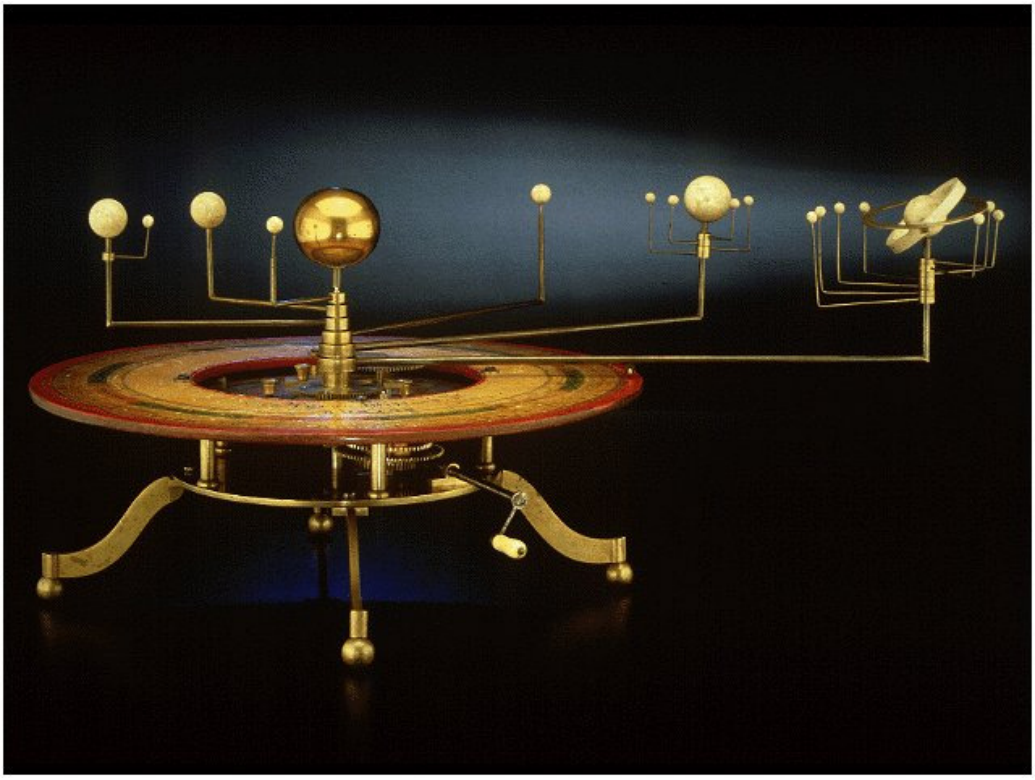
Proseminar Technische Informatik

A survey of virtualization technologies



http://www.zodiacal.com/attic/smithsonian_orrery.jpg

von
Martin Weigelt
[weigelt@mi.fu-berlin.de]

Inhaltsverzeichnis

Was ist Virtualisierung?

Kurzer geschichtlicher Hintergrund

Das Konzept der Virtualisierung entstand 1960 basierend auf den von IBM entwickelten Mainframe Computern. Hier sollte es mehreren Benutzern möglich sein, gleichzeitig auf den Mainframe Computer zuzugreifen. Die dabei verwendete Virtuelle Maschine stellte den Benutzern eine Abstraktion der physischen Maschine zur Verfügung, in dessen Umgebung sie den Eindruck vermittelt bekamen, exklusiven Zugriff auf die Ressourcen des Computers zu haben.
Dies spiegelt die Idee des Time-Sharing-Systems wieder, das 1959 von C. Strachey in einem Paper "Time Sharing in Large Fast Computers", formalisiert wurde.
Die Virtuelle Maschine garantierte den Benutzern eine geschützte und isolierte Umgebung, in der sie Programme entwickeln, testen und ausführen konnten, ohne dass dies ein Effekt auf die native Hardware oder andere Benutzer hatte. Dies stellte einen eleganten Weg dar, die teure Hardware effizient auszunutzen und produktiv zu arbeiten.
In den frühen siebziger Jahren veröffentlichte J. Popek und Robert P. Goldberg ein Paper, in dem die Anforderungen an die Architektur und Software untersucht wurden, um Virtualisierung zu ermöglichen. Daraus gehen drei wichtige Eigenschaften hervor, die Virtuelle Maschinen erfüllen sollten:

1. "*The efficiency property.* All innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the control program."[5]

2. "*The resource control property.* It must be impossible for that arbitrary program to affect the system resources, i.e. memory, available to it; the allocator of the control program is to be invoked upon any attempt."[5]

3. *"The equivalence property.* Any program K executing with a control program resident, with two possible exceptions, performs in a manner indistinguishable from the case when the control program did not exist and K had whatever freedom of access to privileged instructions that the programmer had intended."[5]

Aufgrund der sinkenden Hardwarekosten in den 1970 und 1980 sank auch das Interesse an Virtuellen Maschinen, da nun auch vermehrt Betriebssysteme mit Multitasking eingesetzt wurden.
Erst in den 1990 wurden die Virtuellen Maschinen wieder interessant, da nun eine Menge unterschiedlicher Hardware und Software zur Verfügung stand, die untereinander nicht immer kompatibel ist.

Definitionen

Eine präzise Definition des Begriffs Virtualisierung ist nicht möglich, da der Begriff in unterschiedlichen Kontexten andere Bedeutungen hat. Trotzdem gibt es einige gute, die die Thematik meiner Meinung nach gut schildern.

"Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and others."[4]

Hier wird von Zusammenfügen und Teilen der Computer Ressourcen gesprochen. Das stellt meiner Meinung nach sehr anschaulich dar, wie skalierbar Ressourcen mit Virtualisierung sind, und wie effizient sie sich damit ausnutzen lassen.

„virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others."[1]

In dieser Definition wird Virtualisierung als ein Framework bezeichnet, das Computer Ressourcen in mehrere Ausführungsumgebungen teilt. Die Bezeichnung des Frameworks zeigt meiner Meinung nach gut die neue Abstraktionsebene, die durch Virtualisierung entsteht.

Einsatzzwecke

Virtualisierung wird heutzutage in sehr vielen Bereichen angewendet. Daher konzentriere ich mich hier nur auf die gebräuchlichsten Anwendungen.

Virtualisierung wird heute hauptsächlich eingesetzt, um Kosten zu minimieren, um eine höhere Ausfallsicherheit zu gewährleisten und um Ressourcen besser auszulasten. Insbesondere Unternehmen setzen vermehrt auf Virtualisierung, um ihre vorher unausgelasteten dedizierten Server nun in eine virtuelle Umgebung auf einen leistungsstarken Computer zusammenzufügen. Das lastet die Hardware-Ressourcen zwar besser aus, kann aber bei einem Hardwaredefekt Nachteile haben.

Auch Entwickler setzen gerne Virtualisierung ein, um ihre Software auf unterschiedlichen Plattformen zu testen.

Anwender können in virtuellen Systemen Software testen, Software anderer Systeme nutzen oder sicherheitskritische Anwendungen ausführen.

Virtualisierung vs. Emulierung

Um die Unterschiede und Gemeinsamkeiten von Emulierung und Virtualisierung klar zu machen, werde ich hier einen kleinen Einblick in die Implementierung von Emulatoren geben und dann die Gemeinsamkeiten und Unterschiede beschreiben.

Emulation, zu Deutsch nachahmen, beschreibt ein Verfahren, das zwei inkompatible Systemarchitekturen auf ein System abbildet. Dies erledigen zumeist Programme(sog. Emulatoren),

die die komplette Systemarchitektur, insbesondere den Prozessor, des Gastsystems nachbilden. Dabei werden also alle Komponenten des Gastsystems in Software umgesetzt. So sollen alle Programme, die auf dem „Original“- System ausgeführt werden, auch auf dem Gastsystem, auf jede Eingabe das gleiche Ergebnis liefern. Dadurch ist auch klar, dass Emulation sehr aufwändig und weniger leistungsstark sein kann als Virtualisierung.
Virtualisierung hingegen ist für eine Systemarchitektur ausgelegt und interagiert meistens direkt mit der Hardware. Jedoch machen einige Virtualisierungsprogramme auch Gebrauch von der Emulationstechnik, um Gastsysteme mit einer anderen Systemarchitektur zu unterstützen. Dies ist bei Virtualisierung allerdings eher eine Ausnahme, da wie bereits erwähnt dadurch mit starken Leistungsverlusten gerechnet werden muss.
Das Virtualisierungskonzept zielt also nicht darauf ab, mit möglichst vielen Systemarchitekturen kompatibel zu sein, sondern auf einer einheitlichen Systemarchitektur möglichst effizient zu sein.
Die Virtuelle Maschine ist dabei die Schlüsselkomponente eines jeden Virtualisierungsprogramms. Sie ist dafür verantwortlich, dass die Befehle der Gastsysteme an die native Hardware weitergeleitet werden und die Ergebnisse wieder an das entsprechende Gastsystem zurückgeleitet werden. Eines der Hauptprobleme bei der x86-Architektur ist der Kernel-Modus und die Rechte, um auf die Hardware zuzugreifen. Denn das Gastsystem ist in der Annahme alleinigen Zugriff auf die Systemressourcen zu haben und spricht diese mit den privilegierten Befehlen des Kernel-Modus an. Diese muss die Virtuelle Maschine jedoch emulieren, denn das Gastsystem im User-Mode hat nun nicht mehr die benötigte Berechtigung diese Befehle auszuführen.

Virtualisierungsmethoden

Betriebssystemvirtualisierung mittels OS-Container

Häufig möchte man zum Testen neuer Software, für eine sichere virtuelle Umgebung oder ähnlichen Gründen nicht extra ein komplett neues Betriebssystem installieren und konfigurieren müssen. Hier würde es genügen, basierend auf dem bestehenden Betriebssystem, eine Virtuelle Umgebung zu schaffen, die die bestehende Konfiguration übernimmt und in einen virtuellen Kontext setzt. Das ist die Idee der Betriebssystemvirtualisierung, die eine Virtuelle Maschine auf einem Betriebssystem ausführt, sodass ein Programm keinen Unterschied zwischen der Realen- und der Virtuellen Umgebung bemerkt. Dabei findet die Virtualisierung im Stack Speicher statt. Die Virtuellen Maschinen teilen sich hierbei das Betriebssystem und die darunterliegende Hardware und stellen den Anwender so eine unabhängige und isolierte Umgebung zur Verfügung.
Es existieren schon zahlreiche Lösungen in diesem Bereich. Im Folgenden stelle ich die Implementierung von FreeBSD Jails vor. Des Weiteren gibt es auch noch iCore Virtual Accounts(WinXP), Parallels Virtuozzo Containers(Windows/Linux), OpenVZ(Linux) und andere.

Free BSD Jail

Ein Jail, zu Deutsch Gefängnis, in FreeBSD ist eine Virtualisierungssoftware, um eine isolierte Umgebung aus dem UNIX „root“-Model zu erstellen. In der „Jail“-Umgebung werden dabei Betriebssystemressourcen, wie Prozesse, das Dateisystem, Netzwerk-Geräte usw. verwendet, deren Manipulationen aber nur innerhalb dieser Umgebung sichtbar werden. Wird ein Prozess in dieser Umgebung ausgeführt, dann werden auch alle Prozesse, die dieser Prozess erzeugt wieder in dieser Umgebung ausgeführt. Außerdem kann ein Prozess, der in einer Jail ausgeführt wird, diese nicht verlassen oder auf andere Prozesse in anderen „Jails“ zugreifen.
Die „Jail“-Architektur lässt sich in zwei Bereiche aufteilen: Das Programm im „User-Space“ Jail(8) und der Code, der im Kernel implementiert ist: Der Jail(2) Systemaufruf.
Um neue Jails zu erzeugen wird einfach der Systemaufruf Jail(2) aufgerufen, der dann einen neuen Jail erzeugt. Bei diesem Systemaufruf wird eine neue Datenstruktur erzeugt und an die struct proc des aktuellen Prozesses angefügt. Letztlich wird noch der Systemaufruf chroot(2) gemacht, um den Vorgang abzuschließen.
Die Sichtbarkeit des Prozesses wird dadurch kontrolliert, indem das procfs Dateisystem und der sysctl Baum manipuliert werden[4].

Virtualisierung von Programmiersprachen

Im Laufe der Zeit ist eine große Menge an Software für unterschiedliche Betriebssysteme entstanden, die untereinander aufgrund der Betriebssystemarchitektur nicht kompatibel sind. Nicht allzu oft musste die Software aber mit anderen Betriebssystemen kompatibel sein. Das bedeutete für die Hersteller jedoch, die Software auf das andere System zu portieren. Dieser Vorgang gestaltete sich aber meistens als schwierig, umständlich und teuer.
Um diese Problematik zu lösen entstand die Idee, eine Maschine auf der Anwendungsebene zu realisieren, die eine Menge von Anwendungen interpretieren kann. Abstrakt ist eine Maschine „[…]one that executes a set of instructions as supported by the Instruction Set Architecture (ISA).”[4].
Mit dieser Idee wurde eine Virtuelle Maschine realisiert, die eine neue selbstdefinierte Menge Befehle unterstützt (sog. Bytecode). Diese Befehle werden dann während der Laufzeit des Programms in Befehle des nativen Systems umgesetzt (Just-In-Time Compiling).
Um zu verdeutlichen, was in diesen Virtuellen Maschinen geschieht, werde ich im Folgenden einen kurzen Einblick in die Implementierung der Java Virtual Maschine geben. Außerdem verwenden auch die Programmiersprachen Python, C#, VB.NET u.a. Virtuelle Maschinen, um plattformunabhängig zu sein.

Java Virtual Maschine(JVM)

Aus der Idee eine Prozessor-unabhängige Sprache zu entwickeln, entstand letztlich die Programmiersprache Java. Das Backend bildet dabei der Interpreter für die Java-spezifischen Befehle (Bytecode). Das Backend ist unterdessen fester Bestandteil der Java Laufzeitumgebung(JRE), die schon für viele Betriebssysteme existiert. Dieses Backend ist nun auch besser bekannt als Java Virtual Maschine(JVM).

Oft wird der Java Bytecode durch den Compiler der Java-Entwicklungsumgebung erzeugt (JDK), aber er kann auch durch andere Compiler erzeugt werden. Der Bytecode wird dann von der JVM interpretiert oder mit dem Just-In-Time-Compiler(JIT) ausgeführt. Die Bytecode Befehle werden dabei zur Laufzeit in native Befehle des jeweiligen Betriebssystems transformiert. Dies wird durch die „Virtuelle Hardware“ der Java Virtual Maschine erledigt, die hauptsächlich aus Registern, einem Stack, dem Heap und dem Methoden-Bereich besteht. Das sind die grundlegenden Elemente einer jeden JVM. Die JVM unterstützt dabei bis zu 4GB Speicher mit 32-Bit Adressierung und 32-Bit virtuellen Registern[4]. Außerdem werden die bekannten primitiven Datentypen byte, short, int, long, float, double, char und object unterstützt. Im Unterschied zu anderen Programmiersprachen ist der Befehlszähler nicht in einem speziellen Register, sondern wird in der JVM durch drei Register verwaltet. Ein Register wird verwendet, um den Bereich der lokalen Variablen in der Methode festzuhalten. Ein zweites Register zeigt auf die Ausführungsumgebung des Stacks, die alle lokalen Variablen und die Stack-Befehle beinhaltet. Dann wird noch ein Register benötigt, um auf den Stack-Anfang zu referenzieren, wo alle Operationen und Ergebnisse gespeichert sind.

Außerdem wird der Code vor der Ausführung eines Programms geprüft, um z.B. zu verhindern, dass unbefugte Speicherzugriffe gemacht werden können. Dadurch ist ein effizienter und sicherer Speicherschutz gewährleistet ohne die Verwendung einer MMU.

Insgesamt wurde also durch die Virtualisierung eine sichere, plattformunabhängige und effiziente Umgebung für Anwendungen geschaffen, die hardwarenahen Anwendungen, die z.B. in C und C++ geschrieben wurden, in nichts nachstehen.

Virtualistisierung auf der Befehlssatzarchitektur (Emulation)

Manchmal ist es notwendig, Software von Systemen zu verwenden, die inkompatibel mit dem eigenen System sind. Diese Systeme unterscheiden sich zumeist in der Art ihrer Hardware, also ihres Prozessors, ihrer Speichermedien, ihres Bussystems, ihrer Ein-und Ausgabegeräte usw. Um trotzdem mit diesen Systemen, unter Verwendung des eigenen Systems, arbeiten zu können, ist es notwendig die ausgeführten Hardwarebefehle des Gastsystems in Hardwarebefehle des nativen Systems zu übersetzen. Also wird die komplette Hardware des Gastsystems emuliert.

Dadurch lassen sich einfach mehrere unterschiedliche Systeme auf einem dazu verschiedenen System ausführen. Z.B. kann so auf einer x86-Architektur andere Architekturen, wie x86, Sparc, Alpha o.a. verwendet werden. Diese Technik hat allerdings den signifikanten Nachteil, dass durch die Umsetzung der Hardware des Gastsystems in Software, die Performance des Gastsystems viel schlechter ist, als würde es auf der nativen Hardware ausgeführt werden. Im Folgenden werde ich dazu einen Einblick in die Software QEMU geben. Weitere Verwendung dieser Technik findet sich auch in der Software Bochs, Crusoe, BIRD u.a.

QEMU

QEMU[7] ist ein schneller Prozessor-Emulator, der auch Gebrauch von dynamischen Übersetzungsmechanismen macht. QEMU bietet dabei zwei Betriebsarten an: Komplette System Emulation und Linux User Mode Emulation. Bei der kompletten System Emulation wird ein nicht-modifiziertes Betriebssystem in der Virtuellen Maschine ausgeführt. Bei der Linux User Mode Emulation kann ein Linux-Prozess, der für eine bestimmte Prozessorarchitektur kompiliert wurde, auf einer anderen ausgeführt werden. Außerdem kann QEMU noch zum Debuggen und für systemübergreifendes Kompilieren von Code genutzt werden. Zurzeit unterstützt QEMU bei der kompletten Systeme Emulation die Systeme: Sparc32/64, MIPS, ARM, ColdFire, PowerPC und x86.
Eine bedeutende Rolle spielt bei QEMU der Dynamische Übersetzer, der die Gastbefehle zur Laufzeit in native Befehle umsetzt. Dieser wird in QEMU aber noch durch die Portabilität ergänzt. Das hat den Vorteil, dass der Emulator auch auf anderen Systemen ausgeführt werden kann.
In der Regel ist es schwierig, einen Emulator von einem Host auf den anderen zu übertragen, da der gesamte Codegenerator angepasst werden muss. Dies wird in QEMU dadurch gelöst, dass sogenannte Mikrooperationen eingeführt werden. Die Mikrooperationen dienen dabei als eine Zwischensprache, in die die Befehle des Gastsystems übersetzt werden und aus diesen dann die nativen Befehle konstruiert werden. Dadurch wird die Ausführungsgeschwindigkeit zwar beträchtlich reduziert, aber die Portabilität erhöht. Um diesen Performanceverlust so gering wie möglich zu halten, verwendet QEMU zusätzlich noch einen Cache und versucht durch Tricks wie Konkatenation von Mikrooperationen zu Funktionen, die Ausführungsgeschwindigkeit zu erhöhen.

Virtualisierung auf der Hardware-Ebene(HAL)

Die Virtualisierung auf der Hardware-Ebene ist heutzutage eine der am meist verwendeten Virtualisierungstechniken. Diese Popularität verdankt sie hauptsächlich ihrer Effizienz und durch ihren praktikablen und rentablen Gebrauch auf der x86-Architektur. Die Virtualisierung auf der Hardware-Ebene nutzt dabei die Ähnlichkeit von Gast- und Hostsystem aus, um die Übersetzungszeit der Befehle zu minimieren. Die Schlüsselkomponente bei dieser Technik ist ein Virtual Machine Monitor(VMM) oder auch Hypervisor genannt. Eine VMM ist dabei ein minimales Betriebssystem,

das direkt auf der nativen Hardware des Systems läuft. Eine VMM wird verwendet um Virtuelle Maschinen zu erzeugen und zu verwalten. Beim Erzeugen einer Virtuellen Maschine wird dieser eine Abstraktion der realen Hardware übergeben, auf der dann das Gast-Betriebssystem zugreift.
Eine weitere Aufgabe einer VMM ist es die Virtuellen Maschinen zu verwalten und die realen Hardware-Ressourcen unter den Virtuellen Maschinen zu teilen. Dieser Vorgang ist aber für die Virtuellen Maschinen transparent, d.h. sie gehen davon aus, dass sie im alleinigen Besitz der Systemressourcen sind. Eine VMM speichert und aktualisiert hierbei jeweils immer den Zustand der entsprechenden Virtuellen Maschine.
Eines der Hauptprobleme ist, dass die privilegierten Befehle der Virtuellen Maschinen von einer VMM erkannt und emuliert werden müssen, da sie sonst nicht die nötigen Rechte haben, um ausgeführt werden zu können. Eine Lösung für dieses Problem wären Virtualisierungstechniken, wie „code scanning“ und „dynamically instruction rewriting“ zu verwenden. Des Weiteren ist es schwierig für eine VMM festzustellen, was in einer Virtuellen Maschine passiert und wie sie sich in diesem Zusammenhang verhalten soll. Das ist insbesondere ein Problem bei der Fehlerbehandlung, z.B. wenn ein Seitenfehler auftritt oder wenn es darum geht die Effizienz zu optimieren. Beispielsweise kann man nicht festzustellen, ob die Virtuelle Maschine gerade den Leerlaufprozess ausführt und unnötig den Prozessor belastet.
Eine der größten Herausforderungen einer VMM ist es also, die native Hardware der Virtuellen Maschine in einer effizienten, isolierten und transparenten Weise zugänglich zu machen [2].
Im Folgenden werde ich dieses Konzept an der Software VMware und XEN verdeutlichen.
Weitere Software die dieses Konzept benutzt, ist Microsoft Virtual PC, Linux KVM, Parallels Workstation, Sun Microsystems VirtualBox u.a.

VMware

Das Unternehmen VMware bietet unterdessen eine ganze Palette von Virtualisierungslösungen für die Bereiche: Rechenzentrumsplattformen, Desktop-Produkte, Mac-Produkte, Cloud und Managementprodukte an [9]. Ich werde mich hierbei allerdings nur auf die Produkte VMware Workstation und VMware ESX Server konzentrieren.
Eine VMM in VMware kann in zwei Arten betrieben werden. Zum einen als „hosted“ und zum anderen als „standalone“. Im „hosted“-Betrieb, der bei VMware Workstation verwendet wird, wird zusätzlich zur VMM ein Host-Betriebssystem vorausgesetzt. Diese Kombination optimiert unter anderem die Performance, die Portabilität und erleichtert die Implementation. VMware Workstation arbeitet dabei sowohl als VMM als auch als Anwendung auf dem Host-Betriebssystem. Die „hosted“-Architektur in VMware Workstation besteht aus den Grundkomponenten: VMApp, VMDriver und einer VMM.
VMApp ist eine Anwendung auf der Benutzerebene und VMDriver ist ein Gerätetreiber auf den Host-Betriebssystem. Dadurch wir es möglich, Befehle entweder direkt über das Host-Betriebssystem

auszuführen oder virtuell über die VMM. Für den Wechsel zwischen diesen beiden Fällen wird der VMDriver verwendet. So wird die Ein-und Ausgabe des Gastbetriebssystems zur Optimierung über das Host-Betriebssystem abgewickelt. Bemerkt die VMM also einen Schreib-oder Lesezugriff auf die Festplatte, wird dieser Befehl durch den VMDriver zur VMApp weitergeleitet und schließlich durch einen normalen Systemaufruf des Host-Betriebssystems ausgeführt. VMware nimmt desweiteren auch noch andere Optimierungen vor, die den Virtualisierungsaufwand reduzieren.
VMware ESX Server hingegen wird im „standalone"-Betrieb ausgeführt und arbeitet mit einer VMM, die alleine auf der nativen Hardware läuft. Darauf kann dann eine beliebige Anzahl von Virtuellen Servern isoliert voneinander laufen. Durch das fehlende Host-Betriebssystem, muss die VMM nun aber alle privilegierten Befehle selbst behandeln und kann nicht den Umweg über das Host-Betriebssystem gehen. VMware ESX Server löst das Problem, indem es Teile des Betriebssystem-Kernel-Codes dynamisch überschreibt (dynamically rewriting) und zusätzliche Ausnahmen hinzufügt, um privilegierte Befehle zu behandeln[1]. Insbesondere verwendet VMware auch sogenannte Schattenkopien der Systemstruktur, so dass z.B. Seitentabellen durch Virtuelle Seitentabellen ersetzt werden, die jeden Befehl abfangen, der versucht diese Struktur zu verändern[4]. Auch hier werden noch viele weitere Optimierungen von VMware gemacht.
VMware ESX Server unterstützt auch mehrere virtuelle Prozessoren für jeden physischen Prozessor. Außerdem können Physische Netzwerkkarten auch auf eine leistungsstarke virtuelle Netzwerkkarte zusammengefasst werden.

XEN

XEN ist eine Open-Source VMM für die x86-Architektur[10], die 2001-2002 an der Universität Cambridge entwickelt wurde[3]. Das Besondere an XEN ist, dass es Gebrauch von der Paravirtualisierungstechnologie macht, die ich in diesem Zusammenhang erläutern werde.
Die Idee von der Paravirtualisierung, ist es, die Zusammenarbeit des Gast-Betriebssystems und der darunterliegenden VMM zu optimieren, um dadurch die Performance und Skalierbarkeit zu maximieren. Der Virtuellen Maschine wird hierbei eine erweiterte Sicht auf die native Hardware gegeben.
Im Vergleich zu VMware Workstation wird hier nicht ein Host-Betriebssystem benötigt, sondern es werden modifizierte Gast-Betriebssysteme verwendet, um eine höhere Performance und Skalierbarkeit zu erreichen.
In XEN wird diese Technologie in jeder Virtuellen Maschine verwendet. Die Veränderungen, die an den Gast-Betriebssystemen vorgenommen werden müssen, versucht XEN dabei so gering wie möglich zu halten. Die Binärschnittstelle(ABI) verändert XEN allerdings nicht, wodurch auch keine Änderungen der Programme des Gastbetriebssystems notwendig sind. Um die Hardware Daten-Strukturen zu verändern, arbeitet das Gastbetriebssystem mit der VMM zusammen, indem es die von der VMM bereitgestellte API aufruft. Dieser Systemaufruf wird auch „hypercall" genannt. Das

Besondere bei einem „hypercall“ ist, dass dabei kein Kontextwechsel stattfindet, da die VMM in jedem Adressraum der Gastbetriebssysteme existiert. Dadurch müssen privilegierte Befehle nicht mehr durch die VMM emuliert werden, sondern können direkt durch die VMM aufgerufen werden. Die Isolation zwischen den Virtuellen Maschinen wird gewährleistet, indem XEN die Ressourcen strikt unter den verschiedenen Virtuellen Maschinen teilt. So bekommt jede VM ein Teil der gesamten Kapazität der physischen Ressourcen, wie CPU, Speicher, usw.
Geräte-Treiber der physischen Hardware werden in XEN besonders behandelt, da sie ein Sicherheitsrisiko für die VMM darstellen. Sie werden außerhalb der VMM in einer speziellen Domain(VM) ausgeführt, um einen Absturz der VMM, z.B. durch einen defekten Treiber, zu vermeiden. Außerdem werden in den Gast-Betriebssystemen die hardwarespezifischen Treiber durch paravirtualisierte Treiber ersetzt, die unabhängig von der physischen Hardware sind. Diese Treiber sind sehr leistungsstark, da sie für das Gastsystem keinen zusätzlichen Aufwand erfordern.
Durch die Technologien AMD Parcifica und Intel VT unterstützt XEN für diese Prozessoren nun auch nicht modifizierte Gast-Betriebssysteme.

Anwendungsszenarien

Während im Unternehmensbereich Virtualisierung vermehrt eingesetzt wird, um Kosten zu sparen und die Effizienz zu steigern, wird sie im privaten Bereich häufiger aus Sicherheits- und Kompatibilitätsgründen eingesetzt. Hier seien einige gebräuchliche Anwendungen von Virtualisierung aufgezählt.

Serverkonsolidierung

Im Server-Bereich wird Virtualisierung zumeist dazu verwendet, Server besser auszulasten und Hardwarekosten zu sparen. Das wird dadurch erreicht, dass mehrere dedizierte physische Server nun im Virtuellen Kontext auf einen physischen Server zusammengefasst werden. Dabei ist es wichtig, dass diese weiterhin isoliert voneinander ausgeführt werden, damit bei einem Softwarefehler oder bei einer Kompromittierung, andere virtuelle Server davon nicht betroffen sind. Diese Aufgabe wird nun durch die Virtualisierungssoftware übernommen.
Ein weiterer wichtiger Aspekt ist die Performance, die durch die Virtualisierung so wenig wie möglich beeinträchtigt werden soll. Ein Nachteil der Umstellung auf Virtualisierung stellt der „Single point of Failure“ dar, da nun alle Server auf der gleichen Hardware ausgeführt werden. Dies erfordert ein neues Ausfallkonzept. Das Backup-Konzept wird durch die Virtualisierung allerdings erleichtert, da die virtuellen Systeme nur als Image-Datei auf dem physischen System gespeichert sind und leicht vervielfältigt oder gesichert werden können.

Unter diesen Gesichtspunkten betrachtet, bieten sich zwei der oben vorgestellten Lösungen besonders an. Zum einen XEN durch seine hohe Performance und Isolation, und zum anderen VMware ESX Server durch seine Isolation und Flexibilität.

Verwendung von Software anderer Systemarchitekturen

Manchmal ist es notwendig, Software zu verwenden, die nicht kompatibel mit der eigenen Systemarchitektur ist und zu der keine entsprechende Hardware zur Verfügung steht. Hier kann mithilfe der Virtualisierung auf der Befehlssatzarchitektur, die Hardware emuliert werden und die entsprechende Software darauf betrieben werden. Eine Möglichkeit bietet die Anwendung der Virtualisierungssoftware QEMU, die eine Vielzahl von Systemarchitekturen unterstützt.
Dazu wird mit QEMU zunächst eine virtuelle Festplatte in Form einer Image-Datei erstellt und auf dieser dann das entsprechende Betriebssystem installiert. Anschließend kann die entsprechende Software auf dem virtuellen Betriebssystem installiert und verwendet werden.

Sandbox-Bereich

Eine Sandbox stellt eine isolierte Umgebung für Anwendungen dar. Hier kann der Anwender sicherheitskritische Programme ausführen oder Programme testen. Um diesen Zweck zu erfüllen, ist es aber nicht nötig, ein komplett neues Betriebssystem zu installieren und zu konfigurieren. Hier reicht es aufbauend auf das vorhandene Betriebssystem eine Virtuelle Maschine zu erstellen und darauf Anwendungen auszuführen. So kann z.B. ein Bereich für die Internet-Nutzung eingerichtet werden, um das native Host-System vor einer Kompromittierung zu schützen.
Dafür bietet sich für BSD: Jails an, für Windows XP: Icore Virtual Accounts und für Linux: OpenVZ.

Systemunabhängige Software-Entwicklung

Um Zeit und Kosten bei der Portierung von Software auf unterschiedliche Systeme zu sparen, kann Virtualisierung schon bei der Entwicklung von Software eine Rolle spielen. So wird z.B. Java verwendet, um die Kompatibilität der entwickelten Software mit anderen Systemen zu maximieren. Weitere Aspekte sind die sichere Ausführung des Codes und das effiziente Debuggen von Code.

Optimierung von Software-Tests

Zur Optimierung von Softwaretests wird Virtualisierung auch gerne eingesetzt, um mehrere unabhängige Testumgebungen zu erzeugen. Hier kann sowohl auf der Hardware-Ebene Virtualisiert werden, als auch auf der Befehlssatzarchitektur. Der Vorteil bei der Virtualisierung auf der Befehlssatzarchitektur, speziell bei QEMU wäre, dass sich der Zustand der jeweiligen Virtuellen Maschine jederzeit speichern lässt, und z.B. zum Debuggen verwenden lässt. Der Nachteil wäre dabei jedoch, dass die Performance stark beeinträchtigt wird. Diesen Nachteil könnte man durch die Virtualisierung auf der Hardware-Ebene unter Verwendung von VMware Workstation oder XEN reduzieren.

Quellen

1. http://www.kernelthread.com/publications/virtualization/, eingesehen am 18.06.2010
2. http://queue.acm.org/detail.cfm?id=1017000, eingesehen am 18.06.2010
3. http://queue.acm.org/detail.cfm?id=1189289, eingesehen am 18.06.2010
4. Susanta Nanda, Tzi-cker Chiueh "A Survey on Virtualization Technologies" In: http://www.ecsl.cs.sunysb.edu/tr/TR179.pdf , eingesehen am 18.06.2010
5. Gerald J. Popek, Robert P. Goldberg "Formal Requirements for Virtualisable Third Generation Architectures
6. http://www.freebsd.org/doc/en/books/arch-handbook/jail.html, eingesehen am 18.06.2010
7. http://wiki.qemu.org/download/qemu-doc.html, eingesehen am 18.06.2010
8. http://www.usenix.org/events/usenix05/tech/freenix/full_papers/bellard/bellard_html/index.html, eingesehen am 18.06.2010
9. http://www.vmware.com/de/, eingesehen am 18.06.2010
10. http://tx.downloads.xensource.com/downloads/docs/user/, eingesehen am 18.06.2010