

1 Einführung

Lernziele

- Sie erklären den Zweck und die Rolle eines modernen Betriebssystems.
- Sie erkennen wie Rechnerressourcen durch Applikationen genutzt werden, wenn sie das Betriebssystem verwaltet.
- Sie beschreiben die Funktionen eines aktuellen Betriebssystems in Bezug auf Benutzbarkeit, Effizienz und Entwicklungsfähigkeit.
- Sie erklären die Vorteile abstrakter Schichten und ihrer Schnittstellen in hierarchisch gestalteten Architekturen.
- Sie analysieren die Kompromisse beim Entwurf eines Betriebssystems.
- Sie erläutern die Architektureigenschaften monolithischer, geschichteter, modularer und Mikrokernsysteme.
- Sie stellen netzwerkfähige, Client/Server- und verteilte Betriebssysteme einander gegenüber und vergleichen diese.

Als Einstieg in das Thema legen wir fest, welchen Zwecken ein Betriebssystem dient, wie es sich als Begriff definieren lässt und wo es in einem Rechner einzuordnen ist. Danach diskutieren wir die Anforderungen an den Betriebssystementwurf, mögliche Architekturen und weiterführende Ideen aus der Forschung.

1.1 Zweck

Der Begriff »Betriebssystem« kann unterschiedlich aufgefasst werden. Beispielsweise über die Frage: Was leistet ein Betriebssystem? Zwei Grundfunktionen sind:

- *Erweiterte Maschine*: Das Betriebssystem realisiert von vielen Applikationen geichartig genutzte Teilfunktionen als standardisierte Dienste. Damit wird die Applikationsentwicklung einfacher als beim direkten Zugriff auf die blanke

Rechnerhardware. Die erweiterte Maschine ist eine Abstraktion der Hardware auf hohem Niveau und entspringt einer Top-down-Sicht.

- *Betriebsmittelverwalter*: Das Betriebssystem verwaltet die zeitliche und räumliche Zuteilung von Rechnerressourcen. Im Mehrprogrammbetrieb wird im Zeitmultiplex der Prozessor zwischen verschiedenen ablauffähigen Programmen hin und her geschaltet. Im Raummultiplex wird der verfügbare Speicher auf geladene Programme aufgeteilt. Ausgehend von den Ressourcen entspricht dies einer Bottom-up-Sicht.

Detaillierter betrachtet erfüllt ein Betriebssystem sehr viele Zwecke. Es kann mehrere oder sogar alle der folgenden Funktionalitäten realisieren:

- *Hardwareunabhängige Programmierschnittstelle*: Programme können unverändert auf verschiedenen Computersystemen ablaufen (auf Quellcodeebene gilt dies sogar für unterschiedliche Prozessorfamilien mit differierenden Instruktionssätzen).
- *Geräteunabhängige Ein-/Ausgabefunktionen*: Programme können ohne Änderung unterschiedliche Modelle einer Peripheriegeräteart ansprechen.
- *Ressourcenverwaltung*: Mehrere Benutzer bzw. Prozesse können gemeinsame Betriebsmittel ohne Konflikte nutzen. Die Ressourcen werden jedem Benutzer so verfügbar gemacht, wie wenn er exklusiven Zugriff darauf hätte.
- *Speicherverwaltung*: Mehrere Prozesse/Applikationen können nebeneinander im Speicher platziert werden, ohne dass sie aufeinander Rücksicht nehmen müssen (jeder Prozess hat den Speicher scheinbar für sich allein). Zudem wird bei knappem Speicher dieser optimal auf alle Nutzer aufgeteilt.
- *Massenspeicherverwaltung (Dateisystem)*: Daten können persistent gespeichert und später wieder gefunden werden.
- *Parallelbetrieb (Multitasking)*: Mehrere Prozesse können quasiparallel ablaufen. Konzeptionell stehen mehr Prozessoren zur Verfügung als in der Hardware vorhanden, indem versteckt vor den Anwendungen parallele Abläufe, soweit nötig, sequenzialisiert werden.
- *Interprozesskommunikation*: Prozesse können mit anderen Prozessen Informationen austauschen. Die Prozesse können dabei entweder auf dem gleichen Rechner ablaufen (lokal) oder auf verschiedenen Systemen (verteilt) ausgeführt werden.
- *Sicherheitsmechanismen*: Es können sowohl Funktionen für die Datensicherung, d.h. die fehlerfreie Datenverarbeitung, als auch Datenschutzkonzepte implementiert sein. Der Datenschutz kann zum Beispiel durch das explizite Löschen freigegebener Bereiche im Hauptspeicher und auf Plattenspeichern sicherstellen, dass empfindliche Informationen nicht in falsche Hände fallen. Die Zugangskontrolle zum Rechner (Anmeldedialoge, Benutzerverwaltung) dient ebenfalls dem Datenschutz.

- *Bedienoberflächen*: Moderne Betriebssysteme realisieren grafische Bedienoberflächen mit ausgeklügelten Bedienkonzepten, die Dialoge mit dem System und Anwendungen komfortabel gestalten. Ergänzend existieren Eingabemöglichkeiten für Kommandozeilenbefehle, die geübten Benutzern sehr effiziente Dialogmöglichkeiten, z.B. zur Systemadministration, anbieten.

Die geräteunabhängige Ein-/Ausgabe war eine der wichtigsten Errungenschaften bei der erstmaligen Einführung von Betriebssystemen. Früher war es notwendig, dass Applikationen die Eigenheiten der angeschlossenen Peripheriegeräte im Detail kennen mussten. Mit einem Betriebssystem stehen hingegen logische Kanäle zur Verfügung, die Ein-/Ausgaben über standardisierte Funktionen bereitstellen (siehe Abb. 1–1). Die logischen Kanäle werden häufig mittels sprechender Textnamen identifiziert.

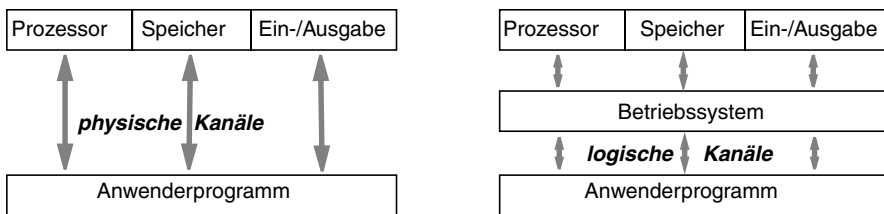


Abb. 1–1 Ein-/Ausgabe ohne und mit Betriebssystem

1.2 Definitionen

Leider existiert keine allgemein verbindliche Definition eines Betriebssystems. Welche Komponenten zu einem Betriebssystem gehören und welche nicht, lässt sich daher nicht endgültig festlegen. Nachfolgend sind drei unterschiedliche Definitionen stellvertretend vorgestellt, die dabei helfen, ein Betriebssystem zu charakterisieren. Eine erste, etwas schwer lesbare Definition nach DIN 44 300 beschreibt ein Betriebssystem wie folgt (Ausschnitt):

... die Programme eines digitalen Rechnersystems, die zusammen mit den Eigenschaften dieser Rechanlage die Basis der möglichen Betriebsarten des Rechnersystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.

Eine zweite, der Literatur entnommene Definition lautet:

Ein Betriebssystem ist eine Menge von Programmen, welche die Ausführung von Benutzerprogrammen auf einem Rechner und den Gebrauch der vorhandenen Betriebsmittel steuern.

Eine dritte Definition betrachtet das Betriebssystem als *Ressourcenverwalter*, wobei die Ressource hauptsächlich die darunter liegende Hardware des Rechners ist. Ein Computersystem lässt sich hierbei als eine strukturierte Sammlung von Ressourcenklassen betrachten, wobei jede Klasse durch eigene Systemprogramme kontrolliert wird (siehe Tab. 1–1).

	Zentrale Ressourcen	Periphere Ressourcen
Aktive Ressourcen	Prozessor(en)	Kommunikationseinheiten 1. Endgeräte (Tastaturen, Drucker, Anzeigen, Zeigergeräte etc.) 2. Netzwerk (entfernt, lokal) etc.
Passive Ressourcen	Hauptspeicher	Speichereinheiten 1. Platten 2. Bänder 3. CD-ROM/DVD etc.

Tab. 1–1 Ressourcenklassen

Ein Betriebssystem lässt sich auch mit einer Regierung (*government*) vergleichen. Wie diese realisiert das Betriebssystem keine nützliche Funktion für sich alleine, sondern stellt eine Umgebung zur Verfügung, in welcher andere Beteiligte nützliche Funktionen vollbringen können. Einige Autoren (z.B. K. Bauknecht, C. A. Zehnder) ziehen die Begriffe *Systemsoftware* bzw. *Systemprogramme* der Bezeichnung *Betriebssystem* vor. In diesem Sinne ist folgende Beschreibung dieser Autoren abgefasst:

»Die Systemprogramme, oft unter dem Begriff *Betriebssystem* zusammengefasst, lassen sich gemäß Abbildung 1–2 gruppieren.

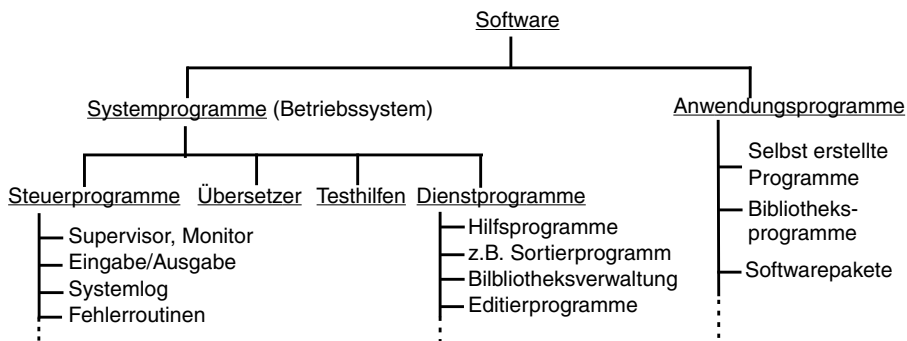


Abb. 1–2 Softwaregliederung

Die eigentlichen *Steuerprogramme* sind für folgende Funktionen zuständig:

- *Steuerung aller Computerfunktionen* und Koordination der verschiedenen zu aktivierenden Programme.

- *Steuerung der Ein-/Ausgabeoperationen* für die Anwendungsprogramme.
- *Überwachung und Registrierung* der auf dem Computersystem ablaufenden Aktivitäten.
- *Ermittlung und Korrektur* von Systemfehlern.«

Auffallend bei dieser Definition ist der Einbezug von *Übersetzern* (Compiler, Binder), *Testhilfen* und *Dienstprogrammen*. Für klassische Betriebssysteme (z.B. Unix und GNU-Tools) trifft dies vollumfänglich zu, während moderne Betriebssysteme oft die Bereitstellung von Übersetzungstools irgendwelchen Drittherstellern überlassen bzw. diese als separate Applikation ausliefern (z.B. Windows und Visual Studio).

1.3 Einordnung im Computersystem

In einem Rechner stellt das Betriebssystem eine Softwareschicht dar, die zwischen den Benutzerapplikationen einerseits und der Rechnerhardware andererseits liegt (siehe Abb. 1–3). Das Betriebssystem selbst besteht aus einem *Betriebssystemkern* und einer Sammlung von Programmen, die *Betriebssystemdienste* bereitstellen. Je nach Betrachtungsweise zählen dazu auch Programme zur Softwareentwicklung, wie Editoren und Compiler. Häufig wird nur der Betriebssystemkern als Betriebssystem bezeichnet, während der Begriff *Systemprogramme* für das Gesamtpaket inklusive der Programmentwicklungswerkzeuge benutzt wird.

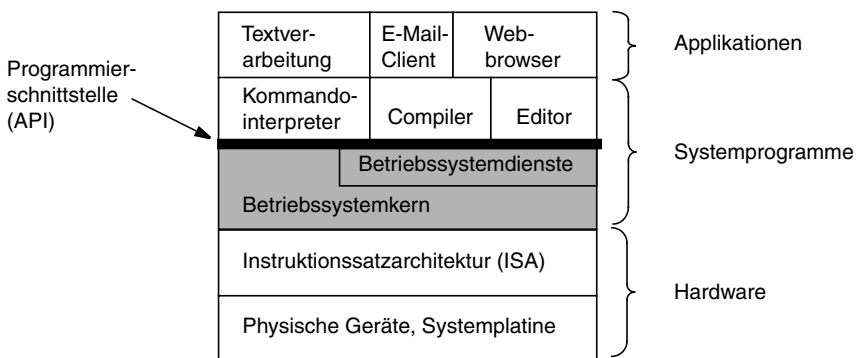


Abb. 1–3 Schichtenmodell eines Rechners

Das Betriebssystem setzt auf der Prozessorarchitektur auf, die durch einen Satz von Maschinenbefehlen und den Registeraufbau charakterisiert wird (sog. Instruktionssatzarchitektur, ISA). Die Systemplatine mit all ihren Bausteinen und den angeschlossenen Peripheriegeräten stellt die Arbeitsumgebung des Prozessors dar. Diese muss ebenfalls dem Betriebssystem in all ihren Details bekannt sein. Von zentraler Bedeutung für den Softwareentwickler ist die Programmierschnittstelle des Betriebssystems (*Application Programming Interface, API*). Die dort

zur Verfügung gestellte Funktionalität kann in Benutzerapplikationen eingesetzt werden. Aus Anwendungssicht unterscheiden sich Betriebssysteme in der *Programmierschnittstelle*, in den unterstützten *Dateiformaten für ausführbare Dateien*, im *Funktionsumfang*, in der *Bedienoberfläche* und der *Maschinensprache*, in die ihr Code übersetzt wurde. Zudem kann oft der Funktionsumfang, d.h. die installierten Systemteile, während des Installationsvorgangs unterschiedlich gewählt werden.

Wie bereits erwähnt, setzt das Betriebssystem direkt auf der Rechnerhardware auf und muss diese daher genau kennen. Denn es verwaltet folgende Hardwareelemente:

- Prozessor
- Arbeitsspeicher (*main memory*)
- Massenspeicher (*mass storage*), z.B. Festplatten, CD-ROM, DVD
- Benutzerschnittstelle (*user interface*)
- Kommunikations- und andere Peripheriegeräte (LAN, WLAN usw.)

Die Betriebssystemtheorie beruht damit auf den Prinzipien der Computertechnik. Computertechnik befasst sich mit:

1. Rechner-Grundmodellen (Von-Neumann-, Harvard-Architektur)
2. Funktionsweise des Prozessors (Instruktionssatz, Registeraufbau)
3. Speichern und ihren Realisierungen (Primär- und Sekundärspeicher)
4. Peripheriegeräten (Tastatur, Bildschirm, Schnittstellenbausteine usw.)

Um die hardwarenahen Teile des Betriebssystems oder nur schon den exakten Ablauf der Programmausführung zu verstehen, ist es daher unerlässlich, sich mit ein paar Details der Computertechnik zu befassen. Einige computertechnische Funktionsweisen, soweit sie für das Verständnis des Betriebssystems nötig sind, werden an passenden Stellen im Buch erklärt. Für weiter gehende Realisierungsdetails der Hardwareelemente sei auf entsprechende Spezialliteratur verwiesen.

1.4 Betriebssystemarten

Ein Betriebssystem stellt eine Umgebung zur Verfügung, in der Anwendungsprogramme ablaufen können. Eine Ablaufumgebung kann recht unterschiedlich realisiert sein:

- Als Laufzeitsystem (*Run-Time System*) einer Programmiersprache (ADA, Modula-2)
- Als virtuelle Maschine zur Ausführung eines Zwischencodes (z.B. Java Virtual Machine, .NET Common Language Runtime)
- Als Basisprogramm eines Rechners (z.B. Unix, Windows)

- Als (sprachunabhängige) Programmbibliothek (z.B. Mikrokontroller-Betriebssysteme)

Häufig findet man Kombinationen dieser vier Varianten. Beispielsweise können Sprach-Laufzeitsysteme Fähigkeiten zur Verfügung stellen, die ansonsten nur Bestandteil von Betriebssystemen sind. Dies beinhaltet Multitasking-Funktionen (z.B. in Java, Ada, Modula-2) und die Speicherverwaltung (verschiedene Sprachen).

1.4.1 Klassische Einteilungen

Eine elementare Klassifizierung von Betriebssystemen basiert auf folgenden Anwendungsarten:

- *Stapelverarbeitung (batch processing)*: Typisches Merkmal ist, dass Programme angestoßen werden, aber ansonsten keine nennenswerte Benutzerinteraktion stattfindet. Die auszuführenden Befehle sind stattdessen in einer Stapeldatei abgelegt, deren Inhalt fortlaufend interpretiert wird. Klassische Großrechnerbetriebssysteme werden auf diese Art und Weise genutzt, z.B. zur Ausführung von Buchhaltungsprogrammen über Nacht.
- *Time-Sharing-Betrieb*: Die zur Verfügung stehende Rechenleistung wird in Form von Zeitscheiben (*time slices, time shares*) auf die einzelnen Benutzer aufgeteilt mit dem Ziel, dass jeder Benutzer scheinbar den Rechner für sich alleine zur Verfügung hat. Historisch gesehen sind Time-Sharing-Systeme die Nachfolger bzw. Ergänzung der Batch-Systeme mit der Neuerung, dass sie Benutzer interaktiv arbeiten lassen (Dialogbetrieb).
- *Echtzeitbetrieb*: Die Rechenleistung wird auf mehrere Benutzer oder zumindest Prozesse aufgeteilt, wobei zeitliche Randbedingungen beachtet werden. Oft sind Echtzeitsysteme reaktive Systeme, indem sie auf gewisse Signale aus der Umgebung (Interrupts, Meldungen) möglichst rasch reagieren.

1.4.2 Moderne Einteilungen

Moderne Betriebssysteme fallen mehr oder weniger in die Gruppe der Echtzeitsysteme, weswegen letztere für uns im Vordergrund stehen. Eine ergänzende Klassifizierung unterteilt Betriebssysteme nach unterstützter Rechnerstruktur:

- Einprozessorsysteme
- Multiprozessorsysteme
- Verteiltes System

Je nach Auslegung kann ein Betriebssystem eine oder mehrere dieser drei Rechnerstrukturen unterstützen. Ergänzend sei noch bemerkt, dass populäre Betriebssysteme netzwerkfähig (*networked operating system*) sind, auch wenn sie nicht

verteilt ablaufen. Beispielsweise unterstützen sie verbreitete Netzwerkprotokolle, die Anbindung entfernter Laufwerke und – teilweise konfigurierbar – eine zentralisierte Benutzerverwaltung.

Beispiele:

Windows unterstützt Einprozessorsysteme und Multiprozessorsysteme. Das Betriebssystem Amoeba ermöglicht transparentes Arbeiten auf einem verteilten System. Für den Benutzer präsentiert es sich wie ein Einzelrechner, besteht in der Tat aber aus mehreren über ein Netzwerk verbundenen Computern.

1.4.3 Geschichte

Abbildung 1–4 zeigt eine kleine Auswahl an Entwicklungslinien gängiger Betriebssysteme, deren Geschichte wir kurz charakterisieren.

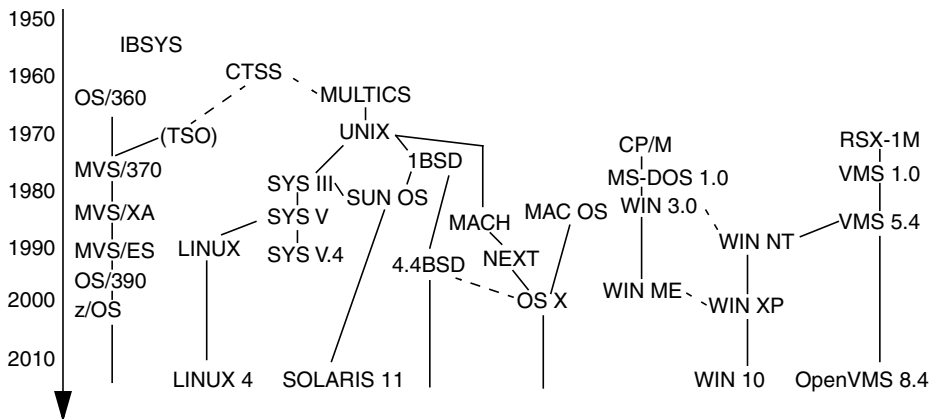


Abb. 1–4 Entwicklungslinien einiger gängiger Betriebssysteme

Ein erstes Betriebssystem für Großrechner war das rudimentäre *IBSYS*, das Stapelverarbeitung ermöglichte. Umfangreicher war bereits das *OS/360* von IBM, das in weiterentwickelter Form als *z/OS* auf heutigen Mainframe-Systemen läuft. Anfänglich hat es nur die Stapelverarbeitung unterstützt, wurde aber bald durch die *TSO* (*Time Sharing Option*) für den Dialogbetrieb ergänzt. Unabhängig davon entstand das *CTSS* (*Compatible Time Sharing System*), das den Dialogbetrieb auf Großrechnern bereits sehr früh erlaubte. Sein Nachfolger war *MULTICS* (*Multiplexed Information and Computing Service*), ein Konsortiumsprojekt, das letztlich nicht sehr erfolgreich war, jedoch viele neue Konzepte realisierte. Darin war es ein Vorbild für das ursprüngliche *Unix*, das jedoch ein wesentlich kompakterer Entwurf war, der die Komplexität des *MULTICS* vermied. Unix hat über viele Zwischenschritte die heutigen Systeme *Linux*, *Oracle Solaris* und *Apple OS X*

geprägt. Das *BSD (Berkeley Software Distribution) Unix* existiert heute als *FreeBSD*, *NetBSD* und *OpenBSD* in geringer Verbreitung weiter. Separate Entwicklungslinien gelten für das *Microsoft Windows*. Ursprünglich hat es als grafische Oberfläche für *MS-DOS* begonnen, wurde aber immer unabhängiger davon. Separat zu dieser originären Windows-Linie entstand das *Windows NT*, das von den *DEC VMS (Virtual Memory System)* Minicomputer-Betriebssystemen abgeleitet wurde, jedoch die API des *Microsoft Windows* realisierte. Mit dem *Windows XP* wurde die originäre Windows-Linie beendet, womit der schwache Unterbau des *MS-DOS* verschwand. Das *VMS* existiert als *OpenVMS* noch heute, ist aber nur minimal verbreitet.

1.5 Betriebssystemarchitekturen

Beim Entwurf eines Betriebssystems sind viele Anforderungen in Einklang zu bringen, die nicht widerspruchsfrei sind, weswegen Kompromisse nötig sind. Neben der Realisierung der in Abschnitt 1.1 beschriebenen Kernfunktionalitäten sind folgende exemplarische Entwurfsziele zu berücksichtigen:

- Fehlerfreiheit des Codes: z.B. durch minimale Komplexität des Quellcodes
- Einfache Operationen (auf API und für alle Schnittstellen)
- Erweiterbarkeit (*extensibility*)
- Skalierbarkeit (*scalability*)
- Orthogonalität: Operationen wirken gleich auf verschiedenartigen Objekten
- Robuste Betriebsumgebung (»crash-proof«, »reliable«)
- Einhaltung der Sicherheitsziele (mehrere Anforderungsstufen denkbar)
- Portabilität (Unterstützung verschiedenartiger Plattformen)
- Echtzeitfähigkeit (z.B. für Multimedia-Anwendungen)
- Effizienz (schnelle Dienstleistung, minimaler Ressourcenbedarf)
- Weiterentwickelbarkeit: Trennung von Strategie (*policy*) und Mechanismus (*mechanism*)

Auf der Suche nach einem optimalen Entwurf sind verschiedenartige Architekturideen entwickelt worden. Diese werden nachfolgend kurz beschrieben und diskutiert. Als Blick in die mögliche Zukunft des Betriebssystembaus wird eine kurze Zusammenfassung einiger interessanter Forschungsarbeiten zum Thema vorgestellt.

1.5.1 Architekturformen

Solange es lediglich um die Systemprogrammierung geht, ist eine Blackbox-Betrachtung des Betriebssystems ausreichend. Nach außen ist damit nur die Programmierschnittstelle sichtbar, jedoch nicht das Systeminnere (siehe A in Abb. 1–5). Dies entspricht einem klassischen Ideal des Software Engineering, das aussagt,

dass die Schnittstelle das Maß aller Dinge ist und die Implementierung dahinter beliebig austauschbar sein soll. Dennoch kann es hilfreich sein, die Innereien eines Betriebssystems zu kennen, damit man nicht Gefahr läuft, gegen die Implementierung zu programmieren. Dies könnte zum Beispiel in einem überhöhten Ressourcenverbrauch oder einer unbefriedigenden Ausführungsgeschwindigkeit resultieren. Daneben ist es stets interessant, unter die »Motorraumhaube« eines Betriebssystems zu gucken. Mit anderen Worten, es geht um eine Whitebox-Betrachtung (siehe B in Abb. 1–5).

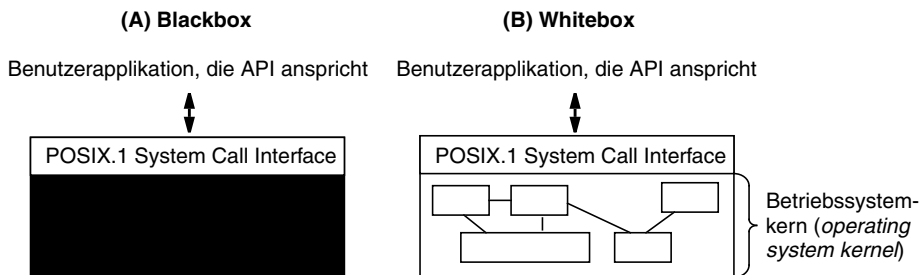


Abb. 1–5 Black- und Whitebox-Betrachtung (Beispiel: Unix)

Damit verbunden sind die Entwurfs- und Konstruktionsprinzipien, die einen erst dann interessieren, wenn man über die Blackbox-Betrachtung hinausgeht. Eine wesentliche Frage ist dabei die Art und Weise, wie die Betriebssystemsoftware strukturiert ist. In der Theorie kennt man *drei Grundstrukturen*, denen sich konkrete Betriebssysteme zuordnen lassen: *monolithische*, *geschichtete* und *Mikrokernsysteme*. Diese werden durch Strukturen für Multiprozessor- und Verteilte Systeme ergänzt. Zuerst soll jedoch auf die Funktionsweise und Bedeutung der Benutzer-/Kernmodus-Umschaltung eingegangen werden, da sie bei der Betrachtung dieser Strukturen eine zentrale Rolle spielt.

1.5.2 Benutzer-/Kernmodus

Als hardwarenahe Softwarekomponente ist ein Betriebssystem eng mit den Möglichkeiten der unterliegenden Plattform verbunden. Es haben sich mit den Jahren unterschiedliche Leistungsklassen von Prozessoren und zugehöriger Hilfslogik etabliert:

- *Mikrocontroller*: Es handelt sich hierbei um einfache Mikroprozessoren, die primär in sehr einfachen eingebetteten Systemen (*embedded systems*) eingesetzt werden. Um die Kosten gering zu halten, verfügen sie lediglich über einen Prozessor mit wenig oder gar keinen weiterführenden Mechanismen zur Unterstützung eines Betriebssystems. Hingegen sind sie zusammen mit verschiedenen Peripherieeinheiten (E/A, Kommunikation, Zeitgeber usw.) in

einen einzigen Halbleiterchip integriert, was Kosten und Platz spart.

Beispiele: Intel 8051, Siemens 80C166, Motorola HC6805

- *Einfache Universalmikroprozessoren*: Sie entsprechen in vielen Punkten den Mikrocontrollern, enthalten jedoch auf dem gleichen Chip keine Peripherieeinheiten. In dieser Gruppe finden wir vor allem die älteren Prozessortypen. Beispiele: Intel 8080/85/86, Motorola 6800, 68000
- *Leistungsfähige Universalmikroprozessoren*: Diese Rechnerchips verfügen über eine ganze Reihe von Hardwareelementen, die ein Betriebssystem unterstützen. Dazu zählen eine MMU (*Memory Management Unit*) und Mechanismen für einen privilegierten Betriebsmodus für die Systemsoftware (Privilegiensystem). Diese Prozessoren eignen sich nicht nur für Desktop-Systeme, Servermaschinen, Tablets und Smartphones, sondern auch für viele *Embedded Systems*, da sie die Verwendung angepasster Desktop-Betriebssysteme mit ihrer reichhaltigen Funktionalität erlauben.

Bei Universalmikroprozessoren werden durch das Privilegiensystem heikle Operationen und Zugriffe geschützt, damit ein Programmierfehler in einem Anwendungsprogramm nicht das ganze Computersystem durcheinanderbringt. Insbesondere bei Multitasking-Anwendungen und Multiuser-Betrieb (mehrere gleichzeitige Benutzer) ist ein solcher Schutz erwünscht. So wird in den meisten Betriebssystemen der Zugriff auf Hardwareteile mittels dieser Schutzfunktionen dem normalen Anwender (bzw. Benutzerapplikationen) verwehrt. Dazu dienen unterschiedliche CPU-Betriebsarten, wobei jede Betriebsart in ihren Pflichten und Rechten genau definiert ist. Im Minimum beinhaltet dies:

- Einen Kernmodus (*kernel mode, supervisor mode*) mit »allen Rechten« für Betriebssystemcode
- Einen Benutzermodus (*user mode*) mit »eingeschränkten Rechten« für Applikationscode

Das Ziel besteht darin, die Applikationen untereinander und den Betriebssystemcode gegen diese zu schützen. Dies bedeutet, dass eine Applikation nicht das ganze System lahmlegen kann. Komfortablere Lösungen unterstützen mehr als zwei Betriebsarten, die dann Privilegienstufen (*privilege level*) genannt werden. Die Möglichkeiten des Privilegiensystems werden stets in Kombination mit der Speicherverwaltung genutzt. So könnte die MMU dafür sorgen, dass nur im Kernmodus ein Zugriff auf Systemcode und Daten möglich ist (mehr Details dazu siehe Abschnitt 8.5). Den Benutzerprozessen ist mithilfe dieser hardwaregestützten Mechanismen in der Regel weder ein direkter Zugriff auf Hardwareteile noch ein Überschreiben von Systemcode oder Systemdaten möglich. Mit der Kenntnis der Fähigkeiten der Benutzer-/Kernmodus-Umschaltung lassen sich die nachfolgend aufgeführten drei Grundtypen von Aufbaustrukturen klarer in ihren Eigenschaften unterscheiden. Oberstes Ziel ist, das unabsichtliche Überschreiben von

Systemdaten und Code zu verhindern, um unkontrollierte Systemabstürze zu vermeiden.

	Benutzermodus	Kernmodus
Ausführbare Maschinenbefehle	Begrenzte Auswahl	Alle
Hardwarezugriff	Nein bzw. nur mithilfe des Betriebssystems	Ja, Vollzugriff
Zugriff auf Systemcode bzw. Daten	Keiner bzw. nur lesend	Exklusiv

Tab. 1-2 Vergleich zwischen Benutzer- und Kernmodus

Im Idealfall werden Schutzmechanismen auch für die Abschottung verschiedener Systemteile untereinander eingesetzt. Die Grenze des Sinnvollen ist allerdings darin zu sehen, dass eine teilweise lahmgelegte Systemsoftware aus Sicht des Anwenders oft nicht besser ist als ein Totalabsturz. Es kann jedoch manchmal nützlich sein, nicht vertrauenswürdige Teile der Systemsoftware, wie Erweiterungen oder Treiber von Drittherstellern, in ihrem Schadenspotenzial einzugrenzen. Ein sekundäres Ziel für den Einsatz der Benutzer-/Kernmodus-Umschaltung können Maßnahmen zur Eindämmung von Systemmanipulationen sein, die zum Ziel haben, vertrauenswürdige Daten zu missbrauchen. So ist Sicherheitssoftware fundamental von den Sicherheitseigenschaften einer Systemplattform abhängig, die softwareseitig durch das benutzte Betriebssystem gegeben ist.

1.5.3 Monolithische Systeme

Die Struktur dieser Systeme besteht darin, dass sie keine oder nur eine unklare Struktur haben (Abb. 1-6).

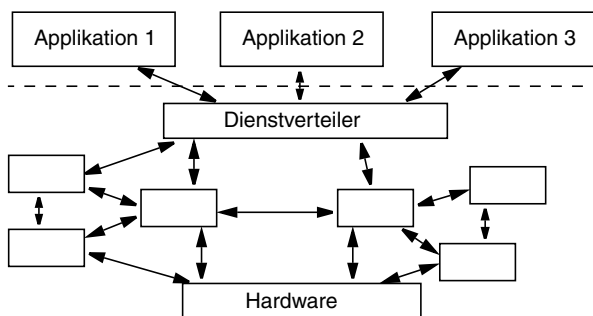


Abb. 1-6 Beispiel für eine monolithische Betriebssystemstruktur

Meist handelt es sich um evolutionär gewachsene Betriebssysteme, bei denen es anfänglich unwichtig war, einzelne Teilfunktionen klar mit Schnittstellen voneinander abzugrenzen. Beispiele dafür sind MS-DOS und ältere Unix-Varianten. Der-

artige Systeme können sehr effizient sein, da sich Schnittstellen frei wählen lassen. Sie sind jedoch schwierig wartbar, wenn ihre Struktur schlecht erkennbar ist.

Modulare Betriebssysteme stellen eine Erweiterung dar, bei der ausgewählte Komponenten derart mit definierten Schnittstellen versehen werden, dass sie den Zugriff auf unterschiedliche Implementierungsvarianten erlauben. Beispielsweise können so über dieselbe Dateischnittstelle verschiedenartige Dateisystemformate unterstützt werden. Meist lassen sich Module dynamisch laden/entladen. Beispiele sind Linux und Oracle Solaris.

1.5.4 Geschichtete Systeme

Bei dieser Strukturierungsform sind die Betriebssystemfunktionen in viele Teilfunktionen gegliedert, die hierarchisch auf mehrere Schichten verteilt sind. Die Festlegung der Schichtenstruktur ist vom einzelnen Betriebssystem abhängig, eine Standardstruktur gibt es nicht.

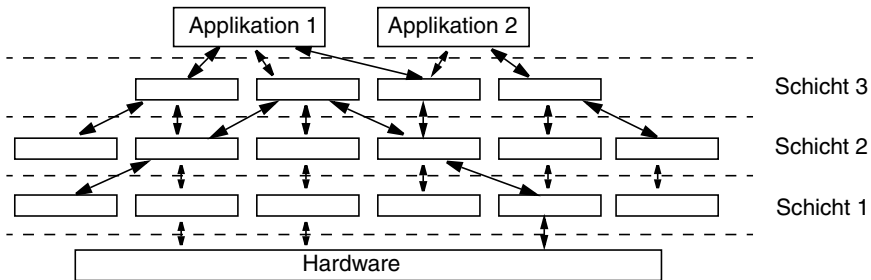


Abb. 1-7 Beispiel einer geschichteten Struktur

Wie in Abbildung 1-7 zu sehen ist, bauen Funktionen einer höheren Schicht strikt nur auf Funktionen einer tieferen Schicht auf. Jede Schicht realisiert eine bestimmte Funktionsgruppe. Systemaufrufe passieren nach unten alle Schichten, bis sie auf die Hardware einwirken. Eingabedaten durchlaufen umgekehrt alle Schichten von unten bis oben zur Benutzerapplikation. Die Schicht 1 in Abbildung 1-7 könnte zum Beispiel eine Hardware-Abstraktionsschicht sein, die eine allgemeine Betriebssystemimplementierung auf eine bestimmte Hardwareplattform anpasst. Beispiele für geschichtete Betriebssysteme sind neuere Unix-Varianten und OS/2. Vorteile dieser Struktur sind, dass sich einzelne Schichten gegen andere Implementierungen austauschen lassen und die Sichtbarkeit der Module durch die Schichten eingegrenzt wird. Durch die Schichtenaufteilung besteht jedoch das Problem, dass manche Funktionen künstlich aufgeteilt werden müssen, da sie nicht eindeutig einer bestimmten Schicht zuordenbar sind.

1.5.5 Mikrokernsysteme (Client/Server-Modell)

Nur die allerzentralsten Funktionen sind in einem Kernteil zusammengefasst, alle übrigen Funktionen sind als Serverdienste separat realisiert (z.B. Dateidienste, Verzeichnisdienste). Der Mikrokern enthält lediglich die vier Basisdienste Nachrichtenübermittlung (*message passing*), Speicherverwaltung (*virtual memory*), Prozessorverwaltung (*scheduling*) und Gerätetreiber (*device drivers*). Diese sind dabei in ihrer einfachsten Form realisiert. Weiter gehende Funktionen sind in den Serverprozessen enthalten, die im Benutzermodus ausgeführt werden. Dadurch werden die komplexeren Systemteile in klar abgegrenzte Teile aufgesplittet, wovon man sich eine Reduktion der Komplexität verspricht. Da zudem ein Großteil des Betriebssystemcodes auf die Benutzerebene (Benutzermodus) verschoben wird, sind überschaubare zentrale Teile des Systems durch den Kernmodus gegen fehlerhafte Manipulationen geschützt. Ebenso ist es nur dem eigentlichen Kern erlaubt, auf die Hardware zuzugreifen. Beansprucht ein Benutzerprozess einen Systemdienst, so wird die Anforderung als Meldung durch den Mikrokern an den zuständigen Serverprozess weitergeleitet. Entsprechend transportiert der Mikrokern auch die Antwort an den anfordernden Prozess zurück (siehe Abb. 1–8). Vorteilhaft ist das derart verwendete Client/Server-Modell für verteilte Betriebssysteme. Für den Benutzerprozess bleibt es verborgen (transparent), ob der Serverprozess lokal oder an einem entfernten Ort ausgeführt wird.

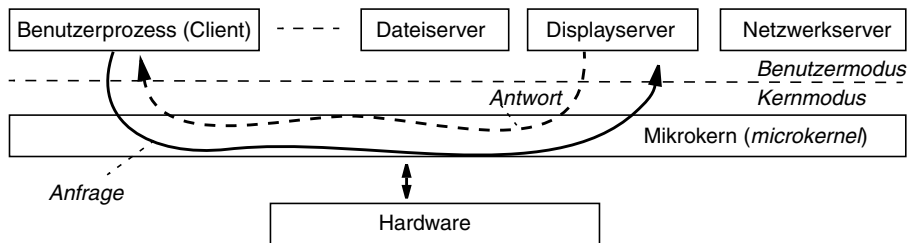


Abb. 1–8 Mikrokern nach dem Client/Server-Prinzip

Kommerzielle Mikrokernbetriebssysteme verlagern neben den vier erwähnten Grunddiensten zur Effizienzsteigerung zusätzliche Funktionen in den Mikrokern. Betrachtet man beispielsweise den Datenverkehr zwischen einem Benutzerprozess und dem Displayserver (siehe Abb. 1–8), so muss für eine bestimmte Operation auf dem grafischen Desktop insgesamt viermal eine Umschaltung zwischen Benutzer- und Kernmodus stattfinden, was durch Verschiebung der Displayfunktionen in den Kern effizienter gelöst werden kann. Beispiele derartiger Systemarchitekturen sind das Mac OS X (basierend auf dem Mach Kernel und BSD Unix) sowie Amoeba.

1.5.6 Multiprozessorsysteme

Multiprozessorsysteme haben durch die Einführung von Multicore-CPU's eine große Verbreitung erfahren. Dabei teilen sich typischerweise alle Rechenkerne die Peripherie und den Hauptspeicher, weswegen man sie *Shared-Memory-Multiprozessoren* nennt.

Drei Betriebssystemtypen sind für diese Rechner denkbar:

1. Jede CPU hat ihr eigenes Betriebssystem:
Der Speicher wird in Partitionen (pro CPU/Betriebssystem) aufgeteilt.
2. Asymmetrische Multiprozessoren (*asymmetric multiprocessing, AMP*):
Das Betriebssystem läuft nur auf einer einzigen CPU (=Master), die Anwendungsausführung nutzt alle restlichen Prozessoren (=Slaves).
3. Symmetrische Multiprozessoren (*symmetric multiprocessing, SMP*):
Nur eine einzige Kopie des Betriebssystems liegt im Speicher. Diese ist von jeder CPU ausführbar.

Am einfachsten ist die Lösung, dass jede CPU ihr eigenes Betriebssystem hat und der Speicher partitionsweise den einzelnen CPUs zugeteilt wird, womit die einzelnen Prozessoren unabhängig voneinander arbeiten. Der Code des Betriebssystems (siehe Abb. 1–9) ist nur einmal im Speicher abgelegt, da unveränderlich. Infolge des fehlenden Lastausgleichs und der fixen Speicheraufteilung skaliert diese Lösung schlecht und hat deswegen keine nennenswerte Verbreitung gefunden.

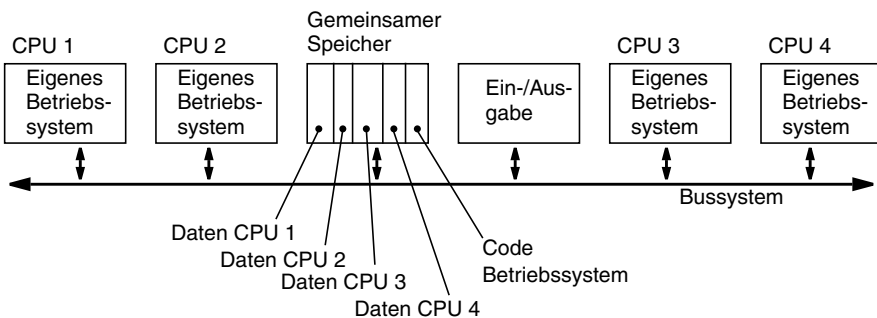


Abb. 1–9 Jede CPU hat ein eigenes Betriebssystem.

Die Variante der *asymmetrischen Multiprozessoren* weist alle Betriebssystemaktivität einer bestimmten CPU zu, die damit der Chef (Master) wird. Die Anwenderprozesse laufen auf den restlichen CPUs, die man Slaves nennt (siehe Abb. 1–10). Vorteilhaft ist die Möglichkeit der flexiblen Zuteilung ablaufwilliger Prozesse an die einzelnen Slaves. Der wesentliche Nachteil ist aber der Flaschenhals, der durch den Master entsteht, da alle Systemaufrufe nur von dieser CPU bearbeitet werden. Nimmt man beispielsweise an, dass die Anwenderprozesse 13% der Zeit in Systemaufrufen verweilen, so entstehen infolge des Flaschenhalses bereits bei

einem System mit 8 Prozessoren Wartezeiten. Es handelt sich also um eine Lösung, die nur für kleine Prozessoranzahlen sinnvoll ist.

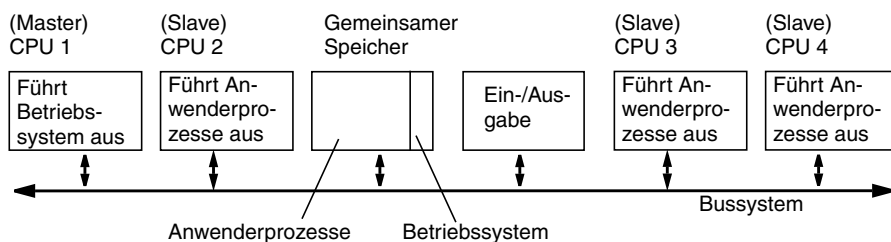


Abb. 1-10 Asymmetrisches (Master/Slave-) Multiprozessorsystem

Die Lösung mit *symmetrischen Multiprozessoren* führt das Betriebssystem genau einmal im Speicher, und zwar sowohl den Code als auch die Daten (siehe Abb. 1-11). Systemaufrufe können von allen CPUs ausgeführt werden. Da die Daten des Betriebssystems für alle gemeinsam zugreifbar sind, entfällt damit der Flaschenhals der Master-CPU.

Allerdings stellt sich damit auch das Problem des koordinierten Zugriffs auf die Systemdaten, um Dateninkonsistenzen zu vermeiden (kritische Bereiche). Der einfachste Weg wäre der, dass zu jedem Zeitpunkt nur eine einzige CPU einen Systemaufruf ausführen darf. Damit wäre das Flaschenhalsproblem aber in einer neuen Form wieder vorhanden und die Leistung limitiert. In der Praxis genügt es aber, wenn die einzelnen Systemtabellen separat abgesichert werden. Systemaufrufe auf unterschiedlichen Tabellen können dann parallel stattfinden. Da viele Systemtabellen in verschiedener Beziehung voneinander abhängen, ist die Realisierung eines derartigen Betriebssystems sehr anspruchsvoll (z.B. Deadlock-Problematik).

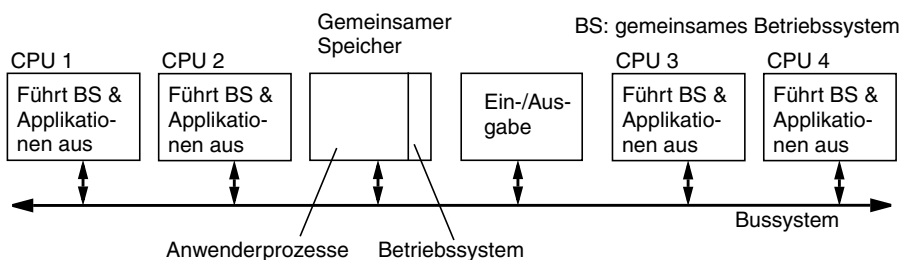


Abb. 1-11 Symmetrisches Multiprozessor-(SMP-) System

1.5.7 Verteilte Betriebssysteme

Verteilte Betriebssysteme nutzen eine Menge von über ein Netzwerk verbundenen Rechnern zur Lösung größerer Aufgaben oder einfach, um eine gemeinsame

Rechenplattform in größerem Rahmen zu realisieren. Idealerweise wird das Betriebssystem so realisiert, dass *Ortstransparenz* herrscht. Dies bedeutet, dass der Benutzer nur ein einziges System sieht (*Single System Image, SSI*), egal an welchem der teilnehmenden Rechner er sich momentan angemeldet hat. Solche Betriebssysteme realisieren die rechnerübergreifende Kommunikation systemintern und unterstützen Mechanismen zum Lastausgleich zwischen den einzelnen Rechnern wie auch Ausfallredundanz. Die verwandten Clustersysteme hingegen unterstützen die Ortstransparenz nur partiell oder gar nicht, womit sie partiell Fähigkeiten verteilter Betriebssysteme besitzen.

1.5.8 Beispiele von Systemarchitekturen

Unix System V

Der innere Aufbau des Unix-Betriebssystems (System V Release 3 hier als Beispiel betrachtet) spiegelt die zwei zentralen Unix-Konzepte *Dateien (files)* und *Prozesse (processes)* über entsprechende Subsysteme wider. Diese sind in Abbildung 1–12 als klar abgegrenzte logische Blöcke zu sehen.

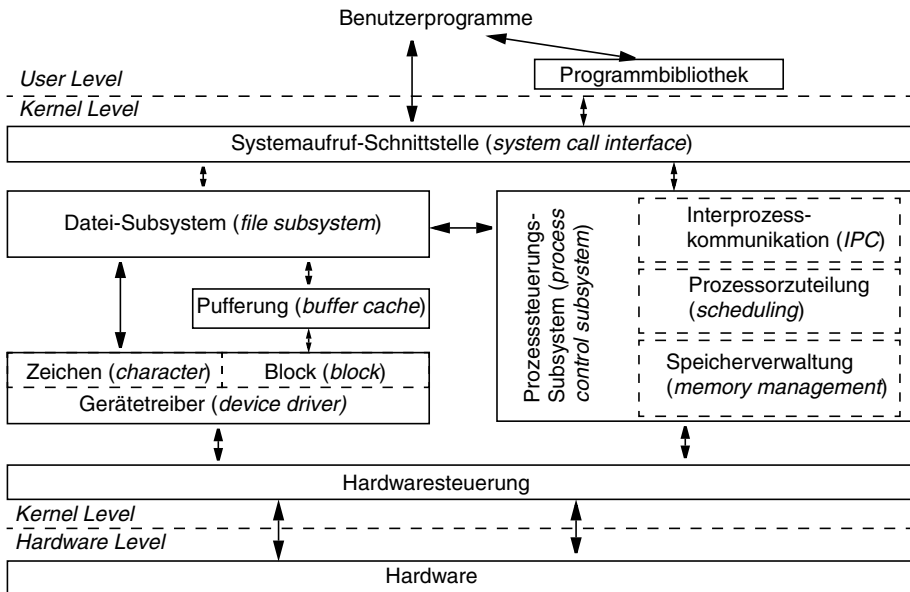


Abb. 1–12 Interne Struktur des Unix (Kern des System V Release 3)

In der Realität sind die Abgrenzungen aber nicht so eindeutig, da einige Module auf interne Funktionen anderer Module einwirken (monolithische Struktur). Die Architektur teilt sich in die drei Schichten *Benutzerebene (user level)*, *Kernebene (kernel level)* und *Hardwareebene (hardware level)* auf. Zuerst stehen die

Benutzerprogramme, die entweder direkt (über Trap-Interrupt von Assemblersprache) oder mithilfe von API-Funktionen aus einer Programmbibliothek (Hochsprache) die Systemdienste nutzen. Für die Interrupt-Verarbeitung liegen Kernroutinen vor, die bei einer Programmunterbrechung aufgerufen werden. Erwähnenswert ist, dass viele Unix-Kommandos gleichartig wie Benutzerprogramme implementiert sind, indem sie als ausführbare Dateien vorliegen und die Systemprogrammierschnittstelle zur Kommunikation mit dem Kern benutzen (Beispiel: Unix Kommandointerpreter, *shells*). Dadurch kann der Kern kompakt und überschaubar gestaltet werden. Die Steuerung von Peripheriegeräten erfolgt durch die Treiber, die entweder zeichenorientiert arbeiten (*character device driver*) oder ganze Datenblöcke manövrieren (*block device driver*). Letztere Gruppe von Treibern kann mithilfe eines Puffers Daten sowohl beim Lesen als auch beim Schreiben zwischenspeichern. Unix ist in C programmiert, ergänzt mit wenigen hardwarenahen Teilen in der Assemblersprache der unterliegenden Hardwareplattform.

Windows 10

Das Betriebssystem Windows 10 besitzt eine Architekturmischform, die sowohl Elemente der Mikrokernidee als auch der geschichteten Strukturierung realisiert (siehe Abb. 1–13). Jedoch unterscheidet es sich nicht groß von vielen Unix-Systemen, indem wesentliche Teile des Systemcodes einschließlich der Treiber in der gleichen Ausführungsumgebung ablaufen und damit unter sich keinen Schutz gegen Fehlzugriffe genießen. Hingegen sind viele Dienste und Hilfsfunktionen in separate Prozesse ausgelagert und daher genauso gegeneinander geschützt wie Benutzerprozesse unter sich. Alle Systemteile im Kernmodus bzw. die in Systemdienstprozesse ausgelagerten Teile sind gegen böswillige Benutzerprozesse abgeschottet. Damit unterscheidet sich Windows 10 deutlich von früheren Windows-Produkten der Reihe 3.x/95/98/ME, die keinen vollständigen Schutz des Systemcodes vor Manipulationen durch fehlerhafte Applikationen boten. Dies ist jedoch eine fundamentale Anforderung für ein stabiles und robustes Betriebssystem. Windows 10 ist in C, zu kleineren Teilen in C++ programmiert. Wenige Softwareteile, die direkt die Hardware ansprechen, sind auch in Assemblersprache codiert.

Nachdem Windows über viele Jahre nur als Gesamtsystem installier- und ladbar war, wurde mit Windows Server 2008 eine Version ohne grafische Oberfläche geschaffen (*windows core*), da ein GUI für den Serverbetrieb nicht unbedingt notwendig ist. Mit Windows 7 wurde zudem ein *MinWin* definiert, das nur aus dem eigentlichen Windows Kernel, den Netzwerkprotokollen, dem Dateisystem und einem minimalen Satz von Diensten (*core services*) besteht und in 40 MB Hauptspeicher Platz findet. MinWin kann unabhängig vom restlichen Windows-Code geladen und getestet werden, womit die höheren Schichten des Systems besser abgekoppelt werden.

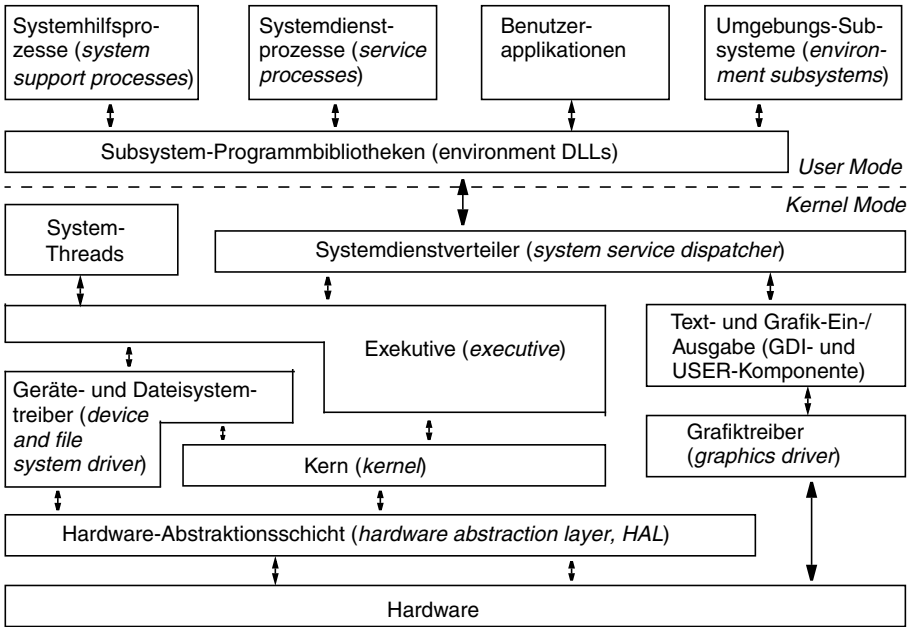


Abb. 1-13 Interne Struktur von Windows 10

Entsprechend wurde auch systemintern eine *MinWin API* definiert, die nun von den höheren Schichten genutzt wird. Windows 10 skaliert gut über die Plattformgröße, indem für Kleinsysteme aus der Embedded-Welt bis zu Servermaschinen passende Systemvarianten verfügbar sind.

Ursprünglich existierten drei Subsysteme, jeweils eines für OS/2, Unix und Win32. Aktuell wird noch das Subsystem für Win32 und neu das *Windows Subsystem for Linux (WSL)* unterstützt. Das WSL erlaubt es, eine angepasste Linux-Distribution (z.B. Ubuntu oder SUSE) unter Windows auszuführen, wobei aber kein Linux-Kernel benutzt wird. Stattdessen wird dessen Funktionalität mithilfe von Adaptions- und Hilfskomponenten direkt durch den NT-Kernel erbracht.

Google Chrome OS

Google realisiert mit diesem Betriebssystem eine neue Idee, wie Anwender mit Programmen arbeiten, nämlich webbasiert. Man ist auch versucht zu sagen, dass das Chrome OS die Versprechungen einlöst, die für das Web 2.0 gemacht wurden. Die Architekturidee besteht im Wesentlichen darin, aus serverbasierten Applikationen (*Cloud-Services, Cloud Computing*) und kostengünstigen Netbooks eine Systemlösung zu realisieren, die übliche Anwendungen, wie Office-Programme, Kalender- und Informationsdienste, dem Anwender als Webapplikationen zur Verfügung stellt, ohne dass er diese Applikationen auf seinem Compu-

ter installieren muss. Der Zugriff auf die Cloud-Services erfolgt via Chrome-Webbrowser. Traditionelle GUI-Applikationen lassen sich nicht installieren, da als Benutzeroberfläche der Webbrowser dient, der seinerseits auf einem Linux-Kernel aufsetzt. Im Jahr 2014 wurde die Möglichkeit integriert, *Chrome Apps* zu installieren, die im *Chrome Web Store* angeboten werden. Nachfolgend wurde 2016 zudem die Installation von *Android Apps* ergänzt, die aus dem *Google Play Store* bezogen werden.

Damit der Benutzer zwischendurch ohne Internetanbindung arbeiten kann, ermöglicht ihm das ebenfalls von Google stammende Browser-Plug-in *Gears*, seine Daten lokal zu speichern. Um einen möglichst schnellen Arbeitsbeginn zu erreichen, setzt das Chrome OS auf einer umfangreichen Firmware auf, die für eine blitzartige Initialisierung der Hardware sorgt. So gesehen liegt eine Dreischichten-Architektur vor: zuunterst die Firmware, dann der Linux-Kernel und zuoberst der Chrome-Webbrowser. Das Chrome OS existiert auch in einer Open-Source-Version, die sich *Chromium OS* nennt und konsequenterweise den Chrome-Browser durch den Chromium-Browser ersetzt. Zudem ist es in verschiedenen Distributionen erhältlich, die oft funktionale Erweiterungen integrieren.

1.5.9 Zukünftige Systemarchitekturen aus Sicht der Forschung

Betriebssysteme bieten viele Ansatzpunkte für Verbesserungen. Aus Benutzersicht ist die Bedienoberfläche das prägende Element eines Betriebssystems. Aus Systemsicht sind die Architektur bzw. Entwurfsprinzipien, die auf die Lösung bekannter Probleme abzielen, von größerem Interesse. Die Gestaltung von Benutzeroberflächen ist ein umfangreiches Wissensgebiet der Informatik, wofür auf entsprechende Spezialliteratur verwiesen sei. Zur Systemarchitektur und ihren Aspekten wird hingegen nachfolgend eine Auswahl an Forschungsarbeiten vorgestellt, die illustrieren, wohin die Entwicklung von Betriebssystemen jenseits der Benutzeroberfläche in Zukunft gehen könnte. Dazu stellen wir eine Reihe von Fragen, auf die mögliche Antworten gefunden wurden.

Auf welcher Schicht sollen Systemfunktionen implementiert werden? Schichten (*layers*) sind ein beliebtes Strukturierungsmittel für Software, wobei die Idee der geschichteten Systeme nahelegt, eine bestimmte Funktion in schichtenspezifische Teilfunktionen aufzuteilen. Die *End-to-End Arguments* von J. H. Saltzer et al. (1984) sagen dagegen aus, dass Funktionen auf tiefer Abstraktionsstufe oft zu teuer sind für ihren Nutzen sowie dass es Funktionen gibt, die nur mit Wissen bzw. mithilfe der Applikation an den Kommunikationsendpunkten gelöst werden können. Diese Beobachtung gilt für Kommunikationssysteme generell, stellt aber auch die Grundlage des Internets dar. Ähnliche Aussagen zu Betriebssystemen wurden von B. W. Lampson bereits 1974 gemacht, wobei er betont, dass Funkti-

onen nie fix auf tiefer Stufe gelöst sein sollten, sondern sich stets durch die Applikation mit einer spezialisierteren Version ersetzen lassen. Teilfunktionen auf tiefer Stufe können hingegen zu hoher Effizienz beitragen.

Wie lassen sich Betriebssysteme flexibel erweitern? Traditionelle Betriebssysteme begrenzen die Performanz, Flexibilität und Funktionalität von Applikationen durch ihre fixen Schnittstellen und Implementierungen von Abstraktionen, wie z.B. Interprozesskommunikation und virtueller Speicher. Die Idee der *Extensible Systems* ist, dass ein minimaler vertrauenswürdiger Kern die Hardwareressourcen via elementare Schnittstellen an nicht vertrauenswürdige Betriebssysteme exportiert. Letztere sind als Programmbibliotheken implementiert (*Library Operating System, LOS*). Die LOS realisieren Systemobjekte und Systemstrategien und laufen im Benutzermodus im abgeschotteten Speicher des sie benutzenden Prozesses. Damit lassen sich Dienste einfacher und applikationsangepasster realisieren, womit sie effizienter erbracht werden. Applikationen können eigene Abstraktionen definieren oder die vom Kern angebotenen minimalen Abstraktionen erweitern oder spezialisieren. Eine *Proof-of-Concept*-Implementierung, genannt *ExoKernel*, wurde von D. R. Engler und M. F. Kaashoek im Jahr 1995 realisiert und mit dem kommerziellen Unix-System Ultrix verglichen, womit die Effizienzsteigerung eindrücklich belegt wurde. Die Idee der Library Operating Systems wurde in einer Nachfolgearbeit unter einem anderen Blickwinkel erforscht, nämlich der Isolierung, wie dies sonst nur mit *Virtual Machine Monitors* (VMM, siehe Abschnitt 12.2.7) erreichbar ist. Das Resultat ist das Betriebssystem *Drawbridge* (D. E. Porter et al., 2011) das die Architektur von Windows 7 so umbaut, dass ein *Security Monitor* Systemfunktionalität, wie Dateisysteme, Netzwerk-Protokollstapel und Peripheriegerätetreiber, bereitstellt, auf die dann die eigentlichen Library Operating Systems aufsetzen. Die LOS realisieren den Großteil der restlichen Systemdienste, sodass im Security Monitor nur ein kleiner Teil des gesamten Betriebssystems, entsprechend 2% des Gesamtcode-Umfangs, gemeinsam ist. Im Gegensatz zu VMM-Lösungen werden erheblich weniger Ressourcen benötigt, wenn das gleiche Betriebssystem stark isoliert mehrere Applikationsumgebungen realisieren soll. Drawbridge ermöglicht ferner eine sehr einfache Applikationsmigration während des Betriebs, vergleichbar mit VMM-Lösungen.

Wie kann ein Betriebssystem sicher erweitert werden? Es gibt Applikationen, für die vorgefertigte Betriebssystemdienste bzw. Betriebssystemschnittstellen schlecht passen. Mit dem Experimental-Betriebssystem *SPIN* (B. N. Bershad et al., 1995) wurde die Idee realisiert, mithilfe einer Infrastruktur zur Systemerweiterung, eines Grundsatzes an erweiterbaren Systemfunktionen und der Verwendung einer typischeren Hochsprache den Applikationen die Spezialisierung von Systemdiensten zu ermöglichen. Die Erweiterungen werden dabei beim Laden oder während des Betriebs dynamisch in vom restlichen Kern logisch getrennte Domänen

(*logical protection domains*) eingebunden. SPIN zeigt, dass eine effiziente Implementierung eines erweiterbaren Betriebssystems bei vollem Schutz möglich ist.

Wie wird die Ausführung unsicherer Codes verhindert? Moderne Sprach-Laufzeitsysteme zeigen, dass mit typsicheren Hochsprachen (z.B. Java, C#) und komplementären Prüfmechanismen bei der Compilierung, beim Laden des Zwischen-codes (z.B. Java-Bytecode, .NET MSIL) und während der Programmausführung nicht erlaubte Zugriffe auf Code und Daten verhindert werden. Leider sind damit Applikationen, die in weniger sicheren Sprachen programmiert wurden, immer noch ein Problem, da sie evolutionär entstanden sind und eine Portierung in eine sichere Sprache unattraktiv ist. Bereits 1996 wurde von G. C. Necula und P. Lee eine Lösung präsentiert, die basierend auf einer *Safety Policy* den Binär-code (*Proof Carrying Code, PCC*) vor der Ausführung prüft, ob er sicher ist. Damit diese Prüfung schnell abläuft, wird dem Binär-code ein kryptografisches Zertifikat mitgeliefert, das eine Sicherheitsüberprüfung ohne detaillierte Codeanalyse erlaubt. G. Morrisett et al. definierten 1999 einen typsicheren Instruktionssatz (*Typed Assembly Language, TAL*), der die Erzeugung typsicherer Binärprogramme aus Hochsprachen als PCC erlaubt.

Kann typsicherer Code die Isolation durch Hardwaremechanismen bei voller Sicherheit ersparen? Traditionelle Betriebssysteme isolieren Prozesse mittels der hardwareunterstützten Mechanismen der MMU (virtueller Speicher) und der CPU (Unterscheidung Benutzer-/Kernmodus). Nachteilig ist, dass jeder Prozesswechsel dadurch zusätzliche Zeit benötigt. 2006 wurde von M. Aiken et al. das Forschungsbetriebssystem *Singularity* vorgestellt, das auf typsicheren Sprachen beruht und ohne Hardwaremechanismen volle Sicherheit gewährleistet, womit es im Vergleich zu herkömmlichen Systemen effizienter abläuft. Programme laufen als *Software Isolated Processes (SIP)* ab und stellen geschlossene Objekträume (*Closed Object Spaces*) dar, da Prozesse indirekt via *Exchange Heap* kommunizieren, wobei der Sender die Referenz auf ein abgelegtes Objekt zwangsweise verliert, bevor dem Empfänger eine Zugriffsreferenz übergeben wird. Das Betriebssystem selbst ist als minimaler vertrauenswürdiger Kern und eine Menge von SIP für höherwertige Systemfunktionen realisiert. Da der Prozesswechsel und die Interprozesskommunikation sehr effizient sind, lassen sich auch Treiber als SIP realisieren, ohne dass die Performanz leidet.

Wie skaliert man Betriebssysteme für Manycore- und Cloud-Systeme? Verbreitete Betriebssysteme wurden für Plattformen entworfen, die nur über einen oder wenige Rechenkerne verfügen. Mit dem *Factored Operating System (fos)* zeigen D. Wentzlaff et al. (2010), dass sich ein verteiltes Betriebssystem hoch skalierbar realisieren lässt, wenn das Gesamtsystem in viele Komponenten aufgeschlüsselt wird, die je für sich Dienste anbieten, die selbst wiederum als eine Dienstmenge (als *fleet* bezeichnet) auf die teilnehmenden Rechner verteilt sind. Je nach aktu-

eller Nachfrage werden diese Dienstmengen ausgeweitet oder geschrumpft. Im Gegensatz zu bekannten *Infrastructure-as-a-Service-(IaaS)*-Lösungen werden Ressourcen in einer einheitlichen, einfach skalierbaren Art und Weise angeboten.

Wie fehlerfrei sind verbreitete Betriebssysteme programmiert? Betriebssysteme sind komplexe Softwareprodukte und entsprechend anfällig für Entwurfs- und Programmierfehler. M. M. Swift et al. haben 2003 die Ursachen für Systemabstürze des Microsoft Windows XP untersucht und dabei festgestellt, dass diese zu 85% durch Treiber verursacht wurden. Treiber stellen jedoch Plug-in-Komponenten eines Betriebssystems dar, die größtenteils durch Drittparteien programmiert werden. Als eine Konsequenz hat Microsoft eine neue Treiberschnittstelle (WDM, siehe Abschnitt 7.3.7) zur Komplexitätsreduktion der Treiberentwicklung eingeführt. 2011 wurden von N. Palix et al. im Linux-Kernel 2.6.33 durch systematische Analysen 736 Fehler identifiziert. Ergänzend dazu steht ihre Beobachtung, dass über 10 Jahre hinweg der Codeumfang des Linux Kernel sich mehr als verdoppelt hat, jedoch die Fehleranzahl in etwa gleich geblieben ist, was für eine deutliche Steigerung der Softwarequalität spricht.

12 Virtualisierung

Lernziele

- Sie unterscheiden sieben Arten der Virtualisierung und beurteilen ihre Nutzen und Anwendung.
- Sie zählen drei Anforderungen an einen Hypervisor auf.
- Sie erläutern die Unterschiede zwischen einer Computervirtualisierung mit und ohne Hardware-Unterstützung sowie der Paravirtualisierung.
- Sie ordnen einen konkreten Hypervisor einer der zwei Grundtypen zu.
- Sie setzen einen Hypervisor in sechs unterschiedlichen Szenarien ein.
- Sie vergleichen Container-Systeme mit Unikernels.

Die Virtualisierung von Rechnerressourcen bringt in verschiedener Hinsicht neue Nutzen. Sie kann auf verschiedenen Stufen erfolgen und ist bereits teilweise in den Konzepten des Prozessmodells und des virtuellen Speichers realisiert. Wir geben einen Überblick über bekannte Virtualisierungsformen und ergänzen diese um neue Virtualisierungskonzepte, wobei wir auf den *Virtual Machine Monitor (Hypervisor)* aufgrund seiner großen Bedeutung in der Praxis näher eingehen.

12.1 Anwendungsbereiche

Bei der Virtualisierung wird eine tatsächlich vorhandene Ressource (ganzer Rechner oder Teile davon) durch eine scheinbar vorhandene Ressource ersetzt, die für den Ressourcennutzer nicht von dem realen Objekt unterscheidbar ist. Die Ziele, die dabei verfolgt werden, sind unterschiedlich. Zum Beispiel ermöglicht eine Abstraktion von der Realität eine einfachere und einheitlichere Nutzung. Oder mithilfe der Virtualisierung lassen sich parallele Abläufe besser isolieren, womit Schutzziele einfacher einzuhalten sind. Ist zudem eine einzelne Ressource für die Mehrfachverwendung nicht vorgesehen, so kann sie über eine Virtualisierung dennoch mehreren Nutzern zugänglich gemacht werden. Nachfolgend befassen

wir uns exemplarisch mit einer Reihe von Virtualisierungsansätzen im Umfeld des Betriebssystems:

- *Virtuelle Prozessoren*: Simulation einer CPU-Hardware durch Software
- *Virtuelle Prozessumgebungen*: Nutzung virtueller Adressräume für Prozesse (Raummultiplex) zusammen mit CPU-Scheduling (Zeitmultiplex)
- *Virtuelles Betriebssystem*: Ausführung unveränderter (Binär-)Programme eines Betriebssystems A unter einem Betriebssystem B
- *Virtueller Desktop*: entfernter Zugriff auf einen grafischen Desktop über das Netzwerk
- *Virtuelle Ressourcen*: Sekundärspeicher- und Netzwerkvirtualisierung
- *Sandboxing*: virtuelle Prozesslaufzeitumgebungen inklusive Betriebssystem
- *Virtuelle Computer*: Virtualisierung der gesamten Computerhardware mithilfe eines Virtual Machine Monitor (Hypervisor)

12.2 Virtualisierungstypen

12.2.1 Virtuelle Prozessoren

Diese Form der Virtualisierung simuliert den Instruktionssatz eines Prozessors und erlaubt damit die Ausführung von Software, die für eine *andere Prozessorfamilie* geschrieben wurde. Dazu gehört die Bereitstellung eines virtuellen Hauptspeicherbereichs, damit die simulierte CPU überhaupt erst Programme ausführen kann. Diese Lösung wird auch *Emulation* genannt und manchmal als *virtuelle Maschine* bezeichnet, da sie Anweisungen der Wunschplattform dynamisch in Anweisungen der Gastgeberplattform umsetzt. Ein frei erhältliches Produkt für die PC- und weitere Plattformen ist *QEMU (Quick Emulator)*. Codeteile aus QEMU kommen auch in einigen Virtual Machine Monitors (siehe Abschnitt 12.3) vor, beispielsweise um dort fehlende Hardware-Unterstützung in Software nachzubilden. Prozessorsimulatoren werden in der Entwicklung von Mikrocontroller-Software gerne benutzt, um Programme auf einem komfortablen Entwicklungssystem auszutesten, bevor sie auf die echte Hardware übertragen werden.

Ein anderer Anwendungsbereich besteht in der *interpretierten Ausführung* von Programmen. Beispielsweise werden Java-Quellcodeprogramme durch den Java-Compiler in einen Zwischencode (Bytecode) übersetzt, der dann von der *Java Virtual Machine (JVM)* interpretiert wird. Die JVM simuliert einen hypothetischen Java-Prozessor, der einen Maschinenbefehlssatz realisiert, der objektorientierte Programme optimal unterstützt. Dadurch, dass die JVM-Software für unterschiedliche Rechnerplattformen implementiert wurde, können im Bytecode vorliegende Programme ohne Änderungen auf all diesen Systemen ausgeführt werden. Dies spielt für über das Internet verteilte Programme eine große Rolle.

Moderne JVMs bzw. Java-Compiler bieten die Möglichkeit, Java-Quellcodeprogramme direkt in Maschinencode der Zielplattform zu übersetzen. Damit kann die Rechenleistung gesteigert werden, die bei der Simulation einer CPU naturgemäß unter den Werten einer direkten Ausführung des plattformspezifischen Maschinencodes liegt.

Ein mit der JVM vergleichbarer Ansatz ist in dem Laufzeitsystem der *Microsoft .NET-Plattform* zu finden. Bei dieser Lösung interpretiert die *Common Language Runtime (CLR)* Anweisungen einer Zwischensprache, die sich *Microsoft Intermediate Language (MSIL)* nennt. Damit diese Plattform viele Sprachen unterstützen kann, wird sie durch ein *Common Type System (CTS)* ergänzt, das für die unterschiedlichen Datentypen die möglichen Werte und ihre Darstellung verbindlich festlegt.

WebAssembly, kurz auch als WASM bezeichnet, realisiert einen virtuellen Prozessor, der innerhalb eines Web-Browsers läuft. Im Vergleich zu Webanwendungen in JavaScript können damit höhere Ausführungsgeschwindigkeiten erreicht werden. Der Quellcode einer Webanwendung (z.B. in C, C++, C# oder Rust) muss zuerst in den WASM-Binärcode übersetzt werden, bevor er ausführbar ist. Heute enthalten die meisten Web-Browser ein WASM-Modul. Einen Schritt weiter zu einer universellen Ablaufumgebung stellt das *WebAssembly System Interface (WASI)* dar, da es die Ausführung von plattformunabhängigem Code außerhalb des Web-Browsers unterstützt. Dazu wird ein virtuelles Betriebssystem definiert, das Systemdienste unabhängig von der unterliegenden Plattform anbietet.

12.2.2 Virtuelle Prozessumgebungen

Moderne leistungsfähige Betriebssysteme unterstützen das *Prozessmodell*, ergänzt um einen *virtuellen Speicher*. Damit können Applikationen unabhängig voneinander entwickelt und isoliert nebeneinander auf dem gleichen Rechner ablaufen. Die Funktionsweise wurde bereits an mehreren Stellen in diesem Buch erläutert, z.B. in den Abschnitten 4.2 und 8.5.

12.2.3 Virtuelles Betriebssystem

Die binäre Applikationsprogrammierschnittstelle des Betriebssystems (*Application Binary Interface, ABI*), also die Systemprogrammierschnittstelle, lässt sich ebenfalls virtualisieren. Dies ermöglicht die Ausführung von Programmen, die für das Betriebssystem A geschrieben wurden, unter einem Betriebssystem B. Ein Beispiel dafür ist die Software *WINE (Wine Is Not an Emulator)*, die es erlaubt, Microsoft Windows-Programme unter Unix mit dem X-Window-System auszuführen, ohne dass sie neu übersetzt oder sonstwie geändert werden müssen. Zurzeit werden Linux, Solaris und diverse BSD-Unix-Systeme unterstützt. Ein

zweites Beispiel ist die *Classic-Umgebung* des Apple Mac OS X, die anfänglich benutzt wurde, um ältere Programme des Mac OS 9 unter Mac OS X auszuführen.

12.2.4 Virtueller Desktop

Wird die Bedienoberfläche eines Rechners A auf einen Rechner B abgebildet, so nennt sich dies ein *virtueller Desktop*. Er unterstützt dieselben Bedienmöglichkeiten eines Systems, aber auf der Hardware eines Zweitsystems. In der Windows-Welt ist dies die Software *Remote Desktop Client (RDP)*, mit der die entfernte Bedienung eines Windows-Rechners ermöglicht wird. Die RDP-Software lässt sich auf unterschiedlichen Plattformen installieren (z.B. Windows, Android, iOS, Mac OS X). Vergleichbare Möglichkeiten bietet in der Unix-Welt das Produkt *VNC (Virtual Network Computing)* oder das Protokoll *XDMCP (X Display Manager Control Protocol)*, mit dem sich auf X beruhende grafische Desktops auf entfernte Systeme umlegen lassen. Diese Virtualisierungsart unterstützt *Thin-Client-Konzepte* auf sehr einfache Art und Weise. So kann ein Serversystem so aufgesetzt werden, dass es vielen Benutzern gleichzeitig einen individuellen grafischen Desktop auf ihren Arbeitsplatzrechnern anbietet. Ein entsprechendes Produkt sind die Microsoft Remotedesktopdienste, die im Windows Server 2012 enthalten sind.

12.2.5 Virtuelle Ressourcen

Eine sehr einfache Lösung beschränkt sich auf die Partitionierung der Ressourcen, indem diese einer Strategie folgend auf nachfragende Prozesse aufgeteilt werden. Speziell auf Serversystemen spielt die Verfügbarkeit von Diensten eine große Rolle. Diese kann dadurch gefährdet sein, dass eine Applikation bei einem Fehlverhalten alle anderen Applikationen behindert, indem sie übermäßig Ressourcen belegt. Dadurch entsteht für andere Dienste ein Ressourcenengpass, der letztlich die Sicherstellung der Dienstgüte verhindert. Ein Lösungsansatz besteht darin, einen speziellen Ressourcenmanager in das Betriebssystem zu integrieren, der für die Ressourcennutzung ein *Quota-System* realisiert. Die Quotenregelung kann auf der Stufe einzelner Prozesse oder Prozessgruppen erfolgen. Werden mehrere Ressourcen für die Quotenverwaltung zusammengefasst, so spricht man von *Ressourcenbehältern (resource containers)*.

Weiter gehende Lösungen nehmen sich spezifischer Ressourcenarten an, wie sie durch Peripheriegeräte realisiert werden. Damit wird die entsprechende Peripherie vom Kernsystem abgetrennt und mittels eigener, d.h. vom Betriebssystem unabhängiger Verfahren verwaltet und über eine neu definierte Schnittstelle dem Kernsystem angeboten. Dies wird vor allem für Sekundärspeicher (HDD, SSD) und Netzwerke benutzt.

Virtuelle Sekundärspeicher (storage virtualization) nehmen sich der Bedürfnisse großer Unternehmungen an, die Bedarfe an Sekundärspeicherkapazität im mehrstelligen Terabyte-Bereich haben. Zudem muss Verfügbarkeit, Archivierung, Datensicherung und gemeinsame Datennutzung sichergestellt werden. Konventionell an Rechner angeschlossene Festplattenlaufwerke sind in einem vernetzten Umfeld mit sehr vielen Rechnern nicht ideal. Zunehmend werden deshalb eigene Speichersubnetze eingesetzt, an denen viele Festplattenlaufwerke angeschlossen sind (*Storage Attached Network, SAN*). Ein SAN dient nur für den Speicherzugriff. Zur flexiblen Verwaltung wird eine Sekundärspeichervirtualisierung eingesetzt. Das heißt, alle verfügbare Festplattenspeicherkapazität erscheint als ein einziges unternehmensweites Dateisystem (*common file system, SAN file system*). Dazu sind entsprechende Dateisystemtreiber für alle eingesetzten Betriebssysteme nötig. Eine etwas weniger weitgehende Lösung besteht darin, dass virtuelle Laufwerke (*virtual volumes*) den einzelnen Nutzern im Stil klassischer Netzwerkdateisysteme angeboten werden.

Virtuelle Netzwerke (*network virtualization*) realisieren, aufbauend auf realen Netzwerken, eigene Verbindungskanäle mit wohldefinierten Verbindungspartnern. Sie können eigene Protokolle nutzen, die zum Beispiel sicherer sind als das reale Netzwerkprotokoll. Zusätzliche Ziele, wie eine kontrollierte Aufteilung der verfügbaren Netzwerkbandbreite, können helfen, die Dienstgüte zu verbessern, sofern alle Kommunikationsteilnehmer nur die virtuellen Netzwerkverbindungen nutzen. Die Netzwerkvirtualisierung kann sich auch nur auf die realen Netzwerkschnittstellen eines einzelnen Computers beschränken. In diesem Fall handelt es sich eigentlich um eine Virtualisierung der Netzwerkschnittstelle und nicht des ganzen Netzes, da auf das Netz hinaus nur die vorhandenen realen Protokolle genutzt werden. Die rechnerexternen Kommunikationspartner merken daher nichts von der Virtualisierung. Rechnerlokal können jedoch vergleichbare Ziele wie bei der Netzwerkvirtualisierung angestrebt werden.

12.2.6 Sandboxing (virtuelles Laufzeitsystem)

Die Laufzeitumgebung einer Applikation wird vom restlichen System abgeschirmt, d.h. virtualisiert (*Container-based Operating Systems, COS*). Diese Möglichkeit wird von einigen Betriebssystemen unterstützt, zum Beispiel von Oracle Solaris mit den *Containers*, vom FreeBSD-Unix mit den *Jails* und von Linux mit dem *Vserver*. Dasselbe Konzept wird aber beispielsweise auch in Webbrowsern im Kleinen genutzt, um einzelne Webseiten oder Plug-ins zu isolieren. Beim Sandboxing sieht jeder Prozess nur sich selbst und das Betriebssystem, wodurch eine wechselseitige Beeinflussung von Applikationen untereinander verhindert wird. Dazu gehört, dass jedem Anwenderprozess nur ein begrenzter Anteil an der Nutzung gemeinsamer Ressourcen gewährt wird. Man könnte auch sagen, dass jede Sandbox ein separater *User Space* darstellt, der sich mit anderen

Sandboxen lediglich den geschützten *Kernel Space* teilt. Erwähnt sei noch, dass die im Unix-Bereich klassische Lösung der *chroot-Umgebung* nicht dieselbe Isolation gewähren kann wie das Sandboxing-Konzept, da die damit realisierbare Isolierung unvollständig ist.

Virtual Hosting

Anwendung findet das Sandboxing bei Anbietern von *Virtual Hosting*, zur erhöhten Isolierung kritischer Applikationen bei sich wechselseitig nicht trauenden Systembenutzern, zur feineren Kontrolle der Ressourcenzuteilung und zur Ermöglichung einfacher Migrierungen von Applikationen. Im Vergleich zu Lösungen mit einem *Virtual Machine Monitor* (siehe Abschnitt 12.3) ist der Zusatzaufwand gering, da stets dieselbe Systemprogrammierschnittstelle genutzt wird und spezielle Hardware unnötig ist. Allerdings wird nicht dieselbe Flexibilität erreicht, da nur Applikationen für dieselbe Betriebssystemplattform lauffähig sind.

Container-Systeme

Für den betrieblichen Einsatz ist eine möglichst einfache Bereitstellung von Anwendungen wichtig, die voneinander isoliert ablaufen. Dies ermöglichen *Container-Systeme*, wie z.B. *Docker* oder *rkt*, die beide auf Linux aufsetzen. Dazu werden Anwendungen, wie z.B. ein Webserver oder ein Datenbanksystem, mithilfe entsprechender Werkzeuge als Standardpakete bereitgestellt. Werden diese Pakete, als *Abbilder (Images)* bezeichnet, auf einer webbasierten Plattform allgemein nutzbar bereitgestellt, so benötigt eine Installation nur noch wenige Arbeitsschritte. Jedes Abbild, als einzelne Datei verfügbar, enthält nicht nur die Anwendung selbst, sondern auch alle Drittsoftware, die bei traditioneller Bereitstellung zusätzlich zu installieren wäre. Für *Docker-Images* ist dies mit dem *Docker Hub* realisiert, der als Onlinedienst sowohl eine öffentliche wie auch eine private Bereitstellung erlaubt.

12.2.7 Virtuelle Computer (Stufe Computerhardware)

Den Kern eines virtuellen Computers bildet der *Virtual Machine Monitor (VMM)* bzw. *Hypervisor*. Er sorgt dafür, dass eine wählbare Anzahl von *Virtual Machines (VMs)* konfiguriert werden kann. Jede VM stellt eine unabhängige Plattform für ein Betriebssystem dar, d.h. scheinbar einen vollständigen Rechner. Die echte Rechnerplattform wird durch einen VMM so vervielfacht, dass außer einer Verlangsamung keine anderen Unterschiede zur einfachen Plattform auftreten. Im Großrechnerbereich ist diese Technologie mehrere Jahrzehnte alt und wurde dazu benutzt, mehrere unabhängige, voneinander isolierte virtuelle Rechner gemeinsam auf einem einzigen realen Computer auszuführen. Diese Anwendung hat nun

auch für die PC-Plattform zunehmendes Interesse gefunden. Sie erlaubt beispielsweise eine Serverkonsolidierung, d.h. die Zusammenfassung vieler Servermaschinen auf wenigen realen Rechnern. Damit können auf mehreren Ebenen Kosten eingespart werden. Diese Virtualisierung ist aber auch für die Softwareentwicklung nützlich, denn es können so mehrere Betriebssystemumgebungen auf dem gleichen Entwicklungssystem konzentriert werden. Zum Beispiel kann eine separate VM als Testumgebung dienen, deren eventueller Absturz infolge der Isolation die Entwicklungsumgebung nicht tangiert. Mehr Details sind im nachfolgenden Abschnitt zu finden.

12.3 Virtual Machine Monitor bzw. Hypervisor

Neben der Speichervirtualisierung (siehe Abschnitt 8.5) ist heute die Bereitstellung virtueller Rechnerumgebungen durch einen *Virtual Machine Monitor* (VMM) bzw. *Hypervisor* die wichtigste Virtualisierungsform. Der VMM erlaubt die Erstellung mehrerer virtueller Umgebungen, *Virtual Machines* (VM) genannt, in denen sich unterschiedliche Betriebssysteme installieren lassen.

12.3.1 Anforderungen

Ein VMM muss nach Popek/Goldberg (1974) drei Kerneigenschaften unterstützen:

- *Ausführungsumgebung (fidelity, equivalence)*: Programme laufen auf einem virtualisierten Rechner identisch ab wie auf einem realen Rechner, mit Ausnahme der Geschwindigkeit.
- *Effizienz (performance, efficiency)*: Die große Mehrheit aller Maschinenbefehle, die innerhalb einer VM abgearbeitet werden, müssen direkt durch den realen Prozessor ausgeführt werden ohne die Intervention durch den VMM.
- *Ressourcenverwaltung (safety, resource control)*: Die Ressourcen werden vollständig durch den VMM verwaltet und den einzelnen VMs zugeteilt bzw. wieder entzogen.

12.3.2 VMM-Funktionsweise

In der Anfangszeit der VMM in der PC-Welt fehlte eine passende Hardwareunterstützung, sodass sich der VMM und das in der VM ablaufende Betriebssystem den Kernmodus teilten. Wie das genau funktioniert, ist nachfolgend vorgestellt. Neue Prozessoren unterstützen nun einen separaten Betriebsmodus für den VMM, was später beschrieben wird.

Ohne Unterstützung durch Hardware

Ein sicherer VM-Monitor setzt eine voll virtualisierbare CPU voraus. Das heißt, dass alle Instruktionen, die auf die Hardware zugreifen oder indirekt die Hardware beeinflussen können, nur privilegiert ausgeführt werden. Dazu muss die CPU zwischen einem privilegierten und nicht privilegierten Betriebsmodus unterscheiden. Typischerweise werden diese als *Kernmodus* (*kernel/supervisor mode*) und *Benutzermodus* (*user mode*) bezeichnet. Prozessorintern wird der privilegierte Modus durch ein gesetztes KM-Bit repräsentiert (KM für Kernel Mode). Dieses Bit kann im Benutzermodus natürlich nicht direkt gesetzt werden. Erfolgt jedoch ein Hardware- oder Software-Interrupt, so wird es automatisch durch die CPU selbst gesetzt (siehe Abb. 12–1). Die Einsprungpunkte der Interrupt-Behandlungsroutinen stehen dabei in einem geschützten Speicher, sodass nur vertrauenswürdige Codeteile im Kernmodus zur Ausführung kommen. Dies setzt wiederum das Vorhandensein einer Speicherverwaltung (MMU-Baustein) voraus, durch die bestimmte Adressbereiche für die Ausführung im Kernmodus reserviert werden.

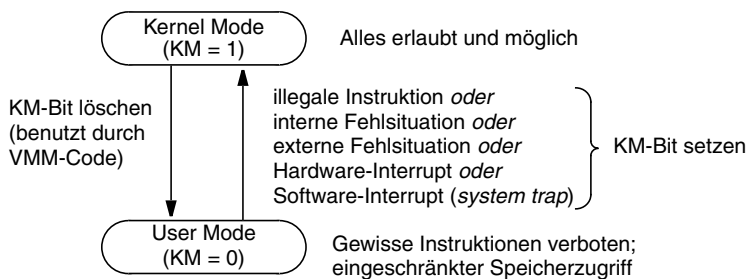


Abb. 12–1 Zustandsmodell für Privilegiensystem der CPU

Was passiert nun, wenn ein Programm im Benutzermodus eine privilegierte Instruktion ausführen will? Die CPU wird in diesem Fall den Befehl nicht ausführen, sondern einen Software-Interrupt (trap) auslösen. Wird ein VMM eingesetzt, so läuft nur der VMM im privilegierten Modus. Aller restlicher Code, einschließlich irgendwelcher Betriebssysteme, läuft im Benutzermodus. Die Unterscheidung eines Betriebssystems zwischen Benutzer- und Kernmodus wird nämlich durch den VMM ebenfalls virtualisiert. Damit steht die Hardware unter alleiniger Kontrolle des VM-Monitors, der diese nur gezielt an die VMs delegiert und diesen jederzeit wieder entziehen kann. Eine wesentliche Voraussetzung für eine voll virtualisierbare CPU stellt die Privilegierung heikler Prozessorinstruktionen dar. Heikel sind sie dann, wenn sie einer VM erlauben würden, dem VMM die Kontrolle über die Hardware zu entreißen. Alle diese Instruktionen müssen einen Software-Interrupt auslösen, wenn sie im Benutzermodus eingesetzt werden. Kernelement ist also, dass privilegierte Instruktionen in den VM-Monitor verzweigen. Dieser

nimmt dann eine virtuelle Ausführung dieser Instruktionen vor, wenn sie im virtualisierten Kernmodus erfolgen. Dabei wird der VM nur kontrolliert ein Teil der realen Hardware zugänglich gemacht. In Abbildung 12–2 ist ein Beispiel mit zwei VMs dargestellt. Beide VMs laufen im nicht privilegierten CPU-Modus. Den Gastbetriebssystemen wird jedoch erlaubt, die privilegierten Instruktionen innerhalb ihrer virtuellen Umgebung zu nutzen (KM* als virtualisiertes KM-Bit). Der eigentliche Zugriff auf die Hardware erfolgt jedoch nur durch den VMM kontrolliert, d.h., er interpretiert die privilegierten Instruktionen im Zustand $KM^*=1, KM=0$ derart, dass der VM der Zugriff auf zugeteilte Ressourcen möglich ist. Ein VMM, der nach diesem Prinzip arbeitet, wird auch als *Software-only VMM* bezeichnet.

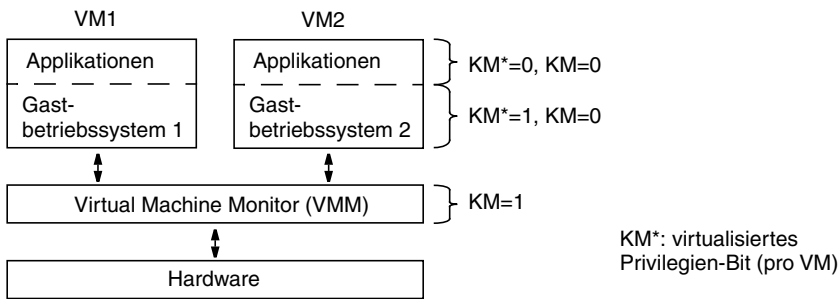


Abb. 12–2 Sichere Virtualisierung mittels Privilegiensystem

Ein VMM kann auch auf Prozessoren realisiert werden, die keine volle Virtualisierung unterstützen. Je nachdem, wie viel Unterstützung eine CPU bereits realisiert, muss jedoch auf eine zunehmend interpretierte Ausführung der Maschinenbefehle und weitere Zusatzmaßnahmen ausgewichen werden. Ist gar keine Unterstützung vorhanden, so ist für einen sicheren Betrieb nur eine volle Interpretation aller Prozessorinstruktionen (Emulation) möglich, wie in Abschnitt 12.2.1 beschrieben. Eine Zwischenform stellt der Intel x86-Prozessor dar. Er bietet gute Schutzmöglichkeiten, hat aber nicht alle heiklen Instruktionen privilegiert. Daher muss für einen sicheren VMM der ausgeführte Instruktionsstrom durch den VMM fortlaufend auf diese heiklen Befehle abgesehen werden. Diese werden dann durch Befehlssequenzen ersetzt, die sie auf sichere Art und Weise ausführen (*binary translation, dynamic rewriting*). Die so realisierten virtuellen Umgebungen werden als *Hybrid Virtual Machines* bezeichnet.

Unterstützt ein Prozessor mehr als zwei Privilegienstufen, so ist eine weitere Implementierungsart möglich. Diese wird *Paravirtualisierung* genannt. Dazu muss aber der Code der Gastbetriebssysteme angepasst werden, was sich bei kommerziellen Betriebssystemen aus Lizenzgründen verbietet. Der Intel x86-Prozessor realisiert vier Privilegienstufen, die auch als Ringe bezeichnet werden. Für

die Paravirtualisierung wird nun beispielsweise dem VMM die Stufe 0, dem Gast-Betriebssystem die Stufe 1 und den Applikationen die Stufe 3 zugeordnet (siehe Abb. 12–3). Der Xen-VMM der Cambridge-University nutzt diesen Ansatz.

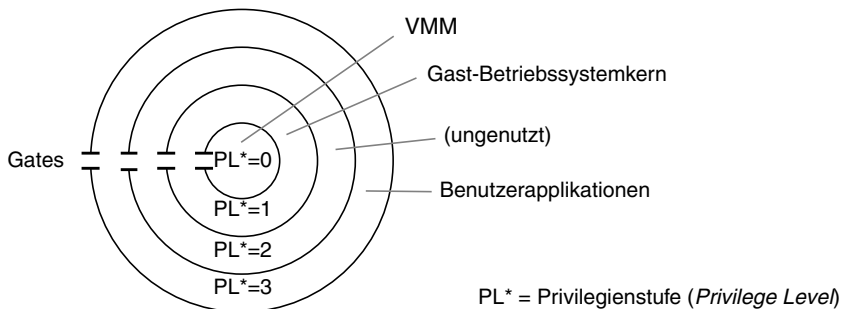


Abb. 12–3 Privilegiensystem des Intel x86-Prozessors mit einer möglichen Nutzung der Privilegienstufen für die Paravirtualisierung

Betrachtet man die 64-Bit-Prozessorvariante, d.h. den Intel x86-64, so lässt sich diese Idee nicht mehr direkt umsetzen, da diese CPU-Architektur nur noch den Ring 0 und 3, aber nicht 1 und 2 realisiert. Für diesen Prozessor muss daher auf die Verfahren der Software-VMM zurückgegriffen werden, um eine Paravirtualisierung zu realisieren.

Mit Unterstützung durch Hardware

Moderne CPUs enthalten eine Zusatzhardware für eine einfachere Realisierung des VMM, der dann als *Hardware-assisted VMM* bezeichnet wird. Dies trifft für viele Prozessortypen zu, auch für die x86- und x86-64-Prozessoren. Ergänzend zu den Privilegienstufen, die das Betriebssystem nutzt, steht eine nochmals höher privilegierte Hypervisor-Stufe zur Verfügung, in die bei der Ausführung kritischer Instruktionen verzweigt wird. Bei x86-Prozessoren wird diese als Root-Betriebsmodus bezeichnet.

12.3.3 VMM-Typen

Der Prozentsatz aller Maschinenbefehle, der direkt auf der CPU ausgeführt wird, stellt ein Unterscheidungsmerkmal verschiedener VMM-Typen dar. Werden alle Prozessorinstruktionen durch Software emuliert, so wird dies als *Complete Software Interpreter Machine (CSIM)* bezeichnet und entspricht der Lösung in Abschnitt 12.2.1. Diese betrachtet man meist nicht als VMM, sondern als Prozessorsimulatoren für begrenzte Anwendungszwecke. Die verbleibenden Lösungen werden in *Hybrid VM (HVM)* und *VMM* unterteilt, wobei VMM letztlich meist auch als Oberbegriff für beide Typen gilt. Eine HVM unterscheidet sich von

einem VMM darin, dass alle privilegierten Instruktionen interpretiert werden müssen (*Software-only VMM*). Ein VMM kann bei einem Teil der privilegierten Maschinenbefehle darauf verzichten, da er durch Prozessorhardware unterstützt wird (*Hardware-assisted VMM*). Welcher Typ implementiert werden kann, wird durch die Möglichkeiten der Hardware vorgegeben.

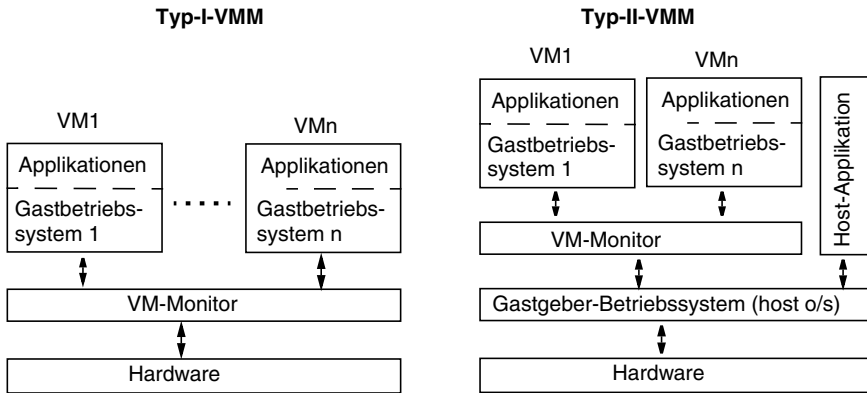


Abb. 12-4 VMM-Realisierungen. Der VMM des Typs II wird durch einen speziellen VMM-Treiber ergänzt (nicht gezeigt), der unter dem Gastgeber-Betriebssystem installiert wird.

Ein VMM kann auf zwei Arten realisiert werden, die als Typ-I- und Typ-II-VMM bezeichnet werden. Beim Typ I setzt der VMM direkt auf der Hardware auf und virtualisiert diese (*bare-metal*). Der VMM stellt ein eigenes Betriebssystem dar, dessen Aufgabe in der Ressourcenverwaltung und der Bereitstellung von VMs liegt. Die Ressourcenverwaltung umfasst die Aufteilung der Rechenzeit (*scheduling*) und die Zuteilung (*allocation*) von Speicher und Peripherie auf die einzelnen VMs. Beispiel eines solchen Produkts ist VMware ESX, das aber nur auf Hardware ablaufen kann, für die es eigene Treiber besitzt. Ein Typ-II-VMM läuft als eine von mehreren Applikationen (*hosted*) auf einem Gastgeber-Betriebssystem (*host operating system*). Damit er jedoch die nötigen Rechte auf der Hardware erhält, benutzt er einen speziellen VMM-Treiber, der unter dem Gastgeber-Betriebssystem installiert ist und dadurch dem Hypervisor Kernmodus-Privilegien eröffnet. Jedes weitere Betriebssystem, das unter dem VMM läuft, wird als Gastbetriebssystem (*guest operating system*) bezeichnet. Der VMM kann auf Mechanismen des unterliegenden Gastgeber-Betriebssystems für die Ressourcenverwaltung zurückgreifen. Typ-II-Produkte sind VMware Workstation, Virtual-Box und Xen. Bei Xen ist ein Linux für die Verwaltung integriert, sodass dieser VMM auch direkt auf die Hardware aufsetzen kann. Wenn die CPU jedoch keine Hardwareunterstützung anbietet, dann beherrscht Xen nur die Paravirtualisierung. Eine zwischen Typ I und Typ II anzusiedelnde Variante sind Hypervisoren, die in das Betriebssystem von Haus aus bereits integriert sind. Realisiert wird dies

in Linux durch die *KVM (Kernel-based Virtual Machine)* und unter Windows mit *Hyper-V*. Diese Zwischenform lässt sich so einsetzen, dass keine Applikationen auf dem Betriebssystem selbst laufen, sondern nur der Hypervisor genutzt wird. In diesem Szenario sind keine Leistungs Nachteile im Vergleich zu reinen Typ-I-VMM zu erwarten.

12.3.4 Unikernel

In Abschnitt 12.2.6 ist die einfache Bereitstellung von Standard-Images beschrieben, die auf Container-Systemen aufsetzen. Eine vergleichbare Lösung lässt sich mit sogenannten *Unikernels* realisieren. Hierbei werden virtuelle Maschinen bereitgestellt, die neben einer einzelnen Anwendung nur gerade diejenigen Teile des darunterliegenden Betriebssystems enthalten, die zwingend von der betreffenden Applikation benötigt werden. Da nur eine einzelne Anwendung läuft, ist eine Isolation des Kernels durch Nutzung des Kernmodus verzichtbar, was Geschwindigkeitsvorteile erzeugt, die den Overhead durch die Virtualisierung kompensieren können.

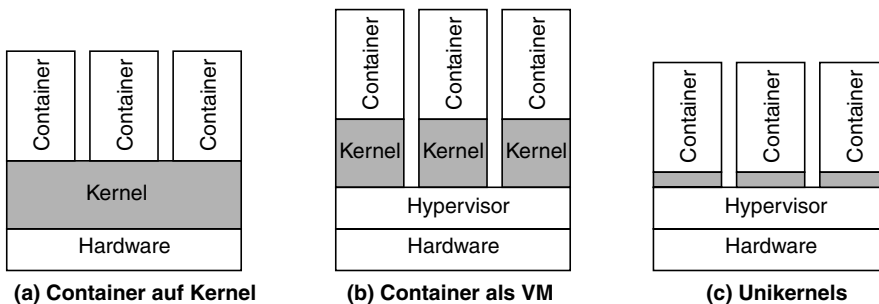


Abb. 12-5 Applikationsausführung ab Container-Image ohne (a) und mit Hypervisor (b) sowie als Unikernel mit Hypervisor (c)

Abbildung 12-5 zeigt die Unterschiede zwischen den Lösungen mit Container direkt auf dem Kernel (a), auf je privatem Kernel in einer VM (b) und mit Unikernels (c). Die bei der Variante (b) nachteilige Duplizierung des Kernel-Codes wird bei Variante (c) minimiert.

12.4 Einsatzgebiete

Ein VMM ermöglicht Einsatzszenarien, die oft mit anderen Technologien nur erschwert oder gar nicht realisierbar sind. Beispiele sind:

- Serverkonsolidierung
- Applikationsisolation

- Applikationsmigration
- Softwaredistribution
- Testumgebung
- Systemüberwachung
- Program Shepherd
- Cloud Computing

Sind Rechnerressourcen nur wenig genutzt, so lassen sich mittels der *Serverkonsolidierung* Hardwareaufwand und damit Kosten sparen. In Abbildung 12–6 ist gezeigt, wie zwei separate Servermaschinen durch ein gemeinsames System ersetzt werden, bei dem ein VMM den Betrieb zweier vollständig isolierter Serverapplikationen erlaubt. Durch die bessere Hardwarenutzung sinkt nicht nur der Hardwareaufwand, sondern auch die Administration wird einfacher.

Werden Rechnerplattformen als Dienstleistung angeboten, so ist eine maximale Isolation der Applikationen einzelner Kunden eine Voraussetzung für den problemlosen Betrieb. Zudem ist eine solche Isolation in vielen weiteren Anwendungsbereichen ebenfalls erwünscht. Das Einsatzszenario ist mit der Serverkonsolidierung vergleichbar (siehe Abb. 12–6).

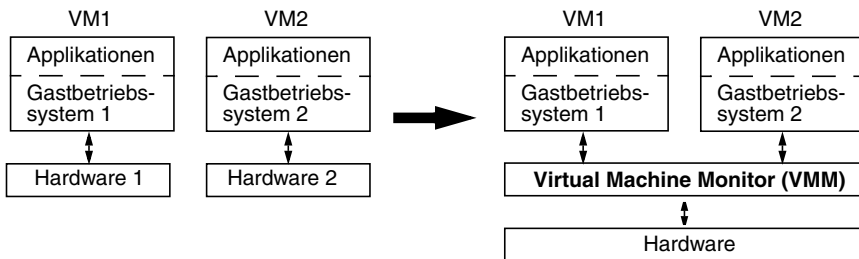


Abb. 12–6 VMM-Einsatzszenario Serverkonsolidierung bzw. Applikationsisolation (Beispiel)

Die Nutzen sind eine verbesserte Betriebsstabilität, die mögliche Realisierung unterschiedlich sicherer Applikationsumgebungen sowie unterschiedlicher Betriebssysteme auf derselben Hardware. Ein anderes Szenario ist die *Applikationsmigration*, bei der eine Applikation mitsamt ihrer Laufzeitumgebung von Rechner 1 auf Rechner 2 migriert wird (siehe Abb. 12–7). Moderne Lösungen unterstützen diese Fähigkeit im laufenden Betrieb (*live-migration*). Die Nutzen sind eine einfache Replizierung im Stil eines *fail-over*, eine mögliche Skalierung der eingesetzten Rechnerhardware (*down-lup-scaling*) oder auch die Multiplizierung, wenn als Resultat dieselbe Ablaufumgebung zweimal vorliegt.

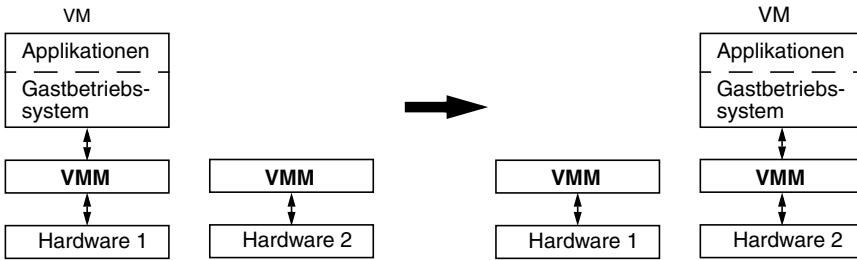


Abb. 12-7 VMM-Einsatzszenario Applikationsmigration

Im Informatikbetrieb kann die Softwaredistribution durch den Einsatz einer VMM-Lösung vereinfacht werden. Auf einem Mastersystem werden das Betriebssystem und alle Applikationen installiert, konfiguriert und getestet (siehe Abb. 12-8). Anschließend wird das virtuelle Abbild der Masterinstallation auf alle Clientmaschinen verteilt. Damit wird der Supportaufwand reduziert und beim Einsatz mehrerer VM auf dem Clientrechner können zudem unterschiedliche Desktop-Arbeitsumgebungen gepflegt werden.

Auch für die Softwareentwicklung kann der VMM-Einsatz nützlich sein. Da sich die Entwicklungs- und Testumgebung auf demselben Rechner ausführen lassen (siehe Abb. 12-9), können Hardware und Kosten gespart werden. Häufig wird damit auch die Handhabung vereinfacht.

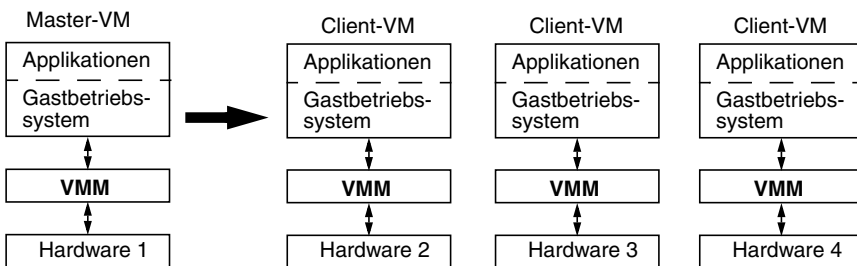


Abb. 12-8 VMM-Einsatzszenario Softwaredistribution (Beispiel)

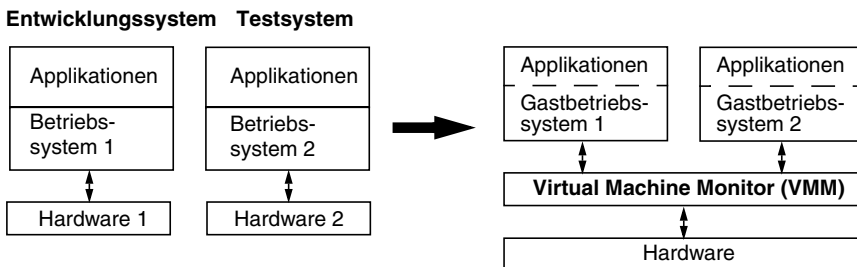


Abb. 12-9 VMM-Einsatzszenario Testumgebung (Beispiel)

Der Zugang auf das Firmennetz über private PCs kann Sicherheitsprobleme aufwerfen. So könnten auf diesem Weg infizierte Dateien vom Internet in das Firmennetz gelangen. Indem für den Firmenzugang eine separate VM eingesetzt wird, kann der Netzzugriff eingeschränkt werden, ohne dass der Nutzer für seine privaten Applikationen irgendwelche Einschränkungen erfährt (siehe Abb. 12-10).

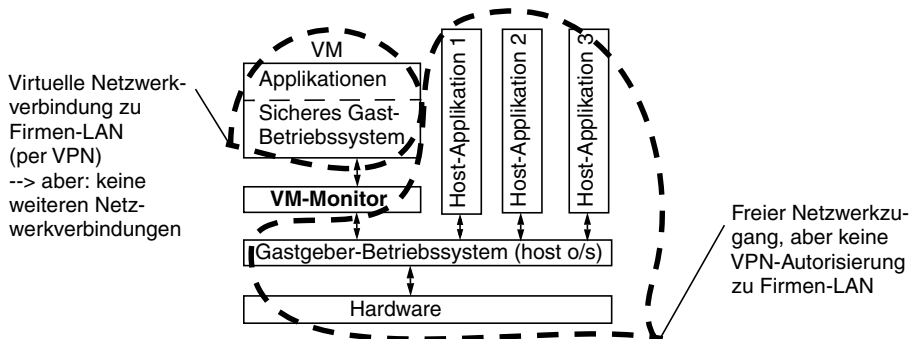


Abb. 12-10 VMM-Einsatzszenario gesicherter Firmenzugang (Beispiel)

Weitere Einsatzszenarien sind:

- Systemüberwachung: Beispielsweise durch ein *Intrusion and Prevention Detection System (IPS)*, das in einer separaten VM abläuft.
- *Überprüfen unsicherer Codes*: Verdächtiger Code wird in einer VM zur Beobachtung ausgeführt, was auf einem produktiven System ein zu großes Risiko wäre. Dies wird auch als *Program Shepherd* bezeichnet und kann verschiedene Analysen beinhalten, wie beispielsweise auf sichere Sprungziele bzw. auf vertrauenswürdige Quellen von Programmsprüngen.
- *Cloud Computing*: Der VMM ist eine der Technologien, auf denen Cloud Computing aufbaut. Daher können VMM-Produkte ein Bestandteil eines Cloud-Computing-Angebots sein. Insgesamt können Clouds jedoch sehr verschiedenartig aufgebaut sein, weswegen wir hier für Details auf die spezialisierte Literatur verweisen.