Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-96-05

Ein Rahmensystem zur Erstellung verteilter Anwendungen

Martin Schaaf

Juli 1996

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Sema Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry of Education, Science, Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- ☐ Intelligent Engineering Systems
- ☐ Intelligent User Interfaces
- ☐ Computer Linguistics
- ☐ Programming Systems
- ☐ Deduction and Multiagent Systems
- ☐ Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Dr. Dr. D. Ruland
Director

Ein Rahmensystem zur Erstellung verteilter Anwendungen

Martin Schaaf

DFKI-D-96-05

This work has been supported by a grant from The Federal Ministry of Education, Science, Research and Technology (FKZ ITW-8902C4).

ISSN 0946-0098

Ein Rahmensystem zur Erstellung verteilter Anwendungen

Martin Schaaf

DFKI GmbH
Erwin-Schrödinger-Straße 57
Postfach 20 80
D-67608 Kaiserslautern

25. Juli 1996

Zusammenfassung

Das Projekt VEGA (Knowledge-Base Validation and Exploration by Global Analysis) zielt auf eine Unterstützung des Wissensingenieurs bei der Wartung von Wissensbasen ab. Dazu bedarf es des Zusammenspiels einer Vielzahl von Systemkomponenten, die mithilfe einer grafischen Oberfläche leichter bedienbar gemacht werden sollen. Desweiteren war geplant, auch extern entwickelte und implementierte Verfahren leicht integrierbar zumachen. Dabei soll dem besonderen Umstand Rechnung getragen werden, daß die Entwickler der Komponenten nicht unbedingt viel Erfahrung bei der Programmierung auf Betriebssystem- und Netzwerkebene haben müssen.

Daher wurde im Rahmen dieser Arbeit ein System entwickelt, das es mehreren unabhängigen Prozessen ermöglicht, auf einfache Art und Weise miteinander zu interagieren und zu kommunizieren. Zur Nutzung des Systems stehen einfache Schnittstellen zur Verfügung. Das System erlaubt den Komponentenentwicklern weiterhin, bei der Wahl ihrer Entwicklungssprache weitgehend von deren Eignung zur Systemprogrammierung unabhängig bleiben zu können.

Das entstandene System ist ein universell einsetzbarer Framework, der ganz allgemein von Entwicklern benutzt werden kann, die ihre Applikation auf eine Menge unabhängiger Akteure verteilen wollen, ohne deren Kommunikation und Verwaltung in den Mittelpunkt der Entwicklung zu stellen. Probleme, die bei der Realisierung von konkurrierender Programmabarbeitung auftraten, wurden mit Hilfe einer speziell entwickelten Thread-Bibliothek gelöst, die für sich allein bereits ein nützliches Hilfsmittel darstellt und verfügbar ist.

The project VEGA (Knowledge-Base Validation and Exploration by Global Analysis) aims to support the knowledge engineer in the maintenance of knowledge bases. For that purpose the cooperation of many system components is needed which should be made more easily usable by a graphical user interface. It was also planned to make externally developed tools integratable without much effort. Furthermore, it should be taken into consideration that many developers are not very experienced with programming at network or operating system level.

Therefore, we have developed a system which allows multiple independent processes to work and communicate easily together by having a simple interface. It gives the tool developers the ability to make their decision of which programming language to use, independent from its suitability for operating system programming.

The resulting system is a universal framework usable by any developer who wants to distribute his application on multiple actors, without focussing on their communication and administration. Internal process concurrency was solved by an own developed thread library which is already a useful and available tool in its own right.

Inhaltsverzeichnis

1 Einleitung

Das Projekt VEGA am DFKI beschäftigt sich mit dem Problem der Wartung und Pflege von Wissensbasen ([Boley *et al.*, 1992; Hinkelmann und Kühn, 1995]). Dabei kommen Methoden aus den verschiedensten Teilgebieten der *Künstlichen Intelligenz* zum Einsatz, z.B. des *Maschinellen Lernens* und der *Logischen Programmierung*. Zu diesen Methoden existiert eine Vielzahl von Verfahren, die teilweise auch konkret implementiert wurden. Ziel ist es nun, basierend auf diesen Verfahren ein prototypisches System zu entwickeln, das den Administrator einer Wissensbasis möglichst gut unterstützt. Im Verlauf der Umsetzung dieser Konzeption ergab sich die Notwendigkeit zu einem heterogenen System, bestehend aus unabhängigen, kommunizierenden Teilen, die in unterschiedlichen Programmiersprachen entwickelt wurden.

Eine Ursache hierfür war, daß man die Bedienung eines solchen Systems durch eine sowohl textbasierte als auch grafische Oberfläche attraktiver gestalten wollte. Dieser Oberfläche kommt die Aufgabe zu, von konkreten Realisierungen zu abstrahieren und statt spezifischen Algorithmen Unterstützung in Form höherer Funktionalitäten anzubieten (vgl. [Hinkelmann und Kühn, 1994; Abecker *et al.*, 1995]). Bereits in [Abecker, 1993] wurde vorgeschlagen, daß man die grafische Komponente als eigenständiges Modul realisieren sollte, wobei Grafik- und Kernalgorithmen in unterschiedlichen Programmiersprachen entwickelt werden können.

Eine weitere Ursache war das Ziel, verschiedene, extern entwickelte Verfahren ohne Reimplementierung zu integrieren und aneinander anzupassen.

Somit stellte sich das Problem der Integration mehrerer unabhängiger Prozesse, was zunächst transparent für die selbstentwickelten Verfahren geschehen sollte und durch einfaches Zusammenschalten mittels UNIX-Pipes auch hätte realisiert werden könnnen. Es war jedoch außerdem zu beobachten, daß bei den Verfahren viele Coderedundanzen existierten, da sich verschiedene Algorithmen, z.B. Formen der Unifikation, an vielen Stellen wiederfinden ließen. Aus diesem Grunde wurde eine Anpassung der gesamten bis dahin entwickelten Software an eine Client-Server Struktur vorgenommen, woraus sich für die Prozeßkommunikation mehr Freiheitsgrade ergaben, da sie jetzt Bestandteil des Gesamtkonzepts wurde.

Es steht nunmehr ein Wissensbankserver zur Verfügung, der die Ressource „Wissensbank“ verwaltet und verschiedene Dienste wie Zugriff und Modifikation von Wissenseinheiten bzw. Unifikation zur Verfügung stellt, die von den eigenen entwickelten Verfahren genutzt werden. An die externe Schnittstelle (zu extern entwickelten Verfahren und Oberflächen) wurde zusätzlich ein Modul gesetzt, welches wir *MCP* (Master Control Program) genannt haben und dessen Aufgabe es ist, zum einen eintreffende Anfragen zu interpretieren, zum anderen die intern entwickelten Verfahren in Funktionalitäten abzubilden, d.h. nach verschiedenen Gesichtspunkten Verfahren auszuwählen (siehe auch Abbildung 1).

Als Problem erwies sich nun, wie die Kommunikation zwischen MCP und externen Clients (Oberfläche bzw. extern entwickelten Verfahren) vonstatten gehen soll. Konkret bedeutet dies:

- Welche Kommunikationsmethoden sollen verwandt werden?

- Welche Kommunikationsressourcen werden benötigt und wer erzeugt bzw. verwaltet sie?

- Wie sollen mehrere Clients bedient werden und, fall diese konkurrierend geschieht, wer übernimmt die Synchronisation? Normale (relationale) Datenbanksysteme bieten solche Funktionalitäten schon seit einiger Zeit ([Date, 1990] und [Ullman, 1988]).

Da es sich hierbei um betriebssystemabhängige Aspekte handelt, entschied man sich, deren Behandlung nicht dem MCP zu überlassen, sondern ein Framework zu entwickeln, das möglichst einfache Schnittstellen zur Verfügung stellt und somit die einfache Verteilung des VEGA-Systems ermöglicht. Hierdurch erreichte man zum einen eine klare Funktionalitätstrennung, zum anderen auch eine Unabhängigkeit vom Entwicklungssystem des MCP und Wissensbankservers (zur Zeit Lucid Common Lisp).

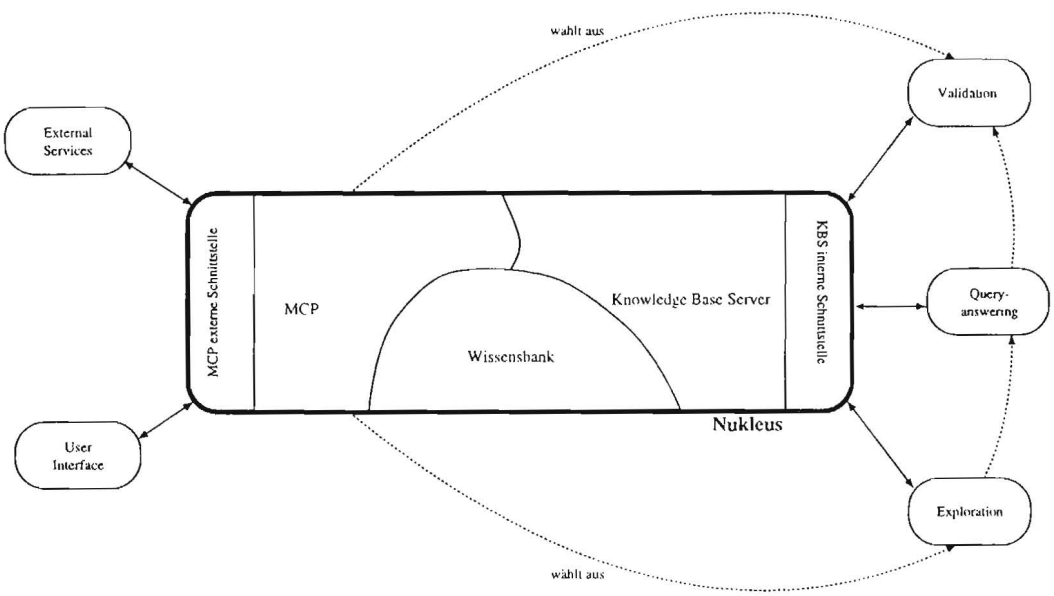

Abbildung 1: Struktur des VEGA-Systems

Die Zielvorgaben, die bei der Entwicklung des Frameworks einzuhalten waren, sind somit:

1. einfache Anforderungen an die Entwicklungssprache bzgl. deren Möglichkeiten zur Systemprogrammierung,

2. geringe Belastung der Entwickler mit den Problemen der Synchronisation von Prozessen und Verwaltung benötigter Ressourcen,

3. Bereitstellung sicherer Dienste,

4. die Mechanismen der Prozeßkommunikation selbst sollen möglichst wenig Rechenzeit verbrauchen.

Der Framework selbst erscheint in Form eines Servers, dem wir den einfallslosen Namen „Ressource Server" gegeben haben, wohlwissend, daß ein Server immer Ressourcen verwaltet. Neben den bereits genannten Anforderungen, sollte der „Ressource Server" in der Lage sein, Requests konkurrierend (quasiparallel) zu verarbeiten. Die eleganteste Art, so etwas zu realisieren, schien die Verwendung einer Standard Thread Bibliothek (z.B. POSIX Threads ([Sun Microsystems Inc., 1994])) zu sein, um jedem angeschlossenen Client einen eigenen prozeßinternen Kontrollfluß zuzuordnen. Leider scheiterte dieser Ansatz an der Nichtverfügbarkeit, Inkompatibilität bzw. Instabilität der Bibliotheken auf verschiedenen Plattformen, was die Portierbarkeit des VEGA-Systems sehr stark eingeschränkt hätte. Aus diesen (und weiteren) Gründen wurde eine eigene Thread Bibliothek T entwickelt, mit deren Hilfe dann die innere Konkurrenz des „Ressource Servers" elegant realisiert werden konnte.

Da bei der Entwicklung von T viele Aspekte des „Concurrent Programming" hervortraten, sowie der „Ressource Server" auch die Kommunikation über Rechnergrenzen hinaus mithilfe eines Netzwerks ermöglicht, wird in Abschnitt 2 zunächst eine Einführung mit Begriffsbestimmungen in diese Thematik gegeben. Im Umgang mit Betriebssystemen und Netzwerken erfahrenen Lesern wird dieses Kapitel nichts Neues bringen.

In Abschnitt 3 wird dann die Thread Bibliothek T vorgestellt, wobei auch eine ausführliche Befehlsreferenz eingeschlossen ist.

Abschnitt 4 befaßt sich mit der Konzeption des „Ressource Servers" und gibt Hinweise, mit denen sich die Implementierung nachvollziehen läßt. Dieser Abschnitt ist in erster Linie an Personen gerichtet, die den Server selbst weiterentwickeln bzw. warten wollen.

Leser, die lediglich an Art und Format der Requests des „Ressource Servers" bzw. einer einfachen Schnittstelle zu Lisp interessiert sind, sollten den Anhang lesen.

2 Grundlagen der Prozeßkommunikation und TCP/IP

2.1 Begriffsbestimmungen

Bevor mit der Behandlung der Thematik miteinander kooperierender Prozesse begonnen werden kann, sind zunächst einige Begriffe aus dem Bereich der Betriebssysteme zu klären. Ihre Bedeutung ist dort nicht immer einheitlich geregelt, so daß ich mich im folgenden an [Coulouris *et al.*, 1994] orientiere. Man beachte, daß im weiteren Verlauf keine Definitionen gegeben werden, sondern lediglich Begriffsbestimmungen, die bereits auf einem intuitiven Verständnis aufbauen.

2.1.1 Programme, Prozesse und Threads

Ein Programm ist eine Instruktionsfolge und beschreibt eine Sequenz auszuführender Operationen. Es ist sozusagen ein „Rezept", nach dem verfahren werden soll. Ein Prozeß ist hingegen die Abstraktion einer Einheit, die ein Programm ausführt, sozusagen der Zustand des „Kochs" (Prozessor). Der Begriff Prozeß ist damit sehr eng mit dem Begriff Prozessor gekoppelt, der die ausführende Einheit eines Rechnersystems ist und den Zustand eines Prozesses mit Hilfe seiner Register speichert und, entsprechend dem Programm, in Folgezustände übergeht. Minimal gehören damit zur Beschreibung eines Prozesses die Zustandsinformationen des Prozessors und die weiterer Register, sofern sie direkt mit dem Prozeß assoziiert sind (z.B. die der bei vielen Systemen vorhandenen *Memory Management Unit*).

Moderne Betriebssysteme wie UNIX ordnen dem Prozeß noch mehr Informationen zu, die allgemein mit dem Begriff der Ausführungsumgebung bezeichnet werden. Beispiele sind Variablen, die vom Erzeuger des Prozesses vererbt werden, bestimmte Dateideskriptoren, Benutzeridentifikationsnummern, benutzte Speicherregionen etc. Die Menge der einem Prozeß zugeordneten Informationen zerfällt damit in zwei Teile:

1. Informationen, die den Zustand der Ausführung beschreiben
2. Informationen, die bei der Erzeugung des Prozesses zugeordnet werden und eher einen statischen Charakter haben.

Ein Prozeß wird in [Coulouris *et al.*, 1994] auch als *Heavyweight Process* bezeichnet.

Ein allgemeineres Prozeßmodell erhält man, wenn man die Unterscheidung zwischen „dynamischen" Zustandsinformationen und Ausführungsumgebungen explizit macht. Erstere sollen im weiteren Verlauf *Threads* genannt werden, was sich aus „Thread Of Execution" ableitet. Ein Prozeß ist somit der Verbund aus Ausführungsumgebung und mindestens einem Thread, der ein Modell für die eigentliche Aktivität ist.
Ausführungsumgebungen sind durch

- einen Adreßraum (Menge von Speicherregionen, normalerweise nur von einem Prozeß nutzbar)[1]

[1] Es gibt jedoch auch Adreßbereiche, die mehreren Prozessen zur Verfügung stehen, z.B. um Coderedundanzen innerhalb des Speichers zu vermeiden. Hierzu zählen die, in denen die „Shared Libraries" abgelegt sind. Auch Teile des Kernels können im weitesten Sinne dazugerechnet werden.

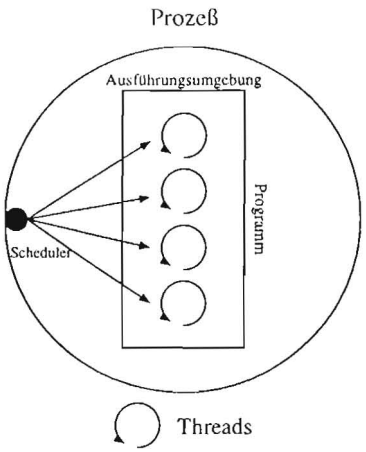

Abbildung 2: Mehrere Threads innerhalb eines Prozesses

- globale Ressourcen (Filedeskriptoren, Zugriffsrechte etc.)

charakterisiert, Threads hingegen durch

- Prozessorzustandsinformationen
- Laufzeitkeller
- ggf. weitere Parameter zur Schedulingstrategie

Der Laufzeitkeller ermöglicht es, für Threads lokale Variablen privat anzulegen und den Code damit reentrant zu machen. Prozeßglobale Variablen sollten nur zur Kommunikation und Synchronisation mit anderen Threads verwendet werden, da der Zugriff auf sie bei einem „multithreaded" Prozeß grundsätzlich einen „kritischen Pfad" (siehe 2.2) darstellen kann. Threads müssen sich, sofern sie globale Ressourcen benutzen, untereinander kennen und kooperieren.

Bei einigen Betriebssystemen ist es möglich, einzelne Threads zu erzeugen. Sie werden dann manchmal *Lightweight Processes* genannt, da ihre Kreation wesentlich weniger aufwendig ist als die eines ganzen Prozesses, der neben mindestens einem Thread auch die Ausführungsumgebung enthält.

2.1.2 Multitasking

Ein Thread wird zur Ausführung gebracht, indem die mit ihm assoziierten Informationen in die entsprechenden Register des Rechners geladen werden und entsprechend den Programminstruktionen fortgefahren wird. Damit scheint die Zahl der gleichzeitig ausführbaren Threads auf die Zahl der aktiven Einheiten wie Prozessor etc. beschränkt zu sein. Mit Unterstützung des Betriebssystems ist es jedoch möglich, durch Einfrieren des Zustands eines Threads und Restaurierung des Zustands eines anderen mehrere

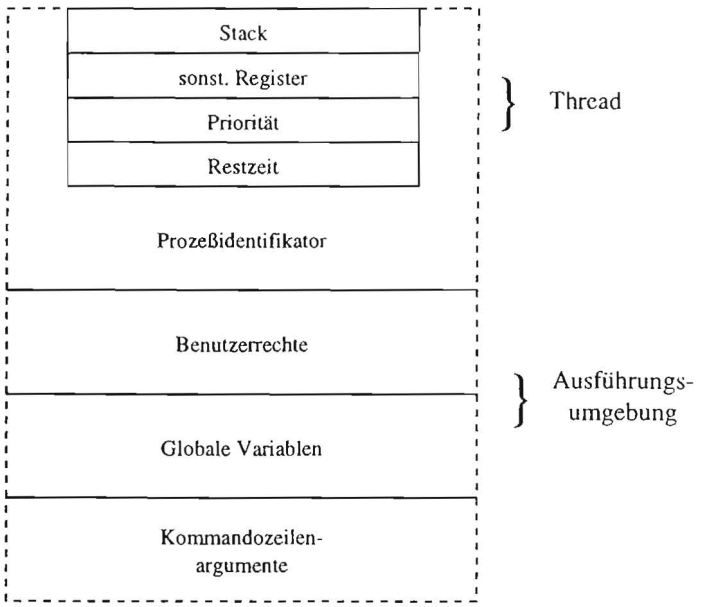

Abbildung 3: Bestandteile eines Prozesses mit einem Thread

virtuelle Prozessoren zu emulieren. Somit ist die quasiparallele Abarbeitung mehrerer Threads (und damit auch mehrerer Prozesse) möglich, die meistens den Gesamtdurchsatz eines Rechnersystems erhöht, da viele Threads einen Großteil ihrer Zeit im Wartezustand verbringen und ihre Rechenzeit dann anderen zur Verfügung gestellt werden kann. Diese Art der parallelen Abarbeitung nennt man (Pseudo–) *Multitasking* oder *Time Sharing*.

Strategien, nach denen das Umschalten mehrerer Prozesse (Threads) bewerkstelligt werden kann, sollen hier nicht erörtert werden; Interessenten seien an [Tanenbaum, 1992a] verwiesen. Grundsätzlich lassen sich jedoch zwei Arten auszeichnen:

1. Kooperation von Prozessen untereinander (kooperatives Pseudomultitasking).

2. zeitscheibengesteuertes Umschalten von Prozessen durch einen Scheduler (preemptives Pseudomultitasking).

Bei 1 geben die Threads ihre Rechenzeit freiwillig ab, d.h. ein Thread ist gehalten, falls er keine Rechenzeit mehr benötigt, das Betriebssystem davon in Kenntnis zu setzen. Oft kann dies beim Aufruf blockierender Systemfunktionen automatisch erkannt werden und wird stillschweigend vom Betriebssystem interpretiert. Trotzdem muß der Programmierer sich häufig darüber Gedanken machen, ob die Rechenzeit in seinem Programm monopolisiert wird. Threadsynchronisation ist einfach, da die Umschaltung auf einen anderen Thread nur zu definierten und bekannten Zeitpunkten vollzogen wird.

Im Gegensatz zu 1 wird bei 2 die Umschaltung zwischen Threads vom Betriebssystem vollzogen, d.h. nach einem bestimmten Zeitintervall wird durch einen Interrupt

der Scheduler des Betriebssystems aufgerufen, welcher je nach Strategie eine Threadumschaltung erzwingt oder aber noch weitere Zeitintervalle verstreichen läßt. Diese Methode befreit den Programmierer von der Aufgabe, sich innerhalb seiner Applikation über das Multitasking Gedanken zu machen, da es für ihn transparent vom Betriebssystem übernommen wird. Er kann jedoch keine Annahmen über die Atomizität einer Sequenz von Anweisungen innerhalb seines Programms machen, da eine Threadumschaltung zu jedem Zeitpunkt erfolgen kann.

Man beachte, daß bis hierher von Threads die Rede war. Die meisten Betriebssysteme kennen jedoch keine Threads, sondern sind lediglich in der Lage, zwischen verschiedenen Prozessen umzuschalten. Dies ordnet sich jedoch obiger Beschreibung unter, wenn vorausgesetzt wird, daß eine Threadumschaltung über Prozeßgrenzen hinweg erfolgen kann.

System Kernel	Thread Name	Prozeß Name
Amoeba	Thread	Process
Chorus	Thread	Actor
Mach	Thread	Task
V System	Process	Team

Abbildung 4: Benennung von Prozessen und Threads bei verschiedenen Betriebssystemen. Quelle: [Coulouris et al.,1994]

Threads können auch durch spezielle Bibliotheken der Programmiersprachen verfügbar gemacht werden, indem jedes Programm mit einem entsprechenden Scheduler gelinkt wird. Solche Realisierungen sind erheblich effizienter als vom Betriebssystem verwaltete Threads innerhalb eines Prozesses, da beim Umschalten, bei der Kreation und dem Zerstören der Threads keine „teuren" Systemaufrufe stattfinden. Allerdings kennt das Betriebssystem den einzelnen Thread dann nicht, und man muß darauf achten, daß prozeßblockierende Systemaufrufe allen Threads die Rechenzeit entziehen, da der ganze Prozeß als „nicht rechenbereit" eingestuft wird.

Die Terminologien „Threads", „Tasks" und „Prozesse" sind nicht einheitlich geregelt. Eine Übersicht über die Benennungen bei verschiedenen Betriebssystemen gibt Abbildung 4.

2.1.3 Ressourcen

Der Begriff *Ressource* läßt sich in Zusammenhang mit Rechnersystemen für nahezu alles verwenden, da jedes Betriebsmittel in irgendeiner Form nur begrenzt verfügbar ist. Beispiele hierfür sind der Prozessor, Speicher, der Drucker oder das Netzwerk. Aber auch physikalisch nicht existente Objekte, wie Lockvariablen (Semaphoren, siehe Abschnitt

2.3.3), Rechenzeit (als Abstraktion des Dienstes, den ein Prozessor anbietet) oder Dateien können Ressourcen sein. Ob ein Objekt Ressource ist, hängt nicht von dessen Beschaffenheit ab, sondern ergibt sich aus dem Umstand, ob es für irgendjemanden relevant ist, daß dieses Objekt nur begrenzt zur Verfügung steht. Erst in diesem Fall kann eine Konkurrenzsituation zwischen mehreren Interessenten (i.a. Threads) entstehen, die einer Auflösung bedarf. Dies war in älteren Systemen dem Betriebssystemkern vorbehalten, doch geht man in jüngster Zeit dazu über, Ressourcen von sogenannten Servern verwalten zu lassen, die sie dann in Dienste abbilden. Näheres dazu in Abschnitt 2.6.

2.2 Probleme beim konkurrierenden Zugriff auf Ressourcen

Wollen mehrere Threads auf ein und dieselbe Ressource zugreifen, so gibt es grundsätzlich die Möglichkeit, diese Konfliktsituation den Beteiligten zu überlassen. Dies ist zum Beispiel beim Betriebssystem UNIX der Fall, wenn zwei Prozesse auf dieselbe Datei zugreifen wollen und beide beim entsprechenden `OPEN` Systemaufruf einen Dateideskriptor erhalten. Die andere Möglichkeit besteht darin, die Ressource durch einen „Manager" verwalten zu lassen. Der direkte Zugriff ist dann von seiten der Tasks nicht mehr gestattet, sondern nur noch diesem vorbehalten. Auf diese Möglichkeit wird in Kapitel 2.6 noch näher eingegangen. Hier soll zunächst einmal auf entstehende Probleme beim konkurrierenden Zugriff ohne „Manager" eingegangen werden. Dazu ein Beispiel (entnommen und angepaßt aus [Nehmer, 1993]):
Gegeben sei eine relationale Datenbank, die in den Hauptspeicher geladen wurde und folgende Struktur besitzt:

```
flug(4711,1,"Konrad Adenauer")
flug(4711,2,"Ludwig Erhardt")
flug(4711,3,"Kurt Schumacher")
flug(4711,4,"Carlo Schmidt")
```

Dabei sollen die Stellen der `flug`-Relation bedeuten, daß beim Flug mit der Nummer `4711` der Platz `1` mit `Konrad Adenauer` besetzt ist. Neue Buchungen werden durch Eintragen weiterer Relationen vorgenommen. Dabei sei nun angenommen, daß zwei Prozessen zwei Eingabeterminals zugeordnet sind und beide parallel (preemptives Multitasking vorausgesetzt) und direkt auf die Datenbank zugreifen können. Zur Vereinfachung setze ich noch die Existenz der folgenden Funktionen in beiden Prozessen voraus:

```
lookup(FlugNr,Platz) -> Name
```

liefert den Namen eines Passagiers, falls dieser Platz auf dem entsprechenden Flug bereits vergeben ist, einen leeren String sonst.

```
assert(FlugNr,Platz,Name)
```

fügt eine neue Relation `flug` mit den entsprechenden Parametern in die Datenbank ein.
Die Prozedur, die eine Buchung für einen Passagier auf dem ersten freien Platz vornimmt, könnte dann folgendermaßen lauten:

```
void book (FlugNr,MaxPlaetze,Name)
{
(1)  found = 0;
(2)  counter = 1;
(3)  while ((counter < MaxPlaetze) && (found == 0)) {
(4)    if (lookup(FlugNr,counter) == "") {
(5)      assert(FlugNr,counter,Name);
(6)      found = 1;
       }
(7)    else counter++;
     }
(8)  if (found == 0) printf("Flugzeug belegt");
(9)  else printf("Platz %d wurde reserviert",counter);
}
```

Da bei preemptivem Multitasking zu jedem Zeitpunkt eine Taskumschaltung erfolgen kann, ist es insbesondere denkbar, daß diese nach dem Vergleich von (4) erfolgt. Wollen also beide Prozesse eine Flugbuchung vornehmen und einer der beiden wird vom anderen nach (4) unterbrochen, während dieser wiederum seine Buchung komplett abschließen kann, so ist nach Abschluß dieser Transaktionen ein Platz doppelt belegt. Der Grund für diesen Fehler liegt darin, daß die Sequenz des Vergleichens, ob ein Platz frei ist, und des Belegens im positiven Fall nicht unterbrochen werden darf, da der entsprechende Prozeß sonst keine konsistente Sicht auf den Zustand des gemeinsam benutzten Speichers hat. Solche Sequenzen nennt man *kritische Abschnitte* [Dijkstra, 1968] (siehe auch Abbildung 5).

Die Idee zur Vermeidung solcher Fehler ist, kritische Abschnitte innerhalb einer Berechnungssequenz atomar zu machen, d.h. durch entsprechende Programmanweisungen dafür zu sorgen, daß sie nicht unterbrochen werden können. Verschiedene Mechanismen werden in 2.3 besprochen und teilweise in der Implementation verwendet.

2.3 Synchronisationsmechanismen

In 2.2 wurde anhand eines Beispiels die Notwendigkeit von Prozeßsynchronisationsmechanismen (genauer Threadsynchronisationsmechanismen) bei preemptivem Multitasking gezeigt. Bei kooperativem Multitasking ist die Situation nicht so dramatisch, da die Threads dort wissen, wann eine Umschaltung „droht“ bzw. sie selber inszenieren. Im folgenden sollen einige Verfahren vorgestellt werden, mit denen eine Threadsynchronisation bewerkstelligt werden kann. Dies soll jedoch nicht erschöpfend geschehen, Interessierte seien an [Tanenbaum, 1992b] verwiesen. Alle Verfahren eignen sich dazu, einen kritischen Abschnitt (siehe Abb. 5) zu sichern, ihn also atomar in der Ausführung zu machen.

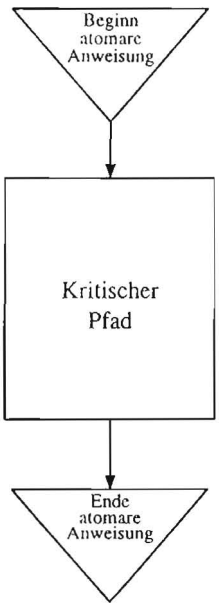

Abbildung 5: Kritischer Abschnitt. Quelle: [Nehmer, 1993]

2.3.1 Interrupt Disabling

Die einfachste Methode, einen Berechnungspfad gegen Unterbrechungen zu sichern, ist, entsprechende Interrupts zu sperren und damit die Multitaskingfähigkeit des Betriebssystems quasi außer Kraft zu setzen, denn der Scheduling–Mechanismus der Time Sharing Betriebssysteme beruht in der Regel auf der Lieferung eines zyklischen Interrupts von seiten der Hardware. Zwar ist diese Methode für den normalen Benutzerprozeß nicht möglich, doch realisieren viele Betriebsysteme genauso einige Verfahren, die noch besprochen werden.

2.3.2 TSL-Hardware Befehl

Diese Methode benötigt etwas Unterstützung von der Hardware. Ein großer Nachteil der Interrupt Disabling Methode ist, daß sie bei Mehrprozessormaschinen nicht funktioniert. Aus diesem Grund haben Prozessoren solcher Maschinen meistens einen speziellen TSL (Test-Set-Lock)-Befehl, mit dem eine Speicherzelle atomar ausgelesen und gesetzt werden kann.

```
tsl [Addr]
```

entspricht

```
mov REG,[Addr]
mov [Addr],1
```

Dabei steht `REG` für ein Register, das implizit vom TSL-Befehl genutzt wird. Die entsprechende Speicherstelle wird also zunächst ausgelesen und anschließend mit 1 belegt. Anhand des ausgelesenen Wertes kann man feststellen, ob die Speicherstelle bereits

vorher belegt war (und man sie in diesem Fall gar nicht verändert hat), oder ob sie jetzt neu belegt wurde.

Ein kritischer Pfad kann nun mit folgenden Prozeduren eingeschlossen werden:

```
void enter_critical_path (int *lock)
{
  int temp;
  do {
    inline {
      tsl [lock] ; dabei wird der Wert von lock nach REG kopiert
                 ; und lock gesetzt
      mov [&temp],REG
    }
  } while (temp == 1)
}

void leave_critical_path (int *lock)
{
  *lock = 0;
}
```

Neben dieser Möglichkeit der Synchronisation gibt es weitere, rein algorithmische Verfahren, wie z.B. Token Passing oder der Dekker-Algorithmus (siehe [Nehmer, 1993] oder [Tanenbaum, 1992b]). Allen haftet jedoch der Nachteil an, daß Prozesse, die einen kritischen Abschnitt nicht betreten können, in einer aktiven Wartestellung verbleiben und damit Rechenzeit verbrauchen, da Lock-Variablen ständig abgefragt werden.

2.3.3 Semaphoren

Das erste praxistaugliche Konzept für die Synchronisation wurde in [Dijkstra, 1968] vorgestellt; es soll in diesem Abschnitt besprochen werden.

Definition 1 *Eine Semaphorvariable ist eine positive Ganzzahlvariable S, für die zwei Operationen P und V existieren, die folgendermaßen definiert sind:*
Sei zunächst W eine Liste von Threads und T1 ein aktiver Thread, der eine der Operationen P und V ausführt, dann gilt
$P(S) = S - 1$, *falls* $S > 0$
$P(S) = 0$, $W = W \cup \{T1\}$ *und T1 blockiert, falls* $S = 0$, *sowie*
$V(S) = S + 1$, *falls* $W = \emptyset$
$V(S) = 0$, $W = W \setminus \{T2\}$ *und T2 entblockiert, falls* $W \neq \emptyset$.

Anschaulich besagt Def. 1, daß Threads eine Semaphorvariable passieren dürfen, solange deren Wert größer als 0 ist, ansonsten werden sie blockiert. Wird der Zähler einer Semaphore erhöht, z.B. nach Verlassen eines kritischen Abschnitts durch einen Thread, so wird aus der Liste der blockierten Threads einer ausgewählt und aktiviert.

In der Praxis werden meist alle ausgewählt und beginnen erneut, sich um die Semaphore zu bewerben. Durch geeignete Initialisierungen der Semaphore kann man damit sehr flexibel die Anzahl der KonkurrentInnen festlegen, die eine Ressource gleichzeitig benutzen dürfen; häufig entspricht der Wert der Semaphore direkt der Anzahl einer Ressource.

Semaphoren zur Synchronisation von Prozessen müssen vom Betriebssystem bereitgestellt werden, da die Operationen P und V ihrerseits selbst kritische Pfade beinhalten, also nicht unterbrochen werden dürfen. Das Betriebssystem selbst garantiert dies unter Verwendung der Methoden aus 2.3.1 oder 2.3.2.

2.3.4 Monitore

Obwohl Semaphoren bereits ein funktionstüchtiges Verfahren zur Threadsynchronisation sind, ist die Programmierung großer Applikationen sehr schwierig, denn jeder Thread hat selbst dafür Sorge zu tragen, daß vor Eintritt in einen kritischen Abschnitt die richtigen Semaphoren in der richtigen Reihenfolge passiert werden. Dabei ist gerade die Reihenfolge von Bedeutung, wie das folgende Beispiel zeigt:

```
thread 1:
sem a,b;
...
P(a);
P(b);
.
kritischer Abschnitt
Thread1
.
V(b);
V(a);
```

```
thread 2:
sem a,b;
...
P(b);
P(a);
.
kritischer Abschnitt
Thread2
.
V(a);
V(b);
```

Wird in obigem Beispiel `thread 1` unterbrochen, nachdem er die Semaphore `a` passiert hat, aber noch vor `b`, und kann `thread 2` anschließend `b` passieren, so können beide nicht mehr weitermachen, da jeder von ihnen eine Ressource belegt, die der andere benötigt. Sie können sie jedoch auch nicht mehr zurückgeben. Es herrscht eine Deadlock-Situation.

Um ein solches Fehlverhalten einzelner Threads zu umgehen, muß der Programmierer das Zusammenspiel aller im Auge behalten, was jedoch konträr zu modernen Programmierprinzipien läuft, da Threads zumeist nach funktionellen Gesichtspunkten gebildet werden.

Ein respektive modernen Software Engineerings besserer Ansatz, eine Ressource zwischen konkurrierenden Threads zu verwalten, ist der, die Ressource in den Mittelpunkt zu stellen und durch entsprechende Funktionen zu abstrahieren. In diesem Fall müssen sich nur noch die abbildenden Funktionen untereinander synchronisieren. Ein interessierter Thread bewirbt sich, indem er eine solche Funktion aufruft und wird ggf.

abgelehnt und dabei auf Wunsch blockiert.

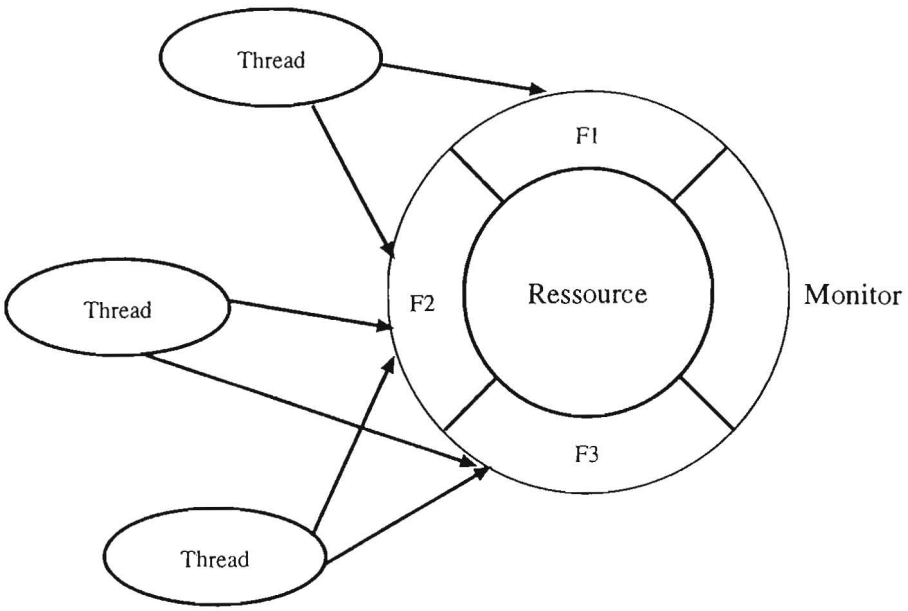

Abbildung 6: Monitor

Ein solches Konzept ist als **Monitorkonzept** bekannt (siehe auch Abbildung 6), die Funktionen samt den verwalteten Ressourcen bilden den Monitor. In Gegensatz zu den bisher vorgestellten Möglichkeiten zur Threadsynchronisation, stellt das Monitorkonzept keine neuen Funktionalitäten zur Verfügung, sondern ist lediglich ein Softwarestrukturkonzept, welches sich mit den anderen Methoden realisieren läßt. Manche Programmiersprachen bieten Monitore, ähnlich Module, als Sprachkonzept an.

2.4 Kommunikationsmodelle

Nachdem in den vorhergehenden Kapiteln hauptsächlich über Threadsynchronisation gesprochen wurde, soll in diesem Kapitel die verwandte Thematik der Kommunikation behandelt werden. Dabei lassen sich vier Grundmodelle herausstellen, deren Eigenschaften sich auf konkrete Kommunikationsmechanismen übertragen lassen. Wie die hier vorgestellten Kommunikationsmodelle praktisch implementiert sind, ist zunächst von untergeordneter Bedeutung. Zwei Basischarakteristika der Kommunikationsmodelle sind:

1. Ist die Kommunikation für den Sender bis zur vollständigen Abwicklung seiner Nachricht blockierend oder nicht. Soll die Kommunikation nichtblockierend sein, so muß das Nachrichtentransportsystem in der Lage sein, Nachrichten zwischenzuspeichern, um so eine Entkopplung von Sender und Empfänger zu ermöglichen.

2. Erwartet der Sender eine Bestätigung seiner Nachricht oder nicht. Im ersten Fall wird die Nachricht häufig Auftrag, im zweiten Fall Meldung genannt.

Empfangsanforderungen an Transportkanäle, die keine Nachrichten beinhalten, führen in der Regel zur Blockierung des Empfängers.

Rendezvous

Sendet der Sender eine Meldung und ist blockiert, bis diese vom Empfänger entgegengenommen wird, so spricht man von einem Rendezvous der beteiligten Kommunikationspartner. Da der Sender keine Rückmeldung erwartet, muß das Transportsystem weder eine Zwischenspeicherung der Nachricht, noch eine Identifikationsmöglichkeit des Senders durch den Empfänger vorsehen. Das Rendezvous kann z.B. sehr einfach durch einen Sender und Empfänger übergreifenden Speicher und eine Semaphore realisiert werden.

Datagramm

Das Datagrammodell erlaubt das Versenden einer Meldung, ohne daß der Sender blokkiert wird, falls die Abnahme nicht sofort erfolgt. Die Meldung selbst heißt in diesem Fall auch Datagramm und muß vom Transportsystem bis zur endgültigen Abnahme durch den Empfänger zwischengespeichert werden.

SRPC

SRPC (Synchroner Remote Procedure Call) ist ein Modell, welches die Übermittlung von Aufträgen ermöglicht. Der Sender wird bis zur Abnahme seiner Nachricht blockiert und erhält vom Empfänger eine Antwort. Die Semantik von Prozeduraufrufen erinnert an dieses Modell, und daher kommt auch sein Name.

ARPC

ARPC (Asynchroner Remote Procedure Call) blockiert den Auftraggeber zwar nicht, kann aber mittels zweier Datagramme emuliert werden, sofern die Möglichkeit für den Empfänger besteht, den Absender eines Datagramms zu identifizieren.

Leider können im Rahmen dieses Textes nicht noch weiterführende Betrachtungen erfolgen, es sei dazu an [Tanenbaum, 1992b] bzw. [Nehmer, 1993] verwiesen.

2.5 Möglichkeiten der Prozeßkommunikation und -synchronisation unter UNIX

In diesem Kapitel soll nun diskutiert werden, inwiefern sich die bis hierhin vorgestellten Konzepte zur Synchronisation und Kommunikation von Threads in realen Betriebssystemen (hier UNIX) wiederfinden lassen. Grundsätzlich schützt das Betriebsystem einzelne Prozesse voneinander und gestattet keine Beeinflussung. Mittels zur Verfügung gestellter Systemressourcen läßt sich eine Prozeßkommunikation dennoch erreichen.

Diesen Ressourcen liegen obige Konzepte zugrunde, wie jedoch nicht anders zu erwarten, wird die Verwendung vieler Methoden durch praktische und historisch gewachsene Aspekte des Betriebssystems häufig erschwert.

War die Unterscheidung zwischen Threads und Prozessen bisher eher nebensächlich, treten an dieser Stelle einige Aspekte auf, die eine Unterscheidung zwischen Threads innerhalb eines Prozesses und solchen, die auf mehrere Prozesse verteilt sind, erfordern. Erstere kann man als eng gekoppelt betrachten, während Threads innerhalb verschiedener Ausführungsumgebungen eher lose gekoppelt bzw. getrennt voneinander sind. Im folgenden wird bei einer Formulierung bzgl. „Threads" implizit von Threads innerhalb eines Prozesses ausgegangen, und eine Formulierung bzgl. „Prozessen" meint Threads innerhalb verschiedener Prozesse mitsamt den damit verbundenen unterschiedlichen Zugriffsrechten etc. Eine nahezu vollständige Übersicht aller unter UNIX möglichen Methoden zur Prozeßsynchronisation und -kommunikation wird in [Brown, 1994] gegeben. Dort sind auch praktische Programmbeispiele zu finden, die die teilweise subtilen Systemaufrufe anschaulich machen.

2.5.1 Pipes

Pipes gehören zu den klassischen Mechanismen der Interprozeßkommunikation unter UNIX. Damit ist ein unidirektionaler Kommunikationskanal gemeint, der zwischen Prozessen, die denselben Erzeuger haben, gelegt werden kann. Eine Pipe ist mit einer echten Röhre vergleichbar, da ein Prozeß auf ein Ende schreiben und ein anderer am anderen Ende von ihr lesen kann. Die typischen Einsatzgebiete der Pipes ist die Verkettung von Kommandozeilenbefehlen von der Shell. Gibt der Benutzer z.B. folgendes ein:

```
% find . -name "*sid*" -print | less
```

so ruft die Shell die beiden Programme „find" und „less" auf (genauer, sie erzeugt zwei parallel arbeitende Unterprozesse, die die Programme „find" und „less" ausführen) und verknüpft deren Standardaus- bzw. -eingabe. Der Shell-Prozeß selbst wartet solange, bis die beiden Unterprozesse terminiert sind und fährt dann mit der Eingabeaufforderung fort. Als Resultat bekommt der Benutzer alle Dateien, inkl. derer in Unterverzeichnissen, zu sehen, die in ihrem Namen die Zeichenkette 'sid' haben (Wirkung von „find") und dies in einer seiner Lesegeschwindigkeit angepaßten Form (Wirkung des Pagers „less"). Für die beiden Programme „find" und „less" ist die Verknüpfung durch die Pipe weitestgehend transparent. Lediglich die Eigenschaft von „less", im Falle fehlender Dateinamen als Kommandozeilenargumente von der Standardeingabe zu lesen, mußte vorausgesetzt werden, denn genau dieser Dateideskriptor (die Standardeingabe wird unter UNIX als abstrakte Datei aufgefaßt) wurde vom Aufrufer verbogen (auf die Pipe) und vererbt.

Während UNIX-Pipes hervorragend dazu geeignet sind, zwei Prozesse, die sich nicht kennen, miteinander zu verbinden (vorausgesetzt diese halten einige Konventionen ein), sind sie für weitergehende Kommunikationsaufgaben weniger geeignet und zwar aus folgenden Gründen:

1. Die Verwaltung der Kommunikationsressource „Pipe“ obliegt einem gemeinsamen Vaterprozeß.

2. Da Pipes allesamt vor der Erzeugung der an der Kommunikation beteiligten Prozesse gebildet werden müssen, besteht keine Möglichkeit, nachträglich noch Verbindungen zu legen. Das Kommunikationsnetz ist also sehr statisch.

3. Sofern der Vaterprozeß keine besonderen Privilegien besitzt, ist eine Verbindung nur zwischen Prozessen der gleichen UserID möglich.

4. Pipes können nicht über ein Netzwerk agieren

Punkte 1, 2 und 3 werden mit den sog. „Named Pipes“ eliminiert. Diese sind Erweiterungen und zwischen normalen Pipes und Dateien angesiedelt. Eine „Named Pipe“ kann wie eine Datei erzeugt und von Prozessen geöffnet werden, ist daher im Dateiverzeichnis enthalten und von anderen Prozessen ansprechbar. Wenn mehrere Prozesse auf sie schreiben oder lesen, sind weitere Maßnahmen zur Synchronisation erforderlich, da eine Nachricht durch die Aktionen mehrerer Prozess zerstückelt werden kann. Trotzdem sind „Named Pipes“ ein probates Mittel, eine einfache Kommunikation zu realisieren, und haben die angenehme Eigenschaft, unter jedem Programmiersystem genutzt werden zu können, welches das Öffnen, Lesen und Schreiben auf Dateien erlaubt, also quasi von allen. Damit entsprechen „Name Pipes“ schon in erheblichem Maße den Anforderungen, die von den Zielvorgaben in Kapitel 1 gefordert wurden. Mit etwas Unterstützung kann auf ihrer Basis weitergehende Kommunikation realisiert werden, wie noch beschrieben werden soll.

2.5.2 Signale

Eine weitere, klassische Methode unter UNIX Prozeßkommunikation zu realisieren, sind *Signale*. Diese können als 1–Bit Nachrichten aufgefaßt werden, womit bereits begründet ist, warum sie nur zur Übermittlung von nicht parametrisierten Informationen geeignet sind. Sender und Empfänger eines Signals müssen sich über dessen Interpretation vorher geeinigt haben.

Signale werden als Softwaretrapmechanismus eingesetzt und hauptsächlich vom Betriebssystem versandt, wenn besondere Ereignisse (Speicherverletzung etc.) eintraten. Aus diesem Grund haben alle Signale bereits Standardinterpretationen, die in den meisten Fällen eine Terminierung des Empfängers nach sich ziehen. Es spricht jedoch nichts dagegen, diese Standardinterpretationen zu verändern und den Signalmechanismus zu eigenen Zwecken zu gebrauchen.

Der Benutzer kommt mit Signalen häufig in Berührung, wenn er einen außer Kontrolle geratenen Prozeß (hier den mit der Nummer 127) beenden muß. Er tut dies, indem er von der Shell eingibt:

```
% ps
  PID TTY STAT  TIME COMMAND
   59 v01 SW    0:01 (tcsh)
```

```
   75 v01 SW    0:00 (startx)
   76 v01 SW    0:00 (xinit)
   79 v01 SW    0:00 (sh)
  101 v01 S     0:08 fvwm
  127 pp1 S     0:01 ich-bin-wild -very
  150 pp1 S     0:00 (tcsh)
  217 pp1 R     0:00 ps
% kill -9 127
```

Dabei ruft das „kill" Kommando lediglich den Systembefehl KILL auf und schickt dem Prozeß mit der Nummer 127 das Signal -9 (SIGKILL). Das Betriebsystem speichert daraufhin den Kontext des betroffenen Programms und ruft eine Behandlungsfunktion auf, die mit diesem Signal assoziiert ist. Im Falle des Signals SIGKILL ist das immer eine Funktion, die den Prozeß terminiert und sich nicht verändern läßt. Für andere Signale kann der Programmierer selbst eine Behandlungsroutine im Programmcode spezifizieren, welche dann aktiviert wird.

Signale sind, wie bereits erwähnt, nicht dazu geeignet, allgemein Nachrichten zu übermitteln[2]. Allerdings gewinnt man mit ihnen eine Möglichkeit, die alle anderen Kommunikationsmechanismen nicht bieten, nämlich, Prozesse asynchron, also ohne deren Wunsch, mit Nachrichten zu versorgen. Dies ist insbesondere beim Auftreten von seltenen Ereignissen hilfreich, müßten sich Prozesse doch ansonsten um die Zustellung solcher Informationen regelmäßig selbst kümmern, was nicht nur unnatürlich ist, sondern nahezu unmöglich, denn ein solches Ereignis kann nach jedem Speicherzugriff oder jeder Maschinenoperation eintreten.

Signale haben jedoch einige Einschränkungen und vor allem einen großen Fallstrick. Zum einen lassen sie sich von „normalen" Benutzern nur innerhalb von Prozeßgruppen (das sind Prozesse, die derselben UserID angehören) versenden, zum anderen dies auch nur lokal auf einem Rechner. Tückisch ist jedoch, daß durch die Definition alternativer, nicht den Prozeß terminierender Behandlungsroutinen ein Programm mehrere Eintrittspunkte hat. D.h., daß neben dem normalen Berechnungsablauf jederzeit ein zweiter treten kann. Teilt die Behandlungsroutine, die innerhalb des Adreßraumes des Prozesses liegt, Variablen mit dem eigentlichen Hauptprogramm, so zieht dies alle Effekte nach sich, die bereits in 2.2 angesprochen wurden. Die Behandlungsroutine und das Hauptprogramm können als konkurrierende Threads angesehen werden. Aus diesem Grund vermeidet man den Zugriff auf globale Variablen von der Behandlungsroutine heraus. Kaskadierte, zu einem Signal gehörende Behandlungsroutinen können in der Regel nicht miteinander in Konkurrenz treten, da das Betriebssystem dasselbe Signal verzögert, wenn eine zugehörige Behandlungsroutine bereits aktiv ist. Als reiner Kommunikationsmechanismus sind Signale normalerweise zu langsam, da mit ihrem Versand auch immer das Neuaufsetzen eines Stacks für den Empfänger einhergeht, was eine teure Operation darstellt.

[2] Obwohl manche Signalmechanismen so erweitert wurden, daß noch einige Parameter mitübertragen werden können, z.B. solche den POSIX.4 Richtlinien folgenden, wie die der System V UNIX Variante SOLARIS von Sun.

2.5.3 System V IPC

Grundsätzlich ist es einem normalen UNIX–Prozeß nicht möglich, auf den Speicher eines anderen Prozesses zuzugreifen, selbst wenn dieser der gleichen UserID angehört. Nur so kann gewährleistet werden, daß ein unkontrollierter Prozeß andere nicht stört bzw. zum Abstürzen bringt. Durch diesen und den Dateischutzmechanismus ist auch die Problematik der Computerviren unter UNIX bei weitem nicht so heikel wie bei Systemen, die einer anderen „Philosophie" folgen.

Trotz allem kann die Kommunikation innerhalb eines Rechnersystems mittels gemeinsamen Speichers zu sehr eleganten und performanten Lösungen beitragen[3], und aus diesem Grunde wurden mit Einführung der System V UNIX Variante Systemfunktionen zur Nutzung von „Shared Memory", Semaphoren und „Message Queues" hinzugenommen. Diese Mechanismen sind unter dem Namen *System V IPC (Interprocess Communication Facility)* bekannt und gehören mittlerweile zum Standardfunktionsumfang fast aller UNIX–Derivate (ggf. mit entsprechenden Kernelerweiterungen).

Die System V IPC's werden ähnlich Dateien verwaltet, d.h. beim Kreieren bzw. Öffnen werden sie mit einem Schlüssel assoziiert. Zum Lesen und Schreiben[4], bzw. Alternieren im Falle von Semaphoren benötigt man einen Identifier, der von den entsprechenden Systemfunktion zum Öffnen und Erzeugen[5] zurückgeliefert wird.

Zur Verwendung dieser Kommunikationsmechanismen ist noch zu sagen, daß sich „Shared Memory" zunächst einmal nicht zum Speichern verzeigerter Strukturen eignet, da die Adresse eines solchen Segments nicht notwendigerweise in jedem Prozeß dieselbe ist. Durch Setzen entsprechender Korrekturtabellen an den Beginn des „Shared Memory", verbunden mit etwas Zeigerarithmetik läßt sich dieses Manko jedoch beseitigen. System V Semaphoren und deren Behandlungsfunktionen bieten eine größere Mächtigkeit als die in 2.3.3 vorgestellten. Zum einen lassen sich mit einer Operation ganze Arrays von Semaphoren verändern, zum anderen ist die Veränderung nicht nur auf das De- und Inkrementieren beschränkt, sondern für beliebige Werte zugelassen. Ein ausgefeiltes Management von Adjustierungen pro Prozeß und Semaphore ermöglicht es, beim Absturz eines Prozesses die Semaphorenwerte wieder dahingehend zu korrigieren, daß andere Prozesse weiterarbeiten können. Für weitere Informationen siehe [Brown, 1994].

Alle System V IPC's bleiben auch nach Terminierung der erzeugenden Prozesse als Ressource erhalten, müssen also explizit beseitigt werden. Da sie mit ähnlichen Schutzbits wie Dateien verwaltet werden, ist ihre Benutzung relativ einfach, man sollte jedoch etwas Erfahrung im Umgang mit Prozeßsynchronisation haben.

Die jedoch nur halbherzig vorgenommene Integration in das UNIX-Dateikonzept ist zugleich auch einer der größten Kritikpunkte an den System V IPC's. Die Identifier, über die "Shared Memory" etc. adressiert werden, sind keine Deskriptoren. Aus diesem

[3] Insbesondere das Senden einer Nachricht von einem Sender zu mehreren Empfängern (Multicasting) ist über gemeinsamen Speicher sehr effizient realisierbar, da Mehrfachübertragungen nicht notwendig sind.

[4] „Shared memory" wird via `shmat` in den Speicherraum eines Prozesses eingebunden, Semaphoren mittels `semop` verändert und „Message Queues" mittels `msgsnd` und `msgrcv` beschrieben und ausgelesen.

[5] semget, shmget und msgget

Grunde ist es nicht möglich, Standardfunktionen zum Umgang mit Deskriptoren, wie `select`, auch hierfür zu benutzen. Programme, die sich der System V IPC's bedienen, müssen diese also immer gesondert behandeln. Im Rahmen der POSIX.4 Definitionen ([Gallmeister, 1995]) wurden Modelle vorgeschlagen, die dieses Manko beseitigen.

Weitere Nachteile der System V IPC's ergeben sich dadurch, daß es sehr wenig Unterstützung bei der Verwaltung der einzelnen Ressourcen gibt, z.B. müssen „Shared Memory" Keys jedem Prozeß bekannt sein, der ein solches Segment in seinen Adreßraum einblenden will. Desweiteren sind sie nicht netzwerkfähig[6]. In 2.5.4 werden noch Methoden untersucht, die weitaus komfortablere Systemfunktionen zur Verfügung stellen und auch bei der dynamischen Erzeugung und Verwaltung besser unterstützen. Trotzdem bieten die „System V IPC's" durch ihren dateisystemähnlichen Schutzmechanismus eine einfache Möglichkeit, sichere und effiziente Kommunikation zu realisieren. Insbesondere lokale Client-Server Strukturen (siehe 2.6) kommen ohne aufwendige Authentifikationsmechanismen aus, da unauthorisierter Gebrauch von Kommunikationskanälen durch das Betriebssystem verhindert werden kann.

2.5.4 TCP/IP und BSD Sockets

Ein Kommunikationsmechanismus, der die bisherige Beschränkung auf lokale Prozesse aufhebt, ist der der „BSD Sockets". Dabei handelt es sich um eine, mit BSD UNIX Version 4.1 eingeführte, allgemeine Schnittstelle zu Netzwerksoftware, d.h. Protokollsoftware. Dies ist in den meisten Fällen die TCP/IP–Protokollsuite, weswegen zunächst darauf eingegangen werden soll.

IP (Internet Protocol) ist der Schicht 3 (Vermittlungsschicht) des OSI–Schichtenmodells (siehe auch [Tanenbaum, 1992a] und Abbildung 7) zuzuordnen, sorgt also für die Auswahl des Paketleitwegs vom Ursprungs- zum Zielort[7]. Dies sind jeweils Prozesse, die auf verschiedenen Rechnern ablaufen können. Adressiert werden sie durch Angabe einer IP–Adresse (einer 32 Bit Nummer, die innerhalb des Internets einen Rechner eindeutig spezifiziert) und einer sogenannten Portadresse (einer 16 Bit Nummer, die den beteiligten Prozeß auf dem jeweiligen Rechner spezifiziert, siehe auch Abbildung 8).

TCP (Transmission Control Protocol) und UDP (ein weiteres, zur TCP/IP-Suite gehörendes Protokoll) sind auf der Schicht 4 (Transportschicht) einzuordnen. Ihre Aufgabe ist es, Daten höherer Schichten entgegenzunehmen, für IP aufzubereiten und diesen Schichten den Eindruck einer Punkt zu Punkt Kommunikation zu geben, obwohl die Datenpakete tatsächlich über mehrere Rechner „geroutet" werden. Der Unterschied zwischen beiden Protokollen ist, daß TCP einen *verbindungsorientierten*, *streamartigen* und *sicheren* Dienst anbietet. UDP hingegen

[6] Neuere Ansätze zur Standardisierung werden zur Zeit erarbeitet, die dann auch die Verteilung auf mehrere Rechner ermöglichen sollen; zur Zeit muß man jedoch viel „von Hand" erledigen.

[7] Eigentlich beinhaltet das IP auch einen Kontrollmechanismus, der den korrekten Empfang jedes Pakets überprüft. Diese eigentlich zur Schicht 2 (Sicherungsschicht) gehörende Eigenschaft macht eine genaue Einordung der TCP/IP- Protokollsuite unmöglich, sie ist allerdings auch 10 Jahre älter als das *OSI–Schichtenmodell*.

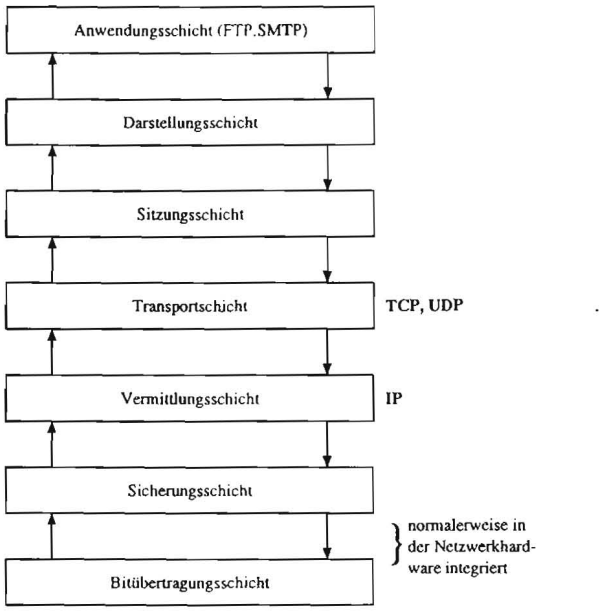

Abbildung 7: Der TCP/IP Protokollstack eingeordnet im OSI–Schichtenmodell

ist *datagrammorientiert*, garantiert jedoch nicht den Empfang aller Datenpakete. Dafür werden Paketgrenzen eingehalten und ein höherer Durchsatz erreicht. UDP eignet sich immer dann, wenn das darunterliegende Netzwerk relativ sicher ist (z.B. Ethernet) oder mehrere Prozesse miteinander kommunizieren sollen (Multicast). Desweiteren, wenn eine Verzögerung der zu übertragenden Daten schlimmer wiegt als deren inkorrekte Übertragung, z.B. Audiodaten, die in Echtzeit übertragen werden müssen.

Die Kommunikationspartner von IP sind also bzgl. des darunterliegenden Netzwerks benachbarte Rechner, die die Pakete ggf. weiterleiten. Via TCP und UDP hingegen kommunizieren die beiden Prozesse, die die Endpunkte der Übertragung darstellen. Zur TCP/IP–Protokollsuite gehören noch weitere, in höheren Schichten anzusiedelnde Protokolle, wie „FTP“ (File Transmission Protocol) und „SMTP“ (Simple Mail Transfer Protocol), die hier jedoch keine Rolle spielen.

Die Definition der TCP/IP–Protokollsuite legt lediglich einen Standard für den Aufbau und das Verhalten entsprechender Protokollsoftware fest. Über die Schnittstelle zwischen ihr und dem Benutzer werden hingegen keine Aussagen gemacht. Sie bleiben den Entwicklern der Betriebssysteme vorbehalten. Dabei stellt sich für diese immer die Frage, ob neue Funktionalitäten durch Erweiterung existierender oder durch Einführung neuer Systemfunktionen realisiert werden sollen. Die Entwickler der BSD UNIX Variante verfolgten in der Version 4.1 beide Wege. Sie führten ganz allgemein als Schnittstelle zu Netzwerksoftware (die in der Regel zum Kernel gehört) sogenannte Sockets ein. Diese sind den in allen UNIXen verwandten Dateideskriptoren sehr ähnlich

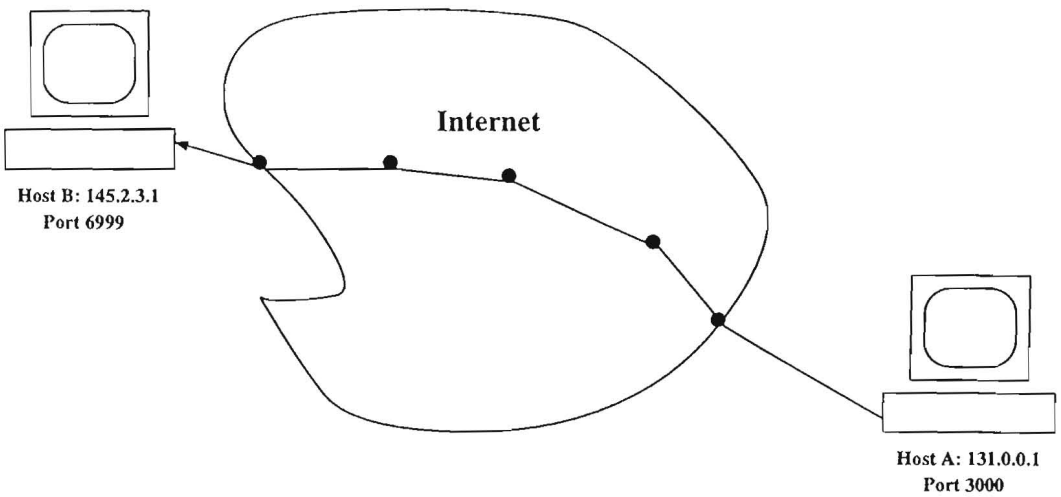

Abbildung 8: Kommunikation zweier Prozesse mittels TCP/IP

und nicht auf die Verwendung mit TCP/IP beschränkt. Zwar existieren rund um die *Sockets resp. deren erweiterter Möglichkeiten eine Reihe Systemfunktionen, die* für normale Deskriptoren nicht vorhanden sind (und auch keinen Sinn machen würden), doch lassen sich nahezu alle für Dateideskriptoren verfügbaren Funktionen auch auf Sockets anwenden. Eine eingehendere Behandlung entprechender Systemfunktionen würde den Rahmen dieses Textes bei weitem überschreiten, Interessenten seien an das hervorragende Buch von [Comer und Stevens, 1993] verwiesen. Auf die Problematik des Rendezvous zweier Kommunikationspartner wird in 2.6 eingegangen.

2.6 Ressourcenverwaltung mittels Client-Server Struktur

Da ein Socket, ähnlich den UNIX Pipes, ein Objekt ist, welches keinen Namen besitzt und ohne den erzeugenden Prozeß nicht vorhanden sein kann, besteht ein grundlegendes Problem bei der Kommunikation über Sockets darin, daß für beide Kommunikationspartner ein Rendezvous hergeleitet werden muß. Während dies bei den Pipes mittels des gemeinsamen Elternprozesses geschieht, haben sich die Entwickler der Socket- Schnittstelle etwas anderes einfallen lassen. Sie unterscheiden zwischen *aktiven* und *passiven* Sockets. Durch Passivierung eines Sockets kann man diesen "empfangsbereit" machen, ohne daß bereits ein Kommunikationspartner festgelegt werden muß. Aktive Sockets hingegen sollen eine Verbindung zu einem passiven Socket etablieren[8]. Zeitlich müssen die passiven Sockets vor den entsprechenden aktiven erzeugt werden. Diese Unterscheidung der beiden Kommunikationsenden zieht auch ein etwas anderes Verhalten der beteiligten Prozesse nach sich. Der Prozeß, der mittels eines passiven Sockets eine Verbindung anbietet, soll im folgenden *Server*, der der mittels eines aktiven Sockets eine

[8]Man beachte, daß diese Unterscheidung nichts mit les- und schreibbar zu tun hat. Sockets sind grundsätzlich bidirektional.

Verbindung aufbaut, *Client* genannt werden.

Eine andere Terminologie für Client und Server erhält man durch die Unterscheidung, wer einen Dienst anbietet, und wer ihn nutzt. Trotzdem wird man feststellen, daß beide Terminologien die Kommunikationspartner häufig gleich benennen. Der zeitliche Ablauf eines Verbindungsauf- und -abbaus wird in Abbildung 9 gezeigt. In Klammern stehen dabei die jeweils zur Verwendung kommenden Systemfunktionen.

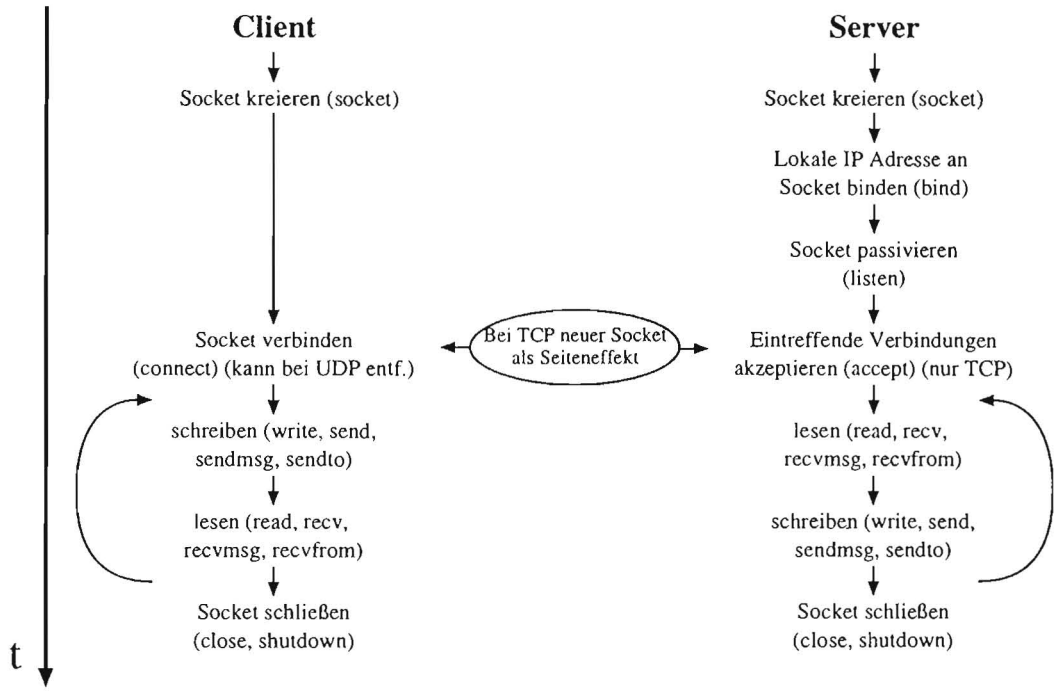

Abbildung 9: Sequenz eines Verbindungsaufbaus mittels Sockets

Eine Applikation, die auf einer Client-Server Struktur basiert, heißt auch verteilte Applikation. Dabei werden Client und Server als zusammengehörig gesehen. Ein typisches Problem, das mit einer verteilten Applikation gelöst wird, ist gegeben, wenn eine Ressource von mehreren Prozessen gleichzeitig genutzt werden soll und durch den Server verwaltet wird. Server verwalten auch physikalische Geräte und bilden sie in Dienste ab, wobei meistens auf Ortstransparenz geachtet wird.

Während Clients häufig ganz „normale" Programme sind, werden an Server mehr Ansprüche gestellt, da sie stabiler laufen müssen, ggf. mehrere Clients bedienen sollen und die von ihnen verwaltete Ressource vor unauthorisiertem Zugriff geschützt werden muß. Dabei haben sich einige Designprinzipien herausgebildet (siehe auch [Comer und Stevens, 1993]), die maßgeblich den Aufwand der Entwicklung eines Servers beeinflussen. Man hat grundsätzlich die Möglichkeit, einen Server zustandsbehaftet oder zustandslos, konkurrierend oder iterativ und verbindungsorientiert oder verbindungslos zu entwerfen. Die Diskussion dieser Prinzipien wird auf Kapitel 4 verschoben.

Ein grundsätzliches Problem beim Verbindungsaufbau zwischen Client und Server besteht darin, daß ein Client wissen muß, an welchem Port der Server seinen passiven Socket eingerichtet hat. Aus diesem Grund wurden im Internet einige Ports für bestimmte Server reserviert (well-known ports), die festgelegte Standarddienste anbieten. Der entsprechende Server bindet bei seiner Initialisierung einen Socket an seine zugewiesene Adresse und wartet dort auf Verbindungen. Je nach Eigenschaft dieses Servers (verbindungsorientiert oder nicht) richtet er bei eingehenden Verbindungswünschen einen neuen Socket ein und vereinbart einen neuen freien Port mit dem Client, um den reservierten Port für andere Clients wieder freizumachen. Die Zuordnung zwischen Diensten und Ports übernehmen dabei verschiedene Systemfunktionen, die ihre Information aus Konfigurationsdateien beziehen und vom Systemverwalter gewartet werden. Will ein Client mit einem Server kommunizieren, dessen Dienst er kennt, aber nicht seinen Port, so kann er diesen über die Funktion `getservbyname` aus der entsprechenden Konfigurationsdatei (hier `/etc/services`) ermitteln[9]. Diese Methode bietet zwar eine gewisse Flexibilität, eignet sich jedoch weder für Server, die an wechselnden Ports auf Verbindungen warten, noch dazu, Zuordnungen zwischen Diensten und Ports zu ändern, denn beim Aufruf eines Dienstes auf einem entfernten Rechner werden die Konfigurationsdateien des lokalen Rechners ausgelesen, die dann nicht mehr konform sind.

[9]Bei NIS-verwalteten (Network Information Service) Systemen wird ggf. eine NIS-Anfrage ausgelöst und die Information durch den NIS-Server geliefert.

3 Einführung in T

T ist eine C-Programmbibliothek zur Unterstützung des Entwurfs von Programmen mit mehreren Kontrollflüssen. Die Entwicklung wurde zum einen durch ein Praktikum bei der Arbeitsgruppe von Prof. Dr. J. Nehmer[10] motiviert. Zum anderen zeigten auch frühere Programmierarbeiten am DFKI, daß die Realisierung von Nebenläufigkeit innerhalb eines Prozesses zu sehr unübersichtlichem Code führen kann und viele Schwierigkeiten in immer wiederkehrender Form gelöst werden müssen, weswegen sich die Entwicklung einer soliden und getesteten Programmierbasis anbot. Leider existiert z.Zt. keine offizielle Dokumentation des von Dipl. Inform. Volker Hübsch[11] entwickelten TP (Thread Package), dessen Funktionalität die Basis von T bildet. Aus diesem Grund soll die hier gegebene Einführung etwas großzügiger ausfallen.

In den folgenden Abschnitten wird zunächst eine kurze Einführung in T gegeben, daran schließt sich eine vollständige Befehlsreferenz im Abschnitt 3.8 an.

3.1 Das Objektmodell von T

Eine sinnvolles Modell für die Programmabarbeitung wurde bereits in Abschnitt 2.1.1 mit dem Begriff „Prozeß“ eingeführt, der als Ausführungsumgebung samt mindestens einem Thread dargestellt wurde. Im UNIX Prozeßmodell ist dies genau ein Thread, weitere Threads kennt das Betriebssystem nicht, sie werden über Programmierbibliotheken verfügbar gemacht, zu denen auch T gehört. Die zentralen Abstraktionseinheiten von T sind das **Objekt** und die **Nachricht**, mit der Objekte untereinander kommunizieren können.

Objekte besitzen eigene Threads, die beim Empfang von Nachrichten aktiviert werden. Der Startcode eines Threads wird durch den *Standardhandler* beschrieben, der Bestandteil eines Objekts ist. Standardhandler sind dabei sind vom Typ `T_Code`, der definiert ist durch:

```
typedef int (*T_Code)(void *environment,T_Object *object,
                      long id, long value)
```

Ein Objekt besitzt außerdem eine Nachrichtenwarteschlange, in der alle noch nicht verarbeiteten Nachrichten in der Reihenfolge des Empfangs festgehalten sind. Sie soll im folgenden mit *IN-Queue* bezeichnet werden. Das erste Element der *IN*-Queue heißt *aktuelle IN-Nachricht*, sie wird beim Aktivieren eines Objekts als Parameter des Standardhandlers übergeben.

Nachrichten, die von einem Objekt entgegengenommen wurden, sind normalerweise nach Abarbeitung des Standardhandlers verloren. Um sie für spätere Aktivierungen aufzubewahren, ist einem Objekt eine weitere Nachrichtenwarteschlange zugeordnet, in der bereits empfangene Nachrichten abgelegt werden können. Sie soll im folgenden mit *SAVE-Queue* bezeichnet werden. Das erste Element dieser Warteschlange wird *aktuelle*

[10] http://www.uni-kl.de/AG-Nehmer/

[11] http://www.uni-kl.de/AG-Nehmer/panda/users/huebsch.id.html

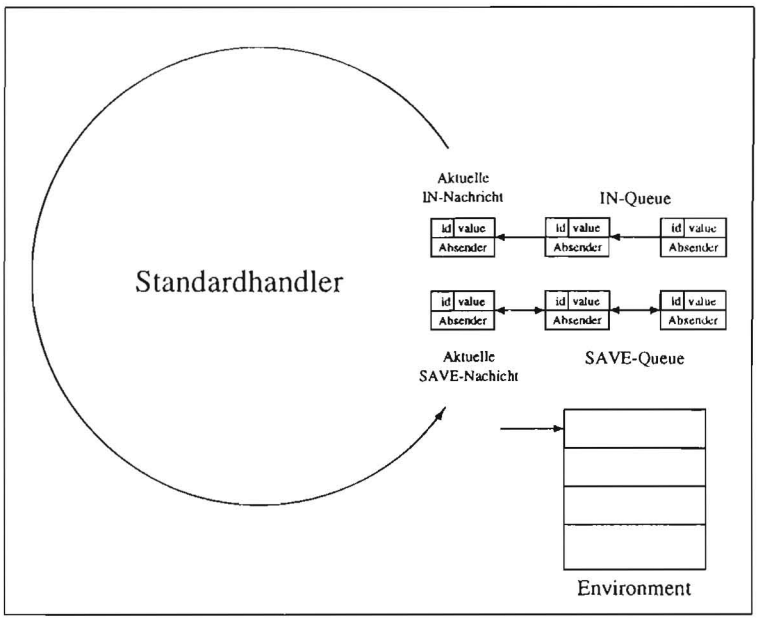

Abbildung 10: Komponenten eines T Objekts

SAVE-Nachricht genannt. Sie ist erst nach Aufruf einer Funktion, die eine Nachricht aus der *SAVE-Queue* kettet, definiert.

Um objektspezifische Daten anlegen und für spätere Aktivierungen aufbewahren zu können, hat ein Objekt eine *Umgebung*, die dem Standardhandler bei Aktivierung übergeben wird. Die Umgebung wird bei Objekterzeugung angelegt, kann jedoch zu jedem Zeitpunkt modifiziert werden.

Die zweite Abstraktionseinheit von T ist die Nachricht. Durch sie können Objekte miteinander kommunizieren und sich dabei gegenseitig aktivieren. Nachrichten sind von der Form (`id`,`value`,`object`), wobei `id` und `value` beliebige `long`-Werte beschreiben und `object` den Absender identifiziert. Nachrichten können explizit empfangen werden oder stellen einen Teil der Parameter des Standardhandlers eines Objekts dar (s.o.).

Eine Nachricht wird immer an das Ende der *IN*-Queue des Empfängers angefügt und bewirkt dabei ggf. die Aktivierung des Objekts und damit den Aufruf des Standardhandlers. Dessen Rückgabewert entscheidet darüber, ob das Objekt aus dem System entfernt wird, oder ob es bestehen bleibt und weitere Nachrichten empfangen kann. Diese Form der Terminierung eines Objekts soll **„ordentliche" Terminierung** genannt werden. Demgegenüber steht die **„erzwungene" Terminierung**, mit der ein Objekt ein anderes unter bestimmten Voraussetzungen terminieren kann. Threadterminierung und die damit verbundenen Probleme sind Gegenstand von Abschnitt 3.5.

3.2 Zyklus eines mit T entworfenen Programmes

Direkt nach dem Start eines mit T gebundenen Programmes existiert genau ein Objekt, welches auch ohne Zustellung einer Nachricht vom System aktiviert wird. Dies ist das *Root*-Objekt, dessen Standardhandler zu diesem Zeitpunkt undefiniert ist. Der Startcode dieses Objekts ist die Funktion `main`, die auch in herkömmlichen C-Programmen den Eintrittspunkt zur Abarbeitung darstellt.

Die Funktion `main` nimmt somit auch im Rahmen von T eine Sonderstellung ein, denn nach ihrer Terminierung terminiert auch der gesamte Prozeß, unabhängig davon, ob noch weitere Threads rechenbereit sind. Um ein solches Verhalten zu unterbinden, kann man dem *Root*-Objekt einen Standardhandler zuweisen, was auf jeden Fall notwendig ist, wenn man Nachrichten an dieses Objekt schicken möchte.

Durch Bindung des Standardhandlers erfährt das *Root*-Objekt eine gleiche Behandlung wie jedes andere Objekt auch, d.h. in Abhängigkeit vom Rückgabewert dieses Standardhandlers terminiert es nicht, sondern wird ggf. als nicht rechenbereit markiert und wartet auf den Empfang einer Nachricht. Nach Terminierung des Standardhandlers wird im Kontext des *Root*-Objekts wieder `main` bzw. eine davon direkt oder indirekt aufgerufene Funktion weiterabgearbeitet.
Anmerkung: Die Funktion `main` bekommt die Kommandozeilenargumente und das Prozeßenvironment als Parameter übergeben.

3.3 Erzeugung, Kommunikation und Synchronisation von Objekten

Ausgehend vom *Root*-Objekt entsteht Nebenläufigkeit durch Erzeugung weiterer Objekte innerhalb eines Prozesses. Dies geschieht mit den beiden Funktionen `T_new_process` und `T_new_sigprocess`. Bei der Erzeugung werden der Standardhandler, der Speicherbereich des Stacksegments, seine Größe, die Prioritätsklasse, die Umgebung, sowie die Schedulingklasse gesetzt (bei `T_new_sigprocess` zusätzlich noch die Nummer des Signals, auf das das Objekt reagieren soll). Beide Funktionen liefern im Erfolgsfall einen Verweis zurück, unter dem das neue Objekt angesprochen werden kann.
T kennt vier Zustände, in denen sich Objekte befinden können. Diese lauten:

1. Zustand **„Rechnend“**;
   Objekte dieses Zustands werden gerade abgearbeitet. Bei Rechnern mit nur einem Prozessor kann es nur ein Objekt im Zustand „Rechnend“ geben.

2. Zustand **„Bereit“**;
   Objekte in diesem Zustand sind rechenbereit und warten auf ihre Aktivierung.

3. Zustand **„Blockiert“**;
   Objekte in diesem Zustand sind nicht rechenbereit und warten auf den Empfang einer Nachricht.

4. Zustand **„Auf Antwort wartend“**;
   Objekte in diesem Zustand sind ebenfalls nicht rechenbereit, warten jedoch nicht auf den Empfang einer beliebigen Nachricht sondern auf eine spezielle Antwort auf einen zuvor abgesetzten Auftrag.

Unmittelbar nach seiner Erzeugung ist jedes Objekt im Zustand „Blockiert“ (außer dem *Root*-Objekt). Ein blockiertes Objekt wechselt in den Zustand „Bereit “, sobald eine Nachricht an der *IN*-Queue anliegt. Wann das Objekt nun aktiviert wird, hängt von der Schedulingklasse des gerade aktiven Objekts und von seiner Priorität ab.

Beispiel:

```
#include <TSystem.h>

#define DUMMY 0
#define GO 1
#define TERMINATE 2

static T_Object *hup_handler;

int sig_hup_handler (void *env, T_Object *object, long id, long value)
{
  T_send(Root,DUMMY,TERMINATE);
  return -1;
}

int control (void *env, T_Object *object, long id, long value)
{
  switch (value) {
  case GO:
    return 0; /* nichts tun */
  case TERMINATE:
    return -1; /* zurueck zu main */
}

void main (int argc, char *argv[])
{
  ...
  hup_handler = T_new_sigprocess(SIGHUP,sig_hup_handler,
                                 (char*)0,T_STACKSIZE, Memory,
                                 T_COOPERATIVE,T_SYSTEM_PRIORITY,
                                 (void *)0,
                                 "SIGHUPHANDLER");
  T_send(T_self(), DUMMY,GO);
  T_bind(control, (void *) 0);
  /* hierher nach neg. Rueckgabewert von 'control' */
  return 0
}
```

Das obige Beispiel zeigt die Erzeugung eines neuen Signalthreads für das Signal `SIGHUP`,

dessen Standardhandler an die Funktion `sig_hup_handler` gebunden wird. Diese Erzeugung wird vom `Root`-Objekt vorgenommen, in dessen Kontext zunächst die Funktion `main` abarbeitet wird. Im weiteren Verlauf wird jedoch der Standardhandler von `Root` mittels `T_bind` an `control` gebunden wird. Da mit dieser Bindung ummittelbar ein Rescheduling einhergeht, sendet sich das `Root`-Objekt zuvor selbst eine Aktivierungsnachricht (die Funktion `T_self` liefert immer einen Objektverweis auf sich selbst). Obiges Programm veranlaßt ein Warten auf das Eintreffen des Signals `SIGHUP`, woraufhin alle Objekte und damit der gesamte schwergewichtige Prozeß terminieren.
Objekte können miteinander kommunizieren, wobei grundsätzlich zwischen nachrichtenorientierter- und auftragsorientierter Kommunikation unterschieden werden muß[12]. Für erstere ist die Funktion `T_send` zuständig. Hiermit kann eine Nachricht an ein beliebiges Objekt (auch an sich selbst!) verschickt werden, ohne daß der Absender blockiert wird, um auf eine Antwort zu warten. Die Nachricht wird lediglich in die *IN*-Queue des Empfängers eingetragen und dieser, sofern nicht schon geschehen, in den Zustand „Bereit" versetzt. Für die auftragsorientierte Kommunikation gibt es die Funktion `T_call`, die den Auftrag ebenfalls in die *IN*-Queue des Empfängers einträgt und diesen ggf. in den Zustand „Bereit" versetzt. Zusätzlich jedoch wird der Absender blockiert, bis der Empfänger den Auftrag entweder mittels `T_reply` quittiert oder gar nicht auf ihn reagiert, d.h. ihn empfängt, aber nicht weiter bearbeitet. Sichert der Empfänger den Auftrag in der *SAVE*-Queue, oder leitet er ihn mittels `T_forward` an ein anderes Objekt weiter, so bleibt die Blockierung bestehen. Die Zustellung der beiden Nachrichtentypen verläuft durch Übergabe der Nachrichteninhalte als Parameter des Standardhandlers. Zusätzlich gibt es aber auch die Möglichkeit Nachrichten explizit durch Aufruf der Funktion `T_receive` zu empfangen, ohne die Standardhandlerfunktion zu beenden. Dabei kann gewählt werden, ob bei nicht vorhandener Nachricht gewartet oder fortgefahren werden soll. Es sei an dieser Stelle jedoch erwähnt, daß eine `T_send/T_receive` Kombination keinen Ersatz für ein `T_call` darstellt, da nicht garantiert ist, daß die empfangene Nachricht eine Antwort auf einen zuvor abgesetzten Auftrag ist. `T_call` hingegen kehrt erst bei „passender" Antwort bzw. „nicht mehr zu erwartender" Antwort zurück.

Neben dem Zweck der Kommunikation ist `T_call` auch die wichtigste Funktion zur Synchronisation. Beantwortet ein Objekt einen gestellten Auftrag nicht, so wird der Auftraggeber trotzdem wieder freigegeben. Dies geschieht jedoch erst, wenn der Auftrag vom Adressaten empfangen wurde (also aktuelle *IN*-Nachricht war) und entweder von dessen Standardhandler abgearbeitet oder durch ein `T_receive` als aktuelle *IN*-Nachricht verdrängt wurde. Es ist jedoch nicht garantiert, daß nach Abarbeitung des *Standardhandlers des adressierten Objekts der Auftraggeber der nächste aktive Thread* wird, da dies von der Prioritäts- und Schedulingklasse abhängt. Durch geeignete Wahl dieser Werte kann man ein solches Verhalten jedoch erreichen (siehe Abschnitt 3.4).

[12] Bei beiden Kommunikationsarten hat die Nachricht die gleiche Struktur. Sie besteht aus zwei `long`-Werten zzgl. dem Verweis auf den Absender.

Die *SAVE*-Queue

Neben der *IN*-Queue für den Nachrichtenempfang existiert eine weitere Nachrichtenwarteschlange, die *SAVE*-Queue. Ihre Aufgabe besteht darin, empfangene Nachrichten zu sichern und für eine spätere Bearbeitung aufzubewahren. Diese Möglichkeit benötigt man z.B. für die Programmierung von Monitoren (siehe Abschnitt 2.3.4). Funktionen zum Umgang mit der *SAVE*-Queue sind `T_save`, `T_resave`, `T_read_saved`, `T_reply_saved` und `T_forward_saved`. Die ersten beiden dienen dem Sichern einer Nachricht, wobei `T_save` die aktuelle *IN*-Nachricht und `T_resave` die aktuelle *SAVE*-Nachricht (sofern vorhanden) sichert. Im Gegensatz zur *IN*-Queue, ist die *SAVE*-Queue eine **geordnete** Warteschlange. Alle Nachrichten werden nach ihrem `value`-Wert aufsteigend sortiert. Zusätzlich erlaubt die Funktion `T_read_saved` nicht nur das Auslesen des ersten Elements der *SAVE*-Queue, sondern bietet die Möglichkeit des gezielten Auslesens durch Angabe verschiedener Suchkriterien. Die Funktionen `T_reply_saved` und `T_forward_saved` dienen dem gleichen Zweck wie jene ohne die Endung `-saved`, beziehen sich jedoch nicht auf die aktuelle *IN*-, sondern die aktuelle *SAVE*-Nachricht.

3.4 Schedulingprinzipien von T

Das Objektmodell von T hat wenig mit dem zu tun, was mit dem Begriff „objektorientierte Programmierung“ assoziiert ist[13]. Der eigentliche Nutzen des T Objektmodells liegt in der Festlegung einer einheitlichen Schnittstelle zur Programmierung mit Threads, mit denen Nebenläufigkeit innerhalb von klassischen Prozessen realisiert werden kann. Der Vorteil einer solchen „leichtgewichtigen“ Nebenläufigkeit im Vergleich zu der bei schwergewichtigen Prozessen üblichen ist die einfache Kommunikation der beteiligten Kontrollflüße untereinander, als auch der direkte Zugriff auf gemeinsame Daten aufgrund des gemeinsamen Adreßraumes z.B. mittels globaler Variablen. Dies bringt jedoch auch einige Risiken mit sich, die bei herkömmlicher, nichtkonkurrierender Programmierung höchstens im Zusammenhang mit Signalhandlern zutagetreten. Aus diesem Grunde ist es wichtig, sich über die Schedulingprinzipien im Klaren zu sein, da Threads im Gegensatz zu Prozessen nicht isoliert voneinander betrachet werden können.

T bietet zwei Formen der Nebenläufigkeit, das kooperative und das preemptive Multitasking. Beide Begriffe wurden schon in Abschnitt 2.1.2 eingeführt und werden hier daher nicht näher behandelt. Dementsprechend existieren zwei Schedulingklassen, denen Objekte angehören können, `T_COOPERATIVE` und `T_PREEMPTIVE`. Zum Zeitpunkt der Erzeugung wird ein Objekts einer dieser beiden Klassen zugeordnet. Es kann jedoch zu jedem Zeitpunkt durch den Aufruf der Funktion `T_set_schedclass` die Klasse wechseln.

[13] Durch geeignete Verbindung von T Objekten mit den Konstrukten einer objektorientierten Sprache wie C++ könnte man das T Modell auch entsprechend erweitern.

Schedulingklasse T_COOPERATIVE

Ein Thread dieser Klasse gibt seine Rechenzeit entweder freiwillig mittels `T_schedule`, durch den Aufruf einer threadblockierenden Funktion oder nach Abarbeitung des Standardhandlers ab. Ansonsten ist die Ununterbrechbarkeit seiner Operationen garantiert.

Schedulingklasse T_PREEMPTIVE

Für einen Thread der Klasse `T_PREEMPTIVE` gilt bzgl. der Rechenzeitabgabe zunächst das gleiche wie für einen Thread der Klasse `T_COOPEERATIVE`. Hinzu kommt, daß ihm die Rechenzeit spätestens nach Ablauf seines Zeitquantums entzogen wird, wenn sich noch ein anderer Thread im Zustand „Bereit" befindet. Aus diesem Grund, kann die Ununterbrechbarkeit einer Anweisungssequenz nicht garantiert werden. Wann eine Reaktivierung stattfindet ist nicht vorhersagbar, insbesondere bei Koexistenz mit Threads der Klasse `T_COOPERATIVE`.

Ist der aktive Thread aus der Klasse `T_COOPERATIVE`, müssen alle anderen Threads, einschließlich derer der Klasse `T_PREEMPTIVE`, warten, bis er seine Rechenzeit abgibt. Kommt ein Scheduling zustande und sind mehrere Threads rechenbereit, so entscheidet die Priorität darüber, welcher Thread als nächstes aktiviert wird. Zur Zeit existiert in T nur ein Prioritätsschema, das besagt, daß ein rechenbereiter Thread der Priorität x erst dann aktiviert wird, wenn es keinen rechenbereiten Thread der Priorität y mit $y > x$ gibt. Bei Threads gleicher Priorität wird nach dem FIFO Prinzip verfahren, d.h. der erste Thread, der mittels einer Nachricht aktiviert wurde, wird beim Schedulen auch als erstes ausgewählt. Dieses Prinzip ist unabhängig von der Schedulingklasse, der ein Thread angehört.

Wechseln der Schedulingklasse

Im Verlaufe der Abarbeitung des Standardhandlers eines Threads kann es sinnvoll sein, die Schedulingklasse zu wechseln, insbesondere kann ein Thread der Klasse `T_PREEMPTIVE` so einen kritischen Pfad sichern. Das Wechseln der Schedulingklasse geschieht mit der Funktion `T_set_schedclass`, mittels `T_get_schedclass` kann die aktuelle Schedulingklasse des aktiven Threads festgestellt werden.

3.5 Aspekte der Objekt- bzw. Threadterminierung

Nachdem nun die Erzeugung von Objekten und die Kommunikation untereinander in T vorgestellt wurden, soll in diesem Abschnitt ein wenig die Problematik der Terminierung von Objekten (Thread Cancellation, siehe auch [Sun Microsystems Inc., 1994]) angegangen werden.

Die mit T realisierbare Nebenläufigkeit erinnert zwar ein wenig an die (quasi-) parallele Verarbeitung schwergewichtiger Prozesse bei Betriebssystemen wie Unix, jedoch

gibt es schwerwiegende Unterschiede. Alle Systemressourcen, die ein schwergewichtiger Prozeß benötigt, werden vom Betriebssystem verwaltet und müssen angefordert werden. Terminiert ein schwergewichtiger Prozeß, ohne allokierte Ressourcen zurückzugeben, zieht dies andere Prozesse nicht in Mitleidenschaft, da dem Betriebssystem allokierte Ressourcen bekannt sind und freigegeben werden können. Ganz anders verhält es sich bei T Objekten. Jedes ist in der Lage, sich selbstständig Ressourcen zu nehmen, ohne sie bei einem „Managerobjekt" anfordern zu müssen. Erschöpft ein Objekt eine dem schwergewichtigem Prozeß zur Verfügung gestellte Ressource, so wirkt sich dies auf alle anderen Objekte aus. Dieses Problem könnte man zwar durch Entwicklung eines Objektmodells mit entsprechenden Privilegstufen beseitigen[14], doch liegt gerade in der Einsparung dieses Overheads ein großer Performanzvorteil von Threads gegenüber Prozessen begründet.

Ein weiterer Punkt, den man beim Programmieren mit T beachten muß, ist, daß Objekte untereinander kommunizieren und dabei ein Objekt ggf. auf Antwort eines anderen wartet. Terminiert dieses nun, kann das dazu führen, daß das wartende Objekt nicht mehr weiter ausgeführt wird und dabei seinseits unter Umständen andere Objekte blockiert.

Daher hat allein der Anwender dafür zu sorgen, daß alle Threads ordnungsgemäß beendet werden und dem Prozeß (und damit allen Threads) nach einer Weile nicht die Ressourcen ausgehen. Diese Forderung ist nicht so schwierig einzuhalten, da der Anwender ja in der Regel alle Threads und deren Verhalten kontrolliert[15].

In T wird zwischen zwei Formen der Terminierung unterschieden, der „ordentlichen" und der „erzwungenen" Terminierung. Bei ersterer handelt es sich um die Terminierung und Auflösung eines Threads nach der Abarbeitung seines Standardhandlers bei negativem Rückgabewert. Diese Form der Terminierung ist relativ unkritisch, da sie kontrolliert vom zu terminierenden Thread selbst initiiert wird. Jedoch ergibt sich hier bereits die Problematik, daß in der *IN*-Queue des Threads noch Aufträge sein könnten, auf deren Beantwortung andere warten. Sollten diese nicht mehr bearbeitet werden, so gibt das System sie frei, wobei ggf. auch die Reaktivierung der Auftraggeber vorgenommen wird. Nicht vermieden wird hierdurch jedoch die Möglichkeit, daß andere Threads noch eine Nachricht an das frisch terminierte Objekt schicken wollen. Das Versenden einer Nachricht an ein nicht existierendes Objekt kann im schlimmsten Fall zu einer Speicherverletzung mit resultierendem Abbruch des gesamten Prozesses führen, weswegen dieser Fehler mit Bedacht umgangen werden sollte. Nicht zu vergessen ist auch die Freigabe von Speicherblöcken, die der zu terminierende Thread im Laufe seiner Aktivität allokiert hat. Trotz allem ist die „ordentliche" Terminierung noch der sauberste Weg, einen Thread zu beenden.

In manchen Fällen kann es notwendig oder eleganter sein, die Terminierung eines Threads von außen zu erzwingen[16]. Dies wird durch die Funktion `T_process_cancel`

[14] Systemressourcen könnten dann von Systemobjekten verwaltet werden.

[15] Im Gegensatz zu Prozessen, die in der Regel vollkommen unabhängig voneinander agieren.

[16] Ein Beispiel hierfür wäre ein Thread, dessen Standardhandler in einer Endlosschleife läuft und lediglich Rescheduling zuläßt. Grundsätzlich kann man solche Endlosschleifen auch durch ständiges Senden einer Nachricht an sich selbst und vollständiger Abarbeitung des Standardthreadhandlers realisieren,

realisiert, wobei diese niemals auf den gerade aktiven Thread angewandt werden kann. `T_process_cancel` veranlaßt die Löschung des adressierten Objekts samt Bereinigung der jeweiligen Nachrichtenwarteschlangen wie oben beschrieben. Im Gegensatz zur „ordentlichen" Terminierung hat der zu terminierende Thread jedoch nicht mehr die Möglichkeit, Aufräumarbeiten zu erledigen. Da dies die Systemintegrität verletzen kann, ist ein Thread in der Lage, die Anwendung eines `T_process_cancel` auf sich zu beeinflussen, indem er mittels `T_set_cancelhandler` eine Funktion bestimmen kann, die bei erzwungener Terminierung aufgerufen wird. Es besteht sogar die Möglichkeit, mittels `T_set_cancelstate` und Parameter `T_CANCEL_DISABLED` die „erzwungene" Terminierung zu verzögern, und zwar bis zum nächsten Aufruf von `T_set_cancelstate` mit Argument `T_CANCEL_ENABLED`.

3.6 Überblick über weitere T Standarddienste

Alle bisher beschriebenen Funktionen sind dem Kern von T zuzurechenen. In diesem Kapitel sollen nun weitere Funktionen beschrieben werden, die den Umgang mit T komfortabler gestalten, ihrerseits jedoch die Kernfunktionen benutzen und bereits vollständig ohne weitere Eingriffe in die Systeminterna auskommen. Es sind Funktionen zur Ein-/Ausgabe, sowie ein Zeitdienst.

3.6.1 Ein-/Ausgabe

Eine wesentliche Motivation für die Entwicklung von T war die Problematik der Realisierung von konkurrierender Ein-/Ausgabe, die von vielen Servern erledigt werden muß. Eine klassische Programmiervariante hierzu ist die Verwendung des `select` Befehls, dem Mengen von Deskriptoren[17] übergeben werden, und der den aufrufenden Prozeß solange blockiert, bis einer dieser Deskriptoren „bereit" ist[18]. Hierduch kann eine konkurrierende Verarbeitung von mehreren Deskriptoren realisiert werden, ohne daß der Prozeß ständig alle Deskriptoren abfragen muß („pollen"), was in Multitasking Umgebungen immer sehr unelegant ist. Das erreichte „Scheduling" ist somit I/O getriggert. Der Nachteil bei der Verwendung von `select` ist, daß das Programmschema immer in einer Schleife resultiert, die den Kern des Programms ausmacht, und in die alle verarbeitenden Funktionen nach möglichst kurzer Zeit zurückkehren müssen, da der Server sonst nach außen nicht mehr konkurrierend erscheint.

Bei der Verwendung von T bietet sich nun die Zuordnung von Objekten (Threads) und Ein-/Ausgabekanälen an, wobei jedoch nicht die bekannten I/O-Funktionen angewandt werden können, da sie ggf. den ganzen Prozeß und nicht nur einzelne Threads blockieren. Glücklicherweise bieten nahezu alle UNIX Derivate auch nichtblockierende Varianten der I/O-Funktionen an, womit es einem Prozeß möglich ist, selbst zu ent-

doch kann dies zu unübersichtlichem, künstlichem Code führen.

[17] Deskriptoren sind die Abstraktion von Ein-/Ausgabekanälen und werden vom Betriebssystem vergeben.

[18] Bereitschaft heißt hier, daß auf den mit dem Deskriptor assoziierten Ein-/Ausgabekanal geschrieben bzw von ihm gelesen werden (je nach Art).

scheiden, wie weiter verfahren werden soll (z.B. bei nicht vorhandenen Daten an einer Pipe).

In einer sehr niedrigen Abstraktionebene angesiedelt ist das *IO*-Objekt. Dies ist ein mittels `T_new_sigprocess` installiertes Objekt, welches u.a auf den Empfang des Signals `SIGIO` reagiert. Dieses Signal kann von Ein-/Ausgabetreibern generiert werden. Ob und zu welchem Ereignis dies geschieht, ist zum einen abhängig vom angesprochenen Gerät, als auch vom konkreten UNIX Derivat[19]. Der Benutzer kann ebenfalls mit dem *IO*-Objekt in Verbindung treten, indem er ihm eine Nachricht schickt, bei der er als `id` einen beliebigen Wert angeben kann, der ihm ebenfalls als `id` zurückgeschickt wird. Mittels `value` wird spezifiziert, wieviele `SIGIO`-Signale seit Prozeßstart bis zur Antwort empfangen werden sollen. Das System führt intern einen Signalzähler mit, bei dessen Überschreitung des `value`-Wertes das *IO*-Objekt antwortet. Diese lautet konkret $(id, value)$ mit id = Echo des zuvor abgesetzten `id`-Wertes, $value$ = aktueller Stand des `SIGIO`-Signalzählers.

Die Semantik von `SIGIO`-Signalen ist systemabhängig, jedoch bauen die im folgenden vorgestellten Funktionen nicht auf dem `IO`-Objekt auf und stellen somit eine komfortablere und portablere Schnittstelle zur Verfügung.

Zum Öffnen und Schließen von Dateien existieren zunächst die Funktionen `T_open` und `T_close`. Beide entsprechen in ihrer Syntax und Semantik weitestgehend ihren äquivalenten Standard C Funktionen `open` und `close`. Das Öffnen einer Datei kann jedoch blockierend wirken, wenn sich dahinter z.B. ein Kommunikationskanal (Pipe) verbirgt, der zum Schreiben geöffnet werden soll. `T_open` wirkt in jedem Fall nur threadblockierend, wobei zusätzlich ein Timeout-Intervall angegeben werden kann, nach dessen Ablauf die Blockierung aufgehoben wird. Intern wird dieses Verhalten unter Zuhilfenahme des in Abschnitt 3.6.2 vorgestellten *Timer*-Objekts realisiert und sollte daher keine Kompatibilitätsprobleme unter den einzelnen UNIX Derivaten aufwerfen.

Nach dem Öffnen einer Datei stehen zum Lesen bzw. Schreiben die Funktionen `T_read` und `T_write` zur Verfügung. Auch sie blockieren ggf. nur den aufrufenden Thread und bieten die Möglichkeit der Angabe eines Timeout-Intervalls. Beiden Funktionen wird bei Aufruf die Adresse eines Speicherbereichs und dessen Länge übergeben, wobei nicht garantiert ist, daß alle Daten auf einmal geschrieben bzw. gelesen werden. Die Anzahl der tatsächlich geschriebenen bzw. gelesenen Bytes wird als Returnwert zurückgegeben und die Funktionen müssen unter Umständen mehrfach iteriert werden (siehe auch das Beispiel und die Systemdokumentation zu `write` und `read`).

Beispiel:

```
#define OPEN_TIMEOUT 5000 /* ms, Zeitspanne, die beim Oeffnen
                             gewartet wird */
#define WRITE_TIMEOUT 5000 /* ms, dito fuer's Schreiben */
```

[19]Soweit dem Autor bekannt, werden unter dem Betriebssystem Linux `SIGIO`-Signale lediglich für Socketdeskriptoren, sowie für Filedeskriptoren verbunden mit speziellen Geräten, geliefert, nicht jedoch für Aktivitäten an „echten" Filedeskriptoren. Hinzu kommt, daß z.B. Terminaltreiber standardmäßig `SIGIO`'s erst nach Einlesen eines Carriage Returns verschicken. Eine Änderung solcher Verhaltensweisen ist hochgradig systemabhängig.

```
int outthread_code (void *env, T_Object *object, long id, long value)
{
  int out_fd, sendcount, length;

  switch (value) {
  case GO:
    while ((out_fd = T_open("outpipe.tmp",
                            O_WRONLY,OPEN_TIMEOUT)) < 0) {
      if (errno == ENXIO)
        printf("outpipe.tmp, open timeout, retry.\n");
      else {
        printf("outpipe.tmp, unable to open.\n");
        T_send(Root,DUMMY,TERMINATE);
        return(-1);
      }
    }
    /* hier absichtlich kein Break */
  case WRITE: /* in 'id' steht die Adresse, des zu schreibenden
                 Strings */
    ....
    sendcount = 0; length = strlen((char *) id);
    while ((res = T_write(out_fd,(char *) (id + (long) sendcount),
                          (size_t) (length - sendcount),
                          WRITE_TIMEOUT)) < 0) {
      if (errno!=EAGAIN) {
        printf("outpipe.tmp, write error\n");
        T_send(Root,DUMMY,TERMINATE);
        return(-1);
      }
      else {
        sendcount+ = res;
        if (sendcount >= length)
          return(0);
      }
    }
    break;
  case TERMINATE: /* es duerften jetzt keine weiteren Nachrichten als
                     TERMINATE Sendungen in der
                     Inqueue stehen */
    return(-1);
    break;
  }
  return(0);
}
```

3.6.2 Das *Timer*-Objekt

Ein weiteres, standardmäßig vom System zur Verfügung gestelltes Objekt ist das *Timer*-Objekt. Hierbei handelt es sich, ähnlich dem *IO*-Objekt, um ein „Echo"-Objekt, welches eine gegebene Nachricht $(id, value)$ nach Ablauf von *value*-tausendstel Sekunden mit der Nachricht $(id_{ret}, value_{ret})$ beantwortet, für die gilt, $id_{ret} = id$ und $value_{ret} =$ abgelaufene Millisekunden seit Prozeßstart.

Das *Timer*-Objekt bedient sich dabei des Echtzeittimers, der vom Betriebssystem jedem Prozeß zur Verfügung gestellt wird. Dieser Timer wird auch vom Scheduler benötigt.

Auch wenn die zugrundegelegte Zeiteinheit des *Timer*-Objekts Millisekunden beträgt, ist es in der Regel nicht möglich, Zeitintervalle mit so genauer Auflösung zu spezifizieren. Das kleinste meßbare Zeitintervall hängt dabei von der Rechnerplattform und dem Betriebssystem ab. Der Wert ist in der vordefinierten Konstante `TIMER_QUANTUM` festgehalten und beträgt meistens 100 Millisekunden. Zeitspezifikationen sollten daher immer ein Vielfaches dieses Wertes sein, da ansonsten entsprechend aufgerundet wird.

> Beispiel: Ist `TIMER_QUANTUM` gleich 100, so bewirkt der Aufruf von `T_call(Timer,0,750)` ein Blockierung des Aufrufers für die Dauer von 8 Zehntelsekunden (!).

3.7 Ein Beispiel: Semaphoren unter T

In Abschnitt 2.3.3 wurden Semaphoren als ein praxistaugliches Konzept zur Synchronisation eingeführt. UNIX System V war eines der ersten Betriebssysteme, das Semaphoren zusammen mit „Shared Memory" und „Message Queues" (siehe Abschnitt 2.5.3) dem Programmierer zur Verfügung stellte. Diesen drei „Interprocess Facilities" widerfuhr auch eine Standardisierung im Rahmen der POSIX.4 Definitionen ([Gallmeister, 1995]), allerdings in leicht abgeänderter Form. Mittlerweile werden sie von vielen Betriebssystemen angeboten und dienen dort der Synchronisation und Kommunikation zwischen Prozessen. Unter T besteht kein Bedarf an Semaphoren, da Objekte die Ununterbrechbarkeit ihrer Aktionen durch den Wechsel in die Schedulingklasse `T_COOPERATIVE` sichern bzw. Erzeuger/Verbraucher Abhängigkeiten mit Hilfe geeigneten Nachrichtenaustauschs realisieren können. Zur Demonstration der Programmentwicklung mit Hilfe von T sollen im folgenden trotzdem die Funktionen, die unter System V bekannt wurden, simuliert werden. Dabei handelt es sich genauer um sehr einfache Varianten von `semget` und `semop`[20]. Sinnvollerweise ist alles, was mit Zugriffsverwaltung der Semaphoren zu tun hat, weggelassen, da es hier um Synchronisation von Threads innerhalb eines Prozesses geht.

```
/* sem.c */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

[20] `semctl` wird nicht berücksichtigt.

```
#include <TSystem.h>

#define SEMOP 0
#define ERROR 1
#define OK 2
#define RETRY 3

int sem_highest_index = -1;
T_Object *sem_monitor = (T_Object *) 0;

typedef struct Semstruct_ds Semstruct;
struct Semstruct_ds {
  int nsems;
  key_t key;
  int *semaphors; /* Array mit Integerwerten */
};
Semstruct **semarray;
/* beschreibt ein Array mit Semaphoren */

typedef struct Semopstruct_ds Semopstruct;
struct Semopstruct_ds {
  int semid;
  struct sembuf *sops;
  unsigned nsops;
};

int sem_monitor_code (void *env, T_Object *object, long id, long value)
{
  Semstruct **sem = (Semstruct **) env, semaphor;
  Semopstruct *s = (Semopstruct *) id;
  short sem_num, sem_op;
  int counter;
  long savedid, savedvalue;
  T_Object *savedobject;

  switch (value) {
  case SEMOP:
    semaphor = *semarray[s->semid];
    for (counter = 0; counter < s->nsops; counter++) {
      /* pruefen, ob alle Operationen ausfuehrbar sind */
      sem_num = s->sops[counter].sem_num;
      sem_op = s->sops[counter].sem_op;
      if ((sem_num < 0) || (sem_num >= semaphor.nsems)) {
        T_reply(id,ERROR);
```

```
        return(0);
      }
      if (semaphor.semaphors[sem_num] + sem_op <= 0) { /* blockieren */
        T_save(id,value); /* Aufrufer blockieren */
        return(0);
      }
    } /* alle Operationen lassen sich blockierungsfrei ausfuehren */
    for (counter = 0; counter < s->nsops; counter++) {
      /* pruefen, ob alle Operationen ausfuehrbar sind */
      sem_num = s->sops[counter].sem_num;
      sem_op = s->sops[counter].sem_op;
      if ((sem_num < 0) || (sem_num >= semaphor.nsems)) {
        T_reply(id,ERROR);
        return(0);
      }
      semaphor.semaphors[sem_num]+= sem_op;
    }
    T_reply(id,OK);
    /* jetzt werden alle blockierten Objekte zum Retry aufgefordert */
    while (T_read_saved(&savedobject,&savedid,&savedvalue,0)) {
      T_reply_saved(savedid,RETRY);
    }
    break;
    /* hier koennten noch weiter Faelle, z.B. zur Terminierung
       behandelt werden */
  }
  return(0); /* und erhalten bleiben */
}

void seminit ()
{
  if (!(*semarray = (Semstruct *) calloc(1,sizeof(Semstruct *))))
    T_error("seminit","Unable to allocate semaphor array",errno);
  semarray[0] = (Semstruct *) 0;
  sem_highest_index = 0;
  if (!(sem_monitor = T_new_process(sem_monitor_code,(char*) 0,
                                    T_STACKSIZE,Memory,
                                    T_COOPERATIVE,
                                    T_SYSTEM_PRIORITY,
                                    (void *) semarray,
                                    "SEM_MONITOR")))
    /* Semaphorenarray wird als Environment uebergeben */
    T_error("seminit","Unable to install semaphor monitor.",1);
}
```

```
int semnew (key_t key, int nsems)
{
  Semstruct *new;

  if (!(new = (Semstruct *) malloc(sizeof(Semstruct))))
    return(-1);
  if (!(new->semaphors = (int *) calloc(nsems,sizeof(int)))) {
    free(new);
    return(-1);
  }
  if (!(*semarray = (Semstruct *) realloc(semarray,
                                          (sem_highest_index + 2) *
                                          sizeof(Semstruct *)))) {
    free(new->semaphors); free(new);
    return(-1);
  }
  semarray[sem_highest_index] = new;
  sem_highest_index++;
  semarray[sem_highest_index] = (Semstruct *) 0;
  new->nsems = nsems;
  new->key = key;
  return(sem_highest_index-1);
}

int semget (key_t key, int nsems, int semflg)
{
  int index, res = -1, schedclass = T_get_schedclass();

  T_set_schedclass(T_COOPERATIVE);
  if (sem_highest_index < 0)
    seminit();
  if (key == IPC_PRIVATE) {
    res = semnew(key,nsems);
    T_set_schedclass(schedclass);
    return(res);
  }
  /* jetzt Semaphorenstruktur suchen */
  for (index = 0; index < sem_highest_index; index++) {
    if (semarray[index]->key == key) {
      T_set_schedclass(schedclass);
      if (semflg & IPC_EXCL) /* leider gefunden */
        return(-1);
      else
```

```
        return(index);
      }
    }
    /* Semaphore mit 'key'  wurde nicht gefunden */
    if (semflg & IPC_CREAT)
      res = semnew(key,nsems);
    T_set_schedclass(schedclass);
    return(res);
}

int semop (int semid, struct sembuf *sops, unsigned nsops)
{
    Semopstruct *s;
    long value, id;

    if ((semid < 0) || (semid >= sem_highest_index)) /* gibt's nicht */
      return(-1);
    if (!(s = (Semopstruct *) malloc(sizeof(Semopstruct))))
      return(-1);
    s->semid = semid; s->sops = sops; s->nsops = nsops;
    id = (long) s;
    /* jetzt die Semaphore holen */
    do {
      value = SEMOP;
      T_call(sem_monitor,&id,&value);
      /* fuehrt die Operationen aus und blockiert ggf. dabei */
      if (value == ERROR)
        return(-1);
    } while (value != OK);
}
```

3.8 T Befehlsreferenz

In diesem Abschnitt folgt nun die Beschreibung der Funktionen der Threadlibrary **T**.

Objekte sind vom Typ `T_Object`, Standardhandler vom Typ `T_Code`, der definiert ist mittels:

```
typedef int(*T_Code)(void *environment,T_Object *object,
                     long id, long value);
```

Dabei ist `environment` die bei der Erzeugung des Threads spezifizierte Umgebung, `object` der Absender der aktuellen *IN*-Nachricht, die aus `id` und `value` besteht.

3.8.1 Funktionen zur Objekterzeugung

```
T_Object *T_new_process (T_Code code, char *stack, int stack_size,
                         T_Memory *memory, int schedclass,
                         int priority, void *environment,
                         char *threadname);
```

T_new_process erzeugt ein neues Objekt und liefert im Erfolgsfall einen Verweis auf dessen Verwaltungsstruktur zurück, ansonsten den Wert (T_Object *) 0. Mit code wird der entsprechende Standardhandler übergeben, der bei der Aktivierung aufgerufen wird. stack, stack_size und memory dienen zum Aufsetzen des Stacks, auf dem der Thread arbeiten soll. Ist stack ungleich dem Nullzeiger, so wird dies als Speicherbereich der Größe stack_size interpretiert und als Stack verwandt. In diesem Fall ist der Parameter memory bedeutungslos. Ist stack jedoch gleich dem Nullzeiger, so muß memory auf einen mittels T_memory_init erzeugten Speicherbereich zeigen, auf dem versucht wird, ein Stacksegment der Größe stack_size zu allokieren. Ein solcher Speicherbereich ist vom System bereits eingerichtet und kann unter der Variable Memory angeprochen werden. Ebenso ist es nicht unbedingt nötig, den zu erwartenden Stackverbrauch im Vorfeld abzuschätzen. Eine sinnvolle Größe kann durch die Konstante T_STACKSIZE eingestellt werden. Die Parameter schedclass und priority dienen der Angabe der Scheduling- und der Prioritätsklasse (siehe 3.4), mit der ein Thread arbeiten soll. T kennt nur eine „Scheduling Policy", die besagt, daß ein rechenbereiter Thread der Priorität x erst dann aktiviert wird, wenn es keinen rechenbereiten Thread der Priorität y mit $y > x$ gibt. Bei Threads gleicher Priorität wird nach dem FIFO Prinzip verfahren, d.h. der erste Thread, der mittels einer Nachricht aktiviert wurde, wird beim Schedulen auch als erstes ausgewählt. Die Werte von priority sollten sich zwischen den vordefinierten Konstanten T_USER_PRIORITY (niedrig) und T_SYSTEM_PRIORITY (hoch) bewegen. Der Parameter environment kann dazu benutzt werden, einen threadspezifischen Speicherbereich anzugeben, der dem Thread bei Aufruf des Standardhandlers zur Verfügung gestellt wird. Der letzte Parameter threadname ist nicht besonders wichtig, er dient lediglich der Benennung des Objekts mit einer maximal 80 Zeichen langen Zeichenkette zu benennen. Eine solche Information kann für Debuggingzwecke nützlich sein, denn vom System werden bei desen Terminierung die zu diesem Zeitpunkt noch exisierenden Objekte unter diesem Namen im Klartext aufgelistet. Aus Gründen der Wirtschaftlichkeit mit Speicher kann dieses Feature auch herauskompiliert werden.

Beispiel:

```
T_Object *outthread_monitor;

typedef struct UP_ds {
  pid_t pid;
  ...
```

```
} UP;

UP Initial = {getpid(),....};

int outthread_monitor_code (void *env, T_Object *object,
                            long id, long value)
{
  UP *initial = (UP *) env;

  printf("Ich bin der Stream Monitor\n");
  return(0);
}

int main ()
{
  if ((outthread_monitor = T_new_process(outthread_monitor_code,
                                         (char*) 0,T_STACKSIZE,
                                         Memory, T_USER_PRIORITY,
                                         (void *) &Initial,
                                         "Stream Monitor"))
      == (T_Object *) 0) {
    printf("Unable to create stream monitor.\n");
    T_exit(errno);
  }
  ...
}
```

Hier wird ein Thread erzeugt, dessen Standardhandler durch die Funktion `outthread_monitor_code` definiert ist. Der Stack wird auf dem Systemspeicher `Memory` allokiert und hat die Größe `T_STACKSIZE` (der Nullzeiger an der Position von `stack` ist wichtig). Die Umgebung des Threads ist die Struktur `Initial`, die auch als globale Variable zugreifbar ist.

```
T_Object *T_new_sigprocess (int signr, T_Code code, char *stack,
                            int stack_size, T_Memory *memory,
                            int schedclass, int priority,
                            void *environment, char *threadname);
```

`T_new_sigprocess` erzeugt ebenfalls ein neues Objekt, und die Parameter haben die gleiche Bedeutung wie beim Kommando zuvor. Hinzugekommen ist der Parameter `signr`, mit dem ein Signal spezifiziert werden kann, welches das Objekt aktiviert. Aus diesem Grund heißen so erzeugte Objekte auch Signalobjekte. Sie sind normale Objekten, die zusätzlich in der Lage sind, Nachrichten vom System zu erhalten (und damit

aktiviert zu werden), was genau dann geschieht, wenn das bei der Erzeugung spezifizierte Signal seit der letzten Threadaktivierung mindestens einmal an den Prozeß verschickt wurde. Daß eine Nachricht nicht von einem anderen Objekt, sondern vom System, kommt, kann der Standardhandler eines Signalobjekts daran erkennen, daß sein Parameter `object` gleich `(T_Object *) 0` ist. In diesem Fall wird in `id` der aktuelle Wert des Signalzählers und in `value` die Anzahl der empfangenen Signale seit der letzten Aktivierung übergeben. Ein Signalzähler ist ein Modulozähler, der bei Überlauf wieder bei 0 beginnt.

Anmerkung:
Unter Unix sind Signale asynchrone Ereignisse, die einen Prozeß sofort unterbrechen und (bei Vorhandensein) den entsprechenden Signalhandler aktivieren lassen. Die damit einhergehende Nebenläufigkeit zwischen eigentlichem Prozeß und Signalhandler erfordert, daß sich beide geeignet synchronisieren müssen, wenn sie auf gemeinsame Daten zugreifen. T Signalobjekte ordnen sich jedoch dem allgemeinen Schedulingprinzip unter, werden also nicht sofort bei Signalempfang aktiviert, sondern lediglich als „rechenbereit" markiert. Auf diese Weise entfällt ggf. jegliches Sperren kritischer Pfade etc., was eine der häufigsten Fehlerquellen beim Umgang mit Signalen darstellt.

In manchen Fällen ist jedoch eine unmittelbare, asynchrone Verarbeitung von Signalen notwendig. In diesem Fall muß ein Signalhandler vom Benutzer selbst gesetzt werden[21]. Wurde für ein Signal ein Handler gesetzt, so wird dieser deaktiviert, wenn später für dasselbe Signal ein Signalobjekt installiert wird. Nach Terminierung dieses Objekts, wird jedoch der alte Signalhandler wieder reaktiviert.

3.8.2 Funktionen zur Objektkommunikation

```
int T_send (T_Object *object, long id, long value)
```

Die Funktion `T_send` dient der nachrichtenorientierten Kommunikation. Es wird an `object` eine Nachricht mit den Werten `id` und `value` versandt. Der adressierte Thread muß existieren, darf also zum Zeitpunkt des Aufrufs dieser Funktion nicht schon terminiert sein. Bei erfolgreicher Zustellung wird eine 0 (!), ansonsten eine -1 zurückgeliefert und das empfangende Objekt ggf. als „rechenbereit" markiert. Der Aufruf kann scheitern, wenn nicht mehr genügend Speicherblöcke zur Allokation der Nachricht zur Verfügung stehen.

```
int T_call (T_Object *object, long *id, long *value)
```

`T_call` dient der auftragsorientierten Kommunikation. Analog zu `T_send` wird eine Nachricht mit den Werten `id` und `value` an `object` versandt, jedoch bleibt der Aufrufer blockiert, bis der Auftrag den adressierten Thread „passiert" hat. „Passieren" heißt

[21] Z.B. mit der Funktion `sigaction`.

hierbei, daß der Auftrag entweder via T_reply beantwortet, oder aber zumindest empfangen und nicht mittels T_save oder T_forward weiterverarbeitet wurde (siehe auch T_receive). Im ersten Fall wird eine 1 zurückgegeben, und die Antwortwerte stehen in id und value. Im zweiten Fall wird eine 0 zurückgegeben. Ist der Aufruf von T_call wegen Speichermangel gescheitert, wird analog zu T_send eine -1 zurückgeliefert.

```
int T_receive (T_Object **object, long *id, long *value, int mode)
```

Mittels T_receive kann, sofern vorhanden, eine Nachricht aus der IN-Queue ausgekettet und zur aktuellen Nachricht gemacht werden. Der Parameter mode, der die Werte T_BLOCK und T_NONBLOCK annehmen kann, bestimmt dabei, ob bei nicht vorhandener Nachricht gewartet oder fortgefahren werden soll. War die bisherige aktuelle Nachricht ein Auftrag, so wird vor deren Überschreiben durch T_receive der Auftraggeber wieder in den Zustand „Bereit" versetzt. Die Rückgabewerte sind 1 bei vorhandener Nachricht (die sich dann in object, id und value befindet) und 0 sonst, was jedoch nur bei mode = T_NONBLOCK vorkommen kann.
Anm.:
Die Funktion T_receive ist eigentlich nicht notwendig, da die IN-Queue automatisch zugestellt wird. Sehr praktisch ist sie jedoch bei der Threadcancellation, wenn sichergestellt werden muß, daß alle noch in der IN-Queue befindlichen Nachrichten vor der Terminierung bearbeitet werden.

```
int T_reply (long id, long value)
```

T_reply sendet ganz allgemein eine Nachricht unter Verwendung der Struktur der aktuellen *IN*-Nachricht an den Absender zurück, d.h. es geht mit dem Versand keine Neuallokation einher. Die Werte können über id und value neu gesetzt werden. War die aktuelle Nachricht ein Auftrag, so wird zusätzlich noch der Auftraggeber wieder aktiviert, der sich bis dahin im „blockiert"-Zustand befand. In jedem Fall ist nach dem Aufruf von T_reply die aktuelle Nachricht undefiniert. Bei erfolgreichem Aufruf wird 1 zurückgeliefert, ansonsten (bei undefinierter aktueller Nachricht) 0.

```
int T_forward (T_Object *object, long id, long value)
```

Mit Hilfe der Funktion T_forward kann die aktuelle Nachricht mit den neuen Werten id und value an object weitergeleitet werden. Auch hierbei wird keine neue Nachrichtenstruktur allokiert, und die bisherige aktuelle *IN*-Nachricht ist anschließend nicht mehr definiert. Die Weiterleitung der Nachricht ist sowohl für den Absender der aktuellen *IN*-Nachricht, als auch für den Empfänger, an den weitergeleitet wird, transparent. Bei erfolgreichem Aufruf wird eine 1 zurückgeliefert, ansonsten (bei undefinierter aktueller Nachricht) eine 0.

3.8.3 Funktionen zur Manipulation der SAVE-Queue

`int T_save (long id, long value)`

Mittels `T_save` wird die aktuelle *IN*-Nachricht mit den ggf. neuen Werten `id` und `value` in die *SAVE*-Queue eingekettet. Diese ist nach `value` aufsteigend sortiert, wobei im Falle der Gleichheit zweier Einträge die neue Nachricht vor der alten eingetragen wird. `T_save` wird immer dann benötigt, wenn die Beantwortung eines Auftrags verzögert werden muß. Nach Aufruf von `T_save` ist die aktuelle Nachricht undefiniert. Die Rückgabewerte sind im Erfolgsfall 1, ansonsten 0, wenn die aktuelle *IN*-Nachricht undefiniert ist.

`int T_resave (long id, long value)`

Mit Hilfe von `T_resave` kann die aktuelle *SAVE*-Nachricht wieder zurück in die *SAVE*-Queue geschrieben werden, was manchmal nötig ist, wenn man mittels `T_read_saved` eine Nachricht zuviel ausgelesen hat. Die Rückgabewerte sind 1 bei Erfolg und 0, falls die akuelle *SAVE*-Nachricht nicht definiert ist. Es gilt, daß nach Aufruf die aktuelle *SAVE*-Nachricht undefiniert ist.

`int T_read_saved (T_Object **object, long *id, long *value, int disc)`

`T_read_saved` kettet eine Nachricht aus der *SAVE*-Queue aus und macht sie zur aktuellen *SAVE*-Nachricht (die alte geht dabei verloren). Die Werte befinden sich anschließend in `object`, `id` und `value`. Welche Nachricht ausgekettet wird, hängt dabei vom Parameter `disc` ab, der mit den Werten `0`, `T_OBJECT`, `T_ID` oder `T_VALUE` geladen werden kann. Im Falle von `0` wird die erste Nachricht der *SAVE*-Queue herangezogen (mit dem kleinsten `value` Wert). Die anderen drei Werte können mittels | (OR) kombiniert werden und bewirken, daß die erste Nachricht gesucht wird, die mit den Werten von `object`, `id` oder `value` übereinstimmt. Im Erfolgsfall wird eine 1 zurückgegeben, konnte keine den Suchspezifikationen entsprechende Nachricht gefunden werden, eine 0.

`int T_reply_saved (long id, long value)`

Mittels `T_reply_saved` wird die aktuelle *SAVE*-Nachricht an den Absender zurückgeschickt, wobei die Werte über `id` und `value` neu gesetzt werden können. War diese Nachricht ein Auftrag, so wird zusätzlich noch der Auftraggeber wieder in den Zustand „Bereit“ versetzt. Nach Aufruf von `T_reply_saved` ist die aktulle *SAVE*-Nachricht undefiniert. War sie das schon vor dem Aufruf von `T_reply_saved`, so scheitert dieser und es wird 0 zurückgeliefert, ansonsten (im Erfolgsfall) 1.

```
int T_forward_saved (T_Object *object, long id, long value)
```

T_forward_save dient analogen Zwecken wie T_forward, weitergeleitet wird jedoch die aktuelle *SAVE*-Nachricht. Bei erfolgreichem Aufruf wird eine 1 zurückgeliefert, ansonsten (bei undefinierter aktueller *SAVE*-Nachricht) eine 0.

3.8.4 Weitere Funktionen

```
void T_schedule ()
```

Normalerweise wird ein Rescheduling nach der Abarbeitung des Standardhandlers, nach Aufruf eines threadblockierenden Befehls oder nach Ablauf eines Zeitquantums, wenn das aktive Objekt der Schedulingklasse T_PREEMPTIVE angehört, durchgeführt. Mit Hilfe von T_schedule ist es möglich, zu jedem Zeitpunkt die Rechenzeit freiwillig abzugeben und, falls möglich, einen anderen Thread zu aktivieren. Erhält der aufrufende Thread die Kontrolle zurück, so wird unmittelbar mit der auf T_schedule folgenden Anweisung fortgefahren.

```
T_Object *T_self ()
```

Mittels T_self kann ein Thread den Verweis auf die eigene Verwaltungsstruktur ermitteln. Dies ist in vielen Fällen sehr hilfreich, z.B. um Nachrichten an sich selbst zu schicken.

```
void T_bind (T_Code code, void *environment)
```

T_bind dient dazu, den Standardhandler und das Environment des aufrufenden Objekts nach dessen Erzeugung zu ändern. Bei „herkömmlichen" Objekten wird die Änderung erst nach der vollständigen Abarbeitung des bisherigen Handlers bei der nächsten Aktivierung wirksam. Anders beim immer existierenden *Root*-Objekt, das vom System generiert wurde. Hierbei wird sofort ein Rescheduling durchgeführt und bei der nächsten Aktivierung der mittels T_bind spezifizierte Handler aufgerufen. In vielen Programmen ist diese Art der Verwendung von T_bind auch die einzige, denn es gilt, daß der gesamte Prozeß terminiert, wenn die Funktion main einmal abgearbeitet wurde. Die folgende Anweisungssequenz trifft man daher häufig am Ende von main an:

```
int root_code (void *env, T_Object *object, long id, long value)
{
  ...
  if (value == WAIT)
    return(0); /* Neuer Root-Handler abgearbeitet, aber Root-Objekt
                  bleibt bestehen */
}
```

```
int main (int argc, char *argv[], char *envp[])
{
  ...
  ... /* beliebige Initialisierungen */
  T_send(T_self(),DUMMY,WAIT);
  T_bind(root_code,(void *) 0);
  return(0); /* Defaulthandler ist abgearbeitet, Prozess terminiert */
}
```

```
void *T_environment ()
```

T_environment liefert einen Verweis auf die aktuelle Umgebung. Dieser Verweis, der auch im Parameter env eines Standardhandlers übergeben wird, braucht daher nicht an Funktionen übergeben werden, die von ihm aufgerufen werden.

```
int T_processes_avail ()
```

Mit Hilfe von T_processes_avail ist es möglich, die noch verbleibende Anzahl von erzeugbaren Objekten zu ermitteln, was der Rückgabewert dieser Funktion ist. Will man mehrere Objekte unmittelbar hintereinander erzeugen und initialisieren, so ist es häufig praktischer, bereits zu Beginn abzufragen, ob noch genügend Ressourcen für die Objekte vorhanden sind. Wie man das System für eine bestimmte Anzahl von Threads konfigurieren kann, steht in Abschnitt 3.8.10.

```
void T_set_priority (int prio);
int T_get_priority ();
```

T_get_priority ermittelt die Prioritätsklasse des aktiven (also des aufrufenden) Objekts. Diese wird normalerweise bei der Erzeugung festgelegt und kann via T_set_priority auch zur Laufzeit geändert werden. Erlaubte Argumentwerte liegen dabei zwischen T_USER_PRIORITY und T_SYSTEM_PRIORITY (siehe auch 3.8.10).

```
void T_set_schedclass (int schedclass);
int T_get_schedclass ();
```

Die Schedulingklasse eines Objekts wird bei der Erzeugung initial angegeben und kann zur Laufzeit via T_get_schedclass ermittelt und mittels T_set_schedclass gesetzt werden. Erlaubte Werte sind T_COOPERATIVE und T_PREEMPTIVE. Gehört ein Objekt der Schedulingklasse T_COOPERATIVE an, so sind all seine Berechnungen ununterbrechbar,

und ein Scheduling erfolgt erst bei freiwilliger Abgabe der Rechenzeit (via T_schedule, dem Aufruf einer threadblockierenden Funktion oder nach Abarbeitung des Standardhandlers).

```
void T_process_cancel (T_Object *object);
```

T_process_cancel erzwingt die Terminierung von object. Das aufrufende Objekt kann sich auf diese Weise allerdings selbst terminieren. Ein solcher Versuch wird vom System ignoriert.
Hat der Empfänger die Terminierung via T_set_cancelstate(T_CANCEL_DISABLED) abgeschaltet, so verbleibt die Anforderung in einer Wartestellung, bis er die Terminierung wieder zuläßt. Wird das betreffende Objekt aufgelöst, so werden zunächst die dem System bekannten Ressourcen (*IN*-Queue, *SAVE*-Queue etc.) freigegeben und anschließend ggf. noch ein „Cancelhandler" aufgerufen.

```
void T_set_cancelstate (int state);
```

Gerät ein Prozeß in eine Berechnungsphase, in der er nicht terminiert werden darf („erzwungene" Terminierung), so kann er dies mittels T_set_cancelstate und dem Parameter T_CANCEL_DISABLED unterbinden. Sollte ein anderes Objekt in diesem Zustand die Terminierung anfordern, so verbleibt die Anforderung in Wartestellung und wird unmittelbar nach dem Wechsel in den Zustand T_CANCEL_ENABLED (ebenfalls via T_set_cancelstate) ausgeführt. Initial ist ein Objekt immer im Zustand T_CANCEL_ENABLED.

```
void T_set_cancelhandler (T_Cancelcode new, T_Cancelcode *old);
```

Mittels T_set_cancelhandler kann der Aufrufer eine Funktion spezifizieren, die unmittelbar vor seiner „erzwungenen" Terminierung (und vor der Freigabe der Nachrichtenwarteschlangen) aufgerufen wird.
Diese Funktion ist vom Type void(*T_Cancelcode)(void *environment, T_Object *object) und bekommt bei Aktivierung das Environment des zugehörigen Objekts und den Verweis auf das letzte Objekt, welches eine Terminierung initiiert hat. Der „Cancelhandler" hat weiterhin Zugriff auf die Nachrichtenwarteschlangen (mittels T_receive und T_read_saved) und ist daher in der Lage, hiermit assoziierte Ressourcen freizugeben. Seine Operationen führt er im Zustand T_CANCEL_DISABLED aus, weswegen die komplette Abarbeitung garantiert ist.

```
void T_exit (int status)
```

`T_exit` erlaubt das kontrollierte Beenden des (schwergewichtigen) Prozesses von jeder Stelle im Programm aus und ist als Alternative zur Standard C–Anweisung `exit` zu verstehen. Bei der Verwendung von `T_exit` hat das System noch die Chance, alle Ressourcen freizugeben, die bei der Initialisierung belegt wurden, und noch entsprechende Statusmeldungen nach `STDOUT` zu schreiben. Der Parameter `status` erlaubt dabei, analog zu `exit`, einen Returnwert anzugeben, der dem Vaterprozeß vom Betriebssystem übergeben wird. Vereinbarungsgemäß sollte dieser Wert bei erfolgreicher Abarbeitung des Programms 0 sein.

3.8.5 T Speicherverwaltung

Bei der Entwicklung von T ergab sich die Notwendigkeit einer eigenen, einfachen Speicherverwaltung, da beim internen Scheduling manchmal Speicherblöcke auch noch kurz nach ihrer Freigabe benötigt werden. Desweiteren sind die meisten internen Datenstrukturen von konstanter Größe, was die Verwaltung effizienter und einfacher macht. Mit den folgenden Funktionen wird die Speicherverwaltung auch dem Benutzer zugänglich gemacht, es sei jedoch darauf hingewiesen, daß die Verwendung von Standard C `malloc` häufig adäquater ist, da T nur das Modell *Speicherpool* anbietet. Speicherpools sind Bereiche, die bereits bei der Initialisierung in feste Speicherblöcke aufgeteilt werden. Spätere Allokationen und Freigaben der Blöcke können dann sehr viel effizienter ausgeführt werden als im „herkömmlichen“ Modell der *Speichersegmente*.

```
void T_pool_init (T_Pool *pool, void *addr, int size,
                  int item_size)
```

`T_pool_init` richtet einen Speicherpool mit Hilfe der mittels `pool` übergebenen Speicherverwaltungsstruktur ein. Der Speicher selbst muß vorher statisch oder dynamisch vorallokiert worden sein. Er wird in `addr`, seine Größe in `size` übergeben. Die Größe der Speicherblöcke, die später via `T_pool_alloc` angefordert werden können, muß bei Speicherpools bereits beim Anlegen spezifiziert werden, was durch den Parameter `item_size` geschieht. Die Anzahl der dann verfügbaren Speicherblöcke $N_{size,item\_size}$ läßt sich leicht mittels $N_{size,item\_size} = size \quad div \quad item\_size$ bestimmen. Die Funktion `T_pool_init` gibt keine Werte zurück, sondern füllt lediglich die Verwaltungsstruktur `pool`. Es liegt in der Verantwortung des Aufrufers, sicherzustellen, daß `addr` auf einen gültigen, ausreichend großen Speicherbereich zeigt.

```
void *T_pool_alloc (T_Pool *pool)
```

`T_pool_alloc` allokiert ein Speichersegment auf durch `T_pool_init` angelegten Pool `pool`. Kann die Anforderung wegen Speichermangels nicht erfüllt werden, wird `(void *) 0`, ansonsten der Zeiger auf den allokierten Block zurückgegeben.

```
void T_pool_free (T_Pool *pool, void *addr)
```

Mittels `T_pool_free` kann ein Speicherblock wieder freigegeben werden. Dabei muß der Pool, an den der Block `addr` zurückgegeben wird, in `pool` angegeben werden. Da keinerlei Überprüfungen stattfinden, muß der Benutzer selbst dafür Sorge tragen, daß diese beiden Parameter konsistent sind. `T_pool_free` gibt nichts zurück, da bei korrekter Benutzung keine Fehler passieren können.

3.8.6 T Systemobjekte

Standardmäßig werden vom System nach dem Hochfahren bereits einige Objekte erzeugt, die dem Benutzer den Umgang erleichtern sollen bzw. den initalen Startpunkt definieren. Im folgenden werden sie kurz charakterisiert:

Root-Objekt

Dieses Objekt stellt den initialen Startpunkt eines jeden mit T gelinkten Programmes dar. Es nimmt insofern eine Sonderstellung ein, als es das einzige Objekt ist, welches auch ohne empfangene Nachricht bereits aktiv ist und initial keinen Standardhandler hat. Stattdessen wird im Kontext des *Root*-Objekts die Funktion `main` abgearbeitet, nach deren Beendigung der gesamte schwergewichtige Prozeß terminiert, unabhängig davon, ob noch weitere Objekte rechenbereit sind. Um mit dem *Root*-Objekt ein „normales" Verhalten zu erreichen, kann man mittels `T_bind` einen Standardhandler installieren. Im Gegensatz zur herkömmlichen Funktionsweise von `T_bind` bewirkt dies ein sofortiges Rescheduling, wobei das *Root*-Objekt ggf. in den Zustand „Blockiert" wechselt. Von anderen Objekten aus ist das *Root*-Objekt unter dem Verweis `Root` ansprechbar.

IO-Objekt

Das *IO*-Objekt dient der Signalisierung von Zustandsänderungen bei I/O-Deskriptoren, die mit entsprechenden Kanälen verbunden sind. Da die Erzeugung solcher Signale in starkem Maße von I/O-Treibern und Betriebssystem abhängt, wird die direkte Verwendung des *IO*-Objekts nicht empfohlen, da sie Portierungsaufwand für die entwickelte Applikation mit sich bringen kann. Es kann vom Benutzer unter dem Verweis `Io` angesprochen werden, wobei für Nachrichten $(id, value)$ gilt: *id* ist ein Wert, den das *IO*-Objekt zurückschicken soll, *value* beschreibt den Signalzähler für das Signal `SIGIO`, der zuletzt vom Aufrufer gesehen wurde. Das Überschreiten dieses Zählers stellt die Triggerbedingung für die Antwort des *IO*-Objekts dar, wobei *value* entsprechend modifiziert zurückgeschickt wird. Ob ein Signal tatsächlich zum angesprochenen I/O-Kanal „paßt", ist nicht garantiert. Dies muß ggf. noch auf nichtblockierende Art und Weise überprüft werden.

Timer-Objekt

Wesentlich portabler als die Verwendung des *IO*-Objekts ist die es *Timer*-Objekts, welches unter dem Verweis `Timer` angesprochen werden kann. Hierbei handelt es sich um eine Schnittstelle zum Intervalltimer des Betriebssystems, welcher vom Berkeley UNIX eingeführt und nunmehr auch vom System V UNIX übernommen wurde. Da er auch intern vom Scheduler benutzt wird, ist die Funktionsweise des *Timer*-Objekts auf allen Plattformen garantiert, auf die T portiert wurde (siehe Abschnitt 5). Um mit ihm in Verbindung zu treten, kann man eine Nachricht (*id*, *value*) schicken, für die gilt: *id* kann ein beliebiger Wert sein und wird vom *Timer*-Objekt zurückgeschickt, *value* beschreibt die Millisekunden, die bis zur Antwort vergehen sollen und beinhaltet nach Antwort die Anzahl der vergangenen Millisekunden seit Programmstart. Auch wenn die Nachrichtenparameter eine Zeitgranularität in der Größenordnung von Millisekunden zulassen, hat das *Timer*-Objekt nur eine Auflösung 10000/`TIMER_QUANTUM` Hz. Sinnvolle Parameter sind daher Vielfache dieser global definierten Konstante, die meistens 100 beträgt. Diese Restriktion entstammt der beschränkten Auflösung der Intervalltimer von Betriebssystemen, die in der Regel bei ca. 100 Hz liegt.

3.8.7 T Standarddienste

Unter Verwendung der oben angesprochenen Systemobjekte und des T Kernels existieren bereits einige Dienste, die dem Benutzer den Umgang mit T erleichtern sollen. Insbesondere in Verbindung mit I/O lassen sich viele Standard C Funktionen nicht mehr benutzen, da sie den ganzen, schwergewichtigen Prozeß blockieren würden. Alternativ dazu sollten die im folgenden vorgestellten Funktionen benutzt werden.

3.8.8 Ein-/Ausgabedienste

Die Standard C Bibliotheken bieten eine Vielfalt von I/O Funktionen an. Darunter fallen Funktionen zum Öffnen, Schreiben, Lesen und Schließen von Dateien, zum Setzen des Dateizeiger (sofern das angesprochene Gerät *seek*–fähig ist), etc. Zusätzlich stehen Befehle zum Schreiben oder Lesen zumeist noch in gepufferter und ungepufferter Variante zur Verfügung[22]. Allen Funktionen gemeinsam ist, daß sie im Endeffekt auf die Systembefehle `open`, `read`, `write`, `lseek` und `close` zugreifen, um letztendlich die gewünschte Operation auszuführen. Dazwischen liegen mehr oder weniger komplexe Pufferungsmechanismen. Ohne weitere Maßnahmen sind die Standard C Funktionen für einen „threaded" Prozeß nicht zu gebrauchen, da sowohl `read` als auch `write` und `open` einen Prozeß blockieren können, obwohl eigentlich nur der aufrufende Thread blockiert werden sollte. Die Lösung liegt in der alternativen Verwendung der unten angegebenen Funktionen, die als Ersatz der Systemfunktionen `open`, `close`, `read` und `write` anzusehen sind.

[22]Das Betriebssystem selbst puffert Schreib/Lese-Zugriffe in aller Regel auch, was sich erst durch bestimmte Einstellungen mittels `fcntl` oder `ioctl` abstellen läßt.

```
int T_open (const char *pathname, int flags, long timeout)
```

T_open öffnet eine Datei (oder ein anderes Gerät) entsprechend pathname und flags. Die exakte Bedeutung dieser beiden Parameter, sowie eventuelle Fehlerwerte, die in der Variablen errno geliefert werden, wurden vom UNIX open übernommen und sind der Systemdokumentation zu entnehmen. Zurückgegeben wird ein Dateideskriptor, über den die geöffnete Datei anschließend angesprochen werden kann, der jedoch, sofern möglich, in einen nichtblockierenden Modus geschaltet wurde. Aus diesem Grund sollte man auf einen so angelegten Deskriptor nicht mehr mittels Standard C Funktionen zugreifen. Der letzte Parameter timeout bestimmt, wie lange auf die erfolgreiche Öffnung der Datei gewartet werden soll. Der gegebene Wert wird als Millisekunden (ms) interpretiert. Ist ein Timeout nicht erwünscht, sollte timeout auf 0 gesetzt werden. T_open kehrt dann erst entweder bei Auftreten eines Fehlers oder bei erfolgreicher Öffnung der Datei zum aufrufenden Thread zurück.

Anmerkung:
In der Regel blockiert die T_open Anweisung nicht, wenn es sich bei der zu öffnenden Datei um eine echte Datei handelt. Jedoch blockiert das Öffnen einer „Named Pipe" zum Schreiben normalerweise den Aufrufer, bis mindestens ein Prozeß diese Datei auch zum Lesen öffnet. Auf diese Weise wird ein Rendezvous zwischen beiden Partnern eingeleitet. Eine gut entworfene Applikation sollte jedoch damit umgehen können, wenn dieses Ereignis niemals eintritt und sich nicht für alle Zeiten blockieren.

```
int T_close (int fd)
```

Mittels T_close kann eine Datei, repräsentiert durch einen Dateideskriptor, wieder geschlossen werden. Der Aufruf dieser Funktion führt niemals zur Blockierung, weswegen T_close den Parameter fd ohne weitere Operationen an close weiterreicht. Mögliche Return- und Fehlerwerte sind daher der Dokumentation des close Befehls zu entnehmen.

Anmerkung:
Wenn Dateien nicht mehr benötigt werden, sollten sie mit dieser Funktion auch geschlossen werden.

```
int T_read (int fd, char *buf, size_t count, long timeout)
```

T_read dient zum Einlesen von Daten von einem Deskriptor[23]. Dieser wird in fd angeben, buf sollte auf den Anfang eines Speicherbereichs der Größe count Bytes zeigen, in dem eingelesene Daten abgelegt werden können. T_read liest maximal count Bytes, ggf. aber weniger. Die Anzahl tatsächlich eingelesener Bytes ist im Erfolgsfalle der Rückgabewert. Eine zurückgegebene 0 deutet auf ein Ende der Datei hin, ein -1 auf einen Fehler; Details hierzu entnehme man der Dokumentation von read. Sind beim

[23] Dies kann auch ein Socketdeskriptor sein.

Leseversuch keine Daten an `fd` verfügbar (aber auch kein Fehler aufgetreten), wird max. `timeout` Millisekunden gewartet und der aufrufende Thread solange blockiert. Ist `timeout` gleich 0, wird bis zum Eintreffen von mindestens einem Byte an `fd` oder bis zum Auftreten eines Fehlers gewartet.

```
int T_write (int fd, const char *buf, size_t count, long timeout)
```

Um Daten auf einen Deskriptor zu schreiben, kann `T_write` verwandt werden. Paramter sind `fd`, der zu beschreibende Deskriptor, `buf`, ein Zeiger auf den Anfang des zu schreibenden Speicherbereichs, `count`, die Anzahl der zu schreibenden Bytes und `timeout`, die Zeitspanne in Millisekunden, die bei erfolglosem Schreiben gewartet wird.

3.8.9 Verbindungsloser UDP Dienst

Ein weiterer Dienst, der z.Zt. jedoch das Beta-Stadium noch nicht überschritten hat, ist ein verbindungsloser UDP Dienst. UDP ist ein auf IP aufsetzendes Protokoll (siehe Abschnitt 2.5.4), mit dessen Hilfe Datagramme zwischen verschiedenen Rechnern ausgetauscht werden können. Im Gegensatz zu TCP ist UDP nicht notwendigerweise verbindungsorientiert, weswegen es sich zur Gruppenkommunikation (Multicast) anbietet. Da außerdem Paketgrenzen eingehalten werden, eignet es sich sehr gut zur Realisierung von RPC-Mechanismen. Leider garantiert die UDP-Schnittstelle nicht die sichere Übertragung. Fehlerhafte Pakete werden stillschweigend verworfen, kommen also in der höheren Schicht nicht an. Ziel des hier angebotenen Dienstes ist es, die Übertragung mittels UDP zu sichern und eine Schnittstelle anzubieten, die der Programmierung mit T entgegenkommt. Dazu wird ein sogenanntes Sliding Window Protokoll (siehe [Tanenbaum, 1992a]) auf UDP aufgesetzt, welches auf effiziente Art und Weise die Kommunikation sichert. Alle zur Verfügung gestellten, potentiell blockierenden Funktionen lassen auch eine nichtblockierende, asynchrone Arbeitsweise zu. Desweiteren gilt, daß sowohl IP-Adressen, als auch Port Nummern von allen Funktionen in „Host Byte Order“ erwartet werden.

```
int T_port_install (T_Port *port, short own_udp_port)
```

`T_port_install` installiert eine Sender-Empfänger Struktur `port` vom Typ `T_Port`, die bereits allokiert sein muß. Der dabei verwendete Kommunikationkanal wird mit dem Parameter `own_udp_port` übergeben und sollte systemweit noch nicht in Benutzung sein. Ein `T_Port` besteht aus (siehe auch Abbildung 11):

1. einer Nachrichtencontainerverwaltung,
2. einem UDP Kommunikationsport,
3. einer Hashtabelle für Verbindungen zu anderen Rechnern und

4. einer Warteschlange, in der empfangene Nachrichtencontainer zur Auslieferung abgelegt werden.

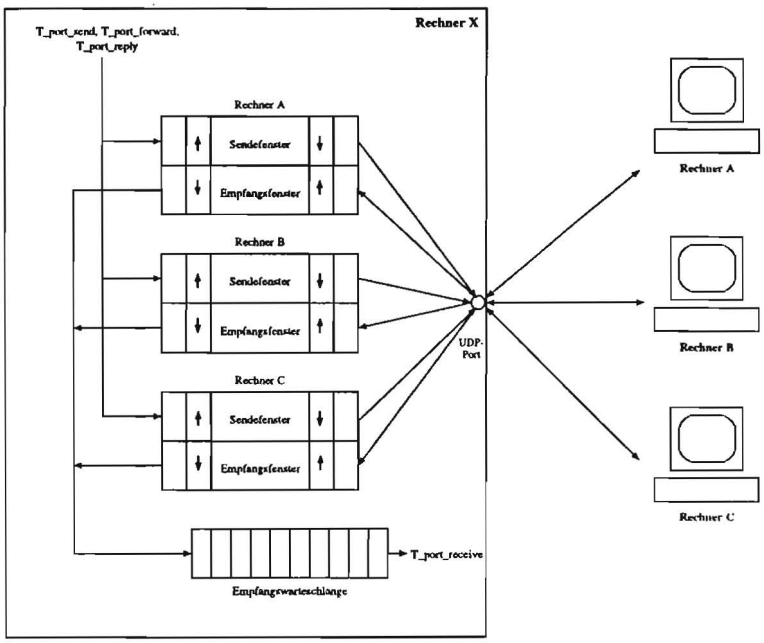

Abbildung 11: Bestandteile von T_Port

Zusätzlich wird für jede Verbindung zu anderen Rechnern eine Datenstruktur zur Unterstützung eines Sliding-Window Protokolls angelegt. Für eine T_Port-Struktur werden zwei Objekte für die Verwaltung der Nachrichtencontainer und der Empfangswarteschlange benötigt. Hinzu kommt jeweils ein Objekt für jede Verbindung zu einer anderen Maschine. Hierauf sollte bei der statischen Konfiguration des Systems geachtet werden (siehe auch Abschnitt 3.8.10). Die erfolgreiche Installation wird mit dem Wert 1, ansonsten mit dem Wert 0, quittiert.

`void T_port_uninstall(T_Port *port)`

`T_port_uninstall` deinstalliert einen `T_Port` und gibt dabei alle allokierten Ressourcen wieder frei. Noch in den Sliding-Windows befindliche Daten werden gelöscht.

`void *T_port_alloc (T_Port *port, long reply)`

`T_port_alloc` allokiert einen Nachrichtencontainer, der später an einen anderen Rechner übermittelt werden kann. Ein solcher Nachrichtencontainer hat die Größe `MAX_PACKET_SIZE` Bytes, welche in der Größenordnung eines Ethernet Pakets liegt. Welche

Nachrichtencontainerverwaltung mit der Allokation beauftragt wird, regelt der Parameter `port`, wobei der Benutzer selbst darauf achten muß, daß beim `T_Port x` allokierte Container nicht über den `T_Port y` verschickt werden. Der Parameter `reply` regelt die Vorgehensweise bei nicht vorhandenen Nachrichtencontainern. Ist `repl` $= 0$, so wird der Aufrufer bis zu deren Verfügbarkeit blockiert, ist hingegen `reply` $\neq 0$, so wird mit dem Returnwert `(void *) 0` zurückgekehrt, und der Aufrufer erhält bei Verfügbarkeit weiterer Container eine Nachricht mit den Werten $(id, value)$ und $id =$ `repl`, $value =$ undefiniert, zurück. Er kann dann einen neuen Allokationsversuch starten. Im Erfolgsfall liefert der Aufruf von `T_port_alloc` einen Zeiger auf den allokierten Nachrichtencontainer zurück.

```
void T_port_free (T_Port *port, void *message)
```

Die Funktion `T_port_free` ist das Pendant zu `T_port_alloc` und gibt einen zuvor allokierten Nachrichtencontainer (übergeben als Zeiger in `message`) wieder frei. Zusätzlich muß der Benutzer via `port` die Nachrichtencontainerverwaltung angeben, der der Container zuvor entnommen wurde. Die Freigabe von Nachrichtencontainern ist nur bei empfangenen Nachrichten nötig, allokierte und versandte Container werden vom System selbständig freigegeben, da keine Aussage darüber gemacht werden kann, wie lange diese im Sliding-Window verbleiben.

```
int T_port_send (T_Port *port, ulong to, short foreignport,
                 void *message, int length, long reply)
```

Via `T_port_send` kann ein zuvor allokierter (und gefüllter) Nachrichtencontainer an einen Empfänger mit der Adresse $(Rechner : Port) =$ `(to:foreignport)` verschickt werden. Zum Einsatz kommt dabei das verbindungslose UDP-Protokoll, welches durch ein geeignetes Sliding Window Protokoll abgesichert wird. Der eigene Port samt Verwaltungsstrukturen, mit deren Hilfe die Kommunikation abgewickelt werden soll, wird durch `port` spezifiziert, der Nachrichtencontainer selbst durch `message`, die Länge der darin enthaltenen Nutzdaten durch `length` (Nachrichtencontainer haben unabhängig von den Nutzdaten immer eine feste Länge `MAX_PACKET_SIZE`). Der Parameter `reply` hat eine ähnliche Bedeutung wie bei `T_port_alloc`. Gilt `reply` $= 0$, so wird der Aufrufer bis zur endgültigen Zustellung seines Containers blockiert, ist hingegen `reply` $\neq 0$, so wirkt `T_port_send` niemals blockierend, sondern meldet mit einer Nachricht $(id, value)$ und $id =$ `reply`, $value =$ undefiniert, die erfolgreiche Übermittlung. In jedem Fall wird der Container von der Verbindungsverwaltung entgegengenommen und ins interne Sliding Window gestellt. Die Returnwerte von `T_port_send` sind somit im Erfolgsfall 1, ansonsten 0 (nur bei `reply` $\neq 0$).

```
void *T_port_receive (T_Port *port, long reply)
```

T_port_receive dient dem Empfang von Nachrichtencontainern. Der Parameter port bestimmt dabei, von welcher Empfangswarteschlange die Nachricht entgegengenommen werden soll, d.h. über welchen lokalen UDP-Port sie empfangen wurde. Ist zum Zeitpunkt des Aufrufs kein Nachrichtencontainer verfügbar, so entscheidet der Parameter reply über das weitere Vorgehen. Gilt reply = 0, so wird der Aufrufer bis zum Eintreffen der nächsten Nachricht blockiert, gilt reply $\neq$ 0, so wird sofort zum Aufrufer zurückgekehrt und er erhält zusätzlich beim Eintreffen neuer Nachrichtencontainer über port eine Nachricht (*id*, *value*) mit dem Inhalt *id* = reply und *value* = undefiniert. Empfangene Nachrichtencontainer sollten nach Gebrauch mittels T_port_free wieder freigegeben werden, da pro T_Port-Sender-Empfänger Struktur nur eine begrenzte Anzahl zur Verfügung stehen.

```
int T_port_forward (T_Port *port, ulong to, short foreignport,
                    void *message, int length, long reply)
```

T_port_forward leitet einen zuvor empfangenen Nachrichtencontainer message mit ggf. geänderten Nutzdaten der Länge length an einen Empfänger mit der Adresse (*Rechner* : *Port*) = (to:foreignport) weiter. Diese Weiterleitung bleibt für den Empfänger transparent, lediglich intern ist natürlich der echte Absender bekannt. Zur Erklärung des Parameter reply siehe T_port_send. Handelt es sich bei message um einen „frischen" Nachrichtencontainer, bei dem die (für den Benutzer nicht sichtbare) Sendermarkierung nicht gesetzt ist, so ist der Absender für den Empfänger nicht mehr feststellbar. Solche Container sollten unbedingt mit T_port_send verschickt und die Benutzung von T_port_forward auf zuvor empfange Container beschränkt werden. Returnwerte sind im Erfolgsfall 1, ansonsten 0 (siehe auch hier T_port_send). Sowohl weitergeleitete als auch zurückgeschickte Nachrichtencontainer dürfen nicht mehr freigegeben werden.

```
int T_port_reply (T_Port *port, void *message, int length, long reply)
```

T_port_reply schickt einen zuvor empfangenen Nachrichtencontainer message mit ggf. geänderten Nutzdaten der Länge length an den Absender zurück. Auch hierbei sollte sichergestellt werden, daß es sich tatsächlich um einen empfangenen Nachrichtencontainer mit gesetzter Absendermarkierung handelt. Sowohl die Bedeutung des Parameters reply, als auch die Returnwerte sind analog der Funktion T_port_send. Sowohl weitergeleitete, als auch zurückgeschickte Nachrichtencontainer dürfen nicht mehr freigegeben werden.

```
ulong T_port_sender_of (void *message)
```

T_port_sender_of ermittelt die IP-Adresse des Absender des Nachrichtencontainers message. Diese wird als *unsigned long*-Wert zurückgegeben und kann z.B. mittels der

Funktion inet_ntoa auf die „dotted“ Notation abgebildet werden. Beachten sollte man, daß die IP-Adresse des Absenders bereits in „Host Byte Order“ zurückgegeben wird und manche Systemfunktionen die „Network Byte Order“ benötigen.

```
short T_port_udpport_of (void *message)
```

T_port_udpport_of ermittelt den Absenderport des Nachrichtencontainers message und gibt ihn bereits nach „Host Byte Order“ konvertiert zurück.

```
int T_port_length_of (void *message)
```

T_port_length_of ermittelt die Länge der Nutzdaten des Nachrichtencontainers message.

3.8.10 Statische T Konfigurationsparameter

In diesem Abschnitt sollen Konfigurationsparameter von T vorgestellt werden, deren Änderung eine Neukompilation nach sich zieht, also „statisch“ sind. Alle Konfigurationsparameter sind symbolische Konstanten und in der Datei TSystem.h definiert.

```
#define T_USER_PRIORITY 0
#define T_SYSTEM_PRIORITY 10
```

Mit den beiden Konstanten T_USER_PRIORITY und T_SYSTEM_PRIORITY wird die Anzahl der Prioritätsstufen festgelegt, die sich ergibt aus: N_{prio} = T_SYSTEM_PRIORITY − T_USER_PRIORITY + 1. Das System legt für jede Prioritätsstufe Listen an, in denen Objekte bei Bereitschaft, Blockierung etc. abgelegt werden, daher ist unbedingt erforderlich, daß gilt:
T_SYSTEM_PRIORITY > T_USER_PRIORITY.
Voreingestellt sind T_USER_PRIORITY = 0 und T_SYSTEM_PRIORITY = 10, also 11 Prioritätsstufen.

```
#define T_MAX_THREADS 32 /* max. Anzahl Threads */
```

T_MAX_THREADS gibt an, wieviele Threads maximal erzeugt werden können. Der Speicher für jeden Thread wird bei Programmstart einmal allokiert; sollte dies nicht möglich sein, wird das Programm unmittelbar abgebrochen. Aus diesem Grund kann die Verfügbarkeit der über diese Konstante eingestellten Anzahl von Threads (Objekte) bei Programmausführung garantiert werden. Wieviel Speicher ein Thread tatsächlich benötigt, hängt von der zugrundegelegten Rechnerplattform und auch von den nachfolgenden Konfigurationsparametern ab. Die Voreinstellung beträgt 32.

```
#define T_STACKSIZE 0x2000 /* die Stackgroesse einzelner Threads */
```

Jedes T Objekt arbeitet auf einem eigenen Stack, dessen Größe pro Objekt mit der Konstanten T_STACKSIZE festgelegt werden kann. Dieser Parameter sollte nicht zu klein gewählt werden, da ein Stacküberlauf die Systemintegrität verletzen kann. Der voreingestellte Wert 0x2000 hat sich in den meisten Fällen als adäquat herausgestellt. Finden jedoch große automatische Strukturen oder Felder Verwendung, so muß er entsprechend erhöht werden, oder aber bei der Objekterzeugung ein eigener Stack für ein solches Objekt angegeben werden (siehe Abschnitt 3.8.1).

```
#define T_MESSAGES_PER_THREAD 16
```

Jede Nachricht, die von einem Objekt versandt wird, benötigt etwas Speicherplatz, bis sie die *IN*-Queue des Empfängers passiert hat und empfangen wurde (ggf. noch länger, wenn sie in der *SAVE*-Queue gesichert wird). Mit der Konstante T_MESSAGES_PER_THREAD kann die Anzahl der vom System allokierbaren Nachrichten auf eine „pro-Thread"-Basis eingestellt werden. Trotzdem steht der letztendlich erzeugte Nachrichtenpool allen Objekten uneingeschränkt zur Verfügung, d.h. ein Objekt ist durchaus in der Lage, mehr als die hier eingestellte Anzahl an Nachrichten zu empfangen/versenden, sofern noch Nachrichtenblöcke frei sind.

```
#define T_ADDITIONAL_MEM_SIZE 0x80000
```

Wie bereits erwähnt, stellt das System in der Variablen Memory ein Speichersegment zur Verfügung, das auch vom Benutzer verwandt werden kann. Dessen Größe ergibt sich aus der Konstante T_ADDITIONAL_MEM_SIZE, die auch auf Null gesetzt werden kann, wenn kein Systemspeichersegment benötigt wird.

```
#define T_MAX_SIGNALS 32
```

Zur Verwaltung der Signalobjekte verwendet das System ein Array, in dem entsprechende Verweise eingetragen werden. Die Anzahl der Felder in diesem Array muß von der Größe der Anzahl der zu verwaltenden Signale entsprechen, die mit der Konstanten T_MAX_SIGNALS festgelegt wird. Da die meisten UNIX Derivate 32 verschiedene Signale kennen, ist dies auch die Voreinstellung.

4 Konzepte des entwickelten Frameworks

Nachdem die begrifflichen Grundlagen eingeführt wurden, soll nun das Problem der Prozeßkommunikation vor dem entsprechenden VEGA-Kontext angegangen werden. Hierzu noch einmal die einzuhaltenden Zielvorgaben aus Kapitel 1:

1. einfache Anforderungen an die Entwicklungssprache bzgl. deren Möglichkeiten zur Systemprogrammierung,
2. geringe Belastung der Entwickler mit den Problemen der Synchronisation von Prozessen und Verwaltung benötigter Ressourcen,
3. Bereitstellung sicherer Dienste,
4. die Mechanismen der Prozeßkommunikation selbst sollen möglichst wenig Rechenzeit verbrauchen.

In Abschnitt 4.1 wird der bisher verschobene Vergleich von konkurrierenden und iterativen Servern gegeben. Vor diesem Hintergrund und von oberen Anforderungen ausgehend, wird in Abschnitt 4.2 dann ein adäquates Szenario entwickelt, dessen Realisation unter Zuhilfenahme der in 3 vorgestellten Thread Bibliothek in Abschnitt 4.3 vorgestellt wird.

Im folgenden werden zunächst noch ein paar Begriffe festgelegt, auf die in den kommenden Abschnitten immer wieder Bezug genommen wird.

1. Prozesse, die miteinander kommunizieren wollen, sollen fortan *Benutzerprozesse* (engl. **U**ser **P**rocesses, kurz **UP**) genannt werden. Im folgenden wird ein Server entwickelt, der die Kommunikation der UP's regelt und auch in der Lage ist, selbständig Prozesse zu erzeugen und mit Kommunikationskanälen zu versorgen. Der Gesamtablauf der verteilten Anwendung könnte wie in Abbildung 12 aussehen.
2. Die Gesamtheit aller mittels eines Servers verbundenen UPs heißt *Prozeßnetz*, kurz *Netz*. Wird in den nächsten Abschnitten auf ein „echtes" Netzwerk Bezug genommen, so wird dies ausdrücklich erwähnt.

Die Requests, die der Server anbietet, sind wie folgt:

1. Dienst zur Erzeugung anderer Benutzerprozesse inklusive Zuordnung zu einem Spitznamen: `NEWPROCESS`–Request
2. Dienst zur Unterstützung der Nachrichtenübertragung zwischen einzelnen UP's: `TRANSMIT`–Request
3. Möglichkeit für einen UP einen anderen oder sich selbst zu terminieren: `KILLPROCESS`–Request
4. Abfragen zur Ermittlung des Status des so erzeugten Prozeßnetzes, insbesondere Suchanfragen zur Identifikation anderer Benutzerprozesse: `GETIDBYPATH`–Request, `GETIDBYNICKNAME`–Request

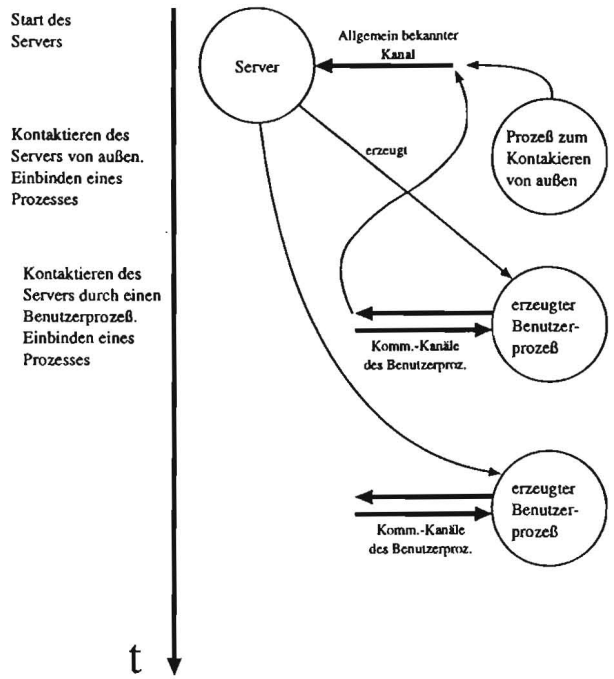

Abbildung 12: Initialer Zustand und Erzeugung von Prozessen

Die exakte Semantik und das Protokoll dieser Requests ist im Anhang A angegeben.

4.1 Konkurrierende versus iterative Server

Eine grundlegende Designentscheidung bei der Entwicklung von Servern ist, ob sie nur einen oder mehrere Clientrequests zur gleichen Zeit bearbeiten können. Im ersten Fall heißen solche Server auch *iterativ*, im zweiten Fall *konkurrierend*.

Die Entscheidung zugunsten einer dieser beiden Varianten muß sehr sorgfältig getroffen werden, da sie sich meistens tief in der Softwarestruktur widerspiegelt. Die Kriterien, nach denen sie erfolgen sollte, sind zum einen die intendierte Funktionalität und zum anderen das Design des Protokolls. Bestehen die Requests nur aus kurzen Sequenzen, kommt eine verbindungslose und ggf. iterative Variante (die meistens sehr viel einfacher zu implementieren ist) in Frage. Bei verbindungsorientierten Protokollen hingegen wählt man häufig konkurrierende Server. Man könnte zwar eine verbindungsorientierte Arbeitsweise mit Hilfe einer paketorientierten simulieren (das „Zerhacken" des Datenstroms kann auch dem Client überlassen werden), diese Vorgehensweise ist jedoch nicht problemangepaßt und führt zu komplexeren Softwarestrukturen.

Bezüglich des Durchsatzes unterscheiden sich iterative und konkurrierende Server zunächst nicht, wenn man davon absieht, daß ein konkurrierender Server immer einige Zeit mit dem Scheduling beschäftigt ist und ein Client seine Requests zügig abliefert. Daher haben auch iterative Server ihre Berechtigung, wenn das verwaltete Objekt

keinen konkurrierenden Zugriff zuläßt und Requests kurz genug sind, um hinreichend schnell interpretiert zu werden. Es muß allerdings noch bedacht werden, inwiefern sichergestellt ist, daß der Client sich beim Request „korrekt" verhält. Ist zu befürchten, daß ein Request nicht innerhalb einer bestimmten Zeitspanne übermittelt wird, so muß man nach entsprechender Zeit mit einem `TIMEOUT` reagieren. Ansonsten besteht die Gefahr, daß der Client den Server „aushungert" und damit alle anderen Clients längerfristig blockiert. Bei konkurrierenden Servern kann dieser Zustand nicht eintreten, allerdings wird auch hier auf `TIMEOUTS` reagiert, um belegte Kommunikationsressourcen wieder zu deallokieren.

Grundsätzlich hat der Entwurf eines konkurrierenden Servers nicht unbedingt etwas mit dessen Verteilung auf mehrere Prozesse zu tun, obwohl in der Praxis tatsächlich viele Server so entworfen sind. Vielmehr ist es eine Frage der Interpretation des Protokolls, welches mit entsprechend entworfenen Automaten konkurrierend abgearbeitet werden kann, obwohl nur ein Prozeß daran beteiligt ist. Dies hat oft Vorteile, wenn dynamische Informationen allen Clients zur Verfügung stehen sollen. Diese müßten bei einer prozeßverteilten Variante in einem allen Prozessen zugänglichen Speicherbereich aufbewahrt oder aber mittels interner Kommunikation zwischen den Serverprozessen ausgetauscht werden. Im Gegenzug hat ein konkurrierender Einprozeßserver den Nachteil, Hardwareparallelität nicht ausnutzen zu können, wenn er nicht in einer „multithreaded"-Version vorliegt, und die Threads auf verschiedene Prozessoren verteilt werden können.

Im hier konkret vorliegenden Fall kann der Server nicht iterativ entworfen werden, da ein Übermittlungsdienst verfügbar gemacht wird, der streamorientiert ist (vgl. `TRANSMIT`–Request in Abschnitt A), d.h. die Anzahl der zu übertragenden Zeichen kann unbegrenzt sein. Damit mehr als nur zwei Prozesse miteinander kommunizieren können, muß der Server daher konkurrierend entworfen werden.

4.2 Szenario

Aufbauend auf den Zielvorgaben und geforderten Funktionalitäten lassen sich nun genauere Anforderungen formulieren, die einzuhalten sind.

1. Keine direkten Systemaufrufe zur Kreation bzw. zum Umgang mit Systemressourcen. „Named Pipes" dürfen zwar benutzt werden, ihre Kreation sollte jedoch nicht der Benutzeranwendung überlassen sein, da hierfür die Systemfunktion `MKNOD` benötigt wird. Die Benutzeranwendung sollte nur einen einfachen Kommunikationsmechanismus „sehen", ohne sich Gedanken über die tatsächlich zum Einsatz kommenden Mechanismen machen zu müssen.

2. Keine Belastung der Benutzeranwendung mit der Verwaltung irgendwelcher Kommunikationsressourcen. Evtl. allokierte Ressourcen müssen wieder freigegeben werden, ohne daß es der Mitarbeit der Benutzeranwendung bedarf. Terminierende Kommunikationspartner müssen erkannt und weitere Sendewünsche unterbunden werden, ohne daß ein inkonsistenter Zustand auftritt.

3. Das für die Kommunikation verwendete Protokoll sollte zur Benutzerseite hin nur solche Informationen umfassen, die für die Anwendung von Bedeutung sind.

Insbesondere interne Verwaltungsinformationen sollten nicht von der Benutzeranwendung gesehen werden.

4. Die Benutzeranwendung sollte unabhängig von anderen agieren können. Insbesondere die Synchronisation bei der Benutzung von Kommunikationsmechanismen mit anderen Anwendungen durch eine Benutzeranwendung sollte transparent sein [24].

5. Die Kommunikation über Rechnergrenzen hinweg muß für die Benutzeranwendung zwar nicht vollkommen transparent sein[25], sollte sich jedoch nicht wesentlich von der lokalen Kommunikation unterscheiden.

Der Punkt 1 schränkt die Kommunikationsmechanismen, die an der Schnittstelle zur Benutzeranwendung stehen sollen, auf Dateien, Pipes oder „Named Pipes" ein. Sie alle erlauben die Benutzung nur durch einfaches Öffnen und Lesen. Da wegen Punkt 2 der Benutzeranwendung ein Prozeß zur Verwaltung an die Seite gestellt werden muß, kann man auf Realisierung mittels Dateien verzichten. Sie stellen sich, einmal vorhanden, der Benutzeranwendung weitestgehend wie eine „Named Pipe" dar, diese ist jedoch meistens effizienter. Die Entscheidung, ob „normale" oder „Named Pipes" verwendet werden sollen, hängt damit zusammen, ob die Benutzeranwendung in der Lage ist, Dateideskriptoren zu benutzen, die nicht explizit mittels eines `OPEN` bzw. äquivalenten Aufrufs angelegt wurden. Dies ist immer der Fall bei den Deskriptoren 0 bis 2, da unter UNIX die Vereinbarung besteht, daß diese Deskriptoren mit dem Standard Input, dem Standard Output sowie dem Standard Error Output korrespondieren. Jedes Programmiersystem hält für diese Kanäle symbolische Variablen bereit. Allgemeiner ist es jedoch, „Named Pipes" zu benutzen und dem Benutzerprozeß keine weiteren offenen Dateideskriptoren als von 0 bis 2 zu vererben, sondern den Namen der „Named Pipes" zu übergeben. Aus diesem Grund werden in allen folgenden Lösungen den Benutzerprozessen drei „Named Pipes" als Kommunikationskanäle zur Verfügung gestellt, die zur einfacheren Verständigung mit `INPIPE`, `OUTPIPE` und `ACKPIPE` bezeichnet werden sollen, was jedoch **nicht** bedeutet, daß die entsprechenden Pipes diese Dateinamen besitzen, sondern unter diesen Namen einem ins Netz eingebundenen Prozeß als Environmenteintrag übergeben werden. Daran gebunden sind dann die echten Namen der Pipes, die in der Regel in einem temporären Verzeichnis angelegt werden.
Achtung: die Namensgebung erfolgte aus der Sicht des Servers, d.h. daß z.B. die `OUTPIPE` ein Kanal aus dem Server heraus und damit für einen Benutzerprozeß ein Eingabekanal ist. Über die `INPIPE` kann ein UP also einen Request stellen, der ggf. zu einem anderen Server einer anderen Maschine weitergeleitet wird. Jeder Request wird unmittelbar beantwortet und dieses Ergebnis an der `ACKPIPE` zur Verfügung gestellt. Über die `OUTPIPE` werden Übertragungen anderer UPs übermittelt. Näheres zum Verhalten des Servers findet man in Abschnitt 4.3. In Abbildung 13 ist der Aufbau und das Zusammenspiel zwischen den Servern und Benutzerprozessen untereinander skizziert. Jeder Server besitzt zusätzlich noch öffentliche, d.h. allgemein zugänliche, irgendwo im

[24] Dies impliziert nicht, daß nun keinerlei Blockierungen mehr vorkommen können.
[25] Insbesondere bei Adressierungsschemata ist dies sehr schwer

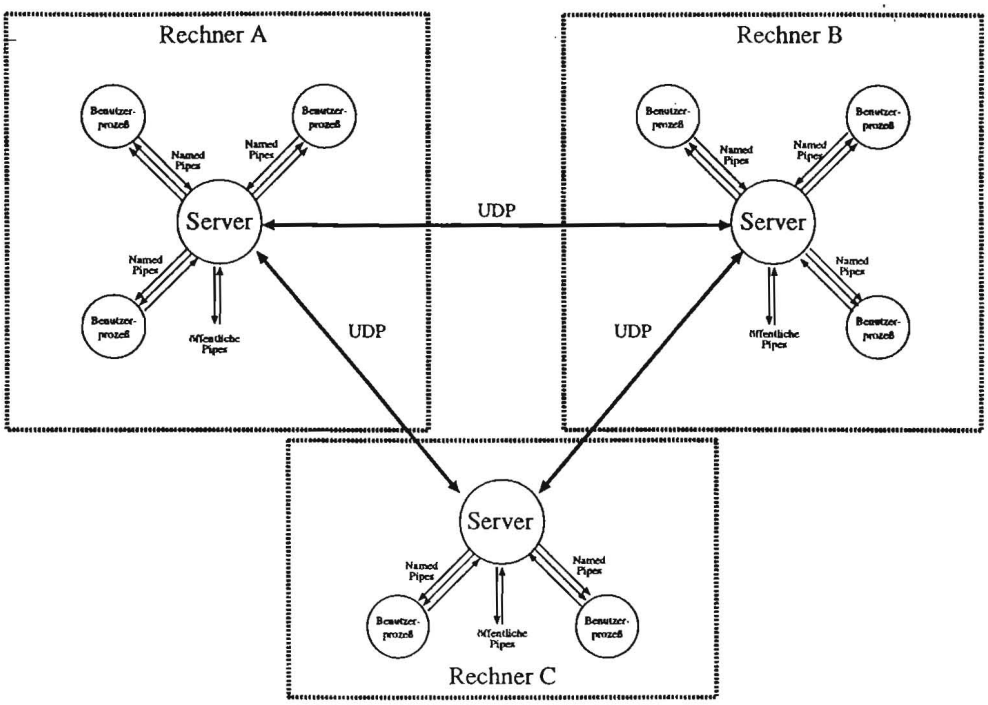

Abbildung 13: Der Server eines entfernten Rechners wird analog eines Benutzerprozesses angesprochen und übernimmt die weitere Verarbeitung.

Dateisystem abgelegte „Named Pipes“, die von jedem Prozeß, der über entsprechende Schutzrechte verfügt, benutzt werden können, um weitere UPs in das Netz einzubinden. Häufig geschieht dies nur bei der Initialisierung, weitere UPs werden dann von zuvor eingebundenen UP erzeugt.

4.3 Ein konkurrierender Server, entwickelt mit T

Wie bereits in Abschnitt 4.1 begründet, muß der Server in der Lage sein, Requests konkurrierend abzuarbeiten. D.h. er muß an allen Eingabekanälen gleichzeitig auf Daten warten und beim Schreiben nur solche Ausgabekanäle bedienen, die aufnahmebereit sind und daher keine Blockierung verursachen können.

Klassischerweise wird ein solches IO-getriggertes Scheduling mit dem Systembefehl `select` realisiert. Dabei werden Mengen von Ein- und Ausgabedeskriptoren[26] übergeben, und `select` kehrt bei „Bereitschaft" eines Deskriptors zurück. Anschließend ist garantiert, daß mindestens ein Byte gelesen bzw. geschrieben werden kann. Der Server kann die so eingelesenen Daten dann zweckmäßigerweise durch einen endlichen Automaten bearbeiten und führt anschließend wieder ein `select` aus. Da bei einem verbindungsorientierten Server Requests häufig nur partiell eingelesen werden, muß der Fortgang der Interpretation durch Speichern des Zustands des interpretierenden Automaten festgehalten werden. Dies führt normalerweise zu einer Softwarestruktur, die in [Gallmeister, 1995] als **Do-While**-Struktur bezeichnet wird. Virtuell existiert dann für jeden Client eine entsprechende Anzahl von endlichen Automaten, deren Kommunikation und Synchronisation untereinander durch globale Variablen realisiert wird, was die Wartung großer Server sehr erschwert.

Bei Verwendung von T (siehe Abschnitt 3) bietet sich nun ein anderes Vorgehen an. Wiederum werden zur Requestinterpretation bzw. Antwortübermittlung endliche Automaten herangezogen, jeder läuft jedoch als selbständiger Thread ab. Die Kommunikation und Synchronisation untereinander geschieht durch Versand und Empfang von Nachrichten und Aufträgen ggf. unter Zuhilfenahme weiterer Threads. Die dadurch erreichte Softwarestruktur ist weitaus robuster und läßt sich leichter beschreiben.

Abbildung 14 zeigt Teile der internen Struktur des Servers. Es ist erkennbar, daß sowohl alle Eingabe- als auch alle Ausgabedeskriptoren von eigenen Threads (In- bzw. Out-Thread) bedient werden. Über deren Struktur wird noch in Abschnitt 4.3.2 gesprochen. An dieser Stelle wichtig ist lediglich die Tatsache, daß die konkurrierende Ein- und Ausgabe des Servers durch geeignete Zuordnung von T Objekten zu Ein- bzw. Ausgabekanälen geschieht. Die Gesamtheit aller Objekte und deren Interaktion ist in Abschnitt 4.3.1 beschrieben.

4.3.1 Aufbau

In diesem Abschnitt sollen nun alle wesentlichen Bestandteile des Servers und deren Interaktionsprinzipien erklärt werden; siehe zunächst Abbildung 15. Zu erkennen sind acht T Objekte, wobei fünf unterschiedlichen Typs sind.

In-Threads nehmen Requests von UPs entgegen und sind für die Zustellung von Antworten verantwortlich, Out-Threads hingegen übertragen Nachrichten anderer UPs an den zugeordneten UP (vgl. `TRANSMIT`-Request). Damit eine Nachrichtenübermittlung von UP A an UP B nicht durch einen Übertragungswunsch von UP C an A gestört

[26] Deskriptoren sind Identifikatoren, die ein Prozeß unter UNIX vom System beim Öffnen einer Datei (auch „Named Pipe") bekommt, und unter denen er sie anschließend ansprechen kann.

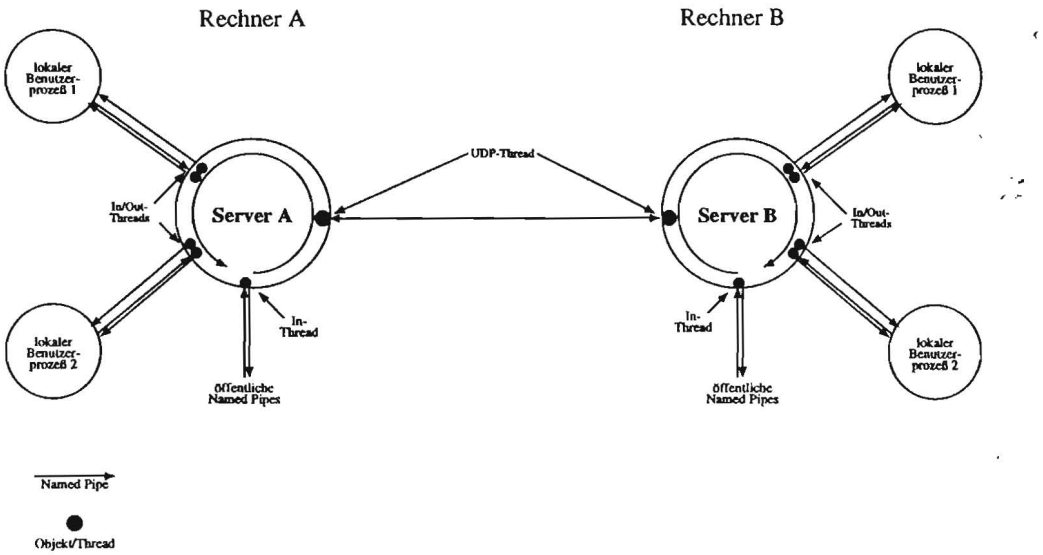

Abbildung 14: Innere Struktur des Servers

werden kann, werden Out-Threads vom Connection-Manager verwaltet. Zusätzlich existiert der UDP-Manager, dessen Aufgabe im wesentlichen darin besteht, eintreffende Requests zu interpretieren und ggf. zuzustellen. Root dient der Initialisierung des Servers und nimmt bei Terminierung eines UPs auch die Terminierung des zugeordneten In- bzw. Out-Threads sowie die Freigabe von Ressourcen („Named Pipes“ etc.) vor. Eine Sonderstellung nimmt der einzelne In-Thread ohne zugehörigen Out-Thread ein. Er interpretiert Requests, die über den öffentlichen Eingabekanal eintreffen. Dabei verhält er sich wie jeder andere In-Thread, d.h. er übermittelt ebenfalls eine Antwort über einen öffentlichen „Acknowledge“-Kanal, die jedoch nicht ausgelesen werden muß.

Im folgenden soll anhand des Aktionsschemas zweier Requests das Zusammenspiel der T Objekte untereinander demonstriert werden. In Abschnitt 4.3.2 wird die Funktion der Objekte detailliert „objektzentriert“ beschrieben.

NEWPROCESS–Request

Ausgangspunkt ist der vollständig eingelesene Request, der im Umgebungsspeicher des interpretierenden In-Threads abgelegt wird.

1. Erzeugen einer neuen UP-Verwaltungsstruktur *up*, die der neue In- bzw. Out-Thread als Umgebung bekommt. Kopieren der bisher eingelesenen Informationen wie Pfad, Argumente etc. in diese Struktur.

2. Erzeugen der neuen „Named Pipes“ in einem temporären Verzeichnis und Vermerken der Deskriptoren in *up*.

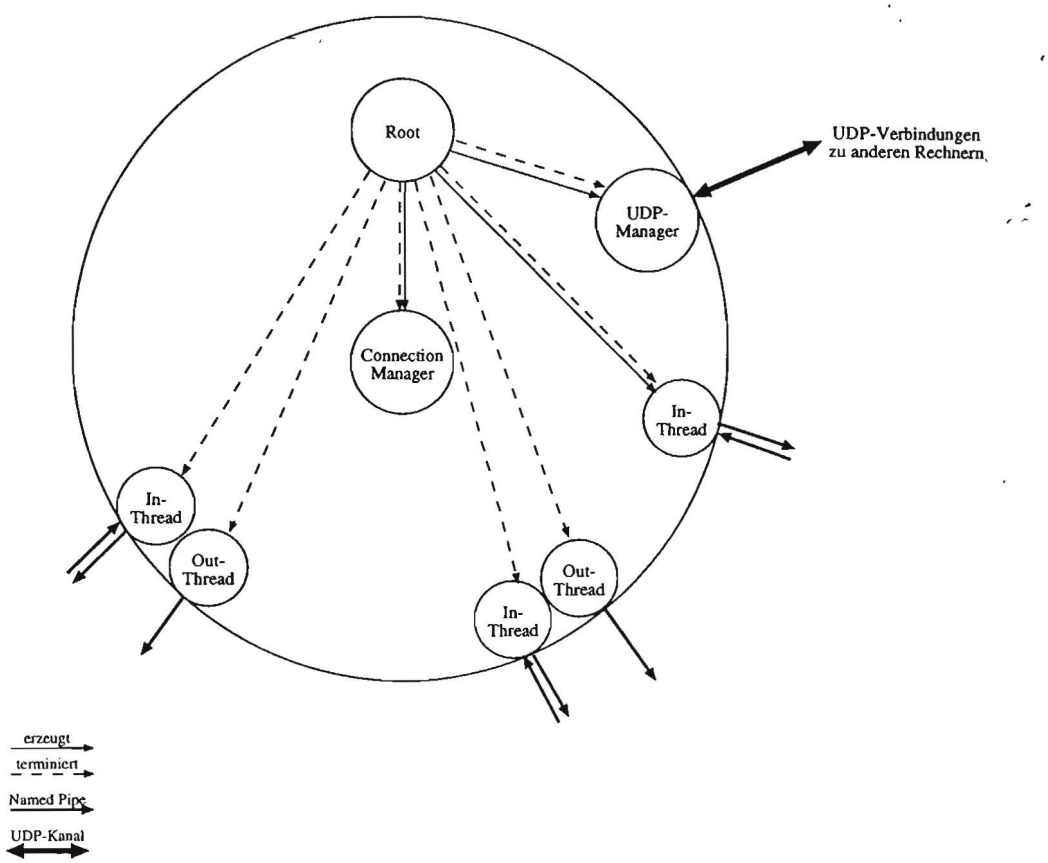

Abbildung 15: T Objekte des Servers

3. Abändern des dem neuen UP zu übergebenden Prozeßenvironments dahingehend, daß die neu erzeugten „Named Pipes“ als Einträge ausgelesen werden.

4. Erzeugen eines neuen In- bzw. Out-Threads zur Kommunikation mit dem neuen UP.

5. Erzeugen des neuen UP's mit entsprechendem Environment mittels `fork` und `exec`.

6. Rückgabe des Returnwertes von `fork` entsprechend vereinbartem Protokoll (siehe Anhang A). **Achtung**: dieser Returnwert sagt wenig über die erfolgreiche Erzeugung des neuen UP's sondern aus, sondern vielmehr darüber, daß alle Ressourcen (Threads, Pipes, Prozeß etc.) kreiert werden konnten. Ob z.B. der Pfad zum neuen UP stimmte, kann erst beim `exec` festgestellt werden, was jedoch von einem neuen Prozeß ausgeführt wird, der keine Antwort mehr übermitteln kann.

Der `NEWPROCESS`-Request ist ein relativ einfacher Request, bei dem es keiner besonderen Threadsynchronisation bedarf.

TRANSMIT-Request

Ausgangspunkt hier ist der Empfang der Pid des UP's, an den übermittelt werden soll (siehe Anhang A), also der Punkt, ab dem sicher ist, um was für einen Request es sich handelt.

1. Einlesen eines Zeichens c durch den In-Thread i des sendenden UP's in einen lokalen Puffer p.

2. Falls c=„\“,
   gehe zu 5,
   falls c=„<nl>“ oder p erschöpft,

   (a) falls Out-Thread des Empfängers *ethread* noch nicht allokiert,
   allokiere *ethread* beim Connection Manager (dabei ggf. Blockierung bis zu dessen Freiwerden).

   (b) Kopieren von p in einen neu allokierten Puffer *p2*.

   (c) Senden der Adresse von *p2* an *ethread*.

3. Falls c=„<nl>“,
   Ende.

4. Gehe zu 1.

5. Einlesen eines Zeichens ohne Interpretation nach p.

6. Gehe zu 1.

Beim TRANSMIT-Request werden eingehende Daten von dem dem sendenden UP zugeordneten In-Thread entgegengenommen und interpretiert. Die Zustellung an den Out-Thread des Empfängers kann jedoch nicht durch einfaches Zustellen der Daten via T_send erfolgen, da sichergestellt werden muß, daß nicht ein anderer Sender an den Empfänger überträgt. Aus diesem Grund werden Out-Threads vom Connection Manger verwaltet, bei dem sie allokiert werden müssen und der die Aufrufer ggf. blockiert. Insbesondere können Out-Threads nicht so ohne weiteres terminiert werden, da sie zum einen noch Speicherressourcen halten können, zum anderen auch potentielle Sender auf ihre Freigabe warten. Aus diesem Grund wird die Terminierung von Objekten immer durch Root vorgenommen, der alle entprechenden Objekte in Kenntnis setzt und den Server in einem konsistenten Zustand hält.

4.3.2 Funktionsweise der Objekte

Nachdem im vorherigen Abschnitt etwas zum Zusammenspiel der Objekte exemplarisch anhand zweier Requests gesagt wurde, soll hier nun der Aufbau einzelner Objekte vorgestellt werden.
Die dabei verwendeten Diagramme sind als Dokumentation zum Sourcecode zu verstehen.

In-Thread

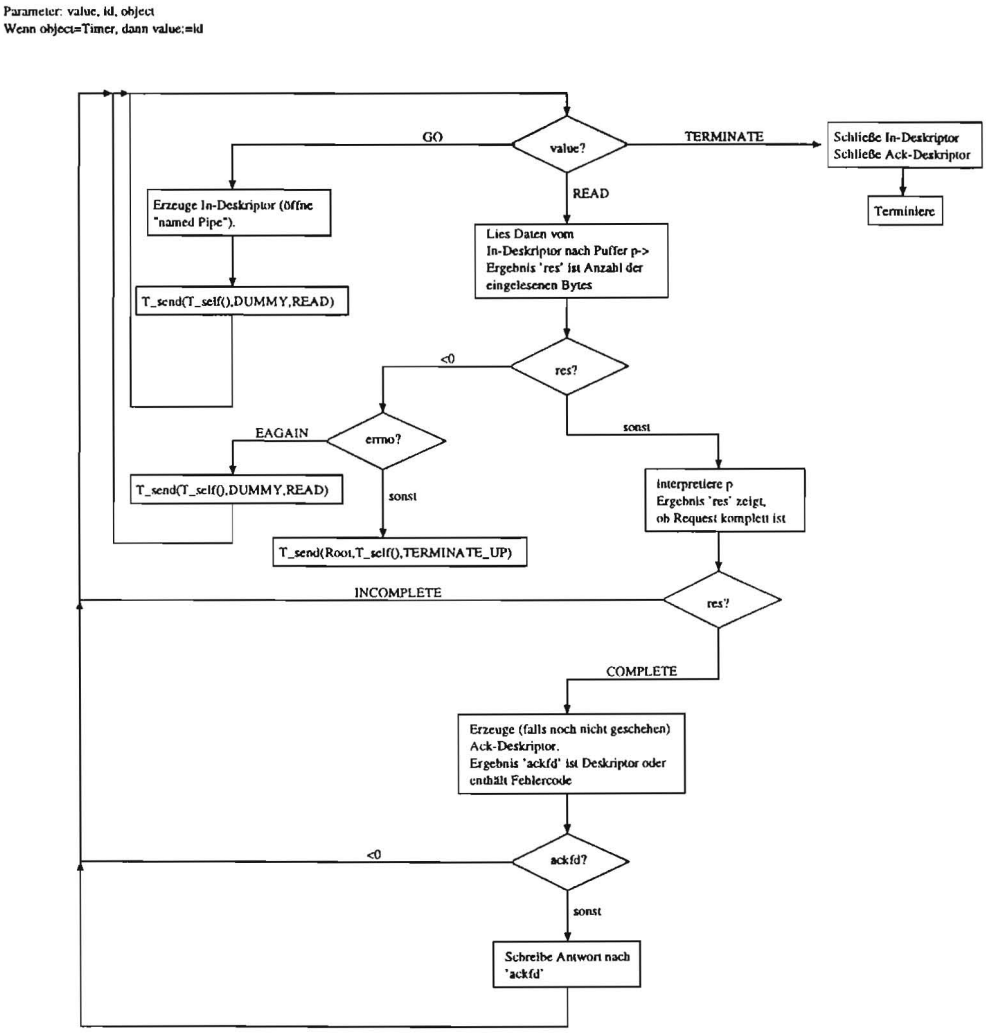

Abbildung 16: Ablaufdiagramm In-Thread

Anm.:

- Der In-Thread ist ein selbst aktivierendes Objekt, d.h. nach Abarbeitung seines Standarhandlers schickt er sich selbst wieder eine Nachricht, um einen erneuten Leseversuch anzustoßen.
- Sollte an der `INPIPE` ein schwerer Fehler auftreten, z.B. daß sie vom angeschlossenen Benutzerprozeß nach erstmaligem Öffnen geschlossen wurde, so veranlaßt der In-Thread die Terminierung von sich samt zugeordnetem Out-Thread und Benutzerprozeß.
- Wie der Benutzerprozeß mit der `ACKPIPE` umgeht spielt keine Rolle. Sollte eine Antwort jedoch nicht zustellbar sein, so wird erst nach Verstreichen des Timeout Wertes ein neuer Request entgegengenommen, es resultieren also zeitliche Verzögerungen.
- In der Abbildung stark vereinfacht wurde die partielle Interpretation von Requests, die durch einen endlichen Automaten vorgenommen wird, dessen Zustand im Environment des In-Threads festgehalten wird.

Out-Thread

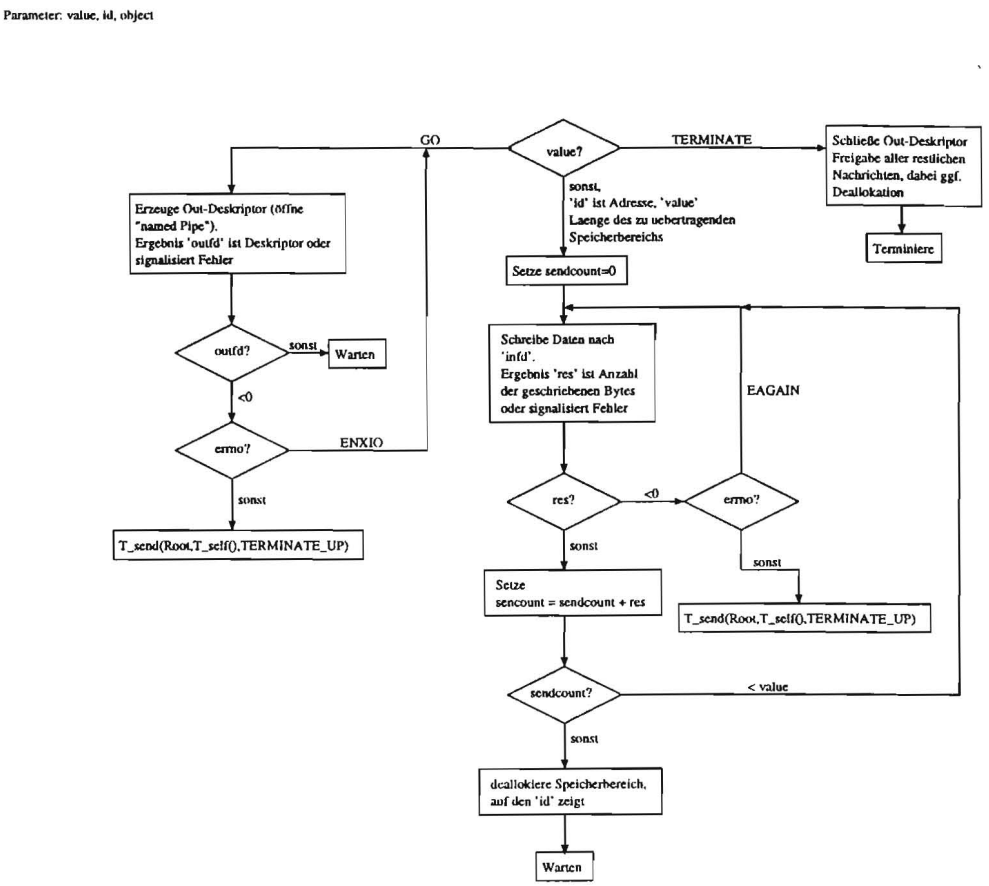

Abbildung 17: Ablaufdiagramm Out-Thread

Anm.:

- Der Out-Thread ist ein passsives Objekt, das sich normalerweise im Wartezustand befindet und erst durch den Empfang von Nachrichten anderer Ojekte aktiviert wird.
- Sollte an der `OUTPIPE` ein schwerer Fehler auftreten, z.B. daß sie vom angeschlossenen Benutzerprozeß nach erstmaligem Öffnen geschlossen wurde, so veranlaßt der Out-Thread die Terminierung von sich samt zugeordnetem In-Thread und Benutzerprozeß.

Connection Manager

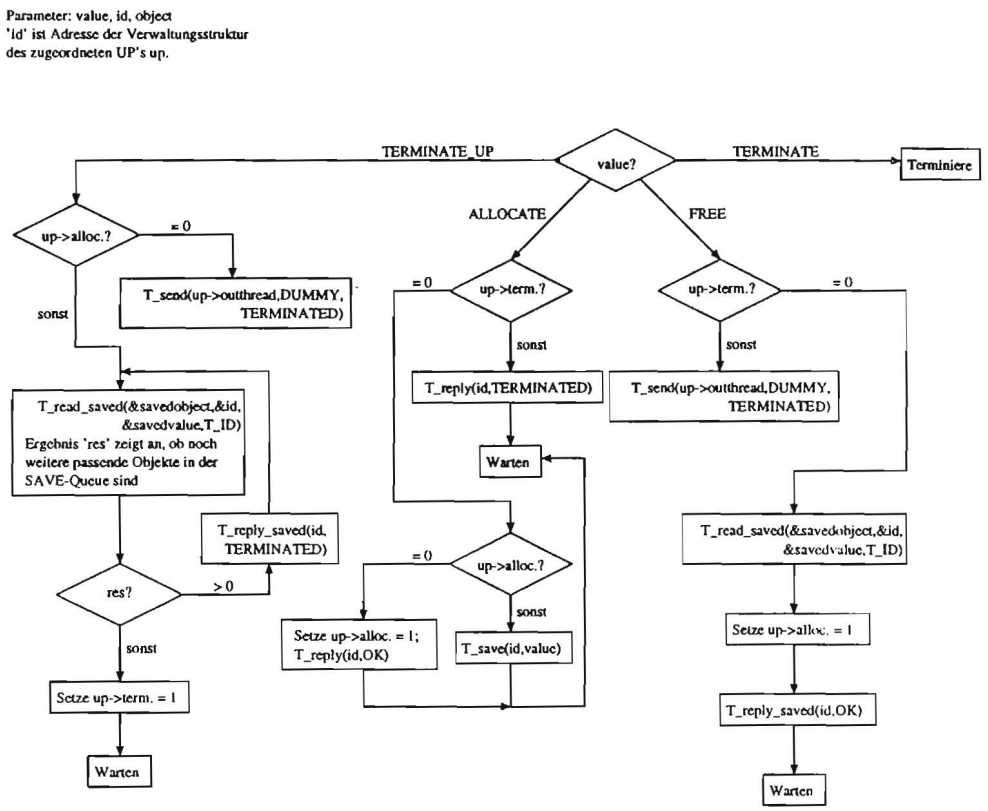

Abbildung 18: Ablaufdiagramm Connection Manager

Anm.:

- Der Connection Manager ist ein passives Objekt, das sich normalerweise im Wartezustand befindet.
- Er übernimmt die Terminierung von Out-Threads, die ggf. verzögert werden muß, wenn ein Out-Thread gerade allokiert ist, d.h. eine Nachricht eines anderen Benutzerprozesses überträgt.
- In einem solchen Fall, werden weitere Allokationsrequests abgewiesen und die eigentliche Terminierung wird vorgenommen, wenn der Out-Thread wieder freigegeben wird.
- Nach der endgültigen Terminierung eines Out-Threads sollten für diesen keine Allokationsrequests mehr gestellt werden, da die Objektadresse nicht mehr gültig ist. Ein solcher Fehler wird vom Connection Manager nicht aufgefangen.

Root

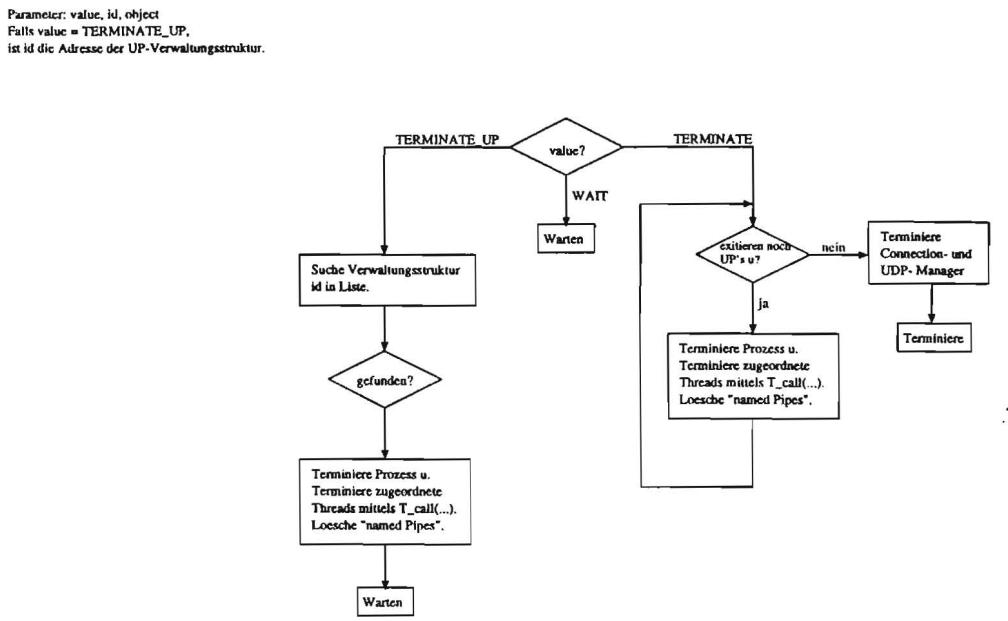

Abbildung 19: Ablaufdiagramm Root

Anm.:

- Das Root-Objekt ist ein passiver Thread, der kurz nach Prozeßinitialisierung, sowie zur Terminierung von einzelnen UP's oder des gesamten Prozeßnetzes aktiviert wird.
- Die Terminierung aller Objekte des Servers wird von ihm initiiert, mit Ausnahme die der Out-Threads, womit der Connection Manager beauftragt wird.
- Soll Root selbst terminieren, werden zuvor auch alle anderen Objekte terminiert.

5 Zusammenfassung und Ausblick

In den vorhergehenden Abschnitten wurde ein Server entwickelt, der eine einfache Kommunikation zwischen ggf. auf verschiedenen Rechnern laufenden Prozessen und somit die Verteilung einer Applikation ermöglicht. Dabei wurde besonderer Wert darauf gelegt, daß von seiten der Applikationsentwickler keinerlei Erfahrung im Umgang mit Betriebssystem und Netzwerken vorhanden sein muß. Alle Funktionalitäten sind über einfache Schnittstellen zugänglich, die in nahezu jedem Programmiersystem vorhanden sind, so daß insbesondere die Portierung solcher Applikationen nur vom Vorhandensein des hier entwickelten Servers auf dem Zielrechner abhängt. Dies war eine der Hauptzielsetzungen, da es gerade im Rahmen prototypischer Softwareentwicklung zu aufwendig wäre, solche Gesichtspunkte zu berücksichtigen. Andererseits muß man zu Demonstrationszwecken lauffähige Versionen für verschiedene Rechner- und Systemplattformen bereit halten.

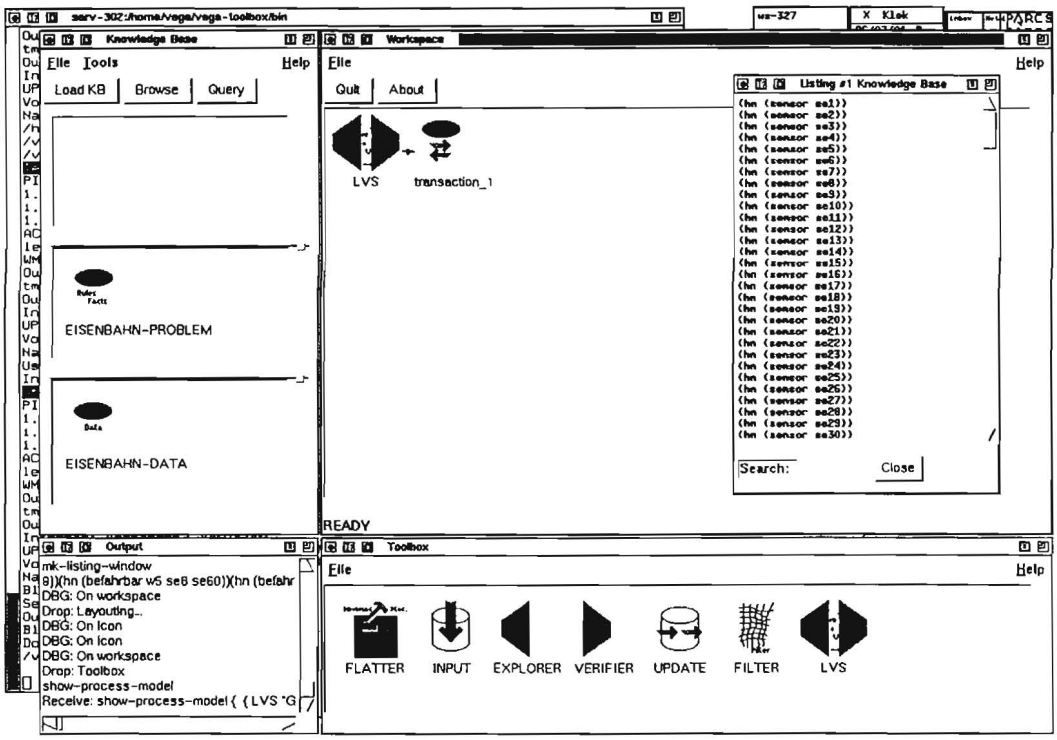

Abbildung 20: Die grafische Oberfläche des VEGA Systems

Zur Zeit wird der Server als Vermittler zwischen dem in Lisp implementierten MCP (Master Control Program), der grafischen Oberfläche in Tcl/Tk (siehe Abbildung 20) und FOIL, einem in C implementierten induktiven Lernverfahren, eingesetzt. Die In-

itialisierung des Systems übernimmt momentan der MCP, der für jeweils einen Wissensbankserver eine Oberfläche erzeugt. Im Hinblick auf die Möglichkeit eine Wissensbank von mehreren Benutzern gleichzeitig bearbeiten zu lassen, ist jedoch geplant, die Initialisierung von der Oberfläche aus vorzunehmen. Dies geschieht dann mit der Option, sich mit einem beliebigen Wissensbankserver zu verbinden oder einen neuen zu erzeugen. Hierzu gehört selbstverständlich auch die Entwicklung eines adäquaten Kohärenzprotokolls, welches die konsistente Sicht jedes Wissensbankservers inklusive aller Oberflächen auf die Wissensbank garantiert. Um dies zu unterstützen ist geplant, den „Ressourcen Server" dahingehend zu erweitern, daß sich angeschlossene Benutzerprozesse einzelnen Prozeßgruppen zuordnen lassen, an die dann mittels Multicast Nachrichten verschickt werden können.

Die Threadbibliothek T ist z.Zt. für die Betriebssysteme SunOS 5.4 (Solaris), SunOS bis Version 5 und Linux ab Version 1.2.13 verfügbar, ebenso wie der Server zur Prozeßkommunikation, der keine weiteren systemabhängigen Teile enthält.

A Protokollspezifikation

In diesem Abschnitt werden die möglichen Requests an den Server inklusive exakter Protokollangabe beschrieben. Dabei wurde das Protokoll jeweils ausgehend von der zu erfüllenden Funktionalität entworfen und nicht nach Gesichtspunkten der Verarbeitbarkeit. Es sollte daher einfach verständlich sein. Anzumerken ist noch, daß alle Informationen in ihrer Zeichenkettenrepräsentation übertragen werden, d.h., daß z.B. eine Prozeßidentifikationsnummer 4500 als Parameter eines Requests als Zeichenkette "4500" übertragen wird.

A.1 NEWPROCESS

Der Request `NEWPROCESS` veranlaßt den Server, einen neuen Prozeß samt zugehörigen „Named Pipes" als Kommunikationsressourcen zu erzeugen. Deren Namen werden im Prozeß–Environment[27] übergeben. Zur eindeutigen Identifikation bekommt dieser neue Prozeß vom Server eine Identifikationsnummer zugeordnet. Desweiteren ist es möglich, später Informationen über den Prozeß alternativ über seinen Aufrufpfad oder über einen Spitznamen (Nickname) zu erhalten, der hier ebenfalls spezifiziert werden muß. Die letzen beiden Felder definieren die an den neuen Prozeß zu übergebenden Argumente und sein Environment, welches um die drei Namen der „Named Pipes" angereichert ist. Die Einträge hierfür heißen `INPIPE`, `OUTPIPE` und `ACKPIPE`. Sollten sie bereits im Environment vorhanden sein, so werden sie entsprechend modifiziert.

Format (Request):

"1"	Rechner	<nl>	Pfad	<nl>	Nickname	<nl>	...
1 Byte(s)	r Byte(s)	1 Byte(s)	p Byte(s)	1 Byte(s)	n Byte(s)	1 Byte(s)	

...	Argumente	<nl>	Umgebung	<nl>
	a Byte(s)	1 Byte(s)	e Byte(s)	1 Byte(s)

Gesamtlänge: $(r + p + n + a + e + 6)$ Bytes

- Das Zeichen '1' ist das Tag, welches diesem Request zugeordnet ist.
- 'Rechner' beinhaltet die Adresse des Rechners, an den der Request gestellt werden soll. Wird der lokale Server angesprochen, so ist dieses Feld leer, ansonsten

[27] Das Environment eines Prozesses unter UNIX ist eine Datenstruktur der Form
`VARIABLE=WERT`,
die vom Erzeugerprozeß ähnlich den Kommandozeilenargumente übergeben werden kann.

kann hier die IP-Adresse, die sog. „dotted number“ oder der DNS[28]–Name des jeweiligen Rechners angegeben werden.

- 'Pfad' beschreibt den absoluten(!) [29] Pfad des Programms.
- 'Nickname' ist eine beliebige eindeutige Zeichenkette, die als Alias für den neuen Prozeß verwendet werden kann.
- 'Argumente' ist die Liste der Argumente, die dem neuen Prozeß zugeführt werden soll. Jedes einzelne Argument wird mit einem Newline (<nl>) abgeschlossen.
- 'Umgebung' ist das Environment für den neuen Prozeß. Alle Einträge werden ebenfalls mit einem Newline (<nl>) abgeschlossen.

Im Schema erkennbar sind Newline-Zeichen (<nl>), die das Ende von Feldern markieren, deren Längen nicht fix sind. Soll eines der Felder leer bleiben (z.B. das der Argumente), so ist lediglich das trennende Newline zu übertragen. Ist das Environment leer, so wird das des Servers übergeben. Beispiel:
`"1serv-302\n/usr/X11/bin/xterm\ntermi\n-e\nvi\n\n\n"`
ist ein C–String, der exakt in dieser Form an den Serverprozeß auf dem Rechner „serv-302“ geschickt werden kann. Aufgerufen wird das Programm „/usr/X11/bin/xterm“ mit den Argumenten „-e“ und „vi“, welches das Environment des Servers erbt und den Spitzname „termi“ zugewiesen bekommt.

Format (Reply an den Requester):

"1"	Status	<nl>	Prozeß–Id	<nl>
1 Byte(s)	stat Byte(s)	1 Byte(s)	pid Byte(s)	1 Byte(s)

Gesamtlänge: $(stat + pid + 3)$ Bytes

- "1" ist das Tag, das die Zugehörigkeit der Antwort zu einem `NEWPROCESS`–Request beschreibt.
- Im zweiten Feld wird der Returnstatus des Requests gemeldet. Der eingelesene String muß als Integer interpretiert werden. Der Wert 0 deutet auf erfolgreichen Empfang und Verarbeitung hin, andere mögliche Werte hängen vom Betriebssystem ab und sind den Anmerkungen der Implementation zu entnehmen.

  Wichtig hierbei ist, daß diese Information lediglich etwas über die erfolgreiche Kreation aussagt. Ob der neue Prozeß zum Zeitpunkt der Rückmeldung noch existiert, oder ob er schon wieder terminiert ist, bleibt unberücksichtigt.

[28] Domain Name Service
[29] Es wird keine Shell zur Interpretation etwaiger Pfadvariablen dazwischengeschaltet.

- "Prozeß–Id" ist eine Nummer zur Identifikation des Prozesses, die man z.B. beim `TRANSMIT`–Request benötigt. Falls der Returnstatus ungleich 0 war, so steht hier eine Klartextfehlermeldung, die ebenfalls vom zugrundeliegenden Betriebssystem abhängt und daher den Anmerkungen zur Implementation zu entnehmen ist.

A.2 TRANSMIT

Dient der Nachrichtenübermittlung zwischen zwei Benutzerprozessen. Als Parameter muß die Prozeß–ID des Empfängers angegeben werden. Die Nachricht selbst kann beliebig lang sein, das Auftreten eines Newline (<nl>) wird als Nachrichtenende interpretiert. Soll jedoch innerhalb einer Nachricht ein <nl>geschickt werden, so muß diesem das Fluchtsymbol "\" vorangestellt werden.
An den Empfänger wird die reine Nachricht bereinigt um evtl. Fluchtsymbole ausgeliefert. Es gibt für ihn keine Möglichkeit in Erfahrung zu bringen, von wem die Nachricht abgesandt wurde (so erreicht man eine höhere Funktionalität, wenn man Prozesse steuern will, die von ihrer Einbindung in das Benutzerprozeßnetzwerk nichts wissen). Aus diesem Grund muß man ggf. das Benutzerprotokoll so anlegen, daß man entsprechende Information bei Bedarf selbst überträgt.

Format (Request an den Server):

"2"	**Rechner**	<nl>	**Prozeß–ID des Empfängers**	<nl>	...
1 Byte(s)	r Byte(s)	1 Byte(s)	pid Byte(s)	1 Byte(s)	

...	**Nachricht an den Empfänger**	<nl>
	n Byte(s)	1 Byte(s)

Gesamtlänge: $(r + pid + n + 4)$ Bytes

- Das Zeichen "2" ist das Tag, welches dem Request `TRANSMIT` zugeordnet ist.
- 'Rechner' beinhaltet die Adresse des Rechners, an den der Request gestellt werden soll. Wird der lokale Server angesprochen, so ist dieses Feld leer, ansonsten kann hier die IP-Adresse, die sog. „dotted number" oder der DNS–Name des jeweiligen Rechners angegeben werden.
- 'Prozeß–ID' beschreibt die Prozeßidentifikationsnummer des zu kontaktierenden Prozesses auf dem angesprochenen Rechner und kann mit dem `GETIDBYPATH`– bzw. `GETIDBYNICKNAME`–Request ermittelt werden.

- Anschließend folgt die Nachricht an den Empfänger. Falls Newline gesendet werden sollen, müssen diese mit einem "\" vorangestellt werden, da sie sonst als Nachrichtenende interpretiert werden.

Im Schema erkennbar sind Newline-Zeichen (<nl>), die das Ende von Feldern markieren, deren Längen nicht fix sind. Beispiel:
`"2serv-302\n5000\nHallo,\\n Du Sack\n"`
ist ein C–String, der exakt in dieser Form an den Server geschickt werden kann. Übertragen wird die Zeichenkette
`"Hallo,<nl>`
`Du Sack"`
an den Empfänger mit der Prozeß–ID 5000.

Format (Reply an den Requester):

"2"	"0"	Status	<nl>
1 Byte(s)	1 Byte(s)	stat Byte(s)	1 Byte(s)

Gesamtlänge: $(stat + 3)$ Bytes

- "2" ist das Tag, das die Zugehörigkeit der Antwort zu einem `TRANSMIT`–Request beschreibt.
- Im zweiten Feld wird der Returnstatus des Requests im Klartext gemeldet. Der String "ok" deutet auf erfolgreiche Übermittlung hin, alle anderen mögliche Zeichenketten hängen vom Betriebssystem ab und sind den Anmerkungen der Implementation zu entnehmen.

A.3 KILLPROCESS

KILLPROCESS veranlaßt den Server, einen anderen, erreichbaren Prozeß zu terminieren, indem diesem das Signal `SIGTERM` geschickt wird. Erreichbar heißt in diesem Zusammenhang, daß der zu terminierende Prozeß auch von diesem oder einem erreichbaren Server gestartet wurde. Andere Prozesse lassen sich mit diesem Request nicht terminieren.

Format (Request an den Server):

"3"	Rechner	<nl>	Prozeß–ID des zu terminierenden Prozesses	<nl>
1 Byte(s)	r Byte(s)	1 Byte(s)	pid Byte(s)	1 Byte(s)

Gesamtlänge: $(r + pid + 3)$ Bytes

- Das Zeichen "3" ist das Tag, welches dem Request `KILLPROCESS` zugeordnet ist.
- 'Rechner' beinhaltet die Adresse des Rechners, an den der Request gestellt werden soll. Wird der lokale Server angesprochen, so ist dieses Feld leer, ansonsten kann hier die IP-Adresse, die sog. „dotted number" oder der DNS–Name des jeweiligen Rechners angegeben werden.
- 'Prozeß–ID' beschreibt die Prozeßidentifikationsnummer des zu kontaktierenden Prozesses auf dem angesprochenen Rechner und kann mit dem `GETIDBYPATH`– bzw. `GETIDBYNICKNAME`–Request ermittelt werden.

Im Schema erkennbar sind Newline-Zeichen (<nl>), die das Ende von Feldern markieren, deren Längen nicht fix sind. Beispiel:
`"3serv-302\n5000\n"`
ist ein C–String, der exakt in dieser Form an den Server geschickt werden kann. Terminiert wird der Prozeß mit der Prozeß–ID 5000 auf dem Rechner mit der Adresse „serv-302".

Format (Reply an den Requester):

"3"	Status	<nl>
1 Byte(s)	stat Byte(s)	1 Byte(s)

Gesamtlänge: $(stat + 2)$ Bytes

- "3" ist das Tag, das die Zugehörigkeit der Antwort zu einem `KILLPROCESS`–Request beschreibt.
- Im zweiten Feld wird der Returnstatus des Requests im Klartext gemeldet. Der String "ok" deutet auf eine erfolgreiche Verarbeitung hin, andere mögliche Zeichenketten hängen vom Betriebssystem ab und sind den Anmerkungen der Implementation zu entnehmen.

A.4 GETIDBYPATH

Mit diesem Request kann die Prozeß–ID anderer Benutzerprozesses ermittelt werden. Als Suchparameter kann der Pfad zum Programm, wie er beim `NEWPROCESS`–Request angegeben wurde, verwendet werden.

Format (Request an den Server):

"4"	**Rechner**	**<nl>**	**Pfad**	**<nl>**
1 Byte(s)	r Byte(s)	1 Byte(s)	p Byte(s)	1 Byte(s)

Gesamtlänge: $(r + p + 3)$ Bytes

- Das Zeichen "4" ist das Tag, welches dem Request `GETIDBYPATH` zugeordnet ist.
- 'Rechner' beinhaltet die Adresse des Rechners, an den der Request gestellt werden soll. Wird der lokale Server angesprochen, so ist dieses Feld leer, ansonsten kann hier die IP-Adresse, die sog. „dotted number" oder der DNS–Name des jeweiligen Rechners angegeben werden.
- Das nächste Feld spezifiziert den Pfad zum Programm, wie er beim `NEWPROCESS`–Request angegeben wurde.

Im Schema erkennbar sind Newline-Zeichen (<nl>), die das Ende von Feldern markieren, deren Längen nicht fix sind. Beispiel:
`"4serv-302\n/usr/X11/bin/xterm\n"`
ist ein C–String, der exakt in dieser Form an den Server geschickt werden kann. Es wird die Prozeß–ID eines Prozesses gesucht, der das Programm „/usr/X11/bin/xterm" auf einem Rechner mit dem DNS–Eintrag „serv-302" ausführt

Format (Reply an den Requester):

"4"	**Prozeß–ID** oder **leer, falls Prozeß nicht gefunden**	**<nl>**
1 Byte(s)	pid Byte(s)	1 Byte(s)

Gesamtlänge: $(pid + 2)$ Bytes

- "4" ist das Tag, das die Zugehörigkeit der Antwort zu einem `GETIDBYPATH`–Request beschreibt.
- Im zweiten Feld wird im Erfolgsfall die Prozeß–ID des nachgefragten Prozesses geliefert, ansonsten ist dieses Feld leer.

A.5 GETIDBYNICKNAME

Mit diesem Request kann die Prozeß–ID eines anderen Benutzerprozesses ermittelt werden. Als Suchparameter kann der Spitzname (Nickname), wie er beim `NEWPROCESS`–Request angegeben wurde, verwendet werden.

Format (Request an den Server):

"5"	**Rechner**	<nl>	**Nickname**	<nl>
1 Byte(s)	r Byte(s)	1 Byte(s)	n Byte(s)	1 Byte(s)

Gesamtlänge: $(r + n + 3)$ Bytes

- Das Zeichen "5" ist das Tag, welches dem Request `GETIDBYNICKNAME` zugeordnet ist.
- 'Rechner' beinhaltet die Adresse des Rechners, an den der Request gestellt werden soll. Wird der lokale Server angesprochen, so ist dieses Feld leer, ansonsten kann hier die IP-Adresse, die sog. „dotted number" oder der DNS–Name des jeweiligen Rechners angegeben werden.
- Das nächste Feld spezifiziert den Spitznamen des Prozesses, der beim `NEWPROCESS`–Request angegeben werden kann.

Im Schema erkennbar sind Newline-Zeichen (<nl>), die das Ende von Feldern markieren, deren Längen nicht fix sind. Beispiel:
`"5serv-302\nstruppi\n"`
ist ein C–String, der exakt in dieser Form an den Server geschickt werden kann. Es wird die Prozeß–ID eines Prozesses gesucht, dessen Spitzname "struppi" ist und auf einem Rechner mit dem DNS–Eintrag „serv-302" läuft.

Format (Reply an den Requester):

"5"	**Prozeß–ID** oder **leer, falls Prozeß nicht gefunden**	<nl>
1 Byte(s)	pid Byte(s)	1 Byte(s)

Gesamtlänge: $(pid + 2)$ Bytes

- "5" ist das Tag, das die Zugehörigkeit der Antwort zu einem `GETIDBYPATH`–Request beschreibt.

- Im zweiten Feld wird im Erfolgsfall die Prozeß–ID des nachgefragten Prozesses geliefert, ansonsten ist dieses Feld leer.

B Schnittstelle zu Lisp

Um den Zugriff auf die Serverfunktionen unter Lisp möglichst einfach zu machen, wurde zusätzlich noch eine Lisp Befehlsbibliothek entwickelt.

Bevor man die Funktionen benutzen kann, muß zunächst die entsprechende Bibliothek hinzugeladen und `init_network` ohne weitere Parameter aufgerufen werden. Eine weitere Voraussetzung ist, daß der entsprechende Lisp Prozeß auch vom Server erzeugt wurde und damit über Kommunikationskanäle verfügt. Alle Parameter der im folgenden beschriebenen Funktionen sind vom Typ *String*.

`defun open-request (host path nickname &key (args NIL))`

`open-request` veranlaßt den Server auf dem Rechner `host` einen neuen Benutzerprozeß mit Programmpfad `path` zu erzeugen und ihm den Spitznamen `nickname` zuzuordnen. Mit dem Key-Parameter `:args` kann zusätzlich eine Liste von Strings übergeben werden, die dem neuen Prozeß als Kommandozeilenargumente zur Verfügung gestellt werden. Zurückgegeben wird exakt der String, der vom Server als Antwort auf einen `NEWPROCESS`-Request an den Requester geliefert wird (siehe Anhang A.1).

`defun transmit-request (host pid data)`

Mittels `transmit-request` können beliebige Daten, übergeben als String in `data` an einen anderen Benutzerprozeß `pid` auf Rechner `host` verschickt werden. Auch hierbei wird im Anschluß an den Request die Antwort des Servers entgegengenommen und als Returnwert des Funktionsaufrufs geliefert.

Da die empfangende Seite keine Voraussagen über die Länge des übermittelten Datenstroms machen kann, wurde im Rahmen der Lisp Schnittstelle die Vereinbarung getroffen, daß der Empfang des Zeichens „#“ das Ende markiert. Sollte dieses Zeichen in `data` vorkommen, so werden entsprechend viele „\“ vorangestellt (der Server interpretiert ebenfalls „\“).

`defun receive ()`

`receive` ist das Pendant zum `transmit-request` und empfängt einen Datenstrom, dessen Ende durch das Zeichen „#“ markiert ist. Dabei werden Fluchtsymbole „\“ innerhalb des Datenstroms entfernt, so daß die Daten exakt so zurückgegeben werden, wie sie beim Sender verschickt wurden. Bei nicht vorhandenen Daten blockiert `receive` den aufrufenden Prozeß bis zu deren Eintreffen.

`defun killprocess-request (host pid)`

`killprocess-request` dient der Terminierung eines anderen Benutzerprozesses im Netz, der durch `host` und `pid` adressiert wird.
Zurückgegeben wird der Antwortstring des Servers (siehe Anhang A.3).

`defun getidbypath-request (host path)`

Um die Prozeßidentifikationsnummer anderer Benutzerprozesses in Netz zu ermitteln, dient `getidbypath-request`. Dabei wird der adressierte Benutzerprozeß durch Angabe der Rechners in `host`, sowie seines Aufrufpfades in `path` analog dessen Erzeugung beim `open-request` spezifiziert.
Zurückgegeben wird jedoch nicht die reine Prozeß-ID, sondern der Antwortstring des Servers in unabgeänderter Form (siehe Anhang A.4).

`defun getidbynickname-request (host nickname)`

`getidbynickname-request` ist analog `getidbypath-request`, nur wird der adressierte Benutzerprozeß nicht über den Aufrufpfad, sondern über seinen Spitznamen `nickname` spezifiziert.
Zum Returnwert siehe A.5.

C Schnittstelle zu Tcl/Tk

Dieser Abschnitt beschreibt, ähnlich Anhang B, eine Befehlsbibliothek für Tcl/Tk zum einfachen Zugriff auf die Serverfunktionen. Diese Bibliothek wurde von Jochen Schäfer[30] erstellt, dem ich für die Bereitstellung dieser Dokumentation danke.

init_pipes

Zweck Diese Funktion initialisiert das TCL-Programm für die Kommunikation mit dem „Ressource Server".

Syntax `init_pipes out-file in-file ack-file`

Es werden bei der Initialisierung die Out- und die In-Pipe, die bei Erzeugung dem Programm zugeordnet wurde, geöffnet. Die Acknowledge-Pipe muß in der jetzigen Implementation nach dem ersten Senden eines Requests geöffnet werden. Den Parametern `out-file`, `in-file` und `ack-file` werden *Namen* von globalen Variablen übergeben, die dann die Dateideskriptoren der geöffneten Pipes enthalten. Außerdem enthält die globale Variable `ACK_NAME` den Dateinamen der Acknowledge-Pipe.

Rückgabewert Diese Funktion gibt den leeren String zurück.

send_open

Zweck Diese Funktion sendet einen `NEWPROCESS`-Request an den „Ressource Server", der dann einen neuen Prozeß erzeugt.

Syntax `send_open file Host Path Nickname Args Env`

file Der Dateideskriptor für die Out-Pipe.

Host Gibt den Rechner an, auf dem das Programm gestartet werden soll.

Path Dies ist der *voll* qualifizierte Dateipfad zu dem Programm, das gestartet werden soll.

Nickname Eindeutiger Name, über den der neue Prozeß angesprochen werden kann.

Args Die Argumentliste, die dem neuen Programm übergeben werden soll.

Env Eine Liste von Environmentvariablen und ihren Werten, die dem neuen Programm übergeben wird. Wird eine leere Liste übergeben, so übergibt der „Ressource Server" dem neuen Programm sein Environment. Wird jedoch ein Environment übergeben, *so beschreibt dieses das gesamte Environment* des neues Programms.

Rückgabewert Diese Funktion gibt den leeren String zurück.

[30] http://www.dfki.uni-kl.de/~jschaefe/

receive_open

Zweck Diese Funktion empfängt die Antwort des „Ressource Servers“ auf einen NEWPROCESS-Request. Die Funktion sollte möglichst bald nach einem NEWPROCESS-Request aufgerufen werden, um das "Verstopfen" der Acknowledge-Pipe zu verhindern.

Syntax `receive_open file`

file Der Dateideskriptor für die Acknowledge-Pipe.

Rückgabewert Diese Funktion gibt in einer Liste den Status des Open-Requests und die PID des gestarteten Prozesses zurück. Ist der NEWPROCESS-Request geglückt, so ist der Status "OK". Ansonsten wird eine Fehlermeldung in Klartext übergeben und der Wert der PID ist undefiniert.

send_transmit

Zweck Diese Funktion sendet einen TRANSMIT-Request an den „Ressource Server“, der dann die gewünschte Botschaft an den Empfänger schickt. Zur Kodierung der Botschaften siehe C.1.

Syntax `send_transmit file Host PID Message`

file Der Dateideskriptor für die Out-Pipe.

Host Gibt den Rechner an, auf dem der Empfänger gestartet wurde.

PID Process ID des Empfängers.

Message Die zu versendende Botschaft.

Rückgabewert Diese Funktion gibt den leeren String zurück.

receive_transmit

Zweck Diese Funktion empfängt die Antwort des „Ressource Servers“ auf einen TRANSMIT-Request. Die Funktion sollte möglichst bald nach dem Request aufgerufen werden, um das "Verstopfen" der Acknowledge-Pipe zu verhindern.

Syntax `receive_open file`

file Der Dateideskriptor für die Acknowledge-Pipe.

Rückgabewert Diese Funktion gibt den Status des NEWPROCESS-Requests zurück. Ist der Request geglückt, so ist der Status "OK". Ansonsten wird eine Fehlermeldung im Klartext übergeben.

send_close

Zweck Diese Funktion sendet einen KILLPROCESS-Request an den „Ressource Server", der dann den entsprechenden Prozeß beendet.

Syntax `send_close file Host PID`

file Der Dateideskriptor für die Out-Pipe.

Host Gibt den Rechner an, auf dem das Programm gestartet wurde.

PID Process ID des zu schließenden Prozesses.

Rückgabewert Diese Funktion gibt den leeren String zurück.

receive_kill

Zweck Diese Funktion empfängt die Antwort des „Ressource Servers" auf einen KILLPROCESS-Request. Die Funktion sollte möglichst bald nach dem Request aufgerufen werden, um das "Verstopfen" der Acknowledge-Pipe zu verhindern.

Syntax `receive_kill file`

file Der Dateideskriptor für die Acknowledge-Pipe.

Rückgabewert Diese Funktion gibt den Status des KILLPROCESS-Requests zurück. Ist der Request geglückt, so ist der Status "OK". Ansonsten wird eine Fehlermeldung in Klartext übergeben.

send_pid_by_path

Zweck Diese Funktion sendet einen GETIDBYPATH-Request an den „Ressource Server", der dann die PID des durch diesen Dateipfad angegebenen Programm zurückgibt.

Syntax `send_pid_by_path file Host Path`

file Der Dateideskriptor für die Out-Pipe.

Host Gibt den Rechner an, auf dem das Programm gestartet wurde.

Path Gibt den voll qualifizierten Dateipfad des Programmes an, für das man die PID ermitteln möchte.

Rückgabewert Diese Funktion gibt den leeren String zurück.

receive_pid_by_path

Zweck Diese Funktion empfängt die Antwort des „Ressource Server" auf einen GETIDBYPATH-Request. Die Funktion sollte möglichst bald nach dem Request aufgerufen werden, um das "Verstopfen" der Acknowledge-Pipe zu verhindern.

Syntax `receive_pid_by_path file`

file Der Dateideskriptor für die Acknowledge-Pipe.

Rückgabewert Diese Funktion gibt die PID des gewünschten Prozesses zurück. Die PID ist 0, wenn das Programm dem „Ressource Server“ nicht bekannt ist.

send_pid_by_nick

Zweck Diese Funktion sendet einen `GETIDBYNICKNAME`-Request an den „Ressource Server“, der dann die PID des durch diesen Nickname spezifizierten Prozesses zurückgibt.

Syntax `send_pid_by_nick file Host Nickname`

file Der Dateideskriptor für die Out-Pipe.

Host Gibt den Rechner an, auf dem das Programm gestartet wurde.

Path Gibt den Nickname des Programmes an, für das man die PID ermitteln möchte.

Rückgabewert Diese Funktion gibt den leeren String zurück.

receive_pid_by_nick

Zweck Diese Funktion empfängt die Antwort des „Ressource Servers“ auf einen `GETIDBYNICKNAME`-Request. Die Funktion sollte möglichst bald nach dem Request aufgerufen werden, um das "Verstopfen" der Acknowledge-Pipe zu verhindern.

Syntax `receive_pid_by_nick file`

file Der Dateideskriptor für die Acknowledge-Pipe.

Rückgabewert Diese Funktion gibt die PID des gewünschten Prozesses zurück. Die PID ist 0, wenn der Prozeß dem „Ressource Server“ nicht bekannt ist.

receive_msg

Zweck Diese Funktion empfängt eine Botschaft, die von einem anderen Prozeß gesendet wurde.

Syntax `receive_msg file`

file Der Dateideskriptor für die Acknowledge-Pipe.

Rückgabewert Diese Funktion gibt die dekodierte (siehe C.1) Botschaft zurück.

C.1 Kodierung der Botschaften

Wie in Anhang B bereits geschildert, wird ein `TRANSMIT`-Request durch \n begrenzt. Dadurch sind Zeilenumbrüche durch das Escapezeichen \ zu kodieren, also wird aus \n \\n. Dies wird dann durch den „Ressource Server“ entsprechend dekodiert, so daß der Empfänger einen Datenstrom aus vielen Zeilen erhält. Da nun also der empfangene Datenstrom erst beendet wird, wenn die Verbindung abgebrochen wird, mußte ein internes Protokoll zwischen MCP und GUI definiert werden. Dafür wurde vereinbart, daß ein Befehl (von GUI oder MCP) durch ein # zu terminieren ist. Im Text enthaltene # und auch \ müssen nun vom Sender entsprechend, d.h durch Voranstellen von \ kodiert und vom Empfänger wieder dekodiert werden.

Literatur

[Abecker *et al.*, 1995] Andreas Abecker, Harold Boley, Knut Hinkelmann, Holger Wache, and Franz Schmalhofer. An Environment for Exploring and Validating Declarative Knowledge. DFKI Technical Memo TM-95-03, DFKI GmbH, November 1995. Also in: Proc. Workshop on Logic Programming Environments at ILPS'95, Portland, Oregon, Dec. 1995.

[Abecker, 1993] Andreas Abecker. Implementierung graphischer Benutzungsoberflächen mit Tcl/Tk und Common Lisp. DFKI Dokument, 12 1993.

[Boley *et al.*, 1992] Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer, and Michael M. Richter. *VEGA - Knowledge Validation and Exploration by Global Analysis.* Project Proposal, DFKI Kaiserslautern, October 1992.

[Brown, 1994] Chris Brown. *UNIX Distributed Programming.* Prentice Hall, 1994.

[Comer und Stevens, 1993] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP Client-Server Programming and Applications BSD Socket Version*, volume 3. Prentice Hall, 1993.

[Coulouris *et al.*, 1994] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems - Concepts and Design.* Addison-Wesley Publishing Company, 2 edition, 1994.

[Date, 1990] C.J. Date. *An Introduction to Database Systems.* Addison-Wesley Publishing Company, 5 edition, 1990.

[Dijkstra, 1968] E.W. Dijkstra. Cooperating Sequential Processes. *Programming Languages*, 1968.

[Gallmeister, 1995] Bill O. Gallmeister. *POSIX.4 Programming for The Real World.* O'Reilly & Associates, Inc., 1995.

[Hinkelmann und Kühn, 1994] K. Hinkelmann and O. Kühn. Vorschlag zur Entwicklung des KES Wissensevolutions-Systems in VEGA. VEGA Diskussionspapier, 1994.

[Hinkelmann und Kühn, 1995] K. Hinkelmann and O. Kühn. Revising and Updating a Corporate Memory. In *EUROVAV-95: Proc. of the European Symposium on Validation and Verification of Knowledged-based Systems*, 1995.

[Nehmer, 1993] Jürgen Nehmer. Betriebsorganisation von Rechnersystemen - Eine Einführung auf der Basis des Client/Server-Modells -. Vorlesungsskript, 10 1993.

[Sun Microsystems Inc., 1994] Sun Microsystems Inc. pthreads and Solaris threads A comparison of two user level APIs. Technical report, Sun Microsystems Inc., May 1994. Early Access Version–pthreads Based on POSIX 1003.4a/D8.

[Tanenbaum, 1992a] Andrew S. Tanenbaum. *Computer-Netzwerke*, volume 2. Wolfram's Fachverlag, 1992.

[Tanenbaum, 1992b] Andrew S. Tanenbaum. *Modern Operating Systems.* Englewood Cliffs NJ: Prentice Hall, 1992.

[Ullman, 1988] J.D. Ullman. *Principles of Database and Knowledge-Base Systems.* Pitman Publishing Ltd., 1988.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

-Bibliothek, Information
und Dokumentation (BID)-
PF 2080
67608 Kaiserslautern
FRG

Telefon (0631) 205-3506
Telefax (0631) 205-3210
e-mail
dfkibib@dfki.uni-kl.de
WWW
http://www.dfki.uni-sb.de/dfkibib

Veröffentlichungen des DFKI

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse oder (so sie als per ftp erhaeltlich angemerkt sind) per anonymous ftp von ftp.dfki.uni-kl.de (131.246.241.100) im Verzeichnis pub/Publications bezogen werden. Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far are obtainable from the above address or (if they are marked as obtainable by ftp) by anonymous ftp from ftp.dfki.uni-kl.de (131.246.241.100) in the directory pub/Publications.
The reports are distributed free of charge except where otherwise noted.

DFKI Research Reports

1996

RR-96-05
Stephan Busemann
Best-First Surface Realization
11 pages

RR-96-03
Günter Neumann
Interleaving
Natural Language Parsing and Generation
Through Uniform Processing
51 pages

RR-96-02
E.André, J. Müller , T.Rist:
PPP-Persona: Ein objektorientierter Multimedia-Präsentationsagent
14 Seiten

1995

RR-95-20
Hans-Ulrich Krieger
Typed Feature Structures, Definite Equivalences, Greatest Model Semantics, and Nonmonotonicity
27 pages

RR-95-19
Abdel Kader Diagne, Walter Kasper, Hans-Ulrich Krieger
Distributed Parsing With HPSG Grammar
20 pages

RR-95-18
Hans-Ulrich Krieger, Ulrich Schäfer
Efficient Parameterizable Type Expansion for Typed Feature Formalisms
19 pages

RR-95-17
Hans-Ulrich Krieger
Classification and Representation of Types in TDL
17 pages

RR-95-16
Martin Müller, Tobias Van Roy
Title not set
0 pages

Note: The author(s) were unable to deliver this document for printing before the end of the year. It will be printed next year.

RR-95-15
Joachim Niehren, Tobias Van Roy
Title not set
0 pages

Note: The author(s) were unable to deliver this document for printing before the end of the year. It will be printed next year.

RR-95-14
Joachim Niehren
Functional Computation as Concurrent Computation
50 pages

RR-95-13
Werner Stephan, Susanne Biundo
Deduction-based Refinement Planning
14 pages

RR-95-12
Walter Hower, Winfried H. Graf
Research in Constraint-Based Layout, Visualization, CAD, and Related Topics: A Bibliographical Survey
33 pages

RR-95-11
Anne Kilger, Wolgang Finkler
Incremental Generation for Real-Time Applications
47 pages

RR-95-10
Gert Smolka
The Oz Programming Model
23 pages

RR-95-09
M. Buchheit, F. M. Donini, W. Nutt, A. Schaerf
A Refined Architecture for Terminological Systems: Terminology = Schema + Views
71 pages

RR-95-08
Michael Mehl, Ralf Scheidhauer, Christian Schulte
An Abstract Machine for Oz
23 pages

RR-95-07
Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, Werner Nutt
The Complexity of Concept Languages
57 pages

RR-95-06
Bernd Kiefer, Thomas Fettig
FEGRAMED
An interactive Graphics Editor for Feature Structures
37 pages

RR-95-05
Rolf Backofen, James Rogers, K. Vijay-Shanker
A First-Order Axiomatization of the Theory of Finite Trees
35 pages

RR-95-04
M. Buchheit, H.-J. Bürckert, B. Hollunder, A. Laux, W. Nutt,
M. Wójcik
Task Acquisition with a Description Logic Reasoner
17 pages

RR-95-03
Stephan Baumann, Michael Malburg, Hans-Guenther Hein, Rainer Hoch,
Thomas Kieninger, Norbert Kuhn
Document Analysis at DFKI
Part 2: Information Extraction
40 pages

RR-95-02
Majdi Ben Hadj Ali, Frank Fein, Frank Hoenes, Thorsten Jaeger,
Achim Weigel
Document Analysis at DFKI
Part 1: Image Analysis and Text Recognition
69 pages

RR-95-01
Klaus Fischer, Jörg P. Müller, Markus Pischel
Cooperative Transportation Scheduling
an application Domain for DAI
31 pages

1994

RR-94-39
Hans-Ulrich Krieger
Typed Feature Formalisms as a Common Basis for Linguistic Specification.
21 pages

RR-94-38
Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne,
Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger,
Klaus Netter, Günter Neumann, Stephan Oepen, Stephen P. Spackman.
DISCO–An HPSG-based NLP System and its Application for Appointment Scheduling.
13 pages

RR-94-37
Hans-Ulrich Krieger, Ulrich Schäfer
TDL - A Type Description Language for HPSG, Part 1: Overview.
54 pages

RR-94-36
Manfred Meyer
Issues in Concurrent Knowledge Engineering. Knowledge Base and Knowledge Share Evolution.
17 pages

RR-94-35
Rolf Backofen
A Complete Axiomatization of a Theory with Feature and Arity Constraints
49 pages

RR-94-34
Stephan Busemann, Stephan Oepen, Elizabeth A. Hinkelman,
Günter Neumann, Hans Uszkoreit
COSMA - Multi-Participant NL Interaction for Appointment Scheduling
80 pages

RR-94-33
Franz Baader, Armin Laux
Terminological Logics with Modal Operators
29 pages

RR-94-31
Otto Kühn, Volker Becker, Georg Lohse, Philipp Neumann
Integrated Knowledge Utilization and Evolution for the Conservation of Corporate Know-How
17 pages

RR-94-23
Gert Smolka
The Definition of Kernel Oz
53 pages

RR-94-20
Christian Schulte, Gert Smolka, Jörg Würtz
Encapsulated Search and Constraint Programming in Oz
21 pages

RR-94-19
Rainer Hoch
Using IR Techniques for Text Classification in Document Analysis
16 pages

RR-94-18
Rolf Backofen, Ralf Treinen
How to Win a Game with Features
18 pages

RR-94-17
Georg Struth
Philosophical Logics—A Survey and a Bibliography
58 pages

RR-94-16
Gert Smolka
A Foundation for Higher-order Concurrent Constraint Programming
26 pages

RR-94-15
Winfried H. Graf, Stefan Neurohr
Using Graphical Style and Visibility Constraints for a Meaningful Layout in Visual Programming Interfaces
20 pages

RR-94-14
Harold Boley, Ulrich Buhrmann, Christof Kremer
Towards a Sharable Knowledge Base on Recyclable Plastics
14 pages

RR-94-13
Jana Koehler
Planning from Second Principles—A Logic-based Approach
49 pages

RR-94-12
Hubert Comon, Ralf Treinen
Ordering Constraints on Trees
34 pages

RR-94-11
Knut Hinkelmann
A Consequence Finding Approach for Feature Recognition in CAPP
18 pages

RR-94-10
Knut Hinkelmann, Helge Hintze
Computing Cost Estimates for Proof Strategies
22 pages

RR-94-08
Otto Kühn, Björn Höfling
Conserving Corporate Knowledge for Crankshaft Design
17 pages

RR-94-07
Harold Boley
Finite Domains and Exclusions as First-Class Citizens
25 pages

RR-94-06
Dietmar Dengler
An Adaptive Deductive Planning System
17 pages

RR-94-05
Franz Schmalhofer, J. Stuart Aitken, Lyle E. Bourne jr.
Beyond the Knowledge Level: Descriptions of Rational Behavior for Sharing and Reuse
81 pages

RR-94-03
Gert Smolka
A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards
34 pages

RR-94-02
Elisabeth André, Thomas Rist
Von Textgeneratoren zu Intellimedia-Präsentationssystemen
22 Seiten

RR-94-01
Elisabeth André, Thomas Rist
Multimedia Presentations: The Support of Passive and Active Viewing
15 pages

DFKI Technical Memos

1996

TM-96-01
Gerd Kamp, Holger Wache
CTL — a description Logic with expressive concrete domains
19 pages

1995

TM-95-04
Klaus Schmid
Creative Problem Solving
and
Automated Discovery
— An Analysis of Psychological and AI Research —
152 pages

TM-95-03
Andreas Abecker, Harold Boley, Knut Hinkelmann, Holger Wache,
Franz Schmalhofer
An Environment for Exploring and Validating Declarative Knowledge
11 pages

TM-95-02
Michael Sintek
FLIP: Functional-plus-Logic Programming
on an Integrated Platform
106 pages

TM-95-01
Martin Buchheit, Rüdiger Klein, Werner Nutt
Constructive Problem Solving: A Model Construction Approach towards Configuration
34 pages

1994

TM-94-04
Cornelia Fischer
PAntUDE - An Anti-Unification Algorithm for Expressing Refined Generalizations
22 pages

TM-94-03
Victoria Hall
Uncertainty-Valued Horn Clauses
31 pages

TM-94-02
Rainer Bleisinger, Berthold Kröll
Representation of Non-Convex Time Intervals and Propagation of Non-Convex Relations
11 pages

TM-94-01
Rainer Bleisinger, Klaus-Peter Gores
Text Skimming as a Part in Paper Document Understanding
14 pages

DFKI Documents

1996

D-96-05
Martin Schaaf
Ein Framework zur Erstellung verteilter Anwendungen
94 pages

D-96-03
Winfried Tautges
Der DESIGN-ANALYZER - Decision Support im Designprozess
75 Seiten

1995

D-95-12
F. Baader, M. Buchheit, M. A. Jeusfeld, W. Nutt (Eds.)
Working Notes of the KI'95 Workshop:
KRDB-95 - Reasoning about Structured Objects: Knowledge Representation Meets Databases
61 pages

D-95-11
Stephan Busemann, Iris Merget
Eine Untersuchung kommerzieller Terminverwaltungssoftware im Hinblick auf die Kopplung mit natürlichsprachlichen Systemen
32 Seiten

D-95-10
Volker Ehresmann
Integration ressourcen-orientierter Techniken in das wissensbasierte Konfigurierungssystem TOOCON
108 Seiten

D-95-09
Antonio Krüger
PROXIMA: Ein System zur Generierung graphischer Abstraktionen
120 Seiten

D-95-08
Technical Staff
DFKI Jahresbericht 1994
63 Seiten

Note: This document is no longer available in printed form.

D-95-07
Ottmar Lutzy
Morphic - Plus
Ein morphologisches Analyseprogramm für die deutsche Flexionsmorphologie und Komposita-Analyse
74 pages

D-95-06
Markus Steffens, Ansgar Bernardi
Integriertes Produktmodell für Behälter aus Faserverbundwerkstoffen
48 Seiten

D-95-05
Georg Schneider
Eine Werkbank zur Erzeugung von 3D-Illustrationen
157 Seiten

D-95-04
Victoria Hall
Integration von Sorten als ausgezeichnete taxonomische Prädikate in eine relational-funktionale Sprache
56 Seiten

D-95-03
Christoph Endres, Lars Klein, Markus Meyer
Implementierung und Erweiterung der Sprache *ALCP*
110 Seiten

D-95-02
Andreas Butz
BETTY
Ein System zur Planung und Generierung informativer Animationssequenzen
95 Seiten

D-95-01
Susanne Biundo, Wolfgang Tank (Hrsg.)
PuK-95, Beiträge zum 9. Workshop „Planen und Konfigurieren", Februar 1995
169 Seiten

Note: This document is available for a nominal charge of 25 DM (or 15 US-$).

1994

D-94-15
Stephan Oepen
German Nominal Syntax in HPSG
— On Syntactic Categories and Syntagmatic Relations —
80 pages

D-94-14
Hans-Ulrich Krieger, Ulrich Schäfer
TDL - A Type Description Language for HPSG, Part 2: User Guide.
72 pages

D-94-12
Arthur Sehn, Serge Autexier (Hrsg.)
Proceedings des Studentenprogramms der 18. Deutschen Jahrestagung für Künstliche Intelligenz KI-94
69 Seiten

D-94-11
F. Baader, M. Buchheit, M. A. Jeusfeld, W. Nutt (Eds.)
Working Notes of the KI'94 Workshop: KRDB'94 - Reasoning about Structured Objects: Knowledge Representation Meets Databases
65 pages

Note: This document is no longer available in printed form.

D-94-10
F. Baader, M. Lenzerini, W. Nutt, P. F. Patel-Schneider (Eds.)
Working Notes of the 1994 International Workshop on Description Logics
118 pages

Note: This document is available for a nominal charge of 25 DM (or 15 US-$).

D-94-09
Technical Staff
DFKI Wissenschaftlich-Technischer Jahresbericht 1993
145 Seiten

D-94-08
Harald Feibel
IGLOO 1.0 - Eine grafikunterstützte Beweisentwicklungsumgebung
58 Seiten

D-94-07
Claudia Wenzel, Rainer Hoch
Eine Übersicht über Information Retrieval (IR) und NLP-Verfahren zur Klassifikation von Texten
25 Seiten

D-94-06
Ulrich Buhrmann
Erstellung einer deklarativen Wissensbasis über recyclingrelevante Materialien
117 Seiten

D-94-04
Franz Schmalhofer, Ludger van Elst
Entwicklung von Expertensystemen: Prototypen, Tiefenmodellierung und kooperative Wissensevolution
22 Seiten

D-94-03
Franz Schmalhofer
Maschinelles Lernen: Eine kognitionswissenschaftliche Betrachtung
54 Seiten

Note: This document is no longer available in printed form.

D-94-02
Markus Steffens
Wissenserhebung und Analyse zum Entwicklungsprozeß eines Druckbehälters aus Faserverbundstoff
90 pages

D-94-01
Josua Boon (Ed.)
DFKI-Publications: The First Four Years 1990 - 1993
75 pages

Ein Rahmensystem zur Erstellung verteilter Anwendungen

Martin Schaaf

D-96-05
Document