Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Thomas Bretscher

Topsy — Teachable Operating System: Dynamischer Modul-Lader

Studienarbeit SA-99.13
Wintersemester 1998/99

Betreuer:
George Fankhauser

Verantwortlicher:
Prof. Dr. Bernhard Plattner

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Topsy — Teachable Operating System: Dynamic Module Loader

Abstract

1. Introduction

Topsy is a small operating system which has been designed for teaching purposes — Topsy stands for **T**eachable **OP**erating **SY**stem — by TIK (Institut für Technische Informatik und Kommunikationsnetze — Computer Engineering and Networks Laboratory) at ETH Zürich (Eidgenössische Technische Hochschule — Swiss Federal Institute of Technology). Topsy constitutes the framework for the practical exercises related to the course Computer Engineering II, which deals with the basic concepts of operating systems.

2. Aims and Goals

Although the traditional Topsy has a modular structure, its parts are statically linked together. Each change of a program requires a complete start-up of the entire system.

A new version should allow to load and link some of the modules dynamically, so one should be able to load additional modules or even to replace existing ones in the running system. An integrated linker has to connect new modules to the already loaded ones. Some sort of module management must be implemented.

Goal of this work is to link user modules dynamically, and the possibility to use dynamic kernel modules (for example device drivers) should be prepared to a certain point.

3. Results

A concept of a simple module manager was developed, usable for both kernel and user modules. This module manager is nearly independent from the thread manager. For commands to load and unload user modules, there is a channel drawn from the user interface to the module manager; but when and how decisions to load or unload kernel modules will be taken, is completely left to following designers working on dynamic kernel module loading.

The software to manage user modules is implemented and integrated in the Topsy kernel, and can be run as a “demo program”, but it is not yet stable and reliable as a kernel extension should be. The user modules present at start-up are still linked statically. The new module’s program code is transferred through a provisional interface.

4. Further Work

Before bringing the dynamic loader and linker into action, a general revision and a systematic test is absolutely necessary, because some lacking error handling mechanisms can lead to kernel crashes.

A module transfer implementation via serial interface is needed for the hardware used in the practical exercises.

After the revision, one can start to implement dynamic kernel modules; most parts of the kernel module management will be done by a slightly extended version of the actual module manager.

Zusammenfassung

1. Einführung

Topsy (**T**eachable **OP**erating **SY**stem) ist ein kleines Betriebsystem, das für Unterrichtszwecke am TIK, ETH Zürich, entwickelt wurde. Topsy ist die Plattform der praktischen Übungen zur Vorlesung Technische Informatik II, die grundlegende Konzepte von Betriebssystemen behandelt.

2. Ziele

Das bisher verwendete Topsy besteht zwar aus Modulen, diese sind aber statisch miteinander gelinkt. Jede Änderung an Programmen erfordert den Neustart des ganzen Systems.

Neu soll ein Teil der Module dynamisch gelinkt werden, sodass sie bei laufendem System hinzugefügt und auch ausgetauscht werden können. Dazu müssen die Module geladen und durch einen neuen, in Topsy integrierten Linker mit den vorhandenen Teilen verbunden werden. Über die geladenen Module und deren Verknüpfungen muss laufend Buch geführt werden.

Ziel dieser Arbeit: User-Module können dynamisch gelinkt werden, für dynamische Kernelmodule (z. B. Gerätetreiber) bestehen gewisse Vorbereitungen.

3. Resultate

Ein einfaches Modulverwaltungs-Konzept, das für Kernel- und Usermodule brauchbar ist, wurde entwickelt, wobei die Verwaltung der Module weitgehend unabhängig von der Verwaltung der Threads ist. Ein Weg der Lade- und Entladebefehle für Usermodule vom Benutzer zur Modulverwaltung ist vorgezeichnet; völlig offen ist hingegen, wann und wie über das Laden von Kernelmodulen entschieden wird.

Die Software zum Laden, Entladen und Verwalten der Usermodule ist erstellt, in den Topsy-Kernel integriert und als “Demo-Version” brauchbar, allerdings nicht ausgereift. Die beim Start vorhandenen Usermodule sind immer noch statisch gelinkt. Der Transfer des Programmcodes der zu ladenden Module erfolgt über eine provisorische Schnittstelle.

4. Weitere Arbeiten

Falls der dynamische Lader/Linker zum Einsatz kommen soll, ist sicher eine Überarbeitung und ein systematisches Austesten der Software notwendig, da noch verschiedene Mechanismen zur Fehlerbehandlung unvollständig und somit Kernel-Abstürze nicht auszuschliessen sind.

Für den Transfer des Programmcodes der zu ladenden User-Module wird noch eine Implementation benötigt, die auf der in der Übung zur Technischen Informatik verwendeten Hardware lauffähig ist.

Nach erfolgter Überarbeitung könnte die Implementation von dynamischen Kernelmodulen in Angriff genommen werden. Die Modulverwaltung sollte dabei mitbenutzbar sein. Wer oder was den Anstoss zum Laden eines neuen Moduls gibt, woher der Programmcode kommt und wie seine Funktionen aufgerufen werden, ist dort aber noch offen.

Inhaltsverzeichnis

Kapitel 1: Einleitung

1.1 Topsy

Topsy ist ein kleines Betriebsystem, das für Unterrichtszwecke am TIK, ETH Zürich, entwickelt wurde. Topsy steht für **T**eachable **OP**erating **SY**stem, was andeutet, dass es für die Studenten verständlich sein soll. Topsy wurde hauptsächlich auf Einfachheit und Klarheit, erst in zweiter Linie auf Effizienz ausgerichtet. Die Portabilität wird durch eine saubere Trennung in hardware-abhängige und -unabhängige Module erreicht. Sehr wichtig ist seine Fähigkeit, mehrere Threads parallel abzuarbeiten, da in der Vorlesung Technische Informatik II diese Problematik am Beispiel Topsy behandelt wird.

Topsy ist — bis auf einige hardwarenahe Teile — in C programmiert. Als Ausgangsbasis für diese Arbeit dient die Version 1.1, deren Source vollständig verfügbar ist (“Topsy home page” [2]). Dokumentation: Topsy-Handbuch [1].

1.2 Ziel der Arbeit

Im bisherigen Topsy müssen jeweils alle Programme kompiliert, Kernel- und Userteil je für sich gelinkt, und dann in einer gemeinsamen Datei geladen werden. Jede Änderung an Programmen erfordert einen Neustart des ganzen Systems. Der Gedanke, den Userteil austauschen zu können, währenddem der Kernel weiterarbeitet, schwebt manchem Studenten in den praktischen Übungen zu Technische Informatik II vor. Die Weiterentwicklung dieser Idee führt zu folgender Aufgabenstellung:

- Zusätzliche oder neue Versionen von bestehenden Userprogramm-Modulen sollen in Topsy geladen und ausgeführt werden können, ohne Topsy neu aufzustarten.
- Vorbereitung für spätere dynamische Konfiguration (ohne neue Kompilierung) des Kernels, indem dann aus einer Auswahl nur die gewünschten/benötigten Treiber und Kernelmodule geladen werden sollen.
- Im Kernel- und im Userbereich sollen soweit möglich dieselben Mechanismen verwendet werden.
- Es soll eine klare Trennung zwischen Lader/Linker und dem Transfer der Programmdaten von “aussen” ins System erreicht werden.

Die offizielle Aufgabenstellung befindet sich im Anhang A.

Kapitel 2: Konzepte

Dynamische Programmmodule sind Programmteile, die — im Gegensatz zu statischen Programmmodulen — bei laufendem System geladen und entladen werden können. Ein System, das ausser dem innersten Kern vollständig aus dynamischen Modulen besteht, ist zum Beispiel Oberon [5]. Moderne Systeme verwenden dynamische Bibliotheksmodule (dynamic libraries), die von verschiedenen Anwendungsprogrammen genutzt werden, während in traditionelleren Systemen jede Applikation ihre Bestandteile selber laden muss und nach dem Beenden sofort wieder freigibt. Häufig nur statisch aufgebaut sind eingebettete Systeme.

2.1 Dynamische Programmmodule in Topsy

Im bisherigen — rein statischen — Topsy ist es möglich, beliebig viele Threads auf jedem Programmcodeabschnitt zu starten. Auch verschiedenartige Threads des gleichen Adressraumes können sich den Code teilen. Dies soll mit den dynamischen Modulen so bleiben, deshalb werden die einzelnen Module nicht bestimmten Threads oder Thread-Familien zugeordnet, sondern nur dem Kernelspace (Kernel-Adressraum) oder dem Userspace. Das Konzept orientiert sich am Oberon-System, wobei dort aber kein Kernelspace vorhanden ist und — zumindest im Originalsystem — zu jeder Zeit nur ein Thread aktiv ist.

- Jedes Modul ist durch einen Namen identifizierbar und ist nur einmal vorhanden (zu beachten bei den globalen Variablen!)
- Es gibt keinen formalen Unterschied zwischen Bibliotheksmodulen und Anwendungsmodulen.
- Die Modulverwaltung ist (fast) unabhängig von der Threadverwaltung:
  - Jeder Thread kann beliebig viele durch Importbeziehungen oder durch Prozedurvariabeln (function pointers) verknüpfte Module benützen
  - Jedes Modul kann von mehreren gleich- oder verschiedenartigen Threads gleichzeitig benützt werden

Das eingeklammerte Wort “(fast)” bezieht sich auf eine eventuelle Kontrolle vor dem Freigeben eines Moduls, ob es noch von einem Thread benutzt wird (siehe 2.3.2 “Prüfung auf laufende Threads” auf Seite 11).

Die dynamischen Module werden vorläufig für Userprogramme verwendet, später ist auch der Einsatz für Teile des Kernels denkbar, z. B. Treiber, die nur bei Bedarf geladen werden.

2.2 Laden und Linken eines Moduls

2.2.1 Laden (Transfer)

Der kompilierte Programmcode eines neuen Modules muss in Topsy transferiert werden. Da Topsy zur Zeit kein Dateisystem hat, wird der Code durch das Hostsystem bereitgestellt und — falls Topsy auf einer eigenen Maschine läuft — über die serielle Schnittstelle oder — falls Topsy auf einer simulierten Maschine läuft — über eine Software-Schnittstelle geleitet (siehe Anhang B “Beschreibung der Entwicklungsumgebung”). Topsy empfängt das Programmmodul und speichert es in seinem Arbeitsspeicher.

Damit das neue Modul von den bestehenden Modulen aufrufbar ist und vorhandene (Bibliotheks-)Module benutzen kann, muss es gelinkt werden. Zudem müssen auch Sprung- und Datenspeicheradressen im Modul selber angepasst werden (Relokation).

2.2.2 Importbeziehungen

Der Linker stellt die Verbindung zwischen dem neuen Modul und vorhandenen (Bibliotheks-)Modulen her. Um die Verknüpfungen zu ordnen, verwendet er dabei mit Vorteil eine Liste mit allen importierten Modulen; diese Liste kann er entweder selber erzeugen, indem er alle Module nach den gewünschten Symbolen durchsucht, oder der Programmierer muss sie in seinen Source-Code explizit oder implizit integrieren. Die erste Möglichkeit wird nicht weiterverfolgt, um Schwierigkeiten mit gleich benannten globalen Symbolen in verschiedenen Modulen nicht unnötig zu verschärfen. Auf diese Weise kann man Funktionen und Variablen importieren, nicht aber Datentypen: diese werden in C nur textuell über die Header-Dateien (“`.h`”) eingebunden.

Wie soll sich nun der Lader/Linker verhalten, wenn ein zu importierendes Modul nicht vorhanden ist? Eine Auswahl von untersuchten Lösungen:

- **1.** Lader rekursiv (wie Oberon): nicht geladene, vom neuen Modul importierte Module werden auch geladen; falls nicht gelungen: Abbruch.
  - **1.1** Zyklischer Import wird abgefangen.
  - **1.2** Zyklischer Import führt zu unendlicher Rekursion.
- **2.** Lader lädt nur befohlene Module (Modulgruppe).
  Falls ein Modul nicht ladbar oder ein Symbol nicht sofort linkbar ist:
  - **2.1** Alle Module der Gruppe verwerfen.
  - **2.2** Nur fehlerhaftes Modul verwerfen.
  - **2.3** Nicht gelinkte Symbole markieren für Link mit einem folgenden Modul der selben Modulgruppe (zyklischer Import denkbar!), falls unmöglich:
    - **2.3.1** Alle Module der Gruppe verwerfen.
    - **2.3.2** Fehlerhaftes (und evtl. folgende) Module verwerfen.
  - **2.4** Nicht gelinkte Symbole als momentan ungültig markieren und aufbewahren für Verlinkung mit späteren Modulen (zyklischer Import denkbar!).
- **3.** Lader lädt genau ein Modul aufs Mal (oder behandelt eine Gruppe wie mehrere aufeinanderfolgende Einzelmodule).
  Falls ein Symbol nicht sofort verlinkbar ist:
  - **3.1** Modul verwerfen.
  - **3.2** Wie 2.4 (zyklischer Import denkbar!).

Variante 1. setzt voraus, dass der Lader ein Modul mit bestimmtem Namen verlangen kann. Da Topsy kein eigenes Dateisystem hat, ist das nicht so einfach realisierbar. Eine minimale Implementation des Laders würde einfach das nächste angebotene Modul laden, die Auswahl der Datei müsste der Benutzer durch einen Befehl ans Host-System vornehmen.

Da Kreis-Importe strukturell unschön sind und beim Freigeben von Modulen Probleme verursachen, kann man diese mit gutem Gewissen verbieten. Dann bringt die Markierung von Symbolen zwecks späterer Verlinkung kaum Nutzen, aber viel Verwaltungsaufwand (Varianten 2.4, 3.2, etwas weniger auch 2.3). Wenn diese Varianten ausgeschlossen werden, muss der Benutzer die Module allerdings in der korrekten Reihenfolge laden.

Variante 2.1 (bei Fehler alle Module der Gruppe verwerfen) bringt keine grossen Vorteile gegenüber 2.2 (nur fehlerverursachendes Modul verwerfen), sofern der Benutzer informiert wird, welche Module tatsächlich erfolgreich geladen wurden. Die Variante 2.2 ist äquivalent zu 3.1 mehrmals nacheinander ausgeführt.

Somit bleibt 3.1 (mit Option für Mehrfachausführung) als einfachste Variante. Als Luxusvariante für eine erweiterte Version ist 2.3.2 (evtl. 2.3.1) denkbar, damit wenigstens innerhalb einer Modulgruppe die Reihenfolge keine Rolle spielt. Was genau unter einer Gruppe verstanden werden kann, kommt im Abschnitt 5.3 “Modulgruppen” auf Seite 23 noch zur Sprache.

2.2.3 Shellkommandos

Damit der Benutzer ein neues Programm starten zu kann, muss das entsprechende Modul Kommando-Prozeduren zur Verfügung stellen. Typischerweise ist das die Funktion namens `main`, es können aber — wie in Oberon — auch mehrere verschiedene Aufrufe pro Modul vorgesehen werden, der Programmierer muss diese im Sourcecode entsprechend kennzeichnen (siehe 5.2 “Konventionen für die C-Sourcen” auf Seite 22). In Frage kommen Funktionen, die keine andern Parameter als die Standard-Argumente in Form eines topsy-spezifischen String-Arrays erwarten.

Verweise auf diese Kommando-Prozeduren werden in einer speziellen zentralen Liste gespeichert, welche bei Bedarf von Userprogrammen, speziell der Shell, gelesen werden kann. Die Shell nimmt vom Benutzer eingetippte Befehle entgegen und interpretiert diese, indem sie — sofern es kein eingebauter Befehl ist — in der Liste die entsprechende Kommando-Prozedur sucht und dann aufruft.

2.3 Freigeben (Entladen) eines Moduls

Damit ein Modul mehrfach genutzt werden kann, wird es nicht automatisch entladen, sondern erst auf Befehl des Benutzers an die Shell. Grundsätzlich kann auch ein anderes Userprogramm den Entladebefehl geben, so liesse sich für spezielle Fälle eine Freigabeautomatik realisieren. Eine Entlademöglichkeit ist nötig, einerseits um den Speicherbedarf zu reduzieren, andererseits um eine neue Version eines Moduls zu laden, weil nicht zwei Module mit gleichem Namen in der Verwaltungsliste stehen dürfen.

2.3.1 Prüfung von Importbeziehungen

Um das unzeitige Entladen eines Moduls durch den Benutzer zu vermeiden, ist eine Kontrolle vorgesehen, ob das freizugebende Modul im Moment von andern geladenen Modulen importiert wird. Diese Prüfung lässt sich nach dem Vorbild Oberon einfach realisieren: ein anfänglich auf null gesetzter, dem Modul zugeordneter Zähler (reference counter) wird jedesmal um eins inkrementiert, wenn ein neues Modul eine Importbeziehung dahin herstellt. Bei jeder Auflösung eines Imports, durch Entladen des importierenden Moduls, wird der Zähler wieder um eins dekrementiert. Die Freigabe ist nur zulässig, wenn der Zähler auf null steht.

Schwieriger zu kontrollieren sind Referenzen über Prozedurvariablen (function pointers) und Pointer auf globale Daten des Moduls. Da der Userspace ein gemeinsamer Adressraum für alle Usermodule und -threads ist, kann wegen der in C möglichen Pointerarithmetik und dem Adress-Operator (&) nicht verhindert werden, dass ohne Wissen des Betriebssystems Pointer auf beliebige Speicherstellen gesetzt werden. Es wäre denkbar, aber aufwendig, eine zentrale Verwaltung für solche Referenzen anzubieten, womit das System die Verknüpfungen überwachen könnte; damit diese nicht umgangen werden kann, müsste jede Sprunganweisung im Programmcode beim Laden durch einen Systemaufruf ersetzt werden.

Den Zugriff auf Variablen hat man damit noch nicht abgedeckt. Übrigens wurde das nicht einmal in Oberon sauber gelöst, was den Verzicht auf eine solche Verwaltung noch etwas erträglicher macht.

Das Problem ist nicht so schwerwiegend, wenn die Verwaltung des virtuellen Speichers sicherstellt, dass Referenzen auf nicht mehr gültige Speicherblöcke erkannt und mit definiertem Verhalten zurückgewiesen werden. Konkret heisst das, dass der fehlbare Thread sofort beendet wird.

2.3.2 Prüfung auf laufende Threads

Ein Problem, das erst mit parallel laufenden Threads wesentlich wird und deshalb in Oberon keine Rolle spielt, sind momentan benutzte Codeabschnitte in den zu entladenden Modulen. Es ist möglich, dass einem Thread sozusagen der Code unter den Füssen weggezogen wird und er auf ungültigen Anweisungen weiterläuft. Was dann genau passiert ist wiederum von der Speicher- und Threadverwaltung abhängig. Um diesen Fall möglichst zu vermeiden, sollte folgende, in der aktuellen Version des Modulmanagers teilweise schon vorbereitete, Funktionalität implementiert werden:

Ein dem Modul zugeteilter Benutzerzähler wird jedesmal um eins inkrementiert, wenn im Modul ein Thread gestartet wird. Dafür muss allerdings noch ein Informationskanal zwischen Modulmanager und Threadmanager geschaffen werden! Wird ein Thread beendet (`kill` oder `exit`), so erinnert sich der Threadmanager an dessen Startmodul und dekrementiert dort den Benutzerzähler wieder um eins. Solange der Zähler nicht null ist, befindet sich ein Thread im Modul oder er kann mittels `return` dorthin zurückkehren, das Modul darf also nicht freigegeben werden. Solange sich der Thread in verschiedenen vom Startmodul importierten Modulen herumbewegt, ist er dank der Importkontrolle auch geschützt.

Diese relativ einfache Lösung weist zwei Unvollkommenheiten auf:

- Wie schon unter 2.3.1 “Prüfung von Importbeziehungen” erwähnt, kann ein Thread über Prozedurvariablen in beliebige andere Module springen, dort ist er nicht geschützt.
- Umgekehrt kann ein Thread in einer nie endenden Funktion eines andern Modules “gefangen” sein, dann erscheint sein Startmodul als benutzt, obwohl er nie dorthin zurückkehren wird.

Kapitel 3: Entwurf der Software

Dieses Kapitel enthält zuerst einige Argumente zur Entscheidung, ob der dynamische Lader/Linker im Kernel- oder im Userspace implementiert werden soll, und nachher eine Übersicht der gewählten Softwarestruktur.

3.1 Kernel- oder Userspace?

Variantenvergleich möglicher Lösungen:

- **1.** Lader im User-Space.
  (+) Kernel wird nicht betroffen, kein tiefer Eingriff ins System.
  (-) Kernel kann den Lader nicht benutzen, da er nicht als zuverlässig angeschaut werden kann.
- **2.** Lader im Kernel-Space.
  (+) Kernel und User können den Lader nutzen.
  (+) Es gibt keine Probleme mit mehrfach geführten Modulverwaltungen oder gleichzeitigen Zugriffen wegen (versehentlich) mehrfach gestarteten Lader-Threads, weil neue Kernel-Threads nicht beliebig gestartet werden können.
  (+) Mit Einbezug des Threadmanagers ist es teilweise möglich zu prüfen, ob der Programmcode eines bestimmten Moduls momentan noch benutzt wird (siehe 2.3.2 “Prüfung auf laufende Threads”).
  (-) Tieferer Eingriff ins System.
  (-) Der Lader muss zuverlässig sein und darf in keiner Situation unzulässiges Verhalten zeigen, da er sonst die Integrität des Kernels gefährdet.
- **3.** Lader doppelt vorhanden.
  Das heisst, der Lader ist vorläufig nur im Userspace vorhanden, der Kernelspace bekäme einen eigenen (gleichartigen) Lader, sobald eine Version vorgesehen ist, die diesen benutzt.
  (-) Aufwendig.
  (-) Die Infrastruktur wie Listenverwaltung und Locks für gegenseitgen Ausschluss ist im Userteil nicht vorhanden.
- **4.** Teile im Kernel, Teile (z. B. Relozierung) unabhängig
  Nicht ein einziger Systemaufruf führt den ganzen Ladevorgang durch, sondern gewisse Teilvorgänge können von einem unabhängigen Thread oder als Funktionsaufruf(e) direkt durch den auslösenden Thread durchgeführt werden.
  (+) Weniger lange Blockaden des Modul-Managers (könnte z. B. während dem Umkopieren des Codes rasch eine Informationsanfrage bearbeiten).
  (-) Probleme des gegenseitigen Ausschlusses werden verschärft.
  (-) Teilweise im User-Space: Teile des Codes müssen auch für Userthreads benützbar sein, also doppelt vorhanden; Daten müssen den Userthreads zugänglich sein.

Der Entscheid fiel zu Gunsten einer Implementation mit Modulmanager inklusive Linker im Kernelspace, nur der Transfer-Teil bleibt im Userspace.

3.2 Speicherverwaltung und dynamische Module

Für den Code und die Daten der dynamischen Module muss Speicherplatz alloziert und wieder freigegeben werden. Wem gehört dieser Speicher?

Da jedes Modul gleichzeitig oder nacheinander von mehreren Threads — auch nicht “verwandten” — benutzt werden kann, sollte der Speicher nicht einem “gewöhnlichen” Thread gehören: sobald dieser beendet ist, wird der Speicher freigegeben. Für Kernelmodule kann der Lader/Linker/Modulmanager als Besitzer auftreten, da dieser unter normalen Umständen nie beendet wird. Für Usermodule muss aber ein Userthread Besitzer sein!

Der herkömmliche statische User-Programmbereich — eine Region Code, eine Region Data — “gehört” dem Thread mit ID 0, eine ungültige Nummer, die aber eindeutig dem Userbereich zugeordnet ist. Man könnte also auch die dynamisch allozierten Module der Thread-ID 0 zuordnen. Die ID 1 wird bisher nicht für Threads verwendet, die Vergabe beginnt bei 2. Beim Message-Empfang wird die Konstante `ANY` mit dem Wert 1 verwendet, wenn kein bestimmter Thread verlangt ist. Da auch 1 dem Userbereich zugeordnet wird, liegt die Verwendung von `ANY` als Modul-Speicherbesitzer nahe.

Der Kernelthreads können grundsätzlich für einen Userthread mit beliebiger Thread-ID Speicher allozieren (genaugenommen zuerst für sich allozieren, dann verschieben), Userthreads können nur für sich selber allozieren.

3.3 Ergänzte Topsy-Struktur

Topsy ist von Anfang an modular aufgebaut worden, um die Verständlichkeit, die Erweiterbarkeit und die Portierbarkeit zu fördern. Die “Module” sind getrennt kompilierte, statisch gelinkte Quellcode-Abschnitte.

3.3.1 Übersicht

Die bisherige Topsy-Struktur sieht folgendermassen aus (Topsy-Manual [1] zu Version 1 oder 1.1, Seite 8):

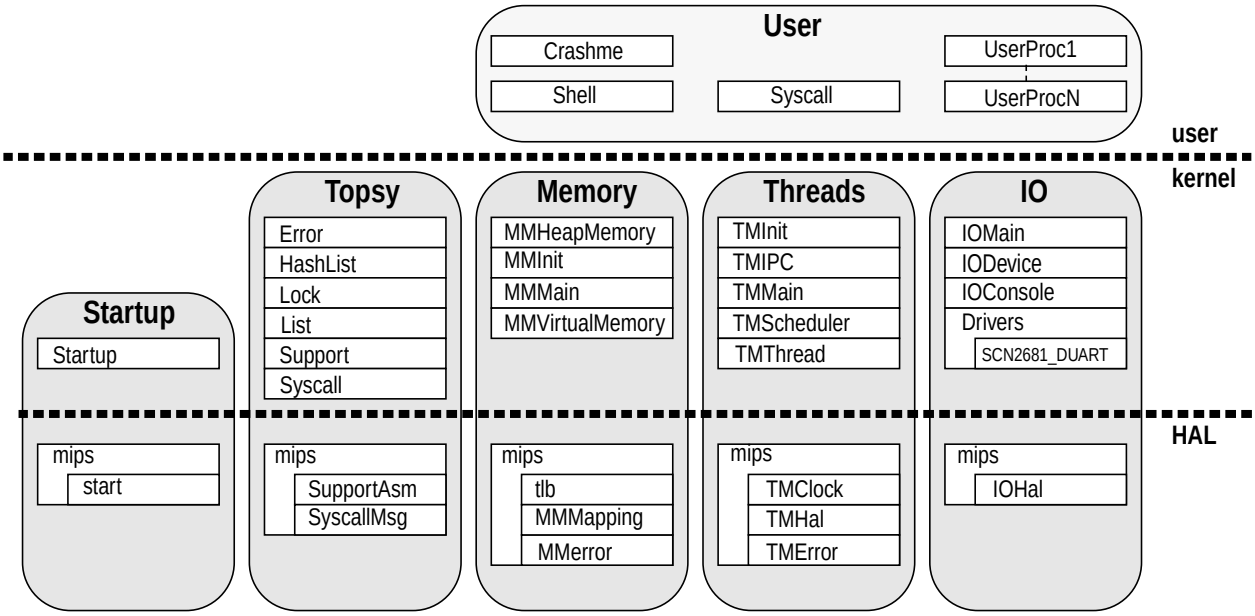

Abb. 1. Bisherige modulare Struktur von Topsy

Topsy mit dynamisch ladbaren Usermodulen enthält ein zusätzliches statisches Kernel-Hauptmodul: `Modules`, genannt Modul-Manager, kennzeichnendes Präfix: `Mod`. Der Modulmanager erledigt die in Form von Systemaufrufen überbrachten Befehle betreffend Modulverwaltung und dynamisches Linken. Seine innere Unterteilung wird im nächsten Abschnitt behandelt.

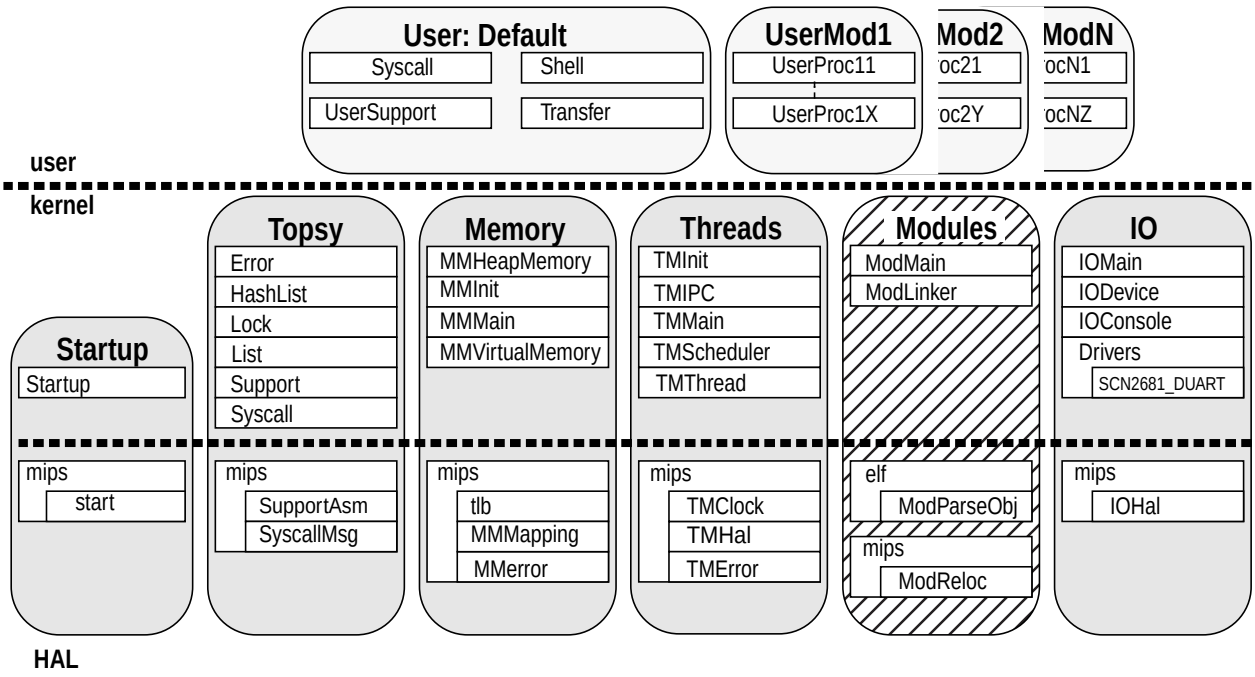

Abb. 2. Neue modulare Struktur von Topsy mit dynamisch ladbaren Usermodulen

Im Userbereich sehen wir die durch überlappende Kästchen angedeuteten dynamisch geladenen Module in prinzipiell beliebiger Anzahl. Der Default-Teil wird in einer ersten Version nach herkömmlicher Art und Weise geladen und gelinkt, soll aber zwecks Vereinheitlichung später auch zu einem oder mehreren dynamisch gelinkten Modulen werden. Als Besonderheit muss deren Programmcode trotzdem weiterhin im gemeinsamen “Start-Pool” enthalten sein, da das System noch keine externen Daten aufnehmen kann.

Der Default-Teil enthält die bekannten (Teil-)Module `Syscall`, `UserSupport` und `Shell`. Eine Neuerung stellt `Transfer` dar: Dieses Teilmodul ist für die Uebertragung der Moduldaten von einer externen Schnittstelle (oder aus einer Datei) in den Topsy-Arbeitsspeicher zuständig. `Transfer` befindet sich im Userbereich, um die Anpassung an verschiedene Schnittstellen zu erleichtern. Falls später auch Kernelmodule darüber geladen werden sollen, müssen diese auf geeignete Art vor dem Linken verifiziert werden.

3.3.2 Das neue Kernel-Modul ‘Modules’

Der Modulmanager besteht aus mehreren Teilmodulen, in denen ein Thread (genannt `modThread`) in einer Schleife läuft, auf Meldungen bzw. Systemaufrufe der andern Kernel- und Userthreads wartet, und diese verarbeitet. Diese Lösung ist analog zum Memory-Manager, zum Thread-Manager und zur Ein- und Ausgabe gewählt. Da nur genau ein Thread die Modulverwaltung erledigt, stellen sich keine Probleme mit gegenseitigem Ausschluss bei Zugriffen auf die zentralen Datenstrukturen.

Zentral verwaltet wird eine Liste aller im Moment geladenen dynamischen Usermodule (später zu ergänzen durch je eine entsprechende Liste pro Adressraum). Die Liste enthält von jedem Modul Name, Importliste und alle exportierten Symbole inklusive deren Speicheradressen. Eine weitere Liste enthält Funktionen, die als Shellkommando verfügbar sind, d. h. sie akzeptieren die Standard-Argumente in Form eines topsy-spezifischen String-Arrays und sie wurden vom Programmierer entsprechend gekennzeichnet.

Der Topsy-Kernel ist eingeteilt in einen allgemein verwendbaren Teil und einen hardwareabhängigen Teil (genannt hardware abstraction layer, HAL, im Falle der Version für MIPS R3000-Prozessor bezeichnet mit `mips`). `Modules` ist eingeteilt in einen allgemeinen Teil (`ModMain` und `ModLinker`), einen hardwareabhängigen Teil (`mips`) und einen modulformatabhängigen Teil (`elf`, siehe 5.1 “Das ‘object file’ zum Laden” auf Seite 21), dazu eine kleine Übersetzungsfunktion als nicht dargestellten weiteren Teil (`mipself`, siehe 6.5.1 “Relokationstypen” auf Seite 30). Der allgemeine Teil ist zwecks Übersichtlichkeit in zwei Untermodule aufgeteilt.

Die Hardware-Abhängigkeit ist in der Relokation begründet, die angepassten Speicheradressen müssen im dem Prozessor entsprechenden Format auf Bit-Ebene in den Programmcode eingesetzt werden. Die Format-Abhängigkeit steckt im Linker, der Symbole wie Funktions- und Variablennamen aus der Moduldatei lesen muss. Die übrigen Teile von Topsy werden durch das Format nicht beeinflusst, da die gesamten statischen Link-Aufgaben durch Kompilierwerkzeuge vor dem Aufstarten von Topsy erledigt werden.

Wird die Hardware oder das Modulformat gewechselt, so sollte nur je ein Teilmodul sowie die kleine Übersetzungsfunktion für die Relokationstypen mit ihren Konstanten neu geschrieben werden müssen. Die angestrebte Portabilität von Topsy wird somit nicht beeinträchtigt.

3.3.3 Änderungen in bestehenden Topsy-Modulen

Es folgt eine Auflistung der durchgeführten bzw. für Erweiterungen voraussichtlich nötigen Änderungen mit genauem Verweis auf die bestehenden C-Sourcen.

a) In jedem Fall erforderliche Änderungen (ausgeführt)

`Configuration.h` (Topsy): Definition von globalen Konstanten betreffend z. B. Namenslängen.

`Topsy.h` (Topsy): Konstante Thread-ID für den Laderthread definieren (`MODTHREAD = -3`, neu dafür `IOTHREAD = -4`, dies, weil der Modulmanagerthread sicher vor den verschiedenen gelinkten IO-Treibern geladen werden muss um eine definierte ID zu bekommen); `FIRST_KERNELTHREAD` neu selbstständig definiert, nicht mehr mit `IOTHREAD` gleich gesetzt.

`TMMain.c` (Threads): Vor dem Start des IO-Threads muss noch der Modulmanager-Thread gestartet werden.

`Messages.h` (Topsy): Neue Meldungstypen definieren: pro Systemaufruf (Syscall) je einen Meldungstyp für den Aufruf und die Rückmeldung (`reply`). Dazu noch Typen für die Rückgabe von Informationen über die dynamischen Module an das übrige Topsy.

`Syscall.h`, `Syscall.c` (Topsy): Pro neuem Systemaufruf wird eine neue Funktion benötigt.

`Shell.c, Shell.h` (User): Die weiterhin im Userspace angesiedelte Shell bekommt neue eingebaute Kommandos (`load`, `free`, `modinfo`) sowie die Fähigkeit, die von ModMain verwaltete Liste mit Kommando-Prozeduren (siehe 2.2.3 “Shellkommandos” auf Seite 10) zu interpretieren und damit Funktionen in neu geladenen Modulen aufzurufen.

b) Änderungen im Zusammenhang mit User-Default

Die folgenden Änderungen werden nur benötigt, wenn das Laden und Linken (und Starten) des User-Default-Teiles geändert wird, was zur Zeit nicht der Fall ist.

`MMDirectMapping.c`, `MMDirectMapping.h`, `MMInit.c` (Memory): Der Aufruf der Funktion `mmInitMemoryMapping` ändert und die ausgeführte Funktion ändert oder entfällt, da es keine im Voraus bekannten Grössen und Adressen der Code- und Datenabschnitte mehr gibt.

`MMVirtualMemory.c`, `MMVirtualMemory.h`, `MMInit.c` (Memory): Der Aufruf der Funktion `mmVmInit` ändert und die ausgeführte Funktion ändert, da es keine im Voraus bekannten Grössen und Adressen der Code- und Datenabschnitte mehr gibt: der ganze Userspace wird als frei initialisiert.

`TMMain.c` (Threads): `threadStart(... userInitAddress, ... "shell", ...)` entfällt, falls dies der Modulmanager selber erledigt. Jener muss zuerst den User-Default-Teil (Syscall, Shell usw.) als Modul(e) laden und daher wissen, an welcher RAM-Adresse diese “Datei” zu finden ist; anschliessend kann er gerade den Shell-Thread (oder einen User-Startup-Thread) starten. Als Alternative könnte der Modulmanager die Startadresse dem Threadmanager mitteilen, und dieser startet wie bisher die Shell.

c) Änderungen im Zusammenhang mit Thread-Überwachung

Falls der Threadmanager mithelfen soll, die momentane Benutzung der Codemodule zu überwachen, braucht es noch mehr Anpassungen, auch diese sind noch nicht realisiert:

`tm-include.h` (Threads): Im Thread-Deskriptor muss ein Verweis auf das Modul, wo der Thread gestartet wurde, eingefügt werden (Nummer oder Zeiger).

`TMThread.c` (Threads): Die Funktion `threadStart` muss den Benutzerzähler des Startmoduls um eins inkrementieren und den Modulverweis im Thread-Deskriptor eintragen. (Woher weiss der Thread-Manager, in welchem Modul sich die Startfunktion befindet?) Die Funktion `threadDestroy` muss den Benutzerzähler des Startmoduls wieder um eins dekrementieren.

Kapitel 4: Interface der neuen Topsy-Bestandteile

Der neue Modulmanager muss auf geeignete Weise mit den bestehenden Teilen von Topsy verknüpft werden. Das neue User-(Teil-)Modul Transfer muss aufgerufen werden und mit der “Aussenwelt” kommunizieren, und schliesslich muss die Shell dem Benutzer neue Befehle zur Verfügung stellen.

Die Interfaces zwischen den einzelnen Teilen des Modulmanagers/Laders/Linkers werden in Kapitel 6 “Implementierung der neuen Topsy-Bestandteile” auf Seite 25 beschrieben.

4.1 Systemaufrufe

Der Modulmanager lässt sich über neue Systemaufrufe (system calls) aktivieren. Dazu muss das sowohl im Kernel- wie auch im Userspace vorhandene (Teil-)Modul Syscall statisch oder dynamisch importiert werden. Die Definitionen der möglichen Rückgabewerte stehen im Header `Messages.h`. Die folgenden neuen Systemaufrufe haben als gemeinsames Kennzeichen die Vorsilbe `mod`.

4.1.1 Neues Modul laden

```
SyscallError modLoad(Address objImagePtr, AddressSpace mode, ModuleInfo
                          answerArray[], int* nbOfAnswers);
```

Erwartet an der Arbeitsspeicheradresse `objImagePtr` (egal, ob User- oder Kernelspace) die Kopie eines kompilierten Modules (“object file”). Der Parameter `mode` kann `USER` oder `KERNEL` (Konstanten aus `Memory.h`) sein und gibt an, in welchen Adressraum das Modul geladen werden soll, wobei ein Userprogramm keinen Befehl zum Laden eines Kernelmodules geben kann. Falls genauere Informationen über den Ladevorgang gewünscht werden, kann mit dem `answerArray` gearbeitet werden, siehe 4.1.6 “Informationsrückgabe an den Aufrufer (answerArray)” und folgende Liste.

Rückgabewerte:

- `MOD_LOADOK`: Ladevorgang erfolgreich, Name des geladenen Moduls im `answerArray` (sobald Modulgruppen unterstützt werden, können dies auch mehrere sein).
- `MOD_LOADCALLFAILED`: Probleme mit Systemaufruf, zu wenig Speicher oder andere Fehler.
- `MOD_LOADNORIGHT`: Ein Userprogramm versuchte ein Kernelmodul zu laden.
- `MOD_LOADMULTIPLEMOD`: Modul mit gleichem Namen existiert bereits, siehe `answerArray[0]`.
- `MOD_LOADIMPORTMISSING`: Nicht alle importierten Module sind geladen, siehe `answerArray`. Rekursives Laden könnte mit Hilfe der zurückgegebenen Angaben auf User-Seite realisiert werden.
- `MOD_LOADOBJERROR`: Formatfehler in der “object file”-Kopie (nicht alle Fehler werden erkannt!).
- `MOD_LOADSYMBOLERROR`: Topsy-Konventionen für Module verletzt, z. B. kein Modulname angegeben oder unvollständige Importliste.
- `MOD_LOADSYMBOLNOTFOUND`: Ein importiertes Symbol wurde in den Symboltabellen der importierten Module nicht gefunden.

4.1.2 Modul freigeben (entladen)

```
SyscallError modFree(ModuleId mid, ModFreeMode freemode);
```

Entlädt das Modul mit der Identifikationsnummer `mid`. Für den Parameter `freemode` sind drei verschiedene Werte zulässig: `FREENORMAL`; `FREEOVERR_ACTIVETHREAD` (entladen, obwohl noch ein Thread im Modul aktiv zu sein scheint); `FREERECURSIVE` (rekursiv entladen: alle nur genau vom zu entladenden Modul importierten Module werden auch entladen, analog zu Oberon, aber noch nicht implementiert).

Rückgabewerte:

- `MOD_FREEOK`: Entladevorgang erfolgreich.
- `MOD_FREECALLFAILED`: Probleme mit Systemaufruf oder andere Fehler.
- `MOD_FREENORIGHT`: Ein Userprogramm versuchte ein Kernelmodul zu entladen.
- `MOD_FREENOTFOUND`: Kein Modul mit dieser Identifikationsnummer vorhanden.
- `MOD_FREEIMPORTED`: Nicht entladen wegen bestehender Importbeziehung.
- `MOD_FREERUNNINGTHREAD`: Nicht entladen wegen eines aktiven Threads im Modul (noch nicht implementiert!), die Sperre kann mittels `freemode = OVERR_ACTIVETHREAD` umgangen werden.

4.1.3 Namen oder Identifikationsnummern von Modulen suchen

```
SyscallError modGetInfo(char* name, ModuleId mid, ModuleInfo* answer);
```

Um die Nummer zu einem bekannten Namen zu suchen ist `mid = 0` zu setzen und der Name zu übergeben. Um den Namen zu einer bekannten Nummer zu suchen ist `name = ""` oder `name = NULL` zu setzen und die gesuchte Nummer in `mid` zu übergeben. Die vollständige Modulbezeichnung mit Namen und Nummer sowie den weiteren in `ModuleInfo` möglichen Angaben wird in `answerArray[0]` geschrieben. Ist das gesuchte Modul nicht vorhanden, werden alle Felder des `answerArray[0]` auf 0 oder NULL-Pointer gesetzt.

In der aktuellen Version reagiert dieser Befehl gleich wie 4.1.4 “Modulliste abrufen”.

Rückgabewerte: `MOD_GETINFOOOK` und `MOD_GETINFOFAILED`.

4.1.4 Modulliste abrufen

```
SyscallError modGetInfoList(ModuleInfo answerArray[], int* nbOfAnswers);
```

Füllt das `answerArray` (siehe 4.1.6 “Informationsrückgabe an den Aufrufer (answerArray)”) mit Namen, Nummern und allen weiteren in `ModuleInfo` möglichen Angaben der momentan geladenen dynamischen Module (noch nicht implementiert).

Die aktuelle Version gibt keine Daten zurück, sondern schreibt eine Modulliste direkt auf das Terminal (Console).

Rückgabewerte: `MOD_GETINFOOOK` und `MOD_GETINFOFAILED` (wie bei `modGetInfo`).

4.1.5 Shellkommandos (Kommandoprozeduren) abrufen

```
SyscallError modGetUserCommands(
                  UserCommandsInfo answerArray[], int* nbOfAnswers);
```

Füllt das `answerArray` vom Typ `UserCommandsInfo` (siehe 4.1.6 “Informationsrückgabe an den Aufrufer (answerArray)”) mit Namen, Adresse und Typ der von den dynamischen Modulen zur Verfügung gestellten Kommando-Prozeduren (noch nicht implementiert).

Rückgabewerte: `MOD_GETUSERCOMMANDSOK` und `MOD_GETUSERCOMMANDSFAILED`.

4.1.6 Informationsrückgabe an den Aufrufer (`answerArray`)

Manche der obigen Systemaufrufe geben weitere Informationen zurück, wenn ihnen entsprechenden Speicherplatz im Adressbereich des Aufrufers zur Verfügung gestellt wird. Der Parameter `answerArray` muss einen Pointer zu einem Array des entsprechenden Typs enthalten, `*nbOfAnswers` eine Variable mit dessen Anzahl Elemente, ähnlich wie bei `ioRead` die maximale Anzahl Byte angegeben wird. Werden keine Informationen gewünscht, soll mindestens einer der beiden Parameter ein NULL-Pointer sein oder `*nbOfAnswers` soll 0 enthalten.

Die tatsächlich geschriebene Anzahl Array-Elemente wird in `*nbOfAnswers` abgelegt. Ist diese Zahl kleiner als vorher, so heisst das, dass keine weiteren Informationen erzeugt wurden; ist die Zahl gleich gross, so kann nicht bestimmt werden, ob weitere Informationen verfügbar gewesen wären.

Die Datentypen der zurückgegebenen Informationen sind ebenfalls in `Messages.h` definiert.

4.2 Das Interface zu Transfer

Das neue User-(Teil-)Modul Transfer wird — als erste Handlung zum Laden eines dynamischen Moduls — typischerweise von der Shell aufgerufen. Transfer stellt folgende zwei Funktionen zur Verfügung:

```
void transfer(Address* startAddrPtr, unsigned long int* lengthPtr);
void transferFree(Address startAddr);
```

Die aktuelle provisorische Version von `transfer` alloziert Speicher, liest die erhältlichen Daten ab der vordefinierten Schnittstelle (siehe Anhang B “Beschreibung der Entwicklungsumgebung” auf Seite 38) und gibt deren Anfangs- und Endadresse zurück. Transfer könnte für beliebige Daten verwendet werden, nicht nur für Module. `transferFree` gibt den allozierten Speicher wieder frei, die angegebene Adresse muss stimmen!

Der Aufruf von Transfer könnte später z. B. um ein Feld für den Modul- oder Dateinamen erweitert werden, falls eine Möglichkeit, bestimmte Dateien zu laden, eingebaut werden soll.

4.3 Das Interface zum Benutzer

Die Systemaufrufe des Modulmanagers werden vom Benutzer typischerweise mit Hilfe der Shell benutzt. Da auch die Implementationen der Shell zur Zeit noch sehr provisorischen Charakter hat, soll an dieser Stelle keine genauere Beschreibung erfolgen.

4.4 Interne Interfaces

Die Interfaces zwischen den einzelnen Teilen des Modulmanagers sind im Text des Kapitels 6 “Implementierung der neuen Topsy-Bestandteile” beschrieben.

Kapitel 5: Modulformat

Die dynamisch ladbaren Module können leider nicht genau gleich erstellt werden wie die konventionellen.

5.1 Das ‘object file’ zum Laden

Da möglichst nur allgemein verwendete Tools zum Kompilieren der Topsy-Programme auf dem Hostsystem verwendet werden sollen, muss ein Standard-Format verwendet werden. Naheliegend wäre das ECOFF-Format, da es bisher für Topsy verwendet wird und alle Tools vorhanden sind. Leider war dazu aber keine detaillierte Dokumentation der internen Formate zu finden; da bisher die kompilierten Dateien fast nur mit Standard-Tools weiterverarbeitet werden, stört das fürs konventionelle Topsy nicht. Der ganze Linkvorgang der statischen Module findet auf dem Host-System statt, einziges vom Institut TIK selbst entwickeltes Tool ist dabei der Bootlinker, der anhand der von Standard-Tools gelieferten Adress- und Grössenangaben den Kernel- und den Userteil verknüpft.

Für die dynamischen Module musste ein eigener Linker entwickelt werden, der in Topsy eingebettet ist. Er muss den byteweisen Aufbau der kompilierten Module kennen, um die Symbole und ihre Adressen herauslesen zu können. Die genauen Spezifikationen konnten für das ELF-Format gefunden werden, siehe [6].

Der dynamische Topsy-Linker verlangt als Eingabe ein “relocatable object file” im ELF-Format, direkt so, wie es der Compiler liefert, bekannt als “`.o`”. Die für ELF ebenfalls definierten Dateitypen “executable file” und “shared object file” werden nicht verwendet. Die generierten Dateien können recht umfangreich werden, wenn sie Debug-Informationen enthalten; der Verzicht auf diese durch geeignete Compiler-Optionen ist zu empfehlen. Falls ein Debugger zum Einsatz kommen soll, besteht auch die Möglichkeit, je zwei Versionen von jedem Modul zu erzeugen: eine mit Debug-Daten für den Debugger und eine ohne solche zum Laden. Je nach verwendeter Hardware ist ein Cross-Compiler nötig, da das Hostsystem und die Topsy-Maschine normalerweise nicht identisch sind. In der aktuellen Situation läuft der Compiler auf einer Workstation mit SPARC-Prozessor und erzeugt Maschinencode für einen MIPS R3000-Prozessor (GNU C Compiler, “Cross” für Ausgabeformat mips-elf-elf, Aufruf typischerweise mit `mips-elf-elf-gcc`).

Der Topsy-Linker kann nicht mit dem MIPS-spezifischen globalen Datenpointer (global pointer, `$gp`) umgehen. Der GNU-C-Compiler benutzt diesen für den Zugriff auf die Sektionen `.sbss` und `.sdata` (“kurze” Daten). Die Erzeugung dieser Sektionen wird vermieden durch die Compiler-Option `-G 0` (d. h. maximale Länge für “kurze” Daten ist Null).

Achtung: der Compiler bemerkt nicht in jedem Fall, wenn eine aufgerufene Funktion nicht existiert (beziehungsweise falsch geschrieben wurde)! Da er die Import-Konventionen nicht kennt, nimmt er an, dass die betreffende Funktion von einem beliebigen anderen Modul implementiert werde. Eine typische Meldung in diesem Fall ist — wenn überhaupt etwas gemeldet wird — `warning: implicit declaration of function ‘...’`, was aber nicht zum Abbruch des Kompiliervorgangs führt.

Falls ein dynamisches Topsy-Modul aus mehreren, untereinander statisch gelinkten Teilmodulen bestehen soll, können mehrere kompilierte Dateien mittels partiellem Linken (auch inkrementelles Linken genannt) zu einem Modul verbunden werden (vgl. Schluss des Abschnittes 5.2 “Konventionen für die C-Sourcen”). Der auf dem Host-System laufende Linker muss dabei als Ausgabe wieder eine “`.o`”-Datei liefern, die weiterhin relozierbar ist. Der GNU Link Editor (auch “Cross”: Aufruf mit `mips-elf-elf-ld`) erledigt dies mit einer der Optionen `-r` oder `-i`.

5.2 Konventionen für die C-Sourcen

Um Kollisionen von gleich benannten Symbolen in verschiedenen Modulen zu vermeiden, ist eine eindeutige Zuordnung zum exportierenden Modul von Vorteil. Der in C an sich flache Namensraum wird durch Konventionen strukturiert. Der dynamische Topsy-Linker ist auf eine korrekte Handhabung angewiesen, doch leider kann sie der übliche C-Compiler nicht überprüfen, da sich dieser weiterhin nur an den Header-Dateien (“`.h`”) orientiert. Die explizite Importliste erleichtert das Zusammensuchen der einzelnen Symbole in verschiedenen Modulen durch den Topsy-Linker.

- Jedes Modul hat einen Namen, der keine Unterstreichungszeichen enthalten darf.
- Die Liste der dynamisch zu importierenden Module ist explizit im Programmcode enthalten (nicht im Header, sonst wird sie mittels `#include` in die andern Module übertragen!), Syntax siehe Beispielprogramm.
- Die Header der zu importierenden Module werden weiterhin mittels `#include` eingebunden.
- Alle dynamisch exportierbaren Funktionen und Variablen sind im Programm und im Header mit dem eigenen Modulnamen als Präfix, abgetrennt durch ein Unterstreichungszeichen, versehen. Im Symbolnamen selber dürfen beliebig viele weitere Unterstreichungszeichen auftreten.
- Kommando-Prozeduren (Funktionen, die als Shellbefehle aufgerufen werden können) dürfen als Parameter nur entweder die Standard-Argumente in Form eines topsy-spezifischen String-Arrays (`ThreadArg*`) oder gar nichts erwarten. Rückgabewerte sind zulässig, aber nicht sinnvoll.

  Die Kommandos werden wie andere dynamisch exportierbare Funktionen behandelt, zusätzlich werden sie in einer Liste angegeben, Syntax siehe Beispielprogramm.
- Die dynamisch importierten Funktionen und Variablen sind mit dem entsprechenden Modulpräfix versehen, also genau so, wie sie in den zugehörigen Headern stehen.

Konstanten werden wie Variablen behandelt, sofern sie wie Variablen deklariert sind. Typen und mittels `#define` definierte Konstanten sind nicht betroffen, da sie in C nur textuell aus dem Header importiert werden. Es wird empfohlen, die Konvention zwecks Übersichtlichkeit trotzdem einzuhalten.

```
/*************** header file: Hello.h
    Demo file for Topsy with dynamic loader */

void Hello_main();

/*************** end of file */

/*************** file: Hello.c
    Demo file for Topsy with dynamic loader */

/* includes are needed for the compiler */
#include "../Topsy/Topsy.h"
#include "DynSyscall.h"   /*  .h-file is DynSyscall.h,
                              but its module name is Syscall */
#include "UserSupport.h"

/* import list for the module manager / dynamic linker:
           first the own module name, then all imports */
void MOD_Import_Hello_Syscall_UserSupport;

/* list of the shell commands (additional commands without prefix): */
void MOD_Commands_Hello_main;
```

```
/* exported functions (and variables) need the module name prefix */

void Hello_main() {
    ThreadId console;  /*  ThreadId is a type imported from Topsy.h
                           which has no module name, so no prefix */

    /* imported functions need the module name prefix */
    Syscall_ioOpen(IO_CONSOLE, &console);
    UserSupport_display(console, "Hello dynamic world!\n");
    Syscall_ioClose(console);
}
/*************** end of file */
```

Die Importliste ist eine Variable ohne Inhalt — Typ `void` — und belegt damit keinerlei Speicher in der Daten-Sektion. Sie wird vom Compiler trotzdem in die Symboltabelle eingetragen! Falls ein anderer Compiler dies nicht tut, würde ich folgende Alternative vorschlagen, wobei der Topsy-Linker natürlich angepasst werden müsste:

```
static const char MOD_Import_Hello[] = "Syscall_UserSupport";
```

Der Beginn `MOD_Import_` und `MOD_Commands_` ist eine Art Schlüsselwort für den Topsy-Linker, Gross- und Kleinschreibungen sind zu beachten.

Bei globalen Variablen ist darauf zu achten, dass sie vom Compiler nicht vorläufig unbestimmten gemeinsamen Sektionen (`COMMON`) zugeordnet werden, da der dynamische Topsy-Linker in der aktuellen Version keine neuen Sektionen erzeugen kann. Dies kann einfach vermieden werden: die globalen Variablen werden vom GNU-C-Compiler dem aktuellen Modul zugeordnet, falls sie mit `static` deklariert werden oder falls sie bei der Deklaration initialisiert werden.

Falls ein dynamisches Topsy-Modul aus mehreren, untereinander statisch gelinkten Teilmodulen bestehen soll (siehe Schluss des Abschnittes 5.1 “Das ‘object file’ zum Laden”), darf in den verschiedenen Programmdateien insgesamt nur eine Importliste mit dem gemeinsamen Modulnamen vorhanden sein. Die Teilmodule werden klassisch über die Header verknüpft, nur die dynamisch zu exportierenden Symbole benötigen das Modulname-Präfix.

5.3 Modulgruppen

Ich verwende den Ausdruck “Modulgruppe” für Module, die zwar nichts direkt miteinander zu tun haben, die aber gleichzeitig geladen werden sollen. Bei der aktuellen Version des dynamischen Laders/Linkers ist dies noch nicht möglich. Da es für eine angenehme Handhabung des Systems aber erwünscht ist, wurden folgende zwei Möglichkeiten betrachtet:

1. Die Module einer Gruppe werden statisch untereinander gelinkt und werden somit zu einem Modul, das aber mehrere Namen mit je einer Importliste enthält. Die exportierbaren Symbole kann der Linker anhand der Präfixe den einzelnen Modulen zuordnen, lokale Symbole nicht. Wird eines der Module später wieder entladen, so wird dessen Liste mit exportierbaren Symbolen gelöscht; der Programmcode selber kann erst entfernt werden, wenn keines der Module mehr vorhanden ist. Da auch die Reihenfolge der Module innerhalb der Gruppe nicht rekonstruierbar ist, gilt Laderverhalten bei Importproblemen (nach 2.2.2 “Importbeziehungen” auf Seite 9): Variante 2.3.1.

   Obige Möglichkeit ist teilweise vorbereitet, ihre Realisierung läuft aber der Einfachheit und der Einheitlichkeit für den Programmierer von Userprogrammen eher zuwider.

• **2.** Die Module einer Gruppe werden vor dem Laden nicht gelinkt, sondern in geeigneter Weise “hintereinandergestellt”, sodass sie von Topsy nacheinander gelesen werden. Jedes Modul kann für sich behandelt werden, das Verhalten ist genau gleich, wie wenn sie in getrennten Ladevorgängen verarbeitet würden. Laderverhalten bei Importproblemen (nach 2.2.2 “Importbeziehungen” auf Seite 9): Variante 3.1 mit Option auf Mehrfachausführung.

  Diese Möglichkeit könnte im Userspace realisiert werden (Änderung des Transfers, mehrfacher Aufruf des Linkers), erfordert aber auf Seite des Hostsystems noch ein geeignetes Tool.

Kapitel 6: Implementierung der neuen Topsy-Bestandteile

Eine Übersicht des Ladevorgangs verdeutlicht die Aufteilung in die verschiedenen Teilmodule (zur Trennung Userspace — Kernelspace: Transfer benutzt natürlich auch hier nicht dargestellte Systemaufrufe des IO-Modules im Kernelspace).

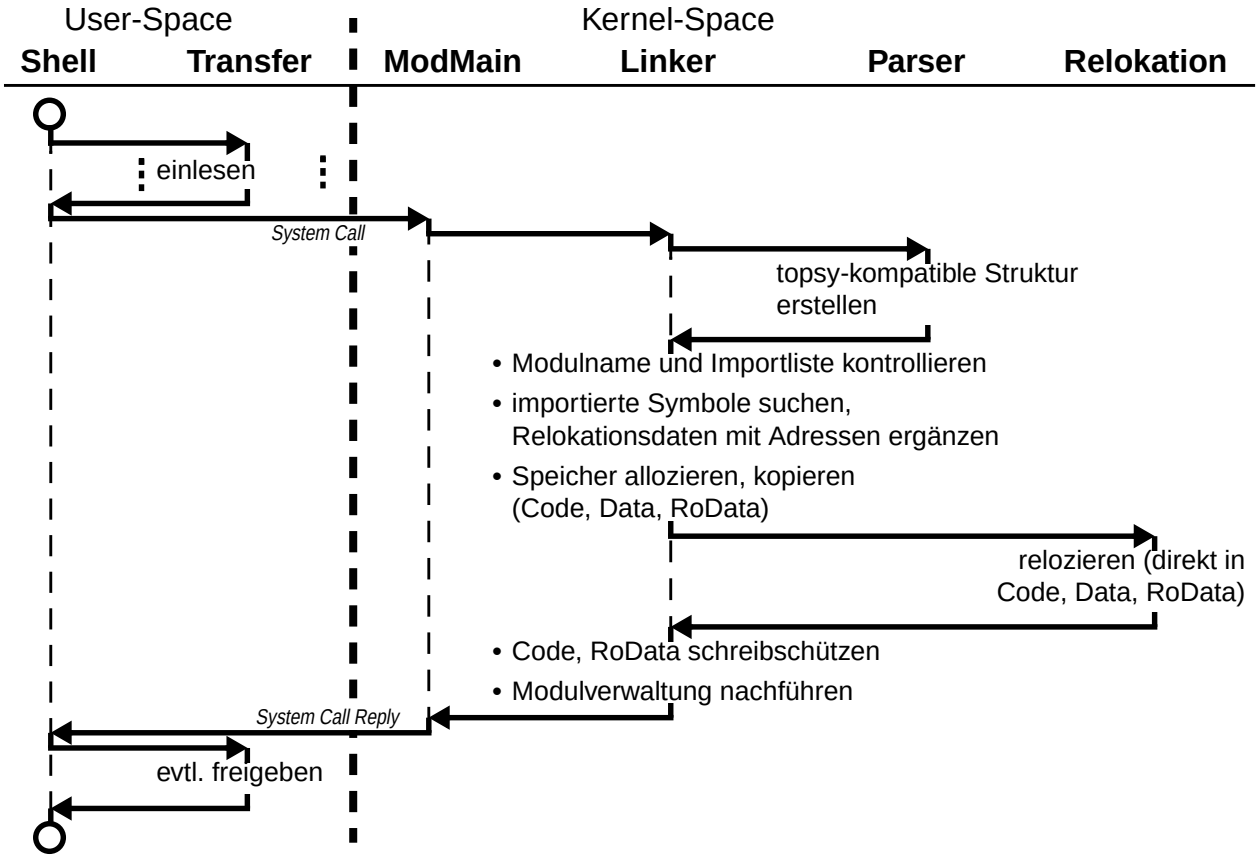

Abb. 3. Ladevorgang eines Usermoduls nach Befehl `load` vom Benutzer an die Shell

6.1 Transfer

Die aktuelle Version von Transfer, ein (Teil-)Modul im Userspace, greift auf den provisorischen Treiber `IO_FILEREADER` zu, der von der simulierten Maschine her eine Datei des Hostsystems lesen kann (siehe Anhang B “Beschreibung der Entwicklungsumgebung” auf Seite 38). Transfer liefert nach dem Einlesen des “object file” die Startadresse von dessen Kopie im Topsy-Arbeitsspeicher zurück an die Shell. Nach Abschluss des Ladevorgangs bekommt Transfer den Befehl, die Kopie wieder freizugeben. Interface-Beschreibung: siehe 4.2 “Das Interface zu Transfer” auf Seite 20.

Da zur Zeit keine Informationen über die Dateigrösse übermittelt werden, kann Transfer nicht wissen, wieviel Speicher alloziert werden soll. Der Einfachheit halber werden jetzt immer 52 kB belegt, was bei üblichen Moduldatei-Grössen von 2 bis 10 kB immer genügen sollte. Eine verbesserte Implementation wird natürlich mit weniger Speicher beginnen und bei Bedarf weitere Seiten allozieren.

6.2 ModMain

ModMain beinhaltet die Organisation der verschiedenen Teile des Modulmanagers im Kernelspace. Hier befinden sich die zentralen Listen zur Verwaltung der geladenen Module. Das Programm besteht hauptsächlich aus der endlosen Schleife, die auf Meldungen (Systemaufrufe) wartet, und diese anschliessend verarbeitet. Je nach verlangter Aktion werden Funktionen der entsprechenden Untermodule aufgerufen. In der aktuellen Version ist zudem noch die Ausgabe der Modulliste direkt auf das Terminal (Console) integriert. Dies soll später durch die Rückgabe von Informationen nach Systemaufrufen ersetzt werden.

6.3 Linker

Das Teilmodul ModLinker übernimmt die format- und hardwareunabhängigen Aufgaben beim Laden und Einbinden oder beim Freigeben eines dynamischen Modules. Die Bezeichnung “Linker” wird den tatsächlichen Fähigkeiten dieses Teilmoduls nicht gerecht, erledigt es doch ziemlich viele verschiedene Aufgaben im Zusammenhang mit dem Laden, dem Linken und der Aktualisierung der Modulverwaltung.

6.3.1 Laden/Linken

Die Funktion `modLoadLink` benutzt verschiedene lokale und auch externe Funktionen, die jeweils einzelne Aufgaben erledigen.

Als erstes ruft `modLoadLink` den Parser auf, der die internen Strukturen des neuen Moduls in topsy-kompatibler Weise darstellt (siehe 6.4 “Parser”) und die Zeiger in `parseResults` entsprechend setzt.

Als nächstes wird Speicher für die Code- und Datensektionen alloziert, verteilt auf drei Regionen: eine für den Code, eine für Nur-Lese-Daten, eine für die übrigen globalen Daten. In jede Region können mehrere Sektionen hintereinander geladen werden; beim üblichen Format werden die initialisierten les- und schreibbaren Daten (Sektion `.data`) mit den nicht initialisierten Daten (Sektion `.bss`) zusammengesetzt. Anschliessend werden Daten und Code — noch ohne relozierte Sprungadressen — an die endgültigen Speicheradressen kopiert. Falls später während dem Linken Fehler auftreten, wird der Speicher wieder freigegeben.

Ursprünglich war vorgesehen, den Speicher erst nach den Kontrollen der Importbeziehungen und Symboltabellen zu allozieren, was den Nachteil hat, dass die endgültigen Adressen der Funktionen und Daten erst später zur Verfügung stehen. Im Hinblick auf die mögliche Unterstützung von Modulgruppen (5.3 “Modulgruppen” auf Seite 23, Möglichkeit 1.) ist aber eine Adressberechnung in einem Schritt wünschenswert, um nicht Zwischenresultate für mehrere Module speichern zu müssen.

Die folgende `do – while` Schleife wird erst benötigt, wenn Modulgruppen unterstützt werden. Die aktuelle Version durchläuft sie genau einmal.

- Speicher für einen Moduldeskriptor wird auf dem Heap des Kernels alloziert und ein Pointer namens `newModule` darauf gesetzt. Die einzelnen Elemente werden auf 0 oder NULL-Pointer initialisiert, die Speicheradressen werden bereits endgültig eingetragen.
- Die Symboltabelle in `parseResults` wird (ab dem Anfang) nach einem (ersten) Modulnamen mit Importliste durchsucht, erkennbar am Präfix `MOD_Import_`. Falls erfolgreich, wird eine Modul-Identifikationsnummer berechnet und je eine leere Liste für Importbeziehungen und für exportierte Symbole im Kernel-Heap erstellt. Falls der Modulname bereits existiert, wird abgebrochen.
- Die importierten Module werden gesucht, und — falls gefunden — Verweise auf diese in die Liste für Importbeziehungen eingetragen.

- Die Symboltabelle in `parseResults` wird wieder von Anfang an nach Symbolen, deren Namenspräfix mit dem vorher gefundenen Modulnamen übereinstimmt, durchsucht. Diese werden mit Namen und endgültiger Speicheradresse in die Liste der exportierten Symbole eingetragen.
- (Noch nicht implementiert: Die eventuell vorhandene Liste der Kommando-Prozeduren mit übereinstimmendem Modulnamen wird gesucht und die darin stehenden Kommandos mit der Liste der exportierten Symbole verglichen. Bei Übereinstimmung wird der Kommandotyp in der Symbolliste entsprechend gesetzt. Alle andern Symbole sind keine Kommando-Prozeduren.)
- Wenn bis hierher keine Fehler aufgetreten sind, wird `*newModule` in die Liste der momentan geladenen Module eingefügt und die Importreferenzzähler der importierten Module werden je um eins inkrementiert. (Noch nicht implementiert: Die Symbole mit entsprechendem Kommandotyp werden in die Liste der Shellkommandos eingefügt.)
- Informationen für die Rückgabe an den Aufrufer werden im `answerArray` eingetragen.

Hier endet die vorläufig ohnehin nur einmal durchlaufene Schleife, die folgenden Punkte werden für eine Modulgruppe, die in sich schon gelinkt wurde (5.3 “Modulgruppen” auf Seite 23, Möglichkeit 1.) nur einmal durchgeführt.

- Die Symboltabelle in `parseResults` wird wieder von Anfang an nach externen, also zu importierenden, Symbolen durchsucht, diese hat vorher der Parser mit dem Typ `MODSYMBOLTYPEEXTERN` bezeichnet. Das Namenspräfix muss mit einem Modulnamen in der Importliste übereinstimmen, dann wird die Speicheradresse der Funktion oder Variable eingetragen — und zwar direkt in der von `parseResults` zugänglich gemachten Datenstruktur, mit der aktuellen ELF-Version also in der von Transfer gelieferten RAM-Kopie des ‘object file’.
- Der Relozierer wird aufgerufen. Er erhält alle benötigten Daten über `parseResults`, da bei den vorherigen Schritten jeweils die Resultate dort eingetragen wurden. Anschliessend werden die Speicherregionen für Code und Nur-Lese-Daten des neuen Moduls mit dem bestehenden Systemaufruf `vmProtect` vor Schreibzugriffen geschützt. Da `vmProtect` in Topsy 1.1 (diese Version diente als Grundlage dieser Arbeit) zwar vorbereitet, aber nicht implementiert ist, bewirkt das im Moment nichts.

Falls einer der vorangegangenen Punkte nicht ausgeführt werden konnte, folgt ein Fehlerbehandlungs- und Aufräumabschnitt. Wurde das neue Modul bereits in die Modulliste eingetragen, wird es daraus entfernt und die Referenzzähler der importierten Module werden wieder erniedrigt. Alle bisher vom Linker allozierten Ressourcen werden freigegeben. Der Parser wird aufgerufen, um die von ihm aufgebauten Strukturen freizugeben. Rückgabewert der Funktion ist einer der im Abschnitt 4.1 “Systemaufrufe” auf Seite 18 aufgeführten Fehlercodes zu `modLoad`.

Nach erfolgreichem Laden wird ebenfalls der Parser aufgerufen, um die von ihm aufgebauten Strukturen wieder freizugeben. Die vom Linker erstellten Strukturen bleiben in diesem Fall erhalten, da sie in die Listen der Modulverwaltung eingebunden worden sind. Die Funktion wird mit der Rückgabe des Wertes `MOD_LOADOK` beendet.

6.3.2 Freigeben/Entladen

Die Funktion `modUnlinkFree` ist relativ einfach aufgebaut. Als erstes muss die Modul-Identifikationsnummer in der Liste gesucht werden, dann wird kontrolliert, ob Importreferenzzähler und Benutzerzähler (siehe 2.3 “Freigeben (Entladen) eines Moduls” auf Seite 10) auf Null stehen.

Ist dies erfüllt oder wird der Benutzerzähler bewusst umgangen, so wird der entsprechende Moduldeskriptor aus der Liste der geladenen Module entfernt und die Importreferenzzähler der vorher

importierten Module werden je um 1 dekrementiert. Anschliessend werden die Speicherregionen des Moduls sowie der Heapspeicher der nicht mehr benötigten Listenelemente und Deskriptoren freigegeben.

6.4 Parser

Das Teilmodul ModParseObj enthält die vom Modulformat, nicht aber von der Hardware abhängigen Teile und ist deshalb ins Unterverzeichnis `elf` eingeordnet. Die Funktion `modParseObj(Address imageAddr, ModObjectImageHeader* parseResults)` erzeugt die in Abb. 4 links

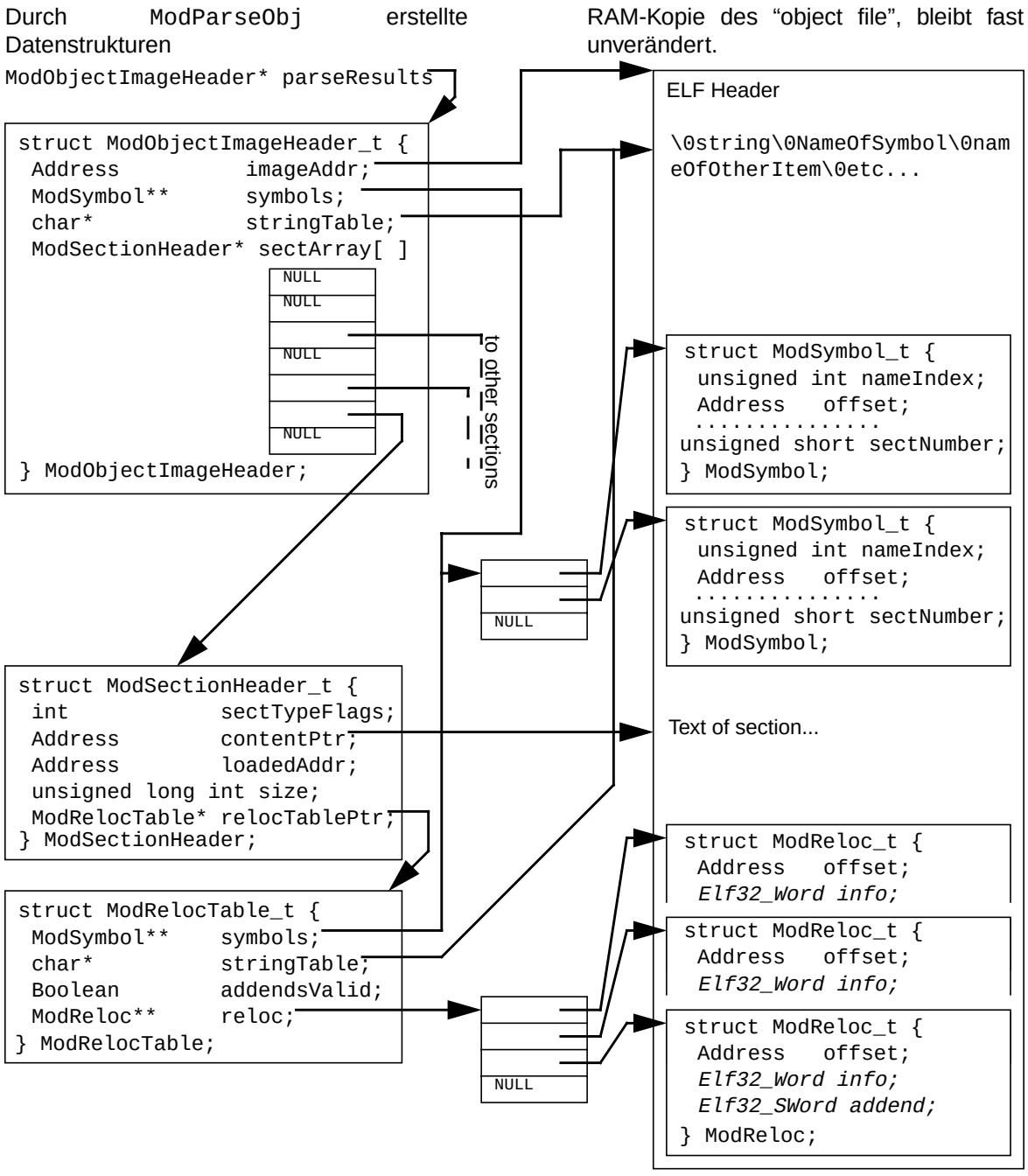

Abb. 4. Strukturen für den Linker und den Relozierer mit Pointern auf ein Modul im ELF-Format

dargestellten Strukturen und gibt einen der in `ModParseObj.h` definierten Codes (`MOD_PARSE...`) vom Typ `Error` zurück. Der Speicher für `parseResults` muss bereitgestellt sein, aller weitere benötigte Speicher wird automatisch alloziert. Die für Linker und Relozierer verständlichen Datenstrukturen sind den ursprünglichen Strukturen im ELF-Format nachempfunden, aber möglichst so, dass eine Verallgemeinerung möglich sein sollte.

Der Parser belässt die einzelnen Symbolbeschreibungen, die Stringtabelle und die einzelnen Relokationsanweisungen im ursprünglichen ELF-Format an ihrer Stelle im Speicher, allerdings werden topsy-spezifische `struct`-Typen darauf "ausgelegt". Die Tabellen werden hingegen neu erstellt und mit Pointern auf die einzelnen Elemente versehen.

Im Header (Typ `ModObjectImageHeader`) steckt ein Array mit fixer Grösse, wo Pointer zu den einzelnen Sektionsbeschreibungen eingetragen werden. Das Array kann die maximale Anzahl zu erwartender Sektionen plus etwas Reserve aufnehmen (`MAXNBOFSECTIONS` definiert in `ModInterface.h`). Der Index im Array entspricht der Sektionsnummer. Bei zum Laden nicht benötigten und bei nicht vorhandenen Sektionen wird ein NULL-Pointer eingesetzt, NULL heisst aber nicht unbedingt "Schluss der Einträge"! Im Gegensatz dazu gibt es pro zu ladender Sektion je ein oder zwei Arrays mit Pointern auf die einzelnen Einträge der Relokations- und Symboltabellen. Diese Arrays werden nach Bedarf angelegt, haben keine fixe Grösse, und sind je durch einen NULL-Pointer abgeschlossen.

Die ursprünglichen ELF-Typen der Modulstrukturen sind zusammen mit Konstanten und Makros in `ELF_Decl.h` definiert. Die meisten Definitionen wurden exakt aus [6] entnommen.

Die Topsy-Typen der Parser-Datenstrukturen sind in `ModInterface.h` definiert und dürfen in der Grösse und der Anordnung der einzelnen `struct`-Elemente verändert werden, um eine Anpassung an die "natürlichen" Strukturen eines andern Modulformates zu erreichen. Die Namen der einzelnen Elemente (und hier nicht gezeigten Konstanten und Makros) sind aber durch Topsy vorgegeben, ausgenommen die in Abb. 4 kursiv gedruckten. Jene sind interne, modulformatspezifische Namen, auf die Linker und Relozierer nicht direkt zugreifen dürfen. Zugänglich sind sie, falls überhaupt von Interesse, durch ebenfalls in `ModInterface.h` definierte Makros mit vorgegebenen Namen.

Ist es bei der Portierung nicht möglich, die Strukturen eines andern Modulformats geeignet in die topsy-kompatiblen Strukturen zu integrieren, müssen mehr Arrays neu angelegt und die benötigten Daten hineinkopiert werden.

Eine Spezialität von ELF: Falls, wie in unserem Beispiel angenommen, die Relokationseinträge ohne "Addends" (siehe 6.5 "Relokation") gespeichert sind, so existiert das Element `ModReloc.addend` gar nicht und wird vom nächsten Relokationseintrag überschrieben. Dies erkennt der Parser am ELF-Sektionstyp `REL` (statt `RELA`), worauf er `ModRelocTable.addendsValid == FALSE` setzt. Damit wird dem Relozierer mitgeteilt, dass er nicht auf `ModReloc.addend` zugreifen darf. Falls es sich um eine Relokationstabelle mit "Addends" handelt (ELF-Sektionstyp `RELA`), so wird der Zugriff auf diese durch das Makro `MODRELOCADDEND( )`ermöglicht. Bei Dateiformaten, die "Addends" entweder niemals oder immer verwenden, wird der entsprechende Parser alle Variablen `addendsValid` auf einen fixen Wert setzen.

6.5 Relokation

Das Teilmodul ModReloc erledigt die prozessorabhängigen Aufgaben und steckt deshalb im Unterverzeichnis `mips`. Die Funktion `relocateSection(ModObjectImageHeader* header, int sectionNo)` erledigt die Anpassung aller Sprung- und Datenzugriffsadressen aufgrund der vom

Linker vorbereiteten Angaben und den im ‘object file’ vorhandenen, vom Parser zugänglich gemachten Relokationstabellen und gibt einen der in `ModReloc.h` definierten Codes (`MOD_RELOC...`) vom Typ `Error` zurück. Sowohl Code- wie auch ein Teil der Datensektionen müssen bearbeitet werden, bei den Daten stehen z. B. Initialisierungswerte für Pointer. Die Adresskorrekturen werden direkt am endgültigen Speicherplatz der einzelnen Sektionen vorgenommen.

Nach einer Kontrolle, ob die übergebene Sektion überhaupt eine Relokationstabelle hat, werden die Startadressen sowie der Tabellentyp — mit oder ohne “Addends” — bestimmt.

Anschliessend wird jeder Eintrag der Relokationstabelle durchgearbeitet. Darin enthalten ist der Offset des zu korrigierenden Objektes innerhalb der Sektion, der Relokationstyp (siehe 6.5.1), die Nummer des zu adressierenden Symbols in der zugehörigen Symboltabelle und — optional — eine vorläufige Zieladresse, genannt “Addend”. Fehlt letzteres, so steht die vorläufige Zieladresse bereits an der zu korrigierenden Speicherstelle und muss dort gelesen werden.

Der zugeordnete Eintrag in der Symboltabelle enthält bei einem externen Objekt dessen tatsächliche Speicheradresse, die vorläufige Zieladresse sollte Null sein. Bei einem Objekt des eigenen Moduls wird auf einen Symboleintrag mit der Sektionsnummer, wozu das zu adressierende Objekt gehört, verwiesen; anhand der Sektionsnummer kann die tatsächliche Startadresse dieser Sektion gefunden werden. Die endgültige Zieladresse ergibt sich aus der Summe von vorläufiger Zieladresse, Offset-Eintrag in der Symboltabelle und der Startadresse der Sektion, in der sich das zu adressierende Objekt befindet. Wie diese Zieladresse in den Code oder die Daten eingesetzt wird, ist vom Relokationstyp abhängig.

Die ebenfalls dem Relozierer zugeordnete Funktion `alignSectionAddress` wird vom Linker einige Male aufgerufen, um eine Adresse der dem Prozessor entsprechenden Alignierung anzupassen (in diesem Fall 4 Byte).

6.5.1 Relokationstypen

Die verschiedenen Arten von Adresskorrekturen werden im ELF-Format mit Nummern bezeichnet, ein anderes Format kann natürlich abweichende Kennzeichnungen verwenden. Die genaue Behandlung ist vom Prozessortyp abhängig. Um die saubere Trennung von modulformat- und prozessorabhängigen Teilmodulen zu erhalten, wurde noch ein Übersetzungsmodul, `Obj2ArchTranslate` genannt und ins Unterverzeichnis `mipself` eingeordnet, dazwischengeschaltet. Für jeden vorkommenden Typ ist dort eine Konstante definiert, die Funktion `translateRelocTypes` nimmt evtl. erforderliche Umsetzungen vor.

Bei ELF-Dateien für MIPS R3000 mit den in 5.1 “Das ‘object file’ zum Laden” auf Seite 21 erwähnten Einschränkungen — z. B. bezüglich `$gp` — kommen folgende Relokationstypen vor:

Typ 2, Topsy-Bezeichnung `RELTYPEWHOLEWORD`. Die Adresse wird als ein vollständiges Datenwort (4 Byte, also 32 Bit) eingesetzt, kommt in Datensektionen vor.

Typ 4, Topsy-Bezeichnung `RELTYPEJUMP`. Für absolut adressierte Sprünge, die um 2 Bit nach rechts geschobenen (also durch 4 dividierte) Adresse wird in die 26 niederwertigsten Bit des Befehlswortes (insgesamt 32 Bit) eingetragen.

Typ 5, Topsy-Bezeichnung `RELTYPEIMMHIGH16`. Die “hohen” 16 Bit der Adresse werden als “Immediate”-Wert an Stelle der “tiefen” 16 Bit ins Befehlswortes eingesetzt.

Typ 6, Topsy-Bezeichnung `RELTYPEIMMLOW16`. Die “tiefen” 16 Bit der Adresse werden als “Immediate”-Wert an Stelle der “tiefen” 16 Bit ins Befehlswortes eingesetzt.

Der Offset-Eintrag in der Symboltabelle und — falls benötigt — die Startadresse der Sektion, in der sich das zu adressierende Objekt befindet, werden addiert; bei den Relokationstypen 4 und 5 werden die binären Werte um 4 bzw. 16 Bit verschoben. Dazu addiert der Relozierer die in Abschnitt 6.5 erwähnte vorläufige Zieladresse. Diese liest er aus dem “Addend” oder er extrahiert sie mit der typabhängigen Bit-Maske (ganzes Wort, 26 “tiefe” Bit oder 16 “tiefe” Bit) aus dem anzupassenden Daten- oder Befehlswort. Das Resultat ersetzt die entsprechenden Bit gemäss obiger Auflistung.

Kapitel 7: Stand der Arbeiten

Ein Konzept, wie Topsy mit dynamisch ladbaren Modulen erweitert werden kann, wurde entwickelt. Die Schnittstellen zwischen den einzelnen Software-Modulen sind weitgehend festgelegt, für die Schnittstelle zum Benutzer sowie die Datenschnittstelle zur Umgebung besteht ein Vorschlag. Die Software wurde soweit implementiert, dass sie als Demoversion betriebsfähig ist, allerdings läuft sie noch nicht so stabil, wie dies Teile des Kernels eigentlich müssten.

7.1 Software-Implementation

Die folgende Tabelle zeigt die Übersicht der Programmteile.

	programmiert	**getestet**
ModMain (Grundfkt. inkl. Modulverwaltung)	ja **(*)**	ja **(*)**
ModMain (Kommando-Verwaltung, Infos)	nein	—
Linker (Grundfkt. inkl. Modulverwaltung)	ja	teilweise
Linker (Kommandos einlesen, Modulgruppen)	nein	—
Parser	ja	teilweise
Relozierer	ja	ja
Syscall-Erweiterungen	ja	teils, entspr. ModMain
Shell-Anpassungen	teils, entspr. ModMain	ja
Transfer (für Testumgebung) (s. Anh. B.1)	ja	ja
Transfer (für Übungsumgebung) (s. Anh. B.2)	nein	—
“Filereader” für Testumgebung	ja	ja
Laden des User-Default-Teiles	nein	—

Tab. 1. Übersicht: Stand der Software-Implementation

(*) Es fehlt noch die Kontrolle, ob Userprogramme bei den Systemaufrufen nicht unzulässig auf den Kernelspace verweisen.

Die dynamische Modulverwaltung besteht erst für Usermodule, nicht für Kernelmodule.

“Teilweise getestet” heisst, dass Fehler bekannt sind, deren Ursachen aber nicht genau untersucht wurden. Eine Liste von bekannten Fehlern:

- Die Moduldeskriptoren in der Verwaltungsliste werden in gewissen Situationen mit unsinnigen Daten überschrieben. Kritisch ist hier die Freigabe der teilweise erstellten Strukturen nach einem Ladeabbruch wegen eines Fehlers sowie das Entladen von Modulen. Der Fehler tritt meistens erst beim nächsten Laden auf, vorher scheint die angezeigte Modulliste noch intakt.
- Der Parser kann abstürzen, wenn er mit einer nicht konformen Datei gefüttert wird. Kritisch sind Dateien, deren Header eine ELF-Datei vorspiegeln, obwohl die Fortsetzung nicht korrekt ist.

Die beim Start vorhandenen Usermodule sind weiterhin statisch gelinkt (3.3.3 b “Änderungen im Zusammenhang mit User-Default” auf Seite 16 nicht ausgeführt). Um die Implementation möglichst einfach zu halten, wurde von Links zwischen statischen und dynamischen Teilen abgesehen, die

“Bibliotheksmodule” Syscall und gegebenenfalls UserSupport müssen für die dynamischen Module ein zweites Mal geladen werden.

Die Überwachung, wieviele Threads in welchem Modul aktiv sind, ist noch nicht implementiert (3.3.3 c “Änderungen im Zusammenhang mit Thread-Überwachung” auf Seite 16 nicht ausgeführt).

Da die Shellkommando-Verwaltung noch nicht in Betrieb ist, wurde die Shell provisorisch mit der Möglichkeit ausgerüstet, Threads mit beliebiger (Userspace-)Startadresse zu starten. Diese Funktion ist vorsichtig zu benutzen.

7.2 Dokumentation

Da die Software noch nicht einsatzbereit ist, sind bis zu einer definitiven Implementation noch einige Änderungen zu erwarten. Deshalb habe ich in Absprache mit dem Betreuer auf das geforderte Kapitel zum Topsy-Handbuch verzichtet.

Die bearbeiteten Sourcen inklusive Test-Anwendungen sind in elektronischer Form verfügbar (Anhang D “Inhalt des elektronischen Archivs” auf Seite 58). Details zur Funktionsweise sind in den Sourcen kommentiert.

Kapitel 8: Ausblick

Folgende Arbeiten könnten an die vorliegenden Resultate anknüpfen.

8.1 Überarbeitung

Falls der dynamische Lader/Linker in den praktischen Einsatz kommen soll, ist sicher eine Überarbeitung und ein systematisches Austesten der Software notwendig, um bekannte und unbekannte Fehler und Unterlassungen zu beheben. Ich hoffe, dass der vorliegende Bericht dabei ein Hilfe sein wird.

Neben den in 7.1 “Software-Implementation” auf Seite 32 erwähnten Unvollständigkeiten können folgende Punkte ergänzt werden:

- Dynamischer Lader optional, Entscheidung zur Compile-Zeit durch eine in `Configuration.h` definierte Konstante (`#define DYNAMICUSER`).
- Die hardwarenahen Teile sollten sowohl für die Big-Endian- wie auch für die Little-Endian-Version des MIPS-Prozessors kompilierbar sein, Entscheidung anhand einer bereits in `Configuration.h` definierten Konstante (`#define TOPSY_BIG_ENDIAN`).

In den Source-Codes sind viele dem Debugging dienende Ausgabebefehle eingefügt (`ioConsolePut...`, `INFO`, `WARNING`). Die einzeln am linken Rand stehenden Befehle können meistens ohne Weiteres entfernt werden; die dem Kontrollfluss entsprechend eingerückten Anweisungen sind vorgesehen, um vorläufig drin zu bleiben (z. B. provisorische Ausgabe der Modulliste). Die für den Benutzer wichtigen Informationen werden von der Shell angezeigt, im Falle von Fehlern teils auch durch “Warnings” direkt vom Kernel.

8.2 Anpassung für die Übungen Technische Informatik II

Die Aufteilung der Userprogramme in einzeln ladbare Module kann den Arbeitsablauf beim Austesten von Übungsprogrammen erleichtern.

Um auf der Übungshardware (siehe Anhang B.2 “Das MIPS-IDT-Board” auf Seite 39) den Einsatz zu ermöglichen, ist das Usermodul Transfer an die dort vorhandene Schnittstelle anzupassen. Zusätzlich muss auf dem Hostsystem die Möglichkeit eingerichtet werden, Module zu “senden”.

Die Ergänzung des Topsy-Handbuches ist noch ausstehend, ebenso natürlich eine studentenfreundliche Anleitung zur Handhabung der Übungshardware.

8.3 Dynamische Kernel-Module

Die Verwaltung von dynamischen Kernelmodulen ist schon vorbereitet. Allerdings ist völlig offen, woher der Anstoss zum Laden eines Kernelmodules kommen soll, wozu es verwendet wird, und wo sein Code bereitgestellt wird. Falls der Code aus einer externen Quelle oder aus dem Userspace stammt, muss er geeignet verifiziert werden.

Eine Anwendungsidee sind Gerätetreiber, bei denen während dem Aufstarten oder gar zur Laufzeit entschieden wird, welche zu laden sind.

Anhang A: Aufgabenstellung

Semesterarbeit für Herrn Thomas Bretscher

Aufgabenstellung: George Fankhauser, Thomas Bretscher
Thema: **Dynamischer Linker und Lader für Topsy**
Beginn der Arbeit: 19.10.1998
Abgabetermin: 6.2. 1999
Betreuung: George Fankhauser, Marcel Waldvogel
Arbeitsplatz: ETZ C96
Umgebung: Sun/Solaris, MipsSim, Gnu-Tools, Topsy V 1.1 Source

A.1 Einleitung

Topsy ist ein portables micro-kernel Betriebssystem, das am TIK für den Unterricht entworfen wurde. In der ersten Version wurde es für die Familie der 32-bit MIPS Prozessoren gebaut. Es zeichnet sich durch eine saubere Struktur, eine hohe Portabilität (Trennung des Systems in hardware-abhängige und -unabhängige Module) und eine gute Dokumentation [1] aus.

Weitere Dokumentation über Topsy ist unter [2] verfügbar.

A.2 Aufgabenstellung

A.2.1 Ziele

a). Zusätzliche oder neue Versionen von bestehenden Userprogrammen sollen auf das MIPS-Board geladen und ausgeführt werden, ohne Topsy neu aufzustarten (Anwendung in Übungen TI 2)

b). Dynamische Konfiguration (ohne neu kompilieren) des Topsy-Kernels, indem nur die gewünschten/ benötigten Treiber und Kernelmodule geladen werden

Falls sich keine besonders grossen Probleme dagegenstellen, sollen a) und b) durch denselben Mechanismus erreicht werden.

Es soll eine klare Trennung zwischen Linker/Lader-Modul und den Mechanismen zum Transfer der Daten (serielle Schnittstelle, Diskblöcke, ROM Images, etc.) erreicht werden.

A.2.2 Vorgehen

Um ein dynamisches Link/Lade-Modul in Topsy zu integrieren, sind sowohl Änderungen am Betriebssystem wie auch an der Entwicklungsumgebung nötig

- **1.** Topsy
  - **1.1** Definition von “dynamischen Modulen”.
    Soweit möglich soll ein Standard-Format des GNU C-Compilers übernommen werden.
  - **1.2** Programme

- **1.2.1** Lader, lädt Modul von beliebiger Speicheradresse in geeigneten Speicherbereich im Userspace oder im Kernelspace und reserviert Speicher für Daten.
  Dazu soll ein Modul-Image Deskriptor definiert werden
- **1.2.2** Linker, passt Sprungadressen und Datenzugriff innerhalb des neuen Moduls an und ermöglicht Prozeduraufrufe im neuen Modul von aussen sowie System-Calls des neuen Moduls.
  Hier ist insbesondere die Verwaltung der Symbole und Import/Export-Tabellen zu beachten.
- **1.2.3** Verwaltung der zur Zeit geladenen Module mit Möglichkeit, Module zu löschen. Hier soll ein (in-core) Modul-Deskriptor definiert werden.

Um das Ganze im Sinne von “a)” vom Userspace aus zu nutzen, braucht es noch:

- **1.2.4** Modifizierte Shell, die Aufrufe von Prozeduren in dynamischen Modulen unterstützt
- **1.2.5** **(*)** Interface zur SUN, um Module in den MIPS-Speicher herunterzuladen

Für dynamisch ladbare Kernelmodule “b)”wird im Weiteren benötigt:

- **1.2.6** Auswahlprogramm für zu ladende Treiber beim Aufstarten und das Lesen von einem “bootdevice”. Diese Anwendung ist speziell für die PC-Version interessant und soll nicht in dieser Arbeit implementiert werden (jedoch im Design berücksichtigt werden).

Der im aktuellen Topsy vorhandene Mechanismus zum Umkopieren Kernel -> User beim Aufstarten (in `mmInit`) soll ersetzt werden durch das automatische Laden eines oder mehrerer Defaultmodule (Inhalt: Shell mit Syscall-Library und evtl. Zubehör).

Alle Programmteile sollen übersichtlich und auf Portabilität ausgerichtet sein. Insbesondere sollen 2 Interfaces definiert werden für:

- Code- und Daten-Fixup für HAL-Teil des Linker/Laders; es soll der MIPS-HAL implementiert werden
- Einen Lademechanismus, der ein Objektfile in den Speicher von Topsy transferiert

- **2.** Entwicklungsumgebung

  Für Userspace-Module werden folgende Komponenten benötigt:

  - **2.1** “Tool” (Makefile) für Kompilierung von einzelnen Modulen sowie zur Einbindung der nicht gelinkten Default-Usermodule in die topsy.ecoff-Datei
  - **2.2** **(*)** Interface zum MIPS, um Module hinunterzuladen (z. B. via gdb/serielle Schnittstelle)

  Für Kernel-Module:

  - **2.3** “Tool” (Makefile) für Einbindung der noch nicht gelinkten Treiber in die topsy.ecoff-Datei

- **3.** Dokumentation (zusätzlich zum Arbeitsbericht)
  - **3.1** Ergänzung Topsy-Handbuch
  - **3.2** **(*)** Anleitung zur Handhabung von Topsy auf dem SUN-MIPS-Verbund

(*) Da die Programmier- und Testarbeiten auf einer simulierten Umgebung durchgeführt werden können, kann in einem ersten Schritt auf das Interface MIPS—Sun verzichtet werden (Punkte 1.2.5, 2.2, 3.2).

Dessen Realisierung ist optional bzw. kann in Zusammenarbeit mit anderen Arbeiten erfolgen. Anstelle der seriellen Kommunikation kann mit einem einfachen Filezugriff ein Objektfile geladen werden.

A.3 Bemerkungen

- Mit dem Betreuer sind wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen soll der Student mündlich über den Fortgang der Arbeit berichten und anstehende Probleme diskutieren.
- Am Ende der zweiten Woche ist ein Zeitplan für den Ablauf der Arbeit sowie eine schriftliche Spezifikation der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Am Ende des zweiten Monats der Arbeit soll ein kurzer schriftlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt.
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden.
- Die Dokumentation ist mit dem Textverarbeitungsprogramm “FrameMaker” zu erstellen.

A.4 Ergebnisse der Arbeit

Neben einem mündlichen Vortrag von 20 Minuten Dauer im Rahmen des Fachseminars Kommunikationssysteme sind die folgenden schriftlichen Unterlagen abzugeben:

- Ein kurzer Bericht. Dieser enthält eine Darstellung der Problematik, eine Beschreibung der untersuchten Entwurfsalternativen, eine Begründung für die getroffenen Entwurfsentscheidungen, sowie eine Auflistung der gelösten und ungelösten Probleme. Eine kritische Würdigung der gestellten Aufgabe und des vereinbarten Zeitplanes rundet den Bericht ab (in vierfacher Ausführung). Der Bericht soll zudem ein neues Kapitel zum Lader/Linker im Topsy Manual [1] enthalten. Es ist in Englisch zu schreiben.
- Ein Handbuch zum fertigen System bestehend aus Systemübersicht, Implementationsbeschreibung, Beschreibung der Programm- und Datenstrukturen sowie Hinweise zur Portierung der Programme.
- Eine Sammlung aller zum System gehörenden Programme.
- Die vorhandenen Testunterlagen und -programme.
- Eine englischsprachige Zusammenfassung von 1 bis 2 Seiten, die einem Aussenstehenden einen schnellen Überblick über die Arbeit gestattet. Die Zusammenfassung ist wie folgt zu gliedern: (1) Introduction, (2) Aims & Goals, (3) Results, (4) Further Work.

Anhang B: Beschreibung der Entwicklungsumgebung

Die Entwicklung und die Tests des Topsy-Systems wurden auf einer simulierten Maschine durchgeführt. Für die Übungen zu Technische Informatik II wird eine “echte” Maschine verwendet, die unter B.2 “Das MIPS-IDT-Board” auf Seite 39 kurz beschrieben ist. Die in dieser Arbeit in Angriff genommenen Erweiterungen sollen später auch auf jener verwendet werden können.

Als Hostsystem diente jeweils eine Sun-Workstation mit UNIX (SunOS) und SPARC-Prozessor.

B.1 Die Simulierte Maschine

B.1.1 Der Simulator

Da die Handhabung des Experimentalboards (siehe B.2 “Das MIPS-IDT-Board”) zum Teil etwas umständlich und langsam ist, schrieb George Fankhauser einen Simulator eines MIPS R3000-Prozessors. Der Simulator ist in Java programmiert und somit prinzipiell auf allen Plattformen lauffähig [3].

Integriert ist ein simuliertes ROM mit einem Bootloader analog zum Experimentalboard, mit dem Unterschied, dass die Programmdaten aus einer wählbaren Datei des Hostsystems anstelle der seriellen Schnittstelle bezogen werden. Aufgestartet wird mit einer Datei im Motorola-SREC-Format. Der zweiten seriellen Schnittstelle des Boards entspricht die Ein- und Ausgabe auf das Terminal (Console). Der Simulator kann während der Ausführung Ausgaben für ein Log generieren, z. B. bei allen Systemaufrufen und Unterbrechungen oder gar bei jeder Instruktion.

B.1.2 Dateitransfer zur simulierten Maschine

Ohne Rücksicht auf die spätere Realisierung auf dem Experimentalboard wurde im Rahmen dieser Arbeit eine weitere Schnittstelle zwischen simuliertem Prozessor und dem Hostsystem implementiert. Die neue Klasse `FileReader` des Simulators verhält sich ähnlich wie eine serielle Schnittstelle nur für Input und liefert Byte für Byte den Inhalt der Hostsystem-Datei namens `/tmp/MipsSimulatorINFILE`. Ein Statusregister zeigt das Dateiende an, über ein Kommandoregister kann der Leser an den Dateianfang zurückgesetzt werden.

Vorhandener Fehler: am Ende der Datei wird ein zusätzliches ungültiges Zeichen angefügt (stört bei ELF-Dateien nicht).

Topsy-seitig greift ein neuer IO-Treiber — ebenfalls `FileReader` genannt — auf die Daten zu. Dieser wurde klassisch in die Treiberstruktur des Kernels integriert.

B.1.3 Software-Tools

Die ganze Kompilation und Teile des Linkens laufen auf dem Hostsystem ab. Dazu verwendet werden:

Für die konventionellen (statisch gelinkten) Teile von Topsy:

- GNU C Compiler, GNU Link Editor und GNU Binary Utilities, Crossversion für MIPS-Code, Format ECOFF. Diese Tools werden bereits für konventionelles Topsy verwendet.
- Topsy-spezifischer Bootlinker, der die Vereinigung von Kernel- und Userteil vorbereitet. Das Javaprogramm wird mit den Topsy-Sourcen geliefert.

- Makefiles und GNU Make. Die Makefiles — ebenfalls Topsy-Bestandteil — mussten an die neue Version angepasst werden.

Für die neuen dynamischen Usermodule:

- GNU C Compiler, GNU Link Editor und GNU Binary Utilities, Crossversion für MIPS-Code, Format ELF. Dazu ein selbst geschriebenes Tool (`analyELF`) zur Anzeige der Sektions- und Symboltabellen von ELF-Dateien auf dem Hostsystem.
- GNU Make und selbst geschriebene Makefiles.

Es ist prinzipiell möglich, für alle Topsy-Module dasselbe Format (ELF) zu verwenden, um die Anzahl verschiedener Tools zu reduzieren. Dazu müssen die Makefiles entsprechend angepasst werden.

Ebenfalls zur Verwendung vorgesehen war ein Debugger mit SimOS-Erweiterung (Entwickelt zur Arbeit mit SimOS von der Stanford University, siehe [4]), zu dem der Mipssimulator eine passende Schnittstelle zur Verfügung stellen sollte. Wegen Problemen beim Zusammenspiel kam er allerdings kaum zum Einsatz.

B.2 Das MIPS-IDT-Board

In dieser Arbeit wurde es zwar nicht verwendet, aber es ist die eigentliche “Heimat” von Topsy: das Experimentalboard mit einem Prozessor vom Typ MIPS R3052 der Firma IDT.

Das Board ist mit einer Workstation über zwei serielle Schnittstellen verbunden, eine für Programmdaten und Debugging, eine für die Verbindung zum Terminal (Console), das in einem Fenster auf der Bedienungsoberfläche der Workstation dargestellt wird. Als weitere Ein- und Ausgabemöglichkeiten sind einige Leuchtdioden sowie drei über ein FPGA beliebig ansteuerbare Kabelanschlüsse vorhanden.

B.3 Bedienungsanleitung zur aktuellen Version

Die folgende Kurzanleitung stellt das Arbeiten auf einer UNIX-Workstation mit MipsSimulator dar (B.1). Vorausgesetzt wird, dass alle benötigten Tools installiert sind.

B.3.1 Auf dem Hostsystem

Kompilieren der statischen Teile: Im Hauptverzeichnis von Topsy `make` aufrufen (bearbeitet auch die statischen Usermodule im Verzeichnis `User`).

Kompilieren der dynamischen Usermodule: Im Hauptverzeichnis `make dyn` aufrufen oder im Verzeichnis `DynamicUser` das eigene Makefile mittels `make` aufrufen.

Starten von Topsy: Mittels geeignetem Java-Interpreter oder -Compiler den Mipssimulator starten (Klasse `Simulator`, Kommandozeilen-Option `-b`, als Parameter Pfad und Dateiname der Datei `topsy.srec` im Topsy-Verzeichnis angeben). Topsy sollte sich mit der Bootmessage, einigen weiteren Ausgaben und dem Prompt melden.

Bereitstellen von Modulen zum Laden: In einem eigenen Terminal die gewünschte Moduldatei (“`.o`”) aus dem Verzeichnis `DynamicUser/mipself.obj/` zur Schnittstelle kopieren (in der aktuellen Version die Datei `/tmp/MipsSimulatorINFILE`).

Aufräumen: `make clean` im Topsy-Hauptverzeichnis löscht kompilierten Code und temporäre Backup-Dateien des statischen und des dynamischen Teils. `make cleandynuser` "reinigt" nur den dynamischen Teil.

B.3.2 Bedienung der Topsy-Shell

Die meisten Fehlermeldungen, die die Shell ausgibt, enthalten die von Syscall gelieferte Fehlernummer. Ihre Bedeutung muss vorläufig noch "von Hand" in der Datei `Topsy/Messages.h` im `typedef enum {....} MessageError` nachgeschaut werden (mit Null beginnend durchnummeriert, einige Werte mittels Kommentaren in der Source bezeichnet).

Laden eines dynamischen Moduls: Zuerst auf dem Hostsystem die Datei bereitstellen, dann Befehl `load` der Topsy-Shell ohne Parameter aufrufen. Einige (viele) Ausgaben über die laufenden Vorgänge kommen direkt aus dem Modulmanager, die Shell meldet am Schluss den Namen des erfolgreich geladenen Moduls. Falls Fehler auftreten wird eine Nummer ausgegeben (siehe oben).

Informationen über geladene Module abrufen: Shell-Befehl `mod` ruft die provisorische Modullisten-Ausgabe direkt vom Modulmanager auf. Wichtig sind die Modulnummer und die exportierten Objekte, von denen neben dem Typ (zur Zeit immer 0) die Speicheradresse hexadezimal und dezimal angezeigt wird.

Starten einer Kommandoprozedur: Zur Zeit werden die Kommandoprozeduren noch nicht verwaltet, der Benutzer muss selber wissen, welche Objekte aufrufbar sind. Die in der provisorischen Modulliste angezeigte dezimale Adresse (z. B. *84544*) kann mit dem Shell-Befehl `start` *84544* aufgerufen werden. Vorsicht, keine eingebaute Sicherung gegen ungültige Adressen!

Die konventionell statisch gelinkten Userprogramme (deren Startbefehle vom Shell-Befehl `info` oder `help` aufgelistet werden) werden weiterhin mit `start` *`befehl`* aufgerufen.

Freigeben eines dynamischen Moduls: Shell-Befehl `free` mit Modul-ID als Parameter aufrufen.

Anhang C: Neue und geänderte Header-Dateien

C.1 Gegenüber Topsy 1.1 geänderte Header-Dateien

Die Auflistung aller geänderten oder für weitere Verbesserungen zu ändernden "`.c`" und "`.h`"-Dateien mit ungefährer Beschreibung ist im Bericht bereits enthalten (siehe 3.3.3 "Änderungen in bestehenden Topsy-Modulen" auf Seite 15). Nur Teil a) wurde ausgeführt.

C.1.1 Kernel

Die Datei `Configuration.h` enthält allgemeine Definitionen zur Topsy-Konfiguration, ihr ist keine bestimmte Programmdatei zugeordnet.

```
/*
     File:                  Topsy/Configuration.h
     Author(s):             George Fankhauser / Thomas Bretscher
     Affiliation:           ETH Zuerich, TIK
     Version:               DynamicLoader
     Creation Date:
     Last Date of Change:   1999/02/22      by: Thomas Bretscher
*/
#ifndef _CONFIGURATION_H_
#define _CONFIGURATION_H_

/* general, messages, prompts, look and feel
 */
#define BOOTMESSAGE "Topsy (c) 1996-1999, ETH Zurich, TIK\nDynamic user module loader \n(Thomas
Bretscher)\n\n"

#if defined(__MIPSEB__) || defined(__BIG_ENDIAN__)
#define TOPSY_BIG_ENDIAN
#endif

/* dynamic loader/linker
 */
#define DYNAMICUSER  /* if undefined: no dynamic loader/linker for user space*/

#define MAXDYNAMICNAMESIZE  24 /* max name size for modules, global symbols*/
                               /* incl. '\0' character! */

/* memory parameters
 */
#define KERNELHEAPSIZE   (64*1024) /* kernel heap size */

/* threads, IPC
 */
#define TIMESLICE         10   /* how long a thread may run, in milliseconds */
#define NBPRIORITYLEVELS   3   /* 0 for highest priority */
#define MAXNBMSGINQUEUE    4   /* message queue length */
#define MAXNAMESIZE       24   /* max thread name size (in characters) */
                               /* incl. '\0' character! */
#define TM_DEFAULTTHREADSTACKSIZE   1024

/* io features
 */
#define NBCYCLESFORDELAY 10    /* each time a write occurs onto a driver, */
                               /* at least NULL-operations are performed  */

#endif
```

Die Datei `Topsy.h` enthält allgemeine Definitionen für das ganze Topsy, ihr ist ebenfalls keine bestimmte Programmdatei zugeordnet.

```
/*
     File:                  Topsy/Topsy.h
     Author(s):             Eckart Zitzler
     Affiliation:           ETH Zuerich, TIK
     Version:               DynamicLoader
     Creation Date:
     Last Date of Change:   1999/01/22  by Thomas Bretscher $
*/

#ifndef _TOPSY_H_
#define _TOPSY_H_

#define MMTHREADID  -1     /* Memory Manager Thread Id. */
#define TMTHREADID  -2     /* Thread Manager Thread Id. */
#define MODTHREADID -3     /* Module Manager (dynamic load/link) Thread Id */
#define IOTHREADID  -4     /* Input/Output Manager Thread Id. */

#define NULL (void*) 0

typedef enum {FALSE = 0, TRUE}  Boolean;

typedef void*  Address;

typedef long int Error;

typedef unsigned long int Register;

typedef long int ThreadId;       /* Kernel threads: < 0, User threads: > 1 */
typedef long int ModuleId;

typedef void* ThreadArg;
typedef void(*ThreadMainFunction)(ThreadArg);

#endif
```

Von der folgenden Datei `Messages.h` sind einige im Zusammenhang mit dieser Arbeit nicht interessanten Abschnitte ausgelassen, markiert mit **`/*...  ...*/`**. Auch dieser Headerdatei ist keine Programmdatei zugeordnet.

```
#ifndef __MESSAGES_H
#define __MESSAGES_H
/*
     File:                  Topsy/Messages.h
     Author(s):             Christian Conrad
     Affiliation:           ETH Zuerich, TIK
     Version:               DynamicLoader
     Creation Date:         11.2.97
     Last Date of Change:   1999/02/19    by Thomas Bretscher
*/

#include "Topsy.h"
#include "Memory.h"
#include "Modules.h"
#include "IO.h"
#include "Configuration.h"


#define SELF         0         /* info about one self */
#define ANY          1         /* Receiving from any thread */
```

```
#define INFINITY    -1         /* Infinite timeout when receiving message */


/*
 * Message ids and errors
 */

typedef enum {
  ANYMSGTYPE,

  TM_START, TM_EXIT, TM_YIELD, TM_KILL,
  TM_STARTREPLY, TM_KILLREPLY, TM_INFO, TM_INFOREPLY,      /* thread manager */

  VM_ALLOC, VM_FREE, VM_MOVE, VM_PROTECT,                  /* memory manager */
  VM_ALLOCREPLY, VM_FREEREPLY, VM_MOVEREPLY, VM_PROTECTREPLY,
  VM_CLEANUP,

  MOD_LOAD, MOD_LOADREPLY, MOD_FREE, MOD_FREEREPLY,        /* module manager */
  MOD_GETINFO, MOD_GETINFOREPLY, MOD_GETUSERCOMMANDS, MOD_GETUSERCOMMANDSREPLY,

  IO_OPEN, IO_CLOSE, IO_READ, IO_WRITE, IO_INIT,   /* input/output */
  IO_OPENREPLY, IO_CLOSEREPLY, IO_READREPLY, IO_WRITEREPLY, IO_INITREPLY,
  IO_TIMER_START, IO_TIMER_STOP, IO_TIMERREPLY,

  UNKNOWN_SYSCALL                                       /* unknown syscalls */
} MessageId;

typedef enum {
  TM_MSGSENDOK/* 0*/, TM_MSGSENDFAILED, TM_MSGRECVOK, TM_MSGRECVFAILED,
  TM_STARTOK, TM_STARTFAILED, TM_KILLOK, TM_KILLFAILED,
  TM_INFOOK, TM_INFOFAILED,

  VM_ALLOCOK/*10*/, VM_ALLOCFAILED, VM_FREEOK, VM_FREEFAILED,
  VM_MOVEOK, VM_MOVEFAILED, VM_PROTECTOK, VM_PROTECTFAILED,
  VM_CLEANUPOK, VM_CLEANUPFAILED,

  MOD_LOADOK/*20*/, MOD_LOADCALLFAILED, MOD_LOADNORIGHT, MOD_LOADMULTIPLEMOD,
  MOD_LOADIMPORTMISSING, MOD_LOADOBJERROR/*25*/, MOD_LOADSYMBOLERROR,
  MOD_LOADSYMBOLNOTFOUND,
  MOD_FREEOK, MOD_FREECALLFAILED, MOD_FREENORIGHT/*30*/,
  MOD_FREENOTFOUND, MOD_FREEIMPORTED, MOD_FREEACTIVETHREAD,
  MOD_GETINFOOK, MOD_GETINFOFAILED,
  MOD_GETUSERCOMMANDSOK, MOD_GETUSERCOMMANDSFAILED,

  IO_OPENOK, IO_OPENFAILED, IO_CLOSEOK/*40*/, IO_CLOSEFAILED,
  IO_READOK, IO_READFAILED, IO_WRITEOK, IO_WRITEFAILED,
  IO_INITOK, IO_INITFAILED
} MessageError;

typedef enum { SYSCALL_SEND_OP, SYSCALL_RECV_OP } MsgOpCode;

/* subtype for various infos on threads */
typedef enum { SPECIFIC_ID, GETFIRST, GETNEXT } ThreadInfoKind;

typedef enum { Info_RUNNING, Info_READY, Info_BLOCKED} ThreadInfoStatus;

/* thread structure for users */
typedef struct ThreadInfo_t {
    ThreadId tid;
    ThreadId ptid ;
    ThreadInfoStatus status ;
    char name[MAXNAMESIZE] ;
} ThreadInfo;

/* subtype for module free modes */
```

```
typedef enum { FREENORMAL, FREEOVERR_ACTIVETHREAD,
            FREEOVERR_IMPORTED,  /*  too dangerous */
            FREERECURSIVE /* not (yet) supported */ } ModFreeMode;
/* module structure for users */
typedef struct ModuleInfo_t {
    ModuleId    mid;
    int         referenceCounter;
    int         userCounter;
    char        name[MAXDYNAMICNAMESIZE];
} ModuleInfo;
typedef struct UserCommandsInfo_t {
    Address     addr;
    CommandType commandType;
    char        name[MAXDYNAMICNAMESIZE];
} UserCommandsInfo;

/* Message structures: Threads, Memory */

/*  ...
... */

/*
 * Message structures: Modules
 */
typedef struct ModLoadMsg_t {
    Address        objImagePtr;
    AddressSpace   mode;
    ModuleInfo*    answerArray;
    int            desiredNbOfAnswers;
} ModLoadMsg;
typedef struct ModLoadReplyMsg_t {
    MessageError   errorCode;
    int            nbOfAnswers;
} ModLoadReplyMsg;

typedef struct ModFreeMsg_t {
    ModuleId       mid;
    ModFreeMode    mode;
} ModFreeMsg;
typedef struct ModFreeReplyMsg_t {
    MessageError   errorCode;
} ModFreeReplyMsg;

typedef struct ModGetInfoMsg_t {
  /* (name==NULL)&&(mid=0) means "all", otherwise "specific module" */
    char*          name;
    ModuleId       mid;
    ModuleInfo*    answerArray;
    int            desiredNbOfAnswers;
} ModGetInfoMsg;
typedef struct ModGetInfoReplyMsg_t {
    MessageError   errorCode;
    int            nbOfAnswers;
} ModGetInfoReplyMsg;

typedef struct ModGetUserCommandsMsg_t {
    UserCommandsInfo* answerArray;
    int            desiredNbOfAnswers;
} ModGetUserCommandsMsg;
typedef struct ModGetUserCommandsReplyMsg_t {
    MessageError   errorCode;
    int            nbOfAnswers;
} ModGetUserCommandsReplyMsg;
```

```
/*
 * Message structures: IO
 */

/* ...
... */

/*
 * Message structures: Generic messages
 */
typedef struct SyscallReply_t {
    MessageError errorCode;
} SyscallReply;

typedef struct UserMessage_t {
    void* p1;
    void* p2;
    void* p3;
} UserMessage;

typedef union SpecMsg_u {            /* Union type of all possible messages */
  UserMessage userMsg;

  TMStartMsg tmStart; TMStartReplyMsg tmStartReply;
  TMKillMsg tmKill; TMKillReplyMsg tmKillReply;
  TMGetInfoMsg tmInfo; TMGetInfoReplyMsg tmInfoReply;

  VMAllocMsg vmAlloc; VMAllocReplyMsg vmAllocReply;
  VMFreeMsg vmFree; VMFreeReplyMsg vmFreeReply;
  VMMoveMsg vmMove; VMMoveReplyMsg vmMoveReply;
  VMProtectMsg vmProtect; VMProtectReplyMsg vmProtectReply;
  VMCleanupMsg vmCleanup;

  ModLoadMsg modLoad; ModLoadReplyMsg modLoadReply;
  ModFreeMsg modFree; ModFreeReplyMsg modFreeReply;
  ModGetInfoMsg modGetInfo; ModGetInfoReplyMsg modGetInfoReply;
  ModGetUserCommandsMsg modGetUserCommands;
  ModGetUserCommandsReplyMsg modGetUserCommandsReply;

  IOOpenMsg ioOpen; IOOpenReplyMsg ioOpenReply;
  IOCloseMsg ioClose; IOCloseReplyMsg ioCloseReply;
  IOReadMsg ioRead; IOReadReplyMsg ioReadReply;
  IOWriteMsg ioWrite; IOWriteReplyMsg ioWriteReply;
  IOInitReplyMsg ioInitReply;
  IOTimerMsg ioTimer;
  SyscallReply syscallReply;
} SpecMessage;

typedef struct Message_t {     /* Main message structure */
    ThreadId from;   /* Sender of the message */
    MessageId id;   /* Type of the message */
    SpecMessage msg;   /* Specific message contents */
} Message;


#endif __MESSAGES_H
```

`Syscall.h` ist der Header zu den Programmdateien `Syscall.c` und `SyscallMsg.S`.

```
/*
     File:                  Topsy/Syscall.h
     Author(s):             Eckart Zitzler, Christian Conrad, George Fankhauser;
                             Thomas Bretscher (new mod_ syscalls)
     Affiliation:           ETH Zuerich, TIK
     Version:               DynamicLoader 1.0
     Creation Date:
     Last Date of Change:   1999/02/12  (1999/03/03) by Thomas Bretscher
*/

#ifndef _SYSCALL_H_
#define _SYSCALL_H_

#include "Topsy.h"
#include "Messages.h"
#include "Memory.h"
#include "Threads.h"
#include "IO.h"

#define SyscallError  MessageError

/* memory management */

SyscallError  vmAlloc(Address *addressPtr, unsigned long int  size);
SyscallError  vmFree(Address address);
SyscallError  vmMove(Address *addressPtr, ThreadId newOwner);
SyscallError  vmProtect(Address  startAdr, unsigned long int  size,
                ProtectionMode  pmode);
SyscallError  vmCleanup(ThreadId threadId);

/* thread management */

SyscallError tmMsgSend( ThreadId to, Message *msg);
SyscallError tmMsgRecv( ThreadId* from,
              MessageId msgId,
              Message* msg,
              int timeOut);
SyscallError tmStart( ThreadId* id,
               ThreadMainFunction mainFunction,
               ThreadArg parameter,
               char *name);
SyscallError tmKill(ThreadId id);
void tmExit();
void tmYield();
SyscallError tmGetInfo(ThreadId about, ThreadId* tid, ThreadId* ptid);
SyscallError tmGetFirst(ThreadInfo* info);
SyscallError tmGetNext(ThreadInfo* info);

/* module management (dynamic loader/linker) */

SyscallError modLoad(Address objImagePtr, AddressSpace  mode,
              ModuleInfo answerArray[], int* nbOfAnswers);
SyscallError modFree(ModuleId mid, ModFreeMode mode);
SyscallError modGetInfo(char* name, ModuleId mid, ModuleInfo* answer);
SyscallError modGetInfoList(ModuleInfo answerArray[], int* nbOfAnswers);
SyscallError modGetUserCommands(
                 UserCommandsInfo answerArray[], int* nbOfAnswers);

/* io interface */

SyscallError ioOpen(int deviceNumber, ThreadId* id);
SyscallError ioClose(ThreadId id);
```

```
SyscallError ioRead(ThreadId id, char* buffer, unsigned long int* nOfBytes);
SyscallError ioWrite(ThreadId id, char* buffer, unsigned long int* nOfBytes);
SyscallError ioInit(ThreadId id);

#endif
```

Folgende Datei gehört zum provisorischen Input-Treiber:

```
/*
     File:                 IO/Drivers/FileReader.h
     Author(s):            Thomas Bretscher
     Affiliation:          ETH Zuerich, TIK
     Version:
     Creation Date:
     Last Date of Change:  1999/01/07       by: Thomas Bretscher


Based on SCN2681_DUART.h for the Serial In/Out
*/
#include “Topsy.h”
#include “MMHeapMemory.h”
#include “IODevice.h”
#include “Configuration.h”
#include “IDT385_IOMap.h”

#define MODE_REGISTER 0x0/* based on SCN2681 register layout */
#define STATUS_REGISTER 0x4
#define COMMAND_REGISTER 0x8
#define TX_REGISTER 0x0c
#define RX_REGISTER TX_REGISTER
#define INTERRUPT_REGISTER 0x14

#define READY_TO_SEND 0x04/* ready to send status */
#define RECV_MASK 0x01/* a byte has arrived */

#define FILEREADER_BUFSIZE 128

#define FILEREADER_REGISTER_SET_SIZE 0x20   /* size for one channel */

typedef struct devFileReaderDesc_t {
    unsigned long in, out;
} devFileReaderDesc;

typedef devFileReaderDesc* devFileReader;

void devFileReader_interruptHandler(IODevice this);
Error devFileReader_init(IODevice this);
Error devFileReader_read(IODevice this, ThreadId threadId, char* buffer, long int* size);
Error devFileReader_write(IODevice this, ThreadId threadId, char* buffer, long int* size);
Error devFileReader_close(IODevice this);
```

In `IDT385_IOMap.h`, wozu keine Programmdatei gehört, stehen die Adressen der IO-Geräte.

```
/*
     Copyright (c) 1996-1997 Swiss Federal Institute of Technology,
     Computer Engineering and Networks Laboratory. All rights reserved.

     TOPSY - A Teachable Operating System.
          Implementation of a tiny and simple micro kernel for
          teaching purposes.

     Permission to use, copy, modify, and distribute this software and its
```

```
     documentation for any purpose, without fee, and without written
     agreement is hereby granted, provided that the above copyright notice
     and the following two paragraphs appear in all copies of this software.



     File:                  IO/Drivers/IDT385_IOMap.h
     Author(s):             George Fankhauser
     Affiliation:           ETH Zuerich, TIK
     Version:
     Creation Date:
     Last Date of Change:   1999/01/11      by: Thomas Bretscher
*/
#include "Configuration.h"

#ifdef TOPSY_BIG_ENDIAN

#define UART_A_BASE 0xbfe00003     /* kernel uncached address */
#define UART_B_BASE 0xbfe00023     /* kernel uncached address */
#define FPGA_BASE   0xbfa40003     /* kernel uncached address */
#define FILEREADERBASE  0xbfa50003 /* kernel uncached address */

#else

#define UART_A_BASE 0xbfe00000     /* kernel uncached address */
#define UART_B_BASE 0xbfe00020     /* kernel uncached address */
#define FPGA_BASE   0xbfa40000     /* kernel uncached address */
#define FILEREADERBASE  0xbfa50000 /* kernel uncached address */

#endif
```

C.1.2 User

`Shell.h` ist die Header-Datei zu `Shell.c` mit den neuen Befehlen zur Modulverwaltung.

```
/*
     File:                  User/Shell.h
     Author(s):             George Fankhauser / Thomas Bretscher (load comm)
     Affiliation:           ETH Zuerich, TIK
     Version:               DynamicLoader
     Creation Date:
     Last Date of Change:    1999/03/02 Thomas Bretscher
*/

#include "Topsy.h"
#include "Syscall.h"

typedef struct ShellFunction_t {
    char* name;
    ThreadMainFunction function;
    ThreadArg arg;
} ShellFunction;


#define SHELL_INVALID   0
#define SHELL_START     1
#define SHELL_EXIT      2
#define SHELL_KILL      3
#define SHELL_INFO      4
#define SHELL_HELP      5
#define SHELL_PS        6
#define SHELL_LOAD      7
#define SHELL_FREE      8
```

```
#define SHELL_MODINFO   9
#define SHELL_NULL     10

#define BACKGRCOMMAND       '&'

static char* commandNames[] = {
     "invalid command\n",
     "start", "exit", "kill", "info", "help", "ps",
     "load", "free", "mod",
     "\n"
};

#define WELCOME "\nTopsy Shell 'Dynamic Loader' (c) 1999, ETH Zurich, TIK\nShell: Thomas Bretscher,
experimental version (new load command)\n\n"
#define EXITMESSAGE "exiting shell...\n"
#define PROMPT "> "
#define HELP "    start <function> <arg>   start a new thread\n    exit                      leave
this shell  \n    kill <id>                kill a thread     \n    ps                        print
thread status\n    load                   load module\n    free                  unload module\n
mod                    info about loaded modules\n    help                  show this helptext\n"

void main(ThreadArg arg);
```

C.2 Neu hinzugefügte Header-Dateien

C.2.1 Kernel

a) Format- und hardwareunabhängige Dateien im Verzeichnis `Modules`

Die folgende Datei `Modules.h` enthält allgemeine Definitionen im Zusammenhang mit dem dynamischen Lader/Linker, ihr ist keine Programmdatei zugeordnet.

```
/*
     File:                  Modules/Modules.h
     Author(s):             Thomas Bretscher
     Affiliation:
     Version:               DynamicLoader 1.0
     Creation Date:         1999/01/23
     Last Date of Change:   1999/02/02 (1999/03/03)  Thomas Bretscher

General includings for dynamic loader/linker.
*/

#ifndef _MODULES_H_
#define _MODULES_H_

#include "Memory.h"

#define FIRST_USERMODULE     1
#define FIRST_KERNELMODULE (-1)

#define MODULE_MODE(x) (x <= 0 ? KERNEL : USER)

typedef enum {
  NOCOMMAND,       /* no command or undefined type */
  SIMPLECOMMAND,   /* (shell-)command ignoring parameters */
  ARGCOMMAND       /* (shell-)command with standard ThreadArg (array)*/
} CommandType;
```

```
#endif
```

`ModMain.h` ist der Header zu `ModMain.c`.

```
/*
     File:                  Modules/ModMain.h
     Author(s):             Thomas Bretscher
     Affiliation:
     Version:               DynamicLoader 1.0
     Creation Date:         1999/01/19
     Last Date of Change:   1999/01/23  (1999/02/23) Thomas Bretscher


Module Manager, architecture and object format independent parts.
The List(s) of dynamic modules with their symbols are here in ModMain's heap
(data types provided by ModLinker).
The function modMain has to run with the thread id MODTHREADID.
*/

#include "Topsy.h"

void modMain(ThreadArg arg);
```

`ModLinker.h` ist der Header zu `ModLinker.c`.

```
/*
     File:                  Modules/ModLinker.h
     Author(s):             Thomas Bretscher
     Affiliation:
     Version:               DynamicLoader 1.0
     Creation Date:         1999/01/20
     Last Date of Change:   1999/02/16 (1999/02/23)  Thomas Bretscher


Dynamic Module Linker, architecture and object format independent parts.
*/

#include "Topsy.h"
#include "Modules.h"
#include "ModInterface.h"
#include "Configuration.h"
#include "Syscall.h"
#include "Messages.h"
#include "List.h"

#define MODIMPORTPREFIX   "MOD_Import_"
#define MODNAMEDELIMITER  '_'
#define USERMEMOWNER ANY  /* defined as Thread ID 1 in Messages.h
         (here: the "virtual" thread holding the memory for user modules) */

typedef struct Module_t {
  ModuleId mid;
  struct Module_t* sameMod; /* circular list to all "hard-linked" parts
                     of a module group
                               normal (only one part): self reference */
  char name[MAXDYNAMICNAMESIZE];
  Address codePtr;          /* start of the vmRegion for code */
  Address readonlyPtr;      /* start of the vmRegion for read only data */
  Address readwritePtr;     /* start of the vmRegion for data */
  int     referenceCounter; /* number of importing modules */
  int     userCounter;  /* number of "living" threads started in this module */
```

```
  List    imports;      /* list of imported modules (pointers to description)*/
  List    symbols;      /* list of exported symbols (functions, data objects)*/
} Module;

typedef struct Symbol_t {
  char        name[MAXDYNAMICNAMESIZE];               /* without prefix */
  CommandType commType; /* NOCOMMAND, SIMPLECOMMAND, ARGCOMMAND (, ... ) */
  /* type */           /* nothing at the moment */
  Address     addr;
} Symbol;


SyscallError modLoadLink(Address objImagePtr, List modList, AddressSpace mode,
               ModuleInfo* answerArray,
               int desiredNbOfAnswers, int* nbOfAnswers);

SyscallError modUnlinkFree(ModuleId mid, ModFreeMode freemode,
                 List modList, AddressSpace mode);
```

b) Modulformatabhängige Dateien im Verzeichnis `Modules/elf`

Die Datei `ELF_Decl.h` enthält Informationen zum ELF-Format, es existiert keine zugehörige Programmdatei.

```
/*
File: ELF_Decl.h
Thomas Bretscher, 1999/02/15 (1999/02/23)

items copied out of the description of the ELF format
http://www.dordt.edu:457/topics/BOOKCHAPTER-8.html
local copy (with inconsistent links): BOOKCHAPTER-8.html

Declarations of data structures of the ELF object file format
not (yet) complete
*/
#ifndef _ELF_DECL_H_
#define _ELF_DECL_H_

/* general */

#define      EI_NIDENT    16

/* file types */
#define      ET_NONE         0        /* No file type */
#define      ET_REL          1        /* Relocatable file */
#define      ET_EXEC         2        /* Executable file */
#define      ET_DYN          3        /* Shared object file */
#define      ET_CORE         4        /* Core file */
#define      ET_LOPROC       0xff00   /* Processor-specific */
#define      ET_HIPROC       0xffff   /* Processor-specific */

/* machine types */
#define      EM_NONE         0       /* No machine */
#define      EM_M32          1       /* AT&T WE 32100 */
#define      EM_SPARC        2       /* SPARC */
#define      EM_386          3       /* Intel386 CPU */
#define      EM_68K          4       /* Motorola 68000 */
#define      EM_88K          5       /* Motorola 88000 */
#define      EM_860          7       /* Intel860 CPU */
#define      EM_MIPS         8       /* MIPS R2000 (?? R3000 ??) */

/* special section indexes */
#define      SHN_UNDEF       0
```

```
#define      SHN_LORESERVE   0xff00
#define      SHN_LOPROC      0xff00
#define      SHN_HIPROC      0xff1f
#define      SHN_ABS         0xfff1
#define      SHN_COMMON      0xfff2
#define      SHN_HIRESERVE   0xffff

/* section types */
#define      SHT_NULL                    0
#define      SHT_PROGBITS                1
#define      SHT_SYMTAB                  2
#define      SHT_STRTAB                  3
#define      SHT_RELA                    4
#define      SHT_HASH                    5
#define      SHT_DYNAMIC                 6
#define      SHT_NOTE                    7
#define      SHT_NOBITS                  8
#define      SHT_REL                     9
#define      SHT_SHLIB                   10
#define      SHT_DYNSYM                  11
#define      SHT_LOPROC                  0x70000000
#define      SHT_HIPROC                  0x7fffffff
#define      SHT_LOUSER                  0x80000000
#define      SHT_HIUSER                  0xffffffff

/* section attribute flags */
#define      SHF_WRITE       0x1
#define      SHF_ALLOC       0x2
#define      SHF_EXECINSTR   0x4
#define      SHF_MASKPROC    0xf0000000

/* Symbol table */
#define ELF32_ST_BIND(i)   ((i)>>4)
#define ELF32_ST_TYPE(i)   ((i)&0xf)
#define ELF32_ST_INFO(b,t) (((b)<<4)+((t)&0xf))
#define      STB_LOCAL               0
#define      STB_GLOBAL              1
#define      STB_WEAK                2
#define      STB_LOPROC              13
#define      STB_HIPROC              15
#define      STT_NOTYPE              0
#define      STT_OBJECT              1
#define      STT_FUNC                2
#define      STT_SECTION             3
#define      STT_FILE                4
#define      STT_LOPROC              13
#define      STT_HIPROC              15

/* Relocation */
#define ELF32_R_SYM(i)    ((i)>>8)
#define ELF32_R_TYPE(i)   ((unsigned char)(i))
#define ELF32_R_INFO(s,t) (((s)<<8)+(unsigned char)(t))

/* various */
#define      STN_UNDEF    0

/* Types ***************************  */

typedef unsigned int       Elf32_Addr;     /* Unsigned program address */
typedef unsigned short int Elf32_Half;     /* Unsigned medium integer */
typedef unsigned int       Elf32_Off;      /* Unsigned file offset */
typedef signed int         Elf32_Sword;    /* Signed large integer */
typedef unsigned int       Elf32_Word;     /* Unsigned large integer */
/*  unsigned char                          Unsigned small integer */
```

```
   typedef struct {   /* ELF Header */
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
   } Elf32_Ehdr;

   typedef struct {   /* Section header */
         Elf32_Word      sh_name;
         Elf32_Word      sh_type;
         Elf32_Word      sh_flags;
         Elf32_Addr      sh_addr;
         Elf32_Off       sh_offset;
         Elf32_Word      sh_size;
         Elf32_Word      sh_link;
         Elf32_Word      sh_info;
         Elf32_Word      sh_addralign;
         Elf32_Word      sh_entsize;
   } Elf32_Shdr;

   typedef struct {   /* Symbol table entry */
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf32_Half      st_shndx;
   } Elf32_Sym;

/* Relocation entries */
   typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
   } Elf32_Rel;


   typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
        Elf32_Sword     r_addend;
   } Elf32_Rela;


/* special cases ********************  */

static const unsigned char e_ident_ELF32BigVersion1[EI_NIDENT] =
 { 0x7f,'E','L','F', 1, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
static const unsigned char e_ident_ELF32LittleVersion1[EI_NIDENT] =
 { 0x7f,'E','L','F', 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

#endif
```

Zur folgenden Datei `ModInterface.h` gehört ebenfalls keine Programmdatei.

```
/*
     File:                  Modules/elf/ModInterface.h
     Author(s):             Thomas Bretscher
     Affiliation:
     Version:               DynamicLoader 1.0
     Creation Date:         1999/01/11
     Last Date of Change:   1999/02/16 (1999/03/03)  Thomas Bretscher
*/
/*
This file is the interface between object format specific and non-specific
parsing of the object file image in the memory.

If they are not marked with "*internal name*", the names of constants, structs
and macros MUST NOT be changed when porting to an other object format (as long
as there are no changes necessary in the "non-specific" source files, which is
not absolutely shure...).
The order and the types of the parts within a struct may vary (for example
short int instead of long int) for reason of compatibility to the "natural"
structures of a specific format.
*/

#ifndef _MODINTERFACE_H_
#define _MODINTERFACE_H_

#include "ELF_Decl.h"
#include "../../Topsy/Topsy.h"

#define MAXNBOFSECTIONS 16

#define SECTFLALLOC    1  /* else not needed in final module */
#define SECTFLLOAD     2  /* else uninitialized data (e.g. .bss)  */
#define SECTFLREADONLY 4
#define SECTFLCODE     8  /* else data */

/* ModSectionHeader only for sections needed in the final module */
/* The final symbol table (in most cases smaller than the original one) will
NOT be stored in a section, but with ModMain's data */
typedef struct ModSectionHeader_t {
  int                     sectTypeFlags;
  Address                 contentPtr;
  Address                 loadedAddr;
  unsigned long int       size;
  struct ModRelocTable_t* relocTablePtr;
} ModSectionHeader;

typedef struct ModObjectImageHeader_t {
  Address              imageAddr;
  struct ModSymbol_t** symbols;      /* pointer to (temporary) symbol array */
  char*                stringTable;  /* pointer to string table of symbol tab*/
  ModSectionHeader*    sectArray[MAXNBOFSECTIONS]; /* NULL if not existing */
} ModObjectImageHeader;

typedef struct ModSymbol_t{    /* to be "laid out" onto the memory ELF image*/
  unsigned int       nameIndex;/* string table index */
  Address            offset;
  /* if MODSYMBOLTYPE(.) != MODSYMBOLTYPEEXTERN: from section beginning
     else: first undef., after symbol searching absolute Address */
  Elf32_Word    size;          /* at present *internal name*,  unused */
  unsigned char info;          /* *internal name* */
  unsigned char unused;        /* *internal name* */
  unsigned short int sectNumber; /* where the symbol belongs to */
} ModSymbol;
```

```
/* Macros to extract data from ModSymbol: */
/* old: #define MODSYMBOLTYPE(symbol)   (((symbol).info) & 0xf)  */
#define MODSYMBOLTYPE(symbol)  (ELF32_ST_BIND((symbol).info) == STB_LOCAL ? \
               (MODSYMBOLTYPELOCAL) : ELF32_ST_TYPE((symbol).info))
/* Constants */
#define MODSYMBOLTYPELOCAL    99          /* a constant not yet used by ELF */
#define MODSYMBOLTYPEEXTERN   STT_NOTYPE  /* an ELF symbol table constant */

typedef struct ModReloc_t {   /* to be "laid out" onto the memory ELF image */
  Address            offset;  /* from section beginning */
  Elf32_Word    info;         /* *internal name*  (Elf32Word is unsigned int)*/
  Elf32_Sword   addend;       /* *internal name*  (Elf32Sword is signed int)*/
} ModReloc;

/* Macros to extract data from ModReloc: */
#define MODRELOCSYMBINDEX(reloc) ((reloc).info>>8) /* symbol table index */
#define MODRELOCTYPE(reloc)      ((unsigned char) ((reloc).info & 0xff))
/* relocation type (to be interpreted by Translate, e. g. ElfMipsTranslate) */
#define MODRELOCADDEND(reloc)    ((reloc).addend)  /* explicit amount to add */


typedef struct ModRelocTable_t {
  ModSymbol**       symbols;      /* pointer to (temporary) symbol array */
  char*             stringTable;  /* pointer to string table of symbol table */
  Boolean           addendsValid; /* if FALSE: ignore addends of this table! */
  ModReloc**        reloc;        /* pointer to (temp) reloc entry array */
} ModRelocTable;

#endif
```

`ModParseObj.h` ist der Header des Parsers (`ModParseObj.c`).

```
/*
     File:                  Modules/elf/ModParseObj.h
     Author(s):             Thomas Bretscher
     Affiliation:
     Version:               DynamicLoader 1.0
     Creation Date:         1999/01/11
     Last Date of Change:   1999/01/15  (1999/02/23) Thomas Bretscher

ELF-specific object-"file" (memory image) parser.

Required format:
ELF32, big-endian (should becom big/little), relocatable object
*/

#include "ELF_Decl.h"
#include "ModInterface.h"
#include "../../Topsy/Topsy.h"

#define MOD_PARSEOK               0
#define MOD_PARSEFAILED           1 /* various reasons */
#define MOD_PARSEWRONGIDENT       2
#define MOD_PARSEFAILEDMEMALLOC   4
#define MOD_PARSESYMBTABLENOTFOUND 8 /* or more then one symbol table */
#define MOD_PARSEFORMATERROR     16

#define MOD_PARSEFREEOK           0
#define MOD_PARSEFREEFAILED     128
#define MOD_PARSEFREEINCOMPLETE  64

/* Function modParseObj builds the Topsy-compatible object headers,
symbol tables and relocation tables.
```

```
The Topsy structs are very close to the ELF structs, so a lot of converting
consists only in setting pointers to the ELF headers. Porting to other
object formats will possibly require more copy operations. */

Error modParseObj(Address imageAddr, ModObjectImageHeader* parseResults);

/* Function modFreeParseData destroys all data structures and frees
memory allocated by modParseObj */

Error modFreeParseData(ModObjectImageHeader* parseResults);
```

c) Prozessorabhängige Dateien im Verzeichnis `Modules/mips`

`ModReloc.h` ist der Header des Relozierers (`ModReloc.c`).

```
/*
     File:                  Modules/mips/ModReloc.h
     Author(s):             Thomas Bretscher
     Affiliation:
     Version:               DynamicLoader 1.0
     Creation Date:         1999/01/15
     Last Date of Change:   1999/02/15  (1999/02/23) Thomas Bretscher

MIPS-R3000-specific relocation.

Symbols addressed relative to the global pointer ($gp) (relocation type 7)
can not be handled, so invoke gcc with the obtion -G 0 (no .sbss and .sdata
sections) when compiling the modules (MIPS version).
There must not be any global (uninitialized) variable associated with section
COMMON (segment common to different modules), this is avoided by initializing
the variables or - if not exported - by using static variables.
*/

#include "ModInterface.h"
#include "Topsy.h"

#define MOD_RELOCOK           0
#define MOD_RELOCNO           2 /* no reloc table for this section */
#define MOD_RELOCSYMBUNKNOWN  3 /* at least one symbol is unknown (rest done)*/
#define MOD_RELOCTYPEUNKNOWN  4 /* unknown reloc type (rest partially done) */
#define MOD_RELOCADDRERROR    5 /* error in address computing */
#define MOD_RELOCFAILED       1 /* other reasons */
/* relocate a section once again (even if aborted) without copying the code
   again, leads to errors! */

/* relocation is done directly in the final code */
Error relocateSection(ModObjectImageHeader* header, int sectionNo);

Address alignSectionAddress(Address x);
```

d) Verbindung formatabhängig — prozessorabhängig im Verzeichnis `Modules/mipself`

Die Datei `Object2ArchTranslate.h` ist der Header zu `Object2ArchTranslate.c`.

```
/*
     File:                  Modules/mipself/Object2ArchTranslate.h
     Author(s):             Thomas Bretscher
     Affiliation:
     Version:               DynamicLoader 1.0
     Creation Date:         1999/02/15 (with old name ElfMipsTranslate.h)
     Last Date of Change:   1999/02/18 (1999/02/23) Thomas Bretscher


Translation of the relocation type numbers from mips-elf object format to
type numbers used in the mips version of ModReloc.
*/

#include "ELF_Decl.h"
#include "ModInterface.h"

#define ACTUALMACHINE  EM_MIPS  /* an ELF machine type constant */

#define RELTYPEWHOLEWORD 2
#define RELTYPEJUMP      4
#define RELTYPEIMMHIGH16 5
#define RELTYPEIMMLOW16  6
#define RELTYPEUNKNOWN   0

int translateRelocTypes(ModReloc t);
```

C.2.2 User

Die Datei `Transfer.h` ist der Header zu `Transfer.c`.

```
/*
     File:                  User/Transfer.h
     Author(s):             Thomas Bretscher
     Affiliation:
     Version:
     Creation Date:         1999/01/07
     Last Date of Change:   1999/02/17  (1999/03/03)  by: Thomas Bretscher
*/
#include "Topsy.h"

/* Reads a "File" from ... (now from FileReader, in future from UART)
and stores in new allocated user memory.
Returns: in *startAddrPtr the memory address
         and in *lengthPtr the length of the "File"

Memory can be freed by calling transferFree.
*/

void transfer(Address* startAddrPtr, unsigned long int* lengthPtr);
void transferFree(Address startAddr);
```

Anhang D: Inhalt des elektronischen Archivs

Die im Laufe dieser Arbeit entstandenen oder veränderten Programme sowie dieser Bericht sind in digitaler Form archiviert.

D.1 Topsy-Sourcen

Vollständige Quellen der Topsy-Version DynamicLoader 1.0: `FullSources/Topsy.tar`

Geänderte und neue Dateien der Topsy-Quellen: `DiffSources/Topsy/`

Der Zusatz *.changes zeigt an, dass noch eine Datei mit Hinweisen auf die konkreten Änderungen besteht (manuell erstelltes ‘diff’).

```
DiffSources/Topsy/Makefile                   *.changes
DiffSources/Topsy/Makefile.elf               neu
DiffSources/Topsy/IO/IO.h                    *.changes
DiffSources/Topsy/IO/Drivers/FileReader.c    neu
DiffSources/Topsy/IO/Drivers/FileReader.h    neu
DiffSources/Topsy/IO/Drivers/IDT385_IOMap.h  *.changes
DiffSources/Topsy/Threads/TMMain.c           *.changes
DiffSources/Topsy/Threads/Threads.h          *.changes
DiffSources/Topsy/Topsy/Configuration.h      *.changes
DiffSources/Topsy/Topsy/Messages.h           *.changes
DiffSources/Topsy/Topsy/Syscall.c            *.changes
DiffSources/Topsy/Topsy/Syscall.h            *.changes
DiffSources/Topsy/Topsy/Topsy.h              *.changes
DiffSources/Topsy/User/FileReaderTest.c      neu
DiffSources/Topsy/User/FileReaderTest.h      neu
DiffSources/Topsy/User/Makefile              *.changes
DiffSources/Topsy/User/Shell.c               *.changes
DiffSources/Topsy/User/Shell.h               *.changes
DiffSources/Topsy/User/Transfer.c            neu
DiffSources/Topsy/User/Transfer.h            neu
```

Neues Verzeichnis für das Kernel-Modul `Modules`:

```
DiffSources/Topsy/Modules/ModLinker.c
DiffSources/Topsy/Modules/ModLinker.h
DiffSources/Topsy/Modules/ModMain.c
DiffSources/Topsy/Modules/ModMain.h
DiffSources/Topsy/Modules/Modules.h
DiffSources/Topsy/Modules/elf/ELF_Decl.h
DiffSources/Topsy/Modules/elf/ModInterface.h
DiffSources/Topsy/Modules/elf/ModParseObj.c
DiffSources/Topsy/Modules/elf/ModParseObj.h
DiffSources/Topsy/Modules/mips/ModReloc.c
DiffSources/Topsy/Modules/mips/ModReloc.h
DiffSources/Topsy/Modules/mipself/Obj2ArchTranslate.c
DiffSources/Topsy/Modules/mipself/Obj2ArchTranslate.h
```

Neues Verzeichnis für dynamisch ladbare Usermodule, enthält Test- und Demomodule sowie ein Makefile:

```
DiffSources/Topsy/DynamicUser/
  DynSyscall.c        HelloDyn.c          SeatReservation.c   UpcallDemo.c
  DynSyscall.h        HelloDyn.h          SeatReservation.h   UpcallDemo.h
  DynSyscallDefs.h    ImportDemo.c        SimpleSyscall.c     UserSupport.c
  DynSyscallMsg.S     ImportDemo.h        SimpleSyscall.h     UserSupport.h
  DynSyscallUnDefs.h  Makefile            Unconventional.c
```

Vollständige Quellen des angepassten Prozessor-Simulators (geschrieben in Java) der Testumgebung: `FullSources/MipsSimulator.tar`

Geänderte und neue Dateien des Mipssimulators.

Der Zusatz *.changes zeigt an, dass noch eine Datei mit Hinweisen auf die konkreten Änderungen besteht (manuell erstelltes 'diff').

```
DiffSources/MipsSimulator/FileReader.java     neu
DiffSources/MipsSimulator/Processor.java      *.changes
```

D.2 Tools

Ein selber programmiertes Tool, um Mips-ELF-Dateien zu parsen und Header, Symbol- und Relokationstabellen anzuzeigen:

```
Tools/analyELF/analyELF.c     Programm
Tools/analyELF/ELF_Decl.h     Datei identisch mit Topsy/Modules/elf/ELF_Decl.h
```

NICHT ENTHALTEN sind Cross-Compiler und Binary Utilities (Binutils)!

Für das Kompilieren der statisch gelinkten Topsy-Teile habe ich den auf verschiedenen Maschinen des TIK und in den Studenten-Arbeitsräumen installierten GNU-Cross-Compiler für Mips-Programme im ECOFF-Format verwendet.

Die bei der Semesterarbeit verwendeten Tools für Mips-Programme im ELF-Format habe ich folgendermassen installiert:

Version: `gcc-2.8.1, gnu binutils 2.9.1`

Konfigurationsbefehl:

```
./configure --target=mips-elf-elf \
--prefix=/home/bretsche/mips-elf-utils --with-gnu-as --with-gnu-ld
```

Anpassung des Makefiles: die ursprüngliche Fassung

```
# This is used instead of ALL_CFLAGS when compiling with GCC_FOR_TARGET.
# It omits XCFLAGS, and specifies -B./.
# It also specifies -I./include to find, e.g., stddef.h.
GCC_CFLAGS=$(INTERNAL_CFLAGS) $(X_CFLAGS) $(T_CFLAGS) $(CFLAGS) \
        -I./include $(TCFLAGS)
```

ergänzen (vor `$(TCFLAGS)`) durch:

```
-I/usr/cygnus/progressive-96q3/H-sparc-sun-solaris2/mips-idt-ecoff/include/
```

dadurch findet der Compiler `stdio.h` usw., wobei das für Topsy aber gar nicht benutzt wird.

Kompilieren:

```
gmake LANGUAGES=c
```

D.3 Dokumentation

Das Verzeichnis `Docu/Bericht/` enthält den abgegebenen Bericht als FrameMaker-5.5-Dokumente sowie als PDF-Datei:

```
Docu/Bericht/
   Bericht.book              BerichtTOC.fm             BerichtAppendix.frame
   BerichtStart.frame        BerichtBody.frame
Docu/Bericht/Bericht.pdf
```

Das Verzeichnis `Docu/Praesentation/` enthält die Folien der Präsentation als FrameMaker-5.5-Dokumente sowie als PDF-Datei:

```
Docu/Praesentation/
   FolieDemoprogramm.frame   FolieLadeprobleme.frame   FolieStand.frame
   FolieKernelUser.frame     FolieModularStruct.frame  FolieStart.frame
   FolieLadeVorgang.frame    FolieModulkonzept.frame
Docu/Praesentation/Folien.pdf
```

Die folgende Datei enthält eine Beschreibung des Dateiformats ELF. Kopie von `http://www.dordt.edu:457/topics/BOOKCHAPTER-8.html` (mit ungültigen Links).

```
Docu/BOOKCHAPTER-8.html
```

Anhang E: Literaturverzeichnis

[1] G. Fankhauser, C. Conrad, E. Zitzler und B. Plattner, Institut für Technische Informatik und Kommunikationsnetze TIK, ETH Zürich. *Topsy — A Teachable Operating System*. 1997.

[2] Institut für Technische Informatik und Kommunikationsnetze TIK, ETH Zürich. *Topsy home page*
`http://www.tik.ee.ethz.ch/~topsy`

[3] G. Fankhauser, Institut für Technische Informatik und Kommunikationsnetze TIK, ETH Zürich. *A MIPS R3000 Simulator in Java*.
`http://www.tik.ee.ethz.ch/~gfa/MipsSim.html`

[4] Computer Science Departement, Stanford University. *SimOS, The Complete Machine Simulator.*
`http://simos.stanford.edu/`
`http://simos.stanford.edu/userguide/userguide-29.html`

[5] N. Wirth und J. Gutknecht, Institut für Computersysteme, ETH Zürich. *Project Oberon, The Design of an Operating System and Compiler.* Addison-Wesley, ACM Press, 1992.

[6] The Santa Cruz Operation, Inc. (SCO). WWW-Artikel in der SCO Online Library, *Developer's Topics*, Kapitel 8: *ELF object files*. 1995.
`http://www.dordt.edu:457/topics/BOOKCHAPTER-8.html`.