Diplomarbeit
Virtualisierung eines x86-PC

Jens Nerche
Technische Universität Dresden

14. September 2000

Inhaltsverzeichnis

Kapitel 1

Einleitung

Heutzutage sind selbst Arbeitsplatzrechner relativ leistungsstark. Die zur Verfügung gestellte Rechenleistung wird von den verbreiteten Anwendungen kaum genutzt, so dass man problemlos mehrere Anwendungen parallel ausführen kann. Da es für x86-PCs jedoch mehrere Betriebssysteme gibt und viele Anwendungen nur für ein spezielles Betriebssystem angeboten werden, kann es passieren, dass für einen Anwendungswechsel auch ein Betriebssystemwechsel nötig ist, denn ein Manko der x86-Architektur ist, dass sie nicht so konzipiert ist, dass sie mehrere gleichzeitig laufende Betriebssysteme unterstützt. Mit der Virtualisierung kann man erreichen, dass mehrere Betriebssysteme simultan benutzt werden. Dadurch wird der Neustart des Rechners zum Systemwechsel unnötig, weiterhin können instabile Betriebssysteme getestet werden, während stabile zur Arbeit benutzt werden.

Die Professur Betriebssysteme kann einen virtuellen PC auch als Werkzeug einsetzen, um Fehler in den in Entwicklung befindlichen Betriebssystemkernen zu finden.

Weitere Ziele, die eine Virtualisierung der Hardware adressieren kann sind: besserer Schutz der Nutzer eines Mulitusersystems voreinander und der Betrieb von einfachen, verbreiteten Betriebssystemen auf skalierenden Multiprozessorsystemen.

Bisher galt die Virtualisierung der x86-Hardware als sehr schwer oder sogar unmöglich [1]. Mit dem Erscheinen des kommerziellen Produktes VMware wurde jedoch die Realisierbarkeit gezeigt. Diese Arbeit führt in das Themengebiet der Virtualisierung ein und diskutiert Techniken, die die in x86-PCs typische Hardware virtualisieren.

1.1 Über dieses Dokument

Im folgenden Kapitel wird vorhandene Hard- und Software analyisiert. Aus diesen Ergebnissen wird in Kapitel 3 ein Entwurf abgeleitet, dessen Implementierung in Kapitel 4 beschrieben wird.

1.2 Danksagung

An dieser Stelle möchte ich allen danken, die zum Gelingen dieser Arbeit beigetragen haben. Das sind neben den Korrekturlesern, die mir viele Hinweise gaben, insbesondere Prof. Härtig und mein Betreuer Michael Hohmuth.

Meinen Freunden danke ich für die schöne Zeit, die ich mit ihnen verbrachte.

Ein besonders herzliches Dankeschön an meine Eltern, die mit ihrer Liebe und Hingabe unter anderem mir eine Ausbildung ermöglichten, die schließlich in dieser Arbeit mündete.

Und natürlich nicht zu vergessen meine Freundin, die sich so lieb um mich gekümmert hat!

1.3 Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kapitel 2

Analyse

2.1 Das Problem

Aus der Einleitung sind zwei Ziele zu entnehmen: Anwendungen, die für verschiedene Betriebssysteme erstellt wurden, sollen gleichzeitig benutzt werden können, und für die Entwicklung von Betriebssystemkernen soll ein Debugwerkzeug geschaffen werden, welches erlaubt, den neuen Kern zu entwickeln und zu testen, während auf dem Rechner ein stabiles System läuft.

2.2 Naiver Ansatz

Der naive Ansatz ist, einfach mehrere Betriebssysteme gleichzeitig auszuführen. Warum das nicht funktioniert, wird im Folgenden erläutert.

Die Aufgaben eines Betriebssystems sind: die Hardware steuern, Ressourcen verwalten und Anwendungen störungsfrei und geschützt voreinander parallel laufen zu lassen. Dabei geht jedes Betriebssystem davon aus, dass es exklusiven und vollständigen Zugriff auf den gesamten Rechner hat. Zum einen suggeriert dies die Hardware, denn sie ist nicht darauf vorbereitet, von mehreren Instanzen verwaltet zu werden, zum anderen wurden im Laufe der Entwicklung der PC-Betriebssysteme keine Anstrengungen unternommen, Mechanismen zu schaffen, um selbige kooperativ oder konkurrierend simultan ausführen zu können[1]. Daraus ergibt sich, dass Betriebssysteme ohne Änderungen an deren Quelltext oder sogar ihrem Design nicht zu "zähmen" sind. Eine spezielle Software ist nötig, die in der Lage ist, mehrere Systeme gleichzeitig kontrolliert auszuführen. Diese Software wird hier als "Monitor für virtuelle Maschinen" oder "Virtual Machine Monitor" (VMM), kurz Monitor bezeichnet.

2.3 Mögliche Lösungen

Grundsätzlich sind zwei Ansätze denkbar:

- Ein Betriebssystem läuft wie üblich. Jedes weitere steht unter der Kontrolle des Monitors. Das erste wird als "Wirt" bezeichnet, die anderen als "Gäste".
- Alle laufenden Betriebssysteme stehen unter der Kontrolle des Monitors. In der eben verwendeten Terminologie gesprochen bedeutet das: Es gibt keinen Wirt, nur Gäste.

[1] Grund dafür mag sein, dass Firmen natürlich kein Interesse haben, dass ein Nutzer neben ihrem Betriebssystem noch ein anderes einsetzt. Im Gegenteil: Es werden manchmal sogar parallel installierte Systeme außer Kraft gesetzt oder gelöscht.

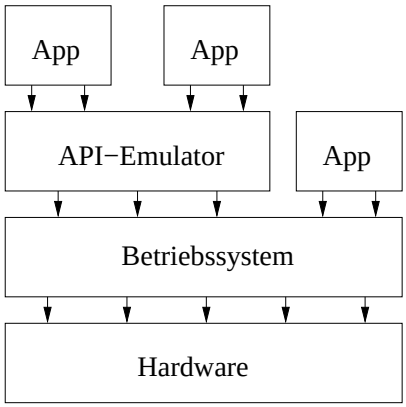

Abbildung 2.1: Emulation der API eines Betriebssystems

Als Lösungen des in Abschnitt 2.1 aufgeworfenen Problems kommen drei Verfahren in Frage: Emulation der Programmierschnittstelle des Betriebssystems (API), Emulation eines kompletten Rechners und Virtualisierung des Rechners. Im Folgenden werden die drei Verfahren genauer betrachtet.

2.3.1 API emulieren

In vielen Fällen ist der Anwender nur an den Applikationen interessiert, nicht am Betriebssystem selbst. Sollen also nur Anwendungsprogramme, keine weiteren Betriebssystemkerne selbst, laufen, bietet sich an, nur die API des zweiten Betriebssystems zu emulieren. Systemrufe werden übersetzt und eventuell nötige Bibliotheken bereitgestellt. Das Verfahren ist sehr schnell, ein zweiter Vorteil ist, dass keine Lizenz des zweiten Betriebssystems nötig ist. Der Nachteil ist hier, dass für jede Kombination Wirt – Gast eine API-Übersetzung existieren muss. Solche Übersetzungen sind aufgrund der teilweise sehr umfangreichen und schlecht dokumentierten APIs nur mit großem Aufwand zu erstellen, und natürlich kann man auf diese Weise keine Betriebssystemkerne testen, wie es für die Betriebssystementwicklung interessant ist.

Ein Beispiel ist WINE ([15]).

2.3.2 Emulieren

Bei der Emulation wird ein ganzer Rechner komplett in Software nachgebildet. Das heißt, CPU und Geräte sind Datenstrukturen, die von der Emulationssoftware verändert werden. Zur Ein- und Ausgabe benutzt der Emulator den Wirt. Alle Befehle werden von der Software geladen, dekodiert und die Wirkung auf die Datenstrukturen ausgeführt.

Auf dem emulierten Rechner läuft ein Betriebssystem mit entsprechenden Anwendungen. Die emulierte Architektur kann eine ganz andere sein als die, auf der die Emulation läuft. Diesem Vorteil steht der Nachteil gegenüber, dass das Verfahren der Emulation sehr langsam und aufwändig ist.

2.3.3 Virtualisieren

Wenn ein Emulator die gleiche Hardware wie die nachahmt, auf der er läuft, liegt der Gedanke nahe, nicht alles in Software nachzubilden, sondern gleich die “echte” Hardware zu verwenden. Dies ist nicht uneingeschränkt möglich, denn es gibt gewisse Restriktionen. Die meisten Befehle können von der CPU unverändert ausgeführt

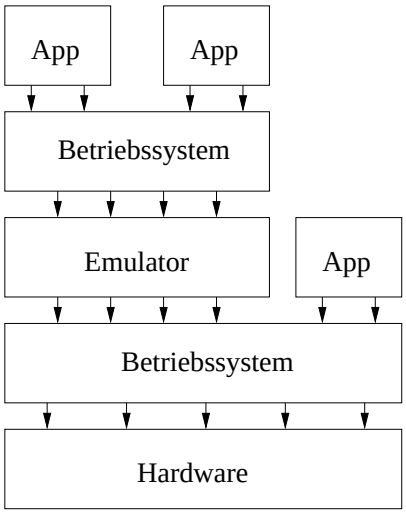

Abbildung 2.2: Emulation der gesamten Hardware

werden, aber einige führen zu Kollisionen, wenn mehrere Betriebssysteme gleichzeitig aktiv sind. Diese "kritischen" Befehle betreffen vor allem die Ein- und Ausgabe sowie die Speicherverwaltung und die Schutzmechanismen. Sie dürfen nicht wirklich, sondern nur virtuell ausgeführt werden, wobei der Monitor die Virtualisierung vornimmt. Er beobachtet die Programmausführung, findet die problematischen Befehle und emuliert sie auf geeignete Art und Weise. Mit anderen Worten: zwischen Betriebssystem und Hardware wird eine zusätzliche Schicht, der VMM, eingeschoben, die die virtualisierte Hardware bereitstellt (Abbildungen 2.3 und 2.4). Die virtualisierte Hardware setzt sich zusammen aus der realen und der emulierten. Zu virtualisieren sind der Prozessor, die Geräte (I/O) sowie der Hauptspeicher und die Schutzmechanismen. Diese Komponenten sind genau zu untersuchen um herauszufinden, was konkret zu virtualisieren ist und auf welche Weise das geschehen kann. Eine detaillierte Vorstellung der Hardware ist z.B. in [3] oder [4] zu finden, für deren Entwurf wird in Kapitel 3 eine formale Beschreibung angegeben.

Gegenüber der Emulation wird bei der Virtualisierung eine hohe Geschwindigkeit erreicht, und es ist keine aufwändige API-Emulation nötig. Von Nachteil ist, dass die x86-PC-Hardware allgemein als nicht bzw. schwer virtualisierbar gilt. Das lässt vermuten, dass für die Virtualisierung, so sie denn gelingt, einiger Aufwand betrieben werden muss.

2.4 Verwandte Arbeiten

Zur Einordnung dieser Arbeit in den Stand der Technik soll ein kurzer Überblick über ähnliche Aktivitäten gegeben werden. Das Hauptaugenmerk liegt dabei auf den Vorbildern S/3x0 und VMware sowie dem freien PC-Emulator Bochs.

2.4.1 IBMs Mainframes

Die Idee der Virtualisierung der Hardware ist nicht neu. Bereits in den 60er Jahren beschäftigten sich Forscher mit der Virtualisierung der S/360-Hardware[2]. Das

[2] Allerdings nicht bei IBM, dort waren Bestrebungen, "Time Sharing Systems" statt im Batchmodus betriebener Systeme zu entwickeln, lange Zeit verpönt. Entsprechend war die Hardware nicht dafür ausgelegt, zusätzlich notwendige Teile mussten von den "Virtualisierern" selbst nachgerüstet werden.

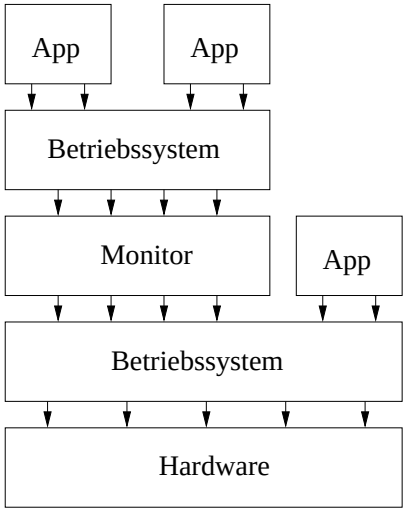

Abbildung 2.3: Virtualisierung mit Wirt und Gast

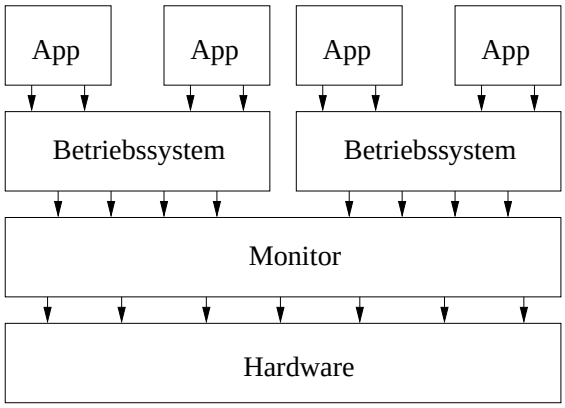

Abbildung 2.4: Virtualisierung ohne Wirt

primäre Ziel war, statt des Batchbetriebes ein "Multi User System" zu bauen. Die Rechenleistung einer Mainframe sollte mehreren Benutzern zur selben Zeit angeboten werden. Dazu ist mindestens virtueller Speicher nötig, doch die Entwickler gingen gleich einen Schritt weiter und virtualisierten nicht nur den Hauptspeicher, sondern gleich die ganze Maschine. Hauptzweck war der Schutz der Nutzer voreinander. 1967 konnte nach ca. dreijähriger Entwicklung das erste System mit virtuellen Maschinen in den Anwendungsgebrauch genommen werden.

Von den damals gesammelten Erfahrungen kann man auch bei der Virtualisierung eines PC profitieren. Ein Beispiel dafür ist die Architektur des Monitors. Das System wurde in zwei bedeutende Teile geteilt:

- Das "Control Program" CP; es virtualisiert die Hardware, so dass mehrere virtuelle auf eine reale Maschinen abgebildet werden; anders gesagt: es dupliziert das "System/360 architecture interface".

- Das "Console Monitor System" CMS; es stellt die eigentliche Betriebssystemfunktionalität zur Verfügung; CMS ist ein Single-User-System, das auch allein auf der Hardware laufen würde.

Daher auch die Bezeichnung CP/CMS. Es existieren klare Schnittstellen zwischen CP und CMS. Die Benutzung der Geräte durch CMS geschieht in spoolartiger Weise.

2.4.2 VMware

Die x86-Hardware galt solange als nicht oder kaum virtualisierbar, bis es einem Entwicklerteam[3] gelang, mit einem verfügbaren, gebrauchsfertigen Produkt das Gegenteil zu zeigen. Es handelt sich dabei um ein kommerzielles, geschlossenes Projekt. Daher erfolgt die Vorstellung nur aus Nutzersicht mit Bezug auf die frei verfügbare Testversion 2.0 für Linux, mit der eine 30tägige Evaluation möglich ist. Es sind keine internen Verfahren oder Details bekannt. VMware soll als Beispiel dafür dienen, wie sich ein Monitor für virtuelle Maschinen in eine bestehende Umgebung eingliedern und dem Anwender präsentieren kann.

Als Systemvoraussetzungen werden mindestens ein Pentium II mit 266 MHz und 96 MB (besser 128 MB) RAM genannt. Dies deutet darauf hin, dass die Virtualisierung eines x86-PC ein ressourcenaufwändiges Problem ist.

Die Installation muss vom Administrator durchgeführt werden. Dabei werden unter /dev Dateien für Geräte angelegt. Des weiteren werden zwei zusätzliche Module für den Betriebssystemkern installiert. Neben dem Hauptpaket sind weitere Tools verfügbar, welche die Geschwindigkeit des Systems (Wirt, Monitor, Gast) erhöhen sollen. Sie wurden aber nicht untersucht.

Nach dem Start stellt sich der VMM als Fenster mit Menü und Knöpfen dar, die Funktionalitäten wie z.B. "Power On", "Reset", "Full Screen" und "Suspend" bereitstellen. Nach dem "Power On" startet ein virtueller Rechner im Real Mode, wie es von realer Hardware bekannt ist. An Geräten stehen neben dem virtuellen Floppy-Laufwerk ein "VMware Virtual IDE Hard Drive" und ein "VMware Virtual IDE CDROM Drive, ATAPI" zur Verfügung. Bei der Netzwerkverbindung kann man zwischen zwei verschiedenen Modi wählen. Die Festplatte wird in Form einer Datei bereitgestellt, die im Dateisystem des Wirtes liegt. Diese Datei wächst bei Bedarf. Eine maximale Größe kann eingestellt werden. Das virtuelle CD-ROM-Laufwerk entspricht dem realen eingebauten Gerät. Im Leerlauf besteht VMware aus nur einem Prozess, im laufenden Betrieb wurden 7 gezählt.

[3] Darunter einige Entwickler des universitären "Disco"-Projektes [2]

Interessant sind die "Suspend" und "Resume"-Funktionen. Dabei wird der aktuelle virtuelle Maschinenzustand in eine Datei gespeichert und kann daraus später wieder vollständig restauriert werden[4].

2.4.3 Bochs

Verwandt mit den VMM sind die Emulatoren. Ein Vertreter dieser Art ist Bochs, welches seit kurzer Zeit unter der LGPL ([10]) verfügbar ist.

Bochs versteht sich selbst als "portable x86-PC-Emulationssoftware", die genug von der x86 CPU, der AT-Hardware und des BIOS emuliert, um DOS, Windows 95, Minix 2.0 und andere Betriebssysteme laufen zu lassen. Getestet wurden als Host unter anderem Sparc-, x86-, PowerPC- und MIPS-Rechner, als Wirtsbetriebssystem Solaris, Linux, BeOS, IRIX, AIX und Windows sowie die im Emulator laufenden Windows 95 und DOS.

Da Bochs frei verfügbar ist, kann es Hilfe in technischer Hinsicht bieten. Falls nötig, können Programmteile, die Lösungen für Teilprobleme der Virtualisierung bieten, diesem Softwarepaket entnommen werden. Die Lizenz erlaubt sowohl die Verwendung in freier wie auch kommerzieller Software.

2.4.4 Andere Arbeiten

Andere Arbeiten seien hier nur stichpunktartig genannt:

- DOS-Boxen in Windows und OS/2, dosemu: Ausführung von DOS-Programmen im V86-Mode
- Windows in OS/2: Ausführen von Windows-Applikationen unter OS/2
- WINE: Ermöglicht das Ausführen von Windowsapplikationen unter Linux durch Emulation der Windows–API ([15])
- SimOS: Umgebung, die eine komplette Maschine simuliert ([16])
- Disco: Ein VMM, um normale Anwendungen auf großen Multiprozessorsystemen laufen zu lassen ([2])
- Brown-Simulator: Ein "High-Level"-Maschinen-Simulator ([18])
- L4Linux: Portierung eines Linuxkerns auf das L4-Interface ([14])
- Sheepshaver: MacOS-Laufzeitumgebung, um auf PowerPC-Rechnern unter BeOS MacOS-Applikationen laufen zu lassen ([19])
- Mac-on-Linux: dito für Linux statt BeOS ([20])
- a386: Bibliothek zur Abstraktion eines i386 im Protected Mode; emuliert dessen privilegierte Befehle ([21])
- User-Mode-Linux: Portierung des Linuxkerns auf sein eigenes Systemrufinterface ([22])

[4] Man bezeichnet das auch als "Hibernation".

2.5 Vorstellung des plex86–Projektes

Im Laufe dieser Arbeit stellte sich heraus, dass die Virtualisierung einer Hardware ein sehr großes und komplexes Projekt ist. Die Suche nach verwandten Arbeiten zum Ermitteln des Standes der Technik führte schnell zum plex86, welches früher "FreeMWare" hieß[5]. Als typisches OpenSource-Projekt wird es von einer internationalen Gemeinde entwickelt. Als Gründer leitet Kevin Lawton, welcher schon Hauptentwickler des PC-Emulators "Bochs" war, auch plex86. Das Internet dient zur Kommunikation und Verbreitung des Quellcodes. Als Architektur wurde die Wirt-Gast-Struktur gewählt. Zur Zeit wird Linux als einziger Wirt unterstützt. Unter diesen günstigen Voraussetzungen fiel der Entschluss leicht, die Diplomarbeit der Weiterentwicklung von plex86 zu widmen.

Im folgenden soll nun stichpunktartig der Stand des Projektes skizziert werden, wie es zu Beginn der Arbeit vorgefunden wurde:

- Aufsatz: "Running multiple operating systems concurrently on an IA32 PC using virtualization techniques ", 1. Fassung
- Nutzertask als einfache Konsolenanwendung
- Linux-Kernelmodul
- Monitor als Teil des Kernelmoduls implementiert
- Einfachster Testkern: Variable in Endlosschleife inkrementieren
- Mechanismen zur Kommunikation:
  - Nutzertask mit Kernmodul
  - Wirt mit Gast
  - Gast mit Monitor
- Laden von Binärobjekten
- Reservierung von RAM und CPU-Zyklen

2.6 Die Hardware

Es würde den Rahmen dieser Arbeit bei weitem sprengen, die Hardware eines x86-PC in dem Maße zu beschreiben, wie es für das folgende Kapitel nötig wäre. Daher wird auf Literatur wie die "Intel Architecture Software Developer's Manuals" ([7], [8], [9]), das "PC-Hardwarebuch" ([3]) oder "80486 Systemsoftware-Entwicklung" ([4]) verwiesen.

[5] Auf Grund vieler Anfragen und der zu großen Ähnlichkeit zum Begriff "Freeware" wurde das Projekt Anfang 2000 umbenannt.

Kapitel 3

Entwurf

In diesem Kapitel wird beschrieben, welche Techniken zur Virtualisierung der x86-Architektur verwendet werden können. Als Grundlage wird in Abschnitt 3.2 eine detaillierte Strukturierung dieser Hardware angegeben. Darauf aufbauend werden in Abschnitt 3.3 Lösungen für die Teilprobleme der Virtualisierung von Prozessor, RAM, Geräten und Systemstrukturen diskutiert. Aus den Teillösungen werden in Abschnitt 3.4 Gesamtlösungen mit Vorschlägen zur Architektur zusammengesetzt.

Da das Kapitel sehr umfangreich ist, soll eine Grafik (Abbildung 3.1) das Vorgehen verdeutlichen.

3.1 Entwurfsziele

Zunächst erfolgt die Definition der Entwurfsziele:

- Der Monitor behält die Maschinensteuerung, der Gast soll die Kontrolle nicht übernehmen können
- Hohe Performance
- Jedes Betriebssystem kann als Gast laufen
- Keine Veränderungen am Wirt, außer zusätzliche Treiber, um die Umschaltung zum Monitor vorzunehmen und die Performance des Systems zu erhöhen
- Keine Veränderungen am Gast, außer zusätzliche Treiber, um die Performance des Systems zu erhöhen
- Keine Veränderungen an der Hardware

Aber zunächst gilt es, eine lauffähige Version zum Testen zu schaffen. Dafür können durchaus mehrere Entwurfsziele zunächst unbeachtet bleiben. Zum Beispiel ist es günstig, zunächst eine einfache, aber wenig performante Lösung zu implementieren, um Charakteristika der Virtualisierung herauszufinden. Zum Thema Virtualisierung, speziell der des x86-PC, gibt es nur wenige Veröffentlichungen. Mit den gesammelten Erfahrungen sowie erfolgten Messungen kann dann das Design überarbeitet werden, um sich den Entwurfszielen zu nähern.

3.1.1 Neuentwurf vs. Anpassung

Es stellt sich die Frage, ob es günstiger ist, vorhandene Software zu nutzen und an die gegebenen Erfordernisse anzupassen oder mit der Entwicklung, speziell dem Entwurf, von vorn zu beginnen. Als Ausgangspunkt würde sich z.B. Bochs (siehe

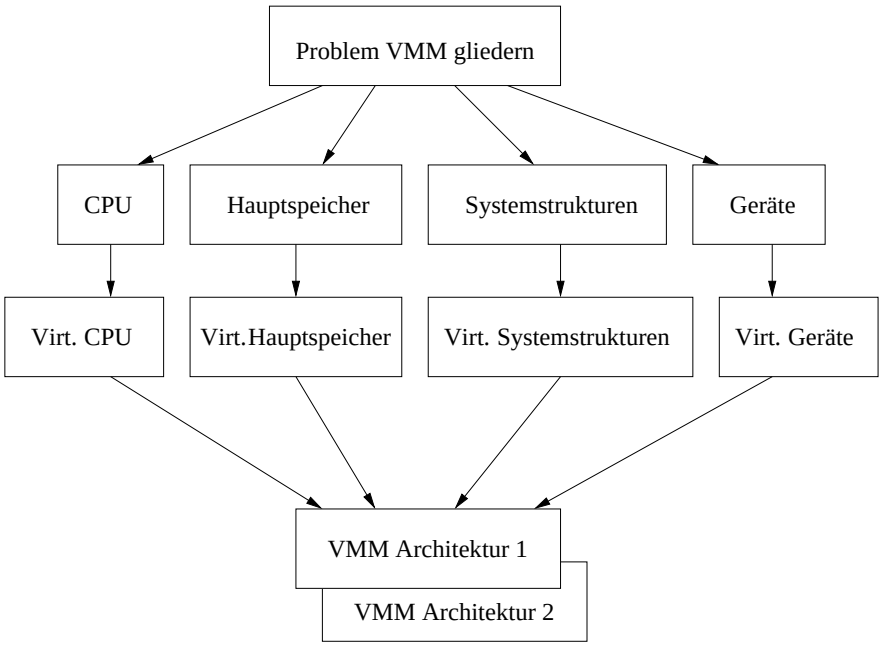

Abbildung 3.1: Aufbau des Entwurfskapitels

Abschnitt 2.4.3) anbieten. Schritt für Schritt könnte man die Emulation durch die Ausführung auf der verwendeten Hardware ersetzen.

Um eine bestmögliche Architektur zu erreichen, ist ein Neuentwurf aber sicher besser, denn dabei muss man nicht auf einschränkende Gegebenheiten der Ausgangssoftware Rücksicht nehmen und kann sich immer frei für die bestmögliche Lösung entscheiden.

Die für plex86 gewählte Lösung ist ein Kompromiss: es wird neu entworfen, jedoch an Stellen, an denen es sich lohnt, auf Kompatibilität zu Bochs geachtet, so dass Teile von diesem übernommen werden können.

3.2 Detaillierte Strukturierung

Eine detaillierte Strukturierung der Hardware und Systemstrukturen soll die Grundlage für den Entwurf bilden, wobei ein Monitor für eine virtuelle Maschine (VMM) eines x86-PC beschrieben wird. Dabei wird von einem VMM als Wurzel ausgegangen, nach und nach werden dann Teile dekomponiert, bis eine hinreichend feine Aufgliederung erreicht wurde.

Bei einem VMM ist es wichtig, dass er vollständig ist. Das heißt, eine virtuelle Maschine sollte ein möglichst genaues Abbild der realen Maschine sein. Die Strukturierung stellt also ein Werkzeug dar, mit dessen Hilfe die Vollständigkeit sichergestellt wird. Gleichzeitig soll sie helfen, ein so kompliziertes Gerät wie einen PC geeignet vereinfacht darzustellen.

Im Abschnitt 3.3 werden Techniken vorgestellt, die virtuelle auf reale Maschinen abbilden.

In dem hier verwendeten Modell besteht der VMM aus einer Anzahl virtueller Maschinen, einer realen Maschine, einer Anzahl Funktionen, die die virtuellen auf die reale Maschine abbilden und Datenstrukturen zur Speicherung von Informationen über die VMM.

3.2.1 Die reale Maschine

Eine reale Maschine besteht aus einem oder mehreren Prozessoren[1], auch CPU genannt, Hauptspeicher, Geräten und Systemstrukturen zur Steuerung der Maschine.

Die CPU

Für die Virtualisierung sind die Befehle, Register, TLBs und Caches des Prozessors von Interesse.

Die Befehle Je nach Art der CPU (Hersteller, Prozessorgeneration) unterscheidet sich die Menge der Befehle. Für einen VMM sind vier Gruppen von Befehlen zu unterscheiden:

- Privilegierte Befehle
- Nichtprivilegierte Befehle
- Sich variabel verhaltende Befehle (Verhalten kann eingestellt werden)
- Befehle, deren Verhalten von der Privilegstufe (Ring) abhängig sind, in dessen Kontext sie ausgeführt werden

Die Register Eine moderne CPU hat eine Vielzahl von Registern. Sie können in verschiedene Arten gruppiert werden:

- Speicherverwaltungsregister
- Steuerregister
- Mehrzweckregister
- Befehls- und Stackzeiger
- Flagregister
- Segmentregister
- Debugregister
- Testregister

Die Namen lassen größtenteils auf die Funktionen schließen. Eine genaue Beschreibung ist zum Beispiel in [3] zu finden.

Der TLB Der Translation Lookaside Buffer speichert Zuordnungen von virtuellen zu physischen Adressen mit der Granularität ganzer Seiten zwischen. Bei allen neueren Prozessoren ist der TLB geteilt in einen für Befehle und einen für Daten.

Die Caches Caches befinden sich außerhalb oder innerhalb des Prozessors. Da das außer eventuell für die Performance keine Auswirkungen hat, kann man Caches der Einfachheit halber als zur CPU gehörig annehmen. Etwas anders kann es sich darstellen, wenn Multiprozessorsysteme unterstützt werden sollen.

Die Cachearchitektur heutiger Systeme sieht mindestens einen First- und Second-Level-Cache vor. Bei einigen findet man sogar einen Third-Level-Cache. Weiterhin können Caches in Befehls- und Datencaches unterteilt sein. Vermutlich kann man daraus weder besonderen Nutzen ziehen, noch ist es hinderlich. Alle Caches sind für Software transparent. Daher wird der Cache im folgenden meist außer acht gelassen.

[1] Eine Vereinfachung für eine erste Version eines VMM ist, dass sich nur eine CPU im Rechner befindet. Dies erspart zusätzlichen Aufwand, der zur Synchronisation notwendig wäre.

Der Hauptspeicher

Verschiedenartige Speicher haben sich im Laufe der Zeit etabliert:

- Konventioneller Speicher
- Erweiterungsspeicher
- Von Geräten eingeblendeter Speicher
- ROM

Die Unterscheidung zwischen konventionellem und Erweiterungsspeicher entfällt, da nur der Protected Mode betrachtet wird. Verbleiben noch der eingebaute Hauptspeicher (RAM) und der Festwertspeicher (ROM).

Die Geräte

Üblicherweise werden in allen Rechnern einige Geräte eingebaut. Sie lassen sich charakterisieren durch:

- Mit diesem Gerät assoziierte Interrupts
- I/O-Ports
- Bereiche im Adressraum des Prozessors für "Memory Mapped IO"
- Die Fähigkeit, direkt (ohne die CPU in Anspruch zu nehmen) auf den Hauptspeicher des Rechners zuzugreifen
- Protokolle, mit dem das Gerät mit seinem Treiber kommuniziert
- Innere Zustände, in denen sich das Gerät befinden kann

Die Systemstrukturen

Relevante Systemstrukturen sind Seitentabelle (Page Table, PT), Globale Deskriptortabelle (GDT), Lokale Deskriptortabelle (LDT) und Interruptdeskriptortabelle (IDT).

Seitentabellen Seitentabellen bestehen aus einzelnen Seitenverzeichnissen, die jeweils 1024 Tabelleneinträge aufweisen. Jeder dieser Einträge hat sieben Komponenten:

- "Page Frame Address"; physische Adresse des Seitenrahmens
- "Available"; für das Betriebssystem verfügbar
- "Dirty"; auf diese Seite wurde (noch nicht) schreibend zugegriffen
- "Accessed": auf diese Seite wurde (noch nicht) zugegriffen
- "User/Superuser"; auf diese Seite darf in allen Schutzstufen/nur von Ring 0 aus zugegriffen werden
- "Read/Write"; nur lesen/lesen und schreiben ist erlaubt
- "Present"; diese Seite befindet sich (nicht) im Speicher

Globale Deskriptortabellen In den Globalen Deskriptortabellen sollten sich mindestens Deskriptoren für ein Code- und ein Stacksegment befinden, maximal sind 8191 Einträge möglich — der erste Eintrag muss immer frei bleiben. Ein Deskriptor beschreibt ein Segment. Die Einträge in der GDT kann man nach "Applikationssegmentdeskriptoren" und "Systemsegmentdeskriptoren" klassifizieren.

Lokale Deskriptortabellen Die lokale Deskriptortabelle kann 4096 Deskriptoren aufnehmen. Allerdings sind hier nur Applikationssegmentdeskriptoren erlaubt.

Interruptdeskriptortabellen Die Interruptdeskriptortabelle nimmt Deskriptoren für Task Gates, Interrupt Gates und Trap Gates auf. Es sind maximal 256 für die 256 Interruptquellen eines x86-PC möglich.

Applikationssegmentdeskriptoren Die einzelnen Bestandteile dieser Deskriptoren sind:

- Basisadresse des Segments
- Größe des Segments
- Typ des Segments
- Typ des Deskriptors; hier immer 1
- Privilegstufe des Deskriptors
- Present
- Für Betriebssystem verfügbar
- Segmentgröße (16/32 Bit)
- Gibt an, ob Größe in Byte oder Speicherseiten zu interpretieren ist

Systemsegmentdeskriptoren Es gibt drei verschiedene Arten von Systemsegmentdeskriptoren: für Task State Segmente, Segmente, die eine lokale Deskriptortabelle enthalten und welche, die Tore darstellen.

Task State Segmente enthalten einen Verweis auf den Vorgänger, soweit vorhanden, einen Stackpointer für jede Privilegstufe, bestehend aus ss und esp, den kompletten Satz der Vielzweckregister, den eip, die EFlags, die Werte aus dem Page Directory Base Register (PDBR) und Local Descriptor Table Register (LDTR), die Segmentselektoren und einen Verweis auf die Basisadresse der I/O-Bitmap.

Für Deskriptoren, die Lokale Deskriptortabellen beschreiben, sind die vier Werte Basisadresse, Größe, Present und DPL (Descriptor Privilege Level) relevant.

Schliesslich noch die Deskriptoren für Tore, sie enthalten neben dem Selektor und dem Offset ebenfalls ein DPL- und ein P-Feld, ferner einen Typ, der die Art des Tores angibt und die Zahl der zu kopierenden Doppelwörter, die die CPU automatisch beim Passieren des Tores vom Stack der aufrufenden auf den Stack der aufgerufenen Prozedur kopiert.

3.2.2 Virtuelle Maschinen

Potenziell können in einem VMM mehrere virtuelle Maschinen laufen. Für den Zweck der Erfahrungssammlung reicht es hier allerdings aus, vorerst nur eine VM pro VMM zu betreiben.

Die virtuelle CPU, Hauptspeicher[2] , Geräte und Systemstrukturen werden analog zur realen Maschine dekomponiert.

3.3 Abbildung virtueller auf reale Maschinen

In diesem Abschnitt werden Funktionen zur Abbildung virtueller auf reale Maschinen vorgestellt und diskutiert. Der Grundsatz lautet, soweit wie möglich die reale Maschine zu nutzen. Was auf diese Weise nicht ausgeführt werden kann oder darf, muss emuliert werden.

3.3.1 Abbildung der virtuellen CPU auf eine reale CPU

Es gibt eine Vielzahl von Prozessorgenerationen und -variationen. Da gerade auch Betriebssysteme dazu tendieren, aus Gründen der Performance und der Genauigkeit sich gut an die Prozessoren anzupassen, dürfen die Unterschiede nicht außer acht gelassen werden. Als erste Näherung dürfte jedoch genügen, von einer komplexen, leistungsstarken CPU eine einfachere zu virtualisieren, das heisst, der Gast "sieht" nur einen einfachen Prozessor, dieser wird aber auf einen komplizierten abgebildet. Zum Beispiel kann von einem geteilten Befehls- und Daten-TLB zu einem gemeinsamen TLB übergegangen werden.

Moderne x86-Prozessoren kennen drei Betriebsmodi:

- Protected Mode
- Real Mode
- Virtual 86 Mode (V86)

Für moderne Betriebssysteme relevant ist nur der Protected Mode. Die beiden anderen Modi werden nur noch aus Kompatibilitätsgründen mit sehr alten Betriebssystemen unterstützt. Daher liegt der Fokus in der Entwicklung vorerst hauptsächlich auf dem Protected Mode.

Befehle

Im Abschnitt 3.2 wurden die Befehle in vier Klassen eingeteilt:

- Privilegierte Befehle müssen abgefangen und emuliert werden, denn sie beeinflussen den Zustand der Maschine oft in einer für den Monitor kritischen Weise. Ein Beispiel ist das Laden des PDBR, womit der Monitor die Kontrolle über den Hauptspeicher verlieren würde.
- Nichtprivilegierte Befehle können unmodifiziert ausgeführt werden. Ausnahmen bilden dabei die beiden folgenden Klassen.
- Befehle mit ringabhängigem Verhalten gilt es *vor* ihrer Ausführung aufzufinden und zu emulieren. Hierbei hilft jedoch nicht wie bei den privilegierten die CPU, das Filtern muss mit Software erledigt werden. Entsprechende Verfahren heissen "Scan Before Execution" (SBE) oder "Prescan".
- Befehle, deren Verhalten konfiguriert werden kann, gehören zur Gruppe der privilegierten oder per Prescan zu findenden.

Die folgenden Abschnitte beschreiben, wie mit den Befehlen dieser Klassen verfahren wird.

[2] Diesen "virtuellen Speicher" nicht mit dem bekannten virtuellen Speicher verwechseln. Es wird die Terminologie "virtueller Hauptspeicher" für den Speicher verwendet, von dem das Gastbetriebssystem annimmt, es handele sich im physisch im Rechner eingebauten RAM. "Virtueller RAM" bedeutet das gleiche.

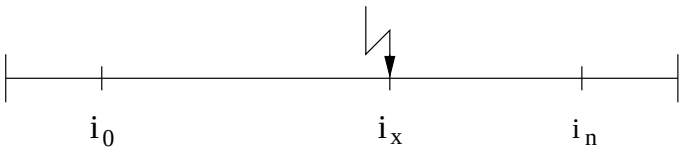

Abbildung 3.2: Programm mit einem kritischen, nichttrappenden Befehl

Privilegierte Befehle

Dem Gast darf zu keiner Zeit die Ausführung solcher Befehle gestattet werden. Ermöglicht werden kann dies, indem dem Gast eine Privilegierungsstufe größer 0 zugewiesen wird. Falls versucht wird, einen privilegierten Befehl in einer solchen höheren Privilegierungsstufe auszuführen, unterbricht die CPU die Programmausführung und führt die Ausnahmebehandlung "Allgemeine Schutzverletzung" durch. Diese Ausnahmebehandlung kann ein Stück Monitorcode sein, der den entsprechenden Befehl emuliert. In Anhang B befindet sich eine Liste der Befehle mit den dazugehörigen Aktionen bei der Emulation. Nach erfolgter Emulation wird mit dem nächsten Befehl im Gast fortgefahren.

Befehle mit ringabhängigem Verhalten

Viel schwerer als die oben beschriebenen Befehle sind diese Instruktionen zu handhaben. Ihr Verhalten hängt von dem Ring ab, in dem sie ausgeführt werden. Da der Gast statt in Ring 0 nun auf der Privilegstufe ungleich 0 läuft, verhalten sich diese Befehle anders, als es der Gast erwartet. Beispiele sind "verr", "verw" oder "popf". Aber auch durch einige Mechanismen zur Virtualisierung können Befehle, deren Verhalten normalerweise unproblematisch ist, zu solchen werden, die nicht original ausgeführt werden dürfen. Ein Beispiel ist das Lesen aus Segmentregistern, der Grund wird in Abschnitt 3.3.3 erläutert. In Abbildung 3.2 ist i_x ein solcher Befehl — das Verhalten des Befehles widerspricht den Erwartungen des Gastes. Durch geeignete Methoden müssen solche Befehle *vor* ihrer Ausführung gefunden und emuliert werden. Abbildung 3.3 verdeutlicht, dass statt des Befehles i_x ein Sprung in die Emulationsbibliothek stattfindet, wo sich die Routinen zur Emulation von Befehlen befinden. Nach Ende der Emulation wird der Gast ab dem Befehl i_{x+1} fortgesetzt. Hier kommen einige Faktoren erschwerend hinzu:

- Das Befehlsformat der IA32-CISC-Architektur ist kompliziert
- Es ist überlappender Code möglich; ein Befehl kann teilweise oder ganz ein Teil eines anderen Befehls sein
- Der Code kann während der Laufzeit modifiziert werden: "Self Modifying Code" (SMC)

Zwei Möglichkeiten sind zum Auffinden dieser Befehle denkbar:

- Den Ausführungspfad des Gastes verfolgen: An bestimmten Punkten — meist vor der Ausführung eines der hier betrachteten Befehls — wird der Gast gestoppt, der Befehl wird emuliert und es wird der Pfad, den der Gast beim Fortsetzen beschreitet, nach weiteren kritischen Befehlen abgesucht. An bedingten Sprüngen kann sich der Pfad verzweigen, so dass mehrere Stellen mit Haltepunkten versehen werden müssen. Jeder Befehl muss dabei von einem Softwaredecoder behandelt werden. Abbruchbedingen für die Pfadverfolgung werden später diskutiert.

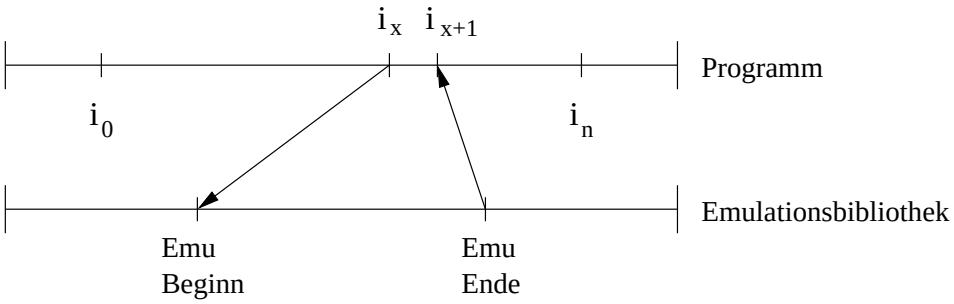

Abbildung 3.3: Erzwungene Emulation eines Befehls

- Lineare Suche in größeren Blöcken: Die Befehle eines Blockes werden von vorn nach hinten untersucht, bevor der Gast den Code dieses Blockes ausführt. Ein Block kann z.B. eine Speicherseite oder sogar der komplette Betriebssystemkern sein.

Während ersteres aufwändiger ist, funktioniert es aber auch noch bei sich überlagernden Befehlen. Genau umgekehrt ist es bei der zweiten Methode. Sich überlagernde Befehle können nur schwer oder nicht gefunden werden, denn Befehle können an einer beliebigen Stelle des Speichers beginnen. Da man außerdem nur den Code betrachtet, der wirklich ausgeführt wird, wenn man dem Ausführungspfad folgt, fiel die Entscheidung zugunsten des ersten Verfahrens.

Der Code wird also auf dem Pfad verfolgt, den der Gast ausführen würde. Das ist im allgemeinen bis zum nächsten Sprung möglich. Falls es ein fester oder bedingter Sprung mit feststehender Zieladresse ist, kann man auch über den Sprung hinaus den Pfad verfolgen. Andernfalls muss man den Sprung emulieren, um den nächsten auszuführenden Befehl zu finden. Zwei Möglichkeiten, den Sprung zu emulieren, bieten sich an:

- Die Hardware führt den Sprung im Einzelschrittmodus aus
- Der Befehl wird in Software emuliert

Da bei der ersten Variante die Hardware die Arbeit verrichtet, ist der Aufwand gering und die Emulation ist sehr genau. Allerdings sind einige teure Kontextumschaltungen nötig, die man sich bei der Emulation in Software erspart. Mit Bochs ist ein erprobter Emulator vorhanden, den man verwenden kann. Somit hält sich der Implementierungsaufwand in Grenzen, so dass Variante zwei nichts im Wege steht und sie verwendet werden kann. Messungen müssen zeigen, welche der beiden Varianten in bestimmten Situationen schneller ist.

Gelangt man bei der Pfadverfolgung an einen bedingten Sprung, kann man beide Pfade verfolgen und so rekursiv absteigen. Abbruchbedingungen für den Abstieg sind:

- Übertreten der Grenzen einer Speicherseite
- Sprünge aus der aktuellen Seite
- Eine maximale Tiefe ist erreicht
- Erreichen von schon gescanntem Code
- Erreichen von Befehlen, die Virtualisierung verlangen

Dafür sind Verwaltungsinformationen nötig.

Findet man Instruktionen, die einer Emulation bedürfen, aber keine Exception auslösen, muss man durch geeignete Maßnahmen die Emulation erzwingen. Folgende Verfahren kann man in Betracht ziehen:

- Hardwarebreakpoints; sie stehen nur in begrenzter Anzahl zur Verfügung und sind deshalb für diesen Zweck ungeeignet
- Softwarebreakpoints; dafür stellt der Befehlssatz eine INT 3 genannte Instruktion zur Verfügung; der Breakpoint tritt in Form einer Exception auf, die der Monitor bearbeitet
- Ungültiger Opcode; beim Versuch, diese Operation auszuführen, wird eine entsprechende Exception ausgelöst, der Monitor kann reagieren
- Sprung in Emulationsbibliothek; dabei wird der zu emulierende Befehl durch einen Sprung direkt zu der Routine ersetzt, die die Emulation durchführt; dabei befindet sich die Emulationsbibliothek entweder im Adressraum des Gastes, oder es werden Tore benutzt; da oft auf Daten aus dem Monitor zugegriffen werden muss, bleibt nur letzteres

Allen vier in Frage kommenden Möglichkeiten ist gemein, dass sie Änderungen am Code des Gastes vornehmen müssen. Im Falle eines Softwarebreakpoints ist es nur ein Byte, bei einem Sprung sind es mehrere Byte. Da man sich nicht darauf verlassen kann, dass ungültige Operationscodes auch ungültig bleiben und INT 3 gegenüber einem Sprung den Vorteil hat, nur ein Byte ersetzen zu müssen, fiel die Wahl auf die Verwendung von Softwarebreakpoints.

Die Änderung des Codes führt direkt zu einem anderen Problem: Sich selbst überprüfender Code wird nicht mehr funktionieren. Der betreffende Code muss also ausführbar sein, aber das Lesen muss verhindert werden. Hier hilft ein Trick weiter, der angewendet werden kann, wenn die CPU über geteilte Daten- und Code-TLBs verfügt. Glücklicherweise ist das bei jedem modernen Prozessor der Fall. Hier kann man im Code-TLB eine Zuordnung von virtueller zu physischer Adresse etablieren, wohingegen diese Zuordnung im Daten-TLB nicht existiert und die betreffende Speicherseite auch nicht in den Adressraum des Gastes eingeblendet ist. Somit ist das Ausführen des Codes zwar möglich, sobald aber lesend (oder schreibend) darauf zugegriffen wird, wird ein Seitenfehler ausgelöst. Der Monitor fängt diesen ab und kann Maßnahmen ergreifen. Die plex86–Gemeinde hat mit einem kleinen Tool ([24]) die Durchführbarkeit dieser Idee überprüft ([25]).

Wenn man den Code ändert, muss man für den Fall, dass der Gast lesend auf ihn zugreift, auch das Original aufbewahren. Zudem sind Verwaltungsinformationen nötig: für jedes Byte einer Speicherseite wird vermerkt, ob der potentiell[3] dort beginnende Befehl schon einer Überprüfung unterzogen wurde oder nicht, und welche Maßnahmen ergriffen wurden. Im schlechtesten Fall sind also pro Speicherseite originalem Code zwei weitere Seiten erforderlich: eine, die den modifizierten Code beinhaltet, der dann von der CPU ausgeführt wird und eine mit den Verwaltungsinformationen. Neben dem Rechenaufwand steigt also auch der Speicheraufwand enorm an. Wenn man annimmt[4], dass die hier betrachteten Befehle nur relativ selten im Code vorkommen, kann man durch geschickte Verfahren viel Speicher sparen.

Für die restlichen Seiten, die nicht gerade in Ausführung befindlichen Code enthalten, gibt es drei Möglichkeiten:

[3] Sich überlappender Code!

[4] Gewissheit können auch hier nur Messungen bringen.

- Der Gast kann nicht auf sie zugreifen. Das U/S-Bit der Seitentabelle ist auf 0 gesetzt. Führt der Codeausführungspfad zu einer solchen Seite, wird der Monitor mit einem Seitenfehler benachrichtigt und kann entsprechend handeln. Jedoch führt auch jeder Lese- und Schreibzugriff auf diese Seite zu einem Seitenfehler. Für Code, der sich intensiv selbst beobachtet oder verändert, ist dies ungünstig.
- Der Gast kann nur lesend zugreifen. Lesezugriffe zur Codebeobachtung sind erlaubt, Schreibzugriffe beim Modifizieren führen zu einem Seitenfehler und müssen emuliert werden. Ein neues Problem tritt hierbei auf: gerade in Ausführung befindlicher Code kann an jede beliebige andere Stelle springen, da die Seiten mit dem Code lesend eingeblendet sind und bei Seitentabellen kein Unterschied zwischen "lesen" und "ausführen" besteht. Somit müssen auch sämtliche Sprungbefehle erzwungen emuliert werden, was den Aufwand erheblich steigert.
- Der Gast kann lesend und schreibend auf seinen Code zugreifen. Auch hier müssen aus dem genannten Grund alle Sprungbefehle emuliert werden, um die Kontrolle nicht zu verlieren. Zu beachten ist außerdem, dass der Gast seinen Code modifiziert haben könnte. Bevor der Ausführungspfad auf einer neuen Seite fortgesetzt werden kann, muss entweder geprüft werden, ob sich in ihr etwas geändert hat, oder der alte, schon virtualisierte Inhalt komplett verworfen werden.

Da man Speicherseiten in einer von-Neumann-Architektur nicht ansieht, ob sie Code oder Daten enthalten, müssten im Falle 1 alle Seiten des Gastes aus seinem Adressraum entfernt werden. Die Folge wäre ein enormes Maß an Seitenfehlern und ein großer Aufwand, sie zu behandeln. Auch die Betrachtung der Code- und Datensegmente hilft wenig, da große, flache Adressräume üblich sind, bei denen Code- und Datensegmente übereinander liegen.

Ob Variante zwei oder drei günstiger ist, hängt von den Eigenschaften des ausgeführten Codes ab. Experimente müssen den Weg zur Entscheidung weisen.

Mit einigen Optimierungen lässt sich die Performance erhöhen. Eine Möglichkeit ist, bereits übersetzten Code zu cachen, eine andere ist, mehrere Seiten als "Seitencluster" zu betrachten und immer gemeinsam zu betrachten, um Mehraufwand gegenüeber der Behandlung einzelner Seiten zu sparen.

Die restlichen Befehle

Wie man sieht, muss viel Aufwand getrieben werden, um kritische Befehle hinreichend genau virtualisieren zu können. Dabei sollte man aber nicht vergessen, dass die meisten Befehle unkritisch sind, d.h. sie können von der CPU unverändert ausgeführt werden. Die Geschwindigkeit entspricht dabei der der originalen Maschine. Daraus ergibt sich der signifikante Performancevorteil gegenüber Emulatoren.

Register

Die IA32 ist so konstruiert, dass ein Befehl, der in ein Register schreibt, das den Systemzustand beeinflusst, eine "Allgemeine Schutzverletzung" hervorruft. Somit ist eine einfache Kontrolle über die Systemregister möglich. Vom Gast gewollte Inhalte dieser Register müssen aufgehoben werden, damit sie beim Lesen zurückgeliefert werden können.

Das Lesen dieser Register ist hingegen problematischer, denn in einigen Fällen dürfen Systemregister von nichtprivilegierten Programmen ausgelesen werden. Es werden aber andere Ergebnisse geliefert, als es der Gast erwartet. Tritt dies auf, muss eine Emulation erzwungen werden.

Je nach Bedeutung der Register ist wie folgt mit ihnen zu verfahren:

- Speicherverwaltungsregister sind TR, LDTR, IDTR und GDTR. Sie beeinflussen den Zustand der Maschine in kritischer Weise und dürfen nicht mit den Werten belegt werden, die der Gast vorsieht.
- Steuerregister sind CR0, CR1, CR2, CR3 und CR4. Das bei den Speicherverwaltungsregistern Gesagte gilt auch hier.
- Vielzweckregister sind EAX, EBX, ECX und EDX. Sie sind unkritisch für den Systemzustand, der Gast kann sie mit seinen Werten belegen.
- Der Befehlszeiger ist EIP, er zeigt auf den nächsten auszuführenden Befehl. Für die performante Programmabarbeitung sollte er den vom Gast vorgesehenen Wert haben, das ist problemlos möglich.
- Der Stackzeiger ist ESP, er kann original belassen werden.
- Das Flagregister EFlags hat direkten Einfluss auf den Zustand der Maschine, es muss vom Monitor kontrolliert werden, Flags können nicht einfach vom Gast übernommen werden.
- Die Werte der Segmentregister CS, DS, ES, FS und GS sind mit den Einträgen in der GDT abzugleichen. Sie sollten so oft wie möglich Werte beinhalten, die der Gast verwenden würde. Dies ist nicht in jedem Falle möglich, wie im Abschnitt 3.3.3 diskutiert wird.
- Die Debugregister DR0, DR1, DR2, DR3, DR4, DR5, DR6 und DR7 werden vom Monitor nicht gebraucht und können dem Gast zur Benutzung überlassen werden. Das ist problemlos möglich, entsprechende Exceptions müssen zum Gast umgeleitet werden.
- Die Testregister TR3, TR4, TR5, TR6 und TR7 hängen direkt mit dem Zustand der Maschine zusammen und müssen emuliert werden.

TLB

Zur Zeit sind keine besonderen Vorkehrungen getroffen, um Einzelheiten des TLB zu virtualisieren. Erfahrungen mit Gastbetriebssystemen werden zeigen, ob dies notwendig ist.

Cache

Der Cache verlangt keine besondere Aufmerksamkeit. Er wird in den folgenden Betrachtungen nicht beachtet.

3.3.2 Abbildung des virtuellen RAM auf realen RAM

Eine der ersten Entscheidungen, die man treffen muss, ist, ob man den verfügbaren Hauptspeicher schon beim Booten in mehrere Partitionen für Wirt und Gäste teilt oder nicht. Damit entscheidet man sich auch für oder gegen Flexibilität in dieser Beziehung. Für eine Partitionierung spricht, dass die Speicherverwaltung für den Monitor einfacher wird und sich DMA-Transfers einfacher oder überhaupt erst virtualisieren lassen. Dagegen spricht, dass man sich schon beim Booten für eine feste (Höchst-)Anzahl virtueller Maschinen entscheiden muss und dass ungenutzter Speicher nicht von anderen Gästen oder dem Wirt verwendet werden kann. Die Nachteile wurden schwerwiegender als die Vorteile beurteilt, daher fiel die Entscheidung für die flexible Lösung und gegen das Partitionieren.

Bei der Virtualisierung des Hauptspeichers kann auf Mechanismen zurückgegriffen werden, die schon von der Hardware für diesen Zweck vorgesehen sind, denn der Hauptspeicher ist die einzige Komponente eines PC, dessen Virtualisierung vorgesehen ist und unterstützt wird. Besagte Mechanismen sind:

- Segmentierung und
- Paging

Vom Paging kann man intensiv Gebrauch machen, da es mächtige und flexible Möglichkeiten bietet. Man kann versuchen, nur durch Manipulieren der Segmente, nur durch Manipulieren der Seitentabellen oder durch beide Varianten kombiniert zum Ziel zu gelangen. Da in verbreiteten Betriebssystemen Paging aggressiv eingesetzt wird und durch alleiniges Manipulieren der Segmente viele Fälle nicht abgefangen werden können (z.B. wenn mehrere virtuelle auf dieselbe physische Seite zeigen), kommt Methode eins nicht in Frage. Mit dem Ziel, die Virtualisierung möglichst einfach zu gestalten und im Vertrauen darauf, dass der Pagingmechanismus flexibel genug ist, alle Anforderungen zu erfüllen, wurde Methode zwei gewählt.

Wie man erkennen kann, führt die Virtualisierung des RAM direkt zur Virtualisierung der Systemstrukturen. Es ist sogar hinreichend, nur die den Hauptspeicher verwaltenden Systemstrukturen zu virtualisieren. Alles weitere wird daher im folgenden Abschnitt erläutert.

3.3.3 Abbildung der virtuellen Systemstrukturen auf reale Systemstrukturen

Wenn die Systemstrukturen genau so übernommen würden, wie es der Gast vorsieht, würde er mehr Kontrolle über den Rechner bekommen, als für den stabilen Betrieb möglich ist. Ein Beispiel: Könnte der Gast Seitentabellen nach eigenen Vorstellungen setzen, könnte er an beliebige Stellen des Hauptspeichers schreiben. Weder der Monitor, noch andere Gäste oder der Wirt würden lange laufen, da ihre Daten und ihr Code unerwartet verändert würde.

Die Gast-Systemstrukturen können also nur modifiziert verwendet werden. Dabei gibt es zwei Möglichkeiten:

- Gast-Systemstrukturen vor Benutzung verändern, aber an vorgesehener Stelle belassen
- Gast-Systemstrukturen unverändert lassen, aber zweiten Satz von Systemstrukturen pflegen, der wirklich im System aktiv ist

Während letzteres mehr Speicher verbraucht, kann jedoch der Lesezugriff durch den Gast weitaus einfacher gehandhabt werden, außerdem ist mit ersterem z.B. die Emulation von Superpages nicht möglich und die Verwendung von bedarfsweiser Übersetzung kann nur mit einem zweiten Satz Strukturen realisiert werden. Somit fiel die Entscheidung für einen zweiten Satz von Systemstrukturen[5]. Die verwendeten Strukturen werden fortan als "Gasttabellen" und "aktive Tabellen" bezeichnet. Dabei ist "Tabelle" ein Synonym für "Struktur".

Sobald von einem Datum zwei Instanzen existieren, stellt sich das Problem der Konsistenz. Es sind folgende Fälle zu betrachten:

- Der Gast setzt eine neue Tabelle.

[5] Zwar wurde auch mit der Möglichkeit experimentiert, Gast-Systemstrukturen zu verändern und zu benutzen. Für eine Verwendung müssen aber weitere Annahmen und/oder Änderungen in der VMM-Architektur getroffen werden. Lesen kann z.B. durch Rückübersetzen und Superpages durch Speicherpartitionierung ermöglicht werden.

- Gast liest aus einer Tabelle.
- Der Gast schreibt in eine Tabelle.

Betrachtet wird zunächst das Setzen einer neuen Tabelle. Dies kann in mehreren Formen geschehen:

- Explizites Laden per Befehl (z.B. "mov CR3, reg32")
- Implizites Laden, z.B. durch Taskumschalten; Seitentabellen und LDT sind tasklokal

Um dem Monitor die volle Kontrolle über virtuelle Systemstrukturen zu gewährleisten, kann es nötig sein, dem Gast Lese- und Schreibzugriffe darauf zu verbieten. Dafür stehen mehrere Möglichkeiten zur Verfügung:

- Das P-Bit der Segmente
- Das P-Bit der Seitentabelleneinträge
- Das U/S-Bit der Seitentabelleneinträge

Die Segmentierungseinheit bietet nur unzuverlässigen Schutz, denn durch Alias-Segmente und die Übersetzung linearer in physische Adressen durch das Paging kann trotzdem leicht auf die zu schützenden physischen Adressen zugegriffen werden. Die erste Möglichkeit kann somit nicht angewendet werden. Das U/S-Bit hat gegenüber dem P-Bit den Vorteil, dass die Seite für den Monitor erreichbar bleibt, während sie für den Gast gesperrt ist. Bei Nutzung des P-Bits müsste die betreffende Seite erst in den Adressraum des Monitors eingeblendet werden (P-Bit setzen), nach dem Zugriff muss sie wieder entfernt werden. Daher fiel die Entscheidung zugunsten des U/S-Bit.

Zu den Eigenschaften der IA32-Prozessoren gehört, dass Einträge aus den Systemstrukturen in prozessorinternen Caches zwischengespeichert werden. Ändert sich ein betroffener Eintrag im Speicher, muss der Cache explizit neu geladen oder für ungültig erklärt werden — er ist also nicht transparent. Aus diesem Grund wäre es günstig, die Übersetzung der Einträge aus den virtuellen in die aktiven Tabellen bedarfsweise vorzunehmen, die aktiven Tabellen würden sich also in gewisser Weise wie die prozessorinternen Caches verhalten. Leider funktioniert dies nur bei den Seitentabellen, nicht bei Deskriptortabellen. Um sicherzugehen, dass ein veränderter Eintrag in den Systemstrukturen in die Caches übernommen wird, muss bei den Seitentabellen der betroffene Eintrag im TLB für ungültig erklärt und anschließend neu etabliert werden. Der entsprechende Befehl — invlpg — ist privilegiert und daher vom Monitor leicht abzufangen, die aktiven Seitentabellen können einfach angepasst werden. Das Laden der Segmentdeskriptorcaches hingegen kann durch Befehle vorgenommen werden, die auch im Nutzermodus erlaubt sind. Hier würde der Monitor von der Veränderung nicht in Kenntniss gesetzt werden, es muss also ein anderer Mechanismus gefunden werden. Im Abschnitt 3.3.3 wird darauf eingegangen.

Deskriptortabellen

Globale Deskriptortabelle Der Schutzmechanismus der Privilegstufen wird durch die Segmentierungseinheit implementiert. Deren korrekte Virtualisierung ist für die Systemsicherheit wichtig. Es muss darauf geachtet werden, dass der Gastcode immer in Ring 3 läuft. Alle anderen Felder können aus der virtuellen in die aktive Tabelle übernommen werden.

Tritt durch das Setzen der GDT oder durch ein Ändern ihrer Einträge eine Inkonsistenz zwischen dem Inhalt des verborgenen Selektorcaches und dem Eintrag in der GDT auf, muss diese vom Monitor beachtet werden. Solange die Selektoren in den Segmentregistern nicht explizit neu geladen werden, muss der alte Deskriptor verwendet werden. Dies ist in bestimmten Fällen nicht gewährleistet: Wird nach dem Laden einer neuen GDT der Gast vom aktiven in den Bereit-Zustand versetzt und ein anderer Gast oder der Wirt wird aktiv, so werden dessen Selektoren geladen. Wenn nun der unterbrochene Gast wieder aktiv wird, werden seine Segmentregister geladen und es werden bereits die neuen Deskriptoren verwendet anstatt der alten, wie es der Gast annimmt. Dies kann zu Fehlfunktionen führen.

Drei Lösungen bieten sich an:

- Die alten Einträge werden an ungenutzten Stellen der GDT gehalten, solange sie verwendet werden. Die Werte in den Segmentregistern müssen entsprechend angepasst werden, was nach sich zieht, dass das Lesen der Segmentregister virtualisiert werden muss, solange die Inkonsistenz besteht.

- Die alte GDT wird weiterhin verwendet, die neuen Einträge werden erst übernommen, wenn ein Laden der Segmentregister geschieht. Dies ist aber nur per Prescan-Technik feststellbar.

- Der Segmentdeskriptorcache wird in Software nachgebildet. Dafür kann die alte GDT verwendet werden, bzw. die Teile davon, welche in Benutzung waren. Wird aus einem Grund ein Neuladen der Segmentregister nötig, geschieht dies in den Schritten: alte GDT setzen, Selektoren laden, neue aktive GDT setzen. Wurden schon Deskriptoren aus der neuen Tabelle verwendet, müssen diese auf die alte GDT — den Segmentdeskriptorcache — übertragen werden. Um festzustellen, welche Deskriptoren verwendet werden, müssen sie verändert werden, so dass das Laden eines Selektors, der auf einen betroffenen Deskriptor zeigt, zu einem Fehler führt. Eine mögliche Variante ist, aus Datensegmenten vorerst nur ausführbare Codesegmente zu machen, bei einer anderen wird der DPL auf 0 gesetzt.

Variante Drei funktioniert mit allen Segmentregistern außer cs, da dieses beim iret-Befehl neu geladen wird. Sie hat aber gegenüber den anderen den Vorteil, dass im Falle einer Inkonsistenz nur das Lesen aus cs mit einer Prescan-Technik aufgefunden werden muss, der Rest ist relativ einfach zu virtualisieren. Der Spezialfall cs muss also nach Variante Eins behandelt werden.

Da vier Privilegstufen auf nur eine abgebildet werden, müssen die damit verbundenen und dadurch weggefallenen Schutzmechanismen in der Software (dem Monitor) geprüft werden.

Tore Generell gilt für Tore, dass sie sich auf der gleichen oder einer niedrigeren Privilegstufe wie das Programm befinden müssen, das es benutzen möchte. Das bedeutet, dass alle Tore in den Ring 3 verschoben werden müssen, damit sie benutzbar bleiben, bzw. in einen Ring kleiner 3, wenn der Monitor an dieser Stelle die Kontrolle übernehmen soll.

Task Status Segmente Die im TSS gespeicherten Werte müssen eventuell an die Gegebenheiten der virtuellen Maschine angepasst werden, wenn das TSS frisch erzeugt wird bzw. der Gast dessen Werte ändert. Bei einem Lesezugriff müssen die virtuellen Werte geliefert werden. Der DPL wird auf 0 gesetzt.

Deskriptoren für Lokale Deskriptortabellen Hier ist nur der DPL anzupassen.

Lokale Deskriptortabelle Die bei der GDT getroffenen Feststellungen gelten analog bei der LDT. Erleichternd ist die Tatsache, dass in der LDT weniger Arten von Deskriptoren erlaubt sind als in der GDT.

Interruptdeskriptortabelle In der aktiven IDT finden sich Einträge für:

- Ausnahmebehandlung; deren Auswertung geschieht teilweise durch den Monitor
- Hardwareinterrupts; eine Behandlung wird gemäß dem Modell zur Virtualisierung der Geräte durchgeführt
- Softwareinterrupts; sie werdem dem aktiven Gast zugestellt.

Um dem Gast die Interrupts zuzustellen, wird dessen virtuelle IDT verwendet.

Seitentabellen

Der Monitor muss sicherstellen, dass nur die von ihm vorgesehenen Seitenrahmen in die aktiven Seitentabellen eingetragen werden. Die grundlegende Idee ist, für diesen Zweck eine dritte Stufe der Adressübersetzung einzuführen, wie es in Bild 3.4 dargestellt wird. Der virtuelle Adressraum des Gastes wird wie üblich mit Hilfe der Segmente in einen linearen Adressraum überführt. Das Paging übersetzt die linearen in physische Adressen. Da diese jedoch virtualisiert wurden, sind es nur aus Sicht des Gastes physische Adressen — sie bilden einen "virtuellen physischen Adressraum", der vom Monitor gebildet wird. Im Bild ist er grau hinterlegt. Um die Adressen aus dem virtuellen physischen Adressraum in den realen physischen Adressraum zu übersetzen, muss der Monitor eine dritte Stufe der Adressübersetzung bereitstellen, denn die Hardware unterstützt nur eine zweistufige Adressübersetzung.

Für die dritte Stufe existiert eine Tabelle, in der jeder Seitenrahmenadresse des virtuellen Gasthauptspeichers eine reale physische Seitenrahmenadresse zugeordnet ist. Durch Ordnen der Tabelle und durchdachte Algorithmen sollte sich der Zugriff sehr performant realisieren lassen, wobei jedoch vorerst nur ein einfacher Tabellenzugriff realisiert wird.

Bild 3.5 zeigt, wie beim Kopieren die Adressübersetzung stattfindet:

- Eine Adresse besteht aus einer Seitenrahmennummer und einem Offset in der angegebenen Seite.
- Der Offset interessiert nicht weiter, er wird einfach übernommen.
- Die Seitenrahmennummer wird als Offset in der genannten Tabelle interpretiert, zu einer Rahmennummer wird so leicht die korrespondierende Rahmennummer gefunden.
- Die neue Adresse setzt sich also aus der übersetzten Rahmennummer und dem alten Offset zusammen.

Durch dieses Schema wird sichergestellt, dass nur für den Gast vorgesehene Speicherseiten an ihn vergeben werden, außerdem arbeitet die Zuordnung verschiedener virtueller Adressen zu derselben physischen Adresse[6] genau wie vom Gast vorgesehen.

Zu beachten ist, dass die letzte Stufe nur ausgeführt wird, wenn das P-Bit gesetzt ist, denn nur dann wird ein Seitenrahmen des Hauptspeichers adressiert. Ansonsten ist die Seite nicht im Hauptspeicher präsent, mit hoher Wahrscheinlichkeit also auf den Hintergrundspeicher ausgelagert.

[6] u.a. "Shared memory"

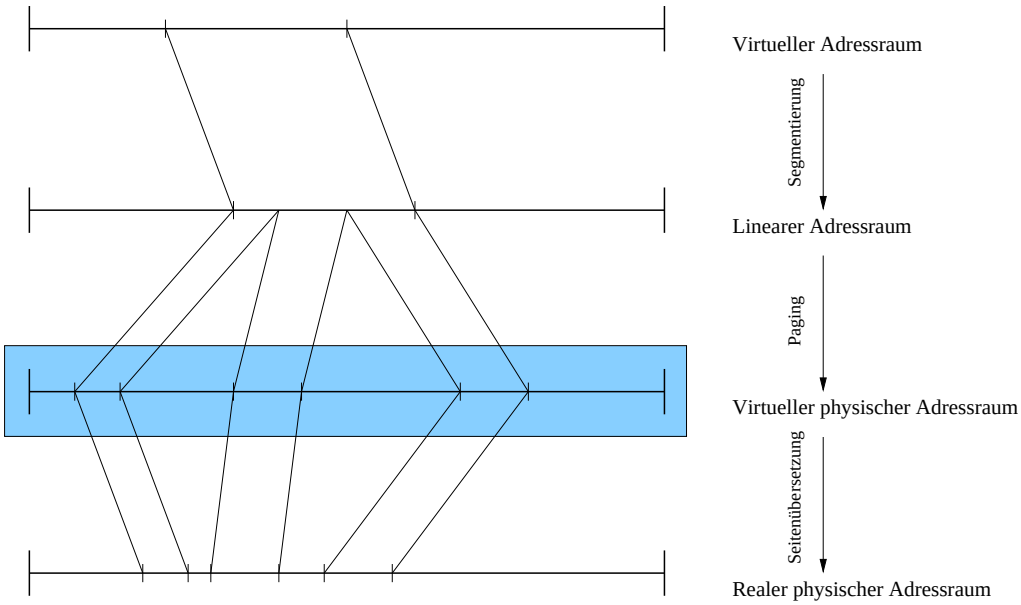

Abbildung 3.4: Einführung eines virtuellen physischen Adressraums

Ein weiterer Fakt, auf den man sein Augenmerk richten sollte, sind Superpages. Da ein Wirt nur selten entsprechende zusammenhängende Bereiche zur Verfügung stellen kann, muss eine große Seite aus 1024 kleinen Seiten zusammengesetzt werden. Im aktiven PD steht somit kein Verweis auf einen 4-MB-Seitenrahmen, sondern auf eine PT, in der wiederum sich die Verweise auf die kleinen Seitenrahmen befinden. Entsprechend werden auch die A- und D-Bits gehandhabt: sie ergeben sich aus einer bitweisen ODER-Operation aller einzelnen A- und D-Bits. Mit der Zerlegung einer Superpage in kleine Seiten geht jedoch ihr Vorteil verloren: die größere Geschwindigkeit, mit der lineare auf physische Adressen abgebildet werden. Die bessere Lösung hier ist, den CPUID-Befehl zu virtualisieren und das PSE als "nicht unterstützt" zu markieren, wenn sie vom Gast nicht ausdrücklich benötigt werden (z.B. zur Fehlersuche in einem Betriebssystemkern).

Wird nicht mit Speicherpartitionierung gearbeitet, sondern Wert auf große Flexibilität gelegt, ist eine dynamische Seitenverwaltung mit einem Seitencache sinnvoll. Dabei werden Wirt und Gästen Speicherseiten während der Laufzeit entzogen und hinzugefügt. Zum Preis eines zusätzlichen Verwaltungsaufwandes kann freier Speicher immer dem Betriebssystem zugeführt werden, das ihn gerade benötigt. Als Faustregel wird verwendet: ist ein Betriebssystem gezwungen, Hauptspeicherinhalte auf den Hintergrundspeicher auszulagern, mangelt es ihm an Hauptspeicher. Je mehr ausgelagert wird, desto größer ist der Bedarf. Eine Faustregel ist es deshalb nur, weil die genaue Politik des Auslagerns vom betriebssystemspezifischen Swapper festgelegt wird.

3.3.4 Abbildung von virtuellen Geräten auf reale Geräte

Ohne virtuelle Geräte zur Ein- und Ausgabe für einen virtuellen Rechner bleibt dieser für den Anwender unbenutzbar. Zur Virtualisierung sind mehrere Architekturen möglich, die in diesem Abschnitt diskutiert werden.

Die Wahl der passenden Architektur hängt vom Zweck ab, den man verfolgt — im Gebrauch zur täglichen Arbeit beispielsweise kann ein Pseudogerät emuliert werden, kommt es hingegen auf eine möglichst genaue Virtualisierung an, soll dem

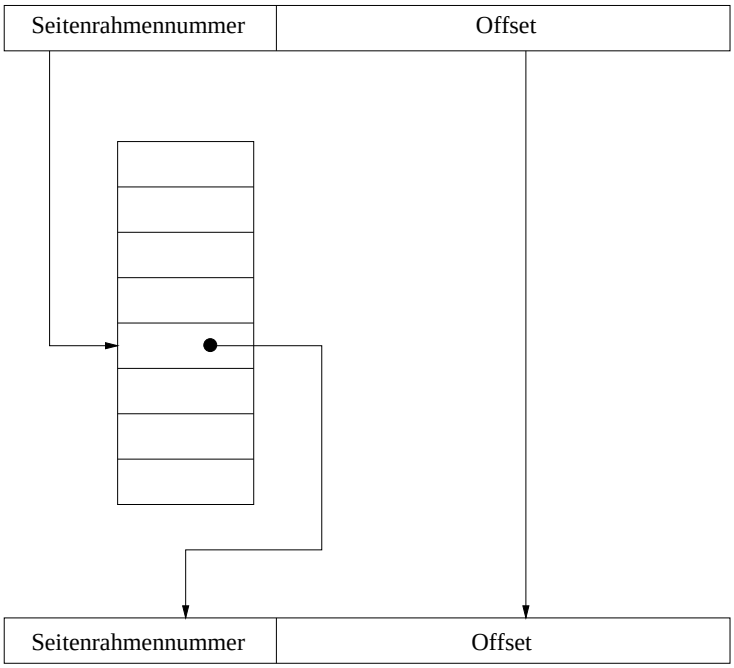

Abbildung 3.5: 3. Stufe der Adressübersetzung unter der Lupe

Gast das wirklich eingebaute Gerät virtuell zur Verfügung stehen. In Testsituationen kann andererseits aber auch verlangt werden, dass ein Gast direkten Zugriff auf das Gerät bekommt, was entsprechende Konsequenzen bezüglich der Sicherheit nach sich zieht. Die folgende Diskussion sollte daher mit dem Gedanken gelesen werden, dass nicht jede Architektur zu jeder Situation passt.

Ports

Ports sind spezielle Adressen im Adressraum des Prozessors. Alle diese Adressen zusammen werden als "I/O-Adressraum" bezeichnet. Auf die Ports wird mit besonderen Befehlen zugegriffen: in, out, ins und outs. Die Adresse des Ports ist im Befehlswort codiert oder steht in einem Register.

In einer Bitmap kann das Betriebssystem angeben, welche Ports für im Nutzermodus laufende Programme offen sind oder gesperrt werden sollen. Sperrt man alle Ports, führt deren Benutzung zu Allgemeinen Schutzverletzungen. Der Monitor kann so die Kontrolle übernehmen.

Eine zweite Möglichkeit, den Zugriff auf I/O-Adressen zu erlauben bzw. zu verbieten, eröffnen die EFlags. Hier kann die Privilegstufe angegeben werden, die mindestens erforderlich ist, um auf die Ports zuzugreifen.

Eingeblendeter Gerätespeicher

Einige Geräte, zum Beispiel Grafikkarten, blenden den in ihnen eingebauten Speicher in den Adressraum der CPU ein, damit diese Zugriff darauf bekommt. Hier bietet sich an, statt des Speichers im Gerät normalen Hauptspeicher zu benutzen und dann je nach Anwendungsfall von Zeit zu Zeit oder auf bestimmte Ereignisse hin ins Gerät zu kopieren. Ein Problem kann die Größe des eingeblendeten Speichers sein — er kann durchaus die Größe des eingebauten RAM erreichen. Ein Auslagern auf Hintergrundspeicher kommt aus Performancegründen nicht in Frage,

somit kann den Gästen nicht der volle Umfang des Speichers auf dem Gerät zur Verfügung gestellt werden.

Hardwareinterrupts

Tritt ein Interrupt auf, unterbricht die CPU die aktuell laufende Task, schaut in der Interrupt Descriptor Table (IDT) nach und springt zur angegebenen Adresse. Je nach Virtualisierungsmodell wird der Interrupt direkt dem Wirt, direkt dem Gast oder situationsabhängig zugestellt. Im ersten und letzten Fall ist es nötig, dass der Monitor dem Gast einen Interrupt zustellt, also eine Hardwareinterrupt emuliert. Dazu wertet er die vom Gast gesetzte virtuelle IDT aus und verfährt entsprechend. Die Interruptbehandlung schliesst auf jeden Fall mit einem iret-Befehl ab.

Direct Memory Access

Der Prozessor ist nicht der einzige Bestandteil eines Rechners, der direkt auf den Hauptspeicher zugreifen kann. Schon in den ersten PCs wurden Hilfsbausteine eingebaut, die Daten vom RAM zu den Geräten oder umgekehrt sowie zwischen verschiedenen Plätzen im Hauptspeicher selbständig transferieren konnten. Da diese Hilfsbausteine nicht weiterentwickelt wurden, war die CPU beim Transfer bald schneller und sie wurden immer seltener verwendet. Andererseits kamen aber Geräte auf, die sogenanntes "Busmastering" beherrschen. Auch dabei werden ohne den Prozessor in oder aus dem Hauptspeicher Daten transferiert. Gemein ist beiden Methoden, dass sie immer mit physischen Adressen arbeiten und keine Rücksicht auf die Speicherschutzmechanismen des Protected Mode nehmen. Einen Gast ungehindert DMA ausführen zu lassen würde also für den Monitor bedeuten, die Kontrolle zu verlieren, da der Gast nunmehr auf jeden beliebigen Speicherbereich zugreifen kann. Die Adressen des Transfers müssen also den Gegebenheiten angepasst werden. Verschiedene Situationen sind dabei denkbar:

- Der Bereich liegt innerhalb einer Seite. Die 3. Stufe der Adressumsetzung reicht aus.
- Die Übertragung überschreitet die Seitengrenze. Für den Gast physisch kontinuierlicher Speicher muss nicht wirklich physisch kontinuierlich sein. Eine Möglichkeit wäre, einen Ausweichplatz zu finden, an dem die geforderte Menge Speicherseiten physisch kontinuierlich vorhanden ist und dann zu kopieren oder die betroffenen Speicherseiten woanders einzublenden.
- Es wird eine speicherpartitionierende Architektur verwendet und die Übertragung findet in der ersten Partition statt. Der DMA-Transfer kann so geschehen, wie es der Gast vorsieht.

Protokoll

Das Protokoll, mit dem ein Gerät kommuniziert, ist für jedes Gerät spezifisch. Daher gibt es sehr viele Protokolle, was den Aufwand bei einigen Modellen für virtuelle Geräte erheblich steigert.

Innerer Zustand

Viele Geräte besitzen einen inneren Zustand. Manchmal kann er abgefragt werden, aber nicht immer. Daher ist es sicher, ein Gerät als Zustandsmaschine zu betrachten und in Software nachzubilden, sofern es die Gerätevirtualisierungsarchitektur erfordert.

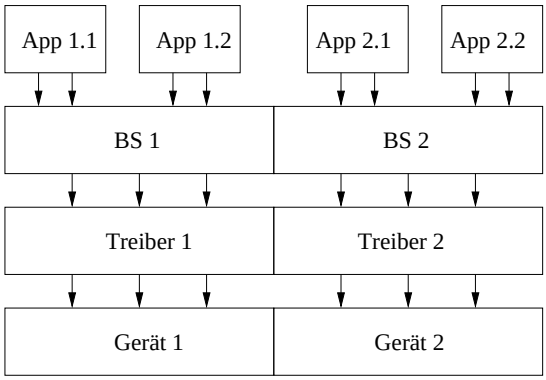

Abbildung 3.6: Partitionierung der Geräte

Es wird angenommen, dass es "sichere Zustände" gibt, in denen die Nutzung eines Gerätes von einem Treiber auf einen anderen übergehen kann. In den anderen, den "unsicheren Zuständen", darf nur ein einziger Treiber ein bestimmtes Gerät benutzen. Am Beispiel eines Modems ist "aufgelegt" ein sicherer Zustand. Ein beliebiger Treiber kann jetzt eine Nummer wählen und Daten senden bzw. empfangen. Während dieser Zeit darf kein anderer Treiber das Modem umprogrammieren, bis wieder "aufgelegt" wurde.

Mögliche Architekturen

Partitionierung der Geräte Die Geräte werden in mehrere Partitionen geteilt. Jedes Gerät wird von nur einem Treiber angesprochen. Es ist relativ einfach zu implementieren, wenn man einen Mechanismus gefunden hat, mit dem Gäste Geräte direkt steuern können. Der Nachteil ist allerdings, dass ein Gerät höchstens einem Betriebssystem zur Verfügung steht — besonders kritisch bei nur einmal vorhandenen Geräten wie Tastatur. Außerdem muß darauf vertraut werden, daß der Gast das Gerät korrekt steuert, denn der Monitor hat hier keine Möglichkeit der Kontrolle.

- Ports: partitioniert
- Eingeblendeter Gerätespeicher: wie üblich den Geräten zugeordnet
- Interrupts: partitioniert; auf "shared interrupts" achten
- DMA: nicht bzw. nur eingeschränkt möglich
- Kommunikation: direkt mit den Treibern
- Innerer Zustand: wie bei einem Rechner mit nur einem Betriebssystem

Bewertung: Könnte theoretisch funktionieren, wird aber an praktischen Problemen (z.B. von beiden Betriebssystemen benötigte, aber nur einmal vorhandene Geräte) scheitern. Weil der Speicher nicht so organisiert ist, wie der Gast es vermutet, darf kein bzw. bei einem speicherpartitionierten Modell nur in der ersten Partition DMA angewendet werden.

Zwei Treiber pro Gerät Ein Gerät wird direkt von zwei Treibern bedient. Damit Gerät und Treiber nicht schnell in inkonsistente Zustände kommen, müssen die beiden Treiber miteinander kommunizieren und sich gegenseitig abstimmen. Kein Treiber ist dafür ausgelegt, alle Treiber müssten entsprechend geändert werden, was einen enormen Aufwand bedeutet.

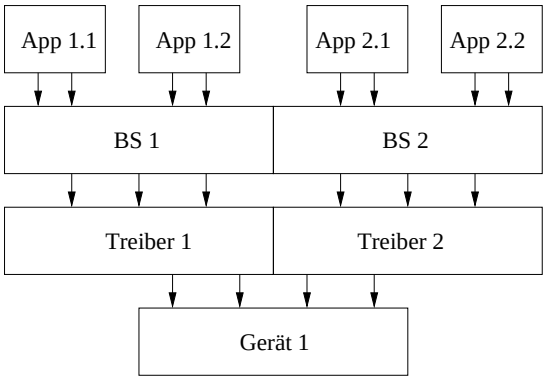

Abbildung 3.7: Zwei Treiber pro Gerät

- Ports: direkt von beiden Treibern genutzt
- Eingeblendeter Gerätespeicher: direkt von beiden Treibern genutzt
- Interrupts: vom Protokoll und inneren Zustand abhänig; beide Treiber erwarten Interrupts, aber nur einer wird ihn bekommen
- DMA: direkt von beiden Treibern genutzt
- Kommunikation: direkt mit den Treibern
- Innerer Zustand: wird ohne Kommunikation der Treiber inkonsistent

Bewertung: Wird nicht funktionieren, wenn die beiden Treiber nicht miteinander kommunizieren, um sich abzustimmen.

Ein Multiplexer für mehrere Treiber Im Gegensatz zum eben erläuterten Verfahren koordinieren sich die Treiber nicht selbst, sondern ein Multiplexer (MUX) übernimmt diese Aufgabe. Dafür muss der MUX die Geräte mit ihren Zuständen, der Kommunikation usw. kennen. Außerdem muss sichergestellt sein, dass die Treiber keine Chance haben, direkt auf ein Gerät zuzugreifen. Der Multiplexer kann die Zustände der Geräte in Software als Zustandsmaschinen nachbilden. Damit ist er in der Lage, das Gerät bei Bedarf in den Zustand zu versetzen, den ein bestimmter Treiber annimmt.

Für Geräte, auf denen nur ausgegeben werden kann, ist ein Spooling-Mechanismus denkbar, der die Geräte virtuell immer verfügbar macht, die Jobs dann aber nacheinander erledigt.

- Ports: von MUX virtualisiert
- Eingeblendeter Gerätespeicher: von MUX virtualisiert
- Interrupts: von MUX virtualisiert
- DMA: von MUX virtualisiert
- Kommunikation: von MUX virtualisiert
- Innerer Zustand: von MUX virtualisiert

Bewertung: Diese Anordnung ist ideal geeignet für die Architektur ohne Wirt mit gleichberechtigten Gästen. Allerdings muss MUX für jedes der vielen möglichen Geräte implementiert werden, was einen sehr hohen Aufwand bedeutet.

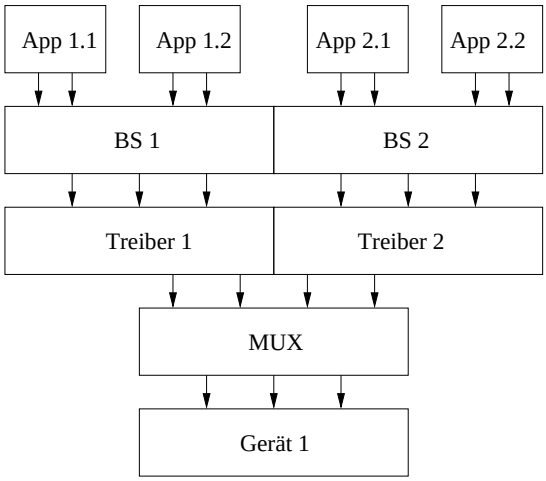

Abbildung 3.8: Ein Multiplexer für mehrere Treiber

Ein Pseudogerät emulieren und auf die Schnittstelle eines Treibers übersetzen Hier läuft ein Betriebssystem wie gewöhnlich. Für ein zweites (drittes, viertes,...) wird ein Übersetzer zwischen die beiden Geräte platziert, der zwischen den Schnittstellen der "Unterseite" des Gasttreibers und der "Oberseite" des Wirttreibers übersetzt. Wie die Terminologie schon andeutet, setzt dies eine Wirt-Gast-Architektur voraus.

- Ports: nur von Treiber 1 benutzt
- Eingeblendeter Gerätespeicher: nur von Treiber 1 benutzt
- Interrupts: nur von Treiber 1 benutzt
- DMA: nur von Treiber 1 benutzt
- Kommunikation: nur von Treiber 1 benutzt
- Innerer Zustand: nur von Treiber 1 benutzt

Bewertung: Ideal für eine Architektur mit Wirt und Gästen. Jedoch muss für jedes Gerät ein Übersetzer für die Schnittstellen der Treiber 2 und 1 geschrieben werden, was einen hohen Aufwand bedeutet. Man kann sich aber auf die Virtualisierung eines Pseudo-Gerätes beschränken, um den Aufwand zu senken.

Zwei Betriebssysteme nutzen denselben Treiber Wie im Bild 3.10 dargestellt, gibt es damit nur noch einen Treiber pro Gerät. Beide Betriebssysteme nutzen diesen Treiber.

- Ports: nur vom einzigen Treiber benutzt
- Eingeblendeter Gerätespeicher: nur vom einzigen Treiber benutzt
- Interrupts: nur vom einzigen Treiber benutzt
- DMA: nur vom einzigen Treiber benutzt
- Kommunikation: nur vom einzigen Treiber benutzt
- Innerer Zustand: nur vom einzigen Treiber benutzt

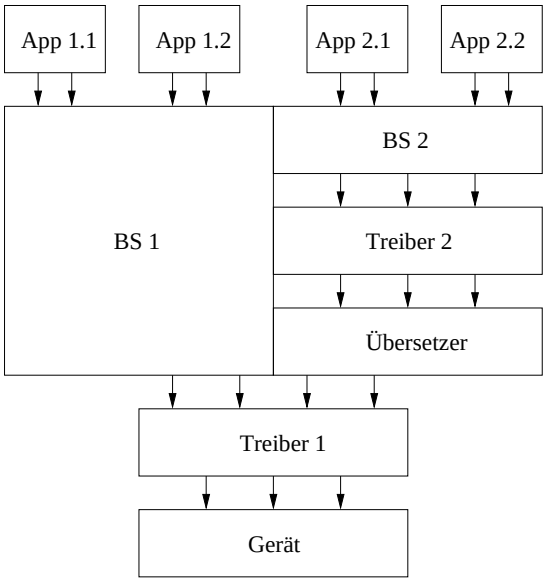

Abbildung 3.9: Emulation eines Pseudogerätes

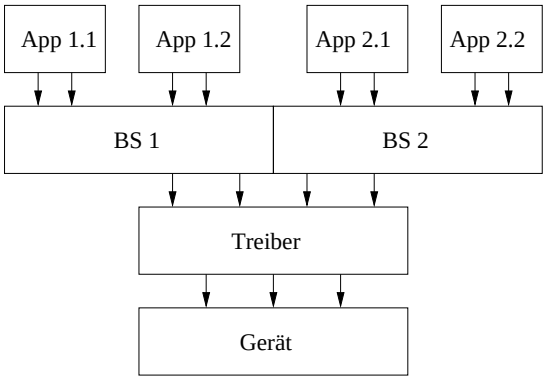

Abbildung 3.10: Zwei Betriebssysteme nutzen denselben Treiber

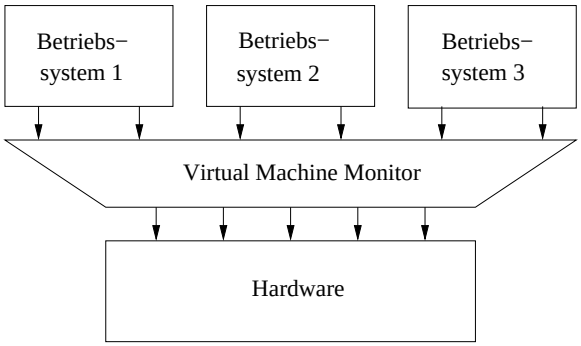

Abbildung 3.11: VMM ohne Wirt mit mehreren Gästen

Bewertung: Dieser Aufbau würde nur dann funktionieren, wenn es gelingt, den Treiber in beiden Betriebssystemen sichtbar zu machen. Jedoch sind die praktischen Hindernisse zu hoch: Beide Betriebssysteme müssen annehmen, dass sie genau mit diesem Treiber arbeiten (zwei gleichartige Betriebssysteme), außerdem kann man meist nur schwer abgrenzen, welcher Code und welche Daten zu diesem Treiber gehören. Nicht zuletzt ist auch hier auf das Konsistenzproblem bei konkurrierendem Zugriff auf ein Gerät zu achten, der Treiber muss also auch die Aufgabe eines Multiplexers übernehmen. Um diese Architektur verwenden zu können, wird man um einen Pseudotreiber für ein Betriebssystem nicht vermeiden können, der Anfragen auf die Schnittstelle des realen Treibers übersetzt — quasi dessen API emuliert.

Eine Entscheidung für eine bestimmte Architektur kann an dieser Stelle nicht getroffen werden, da die Gesamtarchitekturen dafür maßgebend ist, welche Variante die Anforderungen am besten erfüllt.

3.4 Gesamtarchitektur

Nachdem nun die Techniken zur Virtualisierung erarbeitet wurden, können aus den Teillösungen Gesamtlösungen zusammengesetzt werden. Als Architekturen kommen dabei die in Abschnitt 2.3 eingeführten Modelle Wirt-Gast und gleichberechtigte Gäste in Frage.

Unabhängig vom gewählten Modell steht fest, dass der Monitor im Kernmodus laufen muss, da er vollen Zugriff auf den Rechner benötigt, um Befehle abfangen und geeignet emulieren, Geräte steuern und Systemstrukturen entsprechend präparieren zu können.

3.4.1 Gleichberechtigte Gäste

Der Monitor verwaltet hier alle Systemressourcen. Nur er läuft im Kernmodus, die Gäste befinden sich komplett im Nutzermodus und sind einander gleichberechtigt, wie in Abbildung 3.11 dargestellt. Hier sind zwei Modelle denkbar:

- Statisches System: Beim Booten wird festgelegt, welche Betriebssysteme laufen sollen; es können nur welche beendet, aber keine neuen hinzugenommen werden; es ist noch ein Ersetzen des einen durch ein anderes denkbar.
- Dynamisches System: Während der Laufzeit können beliebige Gäste neu hinzukommen; die Konfiguration ändert sich dynamisch im Betrieb, es gibt Mechanismen zum Starten und Beenden von Gästen.

Die Geräte werden hier über einen Multiplexer angesprochen.

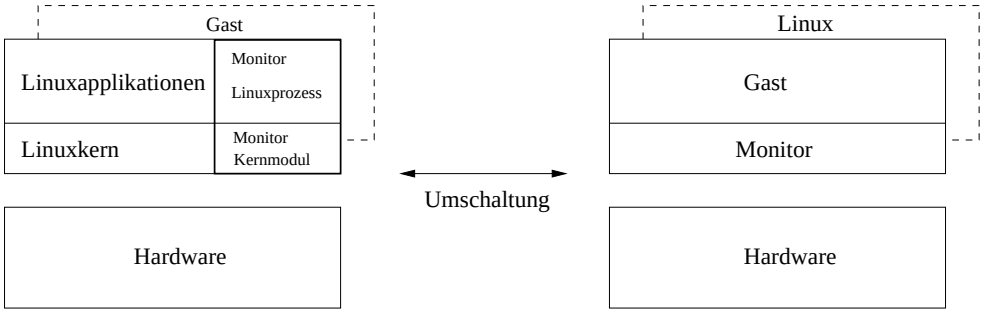

Abbildung 3.12: VMM mit Wirt und Gast

3.4.2 Wirt und Gast

Ein Betriebssystem, Wirt genannt, läuft wie gewohnt auf einem Rechner. Der Monitor läuft als Anwendungstask unter diesem Betriebssystem. Bekommt er Rechenzeit, sichert er den Kontext des Wirts und ersetzt diesen, so dass statt des Wirtes nun er im Kernmodus läuft. Die Gäste befinden sich auch hier im Nutzermodus. Ist die Zeitscheibe zu Ende, wird wieder ein Kontextwechsel hin zum Wirt ausgeführt. Abbildung 3.12 illustriert den Sachverhalt.

Der Wirt verwaltet CPU, Hauptspeicher und sämtliche Geräte. Der Monitor muss Ressourcen, die er dem Gast zur Verfügung stellen will, beim Wirt reservieren. Das heisst:

- Der Monitor stellt einen Teil seiner Zeitscheibe, die ihm vom Wirt zugeteilt wird, dem Gast zur Verfügung.
- Der Monitor muss Speicher, der den Hauptspeicher des Gastes bilden soll, beim Wirt allokieren. Gleiches gilt für Speicher, den der Monitor — etwa für Verwaltungsinformationen — für sich selbst benötigt.
- Der Gast kann Geräte nicht direkt ansprechen, es werden Treiber des Wirtes verwendet.
- Weitere Betriebssysteme werden wie neue Anwendungstasks gestartet und präsentieren sich ebenso dem Nutzer.
- Der Wirt muss eine Umschaltung zum Monitor unterstützen.

Für eine solche Architektur werden diese Komponenten benötigt:

- Eine Task, die im Wirt den VMM repräsentiert
- Eine Erweiterung des Kerns des Wirts, um Routinen ausführen zu können, die im Nutzermodus nicht möglich sind; in der vorliegenden Implementation für Linux als Wirt ist das ein Kernmodul
- Der eigentlichen Monitor, der den Wirt ersetzt, wenn der Gast läuft, und im Kernmodus arbeitet
- Ein Gast, welcher auf einer Privilegstufe größer Null läuft

In den folgenden Unterabschnitten werden die Aufgaben der Komponenten erläutert.

Nutzertask

Eine "normale" Nutzertask ist der einfachste Weg, einen VMM in das Taskmodell des Wirtbetriebssystems einzugliedern und eine Schnittstelle zum Nutzer bereitzustellen. Außerdem übernimmt sie die Aufgabe des Übersetzers vom Gerätemodell, welches in Abbildung 3.9 dargestellt ist. Dabei helfen dynamisch nachladbare Bibliotheken, eine hohe Flexibilität und eine übersichtliche, klare Gliederung sicherzustellen.

Das Kernmodul

Das Kernmodul hat mehrere Aufgaben:

- Speicherseiten beim Wirt allokieren und freigeben
- Datenstrukturen für Monitor aufbauen
- Umschaltung zum Monitor
- Kommunikation mit der Nutzertask

Der Monitorkern

Der Monitorkern übernimmt den Hauptanteil bei der Virtualisierung des PC. Mit Hilfe der in Abschnitt 3.3 vorgestellten Techniken stellt er folgende Funktionalität bereit:

- Emulation von privilegierten Befehlen
- Auffinden und Emulation von nichtprivilegierten Befehlen, die aber dennoch zu einer Disfunktion des Gastes führen können
- Situationsabhängiges Zustellen von Interrupts und Exceptions zu Gast oder Wirt
- Verwaltung der virtuellen Systemstrukturen des Gastes (z.B. Seitentabellen, Deskriportabellen)
- Umschaltung zum Wirt

Der Gast

Für die Unterstützung von Gastbetriebssystemen sind mehrere Meilensteine vorgesehen, um eine schrittweise Annäherung an komplexe Betriebssystemkerne zu erreichen:

1. Protected Mode Testkerne nach dem Multiboot-Standard
2. Fiasco
3. L4
4. Linux

Der Real Mode und V86 Mode folgen — wenn überhaupt — später.

3.5 VMM als Debug-Werkzeug

Soll ein Monitor für virtuelle Maschinen als Debug-Werkzeug eingesetzt werden, muss er um einige Funktionen erweitert werden:

- Suspend/Resume, Breakpointhandling: Anhalten/Fortsetzen des Gastes; Setzen und Entfernen von Breakpoints, an denen der Gast angehalten wird und die Kontrolle an den Debugger übergeht
- Copy in/out n Byte: Eine Anzahl n Byte schreiben/lesen; anders ausgedrückt: den Inhalt des Hauptspeichers des Gastes manipulieren
- Lies/schreib virtuelle/aktive Systemstrukturen
- Lies/schreib Zustand von Geräten

Hierbei wird davon ausgegangen, dass ein unter dem Wirt laufendes Werkzeug bereitsteht und der Monitor nur die nötigsten Funktionen liefert, um das zu untersuchende Betriebssystem (Gast) kontrollieren und steuern zu können. Hier bietet sich beispielsweise der GDB an.

3.6 Ansätze, die nicht funktionieren

Im Laufe der Entwicklung kamen viele Ideen auf, die sich als nicht durchführbar erwiesen. Einige seien hier aufgezählt.

Die erste Idee war, zwei Betriebssysteme wie gewohnt mit ihren Teilen in Kern- und Nutzermodus laufen zu lassen, jedem System einen Treiber einzubauen, der es suspendiert, damit das andere zum Zuge kommt. Unüberwindliche Probleme ergeben sich hier aber, wenn die Speicherverwaltung und die Gerätesteuerung nicht angepasst werden, denn bisher gibt es in keinem Betriebssystem Mechanismen, die eine Kooperation in diesem Sinne erlauben. Neben dem Nachteil des extrem hohen Aufwandes der Anpassung spricht gegen diesen Ansatz, dass in Entwicklung befindliche Betriebssysteme praktisch nicht kontrolliert werden können und Fehlfunktionen dieselben verheerenden Auswirkungen haben wie ohne Monitor. Ein solcher Monitor würde also nicht als Debug-Werkzeug eingesetzt werden können. Der Vorteil wäre allein die sehr hohe Performance.

Beim Versuch, den Adressraum des Gastes mit dem des Wirts zu vereinbaren, wurde das Augenmerk zunächst auf die Segmentierungseinheit gerichtet. Vom 4 GB grossen virtuellen Adressraum reserviert sich ein Betriebssystem ein mehr oder minder großes Stück. Bei Fiasco ist dies der Bereich zwischen 3 und 4 GB. Tritt in dieser Spanne ein Seitenfehler auf, wird der Pager davon nicht in Kenntnis gesetzt, sondern er wird vom Kern als Fehler behandelt. Da auch andere Systeme diesen Bereich gern für sich reservieren, treten unweigerlich Konflikte auf. Gelingt es, diesen reservierten Bereich an eine andere Stelle des virtuellen Adressraums zu verschieben, kann der Pager wie gehabt zum Einsatz kommen. Die Verschiebung sollte durch Segmente geschehen, deren Basis nicht bei 0, sondern einer höheren Adresse liegt. Da am Ende des Segments automatisch ein Umbruch stattfindet, würde der virtuelle Adressraum an die passende Stelle rotiert. Leider funktioniert dies nicht, da es keine festen ungenutzten Bereiche im virtuellen Adressraum gibt, außerdem kann ein Betriebssystem potenziell alle gültigen Adressen generieren. Spätestens hier fiel die Entscheidung, dass für einen funktionierenden Monitor der Wirt komplett stillgelegt werden muss und der Monitor seinen eigenen neuen Kontext erhält.

3.7 Zusammenfassung der Entwurfsentscheidungen

Die erste Implementierung soll vor allem dazu dienen, Erfahrungen zu sammeln und Messungen durchzuführen. Daher wird vorerst auf eine lauffähige, wenn auch einfache, unvollständige und langsame Lösung Wert gelegt. Mit dem gesammelten Wissen kann dann ein Monitor implementiert werden, der die Entwurfsziele erfüllt.

Von den 3 Betriebsmodi der CPU wird nur der Protected Mode unterstützt. Der Gast läuft in Ring 3. Alle privilegierten Befehle erzeugen einen Trap und werden im Monitor, der im Kernmodus läuft, emuliert. Eine Emulation nichttrappender Befehle wird vorerst nicht vorgenommen, da vermutet wird, diese in der ersten Phase nicht zu benötigen. Falls sie dennoch notwendig werden sollte, dann mit folgenden Eigenschaften:

- Ein Befehlsemulator folgt dem Ausführungspfad.
- Ob ein hardwaregesteuerter Einzelschrittmodus oder eine Emulation in Software schneller ist, muss durch Messungen ermittelt werden. Der Einzelschrittmodus ist einfacher zu implementieren, da kein Softwareemulator für die Befehle benötigt wird. Er wird daher verwendet.
- Das Erzwingen der Emulation wird durch Softwarebreakpoints erreicht.

Zur Virtualisierung des Hauptspeichers ist es hinreichend, die Systemstrukturen zu betrachten.

Es wird die in Abschnitt 3.4.2 vorgestellte Wirt-Gast-Architektur mit dem Modell zur Virtualisierung von Geräten verwendet, welches die wirtbasierte Emulation von Pseudogeräten vorsieht.

Neben den virtuellen Systemstrukturen werden aktive verwendet. Wenn möglich, werden Werte in die aktiven Strukturen bei Bedarf eingetragen.

Kapitel 4

Implementierung

Die gegenwärtige Implementierung nutzt Linux als Wirtbetriebssystem. plex86 soll aber nicht auf Linux beschränkt bleiben. Um die Portierung zu vereinfachen, wurde darauf geachtet, betriebssystemabhängige Teile zu separieren und möglichst klein zu halten.

Der folgenden Beschreibung liegt eine Version vom 20. April 2000 zugrunde. Damals umfasste plex86 ca. 15000 Zeilen C-Code und 3200 Zeilen Assemblercode. Hinzu kommen ca. 10700 Zeilen von Bochs übernommener C++ - Code.

Der Quellcode steht unter http://os.inf.tu-dresden.de/~jn4/freemware/plex86-jens.tar.gz zum Download zur Verfügung.

4.1 Architektur

Als Architektur wurde die in Abschnitt 3.4.2 beschriebenen Struktur mit Wirt und Gast realisiert. Bild 4.1 zeigt, wie die einzelnen Komponenten zusammenwirken.

Während der Initialisierung beauftragt der Linuxprozess das VMM – Kernmodul per "ioctl", eine neue VMM bereitzustellen (1). Der Prozess lädt den gewünschten Kern und schaltet dann zum Monitorkontext um (beauftragt das Kernmodul via ioctl) (2). Der Monitor aktiviert den Gast durch "iret" (3). Tritt der Fall ein, dass ein privilegierter Befehl emuliert werden muss, wird infolge der "Allgemeinen Schutzverletzung" der Monitor aktiviert (4), der die Emulation delegieren kann (5), (1). Nach erfolgreicher Emulation wird der Gast wiederum aktiv (1), (2), (3). usw.

4.2 Prozessor

Sowohl der reale als auch der virtuelle Prozessor befinden sich im Protected Mode. Die Schutzverletzungen, die der Gast auslöst, werden durch einen monitoreigenen Exceptionhandler behandelt und die Befehle emuliert. Nach abgeschlossener Emulation wird per iret zum Gast zurückgekehrt. Zur Zeit ist noch keine Form des Prescan eingebaut, nur die privilegierten Befehle werden emuliert, alle anderen direkt ausgeführt.

4.3 Geräte

Von den Geräten werden nur Tastatur und Konsole unterstützt. Während der Emulation der in/out-Befehle wird zur Nutzertask umgeschaltet. Dort haben sich Plugins für bestimmte Bereiche des I/O-Adressraumes registriert. Das jeweils zuständige

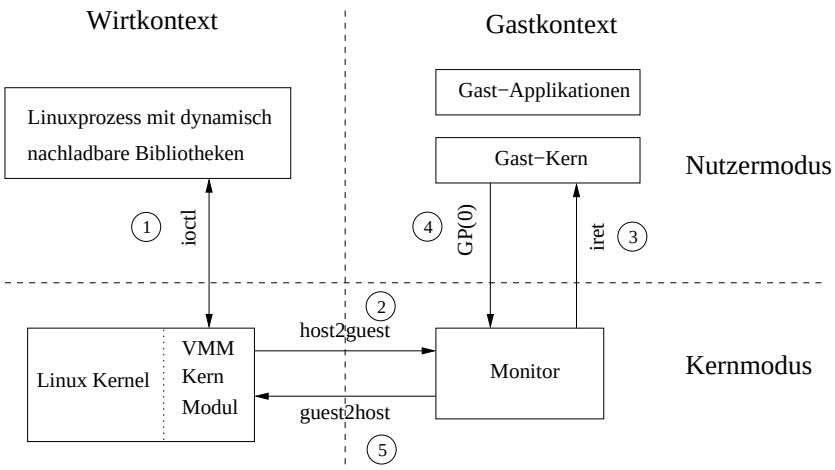

Abbildung 4.1: Zusammenwirken der Komponenten bei einer Wirt-Gast-Architektur

Plugin übernimmt die Bearbeitung. Alle Interrupts werden an den Wirt weitergeleitet. Dafür wird eine entsprechende IDT installiert.

4.4 Systemstrukturen

Es wurde das Modell implementiert, das virtuelle und aktive Strukturen vorsieht. In der GDT können Deskriptoren für Applikationssegmente eingetragen sein. Bei Änderungen werden die alten, noch benutzten Werte an freie Stellen kopiert und die Segmentselektoren angepasst.

Seitentabellen können gesetzt und benutzt werden, jedoch werden noch keine Superpages unterstützt. Seitenfehler werden entgegengenommen und aufgelöst. Der Code zur Virtualisierung der Seitentabellen ist der letzte, den ich zum plex86–Projekt beigetragen habe. Er ist unvollständig, da im Laufe der Implementierung festgestellt wurde, dass für ein akzeptables Speichermanagement größere Änderungen am Monitor nötig wären.

4.5 Wirt – Gast – Umschaltung

Die Umschaltung zwischen Wirt und Gast geschieht durch eine “Nexus” getaufte Kombination aus Code und Daten. Er befindet sich komplett in einer Speicherseite und ist so ausgelegt, dass er an jede beliebige Stelle im Adressraum eingeblendet werden kann.

Der Nexus wird sowohl im Wirts- als auch im Monitorkontext durch entsprechende Einträge in den Seitentabellen eingeblendet. Dann werden vom alten Kontext der komplette Zustand der CPU und die Zeiger auf die Systemstrukturen (GDT, LDT, IDT, TR) gesichert, gleichfalls die Steuerregister. Eben diese Werte werden hernach vom neuen Kontext eingetragen und die Steuerung wird dem Monitor übergeben. Dieser wird dann den Gast aktivieren.

4.6 Speicherverwaltung

Der Monitor reserviert sich via vmalloc und get_free_page Speicherseiten. Sie stehen dem Gast als Hauptspeicher bzw. dem Monitor für seine Datenstrukturen zur

Verfügung. Ausserdem wird beim Laden des Kernmoduls implizit Speicher reserviert, der den Code des Monitors enthält.

Ein Seitenpuffer beinhaltet freie Seiten, die zwischen Monitor und Wirt ausgetauscht werden. Sinkt der Füllstand unter eine bestimmte Marke, füllt der Wirt Seiten nach. Steigt er über einen bestimmten Wert, kann der Wirt Seiten entnehmen. Der Monitor entnimmt Seiten, wenn er diese benötigt und füllt den Puffer mit frei gewordenen Seiten. Auf diese Weise soll ein einfacher Austausch freier Speicherressourcen geschehen.

4.7 Testkerne

Da zu Beginn der Entwicklung noch keine allgemein verfügbaren Betriebssystemkerne laufen, wurden zum Testen einige minimale Kerne geschrieben. Damit wurde das Laden und Ausführen des Codes ("Virtcode"), Umschalten zwischen zwei kooperativen Tasks ("Cooperative"), interruptgesteuerte, preemptive Umschalten zwischen zwei Tasks ("Preemptive"), Setzen von Seitentabellen durch den Gast ("Paging") und Funktionieren von aus dem OSKit zusammengestellten, multibootfähigen Kernen, das Setzen der GDT durch den Gast und verschiedene Aspekte bezüglich gastgesteuerter Seitentabellen ("Solio") erprobt.

Der Gastkern kann als ELF- oder pure Binärdatei vorliegen. Er wird an die Stelle geladen, die später den virtuellen Hauptspeicher darstellt, danach wird er in einer Protected-Mode-Umgebung gestartet.

Versuchsweise wurde auch Fiasco und ein Linux-Kern der Version 2.2.x geladen. Stück für Stück wurde plex86 erweitert, um immer weitere Schritte im Bootvorgang von Fiasco zu unterstützen.

4.8 Einige Details

In diesem Abschnitt sollen einige Details erläutert werden, anhand derer auch mein Beitrag zum plex86-Projekt verdeutlicht wird. Neben Erweiterungen, die eher die Nutzertask betreffen — wie die Konfigurierbarkeit durch Kommandozeile und Datei, laden von Kernen im ELF-Format, Speichern des virtuellen RAM des Gastes in einer Datei oder die Multibootunterstützung für die Gastkerne —, habe ich mich hauptsächlich mit der Emulation von zu virtualisierenden Befehlen beschäftigt. Dafür habe ich zuerst einen Rahmen in Form einer Routine eingeführt, der dann nach und nach von anderen und mir ausgefüllt wurde. An den Beispielen "hlt" und "in" wird gezeigt, wie einzelne Befehle virtualisiert werden. Abschließend wird eine erste Version der virtuellen Segmente betrachtet.

4.8.1 Der Rahmen zur Emulation

Die Rahmenfunktion zur Emulation wird immer dann aufgerufen, wenn eine "Allgemeine Schutzverletzung" anzeigt, dass ein Befehl emuliert werden soll. Anhand des Operationscodes wird zu einer konkreten Funktion verzweigt, die die Instruktion virtualisiert. Sie kehrt mit einem Ergebniswert zurück, der anzeigt, wie weiter verfahren werden soll:

- Fehler: Die virtuelle Maschine wird kontrolliert geschlossen
- Befehl wiederholen: Der Gast wird an der Stelle fortgesetzt, an welcher die Schutzverletzung auftrat
- Befehl überspringen: Der Gast wird nach dem Befehl fortgesetzt, der die Schutzverletzung auslöste

- Monitor neu einblenden und Befehl überspringen: Der Monitor wird an einer neuen Stelle in den Adressraum des Gastes eingeblendet, weiter mit dem nächsten Befehl
- Emulation im Nutzerprozess: Die Emulation wird an eine Bibliothek im Nutzerprozess delegiert

Nach erfolgreicher Emulation wird zum Gast zurückgekehrt.

4.8.2 Zwei Beispiele: hlt und in

"hlt" ist ein privilegierter Befehl, die Ports wurden so konfiguriert, dass nur im Superusermodus auf sie zugegriffen werden kann — "in" verhält sich somit auch wie ein privilegierter Befehl. Beide lösen die Exception 13 (Allgemeine Schutzverletzung) aus, die vom Monitor behandelt wird.

Die originale Semantik von hlt ist, den Prozessor solange anzuhalten, bis ein Interrupt auftritt. In der Implementierung wurde sie allerdings so abgewandelt, dass der VMM kontrolliert geschlossen wird. Somit eignet sich hlt gut als Mittel zum Debuggen.

Die in 4.8.1 erwähnte konkrete Emulationsfunktion dazu sieht so aus:

```
int
emulate_hlt (vm_t * vm, guest_context_t * context)
{
    context->event_info = EMU_INSTR_HLT | (RET_BECAUSE_USEREMU << 8);
    return 4;
}
```

In einer Struktur, die Daten zum Gastkontext enthält, gibt es unter anderem ein Feld event_info. In diesem werden Hinweise zur Emulation auf Wirtseite gespeichert. Hier sind es EMU_INSTR_HLT und RET_BECAUSE_USEREMU. Das return 4 veranlasst die Umschaltung zum Wirt, in dessen Kontext wird das event_info-Feld ausgewertet. Laut originaler Semantik würde dem Gast ab diesem Zeitpunkt keine Rechenzeit mehr zugeteilt werden, da jedoch die abgewandelte Bedeutung angewendet wird, werden einige Debuginformationen ausgegeben und der VMM beendet.

In ähnlicher Weise wird mit dem "in"–Befehl verfahren:

```
int
emulate_in (vm_t * vm, guest_context_t * context, int operand_size, int port)
     /*
      * Emulate in instruction
      */
{
    switch (operand_size)
    {
    case 8:
        context->event_info = EMU_INSTR_IN_8;
        break;
    case 16:
        context->event_info = EMU_INSTR_IN_16;
        break;
    case 32:
        context->event_info = EMU_INSTR_IN_32;
        break;
    }
```

```
    context->event_info |= (RET_BECAUSE_USEREMU << 8) | (port << 16);
    return 4;
}
```

Nachdem das event_info-Feld mit den nötigen Informationen gefüllt wurde, wird in den Wirtkontext umgeschalten. Die Emulation wird zum Nutzerprozess delegiert, wo festgestellt wird, welche Bibliothek zuständig ist:

```
switch (instr)
{
case EMU_INSTR_IN_8:
    plugin_emulate (EVT_INPORT, arg, 1, 1, &value);
    context.eax = (context.eax & ~0xff) | (value & 0xff);
    break;

case EMU_INSTR_IN_16:
    plugin_emulate (EVT_INPORT, arg, 2, 1, &value);
    context.eax = (context.eax & ~0xffff) | (value & 0xffff);
    break;

...
}
```

Jede Bibliothek registriert sich für einen Bereich des I/O-Adressraumes. plugin_emulate legt mit Hilfe der I/O-Adresse und des Registers fest, welche Routine die Emulation übernimmt und ruft sie auf. In der Variablen value wird das Ergebnis zurückgeliefert, welches ins Feld eax der Gastdatenstruktur eingetragen wird. Am Namen ist leicht ersichtlich, dass dieses Feld das gesicherte eax-Register des Gastes ist. Bekommt der Gast das nächste Mal Rechenzeit, findet er also im eax den erwarteten Wert vor.

4.8.3 Virtuelle Segmente

Nachdem Basisadresse und Größe der virtuellen GDT gespeichert wurden, werden deren Einträge in die aktive GDT übernommen. Zur Zeit werden noch einige Deskriptoren ignoriert. Um Sprünge in andere Codesegmente abfangen zu können, werden diese vorerst in Datensegmente umgewandelt. Taskwechsel werden verboten, indem der DPL auf 0 gesetzt wird.

```
/* "install" new gdt; this is the virtual gdt */
vm->common.addr->nexus->guest_gdt_info.limit = limit;
vm->common.addr->nexus->guest_gdt_info.base = base;

if (limit >= MON_GDT_PAGES * PAGESIZE)
    return 0;
nr_descriptors = (limit + 1) / 8;

for (i = 1; i < nr_descriptors; i++)
{
    descriptor_t descr;

    read_guest_descriptor (vm, &descr, i << 3);

    if (descr.type & 0x10)
```

```
        {
            /* Code or data segment descriptor */

            /* For now, we let everything go through unchanged,
             * only the DPL is forced to 3 */
            descr.dpl = D_DPL3;

            /* Change code segments to data segments; this is to make all
             * inter-segment transfers, especially iret, fault. */
            descr.type &= ~0x08;
        }
        else
        {
            /* System segment descriptor */

            switch (descr.type)
            {
            case D_TSS:
            case D_TSS | 2:     /* available/busy TSS */
                /* Force the DPL to zero to disallow task switches */
                descr.dpl = D_DPL0;
                break;

                /* FIXME: Don't allow LDTs or gates for now ... */
            case D_LDT:
            case D_CALL:
            case D_INT:
            case D_TRAP:
            default:
                clear_descriptor (vm, &descr);
                break;
            }
        }

        monitor_gdt[i] = descr;
    }
```

Der Einfachheit halber wurde Variante Eins aus Abschnitt 3.3.3 zur Emulation des Segmentdeskriptorcaches benutzt: Die alten Werte werden in ungenutzten Teilen der GDT gespeichert. (Dieser Code ist nicht mit abgebildet.)

Nach einer Änderung der GDT ist ein erneutes Einblenden des Monitors in den Gastadressraum nötig:

```
/* We have just clobbered the monitor selectors.  Zero them out in the
 * nexus data structure to reflect this.  See unmap_monitor() ... */
vm->common.addr->nexus->mon_jmp_info.selector = 0;
vm->common.addr->nexus->mon_stack_info.selector = 0;
vm->common.addr->nexus->mon_tss_sel = 0;

/* Return to the host to re-map the monitor */
return 3;
```

Da eben Sprünge in Codesegmente verboten wurden, muss auch der ljmp–Befehl emuliert werden:

```
int
emulate_ljmp (vm_t * vm, guest_context_t * context)
     /* ljmp may be an intersegment call
      * we have disabled intersegment calls, so we have to emulate them
      */
{
    descriptor_t descr;
    Bit16u new_cs;
    Bit32u new_eip;
    Bit32u eip = context->eip + 1;

    new_eip = read_guest_dword (vm, &eip);
    new_cs = read_guest_word (vm, &eip);

    read_guest_descriptor (vm, &descr, new_cs);
    if (!(descr.type & D_CODE))
        return 0;

    codeseg_activate (vm, context->cs, 0);
    context->cs = new_cs | 3;
    context->eip = new_eip;
    codeseg_activate (vm, new_cs, 1);

    return 1;
}
```

4.9 Organisation des Quelltextes

Der Quelltext ist in drei Bereiche gegliedert: “user” beinhaltet die Module für den Linuxprozess, “kernel” die für das Betriebssystemkernmodul und “guest” diverse Testkerne. Weiterhin gibt es Konfigurationsdateien wie config.h und plex86.h.

Die in “user” enthaltenen Module sind:

- decode: IA32-Befehlsdecoder
- emulation: Emulation von Befehlen im Wirtbetriebssystemnutzerprozess
- plex86: Hauptprogramm
- plugin: Routinen zur Verwaltung nachladbarer Bibliotheken
- user: Kommunikation mit dem Kernteil des Monitors
- Verzeichnis plugins: Sammlung nachladbarer Bibliotheken

Der Bereich “kernel” enthält:

- emulation: Emulation von IA32-Befehlen
- fault: Behandlung von Exceptions
- fetchdecode: IA32-Befehlsdecoder
- host-beos: BeOS-spezifischer Betriebssystemkerncode
- host-linux: Linux-spezifischer Betriebssystemkerncode
- monitor: Betriebssystemunabhängiger Monitorcode

- nexus: Wirt-Gast-Umschaltung
- page-fault: Behandlung von Seitenfehlern
- prescan: Scan-Before-Execute-Technik
- trap: Behandlung von Traps

Als Gastkerne stehen folgende Testkerne zur Verfügung:

- virtcode: Inkrementiert eine Variable
- cooperative: Zwei kooperativ geschedulte Tasks inkrementieren eine Variable
- preemptive: Zwei preemptiv geschedulte Tasks inkrementieren eine Variable
- paging: Wie preemptive, zusätzlich aber mit eigener Seitentabelle

Nicht mit zum plex86-Projekt zugeordnet, aber auch zum Testen verwendet wurden ein weitgehend aus dem OSKit übernommener Beispielkern (solio) und eine leicht modifizierte Version von Fiasco.

Kapitel 5

Leistungsbewertung

5.1 Testumgebung

Für die Entwicklung und erste Tests wurde ein Rechner mit Pentium (90 MHz), 64 MB RAM, NE2000 Netzwerkkarte und Symbios Logic SCSI Controller werwendet. Als Compiler wurde gcc 2.7.2.3 verwendet, die Linuxkernelversion war 2.2.13.

5.2 Tests

Die Zeiterfassung der Tests wurde über den Time Stamp Counter (TSC) vorgenommen. Bei Prozessoren mit langer Pipeline ergibt sich dabei immer die Frage, wie lange die Ausführung eines Befehles wirklich dauert, was als Start-, was als Endpunkt gilt usw. Der TSC-Wert wird deshalb nur als gute Näherung betrachtet.

In einer Schleife wurde das Messobjekt 1000 Mal ausgeführt. Für jede Ausführung wurden die Takte gemessen. Am Ende der Messung wurde das Minimum und Maximum der Takte ausgegeben. Für jedes Messobjekt wurden mindestens 3 Messungen durchgeführt.

Dieser Testablauf wird damit begründet, dass erstens die Zeit feststellbar ist, die mindestens unter günstigsten Bedingungen (Daten und Code im Cache, TLB gefüllt) benötigt wird, zweitens abschätzbar ist, welche Verlangsamung durch Cache-Misses auftreten und drittens der Einfluss des Schedulings des Wirtes sichtbar wird.

Vermessen wurden zum einen die Umschaltungen zwischen den Kontexten Gast – Monitor, Wirt – Gast sowie Nutzertask – Kernmodus des Wirt und zum anderen die Ausführung bestimmter, ausgewählter Befehle. "null" ist die reine Messschleife, "nop" und "lgdt" die IA32-Befehle und "read cr3" das Auslesen den PDBR mittels "mov". Anwendungsnähere Tests sind printf("Hello world!") und der Algorithmus zur Berechnung von Fibonacci-Zahlen. Dabei ist fibo 2 wie Fibonacci, jedoch wurden die Ergebnisse nicht auf dem Bildschirm ausgegeben.

5.3 Interpretation der Messergebnisse

Zuerst muss gesagt werden, dass die Messwerte mit Vorsicht zu betrachten sind. Es sind erste Messungen, die keinerlei Rückschlüsse auf die Performance in real zu erwartenden Situationen zulassen. Es fehlt im Monitor noch Funktionalität, die eine Verlangsamung zur Folge haben wird, weiterhin sind noch keine Optimierungen erfolgt, um die Geschwindigkeit zu erhöhen.

Diese erste vorsichtige Schätzung dient vor allem um herauszufinden, wieviele Takte die verschiedenen Techniken konsumieren. Kennt man die Charakteristika

Messobjekt		1. min/max		2. min/max		3. min/max	
Gast-Monitor		336	1222	336	1305	336	4027106
Task-Modul		232	5399	232	4927	232	5609
null	plex86	12	30	12	62	12	71
	original	7	30	7	30	7	30
nop	plex86	13	15	13	15	13	15
	original	8	10	8	10	8	10
read cr3	plex86	415	4079644	415	1563	415	4890122
	original	11	12	11	12	11	12
lgdt	plex86	16608	328368	16607	363351	16725	158655
	original	17	39	17	39	17	39
printf	plex86	653567	5297227	681080	22914093	705844	4887383
	original	15984	16200	16002	16189	16002	16216
fibonacci	plex86	453416	5197176	472638	5098740	489870	4778061
	original	11331	11583	11363	11563	11376	11644
fibo 2	plex86	115	289	115	3824966	15	214
	original	110	170	110	170	110	170

Alle Messwerte in Anzahl Takte, die für die Aktion benötigt wurden

von Betriebssystemen, kann man abschätzen, an welchen Stellen sich Optimierungen lohnen. Eine Folgeaufgabe wäre, offene Betriebssysteme (Fiasco, Linux, FreeBSD) entsprechend zu untersuchen.

Wie man sieht, erreicht der Fibonacci-Algorithmus, der nur nativ ausgeführte Befehle verwendet, tatsächlich mit Monitor dieselbe Geschwindigkeit wie ohne Monitor. Je nachdem, wie oft und welche Befehle emuliert werden müssen, kommt es zu einer Verlangsamung. Der Vergleich "printf", "Fibonacci mit Ausgabe" und "Fibonacci ohne Ausgabe" zeigt deutlich, wo Optimierungen erfolgen müssen: bei Zugriffen auf Geräte.

Ein anderer Hinweis ergibt sich bei der Betrachtung der max-Werte: entscheidet der Wirt, einer anderen Task als dem Monitor den Prozessor zuzuweisen, während ein Befehl emuliert wird, steigt dessen Verarbeitungszeit rapide an. Dies kann das Timing des Gastes stören[1]. Dieses Thema sollte genauer untersucht werden.

Zusammenfassend kann man noch sagen, dass die Zahlen die intuitive Auffassung bestätigen: Kontextwechsel sollten, soweit möglich, vermieden werden. Die Emulation von Geräten im Nutzerkontext des Wirtes ist zwar die Variante mit dem geringsten Aufwand, da die Gerätetreiber des Wirtes verwendet werden können, jedoch auch die langsamste. Falls es gelingt, die Gerätetreiber im Monitor zu platzieren, kann eine erhebliche Geschwindigkeitssteigerung erwartet werden.

[1] Während der Weiterentwicklung von plex86 gab es in der Tat erhebliche Schwierigkeiten, Linux' BogoMIPS-Messung durchzuführen.

Kapitel 6

Schlussfolgerungen, Fragen und Ausblick

Die größten Schwierigkeiten bereiten folgende Eigenheiten der IA32: Systemregister sind auch für nichtprivilegierte Programme lesbar, das komplizierte Befehlsformat erlaubt geschachtelte und sich überlappende Befehle und ein Kontextwechsel zwischen Kern- und Nutzermodus dauert relativ lange.

Ein Blick in den Linux-Quelltext zeigt, dass bei weitem nicht alle der gefundenen kritischen Befehle verwendet werden, vermutlich gilt gleiches auch für andere verbreitete Betriebssysteme. Hier sind noch viele Messungen und Experimente nötig, um herauszufinden, welche von den vorgestellten möglichen Techniken zu den besten Ergebnissen im Zusammenhang mit einem bestimmten Gast führen. Vor allem die Unterstützung von Geräten muß weiter ausgearbeitet werden. In diesem Kontext ist auch die Frage interessant, wieviel Performancegewinn erzielt werden kann, wenn alle Entwurfsentscheidungen zugunsten einer hohen Ausführungsgeschwindigkeit fallen. Denn die Messergebnisse zeigen, dass durch die Virtualisierung zum Teil erhebliche Einbußen in der Performance in Kauf genommen werden müssen.

Weitere interessante Fragen sind:

- Sind Echtzeit und Virtualisierung vereinbar?
- Wie kann man auf x86-PCs gefundene Entwurfsmuster auf andere Architekturen übertragen?
- Vereinfacht eine Virtualisierung die Lastbalancierung, da ja die direkte Kopplung von Betriebssystem und Ressourcen aufgebrochen wird?

Die nächsten Schritte sind, die vorhandene Version von plex86 weiterzuentwickeln, um Messungen mit verbreiteten Betriebssystemen durchführen zu können. Mit diesen Ergebnissen kann das Design überarbeitet werden. Je nach den Anforderungen, die sich durch den Einsatz des VMM in bestimmten Umgebungen (im Büro, in wissenschaftlichen Labors, bei Applikationserviceprovidern etc.) ergeben, sollten Optimierungen durchgeführt werden.

Kapitel 7

Zusammenfassung

Diese Arbeit gibt einen methodischen Einstieg in das Themengebiet der Virtualisierung. Sie gibt einen Überblick, zeigt Stellen auf, an denen weitere Forschungen nötig sind und lotet Grenzen aus.

Es wurden eine reale und virtuelle Maschine beschrieben und Funktionen zur Abbildung der virtuellen auf die reale Maschine gefunden. Von den verschiedenen diskutierten Ansätzen wurde jeweils nur einer, meist der einfachste, implementiert. Ziel war vor allem, erste Testkerne laufen zu lassen, um damit zu experimentieren und Erfahrungen zu sammeln.

Das, was über Virtualisierung allgemein und für x86-PCs im Besonderen öffentlich bekannt war, wurde zusammengetragen und um eigene Ideen und Konzepte ergänzt.

Nunmehr steht ein erster Prototyp eines Monitors für virtuelle Maschinen zur Verfügung. Er wurde soweit vorangetrieben, dass einfache Testkerne, wie sie zum Beispiel leicht aus dem OSKit entstehen können, funktionieren. Die gewählte Architektur zielt dabei auf einen geringen Aufwand bei der Virtualisierung von Geräten ab, nimmt damit aber zum Teil erhebliche Performanceeinbußen in Kauf. Eine alternativ erarbeitete Architektur würde bedeutende Geschwindigkeitsvorteile bringen, jedoch einen großen Aufwand bei der Geräteunterstützung erfordern. Die Basistechniken sind bei beiden jedoch gleich oder zumindest sehr ähnlich.

Anhang A

Protokolle

A.1 plex86 $\leftrightarrow$ Linux Kern Modul

- ioctl

PLEX86_ALLOCVPHYS

- Function: PLEX86_ALLOCVPHYS
- Data: Number of MB to allocate
- Intended action: Allocate unpaged memory for the VM
- Intended reply: 0 on success or error code
- Description: The monitor needs memory for several reasons (guests main memory, data structures and so on), allocate this memory here

PLEX86_ALLOCINT

- Function: PLEX86_ALLOCINT
- Data: Number of the interrupt
- Intended action: Allocate an interrupt for emulating in user space
- Intended reply: 0 on success or error code
- Description: This is for emulating devices in user space; plugins can allocate resources like interrupts or I/O-Address-Spaces

PLEX86_RELEASEINT

- Function: PLEX86_RELEASEINT
- Data: Number of the interrupt
- Intended action: Release an interrupt for emulating in user space
- Intended reply: 0 on success or error code
- Description: This is for emulating devices in user space; plugins can allocate resources like interrupts or I/O-Address-Spaces

PLEX86_RESET

- Function: PLEX86_RESET
- Data: -
- Intended action: Reset the in-use count
- Intended reply: 0 on success or error code
- Description: This is for debugging, so that the module can be "rmmoded"

PLEX86_RUNGUEST

- Function: PLEX86_RUNGUEST
- Data: -
- Intended action: Switch to the monitor context
- Intended reply: Emulation code on success or error code
- Description: Switch to monitor context so that a guest can run

PLEX86_TEARDOWN

- Function: PLEX86_TEARDOWN
- Data: -
- Intended action: Close down the monitor
- Intended reply: 0 on success or error code
- Description: At the end, some cleanups happen while the VMM is closed

A.2 Wirt $\leftrightarrow$ Gast (Π_0)

A.2.1 General format

- Intended action: an action, preformed by kernel modul or user space program (plugin)
- Intended reply : a reply to guest, stored in event_info and error
- Description : some words of description

A.2.2 Functions with action performed in host context

RET_BECAUSE_REDIR

- Function: RET_BECAUSE_REDIR
- Subfunction: interrupt vector
- Data: -
- Intended action: Handle a redirected interrupt
- Intended reply: -
- Description: Monitor reflects all interrupts to host; host has to handle interrupt

RET_BECAUSE_EMERR

- Function: RET_BECAUSE_EMERR
- Subfunction: 13 (GP(0))
- Data: -
- Intended action: Close virtual machine
- Intended reply: -
- Description: Unable to emulate a particular thing, it's safe to stop and close virtual machine

RET_BECAUSE_MON_ERROR

- Function: RET_BECAUSE_MON_ERROR
- Subfunction: -
- Data: -
- Intended action: Close virtual machine
- Intended reply: -
- Description: An error happened in monitor itself, it's safe to stop and close virtual machine

RET_BECAUSE_REMAP

- Function: RET_BECAUSE_REMAP
- Subfunction: -
- Data: -
- Intended action: Remap (unmap; map) monitor in guests's address space
- Intended reply: -
- Description: Segments or page tables changed, monitor has to be remapped to a new place

RET_BECAUSE_USEREMU

- Function: RET_BECAUSE_USEREMU
- Subfunction:
  - EMU_INSTR_OUT_8
  - EMU_INSTR_OUT_16
  - EMU_INSTR_OUT_32
  - EMU_INSTR_IN_8
  - EMU_INSTR_IN_16
  - EMU_INSTR_IN_32
  - EMU_INSTR_OUTS_8
  - EMU_INSTR_OUTS_16

  - EMU_INSTR_OUTS_32
  - EMU_INSTR_INS_8
  - EMU_INSTR_INS_16
  - EMU_INSTR_INS_32
  - EMU_INSTR_REP_OUTS_8
  - EMU_INSTR_REP_OUTS_16
  - EMU_INSTR_REP_OUTS_32
  - EMU_INSTR_REP_INS_8
  - EMU_INSTR_REP_INS_16
  - EMU_INSTR_REP_INS_32
  - EMU_INSTR_HLT
  - EMU_INSTR_INT
  - EMU_INSTR_STI
- Data: specific to user space emulation function
- Intended action: Emulate a particular instruction in user space
- Intended reply: -
- Description: Some instructions have to be emulated in user space, mostly by plugins

RET_BECAUSE_KERNELEMU

- Function: RET_BECAUSE_KERNELEMU
- Subfunction:
  - PAGE_CACHE_EMPTY
  - PAGE_CACHE_FULL
- Data: -
- Intended action: refill/empty page cache
- Intended reply: OK/FAIL
- Description: Monitor needs more pages than page cache contains or wants to put more pages into page cache than space is left → refill/empty page cache

A.2.3 Functions with action performed in guest context

MON_ACTION_RAISE_INT

- Function: MON_ACTION_RAISE_INT
- Subfunction: interrupt vector
- Data: -
- Intended action: Raise an interrupt for guest
- Intended reply: -
- Description: Emulate an interrupt exception for guest

MON_ACTION_SET_VIP

- Function: MON_ACTION_SET_VIP
- Subfunction: -
- Data: -
- Intended action: Set VIP-Flag in monitor's eflags
- Intended reply: -
- Description: Just set this flag

A.2.4 See also

See also files:

- plex86.h
- monitor.h

A.3 Monitor ↔ Gast

General Protection Fault

- Function: General Protection Fault
- Data: -
- Intended action: Do emulation or redirect to guest
- Intended reply: -
- Description: Check if some emulation has to be done; if the GP was from an guest application, redirect the GP to guest

Page Fault

- Function: Page Fault
- Data: Page Fault Address
- Intended action: Do emulation or redirect to guest
- Intended reply: -
- Description: Check if some emulation has to be done; if the PF was not for emulation purposes, redirect to guest

Interrupt

- Function: Interrupt
- Data: -
- Intended action: Forward interrupt to monitor inside driver, guest or host
- Intended reply: -
- Description: An interrupt can be handled in monitor, guest or host; the interrupt has to be redirected to the right place, depending on the emulation model and the interrupt number

Anhang B

Emulation privilegierter Befehle

Eine kurze Zusammenfassung, wie privilegierte Befehle virtualisiert werden:

- hlt: Der Gast bekommt keine Rechenzeit mehr, bis ein Interrupt auftritt
- cli: Dem Gast werden keine Interrupts mehr zugestellt
- sti: Dem Gast werden wieder Interrupts zugestellt
- in/ins/out/outs: Der Emulator für das Gerät, welches sich hinter dem betreffenden Port befindet, tritt in Aktion
- invd: Wird ignoriert
- wbinvd: Wird ignoriert
- invlpg: Aktive Seitentabellen anpassen
- lgdt: Speichere die Adresse im virtuellen GDTR, erzeuge eine neue aktive GDT und verwende diese
- lidt: Speichere die Adresse im virtuellen IDTR ,erzeuge eine neue aktive IDT und verwende diese
- lldt: Speichere die Adresse im virtuellen LDTR, erzeuge eine neue aktive LDT und verwende diese
- ltr: Speichere die Adresse im virtuellen TR, erzeuge ein aktives TSS und verwende dieses
- mov r32, CRx: Kopiere das virtuelle CRx in r32
- mov CRx, r32: Kopiere r32 ins virtuelle CRx; passe eventuell den Zustand der virtuellen Maschine an (z.B. Seitentabellen im Falle von CR3, Verhalten im Falle von CR1)
- mov r32, DRx: Kopiere das virtuelle DRx in r32
- mov DRx, r32: Kopiere r32 in das virtuelle DRx; emuliere das Verhalten in Software
- rdmsr: kopiere virtuelles MSR in EDX:EAX
- wrmsr: kopiere EDX:EAX in virtuelles MSR
- lmsw: Passe den Zustand und das Verhalten der virtuellen Maschine an

Anhang C

Glossar

Adressraum: Menge gültiger Speicheradressen, auf die ein Thread zugreifen kann

API: "Application Programming Interface"; beschreibt die Programmierschnittstelle, die ein Dienstanbieter zur Verfügung stellt

Applikation: Anwendung, Anwendungsprogramm

CISC: "Complex Instruction Set Computer"

CPU: "Central Processing Unit"; Prozessor

DMA: "Direct Memory Access"; Zugriff auf den Hauptspeicher ohne CPU, z.B. mit dem alten DMA-Chip oder PCI-Busmaster-Geräten

Emulator: Programm, welches emuliert

emulieren: nachahmen, nachbilden

Gast: In diesem Zusammenhang: Betriebssystem, welches in einer virtuellen Maschine läuft

GDT: "Global Descriptor Table"; Globale Tabelle, in der Segmentdeskriptoren abgelegt sind

LGPL: "GNU Lesser General Public Licence"; ein Softwarelizenzmodell der Free Software Foundation, Inc.

IA: "Instruction Architecture"; Menge von Befehlen, die eine CPU ausführen kann

Opcode: Operationscode; ein Datum, welches einen Befehl der IA32 beschreibt

Page Fault: siehe *Seitenfehler*

Page Table: siehe *Seitentabelle*

PCI: "Peripheral Component Interconnect"; sehr weit verbreitetes Bussystem, in x86-PCs Standard für Erweiterungskarten

Privilegstufe: Schutzebene des Prozessors

Port: Eine Adresse im Adressraum der CPU, die direkt mit einem Gerät assoziiert ist; das Gerät spezifiziert die Bedeutung des Ports

Ring: In diesem Zusammenhang: Schutzebene des Prozessors, Synonym für Privilegstufe

SEC: "Self Examining Code"; ein Programm(-stück), welches sich selbst überwacht

Segmentdeskriptorcache: Unsichtbarer Cache; jedem Segmentselektor ist ein Segmentdeskriptorcache zugeordnet, der die Werte des Deskriptors aufnimmt, auf den der Selektor verweist

Seitenfehler: Unerlaubter Zugriff auf eine Speicheradresse; entweder gehört diese Adresse nicht zum Adressraum oder der Thread greift schreibend auf Nur-Lese-Speicher zu

Seitentabelle: Datenstruktur, die *Adressräume* aufeinander abbildet

SMC: "Self Modifying Code"; ein Programm(-stück), welches sich selbst verändert

Superpage: Speicherseite mit einer Größe von 4 MB

TLB: "Translation Lookaside Buffer"; Bestandteil der CPU für die schnelle Übersetzung von linearen in reale Adressen

Virtualisieren: von realen Gegebenheiten abstrahieren und die Abstraktionen auf die reale Welt abbilden

virtuell: gedacht, nicht wirklich

Wirt: In diesem Zusammenhang: Betriebssystem, welches auf einer realen Maschine läuft und Gäste beherbergen kann

Literaturverzeichnis

[1] Diverse Artikel in comp.arch, nachzulesen z.B. in www.deja.com

[2] Disco: Running Commodity Operating Systems on Scalable Multiprocessors; http://www-flash.stanford.edu/Disco

[3] Hans-Peter Messmer: Das PC Hardware Buch; Addison-Wesley, 1995

[4] Klaus-Dieter Thies: 80486 Systemsoftware-Entwicklung; Hanser, 1992

[5] freemware- und plex86-Mailing-List (Insbesondere Kevin P. Lawton, Ulrich Weigand und Ramon van Handel); http://www.mail-archive.com/freemware-list%40fastxs.net

[6] K. Lawton, U. Weigand, R.v.Handel, C. Adjih, C. Dickey, N. Behnken, J. Nerche: Running multiple operating systems concurrently on an IA32 PC using virtualization techniques; http://www.plex86.org/research/paper.txt

[7] Intel: Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture
http://developer.intel.com/design/PentiumIII/manuals

[8] Intel: Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference
http://developer.intel.com/design/PentiumIII/manuals

[9] Intel: Intel Architecture Software Developer's Manual, Volume 3: System Programming
http://developer.intel.com/design/PentiumIII/manuals

[10] LGPL: erhältlich z.B. im WWW unter http://www.gnu.org/copyleft/lesser.html

[11] L4: The L4 μ-Kernel Family; http://os.inf.tu-dresden.de/L4

[12] Fiasco: http://os.inf.tu-dresden.de/fiasco

[13] OSKit: The OSKit Project; http://www.cs.utah.edu/flux/oskit

[14] L4Linux: Linux on L4; http://os.inf.tu-dresden.de/L4/LinuxOnL4

[15] WINE: Windows on UNIX; http://www.winehq.com

[16] SimOS: The Complete Machine Simulator;
http://simos.stanford.edu

[17] VMware: Try VMware 2.0 Now!;
http://www.vmware.com/vmwarestore/vmwarestore.html#eval

[18] Brown-Simulator: High-level machine simulator;
http://www.cs.brown.edu/software/brownsim

[19] Sheepshaver: MacOS run-time environment for BeOS;
http://www.sheepshaver.com

[20] Mac-on-Linux: Run MacOS under PowerPC Linux;
http://www.ibrium.se/linux/mac_on_linux.html

[21] a386: C programming library which provides a virtual machine;
http://a386.nocrew.org

[22] User-Mode-Linux: Run Linux Inside Itself;
http://user-mode-linux.sourceforge.net

[23] Melinda Varian: VM and the VM Community: Past, Present, and Future;
http://pucc.princeton.edu/~melinda

[24] TLB-Test-Tool: ftp://ftp.freemware.org/pub/freemware/1.44_tlb.gz

[25] Ergebnisse des TLB-Trick-Tests: http://www.plex86.org/news.phtml?id=3