Virtualisierung in Betriebssystemen

Hans Jürgen Ohlbach

27. November 2017

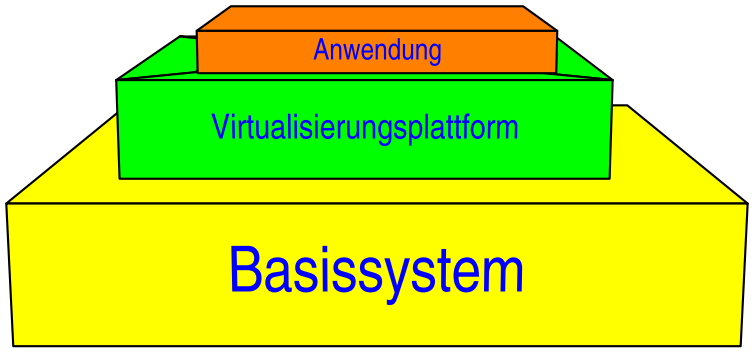
Keywords: Virtualisierung, Paravirtualisierung, Emulation, Partitionierung, API-Emulatoren, Dynamisches Recompilieren

Empfohlene Vorkenntnisse: Betriebssysteme

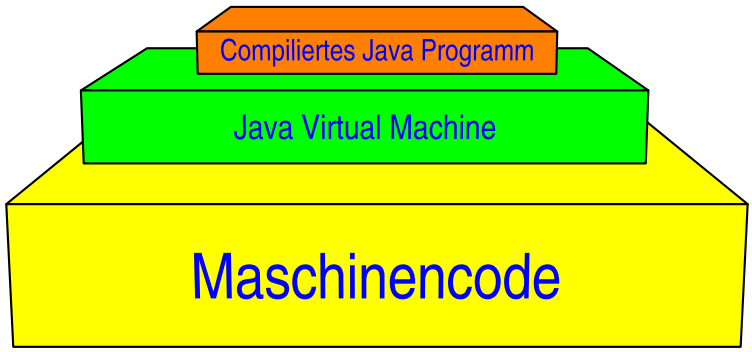
Inhaltsverzeichnis

1 Einführung

Die Grundidee bei der Virtualisierung ist folgende: Man hat zunächst ein Basissystem, z.B. einen konkreten Prozessor, ein Netzwerk, oder irgend ein anderes technisches System. Über dieses Basissystem legt man eine sog. *Virtualisierungsplattform*, die einer Anwendung ein ganz anderes System „vorgaukelt“. Die Anwendung arbeitet also nicht mit dem Basissystem, so wie es ist, sondern agiert so, als ob sie in einer ganz anderen Umgebung eingebettet wäre.



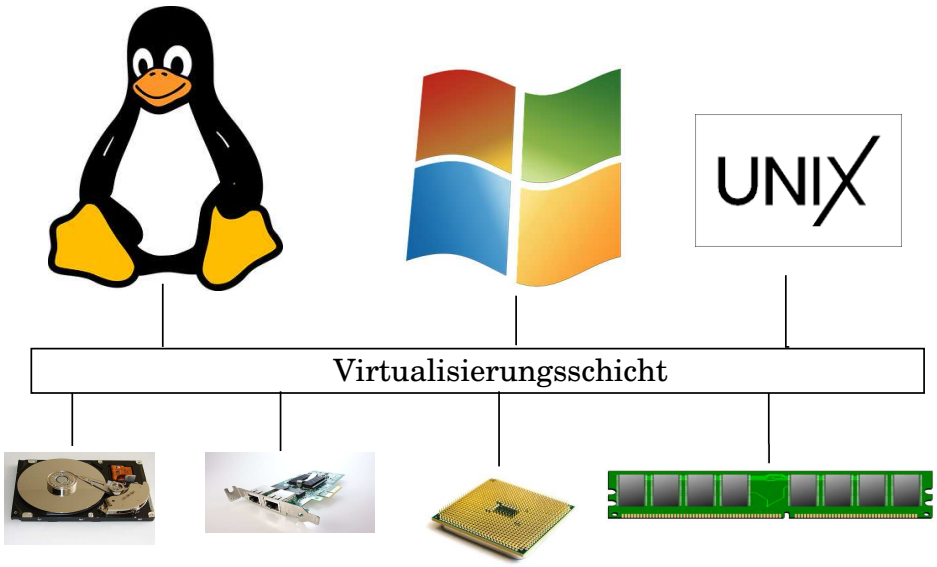
Ein bekanntes Beispiel ist die *Java Virtual Machine* (JVM). Sie lässt einen üblichen Prozessor aussehen wie einen Java Byte Code Prozessor. Die Java Virtual Machine bewirkt, dass jeder Befehl des Java Byte Codes direkt als Folge von Maschinenbefehlen des darunterliegenden Prozessors ausgeführt wird. Ohne ein compiliertes Programm zu ändern, kann man dann das Programm auf ganz verschiedenen Prozessoren ausführen. Man braucht nur die für den jeweiligen Maschinencode angepasste JVM.



Virtualisierung ist also nicht eine Übersetzung (Compilation), wo eine Quellsprache in eine Zielsprache übersetzt wird, sondern die Anwendung wird unmittelbar ausgeführt, allerdings in einer simulierten Umgebung. Bei Programmiersystemen kennt man das schon lange in Form von *Interpretierern*. Ein Interpretierer für z.B. die Programmiersprache Perl führt den Perl-Code unmittelbar aus, ohne Übersetzung. Der Perl-Interpretierer schafft also für das Perl Programm eine virtuelle Perl Umgebung.

Genau genommen ist ein Betriebssystem selbst auch eine Virtualisierungsplattform, nämlich für die Prozesse, die in ihm laufen. Das Betriebssystem gaukelt den Prozessen vor, dass der gesamte Rechner ganz alleine ihnen gehört.

In diesem Miniskript geht es allerdings in erste Linie um die Virtualisierung von Betriebssystemen selbst. Dabei möchte man mehrere Betriebssysteme echt parallel auf einem Rechner laufen lassen, ohne sich beim Booten des Rechners für eines davon entscheiden zu müssen.



Warum sollte man so etwas wollen? Es gibt eine ganze Reihe von Gründen:

- Testen eines Programms für verschiedene Betriebssysteme,
- Testen neuer oder möglicherweise fehlerhafter Programme, die das Betriebssystem zerstören könnten (Schulungsumgebungen),
- Einführung zusätzlicher Dienste, die eventuell ein laufendes System stören könnten,
- Dienste, die alleine einen Rechner nicht auslasten,
- Abschottung verschiedener Dienste gegeneinander
  (Hacker können dann nur eines davon angreifen),
- jedem Benutzer die Möglichkeit schaffen, seine ganz individuelle Umgebung zu konfigurieren,
- Programme weiter zu nutzen, die nur auf veralteter oder nicht verfügbarer Hardware laufen.

Es gibt ganz unterschiedliche Ansätze zur Virtualisierung:

Paravirtualisierung: ein Rechner wird in mehrere virtuelle Maschinen aufgeteilt, in denen verschiedene Betriebssysteme (für dieselbe Architektur) laufen.

Emulation: simulieren einen kompletten Rechner per Software (Beispiel: Java)

Partitionierung: ein Betriebssystemkern läuft nur einmal, stellt aber mehrere unabhängige Laufzeitumgebungen zur Verfügung.

API-Emulatoren: ein Betriebssystem A stellt die Bibliotheken eines Betriebssystems B zur Verfügung, so dass Programme von B laufen können in A.

Dynamisches Recompilieren: (Binary Translation) Zur Laufzeit wird der Code Block für Block in Code der Zielmaschine übersetzt.

In den folgenden Kapiteln sehen wir uns die einzelnen Techniken etwas genauer an.

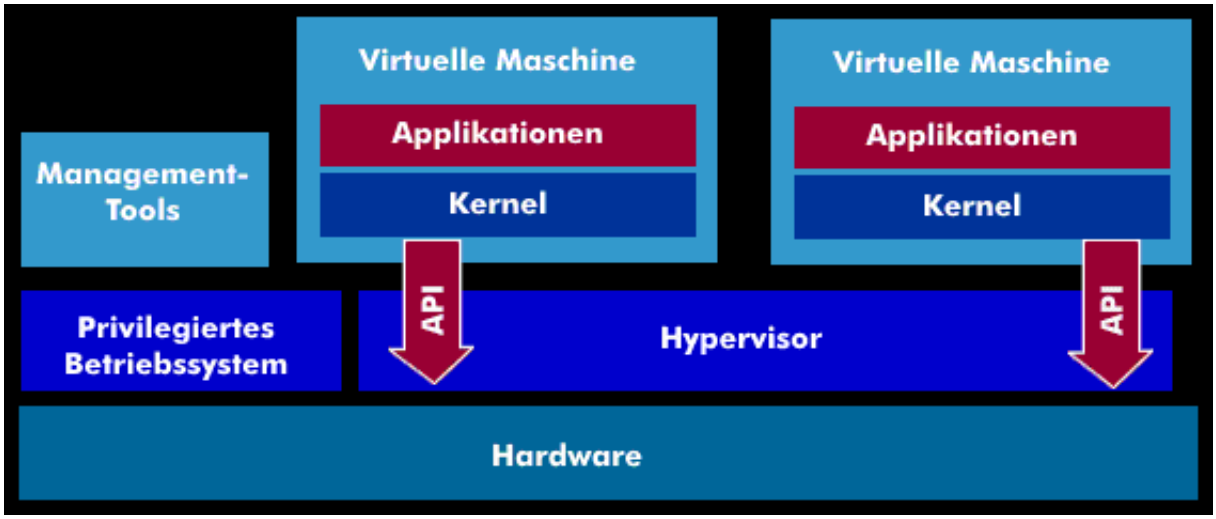
2 Paravirtualisierung

Die Simulation einer komplett anderen Umgebung durch eine Virtualisierungsplattform bedeutet oft einen erheblichen Performanzverlust. Das hat jeder schon mal erlebt, der gleichzeitig mit Interpretierern und Compilern gearbeitet hat. Es gibt aber Situationen, wo das eigentlich nicht nötig wäre.

Als Beispiel, betrachten wir Windows und Linux, die *für dieselbe* Hardware compiliert sind. D.h. man kann alternativ eines von beiden booten, und sie funktionieren problemlos. Ein compiliertes Programm in Windows oder Linux benutzt dann *denselben Maschinencode*. Um die Maschinenbefehle auszuführen, braucht man daher keine Zwischenschicht. Der Unterschied sind die Systemaufrufe, um z.B. einen File zu öffnen, mehr Arbeitsspeicher anzufordern, Sockets einzurichten usw. Diese werden von den verschiedenen Betriebssystemen verschieden verarbeitet. Um *zur Laufzeit* zwischen verschiedenen Betriebssystemen hin- und herzuschalten, müsste man also „nur“ diese Systemaufrufe dem gerade aktiven Betriebssystem zuordnen. Alle „normalen“ Maschinenbefehle können direkt vom Prozessor ausgeführt werden.

Genau das ist die Idee der *Paravirtualisierung*.

Bei der Paravirtualisierung liegt unterhalb der *Gastbetriebssysteme* ein spezieller Virtual Machine Manager, der sog. *Hypervisor*. Er ist in der Lage, zwischen verschiedenen *Gastbetriebssystemen* hin- und herzuschalten.



Der Hypervisor bestimmt, welches Betriebssystem gerade aktiv ist und sorgt dafür, dass die Betriebssystemaufrufe der Anwendungsprogramme auch von dem aktiven Betriebssystem ausgeführt werden.

Aber wie macht der das?
Dazu muss man wissen, dass die Maschinenbefehle in zwei Klassen eingeteilt sind, „normale“, die einfach ausgeführt werden, d.h. im sog. *User-Mode*, und welche, die eine spezielle Berechtigung brauchen, den sog. *Kernel-Mode*. Die Befehle, die Kernel-Mode Berechtigung brauchen sind i.A. genau die Betriebssystemfunktionen. Die Schnittstelle zum Betriebssystem besteht dann aus einem sog. `syscall`-Aufruf, welcher als ein Parameter die Nummer der Betriebssystemfunktion erhält.

Soll jetzt ein Befehl ausgeführt werden, der Kernel-Mode Berechtigung braucht, aber das Kernel-Mode Flag ist gerade nicht gesetzt, dann liegt normalerweise ein Fehler vor, und der Prozess, der das versucht, wird abgebrochen. Das geschieht durch einen *Interrupt*, der den Prozessor zwingt, eine spezielle Interruptroutine auszuführen. Das genau ist der Punkt, wo der Hypervisor eingreifen kann. *Er ersetzt die normale Interruptroutine durch eine eigene, die bewirkt, dass die Betriebssystemfunktion des gerade aktiven Betriebssystems aufgerufen wird.*

Leider kann das nur funktionieren, wenn die virtualisierten Betriebssysteme nicht in den Kernel-Mode umschaltet. Nur dann versuchen sie, die privilegierten Befehle im User-Mode auszuführen, so dass überhaupt ein Interrupt erzeugt wird.

Also:

- Ein Maschinenbefehl, der im User Mode läuft, wird ganz normal, d.h. insbesondere ohne Zusatzaufwand ausgeführt.
- Ein syscall-Aufruf im User-Mode führt zu einem Interrupt.
- Die vom Hypervisor installierte Interruptroutine sorgt dafür, dass die Betriebssystemfunktion des gerade aktiven Betriebssystems aufgerufen wird.

Programme, die fast nur rechnen, und selten Interaktion nach außen brauchen, laufen dann in der virtualisierten Umgebung genau so schnell wie unvirtualisiert.

So wie es jetzt vorgestellt wurde, gibt es allerdings noch weitere Probleme:

1. Die Betriebssysteme müssen für den virtualisierten Einsatz neu compiliert werden, so dass sie nicht in den Kernel-Mode umschalten.
2. Es gibt leider einzelne Maschinenbefehle, die im User-Mode laufen, daher nicht abgefangen werden, und sich virtualisiert anders verhalten als unvirtualisiert. Z.B. der Befehl, der feststellt, ob gerade Kernel-Mode gilt, liefert in der virtualisierten Umgebung immer false. Programme, die so etwas benutzen, reagieren daher in der virtualisierten Umgebung anders als sonst.
3. Ein Betriebssystem geht normalerweise davon aus, dass es den gesamten Arbeitsspeicher zur Verfügung hat. Virtualisiert bekommt ein Gastbetriebssystem jedoch nur eine Teil des Gesamtarbeitsspeichers zugeordnet, sodass die Speicherverwaltung über der Hypervisor geregelt werden muss.

In der ersten Generation von Hypervisoren wurden diese Probleme softwaremäßig gelöst. Durch *dynamisches Recompilieren* wurden die kritischen Maschinenbefehle *zur Laufzeit* umgebaut, so dass sie auch virtualisiert richtig funktionieren. Die Seitentabellen wurden zweistufig gemacht, so dass der Hypervisor Einfluss auf die Speicherplatzzuteiltung des Gastbetriebssystems nehmen konnte.

Seit ca. 2005 haben allerdings die Prozessoren von Intel und AMD Hardwareunterstützung für die Virtualisierung eingebaut. Damit gibt es u.A. nicht mehr nur global Kernel- und Usermode, sondern beides im sog. Root-Mode und im sog. Non-Root-Mode. Der Hypervisor läuft im Root-Mode, und die Gastbetriebssysteme laufen im Non-Root-Mode, aber da wieder unterschieden in Kernel- und Usermode. Jetzt können die Betriebssysteme unverändert auch virtualisiert laufen.

2.1 Typ 1- und Typ 2-Hypervisoren

Typ 1-Hypervisoren setzen direkt auf der Hardware auf, und brauchen kein vorher installiertes Betriebssystem. Beispiele sind Hyper-V (Codename „Viridian“ – früher auch „Windows Server Virtualization“), von Microsoft erstmals 2008 ausgeliefert,VMware ESX/ESXi und Xen.

Typ 2-Hypervisoren brauchen ein vorher installiertes Betriebssystem, und laufen innerhalb dieses *Host-Betriebssystems* als normaler Prozess. Beispiele dafür sind VirtualBox, Windows Virtual PC, VMware Workstation und Parallel Desktop for Mac.

Die Hersteller von Großrechnern, allen voran IBM, bieten jedoch schon seit Jahrzehnten eigene Virtualisierungsumgebungen für ihre Rechner.

3 Emulation

Emulation bedeutet, dass eine Maschine A die komplette Architektur einer Maschine B *simuliert*. Das bedeutet, die Maschinenbefehle von Maschine B müssen durch Maschine A simuliert werden, die ganze Umgebung von Maschine B, Speicher, Filesystem, IO-Kanäle usw. müssen simuliert werden.

Ein Emulator ist daher ein Programm in einem normalen Betriebssystem, welches wiederum Programme oder ganze Betriebssysteme laden und ausführen kann. Im Unterschied zu Paravirtualisierung kann die Prozessorarchitektur des Gastsystems komplett anders sein als die Prozessorarchitektur, auf der das Hostbetriebssystem läuft.

Populär wurden Emulatoren für alte Spielekonsolen. Sie erlauben, alte, lägst vom Markt verschwundene Spielekonsolen in neuer Hardware laufen zu lassen. Ein sehr aktueller Emulator dagegen ist die Java Virtual Maschine. Sie emuliert eine Java Byte-Code Maschine (die bisher nicht als Hardware existiert). Jeder Java Byte-Code Befehl wird dabei durch ein kleines Maschinenprogramm des Hostsystems ausgeführt.

Dadurch, dass jeder Maschinenbefehl des Gastsystems durch evtl. hunderte von Maschinenbefehlen des Hostsystems ausgeführt wird, geht natürlich Geschwindigkeit verloren. Bei den Emulatoren für die alten Spielekonsolen merkt man das nicht, da die neueren Prozessoren sehr viel schneller sind,

manchmal sogar so viel schneller, dass die Geschwindigkeit künstlich gedrosselt werden muss.

Beispiele: Emulatoren von x86-Plattformen:

Bochs emuliert einen kompletten x86-Prozessor auf den unterschiedlichsten Hardwaren, u.a. Mac, SPARC, PlayStation Portable, iPhone, GP2X. Damit kann man z.B. ein Windows Betriebssystem auf der Playstation laufen lassen. Da Bochs ein Emulator ist, geht natürlich erheblich Geschwindigkeit verloren im Vergleich zu den Originalinstallationen.

DOSBox: zur Ausführung älterer, DOS-basierter Software. DOSEMU hat ein ähnliches Ziel, ist aber ein Hypervisor, der daher ein DOS-Betriebssystem braucht. Er ist deutlich schneller als DosBox, aber weniger genau steuerbar.

VirtualPC von Microsoft (emuliert z.B. eine x86-Plattform auf Macintosh-Systemen)

Win4Lin von Netraverse emuliert Windows Betriebssysteme in Linux.

Beispiele: Emulatoren für andere Plattformen:

Hercules, ein Emulator für verschiedene IBM Mainframes, wie das System/360,370/390.

MAME emuliert verschiedene Arcade-Automaten

MESS emuliert verschiedene Spielekonsolen und Heimcomputer-Modelle

PearPC emuliert G3- und G4-PowerPC-Plattformen (PPC)

SIMH emuliert verschiedene Minirechner und Großrechner

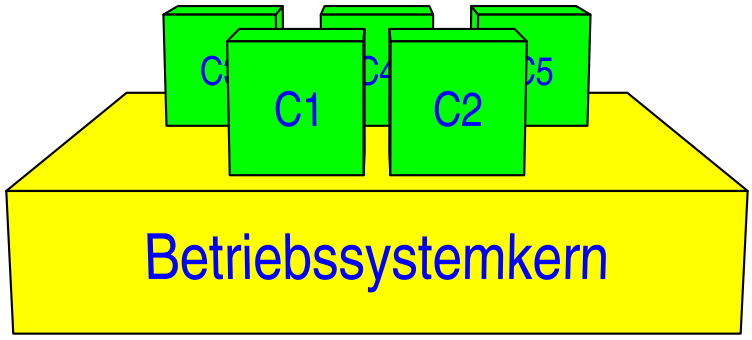
UAE emuliert Commodore-Amiga-Systeme (Motorola 68k-Prozessoren und Custom- Chips)

MaNGOS emuliert die World of Warcraft Server-Software

Antrix/Ascent emuliert ebenfalls die World of Warcraft Server-Software

4 Partitionierung

Bei solchen Systemen läuft *ein* Betriebssystemkern, stellt aber mehrere voneinander abgeschottete Laufzeitumgebungen, sog. *Container*, zur Verfügung. In diesen kann jeder Benutzer z.B. seine eigene Version von Bibliotheken oder Anwendungsprogrammen installieren, ohne dass sich diese gegenseitig stören.



Da die Anwendungen direkt auf dem Prozessor laufen, gibt es kaum Performanzverluste. Die verschiedenen Containersysteme unterscheiden sich natürlich, auf welchem Betriebssystemkern sie laufen, aber auch wie die Sicht auf die Systemumgebung, insbesondere das Filesystem ist.

Beispiele sind:

Docker läuft über einem Linux-Kern und stellt Container für Linux Programme zur Verfügung. Mit Hilfe eines Hypervisor kann Docker auch z.B. unter Windows verwendet werden. **Open Virtuozzo** und **Linux-VServer** haben eine ähnliche Funktionalität wie Docker.

FreeBSD jail läuft auf der Unix-Variante BSD und stellt dort sog. Jails zur Verfügung, in denen voneinander abgeschottet BSD-Programme laufen können.

Solaris Zones stellt im Oracle Solaris Betriebssystem sog. Zones zur Verfügung, in denen Solaris Programme laufen können.

5 API-Emulatoren

Wenn z.B. ein Windows Programm für *dieselbe Prozessorarchitektur* compiliert ist, wie das gerade laufende Betriebssystem, z.B. Linux, dann passen zumindest die Maschinenbefehle. Das Windows Programm könnte theoretisch auch in Linux laufen, tut es aber nicht. Ein Grund ist, dass Windows andere Bibliotheken benutzt als Linux. Ein weiterer Grund ist, dass Windows die compilierten Programmteile anders lädt, und daraus andere Prozesse generiert als Linux.

Das sind aber keine unüberwindlichen Probleme. Man bräuchte

- die Originalbibliotheken von Windows, oder zumindest eine möglichst originalgetreue Nachimplementierung, und
- einen speziellen Lader, der die Windows-Programme als Linux-Prozesse laden kann.

WINE
Genau solche Programme gibt es, genannt *API-Emulatoren* (API = Application Programming Interface). Eines davon ist *WINE* (steht für Wine Is Not an Emulator). WINE läuft auf Linux. Es hat

Nachimplementierungen von Windows-Bibliotheken, incl. der Bibliotheken für die graphische Oberfläche und einen entsprechenden Lader. Leider sind die Originalbibliotheken von Microsoft nicht perfekt dokumentiert, so dass die Nachimplementierungen auch nicht perfekt sind. Falls WINE aber auf der Festplatte eine entsprechende Originalbibliothek findet, dann kann es diese nehmen.

Cygwin
Das Gegenstück zu WINE ist *Cygwin*. Es erlaubt, Linux-Programme unter Windows laufen zu lassen.

6 Dynamisches Recompilieren

Im Prinzip ist es möglich, Maschinencode von einer Maschine A in Maschinencode einer Maschine B zu übersetzen. Damit könnte man Programme, die für einen Prozessortyp compiliert sind, auch auf einem anderen Prozessortyp laufen lassen.

Dynamisches Recompilieren macht das nicht komplett für ein ganzes Programm, sondern *während der Laufzeit*, für jeweils einzelne Blöcke des Programms.

Ein Beispiel ist der *Just in Time Compiler* für Java. Dieser beobachtet zur Laufzeit, welche Methoden eines Java-Programms besonders intensiv genutzt werden, und übersetzt die dann *zur Laufzeit* in den Maschinencode des darunterliegenden Prozessors, und das, ohne das der Benutzer etwas davon merkt, außer dass das Programm schneller wird.

Ein anderes, heute kaum noch gebrauchtes System, ist der *Rosetta Compiler* von Apple. Als Apple von PowerPC auf Intel Prozessoren umschwenkte, erlaubte Rosetta die Ausführung von PowerPC Programmen auf Intel Prozessoren.

Ein weiteres Beispiel ist QEMU. Es emuliert verschiedene Prozessorarchitekturen durch dynamische Rekompilation. Es läuft auf ganze verschiedenen Betriebssystemen. QEMU kann sogar den momentanen Stand des System abspeichern, woanders hin übertragen, und dort weiterlaufen lassen (life migration).

7 Resumé

Virtualisierung war viele Jahrzehnte eine Domäne von Großrechnern. Schon in den 50ern und 60ern wurde in der University of Cambridge, USA, mit dem *Conversational Time Sharing System* (CTSS) mit Virtualisierung experimentiert. IBM hat 1965 in seinem IBM/360-67 Rechner das CTSS-System adaptiert und kommerziell nutzbar gemacht.

Heute gibt es kaum noch einen Serverrechner, der nicht virtualisiert ist, insbesondere um die Hardware besser auszunutzen, aber auch um Systemfehler, Abstürze und Hackerangriffe auf das jeweilige Gastsystem zu beschränken ohne den ganzen Rechner in Gefahr zu bringen.

Aber auch Workstations und Notebooks sind heute leistungsfähig genug, dass man, z.B. mit Virtual Box, mehrere Betriebssysteme parallel betreiben kann.

Die in diesem Miniskript aufgelisteten konkreten Systeme unterliegen permanenter Veränderung. Manche sind vielleicht schon veraltet und nicht mehr brauchbar, manche können vielleicht inzwischen viel mehr als noch letztes Jahr. Wenn man sich für ein konkretes System interessiert, sollte man daher die Originaldokumentation ansehen.

Stichwortverzeichnis