Basistechnologie: Virtualisierung

Maximilian Hoecker

Fakultät für Informatik,
Hochschule Mannheim,
Paul-Wittsack-Straße 10,
68163 Mannheim
maximilian.hoecker@stud.hs-mannheim.de

Zusammenfassung Virtualisierung ist ein breites Themengebiet mit dem sich IT-Landschaften vereinfachen lassen. Erstmals verwendet in den 1950er Jahren, sind inzwischen einige Teilgebiete mit verschiedenen Zielen enstanden. Mit den modernsten Virtualisierungstechniken wie Para- und Hypervisorvirtualisierung lassen sich ganze Systeme performant virtualisieren. Insgesamt werden hier die meistgenutzen Konzepte und Techniken der Virtualisierungswelt behandelt.

1 Einleitung

In der heutigen IT-Welt stehen leistungsstarke Rechner in den Rechenzentren und sogar unter jedem Bürotisch. Diese Rechner haben nicht selten mehrere Gigaherz an Prozessorleistung und 4GB Arbeitsspeicher sind fast schon Standard im Desktopbereich. Betrachtet man sich diese Systeme zur Laufzeit stellt man fest, dass die CPU-Rechenzeit im Vergleich zur Laufzeit sehr klein ausfällt. Ähnliches gilt für den Arbeitsspeicher, der selten zu 100% belegt ist. Um nicht nur die Kosten, sondern auch den Administrationsaufwand in einem Rechenzentrum zu senken, gibt es die Möglichkeit der Virtualisierung.

1.1 Definition

Die Definition des Begriffs *Virtualisierung* gibt es nicht. Sucht man in der Literatur findet man dazu diverse Ansätze. Jeder ist an sich nicht falsch aber auch nicht vollständig.
Amit Singh hat im Jahr 2004 den Begriff wie folgt definiert:

> 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]

Eine andere Definition eines anderen Autors lautet:

> Virtualisierung ist die Technik, Betriebsmittel eines Computers so zu repräsentieren, dass sie von Benutzern und Programmen einfach verwendet werden können, ohne dass die genaue Implementation oder physikalische Eigenschaften des Betriebsmittels bekannt sein müssen. [2, S.106]

Beide sind weder falsch, noch aber vollständig. In der ersten Definition fehlt z.B., dass es auch eine Speichervirtualisierung gibt, die man durch Zusammenschalten von Festplatten im RAID erreicht.
Die zweite Definition bezieht z.B. nicht die Virtualisierung von Anwendungen ein. Ein Großteil der Literatur beschäftigt sich bei der Definition der Virtualisierung nur mit der Abstraktion/Simulation/Emulation von Hardware, was aber nicht die ganze Bandbreite des Fachgebiets abdeckt.

2 Arten der Virtualisierung

Betrachtet man die Virtualisierung im Ganzen, kann man sie in 2 große Teilbereiche aufteilen. Zum Einen in die **Hardwarevirtualisierung** und zum Anderen in die **Softwarevirtualisierung**.
Die erste Idee der Virtualisierung kam 1959 von *Christoper Strachey*. Er veröffentlichte die Abhandlung *Time Sharing in Large Fast Computers*. Strachey beschrieb dort das Konzept des *Time Sharings* bei dem die Rechenzeit der CPU auf Anwender und/oder Programme verteilt wurde. Implementiert wurde dies 1962 im Großrechner Atlas, bei dem es einen damals neuen Architekturansatz gab. Es wurde nun separiert zwischen Prozessen des Betriebsssystems und der Programme.[2, S.53]

2.1 Hardwarevirtualisierung

Ein paar Jahre und Entwicklungen später brachte *IBM* das *S/360* System auf den Markt. Dieses System sollte der Nachfolger des sehr erfolgreichen *IBM 7070* Mainframes werden. Durch die Änderung der Architektur des *IBM S/360* Systems war es eigentlich nicht mehr möglich Programme des alten *IBM 7070* Mainframes auf dem neuen System laufen zu lassen. Aus diesem Grund wurde das System hardware- und betriebssystemseitig erweitert. Zur Laufzeit wurden durch eine Software alle Schritte der älteren Programme überwacht. Bei Bedarf wurden alte Befehle statt direkt in die Recheneinheit, in den Übersetzer gesendet. Dieser übersetzte den alten Befehl in einen neuen und leitete diesen dann weiter an die Recheneinheit.
Larry Moss taufte seine Erfindung damals Emulator.[1] Damit war die erste Form der **Emulation** geschaffen, da dem alten Programm vorgegaukelt wurde,

[1] abgeleitet aus dem lateinischen *aemulare*: Nachahmen

es liefe immernoch auf einer *IBM 7070* Maschine.
Eine weitere Form der Hardwarevirtualisierung ist das **Serverpartitioning**. Beim Serverpartitioning/Serverdomaining unterscheidet man zwischen **festen** (Hardware) und **dynamischen** (Software)**Partionen/Domains**.
Bei einem System aus festen Partitionen, sogenannten **LPARs** (**L**ogical **Par**titions), werden die verfügbaren Hardwareresourcen eines Servers in mehrere logische Teile unterteilt. Jeder Teil für sich besitzt eine gewisse Leistung an CPU, Arbeitsspeicher und Festplattenkapazität, die variieren kann. In jeder einzelnen Partition kann ein komplett unabhängiges Betriebssystem laufen, das keine der anderen Partionen kennen muss. Solche Partionen eines Systems können heute z.B. mit dem z-10 Mainframe System von IBM gemacht werden. Die Resourcenverwaltung ist in einer Art BIOS, dem sogenannten **Licensed Internal Code (LIC)**, implementiert.[3, Seite 48]
Eine kostengünstigere Variante des Serverpartitionings ist das Aufteilen des Servers in *dynamische Paritionen*. Dynamische Partionen werden mit Hilfe eines Hypervisor softwareseitig realisiert, der als Minimalbetriebssystem die komplette Hardware einer Maschine verwaltet und zur Laufzeit aufteilt. In diesem Paper wird im Kapitel 3 näher darauf eingegangen. Bei dieser Art der Aufteilungen werden die Partitionen auch *Virtual Machines(VM)* genannt.
In bestimmten Mainframes ist es auch möglich beide Arten der Partitionierung gleichzeitig zu verwenden. In den neueren Systemen können die logischen Partitionen sogar zur Laufzeit angepasst werden, wenn es das Gastbetriebssystem unterstützt. Aktuell sind Mainframes die einzigste Hardwaregruppe, bei der es die Möglichkeit des logischen Serverpartitionierens gibt. Rechnern mit z.B. einer x86 Architektur ist es nicht möglich sich in logische Partitionen teilen zu lassen.

In Abbildung 1 ist ein Beispiel einer Aufteilung eines Servers mit 6 CPU's, Arbeitsspeicher und 3 Fesplatten zu sehen. *Partition 1* hat 2 ganze CPUs, eine ganze Harddisk und ein wenig RAM für sich. *Partition 2* hingegen nur eine CPU und einen Teil einer Platte. Die nächste logische Einheit (*Partition 3*) hat nur einen Teil einer CPU und nur einen Teil einer Platte. Und zuletzt bekommt *Partition 4* 2 CPUs und eine Festplatte. Diese 4. Partition ist zusätzlich noch dynamisch unterteilt in 2 Virtuelle Maschinen.

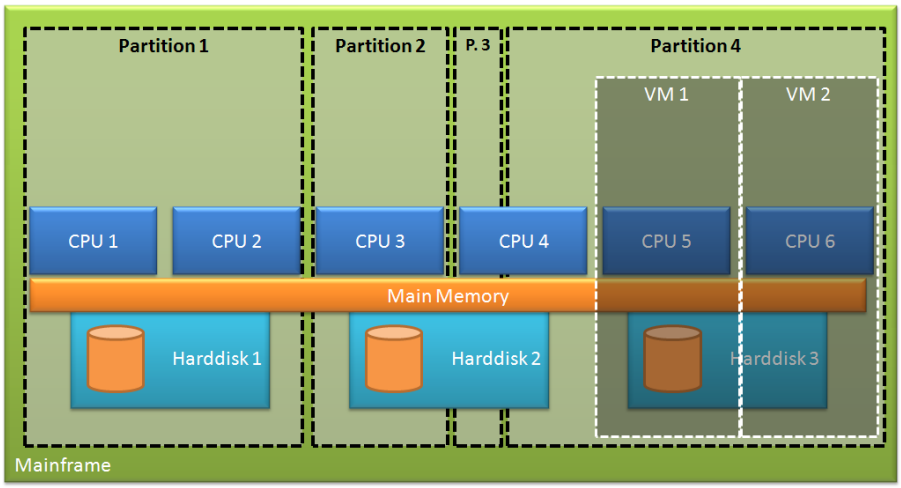


Abbildung 1. Schaubild der Möglichkeiten des Serverpartitionings

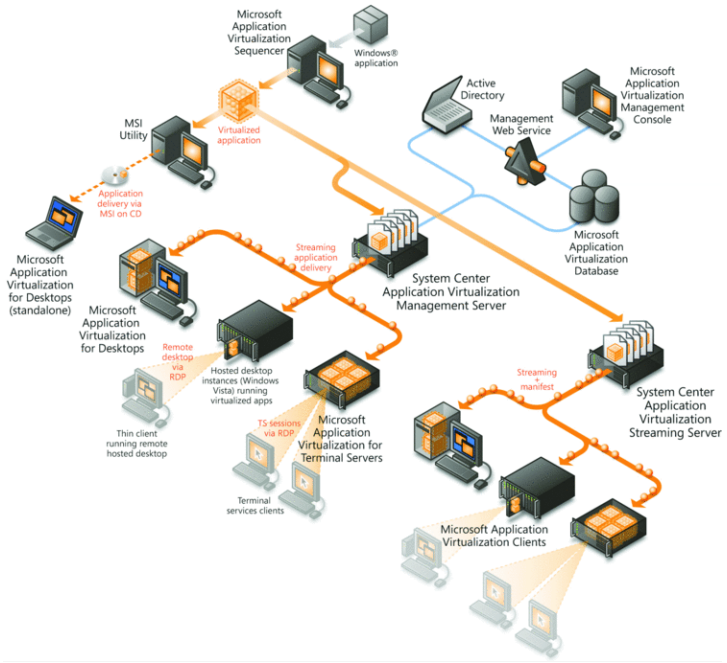
Durch gewisse Mechanismen im Hypervisor ist es auch möglich **Overprovisioning** zu betreiben. Zum Beispiel wird einer der Virtuellen Machines mehr RAM zugewiesen, als eigentlich reell für sie verfügbar wäre. Der Hypervisor weist den virtuellen Maschinen zur Laufzeit die benötigte reelle Menge an RAM zu. Dieser RAM kann auch aus Speicherbereichen anderer virtueller Maschinen gewonnen werden, sofern diese ihn nicht belegt haben. Overprovisioning ist nicht für Produktivumgebungen zu empfehlen, denn falls der RAM einer overprovisioned Machine vollläuft und alle anderen VMs auch 100% ihres Arbeitsspeichers benötigen, kommt es auf den Hypervisor an, ob das System abstürzt. Abstürzen würde er, wenn keine Swap-Möglichkeit bestünde, wie RAM im Hypervisor selbst, oder Platz auf einer Swappartition einer Festplatte.
Zusammengefasst gibt es also 3 Arten der Hardwarevirtualisierung:

- Die Emulation,
- das Logical Partitioning/Domaining und
- die Virtualisierung durch einen Hypervisor

Betrachtet man die 2 Techniken Emulation und Virtualisierung, denkt man auf den ersten Blick, dass diese sich sehr ähnlich sind. Dennoch liegt der Hauptunterschied zwischen beiden darin, dass bei der Virtualisierung von Hardware die abgebildete Architektur in der VM nicht abweichen darf von der des Host-Systems. Bei der Emulation hingegen, darf die Virtuelle Maschine bzw. das Emulierte System eine völlig andere Architektur besitzten als die Hostmaschine.
Ein Vorteil der Virtualisierung gegenüber der Emulation, ist der direkte aber überwachte Zugriff auf Hardware, der vergleichsweise schneller ist. Ein weiterer Vorteil ist, dass man durch die gleiche Architektur weniger Kompatibilitätsprobleme zwischen Host und VM hat.[2, S.107/108]

2.2 Softwarevirtualisierung

Man kann nicht nur Hardware virtualisieren, sondern auch Software. Allgemein bedeutet das Virtualisieren von Software, dass man eine Software unverändert auf einem anderen Betriebssystem laufen lassen kann. Man virtualisiert sozusagen die Schnittstelle zum Betriebssystem. Eine andere Form der Softwarevirtualisierung ist das Emulieren von Treibern.
Besteht eine homogene IT-Landschaft z.B. nur aus Linux Systemen würde es sich nicht lohnen wegen einem Programm, dass nur auf Windows läuft, auf Windows umzusteigen.
WINE (**W**ine **I**s **N**ot an **E**mulator) bietet hier eine Abhilfe. WINE kann man als eine Art Applikationswrapper sehen, der sich um ein Windows Programm legt, damit es in einem Linux System laufen kann. Dieser Wrapper leitet alle Aufrufe die zum Windows-Betriebssystemkernel gehen sollen um. Das Ziel einer jeden Umleitung ist die äquivalente Funktion des Linuxkernels zum Windowskernel. Im Gegensatz zu einem Emulator wandelt WINE keine Aufrufe um. Das heisst, es werden keine Daten konvertiert oder anders speziell vorbereitet, damit sie ausgeführt werden können. Dadurch gibt es automatisch eine Einschränkung. Durch die fehlende Emulation muss der Hostkernel ein x86er Linux Kernel sein, damit die Ergebnisse der Systemaufrufe die selben Resultate liefern.[2, S.80]
Eine ähnliche Anwendung wurde ebenfalls 1993 veröffentlicht. Diese Anwendung namens **WABI** **W**indows **A**pplication **B**inary **I**nterface kam von der Firma SUN Microsystems. **WABI** fungiert als ein Emulator, bei dem Windows Programm unverändert auf Solaris laufen. Dabei ist es egal, ob der Host Rechner eine SPARC oder eine x86er Architektur darunterliegen hat. Mit der Emulation wird jeder Befehl übersetzt in die darunterliegende Architektur.
Vergleicht man beide Ansätze ist der WABI Ansatz der flexibelere, aber auch der langsamere. Die Flexibilität ist gegeben dadurch, dass man jeden Befehl der an einen Kernel gehen soll, emuliert werden kann. Ebenfalls flexibel ist WABI, da man der Host eben ein SPARC oder ein x86er sein kann. Theoretisch wären auch noch alle anderen Plattformen möglich, die Solaris unterstüzt. Langsamer ist WABI, da es eben erst alles übersetzen muss, bevor es wirklich berechnet wird.
Microsoft hat für ihre Systemlandschaft einen weiteren Ansatz, Anwendungen zu virtualisieren. Die Grundidee hinter diesem Virtualisierungsansatz ist, dass es in einer Systemlandschaft oft Kompatibilitätsprobleme gibt zwischen Programmen. Desweiteren soll die Softwareverteilung in einer Systemlandschaft vereinfacht werden. Dabei soll es unrelevant sein, ob eine alte Version bereits auf einem Clientrechner installiert ist oder nicht. Microsoft hat das Grundprinzip von **Microsoft App-V** in einem Schaubild (Abbildung 2) dargestellt.



[5]

Abbildung 2. Architektur und Funktionsweise von Microsoft App-V

Bevor die virtualisierten Anwendungen überhaupt in die Landschaft gelangen können, müssen diese erstmal zu virtualisierten Paketen verarbeitet werden. Eine Anwendung wird also in ein virtuelles Paket gepackt. Dieses Paket beinhaltet alle Informationen zu dem Programm, sowie alle Dateien, die das Programm benötigt. Dieses ganze Paket wird auf einem Rechner, der als **Application Virtualization Sequencer** fungiert, geschnürt. Auf diesem ist ein spezielles Programm vorhanden, das bei der Installation des zu virtualisierenden Programms alle Veränderungen auf dem System protokolliert. Mit den daraus gewonnenen Informationen wird das Paket gebastelt. Mit diesem Paket ist es möglich die Anwendung entweder ganz normal auf einem Clientrechner zu installieren[2], oder es virtualisiert laufen zu lassen. Nimmt man den emfohlenen Ansatz es virtualisiert laufen zu lassen, müssen die Clientrechner erstmal mit dem App-V Zusatzprogramm bestückt werden. Dieses erlaubt das Laufen des virtualisierten Pakets in einer Sandbox. Damit man nicht jedem auf Client einzeln und manuell die Pakete deployen muss, gibt es einen zentralen Server. Dieser nennt sich **System Center Application Virtualization Management Server** und wird in einer *Active Directory* Domäne eingebunden. Falls in der Landschaft kein Active Directory exisitiert, gibt es ein Pendant, das sozusagen Standalone läuft, names **System Center Application Virtualization Streaming Server**. Auf diesen Servern werden die Pakete abgelegt und Clients können sich diese Runterladen, falls die

[2] Als hätte man es mit dem Standardinstaller installiert

Administratoren die Programme für die Anwender freigeschalten haben. Das Herunterladen passiert ebenfalls automatisch. Der Anwender merkt nicht, dass die Programme nicht bei ihm installiert sind. Startet ein Anwender ein virtualisertes Programm, wird das Programm per RTSP oder HTTP Protokoll heruntergeladen und lokal ausgeführt. Nach dem Beenden des Programm bleiben bis auf eventuell gespeicherte Benutzerdateien oder das gecachte Paket nicht auf dem Clientrechner zurück.

Eine weitere Anwendung der Softwarevirtualisierung ist die Virtualisierung von Peripherie-Geräten. Es ist möglich, durch das Schreiben eines speziellen Treibers, Geräte zu simulieren. Dieser Treiber greift statt auf Hardwareadressen auf ein Programm zu, dass sich genauso verhält wie das virtualiserte Gerät. Virtualisiert werden können sowohl blockorientierte als auch zeichenorienterte Geräte. Beispiele dafür sind: Virtuelle CD Laufwerke, Drucker Spooler, iSCSI und rCAPI.

3 Virtualisierung von Betriebssystemen

Den Begriff *Virtualisierung* findet man in der Literatur überwiegend im Zusammenhang mit der Virtualisierung von Betriebssystemen. Die Virtualisierung auf Betriebssystemebene bietet viele Vorteile gegenüber anderen Formen der Virtualisierung. Allgemein muss man sich aber vor Augen halten, dass ein Betriebssystem immer versucht auf Hardware zuzugreifen, egal ob das Betriebssystem virtualisiert ist oder nicht. Ebenso versucht es auf Peripheriegeräte zuzugreifen, unabhängig ob es nativ oder virtualisiert eingesetzt wird.
In erster Linie muss man sich also Gedanken machen, wie man Betriebssysteme virtualisiert, ohne das die Programme etwas davon wissen. Die Funktionsweise der Datenübertragung zwischen Hardware,Kernel und Programmen kann man sich am besten anhand des Ringschutzkonzepts der x86er Architektur erklären. Dieses Konzept hat 4 Ringe, die ineinander geschachtelt sind.

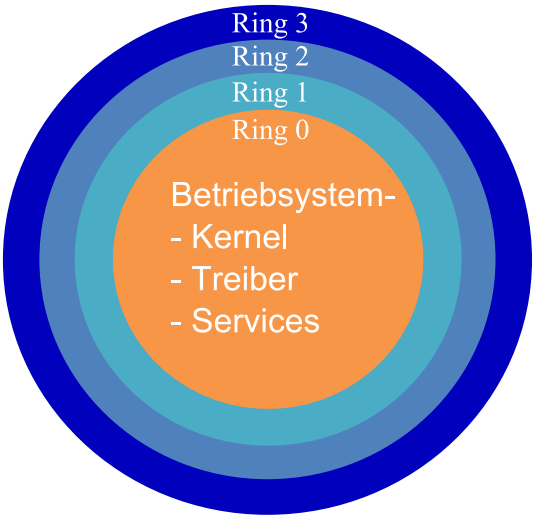


Abbildung 3. Aufbau der CPU Ringe in der x86 Architektur

Jeder Ring ist isoliert von Anderen und hat lediglich eine Schnittstelle zum nächst Inneren und Äußeren. Bei dieser Darstellung der Ringe gilt: je innerer ein Ring ist, desto mehr Privilegien hat er. Das heisst, der innerste Ring (*Ring 0* auch *Kernel-Space* genannt) hat die meisten Privilegien, um auf Hardware zuzugreifen gegenüber den Anderen. In diesem Ring laufen alle Teile eines Betriebssystems, die hardwarenah arbeiten müssen (z.B. Kernel oder Treiber). Der äußerste Ring, also *Ring 3* (auch *User-Space* genannt) hat die wenigsten Privilegien, um auf Hardware zuzugreifen. Im 3. Ring laufen alle Prozesse des Benutzers. Die Ringe 1 und 2 sind in gängigen Betriebssystemen Linux und Windows unbenutzt.
Ein Prozess kann sich bei diesem Konzept nicht selbst in einen höher privilegierten Ring bringen, er kann aber eine Operation eines höher privilierteren Rings nutzen. Versucht er dies, löst der Prozessor eine Exception aus, die abgefangen werden muss. Wird diese Exception nicht abgefangen, stürzt der Prozess ab. Würde eine Exception im Kernel nicht abgefangen werden, bedeutet dies einen Systemabsturz. [4]

3.1 User Mode Linux (UML) im TT Mode

User Mode Linux ist eine Schnittstelle im **architekturunabhängigen Teil** des Linux Kernels. Diese wurde 1999 von Jeff Dike mit einem Patch realisiert. Damals ermöglichte es dieser Patch erstmals, den Kernel selbst als unpriviligierten Prozess zu starten (**User-Mode-Kernel**, s. Abbildung 4). Das bedeutet technisch, dass man mehrere Instanzen des Kernels bzw. des Betriebssystems am laufen hat. Startet man ein Programm auf einem User-Mode-Kernel, werden alle Aufrufe des Programms vom Host Kernel angenommen und bedarfsweise an den Gast-Kernel durchgereicht.[6, Seite 15]

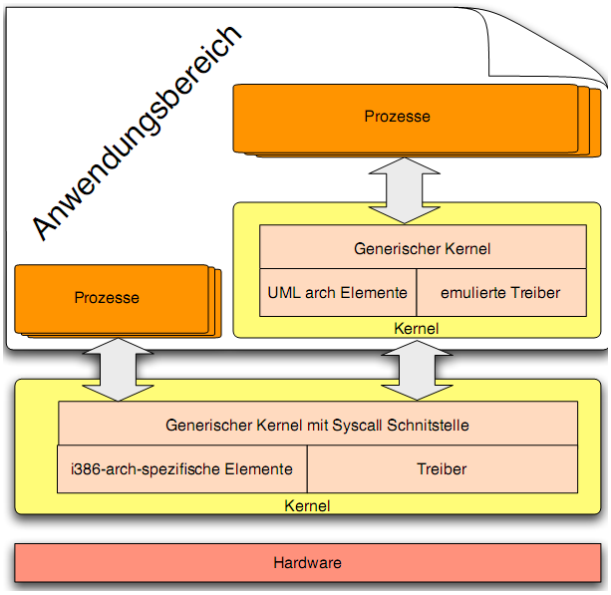


Abbildung 4. Architektur eines UML Systems

Wie in Abbildung 4 zu sehen, laufen die Prozesse einer UML-Instanz auch im Anwendungsspeicher und können somit normalerweise nicht unterschieden werden von Programmen des Hosts. Deswegen wird jeder Prozess eines Gasts als reeller Prozess auf dem Host angelegt.
Damit der Host Kernel weiss aus welchem UML-Kernel welcher Aufruf kommt, gibt es zu jeder UML-Kernel-Instanz einen **Tracing Thread (TT)**. Dieser Thread nimmt alle Aufrufe der Prozesse des UML-Kernels entgegen. Die Aufrufe werden an den Host- und Gastkernel weitergereicht. Bevor ein Aufruf den Hostkernel erreicht, wird der Inhalt des Aufrufs überprüft. Ist der Aufruf ein hardwareunabhängiger, wird dieser mit einem nutzlosen *getpid()* Aufruf 'nullifiziert'[3].[6]

3.2 User Mode Linux (UML) mit SKAS Patch

Ein Resultat aus dem *Tracing Thread Mode* sind *Performanceprobleme* durch die *Prozesskopplung*. Da zu jedem Prozess im Gast ein Hostprozess angelegt wird, hat der Hostkernel schnell sehr viele Prozesse zu verwalten. Ein weiterer Grund für Performancerprobleme sind die *Kontextwechsel*. Muss in einem Gastsystem ein Kontextwechsel vollzogen werden, sind das 4 reele Kontextwechsel(Syscall <-> TracingThread <-> Hostkernel). Dies ist besonders bemerkbar bei I/O-lastigen Anwendungen.

[3] Der Aufruf muss überschrieben werden, da er für einen undeaktivierbaren Debughandler im Kernel vorhanden sein muss

Im TT-Mode fehlt unteranderem die Speichervirtualisierung, dadurch läuft jeder Prozess im Speicherbereich des Hostkernels. Bricht nun ein Prozess aus seinem eigentlichen Speicherbereich aus, kann der Prozess andere Speicherbereiche attackieren und überschreiben.
Die Lösung dieser Probleme stellt der **S**eperate **K**ernel **A**ddress **S**pace(**SKAS**) Patch dar, der auf dem Hostkernel implementiert wird.

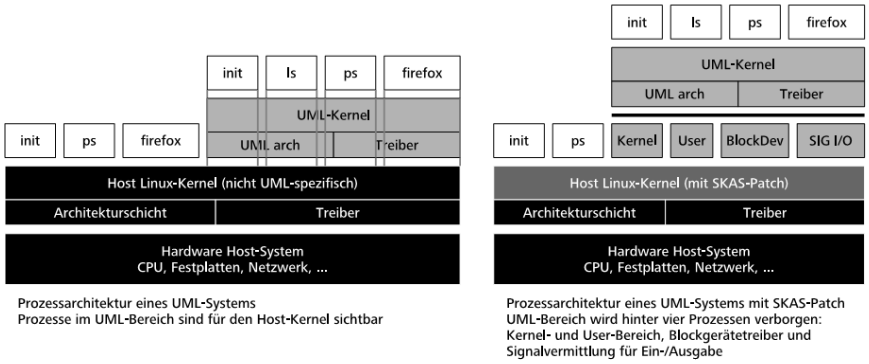


Abbildung 5. Architekturvergleich der UML Versionen

In Abbildung 5 ist ein Architekturvergleich zwischen einem UML System mit und ohne den SKAS Patch zu sehen. Das System mit dem SKAS Patch bzw. dessen Speichervirtualisierung weist nun jedem UML einen eigenen Speicherbereich zu, aus dem es nicht mehr ausbrechen kann. Diesen Schutz übernimmt das Memory Management des Host Kernels. Zusätzlich werden die 4 Kontextwechsel durch eine modifizierte *ptrace()* Funktion (Prozesstracing) im Kernel, auf 2 reduziert. [8]Insgesamt ist das System ca. doppelt so schnell (gemessen bei Kernel-Compiling) [4].
Da die Prozesse eines Gasts nun einen eigenen Speicherbereich haben und nicht mehr auf dem Host laufen, hat der Host nur noch 4 Prozesse zur Verwaltung der Gäste[7]:

- UML Kernel Thread, der Code im UML Kernelspace ausführt
- UML Userspace Thread, der Code im UML Userspace ausführt
- Asyncroner I/O Treiber Thread, ein Thread für die Verwaltung der virtuellen Block Devices
- Emulator-Thread für WRITE-SIGIO, ein Hilfsprozess, der Signale für *write()* calls entgegennimmt und verarbeitet

3.3 Emulation auf Applikationsebene

User-Mode-Linux ist performant, aber nicht sehr flexibel. Die Version des Gast und Hostkernels müssen übereinstimmen, da sonst die API Kompatibilität nicht

[4] Messung von: $http://user-mode-linux.sourceforge.net/old/skas.html$

garantiert werden kann. Eine neue Lösung veröffentlichte 1999 VMware mit ihrer Software *VMware Workstation*. Mit der Software war es zum ersten Mal möglich einen kompletten *x86 Rechner* auf einem x86er Host zu virtualisieren. Dabei war neu, dass der virtualisierte Rechner seine eigene virtuelle Hardware hatte, inklusive eines eigenen BIOS. Das Betriebssystem ist, wie bei vorhergehenden Lösungen auch, im Userspeicher platziert.

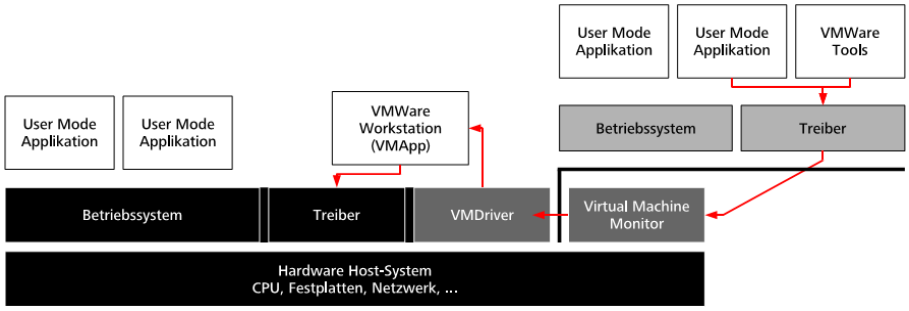


Abbildung 6. Architektur von VMware

VMware führte ebenfalls den **V**irtual **M**achine **M**onitor(**VMM**) ein. Der VMM ist ein Prozess überhalb des Betriebssystems, der hauptsächlich 2 Aufgaben hat. Einerseits verteilt er die vorhandenen und zugewiesenen Hardwareresourcen an die VMs, andererseits scannt er den ausführenden Code vor der Ausführung per **Scan-before-Execution** Verfahren. Dieses Verfahren (auch *PreScan* genannt) überprüft parallel zur Laufzeit den Programmcode der VM **bevor** dieser ausgeführt wird. Tritt dabei ein abzufangender Befehl auf, wird dort ein Breakpoint gesetzt und die Suche endet erstmal. Existieren konditionale Sprünge, werden beide Möglichkeiten verfolgt (bis eine maximale Suchtiefe erreicht ist). Die abgefangenen Befehle werden dann direkt an die äquivalenten Stellen im Betriebssystemkernel des Hosts geleitet. Falls etwas nicht gescannt werden konnte, wird der Rest der Sprünge emuliert.
Die VMware Workstation Architektur besteht unter anderem noch aus dem **VMDriver**, der als Modul im Hostsystem geladen wird. Er dient als Kommunikationsschnittstelle für den I/O Verkehr zwischen Gast <-> VMApp. Als weitere Aufgabe hat der VMDriver das Speichermanagement, also den reellen Arbeitsspeicher an den VMM weiterzuleiten, sowie das Verwalten der virtuellen Geräte.
Der letzte wichtige Architekturteil ist das **VMApp**. Dieses dient als Oberfläche für den Endanwender sowie Kommunikationsschnittstelle zwischen Host- und VM-Treiber. Das VMApp stellt also folgenden Kommunikationsweg her: Hosttreiber <-> VMApp <-> VMXDriver.

3.4 Para-Virtualized-Machines

Der Ansatz von VMware ist ein flexibler, aber kein performanter Ansatz. Das Scan-before-Execution kostet viel Rechenzeit und auf dem Hostbetriebssystem müssen viele zusätzliche Anwendungen und Dienste installiert werden. Desweiteren ist diese Art der Virtualisierung nur eine Art modifizierte Emulation.
Ein anderer Ansatz ist das **Para-Virtualisieren**. Hierbei wird ein ebenfall kompletter x86er Rechner virtualisiert.Es gibt einen **Hypervisor** und einen **modifizierten Gastkernel**. Dieser läuft nicht im Userspace, auch nicht im Kernelspace des Hypervisors, sondern in einem komplett eigenen Speicherbereich.
Eine zentrale Rolle bei dieser Virtualisierung spielt der *Hypervisor*. Er ist im Grunde ein Virtual Machine Monitor, der aus einem minimalen Betriebssystem besteht, das im Ring 0 des CPUs arbeitet. Die Aufgaben sind, wie bei der VMM von VMware auch, die Speicherverwaltung, das Scheduling und die Geräteverwaltung. Im Gegensatz zu der VMM von VMware, weist der Hypervisor von **XEN** die Hardwareresourcen nicht über spezielle Treiber zu, sondern direkt an den Gastkernel. Der Hypervisor von XEN emuliert im Paravirtualisierungsmodus keine Resourcen, aber Geräte (Netzwerkkarte, Festplatten, usw.). Die Resourcen (z.B. RAM) werden über eine Anwendungsschnittstelle des Hypervisors angesprochen. [4]

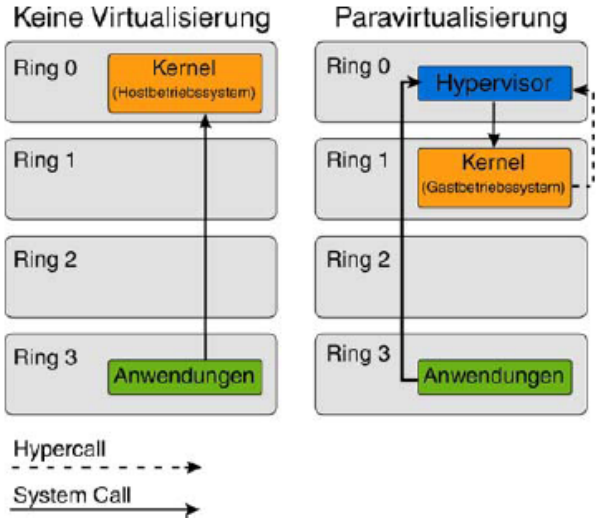


Abbildung 7. Übersicht der CPU Ringe

Da dem Gastkernel, wie Abbildung 7 gezeigt, der Platz im 0. Ring fehlt, weicht der Kernel aus in den *Ring 1*. Wie bereits erwähnt, muss der Gastkernel modifiziert sein. Der Grund dafür ist, dass der Gastkernel in Ring 1 keine Privilegien gegenüber der Hardware hat. Er muss diese über den Hypervisor mittels **Hypercalls** ansprechen. Diese Hypercalls werden syncron getätigt und dann im

Hypervisor zu einem richtigen Systemcall ungewandelt um die unvirtualisierten Resourcen anzusprechen.

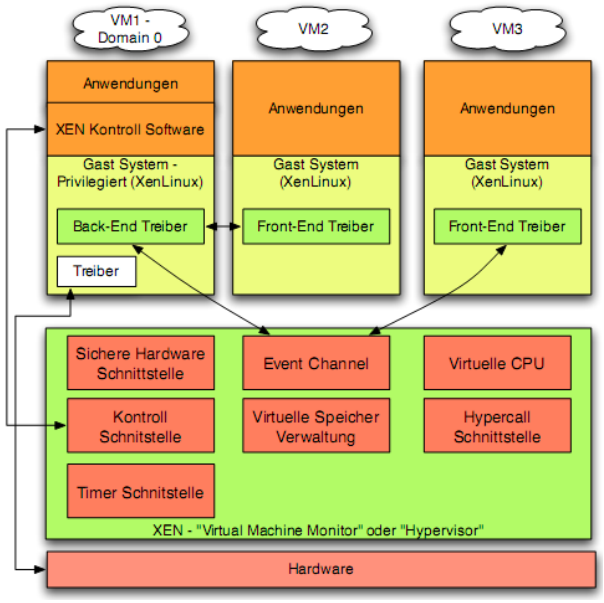


Abbildung 8. Architektur von XEN

In Abbildung 8 erkennt man, dass sich im Hypervisor ausschließlich Programme und Dienste finden, die zur Verwaltung der VMs und der Resourcen dienen. So bietet er verschiedene Schnittstellen an. Unter anderem den **Event Channel**, der das Pendant zu Hardwareinterrupts in einem unvirtualisierten System darstellt. Möchte eine VM (auch **Domain** genannt) also bei einem Hardwareinterrupt benachrichtigt werden (z.B. Timer) kann sie sich darauf registrieren. Implementiert ist das Ganze in Form eines Bitarrays. Tritt ein bestimmtes Event auf, setzt der Hypervisor ein bestimmtes Bit im Bitarray auf 1 ein. Wenn die Domain nach einem Kontextwechsel wieder zum Zug kommt, überprüft diese das Bit/Event. Die Domain kann nun auf das Event reagieren und setzt danach das Bit zurück auf 0. Die 'Registrierung' auf ein bestimmtes Event erfolgt ebenfalls in einem Bitarray.
Ein weiterer wichtiger Part ist die **Kontrollschnittstelle**. Da der Anwender nie im Speicherbereich des Hypervisors arbeitet, kann er diesen nur über die Kontrollschnittstelle steuern. Damit aber nicht jede Domain diese Schnittstelle ansprechen kann, gibt es eine Domain (**Dom0**) die mehr Privilegien hat, als die anderen Domains *(DomU's)*.
Die Dom0 wird beim Systemstart automatisch mitgestartet und ist die Konsole des Anwenders. Da der Hypervisor nur den Arbeitsspeicher und die CPU verwaltet, aber nicht die Geräte wie z.B. Festplatte und Netzwerkkarte, darf die Dom0 direkt auf Hardwaregeräte zugreifen. Dazu kann man in der Dom0 die

normalen Treiber der Geräte verwenden.
Die Domains selbst benötigen natürlich auch Festplatten, Netzwerkgeräte und Grafikkarten, was durch *virtuelle Devices* implementiert ist. Dazu existieren spezielle Treiber, die in 2 Teile geteilt sind. Einerseits einen **Frontend Treiber**, der auf den DomUs benötigt wird, und andererseits den **Backend Treiber**, der nur in der Dom0 existiert. Wenn ein Gerät angesprochen werden soll, ist der Kommunikationsweg Frontend-Treiber<->Backend-Treiber<->Treiber in Dom0<->Hardware. Die Hardwareinterrupts der Geräte sind mit dem *Event Channel* implementiert.

3.5 Hypervisor-Virtualized-Machines

Paravirtualized Machines sind zwar schnell, aber man muss immernoch eine Modifikation am Betriebssystem machen, also am Kernel, damit das Ganze funktioniert. Will man das, sowie die Nutzung von VMware Workstation vermeiden, muss man zu *Hypervisor-Virtualized-Machines* zurückgreifen.
Hypervisor-Virtualized-Machines funktionieren genauso wie Paravirtualized Machines mit einem Hypervisor. Der gravierende Unterschied ist nur, dass das Betriebssystem so wie bei VMware komplett unmodifiziert bleibt. Die einzigste Vorraussetzung dabei ist, dass der Prozessor de Hostrechner entweder den Intel **V**irtual **M**achine E**x**tenstion (VMX) oder den AMD **S**ecure **V**irtual **M**achine (SVM) Befehlssatz unterstützt.
Mit diese Technologien hat der Prozessor nun 2 Betriebsmodi und 5 CPU Ringe (-1,0,1,2,3) , wobei einer (*Ring -1*) ausschließlich für den Hypervisor reserviert ist. Der innerste Ring ist nun der Ring -1, der nun alle Systemcalls zur Hardware aus Ring 0 übernimmt. Die zwei angesprochenen Betriebsmodi sind zum einen der **VMX-Root-Modus** und zum anderen der **VMX-Non-Root-Modus**. Im VMX-Root-Modus rechnet der Hypervisor im Ring -1 und im VMX-Non-Root-Modus rechnet das Gast-OS in den Ringen 0-3. Das Ausweichen des Hypervisors in Ring -1 ist damit zu erklären, dass sich im Ring 0 der Betriebsystemkernel des Gasts einlagert und somit kein Platz wäre für den Hypervisor.

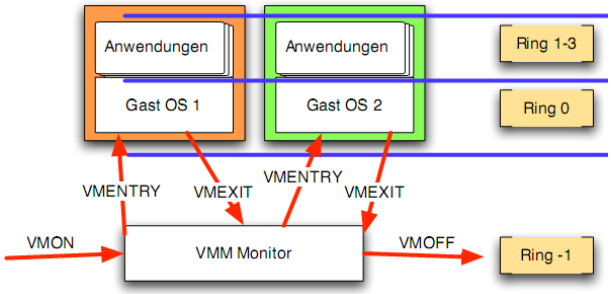


Abbildung 9. Beispiel eines CPU Betriebsmodiverlaufs

Durch den erweiterten Befehlssatz erhält der Prozessor, wie in Abbildung 9 zu sehen, 5 neue Befehle, die zur Steuerung dienen:

- *VMON*: CPU in Virtualisierungsmodus schicken
- *VMENTRY* (von VMM aus): Übergabe der Ringe(0-3) an Gast
- *VMEXIT* (von VMM aus): Abgabe der Ringe an VMM
- *VMOFF*: CPU aus Virtualisierungsmodus holen

Durch diese Befehle kann eine eine VM nicht mehr unterscheiden, ob sie nun nativ oder virtualisiert läuft (Ring 0 ist schließlich immer verfügbar für sie). Dadurch gibt es, bis auf die Kontextwechsel zwischen den VMs, keine Performanceverluste.
Leider gibt es es Unterschiede zwischen beiden Befehlssätzen der CPU-Hersteller. AMD's Pacifica virtualisiert ebenfalls den bei Intel hardwaretechnisch gelösten Speichercontroller in Software. Desweiteren hat Pacifica ebenfalls einen **D**evice **E**xclusion **V**ector (DEV) integriert, der es VMs ermöglicht auch Geräte zu benutzen, die ohne CPU auf Speicher zugreifen können.

4 Zusammfassung und Ausblick

Mit den beschriebenen Technologien lassen sich sowohl viele Probleme in einer IT-Landschaft lösen, als auch Kosten einsparen. Viele dieser Technologien, wie z.B. XEN, sind schon länger auf dem Markt und werden verstärkt im Cloud-Computing Umfeld eingesetzt(Siehe EC2, GoGrid etc.). Aktuell gibt es einen Konkurrenten von XEN namens KVM [10], der sich voll auf Hypervisor-Virtualized-Machines konzentriert und alle anderen Techniken der Betriebssystemvirtualiserung außer acht lässt. Die Linux-Community begrüßt diesen vereinfachten Ansatz, in dem momentan mehrere Disitributionen XEN nicht mehr als Startpaket anbieten. In der Zukunft wird XEN immer mehr vom Markt verdrängt werden und KVM wird XEN ersetzen.

Literatur

1. *An Introduction to Virtualization.* Amit Singh. kernelthread.com. Jan. 2004 `http://www.kernelthread.com/publications/virtualization/`
2. Fischer, Marcus: *XEN Das Umfassende Handbuch.* Galileo Computing. 2009
3. Joachim von Buttlar, Wilhelm G. Spruth *Virtuelle Maschinen: zSeries- und S/390-Partitionierung*, in *Informatik - Forschung und Entwicklung* Volume 19, Number 1 / Juli 2004.
4. Christian Baun, Marcel Kunze, Thomas Ludwig: *Servervirtualisierung Informatik-Spektrum* Volume 32, Number 3 / Juni 2009 S.197-206
5. Microsoft App-V 2009 `http://i.technet.microsoft.com/cc904189.fig01_L.gif`
6. Christian Kern *Virtualisierungstechnologien* 22.07.2005 `http://www.lrr.in.tum.de/~stodden/teaching/sem/virt/ss06/doc/virt06-07-20060531-kern-doc\%20-\%20Paravirtualisierung.pdf`

7. *UML Seminar* Tim Keupen 25.11.2005
http://www.uni-koblenz.de/~vnuml/docs/vnuml/uml.pdf
8. Gerstel, Markus: *Virtualisierungsansätze mit Schwerpunkt Xen* 10/2005
http://markus-gerstel.de/files/2005-Xen.pdf
9. Sesser, Florian: *UML und OpenVZ* 10.07.2007
http://atmvs1.informatik.tu-muenchen.de/dokumente/lehre/seminare/ss07-hsvirtualisierung/sesser_ausarbeitung.pdf
10. Kernel Based Virtual Machine. http://www.linux-kvm.org
11. Xen Hypervisor. http://www.xen.org