausführbarkeit des *Objektcodes* auf der Zielmaschine. Dies ist aufgrund der unterschiedlichen Hardware der potentiellen Zielmaschinen nur über eine Zwischeninstanz möglich, der im folgenden näher betrachteten **virtuellen Maschine**.

2.1.2. Virtuelle Maschinen

Eine **virtuelle Maschine** ist ein Interpreter für eine **abstrakte Zwischensprache**, die nicht unmittelbar auf einer realen, physikalischen Maschine ausführbar ist. Wie eine reale Rechenmaschine besitzt eine virtuelle Maschine Register, verschiedene Speicherbereiche, die von ihr manipuliert werden und einen von ihr interpretierten Befehlssatz. Dieser Befehlssatz definiert die abstrakte Zwischensprache, die von der virtuellen Maschine interpretiert wird; den Objektcode dieser Sprache bezeichnet man auch als **Pseudocode** oder **Bytecode**.

Eine der wichtigsten Eigenschaften dieses Pseudocodes ist seine Unabhängigkeit von einer spezifischen Prozessorarchitektur – ist auf der Zielplattform ein entsprechender Interpreter verfügbar, ist der Pseudocode auf diese Zielmaschine unmittelbar übertragbar und ausführbar. Aus diesem Grund bilden virtuelle Maschinen die Basis für die Realisierung vieler Programmiersprachen, wie PASCAL, Smalltalk, Visual Basic oder Java.

Historische Entwicklung virtueller Maschinen

Virtuelle Maschinen begleiten die Entwicklung der Programmiersprachen in der Informatik von Anbeginn an. Beispielsweise wurde schon 1961 mit UNCOL (*UNiversal Computer Oriented Language*) der Versuch unternommen, eine universelle Zwischensprache für die damals zahlreichen, im Entstehen begriffenen problemorientierten Programmiersprachen zu entwerfen [Steel 61; Sammet 69]. Später wurde in [Kristensen et al. 74; Nori et al. 75] mit dem **P-Code-Interpreter**, einer virtuellen Maschine für die Programmiersprache UCSD PASCAL,[1] der Grundstein für viele weitere Entwicklungen gelegt. So wurde diese virtuel-

[1] UCSD = University of California, San Diego

le Maschine erweitert, um auch als Ausführungsinstanz für andere Programmiersprachen dienen zu können, z.B. für Fortran [Castaneda et al. 79] oder Modula-2 [Powell 84]. Dabei entwickelte [Castaneda et al. 79] eine **universelle virtuelle Maschine** für die gemeinsame Ausführung von Fortran- und PASCAL-Programmen auf derselben Maschine.

Ein wichtiges Ziel in der Entwicklung virtueller Maschinen war damals neben der Übertragbarkeit des Bytecodes die Entwicklung portabler, modular aufgebauter Übersetzer und die Konstruktion maschinenunabhängiger Optimierer [Sites, Perkins 79; Powell 84] – der *Compilerbau* war damals mangels geeigneter Werkzeuge noch sehr kompliziert.

Heute erlebt das Konzept der virtuellen Maschine mit der zunehmenden Bedeutung des **Internet** in einem Umfeld von stark heterogenen, miteinander vernetzten Systemplattformen eine Renaissance. Um dem Prinzip des "*Write Once, Run Anywhere*" zu folgen, entwickelte die Firma Sun Microsystems die Java Plattform mit der Programmiersprache Java [Gosling, McGilton 95; Arnold, Gosling 98]. Java ist eine objektorientierte Programmiersprache, die durch Konzepte wie der dynamischen Bindung, der Realisierung von Polymorphie und Mehrfachvererbung durch partielle Klassenspezifikationen (den Schnittstellen), der Ausnahmebehandlung oder den kompakten, in der Syntax der Programmiersprache C angelehnten Anweisungen eine hohe Ausdrucksmächtigkeit besitzt, aber dabei trotzdem einfach, robust, sicher und vor allem portabel bleibt.

Parallel zu diesen Aktivitäten bei Sun wurde an der Universität Hamburg die Programmierumgebung Tycoon und dessen Nachfolger Tycoon-2 auf Basis virtueller Maschinen entwickelt. Das Ziel war dabei neben der Übertragbarkeit des Objektcodes die Realisierung besonderer Spracheigenschaften, z.B. der **orthogonalen Persistenz** (Abschnitt 2.2).

Allgemeine Architektur virtueller Maschinen

Der grundlegende Aufbau einer virtuellen Maschine ist in Abbildung 2.2 dargestellt. Dabei werden die im folgenden betrachteten Besonderheiten und Merkmale deutlich.

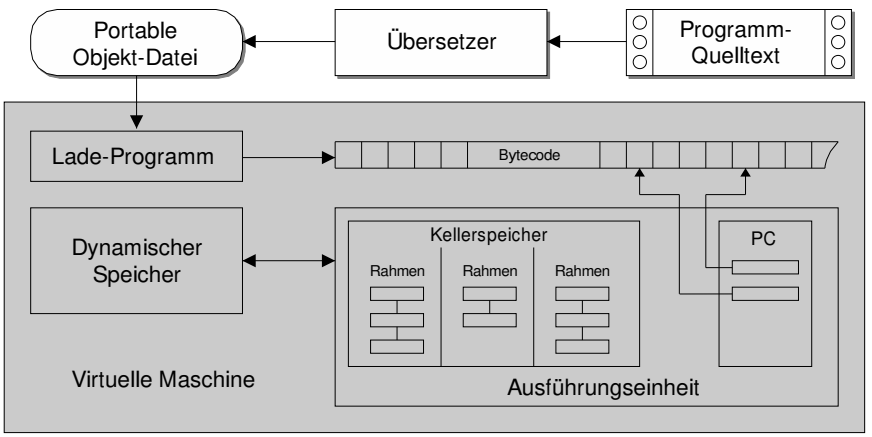


Abbildung 2.2.: Grundlegende Architektur einer virtuellen Maschine

Übertragbare Objektdateien: Der durch den Übersetzer generierte Objektcode wird in einer architekturneutralen Datei, bei objektorientierten Sprachen auch als **Klassendatei** bezeichnet, extern zur Verfügung gestellt. Zum Einlesen dieser Datei benötigt die virtuelle Maschine ein spezielles Ladeprogramm – welches je nach Maschinentyp als *P-Code-Assembler* (bei textuellem P-Code) oder als *Classloader* (bei Klassendateien) bezeichnet wird. Dieses Ladeprogramm nimmt meist schon bei der Bereitstellung des Bytecodes für die virtuelle Maschine einige maschinenspezifische Anpassungen vor, z.B. die Konvertierung von internen Zahlenformaten, wie der Speicherausrichtung (*alignment*) oder der Korrektur der Bytefolge (*byte order*).

Register und Datenbereiche: Eine virtuelle Maschine verfügt über diverse Datenbereiche, von denen einige durch besondere Speicherzellen verwaltet werden, den **Registern**. Zu diesen Registern gehören der **Programmzähler** (*Program Counter*, PC), der auf die nächste auszuführende Instruktion im Bytecode zeigt, sowie Zeiger auf Kellerspeicher und Rahmenbereiche. **Kellerspeicher** (*Stacks*) sind in ein oder mehrere **Rahmen** (*Frames*) unterteilt. Jeder Rahmen kann einer aktuellen Aktivierung (gerade ausgeführte Methode oder Prozedur) zugeordnet werden, er enthält die lokalen Variablen, Parameter und andere Daten dieser Aktivierung. Ein Zugriff auf diese lokalen Daten erfolgt immer ausgehend vom aktuell aktiven Rahmen (LIFO-Struktur des Kellers: *last-in-first-out*). Durch den Kontrollfluß der virtuellen Maschine werden im Verlauf von Methoden- oder Prozeduraufrufen bzw. durch die Beendung von Methoden oder Prozeduren neue Rahmen dynamisch angelegt und auch wieder zerstört.

Dynamische Speicher: Auch frühere virtuelle Maschinen besaßen schon einen **dynamischen Speicher** (*Heap*), in dem einzelne Speicherblöcke je nach Bedarf belegt werden konnten. Im Unterschied zu den früheren Maschinen lassen heutige virtuelle Maschinen (Java oder Tycoon-2) die explizite Freigabe solcher Speicherblöcke nicht mehr zu. Statt dessen übernimmt es die virtuelle Maschine selbst, von Zeit zu Zeit eine **Speicherbereinigung** (*Garbage collection*) vorzunehmen, wobei alle nicht mehr benutzten Speicherblöcke erkannt und wieder freigegeben werden. Durch diesen Automatismus kann in der durch die virtuelle Maschine realisierten Programmierumgebung die ansonsten oft schwierig aufzudeckende Fehlersituation der *hängenden Referenz* vermieden werden [Watt 90].

Multithreading: Eine besondere Eigenschaft der in Abbildung 2.2 dargestellten virtuellen Maschine ist, daß sie die nebenläufige Ausführung mehrerer Threads erlaubt. Um dies zu ermöglichen, muß jeder Thread jeweils eine eigene Instanz verschiedener Register besitzen. Dazu gehören mindestens ein eigener Programmzähler für die dem Thread zugeordneten Instruktionen im Bytecode sowie ein eigener Kellerspeicher, in dem sich der Kontrollfluß des Threads entfalten kann.

Für eine fehlerfreie Funktion einer solchen **threadfähigen virtuellen Maschine** müssen bei allen gemeinsam genutzten Ressourcen interne Synchronisationsmaßnahmen realisiert sein. Dies gilt vor allem für den Zugriff auf Daten im dynamischen Speicherbereich. Bei dem in Abschnitt 2.2 noch näher vorgestellten Tycoon-2-System müssen

während der weiter oben erwähnten Prozedur der Speicherbereinigung sogar alle aktuell laufenden Threads angehalten werden. Der in diesem System benutzte Algorithmus für die Speicherbereinigung basiert auf einem transitiven Kopiervorgang für alle zugreifbaren Daten [Bartlett 88] – nebenläufig durchgeführte Manipulationen könnten dabei an veralteten Objekten stattfinden und damit verlorengehen.[2]

Im Detail können sich virtuelle Maschinen in ihrer Architektur von der Darstellung in Abbildung 2.2 unterscheiden. Dabei stellt sich vor allem die Frage, ob weitere, spezielle Register vorgesehen werden sollten, beispielsweise für die Durchführung von Berechnungen (*Akkumulatoren*). Tatsächlich ist diese Frage kontrovers und hat zu der Entwicklung zweier unterschiedlicher Typen von virtuellen Maschinen geführt:

Kellermaschinen: Viele aktuell entwickelte virtuelle Maschinen, z.B. für Java, Tycoon oder Tycoon-2 sind **Kellermaschinen**. Bei diesen Maschinen wird auf die Verwendung weiterer, spezieller Register verzichtet, die Auswertung von Ausdrücken wird komplett über den Kellerspeicher der Maschine abgewickelt. Oft gibt es noch zusätzliche *Operandenkeller*, so wird bei Java im Zusammenhang mit dem Anlegen eines Rahmens jeweils ein eigener Operandenkeller für alle lokalen Berechnungen angelegt. Der große Vorteil bei Kellermaschinen besteht in deren Unabhängigkeit von einer konkreten Prozessorarchitektur. Bedingt durch die geringe Registerzahl der Maschine wird auch der Instruktionssatz klein und handlich bleiben, was eine effiziente Implementierung der Maschine erlaubt.

Register-Maschinen: Der Einsatz zusätzlicher Spezialregister für eine effiziente Implementierung verschiedener Spracheigenschaften ist die Intention bei den **Register-Maschinen**. Bei diesem Typ von Maschine soll über zusätzliche Register für Werte, auf die ein häufiger Zugriff stattfindet, eine Minimierung von Operandenbewegungen auf dem Kellerspeicher bzw. zwischen den Registern erreicht werden. In [Powell 84] wurde eine so optimierte Register-Maschine für die Programmiersprache Modula-2 entwickelt.

Zu der Frage, welcher Maschinentyp die bessere Performanz besitzt, gibt es einige Untersuchungen. So konnte [Powell 84] eine performante Register-Maschine für die imperative Programmiersprache Modula-2 entwickeln. Für objektorientierte Sprachen wurde in einer früheren Arbeit an der Universität Hamburg [Pakendorf 96] damals noch für TooL, einem objektorientierten Ableger von Tycoon, untersucht, ob die Minimierung von Operandenbewegungen durch eine *dynamische Registerallokation* eine Verbesserung der Performanz bringen kann. Dabei konnte gezeigt werden, daß virtuelle Register-Maschinen nicht bedeutend schneller, oft sogar eher langsamer als virtuelle Kellermaschinen arbeiten. Register-Maschinen haben vor allem dann ein Problem mit der Performanz, wenn eine Abbildung der virtuellen Register auf die Register der physikalischen Maschine nicht möglich ist [Pakendorf 96].

Abschließend seien die unterschiedlichen Fähigkeiten und Eigenschaften verschiedener virtueller Maschinen in Tabelle 2.1 im Vergleich dargestellt. Dabei ist zu erkennen, daß

[2] Es gibt hierfür auch Verfahren, die nebenläufig oder stufenweise arbeiten (*Incremental garbage collection*) [Baker 92]. Einen guten Überblick über diese Verfahren findet man in [Jones, Lins 96].

mit der Einführung neuer Abstraktionen auf Ebene der Programmiersprachen meist auch entsprechende Erweiterungen auf Ebene der virtuellen Maschinen nötig waren (Ausnahmenbehandlungen, Komponenten etc.). Im nächsten Abschnitt folgt zunächst eine kurze Zusammenfassung der bisher betrachteten Architekturkonzepte, bevor diese dann auf das Tycoon-2-System angewendet werden.

Instruktionstyp	**Funktion**	**P-Code**	**Java**	**Tycoon-2**
Kellerspeicher	`pop`, `dup`, `swap` etc.	√	√	√
Arithmetik	Bit-Operationen	√	√	√
	Ganze Zahlen	√	√	√
	Gleitkommazahlen (z.B. IEEE 754)	√ [1]	√	√
	Konvertierung z.B. `long` → `int`	√	√	√
	Fehlerbehandlung	-	√ [2]	√
Verzweigungen	Bedingt (z.B. `if`)	√	√	√
	Unbedingt (z.B. `goto`)	√	√	√
	Ausnahmen (*Exceptions*)	-	√	√
Objektorientierung	Anlegen von Objekten	-	√	√
	Dynamische Methodenauflösung	-	√	√
	Komponenten	-	- [3]	√
Multithreading	Coroutinen	√ [4]	-	-
	Threads	-	√	√
	Synchronisation von Code	-	√	√
	Mehrprozessor-Unterstützung	-	- [5]	√
Sonstiges	Unterstützung von Reflektion	-	√	√
	Dynamisches Laden von Code	-	√	√
	Automatische Speicherbereinigung	-	√	√
	Sicherheitsmechanismen	-	√	-

Tabelle 2.1.: Architektur virtueller Maschinen im Vergleich

[1] Damals noch 4 oder 8 Byte in einem implementationsabhängigen Format.
[2] Über- bzw. Unterlauf durch Werte für ∞ und $-\infty$, illegale Werte durch den Wert `NaN` (*Not a Number*).
[3] Nicht unmittelbar auf Sprachebene, sondern nur als Aufsatz (Java Beans). Für die Integration von Komponenten auf sprachlicher Ebene in Tycoon-2 siehe [Wienberg 99].
[4] In späteren Erweiterungen, z.B. für Modula-2 in [Powell 84].
[5] Die Unterstützung mehrerer Prozessoren ist nur über die direkte Nutzung von Betriebssystem-Threads möglich (*Native thread support*). Dies ist bei Java nur für wenige Systeme realisiert, z.B. für Solaris.

2.1.3. Statische und dynamische Portabilität

Bisher wurden die portablen Systembibliotheken und die virtuellen Maschinen als separate Architekturkonzepte für portable Systeme identifiziert. Beide Konzepte ergänzen sich jedoch gegenseitig: Während portable Systembibliotheken die Übertragbarkeit der *Quelltexte* eines Softwaresystems ermöglichen, erreichen virtuelle Maschinen die unmittelbare Übertragbarkeit des *Objektcodes*. Der eigentliche Unterschied besteht darin, daß im ersten Fall auf dem

gewünschten Wirtsbetriebssystem immer noch ein vollständiger Übersetzungslauf für die übertragenen Programme notwendig ist.

In Anlehnung an den Begriff der statischen und dynamischen Bindung aus dem Bereich der Programmiersprachen (siehe z.B. [Watt 90] Kapitel 4.3.2: *Scope and visibility*) werden an dieser Stelle äquivalente Begriffe für die unterschiedlichen Formen der Übertragbarkeit definiert: Portable Systembibliotheken ermöglichen eine Übertragbarkeit der Programm-Quelltexte und damit eine **statische Portabilität**, während virtuelle Maschinen eine Übertragbarkeit des Objektcodes *zur Laufzeit* erreichen, also eine **dynamische Portabilität**. In Tabelle 2.2 findet sich eine Übersicht über die damit bisher klassifizierten Architekturkonzepte für portable Systeme.

Art der Übertragbarkeit	**Übertragbares Objekt**	**Architekturkonzept**	**Beispiel**
Statisch	Quelltext	Bibliotheksschnittstelle	StarView, NSPR
Dynamisch	Objektcode	Virtuelle Maschine	Java, Tycoon-2

Tabelle 2.2.: Architekturkonzepte für portable Systeme

Für die Entwicklung des Tycoon-2-Systems war die dynamische Portabilität, mit einer auf die Anforderungen der Hochsprache TL-2 zugeschnittenen virtuellen Maschine, ein zentrales Entwurfsziel [Weikard 98]. Die Übertragbarkeit der virtuellen Maschinen selbst sollte durch Einhaltung bestimmter Standards, wie ANSI C und POSIX erreicht werden, was zu der schon in der Einleitung angedeuteten Problematik bezüglich der Betriebssysteme Windows NT und Windows 98 von Microsoft führte.

Eine andere Möglichkeit, die Übertragbarkeit der virtuellen Maschine zu verbessern, besteht in der Kombination der bisher betrachteten Architekturkonzepte für portable Systeme: Mit der Realisierung einer virtuellen Maschine auf Basis einer portablen Systembibliothek wird der Übergang zu einer **portablen virtuellen Maschine** geschaffen. Wie dieses Architekturkonzept auf das Tycoon-2-System übertragen werden kann, wird im folgenden Abschnitt näher beschrieben.

2.2. Entwurf einer portablen Systemarchitektur für Tycoon-2

Nach einer einleitenden Untersuchung über die aktuelle Architektur des Tycoon-2-Systems (Ist-Zustand) wird das Konzept der *portablen virtuellen Maschine* auf diese Architektur übertragen (Ziel-Zustand). Neben den schon im Abschnitt 2.1.1 aufgeführten Vorteilen, die mit einer solchen Schichtung und Kapselung aller betriebssystemabhängigen Bestandteile einhergehen, ergeben sich im Kontext virtueller Maschinen wie bei Tycoon-2 noch weitgehendere Möglichkeiten – vor allem sind auch positive Rückwirkungen auf die Hochsprache TL-2 zu erwarten, was noch in Kapitel 5 näher betrachtet wird.

2.2.1. Aktuelle Systemarchitektur

Zentraler Bestandteil des Tycoon-2-Systems ist die objektorientierte Programmiersprache höherer Ordnung TL-2. Eine detaillierte Einführung in diese Programmiersprache erhält man z.B. in [Wahlen 98; Wienberg 97], an dieser Stelle seien nur kurz einige ihrer Schlüsseleigenschaften aufgeführt:

- Die Sprache TL-2 ist eine reine, objektorientierte Programmiersprache: Alle Objekte verkörpern abstrakte Datentypen erster Klasse, was auch für einfache Typen, wie Zahlen oder Zeichenketten gilt. Es gibt keine besonderen Ausnahmen für *primitive Datentypen*, wie beispielsweise in Java.

- Das einzige Berechnungsparadigma besteht in dem Austausch von Nachrichten zwischen Objekten. Auch primitive Operationen, wie z.B. für die Ganzzahlarithmetik, werden über den Austausch solcher Nachrichten abgewickelt.

- Klassen sind Exemplare von Metaklassen, so daß auch die Erzeugung und Initialisierung neuer Objekte über einen Nachrichtenaustausch stattfinden kann. Die Funktion `new` zum Anlegen neuer Objekte ist in vielen objektorientierten Programmiersprachen ein Sonderfall im Entwurf der Sprache, in TL-2 ist sie eine einfache Methode aus der entsprechenden Metaklasse.

- In TL-2 können auch Funktionen definiert werden: Diese sind als Objekte erster Klasse implementiert, die sogenannte *Auswertungsnachrichten* verstehen können. Auch die Definition von *Funktionen höherer Ordnung* [Watt 90] ist möglich, also von Funktionen, die als Parameter oder Resultat wiederum Funktionen verwenden. Alle Kontrollstrukturen der Sprache, wie Schleifen oder Bedingungen, sind nicht fest in die Sprache eingebaut, sondern über solche Funktionen höherer Ordnung realisiert.

- Der in TL-2 geschriebene statische Typüberprüfer der Sprache beherrscht zwei Ausprägungen der Polymorphie: Die strukturelle Subtypisierung[3] und den begrenzten, parametrischen Polymorphismus [Cardelli, Wegner 85; Ernst 98].

Genauso wie der Typüberprüfer der Sprache TL-2 ist auch der Übersetzer vollständig in TL-2 geschrieben. Er erzeugt Bytecode für die virtuelle Maschine des Tycoon-2-Systems, der TVM (*Tycoon Virtual Machine*), mit der die dynamische Portabilität des TL-2-Objektcodes erreicht wird. Ein wichtiger Bestandteil dieser virtuellen Maschine ist der **Objektspeicher**, der die Rolle des dynamischen Speicherbereichs, wie er bei allen virtuellen Maschinen zu finden ist (Abschnitt 2.1.2), übernimmt. In diesem Objektspeicher findet die Speicherverwaltung aller Objekte statt; auch der Übersetzer und der Typüberprüfer der Sprache TL-2

[3] Die Subtypisierung wird in objektorientierten Sprachen oft mit der Vererbungshierarchie von Klassen gleichgesetzt. Eine solche Subtypbeziehung muß jedoch immer explizit ausgedrückt werden, während bei der *strukturellen Subtypisierung* eine induktive Kontrolle der Typstruktur stattfindet, und so auch Objekte subsumiert werden können, die nicht in expliziter Vererbungsbeziehung stehen [Ernst 98].

sind Bestandteil dieses Objektspeichers. In Abbildung 2.3 ist die Architektur des Tycoon-2-Systems dargestellt.

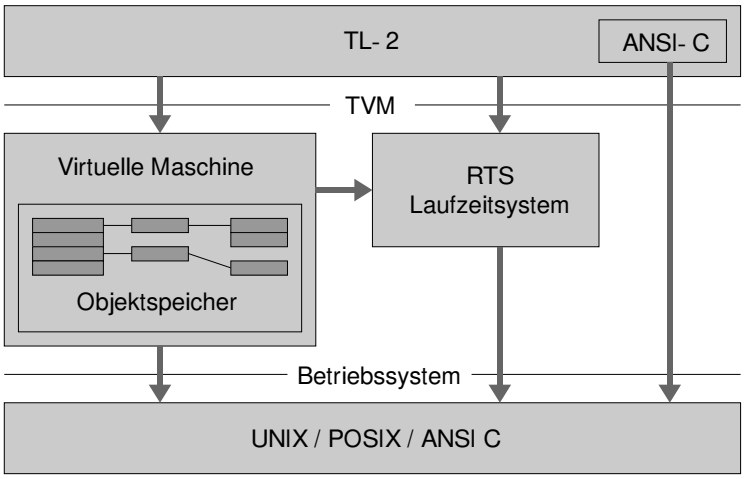


Abbildung 2.3.: Architektur des Tycoon-2-Systems (Ist-Zustand)

Neben den oben beschriebenen Spracheigenschaften von TL-2 gibt es noch eine Reihe weiterer, wichtiger Tycoon-2-Systemeigenschaften, die durch diese Architektur der virtuellen Maschine im Zusammenspiel mit dem Objektspeichersystem ermöglicht werden und die für diese Arbeit von besonderem Interesse sind:

Nebenläufigkeit: Wie schon in Abschnitt 2.1.2 dargestellt wurde, muß eine virtuelle Maschine einige Voraussetzungen erfüllen, um für die nebenläufige Ausführung von Threads geeignet zu sein. In Tycoon-2 werden diese Voraussetzungen dadurch erfüllt, daß jeder TL-2-Thread eine eigene, vollständige Instanz der virtuellen Maschine besitzt, mit allen Registern und jeweils einem eigenen Kellerspeicher. Durch Synchronisationsmaßnahmen wird dabei der Zugriff dieser TL-2-Threads auf gemeinsame Ressourcen, wie dem Objektspeicher, gesteuert. Dazu werden vor allem Bedingungsvariablen (*Condition variables*) und Mutexe benutzt (siehe Abschnitt 4.6.4). In der Implementierung dieser Mechanismen basiert Tycoon-2 auf dem POSIX-Standard IEEE 1003.1c, der unter dem Synonym Pthreads bekannt geworden ist (Abschnitt 4.2).

Multiprozessorfähigkeit: Bei einer direkten Unterstützung des Threadmodells vom jeweiligen Betriebssystem kann eine **Multiprozessorfähigkeit** erreicht werden, so daß die nebenläufigen TL-2-Threads über verschiedene Prozessoren verteilt werden können.

Orthogonale Persistenz: Das Konzept der **Persistenz** bedeutet, daß die Lebensdauer beliebig komplexer Daten die Dauer einzelner Betriebssystemprozesse übersteigen kann: Daten können solange existieren wie die (abstrakten) Prozesse, die sie benötigen. Ist dies für alle semantischen Konzepte einer Sprache durchgängig möglich (Anwendung

des Prinzips der Typvollständigkeit auf die Persistenz [Watt 90]), spricht man von **orthogonaler Persistenz** [Atkinson, Bunemann 87].

Diese Fähigkeit wird in Tycoon-2 dadurch erreicht, daß der Objektspeicher jederzeit in einem **Objektspeicherzustand** (*Object-Store*) gesichert werden kann. Mit diesem kann der Objektspeicher zu einem beliebigen späteren Zeitpunkt vollständig wiederhergestellt werden. Für einige Sprachelemente müssen dabei allerdings besondere Aktivitäten durchgeführt werden: So sind für TL-2-Threads, die im Objektspeicherzustand mitgesichert werden, besondere Synchronisationsmaßnahmen für die Verteilung von Ressourcen beim Neustart der virtuellen Maschine nötig, um Verklemmungen zu verhindern [Weikard 98]. Auch bestimmte Ressourcen, die vom Betriebssystem benutzt und über sogenannte *Handles* verwaltet werden, müssen beim Wiederanlauf der Maschine erneut angefordert werden (Abschnitt 5.2).

Orthogonale Mobilität: Die Brauchbarkeit eines Objektspeicherzustandes für den Wiederanlauf über Betriebssystemgrenzen hinweg wird im wesentlichen durch seine externen Bindungen bestimmt. Durch Einführung einer portablen Systembibliothek für die virtuelle Maschine und durch Nutzung dieser Bibliothek auf TL-2-Sprachebene wird sich die Übertragbarkeit eines Objektspeicherzustandes erhöhen lassen, er wird **mobil**.

Ein wichtiges Ziel dieser Arbeit ist es, eine Übertragbarkeit in Bezug auf den größten Teil der TL-2-Standardbibliotheken zu erreichen. Für einige Ausnahmefälle kann diese Mobilität allerdings nicht realisiert werden, da bestimmte Systemdienste nur auf einigen Betriebssystemen zur Verfügung stehen (z.B. für *Signale*, siehe Abschnitt 4.6.5).

2.2.2. Portable Zielarchitektur

Die in Abbildung 2.3 dargestellte Architektur des Tycoon-2-Systems zeigt auch durch graue Pfeile markierte Abhängigkeiten auf, die bei Beginn dieser Arbeit vorgefunden wurden. So ist unter anderem ersichtlich, daß zum Teil von der TL-2-Sprachebene aus direkte Zugriffe auf das Betriebssystem stattfinden können. Eine eingehendere Analyse der Systemkomponenten aus Abbildung 2.3 bezüglich ihrer Übertragbarkeit kam zu folgenden Ergebnissen:

- Das **Laufzeitsystem** RTS umfaßt eine lose Sammlung verschiedener betriebssystemspezifischer Routinen, die als Systemdienste auf TL-2-Ebene zur Verfügung gestellt werden. Diese Kapselung betriebssystemspezifischer Routinen wird allerdings bisher noch nicht für die virtuelle Maschine oder für die Routinen des Objektspeichersystems genutzt, beide Systemkomponenten benutzen das Laufzeitsystem nur sporadisch.

  Für die Realisierung dieses Laufzeitsystems wurden sowohl portable Systemroutinen (ANSI C oder BSD-Sockets) wie auch verschiedene UNIX-spezifische Routinen benötigt. Zu den letzteren gehören beispielsweise Funktionen aus `unistd.h` für das Durchsuchen von Dateiverzeichnissen oder für die Abfrage spezieller Prozeßinformationen.

- In der **virtuellen Maschine** des Tycoon-2-Systems werden im großen Umfang Routinen für POSIX-Threads benutzt, sowie einige betriebssystemspezifische Funktionen

für Dateien und dynamische Bibliotheken. Daneben finden sich auch einige Spezialitäten, die nicht von jedem C-Übersetzer verarbeitet werden können, beispielsweise der Datentyp `long long int` für 64-Bit Ganzzahlen.

- Das **Objektspeichersystem** benutzt einige systemspezifische Dateioperationen, ist aber schon im Gegensatz zu den anderen Komponenten für eine Übertragbarkeit auf andere Betriebssysteme gut vorbereitet. So findet schon eine automatische Anpassung der Objektspeicher an unterschiedliche, prozessorabhängige Zahlenformate statt (Speicherausrichtung bzw. Bytefolge).

- Die Standardklassen auf TL-2-**Sprachebene** weisen durch die Nutzung des Laufzeitsystems RTS schon einen hohen Grad der Übertragbarkeit auf. Es bleiben vor allem einige semantische Probleme, so würde die Komposition bzw. Dekomposition von Dateinamen bisher auf den Windows-Betriebssystemen scheitern. Einige Abstraktionen des Laufzeitsystems RTS erleichtern die Übertragbarkeit auf andere Systeme erheblich, z.B. die strikte Trennung zwischen Einzel- und *Broadcast*-Signalen bei den Bedingungsvariablen, ohne die eine Übertragung auf das Threadmodell von Windows gar nicht möglich wäre (siehe Abschnitt 4.6.4).

Der letzte Punkt verdeutlicht einen der schon in Abschnitt 2.1.1 erwähnten Vorteile, die eine Schichtenarchitektur mit sich bringen würde. Die Einführung einer portablen Systembibliothek hätte auch die weitgehende Entkopplung der in Abbildung 2.3 dargestellten Abhängigkeiten der Tycoon-2-Systemkomponenten vom Betriebssystem zur Folge und würde die Übertragbarkeit des Tycoon-2-Systems insgesamt verbessern, was ein sehr wichtiges Ziel dieser Arbeit ist. Die daraus resultierende Architektur einer *portablen virtuellen Maschine* als Ziel-Zustand für das Tycoon-2-System ist in Abbildung 2.4 dargestellt:

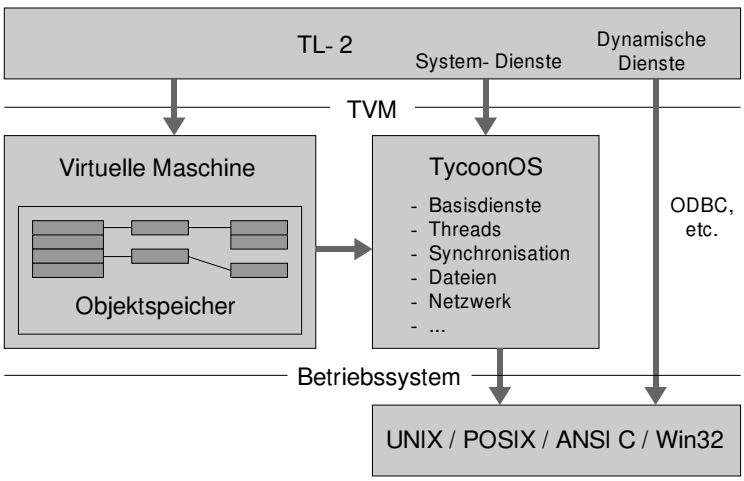


Abbildung 2.4.: Portable Architektur des Tycoon-2-Systems (Ziel-Zustand)

Ausgehend von der Analyse der Tycoon-2-Systemkomponenten lassen sich die im praktischen Teil dieser Arbeit nötigen Hauptaufgaben damit klar spezifizieren. Dabei steht die Realisierung der in Abbildung 2.4 dargestellten portablen Systembibliothek TycoonOS (*Tycoon Operating System*), als Basis für die portable virtuelle Maschine des Tycoon-2-Systems, im Mittelpunkt:

- Alle weiterhin benötigten Systemdienste aus dem ehemaligen Laufzeitsystem RTS müssen in TycoonOS übernommen werden, die Schnittstelle zu diesen Diensten muß dabei in eine übertragbarere Form gebracht werden. Für die TL-2-Standardbibliothek wird dies einige Anpassungen zur Folge haben, von denen einige in Kapitel 5 näher betrachtet werden.

- Alle für die virtuelle Maschine und das Objektspeichersystem notwendigen Systemdienste müssen in TycoonOS realisiert werden. Hierzu zählen vor allem eine übertragbare Schnittstelle für Threads und deren Synchronisationsmechanismen, aber auch für Dateifunktionen und für die Benutzung dynamischer Bibliotheken.

- Es soll die Möglichkeit genutzt werden, mit TycoonOS weitere Abstraktionen und Vereinheitlichungen in das Tycoon-2-System einzuführen. Hierzu zählen einheitliche Dienste zur Ablaufverfolgung (*Tracing*) und einem betriebssystemunabhängigen Fehlermanagement (Abschnitt 5.1.3).

- Neben TycoonOS soll weiterhin die Möglichkeit bestehen bleiben, von TL-2-Ebene aus direkt auf spezifische, externe Betriebssystemdienste (z.B. für ODBC) zugreifen zu können. Diese Fähigkeit sollte allerdings die Mobilität der Objektspeicherzustände nicht negativ beeinträchtigen – auch bei Systemen, auf denen diese **dynamischen Dienste** nicht zur Verfügung stehen. Dies ist im Rahmen des schon in Tycoon-2 bestehenden Ressourcen-Managements realisierbar und wird in Kapitel 5 näher betrachtet.

Eine ausführliche Beschreibung der über die portable Systembibliothek TycoonOS angebotenen Dienste erfolgt in Kapitel 4, eine Auflistung der dazugehörigen Schnittstellen befindet sich in Anhang A.

Im folgenden Kapitel wird zunächst ein Problem behandelt, welches während der Analyse der bestehenden Tycoon-2-Systemkomponenten offensichtlich wurde: Bevor im Rahmen der praktischen Arbeit auch nur die erste Zeile Code auf dem neuen Betriebssystem übersetzt werden kann, muß die Übersetzungssteuerung in Form der *Makefiles* übertragen werden. Diese Übersetzungssteuerung ist aber im höchsten Maße UNIX-spezifisch und nicht ohne weiteres auf die Umgebung von Windows NT übertragbar.

3. Übersetzungssteuerung

In diesem Kapitel wird die Übertragbarkeit der Softwareentwicklungsumgebung für das Tycoon-2-System untersucht. Im Mittelpunkt steht dabei der Prozeß der Übersetzungssteuerung, der bisher mit dem UNIX-spezifischen Werkzeug make realisiert worden ist. Dieses Werkzeug ist aufgrund seiner offenen Lizenz (*GNU Public License*) weit verbreitet und auch unter den Windows-Betriebssystemen verfügbar, allerdings gibt es in der Spezifikation des Übersetzungsprozesses zwischen den Betriebssystemen erhebliche Unterschiede. Deshalb wird in diesem Kapitel ein alternatives Werkzeug für die systemunabhängige Übersetzungssteuerung vorgestellt.

3.1. Softwareentwicklung komplexer Systeme

Ein wichtiges Ziel im Entwicklungsprozeß komplexer Softwaresysteme, dem **Software-Engineering**, ist die Erfüllung der in der Einleitung genannten Softwarequalitätsmerkmale. Dieses Ziel kann über formale Spezifikationen, welche die Funktionalität und die Zuverlässigkeit verbessern, oder über eine geschickte **Modularisierung** der Software, wodurch die Änderbarkeit, die Wiederverwendbarkeit und die Übertragbarkeit erhöht wird, erreicht werden. Ein gutes Beispiel für die Anwendung der Modularisierung bei komplexen Softwaresystemen sind Datenbanksysteme, bei denen dieses Abstraktionsprinzip zu einer *hierarchischen Zerlegung* und damit zu einer Systemschichtung führt (Speicherungsstrukturen, Zugriffspfadstrukturen, Transaktionen etc.), die eine Realisierung solcher Systeme erst möglich macht [Lockemann, Schmidt 87].

Entscheidend für eine erfolgreiche Anwendung des Prinzips der Modularisierung ist es, den richtigen **Grad** der Modularisierung zu finden. Dieser Grad läßt sich anhand einiger Kriterien messen, wie der **Zerlegbarkeit** (in weniger komplexe Teilprobleme), der **Kombinierbarkeit** (Wiederverwendbarkeit standardisierter Teillösungen) oder der **Abschottung** (Begrenzung von Fehlersituationen) [Meyer 88]. Eine bedeutende, auch für das Tycoon-2-System wichtige Konsequenz der Modularisierung ist die Handhabung der Module als *getrennt übersetzbare Einheiten*. Will man ein solches modulares Softwaresystem übersetzen, führt dies zu verschiedenen Problemen für den Softwareentwickler:

- Aufgrund von **Modulabhängigkeiten** muß bei der Übersetzung der verschiedenen Module eine korrekte Reihenfolge eingehalten werden. Dabei ist es besonders bei umfangreichen Programmsystemen wünschenswert, daß nur die tatsächlich nötigen Module neu übersetzt werden.

- Die Identifizierung aller nach einer Änderung neu zu übersetzenden Module und die Einhaltung der entsprechenden Übersetzungsreihenfolge wird dadurch erschwert, daß es meist viele verschiedene Quelltextdateien gibt, die sich über eine nach spezifischen Projektrichtlinien aufgebaute **Verzeichnisstruktur** verteilen.
- Oft müssen im Rahmen des Übersetzungsprozesses zusätzliche betriebssystemspezifische Aktionen durchgeführt werden, wie beispielsweise die Erzeugung wiederverwendbarer, dynamischer Bibliotheken. Manchmal müssen sogar die Quelltextdateien selbst durch Einsatz von **Generatoren** erzeugt werden.

Aufgrund dieser Probleme kann der Softwareentwickler ohne Werkzeugunterstützung kaum noch einen Überblick darüber bewahren, welche Übersetzungsläufe nach bestimmten Modifikationen vorzunehmen sind – der Einsatz zusätzlicher Werkzeuge ist deshalb gerade bei großen Softwaresystemen unumgänglich. In der UNIX-Welt hat sich hierfür make, ein regelbasiertes Werkzeug für die Automatisierung des Übersetzungsprozesses, durchgesetzt.

3.2. Übersetzungssteuerung mit make

Die Entwicklung von make geht auf das Programm UNIX make(1) von [Feldman 78] zurück. Neben einigen kommerziellen Weiterentwicklungen, wie nmake von IBM für OS/2 und Windows, ist heute vor allem GNU make [Stallman, McGrath 93] aufgrund seiner offenen Lizenz[1] über viele Betriebssysteme hinweg weit verbreitet. Diese weite Verbreitung von GNU make würde für den Einsatz dieses Werkzeugs in übertragbaren, betriebssystemübergreifenden Softwaresystemen und damit auch für einen Einsatz im Tycoon-2-System sprechen. Im folgenden wird dieses Werkzeug daher genauer bezüglich seiner Übertragbarkeit auf verschiedene Betriebssysteme hin untersucht.

Die Zieldateien eines Softwaresystems erzeugt make ausgehend von den vorhandenen Quelldateien über eine transitive Kette von Regeln, den **Suffixregeln**. Diese Suffixregeln basieren auf den Namensendungen (*Suffixe*) der beteiligten Dateien: Für jede unterschiedliche Namensendung der Quelldateien muß es eine entsprechende Suffixregel geben, die den für diese Datei notwendigen Übersetzungsschritt spezifiziert. Ob ein Übersetzungsschritt durchgeführt werden muß, entscheidet make anhand des Modifikationsdatums der beteiligten Dateien: Nur im Falle fehlender oder älterer Zieldateien wird ein Übersetzungsschritt für die entsprechende Quelldatei durchgeführt.

Die Spezifikation des Übersetzungsprozesses erfolgt über eine textuelle Notation, durch das sogenannte Makefile. In dieser Steuerdatei muß der Softwareentwickler ausführlich alle Übersetzungsschritte spezifizieren (abgesehen von einigen *automatischen Regeln*); auch die Modulabhängigkeiten eines Softwaresystems müssen an dieser Stelle explizit ausgedrückt werden. Zur Veranschaulichung folgt ein Beispiel für ein Makefile zur Übersetzung des Programms `edit`, basierend auf den Quelltextdateien `main.c`, `kbd.c` und `command.c` sowie den dazugehörigen Schnittstellen (*Header*-Dateien):

[1] GNU make unterliegt der *GNU Public License* (GPL). Damit ist auch für die zukünftige Weiterentwicklung von make eine Offenlegung aller Quelltexte sichergestellt (*Open Source*-Philosophie).

```
# Das Programm 'edit' ist Abhängig von: main.o kbd.o command.o
# Die Objekte werden mit Hilfe von cc gebunden

edit : main.o kbd.o command.o
       cc -o edit main.o kbd.o command.o

# Abhängigkeit der Objekte von Quellen und Header-Dateien.

main.o : main.c defs.h
         cc -c main.c
kbd.o  : kbd.c defs.h command.h
         cc -c kbd.c
command.o : command.c defs.h command.h
         cc -c command.c
```

Schon dieses kleine Beispiel zeigt, daß es einige Mängel in der Übertragbarkeit der Makefiles selbst gibt:

- Die Suffixe der Dateien sind im Makefile fest angegeben, beispielsweise `.o` für Objektdateien. Andere Betriebssysteme benutzen aber zum Teil ganz andere Suffixe; unter Windows enden Objektdateien mit `.obj` und ausführbare Dateien mit `.exe`. Letztere haben unter UNIX gar keinen Suffix.
- Die Anweisungen für verschiedene Aktionen, wie die Aufrufe zum Übersetzen der Programme oder zum Binden der Objektdateien, sind fest angegeben. Diese Anweisungen mit ihren Optionen und Parametern sind jedoch systemspezifisch und unterscheiden sich auf den verschiedenen Betriebssystemen voneinander.

Zwar lassen sich diese einfachen Systemabhängigkeiten mit Hilfe von Variablen und bedingten Anweisungen aus den Makefiles entfernen, bei umfangreicheren Programmsystemen treten jedoch Abhängigkeiten auf, die sich mit solchen Mitteln nicht so einfach entfernen lassen. So sind bei der Analyse der Tycoon-2-Systemkomponenten folgende UNIX-Spezialitäten in den Makefiles auffällig geworden:

- Für die Programmierung von Übersetzungsaktionen steht in make keine eigentliche Programmiersprache zur Verfügung. So muß bei Schleifen in Ermangelung einer Iterationsanweisung auf die UNIX-Shell zurückgegriffen werden:

  ```
  SRC_SUBDIRS = rts tm twl2w3
  all:
      @for i in ${SRC_SUBDIRS} ; do \
          (cd $$i; ... ); \
      done
  ```

- Bei der Verteilung der Programmtexte über eine Verzeichnisstruktur mit verschiedenen Unterverzeichnissen müssen Dateinamen in einer Form angegeben werden, die nicht übertragbar ist, denn die Trennzeichen für Pfade sind auf den verschiedenen Betriebssystemen unterschiedlich (\ unter Windows bzw. / unter UNIX).

- Es müssen in starkem Umfang externe, UNIX-spezifische Zusatzwerkzeuge eingesetzt werden. So wird für die automatische Analyse von Abhängigkeiten zwischen den Quelltexten (kann in der Programmiersprache C durch eine Untersuchung aller `#include`-Anweisungen erfolgen) der GNU C-Übersetzer mit einer speziellen Option aufgerufen. Da beispielsweise Microsoft Visual C++ nicht über vergleichbare Optionen verfügt, ist dieses Verfahren nicht ohne weiteres auf die Windows-Betriebssysteme übertragbar.

- Bei der Erzeugung dynamischer Bibliotheken gibt es erhebliche Unterschiede zwischen den Betriebssystemen, so müssen unter Windows zusätzliche Abhängigkeiten gepflegt werden (für die sogenannten *Modul-Definitionsdateien*). Zwar könnte man in make hierfür unterschiedliche Regeln definieren, die Abhängigkeiten der dynamischen Bibliotheken müßten dann jedoch an unterschiedlichen Stellen in den Makefiles doppelt gepflegt werden.

Diese Systemabhängigkeiten in den Makefiles, wie sie im Tycoon-2-System auftreten, lassen sich nur durch Einsatz weiterer Werkzeuge beheben, die beispielsweise durch einen Zwischenschritt auf Basis systemunabhängiger Übersetzungsspezifikationen Makefiles generieren. Eine wesentlich einfachere, effizientere Alternative ist der make-Ersatz Jam/MR, der unter Umgehung der vorhandenen Makefiles mit einer eigenen, leistungsfähigeren und vor allem übertragbareren Übersetzungsspezifikation arbeitet.

3.3. Übersetzungssteuerung mit Jam/MR

Die Firma Perforce Software entwickelte mit Jam/MR (*Jam – Make(1) Redux*) [Seiwald 94; Perforce 97], einem Werkzeug für die portable Übersetzungssteuerung, ein alternatives Produkt zu make. Dabei unterliegt Jam einer ähnlich freien Lizenz wie make, zusätzlich ist aber auch eine kommerzielle Nutzung explizit erlaubt. Damit hat auch Jam eine weite Verbreitung über die verschiedensten Betriebssysteme gefunden, aktuell gibt es Implementierungen für Windows, Solaris, Linux, HP-UX, OS/2, VMS, Macintosh MPW und auch für BeOS.

Auch bei Jam werden die Spezifikationen zur Übersetzungssteuerung in textueller Form in einer Datei, dem Jamfile, abgelegt. Diese Dateien sind allerdings im Gegensatz zu den Makefiles wesentlich systemunabhängiger, was aufgrund folgender Schlüsseleigenschaften von Jam möglich ist:

- Jam besitzt benannte Regeln, für die mit einer eigenen Programmiersprache Abhängigkeiten und Aktionen definiert werden können. Schleifen oder bedingte Anweisungen gehören ebenso zu dieser Programmiersprache wie die Möglichkeit, lokale Variablen zu definieren.

- Abhängigkeiten zwischen den Quelltextdateien, die in der Programmiersprache C durch gemeinsam genutzte Schnittstellen entstehen können (Quelltexteinschließungen über *Header*-Dateien), werden in Jam über eine Analyse aller `#include`-Anweisungen in den Quelltexten automatisch untersucht – die korrekte Übersetzungsreihenfolge ist so auch ohne Einsatz externer Werkzeuge sichergestellt.[2]

- Es gibt vielfältige Aktionen zum Kopieren von Dateien oder zum Anlegen von Verzeichnissen, so daß nur selten ein Einsatz weiterer Werkzeuge nötig ist. Diese Unabhängigkeit von externen Werkzeugen hat zur Folge, daß Jam sehr effizient arbeitet und trotz seiner Leistungsfähigkeit klein, handlich und unkompliziert in der Benutzung bleibt.

- Im Gegensatz zu make sind die Aktionen zum Übersetzen und Binden von Programmen in Jam für die Verwendung verschiedenster kommerzieller Übersetzer vorbereitet, so z.B. unter Windows für MS Visual C++ oder Borland C++.

- Pfadangaben für Dateien werden in einer betriebssystemunabhängigen Form durch Namenslisten dargestellt, wobei die einzelnen Unterverzeichnisse in diesen Listen durch Leerzeichen voneinander getrennt sind.

- Die jeweiligen systemabhängigen Dateiendungen werden dynamisch ermittelt, was aufgrund benannter Regeln möglich ist. Bei den anonymen Regeln von make müssen die Dateiendungen statisch angegeben werden, um die notwendigen Übersetzungsschritte zu identifizieren.

Als einführendes Beispiel für ein Jamfile soll das bei den Makefiles (Abschnitt 3.2) vorgestellte Programm `edit` dienen. Mit Hilfe von Jam läßt sich die Übersetzungssteuerung für dieses Programm auf nur eine einzelne Codezeile reduzieren:

```
# Das Programm 'edit' ist Abhängig von: main.c kbd.c command.c
Main edit : main.c kbd.c command.c ;
```

Die in diesem Beispiel benutzte Regel `Main` ist für die Erstellung ausführbarer Programmdateien zuständig. Alle systemabhängigen Aktionen zum Übersetzen werden durch diese Regel gekapselt. Auch die Angabe systemabhängiger Dateiendungen ist nicht mehr nötig: Das Programm `edit` erhält auf dem jeweiligen Betriebssystem automatisch den korrekten Suffix für ausführbare Dateien. Die Abhängigkeiten von den *Header*-Dateien werden automatisch erkannt, Objektdateien (z.B. `kbd.o`) werden zwar immer noch genau so wie bei make im Rahmen der Übersetzung benutzt, sie müssen aber nicht mehr explizit im Jamfile angegeben werden.

[2] Über verschiedene Variablen kann dieser Analyseprozeß zwar beeinflußt werden, grundsätzlich ist Jam aber für die Abhängigkeiten zwischen Quelltextdateien auf die Programmiersprachen C bzw. C++ zugeschnitten. Der Einsatz von Jam für andere Programmiersprachen ist nicht ganz unproblematisch, man beachte hierzu die Diskussion um den Einsatz von Jam für die Programmiersprache Java in der *Jamming mailing list* [Perforce 97].

In Jam gibt es eine klare Syntax für die Anwendung von Regeln: Nach dem Namen der Regel folgt optional eine Argumentenliste, wobei mehrere Argumente durch einen Doppelpunkt voneinander getrennt werden. Abgeschlossen wird die Regelanwendung durch ein Semikolon:

```
<RegelName> <Arg1> : <Arg2> : ... : <ArgN> ;
```

Im folgenden wird ein Beispiel für Regeln mit unterschiedlichen Formen von Parameterlisten vorgestellt. Hierbei soll die dynamische Bibliothek[3] `tree` mit Hilfe der beiden Quelltexte `treemake.c` und `treetrav.c` erzeugt werden:

```
LibMember tree : treemake.c treetrav.c ;
BuildLib tree : : libtree.def ;
```

Die Regel `LibMember` legt fest, welche Quelltexte zur dynamischen Bibliothek `tree` gehören und übersetzt diese, falls sie modifiziert wurden (anderenfalls werden die vorhandenen Objektdateien zum Binden der Bibliothek benutzt). Das zweite Argument dieser Regel besteht aus einer Liste von Dateinamen; Jam besitzt vielfältige Anweisungen zur Handhabung und Abarbeitung solcher Listen. Die Regel `BuildLib` erzeugt die dynamische Bibliothek `tree` und erwartet drei Parameter. Der zweite Parameter ist optional und gibt an, von welchen anderen dynamischen Bibliotheken `tree` gegebenenfalls noch abhängt – er ist in diesem Falle leer.

Ein Beispiel für den systemunabhängigen Umgang mit Dateinamen ist im folgenden aufgeführt. Dabei wird zuerst ein neues Verzeichnis angelegt und anschließend die Datei `config.h` auf dieses Verzeichnis kopiert. Die Variable `TOP` enthält den Namen des Wurzelverzeichnisses, auf dem diese ganze Aktion stattfindet (über `$(xx)` wird in Jam die Variable `xx` ausgelesen). Dieses Beispiel zeigt, wie Dateipfade über Namenslisten ohne ein systemabhängiges Trennzeichen (\ bzw. /) spezifiziert werden können:

```
# Systemunabhängiges Kopieren einer Datei

# Neues Verzeichnis auf $(TOP)/ziel/neu anlegen:
MkDir $(TOP) ziel neu ;

# Systemunabhängig Dateinamen für Quelle und Ziel in Variablen ablegen:
makeDirName ALTE_DATEI : $(TOP) quelle config.h ;
makeDirName NEUE_DATEI : $(TOP) ziel neu config.h ;

# Datei kopieren:
File $(NEUE_DATEI) : $(ALTE_DATEI) ;
```

[3] Das Erzeugen dynamischer Bibliotheken wird in Jam erst durch Definition zusätzlicher Regeln möglich, die in einer projekteigenen Jamrules-Datei abgelegt werden können. Beispiele hierfür findet man in der *Jamming mailing list* [Perforce 97].

Bei größeren Programmsystemen wird die Verteilung der Quelltexte über eine komplexe Verzeichnisstruktur in Jam dadurch unterstützt, daß mittels verschiedener Regeln (`SubDir`, `SubInclude`) modularisierte Jamfiles über eine solche Verzeichnisstruktur mitverteilt werden können. Diese modularisierten Jamfiles enthalten nur noch die lokalen Anweisungen für das jeweilige Verzeichnis. Bei einem umfangreichen Softwaresystem können zusätzlich projektspezifische Definitionen und Regeln in einer speziellen Datei (Jamrules) auf dem Hauptverzeichnis des Projekts abgelegt werden.

Zusammenfassend läßt sich sagen, daß sich Jam besser als make für die Übersetzungssteuerung großer, portabler Softwaresysteme eignet.[4] Dies konnte beispielsweise beim Einsatz von Jam im Sybase Open Server der Firma Sybase bestätigt werden [Wingerd, Seiwald 97]. Auch im Rahmen dieser Arbeit wurde make in der Übersetzungssteuerung des Tycoon-2-Systems durch Jam erfolgreich ersetzt, was im folgenden Abschnitt beschrieben wird.

3.4. Eine portable Übersetzungssteuerung für Tycoon-2

Die neue Übersetzungssteuerung für das Tycoon-2-System wurde in der ersten Projektphase eingeführt, wobei zunächst alle Systemabhängigkeiten aus den Quelltexten des Tycoon-2-Systems auskommentiert wurden. Nach einem erfolgreichen Übersetzungsvorgang der virtuellen Maschine von Tycoon-2 auf den Betriebssystemen Solaris, Windows NT, Windows 95 und Windows 98 wurde diese Projektphase abgeschlossen.

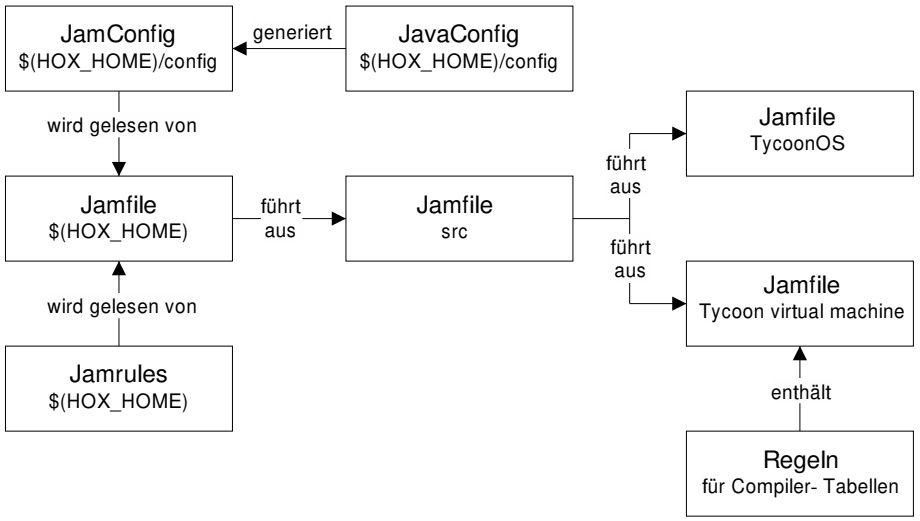


Abbildung 3.1.: Übersetzungssteuerung in Tycoon-2 mit Hilfe von Jam

[4] Stuart Feldman, der Autor von UNIX make(1), wies selbst darauf hin, daß sich make eher schlecht für große Projekte eignet [Feldman 78]. Nachfolgeprodukte wie GNU make oder nmake versuchten diesen Mangel über verschiedene Erweiterungen zu beheben, wie *Wildcarding* bei den Regeln oder den Einsatz von *Präprozessoren*.

In Abbildung 3.1 sind die Zusammenhänge der neuen Übersetzungssteuerung für das Tycoon-2-System dargestellt. Dabei werden zwei Besonderheiten deutlich: Zum einem werden verschiedene Konfigurationsdaten von einem Java-Programm in einer gesonderten Datei abgelegt (JamConfig), was im Abschnitt 3.5 kurz erläutert wird. Zum anderem werden zusätzliche Regeln für die sogenannten *Compiler-Tabellen* im Jamfile für die virtuelle Maschine benötigt.

Diese Tabellen bilden die Schnittstelle zwischen der in C geschriebenen virtuellen Maschine und den in TL-2 realisierten Übersetzer. Sie enthalten Informationen über eingebaute Klassen, Bezeichner und den Instruktionen des Tycoon-2 Bytecodes. Änderungen an dieser Schnittstelle sind immer schwierig durchzuführen, da sie grundsätzlich einen *Bootstrap* zur Folge haben [Wienberg 97]:

1. Die für die neue Schnittstelle nötigen Änderungen müssen auf Ebene der virtuellen Maschine in den entsprechenden *Header*-Dateien eingepflegt werden.

2. Für diese neuen Schnittstellen müssen mit Hilfe von Generatoren neue Tabellen erzeugt werden. Dies sind C-Programme, die mit Hilfe der veränderten *Header*-Dateien die neuen Tabellen erzeugen.

3. Die virtuelle Maschine und der Übersetzer, beide basierend auf der alten Schnittstelle, müssen mit Hilfe der veränderten Tabellen einen neuen Übersetzer erzeugen, der die neue Schnittstelle versteht.

4. Erst jetzt darf die virtuelle Maschine mit den veränderten Schnittstellen übersetzt werden.

5. Nun sollte die neue virtuelle Maschine über die veränderte Schnittstelle mit dem neuen Übersetzer zusammenarbeiten können.

Fehlersituationen sind in diesem Prozeß oft nicht einfach zu handhaben, da man sich verschiedene Versionen der virtuellen Maschine, der Tabellen und des Übersetzers aufbewahren muß. Während vor allem Schritt 3 schwierig und deshalb besser manuell durchzuführen ist, kann Schritt 2 ohne Probleme mit Hilfe von Jam automatisiert werden. Die Abhängigkeiten, die hierbei von Jam beachtet werden müssen, sind in Abbildung 3.2 am Beispiel der Compiler-Tabelle für die Instruktionscodes (*Opcodes*) dargestellt:

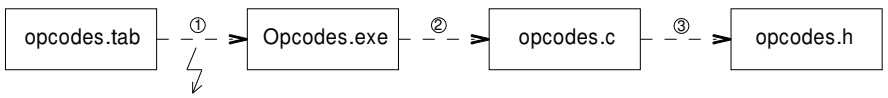


Abbildung 3.2.: Abhängigkeiten der Compiler-Tabelle für die Instruktionscodes

Der hier dargestellte Generator `opcodes.exe` generiert basierend auf seinem Quelltext `opcodes.c` und seiner statisch eingelesenen Schnittstelle `opcodes.h` die Compiler-Tabelle

opcodes.tab. Da die Generatoren ihre Schnittstellendefinitionen statisch einlesen, müssen diese bei Änderungen in der Schnittstelle neu übersetzt werden. Während dies automatisch bei der Übersetzung der virtuellen Maschine von Tycoon-2 geschehen kann, werden die Compiler-Tabellen nur bei Bedarf (mittels `jam Boot`) neu generiert, um einem versehentlichen Überschreiben der Tabellen vorzubeugen. Die in Abbildung 3.2 dargestellten Abhängigkeiten ② und ③ basieren auf den Übersetzungsregeln von C und werden von Jam schon korrekt behandelt, lediglich die Abhängigkeit ① der Compiler-Tabelle von der ausführbaren Datei des Generators ist Jam unbekannt. Wie eine solche neue Abhängigkeit in Jam realisiert werden kann, soll abschließend in diesem Abschnitt dargestellt werden.

Vor der Einführung neuer Regeln muß zunächst das neue, mit `jam Boot` zu generierende Ziel bekanntgemacht werden. Dabei handelt es sich nicht um eine zu erzeugende Datei mit dem Namen `Boot`, sondern vielmehr um ein virtuelles Ziel (*Pseudo-Target*) zur Generierung der benötigten Compiler-Tabellen.

```
# Aktuelles Verzeichnis
SubDir TOP src tm ;

# Systemunabhängiger Dateiname der Tabelle in die Variable
# OPCODES_TAB schreiben, hier: $(TOP)/src/tm/opcodes.tab
makeDirName OPCODES_TAB  : opcodes.tab ;
makeDirName BUILTINS_TAB : builtins.tab ;
...

# Boot ist keine Datei
NOTFILE Boot ;

# Einfache Regel für die Ziele von 'Boot'
BuildTabFiles Boot : $(OPCODES_TAB) $(BUILTINS_TAB) ... ;

# Regel für die Generierung der Tabelle mit einem Generator
BuildTabFile $(OPCODES_TAB)  : $(OPCODES)$(SUFEXE) ;
BuildTabFile $(BUILTINS_TAB) : $(BUILTINS)$(SUFEXE) ;
...
```

Wird Jam ohne die Angabe weiterer Parameter aufgerufen, wird immer das virtuelle Ziel `all` erzeugt. Abhängig hiervon sind alle ausführbaren Dateien, alle Bibliotheken und auch alle zu kopierenden Dateien, wie z.B. bei der Regel `Files` im Beispiel weiter oben. Das Ziel `Boot` ist dagegen nicht abhängig von `all`, so daß die noch zu definierenden Regeln `BuildTabFiles` und `BuildTabFile` nur dann ausgeführt werden, wenn Jam explizit für das Ziel `Boot` aufgerufen wurde.

Eine Regel besteht in Jam aus zwei Komponenten, der eigentlichen **Regel**, die bei einem Aufruf vollständig ausgeführt wird, und den **Aktionen**, die nur dann ausgeführt werden,

wenn das Ziel neu erzeugt werden muß. Es kann auch Regeln ohne Aktionen geben, diese definieren dann lediglich neue Abhängigkeiten für virtuelle Ziele. Regeldefinitionen werden durch das Schlüsselwort `rule`, Aktionen durch das Schlüsselwort `action` eingeleitet:

```
rule <RegelName> { <Anweisungen> }
actions <AktionsName> { <Anweisungen> }
```

Für die `Boot`-Regeln des Tycoon-2-Systems werden zwei Regeln benötigt: Zunächst eine Regel, die die Abhängigkeit zwischen dem virtuellen Ziel `Boot` und den Compiler-Tabellen definiert und schließlich als zweites eine Regel, die die Abhängigkeit der jeweiligen Tabelle vom entsprechenden Generatorprogramm definiert:

1. Die Regel `BuildTabFiles` ist eine einfache Regel ohne weitere Aktionen, sie definiert über die eingebaute Regel `DEPENDS` eine neue Abhängigkeit für das als Argument der Regel angegebene Ziel `$(<)` (in diesem Beispiel ist dies das virtuelle Ziel `Boot`), zu der ebenfalls als Argument angegebenen Liste von Quellen `$(>)` (hier sind dies die Compiler-Tabellen):

   ```
   rule BuildTabFiles {
      DEPENDS $(<) : $(>) ;
   }
   ```

2. Die Regel `BuildTabFile` ist für die Erzeugung der Compiler-Tabellen zuständig. Zunächst werden in der Regel selbst einige Abhängigkeiten definiert:

   ```
   rule BuildTabFile {
     # Generator-Programm aus Quelltext erzeugen (falls nötig)
     Main $(>) : $(>:B).c ;

     # Abhängigkeiten der Tabellen vom Generator
     DEPENDS $(<) : $(>) ;

     # Abhängigkeiten der Tabellen von der Schnittstelle
     # $(x:B).h ersetzt in der Varible x die Dateiendung mit ".h"
     DEPENDS $(<) : $(>:B).h ;

     # Aktion für die Generierung:
     BuildTabFile $(<) : $(>) ;
   }
   ```

   Schließlich wird zu dieser Regel noch eine Aktion definiert, um gegebenenfalls eine neue Compiler-Tabelle zu generieren. Man beachte, daß die Variable `$(>)` den Namen des ausführbaren Generators enthält.

```
actions BuildTabFile {
  if $(UNIX) || $(NT) {
    # Zum Verzeichnis für die Generierung der Tabellen gehen:
    $(PUSHD) $(TABDIR)
    # Generator aufrufen, über Standardausgabe Tabelle erzeugen:
    $(>) >$(<)
    # Wieder auf das alte Verzeichnis zurückgehen:
    $(POPD)
  }

  else if $(WIN95) || $(WIN98) {
    # Hier gibt es kein pushd / popd
    # Eine andere Implementierung ist deshalb nötig...
    ...
  }
}
```

Für ein vertiefendes Verständnis für die Erstellung von Jamfiles oder der Definition neuer Regeln wird auf die Online-Dokumentation von Jam verwiesen [Perforce 97]. Insgesamt konnte Jam die an eine übertragbare Übersetzungssteuerung gestellten Erwartungen voll erfüllen. Abschließend beschäftigt sich der nächste Abschnitt mit der in Abbildung 3.1 angedeuteten Konfiguration der Übersetzungssteuerung.

3.5. Konfiguration

Vor dem Aufruf der Übersetzungssteuerung in Tycoon-2 mit Hilfe von make oder Jam ist eine Konfiguration dieser Steuerung notwendig. Neben der Einstellung verschiedener Umgebungsvariablen für den Betrieb des Tycoon-2-Systems zählen hierzu vor allem die Festlegung bestimmter Übersetzungsparameter für die Optimierung, Fehlerkontrolle (*debugging*) und Ablaufverfolgung (*tracing*) der virtuellen Maschine. Im Verlauf der Realisierung der im nächsten Kapitel vorgestellten portablen Systembibliothek TycoonOS kommen hier noch diverse zusätzliche Optionen hinzu, z.B. für die Ablaufkontrolle verschiedener Systemdienste (Netzwerkkommunikation, dynamische Bindungen etc.).

Bisher wurde diese Konfiguration mit Hilfe von Skript-Dateien der UNIX-Shell durchgeführt, die nicht auf die Windows-Betriebssysteme übertragbar sind. Um hier eine bessere Übertragbarkeit zu erreichen, fanden im Rahmen dieser Arbeit folgende Änderungen in der Konfiguration des Tycoon-2-Systems statt:

- Nur noch die wichtigsten Umgebungsvariablen werden in den Skript-Dateien definiert. Die Funktionalität dieser Skript-Dateien wurde auf die Windows-Betriebssysteme in Form entsprechend angepaßter *Batch-Dateien* übertragen. Ein großes Problem war dabei der absolut mangelhafte Funktionsumfang, der für solche Batch-Dateien auf

diesen Betriebssystemen zur Verfügung steht. So mußte letztendlich ein Umweg über ein C-Programm gewählt werden, daß die aktuellen Werte für die Umgebungsvariablen ermittelt und damit eine entsprechend aufbereitete Batch-Datei generiert.

- Die Konfiguration aller weiteren Parameter für die Übersetzungssteuerung und der virtuellen Maschine geschieht in Zukunft über ein Java-Programm, welches über einen Dialog die komfortable Eingabe aller Übersetzungsparameter gestattet. Dieses Programm generiert eine JamConfig-Datei (siehe Abbildung 3.1 weiter oben) für alle Jam-Parameter und eine `config.h`-Datei für alle Parameter auf Ebene der Programmiersprache C, also für TycoonOS und für die virtuelle Maschine von Tycoon-2. Von diesen generierten Dateien sind entsprechende Prototypen verfügbar, so daß dieser Konfigurationsschritt auch manuell durchgeführt werden kann, falls Java auf der entsprechenden Zielplattform nicht zur Verfügung steht. Abbildung 3.3 zeigt diesen Konfigurationsdialog in Aktion.

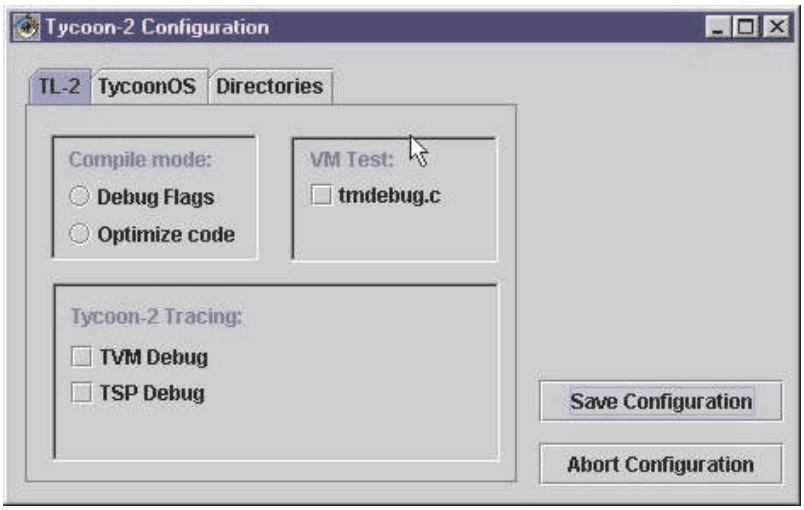


Abbildung 3.3.: Konfiguration von Tycoon-2 mit einem Java-Dialog

In diesem Kapitel wurde ausführlich eine portable Übersetzungssteuerung vorgestellt und deren Einführung in das Tycoon-2-System beschrieben; damit sind die in C realisierten Komponenten des Systems auf den Windows-Betriebssystemen *übersetzungsfähig*. Im nächsten Schritt konnte dann das in Kapitel 2 beschriebene Architekturkonzept für eine portable virtuelle Maschine auf das Tycoon-2-System übertragen werden. An zentraler Stelle steht dabei die Realisierung der Dienste für die portable Systembibliothek TycoonOS, was im folgenden Kapitel ausführlich beschrieben wird.

4. TycoonOS – Eine portable Schnittstelle zum Betriebssystem

In diesem Kapitel werden Entwurf, Realisierung und Schnittstellen der portablen Systembibliothek TycoonOS vorgestellt. Diese Bibliothek soll im Rahmen des in Kapitel 2 entwickelten portablen Architekturkonzeptes das bisher bestehende Laufzeitsystem RTS (*RunTime System*) für die Hochsprache TL-2 ersetzen und zusätzlich alle Betriebssystemdienste kapseln, die für die virtuelle Maschine und das Objektspeichersystem von Tycoon-2 benötigt werden.

Für ein besseres Verständnis der in diesem Kapitel vorgestellten Systemdienste ist eine kurze Einführung in die Betriebssystemarchitektur der neuen Zielplattform Windows NT notwendig. Dabei werden auch einige Probleme im Zusammenhang mit der Benutzung der Win32 API deutlich. Nach einer Betrachtung der verschiedenen Implementierungsaspekte für TycoonOS werden schließlich die verschiedenen Systemdienste dieser Bibliothek ausführlich beschrieben.

4.1. Grundbegriffe

Zunächst erfolgt an dieser Stelle eine kurze Erläuterung der in dieser Arbeit verwendeten Grundbegriffe. Für eine detaillierte Erklärung dieser Begriffe und eine Vertiefung der theoretischen Grundlagen dieses Kapitels wird auf [Tanenbaum 92] verwiesen.

Prozeß: Unter einem Prozeß versteht man die geladene und gestartete Instanz eines Programms einschließlich der dazu benötigten Systemressourcen (aktueller Wert des Programmzählers, Register, Variablen etc.).

Multitasking: Ein Betriebssystem mit der Fähigkeit zum Multitasking erlaubt die pseudoparallele Ausführung einer Menge von Prozessen und damit einen **Mehrprogrammbetrieb**.

Preemptiv: Unter einem *echten* Multitasking versteht man das sogenannte **preemptive** (*verdrängende*) **Multitasking**, wie es beispielsweise bei Windows NT, UNIX oder OS/2 zu finden ist. Dabei hat das Betriebssystem selbst die Kontrolle über alle Systemressourcen und ist für die Zuteilung der Rechenzeiten an die einzelnen Prozesse zuständig. Beim **kooperativen Multitasking**, welches beispielsweise noch bei Windows 3.1 zu finden ist, bestimmen dagegen die einzelnen Prozesse selbst, wie viele Systemressourcen

benutzt werden können. Die Verteilung der Rechenzeiten liegt hier in der Verantwortung der einzelnen Prozesse – ein fehlerhaftes Programm kann damit zum Stillstand des gesamten Systems führen, falls es die Kontrolle über seinen Prozessor nicht wieder abgibt. Ein solches *Hängen* des Systems kann bei Betriebssystemen mit preemptiven Multitasking vermieden werden.

Kontextwechsel: Da bei einem Mehrprogrammbetrieb die Anzahl der Prozesse die Anzahl der zur Verfügung stehenden Prozessoren in der Regel überschreitet, müssen sich mehrere Prozesse denselben Prozessor teilen. Bei einem **Kontextwechsel** (*Scheduling*) wird die Kontrolle eines Prozesses über einen Prozessor an einen anderen Prozeß übergeben. Ein solcher Kontextwechsel verursacht Kosten, da der aktuelle Zustand eines Prozesses, also sein Programmzähler, Registersatz, Kellerspeicher etc. gesichert und später wiederhergestellt werden muß. Ein wichtiges Ziel verschiedener **Strategien** für den Kontextwechsel (den **Scheduling-Strategien**) ist es, diese Kosten zu minimieren, wobei ein Kompromiß zwischen der Anzahl der nötigen Kontextwechsel und der möglichst schnellen Antwortzeit der Prozesse erreicht werden sollte.

Ablaufinvarianz: Eine Konsequenz aus dem Kontextwechsel beim Mehrprogrammbetrieb ist, daß alle Systemaufrufe und Laufzeitbibliotheken des Betriebssystems einen voll **ablaufinvarianten Objektcode** (*reentrant-code*) anbieten müssen. Darunter versteht man Programmcode, der von verschiedenen Prozessen nebenläufig gestartet und abgearbeitet werden kann. Würde eine Funktion mit nicht-ablaufinvariantem Programmcode aufgerufen werden, müßte diese Funktion vollständig und ohne Unterbrechung abgearbeitet werden, was gerade bei einem preemptiven Multitasking nicht sichergestellt werden kann. Der Betriebssystemcode einiger älterer Betriebssysteme, wie beispielsweise von MS-DOS, ist nicht ablaufinvariant, daher durften Hintergrundprozesse (*residente Programme*) deren Programmcode nicht benutzen.

Thread: Unter einem Thread versteht man eine unabhängige Ausführungsinstanz in einem Prozeß. Ein Thread besitzt einen eigenen Kellerspeicher und einen eigenen Registersatz, alle anderen Ressourcen des Prozesses (Speicher, geöffnete Dateien etc.) werden mit den anderen Threads in dem Prozeß geteilt. Aufgrund dieses geringeren Umfangs der Verwaltungsdaten in einem Thread werden diese häufig auch als **leichtgewichtige Prozesse** (*lightweight processes*) bezeichnet – auch ein Kontextwechsel ist bei Threads mit geringeren Kosten verbunden.

Multiprozessorsystem: Ein Multiprozessorsystem ist ein Betriebssystem, welches die Verteilung der Prozesse und Threads über mehrere Prozessoren zuläßt. In einem solchen System ist eine echte Parallelverarbeitung möglich; die durch die Kontextwechsel verursachten Kosten können bei einer hohen Anzahl von Prozessoren vermindert werden. Können die Benutzer- und Betriebssystemprozesse beliebig über alle im System vorhandenen Prozessoren verteilt werden, spricht man von einer **symmetrischen Multiprozessorverarbeitung** (*Symmetric MultiProcessing*, SMP). Eine wichtige Eigenschaft solcher Systeme ist die gemeinsame Nutzung des Hauptspeichers durch alle Prozesso-

ren, womit auch eine Verteilung von Threads desselben Prozesses (teilen sich denselben Speicher) über verschiedene Prozessoren möglich wird.

Synchronisation: Wenn mehrere Prozesse auf gemeinsame Ressourcen des Betriebssystems (z.B. Dateien oder Geräte) zugreifen bzw. wenn mehrere Threads auf gemeinsame Ressourcen des Prozesses (z.B. gemeinsamen Speicher) zugreifen, ist der Einsatz von **Synchronisationsmaßnahmen** notwendig. Erst Mechanismen wie der **gegenseitige Ausschluß**, **Semaphoren** oder **Monitore** erlauben die Lösung klassischer Kommunikationsprobleme für Prozesse und Threads, wie dem Philosophenproblem oder dem Leser-Schreiber-Problem [Hoare 74; Tanenbaum 92; Jessen, Valk 87].

Die für das Tycoon-2-System wichtigen Synchronisationsmaßnahmen werden noch in Abschnitt 4.6.4 vorgestellt. Im nächsten Abschnitt erfolgt eine kurze Einführung in die Betriebssystemarchitektur von Windows NT, der neuen Zielplattform für das Tycoon-2-System.

4.2. Architektur von Windows NT

In Abbildung 4.1 werden die für diese Arbeit relevanten Bestandteile des Betriebssystems Windows NT dargestellt. Eine wichtige Informationsquelle zu der hier vorgestellten grundlegenden Architektur von Windows NT ist das von Microsoft herausgegebene Resource Kit für Windows NT Workstation 4.0 [Microsoft 96b] bzw. Windows NT Server 4.0 [Microsoft 96a].

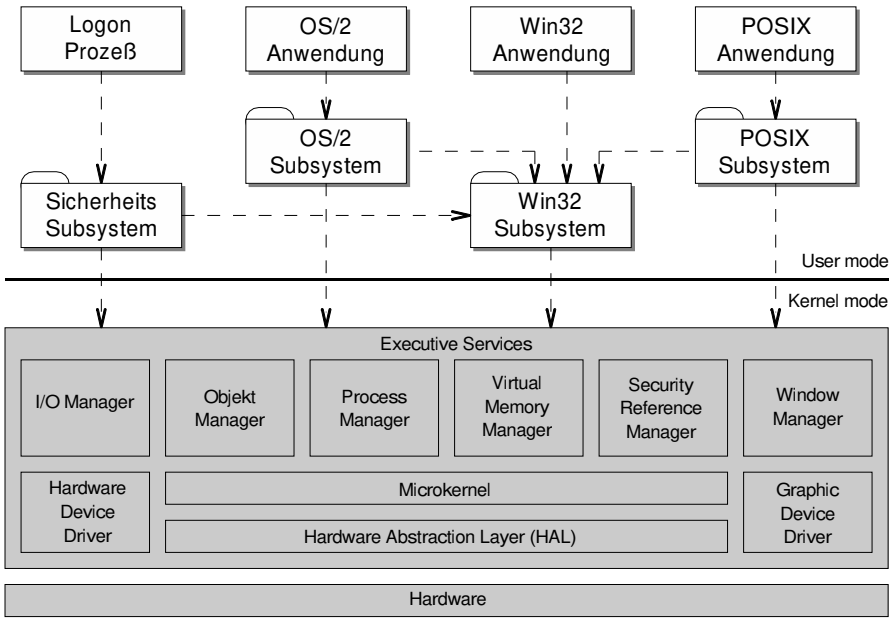


Abbildung 4.1.: Die modulare Architektur von Windows NT

Da an dieser Stelle nur ein grundlegender Überblick über die Konzepte dieser Betriebssystemarchitektur möglich ist, muß für eine Vertiefung auf diese Quellen verwiesen werden. Neben der stark modularen Architektur des Betriebssystems fällt in Abbildung 4.1 auch die strikte Trennung zwischen dem Benutzer- und dem Kernmodus auf:

- Der **Kernmodus** (*Kernel mode*) ist ein hochgradig privilegierter Ausführungsmodus, in dem Programme einen Zugriff auf alle Hardware- und Speicherressourcen des Rechners haben, wobei auch ein Zugriff auf die Adreßräume aller Benutzerprozesse möglich ist. Der im Kernmodus laufende Teil des Betriebssystems wird auch als *Windows NT Executive* bezeichnet. Dieser Teil des Systems umfaßt folgende Komponenten:
  - Den **Hardware abstraction layer** (HAL), eine Bibliothek, die auf der untersten Ebene der Windows NT Executive die direkten Zugriffe auf die Hardware des Rechners kapselt. Daneben greifen nur noch spezielle Gerätetreiber direkt auf die Hardware zu.
  - Den **Microkernel**, der das Herz des Betriebssystems bildet und für den multiprozessorfähigen Kontextwechsel (*Scheduler*) und das Management für Unterbrechungen (*Interrupts*) und Ausnahmebehandlungen (*Exceptions*) zuständig ist. Über diesen vollständig ablaufinvariant realisierten Betriebssystemkern ist eine symmetrische Multiprozessorverarbeitung (SMP) möglich, so daß die Threads aller Prozesse einschließlich des Betriebssystems beliebig über alle vorhandenen Prozessoren verteilt werden können.
  - Verschiedene **Service-Module** für die Verwaltung der Prozesse, des virtuellen Speichers, der verschiedenen Betriebssystemobjekte und der Sicherheitsmechanismen von Windows NT.
- Der **Benutzermodus** (*User mode*) ist ein weniger privilegierter Verarbeitungsmodus, in dem ein direkter Zugriff auf die Hardware des Rechners nicht mehr möglich ist. In diesem Ausführungsmodus können Programme nur noch in ihrem eigenen Adreßraum operieren, alle Systemdienste müssen über wohldefinierte Schnittstellen (*Application Programming Interface*, API) abgewickelt werden. Diese Schnittstellen werden von den im Benutzermodus laufenden **Umgebungs-Subsystemen** des Betriebssystems angeboten, dabei ist jeder Anwendungsprozeß auf die Benutzung eines Subsystems statisch festgelegt – schon bei der Übersetzung des Anwendungsprogramms muß festgelegt werden, in welchem Subsystem das Programm ausgeführt werden kann. Zu den wichtigsten Subsystemen gehören unter anderem:
  - Das **POSIX-Subsystem** (POSIX = *Portable Operating System Interface*), welches, wie schon in der Einleitung dieser Arbeit angedeutet, zum POSIX Standard Level 1 kompatibel ist.[1] Die unter dem Synonym Pthreads bekannte Implementierung für Threads würde dagegen POSIX Level 4 erfordern.[2] Beide IEEE/ANSI-Standards sind seit 1996 auch ISO-Standard [ISO/IEC 96].

[1] IEEE Std 1003.1, POSIX.1: *System application program interface (API) [C language].*
[2] IEEE Std 1003.1c, POSIX.4: *Real-time and threads.*

Von der fehlenden Implementierung der Pthreads abgesehen ist das POSIX-Subsystem recht umfangreich, es bietet etwa 110 Funktionen der von UNIX bekannten C-Standardbibliothek an. Im Windows NT Resource Kit [Microsoft 96b] findet man zahlreiche UNIX-Kommandozeilenwerkzeuge (wie `ls`, `grep` oder `diff`), die mit Hilfe dieses Subsystems auf Windows NT übertragen wurden. Ein großer Nachteil ist allerdings, daß dieses Subsystem in sich *geschlossen* ist, d.h. es gibt für Anwendungen dieses Subsystems keine Möglichkeit, Funktionen aus der Win32 API zu benutzen.

- Das **OS/2-Subsystem**, welches die Ausführung älterer OS/2 Programme (textbasierte 16-Bit Anwendungen) ermöglicht. Im Gegensatz zum POSIX-Subsystem erlaubt das OS/2-Subsystem eine Kommunikation mit dem Win32-Subsystem über den sogenannten *Win32 Thunking* Mechanismus[3] [Microsoft 96b].
- Das **Win32-Subsystem** mit der dazugehörigen Schnittstelle, der Win32 API [Iseminger 99; Hamilton, Williams 96], ist das für Windows NT wichtigste Subsystem, es ermöglicht für alle Prozesse des Benutzermodus den zentralen Zugriff auf das Betriebssystem. Dadurch, daß diese zentrale Schnittstelle auch in den Betriebssystemen Windows 95 und Windows 98 vorhanden ist, kann eine weitgehende dynamische Portabilität der Programme und Bibliotheken über die hier betrachteten Windows-Betriebssysteme erreicht werden.

Innerhalb der "Windows-Welt" bildet das Win32-Subsystem quasi eine portable Systembibliothek (Abschnitt 2.1.1). Dabei gibt es in den verschiedenen Implementierungen der Win32 API allerdings einige Unterschiede, was in Abbildung 4.2 angedeutet wird:

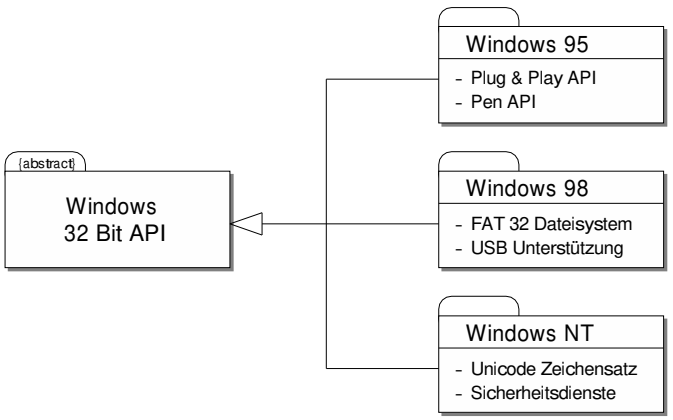


Abbildung 4.2.: Unterschiede in den Implementierungen der Win32 API

[3] Dabei findet über eine Zwischenbibliothek (*Win32 DLL thunking layer*) lediglich eine Abbildung zwischen 16-Bit- und 32-Bit-Aufrufen statt. Innerhalb des OS/2-Subsystems sind verschiedene Systemfunktionen verfügbar, um Funktionen aus einer solchen *Win32 thunking DLL* aufzurufen.

Funktionen für bestimmte Dienste sind dabei nur auf einigen Plattformen verfügbar, beispielsweise gibt es eine USB- oder FAT32-Unterstützung nur auf Windows 98. Für das Tycoon-2-System ist in diesem Zusammenhang zu beachten, daß eine Verwendung der in Abschnitt 4.8.4 beschriebenen Sicherheitsdienste nur unter Windows NT möglich ist.

Ein weiterer Unterschied in den verschiedenen Implementierungen der Win32 API besteht darin, daß spezielle Funktionen zwar auf allen Systemen verfügbar sind, allerdings eine unterschiedliche Anzahl von Parametern aufweisen oder zum Teil sogar eine unterschiedliche Semantik besitzen [Nyman 96]. In dieser Arbeit wird sich für die Realisierung von TycoonOS maßgeblich an der Win32 API von Windows NT orientiert, auf einzelne Unverträglichkeiten zu Windows 95 bzw. Windows 98 wird in diesem Kapitel explizit hingewiesen.

4.3. Implementierungsaspekte

Nach dieser einleitenden Betrachtung der grundlegenden Architektur des Betriebssystems Windows NT können jetzt verschiedene Implementierungsaspekte für die portable Systembibliothek TycoonOS untersucht werden. Dabei werden zunächst verschiedene Realisierungsmöglichkeiten bewertet, wobei einige schon existierende portable Systembibliotheken vorgestellt werden, die Portierungsprojekte von UNIX nach Windows NT unterstützen sollen. Schließlich werden einige Voraussetzungen betrachtet, die TycoonOS an sein Wirtsbetriebssystem stellt.

4.3.1. Portable Systembibliotheken für UNIX und Windows NT

Die **Migration** von Softwaresystemen über verschiedene Betriebssysteme ist ein aktuelles und wichtiges Thema in der Software-Technik. Neben der Übertragung des Tycoon-2-Systems von UNIX auf Windows NT im Rahmen dieser Arbeit gibt es eine Reihe weiterer Projekte, die sich mit allgemeineren Lösungen für die Übertragung von UNIX-Anwendungen auf die Windows-Betriebssysteme befassen [Korn 97; AT&T 99; Cygnus 99a].

Diese Lösungen für die Portierung von Software werden in diesem Abschnitt in der Übersicht vorgestellt, einschließlich einer Bewertung des allgemein verfügbaren POSIX-Subsystems von Windows NT. Von einem Einsatz der hier beschriebenen portablen Systembibliotheken für die Übertragung des Tycoon-2-Systems wurde jedoch zu Gunsten einer eigenen Implementierung von TycoonOS abgesehen, was vor allem an einer mangelnden Implementierung der Pthreads in diesen Bibliotheken und einer möglichen Abhängigkeit vom Bibliotheksheresteller lag (Quelltexte bzw. Lizenzen, siehe Abschnitt 2.1.1). Besonders der letzte Punkt ist im Rahmen einer eventuell kommerziellen Weiterentwicklung von Tycoon-2 zu beachten.

POSIX-Subsystem von Windows NT: Eine Nutzung dieses Subsystems für die Realisierung von TycoonOS wäre ideal. Die optimale Verfügbarkeit als fester Bestandteil des Betriebssystems Windows NT und die Implementierung zahlreicher UNIX-typischer Datei- und Verzeichnisfunktionen sprechen für dieses Subsystem. Weiterhin gibt es eine vollständige Umsetzung der Dateinamenssemantik von UNIX (Trennzeichen für Pfade, Groß- bzw. Kleinschreibung) auf das NTFS-Dateisystem von Windows NT. Der schon

weiter oben angesprochene Mangel an einer Implementierung der POSIX-Threads und vor allem die Abgeschlossenheit gegenüber dem Win32-Subsystem machen allerdings einen Einsatz dieses Subsystems für die Realisierung von TycoonOS unmöglich.

U/WIN: Das U/WIN-Paket von David G. Korn [Korn 97; AT&T 99] (dem Entwickler der Korn Shell) hat sich zum Ziel gesetzt, UNIX-Anwendungen auf allen Windows-Systemen möglichst unverändert übersetzen und ausführen zu können. Dazu werden in dieser Bibliothek vor allem auch semantische Aspekte berücksichtigt, so werden beispielsweise spezielle Gerätenamen umgesetzt (`/dev/fd0` für Diskettenzugriffe) oder spezielle Dateinamen interpretiert (`/dev/null` bzw. `:NUL`).

Leider gibt es in dieser Bibliothek noch keine Funktionalität für Threads bzw. Pthreads, auch ist der Objektcode dieser Bibliothek nicht ablaufinvariant (*threadsafe*) realisiert, weshalb eine Verwendung des U/WIN-Pakets für TycoonOS ausgeschlossen ist. Ein weiterer Nachteil ist die lizenzrechtliche Abhängigkeit von AT&T, da die Quelltexte dieser Bibliothek nicht frei verfügbar sind und kommerzielle Lizenzen kostenpflichtig erworben werden müssen.

Cygnus GNU Tools: Die Cygnus GNU Tools [Cygnus 99a] sind im Gegensatz zu dem U/WIN-Paket einschließlich aller Quelltexte frei verfügbar und machen derzeit eine lebhafte Entwicklung durch (aktuell ist gerade die *Beta*-Version 20.1). Diese Bibliothek konzentriert sich auf eine möglichst weitgehende Abbildung der verschiedenen UNIX-Schnittstellen (API's) auf das Win32-Subsystem, einschließlich der Schnittstellen zum X Windows-System (die graphische Benutzeroberfläche unter UNIX). Mit Hilfe der Cygnus GNU Tools konnte beispielsweise der unter UNIX weit verbreitete Editor XEmacs nach Windows NT portiert werden.

Leider bietet auch diese Systembibliothek keine Funktionalität für Threads an und ist ähnlich wie U/WIN derzeit nicht ablaufinvariant realisiert. Damit kommt auch diese Bibliothek für die Realisierung von TycoonOS nicht in Frage. Von Interesse ist dagegen ein bei Cygnus parallel laufendes Projekt: $\text{Pthreads}_{\text{Win32}}$, eine im Quelltext frei verfügbare Bibliothek speziell zur Realisierung der POSIX-Threads unter Windows [Cygnus 99b]. Dieses Projekt befand sich allerdings bei Beginn dieser Arbeit in einem frühen experimentellen Entwicklungsstadium, außerdem werden hier keine anderen der von TycoonOS benötigten Dienste in einer ablaufinvarianten Form angeboten (Dateisystem, dynamisches Binden etc.).

Netscape Portable Runtime: Diese schon in Kapitel 2 vorgestellte portable Systembibliothek ist eine umfangreiche Bibliothek für betriebssystemnahe Dienste, wie für Threads, Threadsynchronisation, Dateien oder grundlegende Netzwerkfunktionen. Das portable Threadmodell dieser Bibliothek basiert auf dem Konzept der **Monitore** [Hoare 74] und weist dabei einige kleinere Unterschiede zu dem Threadmodell von POSIX auf.

Da die NSPR erst nach Beginn dieser Arbeit veröffentlicht wurde, stand diese Bibliothek für die Realisierung von TycoonOS zunächst nicht zur Diskussion. Neben kleineren Unterschieden zu dem Threadmodell von POSIX gibt es auch einige Mängel in der

Threadsynchronisation (*Lost wakeup*-Problem, siehe Abschnitt 4.6.4), die im Rahmen dieser Arbeit für Windows NT besser gelöst werden konnten.

Neben den hier betrachteten Bibliotheken gibt es noch einige kommerziell verfügbare Systembibliotheken für die Portierung von UNIX-Software auf die Windows-Betriebssysteme, wie z.B. NuTCracker von DataFocus [DataFocus 99], die hier aber nicht weiter betrachtet werden. Auch portable C++-Klassenbibliotheken wie ACE (*The ADAPTIVE Communication Environment* [ACE 99]) oder die in Abschnitt 2.1.1 genannten StarView und IBM Open Class Library sind hier nicht weiter von Interesse, da für die virtuelle Maschine von Tycoon-2 und damit für TycoonOS eine auf die Programmiersprache C basierende Lösung angestrebt wird. Aus diesem Grund wird auch für die Windows-Implementierung von TycoonOS auf die Verwendung der MFC (*Microsoft Foundation Classes*) verzichtet.

Gerade das oben aufgeführte Beispiel der NSPR zeigt die rasante Entwicklung auf dem Gebiet der portablen Systembibliotheken und damit die Relevanz des Themas **Migration** von Softwaresystemen. Ein interessanter Aspekt in diesem Zusammenhang ist, daß Microsoft gerade im September 1999 die Firma Softway Systems [Softway 99] gekauft hat. Diese Firma war für die Entwicklung der Bibliothek Interix bekannt, die unter Windows NT durch eine Erweiterung des POSIX-Subsystems die Übertragung von UNIX-Software erleichtern sollte. Dabei soll sogar eine Öffnung des POSIX-Subsystems gegenüber der Win32 API angestrebt worden sein. Bleibt abzuwarten, welche Konsequenzen dieser Einkauf bei Microsoft für die Entwicklung von Windows 2000 hat.

Nach dieser Vorstellung der portablen Systembibliotheken für UNIX und Windows NT kann die Betrachtung der Implementierungsaspekte für TycoonOS in diesem Abschnitt mit einer Analyse der Anforderungen, die das Tycoon-2-System an sein Wirtsbetriebssystem stellt, abgeschlossen werden.

4.3.2. Anforderungen an das Betriebssystem

Das Tycoon-2-System ist für aktuell am Markt relevante Betriebssysteme wie UNIX oder Windows NT konzipiert; eine Unterstützung von Windows 3.1 oder MS-DOS ist nicht vorgesehen. Dies liegt unter anderem an den folgenden Anforderungen, die von Tycoon-2 an das Wirtsbetriebssystem gestellt werden:

- Das Objektspeichersystem von Tycoon-2 ist für die Nutzung eines virtuellen, linearen **32-Bit-Adreßraumes** ausgelegt. Tycoon-2 benötigt erhebliche Speicherressourcen (etwa ab 80 MB), so daß eine Realisierung dieser Anforderungen auf einem Betriebssystem mit einem 16-Bit-Adreßraum oder einer klein dimensionierten Speichersegmentierung (z.B. 64 KB-Blöcke im *Real mode* unter MS-DOS [Althaus 92]) nicht möglich ist.

- Die Standardbibliotheken der Hochsprache TL-2 benötigen ein Dateisystem, das mit **langen Dateinamen** umgehen können muß. Die 8.3-Begrenzung bei MS-DOS bzw. Windows 3.1 (8 Buchstaben für den Dateinamen, 3 Buchstaben für eine optionale Dateierweiterung) disqualifiziert das unter diesen Systemen übliche Dateisystem FAT ([Hamilton, Williams 96] Kapitel 5, *Dateisysteme*).

Neben diesen eher quantitativen Anforderungen gibt es auch noch einige wichtige qualitative Anforderungen an das Betriebssystem, die aus der in [Weikard 98] entwickelten **multiprozessorfähigen Architektur** der virtuellen Maschine von Tycoon-2 resultieren:

- Das Betriebssystem sollte mit der Fähigkeit zum Mehrprogrammbetrieb ein echtes, preemptives Multitasking mit einem multiprozessorfähigen Threadmodell anbieten.
- Eine Konsequenz daraus ist, daß alle für die Realisierung benutzten Systemaufrufe und Laufzeitbibliotheken des Betriebssystems einen voll ablaufinvarianten Objektcode anbieten müssen. Teilweise muß auch bei modernen Betriebssystemen über spezielle Schalter in der Übersetzungssteuerung sichergestellt werden, daß die ablaufinvarianten Versionen der Laufzeitbibliotheken benutzt werden.
- Das Threadmodell des Wirtsbetriebssystems für Tycoon-2 sollte Synchronisationsmechanismen anbieten, die vergleichbar zu denen bei den POSIX-Threads benutzten sind. Als Resultat dieser Arbeit ist auch eine Anpassung an Threadmodelle, die dem der Win32 API ähnlich sind, einfach möglich (z.B. für das Threadmodell von OS/2).

Insgesamt ist damit klar, warum Windows 3.1 oder MS-DOS für eine Realisierung ausscheiden, dagegen können Windows NT oder Windows 95/98 die hier gestellten Anforderungen voll erfüllen. Mögliche Kandidaten für zukünftige Übertragungsprojekte des Tycoon-2-Systems sind auch OS/2 Warp oder BeOS.

4.4. Schnittstellenübersicht

Nach der Vorstellung dieser Implementierungsaspekte von TycoonOS beschäftigen sich die folgenden Abschnitte mit dem Entwurf, den konkreten Schnittstellen und der Realisierung dieser portablen Systembibliothek. Ausgehend von der in Abschnitt 2.2.2 (Abbildung 2.4) entwickelten portablen Ziel-Architektur des Tycoon-2-Systems stellt Abbildung 4.3 eine erste Verfeinerung von TycoonOS dar.

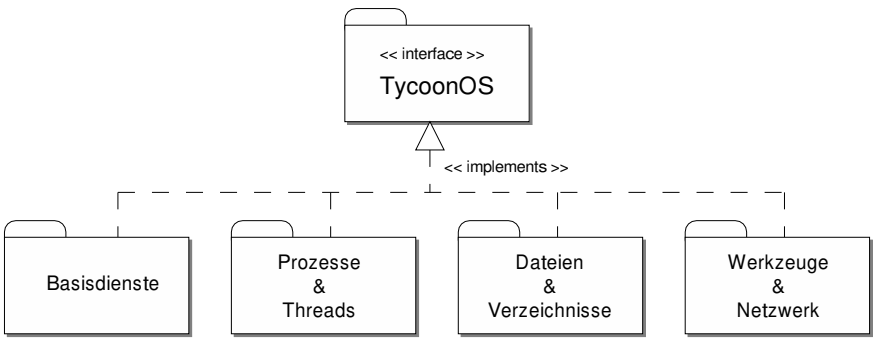


Abbildung 4.3.: Die Dienste von TycoonOS in der Übersicht

Die konkreten Inhalte der in Abbildung 4.3 dargestellten Pakete werden noch im Verlauf der folgenden vier Abschnitte detailliert vorgestellt. Dabei werden vor allem die Darstellung der Schnittstellen für Threads und der Threadsynchronisation einen breiten Raum einnehmen, andere Bestandteile der Bibliothek werden nicht so ausführlich behandelt, wie beispielsweise die Netzwerkfunktionen, da hier Standards zum Einsatz kommen, die auf allen hier betrachteten Betriebssystemen etabliert sind (z.B. BSD Sockets). Eine vollständige Auflistung aller Schnittstellen von TycoonOS findet man im Anhang A.

4.5. Basisdienste

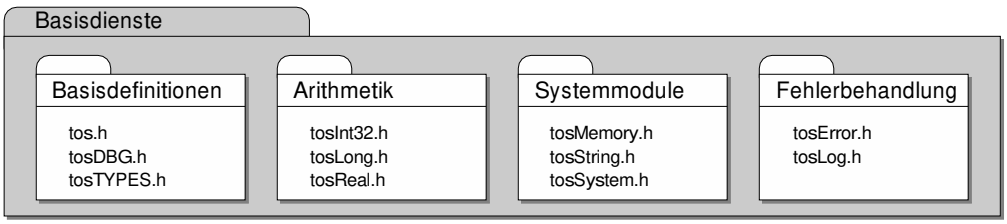


Abbildung 4.4.: Die Basisdienste für TycoonOS

In Abbildung 4.4 sind die Schnittstellen der Basisdienste von TycoonOS dargestellt. Hierzu gehören die Definition verschiedener elementarer Datentypen mit entsprechenden Schnittstellen für Arithmetikfunktionen, allgemeine Dienste für das Fehlermanagement und die Ablaufverfolgung sowie einige weitere systemnahe Dienste für bestimmte Zeichenkettenoperationen, einem direkten Speicherzugriff und zur Bestimmung einiger Parameter der Betriebssystemumgebung.

4.5.1. Basisdefinitionen

Die Schnittstelle `tos.h` bietet über die Funktion `tos_init` den zentralen Einstiegspunkt für alle notwendigen Initialisierungen von TycoonOS an und muß bei der Initialisierung der virtuellen Maschine von Tycoon-2 aufgerufen werden. Von dieser Schnittstelle aus werden auch alle weiteren Basisdefinitionen aus `tosDBG.h` (Schalter für die Ablaufverfolgung) und `tosTYPE.h` (grundlegende Datentypen) geladen.

Über die Definitionen der Schnittstelle `tosDBG.h` kann gezielt für fast alle Dienste von TycoonOS eine Ablaufverfolgung eingeschaltet werden. Dabei können umfangreiche Protokolle der Vorgänge in dieser betriebssystemnahen Schicht erstellt werden, beispielsweise für die Verfolgung von Dateioperationen (`tosFile.h`), das dynamische Laden von Funktionen aus externen Bibliotheken (`tosDll.h`), oder den Aufbau von Netzwerkverbindungen (`tosSocket.h`). Diese Mechanismen zur Ablaufverfolgung werden allerdings nur dann aktiviert, wenn in der Übersetzungssteuerung von Jam der Übersetzungsmodus `debug` eingeschaltet wurde.

Die grundlegenden Datentypen für TycoonOS und die virtuelle Maschine des Tycoon-2-Systems sind in der Schnittstelle `tosTYPES.h` definiert, eine Übersicht über diese Datentypen findet sich in Tabelle 4.1. Erwähnenswert sind in dieser Tabelle lediglich einige übersetzerspezifische Datentypen (ANSI C legt nur die Untergrenzen für einige der in 4.1 aufgeführten Datentypen fest), wie der Datentyp für ganze Zahlen mit einer Wortbreite von 64 Bit. Während sich hierfür bei dem GNU C-Übersetzer unter UNIX der Datentyp `long long int` durchgesetzt hat, muß unter den Windows-Betriebssystemen der Datentyp `__int64` benutzt werden, der spezifisch für Microsoft Visual C++ ist und eventuell bei anderen C-Übersetzern angepaßt werden muß.

Datentyp	**UNIX**	**Windows**
Byte	`unsigned char`	`unsigned char`
Char	`unsigned char`	`unsigned char`
Bool	`enum {FALSE, TRUE}`	`enum {FALSE, TRUE}`
Short	`signed short int`	`signed short int`
Int	`int`	`int`
Long	`long long int`	`__int64`
QWord	`unsigned char`	`unsigned char`
HWord	`unsigned short int`	`unsigned short int`
Word	`unsigned int`	`unsigned int`
DWord	`unsigned long long int`	`unsigned __int64`
Float	`float`	`float`
Double	`double`	`double`
Real	`double`	`double`

Tabelle 4.1.: Datentypen für TycoonOS in der Übersicht

4.5.2. Arithmetikfunktionen

Zu den in `tosTYPES.h` definierten Datentypen gibt es noch einige Schnittstellen für entsprechende Arithmetikfunktionen. Diese Schnittstellen (`tosLong.h`, `tosInt32.h` und `tosReal.h`) sind nur noch von untergeordneter Bedeutung, da die in der Hochsprache TL-2 benutzten atomaren Datentypen über eingebaute Klassen mit entsprechend eingebauten Methoden innerhalb der virtuellen Maschine realisiert worden sind – spezielle primitive Datentypen gibt es in TL-2 nicht mehr (Abschnitt 2.2.1).

Einige Funktionen aus der Schnittstelle `tosInt32.h` werden noch überall dort benutzt, wo für vorzeichenbehaftete Ganzzahlen die volle 32 Bit Wortbreite benötigt wird, beispielsweise bei *Handles* für Dateien. Dies ist deshalb notwendig, da TL-2 diese Datentypen nur mit einer Wortbreite von 30 Bit erfaßt, 2 Bit gehen durch eine Markierung (*tagging*) des Objektspeichersystems verloren [Weikard 98]. Auch die Funktionen für Gleitkommazahlen aus der Schnittstelle `tosReal.h`, wie z.B. `tosReal_sqrt`, `tosReal_cos`, oder `tosReal_sin` werden von der TL-2-Standardbibliothek noch benötigt.

4.5.3. Systemnahe Basisdienste

An zentraler Stelle bei den systemnahen Diensten von TycoonOS steht die Schnittstelle `tosSystem.h` mit Funktionen zur Bestimmung verschiedener Umgebungsparameter des aktuell verwendeten Betriebssystems und der aktuellen Rechnerarchitektur. Dazu gehört die Bereitstellung einer Rechneridentifikation, die sowohl im Objektspeichersystem zur Anpassung externer Objektspeicherzustände an maschinenspezifische Zahlenformate sowie zur richtigen Konfiguration der Ein- und Ausgabeklassen auf Ebene der Hochsprache TL-2 benutzt wird. Im folgenden sind die hierfür nötigen Konstanten und Funktionen dieser Schnittstelle aufgeführt:

```
#define tosSystem_ID_INVALID      0
#define tosSystem_ID_SOLARIS2     1
#define tosSystem_ID_LINUX        2
#define tosSystem_ID_HPUX         3
#define tosSystem_ID_NEXTSTEP     4
#define tosSystem_ID_WINNT        5
#define tosSystem_ID_WIN95        6
#define tosSystem_ID_WIN98        7
#define tosSystem_ID_UNKNOWN      8

extern Int    tosSystem_getID  (void);
extern String tosSystem_getName(void);
```

Bei einer Übertragung von TycoonOS auf ein neues Betriebssystem müssen an dieser Stelle entsprechend neue Identifikatoren vorgesehen werden. Ein weiterer Bestandteil der Schnittstelle `tosSystem.h` ist die statische und dynamische Abfrage der von der Prozessorarchitektur abhängigen Bytefolge bei der Repräsentation von Zahlen:

```
#define tosSystem_BIG_ENDIAN     0
#define tosSystem_LITTLE_ENDIAN  1

/* Dynamisch: */
extern Int tosSystem_getByteOrder  (void);
extern Int tosSystem_getByteOrderOf(Int systemId);

/* Statisch: */
#if defined(rt_LIB_Nextstep_i386) ||
    defined(rt_LIB_Linux_i386)    ||
    defined(rt_LIB_Win32_i386)
  #define tosSystem_BYTEORDER tosSystem_LITTLE_ENDIAN
#else
  #define tosSystem_BYTEORDER tosSystem_BIG_ENDIAN
#endif
```

Während der Initialisierung des Moduls zu dieser Schnittstelle wird das aktuelle Betriebssystem statisch (bedingte Übersetzung), und bei den Windows-Betriebssystemen dynamisch identifiziert, was mittels der Win32 API-Funktion `GetVersionEx` geschieht. Wie die Versionsinformationen dieser Funktion auf die Betriebssystemidentifikatoren abgebildet werden können, ist aus Tabelle 4.2 ersichtlich – bei neuen Windows-Betriebssystemversionen ist an dieser Stelle eventuell eine Anpassung der Identifikation notwendig.

Identifikator	**Versionsinformation**
`tosSystem_ID_WINNT`	`dwPlatformId` = `VER_PLATFORM_WIN32_NT`
`tosSystem_ID_WIN98`	`dwPlatformId` = `VER_PLATFORM_WIN32_WINDOWS` $\wedge$ `dwMajorVersion` $= 4$ $\wedge$ `dwMinorVersion` ≥ 10
`tosSystem_ID_WIN95`	`dwPlatformId` = `VER_PLATFORM_WIN32_WINDOWS` $\wedge$ `dwMajorVersion` $= 4$ $\wedge$ `dwMinorVersion` < 10

Tabelle 4.2.: Versionsinformationen in der Win32 API

Neben der Schnittstelle `tosSystem.h` werden weitere elementare Systemdienste in den Schnittstellen `tosString.h` und `tosMemory.h` angeboten. Die Schnittstelle `tosString.h` stellt einige von der virtuellen Maschine und der Hochsprache TL-2 benötigte Zeichenkettenoperationen zur Verfügung. Hierzu gehört beispielsweise die Funktion `tosString_strdup`, die eine Zeichenkette (ein Feld von Zeichen, welches durch den Wert `0` begrenzt wird) in einen neu angelegten dynamischen Speicherbereich dupliziert.

Die aus dem alten Laufzeitsystem RTS übernommene Schnittstelle `tosMemory.h` ermöglicht über typspezifische *Peek*- und *Poke*-Funktionen den direkten Zugriff auf dynamischen Speicher. Normalerweise arbeitet die Hochsprache TL-2 vollständig auf einem Objektspeicher mit entsprechend sicheren Zugriffsmethoden und einer automatischen Speicherbereinigung, daher ist eine Anforderung und Manipulation von zusätzlichem dynamischen Speicher in der Regel nicht notwendig. Beim Aufruf externer Funktionen müssen jedoch häufig komplexe Datenstrukturen als Parameter für diese Funktionen bereitgestellt werden, und in diesen Fällen bietet die Schnittstelle `tosMemory.h` optimale Mechanismen an, um solche Speicherstrukturen aufzubauen bzw. auszulesen. Zur Verdeutlichung folgt ein kleiner Auszug aus der Schnittstelle von `tosMemory.h`:

```
extern Int   tosMemory_peekInt   (int base, int offs);
extern Char *tosMemory_peekString(int base, int offs);
extern Long  tosMemory_peekLong  (int base, int offs);
...
extern void  tosMemory_pokeInt   (int base, int offs, Int   value);
extern void  tosMemory_pokeString(int base, int offs, Char *value);
extern void  tosMemory_pokeLong  (int base, int offs, Long  value);
```

4.5.4. Fehlerbehandlung und Ablaufverfolgung

Wie schon in Abschnitt 2.2.2 angedeutet, sollte in TycoonOS die Möglichkeit genutzt werden, neue Abstraktionen und Vereinheitlichungen in das Tycoon-2-System einzuführen. Dies betrifft beispielsweise die hier beschriebenen Dienste zur Fehlerbehandlung und zur Ablaufverfolgung in den Schnittstellen `tosError.h` und `tosLog.h`.

Das Auffinden von Fehlerinformationen ist sowohl unter Windows wie auch unter den verschiedenen UNIX-Systemen nicht einheitlich gelöst. Zwar gibt es für Funktionen aus der C-Standardbibliothek die globale Variable `errno`, die die zuletzt aufgetretene Fehlernummer enthält und auf die auch ablaufinvariant (*threadsafe*) zugegriffen werden kann, darüber hinaus gibt es aber auf den verschiedenen Betriebssystemen einige Variationen:

- Alle Funktionen der Win32 API benutzen eine eigene Fehlervariable, die mit der Funktion `GetLastError` ausgelesen und mit der Funktion `SetLastError` gesetzt bzw. gelöscht werden kann. Die Funktionen der WinSocket API, die in den Netzwerkdiensten von TycoonOS benutzt werden, haben ebenfalls ihre eigene Fehlervariable, die mit `WSAGetLastError` gelesen bzw. mit `WSASetLastError` gesetzt werden kann. In beiden Fällen besitzt jeder Betriebssystemthread seine eigene Instanz dieser Variablen, damit auf diese ablaufinvariant zugegriffen werden kann. Die Variable `errno` aus der C-Standardbibliothek bleibt in beiden Fehlersituationen unberührt.

- Unter den UNIX-Betriebssystemen wird für viele Systemaufrufe zwar weitgehend einheitlich `errno` benutzt, trotz der ablaufinvarianten Zugriffsmöglichkeit auf diese Variable bilden aber gerade die POSIX-Threads hier eine Ausnahme: Die meisten Funktionen der POSIX-Threads geben einen Fehlercode unmittelbar als Funktionsergebnis zurück und lassen `errno` unberührt, nur die Funktionen zur Handhabung von Semaphoren signalisieren ihre Fehler über `errno`.

Die Problematik, die in dieser Uneinheitlichkeit liegt, wird vor allem auf der TL-2-Sprachebene deutlich: Hier ist nach dem Aufruf einer externen Funktion gar nicht unmittelbar ersichtlich, aus welchen Subsystemen eventuelle Fehlerinformationen beschafft werden müssen. Eine systemabhängige Fehlerbehandlung wäre die Folge, aber gerade solche Systemabhängigkeiten sollten auf dieser Ebene vermieden werden. Dagegen ist die Ebene der portablen Systembibliothek TycoonOS genau dafür gedacht, solche Systemabhängigkeiten zu kapseln. Hier wird die Fehlerbehandlung über folgende Richtlinien mit Hilfe der Schnittstelle `tosError.h` vereinheitlicht:

- Alle Funktionen der Schnittstelle TycoonOS, die einen Fehler produzieren können, geben einen Ganzzahlwert zurück: -1, falls ein Fehler aufgetreten ist, ansonsten den Wert 0.

- Im Falle eines Fehlers steht in `errno` *immer* eine Fehlernummer, die mit der Funktion `tosError_getCode` *threadsafe* ausgelesen werden kann. Die Funktionsresultate bei den

POSIX-Threads können dabei direkt auf `errno` abgebildet werden, für die Subsystemfehler der Win32 API und der WinSocket API werden die folgenden Konstanten für `errno` eingeführt:

```
#define EWIN32API  6000
#define EWINSOCKET 6001
```

- Für weitere Fehlerinformationen steht ein ganzzahliger Fehlerinfo-Code (über die Funktion `tosError_getCodeDetail`) und ein Fehlertext (über `tosError_getText`) zur Verfügung. In diesen Funktionen aus `tosError.h` ist dabei verborgen, ob und wie Fehlerinformationen beschafft werden können. Die Funktion `tosError_reset` kann schließlich sowohl `errno`, wie auch einen eventuell existierenden Fehlerinfo-Code zurücksetzen.

In Tabelle 4.3 wird gezeigt, wie über diese Richtlinien in Abhängigkeit von dem Wert von `errno` alle Fehlerinformationen auf eine einheitliche Art und Weise beschafft werden können.

Fehlercode (errno)	**Bedeutung**	**Fehlerinfo-Code**	**Fehlertext**
0	Kein Fehler	-	-
1-5999	ANSI C Fehlercode	-	`strerror()`
6000	Win32 API Fehler	`GetLastError`	`FormatMessage()`
6001	WinSocket Fehler	`WSAGetLastError`	-

Tabelle 4.3.: Fehlerinformationen in Abhängigkeit von `errno`

Aus dieser Tabelle geht hervor, daß für die WinSocket-Fehler keine Fehlertexte zur Verfügung stehen, die Funktion `tosError_getText` würde in diesem Fall einfach den Fehlerinfo-Code als Zeichenkette zurückgeben. Die in der Tabelle 4.3 dargestellte Zuordnung für `errno` ist beliebig erweiterbar, falls noch weitere externe Subsysteme in das Tycoon-2-System integriert werden sollen, beispielsweise für ODBC (*Open DataBase Connectivity*). Insgesamt ergibt sich damit auf Ebene der Programmiersprache C in TycoonOS das folgende typische Schema für eine Fehlerbehandlung:

POSIX-Threads:

```
...
int rc = pthread_xxyy(xxyy);
if (rc != 0) {
   errno = rc;
   res   = -1;
}
...
```

Win32 API:

```
...
int rc = PulseEvent(xxyy);
if (rc == 0) {
   errno = EWIN32API;
   res   = -1;
}
...
```

In diesen Beispielen enthält `res` das jeweilige Resultat der TycoonOS-Funktion. Zu beachten ist die unterschiedliche Interpretation der Ergebnisse der Systemfunktionen: Während

Funktionen der Win32 API im Falle eines Fehlers den Wert 0 zurückgeben, wird dieser Wert bei den POSIX-Threads gerade im Falle einer korrekten Funktionsausführung zurückgegeben.

Eine Übertragung der hier angesprochenen einheitlichen Fehlerbehandlung auf die Ebene der Hochsprache TL-2 mit den dort zur Verfügung stehenden Konzepten (Klassen, Objekte oder Ausnahmen) wird noch in Abschnitt 5.1.3 näher behandelt. Abschließend seien an dieser Stelle noch einmal die Definitionen der hier besprochenen Funktionen aus `tosError.h` aufgeführt:

```
extern int tosError_getCode(void);
extern int tosError_getCodeDetail(void);

extern void tosError_setContext(int errCode, int errCodeDetail);

#define tosError_reset() tosError_setContext(0, 0);

extern void tosError_getText(int   errCode,
                             int   errCodeDetail,
                             char *msg,
                             int   msgSize);

extern void tosError_log(int errCode, int errCodeDetail);
```

Die Schnittstelle `tosLog.h` enthält schließlich noch folgende Funktionen zum Schreiben der Meldungen für Fehler und für eine eventuell über `tosDBG.h` eingeschaltete Ablaufverfolgung:

```
extern void tosLog_error(char *module, char *fun, char *msg, ...);
extern void tosLog_debug(char *module, char *fun, char *msg, ...);
```

Fehlermeldungen erfolgen auf die Standard-Fehlerausgabe (`stderr`) und Meldungen für die Ablaufverfolgung werden in die Datei `tycoon.dbg` geschrieben. Für beide Protokolle gibt es in der Schnittstelle `tosLog.h` aber auch Funktionen zum Umleiten der Ausgabe auf andere Dateien oder Geräte:

```
extern void tosLog_redirectError(FILE *newFile);
extern void tosLog_redirectDebug(FILE *newFile);
```

Nach dieser eingehenden Betrachtung der Basisdienste von TycoonOS werden im folgenden Abschnitt die Dienste für Threads und deren Synchronisation beschrieben, dem zentralen Thema dieser Arbeit.

4.6. Prozesse und Threads

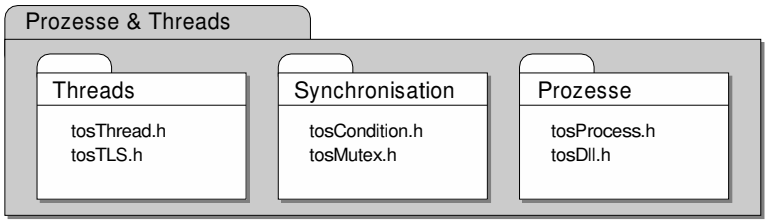


Abbildung 4.5.: Prozesse und Threads im Kontext von TycoonOS

An zentraler Stelle des in Abbildung 4.5 dargestellten Paketes für Prozesse und Threads steht eine portable Schnittstelle zu den für die Implementierung der virtuellen Maschine von Tycoon-2 notwendigen Mechanismen aus dem POSIX-Threadmodell. Hierzu gehören elementare Funktionen für Threads (`tosThread.h`), für die Handhabung von threadspezifischem Speicher (`tosTLS.h`) sowie verschiedene, für das Tycoon-2-System relevante Synchronisationsmechanismen (`tosMutex.h` und `tosCondition.h`). Der Vergleich mit dem Threadmodell der Win32 API steht dabei im Mittelpunkt dieses Abschnittes, wobei auch einige Unterschiede in der Implementierung dieses Threadmodells zwischen Windows NT und Windows 95/98 deutlich werden.

Abschließend werden in diesem Abschnitt noch einige Funktionen für Prozesse aus der Schnittstelle `tosProcess.h` und für das Einbinden von Funktionen aus dynamischen Bibliotheken über die Schnittstelle `tosDll.h` betrachtet.

4.6.1. Grundlegende Threadfunktionen

Zu den grundlegenden Funktionen für Threads gehört das Starten und das Beenden bzw. Abbrechen eines Threads. Hierfür gibt es sowohl bei den POSIX-Threads wie auch in der Win32 API entsprechende Funktionen, wobei es unter den Windows-Betriebssystemen verschiedene Möglichkeiten gibt, einen neuen Thread anzulegen:

Über die Win32 API: Man kann einen neuen Thread mit Hilfe der Win32 API über einen Aufruf der Funktion `CreateThread` anlegen und über die Funktion `ExitThread` wieder beenden. Dies hat zur Konsequenz, daß innerhalb dieses Threads Funktionen der C-Standardbibliothek nicht mehr ablaufinvariant abgearbeitet werden können und nur noch die direkte Benutzung der Win32-API möglich ist. Dies liegt daran, daß für eine Ablaufinvarianz der C-Standardbibliothek beim Anlegen eines neuen Threads gewisse interne Initialisierungen in der Bibliothek für diesen Thread durchgeführt werden müssen; diese werden bei einem direkten Aufruf von `CreateThread` umgangen.

Über die C-Laufzeitbibliothek: Benutzt man zum Anlegen bzw. Beenden eines Threads die C-Bibliotheksfunktion `_beginthread` bzw. `_endthread`, ist dagegen eine sichere,

ablaufinvariante Benutzung dieser Bibliothek sichergestellt. Diese Funktionen besitzen eine vereinfachte Aufrufschnittstelle und haben nur einen geringen Overhead gegenüber den entsprechenden Win32 API-Funktionen, so daß dies die bevorzugte Vorgehensweise für das Anlegen neuer Threads ist.

Benötigt man unbedingt die bei der Funktion `CreateThread` verwendeten Parameter, kann man statt dessen auch die C-Bibliotheksfunktion `_beginthreadex` benutzen, diese besitzt dieselbe Aufrufschnittstelle wie `CreateThread`. Dann muß allerdings auch das Beenden des Threads über die Funktion `_endthreadex` geschehen, desweiteren muß der vom Thread belegte Handle vorher explizit geschlossen werden (mittels `GetCurrentThread` und `CloseHandle`).

Es gibt noch weitere Varianten für das Anlegen und Beenden von Threads unter Windows, beispielsweise mit Hilfe der MFC (*Microsoft Foundation Classes*); diese Varianten sind an dieser Stelle jedoch nicht weiter von Interesse. Für das Tycoon-2-System muß die Variante über die C-Laufzeitbibliothek mit den Funktionen `_beginthreadex` und `_endthreadex` benutzt werden, da die im Ablauf eines Tycoon-2-Threads benutzten TL-2-Bibliotheksfunktionen einen umfangreichen Gebrauch von Funktionen aus der C-Standardbibliothek machen.

Sowohl bei den POSIX-Threads wie auch in der Win32 API gibt es auch Funktionen, um einen Thread von einem anderen Thread aus abzubrechen. Diese Funktionen (`pthread_kill` bzw. `TerminateThread`) werden zwar von TycoonOS in der Schnittstelle tosThread.h angeboten, deren Verwendung ist allerdings kritisch, da unter Umständen nicht alle Systemressourcen von dem abgebrochenen Thread wieder freigegeben werden. Für die zukünftige Weiterentwicklung von TycoonOS könnte hier eine interne Ressourcenverwaltung, wie sie beispielsweise bei den $\text{Pthreads}_{\text{Win32}}$ realisiert ist, für eine Verbesserung sorgen.

Insgesamt ergibt sich damit die in Tabelle 4.4 dargestellte Übersicht über die elementaren Grundfunktionen für Threads, die über die Schnittstelle tosThread.h zur Verfügung gestellt werden.

Funktion	**Pthreads**	**Win32 API**
Thread anlegen	`pthread_create` `pthread_detach`	`_beginthreadex`

Thread beenden	`pthread_destroy`	`GetCurrentThread`
		`CloseHandle`
		`_endthreadex`

Thread abbrechen	`pthread_kill`	`TerminateThread`

Tabelle 4.4.: Anlegen und Beenden von Threads bei den Pthreads und in der Win32 API

Beim Aufruf der Funktion `tosThread_create` zum Anlegen von Threads gibt es noch ein Problem, das bisher noch nicht beachtet wurde: In der Schnittstelle zu dieser Funktion muß ein Verweis auf die *Ausführungsfunktion* angegeben werden, die den Kontrollfluß für den neuen Thread enthält. Dabei können auch Parameter für diese Ausführungsfunktion

beim Anlegen des Threads spezifiziert werden. Die Typen für die Ausführungsfunktion und ihre Parameter sind bei den POSIX-Threads und der Win32 API unterschiedlich, was deshalb problematisch ist, weil genau diese Typen auch bei der Definition der Ausführungsfunktion angegeben werden müssen.

Neben dem allgemeinen Datentyp für die Identifikation eines Threads (`tosThread_T`) müssen in der Schnittstelle tosThread.h deshalb noch weitere Typen zur Verfügung gestellt werden:

- Ein Funktionsprototyp `tosThread_FUNC_T`, der als Parametertyp für die Ausführungsfunktion beim Aufruf der Funktion `tosThread_create` benutzt wird.
- Der Ergebnistyp `tosThread_FUNC_RET`, der bei der Definition der Ausführungsfunktion benutzt werden muß. Dieser Datentyp kann auch für eine Typanpassung des Rückgabewertes (*casting*) in der Ausführungsfunktion benutzt werden.
- Der Typ des Parameters `tosThread_FUNC_PARA`, der an die Ausführungsfunktion übergeben werden kann.
- Bei der Win32 API ist schließlich ist noch eine besondere Spezifikation des verwendeten Mechanismus der Parameterübergabe bei einen Aufruf der Ausführungsfunktion notwendig (`tosThread_FUNC_DECL`). Diese Definition wird bei den POSIX-Threads nicht benötigt und ist dort daher leer.

Aus dieser Aufstellung ergeben sich neben `tosThread_T` folgende Typdeklarationen, die in der Schnittstelle `tosThread.h` angeboten werden:

```
#ifdef rt_LIB_Win32_i386
  #define tosThread_FUNC_RET  unsigned
  #define tosThread_FUNC_PARA void *
  #define tosThread_FUNC_DECL __stdcall
#else
  #define tosThread_FUNC_RET  void *
  #define tosThread_FUNC_PARA void *
  #define tosThread_FUNC_DECL
#endif

typedef tosThread_FUNC_RET
           (tosThread_FUNC_DECL *tosThread_FUNC_T)(tosThread_FUNC_PARA);
```

Die vollständigen Definitionen der Schnittstelle `tosThread.h` können dem Anhang A entnommen werden. Abschließend wird an dieser Stelle in einer etwas gekürzten Form gezeigt, wie die hier beschriebenen Datentypen in der virtuellen Maschine von Tycoon-2 angewendet werden, um neue TL-2-Threads zu erzeugen. Dabei wird die zentrale Interpreterschleife der virtuellen Maschine (`tvm_run`) in der Ausführungsfunktion für den Betriebssystemthread aufgerufen:

```
typedef struct tyc_Thread {
  ...
  tosThread_T tid;
  ...
} tyc_Thread;

tosThread_FUNC_RET tosThread_FUNC_DECL runThread() {
  return (tosThread_FUNC_RET) tvm_run();
}
tyc_Thread * tmthread_new() {
  tyc_Thread * new_TL2_Thread;
  ...
  rc = tosThread_create(&(new_TL2_Thread->tid), runThread, ...);
  ...
  return new_TL2_Thread;
}
```

Nachdem deutlich geworden ist, wie man mit Hilfe der Schnittstelle `tosThread.h` neue Threads starten und beenden kann, stellen sich in diesem Zusammenhang noch drei sehr wichtige Fragen, die in den folgenden Abschnitten behandelt werden sollen: Wie kann auf die Abarbeitung der Threads über Prioritäten Einfluß genommen werden, wie können lokal zu einem Thread Daten gespeichert werden und wie kann ein Zugriff auf alle globalen Ressourcen durch mehrere Threads synchronisiert werden.

4.6.2. Prioritätsebenen und Scheduling

Nachdem ein Thread gestartet wurde, wird dessen Verarbeitung vom **Scheduler** des Betriebssystems übernommen. Da sich in der Regel viele unterschiedliche Prozesse und Threads denselben Prozessor für ihre Verarbeitung teilen müssen, bekommt der Thread zunächst vom Betriebssystem eine bestimmte **Priorität** zugewiesen und wird in eine davon abhängige Warteschlange eingereiht. Je nach aktueller Scheduling-Strategie des Betriebssystems werden diese Warteschlangen abgearbeitet, wobei bei dem hier betrachteten **Prioritäts-Scheduling** mit mehreren Warteschlangen ([Tanenbaum 92]) die grundsätzliche Gemeinsamkeit darin besteht, daß zunächst alle höher priorisierten Warteschlangen verarbeitet werden. Die für diese Arbeit relevanten Scheduling-Strategien werden im folgenden kurz vorgestellt[4]:

First in First out (FIFO): Sind in einer Warteschlange auf derselben Prioritätsebene mehrere Threads eingetragen, hat immer der Thread in der Verarbeitung Vorrang, der **zuerst** in diese Warteschlange eingereiht wurde. Andere Threads derselben Warteschlange (bzw. niedriger priorisierte Threads) kommen erst dann an die Reihe, wenn

[4] Die verschiedenen Scheduling-Strategien und Prioritätsmodelle in den unterschiedlichen Betriebssystemen können in dieser Arbeit nur im Überblick vorgestellt werden, soweit sie für die Threadmodelle von POSIX und der Win32 API von Bedeutung sind. Für eine Vertiefung dieses sehr umfangreichen Themenkomplexes muß auf [Tanenbaum 92] verwiesen werden.

der erste Thread in der Warteschlange fertig abgearbeitet oder durch das Warten auf ein Ereignis *blockiert* ist. Wann immer dieser erste Thread aber wieder Rechenzeit beansprucht, bekommt er diese auch sofort zugewiesen, es sei denn, er wird durch einen höherpriorisierten Thread einer anderen Warteschlange verdrängt. Es gibt keine Beschränkung in der Zuteilung von Rechenzeit an den aktuell aktiven Thread wie beispielsweise beim Zeitscheiben-Verfahren (*Time slicing*). Da ein solcher Thread damit für eine beliebige Dauer seinen Prozessor blockieren kann, ist die Anwendung dieser Strategie in der Praxis kaum vorzufinden.

Round Robin (RR): Diese Strategie arbeitet im Prinzip ähnlich wie die FIFO-Strategie, nur daß zusätzlich ein Zeitscheiben-Verfahren eingeführt wird: Jeder Thread bekommt ein festes **Zeitfenster** (*Quantum*) für seine Rechenzeit zugesprochen, nach Ablauf dieses Zeitfensters wird der Thread durch den nächsten, in der Warteschlange wartenden Thread verdrängt. Dabei findet, abstrakt gesehen, im Vergleich mit der FIFO-Strategie eine Rotation innerhalb der Warteschlange statt, so daß jeder Thread einmal der erste in der Warteschlange ist. Dieses Verfahren garantiert also, daß alle Threads ein Zeitfenster für ihre Ausführung zugesprochen bekommen, was allerdings mit den Kosten eines intensiveren Kontextwechsels verbunden ist.

Der Betriebssystemkern von Windows NT (*Microkernel*) kennt nur die Scheduling-Strategie *Round Robin*, wobei es allerdings eine sehr differenzierte Handhabung in der Zuteilung des Quantums zu einem Prozeß bzw. Thread gibt. Beispielsweise bekommt ein Prozeß, der von der Server-Version dieses Betriebssystems gestartet wurde (Windows NT Server), von Anfang an ein größeres Quantum zugesprochen als eine Klient-Anwendung (Windows NT Workstation). Während Prozesse und Threads aus der Prioritätsklasse *Echtzeit* (siehe unten) nach einer Verdrängung immer wieder dasselbe Quantum zugesprochen bekommen, wird dieses Quantum bei normalen Prozessen mit jeder Verdrängung reduziert und erst bei 0 wieder auf den ursprünglichen Wert zurückgesetzt [Solomon 98].

Bei den POSIX-Threads werden drei verschiedene Scheduling-Strategien berücksichtigt: `SCHED_RR` (*Round Robin*), `SCHED_FIFO` und `SCHED_OTHER`; dabei können diese Strategien dynamisch über die Funktion `pthread_setschedparam` eingestellt werden. Die Bedeutung der Strategie `SCHED_OTHER` ist nicht weiter festgelegt, beispielsweise ist diese bei Solaris mit `SCHED_FIFO` identisch. Unter dem Betriebssystem Linux ist eine Manipulation von Prioritäten nur unter den Scheduling-Strategien `SCHED_RR` oder `SCHED_FIFO` möglich. Da die Implementierung der POSIX-Threads auf diesem System (LinuxThreads) zunächst aber unter der Scheduling-Strategie `SCHED_OTHER` arbeitet, wurde für experimentelle Zwecke in der Schnittstelle `tosThread.h` eine Änderungsmöglichkeit für die Scheduling-Strategien vorgesehen:

```
#define tosThread_SCHED_FIFO  SCHED_FIFO
#define tosThread_SCHED_RR    SCHED_RR
#define tosThread_SCHED_OTHER SCHED_OTHER

extern int tosThread_getSchedClass(tosThread_T tid, Int *class);
extern int tosThread_setSchedClass(tosThread_T tid, Int  class);
```

Das Modell für die Vergabe von Prioritäten ist bei den POSIX-Threads relativ einfach aufgebaut: Es gibt zwei Konstanten, die abhängig von der Scheduling-Strategie eine global gültige minimale (c_{min}) und maximale (c_{max}) Prioritätsebene festlegen. Erfolgt beim Anlegen eines neuen Threads keine Angabe der Priorität, wird dieser immer mit der mittleren Priorität $((c_{max} - c_{min})/2) + c_{min}$ gestartet, ansonsten kann explizit ein beliebiger Wert zwischen c_{min} und c_{max} angegeben werden. Über eine weitere Strukturierung dieser Werte (z.B. für verschiedene *Prioritätsklassen*) sagt der POSIX-Standard nichts weiteres aus, es wird lediglich festgelegt, daß das Intervall $[c_{min} \ldots c_{max}]$ wenigstens 32 Werte umfassen sollte.

Bei Windows NT ist das Prioritätsschema etwas komplexer. Es gibt eine absolute Skala von 32 Prioritätsebenen mit Werten von 0 bis 31, die sich wie folgt aufteilen:

- Es gibt 16 hohe Prioritätsebenen (16-31) für Echtzeitanwendungen. Diese sind höher als die der meisten Systemprozesse, wie z.B. für Maus-Ereignisse oder Festplattenaktivitäten, deshalb kann ein fehlerhafter Thread auf dieser Prioritätsebene unter Umständen diese wichtigen Betriebssystemprozesse blockieren.

- Desweiteren gibt es 15 variable Prioritätsebenen (1-15) für die normalen Anwendungsprozesse.

- Die niedrigste Prioritätsebene 0 ist für das System reserviert und kann nicht an Benutzerprozesse zugewiesen werden.

Diese Prioritätsebenen werden nach zwei Gesichtspunkten unterteilt: Zunächst gibt es vier **Prioritätsklassen**, die beim Anlegen eines neuen Prozesses benutzt werden (Träge, Normal, Hoch und Echtzeit). Innerhalb eines Prozesses können sieben **relative Prioritäten** für die einzelnen, durch diesen Prozeß erzeugten Threads vergeben werden (Träge, Niedrig, Vermindert, Normal, Erhöht, Hoch und Zeitkritisch). Die Verteilung der damit für die Win32 API relevanten 28 Prioritäten auf die oben genannten 31 für Benutzerprozesse vorgesehene Prioritätsebenen zeigt die aus [Hamilton, Williams 96] abgeleitete Tabelle 4.5:

		Prozeß-Prioritäten			
		Echtzeit	Hoch	Normal	Träge
Thread Prioritäten	Zeitkritisch	31	15	15	15
	Hoch	26	15	10	6
	Erhöht	25	14	9	5
	Normal	24	13	8	4
	Vermindert	23	12	7	3
	Niedrig	22	11	6	2
	Träge	16	1	1	1

Tabelle 4.5.: Verteilung der Prozeß- und Thread-Prioritäten in der Win32 API

Legt ein Prozeß einen neuen Thread ohne explizite Angabe der relativen Priorität an, erhält dieser die Priorität *normal*, was abhängig von der Prioritätsklasse des Prozesses der Prioritätsebene 24, 13, 8, oder 4 entspricht (Tabelle 4.5). Der Sinn dieser differenzierten Aufteilung der Prioritäten unter Windows ist die Unterstützung des verantwortungsvollen Umgangs mit Thread-Prioritäten – die Priorität eines Threads sollte immer in der Umgebung seiner Prozeß-Priorität liegen. Auch sollte die Thread-Priorität eines normalen Prozesses die niedrigste Thread-Priorität eines Echtzeitprozesses nicht übersteigen.

Die Prioritäten für Prozesse und Threads können sowohl bei den POSIX-Threads wie auch unter der Win32 API dynamisch mit Hilfe der in Tabelle 4.6 angegebenen Funktionen geändert werden. Eine Besonderheit bei Windows NT ist das *dynamic boosting* der Thread-Priorität: Bei einer Benutzereingabe wird der aktuelle Thread automatisch um zwei relative Prioritätsstufen erhöht, um eine schnellere Antwortzeit zu ermöglichen. Diese Fähigkeit gibt es weder unter Windows 95/98 noch bei den POSIX-Threads.

Funktion	**Pthreads**	**Win32 API**
Prozeß-Priorität lesen	`getpriority` (ab BSD 4.2)	`GetPriorityClass`
Prozeß-Priorität setzen	`setpriority` (ab BSD 4.2)	`SetPriorityClass`
Thread-Priorität lesen	`pthread_getschedparam`	`GetThreadPriority`
Thread-Priorität setzen	`pthread_setschedparam`	`SetThreadPriority`

Tabelle 4.6.: Manipulation von Prioritäten unter Pthreads und der Win32 API

Bisher wird von der Manipulation von Prioritäten im Tycoon-2-System kein Gebrauch gemacht. Für Demonstrationszwecke wurden in der Schnittstelle `tosThread.h` lediglich die Funktionen zur Manipulation von Thread-Prioritäten unter Verwendung der in Tabelle 4.6 aufgeführten Funktionen realisiert:

```
int tosThread_getPriority(tosThread_T tid, Int *pri);
int tosThread_setPriority(tosThread_T tid, Int  pri);
```

Diese Funktionen werden bisher noch nicht auf Ebene der Hochsprache TL-2 angeboten, auch sind die als Parameter anzugebenden Prioritäten noch systemspezifisch und müßten für eine systemunabhängigere Nutzung über entsprechend definierte Konstanten zur Verfügung gestellt werden.

4.6.3. Dynamischer threadspezifischer Speicher

Die Ausführungen in Abschnitt 4.5.4 zur Handhabung von Fehlern über die threadspezifische Variable `errno` unterstreichen die Notwendigkeit der Definition von **threadspezifischen Daten**. In der virtuellen Maschine des Tycoon-2-Systems werden solche threadspezifischen Daten benötigt, um lokal zu jedem Betriebssystemthread den Deskriptor des dazugehörigen TL-2-Threads zu speichern. Solche Deskriptoren enthalten Verweise auf alle relevanten Daten

eines TL-2-Threads, beispielsweise auf seinen aktuellen Ausführungskontext (*Jump buffer*), seinen Kellerspeicher oder seine Ausführungsfunktion [Weikard 98].

Neben den unterschiedlichen, vom jeweiligen C-Übersetzer abhängigen Möglichkeiten, statischen threadspezifischen Speicher zu definieren, gibt es bei den POSIX-Threads und in der Win32 API eine fast einheitliche Schnittstelle zur Handhabung **dynamischen thread-spezifischen Speichers**. Dabei wird für jeden Prozeß eine Tabelle mit Werten verwaltet, für die jeder Thread eine eigene Instanz mit den entsprechenden Belegungen besitzt. Diese Werte (*Schlüssel* bei den POSIX-Threads bzw. *Indizes* bei der Win32 API) werden durch ganze Zahlen mit einer Wortbreite von 64 Bit repräsentiert.

Abbildung 4.6 verdeutlicht diesen Zusammenhang, bei dem die zu speichernden Werte direkt in die jeweiligen Zellen abgelegt werden können. Überschreiten die zu speichernden Daten die Wortbreite von 64 Bit, müssen diese auf dem dynamischen Speicher angelegt werden. In diesem Fall werden nur noch die Zeiger auf die jeweiligen Datenbereiche in den Zellen der Tabelle gespeichert.

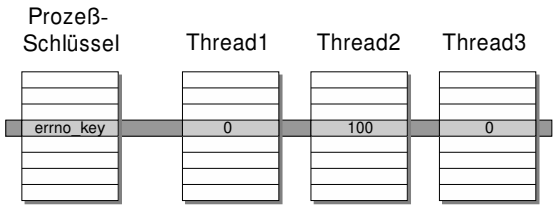


Abbildung 4.6.: Zugriff auf threadspezifische Daten über eine Schlüssel-Tabelle

In der Schnittstelle `tosTLS.h` sind die Funktionen zum Anlegen und Löschen neuer Schlüssel (`tosTLS_keyCreate` bzw. `tosTLS_keyDelete`) sowie zum Eintragen und Auslesen der zu den jeweiligen Schlüsseln gehörenden threadspezifischen Daten (`tosTLS_setValue` bzw. `tosTLS_getValue`) definiert. In Tabelle 4.7 sind die unterschiedlichen Implementierungen für diese Funktionen bei den POSIX-Threads und in der Win32 API dargestellt.

Funktion	**Pthreads**	**Win32 API**
Schlüssel anlegen	`pthread_key_create`	`TlsAlloc`
Schlüssel löschen	`pthread_key_delete`	`TlsFree`
Wert eintragen	`pthread_setspecific`	`TlsSetValue`
Wert auslesen	`pthread_getspecific`	`TlsGetValue`

Tabelle 4.7.: Threadspezifischer Speicher bei den Pthreads und der Win32 API

Neben threadspezifischen Speicher, für den jeder Thread eine eigene Kopie besitzt und auf den ohne besondere einschränkende Bedingungen zugegriffen werden kann, werden gemeinsame Daten für verschiedene Threads auf dem globalen dynamischen Speicher abgelegt. Für einen konsistenten Zugriff auf diese Daten sind die im folgenden betrachteten Synchronisationsmaßnahmen notwendig.

4.6.4. Synchronisation

Für die nebenläufige Ausführung verschiedener Prozesse und Threads in einem Betriebssystem mit Mehrprogrammbetrieb sind Kommunikationsmechanismen zur Synchronisation dieser Prozesse und Threads von elementarer Bedeutung.

Die Win32 API der Windows-Betriebssysteme stellt vielfältige Synchronisationsmechanismen zur Verfügung, wie atomare Operationen, gegenseitigen Ausschluß (Mutex), Semaphoren und vor allem die **Ereignis-Objekte** (*Event objects*), die bei den Windows-Betriebssystemen eine zentrale Rolle spielen [Hamilton, Williams 96; Iseminger 99]. Bis auf die atomaren Operationen können alle Synchronisationsmechanismen mit systemweit eindeutigen Namen versehen werden, so daß sie auch prozeßübergreifend benutzt werden können. Beispielsweise kann eine benannte Semaphore mit einer beliebigen Lebensdauer zum Schutz einer Ressource von unterschiedlichen Prozessen benutzt werden.

Bei den POSIX-Threads kennt man neben dem gegenseitigen Ausschluß und den Semaphoren auch die **Bedingungsvariablen** (*Condition variables*) [ISO/IEC 96; Lewis, Berg 98]. Dahinter verbirgt sich das gleiche Konzept wie bei den Ereignis-Objekten der Win32 API: Das Warten auf ein beliebig komplexes Ereignis. In der Programmierschnittstelle und bei der Initialisierung gibt es zwischen Ereignis-Objekten und Bedingungsvariablen jedoch kleine, im folgenden noch genauer betrachtete Unterschiede, die zu einigen Schwierigkeiten bei der Portierung von Software führen können. Die Synchronisation zwischen Threads in unterschiedlichen Prozessen ist im POSIX-Standard nicht näher festgelegt, bei einigen UNIX-Systemen können allerdings für eine prozeßübergreifende Synchronisation alle Synchronisationsvariablen in Dateien abgelegt werden (siehe [Lewis, Berg 98], *Cross process synchronisation*).

In der threadbasierten Architektur des Tycoon-2-Systems wird eine prozeßübergreifende Synchronisation nicht benötigt, dagegen werden für die Synchronisation zwischen Threads zwei der im POSIX-Standard angebotenen Mechanismen verwendet: Der gegenseitige Ausschluß und die Bedingungsvariablen. Beide Mechanismen müssen in der Implementierung von TycoonOS auf das Threadmodell der Win32 API abgebildet werden, was im folgenden genauer beschrieben wird.

Mutex

Durch einen **Mutex** (gegenseitigen Ausschluß, *MUTual EXclusion*) kann der Zugriff auf eine Ressource (gemeinsamer Speicher, Gerät etc.) auf einen einzelnen Prozeß beschränkt werden – ähnlich wie ein Staffelhölzchen darf immer nur ein Prozeß zur Zeit im Besitz dieser Synchronisationsvariablen sein. Im Gegensatz dazu darf bei einer **Semaphore** [Tanenbaum 92] die Anzahl der gleichzeitig nutzenden Prozesse auch größer sein, sie darf lediglich eine vorher festgelegte Konstante $n \geq 1$ nicht überschreiten. Im Tycoon-2-System wird ein Mutex beispielsweise dafür benutzt, um im Objektspeichersystem das Anlegen neuer Objekte zu serialisieren (der Objektallokator ist einer der nicht ablaufinvarianten Komponenten [Weikard 98]) oder um in der virtuellen Maschine den Zugriff auf den Methoden-Cache zur Beschleunigung der Methodensuche zu synchronisieren.

Sowohl bei den POSIX-Threads wie auch unter der Win32 API gibt es eine direkte Unterstützung für den gegenseitigen Ausschluß, daher sieht die entsprechende Typdefinition in der Schnittstelle `tosMutex.h` recht einfach aus:

```
#ifdef rt_LIB_Win32_i386
  #define tosMutex_T HANDLE
#else
  #define tosMutex_T pthread_mutex_t
#endif
```

Um einen Mutex anlegen und wieder freigeben zu können, gibt es bei den POSIX-Threads und der Win32 API separate Funktionen, wobei das Freigeben unter der Win32 API mit der allgemeinen Funktion `CloseHandle` geschieht. Zu den weiteren typischen Funktionen für einen Mutex gehören das Anfordern und Freigeben einer Sperre (*Lock*) sowie das Prüfen, ob diese Sperre aktuell verfügbar ist. Diese Funktionen sind mit ihren unterschiedlichen Implementierungen bei den POSIX-Threads und der Win32 API in Tabelle 4.8 dargestellt.

Funktion	**Pthreads**	**Win32 API**
Mutex anlegen	`pthread_mutex_init`	`CreateMutex`
Mutex freigeben	`pthread_mutex_destroy`	`CloseHandle`
Sperre anfordern	`pthread_mutex_lock`	`WaitForSingleObject`
Verfügbarkeit der Sperre prüfen	`pthread_mutex_trylock`	`WaitForSingleObject`
Sperre freigeben	`pthread_mutex_unlock`	`ReleaseMutex`

Tabelle 4.8.: Gegenseitiger Ausschluß bei den Pthreads und der Win32 API

Einer kleinen Anmerkung bedarf die Funktion `WaitForSingleObject` unter der Win32 API: Dabei handelt es sich um eine allgemeingültige Warte-Funktion, die auf alle Synchronisationsobjekte wie Mutexe, Ereignis-Objekte oder Semaphoren angewendet werden kann. Ein Parameter dieser Funktion legt die maximale Wartezeit fest, dabei kann die Konstante `INFINITE` (unendlich) oder auch der Wert `0` benutzt werden. Bei letzterem überprüft diese Funktion lediglich die aktuelle Verfügbarkeit des Objektes, auf das gewartet werden soll und weist damit die gleiche Semantik wie die Funktion `pthread_mutex_trylock` auf.

In TycoonOS werden die Funktionen aus Tabelle 4.8 über folgende Schnittstelle angeboten:

```
extern int tosMutex_init   (tosMutex_T *mx);
extern int tosMutex_destroy(tosMutex_T *mx);
extern int tosMutex_lock   (tosMutex_T *mx);
extern int tosMutex_trylock(tosMutex_T *mx);
extern int tosMutex_unlock (tosMutex_T *mx);
```

Während bei einem Mutex die Übertragung auf die Windows-Betriebssysteme relativ unkompliziert ist, gibt es bei der Abbildung der im folgenden betrachteten Bedingungsvariablen auf die Win32 API erheblich größere Probleme.

Bedingungsvariablen

Eine **Bedingungsvariable** (*Condition variable*) kann einen Prozeß oder einen Thread so lange blockieren, bis eine Bedingung erfüllt ist. Für die konsistente Abfrage und Veränderung dieser Bedingung wird zusätzlich ein Mutex benötigt; kein Prozeß bzw. Thread sollte ohne dessen Besitz die Bedingung manipulieren.

Um diesen Vorgang besser zu verstehen, folgt hier ein kurzes Beispiel für einen Thread, der auf das Eintreten der Bedingung `count > 0` wartet:

1. Zunächst muß der Thread einen Mutex `mx` anfordern, um die konsistente Manipulation der Variablen `count` zu sichern; dann testet der Thread die Bedingung `count > 0`.

2. Ist die Bedingung wahr, braucht der Thread nicht zu warten, er kann beispielsweise die durch `count` gezählte Ressource verbrauchen und `count` um eins herabzählen. Durch den Besitz von `mx` ist beim Herabzählen sichergestellt, daß der Wert von `count` immer noch größer als 0 ist. Anschließend muß der Thread den Mutex `mx` wieder freigeben.

3. Ist die Bedingung nicht wahr, muß der Thread warten. Dabei muß der Mutex `mx` vor dem Warten freigegeben werden, sonst könnte die Bedingung niemals durch einen anderen Thread verändert werden. Dieses Freigeben wird vom System atomar innerhalb der Wartefunktion durchgeführt.

4. Führt ein anderer Thread eine Änderung an `count` durch, signalisiert er dieses anschließend, was zum Aufwachen des wartenden Threads führt.

5. Für den aufwachenden Thread belegt das System wieder den Mutex `mx`, so daß dieser wieder die Bedingung `count > 0` prüfen kann. Ist die Bedingung wahr, kann mit der Verarbeitung unter Punkt 2, sonst unter Punkt 3 fortgesetzt werden.

Bei dem hier beschriebenen Ablauf handelt es sich um die Verbraucherseite des *Erzeuger-Verbraucher*-Problems [Tanenbaum 92]. Der Pseudocode für diesen Ablauf unter Benutzung der Schnittstelle zu den POSIX-Threads ist im folgenden dargestellt:

Erzeuger:

```
LOOP:
  pthread_mutex_lock(mx);
  doErzeuge();
  count = count + 1;
  pthread_mutex_unlock(mx);
  pthread_cond_signal(cv);
END
```

Verbraucher:

```
LOOP:
  pthread_mutex_lock(mx);
  while (!(count > 0)) {
    pthread_cond_wait(cv, mx);
  }
  doVerbrauch();
  count = count - 1;
  pthread_mutex_unlock(mx);
END
```

Bei dem hier betrachteten Synchronisationsvorgang mit Bedingungsvariablen ist ein sehr wichtiger Aspekt nur beiläufig erwähnt worden: Das Freigeben des Mutex und das anschließende Warten des Verbraucherthreads unter Punkt 3 muß **atomar** geschehen. Ist dies nicht sichergestellt, könnte ein Signal des Erzeugers schon vor Beginn des Wartens erfolgen und verlorengehen (*Lost wakeup* [Lewis, Berg 98]). Im Gegensatz zu diesem Problem muß das Ende des Wartens und das erneute Anfordern des gegenseitigen Ausschlusses nicht atomar erfolgen, schlimmstenfalls ändert sich zwischenzeitlich die Bedingung und das Aufwachen war umsonst.

Bisher wurden das **Signalisieren** und **Warten** auf ein Ereignis als elementare Operationen für Bedingungsvariablen identifiziert. Betrachtet man sich den Fall, daß mehrere Threads nebenläufig auf dasselbe Ereignis warten, würde ein einfaches Signal immer nur einen einzelnen Thread aktivieren. Darüberhinaus gibt es aber auch die Möglichkeit, ein **Broadcast-Signal** an alle aktuell wartenden Threads zu senden; diese Reaktivierung aller Threads findet dabei in einer nicht näher definierten Reihenfolge statt.

Broadcast-Signale finden auch in der virtuellen Maschine des Tycoon-2-Systems Anwendung, beispielsweise werden Bedingungsvariablen dafür eingesetzt, alle aktuell laufenden Threads vor einer Speicherbereinigung an einem *Synchronisationspunkt* [Weikard 98] anzuhalten. Durch ein *Broadcast*-Signal werden nach der Speicherbereinigung alle wartenden Threads auf einmal reaktiviert.

In der Win32 API können Bedingungsvariablen am ehesten mit Ereignis-Objekten nachgebildet werden. Über die Funktion `PulseEvent` kann ein einzelnes Signal gesendet werden, während das Warten auf einen Thread mit Hilfe der schon weiter oben vorgestellten Funktion `WaitForSingleObject` geschieht. In Abbildung 4.7 ist dieser Prozeß in der Win32 API skizziert.

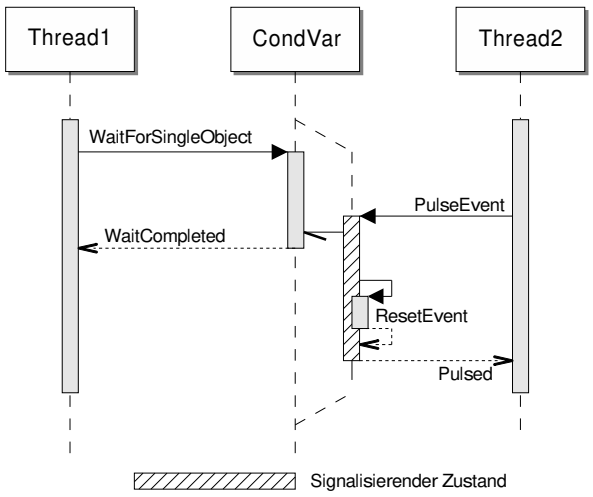


Abbildung 4.7.: Synchronisation mit Ereignis-Objekten

In der Win32 API ist dabei zu beachten, daß bei allen Synchronisationsvariablen (Ereignis-

Objekte, Semaphoren, Mutexe etc.) zwischen den Zuständen **signalisierend** und **nicht-signalisierend** unterschieden wird: Im signalisierenden Zustand wird eine Synchronisationsvariable nicht blockieren, alle schlafenden Threads werden reaktiviert, solange die Variable im signalisierenden Zustand bleibt. Dagegen wird eine Synchronisationsvariable im nicht-signalisierenden Zustand blockieren, alle anfragenden Threads werden warten müssen. Über die Funktion `SetEvent` kann man den signalisierenden Zustand explizit setzen (anstatt mit `PulseEvent` in Abbildung 4.7) und über die Funktion `ResetEvent` kann dieser Zustand dann wieder auf nicht-signalisierend zurückgesetzt werden.

Für das Zurücksetzen des signalisierenden Zustandes bei Ereignis-Objekten gibt es in der Win32 API zwei unterschiedliche Möglichkeiten: Bei **automatischen Ereignis-Objekten** wird nach dem Benachrichtigen des ersten Threads der Zustand automatisch über einen internen Aufruf von `ResetEvent` auf nicht-signalisierend zurückgesetzt. In Abbildung 4.7 ist dies der Fall, bei `CondVar` handelt es sich offensichtlich um ein automatisches Ereignis-Objekt. Bei **manuellen Ereignis-Objekten** geschieht dieses Zurücksetzen nicht automatisch, das Ereignis-Objekt bleibt solange im signalisierenden Zustand, bis einer der Threads explizit `ResetEvent` aufruft – bis zu diesem Zeitpunkt werden alle wartenden Threads gelöst. Dieser Vorgang ist in Abbildung 4.8 dargestellt.

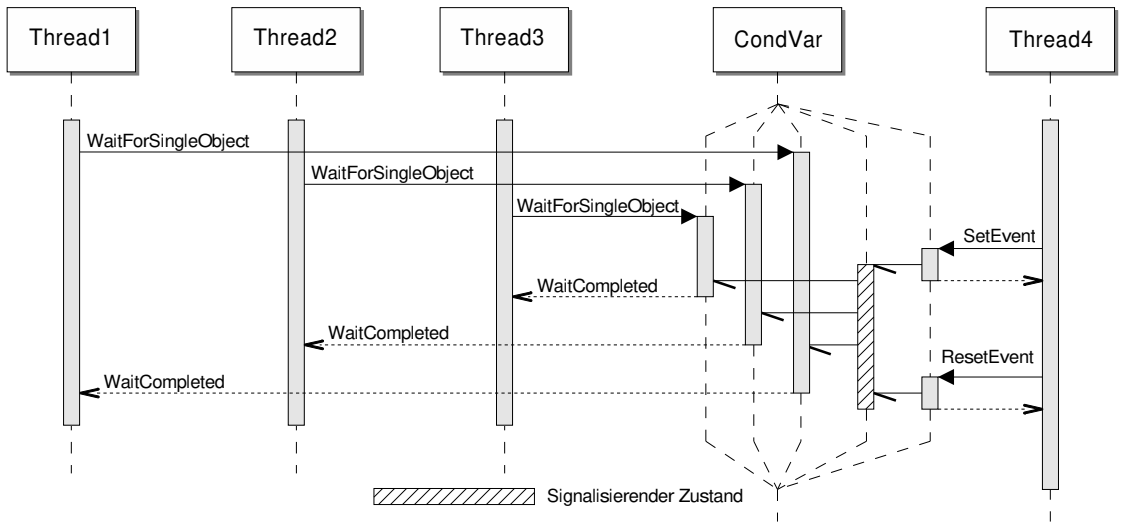


Abbildung 4.8.: Manuelle Ereignis-Objekte in der Win32 API

Bei manuellen Ereignis-Objekten stellt sich im Vergleich zu den *Broadcast*-Signalen der POSIX-Threads die Frage, wie sichergestellt werden könnte, daß alle wartenden Threads reaktiviert werden, bevor ein `ResetEvent` durchgeführt wird. Benutzt man die in Abbildung 4.8 dargestellten Funktionen `SetEvent` bzw. `ResetEvent`, ist dieses Problem nicht ohne zusätzlichen Aufwand zu lösen. Dagegen bietet sich alternativ die Funktion `PulseEvent` an, die bei der Anwendung auf manuelle Ereignis-Objekte eine entsprechend sichere Implemen-

tierung besitzt: In einer atomaren Operation werden zunächst alle wartenden Threads benachrichtigt und erst anschließend wird das Ereignis-Objekt durch einen internen Aufruf von `ResetEvent` in den nicht-signalisierenden Zustand zurücksetzt. In Abbildung 4.9 ist dieser Vorgang dargestellt.

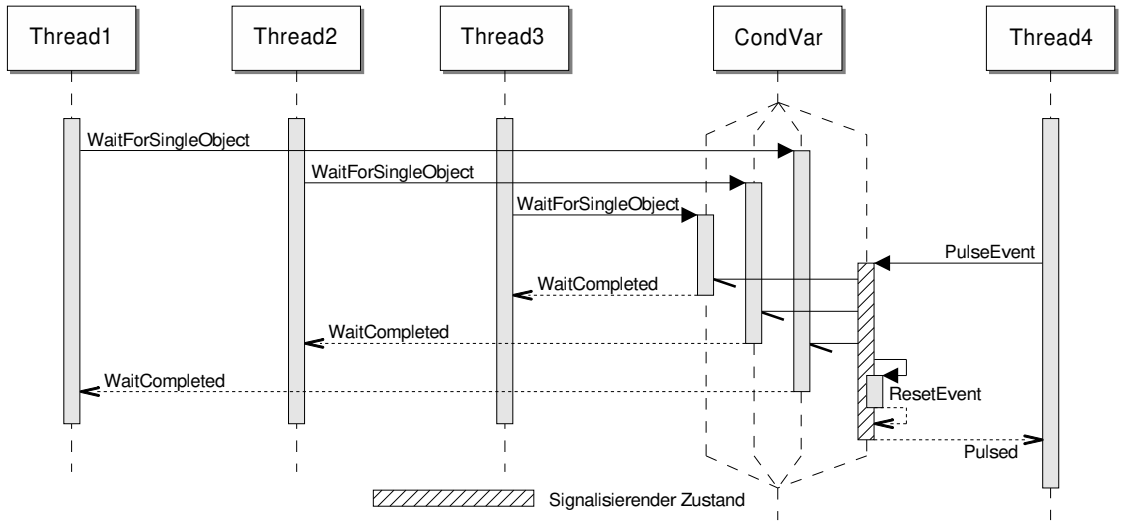


Abbildung 4.9.: Broadcast-Signal mit `PulseEvent` in der Win32 API

Zusammenfassend betrachtet können die Bedingungsvariablen der POSIX-Threads auf die Ereignis-Objekte der Win32 API auch unter Berücksichtigung der *Broadcast*-Signale abgebildet werden. Allerdings gibt es noch zwei Probleme, die dabei berücksichtigt werden müssen:

- Wie schon weiter oben erläutert wurde, muß beim Warten auf eine Bedingungsvariable zuerst der noch blockierte Mutex freigegeben werden. Hierzu kann man in der Win32 API zwar die Funktion `ReleaseMutex` benutzen, eine *atomare* Hintereinanderausführung von `ReleaseMutex` und `WaitForSingleObject` ist jedoch nicht möglich. Damit kann das Problem des *Lost wakeup* [Lewis, Berg 98] auftreten, was in Abbildung 4.10 dargestellt wird. Dabei ist ersichtlich, daß die Funktion `PulseEvent`, falls noch kein Thread wartet, das manuelle Ereignis-Objekt `CondVar` wieder unverrichteter Dinge in den nicht-signalisierenden Zustand zurücksetzt und damit die nachfolgende Anfrage von `Thread1` blockiert.

- Bei den POSIX-Threads kann dynamisch ein und dieselbe Bedingungsvariable sowohl für das einfache Signalisieren wie auch für *Broadcast*-Signale benutzt werden. In der Win32 API muß schon bei der Initialisierung eines Ereignis-Objektes festgelegt werden, ob es sich bei dem Objekt um ein manuelles oder ein automatisches Ereignis-Objekt handeln soll. Diese Eigenschaft ist anschließend nicht mehr dynamisch änderbar.

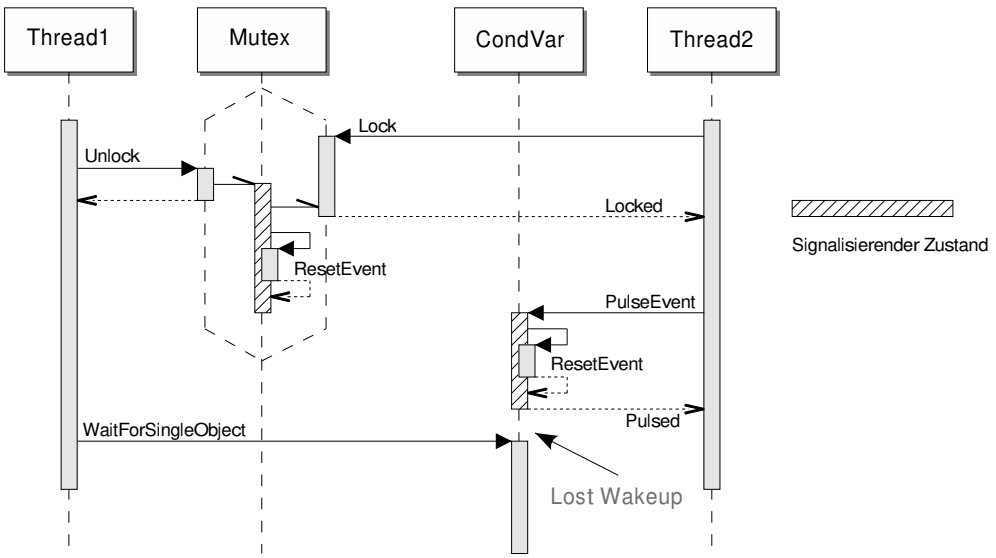


Abbildung 4.10.: Problem des *Lost wakeup* bei Bedingungsvariablen in der Win32 API

Das erste Problem läßt sich nur unter Windows NT lösen, da es hier in der Win32 API die Funktion `SignalObjectAndWait` gibt, die in einem atomaren Vorgang zunächst den Mutex freigeben und anschließend auf die Signalisierung des Ereignis-Objektes warten kann. Unter Windows 95 und Windows 98 ist diese Funktion zwar in der Schnittstelle zur Win32 API vorhanden, aber nicht implementiert – ein Aufruf dieser Funktion produziert lediglich einen Fehler.[5]

Bei der Realisierung von TycoonOS wird unter Windows NT die sichere Implementierung benutzt, bei Windows 95 und Windows 98 wird auf die unsichere Variante mit Ausgabe einer Warnung zurückgegriffen. Im Zusammenhang mit dieser Warnung sind bei der Speicherbereinigung des Tycoon-2-Systems in der Vergangenheit öfter Fehler und Programmabstürze aufgetreten, ein stabiler Betrieb ist unter diesen Betriebssystemen derzeit nur bei deaktivierter Speicherbereinigung und einem gleichzeitigen Verzicht auf Multithreading möglich. Zu beachten ist, daß das Problem des *Lost wakeup* weder in der Netscape portable runtime (NSPR) noch bei den (relativ jungen) $\text{Pthreads}_{\text{Win32}}$ [Cygnus 99b] gelöst wird. Beide Bibliotheken greifen auch unter Windows NT nicht auf eine sichere Implementierung mittels `SignalObjectAndWait` zurück.

Das zweite oben beschriebene Problem tritt im Tycoon-2-System nicht auf: Schon im Entwurf der virtuellen Maschine [Weikard 98] wurde von Anfang an eine strikte Trennung zwischen Bedingungsvariablen für einfache Signale und für *Broadcast*-Signale vorgesehen. Um diese Trennung auch auf Ebene der Hochsprache TL-2 durchzusetzen, wurden unter-

[5] Als Grund für diesen Mangel findet man in einer technischen Abhandlung (*Emulating Operating System Synchronization in Win32 Applications*) in der MSDN [MSDN 99] den Hinweis, daß diese Funktion schwer zu implementieren war.

schiedliche eingebaute Methoden (*Builtins*) vorgesehen, so daß über die entsprechenden TL-2-Klassen `Condition` bzw. `BroadcastingCondition` (Abschnitt 5.1.1) eine Mischung nicht mehr möglich ist. Allgemein verwendbare Systembibliotheken wie die NSPR oder die $\text{Pthreads}_{\text{Win32}}$ müssen natürlich mit einer solchen Mischung rechnen – hier wird das *Broadcasting* mit Hilfe von Semaphoren realisiert. Dabei muß unter anderem bei jeder Bedingungsvariablen über die aktuelle Anzahl wartender Threads intern Buch geführt werden.

In der Schnittstelle von TycoonOS zeigt sich die strikte Trennung zwischen Bedingungsvariablen für einfache Signale und *Broadcast*-Signale durch folgende Typdefinition:

```
#ifdef rt_LIB_Win32_i386
  #define tosCondition_T HANDLE
#else
  #define tosCondition_T pthread_cond_t
#endif

/*
 *  Single    <: T
 *  Broadcast <: T
 */
typedef tosCondition_T tosCondition_Single;
typedef tosCondition_T tosCondition_Broadcast;
```

Mit diesen Datentypen können unterschiedliche Funktionen zum Anlegen von Bedingungsvariablen definiert werden; das Löschen von Bedingungsvariablen kann dagegen mit ein und derselben Funktion erfolgen:

```
extern int tosCondition_initSingle   (tosCondition_Single    *cond);
extern int tosCondition_initBroadcast(tosCondition_Broadcast *cond);

extern int tosCondition_destroy(tosCondition_T *cond);
```

Für das Signalisieren auf Bedingungsvariablen können schließlich die jeweils gültigen Datentypen in den Parametern der entsprechenden Funktionen vorgegeben werden. Da TycoonOS mit Hilfe der Programmiersprache C realisiert worden ist, kann eine mutwillige Umgehung dieser Trennung allerdings nicht verhindert werden.

```
extern int tosCondition_signal   (tosCondition_Single    *cond);
extern int tosCondition_broadcast(tosCondition_Broadcast *cond);

extern int tosCondition_wait(tosCondition_T *cond, tosMutex_T *mx);

extern int tosCondition_timedwait(tosCondition_T *cond,
                                  tosMutex_T     *mx,
                                  Long            mseconds);
```

Abschließend sind in Tabelle 4.9 die unterschiedlichen Implementierungen dieser Funktionen bei den POSIX-Threads und in der Win32 API dargestellt.

Funktion	**Pthreads**	**Win32 API**
Anlegen (einfach)	`pthread_cond_init`	`CreateEvent(automatic)`
Anlegen (broadcast)	`pthread_cond_init`	`CreateEvent(manual)`
Freigeben	`pthread_cond_destroy`	`CloseHandle`
Signalisieren (einfach)	`pthread_cond_signal`	`PulseEvent`
Boadcasting	`pthread_cond_broadcast`	`PulseEvent`
Auf Freigabe warten	`pthread_cond_wait`	`ReleaseMutex`, `WaitForSingleObject` (Win98) `SignalObjectAndWait` (NT)
Verfügbarkeit prüfen	`pthread_cond_timedwait`	`ReleaseMutex`, `WaitForSingleObject` (Win98) `SignalObjectAndWait` (NT)

Tabelle 4.9.: Bedingungsvariablen bei den Pthreads und in der Win32 API

Nach dieser ausführlichen Diskussion um die Mechanismen für die Synchronistion werden aus den TycoonOS-Schnittstellen für Prozesse und Threads jetzt noch einige Funktionen für Prozesse und dynamische Bibliotheken vorgestellt.

4.6.5. Prozesse

Im Prozeßmodell zwischen UNIX und Windows NT gibt es einige erhebliche Unterschiede, beispielsweise spannen die Vater- bzw. Kindprozesse unter UNIX **Prozeßhierarchien** auf, welche es in dieser Form unter Windows NT nicht gibt. Mit seiner threadbasierten Architektur macht das Tycoon-2-System allerdings auch keinen unmittelbaren Gebrauch von diesen Prozeßmodellen, lediglich zum Starten und Stoppen externer Werkzeuge werden einige Funktionen benötigt, die in TycoonOS über die Schnittstelle `tosProcess.h` angeboten werden. In Tabelle 4.10 sind die verschiedenen Funktionen dieser Schnittstelle aufgeführt.

Funktion	**UNIX**	**Win32 API / ANSI C**
Prozeß-ID lesen	`getpid`	`_getpid`
Eltern-Prozeß-ID lesen	`getppid`	–
Verbrauchte CPU-Zeit lesen	`times`	`GetProcessTimes`
Prozeß starten	`system`	`system`
Prozeß abbrechen	`kill`	`TerminateProcess`
Signal senden	`kill`	–

Tabelle 4.10.: Funktionen für Prozesse bei UNIX und in der Win32 API

Neben den Funktionen zum Starten und Stoppen von Prozessen gibt es auch einige Funktionen zum Abfragen verschiedener Informationen zu einem Prozeß, wie beispielsweise seiner aktuell verbrauchten Prozessorzeiten im Benutzer- bzw. Kernmodus (auch UNIX kennt die in Abschnitt 4.2 vorgestellte Trennung zwischen dem Benutzer- und dem Kernmodus). Vor Aufruf der ANSI-C Standardfunktion `system` zum Starten eines Prozesses wird in der TycoonOS-Funktion zusätzlich auf den Programmnamen die in Abschnitt 4.7.1 beschriebene Funktion zur Normalisierung von Dateinamen angewendet.

Auffällig sind in Tabelle 4.10 zwei Funktionen, die unter Windows nicht zur Verfügung stehen: Die Funktionen zum Lesen der Identifikation des Elternprozesses und zum Senden von **Signalen**. Mit der letzteren können verschiedene, im POSIX-Standard spezifizierte Unterbrechungsanforderungen an Prozesse gesendet werden, sie bilden unter UNIX einen wichtigen Mechanismus zur Prozeßkommunikation. Auch das Stoppen von Prozessen erfolgt über solche Signale (z.B. `SIGKILL`), woher auch der Name der Funktion zum Senden eines Signals stammt (`kill`).

Im Tycoon-2-System werden solche systemspezifischen Mechanismen für die Realisierung des Tycoon-2 WEB-Servers benötigt, daher sind diese Funktionen auch in der Schnittstelle von TycoonOS aufgenommen worden. Unter den Windows-Betriebssystemen stehen diese Mechanismen allerdings nicht zur Verfügung, auch unter Linux ist eine Verwendung der benutzerdefinierten Signale (`SIGUSR1` bzw. `SIGUSR2`) im Zusammenhang mit der dortigen Implementierung der POSIX-Threads nicht möglich [LinuxThreads 99].

4.6.6. Dynamische Bibliotheken

Im Zusammenhang mit dem am Anfang von Kapitel 3 vorgestellten Konzept der Modularisierung von Software spielen die **dynamischen Bibliotheken** eine wichtige Rolle: Sie erlauben die Bereitstellung von Ressourcen und Funktionalitäten für mehrere Anwendungen, die sich damit die gleichen Ressourcen bzw. den gleichen Objektcode teilen können. Alle modernen Betriebssysteme unterstützen entsprechende Konzepte für dynamische Bibliotheken, wie beispielsweise mit den *Shared libraries* unter UNIX oder den *Dynamic link libraries* (DLL's) unter den Windows-Betriebssystemen.

In den Ausführungsmodellen der hier betrachteten Betriebssysteme werden dynamische Bibliotheken immer in den Adreßraum des aufrufenden Prozesses geladen, was unter anderem zur Folge hat, daß jeder Prozeß eine eigene Kopie eventueller globaler Daten der dynamischen Bibliothek erhält. Darüberhinaus wird auch meist die Möglichkeit angeboten, gemeinsamen Speicher (*Shared memory*) für die dynamische Bibliothek zu definieren, der von allen Anwendungen, die diese Bibliothek dynamisch laden, gleichzeitig genutzt werden kann. Dabei sind allerdings entsprechende Synchronisationsmaßnahmen für den Zugriff auf diesen gemeinsamen Speicher vorzusehen.

Sowohl die portable Systembibliothek TycoonOS wie auch die virtuelle Maschine des Tycoon-2-Systems stellen ihren Objektcode in Form dynamischer Bibliotheken zur Verfügung (`libtos` bzw. `libtm2`). Für die Programmierung und Übersetzung solcher dynamischen Bibliotheken müssen verschiedene, betriebssystemabhängige Faktoren berücksichtigt werden:

- Je nach Betriebssystem gibt es unterschiedliche Dateiendungen für dynamische Bibliotheken, so beispielsweise `.dll` bei den Windows-Betriebssystemen und `.so` unter UNIX.
- Für die Erzeugung dynamischer Bibliotheken sind unterschiedliche, betriebssystemabhängige Übersetzungsaktionen notwendig.
- Im Quelltext für eine dynamische Bibliothek muß spezifiziert werden, welche Bezeichner (Funktionen bzw. globale Variablen) nach außen sichtbar sein sollen. Während dies unter UNIX automatisch für alle öffentlichen Bezeichner im Quelltext geschieht, müssen unter den Windows-Betriebssystemen alle zu exportierenden Bezeichner explizit spezifiziert werden. Dies kann entweder unmittelbar bei der Definition eines solchen Bezeichners, über die Angabe eines vom jeweiligen Übersetzer abhängigen Schlüsselwortes (z.B. `_declspec(dllexport)` bei Microsoft Visual C++) oder über eine separate **Modul-Definitionsdatei** erfolgen.

Die ersten beiden Probleme wurden für diese Arbeit im Rahmen der in Kapitel 3 beschriebenen portablen Übersetzungssteuerung gelöst. Für den Export aller öffentlichen Bezeichner unter den Windows-Betriebssystemen wurde in TycoonOS der Weg über eine Modul-Definitionsdatei gewählt, da dies die hierfür übliche Methode bei diesen Betriebssystemen ist. Zudem ist diese Lösung unabhängig vom jeweiligen Übersetzer und könnte auch für eine zukünftige Portierung nach OS/2 benutzt werden. Im folgenden ist ein kleiner Ausschnitt aus der Modul-Definitionsdatei für TycoonOS dargestellt:

```
LIBRARY "libtos"
DESCRIPTION "Portable Tycoon-2 operating system layer (TycoonOS)"
EXPORTS

tos_init
tosLog_redirectError
tosLog_redirectDebug
...
```

Bei dieser Lösung mit Hilfe einer Modul-Definitionsdatei sind allerdings auch einige Dinge zu beachten:

- Es gibt eine neue Abhängigkeit der dynamischen Bibliothek von ihrer Modul-Definitionsdatei in der Übersetzungssteuerung, welche nur bei den Windows-Betriebssystemen beachtet werden muß: Wird die Modul-Definitionsdatei geändert, muß die dynamische Bibliothek neu erzeugt werden. Diese Abhängigkeit konnte im Tycoon-2-System mit Hilfe von Jam über eine entsprechende Erweiterung der Übersetzungsregeln gelöst werden (Kapitel 3).
- Es kann Funktionen geben, die nur unter bestimmten Bedingungen definiert werden, ansonsten aber undefiniert bleiben, z.B. spezielle Funktionen zur Fehlerkontrolle. Im

Tycoon-2-System wird bei einer solchen *bedingten Übersetzung* für jeden Zweig der Bedingung eine entsprechende Funktionsdefinition vorgesehen, eventuell nur ein Funktionsrumpf ohne weitere Anweisungen. Dies tritt beispielsweise bei den optionalen Funktionen zur Leistungsmessung in der Schnittstelle `tmprofile.h` auf. Gegebenenfalls müssen in solchen Situationen mehrere Modul-Definitionsdateien bereitgestellt werden, von der Übersetzungssteuerung muß dann immer die jeweils korrekte Datei ausgewählt werden.

Für die Bindung von Funktionen aus dynamischen Bibliotheken zur Laufzeit eines Programms werden auf allen Betriebssystemen entsprechende Funktionen angeboten. Hierzu gehören das Öffnen und Schließen einer dynamischen Bibliothek sowie die Beschaffung einer Referenz auf die zu importierende Funktion aus einer geöffneten dynamischen Bibliothek. Diese Funktionen werden in TycoonOS über die Schnittstelle `tosDll.h` angeboten, eine Übersicht über die Implementierung dieser Funktionen auf den verschiedenen Betriebssystemen wird in Tabelle 4.11 dargestellt.

Funktion	**Solaris/Linux**	**HP-UX**	**Win32 API**
Bibliothek öffnen	`dlopen`	`shl_load`	`LoadLibrary`
Bibliothek schließen	`dlclose`	`shl_unload`	`FreeLibrary`
Funktionsadresse laden	`dlsym`	`shl_findsym`	`GetProcAddress`
Fehlertext beschaffen	`dlerror`	–	`FormatMessage`

Tabelle 4.11.: Funktionen für dynamische Bibliotheken bei UNIX und der Win32 API

Aus dieser Tabelle ist ersichtlich, daß es unter den verschiedenen Versionen der UNIX-Betriebssysteme unterschiedliche Funktionen für dynamische Bibliotheken gibt. Dies liegt daran, daß diese Funktionen nicht standardisiert sind (weder POSIX noch ANSI), allerdings haben sich die beiden Lösungen für Solaris und HP-UX weitgehend etabliert. Beide Varianten werden auch in TycoonOS unterstützt, lediglich für die Beschaffung des Fehlertextes steht unter HP-UX keine entsprechende Funktion zur Verfügung. Die Funktion zum Öffnen einer dynamischen Bibliothek wendet auf den Namen dieser Bibliothek die in Abschnitt 4.7.1 beschriebene Funktion zur Normalisierung von Dateinamen an. Die systemspezifische Dateiendung für den Bibliotheksnamen wird nach dieser Normalisierung innerhalb von TycoonOS automatisch angepaßt.

Nach dieser umfangreichen Darstellung der portablen Schnittstellen für Threads, Synchronisation, Prozesse und dynamischen Bibliotheken werden im folgenden die Schnittstellen für die Handhabung von Dateien und für Verzeichnisse vorgestellt.

4.7. Dateien und Verzeichnisse

Die in Abbildung 4.11 dargestellten Schnittstellen für Datei- und Verzeichnisdienste umfassen einige elementare Funktionen für Dateien (`tosFile.h`) und zur Handhabung von

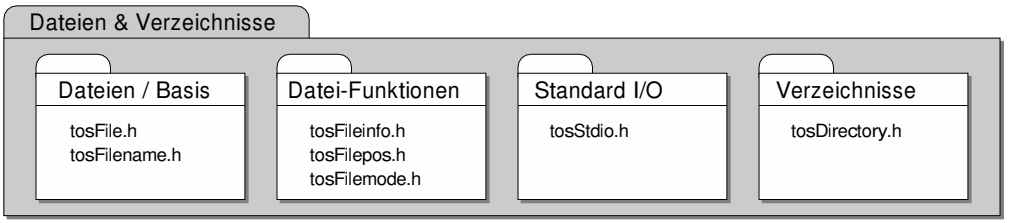


Abbildung 4.11.: Datei- und Verzeichnisdienste von TycoonOS

Dateinamen (`tosFilename.h`) sowie Funktionen zur Abfrage und Veränderung verschiedener Dateiattribute (`tosFileinfo.h`, `tosFilepos.h` und `tosFilemode.h`). Schließlich gibt es noch mit `tosDirectory.h` eine Schnittstelle mit Funktionen zum Anlegen, Löschen und Durchsuchen von Dateiverzeichnissen.

4.7.1. Dateifunktionen

Über die Schnittstelle `tosFile.h` werden grundlegende Funktionen für Dateien angeboten, etwa für das Öffnen und Schließen von Dateien oder für elementare Ein- bzw. Ausgabeoperationen. Neben verschiedenen Funktionen aus dem ANSI C Standard für diese Operationen gibt es einige kleinere Differenzen zwischen den Betriebssystemen, beispielsweise unterschiedliche Funktionsnamen für die Synchronisation von Dateien mit ihrem Dateipuffer (`_commit` bzw. `fsync`) oder auch unterschiedliche Optionen für das Öffnen von Dateien. Tabelle 4.12 zeigt die wichtigsten Funktionen dieser Schnittstelle im Überblick.

Funktion	**UNIX**	**Win32 API / ANSI C**
Datei öffnen	`open`	`open`
Datei schließen	`close`	`close`
Datei synchronisieren	`fsync`	`_commit`
Datei-Informationen abfragen	`stat`	`_stat`
Datei kopieren	–	`CopyFile`
Datei verschieben	`rename`	`MoveFileEx`
Datei löschen	`remove`	`remove`
Symbolischen Link erzeugen	`symlink`	–

Tabelle 4.12.: Verschiedene Dateifunktionen bei UNIX und in der Win32 API

Für das Kopieren von Dateien gibt es unter UNIX keine Systemfunktion, hier wurde eine entsprechende Implementierung innerhalb von TycoonOS vorgenommen. Um die Semantik der Funktion `rename` zum Verschieben von Dateien unter der Win32 API nachzubilden, ist eine Benutzung der Funktion `MoveFileEx` notwendig, da nur diese die Option anbietet, eine schon existierende Zieldatei zu ersetzen und um ein Verschieben der Datei über Laufwerksgrenzen hinweg zu erlauben. Diese Funktion steht nur in der Win32 API von Windows NT zur

Verfügung, unter Windows 95 bzw. Windows 98 muß mit einer Kombination von `DeleteFile` und `MoveFile` gearbeitet werden, womit die Semantik von `rename` allerdings nicht vollständig nachgebildet werden kann (kein Verschieben über Laufwerksgrenzen hinweg).

Die Funktionen für *symbolische Links* stehen nur unter UNIX zur Verfügung, hier wurde bei den Windows-Betriebssystemen keine entsprechende Implementierung vorgenommen (z.B. mit den sogenannten *Dateiverknüpfungen*), da in zukünftigen Versionen von Windows ein besseres Konzept direkt auf Dateisystemebene angeboten werden soll.

In der Schnittstelle `tosFilename.h` werden verschiedene Funktionen für die systemunabhängige Komposition bzw. Dekomposition von Dateinamen zur Verfügung gestellt. Diese Funktionen ermöglichen beispielsweise das einfache Ersetzen von systemabhängigen Dateiendungen (`tosFilename_setExtension`), wie es bei den dynamischen Bibliotheken notwendig ist (Abschnitt 4.6.6). Eine wichtige Funktion dieser Schnittstelle ist die **Normalisierung** von Dateinamen, welche durch folgende Anpassungen an das auf dem jeweiligen Betriebssystem gültige Dateinamensformat realisiert wird:

- Alle Trennzeichen für Pfade (\ unter Windows bzw. / unter UNIX) müssen systemspezifisch angepaßt werden.
- Laufwerksbuchstaben dürfen nur unter den Windows-Betriebssystemen benutzt werden, unter UNIX werden diese Laufwerksbuchstaben durch TycoonOS auf einfache Unterverzeichnisnamen abgebildet.

Weitere semantische Interpretationen von Dateinamen, wie sie beispielsweise in [Korn 97; AT&T 99] vorgenommen werden (Interpretation spezieller Datei- und Gerätenamen), finden nicht statt – die Normalisierung von Dateinamen sollte an dieser Stelle zunächst so einfach wie möglich gehalten werden. Tabelle 4.13 zeigt einige Beispiele für die Normalisierung von Dateinamen in TycoonOS.

UNIX-Dateiname	**Windows-Dateiname**
`test/hallo.c`	`test\hallo.c`
`/usr/bin/ls`	`\usr\bin\ls`
`/c/windows/notepad.exe`	`\c\windows\notepad.exe`
`/c//windows/notepad.exe`	`c:\windows\notepad.exe`

Tabelle 4.13.: Verschiedene Formate von Dateinamen unter UNIX und Windows

Die Normalisierung von Dateinamen kann entweder explizit mit Hilfe der Funktionen `tosFilename_toUnix` und `tosFilename_toWin` vorgenommen werden, oder implizit für das aktuelle Wirtsbetriebssystem über die Funktion `tosFilename_normalize`. Eine Übersicht über alle weiteren Funktionen der Schnittstelle `tosFilename.h` findet man im Anhang A.

Weitere Funktionen für Dateien werden in den Schnittstellen `tosFileinfo.h` für die Abfrage verschiedener Datei-Informationen, wie z.B. für das letzte Änderungsdatum einer Datei, `tosFilemode.h` für das Setzen verschiedener Zugriffsmodi für Dateien (soweit dies

in einer übertragbaren Form möglich ist) und `tosFilepos.h` für die portable Handhabung des Dateipositions-Zeigers für Ein- und Ausgabeoperationen zur Verfügung gestellt. Alle drei Module für diese Schnittstellen konnten mit nur geringen Differenzen zwischen den Betriebssystemen realisiert werden, weshalb sie an dieser Stelle nicht detaillierter vorgestellt werden. Erwähnenswert ist lediglich noch die Schnittstelle `tosStdio.h`, die als Erweiterung der ANSI C Schnittstelle `stdio.h` folgende Aufgaben erfüllt:

- Es werden standardisierte Schalter für das Öffnen von Dateien angeboten. Dies ist unbedingt notwendig, da beim Öffnen einer Datei ohne Angabe solcher Schalter unter UNIX und Windows die Dateien in unterschiedlichen Modi geöffnet werden (binär bzw. textuell), was zu erheblichen Problemen führen kann (Interpretation von Zeilenvorschüben in binären Dateien). Durch den ANSI C Standard wird dieses Verhalten beim Öffnen von Dateien leider nicht festgelegt.

- Für den beim Öffnen einer Datei angegebenen Dateinamen findet die weiter oben beschriebene Normalisierung von Dateinamen statt.

- Die nicht standardisierte Funktion `ftruncate` zum destruktiven Setzen des Dateipositions-Zeigers (alles nach dem Zeiger wird gelöscht) wird über diese Schnittstelle angeboten, da sie mit den gleichen Datei-Handle arbeitet wie die anderen Funktionen aus `stdio.h`. Im Tycoon-2-System wird diese Funktion beim Sichern eines Objektspeicherzustandes benötigt. Unter Windows ist die Implementierung dieser Funktion nicht ganz trivial, da hierfür die Win32 API Funktion `SetEndOfFile` benutzt werden muß und dabei vorher der Datei-Handle der C-Laufzeitbibliothek in einen betriebssystemnahen Handle konvertiert werden muß.

4.7.2. Verzeichnisfunktionen

Nach der Betrachtung der für das Tycoon-2-System wichtigen Dateifunktionen werden jetzt noch einige Funktionen zum Anlegen, Löschen und Durchsuchen von Dateiverzeichnissen vorgestellt. In Tabelle 4.14 werden diese über die Schnittstelle `tosDirectory.h` angebotenen Funktionen in der Übersicht dargestellt.

Funktion	**UNIX**	**Win32 API / ANSI C**
Vollständigen Dateipfad bestimmen	`realpath`	`_fullpath`
Wurzelverzeichnis wechseln	`chroot`	–
Verzeichnis anlegen	`mkdir`	`_mkdir`
Verzeichnis löschen	`rmdir`	`_rmdir`
Dateisuche vorbereiten	`opendir`	–
Datei suchen	`readdir`	`FindFirstFile`, `FindNextFile`
Dateisuche abschließen	`closedir`	`FindClose`

Tabelle 4.14.: Verzeichnisfunktionen von UNIX und der Win32 API

Wie aus dieser Tabelle ersichtlich ist, gibt es neben einigen kleineren Differenzen, wie beispielsweise dem Fehlen der Funktion `chroot` unter den Windows-Systemen (diese Funktion wird aber auch im Tycoon-2-System kaum genutzt), vor allem Unterschiede beim Suchen von Dateien in Verzeichnissen:

- Unter den UNIX-Betriebssystemen werden Funktionen aus dem POSIX-Standard benutzt, die die Ergebnismenge einer solchen Suche wie eine Datei behandeln, die ausgelesen werden kann. Ein solcher **Ausgabekanal** wird unter Angabe der Suchparameter geöffnet (`opendir`), so daß anschließend mit jeder Leseoperation auf diesem Kanal die nächste gefundene Datei ausgelesen werden kann (`readdir`). Falls keine weitere Datei mehr gefunden wird, kann der Kanal wieder geschlossen werden (`closedir`).
- Bei der Win32 API wird die Dateisuche über ein Funktionspaar abgewickelt: Zuerst wird das erste Auftreten der gesuchten Datei ermittelt (`FindFirstFile`), wobei alle Suchparameter angegeben werden; jedes weitere Auftreten der gesuchten Datei kann dann immer auf Basis der zuletzt gefundenen Datei ermittelt werden (`FindNextFile`). Wird keine weitere Datei mehr gefunden, können alle Ressourcen mittels `FindClose` wieder freigegeben werden.

In TycoonOS wird für die Suche von Dateien über Verzeichnisse der Weg über einen Ausgabekanal gewählt, um möglichst dicht an den bisher verwendeten Suchalgorithmen für Dateien z.B. im *Classloader* zu bleiben. Unter den Windows-Betriebssystemen muß dabei die Lesefunktion in der Lage sein, zwischen dem Ersten und den weiteren Folgezugriffen zu unterscheiden. Dies geschieht aufgrund folgender Informationen, die TycoonOS über die Schnittstelle `tosDirectory.h` anbietet und mit denen auch die Dateisuche gegenüber POSIX etwas vereinfacht wird (Angabe des Paramters `lastEntry` bei der Funktion `readdir` entfällt):

WIN32 API:

```
#include <windows.h>
typedef struct
{
  char   searchPath[_MAX_PATH];
  HANDLE searchHandle;
  WIN32_FIND_DATA lastEntry;
}
tosDirectory_T;
```

POSIX:

```
#include <dirent.h>
typedef struct
{
  char   searchPath[MAXNAMLEN];
  DIR   *searchHandle;
  struct dirent lastEntry;
}
tosDirectory_T;
```

Die Schnittstelle der so über den Datentyp `tosDirectory_T` vereinfachten Funktionen in `tosDirectory.h` wird im folgenden dargestellt, die Funktion `tosDirectory_sizeOfT` gibt dabei die Größe des oben angegebenen Datentypen zurück, da die Werte dieses Typen dynamisch angelegt werden müssen:

```
extern int   tosDirectory_sizeOfT(void);
```

```
extern int   tosDirectory_open (tosDirectory_T *pDir, char *dirName);
extern char *tosDirectory_read (tosDirectory_T *pDir);
extern int   tosDirectory_close(tosDirectory_T *pDir);
```

Nach der Betrachtung der Datei- und Verzeichnisdienste erfolgt jetzt noch als Abschluß dieses Kapitels die Vorstellung der Schnittstellen für Netzwerk- und Sicherheitsdienste sowie weiterer Werkzeuge der portablen Systembibliothek TycoonOS.

4.8. Netzwerkdienste und weitere Werkzeuge

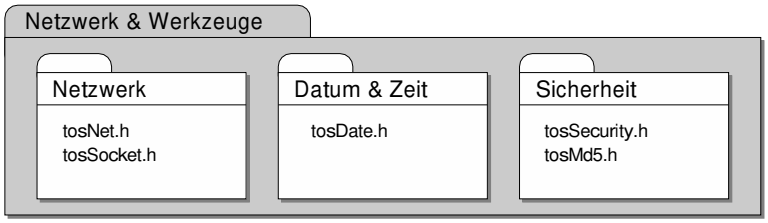


Abbildung 4.12.: Netzwerkdienste und weitere Werkzeuge von TycoonOS

Abbildung 4.12 zeigt die in diesem Kapitel vorgestellten Schnittstellen von TycoonOS für Netzwerkdienste (`tosNet.h` und `tosSocket.h`) und für verschiedene Werkzeuge wie Datums- und Zeitfunktionen (`tosDate.h`), Dateisignierung (`tosMd5.h`) und elementare Sicherheitsdienste (`tosSecurity.h`).

4.8.1. Netzwerkdienste

In dem Bereich der Netzwerkdienste bietet TycoonOS zwei Schnittstellen an, die auf dem weit verbreiteten Standard von Berkeley für TCP/IP-Netzwerke basieren [Comer, Stevens 94; Comer 95]. Hierzu gehören `tosNet.h` mit einigen Funktionen zur Abfrage der **Netzwerk-Datenbank** und `tosSocket.h` mit Funktionen für die netzwerkweite Kommunikation über **Sockets**. Während die erste Schnittstelle Funktionen zur Bestimmung von Rechnernamen und Netzwerkadressen anbietet, erlauben die Sockets der zweiten Schnittstelle als **Kommunikationsendpunkte** eine verbindungsorientierte (*Network stream socket*) oder auch eine verbindungslose (*Datagram socket*) Kommunikation über ein TCP/IP-Netzwerk. Beide Schnittstellen basieren auf der Version 3.4 der *Berkely Software Distribution* (3.4BSD) [Leffler et al. 89].

Auf den Windows-Betriebssystemen wird der Berkeley-Standard für TCP/IP-Netzwerke über die Schnittstelle zu der WinSocket-Bibliothek erfüllt, so daß alle Funktionen der beiden TycoonOS-Schnittstellen `tosNet.h` und `tosSocket.h` ohne Probleme auf diese Systeme übertragen werden konnten. Kleinere Unterschiede gibt es nur bei folgenden Details:

- Nach dem Aufruf von Funktionen aus der WinSocket-Bibliothek ist die in Abschnitt 4.5.4 beschriebene Abbildung von Fehlerinformationen (`WSAGetLastError`) auf `errno` notwendig.
- Vor einer Benutzung der WinSocket-Bibliothek durch TycoonOS muß eine besondere Initialisierung dieser Bibliothek vorgenommen werden. Dabei muß über den Aufruf der Funktion `WSAStartup` die benötigte Versionsnummer der Bibliothek geprüft werden. Für eine vollständige Konformität von Windows zum BSD-Standard ist eine WinSocket-Bibliothek mit der Version ≥ 2.0 notwendig.
- Da die WinSocket-Bibliothek als separates Subsystem nicht in das C-Laufzeitsystem integriert ist, können die C-Standardfunktionen für die Ein- und Ausgabe (`read` bzw. `write`), anders als bei UNIX, nicht für Sockets benutzt werden. Statt dessen müssen die in der WinSocket-Bibliothek angebotenen Funktionen `recv` und `send` benutzt werden.

Für eine Vertiefung der Thematik rund um die Netzwerkkommunikation muß auf die oben angegebenen Quellen verwiesen werden. An dieser Stelle folgt abschließend in Tabelle 4.15 noch eine Übersicht über die in den Schnittstellen `tosNet.h` und `tosSocket.h` angebotenen Funktionen.

Funktion	**UNIX**	**WinSocket**
Initialisierung	–	`WSAStartup`
Eigenen Rechnernamen abfragen	`gethostname`	`gethostname`
Rechneradressen abfragen	`gethostbyname`	`gethostbyname`
Socket öffnen	`socket`	`socket`
Socket schließen	`close`	`closesocket`
Socket an Adresse binden	`bind`	`bind`
Verbindungsaufnahme	`connect`	`connect`
Auf Verbindung warten	`accept`	`accept`
Daten lesen	`read`	`recv`
Daten schreiben	`write`	`send`

Tabelle 4.15.: Netzwerkfunktionen unter UNIX und Windows

4.8.2. Datums- und Zeitroutinen

Die in der Schnittstelle `tosDate.h` angebotenen Funktionen für die Ermittlung und Konvertierung von Datums- und Zeitangaben wurden im wesentlichen aus dem alten Laufzeitsystem RTS übernommen. Alle Funktionen konnten dabei auf die Windows-Betriebssysteme übertragen werden, lediglich die Funktion `tosDate_fromString` zum Lesen eines Datums aus einer formatierten Zeichenkette ist in TycoonOS nur für die UNIX-Betriebssysteme implementiert, da die hierfür benötigte Funktion `strptime` unter Windows keinen gleichwertigen Ersatz besitzt.

Tabelle 4.16 verschafft einen Überblick über die in dieser Schnittstelle angebotenen Funktionen, darüber hinaus gibt es noch eine Reihe verschiedener Funktionen zur Ermittlung spezifischer Teilinformationen aus einer Datums- bzw. Zeitangabe (Sekunde, Minute, Tag, Wochentag, Zeitzone etc. – siehe Anhang A).

Funktion	**UNIX**	**Win32 API / ANSI C**
Zeitzone bestimmen	`timezone`	`_timezone`
Zeitzonen-Name bestimmen	`tzname`	`_tzname`
Zeit in Sekunden seit 1.1.1970	`time`	`time`
Lokales Datum aus Zeitstempel	`localtime_r`	`localtime`
Lokales Datum aus Zeichenkette	`strptime`	–
Lokales Datum formatiert ausgeben	`strftime`	`strftime`

Tabelle 4.16.: Datums- und Zeitfunktionen unter UNIX und Windows

Die in dieser Schnittstelle angebotenen Funktionen für Datum und Zeit sind nicht sehr umfangreich, so fehlen Funktionen für verschiedene Datumsberechnungen im Rahmen des gregorianischen Kalenders, beispielsweise für die Bestimmung von Schaltjahren. Zwar könnten die entsprechenden Funktionen auch auf Ebene der Hochsprache TL-2 angeboten werden, für eine zukünftige Weiterentwicklung von TycoonOS wäre es aber auch zu überlegen, eine schon bestehende, möglichst umfangreiche und portable Bibliothek für Datumsberechnungen zu benutzen. Beispielsweise umfaßt die C-Bibliothek Date-Calc [DateCalc 99] umfangreiche Datumsberechnungen für den gregorianischen Kalender, einschließlich Feiertagsbestimmungen oder die Bildung von Datumsdifferenzen. Dabei ist diese Bibliothek konform zu verschiedenen Standards in der Darstellung von Datums- und Zeitangaben, wie der ISO 8601 [ISO 95] oder der entsprechenden deutschen Norm DIN EN 28601 für Zeitangaben, Kalender und Wochennummerierungen.

4.8.3. Datensignatur

Die Schnittstelle `tosMd5.h` bietet eine **Einwegfunktion** (*One way function*) für die Signierung von Daten nach dem **MD5**-Algorithmus an (*Message Digest*, [Rivest 92]). Einen guten und vor allem verständlichen Überblick über Algorithmen für die Signierung und Verschlüsselung von Daten findet man in [Kaufman et al. 94]. Im Tycoon-2-System wird die in dieser Schnittstelle angebotene Einwegfunktion für die Anfertigung eines eindeutigen *Fingerabdrucks* von den Quelltextdateien der TL-2-Klassen benutzt. Mit Hilfe solcher Signaturen kann der TL-2-Übersetzer im Rahmen der Übersetzungssteuerung alle modifizierten Quelltexte über eine veränderte Signatur erkennen.

Die zentralen Funktionen dieser Schnittstelle werden in folgender Übersicht dargestellt. Dabei wird ersichtlich, daß die Daten für die Signierung auch in mehreren Blöcken eingelesen werden können (`tosMd5_update`), wobei auch schon einige Aufbereitungen dieser Daten für den MD5-Algorithmus stattfinden. Erst nach dem Einlesen aller Daten wird die Prüfsummenbildung mit Hilfe der Funktion `tosMd5_final` durchgeführt. In der Implementierung

basiert das Modul dieser Schnittstelle auf der Referenzimplementierung aus [Rivest 92], die allerdings dem Urheberrecht der Firma RSA Data Security Inc. unterliegt.[6]

```
/* Allokation und Initialisierung eines Kontextes =========*/

extern void tosMd5_init(MD5_CTX *context);

/* Neue Daten in Kontext einpflegen =======================*/

extern void tosMd5_update(MD5_CTX       *context,
                          unsigned char *buffer,
                          unsigned int   offset,
                          unsigned int   len);

/* Prüfsumme schreiben, Kontext entleeren =================*/

extern void tosMd5_final(MD5_CTX       *context,
                         unsigned char  signature[16]);
```

4.8.4. Zugriffsrechte und Sicherheitsdienste

Eine wichtige Anforderung an ein Mehrbenutzerbetriebssystem ist die Bereitstellung von grundlegenden Sicherheitsmechanismen für die **Authentisierung** der am System teilnehmenden Benutzer und deren **Autorisierung** für bestimmte Aktivitäten, wie das Starten von Programmen oder das Bearbeiten von Dateien.

Unter der Authentisierung versteht man dabei die **Identifikation** der Benutzer, was meist durch Angabe einer Benutzerkennung und eines geheimen Paßwortes geschieht. Mit der Autorisierung werden für den Benutzer dedizierte **Zugriffsrechte** erteilt, wobei es betriebssystemabhängig eine unterschiedliche Zusammensetzung der einzelnen Rechte (Lesen, Schreiben, Ausführen etc.) und der zu schützenden Objekte (Dateien, Prozesse, Geräte etc.) gibt.

Unter UNIX wird jeder Benutzer durch eine Ganzzahl identifiziert, der `uid` (*User identification*); der Wert 0 ist dabei grundsätzlich für den Systemadministrator reserviert. Daneben gibt es Identifikatoren für Gruppen, die ebenfalls durch Ganzzahlen dargestellt werden und die eine Zuteilung von Zugriffsrechten an mehrere Benutzer erlauben (die Angehörigen der Gruppe). Jeder Prozeß erhält automatisch die `uid` des Benutzers, der ihn gestartet hat; jede Datei besitzt verschiedene Zugriffsmodi für das Lesen, Schreiben und Ausführen jeweils getrennt für den Eigentümer, die Eigentümergruppe und für alle anderen Benutzer.

Auch unter Windows NT[7] gibt es Benutzer und Benutzergruppen, die sich durch einen Identifikator ausweisen können, der *Security ID* (SID). Eine solche SID ist komplexer aufgebaut als eine `uid` unter UNIX, so enthält sie Informationen darüber, wer die jeweilige

[6] Im Gegensatz hierzu unterliegen die meisten Quelltexte des Tycoon-2-Systems der *GNU Public License*.

[7] Windows 95 und Windows 98 besitzen nur ein eingeschränktes Sicherheitssystem und bieten in der Win32 API hierfür auch keine Funktionen an. Die entsprechenden TycoonOS-Dienste stehen auf diesen Betriebssystemen daher nicht zur Verfügung.

SID eingerichtet hat. Unter Windows NT besitzt grundsätzlich jedes Betriebssystemobjekt (Benutzer, Datei, Prozeß, Thread, Gerät, Semaphore etc.) einen Zugangscode (*Access token*) – greifen zwei Objekte aufeinander zu (Benutzer auf Gerät, Thread auf Datei etc.), vergleicht ein Betriebssystemprozeß (*Security Resource Monitor*, SRM) die entsprechenden Zugangscodes miteinander und entscheidet aufgrund der dort eingetragenen Zugriffskontrollliste (*Accesss Control List*, ACL), ob dieser Zugriff erlaubt ist.

Insgesamt sind die Sicherheitsmechanismen unter Windows NT deutlich umfangreicher als unter UNIX, dabei zeigt sich unter Windows NT eine sehr orthogonale Architektur dieser Mechanismen, die zudem flexibel erweiterbar sind. So können hier im Gegensatz zu den statischen Zugriffsmodi für Prozesse und Dateien bei UNIX dynamisch beliebige Zugriffsmodi durch eine Erweiterung der ACL's definiert werden. Beispielsweise werden im U/WIN-Paket ([Korn 97; AT&T 99], siehe Abschnitt 4.3.1) über entsprechend aufgebaute Zugriffskontrollisten unter Windows NT die Zugriffsmodi von UNIX für Dateien nachgebildet. Ein Nachteil bei Windows NT besteht im Gegensatz zu UNIX darin, daß die Quelltexte des Betriebssystems nicht frei zugänglich sind und damit noch viele Fehler im verborgenen liegen, die eventuell zu neuen Sicherheitslücken führen können.

Für eine Anwendung elementarer Sicherheitsmechanismen im Tycoon-2-System stellt TycoonOS in der Schnittstelle `tosSecurity.h` einige Funktionen für die Benutzer- und Gruppenidentifikation zur Verfügung. Hierzu zählt die Ermittlung der zu den Identifikatoren gehörenden Benutzer- bzw. Gruppennamen sowie eine Erfassung und Veränderung der zum aktuell laufenden Prozeß gehörenden Identifikatoren. Auch eine Änderung der Eigentümer (Benutzer bzw. Gruppe) für Dateien ist vorgesehen. Tabelle 4.17 gibt einen Überblick über die in `tosSecurity.h` angebotenen Funktionen.

Funktion	**UNIX**	**Win32 API**
Name einer Benutzer-ID ermitteln	`getpwuid_r`	`LookupAccountSid`
Name einer Gruppen-ID ermitteln	`getgrgid_r`	`LookupAccountSid`
Benutzer-ID aus Namen ermitteln	`getpwuid_r`	`LookupAccountName`
Gruppen-ID aus Namen ermitteln	`getgrgid_r`	`LookupAccountName`
Benutzer-ID des aktuellen Prozesses ermitteln	`getuid`	`OpenProcessToken` + `GetTokenInformation`
Gruppen-ID des aktuellen Prozesses ermitteln	`getgid`	`OpenProcessToken` + `GetTokenInformation`
Aktuelle Benutzer-ID setzen	`setuid`	–
Aktuelle Gruppen-ID setzen	`setgid`	–
Eigentümer von Datei ändern	`chown`	–

Tabelle 4.17.: Sicherheitsmechanismen unter UNIX und Windows

Das Setzen einer neuen Prozeßidentifikation und die Änderung der Eigentümer einer Datei funktionieren nicht in der TycoonOS-Implementierung unter Windows NT. Auch für umfangreichere Sicherheitsdienstleistungen laufen die Unterschiede zwischen den Betriebs-

systemen zu weit auseinander, hier sollten statt dessen externe, plattformübergreifende Sicherheitsdienstleistungen eingesetzt werden, wie beispielsweise das am MIT (*Massachusetts Institute of Technology*) entwickelte **Kerberos**-System [Kohl, Neuman 93; Steiner et al. 88]. Dieses System ist unter UNIX frei verfügbar und soll auch als zentraler Bestandteil für die verteilte Zugriffskontrolle im Netzwerk in Windows 2000 integriert werden. Für die Einbindung der Sicherheitsdienstleistungen von Kerberos in Tycoon-1 gibt es schon Erfahrungen, die durchaus auch auf das Tycoon-2-System übertragbar wären [Willomat 97].

4.9. Gesamtübersicht

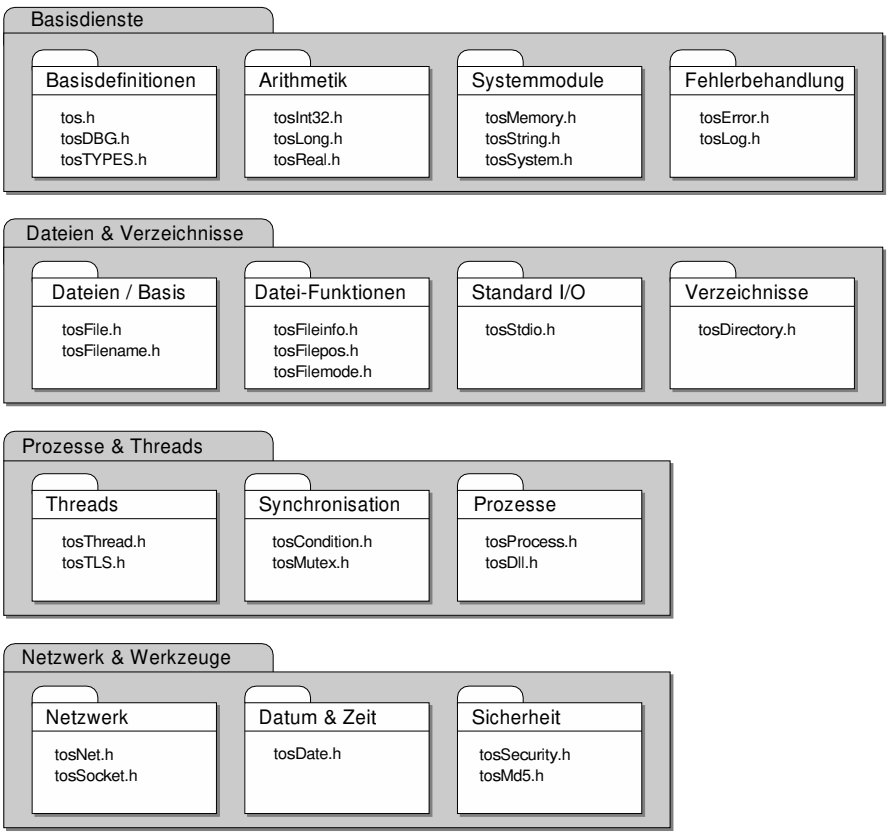


Abbildung 4.13.: Gesamtübersicht über die Dienste von TycoonOS

Abbildung 4.13 zeigt abschließend noch einmal eine Gesamtübersicht über alle hier vorgestellten Schnittstellen der portablen Systembibliothek TycoonOS. Die möglichen Erweiterungen dieser Systembibliothek wurden schon im Verlauf dieses Kapitels angedeutet und werden abschließend noch im Ausblick dieser Arbeit diskutiert.

5. Konsequenzen auf hochsprachlicher Ebene

In diesem Kapitel werden einige Auswirkungen und Konsequenzen auf Ebene der Hochsprache TL-2 betrachtet, die aus der Einführung des portablen Architekturkonzeptes für das Tycoon-2-System und der in Kapitel 4 beschriebenen übertragbaren Systembibliothek TycoonOS folgen. Dabei werden die in diesem Zusammenhang interessanten TL-2-Standardklassen für Systemdienste in der Übersicht vorgestellt.

Abschließend werden einige Betrachtungen bezüglich der Mobilität der Objektspeicherzustände von Tycoon-2 im Zusammenhang mit der Benutzung von Betriebssystemressourcen angestellt.

5.1. Systemdienste

In diesem Abschnitt werden einige Standardklassen für die Systemdienste in der Hochsprache TL-2 vorgestellt, soweit sie von der Einführung der portablen Systembibliothek TycoonOS betroffen sind. Dabei wird auch deren Anbindung an TycoonOS und die in Abschnitt 4.5.4 vorgestellte einheitliche Fehlerbehandlung betrachtet. Eine ausführliche Einführung in die Sprache TL-2 mit ihren Standardklassen kann an dieser Stelle jedoch nicht erfolgen, hierfür sei auf [Wahlen 98; Wienberg 97] verwiesen.

5.1.1. Threads und Threadsynchronisation

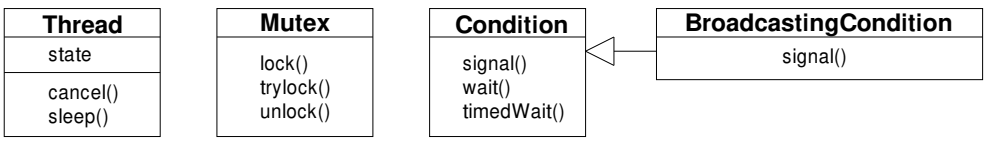


Abbildung 5.1.: TL-2-Klassen für Threads und Threadsynchronisation

In Abbildung 5.1 sind die TL-2-Klassen für Threads und verschiedene Synchronisationsmechanismen für Threads dargestellt. Dabei gibt es Klassen für die schon in Abschnitt 4.6 vorgestellten Mechanismen des gegenseitigen Ausschlusses (Mutex) und der Bedingungsvariablen. Auch die in Abschnitt 4.6 vorgestellten Funktionen für Threads und deren Synchronisation sind in Abbildung 5.1 dargestellt. Diese Methoden basieren auf eingebaute Metho-

den (*Builtins*) der virtuellen Maschine, die dort mit Hilfe der entsprechenden TycoonOS-Funktionen realisiert worden sind.

Schon in Abschnitt 4.6.4 wurde die Notwendigkeit einer strikten Trennung zwischen einfachen Signalen und *Broadcast*-Signalen mit Blick auf das Threadmodell von Windows NT beschrieben. An dieser Stelle wird eine solche Trennung über die Einführung der separaten TL-2-Klassen `Condition` und `BroadcastingCondition` erreicht: Beide Klassen besitzen eine `signal`-Methode, wobei diese in der Klasse `Condition` als einfaches Signal und in der Klasse `BroadcastingCondition` als *Broadcast*-Signal implementiert ist. So benutzen die Exemplare beider Klassen immer nur die jeweils korrekte Form der Signalisierung, eine Vermischung ist nicht möglich.

5.1.2. Dateisystemklassen

Das Tycoon-2-System besitzt in der TL-2-Standardbibliothek umfangreiche Klassen mit Ein- und Ausgabeoperationen für Dateien. Eine zentrale Position nimmt dabei die Klasse `File` ein, die viele der in Abschnitt 4.7 beschriebenen Dateifunktionen kapselt und auch für das Öffnen und Schließen von Dateien zuständig ist. Darauf aufbauend gibt es diverse Klassen, wie beispielsweise `Writer`, `Output` oder `BufferedOutput` für Ausgabeoperationen auf Dateien; diese Klassen sind hier allerdings nicht weiter von Interesse.

Von größerer Bedeutung im Zusammenhang mit dieser Arbeit sind dagegen die in Abbildung 5.2 dargestellten Klassen für eine systemunabhängige Handhabung von Dateinamen. Über diese Klassen wird festgelegt, wie von TL-2 aus auf ein Dateisystem zugegriffen werden kann, also wie Dateinamen aufgebaut oder einzelne Komponenten dieser Dateinamen (Pfade oder Suffixe von Dateinamen) verändert werden können.

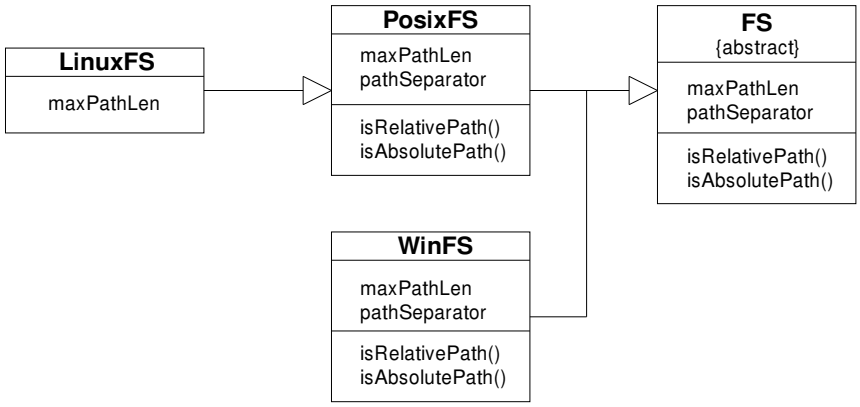


Abbildung 5.2.: TL-2-Klassen für den Zugriff auf das Dateisystem

An zentraler Stelle steht dabei die abstrakte Klasse `FS`, über die die TL-2-Schnittstelle für den Zugriff auf ein Dateisystem spezifiziert wird. Zu dieser Klasse gehören Felder wie `maxPathLen` mit einem Wert für die maximale Dateinamenslänge, oder `pathArcSeparator`

mit einer Angabe über die schon in Abschnitt 4.7.1 beschriebenen Trennzeichen für Pfade bei Dateinamen. Weiterhin werden hier die Schnittstellen verschiedener Methoden definiert, beispielsweise zum Suchen von Dateien (`search`), zur Zusammensetzung von Dateinamen (`composePath`) oder für die Abfrage verschiedener Informationen zu einem Dateinamen (`isValidPath`, `isAbsolutePath`, `isRelativePath` etc.). Eine Implementierung dieser Zugriffsmethoden findet in den jeweils von `FS` abgeleiteten konkreten Klassen statt:

- Für die UNIX-Betriebssysteme werden in der von `FS` abgeleiteten Klasse `PosixFS` die nach dem POSIX-Standard [ISO/IEC 96] festgelegten Zugriffsmethoden auf das Dateisystem implementiert.

- Kleinere Abweichungen von diesem Standard gibt es bei dem unter Linux verwendeten Dateisystem ext2 (*Extended filesystem version 2*), die über die Klasse `LinuxFS` berücksichtigt werden.

- Im Rahmen dieser Arbeit wurde mit `WinFS` eine weitere Klasse von `FS` abgeleitet, um die auf den Windows-Betriebssystemen üblichen Zugriffsmethoden auf das Dateisystem zu realisieren. Neben der Handhabung des von UNIX abweichenden Trennzeichens für Pfade war dabei vor allem auch die Berücksichtigung von Laufwerksbuchstaben bei einigen Methoden problematisch, z.B. bei der korrekten Identifikation absoluter und relativer Pfadangaben in den Methoden `isAbsolutePath` bzw. `isRelativePath`.

Eine Benutzung der hier vorgestellten Klassen kann immer erst dann erfolgen, wenn das aktuelle Betriebssystem ermittelt wurde um festzustellen, welche von `FS` abgeleitete Klasse für den Zugriff auf das Dateisystem benutzt werden muß. Eine solche Identifikation erfolgt bei jedem Neustart des Tycoon-2-Systems im Rahmen der dabei ausgeführten `Commit`-Operation (siehe [Weikard 98]). Diese Operation initiiert das Wiederherstellen aller Ressourcen im Rahmen des noch in Abschnitt 5.2 vorgestellten Ressourcen-Managements und aktualisiert verschiedene Systeminformationen in einem global angelegten Exemplar der Klasse `Tycoon`.[1] Dabei wird neben der Bereitstellung der aktuellen Programmparameter auch ein Exemplar der aktuell gültigen Dateisystemklasse in das Feld `tycoon.os.fs` eingetragen. Mit diesem Feld kann beispielsweise das aktuell zu benutzende Trennzeichen für Pfade über einen Aufruf von `tycoon.os.fs.pathArcSeparator` ermittelt werden.

Alle Dateisystemklassen der TL-2-Standardbibliothek benutzen direkt die in Abschnitt 4.7 beschriebenen Dateifunktionen aus der Schnittstelle von TycoonOS, eine Verwendung eingebauter Methoden ist hier nicht notwendig. Wie solche externen Systemaufrufe auf TycoonOS und auch auf andere dynamische Bibliotheken von TL-2 aus durchgeführt werden können, zeigt der folgende Abschnitt.

[1] Dieses globale Exemplar der Klasse `Tycoon` stellt auch geöffnete Standardkanäle für die Ein- bzw. Ausgabe bereit (`tycoon.stdin` bzw. `tycoon.stdout`). Die Klasse `Tycoon` ist dabei mit der aus Java bekannten Klasse `System` vergleichbar.

5.1.3. Dynamische Bibliotheken

In der TL-2-Standardbibliothek werden neben den Funktionen aus der in Kapitel 4 beschrieben portablen Systembibliothek TycoonOS auch noch einige Funktionen aus der C-Standardbibliohtek benötigt, wie z.B. `strcpy` oder `getenv`. Die aus diesen beiden dynamischen Bibliotheken benötigten Funktionen werden über die in Abbildung 5.3 dargestellten TL-2-Klassen `TycoonOS` und `AnsiC` angeboten. Beide Klassen leiten sich von der Klasse `Dll` ab, der allgemeinen Klasse für dynamische Bibliotheken, die auch für die Ableitung weiterer Klassen für externe Subsysteme benutzt werden muß.

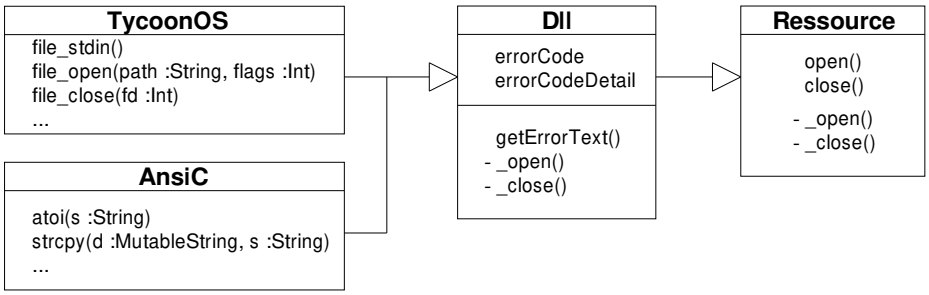


Abbildung 5.3.: TL-2-Klassen für dynamische Bibliotheken

Die eigentliche Definition der externen Funktionen geschieht in den von der Klasse `Dll` abgeleiteten Klassen. Eine solche externe Funktion kann in der Hochsprache TL-2 mit Hilfe des Schlüsselwortes `extern` definiert werden:

```
tl2_getenv(:String) :String extern "getenv"
```

Bevor allerdings eine solche externe Funktion aufgerufen werden kann, muß zunächst die dazugehörige dynamische Bibliothek geöffnet werden, was auch schon in Abschnitt 4.6.6 beschrieben wurde. Weiterhin sollten nach dem Aufruf einer externen Funktion alle eventuell vorhandenen Fehlerinformationen im Rahmen der in Abschnitt 4.5.4 vorgestellten einheitlichen Fehlerbehandlung bereitgestellt werden. Beide im folgenden näher betrachtete Aufgaben übernimmt die hierfür zuständige Klasse `Dll`.

Öffnen dynamischer Bibliotheken: Das Öffnen bzw. Schließen dynamischer Bibliotheken geschieht mit Hilfe der TL-2-Klasse `Dll` über einen Umweg:

Beide Operationen erfolgen in den in Abbildung 5.3 dargestellten privaten Methoden `_open` bzw. `_close` der Klasse `Dll`. Diese Methoden überschreiben die entsprechenden privaten Methoden der Klasse `Ressource`, die dort lediglich die Rolle eines Platzhalters einnehmen. In der Klasse `Ressource` befinden sich die eigentlichen öffentlichen Methoden für das Öffnen bzw. Schließen beliebiger Ressourcen, die in ihrer Implementierung die jetzt überschriebenen privaten Methoden `_open` bzw. `_close` benutzen.

Durch diesen Umweg über die Klasse `Ressource` werden geöffnete dynamische Bibliotheken in das Ressourcen-Management von Tycoon-2 aufgenommen, was noch in Abschnitt 5.2 näher betrachtet wird. Der Gewinn daraus besteht in diesem Falle darin, daß bei einem Neustart des Tycoon-2-Systems alle zuvor offenen dynamischen Bibliotheken vom System automatisch wieder geöffnet werden.

Fehlermanagement: In Abschnitt 4.5.4 wurde schon beschrieben, wie mit Hilfe der Variablen `errno` eine zentrale Handhabung von Fehlern beim Aufruf externer Funktionen möglich ist, auch wenn die entsprechenden Fehlerinformationen in irgendwelchen Subsystemen (z.B. Win32 API oder WinSocket) verborgen sind. Die Klasse `Dll` enthält auch verschiedene Felder und Methoden zum Abfragen solcher Fehlerinformationen (Abbildung 5.3). Für eine Beschaffung dieser Informationen ist es allerdings auf TL-2-Ebene nicht sinnvoll, unmittelbar über die TycoonOS-Funktion `tosError_getCode` auf `errno` zurückzugreifen: Durch den in Abschnitt 4.5.4 diskutierten ablaufinvarianten Zugriff auf `errno` ist zwar sichergestellt, daß es für jeden TL-2-Thread eine eigene Instanz dieser Fehlervariablen gibt, mehrere Exemplare der Klasse `Dll` innerhalb eines Threads würden jedoch auf die gleiche Instanz dieser Fehlervariablen zugreifen, so daß es zu Fehlinterpretationen kommen kann.

Statt des direkten Zugriffs auf die klassenweit in einem Thread gültige Fehlervariable `errno` (*Klassenvariable*) sollten nach jedem Aufruf einer externen Funktion der Fehler- bzw. Fehlerinfo-Code direkt bei den jeweiligen Exemplaren der Klasse `Dll` gespeichert werden (*Instanzvariable*).

Um diese einheitliche Fehlerbehandlung über die Klasse `Dll` zu realisieren, wurde im Rahmen dieser Arbeit folgende Modifikation der virtuellen Maschine von Tycoon-2 notwendig: Vor dem Aufruf einer externen Funktion aus der virtuellen Maschine heraus, der an zentraler Stelle über die Funktion `tmccall_execute` ausgeführt wird, muß der zuletzt bei dem aktuellen Exemplar der Klasse `Dll` aufgetretene Fehlerkontext mit Hilfe der TycoonOS-Funktion `tosError_setContext` wiederhergestellt werden. Dabei werden die betroffenen Fehlervariablen (über `errno` bzw. `SetLastError` und `WSASetLastError`) auf die zuletzt aufgetretenen Werte zurückgesetzt. Nach dem Aufruf der externen Funktion muß der Fehlerkontext mit den TycoonOS-Funktionen `tosError_getCode` und `tosError_getCodeDetail` wieder in das aktuelle Exemplar der Klasse `Dll` übertragen werden.

Dieser Vorgang wird durch folgenden, vereinfachten Ausschnitt aus dem Quelltext der virtuellen Maschine verdeutlicht, wobei die Zugriffe auf die TL-2-Felder hierbei stark vereinfacht dargestellt sind:

```
...
tosError_setContext(tyc_DLL->errCode, tyc_DLL->errCodeDetail);
pResult = tmccall_execute((tyc_ExternalMethod*)pMethod, ...);
tyc_DLL->errCode = tosError_getCode();
tyc_DLL->errCodeDetail = tosError_getCodeDetail();
...
```

Diese Form der Fehlerbehandlung hat auf Ebene der Hochsprache TL-2 zwei wichtige Konsequenzen: Zum einen ist das explizite Zurücksetzen von Fehlerinformationen in den Exemplaren der Klasse `Dll` über die Methode `Dll.resetError` sehr wichtig, um Fehlerinformationen auch wieder löschen zu können. Zum anderem dürfen nur noch Klassen, die von `Dll` abgeleitet wurden, externe Systemaufrufe ausführen, was allerdings eine akzeptable Bedingung ist, da das vor einem Aufruf notwendige Öffnen der entsprechenden dynamischen Bibliothek gerade von dieser Klasse übernommen wird.

Eine weitere Frage, die sich aus dieser Form der Fehlerbehandlung stellt, betrifft die Anwendung des Konzeptes der Ausnahmen – eigentlich sind diese in der Hochsprache TL-2 für eine Behandlung von Fehlern bzw. Ausnahmesituationen vorgesehen. Auf die Benutzung dieses hochsprachlichen Konzeptes bei einem Aufruf systemnaher, externer Funktionen wurde jedoch bewußt verzichtet. Im Rahmen spezifischer Anwendungsentwicklungen besteht immer noch die Möglichkeit, aus den in der Klasse `Dll` zur Verfügung gestellten Fehlerinformationen Ausnahmen zu generieren. Dies könnte durch die Einführung weiterer, umhüllender Klassen (*wrapper*) geschehen, die für alle Methoden mit externen Systemaufrufen eine einheitliche Ausnahmebehandlung vorsehen, was hier am Beispiel der TycoonOS-Funktion zum Kopieren einer Datei gezeigt wird:

```
fileCopy_withEx(from :String, to :String) :Void {
  if( tycoonOS.fileCopy(from, to) < 0
      then: { DllCallError.new.raise })
}
```

5.1.4. Netzwerkdienste

Die Vorstellung verschiedener Standardklassen für Systemdienste in der Hochsprache TL-2 soll mit der Betrachtung einiger Netzwerkdienste abgeschlossen werden. Abbildung 5.4 zeigt die hierfür zur Verfügung gestellten Klassen, die die TL-2-Schnittstelle zu den in Abschnitt 4.8.1 vorgestellten Netzwerkdiensten von TycoonOS bilden, in der Übersicht.

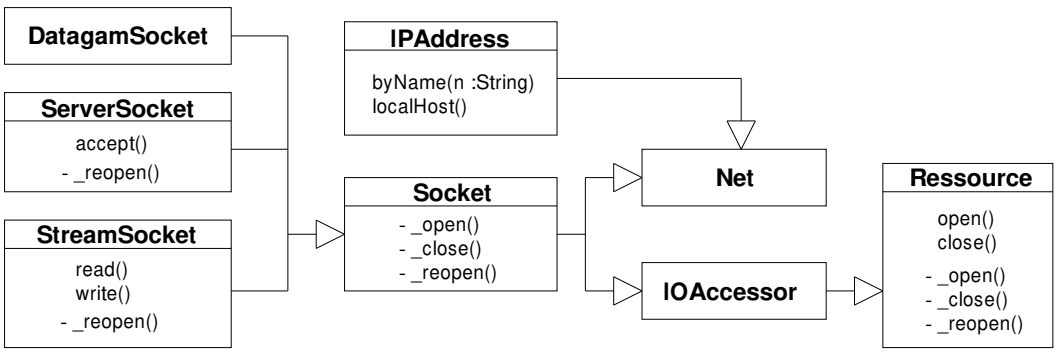


Abbildung 5.4.: TL-2-Klassen für Netzwerkdienste

Im Detail sollen diese Klassen hier nicht näher behandelt werden, einige Aspekte aus Abbildung 5.4 sind jedoch für die folgenden Ausführungen zum Ressourcen-Management im Tycoon-2-System von besonderem Interesse:

- Je nach Verwendungszweck für einen Socket gibt es unterschiedliche TL-2-Klassen. Diese Trennung ist durchaus sinnvoll, da beispielsweise ein Server-Socket lediglich für die Kontaktaufnahme mit den Klienten für den Diensterbringer verwendet werden darf (über die Methode `accept`), für Ein- bzw. Ausgabeoperationen kann dieser Socket dagegen nicht benutzt werden.

- Daß Öffnen bzw. Schließen eines Sockets wird, ähnlich wie bei den dynamischen Bibliotheken, über das Ressourcen-Management mit Hilfe der Klasse `Ressource` abgewickelt. Da über einige der Socket-Klassen auch Ein- bzw. Ausgabeoperationen ausgeführt werden können, ist hierbei zusätzlich noch die Klasse `IOAccessor` zwischengeschaltet. Über diese Basisklasse für Ein- und Ausgabeoperationen können beispielsweise auftretende I/O-Fehler zentral behandelt werden.

- Im Vergleich zu den dynamischen Bibliotheken fällt in Abbildung 5.4 für Sockets auf, daß es für das Öffnen die zusätzliche private Methode `_reopen` gibt. Mit dieser Methode können bei einem Neustart des Tycoon-2-Systems im Zusammenhang mit dem Ressourcen-Management differenzierte Aktionen für die Wiederherstellung eines Sockets ausgeführt werden. Dieses Verfahren ist notwendig, da Sockets neben dem eigentlichen Öffnen auch meist noch an spezielle Ports und IP-Adressen gebunden werden müssen [Comer, Stevens 94; Comer 95].

Unter welchen Umständen die Wiederherstellung einer Ressource beim Neustart gelingt und welche Grenzen dabei für mobile Objektspeicherzustände aufgezeigt werden können, soll im folgenden zum Abschluß dieser Arbeit am Beispiel eines TL-2-Threads, der über einen Server-Socket auf Anfragen wartet, dargestellt werden.

5.2. Ressourcen-Management und mobile Objektspeicher

Eine wichtige Eigenschaft des Tycoon-2-Systems besteht in der Fähigkeit, jederzeit den aktuellen Systemzustand in einen Objektspeicherzustand sichern zu können – damit kann zu einem beliebigen späteren Zeitpunkt dieser Systemzustand wiederhergestellt werden. Die in diesem Kapitel betrachteten Klassen für verschiedene Systemdienste stellen in diesem Zusammenhang eine besondere Problematik dar, da die Exemplare dieser Klassen (Threads, Bedingungsvariablen, Dateien, Sockets etc.) immer mit entsprechenden externen Ressourcen des Betriebssystems verbunden sind. Diese externen Ressourcen stehen außerhalb der Kontrolle des Tycoon-2-Systems und müssen daher bei einem Neustart des Systems durch spezifische Operationen wiederhergestellt werden, bevor sie wieder benutzt werden können. Diese Aufgabe, also die Buchführung über alle externen Ressourcen im Objektspeichersystem und die Wiederherstellung dieser Ressourcen beim Neustart des Systems, ist die Aufgabe des in [Weikard 98] entwickelten **Ressourcen-Managements** für Tycoon-2.

Die Sicherstellung der korrekten Funktion des Ressourcen-Managements unter dem Betriebssystem Windows NT war ein wichtiger Aspekt dieser Arbeit, insgesamt sollte eine hohe Mobilität der Objektspeicherzustände zwischen den Betriebssystemen erreicht werden.

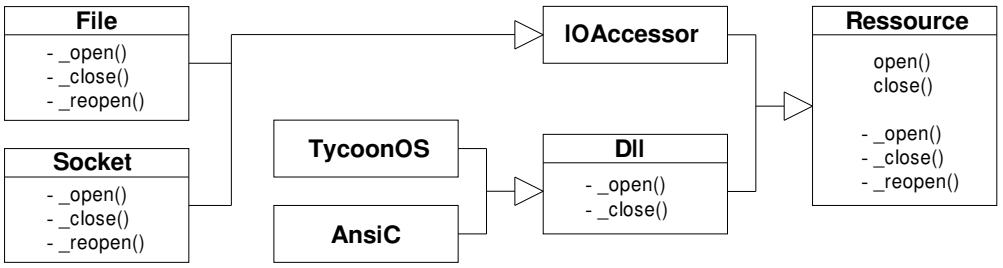


Abbildung 5.5.: TL-2-Klassen für das Ressourcen-Management

In Abbildung 5.5 wird ein für diese Arbeit relevanter Ausschnitt der am Ressourcen-Management beteiligten TL-2-Klassen dargestellt. An zentraler Stelle steht dabei die schon weiter oben vorgestellte Klasse `Ressource`, bei der über die öffentliche Methode `open` eine neue externe Ressource in das Ressourcen-Management aufgenommen und mit `close` wieder entfernt werden kann. Das eigentliche Öffnen und Schließen der Betriebssystemressourcen wird durch Aufruf der privaten Methoden `_open` bzw. `_close` durchgeführt, welche durch entsprechende Methoden in den eigentlichen Systemklassen (`File`, `Socket` etc.) überschrieben werden müssen.

Nachdem eine neue Ressource beim Ressourcen-Manager über den Aufruf der Methode `open` eingetragen wurde, wird für diese Ressource beim Neustart des Systems die private Methode `_reopen` aufgerufen. Der Ressourcen-Manager führt diesen Methodenaufruf für alle bei ihm eingetragenen Ressourcen aus, und zwar in der Reihenfolge, in der sich die Ressourcen mittels `open` angemeldet haben. Die in der Klasse `Ressource` implementierte Methode `_reopen` ruft einfach die private Methode `_open` der entsprechenden Ressource auf, für speziellere Aktionen beim Wiederherstellen der Betriebssystemressource kann die Methode `_reopen` überschrieben werden.

Tritt beim Wiederherstellen einer Ressource ein Fehler auf, wird vom Tycoon-2-System eine Warnung ausgegeben. Anschließend wird die Ressource über einen Aufruf der Methode `Ressource.close` geschlossen. Zu beachten ist, daß damit das Tycoon-2-System trotzdem gestartet wird und ein weiterer störungsfreier Betrieb möglich ist, solange die geschlossene Ressource nicht benötigt wird.

Abschließend folgt zur Verdeutlichung dieser Thematik noch ein Beispiel. Dabei wird ein TL-2-Thread gestartet, der einen Server-Socket auf Port 4711 des lokalen Rechners öffnet:

```
let t1 = Thread.new(fun() {
  let s = ServerSocket.new(IPAddress.localHost, 4711, 5)
  s.open
  ...
})
```

In einer Schleife wartet dieser Thread auf einkommende Nachrichten an diesem Port und beantwortet diese. Das Abbruchkriterium für den Thread ist das Eintreffen einer Nachricht, die die Zeichenkette "shutdown" enthält:

```
let msg = MutableString.new(500)
until( {msg.locateSubString("shutdown", 0) != nil} {
  let s2 = s.accept
  tycoon.stdout << "Request receives...\n"
  s2.readBuffer(msg, 0, 500)
  ... (* REPLY *)
  s2.close
}
```

Falls an irgend einer Stelle in der Ausführung des Threads `t1` der aktuelle Systemzustand gesichert und nach einer beliebig langen Unterbrechung ein Neustart des Tycoon-2-Systems durchgeführt wird, geschehen folgende Dinge:

- Da auch TL-2-Threads beim Ressourcen-Manager eingetragen sind, wird der Beispiel-Thread `t1` beim Wiederanlauf des Systems neu gestartet. Dabei wird die Ausführung des Threads an dem Punkt fortgesetzt, bei dem er sich beim Sichern des Systemzustandes befand.

  Ein dabei auftretendes Problem, welches an dieser Stelle aber nicht weiter vertieft werden soll, besteht darin, daß sich der Thread `t1` beim Sichern des Systemzustandes mit großer Wahrscheinlichkeit in einem blockierenden Betriebssystemaufruf (`accept`) befindet. Aus Sicht der Anwendung wird diese Problematik durch den Abbruch dieses blockierenden Systemaufrufes und das Auslösen einer Ausnahme beim Wiederanlauf gelöst. Damit kann im Rahmen der Ausnahmebehandlung eine einfache Wiederholung des blockierenden Systemaufrufes erfolgen:

  ```
  until( {msg.locateSubString("shutdown", 0) != nil} {
    try({
      let s2 = s.accept
      ...
    }
    fun(e :Exception) {
      tycoon.stdout << "Server socket re-started\n"
    })
  })
  ```

- Unmittelbar nach dem TL-2-Thread wird vom Ressourcen-Manager auch der Server-Socket `s` wieder neu geöffnet und an den alten Port mit der alten IP-Adresse gebunden. Für diesen Vorgang wird in der Klasse `ServerSocket` die private Methode `_reopen` aufgerufen, die wie folgt realisiert ist:

```
_reopen :Void {
  _handle := _checkReturnCode(_dll.socket_newServerSocket())
  _bind
  _listen
}
```

Nach diesen Aktionen beim Neustart des Tycoon-2-Systems nimmt der TL-2-Thread `t1` wieder seine Arbeit auf, so lange, bis er eine `"shutdown"`-Nachricht erhält. In der Praxis konnte mit diesem Beispiel gezeigt werden, daß der TL-2-Thread `t1` auch schadlos den Wiederanlauf auf einem anderen Betriebssystem überstehen kann. Mit einem unter Windows NT erstellten Objektspeicherzustand konnte problemlos ein Neustart des Systems unter Solaris durchgeführt werden.

Mit dieser betriebssystemübergreifenden Mobilität der Objektspeicherzustände wurde ein wichtiges Ziel dieser Diplomarbeit erreicht. Dazu konnte neben einem auf Windows NT korrekt laufenden Ressourcen-Management auch TycoonOS als Basis für die in diesem Kapitel beschriebenen TL-2 Systemdienste einen wesentlichen Beitrag leisten.

Abschließend sei auch noch auf die Grenzen der Mobilität der Objektspeicherzustände hingewiesen, die an dem oben aufgeführten Beispiel angedeutet werden können: Falls der Neustart des Tycoon-2-Systems auf einem anderen Rechner mit einer anderen lokalen IP-Adresse durchgeführt wird, mißlingt das Binden des Sockets an die im Objektspeicherzustand unter `s` eingetragene Adresse – man kann keinen Server-Socket auf einem fremden Rechner öffnen, soweit diese veraltete Adresse beim Neustart überhaupt noch bekannt ist. In dieser Situation könnte das erneute Öffnen des Server-Sockets und das Binden an die neue lokale Adresse von der Anwendungslogik übernommen werden. In dem hier aufgeführten Beispiel erfolgt dies im Rahmen der Ausnahmenbehandlung:

```
until( {msg.locateSubString("shutdown", 0) != nil} {
 try({
   let s2 = s.accept
   ...
 }
 fun(e :Exception) {
 if( s.address = IPAddress.localHost
    then: { tycoon.stdout << "Server socket restored on the same host\n" }
    else: { tycoon.stdout << "Server socket re-opened on different host\n"
            s.close
            s := ServerSocket.new(IPAddress.localHost, 4711, 5)
            s.open
          })
 }) (* try *)
}) (* until *)
```

Da das Öffnen eines Server-Sockets immer nur auf dem jeweiligen lokalen Rechner erfolgen darf, könnte dieses Problem auch direkt in der Klasse `ServerSocket` durch eine ent-

sprechende Erweiterung der privaten Methode `_reopen` gelöst werden. Hierzu ist generell anzumerken, daß die bei der Entwicklung vom Tycoon-2 WEB-Server entstandenen Klassen für Netzwerkdienste bei einer zukünftigen Benutzung noch weiter ausgebaut werden müßten; so sind beispielsweise in der Klasse für *Datagramme* noch überhaupt keine Methoden realisiert. Eine Betrachtung solcher Erweiterungsmöglichkeiten der TL-2-Standardklassen, von TycoonOS und vom Tycoon-2-System im allgemeinen erfolgt im Ausblick dieser Arbeit.

6. Bewertung und Ausblick

Im Rahmen dieser Arbeit wurde das Tycoon-2-System erfolgreich auf das Betriebssystem Windows NT übertragen. Mit der Identifikation und Anwendung des Architekturkonzeptes der *portablen virtuellen Maschine* auf das Tycoon-2-System, also der Implementierung der virtuellen Maschine von Tycoon-2 auf Basis der portablen Systembibliothek TycoonOS, konnte eine *statische Portabilität* dieser virtuellen Maschine erreicht werden. Auch die *dynamische Portabilität* der TL-2-Klassen und der persistenten Objektspeicher konnte auf Windows NT ausgedehnt werden.

Die Übertragbarkeit des gesamten Tycoon-2-Systems wurde erhöht und basiert jetzt auf den folgenden drei Säulen, die sich auch in den zentralen Kapiteln dieser Diplomarbeit widergespiegelt haben:

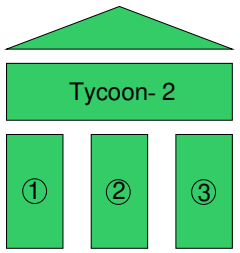


Abbildung 6.1.: Die drei Säulen des portablen Tycoon-2-Systems

1. Durch die Einführung der portablen Übersetzungssteuerung Jam/MR konnten die systemabhängigen Makefiles ersetzt werden. Dabei wurden mit Hilfe von Jam neue Regeln für die systemunabhängige Übersetzung dynamischer Bibliotheken und die Modellierung neuer Abhängigkeiten definiert, die für einen Bootstrap des Tycoon-2-Systems benötigt werden. Jam konnte die an eine übertragbare Übersetzungssteuerung gestellten Erwartungen voll erfüllen.

2. Mit der Einführung der portablen Systembibliothek TycoonOS konnten alle betriebssystemspezifischen Dienste für die virtuelle Maschine des Tycoon-2-Systems gekapselt werden. Dabei ist insbesondere eine Übertragung der Synchronisationsmechanismen aus dem POSIX-Threadmodell auf die Win32 API von Windows NT gelungen, wobei im Gegensatz zu anderen portablen Systembibliotheken auch das Problem des *Lost wakeup* berücksichtigt wurde.

3. Über TycoonOS konnten auch auf Ebene der Hochsprache TL-2 portable Systemdienste zur Verfügung gestellt werden, mit denen eine weitgehende Übertragbarkeit der Standardbibliothek von TL-2 erreicht wurde. Als unmittelbare Konsequenz hieraus wurde die Mobilität der Objektspeicherzustände vom Tycoon-2-System erhöht.

Der Erfolg dieser portablen Architektur des Tycoon-2-Systems zeigte sich in der danach vorgenommenen Übertragung auf das Betriebssystem Linux: Einschließlich kleinerer Anpassungen und Tests ließ sich diese Übertragung innerhalb von nur 2,5 Manntagen bewältigen. Dabei waren einige Änderungen im Zusammenhang mit dem Threadmodell von Linux (*LinuxThreads*) notwendig. Einschränkend ist zu berücksichtigen, daß sich die Linux-Implementierung von TycoonOS sehr dicht an die schon vorhandene Solaris-Implementierung anlehnt. Außerdem befanden sich in älteren Quelltexten des Tycoon-2-Systems schon einige Hinweise und Ansätze für eine Linux-Portierung.

Derzeit gibt es damit stabile Implementierungen von TycoonOS und dem Tycoon-2-System auf den Betriebssystemen Solaris, Windows NT und Linux. Weiterhin ist eine Implementierung für NextStep, HP-UX und IRIX bereits in Teilen realisiert, aber noch nicht umfassend getestet. Unter Berücksichtigung der in dieser Arbeit dargestellten Mängel in der Implementierung der Win32 API ist das Tycoon-2-System auch bedingt auf den Betriebssystemen Windows 95 und Windows 98 nutzbar – soweit auf die Verwendung der Sicherheitsdienste verzichtet und die Probleme bei der Threadsynchronisation (*Lost wakeup*) berücksichtigt werden. Eine Besserung ist hier erst dann in Sicht, wenn diese *Consumer*-Versionen von Windows einen auf Windows NT basierenden Betriebssystemkern erhalten, was Microsoft für die Zukunft in Aussicht stellt.

Eine Übertragung auf Betriebssysteme mit einer deutlich von Windows oder UNIX verschiedenen Architektur, wie z.B. BeOS der Firma Be [Be 99] oder Mac OS von Apple [Apple 99], wäre sicher noch eine größere Herausforderung. Hier wird die Übertragbarkeit des Tycoon-2-Systems auch an seine Grenzen stoßen, falls es beispielsweise zu große Unterschiede im Threadmodell oder zu starke Abweichungen im Dateisystem des Zielbetriebssystems zu der bisherigen Implementierung von Tycoon-2 gibt.

6.1. Ausblick

Die Betrachtung der zukünftigen Entwicklungsmöglichkeiten von Tycoon-2 zum Abschluß dieser Arbeit umfaßt eine Zusammenfassung der schon in Kapitel 4 angedeuteten Erweiterungsmöglichkeiten von TycoonOS sowie einige allgemeine Betrachtungen zu den Entwicklungsmöglichkeiten des Tycoon-2-Systems einschließlich der virtuellen Maschine und der Hochsprache TL-2.

Für die zukünftige Weiterentwicklung von TycoonOS wäre zunächst die Einführung einer Ressourcen-Verwaltung, wie sie beispielsweise bei den $\text{Pthreads}_{\text{Win32}}$ realisiert worden ist, sinnvoll – diese könnte zu einem schonenderen Umgang mit den Betriebssystemressourcen bei bestimmten Operationen (z.B. dem Abbrechen von Threads mit `tosThread_kill`) führen. Dies ist besonders bei Anwendungsprozessen mit einer langen Lebensdauer notwendig, z.B. für Server-Prozesse wie den Tycoon-2 WEB-Server.

Weiterhin wäre eine Integration leistungsfähiger Sicherheitsdienstleistungen mit Sicht auf den Einsatz von Tycoon-2 in großen verteilten Systemen wünschenswert. Die hierfür von TycoonOS angebotenen Dienste sind sehr elementar, da es hier zu große Unterschiede zwischen den Betriebssystemen gibt. Eine bessere Möglichkeit könnte in der schon in Kapitel 4 angedeuteten Integration externer, plattformübergreifender Sicherheitsdienstleistungen bestehen, wie beispielsweise dem am MIT entwickelten **Kerberos**-System [Kohl, Neuman 93; Steiner et al. 88].

Schließlich sind auch die in TycoonOS angebotenen Funktionen für Datum und Zeit nicht sehr umfangreich. Das Anlegen eines Exemplars der Klasse `Date` mit Hilfe einer formatierten Zeichenkette ist derzeit auf den Windows-Betriebssystemen nicht möglich (Abschnitt 4.8.2); ein neues Datum kann auf diesen Systemen nur über die Methoden `now` und `fromTime` erzeugt werden. Bei einer Erweiterung der Datums- und Zeitfunktionen ist allerdings zu entscheiden, ob auf Ebene von TycoonOS eine zusätzliche Bibliothek, wie z.B. die C-Bibliothek Date-Calc [DateCalc 99], genutzt werden sollte, oder ob umfangreichere Funktionen für Datumsberechnungen auf Ebene der Hochsprache TL-2 zu realisieren sind.

Für die Weiterentwicklung des Tycoon-2-Systems insgesamt ist noch ein wichtiger Aspekt zu berücksichtigen, der die portable Entwicklungsumgebung dieses Systems betrifft, aber auf den in dieser Arbeit nicht näher eingegangen wurde: Die Einführung einer Versionsverwaltung für alle Quelltexte des Tycoon-2-Systems. Bisher wird hierfür das auf SCCS basierende Teamware [Sun 99] von Sun eingesetzt. Dieses Produkt steht auf den Windows-Betriebssystemen allerdings nicht zur Verfügung, so daß die Möglichkeit untersucht werden sollte, auf portable Werkzeuge zur Versionsverwaltung zurückzugreifen, die auf allen für das Tycoon-2-System relevanten Betriebssystemen verfügbar sind. Mögliche Alternativen zu SCCS und Teamware wären dabei RCS (*Revision Control System*) oder CVS (*Concurrent Versions System*) [Tichy 82; Bolinger, Bronson 95].

Abschließend ist anzumerken, daß es natürlich diverse interessante Erweiterungsmöglichkeiten für das Tycoon-2-System gibt. Nach Einführung des Konzeptes der Komponenten und der neuen Syntax für die Hochsprache TL-2 in [Wienberg 99] sowie der portablen Architektur für die virtuelle Maschine von Tycoon-2 in dieser Arbeit wäre zunächst eine gewisse Konsolidierung notwendig. Dabei sollten die teilweise noch vorhandenen Implementierungsschwächen des Systems beseitigt werden, beispielsweise wäre die Einführung einer nebenläufig ausführbaren Speicherbereinigung (*Garbage collection*), aber auch eine Überarbeitung verschiedener TL-2-Klassen wünschenswert. Dazu gehören neben den Klassen für die Netzwerkdienste, mit den schon in Abschnitt 5.2 angedeuteten Verbesserungen für Sockets, auch die Klassen für den Tycoon-2 WEB-Server (`Http`, `Stml`, `Ftp` etc.), die in ihrer Übertragbarkeit zwischen den Betriebssystemen noch erheblich verbessert werden können.

Ein mögliches größeres Entwicklungsziel könnte der Versuch sein, sich für Tycoon-2 die vielfältigen, im Internet vorhandenen Java-Ressourcen nutzbar zu machen, beispielsweise durch Einführung einer zu CORBA konformen *Middelware* für Tycoon-2, oder durch eine Erweiterung der virtuellen Maschine von Tycoon-2 mit dem Ziel, eine *universelle virtuelle Maschine* für die gemeinsame Ausführung von Tycoon-2- und Java-Bytecode zu entwickeln.

Anhang

A. Die Schnittstellen von TycoonOS

A.1. Basisdefinitionen

A.1.1. tos.h – Basisdefinitionen und Initialisierungen

```
#ifndef TOS_H
#define TOS_H

#include "tosDBG.h"                /* Debug-Schalter   */
#include "tosTYPES.h"              /* Datentypen       */

/*== Zentrale Initialisierung für TycoonOS ================================*/

extern void tos_init(void);

#endif /* TOS_H */
```

A.1.2. tosDBG.h – Debug-Schalter

```
#ifndef TOSDBG_H
#define TOSDBG_H

/*== Debug-Schalter ======================================================

  Bitte die gewünschten Debug-Schalter anknipsen. Hat nur ein Effekt,
  falls in Jam 'buildmode = debug' definiert ist.

*/
#ifdef BUILDMODE_DEBUG
  #define tosFilename_DEBUG
  #define tosDll_DEBUG
  #define tosTLS_DEBUG
  #define tosMutex_DEBUG
  #define tosCondition_DEBUG
  #define tosThread_DEBUG
  #define tosThreadPri_DEBUG
  #define tosProcess_DEBUG
  #define tosStdio_DEBUG
  #define tosDirectory_DEBUG
  #define tosFile_DEBUG
```

```
  #define tosFilemode_DEBUG
  #define tosFileinfo_DEBUG
  #define tosFilepos_DEBUG
  #define tosNet_DEBUG
  #define tosSecurity_DEBUG
  #define tosSocket_DEBUG
#endif

/*== Automatische Schalter (nicht manuell setzen) ========================*/

#if defined(tosFilename_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosDll_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosTLS_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosMutex_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosCondition_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosThread_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosThreadPri_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosProcess_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosStdio_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosDirectory_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosFile_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosFilemode_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosFileinfo_DEBUG)
  #define tos_DEBUG
#endif
```

```
#if defined(tosFilepos_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosNet_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosSecurity_DEBUG)
  #define tos_DEBUG
#endif
#if defined(tosSocket_DEBUG)
  #define tos_DEBUG
#endif

#endif /* TOSDBG_H */
```

A.1.3. tosTYPES.h – Typdefinitionen

```
#ifndef TOSTYPES_H
#define TOSTYPES_H

/*== Compiler-Spezifisch =================================================*/

#if defined(rt_LIB_Win32_i386)
  #define tosTYPES_LONG           __int64
  #define tosTYPES_ULONG unsigned __int64
#else
  #define tosTYPES_LONG           long long
  #define tosTYPES_ULONG unsigned long long
#endif

/*== Basis-Definitionen ==================================================*/

#ifndef NULL
  #define NULL ((void*)0)
#endif

/*== Basis-Datentypen ====================================================*/

typedef unsigned char Byte;
typedef unsigned char Char;

#undef FALSE
#undef TRUE

typedef enum {FALSE, TRUE} Bool;

typedef char * String;
```

```
/*== Vorzeichenbehaftete Datentypen ======================================*/

typedef signed short   Short;
typedef int            Int;
typedef tosTYPES_LONG  Long;


/*== Vorzeichenlose Datentypen ===========================================*/

typedef unsigned char   QWord;
typedef unsigned short  HWord;
typedef unsigned int    Word;
typedef tosTYPES_ULONG  DWord;


/*== Gleitkomma-Typen ====================================================*/

typedef float  Float;
typedef double Double;
typedef Double Real;

#endif /* TOSTYPES_H */
```

A.2. Arithmetikfunktionen

A.2.1. tosInt32.h – Ganze Zahlen 32 Bit

```
#ifndef TOSINT32_H
#define TOSINT32_H

/*== Konvertierung =======================================================*/

extern int tosInt32_fromInt(int x);
extern int tosInt32_asInt(int x);

extern int tosInt32_fromString(const char *string);

/*== Ordnungsrelation / Operationen ======================================*/

extern int tosInt32_order(int x, int y);

extern int tosInt32_add(int x, int y);
extern int tosInt32_sub(int x, int y);
extern int tosInt32_mul(int x, int y);
extern int tosInt32_div(int x, int y);
extern int tosInt32_mod(int x, int y);

#endif /* TOSINT32_H */
```

A.2.2. tosLong.h – Ganze Zahlen 64 Bit

```
#ifndef TOSLONG_H
#define TOSLONG_H

#include "tos.h"

/*== Konvertierung ========================================================*/

extern Long tosLong_fromInt(int x);
extern int tosLong_asInt(Long x);


/*== Ordnungsrelation / Operationen =======================================*/

extern int tosLong_order(Long x, Long y);

extern Long tosLong_add(Long x, Long y);
extern Long tosLong_sub(Long x, Long y);
extern Long tosLong_mul(Long x, Long y);
extern Long tosLong_and(Long x, Long y);
extern Long tosLong_div(Long x, Long y);
extern Long tosLong_mod(Long x, Long y);

#endif /* TOSLONG_H */
```

A.2.3. tosReal.h – Gleitkommazahlen

```
#ifndef TOSREAL_H
#define TOSREAL_H

#include "tos.h"

/*== Konvertierung ========================================================*/

extern double tosReal_fromInt(int x);
extern int    tosReal_asInt(double d);

extern double tosReal_fromLong(Long x);
extern Long   tosReal_asLong(double d);

extern double tosReal_fromString(const char *string);


/*== Konstanten ===========================================================*/

extern Float  tosReal_floatNaN(void);
extern Double tosReal_doubleNaN(void);
```

```
/*== Ordnungsrelation / Operationen ======================================*/

extern int tosReal_order(double x, double y);

extern double tosReal_add(double x, double y);
extern double tosReal_sub(double x, double y);
extern double tosReal_mul(double x, double y);
extern double tosReal_div(double x, double y);

extern double tosReal_sqrt(double x);
extern double tosReal_ln(double x);
extern double tosReal_sin(double x);
extern double tosReal_cos(double x);
extern double tosReal_tan(double x);
extern double tosReal_asin(double x);
extern double tosReal_acos(double x);
extern double tosReal_atan(double x);

extern double tosReal_pow(double x, double y);
extern double tosReal_expE(double x);

#endif /* TOSREAL_H */
```

A.3. Fehlerbehandlung und Ablaufverfolgung

A.3.1. tosError.h – Allgemeine Fehlerbehandlung

```
#ifndef TOSERROR_H
#define TOSERROR_H

#include <assert.h>
#include <stdlib.h>

#include "tos.h"

/*== Fehlerkonstanten ====================================================*/

#define EWIN32API        6000
#define EWINSOCKET       6001

/*== Fehlermanagement für TycoonOS und TVM ================================*/

#define tosError_ABORT() assert(FALSE); abort();
#define tosError_reset() tosError_setContext(0, 0);

extern int tosError_getCode(void);
extern int tosError_getCodeDetail(void);
```

```
extern void tosError_setContext(int errCode, int errCodeDetail);

/*== Fehlertext-Routinen ================================================*/

extern void tosError_getText(int   errCode,
                             int   errCodeDetail,
                             char *msg,
                             int   msgSize);

extern void tosError_log(int errCode, int errCodeDetail);

#endif /* TOSERROR_H */
```

A.3.2. tosLog.h – Fehlerprotokoll und Ablaufverfolgung

```
#ifndef TOSLOG_H
#define TOSLOG_H

#include <stdio.h>


/*== Ausgabe-Umleitung ==================================================*/

extern void tosLog_redirectError(FILE *newFile);
extern void tosLog_redirectDebug(FILE *newFile);


/*== Nachrichtenprotokolle ==============================================*/

extern void tosLog_error(char *module, char *fun, char *msg, ...);

extern void tosLog_debug(char *module, char *fun, char *msg, ...);

#endif /* TOSLOG_H */
```

A.4. Systemmodule

A.4.1. tosMemory.h – Direkter Speicherzugriff

```
#ifndef TOSMEMORY_H
#define TOSMEMORY_H

#include "tos.h"

extern void tosMemory_formatString(char *buffer,
                                   char *fmt,
                                   int value);

extern int tosMemory_isNull(int base);


/*== Speicherwerte auslesen ================================================*/

extern int           tosMemory_peekInt         (int base, int offs);
extern char         *tosMemory_peekString      (int base, int offs);
extern char          tosMemory_peekChar        (int base, int offs);
extern unsigned char tosMemory_peekUnsignedChar(int base, int offs);
extern short         tosMemory_peekShort       (int base, int offs);
extern double        tosMemory_peekReal        (int base, int offs);
extern Long          tosMemory_peekLong        (int base, int offs);


/*== Speicherwerte setzen ==================================================*/

extern void tosMemory_pokeInt         (int base, int offs, int value);
extern void tosMemory_pokeString      (int base, int offs, char * value);
extern void tosMemory_pokeChar        (int base, int offs, char value);
extern void tosMemory_pokeUnsignedChar(int base, int offs, unsigned
                                                           char value);
extern void tosMemory_pokeShort       (int base, int offs, short value);
extern void tosMemory_pokeReal        (int base, int offs, double value);
extern void tosMemory_pokeLong        (int base, int offs, Long value);


/*== Konvertierung =========================================================*/

extern int tosMemory_IntToInt32(int value);

#endif /* TOSMEMORY_H */
```

A.4.2. tosString.h – Werkzeuge für Zeichenketten

```
#ifndef TOSSTRING_H
#define TOSSTRING_H

/*== Verschiedene Werkzeuge ================================================*/

extern char *tosString_strdup(const char *s);

extern char *tosString_extract(char       *dest,
                               int         pos,
                               int         len,
                               const char *source);


/*== Zeichenkettenoperationen für TL-2 =====================================*/

extern int tosString_locateSubString(char *s, char *sub, int startingAt);

extern int tosString_locateLastSubString(char *s, char *sub, int before);

extern int tosString_locateSomeChar(char *s, char *chars, int startingAt);

extern int tosString_locateSomeCharBefore(char *s, char *chars, int before);


#endif /* TOSSTRING_H */
```

A.4.3. tosSystem.h – Verschiedene Systemparameter

```
#ifndef TOSSYSTEM_H
#define TOSSYSTEM_H

#include "tos.h"

/*== Betriebssystem-Identifikation =========================================*/

#define tosSystem_ID_INVALID      0
#define tosSystem_ID_SOLARIS2     1
#define tosSystem_ID_LINUX        2
#define tosSystem_ID_HPUX         3
#define tosSystem_ID_NEXTSTEP     4
#define tosSystem_ID_WINNT        5
#define tosSystem_ID_WIN95        6
#define tosSystem_ID_WIN98        7
#define tosSystem_ID_UNKNOWN      8
```

```
/*== Lesen der Betriebssystem-ID ========================================*/

    tosSystem_getID     :  Aktuelle ID.
    tosSystem_getName   :  Dazugehöriger Name.
    tosSystem_NAME_SIZE :  Maximale Länge des Namens.
*/

#define tosSystem_NAME_SIZE 16

extern Int    tosSystem_getID  (void);
extern String tosSystem_getName(void);


/*== Bytefolge ==========================================================*/

#define tosSystem_BIG_ENDIAN     0
#define tosSystem_LITTLE_ENDIAN  1

/* Statisch: */

#if defined(rt_LIB_Nextstep_i386) ||
    defined(rt_LIB_Linux_i386)    ||
    defined(rt_LIB_Win32_i386)
  #define tosSystem_BYTEORDER tosSystem_LITTLE_ENDIAN
#else
  #define tosSystem_BYTEORDER tosSystem_BIG_ENDIAN
#endif

/* Dynamisch: */

extern Int tosSystem_getByteOrder(void);
extern Int tosSystem_getByteOrderOf(Int systemId);


/*== Initialisierung ====================================================*/

extern void tosSystem_init(void);

#endif /* TOSSYSTEM_H */
```

A.5. Threads und Threadsynchronisation

A.5.1. tosThread.h – Basisfunktionen für Threads

```
#ifndef TOSTHREAD_H
#define TOSTHREAD_H

#ifdef rt_LIB_Win32_i386
  #include <windows.h>
#else
  #include <pthread.h>
#endif
#if defined(rt_LIB_Solaris2_SPARC)
  #include <thread.h>
#endif

#include "tos.h"

/*== Datentypen für Threads und Thread-Deklarationen ======================

   T          - ID eines Threads.
   FUNC_T     - Funktions-Prototyp einer Ausführungsfunktion.
   FUNC_RET   - Ergebnis-Typ einer Ausführungsfunktion.
   FUNC_PARAM - Parameter-Typ einer Ausführungsfunktion.
   FUNC_DECL  - Parameter-Mechanismus einer Ausführungsfunktion.
*/

#ifdef rt_LIB_Win32_i386

  #define tosThread_T         HANDLE

  #define tosThread_FUNC_RET  unsigned
  #define tosThread_FUNC_PARA void *
  #define tosThread_FUNC_DECL __stdcall

#else

  #define tosThread_T         pthread_t

  #define tosThread_FUNC_RET  void *
  #define tosThread_FUNC_PARA void *
  #define tosThread_FUNC_DECL

#endif

typedef tosThread_FUNC_RET
           (tosThread_FUNC_DECL *tosThread_FUNC_T)(tosThread_FUNC_PARA);
```

```
/*== Starten bzw. Beenden von Threads ====================================

   tosThread_create         : Erzeugt einen neuen Thread.
   tosThread_exit           : Beendet den aktuellen Thread.
   tosThread_mainWaitedExit : Beendet das Hauptprogramm erst dann, wenn
                              alle Threads beendet sind.
   tosThread_kill           : Thread abbrechen.
*/

extern int tosThread_create(tosThread_T      *pTid,
                            tosThread_FUNC_T  threadFunction,
                            void             *pArg);

extern void tosThread_exit(void *exitCode);

extern void tosThread_mainWaitedExit(void);

extern int tosThread_kill(tosThread_T tid);


/*== Verschiedene Werkzeuge ================================================*/

extern void tosThread_sleep(Word sec);

#ifdef rt_LIB_Linux_i386

  #define tosThread_SCHED_FIFO  SCHED_FIFO
  #define tosThread_SCHED_RR    SCHED_RR
  #define tosThread_SCHED_OTHER SCHED_OTHER

  extern int tosThread_getSchedClass(tosThread_T tid, Int *class);
  extern int tosThread_setSchedClass(tosThread_T tid, Int  class);

#endif

extern int tosThread_getPriority(tosThread_T tid, Int *pri);
extern int tosThread_setPriority(tosThread_T tid, Int  pri);


/*== Initialisierung =======================================================*/

extern void tosThread_init();

#endif /* TOSTHREAD_H */
```

A.5.2. tosTLS.h – Dynamischer threadspezifischer Speicher

```
#ifndef TOSTLS_H
#define TOSTLS_H

#ifdef rt_LIB_Win32_i386
  #include <windows.h>
#else
  #include <pthread.h>
#endif
#if defined(rt_LIB_Solaris2_SPARC)
  #include <thread.h>
#endif

/*== Datentyp ============================================================*/

#ifdef rt_LIB_Win32_i386
  #define tosTLS_T DWORD
#else
  #define tosTLS_T pthread_key_t
#endif


/*== Schlüsselallokation =================================================*/

extern int tosTLS_keyCreate(tosTLS_T *key);
extern int tosTLS_keyDelete(tosTLS_T key);


/*== Zugriff auf Werte ===================================================*/

extern int    tosTLS_setValue(tosTLS_T key, void *val);
extern void * tosTLS_getValue(tosTLS_T key);

#endif /* TOSTLS_H */
```

A.5.3. tosMutex.h – Gegenseitiger Ausschluß

```
#ifndef TOSMUTEX_H
#define TOSMUTEX_H

#ifdef rt_LIB_Win32_i386
  #include <windows.h>
#else
  #include <pthread.h>
#endif
#if defined(rt_LIB_Solaris2_SPARC)
  #include <thread.h>
#endif
```

```
/*== Datentyp ============================================================*/

#ifdef rt_LIB_Win32_i386
  #define tosMutex_T HANDLE
#else
  #define tosMutex_T pthread_mutex_t
#endif

/*== Anlegen / Zerstören =================================================*/

extern int tosMutex_init   (tosMutex_T *mx);
extern int tosMutex_destroy(tosMutex_T *mx);

/*== Blockieren / Warten =================================================*/

extern int tosMutex_lock   (tosMutex_T *mx);
extern int tosMutex_trylock(tosMutex_T *mx);
extern int tosMutex_unlock (tosMutex_T *mx);

#endif /* TOSMUTEX_H */
```

A.5.4. tosCondition.h – Bedingungsvariablen

```
#ifndef TOSCONDITION_H
#define TOSCONDITION_H

#if defined(rt_LIB_Win32_i386)
  #include <windows.h>
#else
  #include <pthread.h>
#endif
#if defined(rt_LIB_Solaris2_SPARC)
  #include <thread.h>
#endif

#include "tos.h"
#include "tosMutex.h"

/*== Datentyp ============================================================*/

#ifdef rt_LIB_Win32_i386
  #define tosCondition_T HANDLE
#else
  #define tosCondition_T pthread_cond_t
#endif
```

```
/*
 *  Single    <: T
 *  Broadcast <: T
 */
typedef tosCondition_T tosCondition_Single;
typedef tosCondition_T tosCondition_Broadcast;


/*== Anlegen / Zerstören ================================================*/

extern int tosCondition_initSingle   (tosCondition_Single    *cond);
extern int tosCondition_initBroadcast(tosCondition_Broadcast *cond);

extern int tosCondition_destroy(tosCondition_T *cond);


/*== Senden / Warten ====================================================*/

extern int tosCondition_signal   (tosCondition_Single    *cond);
extern int tosCondition_broadcast(tosCondition_Broadcast *cond);

extern int tosCondition_wait(tosCondition_T *cond, tosMutex_T *mx);

extern int tosCondition_timedwait(tosCondition_T *cond,
                                  tosMutex_T     *mx,
                                  Long            mseconds);

#endif /* TOSCONDITION_H */
```

A.6. Dateisystemfunktionen

A.6.1. tosFile.h – Elementare Funktionen für Dateien

```
#ifndef TOSFILE_H
#define TOSFILE_H

#include "tos.h"


/*== Standard-Dateien ====================================================*/

extern int tosFile_stdin(void);
extern int tosFile_stdout(void);
extern int tosFile_stderr(void);


/*== Öffnen / Schließen ==================================================*/

#define TYC_O_RDONLY 0
#define TYC_O_WRONLY 1
#define TYC_O_RDWR   2
#define TYC_O_APPEND 4
#define TYC_O_SYNC   8
#define TYC_O_CREAT  16
#define TYC_O_EXCL   32
#define TYC_O_TRUNC  64

extern int tosFile_open(char *pszPath, int flags);
extern int tosFile_sync(int fd);
extern int tosFile_close(int fd);


/*== Datei Ein- und Ausgabe ==============================================*/

extern int tosFile_read1 (int fd, void * pBuffer, int start, int n);
extern int tosFile_write1(int fd, const void * pBuffer, int start, int n);

extern int tosFile_readChar (int fd);
extern int tosFile_writeChar(int fd, char ch);


/*== Datei-Informationen =================================================*/

extern Bool tosFile_exists(char *pszPath);
extern Bool tosFile_isCharacterDevice(int fd);
```

```
/*== Datei-Operationen ================================================*/

extern int tosFile_copy  (char *from, char *to);
extern int tosFile_rename(char *from, char *to);
extern int tosFile_remove(char *path);

/*== Symbolische Links ================================================*/

extern int tosFile_mkSymLink(char *name1, char *name2);    /* Nur für UNIX */

#endif /* TOSFILE_H */
```

A.6.2. tosFileinfo.h – Datei-Informationen

```
#ifndef TOSFILEINFO_H
#define TOSFILEINFO_H

#include <sys/types.h>
#include <sys/stat.h>

#include "tos.h"

/*== Konstanten / Typen ===============================================*/

#ifdef rt_LIB_Win32_i386
  typedef struct _stati64 tosFileinfo_Buffer;
#else
  typedef struct stat tosFileinfo_Buffer;
#endif

extern Int tosFileinfo_bufferSize(void);

extern Int tosFileinfo_getBuffer(char *path, tosFileinfo_Buffer *buf);

/*== Datei-Informationen ==============================================*/

extern Bool tosFileinfo_isFile        (tosFileinfo_Buffer *buf);
extern Bool tosFileinfo_isDirectory   (tosFileinfo_Buffer *buf);
extern Bool tosFileinfo_isSymbolicLink(tosFileinfo_Buffer *buf);

extern Long tosFileinfo_lastModified    (tosFileinfo_Buffer *buf);
extern Long tosFileinfo_lastAccessed    (tosFileinfo_Buffer *buf);
extern Long tosFileinfo_lastStatusChange(tosFileinfo_Buffer *buf);
extern Long tosFileinfo_size            (tosFileinfo_Buffer *buf);

#endif /* TOSFILEINFO_H */
```

A.6.3. tosFilemode.h – Dateizugriffsmodi

```
#ifndef TOSFILEMODE_H
#define TOSFILEMODE_H

#include <sys/types.h>
#include <sys/stat.h>

#include "tos.h"


/*== Konstanten / Typen ================================================*/

#if   defined(rt_LIB_Nextstep_i386)
  typedef unsigned short tosFilemode_T;
#elif defined(rt_LIB_Win32_i386)
  typedef unsigned short tosFilemode_T;
#else
  typedef mode_t tosFilemode_T;
#endif

#ifdef rt_LIB_Win32_i386
  #define tosFilemode_DEFAULT  _S_IREAD | _S_IWRITE
#else
  #define tosFilemode_DEFAULT  S_IFREG | S_IRWXU | S_IRGRP | S_IROTH
#endif


/*== Modi setzen bzw. schreiben ========================================*/

extern int tosFilemode_get(char *pszPath, tosFilemode_T *mode);
extern int tosFilemode_set(char *pszPath, tosFilemode_T  mode);

#endif /* TOSFILEMODE_H */
```

A.6.4. tosFilename.h – Handhabung von Dateinamen

```
#ifndef TOSFILENAME_H
#define TOSFILENAME_H

#ifdef rt_LIB_Win32_i386
  #include <stdlib.h>
#else
  #include <sys/param.h>
#endif

#include "tos.h"
```

```
/*== Konstanten ==========================================================*/

#ifdef rt_LIB_Win32_i386
  #define tosFilename_MAXLEN  _MAX_PATH
#else
  #define tosFilename_MAXLEN  MAXPATHLEN
#endif


/*== Dekomposition =======================================================*/

extern void tosFilename_split(const char * pszFilename,
                                    char * pszPath,
                                    char * pszName,
                                    char * pszExtension);

extern void tosFilename_getPath(const char * pszFilename,
                                      char * pszPath);

extern void tosFilename_getName(const char * pszFilename,
                                      char * pszName);

extern void tosFilename_getExtension(const char * pszFilename,
                                           char * pszExtension);


/*== Komposition =========================================================*/

extern void tosFilename_new(char * pszFilename,
                            const char * pszPath,
                            const char * pszName,
                            const char * pszExtension);

extern Bool tosFilename_setPath(char * pszFilename,
                                const char * pszPath);

extern Bool tosFilename_setName(char * pszFilename,
                                const char * pszName);

extern Bool tosFilename_setExtension(char * pszFilename,
                                     const char * pszExtension);


/*== Normalisierung ======================================================*/

extern void tosFilename_toWin(const char * pszUnixFilename,
                                    char * pszWinFilename,
                                    int    size);
```

```
extern void tosFilename_toUnix(const char * pszWinFilename,
                               char * pszUnixFilename,
                               int    size);

extern void tosFilename_normalize(const char * pszFilename,
                                  char * pszSystemFilename,
                                  int    size);


/*== Temporäre Dateinamen ================================================*/

extern int tosFilename_tempName(const char *dir,
                                const char *pfx,
                                      char *tmpName,    /* Resultat       */
                                      int   size);      /* Resultatsgröße */

extern int tosFilename_tempNameSize(void);

#endif /* TOSFILENAME_H */
```

A.6.5. tosFilepos.h – Zeiger für Dateipositionen

```
#ifndef TOSFILEPOS_H
#define TOSFILEPOS_H

#include "tos.h"

typedef Long tosFilepos_T;


/*== Positionierung ======================================================*/

extern tosFilepos_T tosFilepos_seek   (int fd, tosFilepos_T pos);
extern tosFilepos_T tosFilepos_seekEnd(int fd, tosFilepos_T pos);
extern tosFilepos_T tosFilepos_seekCur(int fd, tosFilepos_T pos);

#endif /* TOSFILEPOS_H */
```

A.6.6. tosStdio.h – Portable Standard Ein- und Ausgabe

```
#ifndef TOSSTDIO_H
#define TOSSTDIO_H

#include <stdio.h>

/*== Typen ===============================================================*/

#define tosStdio_T FILE
```

```
/*== Modi für das Öffnen ================================================*/

#define tosStdio_READWRITE_MODE "rb+"
#define tosStdio_READ_MODE      "rb"
#define tosStdio_WRITE_MODE     "wb"
#define tosStdio_CREATE_MODE    "wb+"


/*== Öffnen / Schließen =================================================*/

extern tosStdio_T *tosStdio_open(const char *szFileName, const char *szMode);

#define tosStdio_close fclose


/*== Ein- / Ausgabe =====================================================*/

#define tosStdio_read  fread
#define tosStdio_write fwrite
#define tosStdio_flush fflush


/*== Positionierung =====================================================*/

#define tosStdio_SEEK_CUR  SEEK_CUR
#define tosStdio_SEEK_END  SEEK_END
#define tosStdio_SEEK_SET  SEEK_SET

#define tosStdio_seek  fseek
#define tosStdio_tell  ftell

extern int tosStdio_truncate(tosStdio_T *file, int truncPos);

#endif /* TOSSTDIO_H */
```

A.7. Verzeichnisfunktionen

A.7.1. tosDirectory.h – Dateiverzeichnisse

```
#ifndef TOSDIRECTORY_H
#define TOSDIRECTORY_H

#include "tos.h"
```

```
/*== Typen ==================================================================*/

#ifdef rt_LIB_Win32_i386

  #include <windows.h>

  typedef struct {
            char            searchPath[_MAX_PATH];
            HANDLE          searchHandle;
            WIN32_FIND_DATA lastEntry;
          } tosDirectory_T;

#else

  #include <dirent.h>

  typedef struct {
            char   searchPath[MAXNAMLEN];
            DIR   *searchHandle;
            struct dirent lastEntry;
          } tosDirectory_T;

#endif


/*== Verzeichnis-Operationen ================================================*/

extern int tosDirectory_realpath(char *pszPath,
                                 char *pszResolvedPath,
                                 int   n);

extern int tosDirectory_chroot(char *pszRoot); /* Nur für UNIX */

extern int tosDirectory_create(char *pszPath);
extern int tosDirectory_remove(char *pszPath);


/*== Such-Kanal für Dateien =================================================*/

extern int   tosDirectory_sizeOfT(void);
extern int   tosDirectory_open   (tosDirectory_T *pDir, char *dirName);
extern char *tosDirectory_read   (tosDirectory_T *pDir);
extern int   tosDirectory_close  (tosDirectory_T *pDir);

#endif /* TOSDIRECTORY_H */
```

A.8. Prozesse und dynamische Bibliotheken

A.8.1. tosProcess.h – Einfache Prozeß-Routinen

```
#ifndef TOSPROCESS_H
#define TOSPROCESS_H

#include "tos.h"


/*== Prozeß-Ientifikation ================================================*/

extern Int tosProcess_getpid(void);
extern Int tosProcess_getppid(void); /* Nur für UNIX */


/*== Prozeß-Informationen ================================================*/

extern Long tosProcess_getUserCPUtime(void);
extern Long tosProcess_getKernelCPUtime(void);


/*== Prozeß-Kontrolle und Ausführung =====================================*/

extern Int tosProcess_kill(int pid, int value);

extern Int tosProcess_signal(int pid, int value);  /* Nur für UNIX */

extern Int tosProcess_execute(const char *cmdname, char *const argv[]);

extern Int tosProcess_system (const char *cmdname);

#endif /* TOSPROCESS_H */
```

A.8.2. tosDll.h – Dynamische Bibliotheken

```
#ifndef TOSDLL_H
#define TOSDLL_H

#include "tos.h"

/*== Definitionen ========================================================*/

#ifdef rt_LIB_Win32_i386
  #define tosDll_STD_LIBC "msvcrt"
#else
  #define tosDll_STD_LIBC "libc"
#endif
```

```
/*== Öffnen und Schließen ================================================*/

extern void *tosDll_open(char *pszPath);
extern Int tosDll_close(void *hDLL);

/*== Funktionsadresse beschaffen =========================================*/

extern void *tosDll_lookup(void * hDLL, char * pszSym);

#endif /* TOSDLL_H */
```

A.9. Netzwerkdienste

A.9.1. tosNet.h – Abfrage der Netzwerkdatenbank

```
#ifndef TOSNET_H
#define TOSNET_H

#include "tos.h"


/*== Konstanten ============================================================*/

#define tosNet_IPADDR_LENGTH 4

extern int tosNet_getIPAddrLen(void);
extern int tosNet_getMaxHostNameLen(void);


/*== Rechner-Informationen beschaffen ======================================*/

extern int tosNet_getHostName(char * pszName, int cbName);

extern int tosNet_getHostByName(const char *pszName,
                                Byte       *pbAddr,
                                int         cbAddr);


/*== Initialisierung =======================================================*/

extern void tosNet_init(void);

#endif /* TOSNET_H */
```

A.9.2. tosSocket.h – Netzwerkkommunikation mit BSD Sockets

```
#ifndef TOSSOCKET_H
#define TOSSOCKET_H

#include "tos.h"


/*== Datentypen ============================================================*/

typedef Int tosSocket_T;

extern Int tosSocket_nil(void);
```

```
/*== Socket anlegen ======================================================*/

extern tosSocket_T tosSocket_newStreamSocket(void);
extern tosSocket_T tosSocket_newDatagramSocket(void);
extern tosSocket_T tosSocket_newServerSocket(void);

extern int tosSocket_close(tosSocket_T s);


/*== Socket verbinden ====================================================*/

extern int tosSocket_connect(tosSocket_T  s,
                             Byte        *pbIPAddress,
                             int          cbIPAddress,
                             int          port);

extern int tosSocket_listen(tosSocket_T s, int queuelen);

extern int tosSocket_bind(tosSocket_T  s,
                          Byte        *pbIPAddress,
                          int          cbIPAddress,
                          int          port);

extern tosSocket_T tosSocket_accept(tosSocket_T s);


/*== Ein- und Ausgabe für Stream-Sockets =================================*/

extern int tosSocket_read(tosSocket_T  s,
                          void        *pBuffer,
                          int          start,
                          int          n);

extern int tosSocket_write(tosSocket_T  s,
                           const void  *pBuffer,
                           int          start,
                           int          n);


extern int tosSocket_readChar(tosSocket_T s);
extern int tosSocket_writeChar(tosSocket_T s, char ch);


/*== Socket-Informationen ================================================*/

extern int tosSocket_address(tosSocket_T  s,
                             Byte        *pbIPAddress,
                             int          cbIPAddress);
```

```
extern int tosSocket_remoteAddress(tosSocket_T  s,
                                   Byte        *pbIPAddress,
                                   int          cbIPAddress);

extern int tosSocket_port(tosSocket_T s);
extern int tosSocket_remotePort(tosSocket_T s);

#endif /* TOSSOCKET_H */
```

A.10. Verschiedene Werkzeuge

A.10.1. tosDate.h – Datums- und Zeit-Funktionen

```
#ifndef TOSDATE_H
#define TOSDATE_H

#include "tos.h"
#include <time.h>

/*== Zeitstempel ==========================================================*/

extern Long tosDate_time(void);
extern Long tosDate_clock(void);
extern long tosDate_clocksPerSecond(void);


/*== Datentypen und Konvertierung =========================================*/

#define tosDate_tm     tm
#define tosDate_time_t time_t

extern void tosDate_normalize(struct tosDate_tm *ptm);

extern void tosDate_fromTime(struct tosDate_tm *ptm, Long lTime);
extern Long tosDate_asTime(struct tosDate_tm *ptm);

extern int tosDate_fromString(struct tosDate_tm *ptm, char *buf, char *fmt);


/*== Handle für Datums- und Zeit-Informationen ============================*/

extern int tosDate_sizeOfHandle(void);

extern void tosDate_setHandle(struct tosDate_tm *ptm,
                              int year, int month, int day,
                              int hour, int minute, int second,
                              int offset,
                              int isDST);
```

```
/*== Datums- und Zeit-Informationen beschaffen ============================*/

extern int tosDate_format(struct tosDate_tm *ptm,
                                       char *fmt,
                                       char *buffer,
                                        int  n);

extern int   tosDate_offset(struct tosDate_tm *ptm);
extern char *tosDate_zone  (struct tosDate_tm *ptm);

extern int tosDate_second(struct tosDate_tm *t);
extern int tosDate_minute(struct tosDate_tm *t);
extern int tosDate_hour  (struct tosDate_tm *t);

extern int tosDate_day  (struct tosDate_tm *t);
extern int tosDate_month(struct tosDate_tm *t);
extern int tosDate_year (struct tosDate_tm *t);

extern int tosDate_weekDay(struct tosDate_tm *t);
extern int tosDate_yearDay(struct tosDate_tm *t);

extern int tosDate_isDST(struct tosDate_tm *t);


/*== Initialisierung =====================================================*/

extern void tosDate_init(void);

#endif /* TOSDATE_H */
```

A.10.2. tosMd5.h – Signierung von Daten

```
#ifndef TOSMD5_H
#define TOSMD5_H


/*== GLOBAL.H - RSAREF Typen und Konstanten ==============================*/

#ifndef PROTOTYPES
  #define PROTOTYPES 0
#endif

typedef unsigned char       *POINTER;
typedef unsigned short int   UINT2;
typedef unsigned long  int   UINT4;
```

```
#if PROTOTYPES
  #define PROTO_LIST(list) list
#else
  #define PROTO_LIST(list) ()
#endif

/*== MD5-Kontext ========================================================*/

typedef struct {
  UINT4 state[4];                                   /* state (ABCD) */
  UINT4 count[2];        /* number of bits, modulo 2^64 (lsb first) */
  unsigned char buffer[64];                         /* input buffer */
} MD5_CTX;

/*== Tycoon-2-Schnittstelle =============================================

     tosMd5_contextSize,
     tosMd5_init          : Allokation und Initialisierung eines Kontextes.
     tosMd5_update        : Neue Daten in Kontext einpflegen.
     tosMd5_final         : Prüfsumme schreiben und Kontext entleeren.
*/

extern unsigned int tosMd5_contextSize(void);
extern void tosMd5_init(MD5_CTX *context);

extern void tosMd5_update(MD5_CTX       *context,
                          unsigned char *buffer,
                          unsigned int   offset,
                          unsigned int   len);

extern void tosMd5_final(MD5_CTX       *context,
                         unsigned char  signature[16]);

#endif /* TOSMD5_H */
```

A.10.3. tosSecurity.h – Elementare Sicherheitsfunktionen

```
#ifndef TOSSECURITY_H
#define TOSSECURITY_H

#ifdef rt_LIB_Win32_i386
  #include <windows.h>
#else
  #include <sys/types.h>
  #include <unistd.h>
#endif

#include "tos.h"
```

```
/*== Datentypen =============================================================*/

typedef Int tosSecurity_SID;

/*== Typ-Konvertierungen ====================================================*/

extern Int tosSecurity_uidToString(tosSecurity_SID *uid,
                                   char            *res,
                                   Int             *sizeOfRes);

extern Int tosSecurity_gidToString(tosSecurity_SID *gid,
                                   char            *res,
                                   Int             *sizeOfRes);

extern Int tosSecurity_uidFromString(char            *uname,
                                     tosSecurity_SID *resUid,
                                     Int             *sizeOfSID);

extern Int tosSecurity_gidFromString(char            *gname,
                                     tosSecurity_SID *resGid,
                                     Int             *sizeOfSID);

/*== Aktuelle Zugriffscodes =================================================*/

extern Int tosSecurity_getCurrentUID(tosSecurity_SID *resUid,
                                     Int             *sizeOfSID);

extern Int tosSecurity_getCurrentGID(tosSecurity_SID *resUid,
                                     Int             *sizeOfSID);

extern Int tosSecurity_setCurrentUID(tosSecurity_SID *uid);
extern Int tosSecurity_setCurrentGID(tosSecurity_SID *gid);


/*== Datei-Zugriffscode ändern ==============================================*/

extern Int tosSecurity_chown(const char      *path,
                             tosSecurity_SID *owner,
                             tosSecurity_SID *group);


/*== Initialisierung ========================================================*/

extern void tosSecurity_init(void);

#endif /* TOSSECURITY_H */
```

Literaturverzeichnis

ACE 99: The ADAPTIVE Communication Environment (ACE) . http://www.cs.wustl.edu/~schmidt/ACE.html, 1999.

Althaus 92: Althaus, M. *PC/XT/AT Referenz-Handbuch.* SYBEX-Verlag, 1992. ISBN 3-88745-263-1.

Apple 99: Apple Computer . http://www.apple.com, 1999.

Arnold, Gosling 98: Arnold, Ken und Gosling, James. *The Java Programming Language.* The Java Series. Addison-Wesley, second edition, 1998.

Atkinson, Bunemann 87: Atkinson, M.P. und Bunemann, P. *Types and Persistence in Database Programming Languages.* ACM Computing Surveys, Jg. 19, Juni 1987, Nr. 2.

AT&T 99: U/WIN Research . http://www.research.att.com/sw/tools/uwin, 1999.

Baker 92: Baker, H.G. *The Treadmill, Real-time Garbage Collection without Motion Sickness.* ACM SIGPLAN Notices, Jg. 27, Maerz 1992, Nr. 3, S. 66–70.

Balzert 98: Balzert, H. *Lehrbuch der Software-Technik, Band 2: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung.* Spektrum Akademischer Verlag, 1998. ISBN 3-8274-0065-1.

Bartlett 88: Bartlett, J.F. *Compacting Garbage Collection with Ambiguous Roots.* Technical report, Digital Equipment Corporation, Februar 1988.

Be 99: Welcome to Be Incorporated . http://www.be.com, 1999.

Bolinger, Bronson 95: Bolinger, D. und Bronson, T. *Applying RCS and SCCS.* O'Reilly, 1995. ISBN 1-56592-117-8.

Busch et al. 93: Busch, A., Kuehnel, T., und Jahnke, A. *StarView 2.0.* Star Division, Hamburg, 1993.

Cardelli, Wegner 85: Cardelli, L. und Wegner, P. *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, Jg. 17, Dezember 1985, Nr. 4, S. 471–522.

Castaneda et al. 79: Castaneda, F., Chow, F., Nye, P., Sleator, D., und Wiederhold, G. *PC-FORT: A Fortran-to-Pcode translator.* Technical report, Standford University, Department of Computer Science, 1979.

Comer, Stevens 94: Comer, D. und Stevens, D.L. *Internetworking with TCP/IP; Vol II: Design, Implementation, and Internals.* Prentice-Hall, second edition, 1994. ISBN 0-13-134677-6.

Comer 95: Comer, D. *Internetworking with TCP/IP; Vol I: Principles, Protocols, and Architecture.* Prentice-Hall, third edition, 1995. ISBN 0-13-216987-8.

Cube 98: Cube, A. *Großprojekte in Word.* Magazin für Computertechnik, c't, Oktober 1998, Nr. 21, S. 216ff.

Cygnus 99a: Cygnus GNU Tools . http://sourceware.cygnus.com/cygwin, 1999.

Cygnus 99b: Pthreads for Win32 . http://sourceware.cygnus.com/pthreads-win32/, 1999.

DataFocus 99: MKS UNIX-Windows Co-Existence HomePage . http://www.datafocus.com, 1999.

DateCalc 99: Date::DateCalc – A package for all kinds of date calculations for the gregorian calendar . http://www.engelschall.com/u/sb/download, 1999.

Ernst 98: Ernst, M. *Typprüfung in einer polymorphen objektorientierten Programmiersprache.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, 1998.

Feldman 78: Feldman, S.I. *Make – A Program for Maintaining Computer Programs.* Technical Report Computing Science Technical Report No. 57, Bell Laboratories, August 1978.

Geisler 95: Geisler, Andreas. *Basisdienste zur Gestaltung einer reflektiven grafischen Entwicklungsumgebung für eine persistente Programmiersprache.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juni 1995.

Gosling, McGilton 95: Gosling, James und McGilton, Henry. *The Java Language Environment – A Whitepaper.* Technical report, Sun Microsystems, Oktober 1995.

Hachmann 96: Hachmann, Ilona. *Generische Visualisierungsdienste am Beispiel von StarView und Tycoon.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, April 1996.

Hamilton, Williams 96: Hamilton, D. und Williams, M. *Windows NT4 Programmierung für Insider.* SAMS Publishing, 1996. ISBN 3-87791-886-7.

Hoare 74: Hoare, C.A.R. *Monitors: An Operating System Structuring Concept.* Communications of the ACM, Jg. 17, Oktober 1974, Nr. 10, S. 549–557.

Hollich 99: Hollich, V. *Money und Quicken: Die Kontopfuscher.* Magazin für Computertechnik, c't, Juli 1999, Nr. 14, S. 52ff.

IBM 95: VisualAge C++ for OS/2, Open Class Library Reference . International Business Machines Corporation, Part No. S25H-6965, 1995.

Iseminger 99: Iseminger, D. *Microsoft Win32 Developer's Reference Library.* MSDN Library. Microsoft Press, 1999. ISBN 0-7356-0816-4.

ISO 95: 8601:1988 Data elements and interchange formats — Information interchange — Representation of dates and times . TC 154, Stage: 90.92, September 1995.

ISO 96: 9001:1994 Quality systems — Model for quality assurance in design, development, production, installation and servicing . TC 176/SC 2, Stage: 90.92, November 1996.

ISO 99: 9000-1:1994 Quality management and quality assurance standards — Part 1: Guidelines for selection and use . TC 176/SC 2, Stage: 90.60, Juli 1999.

ISO/IEC 93: 9126:1991 Information technology — Software product evaluation — Quality characteristics and guidelines for their use . JTC 1/SC 7, Stage: 90.92, August 1993.

ISO/IEC 96: 9945-1:1996 [IEEE/ANSI Std 1003.1, 1996 Edition] Information technology — Portable Operating System Interface (POSIX) - Part1: System Application Program Interface (API); This edition incorporates extensions for realtime applications (1003.1b-1993, 1003.1i-1995) and threads (1003.1c-1995) (pthreads) . JTC 1/SC 22, Stage: 60.60, November 1996.

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

Jones, Lins 96: Jones, R. und Lins, R. *Garbage Collection: algorithms for automatic dynamic memory management.* Wiley, 1996. ISBN 0-471-94148-4.

Kaufman et al. 94: Kaufman, C., Perlman, R., und Speciner, M. *Network Security - PRIVATE Communication in a PUBLIC World.* Prentice-Hall PTR, 1994. ISBN 0-13-061466-1.

Kohl, Neuman 93: Kohl, J. und Neuman, C. *The Kerberos Network Authentication Service (V5)* . RFC 1510, September 1993.

Korn 97: Korn, David G. *Porting UNIX to Windows NT.* In: USENIX (Hrsg.). *1997 Annual Technical Conference, January 6–10, 1997. Anaheim, CA*, Januar 1997.

Kristensen et al. 74: Kristensen, B., Madsen, O., und Jensen, B. *A PASCAL environment machine (P-Code).* DAIMI PB-28, Institute of Mathematics – University of Aarhus, 1974.

Leffler et al. 89: Leffler, McKusick, Karels, und Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989. ISBN 0-201-06196-1.

Lewis, Berg 98: Lewis, B. und Berg, D. *Multithreaded Programming with Pthreads.* Sun Microsystems, 1998. ISBN 0-13-680729-1.

Lewis et al. 98: Lewis, T., Sixpack, J., Lemming, L., und Nader, R. *Binary Critic.* IEEE Computer, Juli 1998, S. 107ff.

LinuxThreads 99: The Linux Threads library . http://pauillac.inria.fr/~xleroy/linuxthreads/index.html, 1999.

Lockemann, Schmidt 87: Lockemann, P.C. und Schmidt, J.W. (Hrsg.). *Datenbank-Handbuch.* Springer-Verlag, 1987.

Loviscach et al. 99: Loviscach, J., Brors, D., und G., Himmelein. *Das Office 2000-Problem.* Magazin für Computertechnik, c't, Juli 1999, Nr. 15, S. 122ff.

Mathiske et al. 95: Mathiske, B., Matthes, F., und Schmidt, J.W. *On Migrating Threads.* In: *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, Juni 1995. (Also appeared as TR FIDE/95/136).

Mathiske 96: Mathiske, B. *Mobility in Persistent Object Systems.* Dissertation, Fachbereich Informatik, Universität Hamburg, Germany, Mai 1996. (in German).

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

Matthes 97: Matthes, Florian. *Business Conversations: A High-Level System Model for Agent Coordination.* In: *Proceedings of the Sixth International Workshop on Database Programming Languages, Estes Park, Colorado.* Springer-Verlag, August 1997.

McCall et al. 77: McCall, J.A., P.K., Richards, und G.F., Walters. *Factors in Software Quality.* Rome Air Development Center, 1977.

Meyer 88: Meyer, B. *Object-oriented Software Construction.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

Microsoft 96a: Microsoft. *MS Windows NT Server 4.0 Resource Kit.* Microsoft Press, 1996. ISBN 1-57231-344-7.

Microsoft 96b: Microsoft. *MS Windows NT Workstation 4.0 Resource Kit.* Microsoft Press, 1996. ISBN 1-57231-343-9.

Müßig 94: Müßig, S. *Beiträge zur typsicheren generischen Datenvisualisierung.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juli 1994.

Mindcraft 99: Mindcraft. *Web and File Server Comparison: Microsoft Windows NT Server 4.0 and Red Hat Linux 5.2 .* http://www.mindcraft.com/whitepapers/nts4rhlinux.html, April 1999.

MSDN 99: Microsoft MSDN Library . http://www.microsoft.com/msdn/, 1999.

Nori et al. 75: Nori, K., Ammann, U., Jensen, K., und Nageli, H. *The PASCAL (P) Compiler Implementation Notes.* Technical report, Institut für Informatik, ETH Zürich, 1975.

Nyman 96: Nyman, N. *Microsoft Windows Operating System White Paper – Differences in Win32 API Implementations Among Windows Operating Systems* . http://channels.microsoft.com/ntserver/appservice/techdetails/techspecs/apidif.asp, Oktober 1996.

Pakendorf 96: Pakendorf, M. *Optimierung der persistenten objektorientierten Programmiersprache TooL.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juni 1996.

Perforce 97: Jam/MR 2.2 . http://www.perforce.com/jam/jam.html, Perforce Software, Inc., 1997.

Powell 84: Powell, M.L. *A portable optimizing compiler for Modula-2.* ACM SIGPlan Notices, Jg. 19, Juni 1984, Nr. 6, S. 310–318.

Rivest 92: Rivest, R. *The MD5 Message-Digest Algorithm* . RFC 1321, April 1992.

Sammet 69: Sammet, J.E. *Programming Languages: History and Fundamentals.* Prentice-Hall, 1969.

Seiwald 94: Seiwald, C. *Jam — Make(1) Redux.* In: *Proceedings of the 1994 USENIX UNIX Applications Development Symposium*, Berkeley, CA, USA, April 1994, S. 79–88.

Sites, Perkins 79: Sites, R. und Perkins, D. *Universal P-Code definition, version [0.3].* UCSD/CS-79/ 037, University of California, San Diego (UCSD), 1979.

Softway 99: Softway Systems, Welcome to Interix . http://www.interix.com, 1999.

Solomon 98: Solomon, D.A. *Inside Windows NT.* Microsoft Press, second edition, 1998. ISBN 1-57231-677-2.

Stallman, McGrath 93: Stallman, R.M. und McGrath, R. *GNU Make: A Program for Directing Recompilation*, Juli 1993.

Steel 61: Steel, T.B. *A First Version of UNCOL.* In: *Proceeding of the Eastern Joint Computer Conference.* Association of Computing Machinery, 1961, S. 371–377.

Steiner et al. 88: Steiner, G., Neuman, C., und Schiller, J. *Kerberos: An authentication service for open network systems.* In: *Proceedings of the USENIX Winter Conference*, Februar 1988, S. 191–202.

Sun 99: Sun Workshop Teamware . http://www.sun.com/software/Developer-products/teamware/spec.html, 1999.

Tanenbaum 92: Tanenbaum, A.S. *Moderne Betriebssysteme.* 2. Auflage, Carl Hanser Verlag & Prentice-Hall International Inc., 1992. ISBN 3-446-18402-3.

Tichy 82: Tichy, W. *Design, Implementation and Evaluation of a Revision Control System.* In: *Proceedings of the Sixth International Conference on Software Engineering*, 1982, S. 58–67.

Tsuji et al. 96: Tsuji, H., Love, B., Law, W., und Olson, B. *Power GUI Programming with Visual AgeTM for C++.* John Wiley & Sons, 1996. ISBN 0-4711-6482-8.

Wahlen 98: Wahlen, J. *Entwurf einer objektorientierten Sprache mit statischer Typisierung unter Beachtung kommerzieller Anforderungen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juni 1998.

Watt 90: Watt, D.A. *Programming language concepts and paradigms.* Prentice Hall international series in computer science. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

Weber, Hüskes 96: Weber, V. und Hüskes, R. *Gegen-Etwurf — StarOffice 3.0 fordert Microsoft Office 95 heraus.* Magazin für Computertechnik, c't, Januar 1996, Nr. 1, S. 140ff.

Wegner 98: Wegner, H. *Objektorientierter Entwurf und Realisierung eines Agentensystems für kooperative Internet-Informationssysteme.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Mai 1998.

Weikard 98: Weikard, M. *Entwurf und Implementierung einer portablen multiprozessorfähigen virtuellen Maschine für eine persistente, objektorientierte Programmiersprache.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juli 1998.

Wienberg 97: Wienberg, A. *Bootstrap einer Persistenten Objektorientierten Programmierumgebung.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, August 1997.

Wienberg 99: Wienberg, A. *Komponenten in einer objektorientierten Programmiersprache: Modell, sprachliche Umsetzung und Visualisierung.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, August 1999.

Willomat 97: Willomat, A. *Authentisierung in Tycoon: Eine polymorphe Entwicklungsschnittstelle zu Kerberos.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Oktober 1997.

Wingerd, Seiwald 97: Wingerd, L. und Seiwald, C. *Constructing a Large Product with Jam.* Lecture Notes in Computer Science, Jg. 1235, 1997, S. 36ff.

Erklärung

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

Hamburg, den 25. Januar 2000

(André Willomat)