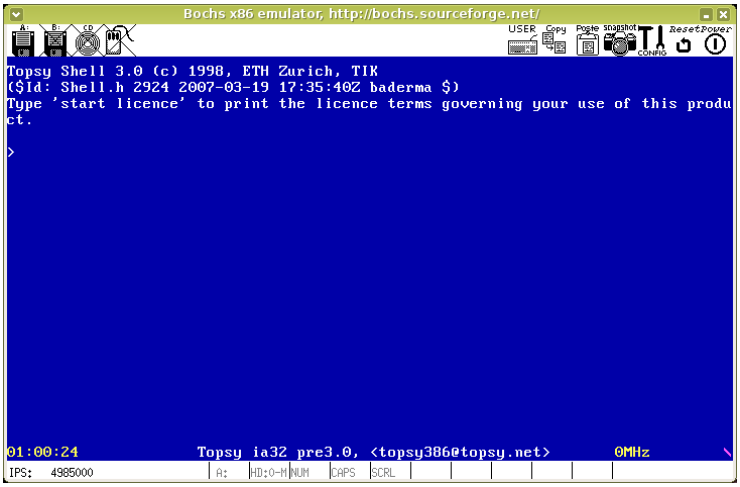
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Portierung von Topsy v3 auf den Intel Pentium 4

Voraussetzungen für PC/AT-kompatible Multiprozessor-Unterstützung



Semesterarbeit von Sebastian Ryffel

Betreuer: Dr. Lukas Ruf
Professor: Prof. Dr. Bernhard Plattner

SA-2007-19 von März bis August 2007
20. August 2007

Zusammenfassung

Das Ziel dieser Semesterarbeit ist die Portierung der dritten Version des Teachable Operating System (Topsy v3) auf PC/AT-kompatible PCs mit Prozessoren der Intel ia32-Architektur ab dem Pentium 4. Topsy v3 ist ein schlankes multi-tasking und multi-threading Microkernel Betriebssystem, welches sich besonders für embedded Systeme und als Node OS eignet. Es unterstützt Paging und mehrere virtuelle Adressräume.

Diese Arbeit konzentriert sich vor allem auf den Microkernel. Zuerst wurde der Page Mapping Layer portiert, welcher alle architektur-spezifischen Teile des Memory Managements beinhaltet. Das Thread Management wurde grundlegend überarbeitet und besitzt eine neue Architektur. Zusammen mit dem neuen reentranten Interrupt Handler ermöglicht es Nested Exceptions beliebiger Tiefe. Neu kann Topsy die Funktionen des lokalen APICs und des I/O-APICs verwenden. Damit schafft diese Arbeit die Voraussetzungen für den zukünftigen Betrieb von Multicore Systemen gemäss der MultiProcessor Spezifikation von Intel.

Abstract

The goal of this semester thesis is to port the third version of the Teachable Operating System (Topsy v3) to PC/AT compatible computers with Pentium 4 or later processors with Intels ia32-architecture. Topsy v3 is a multi-tasking multi-threading microkernel OS. With its modular network subsystem it is especially suited to run as a node operating system. It supports paging and multiple virtual address spaces. This thesis should serve as a firm foundation for later multicore operation as specified by the MultiProcessor Specification.

This thesis concentrates mainly on the microkernel. First the Page Mapping Layer, where the architecture dependent Memory Management happens, had to be ported. The Thread Management was redesigned. The new Interrupt Handler is reentrant and allows nested exceptions. Topsy now can take advantage of the features of the local APIC and the I/O-APIC, which are important requirements for the use of multiple processors.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Topsy v3 ist die dritte Version des Teachable Operating System, welches an der ETH ursprünglich zu Unterrichtszwecken entwickelt wurde. Es ist ein multi-tasking und multi-threading Microkernel Betriebssystem, bei dessen Design besonderer Wert auf Einfachheit und Schlankheit gelegt wurde. Damit eignet es sich besonders für eingebettete Systeme und dank dem neuen modularen Netzwerk Stack als Betriebssystem für Netzwerk Knoten (NodeOS).

Topsy ist ein modernes Betriebssystem, welches dank dem Microkernel Ansatz aus der Masse hervorsticht. Bis jetzt lief Topsy v3 nur auf MIPS- und Intel XScale-Rechnern.

1.1 Motivation

Für ia32-kompatible Prozessoren existierte bis anhin nur eine Portierung für die Topsy v1 Architektur. Mit dieser Arbeit läuft Topsy auf allen PC/AT-kompatiblen Computern mit Prozessoren der ia32-Architektur. Diese beinhalten heutzutage zumeist schon mehrere Kerne pro Prozessor. Deshalb war die ursprüngliche Absicht der Arbeit, Topsy vollständig multicore-fähig zu machen. Schon bald aber zeigte sich, dass dies im Rahmen einer Semesterarbeit aus zeitlichen Gründen nicht realistisch ist.

1.2 Dokumentstruktur

Im Kapitel ‘Verwandte Arbeiten’ werden Arbeiten vorgestellt, auf welchen diese aufbaut. Anhand deren Leistungen und den vorhandenen Lücken leiten sich der Umfang und die Ziele dieser Arbeit ab.

Darauf aufbauend werden im Kapitel 3 die Aufgabenstellung und die daraus resultierenden Herausforderungen erläutert.

Das gewählte Design des Ports ist im Kapitel ‘Architektur’ beschrieben. Wichtige Designentscheide und der Aufbau werden erklärt.

Daraus abgeleitet folgt die Beschreibung der ‘Implementation’. In diesem Kapitel sind alle wichtigen Änderungen an der Codebasis dokumentiert. Besonders diejenigen, die Anpassungen der anderen Portierungen bedingen.

Die Methoden und Resultate der durchgeführten Tests zur Verifizierung des Codes sind im Kapitel ‘Evaluation’ dokumentiert.

Diese Arbeit bringt Topsy einen Schritt weiter in Richtung Multicore Unterstützung. Zudem läuft Topsy nun auf vielen modernen Prozessoren, womit sich neue Anwendungsbereiche erschliessen. Mögliche Stossrichtungen für zukünftige Arbeiten werden im Kapitel ‘Weiterführende Arbeiten’ aufgezählt.

Das letzte Kapitel fasst die Arbeit zusammen und schliesst mit einem Fazit.

1.3 Danksagungen

Ich möchte mich bei Herrn Dr. Lukas Ruf ganz herzlich für die Betreuung und Unterstützung bedanken.

Ich danke Herrn Prof. Dr. Bernhard Plattner für die Ermöglichung der Arbeit.

Auch vielen Dank ans Computer Engineering and Networks Laboratory (TIK) für den Support und die zur Verfügung gestellte Infrastruktur.

Kapitel 2

Verwandte Arbeiten

Topsy ist ein für Unterrichtszwecke geschriebenes, multihreading fähiges Microkernel Betriebssystem, dessen Interprozesskommunikation auf dem Message-Passing Konzept aufbaut. Im Rahmen diverser Semester- und Diplomarbeiten wurde Topsy stetig erweitert und auf neue Plattformen portiert.

In diesem Kapitel werden die für diese Arbeit relevanten Arbeiten vorgestellt.

2.1 Topsy v1

Topsy v1 wurde 1996 von George Fankhauser für den MIPS R3000 entwickelt [1].

2.1.1 Aufbau von Topsy v1

Die Abbildung 2.1 auf Seite 6 zeigt auf der linken Hälfte den Aufbau von Topsy v1. Er besteht aus drei Schichten.

User

Zuoberst ist der User Layer. Dort laufen die Shell und weitere, unprivilegierte Threads des Benutzers. Die Interprozesskommunikation, also die Kommunikation unter den Threads, läuft über Nachrichten. Der gleiche Mechanismus wird auch für die Schnittstelle zu den Threads der Kernel Schicht verwendet.

Kernel

Das Herzstück von Topsy ist die Kernel Schicht. Der Kernel ist gemäss dem Microkernel-Ansatz auf unabhängige Threads aufgeteilt, welche ihrerseits über Nachrichten kommunizieren. Im laufenden Betrieb sind in Topsy v1 vier Kernel Threads aktiv:

Memory Management Thread (MM-Thread) verwaltet den verfügbaren Speicher. Die anderen Threads delegieren das Allozieren und später das Freigeben von Speicherbereichen an diesen Thread.

Thread Management Thread (TM-Thread) verwaltet alle Threads. Er kann Threads erstellen, starten oder beenden. Wenn ein Thread terminiert, wird er von diesem Kernel Thread automatisch wieder aus dem System entfernt. Threads können freiwillig die CPU abgeben, indem sie eine Yield Nachricht an den TM-Thread senden.

Input/Output Thread (IO-Thread) verwaltet Input/Output-Ressourcen wie zum Beispiel die Konsole oder die Tastatur. Um ein Gerät zu öffnen, davon zu lesen, schreiben oder es wieder zu schliessen, muss eine entsprechender Syscall zum IO-Thread initiiert werden.

Idle Thread ist immer lauffähig und wird vom Scheduler nur gewählt, wenn alle anderen Threads blockiert sind. Der Idle Thread hält dann die CPU an.

In der Kernel Schicht sind zwei weitere Module angesiedelt: Das Topsy-Modul und der Startup-Prozess. Das Topsy Modul kann als eine Art Kernel Library gesehen werden. Es beinhaltet das Syscall Interface und stellt den Kernel Threads verschiedene Datentypen und die dazugehörigen Routinen zur Verfügung.

Im Startup-Modul befindet sich die Routine `topsyMain()`, welche nach dem Booten als erste architektur-unabhängige Funktion aufgerufen wird. Dort wird die Konsole initialisiert und der MM- und TM-Thread gestartet.

Hardware Abstraction Layer

Der Hardware Abstraction Layer (HAL) stellt dem Kernel auf jeder Architektur eine einheitliche Schnittstelle zur Verfügung. Zu seinen wichtigsten Funktionalitäten gehören das Interrupt Management und Handling, sowie die Realisierung der Interprozesskommunikation und Memory Management.

2.2 Topsy i386

Die erste Portierung von Topsy wurde 1998 von Lukas Ruf durchgeführt. Während seiner Semesterarbeit portierte er das MIPS-Topsy v1 auf PC/AT kompatible Computer mit Prozessoren der ia32-Architektur [2]. Damit zeigte er die Portierbarkeit von Topsy auf und legte den Grundstein für die vorliegende Arbeit. Lukas Rufs Version von Topsy wird Topsy i386 genannt.

Topsy i386 implementiert einen Boot-Prozess, welcher die Hardware initialisiert, Topsy lädt und in den architektur-unabhängigen Code springt. Zudem wurde der ganze HAL-Layer, also Memory-, Thread-Management, Input/Output und Interprozesskommunikation (IPC), portiert. Des weiteren beinhaltete die Arbeit Treiber für Videoausgabe und Tastatur.

2.2.1 Schwächen von Topsy i386

In diesem Kapitel werden einige Schwächen der ersten ia32-Portierung aufgeführt.

Speicherschutz

Topsy i386 kennt kein Paging und dementsprechend laufen alle Threads im gleichen Adressraum. Dieser Adressraum ist grundsätzlich für alle Threads lesbar und schreibbar.

Um dennoch Speicherschutz zu gewährleisten, verwenden Kernel und User Threads verschiedene Segmentselektoren. Mit Segmentselektoren kann der verfügbare Speicher einer Applikation auf einen beliebigen Bereich eingeschränkt werden. Für Threads der User Schicht wird dieser so gewählt, dass der Code und die Daten des Kernels ausserhalb liegen. So wird der Speicher des Kernels vor fehlerhaften und böswilligen Threads des Users abgeschottet. Letztere sind aber weder untereinander noch vor dem Kernel geschützt. Zudem können User Threads auf den Stack der Kernel Threads zugreifen und diesen sogar verändern. Der Speicherschutz ist also nur minimal.

Aktueller Zustand

Topsy i386 bootete und lief zuletzt 2001 auf einem Pentium II mit 200MHz. Seither hat sich Topsy stetig weiter entwickelt und viele neue Features bekommen, ohne dass die ia32-Portierung angepasst worden wäre.

Multi Prozessor Unterstützung

Topsy i386 läuft nur auf einem Prozessor. Um dies zu ändern, müssten die Kriterien der Multi-Prozessor Spezifikation [7] von Intel erfüllt werden. Diese definiert ein PC/AT-kompatibles Multiprozessor-Design und bewahrt dabei vollständige PC/AT-Binärkompatibilität. Sie spezifiziert alle für das Betriebssystem relevanten Komponenten. Obwohl diese Komponenten in allen modernen Prozessoren vorhanden sind und auch in Singleprozessor-Systemen Vorteile bringen, kann Topsy i386 deren Funktionalitäten nicht nutzen.

2.3 Topsy v3

Die Diplomarbeit 'Topsy v3' von Claudio Jeker und Boris Lutz "brachte Topsy einen Schritt weiter in Richtung NodeOS, in dem sie es mit einem modernen Speicher-, Prozess- und IO-Management ausstatteten" [3].

Wie in der Abbildung 2.1 ersichtlich, lässt sich Topsy v3 in drei Schichten unterteilen: Unprivileged, Privileged und Microkernel.

Die Unprivileged Schicht entspricht in etwa der User Schicht des alten Topsys. In ihr werden die Applikationen des Benutzers ausgeführt. Die Kernel Threads sind in der Privileged Schicht angesiedelt. Der Microkernel besteht im Wesentlichen nur noch aus einem Interrupt Handler und dem für die IPC zuständigen Nachrichten Dispatcher.

2.3.1 Neuerungen in Topsy v3

Drei wesentliche Neuerungen wurden eingeführt, die nachfolgend einzeln betrachtet werden:

Protection Domains

In Topsy v3 läuft jede Anwendung in einer sogenannten Protection Domain (PD). Eine Protection Domain definiert für alle Threads der Anwendung einen gemeinsamen virtuellen Adressraum und Grenzwerte für Prioritäten und Zugriffsberechtigungen. So werden die Threads einer Anwendung vor äusseren Einwirkungen durch Threads anderer Anwendungen geschützt. Dies stellt neue Anforderungen an Topsy und insbesondere an das Speichermanagement, welches nun virtuelle Adressräume unterstützen muss.

Memory Management

Für die Implementation verschiedener virtueller Adressräume verwendet das neue Memory Management Paging. Dazu wurde das Memory Management von Grund auf neu entwickelt und durch ein ausgeklügeltes, objektorientiertes System ersetzt. Wie in Topsy v1 müssen Speicherbereiche über Nachrichten an den Memory Manager angefordert und wieder freigegeben werden. Diese werden dann bei Bedarf gemappt und belegen vorher noch keinen physikalischen Speicher.

Speicher kann auch zwischen verschiedenen Protection Domains geshared werden. Zum Beispiel, wenn mehrere Threads den gleichen Programmcode teilen oder Shared-Memory verwenden. Topsy v3 beherrscht auch Copy-on-Write, bei dem eine zu kopierende Region erst beim ersten Schreibzugriff wirklich kopiert wird.

Interprozesskommunikation

Da nun nicht mehr alle Threads den gleichen Adressraum haben, musste für Topsy v3 auch die Interprozesskommunikation neu designed werden. Nachrichten sollten möglichst effizient zwischen Protection Domains ausgetauscht werden können, ohne dabei eine Lücke in den Speicherschutz zu reissen.

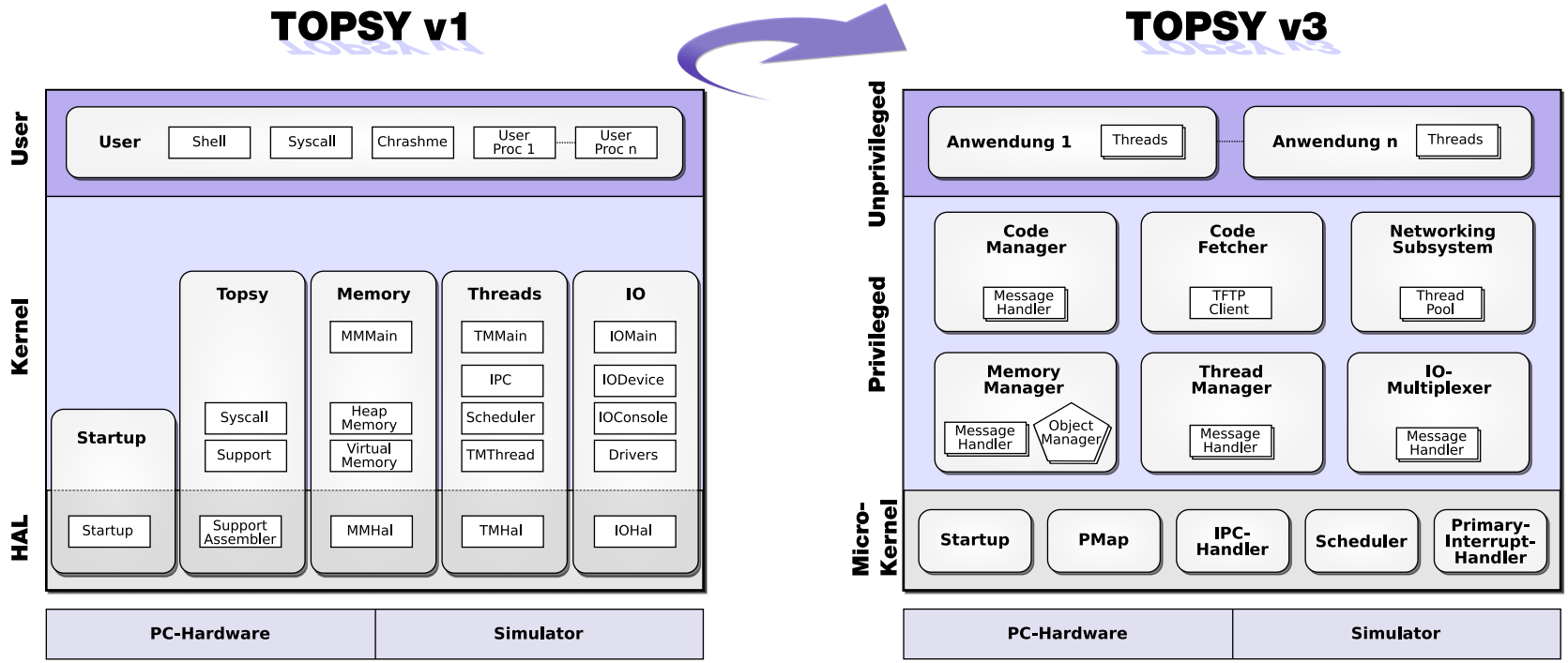


Abbildung 2.1: Topsy v1 vs Topsy v3

Die gewählte Lösung basiert auf einem zweistufigen Ansatz. Kleine Nachrichten werden von einer Protection Domain in die anderen kopiert. Für grössere kann ein Transportbehälter (Enveloppe) angehängt werden. Ein Loan-out-Mechanismus erlaubt es, diesen dann temporär in eine andere Protection Domain zu mappen. So kann der aufwendige Kopiervorgang umgangen werden.

2.3.2 Zusammenfassung

Claudio Jeker und Boris Lutz trieben den Microkernel Ansatz noch weiter und führten Topsy auf ein neues Level. Topsy ist nun nicht mehr nur multithreading-, sondern auch multitasking-fähig. Die Neuerungen haben weitreichende Konsequenzen für alle Subsysteme. Topsy v3 wurde nur für die MIPS Architektur, und für dem IXP1200 von Intel implementiert.

2.4 Linux Kernel

Der Linux Kernel [4] läuft auf diversen Plattformen, wobei die Zielplattform und Prozessorarchitektur dieser Arbeit die älteste und in [5] auch sehr gut dokumentiert ist. Zusammen mit der Verfügbarkeit des Quellcodes ist der Linux Kernel eine gute Anlaufstelle bei Unklarheiten in den Spezifikationen und dient dieser Arbeit als Referenzimplementation.

Durch sein monolithisches Design, in dem der Kernel Teil eines jeden Prozesses ist, unterscheidet er sich fundamental vom Microkernel-OS Topsy. Dennoch konnten einige interessante Techniken übernommen werden.

Kapitel 3

Herausforderungen

Das Ziel dieser Arbeit ist, die Lücke zwischen der aktuellen Version von Topsy und dem ia32-Port zu schliessen. Schliesslich soll Topsy v3 auf einem PC/AT-kompatiblen Rechner mit einer CPU ab Pentium 4 [6] stabil laufen. Beispiele dafür sind, neben dem Pentium 4 selber, der Pentium M [6] oder der der Multicore Prozessor Intel Core Duo [6].

In diesem Kapitel wird zuerst die Aufgabenstellung konkretisiert und nachher auf einige spezielle Herausforderungen eingegangen.

3.1 Aufgabenstellung

Wie bei den meisten Portierungen fällt der grösste Teil der Arbeit am Hardware Abstraction Layer an. Dieser muss an die neuen Schnittstellen von Topsy v3 angepasst werden und die neuen Features unterstützen. Da ein Design-Ziel von Topsy die Portierbarkeit ist und die neue Version schon auf mehreren Architekturen läuft, darf an den architektur-unabhängigen Teilen so wenig wie möglich geändert werden.

3.2 Herausforderungen

Obwohl sich die Arbeit vor allem auf den HAL konzentriert, ist es nötig, die darüber liegenden Schichten bis ins Detail zu verstehen. Nur mit dem Wissen, zu welchem Zweck, in welchem Kontext und in welcher Reihenfolge die zu implementierenden Funktionen aufgerufen werden, können diese richtig portiert werden. Dies setzt ein grosses Detailverständnis der Topsy-Mechanismen voraus.

Diese müssen möglichst effizient auf der Ziel-Plattform umgesetzt werden. Die Intel Architektur lässt dem Entwickler viele Freiheiten, indem sie für vieles verschiedene Lösungsvarianten zulässt. Dies und der Umstand, dass die Spezifikation [6] über die Jahre immer wieder rückwärtskompatibel erweitert wurde, macht sie äusserst komplex. Sie umfasst mittlerweile über dreieinhalbtausend Seiten, ohne die Multi Prozessor Spezifikation [7].

Die Einführung von Protection Domains in Topsy v3 und die damit verbundene Verwendung von Paging hat weitreichende Konsequenzen für alle Subsysteme. Zum einen muss Paging aktiviert und die entsprechenden Datenstrukturen verwaltet werden. Zum anderen müssen alle Subsysteme so angepasst werden, dass alle benötigten Datenstrukturen zugreifbar sind, ohne unnötig viel gemeinsamen Speicher zu verwenden. Dies gilt besonders für die Interprozesskommunikation.

Multi-Prozessor-Unterstützung, wie sie in der ursprünglichen Aufgabenstellung noch formuliert wurde, konnte aus Zeitgründen nicht realisiert werden. Dieses Ziel wurde aber nicht aus den Augen verloren. Diese Arbeit soll die Grundlage schaffen, um später Topsy auf mehreren Prozessoren auszuführen.

Die Unterstützung des Advanced Programmable Interrupt Controllers (APIC) und I/O-APIC ist eine nötige Voraussetzung, die die Multi-Prozessor Spezifikation stellt. Deshalb war es ein Ziel dieser Arbeit, diese Bausteine zu initialisieren und so zu konfigurieren, dass Topsy auch auf Systemen mit mehreren Prozessoren funktioniert. Dies bringt Vorteile, auch für Einprozessor-Systeme.

Der APIC wurde mit dem Pentium Pro eingeführt, ist in allen neueren Prozessoren integriert und wird deshalb auch lokaler APIC genannt. Dieser empfängt Interrupts von internen Quellen, den Interrupt Pins und den externen I/O-APIC-Bausteinen und leitet sie an den Prozessor weiter. In Multi-Prozessor-Systemen sendet und empfängt er auch Nachrichten an und von anderen Prozessoren. Der I/O-APIC ist Teil des Chipsets. Er empfängt externe Interrupts des Systems und angeschlossenen Geräten und sendet sie an den lokalen APIC von einem oder mehreren Prozessoren.

Eine besonders grosse Herausforderung war die Überarbeitung des Thread Management. Der alte Interrupt Handler war nicht reentrant. Das heisst, es durfte immer nur maximal ein Thread im Interrupt Handler sein. Im Interrupt Handler dürfen also keine Unterbrechungen auftreten. Der nächste Interrupt oder die nächste Exception darf erst behandelt werden, wenn der Handler abgearbeitet ist. Dies führt zu unnötigen Verzögerungen bei der Bearbeitung von wichtigen Interrupts, was besonders bei Mehr-Prozessor-Systemen suboptimal ist.

Mit der Einführung von Virtual Memory ist eine neue Ausnahmebedingung möglich geworden: die Page Fault Exception. Ein Page Fault wird ausgelöst, wenn auf eine Page zugegriffen wird, die nicht im System vorhanden ist. Dieser muss sofort behandelt werden. Wenn der Zugriff fehlerhaft war, muss der Thread beendet, anderenfalls eine neue Page an dieser Stelle eingefügt und der Thread an der gleichen Stelle nochmals gestartet werden. Im alten Interrupt Handler traten mit aktiviertem Paging oft Page Faults auf, die nicht bearbeitet werden konnten. Um solche Nested Exceptions zu behandeln, muss der Interrupt Handler reentrant sein.

Nested Exceptions sind in Topsy v3 ein neues Konzept und wurden jetzt mit dieser Arbeit erstmals eingeführt. Deshalb waren einige Änderungen am architektur-unabhängigen Teil nötig. Für reentrante Interrupt Handler müssen alle globalen Datenstrukturen durch Locks vor quasi-zeitgleichen Zugriffen geschützt werden. Schwieriger ist die Speicherung des Unterbrechungszustandes, von denen es mit Nested Exceptions beliebig viele geben kann. In den Unterbrechungszuständen müssen unter anderem Informationen des Schedulers gespeichert werden. Diese geben an, ob der Thread blockiert ist, auf ein Ereignis wartet oder lauffähig ist. Wenn nun ein Ereignis eintritt, müssen die Informationen des richtigen Zustandes geändert und der Thread eventuell aktiviert werden.

Reentrante Interrupt Handler sind eine Notwendigkeit in Multicore Systemen. Da es keine Möglichkeit gibt, Interrupts für alle CPUs gleichzeitig zu maskieren, kann es immer sein, dass zwei CPUs gleichzeitig im Interrupt Handler sind.

Diese Arbeit ist mehr als die blosse Portierung des HAL. Besonders die Einführung von Nested Exceptions und die Vorbereitungen für de Multicore Betrieb bedingen weitreichende Architekturänderungen am architektur-unabhängigen Teilen des Microkernels und an den Kernel Threads.

Kapitel 4

Architektur

Dieses Kapitel erklärt verschiedene Lösungsansätze zu den im vorherigen Kapitel eingeführten Problemen und begründet die gewählten Design-Entscheide. Dabei werden die Subsysteme “Memory Management”, “Thread Management” und “Interprozesskommunikation” in den folgenden drei Unterkapiteln separat behandelt.

4.1 Memory Management

Das Memory Management der ia32-Architektur baut auf den zwei Konzepten Segmente und Paging auf. Deren Funktionsweisen, Vor- und Nachteile werden zuerst erklärt und nachher wird auf ihre Verwendung in Topsy v3 eingegangen.

4.1.1 Einführung ins Memory Management der ia32-Architektur

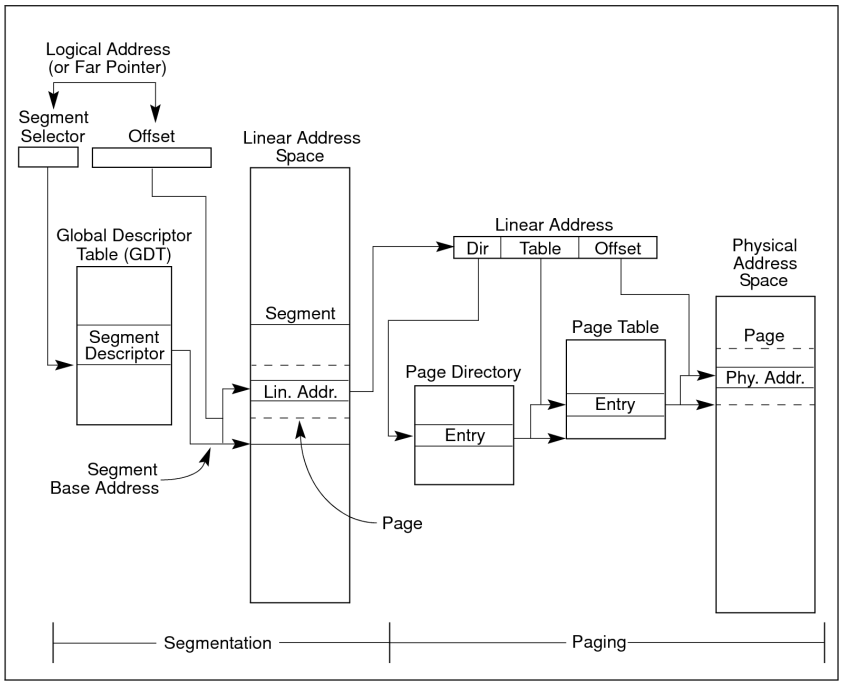


Abbildung 4.1: Übersicht über Segmente und Paging der ia32-Architektur (aus [6])

Die Abbildung 4.1 gibt einen Überblick über das Memory Management der ia32-Architektur. Für die Übersetzung von Adressen in den physikalischen Adressraum werden zwei Mechanismen verwendet: Segmente und Paging.

Segmente

Segmentierung ist ein Mechanismus, mit dem der verfügbare Speicher in kleinere, geschützte Bereiche, genannt Segmente, eingeteilt werden kann. Diese können für den Code, die Daten und den Stack eines Programms verwendet werden. Indem zum Beispiel jedem Prozess (oder Thread) eigene Segmente zugeteilt werden, kann der Prozessor garantieren, dass keiner ausserhalb seiner Segmente in den Bereich der anderen Prozesse schreibt. Segmente können als Code, Daten oder Stack deklariert und so die auf ihnen erlaubten Operationen eingeschränkt werden.

Paging

Wenn Paging aktiviert ist, unterteilt der Prozessor den adressierbaren Adressraum in kleine Einheiten, genannt Pages. Diese können dann an eine beliebige Stelle im RAM oder auf einem anderen Gerät gemappt werden.

Paging erlaubt nicht nur mehrere virtuelle Adressräume. Es kann zusätzlich für jede Page definiert werden, ob sie auch mit User oder nur mit Kernel Privilegien lesbar und/oder schreibbar ist. Berechtigungen können somit viel feingranularer gesetzt werden als bei Segmenten.

Adressierung

Um ein Byte in einem Segment zu lokalisieren, muss eine *logische Adresse* angegeben werden. Diese besteht aus einem 16-Bit breiten Segment Selektor und einem 32-Bit-Offset. Der Selektor identifiziert unter anderem eindeutig ein Segment, dessen Basis-Adresse in der Global Descriptor Table steht. Dazu wird der Offset addiert und so die 32-Bit breite *lineare Adresse* gebildet. Bei aktiviertem Paging übersetzt der Prozessor diese mit Hilfe des Page Directories und der Page Tables in eine *physikalische Adresse*. Sonst wird die lineare direkt auf die physikalische Adresse gemappt.

4.1.2 Segmente

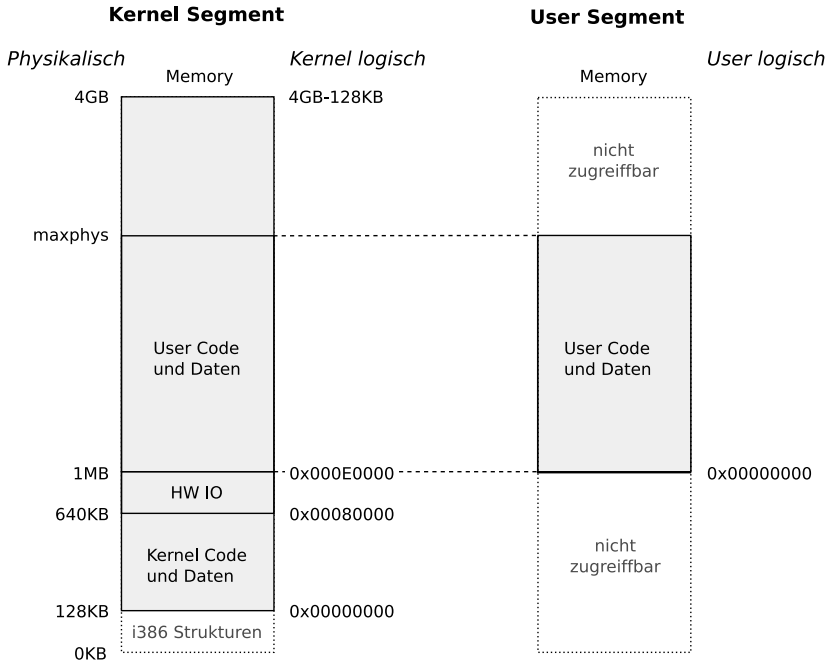


Abbildung 4.2: Kernel und User Segmente in Topsy i386

Topsy i386 benutzt Segmente, um einen minimalen Speicherschutz zu garantieren. Die logischen Adressräume des Kernel- und des Userspaces werden durch zwei verschiedene Segmente definiert. Die Abbildung 4.2 zeigt auf der linken Seite das Segment des Kernels, welches den ganzen linearen Adressraum ausser die ersten 128KB umfasst. Das User Segment geht von der physikalischen Adresse 1MB bis zum Ende des vorhandenen Speichers.

Segmente sind ein Relikt aus der Zeit, als es noch kein Paging gab. Speicherschutz kann mittels Paging viel eleganter und flexibler realisiert und Berechtigungen können feingranularer gesetzt werden. Die maximale Anzahl Segmente ist durch die Grösse der Global Descriptor Table limitiert. Für Paging gibt es keine solche Einschränkung.

Topsy v3 kennt, wie die meisten anderen Architekturen, keine Segmente. Es müsste also ein zusätzlicher Abstraktionslayer ins Memory Management von Topsy v3 eingeführt werden, um im ia32-Port Segmente zu benutzen.

Dies würde keine Vorteile bieten, weshalb ich mich gegen die Verwendung von Segmenten entschieden habe. Segmente können nicht wie Paging deaktiviert werden. Sie können aber so gewählt werden, dass sie den gesamten linearen Adressraum umfassen. Dann entspricht die logische immer der linearen Adresse.

4.1.3 Paging

Die Architektur des Paging-Mechanismus wurde durch Topsy v3 und die ia32-Architektur definiert. Die Herausforderungen lagen deshalb mehr in der Implementation.

4.2 Thread Management

Das Thread Management ist ein Schlüsselgebiet jedes Betriebssystems. Dabei stehen Sicherheit und Geschwindigkeit an oberster Stelle. Aus verschiedenen, im Kapitel 3.2 erläuterten Gründen musste es grundlegend überdacht und verändert werden.

4.2.1 Einführung

In Topsy v3 gibt es zwei grundsätzlich verschiedene Arten von Threads: *User Threads* und *Kernel Threads* (siehe Kapitel 2.2 und 2.3). Kernel Threads sind Teil des Kernels und haben dementsprechend mehr Privilegien. Während der regulären Ausführung laufen User Threads im unprivilegierten *User Mode*. Wenn eine Unterbrechung in Form eines Interrupts oder einer Ausnahmebedingung stattfindet, springt der Thread in den Interrupt Handler und wechselt in den privilegierten *Kernel Mode*. Kernel Threads laufen immer im Kernel Mode.

Unterbrechungen

Es gibt 4 verschiedene Klassen von Unterbrechungen:

1. *Timer Interrupt*: Der Timer löst periodisch einen Interrupt aus und unterbricht damit den laufenden Thread. Der Scheduler bestimmt dann, welcher Thread als nächster die CPU bekommt.

2. *Exceptions*: sind zumeist die Folge von fehlerhaftem Verhalten. Der Thread wird blockiert und meistens vom TM-Thread gelöscht. Bei einem Pagefault muss der MM-Thread über die Rechtmässigkeit des Zugriffs entscheiden und dann entweder die Page mappen und den Thread wieder aktivieren oder ihn beenden.

3. *IPC und Syscalls*: Um Nachrichten zu versenden oder zu empfangen, muss ein Thread einen speziellen Software-Interrupt auslösen. Beim Senden darf der Thread nachher weiter arbeiten. Beim Empfangen wird er blockiert, sofern die Nachricht noch nicht eingetroffen ist.

4. *Übrige Interrupts*: Für alle übrigen Interrupts, die zum Beispiel von der Tastatur oder anderen Geräten kommen, wird der entsprechende Handler ausgeführt und eine neue Scheduler-Entscheidung gefällt.

Ausführungszustände

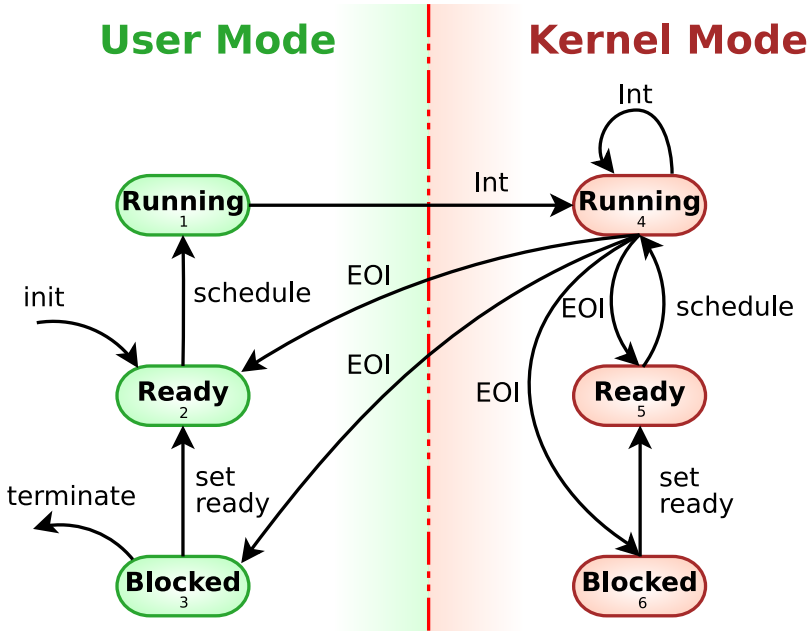


Abbildung 4.3: Mögliche Zustände eines User Threads

Jeder Thread im System ist in mindestens einem der in Abbildung 4.3 für User Threads dargestellten Zustände. Kernel Threads sind auch in den Zuständen (1) bis (3) im privilegierten Kernel Mode. Sie führen dann aber im Gegensatz zu den Zuständen (4) bis (6) nicht den Interrupt Handler, sondern den Code des Kernel Threads aus.

Die Pfeile in der Abbildung 4.3 kennzeichnen die möglichen Übergänge und ihre Bedingungen. Wenn ein neuer Thread kreiert wird, ist er im Zustand (2): ready. Wenn der Scheduler ihm die CPU zuteilt, wechselt er nach (1) und läuft, bis er durch eine Unterbrechung (Int) in den Interrupt Handler (4) im Kernel Mode gezwungen wird.

Bei einer Unterbrechung muss der Zustand des Threads (Kontext) gespeichert werden, um diesen nachher wieder herzustellen, damit der Thread nachher an der gleichen Stelle weiterarbeiten kann. Dieser Vorgang wird Kontext Wechsel genannt.

Das Ende der Abarbeitung der Unterbrechung ist im Diagramm mit End of Interrupt (EOI) bezeichnet. Der Thread wird entweder blockiert (3), wenn er momentan nicht lauffähig ist, oder als bereit markiert (2), so dass ihm der Scheduler bei Gelegenheit wieder eine CPU zuteilen kann (1).

In dieser Arbeit werden mit Nested Exceptions zwei weitere Zustände eingeführt. Neu können Threads im Zustand (4) unterbrochen werden. Dann muss, vergleichbar mit dem Übergang von (1) zu (4), ein weiterer Kontext gespeichert werden. Dies ist beliebig oft möglich, es müssen also dementsprechend viele Kontexte verwaltet werden können

4.2.2 Kontext Wechsel

Für Microkernel-Designs sind Kontext Wechsel besonders kritisch, da sie viel häufiger auftreten als bei monolithischen Systemen. Die Anforderungen an den Mechanismus bestehen in hoher Geschwindigkeit, Sicherheit und Unterstützung für beliebig viele Kontexte pro Thread.

Das neue Design sollte unnötige Kopieroperationen, wie sie in Topsy i386 vorkommen, vermeiden. Es sollte nur gesichert und wieder hergestellt werden, was auch wirklich gebraucht wird.

Zur Gewährleistung der Sicherheit darf kein Thread im User Mode auf Kontexte zugreifen können. Dies gilt auch für seinen eigenen. Kontexte müssen auf ihre Gültigkeit überprüft werden können.

Die ia32-Architektur bietet zwei Alternativen: hardware-unterstützte und software-gesteuerte Kontext Wechsel. Kontext Wechsel sind in Topsy i386 vollständig in Software implementiert. Die Unterstützung durch die Hardware speichert den gesamten Zustand des Prozessors und ist zum einen unflexibel und ersetzt zum andern die Software nicht vollständig, da zum Kontext mehr als die Register der CPU gehören. Die Realisierung in Software ist einfacher, bietet mehr Kontrolle über den Vorgang und wurde deshalb favorisiert.

4.2.3 Stacks

Die wichtigsten Bestandteile eines Threads sind neben dem Code und den Daten der Stack, die Thread-Datenstruktur des Betriebssystems und die Kontexte. Diese Daten sollten im Interrupt Handler möglichst früh zur Verfügung stehen. Dieser wird im Protection Domain des TM-Threads ausgeführt, denn nur er hat Zugriff auf die Informationen, welche zum Beispiel für das Scheduling benötigt werden.

In Topsy i386 hat jeder Thread einen Stack. Auf ihm wird bei einer Unterbrechung der Kontext gespeichert und nachher in die dafür vorgesehene Datenstruktur des TM-Threads kopiert. Dies ist kompliziert, da für Kernel Threads jedes Mal spezielle Anpassungen nötig sind, und langsam, da das Kopieren des Kontextes vermeidbar wäre. Dieses Konzept lässt sich nicht einfach auf Nested Exceptions erweitern.

Stacks werden in Topsy erst bei Bedarf gemappt. Wenn einer über den gemappten Bereich wächst, gibt es einen Page Fault. Der Memory Manager mappt dann die nächste Page. Dies könnte auch im Interrupt Handler passieren, sogar bevor der Kontext vollständig gespeichert ist. Dies gilt es zu vermeiden, da dann der Zustand des Threads nicht konsistent ist. Der ganze Stack muss also immer schon zu Beginn gemappt werden.

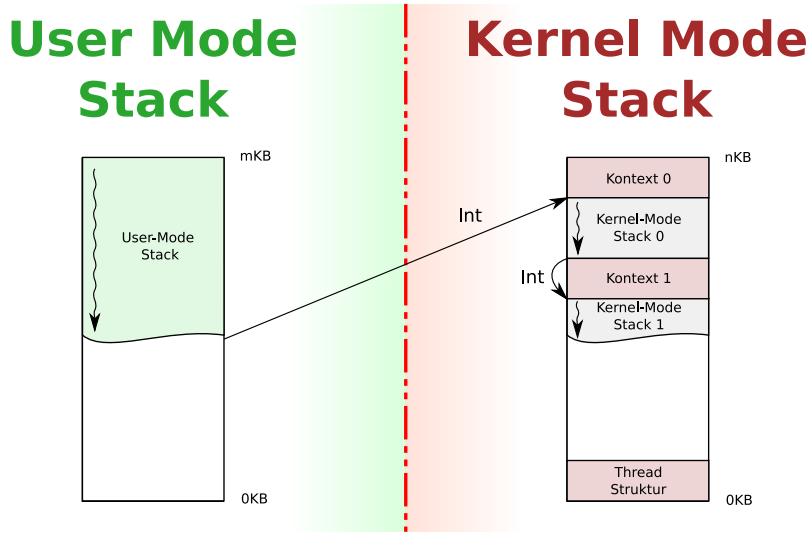


Abbildung 4.4: User Mode und Kernel Mode Stack eines Threads

Das gewählte Stackdesign wird in der Abbildung 4.4 illustriert: Durch die Verwendung eines separaten Kernel Mode Stacks werden alle oben genannten Probleme gelöst. Die ia32-Architektur kann bei einer Privilegienänderung automatisch auf einen anderen Stack wechseln. Dieser Stack ist auch der ideale Ort zum Speichern der Kontexte. Er ist vom Usermode nicht zugreifbar, die gespeicherten Kontexte können also gerade dort belassen werden. Der Kernel Code ist nicht rekursiv und benötigt nur einen kleinen Stack. Diesen schon zu Beginn zu mappen ist kein Problem.

Die Thread-Struktur wird neu am Ende des Stacks abgelegt. Da dieser an einem Vielfachen seiner Grösse ausgerichtet wird, kann im Kernel Mode jederzeit aus dem Stackpointer die Position der Thread-Struktur abgeleitet werden. Zu Beginn des Interrupt Handlers wird in die Protection Domain des TM-Threads gewechselt, da nur dort alle benötigten Daten verfügbar sind. Dass der Stack- und andere Pointer auf den Stack nach dem Wechsel nicht an einen anderen Ort zeigen, muss der Kernel Mode Stack im Protection Domain des Threads an die gleiche Stelle gemappt werden wie im TM-Thread.

Limitierungen

Ein Problem sind Stack Overflows im Kernel. Bei einem Stack Overflow im User Mode gibt es einen Page Fault, und der Thread wird vom MM-Thread beendet. Ein Stack Overflow auf dem Kernel Stack überschreibt zuerst die Thread Struktur, was vielfältige Probleme verursachen könnte. Es ist sogar vorstellbar, dass Topsy abstürzt. Dies ist aber vertretbar, da der Overflow nur durch einen Fehler im Kernel verursacht werden könnte.

4.2.4 Reentranter Interrupt Handler

Mit dem im Kapitel 4.2.3 beschriebenen Stack- und Kontextwechsel-Design sind wichtige Voraussetzungen für Nested Exceptions gegeben. Neu können nicht nur ein, sondern beliebig viele Kontexte verwaltet werden. Sobald der Kontext vollständig gespeichert ist, können im Interrupt Handler Nested Exceptions zugelassen werden.

Für jeden Kontext muss der Schedulerstatus gespeichert werden, welcher besagt, ob der Kontext wieder hergestellt werden darf oder ob er blockiert ist.

Der Scheduler selber interessiert sich nur für den Status des aktuellsten Kontextes jedes Threads und muss nichts von den übrigen wissen. Eine Ausnahme sind die Funktionen, welche den Schedulerstatus des Threads verändern wollen. Beispielsweise muss ein Thread aufgeweckt werden, wenn die Nachricht eintrifft, auf die er gewartet hat. Dies aber nur unter der Bedingung, dass der aktive Kontext auf die Nachricht wartet. Denn wenn der Thread zusätzlich wegen einer Nested Exception auf die Lösung eines Page Faults durch den MM-Thread wartet, darf nur der Schedulerstatus des auf die Nachricht wartenden Kontextes angepasst und der Thread selber auf keinen Fall geweckt werden, da der Page Fault ja zuerst noch behandelt werden muss.

Da Nested Exceptions in Topsy ein neues Konzept sind, musste dafür ein zusätzlicher Abstraktionslayer eingeführt werden. Dieser ist architektur-abhängig, da eventuell die entsprechenden architektur-abhängigen Kontexte verändert werden müssen.

4.3 Interprozesskommunikation

Interprozesskommunikation mittels Nachrichten ist ein häufig verwendetes Merkmal in Microkernel-Betriebssystemen und in Topsy einer der zentralsten Mechanismen. In Topsy v3 musste er, besonders für die Kommunikation zwischen Protection Domains und wie im Kapitel 2.3 beschrieben, grundlegend überarbeitet werden.

4.3.1 Speicherverwaltung

Wie schon angedeutet, finden das Versenden und Empfangen von Nachrichten in einem speziellen Interrupt Handler, dem Message Dispatcher, statt. Dies geschieht in der Protection Domain des TM-Threads, welcher nur Zugriff auf die vom Microkernel bereitgestellten Memory-Funktionen, aber nicht auf die Daten und den Code des MM-Threads hat.

Die Nachricht muss zuerst in die Protection Domain des TM-Threads gemappt werden. Später muss sie an die richtige Stelle in der PD des Zielthreads kopiert werden. Dazu muss die entsprechende Page des Zielthreads auch in den PD des TM-Threads gemappt werden.

Dafür werden in Topsy i386 fixe Pages im virtuellen Adressraum des TM-Threads verwendet. Dies funktioniert nicht mehr, wenn der Vorgang unterbrochen und von einem anderen Thread eine andere Nachricht gesendet werden kann. Während dem ganzen Message Dispatcher keine Interrupts zuzulassen, kommt nicht in Frage.

Eine Möglichkeit wäre, den MM-Thread zu bitten, die Pages an eine beliebige freie Stelle zu mappen. Das wäre sicher und würde auch funktionieren. Es wäre aber nicht sehr performant, da dazu eine Nachricht und ein Kontextwechsel zum MM-Thread und nachher eine Antwort und ein Kontextwechsel zurück nötig wären.

Die gewählte Lösung teilt dem TM-Thread zu diesem Zweck einen Teil seines virtuellen Speichers zur eigenen Verwaltung zu. Dieser wird in einem durch ein Lock geschützten Puffer verwaltet und nachher vom TM-Thread selber an die richtige physikalische Stelle gemappt. So kann er schnell und unkompliziert eine Page mappen. Der kritische Punkt ist die Grösse des Puffers, sie ist aber gut abschätzbar und muss nicht gross sein.

Kapitel 5

Implementation

Dieses Kapitel dokumentiert die Umsetzung des im Kapitel 4 aufgezeigten Designs. Zur besseren Lesbarkeit sind die zwei Kapitel gleich strukturiert. Die wichtigsten Konzepte werden hier erläutert, für eine abschliessende Referenz sei aber auf den kommentierten Quelltext verwiesen [8].

5.1 Übersicht

Als Einleitung in das folgende Kapitel beschreibe ich das Setup mit allen verwendeten Programmen und Tools. Im Unterkapitel 'Vorgehen' folgt ein kurzer Abriss des Ablaufs der Implementationsphase. Zuletzt will ich auf die Organisation des Quelltextes von Topsy v3 im Allgemeinen und der ia32-Portierung im Speziellen eingehen.

5.1.1 Setup

Auf meiner Entwicklungs- und Testmaschine lief Gentoo Linux[1] 2007.0 mit gcc[2] in der Version 4.1.2, binutils[3] v2.17 und make[4] v3.81.

Zum Testen des Codes verwendete ich primär bochs[5] (v2.3), da dieser einen sehr mächtigen Debugger integriert. Ich testete Topsy aber auch mit QEMU[6] (v0.9) und vor allem mit dem VMPlayer[7] (v1.1.2). Der VMPlayer ist gratis erhältlich und kann die gleichen Images abspielen wie die VMWare Workstation und alle anderen VMWare Produkte. Es fehlen ihm aber die Entwicklertools und die Möglichkeit, Images zu erstellen.

Genauere Infos zum Setup und eine Installationsanleitung finden sich im File `README.txt`[8].

5.1.2 Vorgehen

Nach der Einarbeitungszeit, welche ich vor allem für das Lesen der ia32-Spezifikationen [6] und das Vertrautwerden mit dem Topsy-Code ([1], [2], [3]) verwendete, begann ich mit der Arbeit an Topsy i386.

Zuerst waren Anpassungen nötig, dass es überhaupt mit den neuen GNU-Tools kompiliert. Nachher debuggte ich den Startup-Prozess, bis dieser wieder den architektur-unabhängigen Teil erreichte.

[1] Gentoo Linux: http://www.gentoo.org
[2] GNU Compiler Collection: http://gcc.gnu.org
[3] GNU Binutils: http://www.gnu.org/software/binutils
[4] GNU Make: http://www.gnu.org/software/make
[5] Bochs: http://bochs.sourceforge.net/
[6] QEMU: http://www.qemu.org/
[7] VMWare Player: http://www.vmware.com/products/player/
[8] `README.txt` auf der beiliegenden CD

Um nicht zu viel Zeit mit der alten Codebasis aufzuwenden, arbeitete ich dann an Topsy v3 weiter. Der Boot-Code konnte beinahe unverändert übernommen und weiter debugged werden. Dann musste das alte Memory Management entfernt und der HAL des neuen implementiert werden.

Dann kümmerte ich mich um den lokalen APIC, die Ersetzung des alten Timers durch denjenigen des lokalen APICs und schliesslich noch um den I/O-APIC.

Die meiste Arbeit dann in die Überarbeitung des Thread Management, das neue Stacklayout, den reentranten Interrupt Handler und die entsprechenden Anpassungen der Interprozesskommunikation und des Schedulers ein.

5.1.3 Organisation des Quelltextes

Der Quelltext von Topsy ist nach den Subsystemen organisiert. Abbildung 5.1 zeigt den Inhalt des Verzeichnisses `Topsy-v3_P4`.

```
Topsy-v3_P4
|-- Boot            : Architektur-abhängige Boot-Prozesse
|-- Config          : verschiedene Konfigurationsfiles (d)
|-- Docu            : Dokumentation (d)
|-- IO              : Input/Output
|-- Makefiles       : Makefiles (d)
|-- Memory          : Altes Memory Management (Topsy v1)
|-- Net             : Netzwerk-Stack (*)
|-- Startup         : Architektur-unabhängiger Topsy-Startup
|-- Threads         : Thread-Management
|-- ThreadsLight    : Lightweight Threads (*)
|-- Tools           : Tools fürs Builden von Topsy
|-- Topsy           : Konfiguration und Libraries
|-- User            : User Applikationen, zB die Shell
`-- VMemory         : Neues Virtual-Memory Management
```

`(*)` : Nicht Gegenstand dieser Arbeit
`(d)` : Enthält keinen Code

Abbildung 5.1: Organisation des Quelltextes: `Topsy-v3_P4`

Alle in der Abbildung 5.1 aufgeführten Verzeichnisse mit Code, enthalten die architekturunabhängigen Files und für jede Architektur ein Unterverzeichnis mit den architekturabhängigen Files. Im Folgenden ist jeweils nur das diese Arbeit betreffende Unterverzeichnis `ia32` aufgeführt.

Boot

```
Topsy-v3_P4/Boot/
`-- ia32
    |-- LowCore             : Boot-Code
    |   |-- CoreBoot        : Code des MBRs, springt in CoreLoad
    |   |   `-- ...
    |   |-- CoreLoad        : Initialisiert HW und lädt Topsy
    |   |   `-- ...
    |   `-- Include         : Includes
    |       `-- ...
    |-- Tools
    |   |-- CreateNetImg
    |   |   `-- ...
    |   |-- KernPatch       : Alter Binary-Patcher
    |   |   `-- ...
    |   |-- Keyboard        : Erstellt Tabelle für Keyboard-Treiber
    |   |   `-- ...
    |   `-- ToolLib
    |       `-- ...
    `-- link.ia32.scr       : Linkskript für Kernel Image
```

Abbildung 5.2: Organisation des Quelltextes: `Topsy-v3_P4/Boot/`

Das Verzeichnis `Boot` (Abb. 5.2) enthält den architektur-abhängigen Bootloader und das Linkskript für das Kernel Image.

IO

```
Topsy-v3_P4/IO/
|-- Drivers                  : Architektur-unabhängige Treiber
|   |-- Ethertap.{c,h}       :
|   `-- Loopback.{c,h}       : Loopback-Treiber
|-- IO.h                     : Include
|-- IOConsole.{c,h}          : Textausgabe auf Konsole
|-- IODevTable.h             : Struktur der Device-Tabelle
|-- IODevice.{c,h}           : Main der Device-Threads
|-- IOMain.{c,h}             : IO-Thread
`-- ia32
    |-- Debug.{c,h}          : Debug
    |-- Drivers              : Architektur-abhängige Treiber
    |   |-- HWKeyTable.S
    |   |-- IOPci*.c         : PCI-Treiber
    |   |-- KeyCodeTable.S
    |   |-- Keyboard.{c,h}
    |   |-- TTY.{c,h}        : Konsolen-Treiber
    |   |-- Video.{c,h}      : Video-Treiber
    |   |-- classlist.h
    |   |-- devlist.h
    |   |-- keyboard.{S,h}
    |   |-- names.c
    |   |-- pci.ids
    |   |-- performanceTest.{c,h}
    |   `-- video.{S,h}
    |-- IODevTable.c         : Tabelle der Geräte-Konfigurationen
    |-- IOHal.{c,h}
    `-- Tools
        `-- gen-devlist.c
```

Abbildung 5.3: Organisation des Quelltextes: `Topsy-v3_P4/IO/`

Im Verzeichnis `IO` (Abb. 5.3) sind alle Gerätetreiber und der IO Kernel Thread gespeichert.

Memory

```
Topsy-v3_P4/Memory/
|-- MMHeapMemory.{c,h}
|-- MMHeapMemory_new.c
|-- MMInit.{c,h}
|-- MMMain.{c,h}
|-- MMVirtualMemory.{c,h}
|-- Memory.h
`-- ia32
    |-- MMDirectMapping.c
    |-- MMError.{c,h}
    |-- MMHal.{c,h}          : GDT-, IDT-, TSS-Verwaltung
    |-- MMMapping.h
    |-- MemoryLayout.h       : Konfiguration des Memory-Management
    `-- mmHalAsm.{S,h}       : Diverse Assembler-Routinen
```

Abbildung 5.4: Organisation des Quelltextes: `Topsy-v3_P4/Memory/`

Das Memory Management von Topsy v1 wird bis auf die beschriebenen Files nicht mehr verwendet. Es ist im Verzeichnis `Memory` (Abb. 5.4).

Startup

```
Topsy-v3_P4/Startup/
|-- Startup.{c,h}       : topsyMain(), erste arch-unabhängige Funktion
|-- headerLayout.h      : Topsyheader
`-- ia32
    |-- BinaryFile.h    : Alter BinFileDesc, abgelöst durch topsyheader
    |-- Init.{c,h}      : Initialisiert HAL und springt in topsyMain()
    `-- start.S         : Bildet topsyheader des Binary
```

Abbildung 5.5: Organisation des Quelltextes: `Topsy-v3_P4/Startup/`

`topsyMain()` und der architektur-unabhängige Startupcode befinden sich im Verzeichnis `Startup` (Abb. 5.5).

Threads

```
Topsy-v3_P4/Threads
|-- TMIPC.{c,h}                 : IPC Funktionen für Kernel
|-- TMInit.{c,h}                : Initialisiert TM und startet
|                                   die Kernel Threads
|-- TMMain.{c,h}                : Thread-Management-Thread
|-- TMProtectionDomain.{c,h}    : Protection Domain
|-- TMScheduler.{c,h}           : Scheduler
|-- TMThread.{c,h}              : Erstellen, Verwalten und Löschen
|                                   von Threads
|-- TMTime.{c,h}
|-- TMTimer.{c,h}               : Alter Timer
|-- TMTimerNG.{c,h}             : Sendet Nachricht, wenn bestimmte
|                                   Zeit abgelaufen
|-- Threads.h
|-- config.in
|-- ia32
|   |-- TM8254Timer.c           : Funktionen des alten Timers
|   |-- TM8254TimerAsm.S        : Assembler Routinen
|   |-- TMClock.{c,h}           : Verwaltet Echtzeit-Uhr
|   |-- TMClockAsm.S            : Assembler-Routinen der RTC
|   |-- TMError.{c,h}           : Error- und Exception-Handler
|   |-- TMHal.{c,h}             : HAL des Thread Management
|   |-- TMHalApic.{c,h}         : APIC und I/O-APIC Implementation
|   |-- TMHalApicAsm.{S,h}      : Assembler Routinen
|   |-- TMHalAsm.{S,h}          : Assembler Routinen
|   `-- TMSMPHal.{c,h}          : Stubs für SMP
`-- tm-include.h                : Strukturen des Thread Managements
```

Abbildung 5.6: Organisation des Quelltextes: `Topsy-v3_P4/Threads`

Die gesamte Thread Verwaltung, der Scheduler und der Interrupt Handling Code sind im Verzeichnis `Threads` (Abb. 5.6).

Topsy

```
Topsy-v3_P4/Topsy/
|-- Assert.{c,h}            : Assert-Makros
|-- Configuration.h         : Konfiguriert Topsy
|-- Error.{c,h}             : Error Handling
|-- HashList.{c,h}
|-- Lifo.{c,h}
|-- List.{c,h}
|-- Lock.{c,h}              : Lock Implementation
|-- Messages.h              : Alle Nachrichtenformate
|-- Socket.{c,h}
|-- Support.{c,h}           : Diverses
|-- Syscall.{c,h}           : Syscalls
|-- SyscallMsg.h
|-- System.{c,h}
|-- Topsy.h
|-- Version.h
|-- doubleq.h
`-- ia32
    |-- CoreLoad.h
    |-- Endian.h
    |-- Exception.{c,h}
    |-- Limits.h
    |-- PrintReg.c
    |-- SupportAsm.{S,h}    : Assembler Routinen für Support.c
    |-- SyscallMsg.S        : Implementiert Syscall zum Senden
    |                           und Empfangen von Nachrichten
    |-- Tools.{c,h}
    |-- asm.h
    |-- cpu.h               : Arch-abhängige Konfiguration
    `-- limits.h
```

Abbildung 5.7: Organisation des Quelltextes: `Topsy-v3_P4/Topsy/`

Das Verzeichnis `Topsy` (Abb. 5.7) beinhaltet diverse Headerfiles mit Konfigurationsoptionen, Datenstrukturen wie Listen und Locks und die Kernel Library.

User

```
Topsy-v3_P4/User/
|-- CrashMe.{c,h}                    : Crash-Tests
|-- Demo.{c,h}                       : Demo
|-- HeapAllocator.{c,h}              : Heap Allocator für User Threads
|-- HeapFreeTree.{c,h}
|-- HelloWorld.{c,h}                 : Hello World
|-- IRQControl.{c,h}
|-- Licence.{c,h}                    : Zeigt GPL
|-- Lock.{c,h}
|-- PacketDriverTest.{c,h}
|-- PacketFraming.{c,h}
|-- PacketStackControl.{c,h}
|-- PacketStackControl_loopback.c
|-- PacketStackGlobals.h
|-- Packetizer.{c,h}
|-- SeatReservation.{c,h}            : Demo Reservationssystem
|-- Shell.{c,h}                      : Shell
|-- StartMeFirstUser.c               : Initialisiert und startet Shell
|-- ThreadsDemo.{c,h}                : Thread Demo
|-- UserSupport.{c,h}
|-- datafile.{c,h}
|-- ia32
|   `-- userStart.S                  : topsyheader, ia32-Start
|-- link.ia32.scr                    : Link-Skript des User Images
|-- link.ixp2400.scr
|-- link.linux.scr
|-- link.mips.scr
|-- link.mips.simos.scr
|-- link.sa1100.scr
|-- link.solaris.scr
|-- malloc_api.{c,h}                 : malloc
|-- testtimer.{c,h}
|-- tmTimerTest.{c,h}
`-- ualloc_api.c
```

Abbildung 5.8: Organisation des Quelltextes: `Topsy-v3_P4/User/`

Der gesamte Code des User Images wie die Shell und die Applikationen sind im Verzeichnis `User` (Abb. 5.8). Auch das Linkskript des User Images und `userStart.S`, welches den Topsy-header des User Images bildet, sind dort gespeichert.

VMemory

```
Topsy-v3_P4/VMemory/
|-- HeapAllocator.{c,h}          : Heap Implementation
|-- HeapFreeTree.{c,h}
|-- ia32
|   |-- vm_bootup_hal.c          : Initialisiert HAL-Layer
|   |-- vm_hal.{c,h}             : Hilfsfunktionen des HAL-Layer
|   |-- vm_pmap.c                : Implementiert pmap-Layer
|   |-- vm_pmap_asm.{S,h}        : Assembler Routinen
|   |-- vm_pmap_exc.c            : Exception-Management
|   `-- vm_pmap_hal.h            : Strukturen des HAL-Layers
|-- kalloc_api.{c,h}
|-- steal_alloc.{c,h}
|-- vm.h
|-- vm_alloc.{c,h}
|-- vm_anon.{c,h}
|-- vm_bootup.{c,h}              : Initialisiert VM
|-- vm_calls.{c,h}               : Implementiert VM-Interface
|-- vm_dmap.{c,h}
|-- vm_domain.{c,h}
|-- vm_extern.h
|-- vm_file.{c,h}
|-- vm_hash.{c,h}
|-- vm_loader.{c,h}
|-- vm_loan.{c,h}
|-- vm_main.{c,h}                : Memory Management Thread
|-- vm_map.{c,h}
|-- vm_object.{c,h}
|-- vm_page.{c,h}
|-- vm_pmap.h                    : PMap Interface
`-- vm_region.h
```

Abbildung 5.9: Organisation des Quelltextes: `Topsy-v3_P4/VMemory/`

Das Memory Management von Topsy v3 und der MM-Thread befinden sich im Verzeichnis `VMemory` (Abb. 5.9).

5.2 Memory Management

Alle am Memory Management vorgenommenen Änderungen konzentrieren sich, abgesehen von Bugfixes, auf die Files im Verzeichnis `Topsy-v3_P4/VMemory/ia32` (Abb. 5.9).

5.2.1 Memory Management in Topsy v3

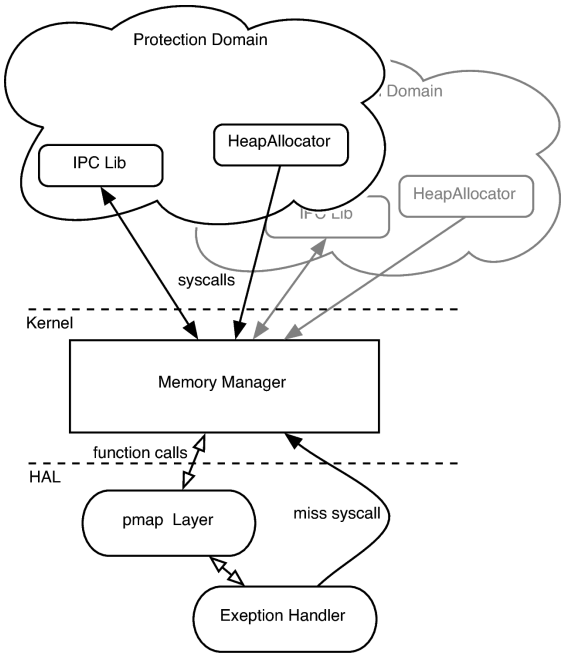


Abbildung 5.10: Aufbau des Memory Management in Topsy v3 (aus [3])

Abbildung 5.10 zeigt den Aufbau des objektorientierten Memory Management von Topsy v3. Die Threads der verschiedenen Protection Domains greifen über die Bibliotheken des Heap Allocators oder des IPCs mittels Syscalls auf den Memory Management Threads zu.

Der MM-Thread benützt dann den pmap-Layer des HALs. Dieser kümmert sich um das Mapping der Pages von virtuellen auf physikalische Adressen. Diese Funktionalität braucht auch der Exception Handler, dessen Architektur im Kapitel 4.2 genauer beschrieben ist.

In [3] ist das pmap-Interface beschrieben. Es besteht aus 12 Funktionen, welche für die ia32-Architektur implementiert werden mussten.

5.2.2 Paging

Die Implementation des eigentlichen Paging beinhaltet die Verwaltung der Page Directories und der Page Tables, Funktionen zur Aktualisierung des Page-Caches (Translation Lookaside Buffer, TLB) und der Page Fault Handler.

Die ia32-Architektur kennt sowohl 4KB als auch 4MB grosse Pages. Da Topsy relativ wenig Memory braucht, habe ich mich für die 4KB-Option entschieden.

Verwaltung der Page Directories und Page Tables

Wie in den Abbildungen 4.1 und 5.11 ersichtlich, sind die Page Directories und Page Tables im Wesentlichen 4KB grosse Tabellen mit 1024 Pointern auf Page Tables respektive physikalische Pages. Die Tabellen werden bei Bedarf dynamisch alloziert und wieder freigegeben.

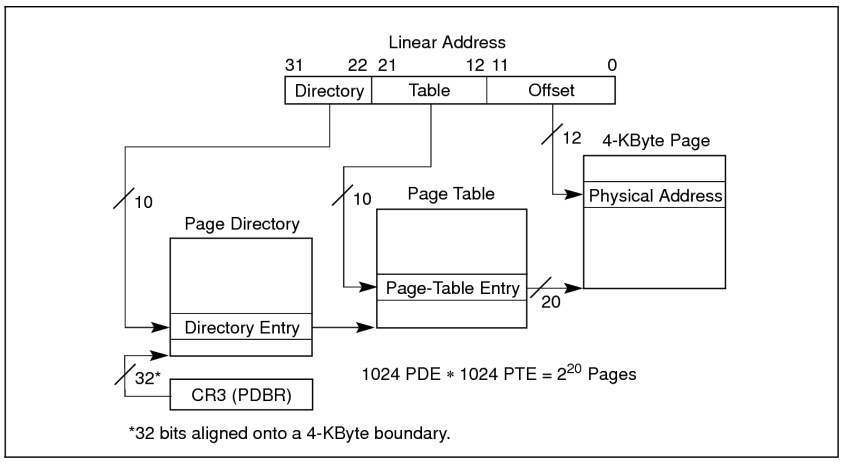


Abbildung 5.11: Lineare Adressübersetzung für 4KB grosse Pages (aus [6])

Dafür wird beim Startup mittels `vm_allocate_bootup_memory()` ein direkt gemappter Bereich der Grösse `pmap_buf_size` alloziert. Dessen virtuelle beziehungsweise physikalische Anfangsadresse wird in `pmap_vbase` respektive `pmap_pbase` gespeichert.

```
     +-------------+  TOP = pmap_vbase + pmap_buf_size
12   |  PTbl 1     |
11   |  PTbl 2     |
10   | ->next free |  pmap_free_ptbl = 10
 9   |  PTbl 3     |  pmap_ptbl_seg_bottom = 9
 8   |             |
 7   :             :
 6   :             :
 5   |             |
 4   |  PDir 3     |  pmap_pdir_seg_top = 4
 3   |  PDir 2     |
 2   | ->next free |  pmap_free_pdir = 2
 1   |  PDir 1     |
     +-------------+  BOTTOM = pmap_vbase (pmap_pbase)
```

Abbildung 5.12: Beispiel der Benutzung des pmap-Buffers

Dieser Bereich wird nun ‘von oben’ mit Page Tables und ‘von unten’ mit Page Directories gefüllt. Der Füllstand wird mit den Variablen `pmap_ptbl_seg_bottom` und `pmap_pdir_seg_top` nachgeführt. Wenn innerhalb der Bereiche eine Tabelle entfernt wird, wird sie in die entsprechende Freelist eingetragen. Die Abbildung 5.12 zeigt beispielhaft eine mögliche Belegung und die dazugehörigen Werte der Variablen. Der Buffer hat zwölf Slots à 4KB, wobei sechs davon belegt sind.

Da sowohl Page Tables als auch die Pages selber an der 4KB-Grenze ausgerichtet sein müssen, können die niederwertigen 12Bits der 32Bit grossen Einträge für Flags verwendet werden. Deren Format ist in der Abbildung 5.13 dargestellt.

Topsy kennt Flags für lesbar, schreibbar, ausführbar, cachbar und bufferbar. Diese können eins zu eins auf die verfügbaren Flags gemappt werden. Die einzige Ausnahme sind Pages, die laut den Topsy Flags weder lesbar noch schreibbar sind. Solche werden als Supervisor-Pages gemappt, welche nur im Kernel Modus lesbar und schreibbar sind. Diese Praxis ist konsistent mit den anderen Architekturen, es wäre dennoch sinnvoll, ein zusätzliches Flag für Pages einzuführen, welche nur vom Kernel Mode aus lesbar und schreibbar sein dürfen.

Die Tabelle 5.1 listet den kombinierten Effekt der Berechtigungs-Flags des Page Directory und der Page Table auf. Die Implementation kennzeichnet alle Page Directory Einträge als lesbar, schreibbar und User. Die effektiven Berechtigungen der Pages können so vom Page Table Eintrag gesetzt werden.

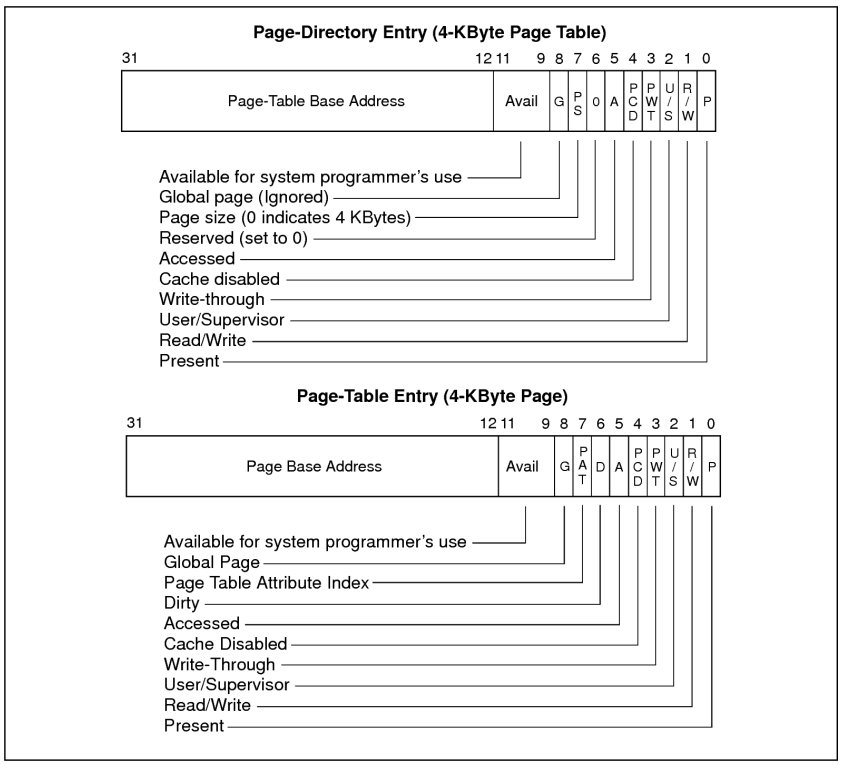


Abbildung 5.13: Format der Page-Directory und Page-Table Einträge (aus [6])

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	**Access Type**	**Privilege**	**Access Type**	**Privilege**	**Access Type**
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

Tabelle 5.1: Kombination von Page Table und Page Flags (aus [6])

Exception Handler

Die Exception Handler befinden sich im File `Topsy-v3_P4/VMemory/ia32/vm_pmap_exc.c`. Der prominenteste Vertreter ist der Page Fault Handler, er führt folgende Aktionen aus:

1. Er findet die Adresse, welche die Ausnahmebedingung verursachte. Sie steht im Register `%cr2` der CPU.
2. Er ermittelt die Art des Fehlers. Die Flags des Fehlercodes auf dem Stack weisen auf die

Ursache hin.

3. Er blockiert den Thread und sendet die gesammelten Daten an den MM-Thread.

4. Der MM-Thread mappt dann die Page, sofern der Zugriff gültig war. Andernfalls beendet er den fehlerhaften Thread.

Nummer	Name	Ursache
T_EX_05	BOUND Range Exceeded Exception	Array-Index ausserhalb des gültigen Bereichs
T_EX_09	Coprocessor Segment Overrun	Page- oder Segment-Fehler im Intel 387 Mathematik Coprozessor, nur in Intel386-CPU basierten Systemen
T_EX_0A	Invalid TSS Exception	Problem mit Task State Segment
T_EX_0B	Segment Not Present	Segment nicht vorhanden
T_EX_0C	Stack Fault Exception	Limit überschritten oder Stack Segment ist nicht verfügbar
T_EX_0E	Page Fault	Page Fault
T_EX_11	Alignment Check Exception	Wenn Alignment Check aktiviert ist

Tabelle 5.2: Memory-Exceptions

Die anderen Memory-Exceptions sind in der Tabelle 5.2 aufgeführt und beenden den verursachenden Thread sofort.

5.2.3 Segmente

Auch wenn Segmente in Topsy v3 nicht mehr zur Partitionierung des Speichers und zum Speicherschutz verwendet werden, sind sie wichtig. Das Privilegien-Feld des aktuellen Code-Segments definiert, mit welchen Privilegien der Code läuft. Aus diesem Grund benötigen Kernel und User Threads verschiedene Code-Segmente.

In Assembler-Code werden gerne Segmente zur vereinfachten und sichereren Programmierung verwendet, wenn nur in einem bestimmten Bereich operiert wird. Dazu dienen die Segmente ‘OS Video’, welches das VGA-Memory umfasst, ‘OS GDT’, mit der Global Descriptor Table und ‘IDT Alias’, welches die Interrupt Descriptor Table beinhaltet.

Die dritte Kategorie sind die speziellen Segmente. Das ‘BSP TSS’ beinhaltet das Task State Segment des Bootstrapprocessors. Dieses Segment wird für Kontextwechsel des Prozessors verwendet, welcher das System bootet. Im Multicore Betrieb muss für jede weitere CPU ein eigenes TSS angelegt werden. Dafür kann die Funktion `int pmap_get_new_tss(unsigned short *tss_sel, unsigned short *tss_alias, struct protection_domain *d)` verwendet werden.

Selektor	Verwendung	Start	Grösse
0x00	Dummy	–	–
0x08	OS Code	0x0	4GB
0x10	OS Daten	0x0	4GB
0x18	OS Global	0x0	4GB
0x20	OS Video	0xB8000	0xFBF
0x28	OS GDT	0x1000	64KB
0x30	IDT Alias	0x10000	4KB
0x38	BSP TSS	0x1E000	104B
0x40	BSP TSS Alias	0x1E000	104B
0x48	User Code	0x0	4GB
0x50	User Data	0x0	4GB

Tabelle 5.3: Segmente in Topsy v3

5.2.4 Syscalls

Es war nötig, einige neue Syscalls zu implementieren und alte zu erweitern. Die Änderungen betreffen den MM-Thread und wurden in `VMemory/vm_main.c` vorgenommen.

Der MM-Thread unterstützt folgende neuen Nachrichtentypen:

- **`VM_MAP_FIND_NEAR`**
  Findet im PD mit der Id `pdId` eine Region `region` der Grösse `size` in der Nähe des Pointers `hint`.

  Aufruf: In der Enveloppe müssen folgende Argumente gespeichert werden: `ThreadId pdId, address_t hint size_t size, Region_t region` und `int direction`.

  Rückgabe: Die Rückgabe enthält einen Fehlercode. Wenn der Aufruf erfolgreich war, ist dieser gleich `VM_OK` und die Pointer zeigen auf den Start und das Ende der gefundenen Region:
  ```
  id = VM_REPLY
  reply.msg.vm_reply.err : Fehlercode
  reply.msg.vm_reply.ptr1 : region.start
  reply.msg.vm_reply.ptr2 : region.end
  ```

- **`VM_REGION_SHARE`**
  Macht die Region `region` aus dem PD `sourcePdId` auch im PD `destPdId` in der Nähe von `hint` verfügbar.

  Aufruf: In der Enveloppe müssen folgende Argumente gespeichert werden: `ThreadId sourcePdId, ThreadId destPdId, Region_t region, address_t hint, int flags` und `int direction`

  Rückgabe: wie oben.

- **`VM_REGION_ADD_NEAR`**
  Fügt dem PD `pdId` die Region `region` in der Nähe von `hint` hinzu.

  Aufruf: In der Enveloppe müssen folgende Argumente gespeichert werden: `ThreadId pdId, Region_t region` und `address_t hint`

  Rückgabe: wie oben.

- **`VM_REGION_ENTER`**
  Stellt sicher, dass die Region, in welchem `hint` im PD `pdId` liegt, im pmap-Layer eingetragen ist. Operationen auf der Region können dann keine Page Faults mehr provozieren.

  Aufruf: Der gewünschte PD wird in `msg.msg.vm_get.pdId` und der Pointer in `msg.msg.vm_get.hint` gespeichert.

  Rückgabe: Fehlercode in `VM_REPY`.

5.3 Thread Management

Beim Thread Management mussten nicht nur die HAL-Files in `Topsy-v3_P4/Threads/ia32`, sondern auch architektur-unabhängige Files verändert werden. Nachfolgend werden die Implementationen des Stackmanagements, des Interrupt Handlers und der APICs erklärt.

5.3.1 Stacks

Das Stackmanagement von User und Kernel Threads unterscheidet sich fundamental. User Threads besitzen zwei Stacks, zwischen denen bei Privilegienänderungen gewechselt wird. Kernel Threads laufen immer im Kernel Mode und benötigen daher nur einen Stack. In diesem Unterkapitel wird deshalb zuerst das Stackmanagement der User Threads und nachher das der Kernel Threads beschrieben.

User Threads

Die Abbildung 5.14 zeigt die Stacks eines User Threads und sinnbildlich für seinen Ablauf den Verlauf des Stackpointers. Auf der linken Hälfte befindet sich der 265KB grosse User Mode Stack (UMS) und auf der anderen Seite der viel kleinere, 8KB grosse Kernel Mode Stack (KMS). Zudem besitzt die CPU ein Task Register, über welches das Task State Segment (TSS) referenziert wird. Dies speichert das Stacksegment `ss0` und den Stackpointer `esp0` des KMS des aktuellen Threads.

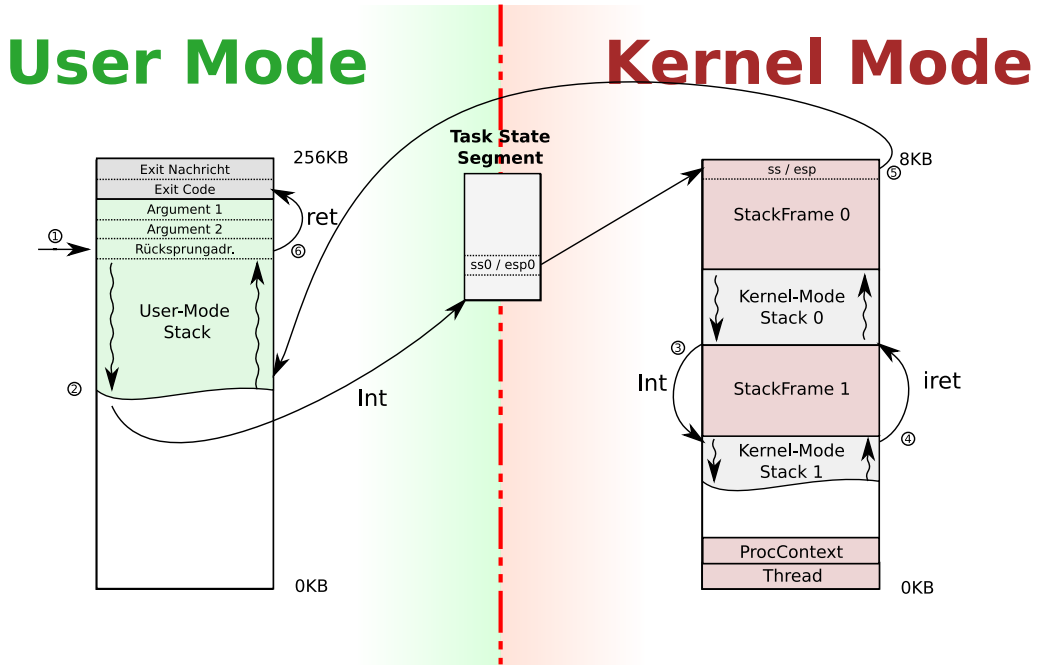


Abbildung 5.14: User Mode- und Kernel Mode Stack eines User Threads

Ablauf eines User Threads: Die Nummerierung entspricht derjenigen der Abbildung 5.14.

1. Wenn der Thread im User Mode gestartet wird, zeigt sein Stack-Pointer auf die Position `(1)`. Dort steht eine Rücksprungadresse (mehr dazu unter Punkt 6.). Dem Thread können zwei 4 Byte grosse Argumente übergeben werden.

2. Der Thread startet seine Ausführung, wobei der UMS wächst. Bei der Position `(2)` wird er durch einen Interrupt unterbrochen.

   Auf Grund des Privilegienwechsels benützt die CPU nun automatisch den im TSS angegebenen KMS und speichert dort die Position des alten Stacks und den Zustand, genannt `Stack Frame`, des eben unterbrochenen Threads.

   Die Funktion `__get_thread_from_esp()` liefert im Kernel Mode jederzeit die Thread-Struktur des aktuell laufenden Threads. Da alle Kernel Stacks an der 8KB-Grenze ausgerichtet sind, muss dafür nur der Stackpointer abgerundet werden. Dieser Mechanismus wurde von Linux entlehnt.

   Der Interrupt Handler läuft nun im Kernel Mode und wechselt in den Protection Domain des TM-Threads. Interrupts sollten möglichst schnell abgearbeitet werden, da sonst nachfolgende verzögert werden können. Deshalb erhöht der Handler seine Scheduling Priorität auf `INTHANDLER_PRIORITY`, was typischerweise die höchste verfügbare ist.

3. Sobald das Stack Frame vollständig aufgebaut ist, können weitere Interrupts zugelassen werden. Der Interrupt Handler bearbeitet den Interrupt, bis er im Punkt `(3)` wieder unterbrochen wird.

Nun findet kein Privilegien- und deshalb auch kein Stack-Wechsel statt. Aus diesem Grund muss auch keine alte Stack-Position gespeichert werden. Das neue Stack Frame wird wie vorher aufgebaut, es fehlen aber die Werte `ss` und `esp`. Ansonsten unterscheiden sich verschachtelte Unterbrechungen nicht von gewöhnlichen.

Anmerkung: Bei Unterbrechungen, welche keinen Privilegienwechsel nach sich ziehen, fehlen `ss` und `esp` im Stack Frame. Dies ist namentlich bei Nested Exceptions und Unterbrechungen von Kernel Threads der Fall. Deshalb kann im Allgemeinen nicht davon ausgegangen werden, dass diese zwei Felder der zugehörigen Datenstruktur `ProcStackFrame` auf gültige Werte zeigen.

4. Der Interrupt Handler ist beim Punkt `(4)` fertig. Er prüft nun die Lauffähigkeit des Stack Frames 1 anhand der darin enthaltenen Scheduling Informationen und aktualisiert die Scheduling Informationen des Threads.

   Der Scheduler bestimmt nun den nächsten Thread, welcher die CPU bekommt. Sobald der hier besprochene Thread an der Reihe ist, wird sein Stack Frame 1 wieder hergestellt, Interrupt Return (`iret`) ausgeführt und der erste Interrupt Handler läuft weiter.

5. An der Position `(5)` passiert genau dasselbe wie bei `(4)`. Insbesondere wird die alte Priorität wieder hergestellt. Die `iret` Instruktion erkennt anhand des Codesegments im Stack Frame, dass ein Privilegienwechsel ausgeführt wird, und popt dann auch `ss` und `esp` vom Stack.

   Der User Thread fährt nun an der gleichen Stelle weiter, an der er unterbrochen wurde.

6. Wenn die Hauptfunktion des Threads zurückkehrt (Punkt `(6)`), springt das Programm in den `Exit Code`, welcher ebenfalls auf dem Stack des Threads liegt. In diesem wird die `Exit Nachricht` an den TM-Thread gesendet, welcher den Thread aus dem System entfernt.

Kernel Threads

Kernel Threads unterscheiden sich von User Threads nur durch ihren Aufbau. Beim Erstellen von Threads muss zwischen User und Kernel Threads unterschieden werden. Während der Laufzeit ist dies aber transparent, die Abläufe sind genau gleich.

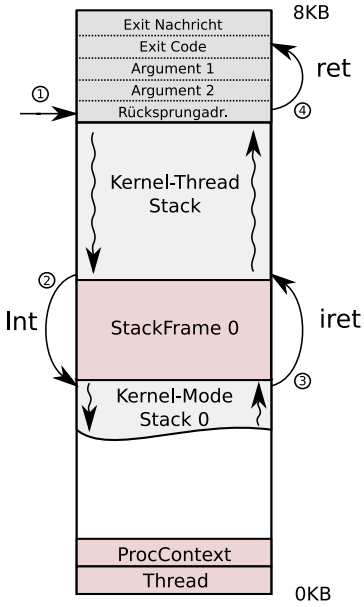


Abbildung 5.15: Kernel Thread-Stack eines Kernel Threads

Ablauf eines Kernel Threads: (siehe Abbildung 5.15)

1. Auch den Kernel Threads können zwei Argumente übergeben werden, und der Stack Pointer zeigt beim Start auf die Rücksprungadresse bei Position `(1)`. Wie schon angedeutet, besitzen Kernel Threads aber nur einen Stack, den Kernel Thread Stack (KTS). Der KTS ist wie der KMS der User Threads 8KB gross.
2. Ein Interrupt eines Kernel Threads sieht genau gleich aus wie eine verschachtelte Unterbrechung eines User Threads.
3. Auch die Rückkehr vom Interrupt unterscheidet sich nicht von der eines User Threads.
4. Wenn der Kernel Thread fertig ist, wird er mit dem gleichen Mechanismus beendet wie ein User Thread.

Stack Management

Das Neue Stack Management bedingte einige Änderungen am architektur-unabhängigen Teil und macht insbesondere das Stack Management selber architektur-abhängig.

Die unten aufgeführten Funktionen müssen für alle anderen Architekturen noch implementiert werden. Im zweiten Abschnitt der Funktionsbeschreibung ist jeweils die Lösung der ia32-Architektur beschrieben.

Mögliche Werte von `flags` sind `TM_KERNEL`, wenn ein Kernel Thread erstellt werden soll, und `TM_INIT`, wenn die Kernel Threads noch nicht laufen. Wenn die Kernel Threads noch nicht laufen, dürfen die Funktionen des Memory Managers direkt aufgerufen werden, ansonsten müssen die Syscalls des MM-Threads verwendet werden.

Die Funktionen liefern `TM_OK`, wenn sie erfolgreich waren, und sonst `TM_FAILED`.

- **`int tmHalInit()`**
  Diese Funktion initialisiert den HAL-Layer des Thread Management. Hier können Datenstrukturen initialisiert werden. Diese Funktion wird von `tmInit()` aufgerufen, der Heap Allocator ist also bereits initialisiert.

  ia32: Es gibt hier nichts Wesentliches zu tun.

- **`int tmHalInitThread(Thread* *threadPtr, ThreadId destPDId, ThreadId tmPDId, Address* stackBaseAddress, Address* stackBaseMappedAt, size_t* stackSize, int flags)`**
  Diese Funktion alloziert die Thread Struktur und den Stack und gibt die entsprechenden Werte zurück.

  ia32: Diese Funktion erstellt das im Kapitel 5.3.1 besprochene Layout. Dabei wird der Kernel Mode Stack im TM-Thread und im Zielthread an die gleiche Stelle gemappt. Auch der User Mode Stack muss in den PD des TM-Threads gemappt und der Pointer `stackBaseMappedAt` entsprechend gesetzt werden.

  Dann werden die Datenstrukturen `Thread` und dessen architektur-abhängiger Teil `ProcContext` angelegt und initialisiert.

  Ein TSS muss nicht angelegt werden, da nicht pro Thread, sondern pro CPU ein TSS vorhanden ist.

- **`int tmHalCleanupAfterThreadBuild(Thread* threadPtr, Address stackBaseMappedAt, int flags)`**
  Wenn der Thread erstellt und lauffähig ist, werden eventuell bestimmte Ressourcen nicht mehr gebraucht, welche in `tmHalInitThread()` alloziert wurden. Diese können jetzt freigegeben werden.

  ia32: Wenn der TM-Thread in einem anderen PD als der Zielthread ist, kann dessen Stack aus dem virtuellen Speicher des TM-Threads gelöscht werden.

- **int tmHalDestroyThread(Thread* threadPtr, ThreadId destPDId, ThreadId tmPDId)**
  Wenn der Thread aus dem System entfernt wird, muss diese Funktion alle verbleibenden Ressourcen freigeben. Dazu gehören die Thread Struktur wie auch die Stacks.

  ia32: Es wird zuerst der User Mode Stack gelöscht, nachher der Kernel Mode Stack aus dem PD des Threads und zuletzt auch aus dem PD des TM-Threads entfernt. Damit ist auch die Thread Struktur freigegeben.

5.3.2 Scheduler

Die Unterstützung von Nested Exceptions zog Anpassungen an den Scheduler Funktionen nach sich. Diese sind hier dokumentiert, weil sie für die anderen Architekturen noch implementiert werden müssen.

`BlockCause` gibt ein Hinweis auf das gewünschte Stack Frame, welches verändert werden soll (siehe Kapitel 4.2.4). Dabei sind folgende Unterbrechungsarten definiert: `TIMER_INT`, `PAGE_FAULT`, `MSG_RECV`, `MSG_SEND` und `ACTIVE`.

Die Funktionen liefern `TM_OK`, wenn sie erfolgreich waren, und sonst `TM_FAILED`.

- **int tmHalThreadSetReady(ThreadPtr threadPtr, BlockCause hint)**
  Setzt das entsprechende Stack Frame auf `READY`.
- **int tmHalThreadSetBlocked(ThreadPtr threadPtr, BlockCause hint)**
  Blockiert das Stack Frame.
- **void tmHalThreadSetResched(ThreadPtr threadPtr, BlockCause hint)**
  Markiert das Stack Frame so, dass der Interrupt Handler den Scheduler den nächsten Thread wählen lässt. Ansonsten läuft der aktuelle Thread weiter.
- **void tmHalThreadYield(ThreadPtr threadPtr, BlockCause hint)**
  Gibt die CPU ab, ausser der Thread ist im Interrupt Handler. Dann wird dieser zuerst abgearbeitet.
- **SchedStatus tmHalThreadGetSchedStatus(ThreadPtr threadPtr, BlockCause hint)**
  Liefert den Scheduler Status (`READY`, `RUNNING`, `BLOCKED`) des Stack Frames.
- **void tmHalIpcResetPendingFlag(Thread* threadPtr)**
  Diese Funktion wird aufgerufen, bevor ein Thread wieder hergestellt wird. Wenn der Thread auf eine Nachricht gewartet hat und jetzt durch sie geweckt wird, muss der Status seiner Nachrichten-Queue von `PENDING` auf `NOT_WAITING` aktualisiert werden.

  Wenn der Thread aus einem anderen Grund geweckt wird (zB ein Page Fault), bleibt der Status erhalten.

Architekturen, welche keine Nested Exceptions kennen, können einfach im File `TMHal.h` Makros definieren, welche die alten Scheduler Funktionen aufrufen. Dieses File befindet sich im Unterverzeichnis mit den architektur-abhängigen Files des `Threads` Verzeichnisses.

5.3.3 Reentranter Interrupt Handler

Den Interrupt Handler reentrant (auch thread-safe genannt) zu machen, war nicht einfach. Zum Glück sind viele Teile von Topsy v3 schon geschützt, zum Beispiel der Heap Allocator und weitere Teile des Memory Management.

Die Thread Struktur ist neu quasi-gleichzeitigen Zugriffen ausgesetzt. Sie beinhaltet nicht nur Statusinformationen des Schedulers, sondern auch die Nachrichten-Queue. Besonders diese zwei Strukturen müssen unbedingt geschützt werden. Sie sind aber nicht unabhängig, sondern eng miteinander verknüpft. Deshalb können sie nicht einzeln, sondern die Thread Struktur muss als ganzes durch ein Lock geschützt werden.

In Topsy gibt es drei Möglichkeiten, Locking zu implementieren. Die einfachste ist das Deaktivieren von Unterbrechungen in kritischen Bereichen. Dies wäre praktikabel und effizient für Singlecore-Systeme, funktioniert bei mehreren Prozessoren aber nicht mehr. Die zweite Möglichkeit ist die Verwendung von `BigLock`, einem Lock, welches gleichzeitig mehrere lesende Zugriffe, aber immer nur einen schreibenden zulässt. Solche Situationen kommen aber kaum vor, weshalb ein normales `Lock` verwendet werden kann. `Lock` ist als simples Spinlock[9] implementiert.

Das Hinzufügen der Lockvariable zur Threadstruktur und das Locking der kritischen Bereiche genügt aber noch nicht. Das Lock muss in Bereichen des Interrupt Handlers erworben werden, in denen Interrupts deaktiviert sind. Wenn ein anderer Thread das Lock noch besitzt, bleibt Topsy im Spinlock hängen. Es muss also verhindert werden, dass ein Thread in kritischen Bereichen unterbrochen wird.

Die Kombination aus Lock und maskierten Interrupts ist nicht gefahrlos. Der Programmierer muss garantieren, dass seine Implementation in kritischen Bereichen nicht unterbrochen werden kann. Dies gilt insbesondere für Exceptions wie Page Faults und Non Maskable Interrupts (NMI). Diese Garantien können im Interrupt Handler gegeben werden, indem in den kritischen Bereichen nicht auf Strukturen zugegrifen wird, von denen nicht bekannt ist, ob sie gemappt sind und keine NMIs verwendet werden. Der wachsende Stack kann keinen Page Fault verursachen, da dieser zu Beginn vollständig gemappt wird.

Zur Vereinfachung wurden die Makros `LOCK_THREAD(threadPtr, flags)` und `UNLOCK_THREAD(threadPtr, flags)` eingeführt, welche für alle Architekturen implementiert werden müssen (`Threads/<arch>/TMHal.h`). Diese Makros kümmern sich um die De- respektive Re-Aktivierung der Interrupts und das Erwerben respektive Freigeben des Locks.

Die gewählte Lösung schützt die Thread Struktur vor zeitgleichen Zugriffen und funktioniert sowohl in Single als auch in Multi Core Systemen.

5.3.4 Fazit und Limitationen des neuen Thread Management

Das neue Thread und Stack Management ist in seinem Aufbau komplizierter als das alte. Es bietet jedoch erweiterte Möglichkeiten wie Nested Exceptions und erhöhte Sicherheit, da User Threads nicht auf ihren Kernel Mode Stack zugreifen können.

Da die Kernel Mode Stacks die Thread Struktur enthalten, müssen sie immer in den PD des TM-Threads gemappt sein. Dies limitiert die maximale Anzahl gleichzeitig lauffähiger Threads. Ein KMS ist 8KB gross und muss unterhalb und oberhalb mindestens eine freie Page haben, um Stack Faults festzustellen. Er braucht also 16KB des VM. Bei 4GB verfügbarem VM können etwa eine Viertelmillion Threads angelegt werden.

Im Betrieb ist das neue System unkomplizierter, da keine Daten kopiert werden müssen und nicht zwischen User und Kernel Threads oder Nested und normalen Exceptions unterschieden werden muss.

5.3.5 Lokaler APIC

Der lokale APIC muss einerseits initialisiert und konfiguriert und andererseits der alte Timer durch den modernen des lokalen APICs ersetzt werden. Der C-Code für den APIC ist in den Files `Threads/ia32/TMHalApic.{c,h}` und die Assembler Routinen in den Files `Threads/ia32/TMHalApicAsm.{S,h}`.

Der lokale APIC wird konfiguriert durch Memory Mapped Register, welche an bestimmten Offsets der Position `apic_base` liegen. Einige der Register sind nur lesbar andere auch schreibbar. Für genauere Informationen sei auf das Kapitel 8 im Band 3A von [6] verwiesen.

[9] Ein Spinlock prüft in einer ewigen Schleife ständig, ob das Lock freigegeben wurde und kehrt dann zurück.

Kompilieren

Wenn im Makefile `Makefiles/Makefile.ia32.i386` die Variable `APIC` auf `"YES"` gesetzt wird, wird der Kernel mit APIC-Support kompiliert und das Makro `APIC` definiert.

Initialisierung

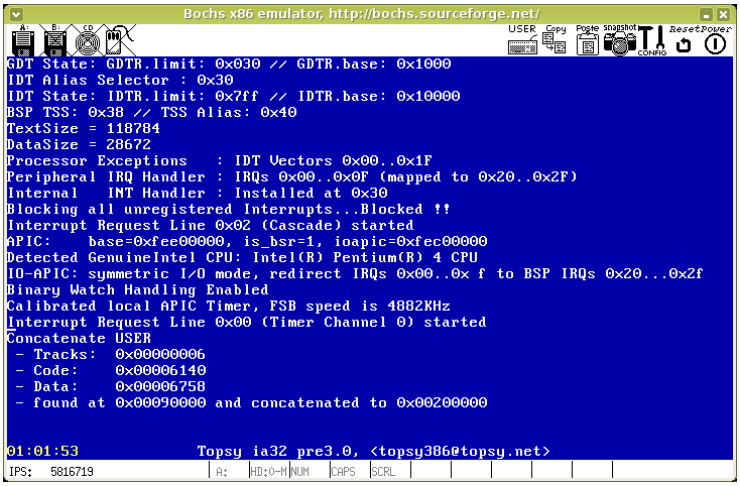


Abbildung 5.16: Startup von Topsy v3

Die Initialisierungsfunktionen werden im architektur-abhängigen Startup in `Startup/ia32/Init.c` aufgerufen. Einen Screenshot des Outputs zeigt die Abbildung 5.16.

In `ApicInit()` wird zuerst geprüft, ob ein APIC vorhanden ist, und dieser dann aktiviert. Dann probiert der Code, den CPU-Typ herauszufinden. Wenn sie nicht von Intel stammt, wird abgebrochen. Dies hat den Grund, dass später auf sogenannte “Model-Specific Registers” zugegriffen wird, die sonst eventuell bei diesem Prozessor ungültig sind. Dann werden die einzelnen Komponenten konfiguriert und die gewünschten Interrupt Vektoren (Nummern) gesetzt. Die Priorität eines Interrupts ist gleich seiner Nummer geteilt durch 16. Der lokale APIC sendet einen Interrupt nur an den Prozessor, wenn dieser Interrupts aktiviert hat und nicht gerade einen mit höherer Priorität behandelt. Momentan haben (fast) alle Interrupts die gleiche Priorität. Das Task Priority Register definiert, welche minimale Priorität ein Interrupt haben muss, um einen anderen unterbrechen zu können. Es wird auf 0 initialisiert, so dass alle Interrupts andere unterbrechen dürfen. Zuletzt werden die Interrupt Register zurückgesetzt und damit alle hängigen Interrupts entfernt.

`ApicTimerInit()` kalibriert den Timer des lokalen APICs. Zuerst muss die Geschwindigkeit des Front Side Buses herausgefunden werden. Dazu wird der Timer mit einem grossen Initialwert gestartet und mit Hilfe der Echtzeituhr eine Sekunde gewartet. Die Differenz der Timerwerte ist die Frequenz.

Später wird in `tmInit()` (`Threads/TMInit.c`) die Funktion `tmInstallErrorHandlers()` ausgeführt, welche die Interrupt Handler des APICs installiert. Der eine ist `ApicErrorISR()` mit der Interrupt Nummer (Vektor) `APIC_IR_ERRIV`, welcher bei Fehlern im APIC ausgelöst wird und eine Fehlermeldung ausgibt. Der andere ist der Spurious Interrupt Handler `ApicSpivISR()` mit dem Vektor `APIC_IR_SPIV`. Dieser spezielle Interrupt dient dazu, auf einen Prioritäten-Konflikt aufmerksam zu machen. In Topsy wird er ignoriert und der Handler wird nur registriert, wenn das Makro `APIC_SPIV_HANDLING_ISR` definiert ist. Andernfalls wird immer sofort `iret` ausgeführt, was den Overhead absolut minimiert (siehe Quelltext in `Threads/ia32/TMHalAsm.S`).

Verwendung des Timers des lokalen APICs

Die Verwendung des Timers des lokalen APICs unterscheidet sich nicht von derjenigen des alten Timers. Mittels `setClockValue()` kann das gewünschte Intervall in Millisekunden gewählt werden. Wenn der Kernel mit APIC-Support kompiliert wird, wird der APIC Timer und sonst der alte verwendet.

Interrupt Handler

Die Abarbeitung eines Interrupts muss dem APIC mit `__endOf_APIC_IRQ()` signalisiert werden. Dann weiss dieser, dass er den nächsten senden darf. Die einzige Ausnahme ist der Spurious Interrupt, welcher nicht bestätigt werden muss.

5.3.6 I/O APIC

Der I/O APIC ist ein Baustein, welcher Interrupts flexibel an verschiedene CPUs weiterleiten kann. Bein Einschalten verhält er sich rückwärts kompatibel zum alten Programmable Interrupt Controller. In diesem Modus sind sämtliche APIC-Komponenten passiv und werden umgangen.

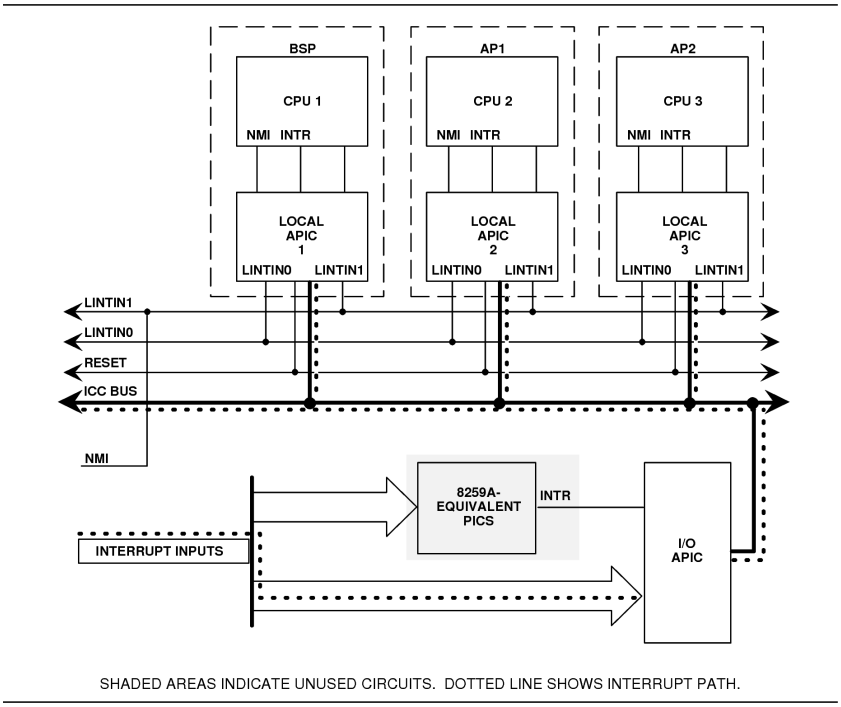


Abbildung 5.17: Symmetric I/O des I/O-APIC (aus [7])

Der flexibelste Modus ist der Symmetric I/O Mode, welcher in Abbildung 5.17 dargestellt ist. Dabei werden alle Interrupts an den I/O-APIC gesendet. Dieser besitzt eine interne Tabelle, in der für jeden eingehenden Interrupt eine oder mehrere CPUs und ein Interrupt-Vektor definiert ist. Anhand dieser Tabelle leitet der I/O-APIC die eingehenden Interrupts weiter.

In `IOApicInit()`, welches gleich nach der Initialisierung des lokalen APICs ausgeführt wird, wird zuerst der Baustein aktiviert, in den Symmetric I/O Mode gesetzt und dann die Weiterleitungstabelle konfiguriert. Die Interrupts 0 bis 16 werden an den BSP mit den Vektoren 32 bis 48 gesandt und alle anderen ausmaskiert. Dies wird so gemacht, dass das Verhalten äquivalent zu dem des alten PIC ist.

5.4 Interprozesskommunikation

Die Implementation des IPC-Mechanismus muss sich zwei neuen Herausforderungen stellen: dem virtuellen Memory und gleichzeitigen Zugriffen.

Der Hauptteil des IPC-Managements geschieht im Interrupt Handler. Die verwendete Nachrichten Queue ist Teil der Thread Struktur. Die Thread Safety wurde deshalb im Kapitel 5.3.3 schon adressiert.

In diesem Kapitel geht es im Weiteren um das neue Memory Management.

5.4.1 Memory Management

Das im Kapitel 4.3.1 skizzierte Protokoll wurde implementiert. Die Funktion `int vm_pmap_map_page_in_other_pmap(pmap_t from_pmap, address_t from_virt, pmap_t to_pmap, address_t to_virt, int flags)` dient dazu, temporär die Page der Nachricht in den PD des TM-Threads zu mappen. Später wird diese mittels der Funktion `void moveMessageToPD(Thread* threadPtr, Message* msgPtr, Message* destPtr)` in den PD des Empfängers verschoben.

Diese Routinen benötigen jeweils eine oder zwei Pages im virtuellen Adressraum des TM-Threads. Diese können mit `address_t vm_pmap_pagebuff_get_page(void)` angefordert und mit `void vm_pmap_pagebuff_release_page(address_t v_addr)` wieder freigegeben werden. Diese Funktionen schützen den Page Buffer durch ein Lock und sind thread safe.

Der Page Buffer wird bei der Initialisierung des pmap-Layers in der Funktion `void vm_pmap_pagebuff_init(struct vm_segmap* segmap)` angelegt.

5.5 Boot Prozess

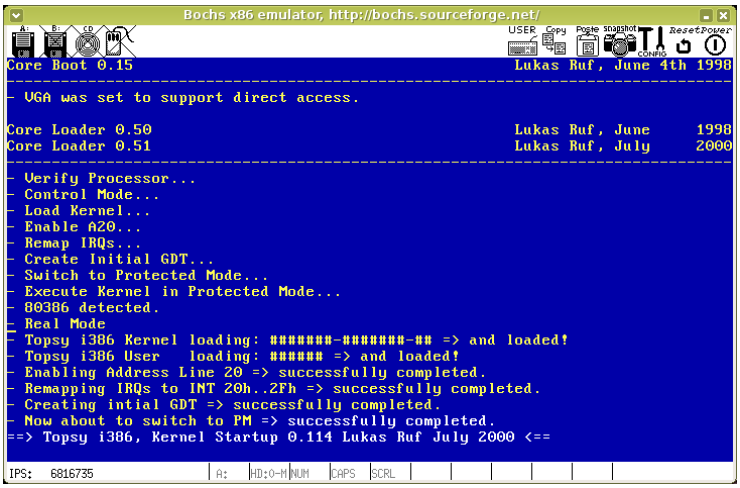


Abbildung 5.18: Core Boot und Core Loader von Topsy v3

Der Boot-Prozess wurde praktisch unverändert von Topsy i386 übernommen. Da an seinem grundsätzlichen Design wenig verändert wurde, sei auf die Dokumentation [2] verwiesen. Die Abbildung 5.18 zeigt den Ablauf von Core Boot und Core Load. Grossen Einfluss auf den Boot Prozess hatte die Verschiebung der Segmente und die Migration zum Topsyheader.

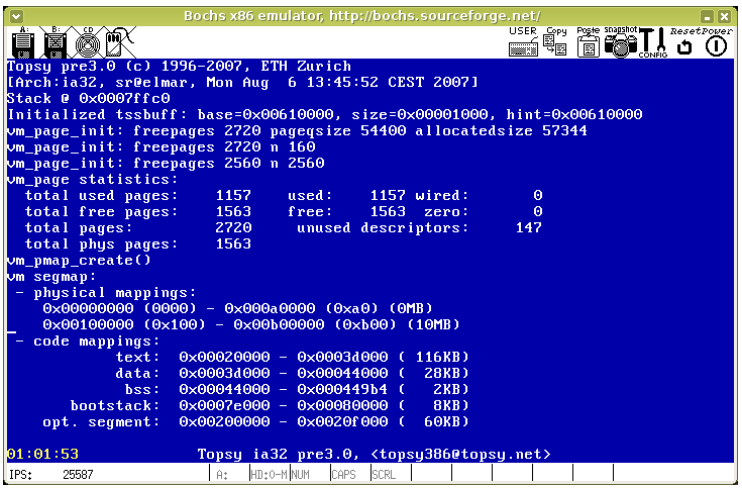


Abbildung 5.19: Architektur-unabhängiger Startup von Topsy v3

5.5.1 Relozierung zur Link-Zeit

Aus verschiedenen Gründen mussten die Link Skripte angepasst werden. Topsy v3 unterscheidet nicht nur zwischen Code und Daten eines Programms, sondern kennt auch BSS, welches uninitialisierte Daten beinhaltet. Letzteres musste dem Binary hinzugefügt und die Bereiche mussten an der Page Grenze ausgerichtet werden. Diese Bedingungen stellt das Memory Management von Topsy v3 (siehe Abbildung 5.19, welche die Initialisierung des Memory Management zeigt).

Sowohl das Kernel als auch das Userimage starteten ihre jeweiligen Segmente am Offset 0 (siehe Kapitel 4.1.2). Neu umspannen beide Segmente aber den ganzen linearen Adressraum. Somit startet der Kernel bei 128KB und das User Image wurde an die Position bei 2MB verschoben. Die Position im File ist also nicht mehr automatisch die Adresse im Memory, was im Link Skript berücksichtigt werden muss.

5.5.2 Topsyheader

Der Topsyheader ist ein in Topsy v3 eingeführter, obligatorischer Header für Binaries. Sein Format ist für alle Architekturen identisch und ersetzt den `BinFileDesc` des ia32-Ports. In Topsyheader werden vor allem Angaben zur Grösse und Position der Bereiche des Binaries gespeichert, sie sind in [3, Kapitel 18] dokumentiert. Die Informationen werden am Schluss des Build-Prozess vom Programm PatchBinaries in das Binary geschrieben.

Der Topsyheader der Binaries wird für das Kernel Image im File `Startup/ia32/start.S` und fürs User Image in `User/ia32/userStart.S` erstellt. Die Felder für architekurspezifische Informationen werden im ia32-Port für die Anzahl vom Binary belegten Tracks (Grösseneinheit auf Disketten) verwendet.

Der Boot Prozess ist an vielen Stellen auf den Header angewiesen und musste angepasst werden. Schon im Bootloader liest die Funktion `rm_load` (File `Boot/ia32/LowCore/CoreLoad/rm_load.inc`) die Grösse des Images und kopiert die angegebene Anzahl Tracks von der Diskette ins Memory (siehe Abbildung 5.18). Das dabei gewählte Kopierverfahren ist zwar schnell, das Image ist im Memory aber nicht mehr zusammenhängend und muss später nochmals umkopiert werden. Dies geschieht im `main()` (File `Startup/ia32/Init.c`) anhand des Topsyheaders des Kernel und User Images.

Das letzte Mal wird der Topsyheader im architektur-abhängigen Teil in der Funktion `vm_get_segmap()` (File `VMemory/ia32/vm_bootup_hal.c`) gebraucht, um dem Memory Manager die Grösse und Position der belegten Speicherbereiche anzugeben.

Kapitel 6

Evaluation

Testen und Evaluieren sind ein integraler Teil des Programmierens. In diesem Kapitel lege ich dar, mit welchen Mitteln ich die Korrektheit der vorgestellten Implementation verifizierte. Zuerst werden die verwendeten Methoden und Tests vorgestellt und nachher die Resultate präsentiert und diskutiert.

6.1 Methodik

Die Evaluation und das Testen von Topsy stützt sich auf drei Pfeiler: Verifikation und Kontrolle durch die Werkzeuge des Simulators, Langzeittests und das Last- und Crashtest-Programm von Topsy.

Die Beschäftigung mit dem Testen beginnt nicht erst am Schluss, sondern schon vor und während der Implementation. In Topsy ist es üblich, Preconditions und anderes mit Asserts durchzusetzen. Dies zwingt Entwickler, Funktionen nur in der vorgesehenen Weise zu verwenden (Design by Contract).

6.1.1 Simulator

Wenn Topsy in einem Simulator wie zum Beispiel bochs läuft, hat man die volle Kontrolle über alle Vorgänge. Das System kann jederzeit angehalten und sein Zustand inspiziert werden. Bei hardwarenahem Code, zum Beispiel für Kontextwechsel, ist dies oft die einzige Testmöglichkeit.

Der Debugger von Bochs erlaubt das Setzen von Breakpoints. Diese können nicht nur auf eine Stelle im Code, sondern auch auf Lese- oder Schreibzugriffe an bestimmte Adressen gesetzt werden. Bochs integriert auch Tools zur Analyse des Speichers, des Page Mapping, der Segmente, des Stacks, der CPU und weiteres.

Damit können Situationen analysiert werden, die schwierig herzustellen sind, zum Beispiel wenn das Timing eine grosse Rolle spielt. Gerade Interrupts und besonders Nested Exceptions können so untersucht werden.

6.1.2 Langzeittests

Programmierfehler im HAL bemerkt man meistens schon beim Booten. Denn der HAL-Layer hat genau definierte Schnittstellen nach unten und oben. Weder die Hardware noch die Kernel Threads sind tolerant gegenüber Abweichungen und reklamieren sofort. Der HAL muss funktionieren, damit in den oberen Schichten überhaupt irgendetwas läuft. Die Abhängigkeit und die Frequenz, mit der die Funktionen aufgerufen werden, sind so hoch, dass auch sehr unwahrscheinliche Fälle mit hoher Wahrscheinlichkeit auftreten.

Sobald das System gebootet hat und die Shell läuft, müssen alle Komponenten funktionieren. Zu diesem Zeitpunkt laufen 8 Threads, es müssen also konstant Kontexte gespeichert und gewechselt werden. Es sind sowohl User als auch Kernel Threads aktiv. Zudem werden ständig

Nachrichten versendet, da die Shell die Konsole nach neuem Input pollt. Dafür sind auch Nachrichten an den Memory Manager nötig, da die Konsole die Antwort in eine Enveloppe schreibt, welche von diesem angelegt und wieder freigegeben werden muss.

Wenn Topsy über längere Zeit läuft, können viele Fehler wie Memory Leaks, Fehler im Thread Management und in der IPC nahezu ausgeschlossen werden.

6.1.3 Lasttests

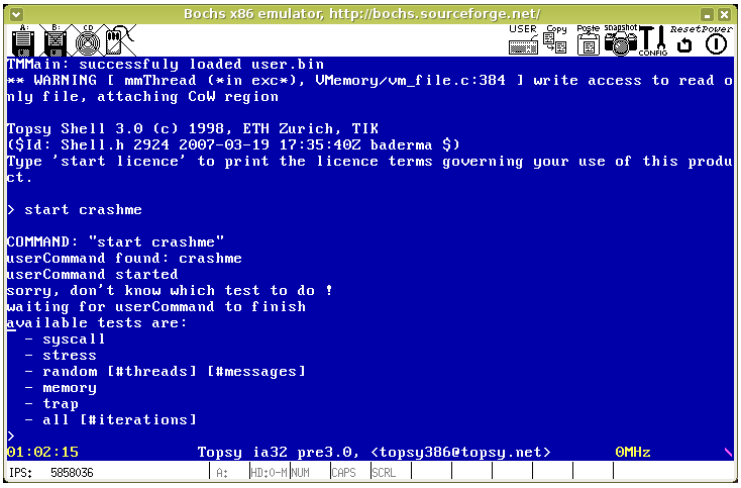


Abbildung 6.1: Module von `crashme`

Schon seit der ersten Version ist in Topsy das Programm `crashme` integriert. `crashme` wird von der Shell aus als User Prozess gestartet (siehe Abbildung 6.1) und versucht, das System zum Absturz zu bringen. Es ist dem Unix-Testprogramm nachempfunden, welches beinahe jedes verfügbare Unix aus dem Tritt brachte, bis die Entwickler sich `crashme` anschauten. Diese verschiedenen Module testen die Robustheit und Belastbarkeit von Topsy:

- **`syscall`**
  Startet einen Kindprozess, welcher die Tests durchführt und auf dessen Ende wartet. Anhand des Rückgabewerts weiss der Elternprozess, ob die Tests erfolgreich waren oder nicht oder ob der Kindprozess vorzeitig beendet wurde.

  Der Kindprozess alloziert einige Memory Blöcke, startet einen Thread, testet den Yield-Syscall, empfängt eine Nachricht vom eben gestarteten Thread, killt ihn dann und gibt zuletzt die allozierten Speicherbereiche wieder frei.

  `syscall` ist eine Art Unit Test der verfügbaren Syscalls, testet aber nur ganz profane Abläufe und weder Grenzfälle noch Unerlaubtes.

- **`stress`**
  Dieser Test startet ein Thread, welcher rekursiv neue Threads erstellt. Sobald das Erstellen eines Threads fehlschlägt, wird abgebrochen und alle beendet.

  Der Name des Tests ist berechtigt, denn das Erzeugen eines Threads ist eine teure Operation und das System wird damit bis an seine Grenzen belastet. Der Test läuft so lange, bis kein Speicher für einen zusätzlichen Thread mehr verfügbar ist. Auch in dieser Situation muss Topsy weiterlaufen und andere Protection Domains dürfen dadurch nicht beeinträchtigt werden. Zuletzt muss der Speicher aller Threads wieder freigegeben werden, damit der Test mehrmals durchlaufen kann.

- **`random [#threads] [#msgs]`**
  Startet eine Anzahl Threads (Argument `[#threads]`), welche eine Anzahl Nachrichten (Argument `[#msgs]`) versenden. Die Nachrichten werden mit pseudozufälligen Daten gefüllt. Die meisten Nachrichten und Destinationen sind also ungültig.

Der Test gilt als erfolgreich, wenn Topsy nicht abstürzt oder hängen bleibt.

- **memory**
  Dieser Test besteht aus zwei Phasen. In der ersten werden achtzehn Threads gestartet, welche jeweils von einer bestimmte Stelle im Memory lesen. In der zweiten Phase werden zwanzig Threads einer leeren Funktion gestartet.

  Die erste Phase versucht systematisch von grenzfälligen Stellen im Memory zu lesen. In einigen Fällen darf dies nicht erlaubt sein, und die Threads müssen ordnungsgemäss beendet werden.

  Die zweite Phase lasse ich lieber beiseite.

- **trap**
  `trap` besteht auch aus zwei Phasen, hat sonst aber nichts mit dem `memory`-Test gemeinsam. Zuerst werden achtzehn Threads gestartet, wobei die übergebenen Funktionspointer auf verschiedene Stellen im Speicher zeigen. In der zweiten Phase werden 48 Threads kreiert und die Funktionspointer zufällig gewählt.

  Dieser Test beansprucht vor allem das Thread Management und Exception Handling. Der TM-Thread muss robust gegen fehlerhafte Parameter sein und darf ihnen nicht blind vertrauen. Wenn die Threads tatsächlich gestartet werden, tritt meistens sofort eine Exception auf. Diese muss unter allen Bedingungen korrekt gehandhabt werden.

  Topsy muss alle Testthreads beenden, was nur gelingt, wenn alle Exception Handler richtig reagieren. Der Test gilt als bestanden, wenn er durchläuft, ohne dass Topsy beeinträchtigt wird.

6.2 Resultate

Nachdem nun alle Methoden der Verifikation vorgestellt wurden, beschreibt dieses Kapitel die erzielten Resultate. In allen Tests standen dem System 12MB an Speicher zur Verfügung. Dies ist nahe am absoluten Minimum gewählt, damit allfällige Memory Leaks früher entdeckt werden.

6.2.1 Langzeittests

Ich habe Topsy sowohl in Bochs als auch im VMPlayer über drei Tage laufen lassen. In 90 Stunden wurden in Bochs 287’897’998 Nachrichten versendet und das System wurde 637’391’608 Mal unterbrochen, 7’784’271 waren verschachtelte Unterbrechungen. VMWare ist noch ein bisschen schneller, die Zahlen dürften noch etwas höher liegen.

Diese Zahlen zeigen eindrücklich, wieviel geschieht, obwohl die Shell nichts zu tun hat. Dies ist zwar gut für die Aussagekraft des Tests, zeigt aber zugleich auch, dass die Shell und das IO Subsystem noch verbessert werden können.

6.2.2 Lasttests

Die Tests des Tools `crashme` wurden mehrmals durchlaufen. Die Resultate und deren Interpretation folgen zu jedem Modul separat.

Die Tests wurden nacheinander und in keiner speziellen Reihenfolge ausgeführt.

syscall

Zum `syscall` Test gibt es nicht viel zu sagen. Wenn diese Syscalls nicht funktionieren würden, könnte man das System gar nicht aufstarten und den Test ausführen.

Der Test ist wohl eher zur Dokumentation der Syscalls geschrieben worden.

stress

Bei 12MB Speicher ist nicht mehr viel Platz für zusätzliche Prozesse verfügbar. Der Test kann dennoch fast 600 neue Threads anlegen. Wenn der Speicher voll und keine Page mehr verfügbar ist, reagiert Topsy unterschiedlich, je nach dem, wo es sich gerade befindet. Folgende Varianten konnten beobachtet werden:

a) Beim Syscall an den MM-Thread, der eine Enveloppe an eine Nachricht anhängt, hält das System mit einer `PANIC` an. Diese wird in der Funktion `vm_loan_alloc()` des Loan Managers ausgelöst, weil dieser keine freie physikalische Page mehr findet.

Der limitierende Faktor in diesem Fall ist also das physikalische Memory, obwohl jeder Thread nur drei Pages braucht: zwei für den KMS und einen für die erste Seite des UMS. Da der UMS sehr grosszügig dimensioniert ist, ändert sich die Situation schon ab etwa 200MB verfügbarem Speicher. Dann wird der VM des PDs, in dem all diese Threads laufen, zur limitierenden Ressource, da alle Threads den Platz für ihren UMS reservieren müssen.

b) Wenn die `PANIC` auskommentiert und stattdessen ein Fehler zurückgegeben wird, schlägt das Anlegen des Stacks für einen neuen Thread fehl. Die Erstellung wird daraufhin abgebrochen und alle Threads terminieren.

Anhand der Memory Mappings im PD des TM-Threads und der Shell kann man erkennen, dass alle Stacks der Threads wieder entfernt wurden. Der Test kann beliebig oft wiederholt werden, es werden gleich viele neue Threads angelegt und wieder beendet. Es ist also sehr unwahrscheinlich, dass in diesem Code ein Memory Leak existiert.

Lösungsmöglichkeit: Bei vollem Speicher einfach das Allozieren fehlschlagen zu lassen, ist riskant. Es müsste stattdessen möglichst schnell Speicher freigegeben werden, indem zum Beispiel der speicherintensivste User Thread beendet wird. Da mir die Zeit für die Implementation und Evaluation eines solchen Mechanismus nicht reicht, lasse ich die `PANIC` stehen.

random

Die zufällig generierten Nachrichten, die der `random` Test verschickt, gehen auch an einen zufälligen Thread, der nicht existieren muss. Solange die adressierten Threads existieren, läuft der Test gut und die Threads verwerfen die Nachrichten zumeist schon wegen des unbekannten Typs.

Wenn der Empfänger einer Nachricht nicht existiert, hält der Message Dispatcher `msgDispatcher()` das System mit einer `PANIC` an. Dies ist meiner Meinung nach die falsche Reaktion. Es kann sein, dass der Empfänger einer Nachricht nicht (mehr) läuft. Es muss auch eine Möglichkeit geben, dies einfach zu prüfen. Es würde reichen, in solchen Fällen eine Fehlermeldung zurückzugeben.

Der Message Dispatcher ist Teil des architektur-unabhängigen Codes. Deshalb und aus Zeitmangel lasse ich ihn so stehen.

memory

11 der 18 Threads werden beim Lesezugriff beendet, die anderen laufen weiter, da die übrigen Adressen für die Threads lesbar sind. Das Exception Management funktioniert also.

Man sollte den Test noch erweitern und den Schreibzugriff testen. Es müssten dann mehr Threads vorzeitig beendet werden.

trap

Alle Threads werden erfolgreich beendet. Die meisten verursachen schon beim Laden der ersten Instruktion einen Page Fault und werden beendet. Ansonsten treten viele verschiedene Exceptions auf. Dabei sind viele “General Protection Exceptions”, für die die Spezifikation [6] 31 verschiedene Ursachen auflistet, einige “Invalid Opcode Exceptions” und wenige andere.

6.2.3 Zusammenfassung

Topsy besitzt zwar keine Unit Tests[1], mit Hilfe der hier vorgestellten Tools können aber aussagekräftige Resultate erzielt werden. Mittels des `crashme` Programms und der Assert-Statements kann das System als Ganzes geprüft und können Regression Tests implementiert werden.

Natürlich sind die hier vorgestellten Resultate kein Korrektheits-Beweis. Sie liefern aber starke Indizien für die Korrektheit des Codes und zeigen einige klare Limitationen, welche in Zukunft behoben werden müssen.

[1] Unit Tests sind Prozesse, welche benutzt werden, um Codestücke individuell auf ihre Korrektheit zu prüfen.

Kapitel 7

Weiterführende Arbeiten

Im Zuge dieser Arbeit ist viel Code entstanden und Topsy v3 ist der Vision der Multicore Unterstützung einen Schritt näher gekommen. Es sind aber noch viele Verbesserungen möglich. Die aus meiner Sicht wichtigsten Punkte will ich in diesem Kapitel aufführen, ohne dabei die in [3] genannten Punkte zu wiederholen.

7.1 Multicore

Bis Topsy v3 effektiv auf mehreren Prozessoren läuft, braucht es noch ein paar Erweiterungen. Die zusätzlichen Prozessoren müssen gebootet und verwaltet werden. Die wichtigsten Änderungen sind diejenigen im Scheduler. Dieser sollte überdacht werden. Es muss sich zeigen, ob das in [3] vorgeschlagene hierarchische Scheduling für ein MultiCore Setup geeignet ist.

7.2 Gegenwärtige Limitierungen

Topsy v3 hat noch Limitierungen, die in Zukunft adressiert werden müssen.

Die ia32-Architektur ist zwar komplex, bietet dafür aber mehr Features als andere. Da die Autoren von Topsy v3 andere Zielplattformen im Auge hatten, finden einige Funktionalitäten keine Entsprechung in Topsy v3.

7.2.1 Memory Management

Die ia32-Architektur kennt ein ‘Global’ Flag für Pages, welches diejenigen kennzeichnet, die bei einem Wechsel des virtuellen Adressraums nicht aus dem Cache gelöscht werden müssen. Man könnte die Performance steigern, indem Teile des Interrupt Handlings, die Hardware I/O Region und die Pages mit den APIC Registern als global eingetragen werden.

7.2.2 Thread Management

Das neue Thread Management muss weiter optimiert werden. Ein wesentlicher Vorteil von softwarebasierten Kontextwechseln ist die grössere Kontrolle über den Vorgang. Es wäre von Vorteil, Kontexte vor dem Wechsel zu überprüfen und mehr Statistiken über den Zustand des Systems zu führen.

Mit der Implementation der MultiProcessor Spezifikation [7] werden weitere Interrupts eingeführt. Spätestens dann sollten diese adäquat priorisiert werden.

7.2.3 Interprozesskommunikation

Auf einige unwahrscheinliche Spezialfälle reagiert Topsy v3 noch nicht optimal. Dazu gehört unter anderem das Versenden einer Nachricht an einen nicht existierenden Thread (siehe Kapitel 6.2.2). Wenn ein Thread versucht, einem anderen mit voller Nachrichten Queue eine Nachricht zu schicken, wird er beendet, sofern er ein User Thread ist. Anderenfalls passiert nichts und die Nachricht geht verloren. Beides entspricht nicht dem Verhalten eines fehlertoleranten Systems. Die Zustellung könnte zum Beispiel später nochmals versucht werden.

Topsy v3 wurde 2002 entworfen, als Computer im Allgemeinen und Kopieroperationen im Speziellen noch viel langsamer waren. Tests ergaben dazumal, dass ab einer Nachrichtengrösse von 16Bytes das Anhängen einer Enveloppe performanter als das Kopieren der ganzen Nachricht war. Dies ist heute sicher nicht mehr der Fall. Viele Syscalls müssen für wenige zusätzliche Bytes eine Enveloppe verwenden.

7.2.4 Input/Output Subsystem

Aus Zeitmangel wurden nur der Tastatur- und Video-Treiber portiert. Die anderen müssen eventuell noch angepasst und getestet werden.

Der Tastaturtreiber und andere verwenden zum Lesen und Schreiben einzelner Zeichen eine Enveloppe. Dies ist ein grosser Overhead und könnte durch die Unterscheidung zwischen Zeichen- und Block-Geräten vermieden werden.

7.2.5 User Threads

Um den Tastaturinput zu lesen, muss die Shell pollen. Eine bessere Lösung wäre, der Aufrufer würde blockiert, bis ein Zeichen eintrifft.

Von der Tastatur kommende Zeichen werden schon im Interrupt Handler auf die Konsole geschrieben. Dies ist eigentlich unerwünscht und macht das nachträgliche Editieren der aktuellen Kommandozeile schwieriger. Dies wäre aber eine sehr praktische Funktion.

Kapitel 8

Zusammenfassung und Fazit

Die Einführung von virtuellen Adressräumen in Topsy v3 und die Vorbereitungen für den Multicore Betrieb zogen umfangreiche Änderungen nach sich. Diese werden in diesem Kapitel zusammengefasst. Nachher ziehe ich das Fazit.

8.1 Zusammenfassung

Für die Portierung von Topsy v3 auf die ia32-Prozessor Architektur musste nicht nur der Hardware Abstraction Layer angepasst werden. Um die Bedingungen für den späteren Multicore Betrieb zu schaffen, musste ich einige Architekturänderungen und Anpassungen des architekturunabhängigen Teils vornehmen. Nach der Einarbeitung in die ia32-Architektur und die Topsy-Mechanismen war die Implementation des Pagings dank der guten Dokumentation und des sorgfältigen Designs des neuen Memory Managements problemlos.

Eine grössere Herausforderung stellte das neue Thread Management dar. Aus Sicherheitsüberlegungen und auch als eleganteste Lösung bot sich eine Architektur mit User und Kernel Mode Stack an. Deshalb ist das Stackmanagement neu architektur-abhängig und muss für andere Portierungen erst implementiert werden. Von den kürzeren Reaktionszeiten auf wichtige Interrupts profitieren alle Architekturen auf Grund des neuen reentranten Interrupt Handlers. Damit wurde zugleich eine Notwendigkeit für Multicore Systeme abgedeckt. Zwei weitere erfüllte Bedingungen sind der Support für den lokalen APIC und den I/O-APIC.

Der reentrante Interrupt Handler und die Verwendung von Paging stellen auch neue Anforderungen an die IPC. Zudem mussten zahlreiche ‘kleine’ Änderungen in allen Subsystemen vorgenommen und Bugs gepatcht werden.

8.2 Fazit

Topsy v3 läuft stabil auf einem PC mit einer Intel Pentium 4-CPU oder einer neueren. Es wurden die Bedingungen für Multicore Unterstützung gemäss der MultiProcessor Spezifikation [7] von Intel geschaffen. Um die Vision zu verwirklichen, bleibt aber noch viel zu tun. Sowohl an Topsy selber als auch an den verschiedenen Portierungen kann noch Einiges ergänzt werden.

Mir persönlich hat die Arbeit Spass gemacht. Es war spannend, an diesem schön entworfenen Betriebssystem zu entwickeln. Ich habe viel über Betriebssystemkonzepte und -Architekturen gelernt und sehr von der Betreuung profitiert. Es war für mich eine wichtige Erfahrung und äusserst aufschlussreich, den Blick fürs ganze System zu gewinnen. Ein grosses Dankeschön richte ich an all die fleissigen Hacker, welche an Topsy arbeiteten, und wünsche dem Projekt weiterhin gutes Gedeihen.

Anhang A

Projektplan

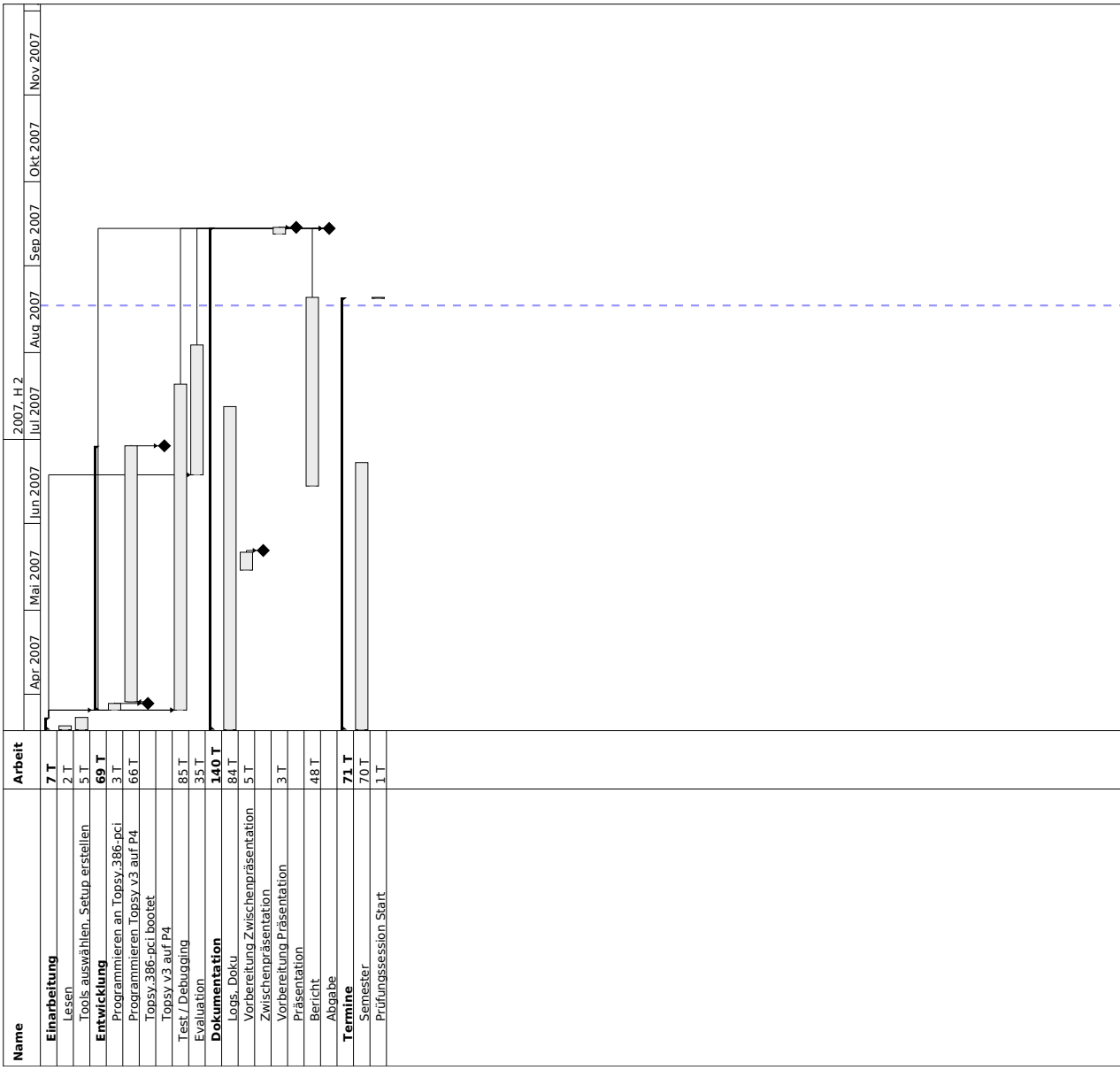


Abbildung A.1: Projektplan

Anhang B

Originale Aufgabenstellung

Auf den folgenden drei Seiten ist die originale Aufgabenstellung abgebildet.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Somersemester 2007

Semesterarbeit
für
Sebastian Ryffel

Betreuer: Lukas Ruf

Ausgabe: 19.03.2007
Abgabe: 07.07.2007

Topsy v3 on Multi-Core Processors

1 Einführung

Chip-Multicore Prozessoren kommen heutzutage bereits in handelsüblichen Arbeitsplatzrechner zum Einsatz. Beispiele sind der Dual-Core Intel Xeon oder der Quad-Core AMD Opteron Prozessor.
Topsy v3 [3] – a Teachable Operating System [1] – wurde an der ETH Zürich entwickelt und wird an verschiedenen Universitäten und Fachhochschulen im Unterricht eingesetzt. Topsy v3 folgt einer Schichtenarchitektur und verfolgt ein sehr ausgeprägtes Mikrokernel-Konzept, das sog. Protection Domains in den beiden Dimensionen *Privilegien* und *Ausführungspriorität* priorisieren kann.
Während Topsy v3 bereits schon für verschiedene Prozessorplattformen verfügbar ist (u.a. für die Prozessoren Intel XScale und MIPS R3000), existiert für ia32-kompatible Prozessoren mit Topsy386[4] nur eine Topsy v1-Architektur-kompatible Portierung.

2 Aufgaben: Topsy v3 on Multi-Core Processors

In dieser Semesterarbeit von Sebastian Ryffelsoll Topsy v3 auf eine moderne ia32-kompatible Chip-Multicore Prozessorplattform portiert werden. Diese Portierung umfasst die Anpassungen des Hardware Abstraktionsschicht (HAL, Hardware Abstraction Layer) und, wo benötigt, die Implementierung und Anpassung geeigneter Mechanismen in der portablen Kernelschicht von Topsy v3.

3 Vorgehen

Die Portierung von Topsy v3 erfolgt in drei Phasen. In der ersten Phase wird der HAL Topsy386 auf einen binär kompatiblen ia32-Prozessor-Simulator und daran anschliessend auf die Ziel Hardware portiert. Diese Portierung gestattet das Anpassen der Basisfunktionalität, die für Topsy v3 ebenfalls benötigt wird, und erlaubt eine Einarbeitung in den Source Code und die Prozessorarchitektur.
In der zweiten Phase werden die Modifikationen für Topsy v3 am HAL und am Kernel in den dann existierenden Code eingearbeitet, so dass Topsy v3 für ia32-kompatible Plattformen zur Verfügung steht.
Die dritte Phase beinhaltet die Umsetzung der benötigten Anpassungen, um Chip-Multicore-Eigenschaften soweit als möglich auszunutzen.
Zur Erreichung dieser Ziele wird die Arbeit nach folgendem Vorgehen durchgeführt:

- Richten Sie sich eine Entwicklungsumgebung (GNU Tools) unter Linux ein.
- Machen Sie sich vertraut mit den Unterlagen (Dokumentation und Source Code) zum bestehenden Topsy386 und Topsy v3, gleich wie zur ia32-Prozessorplattform.
- Erstellen Sie einen Zeitplan, in welchem Sie die von Ihnen zu erreichenden Meilensteine Ihrer Arbeit identifizieren.
- Analysieren Sie verschiedene Komponenten von Topsy386, um diejenigen zu identifizieren, die für eine moderne ia32-Prozessorplattform angepasst werden müssen.
- Identifizieren Sie die anzupassenden Funktionen und Parameter in Topsy v3, die für die Portierung angepasst werden müssen.
- Implementieren Sie diese für eine moderne ia32-kompatible Chip-Multicore Plattform.
- Beachten Sie insbesondere die Problematik des Startvorgangs und der Initialisierung der ia32-kompatiblen Prozessoren.

optional Entwickeln Sie Mechanismen, um spezifische Eigenschaften einer Chip-Multicore Architektur unter Topsy v3 zu kontrollieren und auszunutzen.

- Zeigen Sie die erfolgreiche Implementierung auf einem Chip-Multicore Prozessor durch eine Ausführung der Demo-Applikationen unter Topsy v3.
- Evaluieren Sie das Erreichte.
- Dokumentieren Sie das Erreichte.

Auf eine klare und ausführliche Dokumentation wird besonders Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte und untersuchten Varianten vor dem definitiven Variantenentscheid ausführlich schriftlich festzuhalten.

4 Organisatorische Hinweise

- Am Ende der zweiten Woche ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Mit dem Betreuer sind regelmässige, zumindest wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen sollen die Studenten mündlich über den Fortgang der Arbeit und die Einhaltung des Zeitplanes berichten und anstehende Probleme diskutieren. Die Sitzungen können mittels geeigneten Telekommunikationsmitteln abgehalten werden, falls möglich.
- Am Ende des ersten Monates muss eine Vorabversion des Inhaltsverzeichnis zur Dokumentation dem Betreuer abgegeben und mit diesem besprochen werden.
- Nach der Hälfte der Arbeitsdauer soll ein kurzer mündlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt. Dieser Zwischenbericht besteht aus einer **viertelstündigen**, mündlichen Darlegung der bisherigen Schritte und des weiteren Vorgehens gegenüber Professor Plattner.
- Am Schluss der Arbeit muss eine Präsentation von **20 Minuten** im Fachgruppen- oder Institutsrahmen gegeben werden. Anschliessend an die Schlusspräsentation soll die Arbeit Interessierten praktisch vorgeführt werden.

- Ein einheitlicher Coding Style muss eingehalten werden.
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden; Quellen müssen korrekt zitiert werden.
- Die Dokumentation muss mit dem Satzsystem LaTeX Graphiken müssen mit Open Source Werkzeugen erstellt werden, während für die Präsentation auch Staroffice oder Microsoft PowerPoint verwendet werden kann.
- Quellcode der Arbeit muss regelmässig (mind. täglich) auf dem Topsy subversion Server gesichert werden (`https://svn.topsy.net`).
- Es ist ein mit Bindespiralen (am TIK vorhanden) gebundener Schlussbericht über die geleisteten Arbeit abzuliefern (2 Exemplare). Dieser Bericht besteht aus einer Zusammenfassung, einer Einleitung, einer Analyse von verwandten und verwendeten Arbeiten, sowie einer vollständigen Beschreibung der Konfiguration von den eingesetzten Programmen. Der Bericht ist in Deutsch oder Englisch zu halten. Die Zusammenfassung muss in Deutsch und Englisch verfasst werden.
- Die Arbeit muss auf CDROM archiviert abgegeben werden. Stellen Sie sicher, dass alle Programme sowie die Dokumentation und die Präsentationen sowohl in der lauffähigen, resp. druckbaren Version als auch im Quellformat vorhanden, lesbar und verwendbar sind.
  Mit Hilfe der abgegebenen Dokumentation muss der entwickelte Code zu einem ausführbaren Programm erneut übersetzt und eingesetzt werden können.
- Diese Arbeit steht unter der GNU General Public License [2] (GNU GPL) v2.
- Diese Arbeit wird als Semesterarbeit an der ETH Zürich durchgeführt. Es gelten die Bestimmungen hinsichtlich Kopier- und Verwertungsrechte der ETH Zürich.

Literatur

[1] G. Fankhauser, C. Conrad, E. Zitzler, and B. Plattner. *Topsy – A Teachable Operating System*. Computer Engineering and Networks Laboratory (TIK), 1997.
[2] GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`, Oct. 2005.
[3] C. Jeker and B. Lutz. *Memory, IO and Process/Thread Management for Topsy v3*. Computer Engineering and Networks Laboratory (TIK), Mar. 2002.
[4] L. Ruf. *Topsy i386 – A Teachable Operating System. The Port to the ia32 Architecture*. Computer Engineering and Networks Laboratory (TIK), 1998.

Zürich, den 19.03.2007

Lukas Ruf

Literaturverzeichnis

[1] G. Fankhauser, C. Conrad, E. Zitzler, B. Plattner. *Topsy – A Teachable Operating System*.
Computer Engineering and Networks Laboratory, ETH Zürich
1996 - 2001

[2] Lukas Ruf. *Topsy i386, The Port to the ia32 Architecture*.
Semesterarbeit, Computer Engineering and Networks Laboratory, ETH Zurich.
April 1998 - Juni 1998

[3] Claudio Jeker und Boris Lutz. *Topsy v3, Teachable Operating System*.
Diplomarbeit, CSG TIK, ETH Zürich.
Oktober 2001 - März 2002

[4] Linus Torvalds. *Linux Kernel Source Code*.
www.kernel.org, Linux 2.6.17.6.
Juli 2006

[5] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*.
3rd Edition, O'Reilly Media, Inc.
November 2005

[6] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.
Volume 1-3. Intel Corporation, P.O. Box 5937, Denver, CO 80217-9808
November 2006

[7] Intel. *MultiProcessor Specification*.
Version 1.4. Intel Corporation, Literature Center, P.O. Box 7641, Mt. Prospect IL 60056-7641
Mai 1997

[8] *Topsy v3, Quelltext der ia32-Portierung*.
Subversion. https://svn.topsy.net:8443/svn/Topsy/Topsy/branches/Topsy-v3-P4
Juli 2007