1. Die Software-Plattform

Die Systemsoftware stellt praktisch eine Plattform zum “Laufenlassen” von Anwendungen dar (Abb. 1.1 und 1.2). Im allgemeinen Sprachgebrauch wird die Software-Plattform mit dem Betriebssystem gleichgesetzt (so spricht man von einer Windows-Plattform, einer Linux-Plattform usw.). Es gehört zur Systemphilosophie (und vor allem auch zum Marketing), welche Funktionen in welchen Programmen untergebracht sind und wie diese verkauft werden. Die Grenzfälle:

- Alles aus einer Hand. Es gibt einen einzigen Programmkomplex (das “Betriebssystem” im herkömmlichen Sinne), der von einem Hersteller angeboten wird.
- Das Baukastenprinzip (modulare Struktur). Das eigentliche Betriebssystem hat nur einen minimalen Umfang (es betrifft z. B. lediglich die Funktionen “Laufzeitvergabe” und “Betriebsmittelverwaltung”). Alle anderen Funktionen werden von zusätzlichen Programmen erledigt (und die können womöglich von verschiedenen Anbietern bezogen werden).

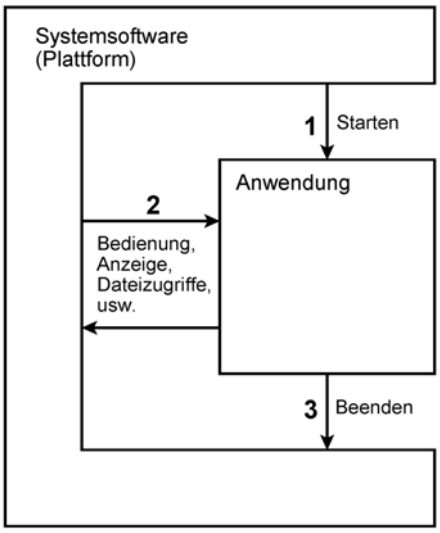


Abb. 1.1 Die Software-Plattform. 1 - das System startet die Anwendung (die Anwendung erhält hiermit praktisch die Verfügung über die Hardware); 2 - die Anwendung ruft Systemfunktionen und Dienstleistungen auf bzw. wird vom System aus beeinflusst; 3 - die Anwendung wird beendet (sie gibt hiermit die Hardware wieder frei).

Heutzutage gibt es wohl kaum eine Plattform, zu der nicht Komponenten von Drittanbietern erhältlich sind – nur der Anteil macht den Unterschied. Alle modernen Plattformen sind in sich modular strukturiert; sie bestehen aus einer Vielzahl von Programmen, und es gibt vielfältige Wahlmöglichkeiten beim Installieren und Konfigurieren. Es ist im Grunde nur die Frage, wer zu welchen Bedingungen solche Module anbieten darf (und wenn ja, wie aufwendig das ist).

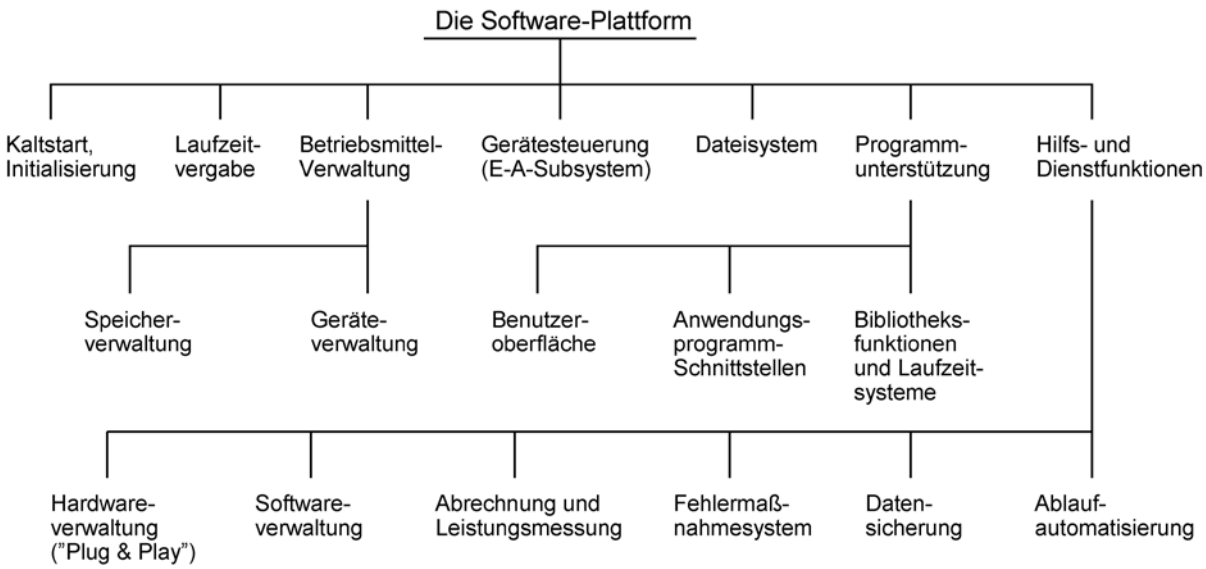


Abb. 1.2 Typische Funktionen einer Software-Plattform.

Jede Plattform ist durch eine bestimmte "Philosophie" gekennzeichnet. Die Philosophie geht typischerweise auf die Entwicklungsgeschichte des Systems zurück. Sie wird maßgeblich durch folgende Gegebenheiten geprägt:

- durch das ursprünglich vorgesehene Einsatzgebiet,
- durch die ursprünglich zugrunde gelegten Hardware-Plattformen oder Architekturen,
- durch außerfachliche Zwänge (Budgets, Termine, Patentrecht usw.),
- durch freie Entwurfsentscheidungen.

Das typische Betriebssystem

Jeder Entwickler hat eine eigene Auffassung darüber, welche Funktionen ins Betriebssystem gehören und welche besonderen Dienst- und Hilfsprogrammen außerhalb des Betriebssystems zu überlassen sind. Wenn man die grundsätzlichen Funktionen betrachtet, so unterscheiden sich die üblichen Betriebssysteme aber nur wenig voneinander (Abb. 1.3). Es gibt zwei "Kunden", die Leistungen des Betriebssystems in Anspruch nehmen:

- Die Anwendungsprogramme. Diese rufen die Systemfunktionen über Softwareschnittstellen auf, die üblicherweise als *Application Program Interface* (API) bezeichnet werden.
- Die Nutzer. Die zur (manuellen) Bedienung des Systems vorgesehenen Funktionen werden allgemein unter den Begriffen "Benutzeroberfläche" oder "Bedienschnittstelle" (User Interface) zusammengefasst. Aus dem Unix-Bereich ist weiterhin der bildhafte Begriff *Shell* (Schale) bekannt. Alle Benutzeroberflächen oder Shells sind – aus der Sicht dessen, der vor dem Computer sitzt – irgendwie komfortabel, manche mehr und manche weniger.

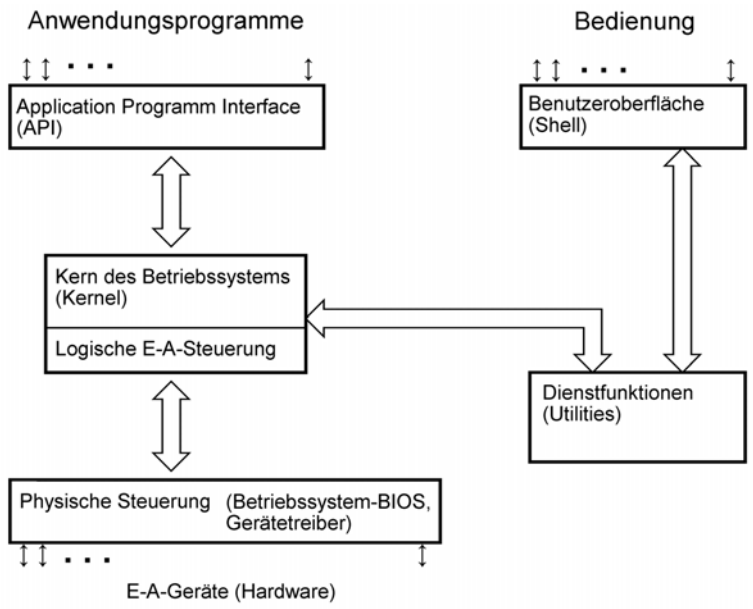


Abb. 1.3 Die typische Struktur eines Betriebssystems. Auf dieser Eben sehen alle Systeme nahezu gleich aus ...

Zwei Zustände

Die Anwendungen dürfen Einiges nicht, was das System tun muss, beispielsweise auf Register der Hardware zugreifen und die Belegung des Arbeitsspeichers verändern. Eine seit Langem bewährte Lösung, Anwendungs- und Systemprogramme gegeneinander abzugrenzen, besteht darin, in der Hardware zwei Zustände einzuführen: den Anwenderzustand (User State) und den Systemzustand (System oder Supervisor State). Im Systemzustand ist alles erlaubt (Zugriffe auf die Hardware usw.), und alle Maschinenbefehle dürfen ausgeführt werden. Im Anwenderzustand werden bestimmte Zugriffe und das Ausführen bestimmter Maschinenbefehle verhindert. Wer es trotzdem versucht, löst eine Fehlermeldung an das Betriebssystem aus. Zum Umschalten gibt es bestimmte Maschinenbefehle. Nähere Einzelheiten in Abschnitt 4****. Wichtig ist zunächst:

- Die Anwendung darf nicht alles.
- Wenn sie eine Dienstleistung braucht – z. B. einen Dateizugriff –, so muss sie das Betriebssystem rufen.
- Hierzu muss sie die Softwareschnittstelle des Application Programming Interface (API) ansprechen. Das läuft letzten Endes darauf hinaus, Maschinenbefehle auszuführen, die einen Wechsel in den Systemzustand veranlassen.
- Hat das System seine Dienstleistung erbracht, so führt es Maschinenbefehle aus, die eine Rückkehr zur Anwendung bewirken. Dabei wird der Systemzustand wieder verlassen.

Der Kern des Betriebssystems

Die eigentlichen Betriebssystemfunktionen bilden den Kern (Kernel[1)]). Insbesondere bei Betriebssystemen, die mit vielfältigen peripheren Geräten zusammenwirken sollen, wird die E-A-Steuerung nochmals gegenüber dem eigentlichen Kern isoliert. Die Kern-Routinen verwirklichen dann die logischen Gerätesteuerfunktionen. Die physische Gerätesteuerung wird hingegen gesonderten Programmen übertragen, die über reguläre Schnittstellen vom Kern aus ansprechbar sind (Physical Input/Output System, Gerätetreiber). In der Betriebssystementwicklung wird seit Jahrzehnten darum gestritten, was wohin gehört:

- Welche Funktionen gehören in den Kern, welche nicht?
- Welche Funktionen sollen überhaupt im Systemzustand ausgeführt werden, welche sind im Anwenderzustand eigentlich besser aufgehoben?

Jeder Praktiker weiß, dass komplexe Programme einfach nicht 100%ig fehlerfrei hinzubekommen sind. Damit, dass Anwendungen abstürzen, hat man sich – zumindest allem Anschein nach – seit Längerem abgefunden[2)]. Wenn aber schon eine Anwendung abstürzt, so soll sie das wenigstens für sich tun, also nicht auch noch andere Anwendungen mit hineinziehen. Vor allem aber sollen wenigstens die wichtigsten Funktionen des Systems überleben, weil es nur so möglich ist, den Fehler zu analysieren und ggf. Gegenmaßnahmen zu veranlassen. Das Hauptziel besteht darin, einen Kern zu haben, der unter keinen Umständen abstürzen kann. Hierzu liegt es nahe, den Kern so klein und einfach wie möglich auszulegen[3)]. Andererseits soll das System aber auch etwas leisten ...

Voll ausgebaute Kerne (Monolithic Kernels)

Der Systemkern ist ein vergleichsweise großes und sehr komplexes Programm, das alle Systemfunktionen erbringt, bei denen es auf Geschwindigkeit ankommt. Diese Auslegung wird – aus Leistungsgründen – herkömmlicherweise bevorzugt (Abb. 1.4).

Minimale Kerne (Microkernels)

Diese Auslegung ergibt sich aus der bereits angesprochen Überlegung heruas, den Kern so klein wie möglich zu halten (Überschaubarkeit). Im Extremfall ist er wirklich nur auf die notwendigsten Funktionen der Laufzeitvergabe, Speicherverwaltung und Gerätesteuerung beschränkt. Alles Andere wird von Programmmodulen erledigt, die über wohldefinierte Schnittstellen mit dem Kern zusammenwirken.

1: Sprich: Körnel.

2: Vgl. die – zweifellos sehr aufwendigen – Vorkehrungen zur Fehlerberichterstattung und zum automatischen Einspielen von Updates (und auch die Anzahl der Updates) ...

3: Was nicht da ist, kann auch nicht versagen ...

Client-Server-Betriebssysteme

Hat man sich für einen minimalen Kern entscheiden, so ergibt sich die Frage, in welchem Zustand die anderen Programmmodule laufen sollen. Am besten wäre es wohl im Anwenderzustand, denn da können sie – wenn sie denn abstürzen – keinen weiteren Schaden anrichten (Abb. 1.5). Diese Programme werden als Server angesehen, die für den Kern – als Client – Dienstleistungen erbringen. Da alle Programme nur über reguläre Schnittstellen zusammenwirken, ist es möglich, sie unabhängig voneinander weiterzuentwickeln (neue Dateisysteme, Bedienoberflächen usw.) und ins System einzubringen[4]. Diese Auslegung ist besonders im akademischen Bereich beliebt[5]. Seit vielen Jahren wird daran geforscht. Manchmal ergibt sich sogar eine Anwendung in der Praxis ...

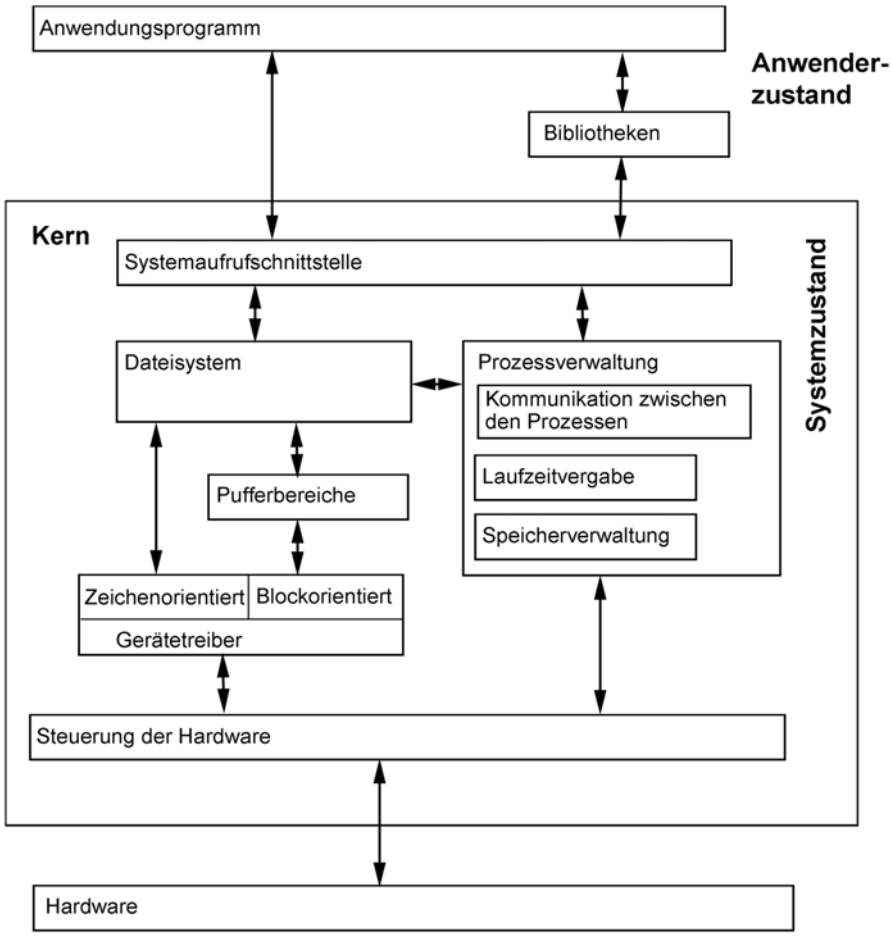


Abb. 1.4 Die grundsätzliche Struktur der Unix-Betriebssysteme. Der Unix-Kernel führt alle entscheidenden Funktionen selbst aus.

4: Also ohne umfassende Neuinstallation oder umfangreiche Updates.

5: Weil so das gesamte System in überschaubare Häppchen zerfällt, die sich gut als Themen für Abschlussarbeiten, Zeitschriftenartikel und Konferenzbeiträge eignen ...

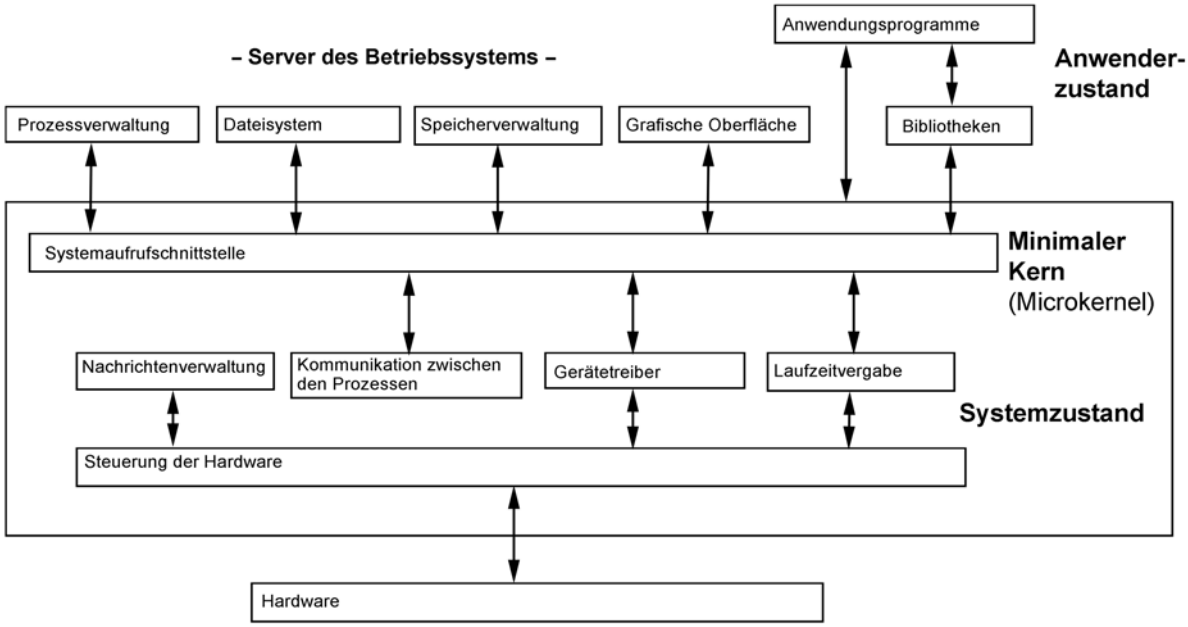


Abb. 1.5 Ein Client-Server-Betriebssystem mit Microkernel. Die Dienstfunktionen werden im Anwenderzustand ausgeführt.

Da bei der Client-Server-Organisation alles über reguläre Programmschnittstellen geht und übliche Systemfunktionen zumeist mit mehreren solchen Aufrufen – einschließlich der Zustandsübergänge – verbunden sind, weisen akademisch ausgelegte Systeme nach wie vor Leistungsschwächen auf. Ein Beispiel ist der sog. Mach-Microkernel, der von der Carnegie-Mellon-Universität entwickelt wurde. Die "akademische" Struktur entspricht näherungsweise Abb. 1.5. Es gibt aber auch im praktischen Einsatz befindliche Systeme, die auf diesem Kern aufbauen[6]. Dann werden aber typischerweise auch die andern leistungsentscheidenden Dienstfunktionen – wie die Speicherverwaltung, die Netzwerkkommunikation und die Dateisysteme – im Systemzustand ausgeführt.

Windows

Die modernen Windows-Versionen sind – ihrer grundsätzlichen Struktur nach – Kompromisslösungen (Abb. 1.6). Die leistungsentscheidenden Funktionen[7] werden im Systemzustand ausgeführt. Ergänzende Dienstfunktionen laufen im Anwenderzustand. Dadurch, dass viele komplexe Funktionen im Systemzustand ausgeführt werden, wird das System leistungsfähiger. Das Absturzverhalten wird zwar ungünstiger, es sieht aber damit auf den ersten Blick schlimmer aus als es wirklich ist. Denn manche Funktionen sind so wichtig, dass man im Fehlerfall ohnehin nicht weiterarbeiten kann. So braucht nahezu jede Anwendung das Dateisystem, die Fensterverwaltung und die grafische Oberfläche. Wenn beispielsweise das Dateisystem abstürzt, so würde dies in einem System gemäß Abb. 1.6 zunächst dazu führen, dass

6: U. a. die modernen Macintosh-Betriebssysteme.

7: Wozu auch die Fensterverwaltung und die elementaren Grafikfunktionen gehören.

nur die betreffende Anwendung beendet wird und alle anderen weiterlaufen. Aber die anderen Anwendungen werden auch irgendwann einmal das Dateisystem benötigen ... Es ist einfach eine Frage der Wahrscheinlichkeit, ob der Fehler bei den dann angesprochenen Funktionen auch wirksam wird oder nicht. Der Kompromiss zugunsten des Leistungsvermögens hat also durchaus seine Berechtigung.

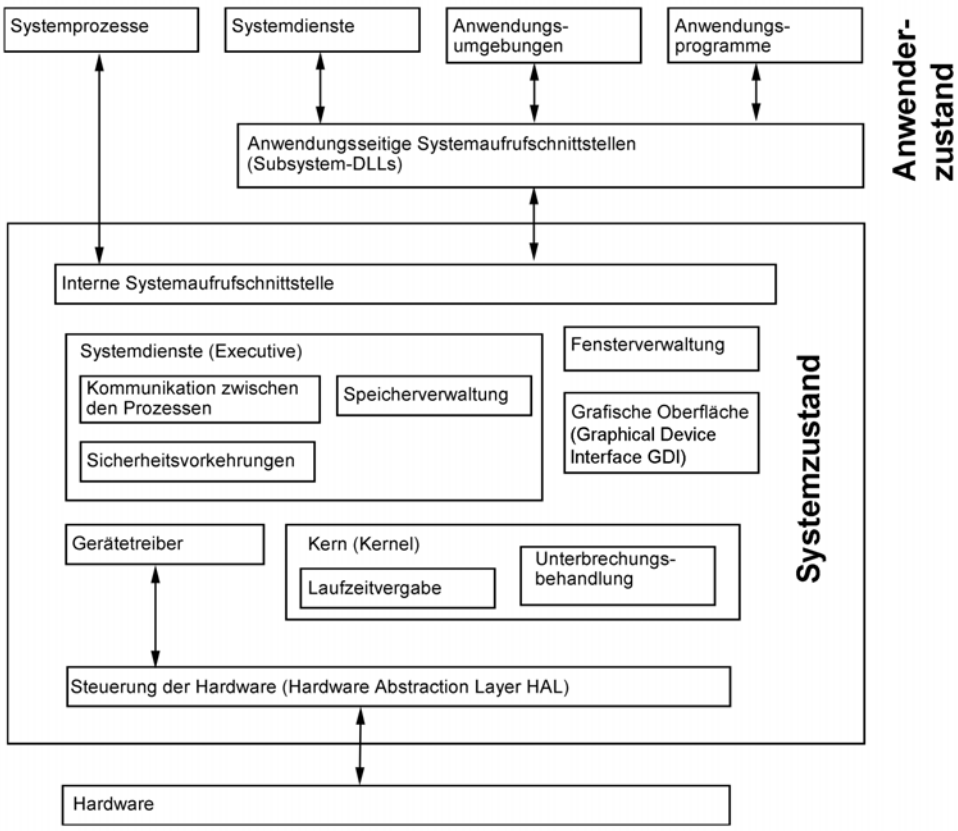


Abb. 1.6 Die grundsätzliche Struktur der modernen Windows-Betriebssysteme (ab Windows NT/2000).

Kleine Kerne

Viele Programme laufen – aus Leistungsgründen – im Systemzustand, der eigentliche Kern wird aber – wegen der Überschaubarkeit – klein gehalten (vgl. auch Abb. 1.5). Es ist im Grund nur eine Frage der Zerlegung der Programmieraufgabe – und des Marketing. Dieses wirbt gelegentlich mit Hinweisen der Art, der Kern des Betriebssystems XYZ könne vom Mobiltelefon bis zum Supercomputer so ziemlich alles steuern ... Kann sein. Wenn dem so ist, so heißt das aber nicht mehr als dass man wirklich nur die allerwichtigsten Grundfunktionen in einem einzelnen Programmstück zusammengefasst hat – ein solcher Kern ist so einfach, dass man ihn eigentlich auch selbst schreiben könnte ...

2. Funktionen der Plattform

2.1 Laufzeitvergabe

2.1.1 Wie viele Programme?

Die klassische Datenverarbeitung

Es werden Programme abgearbeitet, um aus gegebenen Daten (Eingangsdaten) die jeweils gewünschten Ergebnisse (Ausgangsdaten) zu bestimmen; das klassische Programm entspricht dem Schema Eingabe – Verarbeitung – Ausgabe (Abb. 2.1). Hierbei wird ein Programm zu einer Zeit ausgeführt. Ggf. müssen, um ein Anwendungsproblem zu lösen, mehrere Programme nacheinander aufgerufen werden, wobei die Ergebnisse der vorhergehenden Programme zu Eingaben der nachfolgenden werden (Zwischenspeicherung).

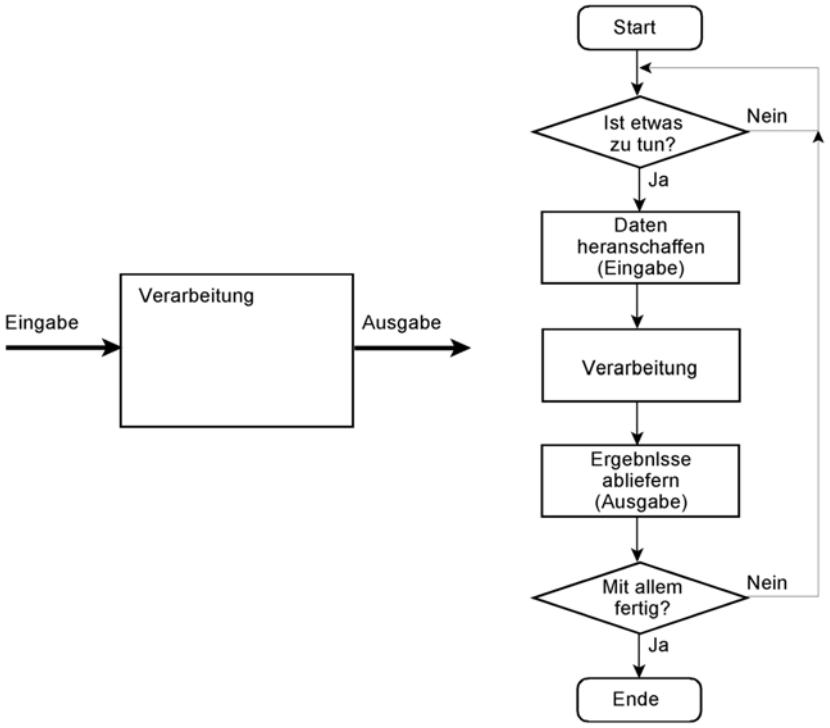


Abb. 2.7 Das grundsätzliche Ablaufschema der klassischen Datenverarbeitung.

Dieser Ablauf wurde bald als unbefriedigend empfunden, und man hat nach Lösungen gesucht, mehrere Programme gleichzeitig ausführen zu können. Typische Entwicklungsziele:

- Wartezeiten (vor allem während der Ein- und Ausgabe) sollten mit nützlicher Arbeit ausgefüllt werden.
- Mehrere Anwender sollten die Maschine gleichzeitig nutzen können.

Personalcomputer

Die ersten Personalcomputer waren Einzelplatzsysteme, die – wie die klassischen EDV-Anlagen – zu einer Zeit ein Anwendungsprogramm ausführen konnten. Nur sitzt hier der Anwender die ganze Zeit über am Bildschirm (interaktiver Betrieb im Gegensatz zur sog. Stapelverarbeitung der EDV[8]). Wenn man nur jeweils ein Programm ausführen kann, ist es recht umständlich, verschiedene Anwendungen im Verbund einzusetzen (z. B. einen Text zu schreiben, die zugehörigen Abbildungen zu zeichnen, Berechnungen auszuführen und gespeicherte Datenbestände zu durchsuchen). Deshalb kam bald der Wunsch auf, mit mehreren Programmen gleichzeitig arbeiten und zwischen ihnen freizügig umschalten zu können.

Embedded Systems

Manche Anwendungsaufgaben können gemäß dem einfachen Schema Eingabe – Verarbeitung – Ausgabe erledigt werden, andere sind hingegen nicht ohne weiteres auf dieses Schema zurückzuführen[9]:

- es sind mehrere unabhängige Eingaben zu verarbeiten,
- es sind gleichzeitig mehrere Ausgaben zu liefern,
- es sind mehrere Informationswandlungen gleichzeitig auszuführen,
- es sind strikte Zeitbedingungen einzuhalten (Millisekunden oder weniger).

Das Grundproblem: Mehreres gleichzeitig erledigen

Mehrere Anwendungsfunktionen sind so auszuführen, dass sie der Anwendungsumgebung (ggf. einschließlich eines Nutzers/Bedieners) so erscheinen, als ob sie gleichzeitig (simultan) ablaufen würden. Dieses Problem tritt in allen Bereichen der Computertechnik (EDV, PCs, Embedded Systems usw.) auf. Auf einem Prozessor kann aber zu einer Zeit nur ein Programm laufen.

Jedem Programm ein eigener Prozessor?

Diese naheliegende Alternative war in der Vergangenheit offensichtlich viel zu teuer – ganz abgesehen von Praxisfragen, z. B. dem Platzbedarf, wenn der einzelne Prozessor einen ganzen Schrank belegt. Das gilt in vielen Anwendungsbereichen auch noch heute, obwohl es der Stand der Technik ermöglicht, mehrere Prozessorkerne auf einem einzigen Schaltkreis unterzubringen. Für vieleAnwendungen sind solche Prozessoren einfach zu teuer. Und dort, wo man sie sich leisten kann, wäre eine derartige Nutzungsweise nicht immer zweckmäßig. Wenn ein Programm auf eine Eingabe wartet (was häufig der Fall ist), so hat der betreffende Prozesor nichts zu tun. Und er kann auch nicht dazu beitragen,den Ablauf der anderen Programme zu beschleunigen.

8: Die Eingangsdaten liegen auf Datenträger vor (Magnetband, Platte o. dergl.), die Ergebnisse werden gedruckt (Steuerbescheide, Rechnungen usw.) oder auf Datenträgern abgelegt. Das alles läuft weitgehend automatisch ab.

9: Nicht selten ergeben sich solche Anforderungen weniger aus dem eigentlichen Anwendungsproblem, sondern daher, weil man dem Rechner möglichst viele – wenn nicht gar alle – Funktionen übertragen will (um Hardware einzusparen oder um zusätzliche Funktionen vorzusehen, etwa in Hinsicht auf Bedienkomfort und Kommunikation).

Zudem ergeben sich schwierige Kommunikationsprobleme (z. B. wenn zwei Anwendungen auf dieselbe Datei zugreifen wollen). Deshalb verwenden die modernen Betriebssysteme kompliziertere Verfahren, um die zu erledigende Arbeit auf die vorhandenen Prozessorfunktionen aufzuteilen.

Es wird alles nicht wirklich, sondern nur scheinbar gleichzeitig erledigt
Die Programme werden abschnittsweise zeitversetzt ausgeführt. Beispielsweise läuft zunächst Programm A. Nach 20 Millisekunden wird auf Programm B umgeschaltet, dann auf Programm C usw. Das Umschalten wird vorzugsweise mit Software implementiert (Betriebssystem).

Hardwarelösungen wurden zwar schon recht früh entwickelt (das klassische Beispiel: die E-A-Prozessoren des Großrechners CDC 6600 (1963)), haben sich aber bisher am Markt nicht durchsetzen können. Sie sind aufwendiger und nicht so flexibel, haben aber auch bedeutsame Vorteile. Deshalb versucht man sich immer wieder daran (Beispiel: Intels HyperThreading-Technologie).

Tasks – Prozesse – Threads – Fibers
Eine *Task* (Aufgabe) ist ein in sich abgeschlossenes, lauffähiges (Anwendungs-) Programm. Das (scheinbar) gleichzeitige Ausführen mehrerer Tasks heißt Multitasking. Task, Multitasking, Multiprogramming usw. sind übliche Allgemeinbegriffe. Zu jeder Software-Plattform gehören aber jeweils spezifische Fachausdrücke. So spricht man im Zusammenhang mit dem Multitasking oft von Prozessen und Threads. Zunächst sollen die einschlägigen Fachbegriffe der Windows-Betriebssysteme kurz erläutert werden[10]:

Ein *Programm* ist eine ausführbare Folge von Maschinenbefehlen (Code), ergänzt durch die jeweils erforderlichen konstanten Daten.

Ein *Prozess* ist die Sammelbezeichnung für die Ressoucen, die es ermöglichen, Threads auszuführen – mit anderen Worten, ein Prozess ist etwas, das die Gelegenheit bietet, Programme laufen zu lassen. Die Ressourcen umfassen u. a. einen virtuellen Speicheradressraum sowie Steuer- und Pufferbereiche (Abb. 2.2).

Ein *Thread* ist ein laufendes Programm unter Einschluss der ihm zugeordneten Ressourcen, die zur Programmausführung erforderlich sind. In einem Prozess können ein Thread oder mehrere Threads ausgeführt werden. Werden mehrere Threads gleichzeitig (parallel) ausgeführt, spricht man vom Multithreading. Die Ressourcen eines Thread umfassen u. a. den Registersatz des Prozessors (Prozessorzustand, Hardware-Kontext), zwei Stackbereiche (je einen für den Anwender- und für den Systemzustand) sowie einen eigenen Speicherbereich für Bibliotheksprogramme, z. B. DLLs (Abb. 2.3).

10: Die Begriffsbildungen im Bereich Unix/Linux sind ähnlich. U. a. gibt es auch hier Prozesse und Threads.

Sind mehrere Threads lauffähig, so können diese scheinbar gleichzeitig auf einem Prozessor ausgeführt werden. Die Threads werden hierzu vom System in die Taskumschaltung einbezogen. Stehen mehrere Prozessorfunktionen zur Verfügung, können die Prozesse und Threads in verschiedenen Prozessoren gestartet werden. Die Zerlegung komplexer Anwendungen in parallel ablauffähige Threads ist eine grundsätzliche Voraussetzung, um Maschinen mit mehreren Prozessorkernen überhaupt sinnvoll nutzen zu können.

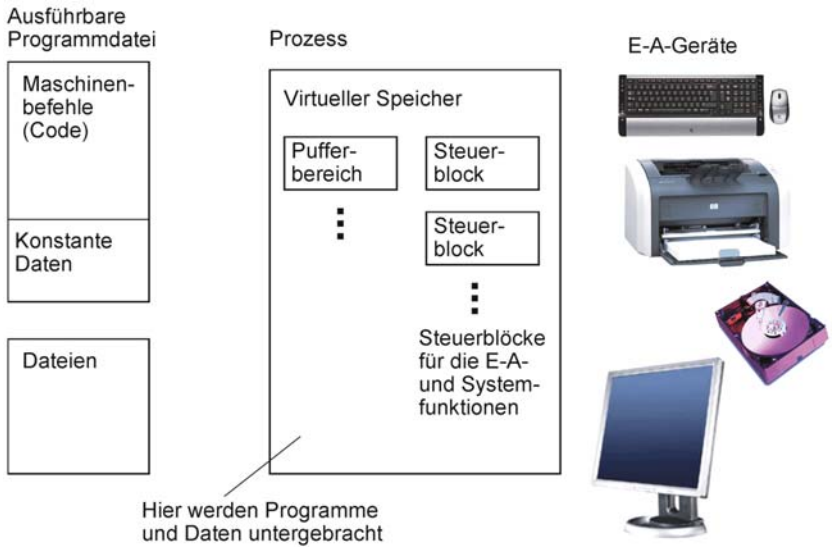


Abb. 2.8 Der Prozess ist eine Sammung systemseitiger Ressourcen, die erforderlich sind, um Programme auszuführen. Hierzu braucht man nicht nur Speicherplatz für den Programmcode, sondern auch Pufferbereiche und Steuerblöcke, mit deren Hilfe das System die Nutzung der E-A-Geräte, die Dateizugriffe und die Kommunikation zwischen den Prozessen organisiert.

Betrachten wir die Ausführung einer üblichen Anwendung:

- Beim Starten (durch entsprechendes Anklicken) beziehen wir uns auf eine Programmdatei (Beispiel: AcroRd32.exe für den Acrobat Reader).
- Das System richtet, um diese Anwendung laufen zu lassen, zunächst einen Prozess ein.
- In diesem Prozess wird ein Thread angelegt und das Programm gestartet.
- Das laufende Programm kann weitere Threads und Prozesse erzeugen.

Als *Fiber*[11)] werden Programme bezeichnet, die innerhalb von Threads ausgeführt werden können. Eine Fiber besteht aus dem jeweils auszuführenden Programmcode und einem eigenen Stackbereich. Ansonsten nutzen sie die Ressourcen des jeweiligen Threads und des Prozesses. Fibers werden nicht über Handles, sondern über Adressen angesprochen. Die Laufzeit der Fibers wird nicht vom Betriebssystem vergeben. Stattdessen muss die Ausführung einer Fiber programmseitig angewiesen werden. Hierbei gibt es zwei Verfahrensweisen:

11: Wörtlich = Faser.(Weil Threads (Fäden) aus Fasern bestehen ...)

- Master-Slave Scheduling. Eine Fiber ist gleichsam der Chef. Sie bestimmt, welche der anderen Fibers jeweils Laufzeit erhält.
- Peer-to-Peer Scheduling. Alle Fibers sind gleichberechtigt. Die Fiber, die gerade Laufzeit hat, bestimmt, welche als nächstes Laufzeit erhält. Eine typische Form der Laufzeitvergabe ist das zyklische Weiterschalten von Fiber zu Fiber (Round Robin Scheduling).

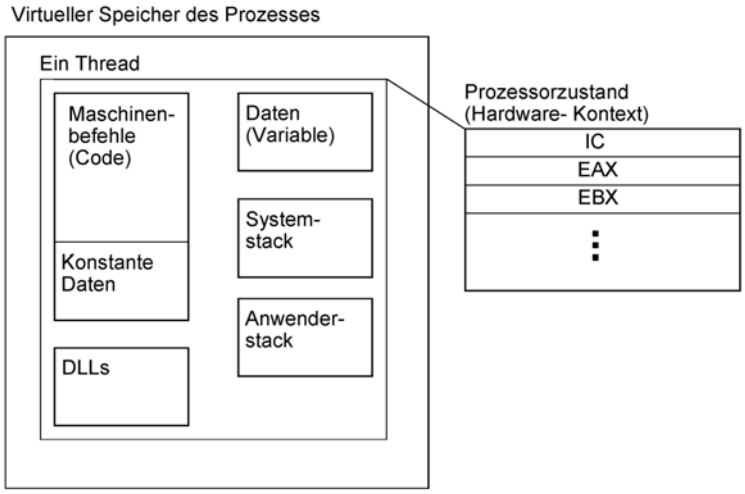


Abb. 2.9 Im Prozess ist ein Thread zum Ausführen eines Programms erzeugt worden. Im virtuellen Speicher des Prozesses können mehrere Threads untergebracht sein. Zum Thread gehören sowohl die entsprechenden Speicherbereiche als auch die programmseitig zugänglichen Registerinhalte des Prozessors.

Zusammenfassung:

- Ein Prozess ist eine komplette Ausführungsumgebung einschließlich dem virtuellen Speicher sowie den Vorkehrungen zur Dateiverwaltung und zum Steuern der Ein- und Ausgabe.
- Ein Thread ist die Ablaufumgebung für ein einzelnes Programm innerhalb eines Prozesses.
- Prozesse erfordern vergleichsweise viel, Threads vergleichsweise wenig Verwaltungsaufwand. Threads heißen deshalb gelegentlich auch “leichtgewichtige” (lightweigth) Prozesse. Fibers brauchen noch weniger Verwaltungsaufwand. Sie können ihrerseits als “leichtgewichtige” Threads angesprochen werden.
- Weil die Threads so wenig Verwaltungsaufwand (Overhead) erfordern, ist es zweckmäßig, das Anwendungsprogramm von vornherein in mehrere Threads aufzuteilen (Parallelisierung).
- Fibers ermöglicht es dem Anwendungsprogrammierer, eine Art elementaresMultitasking auf eigene Rechnung zu implementieren, wobei er die Ausführungsreihenfolge vollständig unter Kontrolle hat[12)].

12: Die Ausführungsreihenfolge der Threads bestimmt hingegen das System; die Anwendung hat darauf keinen Einfluss.

- Eine Anwendung, die nur als ein einziger Thread ausgeführt wird, kann auch nur auf einem einzigen Prozessor laufen. Mehrere Threads können hingegen auf mehreren Prozessoren parallel ausgeführt werden. Somit sind mehrere Prozessorkerne auch in der Lage, den Ablauf der einzelnen Anwendung zu beschleunigen[13].

Zum Anlegen von Prozessen, Threads und Fibers gibt es es entsprechende Systemfunktionen[14]. Der Prozess, der einen weiteren Prozess erzeugt, heißt Elternprozess (Parent Process), der erzeugte Prozess Kindprozess (Child Process). Es gibt Hilfsprogramme, die Näheres zu den im PC laufenden Prozessen und Threads darstellen (Abb. 2.4 bis 2.6)[15].

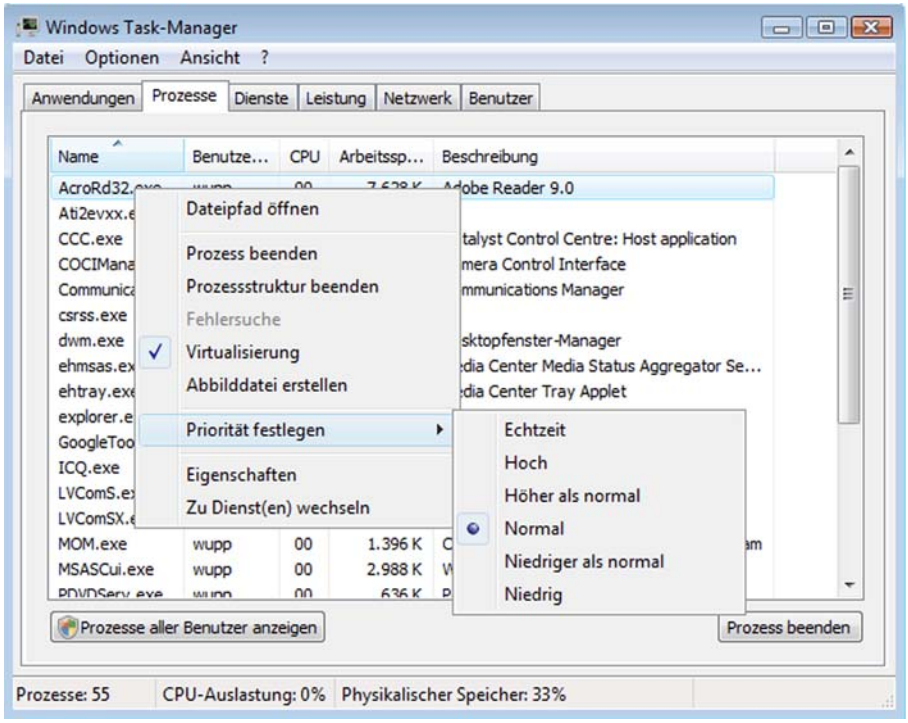


Abb. 2.10 Die elementare Prozessanzeige in Windows – der Taskmanager. Unter "Eigenschaften" kann man u. a. die Priorität abfragen und ändern.

13: Deshalb verteilen die Systeme vorrangig die Threads auf die verfügbaren Prozessorfunktionen und ordnen nicht einfach jeder Anwendung einen eigenen Prozessor zu.

14: In Windows heißen sie *CreateProcess* und *CreateThread*, in Unix/Linux *fork* und *clone*.

15: Höher entwickelte Programme – die u. a. anzeigen können, von wem welcher Prozess erzeugt wurde – sind aber nicht ganz mühelos zugänglich. Man muss im Internet suchen und womöglich eine entsprechende Programmentwicklungsumgebung installieren ...

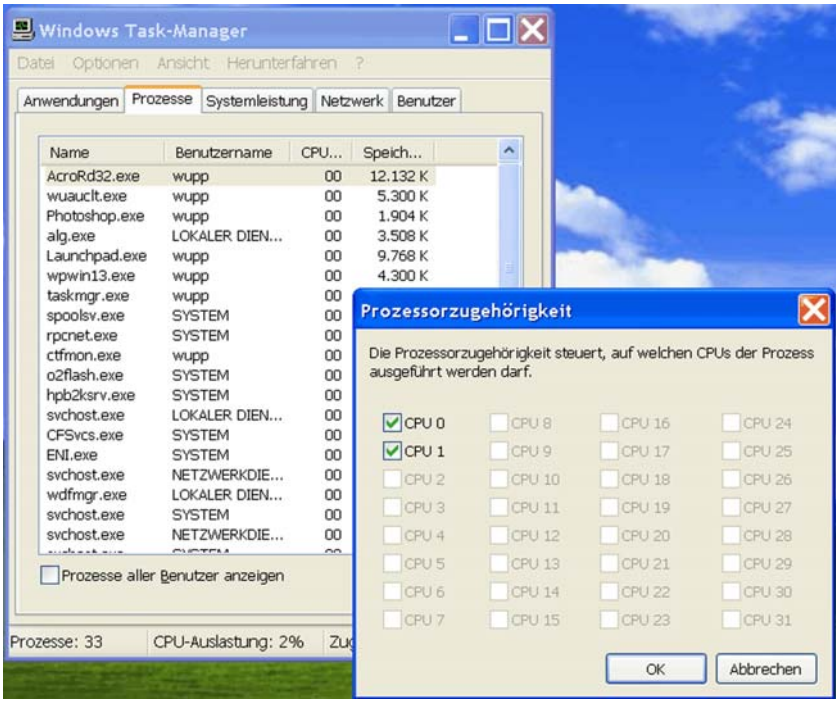


Abb. 2.11 Hat die Maschine mehr als eine Prozessorfunktion, kann man auch festlegen, auf welchen Prozessoren der Prozess laufen darf.

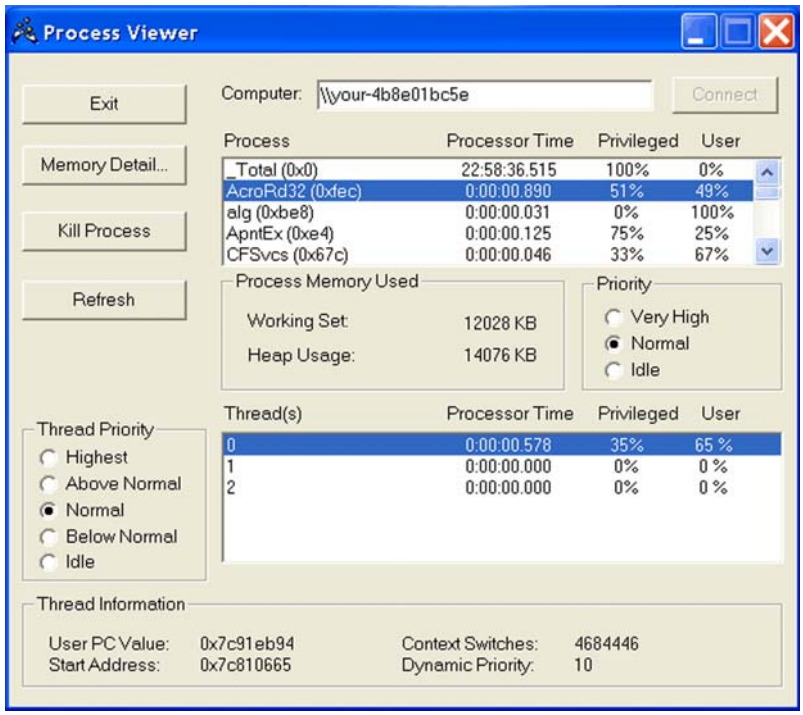


Abb. 2.12 Der Process Viewer ist ein Spezialprogramm (Microsoft), das nähere Einzelheiten erkennen lässt (u. a. auch die zum Prozess gehörenden Threads).

2.1.2 Nutzer und Anwendungsprogramme

Die folgenden beiden Fragen gehören zu den ersten, die sich der Entwickler einer Software-Plattform vorlegen muss:

- Wie viele Nutzer (User) sollen gleichzeitig mit dem System arbeiten können?
- Wie viele Anwendungsprogramme (Tasks) sollen gleichzeitig lauffähig sein?

Tabelle 2.1 nennt die grundsätzlichen Antworten, die es auf beide Fragen gibt.

Anzahl der Nutzer	Anzahl der gleichzeitig lauffähigen Anwendungsprogramme	Bezeichnung	Beispiele
1	1	Single User, Single Task	DOS
1	Mehrere	Single User, Multitasking	OS/2, alle Arbeitsplatz-versionen von Windows
Mehrere	Mehrere	Multi User, Multitasking	Alle Server-Versionen von Windows, Unix

Tabelle 2.1 Unterscheidung von Software-Plattformen nach den Arbeits-möglichkeiten für Nutzer und Anwendungsprogramme.

Single User, Singe Task
Ein einzelner Nutzer sitzt vor dem Computer, und es kann jeweils nur ein Anwendungsprogramm laufen. Das ist der einfachste Fall. Er stellt auch an die Laufzeitvergabe die geringsten Anforderungen. Die Plattform wirkt lediglich als Programmstarter (Program Launcher) für die jeweilige Anwendung.

Single User, Multitasking
Ein einzelner Nutzer sitzt vor dem Computer, es können aber mehrere Anwendungsprogramme gleichzeitig lauffähig sein. Hier muss die Plattform die Laufzeit wirklich verwalten. Grundsätzlich handelt es sich darum, die Laufzeit sozusagen stückweise zu verteilen. Beispielsweise wird Programm A 100 ms lang laufen gelassen. Dann greift die Laufzeitvergabe ein und teilt Programm B 50 ms zu, dann Programm C 200 ms usw. Dem Benutzer erscheint es so, als ob alle drei Programme praktisch gleichzeitig arbeiten. Dies zu organisieren ist nicht einfach:

- Das System muss sich merken, an welcher Stelle einem Programm die Laufzeit entzogen wurde.
- Sinngemäß sind Maschinenzustände, Zwischenergebnisse usw. zu retten (beim Entzug der Laufzeit) und wieder einzustellen (wenn dem Programm erneut Laufzeit zugesprochen wird).

- Die Speicher- und Geräteverwaltung muss mit der Laufzeitvergabe in Einklang stehen. Beispiel: Programm A hat die ersten Zeichen auf den Drucker ausgegeben. Nun erhält Programm B Laufzeit und möchte auch drucken. Es ist ersichtlich, dass – sollte das System das so durchgehen lassen – auf dem Papier ein Wust durcheinandergewürfelter Zeichen zu sehen sein würde.
- Es darf nicht sein, dass bestimmte Programme stets Laufzeit erhalten und andere kaum oder gar nicht zum Zuge kommen,
- Es darf nicht sein, dass das Programm, das gerade läuft, Daten eines anderen Programms verfälscht.

Dass es nur einen Nutzer gibt, vereinfacht die Aufgabe etwas, denn es ist nur die Kommunikation über einen Satz von Bedien- und Anzeigeeinrichtungen zu unterstützen.

Multi User, Multitasking
Mehrere unabhängige Nutzer können gleichzeitig mit dem System arbeiten. Typischerweise sind an ein solches System mehrere Bildschirme, Tastaturen usw. angeschlossen (Mehrplatzsysteme). Diese Auslegung stellt die höchsten Anforderungen an die Systemplattform. Die einzelnen Nutzer sind sowohl angemessen mit Laufzeit zu versorgen als auch voreinander zu schützen. Es darf z. B. nicht sein, dass ein Nutzer auf vertrauliche Datenbestände eines anderen zugreifen kann.

2.1.3 Taskzustände

Die Wirkprinzipien der “richtigen” Betriebssysteme sind kompliziert. Das liegt aber vor allem am Umfang und an den vielen Spitzfindigkeiten. Es ist aber keinesfalls schwierig, die eigentlichen Grundlagen zu verstehen. Um das Umschalten zwischen verschiedenen Programmen zu erläutern, wollen wir zum Allgemeinbegriff “Task” zurückkehren. In der Rechnerarchitektur bezeichnet der Begriff “Task“ die Umgebung (Register, Speicherbereiche usw.), in der ein einzelnes Programm lauffähig ist. Multitasking bedeutet, dass mehrere Programme gleichzeitig lauffähig sein können. In einem Einzelprozessor kann aber jeweils nur ein Programm tatsächlich arbeiten. Die betreffende Task heiß die arbeitende (Running) Task. Andere Programme können arbeitsfähig sein, das heißt, mit der Befehlsausführung beginnen, sobald die betreffende Task Laufzeit erhält. Solche Tasks heißen aktive (Busy) Tasks. Weiterhin ist es möglich, Task-Umgebungen im Speicher zu halten, die nicht unmittelbar arbeitsfähig sind und sie nur unter bestimmten Bedingungen arbeitsfähig zu machen. Solche Tasks heißen inaktive (Not Busy oder Idle) Tasks.

Im einfachsten Fall hat die Task nur zwei Zustände (Abb. 2.7): den Ruhezustand (Idle) und den Laufzustand (Running). Dieses Schema ist dann anwendbar, wenn das in der Task laufende Programm nur vergleichsweise wenig Zeit braucht, um seine Arbeit zu erledigen. Richtwerte zur maximalen Laufzeit: ca. 5...100 ms. Hierbei bestimmt die Laufzeit der jeweils aktiven Task die Latenzzeiten der anderen Tasks (das jeweils aktive Programm muss erst durchgelaufen sein, bevor ein anderes starten kann).

In folgenden Fällen ist es erforderlich, der jeweils laufenden Task Laufzeit zu entziehen (Taskumschaltung):

- es gibt Tasks mit längerer Laufzeit,
- es gibt Tasks, die “ewig“ laufen (Endlosschleifen),
- es gibt Tasks, die nicht so lange warten können, bis die arbeitende Task wieder ihren Ruhezustand einenommen hat (zu lange Latenzzeit).

Die grundsätzliche Lösung besteht darin, einen weiteren Taskzustand einzuführen (Abb. 2.8): die Task ist zwar noch aktiv, hat aber momentan keine Laufzeit (Busy). Dieser Zustand wird vom System aus eingestellt und auch wieder verlassen (Laufzeitvergabe)

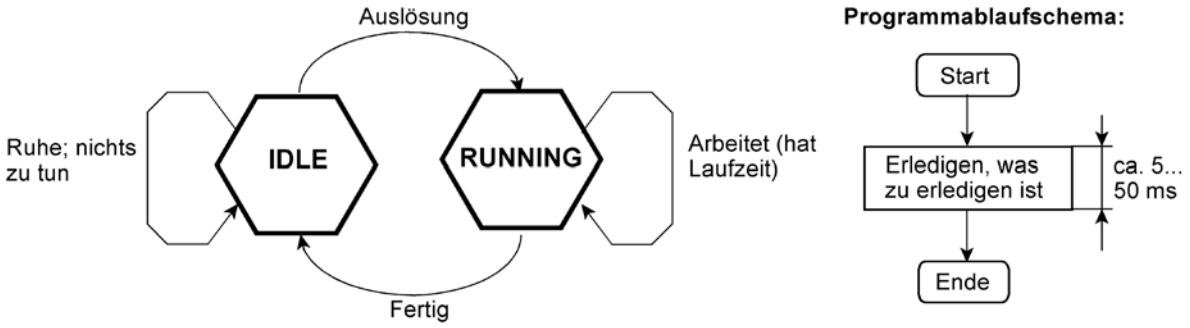


Abb. 2.13 Der einfachste Fall: nur zwei Taskzustände.

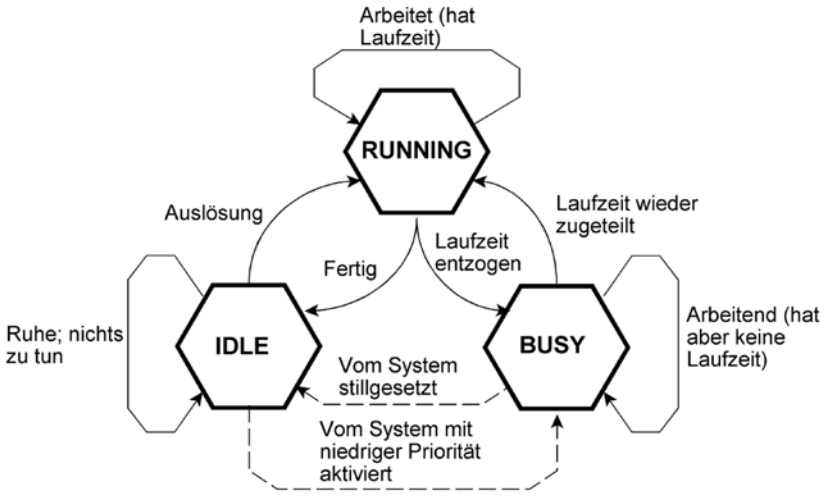


Abb. 2.14 Die grundsätzlichen Taskzustände beim Multitasking mit Laufzeitvergabe.

In höher entwickelten Systemen gibt es weitere Taskzustände, die die elementaren Zustände von Abb. 2.8 ergänzen. Sie haben mit Feinheiten der Laufzeitvergabe, mit der Unterstützung des Stromsparens und mit der Einhaltung von Realzeitanforderungen zu tun. Abb. 2.9 veranschaulicht ein Beispiel. Hier sind zwei zusätzliche Zustände vorgesehen:

1. Suspendiert (Suspended)

Befindet sich eine Task in diesem Zustand, so bekommt sie nie Laufzeit. Einleiten: mit Systemanweisung SUSPEND. Verlasssen: von einer anderen Task aus mit Systemanweisung WAKEUP (Normalfall) oder durch direkten Eingriff des Systems (z. B. zwecks Fehlerbehandlung). Typischer Nutzungsfall: Stromsparen. Wenn die von der betreffenden Task gesteuerte Hardware in einen Stromsparzustand versetzt wird, kann auch die zugehörige Task stillgelegt werden; es ist dann nicht mehr nötig, sie bei der Laufzeitvergabe zu berücksichtigen.

2. Verzögert (Delayed)

Die Task ist noch aktiv, erhält aber solange keine Laufzeit, bis eine bestimmte Verzögerungszeit abgelaufen ist. Einleiten: mit Systemanweisung DELAY[16)] (wobei die Verzögerungszeit (z. B. in Millisekunden)) als Parameter übergeben wird. Verlasssen: nach Ablauf der Verzögerungszeit (Normalfall) oder durch direkten Eingriff des Systems (z. B. zwecks Fehlerbehandlung). Typischer Nutzungsfall: es sind soundso viele Millisekunden zu warten. Währenddessen können andere Tasks Laufzeit erhalten. Ein weiterer Vorteil gegenüber dem programmseitigen Auszählen des Zeitintervalls (Wartschleife): die Verzögerungszeit wird unabhängig vom Programmablauf mit Hilfe eines Systemzeitgebers bestimmt. Sie kann deshalb – unabhängig von den Zeitverhältnissen der Befehlsausführung (Instruction Timing) – mit hoher Genauigkeit eingehalten werden.

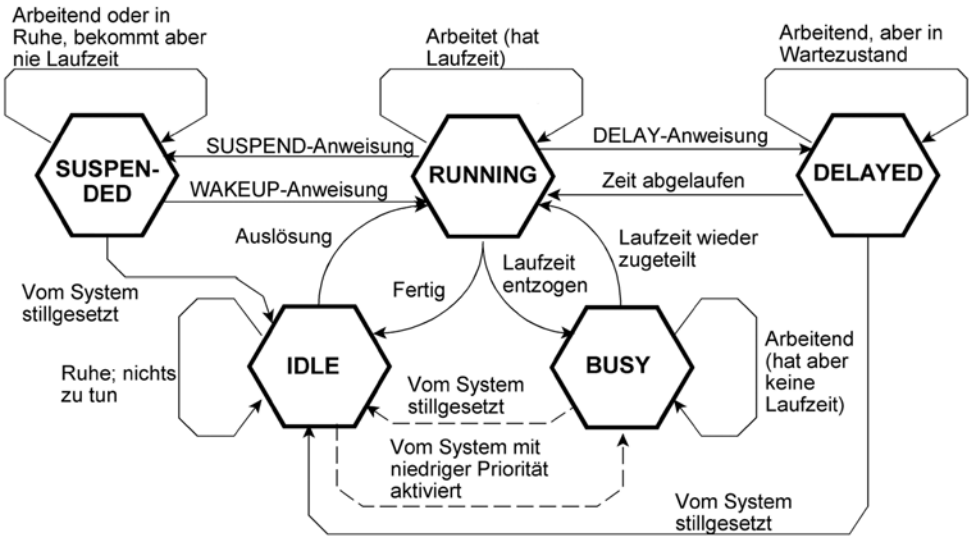


Abb. 2.15 Typische erweiterte Taskzustände.

16: Abb. 2.9 betrifft ein anwendungsspezifisches Realzeitsystem, das zwar vergleichsweise einfach ist, aber viele typische Merkmale moderner Betriebssysteme aufweist. Die der DELAY-Funktion entsprechende Windows-Funktion heißt SLEEP.

Solche Zustände – und die entsprechenden Systemfunktionen zum Auslösen von Zustandsübergängen – gibt es in vielen höher entwickelten Systemen, auch in Windows und Unix/Linux. Abb. 2.10 veranschaulicht die Zustände eines Windows-Threads:

Eingerichtet (Initialized)
Der Speicher wurde zugewiesen, die Programmdatei wurde geladen, die steuernden und beschreibenden Datenstrukturen wurden eingerichtet.

Lauffähig (Ready)
Der Thread wartet darauf, Laufzeit zu erhalten. Hierzu wurde er in die Laufzeitwarteschlange des Systems eingetragen.

Ausführungsbereit (Standby)
Der Thread wird bei der folgenden Taskumschaltung Laufzeit erhalten (ist also praktisch zur Ausführung vorgemerkt). Es kann sich jeweils nur ein einziger Thread in diesem Zustand befinden.

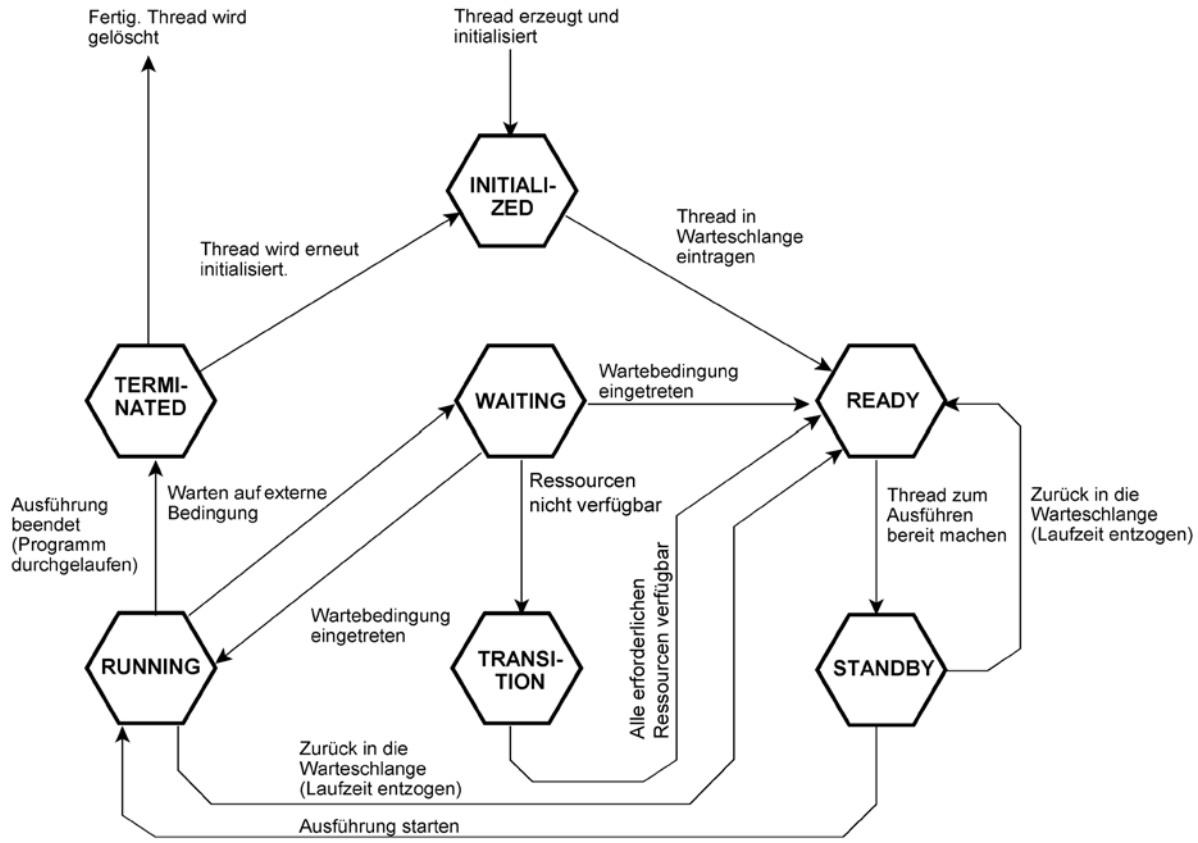


Abb. 2.16 Die Zustände eines Windows-Threads.

Arbeitend (Running)
Der Thread hat tatsächlich Laufzeit. Das Programm wird ausgeführt. Von diesem Zustand aus sind folgende Übergänge möglich:

- Zum Endzustand (Terminated), wenn das Programm bis zum Ende durchgelaufen ist.
- Zum Wartezustand, wenn im Programmabauf eine Wartebedingung auftritt, beispiels weise weil auf Ergebnisse eines anderen Threads oder auf die Verfügbarkeit eines E-A-Gerätes gewartet werden muss.
- Zum Zustand der Lauffähigeit (Ready), wenn das System dem Thread Laufzeit entzogen hat, um andere Threads laufen zu lassen.

Wartend (Waiting)
Es ist eine Wartebedingung aufgetreten. Der Zustand wird verlassen, sobald die Wartebedingung erfüllt ist. Je nachdem, ob dem Thread Laufzeit zugesprochen oder entzogen wurd, erfolgt dann ein Übergang in den Arbeitszustand (Running) oder in den Zustand der Lauffähigkeit (Ready).

Übergang (Transition)
Dieser Zwischenzustand ergibt sich dann, wenn der Thread ausführungsbereit ist, der aber noch nicht alle Ressourcen, die er benötigt, verfügbar sind. Sind alle Ressourcen verfügbar geworden, ist der Thread lauffähig (Ready).

Beendet (Terminated)
Der Thread gelangt in diesen Zustand, wenn die Programmausführung beendet ist. Er verbleibt dort, solange noch aktive Handles auf den Thread verweisen[17]. Ist das nicht der Fall, wird der Thread gänzlich gelöscht.

2.1.4 Taskumschaltung

Multitasking-Vorkehrungen stellen unabhängige Umgebungen für unabhängige Programme zur Verfügung (Abb. 2.11). Auf einem Prozessor kann nur ein Programm zu einer Zeit laufen. Wenn ein Programm läuft, so belegt es die Register des Prozessors. Taskumschaltung heißt, die entsprechenden Angaben zu retten und durch die Angaben jener Task zu ersetzen, die als nächste Laufzeit erhalten soll (Abb. 2.12).

Die “Umgebung“ (der Kontext) einer Task besteht aus den Inhalten der Prozessorregister, ihrem Stackbereich, eigenen Arbeitsbereichen usw. Hinzu kommen Steuerangaben, die den Taskzustand beschreiben. Für jede Task sind Speicherbereiche erforderlich, die die Angaben der Task-Umgebung aufnehmen können.

17: Das gibt anderen Threads die Möglichkeit, die internen Zustände des Threads zu analysieren.

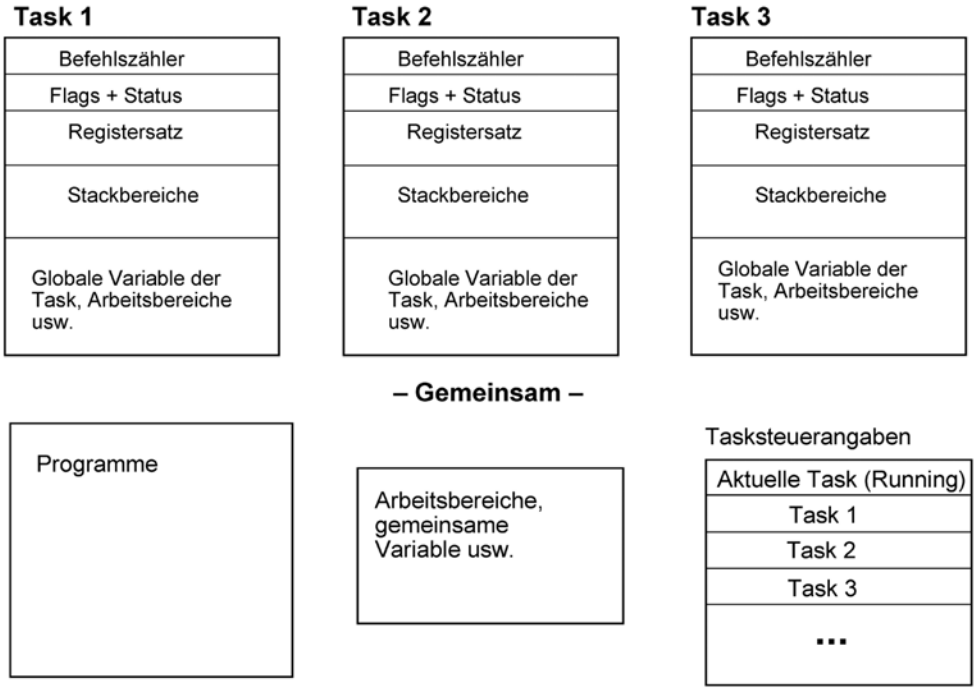


Abb. 2.17 Task-Umgebungen im Prozessor.

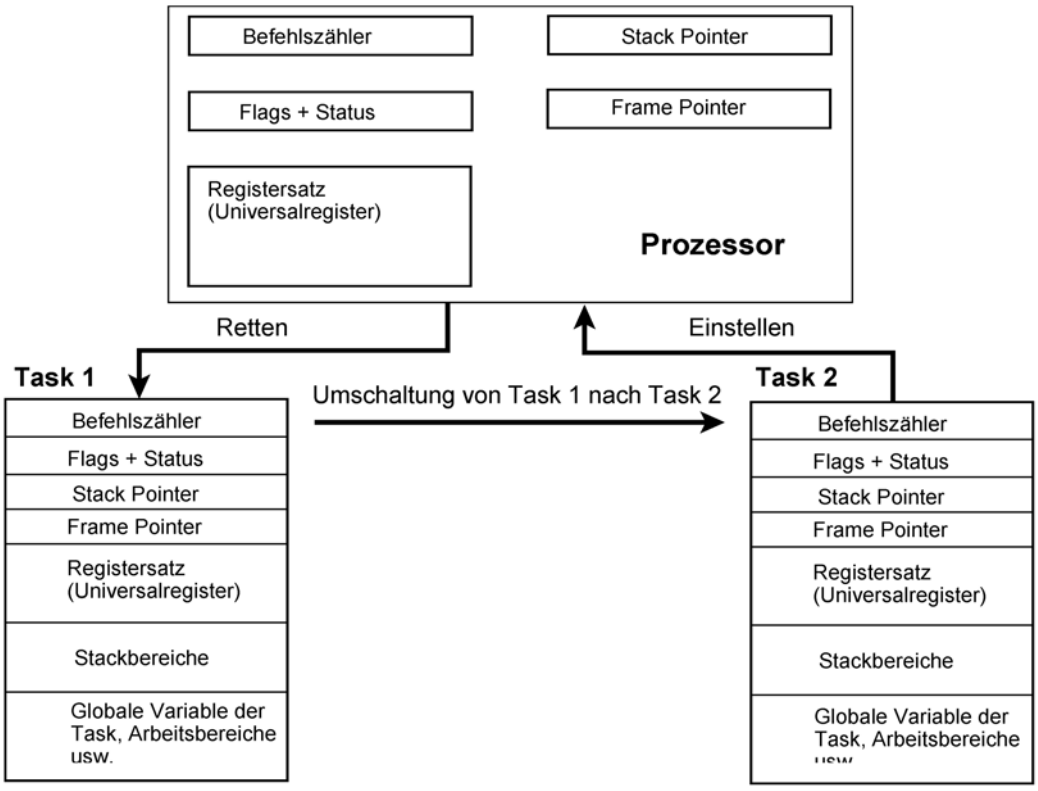


Abb. 2.18 Der grundsätzliche Ablauf einer Taskumschaltung.

Jede Task hat einen Rettungsbereich für die Prozessorregister. Wird einer Task Laufzeit entzogen, so werden die Registerinhalte in den betreffenden Rettungsbereich transportiert. Es

ist alles zu retten (einschließlich Befehlszähler, Stackpointer usw.), so dass später – nachdem erneut Laufzeit zugeteilt wurde – die Task ihre Arbeit wieder aufnehmen kann. Die Registerinhalte der Task, die Laufzeit erhalten soll, werden aus derem Rettungsbereich in den Prozessor gebracht. In diesem Zusammenhang erledigt sich auch die Umschaltung des Stacks, der privaten Arbeitsbereiche usw. (durch entsprechendes Laden des Stackpointers, des Frame Pointers, der Adressregister usw.). Als Letztes wird der Befehlszähler aus dem Rettungsbereich geladen. Anschließend wird der erste Befehl der neuen Task ausgeführt.

Wie viel Zeit eine Taskumschaltung erfordert, hängt vor allem vom Umfang der jeweiligen Taskumgebung ab. Komfortable Systeme haben extrem umfangreiche Taskumgebungen. Die Umgebung eines Prozesses ist viel komplexer als die eines Threads. Beispielsweise ist das Umschalten zwischen zwei Windows-Anwendungen ein Umschalten zwischen Prozessen. Hierbei müssen nicht nur die Prozessorregister, sondern auch Fenster, geöffnete Dateien, die Seitentabellen des virtuellen Speichers, anhängige Kommunikationsvorgänge usw. berücksichtig werden. Deshalb dauert eine solche Kontextumschaltung viele Milllisekunden[18]. Dem gegenüber erfordert es viel weniger Zeit, zwischen zwei Threads innerhalb einer Anwendung umzuschalten[19].

2.1.5 Laufzeitvergabe

Programmseitiges (kooperatives) Multitasking
Das gerade laufende Programm entscheidet selbst, wann es andere Programme zum Zuge kommen lässt. Beispielsweise gibt es hierfür einen entsprechenden Systemaufruf (nennen wir ihn – ohne Rücksicht auf ein bestimmtes System – BREAK[20]). Der Programmierer muss solche BREAK-Anweisungen gelegentlich in sein Programm einstreuen (hierfür gibt es Erfahrungswerte). Das System reagiert auf BREAK folgendermaßen:

- Steht ein weiteres Programm bereit, das auch Laufzeit haben möchte, so erhält es tatsächlich Laufzeit.
- Fordert kein anderes Programm Laufzeit an, wird das ursprüngliche Programm im Anschluss an BREAK fortgesetzt.

Das ist ein seit langem bewährtes Prinzip, das in vielen Embedded Systems angewendet wird[21]. Es kommt ohne zusätzliche Hardware und ohne Unterbrechungssystem aus. Zudem hat es der Programmierer in der Hand, die Kontrolle über die Hardware nur dann abzugeben, wenn es

18: Richtwert: Das Umschalten zwischen zwei Windows-Prozessen dauert etwa 20 ms, und zwar weithegend unabhängig vom Leistungsvermögen des Prozessors.

19: Die Thread-Umschaltung lässt die virtuellen Speicher in Ruhe. Es wird lediglich ein Zeiger auf das Seitentabellenverzeichnis desjeweiligen Prozesses gerettet.

20: In Windows: SLEEP.

21: Historische Beispiele im PC-Bereich: Windows 3.x und die ersten Macintoshs.

seinem Programm wirklich nicht schadet. Das Verfahren hat aber auch schwerwiegende Nachteile:

- Vergleichsweise hohe Latenzzeiten. Ein wartendes Programm erhält erst dann Laufzeit, wenn im gerade laufenden Programm eine BREAK-Anweisung ausgeführt wird (typischerweise im Abstand von mehreren bis vielen ms),
- Manche Programme sind nicht wirklich kooperativ (da sie z. B. zu wenige (oder gar keine) Umschaltanweisungen enthalten). Historische Beispiele: Viele Windows 3.x-Anwendungen[22)].

Zwangsweises Multitasking

Die Taskumschaltung wird vom System erzwungen (engl. Preemptive[23)] Multitasking); der Anwendungsprogrammierer muss sich nicht darum kümmern. Hierfür braucht man aber gewisse Vorkehrungen in der Hardware:

- einen vom Programmablauf unabhängigen Zeitgeber,
- ein Unterbrechungssystem,
- besondere Befehle, die es ermöglichen, zeitweilig zu verhindern, dass dem Programm Laufzeit entzogen wird.

Wir wollen hier nur den einfachsten Fall beschreiben, die Zuteilung sog. Zeitscheiben (Time Slicing). Der besagte Zeitgeber erzeugt in festen Abständen, z. B. alle 10 ms, eine Unterbrechung. Diese veranlasst, dass das Laufzeitvergabeprogramm[24)] gerufen wird, das dem nächsten lauffähigen Programm die Laufzeit zuteilt. Nehmen wir an, es seien drei Programme A, B, C lauffähig. Dann wird die Laufzeit beispielsweise wie folgt zugewiesen: für 10 ms an Programm A – für 10 ms an Programm B – für 10 ms an Programm C – für 10 ms an Programm A usw. Ein solcher Zeitabschnitt heißt *Zeitscheibe* (Time Slice). Dieses Prinzip der Laufzeitvergabe ist typisch für alle höher entwickelten Plattformen.

Vorrangsteuerung (Prioritäten)

Das eben beschriebene zyklische Weiterschalten zwischen den lauffähigen Programmen (Fachbegriff: Round Robin) hat den Vorteil, "fair" zu sein (jedes Programm bekommt den gleichen Anteil). Es führt aber nach wie vor zu beachtlichen Latenzzeiten (wurde z. B. gerade Programm A mit Laufzeit versorgt, so dauert es 20 ms, bis Programm C drankommt). Auch kann

---

22: Die Programmierer hätten eigentlich an allen geeigneten Stellen SLEEP-Anweisungen einstreuen sollen. Das haben aber viele nicht getan. Auch heute gehört es zum guten Programmierstil, an Stellen, an denen das Programm ohnehin nicht mehr tun kann, als etliche Millisekunden zu warten, eine SLEEP-Anweisung zu setzen.

23: Der in manchen Schrifte zu findende Ausdruck "präemptiv" mag sonst etwas heißen, richtiges Deutsch ist es jedenfalls nicht ...

24: Engl. Scheduler.

es gelegentlich sein, dass wir z. B. die Ergebnisse von Programm B unbedingt brauchen, während es bei Programm A auf einige hundert ms nicht ankommt. Hierfür hat man Maßnahmen der Vorrangsteuerung über Prioritäten (Priorities) eingeführt. Das kann z. B. so aussehen, dass einem Programm höherer Priorität mehrere Zeitscheiben zugewiesen werden. Beispiel: wir weisen dem Programm B eine höhere Priorität zu[25]. Der Scheduler vergibt nun zyklisch 10 ms an Programm A, 20 ms an Programm B, 10 ms an Programm C usw. Weitere Prioritätsfestlegungen können z. B. bewirken, dass bestimmte Unterbrechungen unmittelbar – also gleichsam außer der Reihe – die Zuteilung von Laufzeit an ein bestimmtes Programm veranlassen[26]. Richtwerte zur Dauer von Zeitscheiben (Windows):

- Arbeitsplatzsysteme: 15 bis 30 Millisekunden,
- Server: 90 bis 180 Millisekunden.

Freiwillig Laufzeit abgeben
Auch beim zwangsweisen Multitasking gibt es entsprechende Systemfunktionen. Sie werden dann aufgerufen, wenn der Thread darauf warten muss, dass irgend etwas Anderes erledigt wird. Sie bewirken, dass der Thread aus dem Arbeitszustand (Running) in den Wartezustand (Wait) übergeht.

Vorder- und Hintergrund
Diese Begriffe aus der "klassischen" EDV kennzeichnen eine grobe Einteilung von Programmen gemäß ihrer Priorität.

Vordergrund (Foreground)
Diese Programme haben höhere Prioritäten. Typischerweise lässt man jene Programme im Vordergrund laufen, mit denen die Nutzer (am Bildschirm) unmittelbar zusammenarbeiten.

Hintergrund (Background)
Die Programme müssen sich mit dem Rest an Laufzeit begnügen, der übrigbleibt (sie erhalten immer dann Laufzeit, wenn die Vordergrundprogramme gerade nichts zu tun haben). Ein typisches Beispiel ist eine im Hintergrund laufende Datensicherung. Wichtig: Hintergrundprogramme sollten "automatisch" durchlaufen, also keine Bedieneingriffe erfordern.

2.1.6 Gegenseitige Behinderungen

Deadlocks
Wenn beispielsweise Task A darauf wartet, dass Task B eine bestimmte Datei freigibt, B aber darauf, dass A endlich eine bestimmte Ausgabe erledigt hat, so bleiben beide Tasks hängen. Ein solcher Zustand heißt Verklemmung oder Deadlock. Es ist ein harter Fehlerzustand. Die Gefahr

25: Zur anwendungsseitigen Prioritätseinstellung unter Windows vgl. Abb.2.4.

26: Beispiel: Windows erhöht zeitweilig die Priorität eines Threads, nachdem Wartebedingungen erfüllt oder E-A-Abläufe beendet wurden (Priority Boosting).

von Deadlocks besteht dann, wenn – wie im Beispiel angedeutet – Tasks voneinander abhängig sind.

Livelocks
Als Livelock (Blockierung) bezeichnet man Zustände der Behinderung, die nicht durch gegenseitige Abhängigkeiten hervorgerufen werden. So kann z. B. ein System "klemmen", wenn Task A eine hohe Priorität hat und Task B eine niedrige, Task A aber das System laufend belegt, so dass Task B nie zum Zuge kommt.

Hinweise:

1. Sind die Tasks vollkommen unabhängig voneinander, so kann es auch keine Deadlocks geben.
2. Eine Laufzeitvergabe nach dem Round-Robin-Prinzip (zeitgesteuerte zyklische Weiterschaltung) vermeidet Livelocks. Trotzdem kann z. B. eine ungeschickte Prioritätseinstellung zu einer merklichen (bisweilen untragbaren) Verschlechterung der Systemleistung führen (typischerweise an extrem langen Antwortzeiten erkennbar).
3. Round Robin schützt nicht grundsätzlich gegen Deadlocks.
4. Der Theorie nach ist es möglich, bestimmte Deadlock-Situationen aus der Analyse der Programm-Quelltexte zu erkennen.
5. Manche Deadlocks können durch Ändern von Prioritäten tatsächlich abgestellt werden, manche nicht – das Experimentieren mit Prioritäten kann dann bestenfalls die Wahrscheinlichkeit verringern, dass eine Deadlock-Situation auftritt, die Gefahr selbst ist aber nach wie vor latent (man verschiebt das Auftreten lediglich auf einen späteren Zeitpunkt).

2.2 Speicherverwaltung

2.2.1 Was braucht ein Programm, um laufen zu können?

Die Speicherverwaltung hat die Aufgabe, den Programmen eine angemessene Speicherkapazität zur Verfügung zu stellen. Wie viel Speicher (z. B. in Bytes ausgedrückt) braucht aber ein Programm? - Es sind unterzubringen:

- das Programm selbst,
- die zugehörigen konstanten Daten,
- die veränderlichen Daten (Variablen),
- Arbeits- und Übergabebereiche,
- bedarfsweise Symbol- und Verweistabellen.

In einfachen Systemen kann man die verfügbare Speicherkapazität fest aufteilen (statische Speicheraufteilung). Moderne Hochleistungssysteme sind hingegen dadurch gekennzeichnet,

dass sich die Speicherbelegung ständig ändert (dynamische Speicheraufteilung). Abb. 2.13 veranschaulicht ein Prinzip, das häufig implementiert wird.

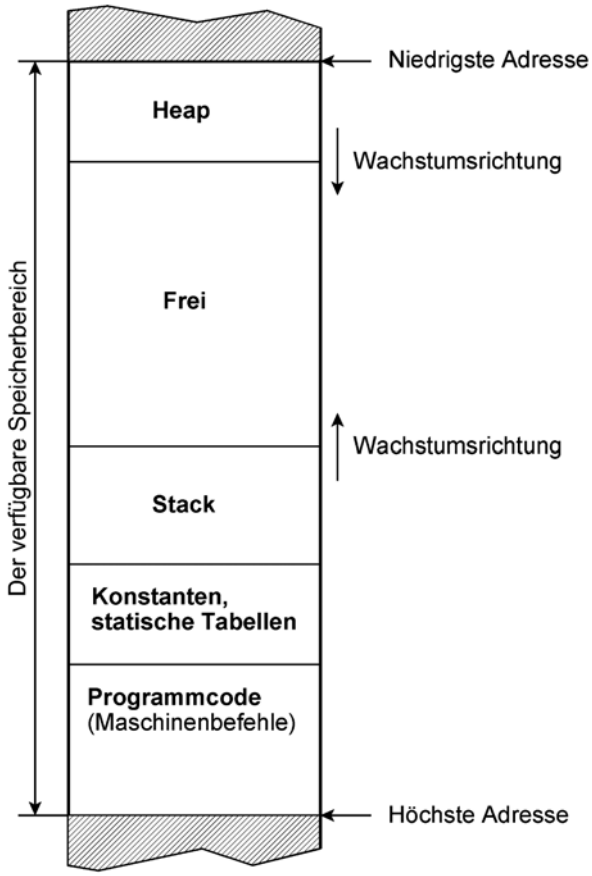


Abb. 2.19 Zum Prinzip der Speicheraufteilung (Beispiel).

Die Speicherverwaltung beginnt damit, dass ein hinreichend großer Speicherbereich bereitgestellt wird. Er wird folgendermaßen belegt:

1. Der Programmcode wird ganz hinten untergebracht.
2. Davor kommen die statischen – in ihrer Größe unveränderlichen – Datenbereiche (Konstanten, globale Variable, Symboltabellen usw.).
3. Im Anschluss daran – zu den niederen Adressen hin – wird ein Bereich veränderlicher Größe eingerichtet, der sog. Stack[27]. Er nimmt dynamische Variable, Parameter, lokale Variable, Zwischenergebnisse und Rückkehradressen auf. Er wächst in Richtung niederer Adressen.
4. Am Anfang des Speicherbereichs wird oftmals eine weitere veränderliche Struktur angeordnet, der sog. Heap[28]. Er wächst in Richtung höherer Adressen.

27: Wörtlich = Stapel.

28: Wörtlich = Haufen.

Sowohl Stack als auch Heap wachsen oder schrumpfen während der Ausführung des Programms. Durch die Anordnung an entgegengesetzten Enden ist stets gewährleistet, dass sich ein möglichst großer freier Bereich zwischen Stack und Heap befindet. Nur in dem – vergleichsweise unwahrscheinlichen – Fall, dass beide Strukturen wachsen und wachsen, kann es vorkommen, dass irgendwann einmal nichts mehr frei ist, dass also der Stack versucht, ein Stück des Heap zu belegen oder umgekehrt. Die Schutzvorkehrungen der Hardware oder das Laufzeitsystem der Software sollten diesen Fall erkennen und entsprechend reagieren (z. B. mit dem Abbruch der Programmausführung und einer entsprechenden Fehlermeldung).

2.2.2 Stack und Heap

Beide Informationsstrukturen sind von grundsätzlicher Bedeutung. Stacks werden automatisch verwaltet[29], Heaps müssen zu Fuß (soll heißen: vom Anwendungsprogrammierer)[30] verwaltet werden. Tabelle 2.2 gibt einen Überblick über die vorzugsweise Nutzung beider Strukturen.

	Stack	**Heap**
Nutzung (gespeichert werden...)	Rückkehradressen, lokale Daten (verschwinden bei Rückkehr aus der jeweiligen Funktion)	Dynamische Daten (bleiben solange erhalten, bis sie explizit (vom Programm) wieder freigegeben werden)
Belegung und Freigabe (Auf- und Abbau)	Automatisch gemäß dem LIFO-Prinzip	Typischerweise (vgl. Programmier-sprache C) vom Programmierer anzufordern und freizugeben
Besondere Eignung	Für kleinere und einfachere Datenstrukturen (zu beispielsweise 32 oder 64 Bits)	Für größere und kompliziertere Datenstrukturen (z. B. von 256 Bytes an aufwärts)

Tabelle 2.2 Zur Verwendung von Stack und Heap.

Der Stackmechanismus gewährleistet ein automatisches Wachsen und Schrumpfen des Stacks. Die Daten werden immer nur an einem Ende eingeschrieben und entnommen. Der Stack ist deshalb stets ein zusammenhängend belegter Speicherbereich ohne Lücken. Näheres im folgenden Abschnitt 2.3.3. Der Theorie nach kann man mit dem Stackmechanismus alle einschlägigen Probleme lösen. Es gibt sogar komplette Rechnerarchitekturen, die darauf beruhen. Der Stack wird aber uneffektiv, wenn das Programm mit größere Datenstrukturen umgehen muss, beispielsweise mit Bildpuffern, Warteschlangen und Matrizen. Der Heap ist als ergänzender Speicherbereich gedacht, um solche Datenstrukturen aufzunehmen. Er hat keinen starren Zugriffsmechanismus, sondern kann auf einfache Weise freizügig vom Programm verwaltet werden. Hierzu kommt man mit zwei elementaren Funktionen aus:

29: Manche Prozessoren – so auch die in PCs eingesetzten – unterstützen dies mit entsprechenden Registern und Maschinenbefehlen.

30: In manchen Sprachumgebungen (z. B. Java) erledigen das der Compiler und die Laufzeitumgebung.

- Speicherplatz anfordern (Allocate).
- Speicherplatz wieder freigeben (Free).

Die Allocate-Anweisung bewirkt, dass das System einen Bereich passender Größe sucht und einen Adresszeiger zurückgibt. Das Anwendungsprogramm kann dann in diesem Bereich tun, was es will. Wird die betreffende Datenstruktur nicht mehr gebraucht, so wird der zugehörige Bereich wieder freigegeben. Das Grundproblem dieses einfachen Verwaltungsprinzips: werden immer wieder Bereiche verschiedener Größe angefordert und freigegeben, so werden nach einiger Zeit kaum noch größere zusammenhängende Bereiche zu finden sein (Fragmentierung). Nähers dazu in Abschnitt 2.3.5.

2.2.3 Der Stack

Ein Stack ist eine Speicheranordnung, die eine gewisse Anzahl gleich langer Informationsstrukturen (Stack-Elemente) aufnehmen kann. Es gibt keinen wahlfreien Zugriff, sondern die Speicheranordnung wird implizit von einem Adresszähler (Stackpointer) adressiert.

Stackzugriffe
Es gibt nur zwei grundlegende Zugriffsabläufe:

- Push. Ein Push-Ablauf legt ein Element auf den Stack
- Pop. Ein Pop-Ablauf entnimmt das zuletzt (vom letzten Push) auf den Stack gelegte Element (beim nächsten Pop wird dann das vom vorletzten Push abgelegte Element entnommen usw.).

Die Stack-Organisation wird deshalb gelegentlich auch als LIFO (Last In, First Out) bezeichnet.

Wachstumsrichtung
Es ist eine reine Konventionsfrage, ob bei Push-Abläufen der Inhalt des Stackpointers erhöht und bei Pop-Abläufen vermindert wird oder umgekehrt. Üblicherweise wachsen Stacks in Richtung niederer Adressen. Der Inhalt des Stackpointers wird bei Push-Abläufen vermindert und bei Pop-Abläufen erhöht.

Zähl- und Zugriffsreihenfolge
Ebenso ist es eine reine Konventionsfrage, ob bei einem Push zunächst der Stackpointer verändert und dann das neue Element gespeichert wird oder umgekehrt. Die meisten Prozessoren verhalten sich folgendermaßen:

- Der Stackpointer zeigt auf das oberste Element im Stack (Top of Stack, TOS).
- Push: der Stackpointer-Inhalt wird zunächst vermindert (Predecrement), dann wird das Element gespeichert.
- Pop: das Element wird entnommen, dann wird der Stackpointer-Inhalt erhöht (Postincrement).

Stack-relative Adressierung
Es ist oft von Vorteil, wenn man zu Elementen des Stack auch wahlfrei zugreifen kann. So kann man auch untere Elemente im Stack erreichen, ohne die oberen zuvor entfernen zu müssen. Solche Zugriffe beziehen sich zweckmäßigerweise auf den Stackpointer, so dass das erste, zweite usw. Element im Stack für Lese- und Schreibzugriffe zugänglich ist, wobei der Stackpointer nicht verändert wird (explizite Stackzugriffe nach dem Prinzip Basis + Displacement mit dem Stackpointer als Basisadressregister).

Variabel lange Stackelemente
Gibt es nicht – typischerweise sind alle Elemente in einem Stack gleich lang. Kürzere Angaben werden zwecks Ablage auf dem Stack entsprechend erweitert, längere in Form mehrerer Elemente (z. B. 32- oder 64-Bit-Worte) gespeichert.

Stack Frames
Ein Stack Frame ist ein fester Bereich im Stack. Er dient vor allem dazu, die lokalen (statischen) Variablen des laufenden Programms aufzunehmen.

Statische und dynamische Variable
Statische Variable werden im Programmtext deklariert (jeder Variablenname wird angegeben, und es wird ihm ein Datentyp zugewiesen). Hierzu ein Beispiel (wir verwenden der leichteren Lesbarkeit halber eine an Pascal und Ada orientierte Syntax):

```
Artikel_Nr: Integer;                    -- 4 Bytes (ganze 32-Bit-Binärzahl)
Bezeichnung: String(64);                -- 64 Bytes (Zeichenkette)
Preis: Unpacked_BCD(16);                -- 16 Bytes (BCD-Zahl)
Länge, Breite, Höhe: Small_Integer;     -- je 2 Bytes (ganze 16-Bit-Binärzahlen)
Gewicht, Spezifisches_Gewicht: Float;   -- je 4 Bytes (32-Bit-Gleitkommazahlen)
usw.
```

Der Compiler muss allen Variablen entsprechenden Speicherplatz zuweisen.

Dynamische Variable entstehen hingegen im Laufe der Verarbeitung (also ohne dass sie der Programmierer ausdrücklich deklarieren muss). Beispiel: der Programmierer schreibt hin:

```
Gewicht := Länge * Breite * Höhe * Spezifisches_Gewicht;
```

Der Compiler muss diese Formel in eine Folge von Maschinenbefehlen umsetzen. Da die einzelnen Befehle nur ganz elementare Operationen ausführen können, fallen im Verlauf der Rechnung Zwischenergebnisse an. Das sind die dynamischen Variablen, die typischerweise auf dem Stack abgelegt werden.

Sowohl statische als auch dynamische Variable werden im Stack untergebracht
Das muss nicht unbedingt so sein, hat sich aber bewährt. Und zwar vor allem deshalb, weil man gern Programme in Programme schachtelt (Unterprogrammtechnik). Dann liegt es nahe, die verfügbare Speicherkapazität im Sinne eines Stack zu verwalten (vgl. auch Abb. 2.13). Zuerst kommt der Stack Frame des ersten Programms. Darüber (in Richtung zu den niederen Adressen hin) werden die gerade aktuellen dynamischen Variablen auf den Stack gelegt. Wenn nun das Programm ein Unterprogramm aufruft, kommt dessen Stack Frame auf den Stack, darüber werden dessen dynamische Variable abgelegt usw.

Das Zugriffsproblem
Gemäß dem Rechenablauf wächst oder schrumpft der Stack. Andererseits sind aber immer die gleichen statischen Variablen zu adressieren. Würde man sich aber stets auf den Stackpointer (als Basisadresse) beziehen, so würden sich bei jedem Zugriff andere realtive Adresen (Displacements) zu den statischen Variablen ergeben. Deshalb sieht man typischerweise ein weiteres Adressregister vor, den sog. Frame Pointer oder Base Pointer (Abb. 2.14). Der Base Pointer (Frame Pointer) zeigt stets auf den Anfang des aktuellen Stack Frame (d. h. auf das Wort an der jeweils höchsten Adresse). Alle Inhalte des aktuellen Stack Frame sind somit über negative Displacements (bezogen auf den Base Pointer) erreichbar. Ruft das aktive Programm seinerseits ein Unterprogramm, so wird der aktuelle Inhalt des Stackpointers in den Base Pointer übernommen, und oberhalb des dynamischen Bereichs des rufenden Programms wird der Stack Frame des gerufenen aufgebaut.

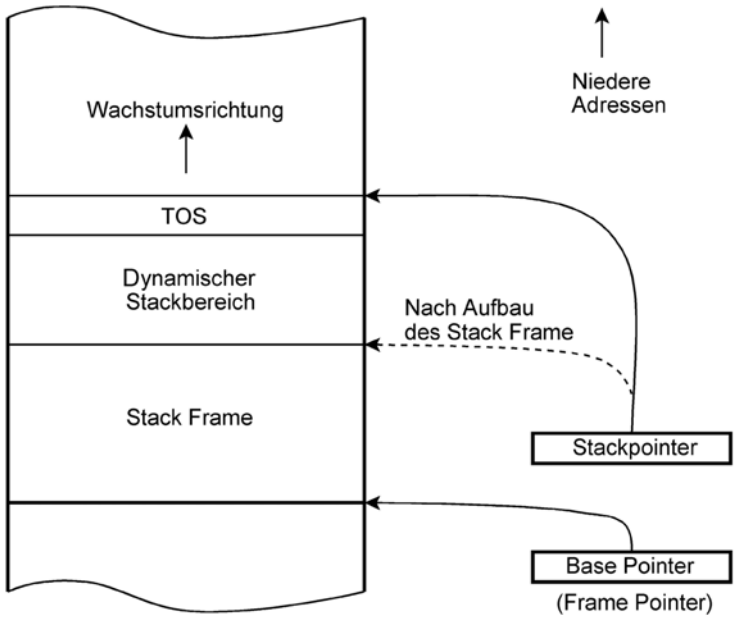


Abb. 2.20 Stack-Organisation mit Stack Frame.

Die UNIX-Stackorganisation

Diese Form der Stackorganisation ist zum Industriestandard geworden. Für jede Prozess (bzw. Thread) werden zwei Stacks verwaltet (Abb. 2.15):

- der Anwenderstack (User Stack) zum Aufrufen von Anwendungsprogrammen,
- der Systemstack (Kernel Stack) zum Aufrufen der Systemfunktionen.

Weshalb zwei Stacks?

Im Grunde würde ein Stack je Programm (Task, Prozess oder Thread) genügen. Dann hätte aber jeder Programmfehler, der eine Verfälschung des Stackinhalts oder der zugehörigen Adressregister bewirkt, einen totalen Absturz zur Folge. Auch das Betriebssystem könnte dann nicht mehr arbeiten. Deshalb ist man bestrebt, das System so auszulegen, dass es bei einem beliebigen Absturz der Anwendung noch betriebsfähig bleibt. Der Systemstack kann nur im Systemzustand verändert werden[31]. Fehler im Anwenderprogramm können sich deshalb nur auf den Anwenderstack auswirken. Der Systemstack bleibt intakt, so dass das System in der Lage ist, den Fehler zu behandeln[32].

Ein Programmaufruf läuft folgendermaßen ab (Abb. 2.16):

1. Das rufenden Programm legt die zu übergebenden Parameter auf den Stack. Aus Tabelle 2.3 sind typische Konventionen des Unterprogrammrufs ersichtlich.
2. Der Aufruf wird ausgeführt. Dabei gelangt die Rückkehradresse auf den Stack.
3. Das gerufenene Programm kopiert den bisherigen Frame Pointer auf den Stack (Adresszeiger als Rückverweis). Typischerweise wird der aktuelle Inhalt des Stackpointers zum neuen Frame Pointer[33].
4. Das gerufene Programm kopiert seine lokalen Variablen in den Stack (bzw. schafft auf dem Stack soviel Platz, dass die lokalen Variablen hineinpassen).
5. Der aktuelle Frame bzw. Base Pointer wird eingerichtet.

31: Vgl. auch den Übergangszustand (Transition) in Abb. 2.12. Der Systemstack muss im Arbeitsspeicher stehen, um überhaupt Systemaufrufe der arbeitenden Task annehmen zu können. Deshalb geht es bei ausgelagertem Systemstack zeitweilig nicht weiter.

32: Was aber, wenn das System selbst Fehler enthält und so seinen eigene Stack beschädigt? – Dann weiß es sich auch nicht mehr zu helfen und zeigt ggf. mit letzer Kraft den berühmten blauen Bildschirm ...

33: Dieser Ablauf wird gelegentlich von der Hardware unterstützt (z. B. IA-32-ENTER-Befehl).

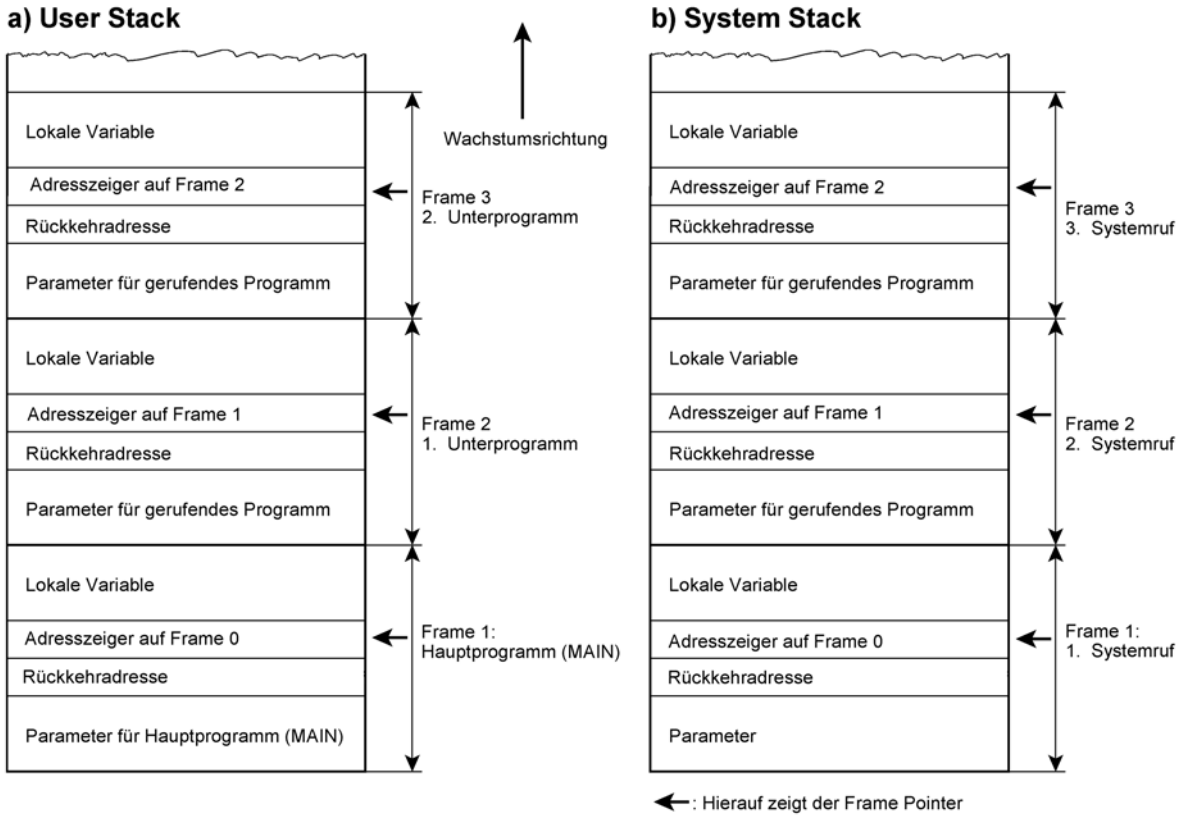


Abb. 2.21 Die Unix-Stackorganisation[34].

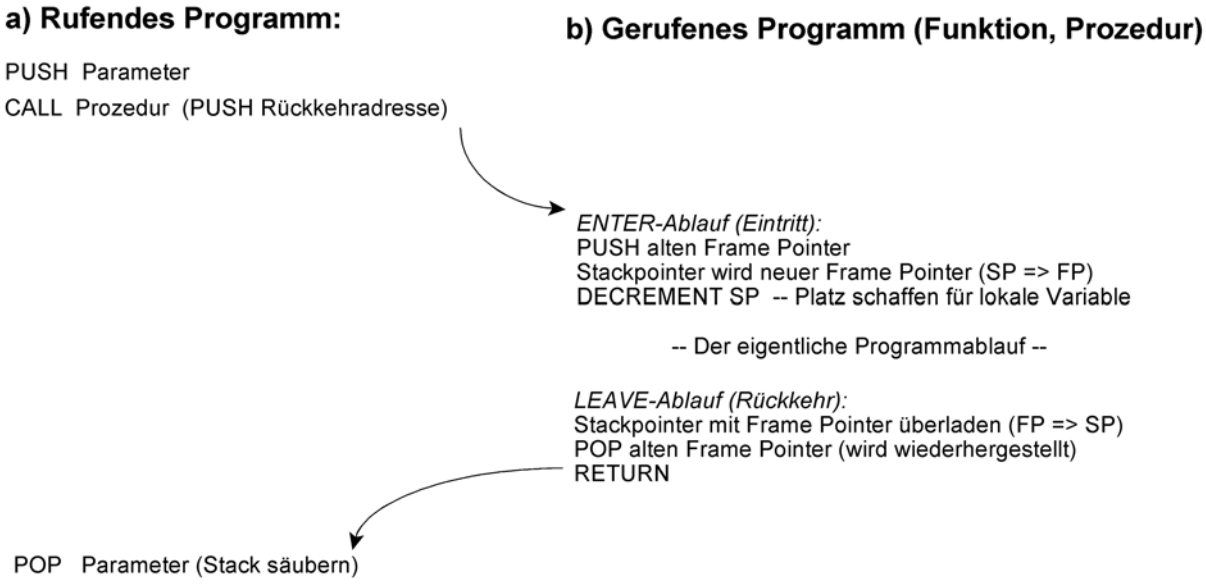


Abb. 2.22 Unterprogrammaufruf in einer Laufzeitumgebung, die auf Stack Frames beruht.

34: Hinweis: Einige Angaben in den Frames 1 (z. B. der Adresszeiger auf Frame 0) sind bedeutungslos. Sie wurden nur deshalb eingeführt, um alle Frames gleichartig aufbauen zu können.

	Pascal	**C**
Reihenfolge der Parameterübergabe	Von links nach rechts	Von rechts nach links
Wer stellt bei der Rückkehr die ursprüngliche Stackbelegung wieder her (Stack Cleanup)?	Das gerufene Programm	Das rufende Programm
Vor- und Nachteile der Stack-Cleanup-Konvention	• Cleanup-Ablauf nur einmal vorhanden (im gerufenden Programm). • Funktionsaufrufe typischerweise nur mit fester Parameteranzahl	• Cleanup-Ablauf in jedem rufenden Programm erforderlich. • Der erste Parameter (ganz links im Funktionsaufruf) kommt stets auf dem TOS zu liegen (das erleichtert Implementierung von Funktions-aufrufen mit variabler Parameteranzahl)

Tabelle 2.3 Typische Konventionen des Unterprogrammrufs.

Die Stackbelegung soll im Folgenden anhand eines Beispiels dargestellt werden (Abb. 2.17).

Eine Funktion wird deklariert (Programmiersprache C):

```
int MAUSI (int A, int B, double C);

{
int X, Y;
double Z;
float H, I;
...

....

return (Y);

}
```

Jetzt wird im Anwendungsprogramm die Funktion aufgerufen:

```
OMEGA = MAUSI (ALPHA, BETA, GAMMA);
```

Zunächst werden die Parameter auf den Stack gelegt, dann wird die Funktion aufgerufen:

```
PUSH_DOUBLE GAMMA      -- alle Parameter auf den Stack
PUSH BETA
PUSH ALPHA
CALL MAUSI             -- das Unterprogramm wird aufgerufen
```

Eintritt in die Funktion. Es wird zunächst der Eintrittscode ausgeführt (ENTER-Ablauf):

```
MAUSI:   PUSH  FP          --  Frame Pointer auf Stack
         MOV   SP, FP      --  neuen FP einrichten
         DEC   SP, 24      -- die lokalen Variablen brauchen 24 Bytes

 Das war nur die Einleitung. Hier schließt sich der eigentliche Funktionskörper von MAUSI an.
```

Rückkehr (LEAVE-Ablauf):

```
MOV Y, (FP + 20)    -- Rückgabewert in Stack schreiben
MOV FP, SP          -- Zurückstellen des Stackpointers
POP FP              -- alten FP aus Stack zurückholen
RETURN              -- Rückkehr zum rufenden Programm
```

Weiter mit dem rufenden Programm:

```
POP_DOUBLE          -- Stack freimachen (Cleanup)
POP_DOUBLE
```

Der Rückgabewert der Funktion steht typischerweise in einem Register des Prozessors.

Programmoptimierung und Programmfehlersuche (Debugging)
Offensichtlich sind beim Aufruf einer Funktion etliche Maschinenbefehle auszuführen, die nichts mit der eigentlichen Funktion zu tun haben – es ist reiner Verwaltungsaufwand (Overhead).

Ist wirklich auf Leistung zu programmieren (z. B. bei Spielen), wendet man deshalb verschiedene Tricks an. Beispielsweise werden kurze Funktionen nicht als Unterprogramme gerufen, sondern als sog. Inline-Code immer wieder in den Programmtext eingefügt. Viele Compiler rufen Funktionen ohne Parameter und lokale Variable nach einem verkürzten Verfahren auf. Also arbeitet man mit globalen Variablen – richtige Profis können das eben ...

Zum Fehlersuchen (Debugging) müssen diese Optimierungen jedoch ausgeschaltet werden. Dann kommt es darauf an, immer einen Stack Frame zu erzeugen, damit man im Fehlerfall anhand des Speicherabzugs (Memory Dump) erkennen kann, was eigentlich los gewesen ist. Abb. 2.18 zeigt das Steuermenü eines ensprechenden Compilers.

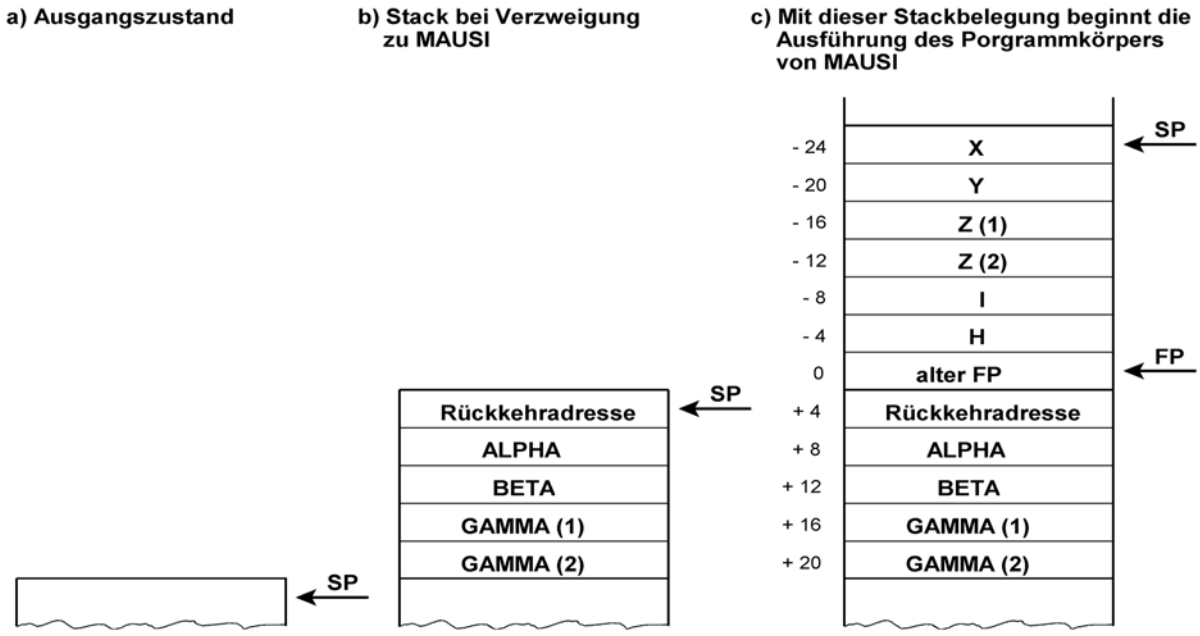


Abb. 2.23 Die Stackbelegung beim Unterprogrammruf anhand eines Beispiels.

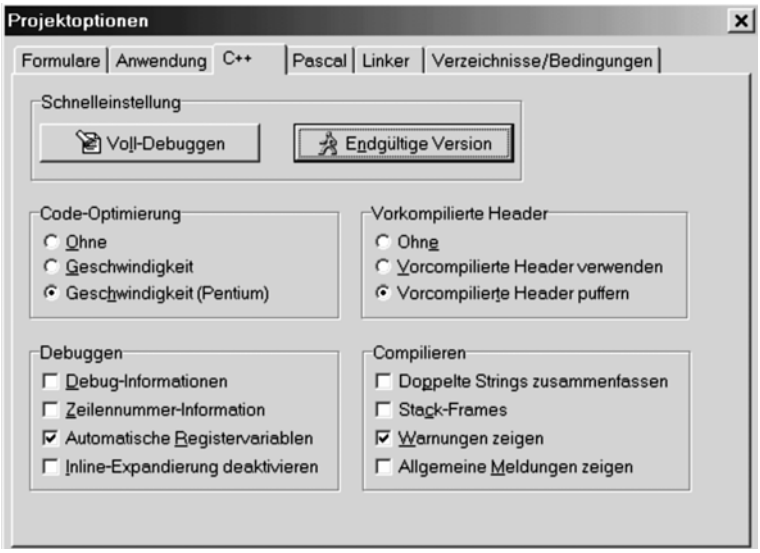


Abb. 2.24 Das Steuermenü eines Compilers. Die Betriebsweise kann zwischen Fehlersuchbetriebund optimierter Compilierung umgeschaltet werden.

2.2.4 Virtuelle Speicher

Virtuell = scheinbar. Der Software soll eine große – im Idealfall praktisch unbegrenzte – Speicherkapazität bereitgestellt werden, die Kosten sollen aber erträglich bleiben. Es ist offensichtlich, dass es total scheinbar nicht geht, sondern dass die zu unterstützende Speicherkapazität irgendwo vorhanden sein muss. Die Grundsatzlösung besteht darin, Arbeitsspeicher (teuer) und Massenspeicher – vor allem Festplatten – (kostengünstig) im

Verbund zu betreiben. Im Idealfall gibt es gar keine besondere Programmschnittstelle, sondern nur Maschinenbefehle für Speicherzugriffe (Laden, Speichern, Operationen mit Speicheroperanden) – der Anwendungsprogrammierer greift einfach auf die gesamte Speicherkakapzität zu, ohne sich besonders darum zu kümmern. Mit dem Prinzip des seitenorientierten virtuellen Speichers ist diese Ideallösung praktisch erreicht. Dieses Prinzip erfordert aber entsprechende Vorkehrungen in der Hardware (vor allem, um bei jedem Zugriff zu erkennen, ob eine Umlagerung notwendig ist oder nicht) - sonst wäre der Geschwindigkeitsverlust zu groß.

Zu Fuß umlagern

Wenn es in der Hardware nichts gibt, muss sich der Programmierer selbst darum kümmern. Der Arbeitsspeicher wird entsprechend aufgeteilt. Was unbedingt anwesend sein muss, wird fest eingelagert (residente Programme und Datenbereiche, Stack(s), Heap(s)). Alles Andere wird in Form von Dateien auf den Massenspeichern gehalten (transiente Programme und Datenbereiche). Um die Programme ausführen und die Daten bearbeiten zu können, werden im Arbeitsspeicher feste Bereiche, die sog. Transientbereiche, reserviert. Ist ein Programm auszuführen, so wird es in einen hinreichend großen Transientbereich geladen. Auch das erste der PC-Betriebssysteme – das DOS – arbeitet nach diesem Prinzip. DOS-Anwendungen sind im Grunde transiente Programme, die zwecks Ausführung in die sog. TPA (Transient Program Area) geladen werden. Bedarfsweise ladbare Bibliotheksprogramme und Programm-Module gibt es aber auch in höher enwickelten Systemen,wie Windows (DLLs) und Unix/Linux.

Ein Sonderproblem – die Verschieblichkeit

Nutzt man das Prinzip des Transientbereichs in überschaubaren kleinen Systemen, die ein feststehendes Sortiment von Anwendungen auszuführen haben, so kann man die Transientbereich an festen Adressen anordnen und alle transienten Programme für die jeweiligen Anfangsadressen übersetzen. In PCs und in gröeßeren Systemen werden aber immer wieder andere Anwendungen ausgeführt. Um den Speicher gut auszunutzen, soll es möglich sein, das jeweilige Programm dorthin zu laden, wo gerade Platz ist. Ein solches Programm muss in der Lage sein, mit beliebigen Adressen zurechtzukommen; das Programm muss – wie der Fachbegriff lautet – verschieblich[35)] sein. Manche Prozessoren unterstützen die Verschieblichkeit durch entsprechende Speicheradressierungsvorkehrungen. Bei der x86-Architektur ist diese Unterstützung aber auf kleine Programme (höchstens 64 kBytes) beschränkt. Um diese Einschränkungen zu umgehen, werden die nichtverschieblichen Adressbezüge beim Laden aufgelöst. Näheres in Abschnitt 3.***.

Prozesse umlagern

Diese Lösung wurde manchmal gewählt, um den Multitasking-Betrieb auf Maschinen zu verwirklichen, deren Arbeitsspeicherkapazität vergleichsweise gering ist und die keine

35: Engl. relocatable. Das Gegenteil: nicht verschieblich = non-relocatable.

Unterstützung für eine virtuelle Speicherorganisation aufweisen[36]. Im einfachsten Fall hält man neben dem System nur den einen Prozess im Arbeitsspeicher, der gerade Laufzeit hat und lagert alle anderen Prozesse auf die Festplatte aus[37]. Die Laufzeitvergabe ist also stets mit dem Auslagern der Speicherbelegung des einen und dem Hereinholen der Speicherbelegung des anderen Prozesses verbunden (Process Swapping). Das Prinzip ist einfach und durchaus wirkungsvoll (der laufende Prozess hat praktisch den gesamten Arbeitsspeicher für sich, und wenn er läuft, gibt es keinen Verwaltungsaufwand mehr). Die Umschaltzeiten sind aber nur bei geringen Speicherkapazitäten (einige zehn kBytes je Prozess) erträglich[38].

Der segmentbezogene virtuelle Speicher

Ein Segment ist ein zusammenhängender Speicherbereich jeweils festgelegter Größe. Segmente werden als Ganzheiten verwaltet. Dem Programmierer erscheint jedes Segment als ein eigener Speicher, der mit Adresse 0 beginnt (Segmentanfang) und eine jeweils bestimmte Größe hat (Segmentlänge).

Die Segmente werden typischerweise von Hand oder vom Laufzeitsystem auf Grundlage der Nutzungsweise eingerichtet. So liegt es nahe, Datensegmente, Stacksegmente und Programmsegmente vorzusehen. Jedes Segment muss für sich handhabbar sein, mit anderen Worten, es muss in den Arbeitsspeicher passen – und zwar zusammen mit den Programmen und Daten, die zum Betrieb des Systems benötigt werden.

Segmente werden mit Segmentdeskriptoren beschrieben, die in Segmenttabellen zusammengefasst sind (Abb. 2.19)[39]. Um auf ein Segment zuzugreifen, ist es über einenSegmentselektor anzusprechen. Ist das Segment im Arbeitsspeicher anwesend, enthält der Deskriptor einen Zeiger auf den betreffenden Speicherbereich (Anfangsadresse und Länge). Ist das Segment nicht anwesend, enthält der Deskriptor eine Positionsangabe für den Massenspeicher. Auf Grundlage dieser Angabe kann das Segment gefunden und in den Arbeitsspeicher transportiert werden. Der Segmentdeskriptor wird dann entsprechend umgestellt.

Ein Vorteil dieser Lösung ist der geringe Aufwand in der Hardware. Die Programmschnittstelle läuft im Grunde über eine einzige Bitposition im Segmentdeskriptor – das sog. Präsenzbit P (engl. Presence). Ist es gesetzt, so ist das Segment im Arbeitsspeicher anwesend, ist es gelöscht, muss das Segment vom Massenspeicher geholt werden. Das ist Sache der Systemsoftware. Die

---

36: Auch einige der ersten Unix-Systeme wurden so implementiert.

37: Manche Systeme halten – wenn genug Platz ist – auch mehrere Prozesse im Arbeitsspeicher. Wenn aber der gerade laufende Prozess mehr Platz benötigt, wird erforderlichenfalls ein anderer Prozess ausgelagert.

38: Gegenbeispiel: Bei 300 MBytes/s (SATA II) kostet das Umlagern der Anwenderspeicher zweier Windows-Prozesses (2 GBytes raus, 2 GBytes rein) etwa 13 s ...

39: Die Weiterentwicklung dieses Prinzips führt auf die sog. objektorientierte Speicherorganisation. Näheres in Abschnitt 2.3.7.

Hardware muss nicht mehr tun als eine Unterbrechung auszulösen, wenn versucht wird, auf ein Segment zuzugreifen, dessen Präsenzbit gelöscht ist[40].

Das Hauptproblem ist die mit dem Einlagern der Segmente verbunden Speicherverwaltung. Um ein Segment einlagern zu können, muss erst einmal ein genügend großer zusammenhängender Speicherbereich verfügbar sein (oder – durch Auslagern anderer Segmente – bedarfsweise freigemacht werden). Wegen der unterschiedlichen Größe der Segmente kommt es mit der Zeit zur Fragmentierung des Speichers. Diese Form der Speicherverwaltung funktioniert letzten Endes nur mit Garbage Collection (die viel Zeit kostet ...)[41].

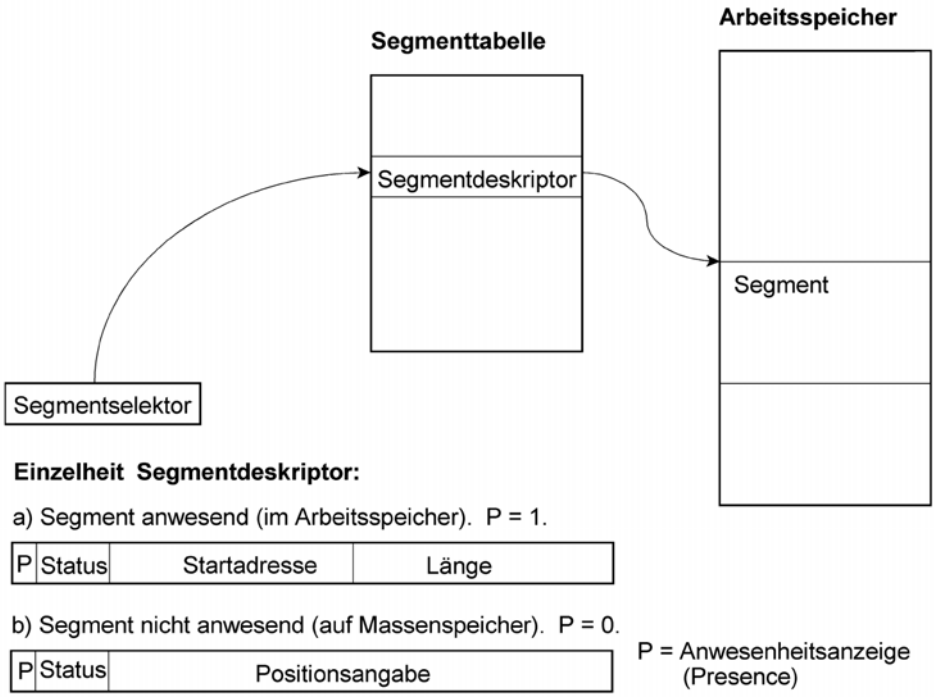


Abb. 2.25 Segmentorientierte Speicherorganisation als Grundlage eines virtuellen Speichers.

Der seitenorientierte virtuelle Speicher

Die Speicherverwaltung nimmt auf den Speicherinhalt – also auf Programme, Daten usw. – überhaupt keine Rücksicht, sondern verwaltet den Adressraum in Form gleichgroßer Bereiche, der sog. Seiten (Pages). Hierbei gibt es zwei Arten von Adressen:

- Die virtuelle Adresse. Sie adressiert den virtuellen Speicher aus Sicht des Programms.
- Die physische Adresse. Sie adressiert den Arbeitsspeicher.

40: Die segmentorientierte Speicherverwaltung ist Teil der Architektur IA-32. Die modernen Betriebssysteme nutzen diese Vorkehrungen aber nicht aus. Sie wurden deshalb in der 64-Bit-Weiterentwicklung (AMD64 / Intel® 64) weggelassen.

41: S. dazu den folgenden Abschnitt 2.3.5.

Adressumsetzung (Address Translation
Um zum Arbeitsspeicher zugreifen zu können, müssen die virtuellen Adressen in physische Adressen umgesetzt werden.

Seiten und Seitenrahmen (Pages, Page Frames)
Beides sind Behälter für Speicherinhalte. Der virtuelle Speicher ist in Seiten eingeteilt, der physische Arbeitsspeicher in Seitenrahmen. Jeder Seitenrahmen kann eine Seite aufnehmen. Seite und Seitenrahmen sind jeweils gleich groß und an integralen Adressen angeordnet.

Die grundsätzlichen Aufgaben der Speicherverwaltung:

- die Adressumsetzung (von der virtuellen zur physischen Adresse),
- das Ein- und Auslagern von Seiten (Page Swapping).

Die Byteadresse innerhalb der Seite oder des Seitenrahmens ist gleich und wird deshalb bei der Adressumsetzung nicht berücksichtigt. Das betrifft bei einer Seitengröße von n Bytes die ld n niedrigstwertigen Adressbits. Die eigentliche Umsetzungsaufgaben (Abb. 2.20): der verbleibende höchstwertige Teil der virtuellen Adresse (Seitenadresse; Virtual Page Number VPN) ist umzusetzen in den verbleibenden höchstwertigen Teil der physischen Adresse (Seitenrahmenadresse; Page Frame Number PFN oder Physical Page Number PPN). Diese Aufgabe wird mit Tabellenstrukturen gelöst.

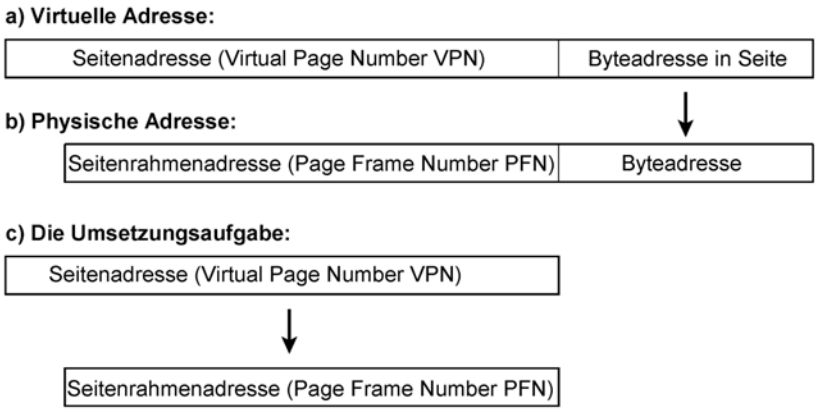


Abb. 2.26 Adressen und Adressumsetzung.

Was sein muss, muss sein ...
In letzter Konsequenz braucht jede Seite eine Umsetzungsangabe. Ist die virtuelle Adresse insgesamt V Bits lang und umfasst eine Seite P Bytes (Seitenlänge), so ergibt sich die Anzahl der Umsetzungsangaben zu $2^{V - ld P}$.

Beispiel: virtuelle Adresse 32 Bits, Seitenlänge 4 kBytes. Somit sind 1 M Umsetzungsangaben erforderlich. Die – dem Prinzip nach – einfachste Realisierung ist eine Umsetzungstabelle mit

1 M Einträgen zu 4 Bytes (20 Bits für die Seitenrahmenadresse PFN, Rest für Steuer- und Schutzzwecke), also insgesamt 4 MBytes. Das ist entschieden zu viel, um im Arbeitsspeicher gehalten zu werden (zumindest war es zu jener Zeit so, als solche Lösungen erfunden wurden). Der Ausweg: ein mehrstufiges Tabellenschema, in dem jede Stufe einen Teil der VPN umsetzt (Abb. 2.21).

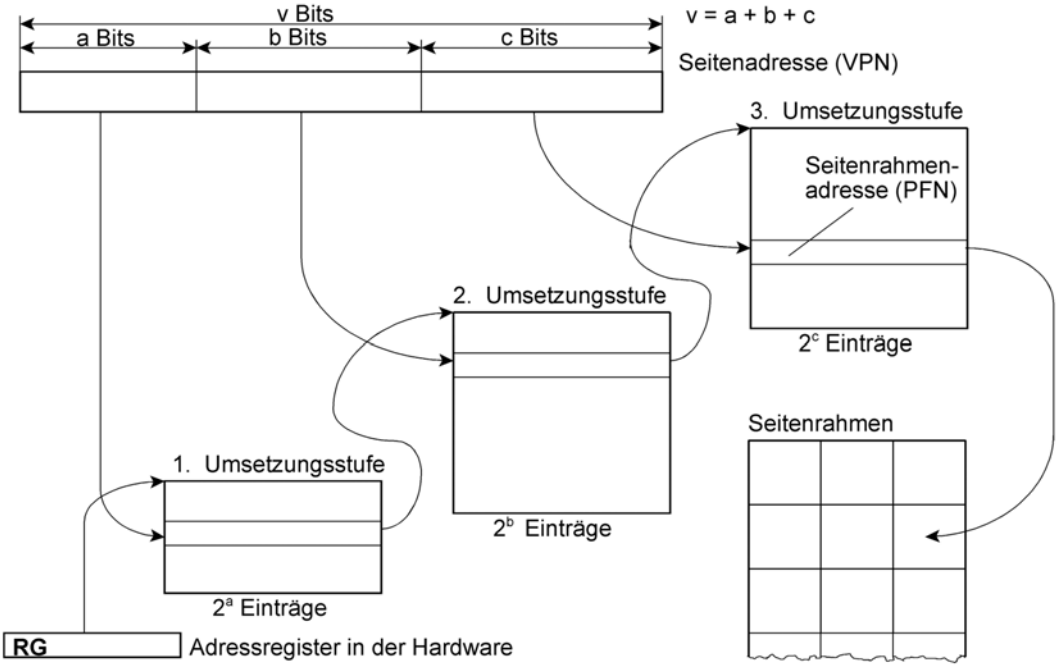


Abb. 2.27 Adressumsetzung über adressierbare Umsetzungstabellen. Hier am Beispiel einer dreistufigen Umsetzung.

Um n Bits der Seitenadresse (VPN) umzusetzen, braucht man Tabellen mit 2^n Einträgen. Die erste Stufe, die a Adressbits umsetzt, umfasst eine einzige Tabelle mit 2^a Einträgen, die zweite Stufe, die b Adressbits umsetzt, umfasst 2^a Tabellen mit jeweils 2^b Einträgen usw. Die letzte Stufe enthält in ihren Einträgen die jeweilige Seitenrahmenadresse (PFN). Es sind alle Bits der VPN umzusetzen. Stufenzahl und Tabellengrößen sind Erfahrungssache. Typischerweise richtet man es so ein, dass die einzelne Tabelle so groß ist wie eine Seite, so dass die Tabellen von der zweiten Umsetzungsstufe an ebenso wie die gewöhnlichen Seiten der Auslagerung unterworfen und auf einem Massenspeicher gehalten werden können. Die Tabelle der ersten Umsetzungsstufe muss stets im Arbeitsspeicher stehen. Diese Tabelle wird über ein Adressregister des Prozessors adressiert.

Aus Abb. 2.21 ist aber auch ersichtlich, dass jedem einzelnen Speicherzugriff mehrere (hier drei) Tabellenzugriffe vorangehen müssen. Vier Speicherzugriffe anstelle von einem – das wäre in der Praxis entschieden zu viel ...

Der seitenorientierte virtuelle Speicher muss also durch Hardware unterstützt werden. Die entsprechende Einrichtung heißt Translation Lookaside Buffer (TLB), Address Translation Cache (ATC) o. ä.

Was noch in den Tabelleneinträgen steht
Im Rahmen der Tabelleneinträge werden typischerweise ergänzende Angaben gespeichert, die folgende Funktionen haben:

- Speicherschutz,
- Unterstützung des Seitenaustauschs (Page Swapping),
- Unterstützung der Cache-Verwaltung.

Mehrere Adressräume
Ein virtueller Adressraum ist gegeben durch seine Adressumsetzungstabellen und durch den jeweils belegten Speicherplatz (im Arbeitsspeicher oder auf der Festplatte). Um mehrere Adressräume bereitzustellen, genügt es, ausreichend Speicherplatz vorzuhalten und die Seitenumsetzungstabellen entsprechend auszuwechseln. Der wesentliche Vorteil besteht darin, dass man vom Adressraum A aus den Adressaum B unmöglich erreichen kann (Schutzwirkung). Deshalb liegt es nahe, in einer Multitasking-Umgebung für jeden Prozess einen eigenen virtuellen Adressraum vorzusehen (Beispiel: Windows).

2.2.5 Fragmentierung

Ein Problem, das immer dann auftritt, wenn Speicherplatz belegt und wieder freigegeben wird. Es kann sowohl den Arbeitsspeicher als auch die Massenspeicher betreffen.Es bedeutet, dass die freie Speicherkapazität nicht fortlaufend (sozusagen am Stück) bereitsteht, sondern auf mehrere (bis sehr viele) kleine Bereiche verteilt ist.

Machen wir uns das an einem Beispiel klar. Wir haben einen Lagerplatz, auf dem jeweils gleich große Container abgestellt werden (Abb. 2.22). Anfangs ist der Platz leer. Nun werden ständig Container angeliefert. Es liegt nahe, sie einfach aneinandergereiht abzustellen. Es werden aber auch immer wieder Container abgeholt, allerdings in ganz anderer Reihenfolge und Anzahl. Demzufolge ergeben sich mehrere freie Flächen zwischen den belegten: der Lagerplatz wird fragmentiert. Kommt nun eine größere Fuhre an, so weiß man nicht mehr, wohin damit.

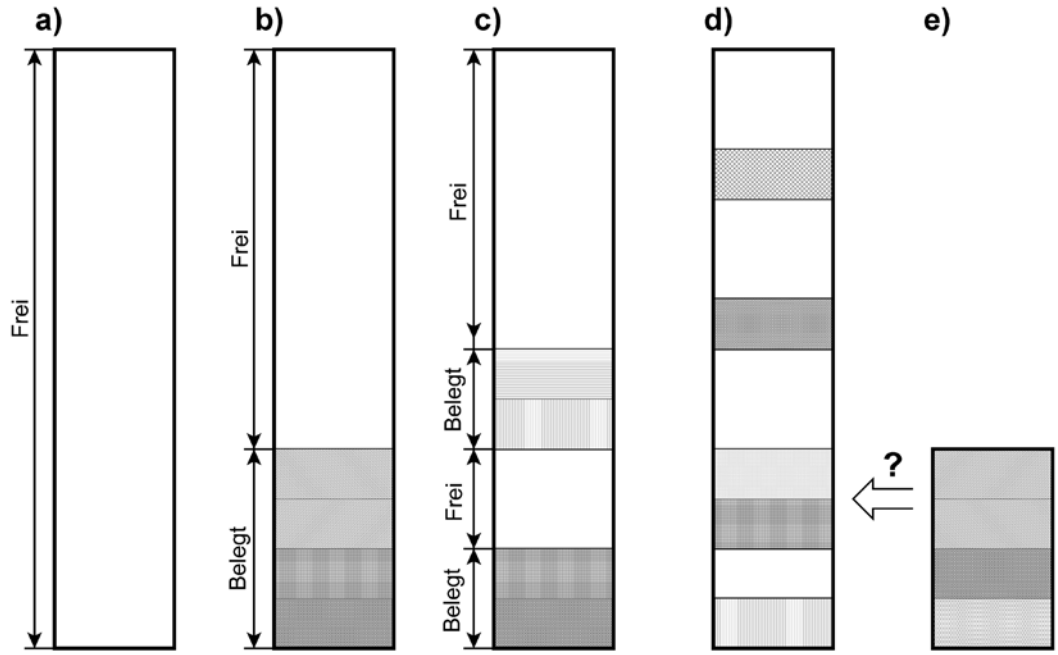


Abb. 2.28 Die Fragmentierung am Beispiel eines Lagerplatzes für Container.

Abb. 2.22 veranschaulicht typische folgende Zustände und Vorgänge:

a) Der Platz ist leer.
b) Der Platz wird nach und nach mit den ersten Containern gefüllt.
c) Es sind weitere Container hinzugekommen, aber auch einige wieder abgeholt worden. Beginn der Fragmentierung. Noch ist aber eine große zusammenhängende Freifläche vorhanden.
d) Einige Zeit später. Es sind weitere Container geliefert, aber auch welche abgeholt worden. Daraus hat sich eine Vielzahl kleiner Freiflächen ergeben (starke Fragmentierung).
e) Jetzt kommt ein größerer Posten. An sich ist ausreichend Platz, aber keine der Freiflächen ist groß genug, um alle gelieferten Container aufzunehmen.

Der Trivialfall, dass der verfügbare Platz insgesamt nicht ausreicht, steht hier nicht zur Diskussion. Viemehr ist an sich genügend Stellfläche vorhanden, nur ist sie auf viele kleine verstreut liegende Flächen aufgeteilt. Anhand von Abb. 2.22 können wir uns auch die grundsätzlichen Auswege klarmachen:

- Verstreute Speicherung: wir reißen die neue Lieferung auseinander und stellen die Container einzeln dahin, wo gerade Platz ist.
- Defragmentierung: wir schieben die bereits abgestellten Container zusammen, so dass sich wieder eine einzige große Freifläche ergibt.

Es ist ohne weiteres einzusehen, dass wir unser Gleichnis mühelos auf Datenstrukturen und Speicherbereiche übertragen können. Auch die Problemstellen sind leicht zu erkennen:

- Die verstreute Speicherung kostet Verwaltungsaufwand (wir brauchen z. B. einen Computer, der sagt, welcher Container wo steht). Zudem kostet sie Zeit. Um eine Ladung aus mehreren Containern loszuwerden oder – zwecks Versand – zusammenzustellen, müssen wir nahezu den gesamten Lagerplatz abfahren.
- Die Defragmentierung kostet sogar richtig Zeit. Man könnte aber hierfür z. B. eine Nachtschicht einrichten, die dafür sorgt, dass am nächsten Tag wieder ein aufgeräumter Lagerplatz vorhanden ist.

Fragmentierung auf Datenträgern
Die verstreute Speicherung wird von der Datenträgerverwaltung unterstützt. Je stärker die Fragmentierung, desto größer aber die Zugriffszeiten (in voller Analogie zu unserem Lagerplatz-Beispiel – dem Hin- und Herfahren des Staplers entspricht hier das mechanische Positionieren). Deshalb ist von Zeit zu Zeit eine Defragmentierung sinnvoll (mittels Dienstsoftware). Bei den heutigen Festplattenkapazitäten ist das eine zeitaufwendige Angelegenheit. Aber auch hier gilt die Analogie zum Lagerplatz: man kann es nach Feierabend oder über Nacht erledigen lassen, in neueren Systemen auch im Hintergrund während der laufenden Arbeit.

Jedes Dateisystem ist von Defragmentierung betroffen
Allerdings manche mehr und manche weniger. FAT-Dateisysteme sind in dieser Hinsicht viel anfälliger als Systeme, die den freien Speicherplatz über Bitketten (Bitmaps) verwalten. Es ist aber ein Irrglaube, dass es z. B. bei NTFS oder einem Unix/Linux-Dateisystem nie zur Defragmentierung kommt. Der Zeitpunkt wird nur hinausgeschoben. Dies gelingt u. a. deshalb, weil es die Bitkette ermöglicht, ohne viel Zeitaufwand freie Bereiche geeigneter Größe aufzufinden (eine Bitkette mit einem Bit je Zuordnungseinheit ist viel schneller durchmustert als eine FAT mit 16 oder 32 Bits je Zuordnungseinheit – deshalb leistet man sich beim FAT-Dateisystem solche Optimierungen nicht).

Fragmentierung im Arbeitsspeicher
Vor allem Heap-Strukturen (vgl. Abschnitt 2.3.1.) sind davon betroffen. Das Gegenmittel – nämlich das Zusammenschieben – heißt hier aber nicht Defragmentierung, sondern Garbage Collection[42]. Die Fragmentierung ist vor allem von Bedeutung bei Programmen, die sehr große zeitweilige Datenstrukturen im Arbeitsspeicher halten, weniger also bei “gewöhnlichen” Windows-Anwendungen. Die Garbage Collection ist typischerweise Angelegenheit des Laufzeitsystems und – von der Theorie her – eine Wissenschaft für sich.

42: Wörtlich: Müllsammlung.

Die Programmiersprache Java schließt u. a. eine im Hintergrund stets wirksame Garbage Collection ein (das ist an sich schön, kostet aber Laufzeit). In C- und C++-Programmen ist hingegen der Programmierer selbst für das Anfordern und Freigeben von Speicherplatz auf dem Heap verantwortlich.

Belegungsverfahren bei verstreuter Speicherung
Des Zeitaufwandes wegen ist man bestrebt, das Defragmentieren soweit wie möglich hinauszuschieben. Man versucht deshalb, Dateien zusammenhängend zu speichern. Bei der Zuweisung von Bereichen im Arbeitsspeicher kommt es darauf an, für künftige, nachfolgende Zuweisungen noch genug Platz zu lassen. Man kennt verschiedene Belegungsverfahren. Wir wollen nur die drei gebräuchlichsten kurz vorstellen:

First Fit
Der erste freie Bereich, in den unsere Struktur hineinpasst, wird auch belegt. Der Vorteil: es geht schnell. Der Nachteil: noch mehr Fragmentierung. Die freien Bereiche werden immer kleiner. Beispiel: wir brauchen auf einer Festplatte Platz für 3 Cluster. Das Dateisystem trifft auf einen freien Bereich aus 8 Clustern. Werden die 3 Cluster hier untergebracht, so entsteht ein freier Rest-Bereich von 5 Clustern.

Best Fit
Es wird so lange gesucht, bis ein Bereich gefunden ist, der den geringsten Verschnitt übrigläßt. Beispiel: wir brauchen Platz für 3 Cluster und finden Bereiche zu 4, 7 und 16 Clustern vor. Dann werden wir einen Bereich von 4 Clustern belegen. Der Vorteil: es bleiben recht wahrscheinlich Bereiche übrig, die auch künftig zu gebrauchen sein werden. Der Nachteil: Zeitaufwand. Zudem können sehr kleine Lücken übrigbleiben, die wirklich nicht mehr verwertbar sind.

Last Fit
Es wird nach dem jeweils größten Bereich gesucht. Der Grundgedanke: wird dieser teilweise belegt, so bleibt wahrscheinlich ein – im Gegensatz zu Best Fit – immer noch brauchbarer Bereich übrig. (Im obigen Beispiel: wenn wir von 16 Clustern 3 belegen, bleiben noch 13 frei.)

2.2.6 Prozesse und Threads

Moderne Systeme nutzen das Prinzip des seitenorientierten virtuellen Speichers, um jedem Anwenderprozess einen ausreichend großen Speicheradressraum zur Verfügung zu stellen. In 32-Bit-Systemen sind dies typischerweise zwei oder drei GBytes (Abb. 2.23). Obwohl die heutigen Computer vergleichsweise viel Arbeitsspeicherkapazität haben und der Auslagerungsspeicher auf der Festplatte (fast) nichts kostet, ist man bestrebt, die Speicherkapazität gut auszunutzen. Heute geht es in erster Linie nicht mehr darum, wie viele Bits Speicherkapazität man sich leisten kann, sondern darum, dass man die Verwaltungsaufwendungen und Transportzeiten gering hält. Auch hierzu werden die Vorkehrungen des seitenorientierten virtuellen Speichers ausgenutzt.

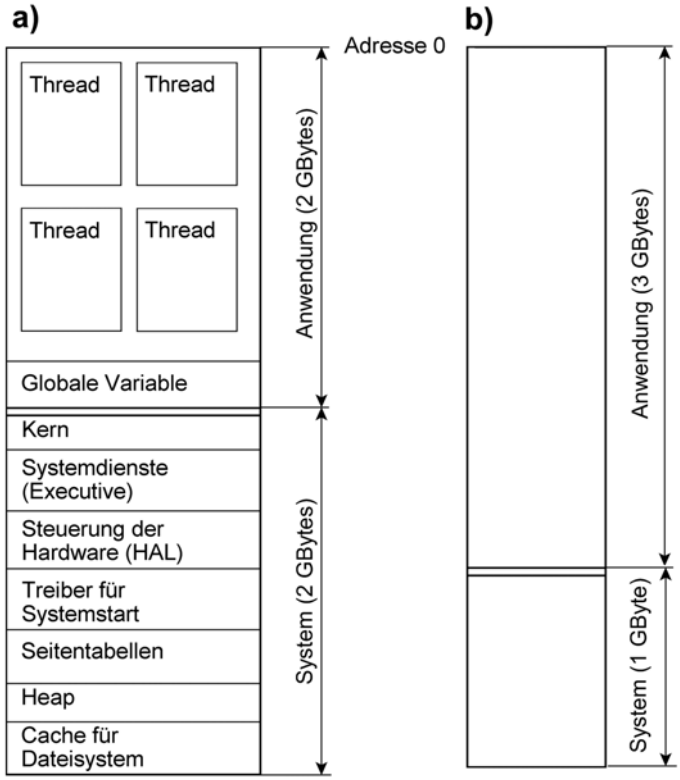


Abb. 2.29 Typische Speicherbelegungen von 32-Bit-Systemplattformen. a) Windows-Arbeitsplatzsysteme; b) Linux, Windows-Serversysteme (wahlweise als Alternative zu a))[43]. Der Anwenderbereich nimmt die Speicherbereiche der einzelnen Threads auf, also den Programmcode, die Stacks, ggf. erforderliche DLLs usw. (vgl. Abb. 2.5).

Vier GBytes insgesamt und zwei GBytes für die Anwendung bedeuten zunächst einmal nur, dass für jeden Prozess ein vollständiger Satz an Seitentabellen verwaltet wird. Was den Speicherplatz selbst betrifft, so wird nur so viel belegt, wie wirklich gebraucht wird. Diese Verfahrensweise erlaubt es, mit der Zuweisung von Speicherplatz großzügig zu sein. So werden für Stacks und Heaps typischerweise jeweils ein Megabyte vorgesehen[44].

Das Betriebssystem selbst wird natürlich nicht für jeden Prozess einzeln gespeichert. Vielmehr werden die Seitentabellen so eingerichtet, dass die gemeinsamen Systemfunktionen und die gemeinsam genutzten Speicherbereiche von allen Prozessen aus unter gleichen physischen Adressen erreichbar sind. Abb. 2.24 veranschaulicht, wie man mit den Mitteln der Seitenverwaltung gemeinsame Speicherbereiche für mehrere Prozesse einrichten kann.

43: Für das System reicht das eine GByte zumeist bei Weitem aus.

44: Sofern in den jeweiligen Systemaufrufen nichts Anderes angegeben ist.

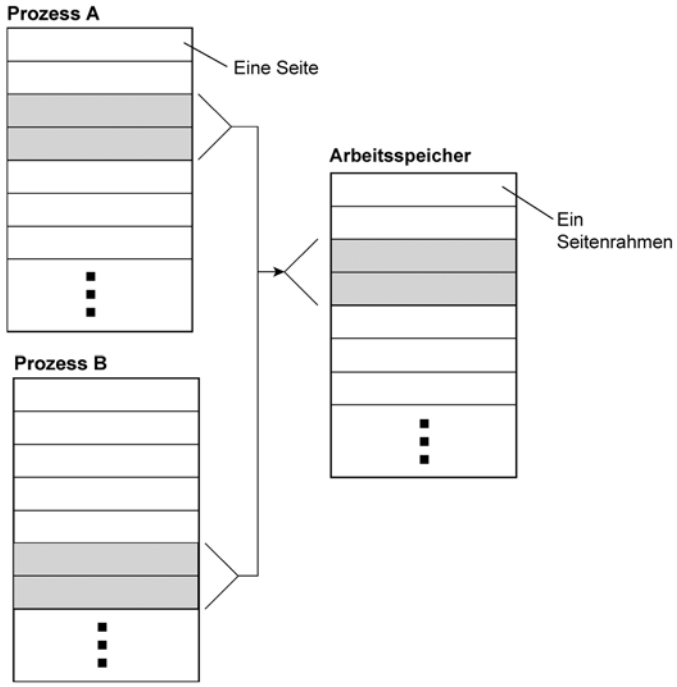


Abb. 2.30 Gemeinsame Nutzung von Speicherbereichen (Shared Memory). Die Seitentabellen der beiden Prozesse werden so eingerichtet, dass verschiedene virtuelle Adressen auf einen einzigen Bereich im Arbeitsspeicher verweisen.

Die Speicherbelegung kann man sich mit besseren Systeminformationsprogrammen ansehen (Abb. 2.25).

Wie das Anwendungsprogramm auf Speicherbereiche zugreift

Der Speicher kann nur dann gut ausgenutzt werden, wenn die Speicherverwaltung die von den Programmen angeforderten Speicherbereiche dort anordnen kann, wo es jeweils zweckmäßig ist. Da sich die "Auftragslage" im PC ständig ändert, wäre es keine gute Lösung, den Bereichen feste Adressen zuzuweisen. Ein Programm, das auf einen Speicherbereich zugreifen will, muss deshalb zunächst anfragen, unter welchen Adressen dieser Bereich derzeit zugänglich ist. Hierzu gibt das System auf die Anforderung eines Speicherbereichs nicht einen einfachen Adresszeiger (Pointer) zurück (der z. B. auf die Anfangsadresse des Bereichs verweist), sondern einen Zeiger auf einen Speicherblock, der seinerseits den eigentlichen Adresszeiger enthält. Eine solche Zeigerangabe heißt Handle[45)]. In Systemen wie Windows sind Speicherbereiche nur über Handles zugänglich.

45: Wörtlich = Handgriff.

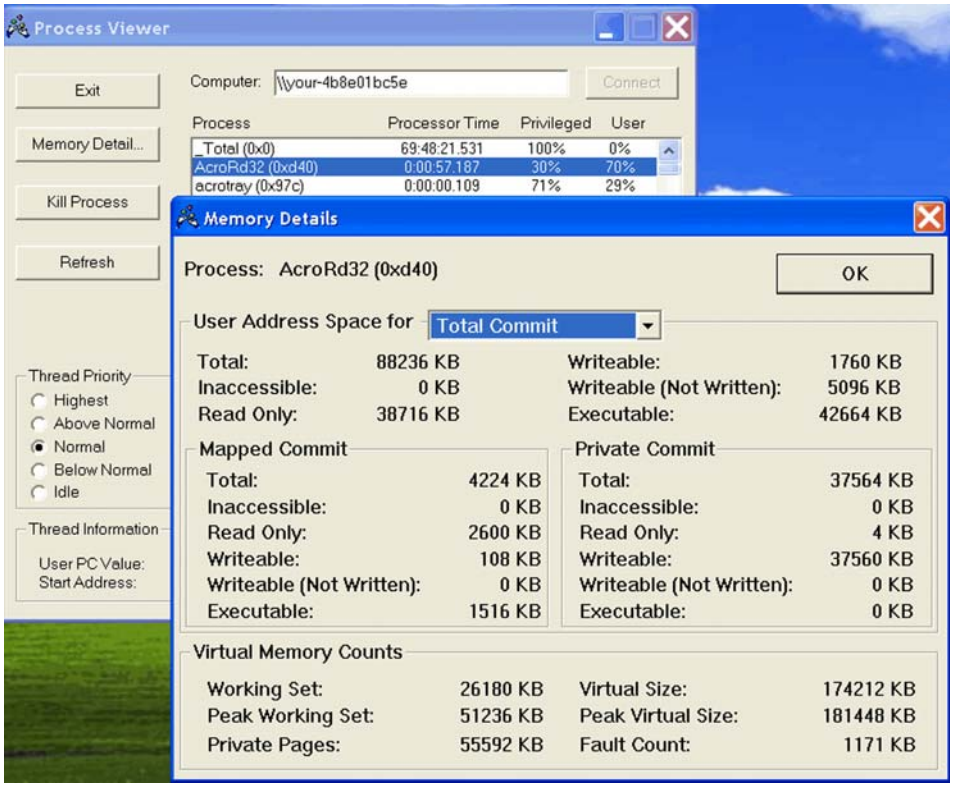


Abb. 2.31 Wie viel Speicher belegt eine Anwendung? – Der Process Viewer von Microsoft zeigt es. Im virtuellen Speicher belegen die Seiten der Anwendung (Private Pages) 55 592 kBytes. Das sind 13 898 Seiten von 4 kBytes Länge. Für den Programmcode (Executable) werden 42 666 kBytes benötigt. Das eigentliche Anwendungsprogramm hat zwar nur 334 kBytes, es gibt aber jede Menge an DLLs[46] ...

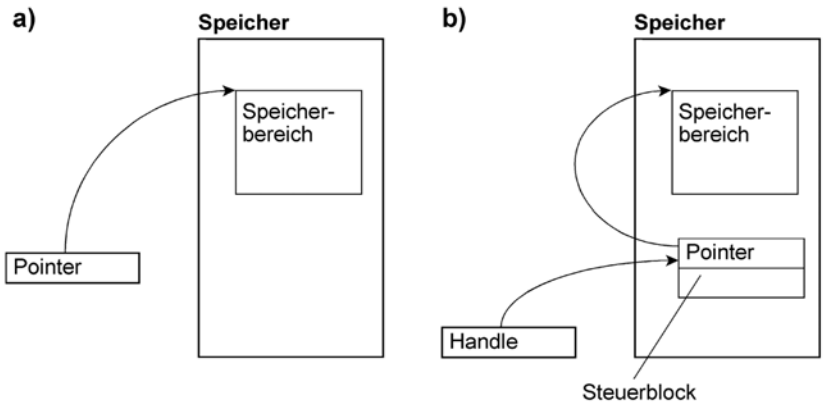


Abb. 2.32 Pointer und Handle. a) der Pointer ist eine Adressangabe, die auf den Anfang des Speicherbereichs verweist. b) ein Handle adressiert einen vom System verwalteten Steuerblock, der seinerseits einen Pointer enthält.

46: Wie ein Blick ins Verzeichns der Anwendung zeigt.

2.2.7 Objektorientierte Speicherorganisation

Alle Programme und Datenbereiche werden als Objekte aufgefaßt. Objekte sind Behälter für Information, die jeweils als Ganzheit behandelt werden. Jedes Objekt hat eine Namen bzw. – zur Laufzeit – eine binär codierte Ordinalzahl[47], die das jeweilige Objekt aus der Menge aller Objekte auswählt. Die Zugriffsangaben in den Maschinenbefehlen sind keine Adressen, sondern Objektnummern (Object Identifers). Die Objekte werden durch Objektdeskriptoren beschrieben, die in Objekttabellen zusammengefaßt sind. Die Objektverwaltung sorgt automatisch dafür, daß die jeweiligen Objekte in den Speicher geschafft oder ausgelagert werden.

Objektorientierung bedeute im Grunde, alle Adressbezüge erst zur Laufzeit aufzulösen. Ein Objekt kann beim ersten Aufruf über Adresse A erreichbar sein, beim zweiten Aufruf über Adresse B usw. – das Anwendungsprogramm merkt davon gar nichts; es kennt nur seine Objektnummern.

Die durchgehende Objektorientierung[48] erfordert ein zweistufiges Zugriffsschema (Capability Based Addressing; Abb. 2.27). Die Variablen im Programm sind keine Objektnummern, sondern sie bezeichnen Eintrittspunkte in eine zur Laufzeit aktuelle Zugriffstabelle (Capability Table, Access Reference Table), die ihrerseits die Objektnummern enthält, mit denen wiederum die Objekttabelle aufgesucht wird. Die Zugriffstabelle entspricht praktisch einem Stack Frame, der anstelle der Werte der lokalen Variablen deren Objektnummern enthält. Auf diese Weise wird die Menge der Objekte von den Programmen vollständig entkoppelt (mit anderen Worten: voreinander geschützt), und es ist möglich, Schutzrechte bis auf die einzene Variable genau zuzuweisen (wer sie lesen darf, wer sie ändern darf usw.).

Die Objektdeskriptoren von Abb. 2.27 ähneln den Segmentdeskriptoren von Abb. 2.19 Tatsächlich haben beide Auslegungen viele Gemeinsameiten. Die segmentorientierte Speicherverwaltung befasst sich nur mit ganzen Speicherbereichen, die objektorientierte Speicherverwaltung betrifft hingegen alle Datenstrukturen bis zur einzelnen Variablen. Aus dieser Tatsache ergibt sich aber auch der wesentliche Nachteil – ein System, das vor jedem einzelnen Datenzugriff noch zwei Tabellenzugriffe ausführen muss, kann offensichtlich nicht allzu viel leisten. Um diesen Nachteil abzustellen, wäre eine massive Unterstützung seitens der Hardware erforderlich. Deshalb hat man beispielsweise in Windows dieses Prinzip nicht implementiert[49], sondern einen Kompromiss gefunden: die Speicherbereiche werden über Handles angesprochen (vgl. Abb. 2.26), die einzelnen Variablen über Adressen.

47: Mit anderen Worten, eine laufende Nummer.

48: Das heißt eine Auslegung, bei der jede einzelne Variable als Objekt behandelt wird.

49: Obwohl es seit Jahrzehnten bekannt ist. Alle bisherigen Implementierungen waren aber von unannehmbaren Leistungsschwächen betroffen.

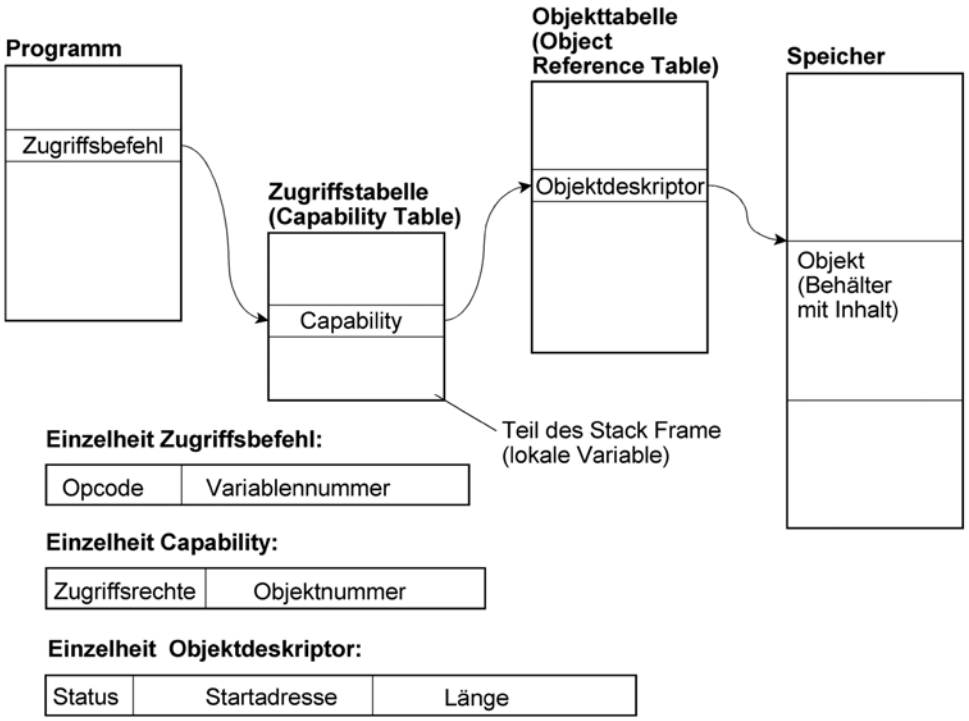


Abb. 2.33 Zweistufiger objektorienterter Zugriff (Capability Based Addressing).

2.3 Reale und virtuelle Maschinen

Eine reale Maschine ist gegenständlich vorhanden – es ist letzten Endes der Prozessor, dessen Hardware tatsächlich Maschinenbefehle ausführt. Der Grundgedanke der virtuelle Maschine besteht darin, einen einzigen Prozessor so zu betreiben, als ob mehrere unabhängige Prozessoren zur Verfügung stünden.

Der wesentliche Unterschied zum Multitasking (Tabelle 2.4): die Task muss sich an die Bedingungen des Systems halten, auf einem unabhängigen Prozessor hingegen kann man ein beliebiges System ausführen. Virtuelle Maschinen ermöglichen also nicht nur die Parallelausführung mehrere Anwendungen, sondern auch die gleichzeitige Nutzung mehrerer Betriebssysteme.

2.3.1 Virtuelle Maschinen durch hardwareseitiges Umschalten

Jeder Prozessor besteht im Grunde aus Speichermitteln (vor allem Registern) und Verknüpfungsschaltungen. Sieht man mehrere Sätze von Speichermitteln vor und schaltet zwischen ihnen schnell um, so ergeben sich entsprechend viele gleichzeitig verfügbare Prozessorfunktionen (das Ausführungsbeispiel im PC-Bereich: Intels HyperThreading-Technologie). Die Triviallösung: jede Prozessorfunktion ist eine eigenständige Maschine mit eigenem Betriebssystem. Der grundsätzliche Nachteil dieser Lösung ist aber offensichtlich – der Aufwand wäre viel zu hoch. Damit die Grundsatzlösung auch in der Praxis so einfach bleibt wie

es sich in der Theorie anhört, müsste jede Maschine einen eigenen Arbeitsspeicher und eine eigene E-A-Ausstattung haben. Das liefe dann auf mehrere unabhängige (zwecks Datenaustausch ggf. vernetzte) Einzel-PCs hinaus, die lediglich in einem gemeinsamen Gehäuse untergebracht sind. Einfachlösungen hätten nicht den Komfort der typischen VM-Systeme (wo man beispielsweise Windows- und Linux-Fenster auf dem gleichen Bildschirm haben kann), entsprechend komfortable Lösungen (die man natürlich bauen könnte) wären viel zu teuer. Hinzu kommen Platz-, Strom- und Kühlungsbedarf.

Multitasking	**Virtuelle Maschinen**
• Mehrere Anwendungen können zeitmultiplex ausgeführt werden. • Jede Anwendung hat fast die gesamte Maschine zur Verfügung – aber eben nicht alles (nur bestimmte Speicherbereiche, E-A-Adressen, Register usw.). • Die Systemplattform organisiert das Umschalten zwischen den Tasks. • Die Tasks müssen sich an die Konventionen der Plattform halten. • Nur ein Betriebssystem für alle Tasks. • Umschaltung zwischen den Tasks vergleichsweise schnell. *Eine "feinfühlige", gezielt dosierbare Ressourcenverwaltung, die aber den einzelnen Anwendungen Beschränkungen auferlegt.*	• Mehrere Anwendungen können zeitmultiplex ausgeführt werden. • Jede Anwendung hat (scheinbar(virtuell)) die gesamte Maschine zur Verfügung. • Die Plattform organisiert das Umschalten zwischen den virtuellen Maschinen. • In jeder virtuellen Maschine kann ein eigenes Betriebssystem laufen. • Umschaltung zwischen virtuellen Maschinen ggf. zeitaufwendig (hier kommt es auf die Einzelheiten der Auslegung an). *Eine pauschale Ressourcenverwaltung, die sich nicht mit Kleinigkeiten abgibt – dafür aber ihre Zeit braucht.*

Tabelle 2.4 Tasks und virtuelle Maschinen im Vergleich.

2.3.2 Virtuelle Maschinen als Gäste

Eine einleuchtende Idee, um das Problem zu lösen, auf ein und demselben Prozessor mehrere Programme laufen zu lassen, ja sogar mehrere Software-Plattformen. Wir stellen – zunächst gedanklich – jedem Programm den gesamten Prozessor zur Verfügung: (fast) alle Maschinenbefehle, (fast) alle Register usw. Jedes Programm sieht einen Arbeitsspeicher, der ganz vorn anfängt (mit Adresse 0) und ganz hinten aufhört (idealerweise am Ende des Speicheradressraums). Jedes Programm hat so einen eigenen *virtuellen* (scheinbaren) Prozessor ganz für sich allein. Damit das funktioniert, brauchen wir ein übergeordnetes Steuerprogramm. Dessen grundsätzliche Wirkungsweise ist nicht einmal sonderlich kompliziert. Nehmen wir an, es seien drei virtuelle Maschinen A, B, C zu implementieren. Dann richten wir drei Dateien DA, DB, DC ein, die jeweils den gesamten Arbeitsspeicherinhalt sowie alle erforderlichen Zustandsangaben des Prozessors aufnehmen können. Wir starten zunächst die virtuelle Machine A, indem wir ihr Speicherabbild aus der Datei DA in den Arbeitsspeicher laden und den Prozessorzustand – wie bei einer Taskumschaltung – in die Prozessor bringen. Nach einer gewissen Zeit halten wir den Verarbeitungsablauf an und schaffen den Arbeitsspeicherinhalt

sowie den aktuellen Prozessorzustand wieder in die Datei DA. Anschließend laden wir die Datei DB in den Arbeitsspeicher, um die virtuelle Maschine B zu starten usw.

In der Praxis kann man diesen einfachen Ablauf aber nicht 1:1 verwirklichen, denn dann würde die Systemleistung zu sehr abfallen[50]. Es geht also darum, das pauschale Umlagern zu vermeiden. Diese Aufgabe kann auf Grundlage folgender Überlegungen gelöst werden:

- Wenn der virtuelle Speicher einer Anwendung einmal eingerichten ist (Seitentabellenstruktur + Speicherinhalt) und die Anwendung nur für sich läuft (wenn sich also die ausgeführten Befehle nur mit dem eigenen Speicherinhalt beschäftigen), passiert im Grunde gar nichts.
- Sinngemäß passiert nichts, wenn die Anwendung ihr eigenes Betriebssystem ruft und auch dieses nur auf zugewiesene Speicherinhalte einwirkt.
- Bei Ein- und Ausgabevorgängen und beim Vergeben von Laufzeit muss eingegriffen werden.
- Jede virtuelle Maschine hat ihre eigene Peripherie (Tastatur, Maus, Bildschirm(e), Laufwerk(e) usw.). Das sind aber auch nur scheinbare (virtuelle) Einrichtungen. Entsprechende Zugriffe müssen auf die tatsächlich angeschlossenen (realen, physischen) Geräte umgeleitet werden.
- Jedes beliebige Programm sieht von einem Gerät eigentlich nur dessen Registersatz oder eine entsprechende Speicherausstattung (z. B. den Bildspeicher). Wenn man diese Zugriffe so umlenkt, dass sie nicht mehr die Hardware betreffen, sondern ein weiteres Programm[51], ist es möglich, die Funktionen des realen Gerätes nachzubilden oder die Zugriffe auf ein physisches Gerät umzuleiten.

Das Umschalten zwischen den virtuellen Maschinen übernimmt eine weitere Schicht von Systemprogrammen, die in ihrer Gesamtheit als VM-Monitor oder Hypervisor bezeichnet wird (Abb. 2.28). Die Prinzipien sind seit Jahrzehnten bekannt[52]. Zwischenzeitlich sind sie auch im PC-Bereich von Bedeutung[53]. Moderne Prozessoren haben zusätzliche Befehle, die die Implementierung entsprechender Systemprogramme unterstützen.

50: In heutigen PCs wären Gigabytes umzulagern. Und das dauert auch bei den schnellsten Interfaces entschieden zu lange.

51: Fachbegriff: die Zugriffe werden abgefangen (engl. trapped, intercepted).

52: Das erste System, das am Markt erfolgreich war: VM/370 von IBM (70er Jahre).

53: Zum Einrichten und Nutzen virtueller Maschinen gibt es eine geradezu unüberschaubare Literatur. Zumeist sind Datenträger beigegeben, die es beispielsweise ermöglichen, Windows und Linux parallel in zwei virtuellen Maschinen laufen zu lassen.

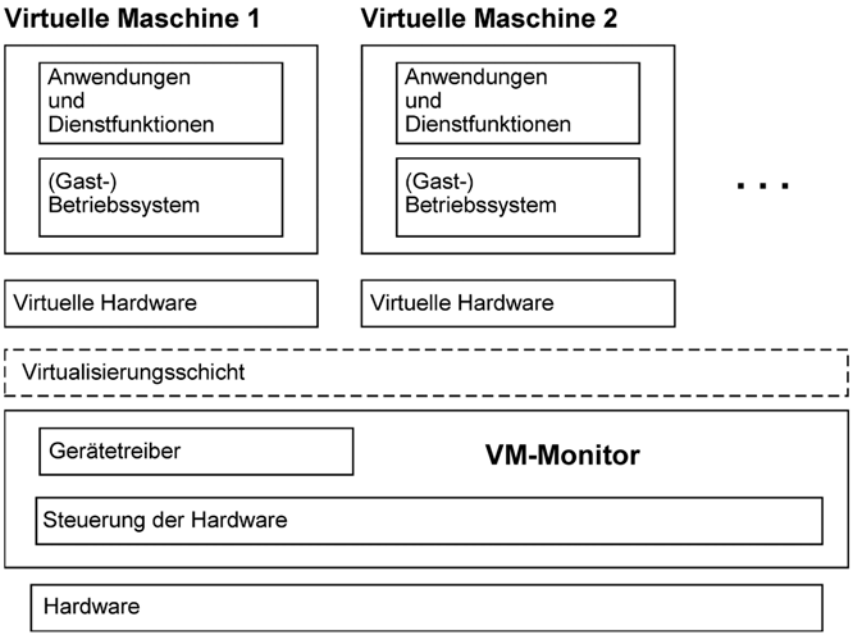


Abb. 2.34 Die Ausführung virtueller Maschinen auf einer realen Maschine.

Der Wirt und die Gäste

Der Programmkomplex, der den Betrieb der virtuellen Maschinen gewährleistet und der tatsächlich mit der Hardware zusammenwirkt (der VM-Monitor oder Hypervisor), wird auch als Wirts- oder Hostsystem bezeichnet. Jede virtuelle Maschine hat ihr eigenes Betriebssystem, das als Gastsystem bzeichnet wird. Billigausführungen von VM-Systemen verwenden ein vorhandenes Betriebssystem als Hostsystem, beispielsweise Windows oder Linux. Bessere Ausführungen haben einen eigenen optimierten Systemkern.

Eine einmal eingerichtete Anwendung in einer virtuellen Maschine läuft, ohne dass sich irgendwer darum kümmern muss. Das gilt sinngemäß für Systemfunktionen, die sich nur auf Speicherinhalte beziehen und nicht mit irgend welchen E-A-Vorgängen verbunden sind. Jede virtuelle Maschine kann im Grunde ein x-beliebiges Betriebssystem haben. Das VM-System muss also keine Systemaufrufe abfangen oder nachbilden (denn die sehen bei jedem System anders aus), sondern die Schnittstellen zwischen Hard- und Software, also die Architektur des Prozessors und der jeweiligen E-A-Geräte. Das betrifft programmseitige Zugriffe auf die Seitentabellen, auf Bildspeicher und Geräteregister sowie das Reagieren auf Unterbrechungen. Das Abfangen sieht im Idealfall so aus, dass, wenn eine virtuelle Maschine Laufzeit hat und versucht, einen der abzufangenden Befehle auszuführen, eine Unterbrechung ausgelöst wird. Hierdurch wird der VM-Monitor aktiviert. Er analysiert den betreffenden Befehl und bildet dessen Wirkung programmseitig nach (Emulation). Zum Abfangen mancher Befehle können die Vorkehrungen der herkömmlichen Prozessorarchitektur ausgenutzt werden. Die erweiterten Vorkehrungen der modernen Prozessoren unterstützen alle für den Betrieb virtueller Maschinen erforderlichen Abfangmaßnahmen.

Als Beispiel sei kurz beschrieben, wie ein Festplattenzugriff in einer virtuellen Maschine abläuft: Die Software sieht die Festplattensteuerung im Grunde als einen Registersatz, der mit E-A-Zugriffen angesprochen werden kann. Solche Zugriffe lassen sich über die Erlaubnisbitliste[54)] des Prozessors abfangen. Die zugehörige Unterbrechung wird von einem Programm der virtuellen Hardware (vgl. Abb. 2.28) behandelt. Dieses bildet die Festplattensteuerung programmseitig nach. Eine virtuelle Festplatte ist im Grunde eine große Datei auf einem realen Laufwerk. Demgemäß wird der VM-Monitor veranlasst, die entsprechenden wirklichen Zugriffe auszuführen. Ist der VM-Monitor ein handelsübliches Betriebssystem, muss er dazu über dessen Anwenderprogrammschnittstellen (APIs) aufgerufen werden. Hierzu dient die in Abb. 2.28 dargestellte Virtualisierungsschicht.

Weshalb sind virtuelle Maschinen so in Mode?

Im professionellen Einsatz ist es vor allem die bessere Ausnutzung der Hardware und die Vereinfachung der Administration einschließlich der Systemwiederherstellung. Man kann mit mehreren Systemplattformen gleichzeitig arbeiten und deren jeweilige Stärken gezielt nutzen. Ein weiterer grundsätzlicher Vorteil liegt darin, dass die Gastsysteme (nahezu) 100% gegeneinander isoliert (und somit auch voreinander geschützt) sind. Ein VM-Monitor ist zwar nicht gerade trivial, er hat aber – verglichen mit einem "ausgewachsenen" Betriebssystem – nur eine einzige im Grunde bescheidene und überschaubare Aufgabe: bilde die Hardware-Software-Schnittstellen des Prozessors und der E-A-Steuerung nach, und zwar so, dass es bei der Nutzung der realenRessourcen nicht zu Konflikten kommt. Vor allem dann, wenn der Prozessor eine entsprechende architekturseitige Unterstützung aufweist, ist zu erwarten, dass sich hierfür eine zuverlässige Lösung finden lässt. Mit anderen Worten: der VM-Monitor wird viel seltener abstürzen als ein kompliziertes, umfangreiches Betriebssystem. Ist ein Betriebssystem abgestürzt, so hängt nur die jeweilige virtuelle Maschine, während die anderen weiterlaufen. Dieser Vorteil macht sich auch beim Inbetriebnehmen und Einfahren von Programmen, bei Schulungen usw. bemerkbar.

Zudem wird in der Literatur, in Internet-Foren usw. hervorgehoben, dass von außen (vor allem: übers Internet) kommende Einflüsse nur die jeweilige virtuelle Maschine betreffen können. Ein Virus kann nur deren Dateien verändern, ein Spionageprogramm nur deren Datenbestände ausspähen. Dem Prinzip nach sind solche Aussagen nicht falsch. Es sei aber bemerkt, dass im Bereich der virtuellen Maschinen alles mit Software erledigt wird. Sie kann fehlerhaft sein (Sicherheitslücken). Auch hört man gelegentlich, dass manche Softwarefirmen gewissen Behörden (vor allem in den USA) gern einen Dienst erweisen[55)] ...

54: Ein Merkmal der IA-32-Prozessorarchitektur.

55: Ein Programm, das im Systemzustand läuft (und der VM-Monitor muss im Systemzustand laufen) hat die volle Kontrolle über alle Ressourcen; es kommt an alle Dateien aller virtuellen Maschinen heran ...

2.3.3 Virtuelle Maschinen durch Emulation

Ein Emulator ist ein Programm, das die Funktionsweise einer bestimmten Rechnerarchitektur – vor allem: die Wirkungen der Maschinenbefehle – mittels Software nachbildet. Ein typischer Einsatzfall: wir haben ein Program, das zur Ausführung auf Maschinen einer bestimmten Rechnerarchitektur X (Zielarchitektur) vorgesehen ist und als Maschinencode vorliegt, es gibt aber keinen entsprechenden Computer, der dieses Programm ausführen könnte. Der Ausweg: wir nehmen eine Maschine einer anderen Architektur (Plattform-Architektur) und schreiben ein Programm, das die Funktionsweise der Zielarchitektur nachbildet (emuliert).

An sich ist ein Emulator nicht besonders kompliziert. Wir müssen für alle Speichermittel der Zielmaschine entsprechende Speicherbereiche vorsehen, also für den Programmspeicher, für den Datenspeicher, für den Registersatz usw. (Abb. 2.29). Diese Speichermittel werden zweckmäßigerweise mit Feldstrukturen (Arrays) nachgebildet. Das zu emulierende Programm wird in das Befehls-Array geladen. Der Emulator holt Maschinenbefehl für Maschinenbefehl aus diesem Array und bildet dessen Wirkungsweise nach. Es ergibt sich eine recht einfache Programmschleife (Abb. 2.30). Um mehrere Maschinen zu emulieren, sind mehrere Datenstrukturen gemäß Abb. 2.29 bereitzustellen.

Die Stärke der Emulation liegt im Beobachten und Analysieren. Da alles über Software läuft, sind auch alle Einzelheiten des Programmablaufs der Auswertung zugänglich. Auch schwerste Fehler im Zielprogramm bringen einen Emulator nicht zum Absturz; es ist auch in solchen Fällen möglich, den Programmablauf in allen Einzelheiten zu beobachten (Debugging). Gelegentlich lohnt sich sogar die Emulation der eigenen Architektur (also der des Prozessors im PC)[56)].

Der Nachteil: die Emulation ist viel zu langsam, so dass sie nur für Sonderzwecke (s. Fußnote) in Betracht kommt. Richtwert: Das Emulieren eines Maschinenbefehls erfordert etwa 10 bis 50 Maschinenbefehle.

56: Anwendungsbeispiele: (1) Virensuchprogramme. Verdächtige Befehlsfolgen werden emuliert, um zu erkennen, ob sie wirklich in der Lage sind, Schaden anzurichten. (2) VM-Software. Emulation von Gerätefunktionen und Analysieren von Befehlswirkungen, die sich nicht hardwareseitig abfangen lassen. (3) Zwischenlösung beim Übergang von einer Prozessorarchitektur auf die andere (damit die alten Programme auf den neuen Maschinen laufen können). Die Macintoshs haben zwei derartige Übergänge hinter sich ...

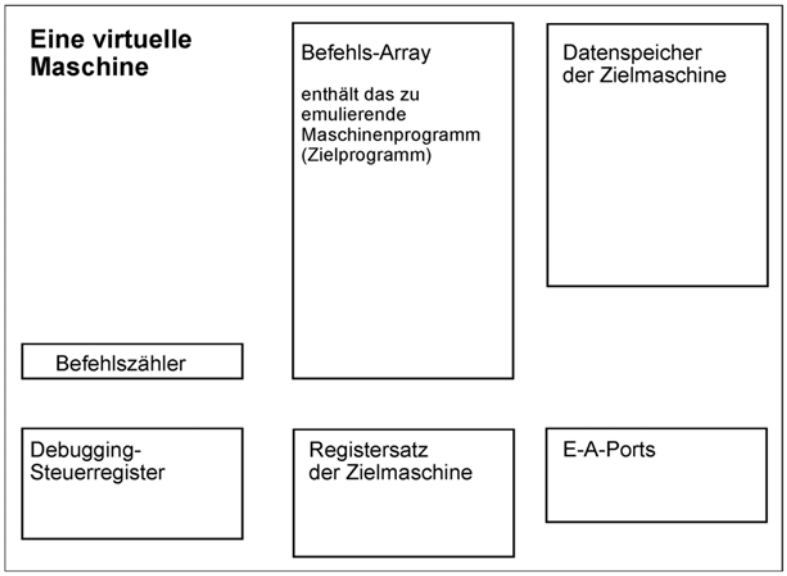


Abb. 2.35 Datenstrukturen eines typischer Emulators. Für jede virtuelle Maschine ist ein Satz solcher Datenstrukturen erforderlich.

Virtuelle Maschinen auf Grundlage fiktiver Befehlslisten

Dieses Prinzip hat man sich vor allem deshalb einfallen lassen, um die Entwicklung von Compilern zu erleichtern. Jede höhere Programmiersprache lebt schließlich auch von dem Versprechen, dass man ein Programm nur einmal schreiben muss und sich dann ohne Weiteres für verschiedene Plattformen übersetzen lassen kann. Jede Plattform erfordert aber einen Compiler. Um den Entwicklungsaufwand zu verringern, sind die Compiler-Autoren auf folgendes Prinzip gekommen:

- Der Compiler erzeugt das Programm zunächst in einer Zwischensprache. Das ist typischerweise die Assemblersprache eines fiktiven Prozessors, also eines Prozessors, den es als Hardware gar nicht gibt.
- Wollen wir das Programm auf einem System X laufen lassen – und kommt es nicht auf wirklich höchste Geschwindigkeit an –, so schreiben wir einen Interpreter (Kapitel 4) für diese Zwischensprache, mit anderen Worten, einen Emulator.
- Kommt es hingegen auf Leistung an, so schreiben wir einen weiteren Compiler, der das Programm des fiktiven Prozessors in den Maschinencode des Zielsystems umwandelt.

Beispiele solcher Zwischensprachen bzw. virtueller Maschinen: der sog. P-Code (für die Programmiersprache Pascal), (2) die Java Virtual Machine (JVM).

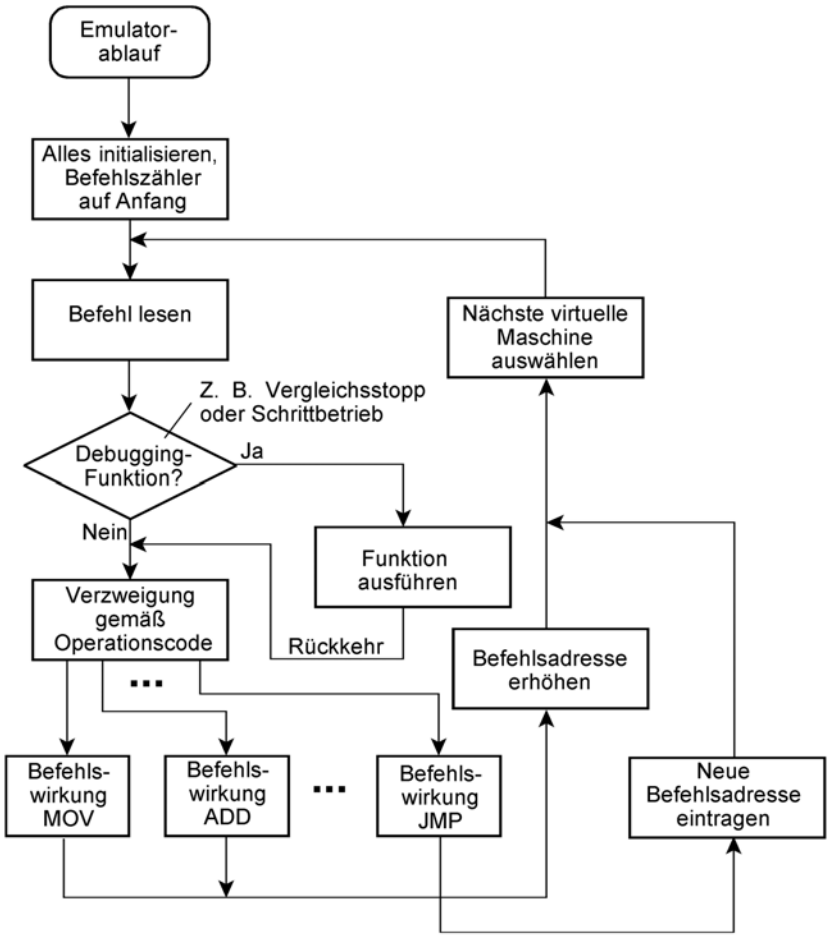


Abb. 2.36 Ein typischer Emulator im Ablaufdiagramm.

2.3.4 Sprachumgebungen als virtuelle Maschinen

Das ist der nächste Entwicklungsschritt. Weshalb muss das Anwendungsprogramm überhaupt die doch recht primitiven Maschinenbefehle – wie ADD, MOVE usw. – zu sehen bekommen? Stattdessen bieten wir von vornherein eine hochentwickelte Programmschnittstelle an, die sowohl elementare Operationen als auch den Umgang mit komplexen Datenstrukturen unterstützt. Dieses Prinzip wurde z. B. im System AS/400[57)] verwirklicht (Abb. 2.31).

57: Neuerdings iServer, eSeries usw.

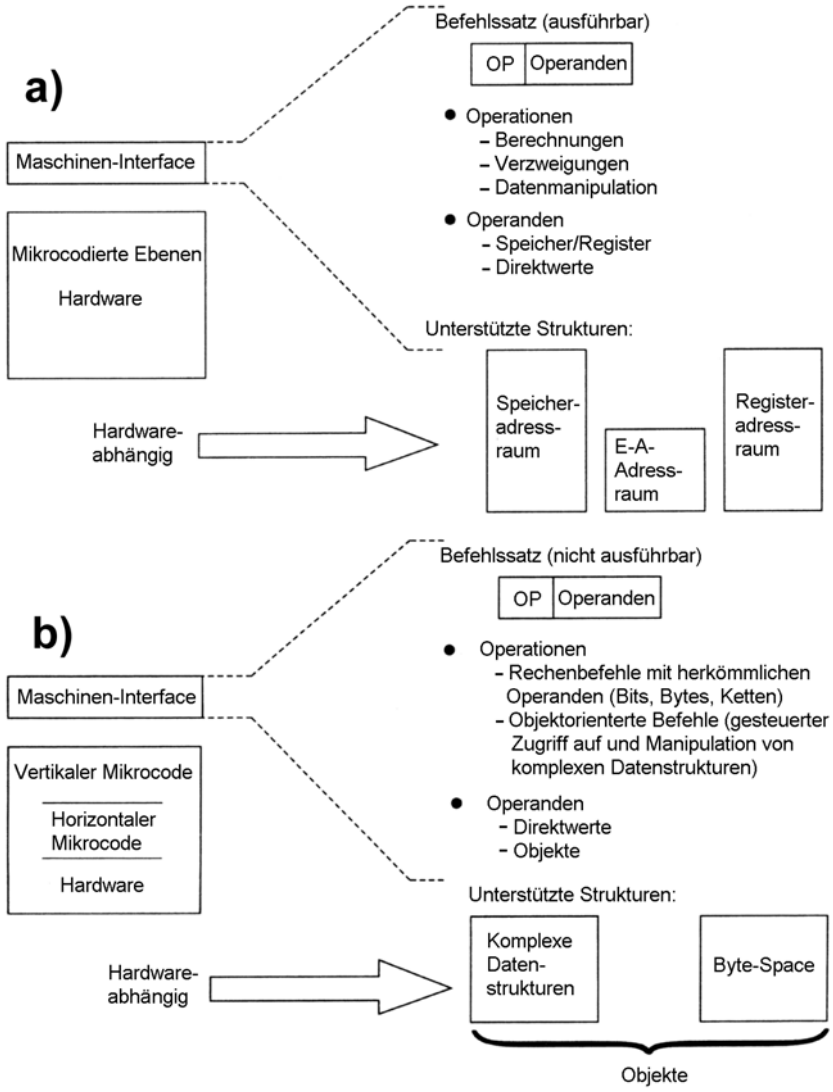


Abb. 2.37 Programmschnittstellen von Prozessoren (Maschinen-Interfaces) im Vergleich (IBM).

a) Herkömmliche Schnittstelle. Die Befehle, die der Anwender aufruft, wirken direkt auf die Hardware. Der Anwender sieht verschiedene Adressräume (Speicher, E-A, Register).
b) Die AS/400-Schnittstelle. Die Befehle, die der Anwender aufruft, werden von internen Programmen (hier Mikrocode genannt) interpretiert. Der Anwender sieht nur Objekte (vgl. Abschnitt 2.3.7). Das können Bytes sein, aber auch komplexe Datenstrukturen. U. a. ist ein hochentwickeltes Datenbanksystem bereits auf dieser Ebene eingebaut.

Die Vorteile: (1) deutlich höhere Sicherheit – kein Anwendungsprogramm ist in der Lage, in der Hardware irgend etwas zu verstellen, (2) Hardware-Unabhängigkeit. So hatten die ersten AS/400-Modelle einen speziellen Prozessor, während die neueren Typen mit PowerPC-Prozessoren bestückt sind (die meisten Nutzer haben von der Umstellung gar nichts bemerkt).

Der Nachteil: der hohe Entwicklungsaufwand. Die Sache wird nur dann etwas, wenn alles aus einer Hand kommt und auf die typischen Anwendungserfordernisse abgestimmt ist. Das Problem bei allen derartigen Lösungen mit einem hohen "Vorfertigungsgrad": wenn die fertigen Schnittstellen saugend zum Problem passen, dann ist die Leistung der von herkömmlichen Systemen sogar überlegen (denn die Schnittstellen-Entwickler können intern unbedenklich alle möglichen Tricks anwenden und so die Hardware voll ausnutzen). Passt es aber nicht, so ist der Anwender gezwungen, seine Funktionen aus den elementaren Operationen der Schnittstelle zusammenzustückeln – und dann wird es langsam (weil zwischen den Befehlen der Schnittstellen und der Hardware die Schichten des internen Mikrocodes liegen).

2.4 E-A-Subsysteme

Das E-A-Subsystem hat die Aufgabe, die angeschlossenen Geräte über ihre jeweiligen Interfaces so anzusteuern, dass sie die gewünschten Funktionen erbringen. Das Problem: jedes Gerät hat seine Eigenheiten, die Anwendungsprogramme sollen aber nicht von solchen Eigenheiten abhängig sein. Dies wird durch ein Übereinanderstapeln verschiedener Software-Schichten erreicht: das Anwendungsprogramm ruft ein erstes Steuerprogramm auf, dieses ein weiteres usw. "Ganz unten" – in den E-A-Steuerschaltungen und in den Geräten – treten dann Mikrocontroller in Tätigkeit, deren Firmware die eigentlichen Informationswandlungen, die mechanischen Bewegungen usw. steuert.

2.4.1 Die logische E-A-Steuerung

Die logische E-A-Steuerung bezieht sich auf abstrakte Konzepte der Ein- und Ausgabe, nicht auf konkrete Geräte. Es liegt nahe, die verschiedenen E-A-Geräte großzügig in Klassen einzuteilen und für jede Klasse ganz allgemeine Funktionen zu definieren. Dies sei an einigen einfachen Beispielen gezeigt:

Bildschirm
Der Bildschirm ist eine rechteckige Anordnung zum Anzeigen von Bildpunkten. Um ihn anzusteuern, genügt an sich eine einzige Anweisung:

PIXEL (X, Y, Wert)

X ist die Angabe der waagerechten Position, Y die der senkrechten. Der Wert gibt die jeweilige Farbe an. Wert = 0 bedeutet, dass "nichts" dargestellt wird (entspricht dem Löschen des betreffenden Bildpunktes).

Drucker
Die zu druckende Seite läßt sich wie ein Bildschirm behandeln.

Massenspeicher
Jeder Massenspeicher ist eine Anordnung, die eine Anzahl von Sektoren (zu beispielsweise 512 Bytes) speichern kann. Die Sektoren sind fortlaufend nummeriert. Es genügen zwei Anweisungen:

- Schreiben: WRITE (Sektor-Nummer),
- Lesen: READ (Sektor-Nummer).

2.4.2 Die physische E-A-Steuerung

Die physische Steuerung betrifft die jeweilige Hardware. Die "physische" Steuer-Software gehört unmittelbar zum E-A-Gerät oder zu dessen Schnittstellensteuerung (Hostadapter, Controller). Ihre Aufgabe besteht darin, die Anweisungen der logischen E-A-Steuerung in Programmabläufe umzusetzen, die die gewünschten Wirkungen in einer bestimmten Hardware hervorbringen (Abb. 2.32).

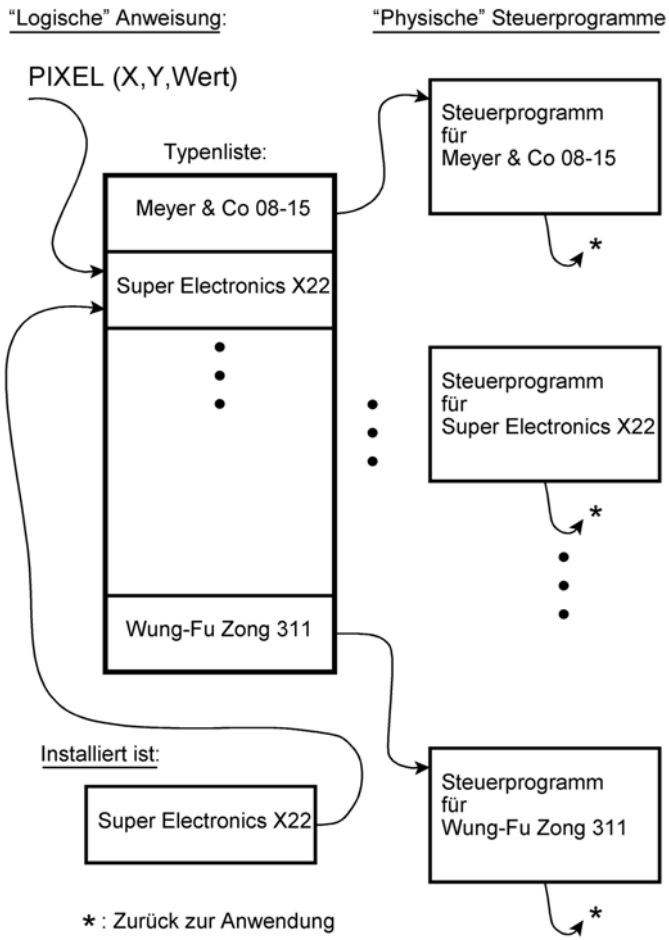


Abb. 2.38 Von der logischen zur physischen E-A-Steuerung (Beispiel).

Im Beispiel von Abb. 2.32 geht es um die Nutzung eines Bildschirms. Handelt es sich um einen typischen PC, so ist der jeweils installierte Videoadapter entsprechend anzusteuern. Jeder Hersteller eines Videoadapters muss hierzu ein Steuerprogramm liefern, das die Anweisung PIXEL (X, Y, Wert) in Zugriffe auf seine Hardware umsetzt, und zwar so, dass hierdurch tatsächlich ein Bildpunkt in der angegebenen Farbe auf der Position X waagerecht und Y senkrecht angezeigt wird. Das System hält eine Typenliste der unterstützten Hardware. Zudem weiß es, welche Hardware gerade installiert ist (das wurde z. B. im Rahmen von Plug&Play-Abläufen nach dem Einschalten erkannt). Dementsprechend sieht es in der Typenliste nach und verzweigt zum jeweiligen Steuerprogramm.

Genügen derart einfache E-A-Funktionen?
Im Prinzip ja. Zwei Nachteile sind aber offensichtlich:

- Es gibt viele Funktionen, die nahezu jede Anwendung benötigt (z. B. beim Bildschirm das Löschen der Anzeige, das Darstellen von Zeichen, von rechteckigen Fenstern usw.). Hätte man z. B. nur eine Funktion PIXEL (X, Y, Wert), so müsste jeder Anwendungsprogrammierer alles Weitere selbst schreiben.
- Jeder Funktionsaufruf läuft über die Plattform, z. B. ähnlich Abb. 2.32. Es ist klar, dass das Nachsehen in der Typenliste, das Verzweigen zum Steuerprogramm usw. Zeit kostet. Um einen Bildschirm von z. B. 1280 • 1024 Bildpunkten zu löschen, müsste unsere PIXEL-Funktion 1 310 720 mal aufgerufen werden. Demgegenüber bereitet es keine besonderen Schwierigkeiten, ein Löschprogramm zu schreiben, das den Bildspeicher einer bestimmten Grafhikkarte über direkte Hardwarezugriffe auf schnellstem Wege mit Nullen füllt. Damit der Anwendungsprogrammierer dieses Programm aber aufrufen kann, müsste die Plattform einen passenden Funktionsaufruf vorsehen (den man z. B. CLEAR nennen könnte).

Deshalb stellt man typischerweise einen recht umfangreichen Vorrat an allgemein brauchbaren E-A-Funktionen zur Verfügung.

Was muss die Anwendung von der Hardware wissen?
Das ist zunächst Ansichtssache. Es gibt zwei Auffassungen:

- Die Anwendung erfährt nichts. Die E-A-Steuersoftware sorgt dafür, dass unsinnige oder nicht erfüllbare Anforderungen entweder näherungsweise ausgeführt oder ganz einfach ignoriert werden. Bisweilen leistet man sich den Luxus, die Anwendung hierüber zu informieren. Beispiel: Die Angaben X, Y, Wert in unserer PIXEL-Anweisung werden als 32-Bit-Binärzahlen übergeben. Unser Bildschirm könnte also aus rund 4 Milliarden • 4 Milliarden Bildpunkten bestehen. Ist nun die Hardware in der Lage, z. B. 1280 • 1024 Bildpunkte darzustellen, so werden alle jeweils größeren Werte von X bzw. Y einfach ignoriert.

- Es gibt weitere Funktionen, worüber die Anwendung abfragen kann, was die installierte Hardware tatsächlich leistet. Der Anwendungsprogrammierer kann diese Werte zu Optimierungszwecken ausnutzen (er wird z. B. eine Bildschirmdarstellung anders gestalten, wenn er statt 1280 • 1024 Bildpunkten nur 640 • 480 Bildpunkte zur Verfügung hat oder statt rund 16 Millionen Farben nur 256). Unter Umständen kann er auch die Weiterarbeit ablehnen (z. B. anzeigen, dass das Video-Subsystem zu schäbig ausgestattet ist und ein Weiterarbeiten gar keinen Sinn hat).

Beide Auffassungen haben ihre Berechtigung und werden in modernen Plattformen je nach Zweckmäßigkeit implementiert.

2.4.3 Die "intelligente" E-A-Steuerung

Dieses Prinzip wurde bereits in den EDV-Anlagen (Mainframes) der 60er Jahre verwirklicht (Abb. 2.33).

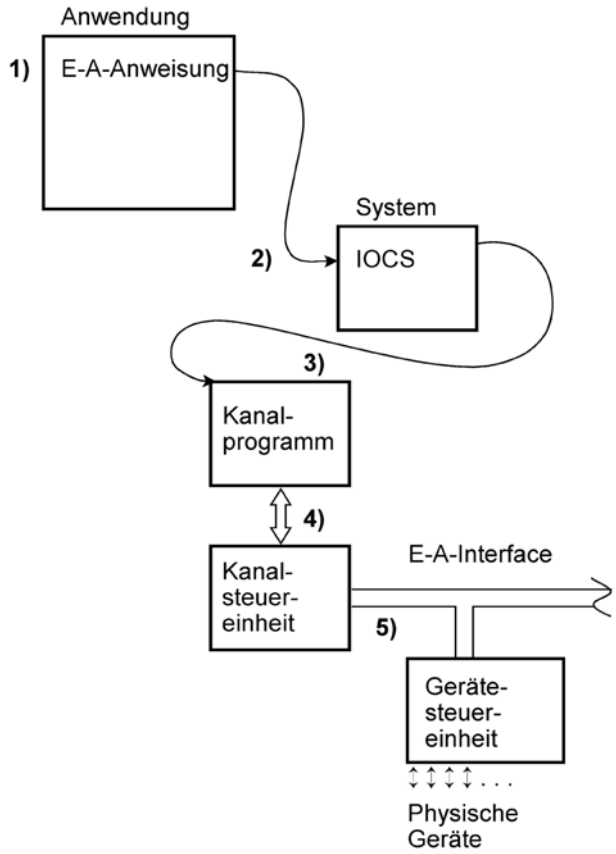


Abb. 2.39 "Intelligente" E-A-Steuerung am Beispiel der "klassischen" Mainframes.

Gemäß Abb. 2.33 laufen die E-A-Vorgänge in folgenden Schritten ab:

1) In einem Anwendungsprogramm ist eine E-A-Anweisung auszuführen.
2) Das führt zum Aufruf des Betriebssystems.
3) Dessen E-A-Steuersystem (I/O Control System IOCS) setzt ein Kanalprogramm auf (das Kanalprogramm besteht nicht aus "gewöhnlichen" Maschinenbefehlen, sondern aus speziellen Kanalkommandoworten).
4) Dieses Kanalprogramm wird von einer besonderen Hardware, einer Kanalsteuereinheit, ausgeführt.
5) Die Kanalsteuereinheit überträgt (standardisierte) E-A-Kommandos über das (standardisierte) E-A-Interface an die betreffende Gerätesteuereinheit. Diese endlich wirkt direkt mit dem Gerät zusammen.

Hierbei sind mehrere standardisierte Schnittstellen zu durchlaufen:

- den Aufruf des Betriebssystems,
- die Kanalkommandoworte (praktisch eine besondere Art von Maschinenbefehlen),
- das E-A-Interface mit seinem Satz an Kommandos, Zustandsmeldungen usw.

Alle Aufgaben der E-A-Steuerung sind letztlich mit Folgen weniger Kommando-Grundtypen zu erledigen. (Diese sind an sich einfach: Lesen, Schreiben, Steuerangaben zum Gerät übertragen, Zustandsmeldungen abholen usw.)

Der Vorteil für einen Geräteentwickler: er muss nur seine Kommandospezifikation implementieren – "wie es im System aussieht, kann ihm egal sein". Ein bewährter Trick: man legt sein Gerät so aus, dass es sich – hinsichtlich der Kommandos, der Zustandsmeldungen usw. – wie ein bereits bekanntes Gerät verhält. So hat man die vielfältigsten speziellen Geräte gebaut, die sich am E-A-Interface wie Plattenspeicher oder wie bestimmte Bildschirmgeräte verhalten.

Was spricht gegen eine solche Lösung?

Die Aufwendungen
Neben dem eigentlichen Prozessor, der die Anwendungs- und Systemprogramme ausführt, sind an jedem E-A-Vorgang noch wenigstens zwei weitere programmgesteuerte Einrichtungen beteiligt: nämlich die Kanal- und die Gerätesteuereinheit. Beide Hardwarekomplexe sind nicht gerade trivial. Keine E-A-Funktion der Geräte kann vom Prozessor aus direkt beeinflusst werden; es ist stets der Weg über Kanalprogramme und E-A-Kommandos zu gehen. Folglich müssen die – bisweilen recht komplizierten – Abläufe der Gerätesteuerung (stellen Sie sich beispielweise einen Laserdrucker oder einen Plattenspeicher vor) in der Gerätesteuereinheit erledigt werden. Manche Gerätesteuereinheiten sind deshalb kaum weniger aufwendig als die "eigentlichen" Prozessoren.

Die Latenzzeiten
Es ist nicht möglich, vom Anwendungsprogramm “schnell mal” auf E-A-Einrichtungen zuzugreifen, beispielsweise um in einem Bildpuffer ein paar Pixel zu setzen – alle E-A-Vorgänge laufen nur über die Kanalkommandos und die Kanalschnittstellen. Die typischen Mainframes leisten Hervorragendes, wenn es um große Datenmengen geht (wenn -zigtausende Bestellungen, Versicherungsfälle o. dergl. zu bearbeiten, Rechnungen zu drucken, Überweisungen zu tätigen sind usw.). Bewegte Bildschirmdarstellungen in 3D und Farbe sind jedoch nicht ihre Stärke, ebensowenig Aufgaben der Prozesssteuerung unter Realzeitbedingungen. Um einigermaßen annehmbare Reaktionszeiten zu erreichen, müssen die Geräte weitgehend autonom arbeitsfähig sein. Die herkömmliche Lösung: komplizierte und teure E-A-Geräte, die aber trotz allem nicht so flexibel sind und so ansehnliche Oberflächen bieten wie die modernen PCs. Die moderne Lösung: Vernetzung. Die Mainframe-Computer wirken als Hochleistungs-Server, während alles, was interaktiv ist, von angeschlossenen Netzcomputern erledigt wird.

2.4.4 Die E-A-Steuerung der ersten PCs

Wenn es um geringste Kosten geht, kann man sich den soeben skizzierten Aufwand nicht leisten. In den ersten Mikrocomputer-Konfigurationen musste der Prozessor praktisch alles allein tun. Um die Anwendungsprogramme davon zu entlasten, hat man die ganz elementaren E-A-Funktionen in Firmware realisiert. Beim PC kennen wir diese Firmware als BIOS (= Basic Input/Output System).

Was leistet das BIOS gegenüber einem aufrufenden Programm?
Das BIOS stellt Elementarfunktionen abstrakter (“logischer”) Geräte zur Verfügung. Die Plattenspeicherfunktionen betreffen das Lesen und Schreiben von Sektoren, die Bildschirmfunktionen das Darstellen von Bildpunkten oder Zeichen usw. Das rufende Programm wird lediglich von den technischen Einzelheiten der Ansteuerung des Interfaces, der Zugriffe auf bestimmte Adressen usw. verschont.

Die herkömmliche Arbeitsteilung sieht wie folgt aus: die Anwendung ruft das Dateisystem, das Dateisystem ruft das BIOS, das BIOS steuert die eigentlichen Zugriffe auf das Laufwerk.

Wie kommt das BIOS mit Hardware verschiedener Hersteller zurecht?
Hierfür hat man drei Lösungen gefunden:

- Einstellbare Parameter. Anwendung: vor allem bei Festplatten. Alle Festplatten werden auf nahezu gleiche Weise angesteuert. Die einzelnen Typen unterscheiden sich praktisch nur in der Anzahl der Oberflächen, der Spuren und der Sektoren je Spur. Die herkömmliche Lösung in AT-kompatiblen PCs: im ROM sind bis zu 47 Laufwerkstypen vordefiniert. Für den Fall, das in dieser Auswahl nichts Passendes zu finden ist, besteht die Möglichkeit, die Werte der tatsächlich installierten Laufwerke einzugeben (BIOS-Setup) und im Konfigurationsspeicher (CMOS-RAM mit Batteriestützung) zu halten.

- ROM-Erweiterungen. Steckkarten können ihr eigenes BIOS mitbringen. Nach jedem Einschalten prüft das BIOS, ob solche ROM-Erweiterungen installiert sind. Ist dies der Fall, werden sie in die BIOS-Programmschnittstelle einbezogen.
- Ladbare Erweiterungen. ROM-residente Programme sind an sich nur erforderlich, um nach dem Einschalten die ersten Zugriffe auf einen Datenträger zu veranlassen, der weitere Programme enthält. Sobald es möglich ist, Daten von außen zu holen, können beliebige Programme geladen werden, darunter auch solche, die BIOS-Funktionen ersetzen oder ergänzen.

Gerätetreiber

Schon bald wollte man an die PCs mehr anschließen als nur die gängige Peripherie. Auch die Programmierer hatten ehrgeizige Pläne. Die vergleichsweise einfache BIOS-Unterstützung genügte hierfür nicht mehr. Deshalb wurde eine weitere Software-Schicht eingeführt: die Gerätetreiber (Device Drivers). Ein Gerätetreiber ist ein E-A-Steuerprogramm, das vom Betriebssystem aufgerufen wird und hierzu nach bestimmten Regeln strukturiert ist. Im Gegensatz zum BIOS stellen Gerätetreiber dem aufrufenden Programm typischerweise “abstraktere” Funktionen zur Verfügung, beispielsweise Dateizugriffe.

Gerätetreiber unter DOS

Gerätetreiber, die vom DOS unterstützt werden, müssen beim Starten des Computers in den Arbeitsspeicher gebracht werden (Installieren der Gerätetreiber). Die Forderung, dass ein solches Programm im Arbeitsspeicher ständig anwesend (resident) sein muss, ist leicht erklärlich: es muss schließlich unmittelbar auf die Außenwelt, das heißt auf Signale von der Peripherie, reagieren können. Ein Laden auf Anforderung würde viel zu lange dauern. Abb. 2.34 veranschaulicht, wie ein Gerätetreiber im Prinzip aufgebaut ist. Er besteht aus einem fest formatierten Kopf, einer Adresstabelle und den Programmstücken, die die eigentlichen Funktionen realisieren. Ein solcher Treiber wird über bestimmte DOS-Funktionen gerufen. Dabei wird die laufende Nummer der gewünschten Treiber-Funktion als Parameter übergeben. DOS errechnet aus der übergebenen Funktionsnummer die jeweilige Tabellenadresse, entnimmt die Startadresse der betreffenden Funktion und verzweigt zu dieser.

DOS-Gerätetreiber geören typischerweise zu den Anwendungen

Diese Einfachauslegung (keine zentrale Verwaltung) war seinerzeit ein Problem. Vor allem Drucker, aber auch Grafikkarten usw. wurden mit einer größeren Zahl an Treiberdisketten ausgeliefert, und auch viele Anwendungen enthielten eine mehr oder weniger lange Liste von Treibern für die jeweils unterstützten Geräte. Es konnte durchaus sein, dass man zu einem neuen Gerät keinen Treiber für eine bestimmte Anwendung bekam. Dann konnte man das Gerät nicht einsetzen.

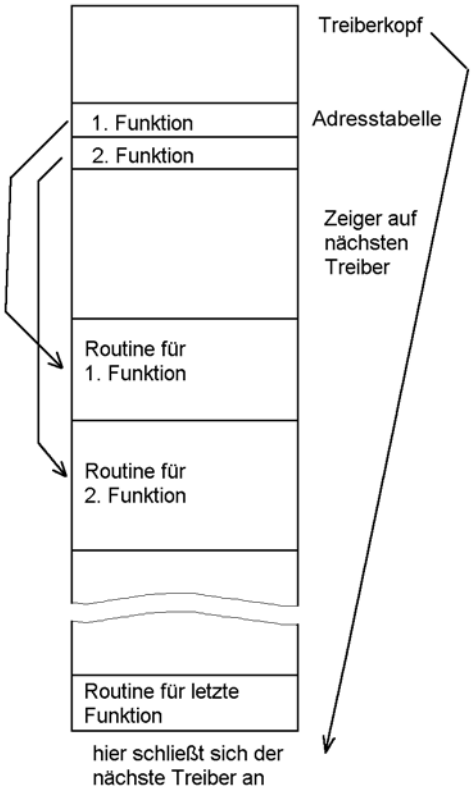


Abb. 2.40 Die grundsätzliche Struktur eines Gerätetreibers.

2.4.5 Die E-A-Steuerung der modernen PCs

Die programmseitige E-A-Steuerung beruht nach wie vor auf Gerätetreibern. Nur werden diese – im Gegensatz zum DOS – zentral von der Systemplattform verwaltet. Es ist aber nach wie vor so, dass viele Geräte eigene Treiber brauchen, die eigens installiert werden müssen[58]. Zudem sind moderne Plattformen nicht gerade einfach – eine zusätzliche Fehlerquelle. Nicht selten ist deshalb auch Treibersoftware fehlerhaft (und nur durch gelegentliche Updates auf einen befriedigenden Stand zu bringen[59]).

Gerätetreiber unter Windows

Die Entwicklungsgeschichte ist durch eine Abfolge verschiedener Treiberkonzepte gekennzeichnet:

58: Obwohl moderne Flash-ROMs mit mehreren MBytes so teuer auch nicht sind. Jedes Gerät könnte also seine Treibersoftware ohne weiteres unverlierbar bei sich tragen. Es würde aber dann womöglich ein paar Euro mehr kosten – und man könnte es sich kaum noch leisten, unausgereifte Lösungen in Riesenstückzahlen auf den Markt zu werfen ...

59: *Praxistipp:* Manche Geräte funktionieren nur dann zufriedenstellend, wenn man sofort nach der Treiberinstallation ins Internet geht und die aktuellen Updates herunterlädt.

16-Bit-Windows (Win16-API; Versionen 3.x, 95/98)
Treiber werden vorzugsweise in Form von DLLs implementiert.

32-Bit-Windows (Win32-API; Versionen 3.x, 95)
Die bevorzugte Treiberstruktur heißt VxD. Die Abkürzung steht für Virtual Device Driver, wobei das x durch bestimmte Kennbuchstaben ersetzt werden kann. Beispiele: VDD bezeichnet einen Bildschirmtreiber (D = Display), VPD einen Druckertreiber (P = Printer).

Windows NT
Nach einem Schichtenmodell organisierte Kerneltreiber (Layered Driver Model).

Windows 98/Me/2000/XP usw.
Die Treiberspezifikation heißt WDM = Windows Driver Model. WDM entspricht der Treiberorganisation von Windows NT, die um Plug&Play- und Stromsparvorkehrungen (Power Management) erweitert wurde.

In der Reihenfolge dieser Aufzählung nimmt die Wirksamkeit der Schutzvorkehrungen zu. Die Treiberentwicklung wird immer aufwendiger (zunehmende Kompliziertheit, kostspieligere Entwicklungsumgebungen):

- Win16-API, DLL-Treiber: praktisch kein Schutz (“Programm darf alles”). Programme, die auf Sondergeräte zugreifen, können mit herkömmlichen Werkzeugen selbst entwickelt werden (Direktzugriffe aus der Anwendung heraus).
- Win32-API, VxD-Treiber: die Schutzvorkehrungen der Hardware können ausgenutzt werden, um bestimmte Ressourcen zu reservieren oder bestimmte Zugriffe abzufangen (z. B. in dem Sinne, dass nur der Treiber XYZ auf bestimmte E-A-Adressen zugreifen darf). Was nicht reserviert oder abgefangen wird, ist aber nach wie vor verfügbar und kann z. B. von einer Anwendung aus mit E-A-Befehlen (IN und OUT) angesprochen werden.
- Windows NT und Nachfolger (WDM): hier hat das System alles unter Kontrolle. Physische Zugriffe können nur von sog. Kernel-Treibern ausgeführt werden. Es ist nicht möglich, von der Anwendung aus auf die Hardware zuzugreifen.

Das E-A-System der modernen Windows-Versionen (von NT an) wurde im Hinblick auf die Anforderungen großer Systeme entwickelt. In solchen Systemen ist eine Vielzahl von Geräten unterschiedlichen Typs zu unterstützen (Laufwerke, Netzwerkanschlüsse, Drucker usw.), die von vielen gleichzeitig laufenden Anwendungen genutzt werden (Einsatzbeispiel: die Server der Internet-Dienstanbieter). Solche Anforderungen lassen sich nur durch mehrere Software-Schichten mit regulären Schichtübergängen erfüllen (Abb. 2.35 und 2.36).

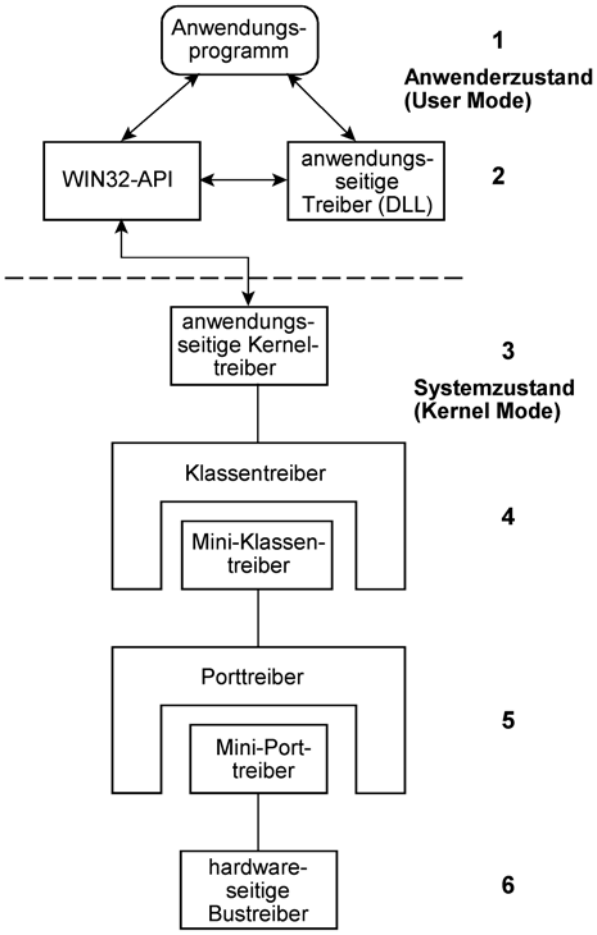


Abb. 2.41 Ein allgemeines Schema der Gerätetreiberorganisation (nach Microsoft). Nähere Erläuterung im nachfolgenden Text.

Gemäß Abb. 2.35 werden mehrere Treiberschichten durchlaufen. Die verschiedenen Arten der E-A-Einrichtungen haben jeweils eine eigene Schichtanordnung (Driver Stack). Die meisten Driver Stacks enthalten nicht alle hier dargestellten Treibertypen.

1), 2) Das Anwendungsprogramm ruft die E-A-Funktionen entweder direkt über die Anwendungsprogrammschnittstelle des Systems oder über einen Treiber, der im Anwenderzustand arbeitet. Solche Treiber werden üblicherweise als DLLs bereitgestellt.

3) Manche anwendungsseitigen E-A-Funktionen werden im Systemzustand ausgeführt (das ist dann der Fall, wenn man auf Systemressourcen zugreifen muss).

4) Klassentreiber erbringen die hardware-unabhängigen Funktionen für bestimmte Arten von E-A-Einrichtungen. Sie haben typischerweise die meiste Arbeit zu leisten.

5) Porttreiber steuern E-A-Ports, Verteiler (Hubs), Schnittstellensteuerungen, Interfaceadapter usw. Sie sind in der Lage, Unterbrechungen zu empfangen und zu verarbeiten.

6) Bustreiber wirken direkt auf die Hardware. Das System stellt solche Treiber für alle im PC-Bereich üblichen Bussysteme bereit.

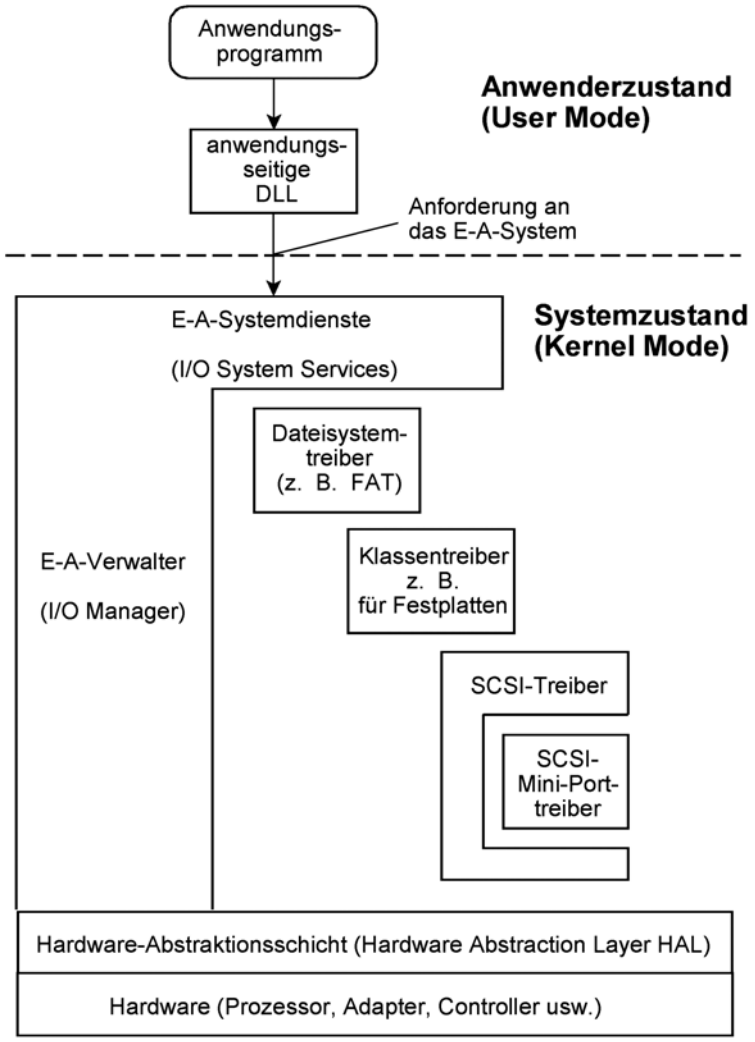


Abb. 2.42 Gerätetreiberorganisation am Beispiel eines Dateisystemtreibers (nach Microsoft).

Das E-A-System besteht aus dem E-A-Verwalter (I/O-Manager) und den einzelnen Kernel-Treibern. Der E-A-Verwalter enthält die E-A-Systemdienste (I/O System Services), die die elementaren Schnittstellen zwischen dem Rest des Systems und den Treibern bereitstellen. Das betrifft u. a. solche elementare E-A-Funktionen, wie Datei erzeugen, Datei lesen, Datei schreiben usw. Die E-A-Schnittstellen zum Anwenderprogramm werden in DLLs bereitgestellt, die Schnittstelle zur eigentlichen Hardware ist die Hardware-Abstraktionsschicht (Hardware Abstraction Layer HAL). Die HAL enthält u. a. die elementaren Zugriffsfunktionen (Lesen und Schreiben) für E-A-Ports und Register. Dazwischen liegen die Schichten der Kerneltreiber. Minitreiber sind in andere Treiber eingebunden (wrapped). Sie wirken nur mit dem betreffenden übergeordneten Treiber und mit der zu steuernden Einrichtung zusammen. Diese Treiberart hat man eingeführt, um die Treiberentwicklung für allgemein übliche Geräte (Hostadapter, Grafikkarten usw.) zu erleichtern. Wer beispielsweise einen neuen SAS-Adapter herausbringen will, muss lediglich einen kleinen, überschaubaren Minitreiber schreiben (der nur die

Besonderheiten dieses Adapters behandelt) und nicht etwa einen ausgewachsenen Gerätetreiber (der alle Einzelheiten der SCSI-Standards zu berücksichtigen hat).

Die DirectX-APIs

Diese Programmschnittstellen sind zur Unterstützung von Multimedia-Anwendungen, vor allem von Videospielen, vorgesehen (Tabelle 2.5). „Direct" ist hier wörtlich zu nehmen – es geht darum, dem Anwendungsprogrammierer einen möglichst direkten Zugriff auf die Multimedia-Hardware zu ermöglichen. Hierzu ist DirectX nach einem besonderen Schichtenmodell organisiert (Abb. 2.37 und 2.38).

DirectX-Komponente	Funktionen
DirectDraw	Unmittelbare Beeinflussung des Bildspeichers
Direct3D	Zugang zu Graphiksoftware und Beschleunigungshardware für 3D-Darstellungen
DirectSound	Stereo- und 3D-Sound, Speicherverwaltung für die Soundkarte
DirectPlay	Unterstützung von Spielen mit mehreren Spielern (Multiplayer Games)
DirectInput	Eingabe von verschiedenen Einrichtungen, insbesondere von Spielgeräten, Unterstützung von Rückmeldungen (Forced Feedback)
DirectShow	Zugang zu Beschleunigungshardware für MPEG-1-Videowiedergabe
DirectSetup	Automatisches Installieren der DirectX-Programme

Tabelle 2.5 DirectX-Komponenten im Überblick (nach Microsoft).

Die Anwendungen nutzen DirectX über eine Vielzahl von Funktionsaufrufen. Bestimmte Funktionen werden von der Hardware des jeweiligen PCs unterstützt, andere nicht. Beim Hochfahren legt DirectX eine entsprechende Liste an. Wird eine Funktion aufgerufen, so sieht DirectX in dieser Liste nach. Wird die Funktion von der Hardware unterstützt, so wird der Aufruf über die HAL-Schicht auf schnellstem Wege an die Hardware weitergereicht. Wird sie hingegen nicht unterstützt, so wird eine entsprechende Funktion in der HEL-Schicht aufgerufen, die die gewünschten Wirkungen programmtechnisch nachbildet (emuliert).

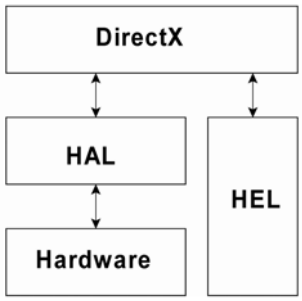


Abb. 2.43 Das Schichtenmodell der DirectX-APIs (nach Microsoft). HAL = Hardware Abstraction Layer; HEL = Hardware Emulation Layer.

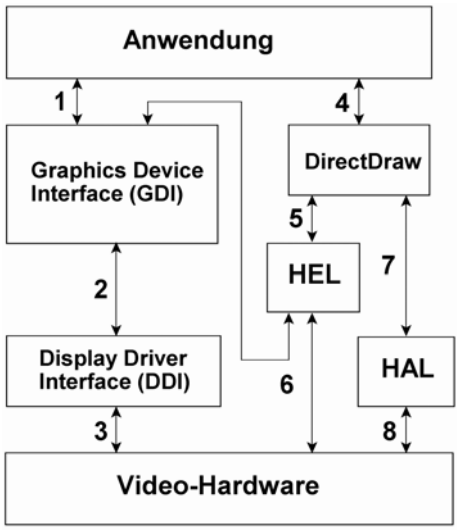


Abb. 2.44 DirectX-APIs am Beispiel der Komponente DirectDraw (nach Microsoft). 1, 2, 3 - der übliche Zugang zur Hardware; . 4...8 - schnellere Zugriffswege.

Windows-Anwendungen haben herkömmlicherweise lediglich die Funktionen der universellen Grafikschnittstelle (Graphics Device Interface GDI) zur Verfügung (1). Diese sind vollkommen unabhängig von der Video-Hardware. Die Videohardware wird ihrerseits über Gerätetreiber gesteuert, die von der GDI-Schnittstelle aus über eine ebenfalls vereinheitlichte Treiberschnittstelle (Display Driver Interface DDI) angesprochen werden (2, 3). Es dauert offensichtlich seine Zeit, bis sich ein Wunsch der Anwendung – z. B. eine Linie oder ein Rechteck darzustellen – tatsächlich im Bildspeicher (und damit auf dem Bildschirm) bemerkbar machtBeim Spielen dauert der Zugriff über die Schnittstellen 1, 2, 3 oft entschieden zu lange. Solche Anwendungen können deshalb u. a. die DirectDraw-Schnittstelle im Rahmen von DirectX verwenden (4). Wird im jeweiligen PC die betreffende Funktion nicht von der Hardware unterstützt, so wird sie über die HEL-Schnittstelle (5) von besonderen Treiberprogrammen nachgebildet (emuliert). Diese Programme greifen teils direkt auf die Hardware zu, teils nutzen sie Funktionen der GDI-Schnittstelle (6). Wird hingegen die gewünschte Funktion von der Hardware unterstützt, so leitet DirectDraw den Aufruf auf die HAL-Schnittstelle um (7). Die entsprechenden Treiberprogramme wirken dann direkt auf die Hardware (8).

Hinweis:
Nicht nur die Anzahl der Kästchen im Zugriffsweg zählen – es kommt vor allem darauf an, *was* dort programmiert ist. Es leuchtet ein, dass total universelle Schnittstellen, die das Anwendungsprogramm tatsächlich von der Hardware vollkommen abschirmen (wie GDI und DDI), viel kompliziertere Programme erfordern als Schnittstellen, die für den möglichst direkten Durchgriff auf die Hardware ausgelegt sind.

Gerätetreiber unter Unix/Linux
Die Gerätetreiber gehören – wie bei den modernen Windows-Versionen – zum Kern des Systems (Kerneltreiber). Während man aber für Windows im Grunde x-beliebige Steuerprogramme schreiben kann, hat Unix traditionell eine ganz elementare universelle Schnittstelle: alle Geräte sind Dateien, alle Daten sind Aneinanderreihungen von Bytes. Beide Auslegungen haben Vor- und Nachteile.

Bei Windows steht der PC-typischen Flexibilität (Direktzugriff auf alles, sobald das Programm im Systemzustand läuft) das offensichtliche Problem der Sicherheit gegenüber – es ist nicht nur eine Sicherheitslücke, sondern geradzu ein Scheunentor ... Abhilfe: die Treiberzertifizierung (Microsoft).

Der überschaubare Zugangsweg von Unix bietet bessere Ansätze für Sicherheitskontrollen, hat aber gelegentlich höhere Latenzzeiten zur Folge (das betrifft z. B. Bildspeicherzugriffe).

Die Unix-Systeme unterscheiden traditionellerweise zwischen drei Arten von Geräten (Tabelle 2.6). Einfache E-A-Ports, Mikrocontroller-Schnittstellen usw. können typischerweise als zeichenorientierte Geräte aufgefasst werden. Ein solches Gerät wird vom System wie eine Datei behandelt. Der Anwender sieht lediglich eine Datei mit einem speziellen Namen, auf die er die üblichen Dateioperationen (OPEN, CLOSE, READ, WRITE usw.) anwenden kann. Jede solche Datei hat einen bestimmten Verzeichniseintrag im Dateisystem (z. B. /dev/ser1 oder/dev/lp1).

Geräteart	Zugriffsarten
Zeichenorientierte Geräte (Character Devices)	Byteweise Lese- und Schreibzugriffe
Blockorientierte Geräte (Block Devices)	Blockweise Lese- und Schreibzugriffe. Ein Block entspricht typischerweise 1 kBytes
Netzwerkschnittstellen (Network Interfaces)	Senden und Empfangen von Datenpaketen

Tabelle 2.6 Gerätearten in Unix-Systemen.

2.4.6 Geräteverwaltung

Die Geräteverwaltung muss dafür sorgen, dass die einzelnen Anwendungsprogramme die E-A-Geräte (z. B. Drucker, Bildschirme, Scanner usw.) benutzen können, sich aber dabei nicht gegenseitig behindern. Offensichtlich wird es nichts Rechtes werden, wenn z. B. zwei im Multitasking laufende Programme gleichzeitig drucken oder Faxe senden wollen – und dazu auf denselben Drucker oder auf dasselbe Modem zugreifen. Hierzu ist es notwendig, zu erkennen, was zusammengehört. Eine einfache Lösung: die Programme wissen es selbst. Dann genügen zwei Systemanweisungen: Reservieren (RESERVE) und Freigeben (RELEASE)[60].

60: Das sind Allgemeinbezeichnungen, mit denen das Prinzip veranschaulicht werden soll; keine Anweisungen bestimmter Systeme.

In einer Anwendung, die den Drucker Nr. 1 nutzen möchte, muss dann z. B. folgender Ablauf programmiert sein:

RESERVE Printer_1

Druck des jeweiligen Dokuments

RELEASE Printer _1

Hierbei wird die RESERVE-Anweisung so lange ausgeführt, bis der Drucker tatsächlich verfügbar geworden ist. Ist er verfügbar, so wird er als “besetzt” gekennzeichnet. Eine weitere Anwendung, die in einer anderen Task läuft und auch den Drucker braucht, findet ihn dann als nicht verfügbar vor.

Nach dem Drucken wird der Drucker mittels der RELEASE-Anweisung wieder freigegeben, so dass er ggf. von Anwendungen in anderen Tasks genutzt werden kann.

Das Reservieren und Freigeben ist typischerweise nicht Sache der Anwendung, sondern es liegt in der Verantwortung der Plattform (die Anwendung übergibt komplette Aufträge und überlässt die Einzelheiten dem System).

2.5 Dateisysteme

2.5.1 Die Datei und ihre Entwicklungsgeschichte

Eine Datei (File) ist eine zusammengefasste, als Ganzheit handhabbare Sammlung von Bytes mit innerer Struktur und veränderlicher Größe. Sie ist gekennzeichnet durch einen Dateinamen (File Name oder Identifier), durch Angaben zu Struktur und Zugriffsorganisation (Dateibeschreibung bzw. File Description) sowie durch ihren eigentlichen Inhalt. Die Datei ist ein ganz naheliegendes Konzept. Es ist die Übertragung altbewährter Prinzipien der Tabelle, der Liste oder der Kartei auf die Computertechnik. So enthält eine Stückliste (z. B. zu einer Maschine) Angaben über die Bezeichnung jedes Einzelteils, über den verwendeten Werkstoff, die Rohmasse, die benötigte Stückzahl usw. Sinngemäß kann man eine Materialbedarfsdatei aufbauen, in der für jedes Einzelteil ein Datensatz vorgesehen ist. Das Dateikonzept hat sich im Laufe der Zeit entwickelt und ist sowohl durch die Anforderungen der Anwender als auch durch die technische Entwicklung immer weiter vervollkommnet worden.

Herkömmlicherweise sind Dateien zumeist in Form von *Datensätzen* (Records) organisiert. Dementsprechend gibt es Funktionen zum Schreiben und Lesen von Datensätzen. Beim Schreiben wird der Datensatz vom Arbeitsspeicher zum Massenspeicher transportiert; beim Lesen gelangt der Datensatz vom Massenspeicher in den Arbeitsspeicher.

Hinsichtlich der Zugriffsmöglichkeiten unterscheidet man zwischen Dateien mit sequentiellem Zugriff (Sequential Files) und solchen mit wahlfreiem Zugriff (Random oder Direct Access Files). Bei sequentiellem Zugriff sind nur aufeinanderfolgende Datensätze ohne weiteres zugänglich (die elementaren Funktionen sind "Zugriff zum ersten Datensatz" und "Zugriff zum nachfolgenden Datensatz"). Hingegen kann man beim wahlfreien Zugriff vom Anwendungsprogramm aus beliebige Datensätze durch Angabe ihrer laufenden Nummer oder eines bestimmten Kennzeichens (Schlüssels) erreichen (die elementaren Funktionen sind dann "Zugriff zu Datensatz n" bzw. "Zugriff zum ersten/nächsten Datensatz mit Schlüssel k").

Man erwartet von einem modernen Dateisystem, dass die einzelnen Dateien ihre Größe beliebig ändern dürfen (in Systemen älteren Ursprungs muss der Anwendungsprogrammierer festlegen, wieviele Datensätze oder Bytes die jeweilige Datei umfasst). Diese Anforderung bestimmt die Auslegung der Dateiorganisation auf Massenspeichern in wesentlichem Grade. Es ist klar, dass solche Dateien durch beschreibende Angaben ergänzt werden müssen, die ihre Lage, ihre aktuelle Größe usw. genau bezeichnen, und dass die Dateizugriffe einen nicht unerheblichen Verwaltungsaufwand erfordern.

Aus Sicht der Plattform ist eine Datei nichts weiter als ein in seiner Größe veränderlicher Haufen Bytes auf einem Massenspeicher, der unter einem Namen (dem *Dateinamen, Dateibezeichner*) geführt wird. Diese recht roh klingende Definition besagt aber das Wesentliche:

- Für die Plattform sind Dateien nur Aneinanderreihungen von Bytes; die interne Strukturierung (z. B. in Form von Datensätzen (Records)) obliegt dem Anwendungsprogramm.
- Die Datei kann ihre Größe beliebig ändern, im Besonderen wachsen. Die typischen Programmschnittstellen unterstützen das Hinzufügen von Bytes und das Löschen von Dateien, aber nicht das Entfernen von Bytes (das Umbauen von Dateien ist Sache des Anwendungsprogramms.
- Jede Datei hat einen Dateinamen, unter dem sie verwaltet wird.

Elementare Dateizugriffe

Es sind nur wenige Funktionen definiert[61]. Ganz elemenar sind die Zugriffsfunktionen Schreiben, Lesen und Positionieren sowie die Verwaltungsfunktionen Öffnen und Schließen[62]. Die Verwaltungsfunktionen hat man deshalb eingeführt, um den Verwaltungsaufwand jeweils nur einmal zu haben (und nicht bei jedem Zugriff). Aus Sicht der Zugriffsfunktionen ist die Datei im Grunde ein adressierbarer Speicherbereich, dessen Länge bis aufs Byte genau bestimmt ist (Abb. 2.39).

61: Die Funktionsbezeichnungen im folgenden Überblick sind Allgemeinbezeichnungen ohne Bezug auf ein bestimmtes System.

62: Das Anlegen neuer Dateien, das Löschen von Dateien usw. werden teils mit den genannten elementaren Funktionen, teils mit weiteren Funktionen erledigt. Beispiel: die Open-Funktion legt eine neue Datei an, wenn der angegeben Dateiname nicht gefunden wurde.

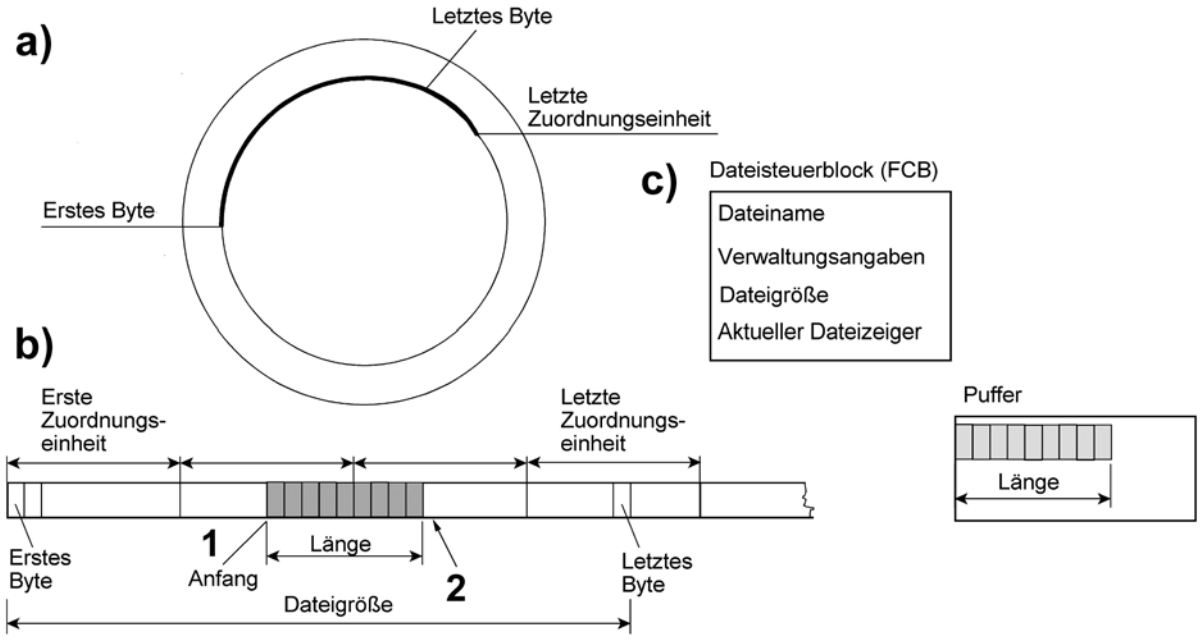


Abb. 2.45 Dateizugriffe. a) die Datei auf dem Massenspeicher; b) die Datei aus Sicht der Zugriffsfunktionen; c) die zugehörigen Datenstrukturen (Dateisteuerblock und Puffer). 1 - der Dateizeiger vor dem Zugriff (zeigt auf das erste Byte des Blockes, der zu lesen oder zu schreiben ist); 2 - der Dateizeiger nach dem Zugriff (zeigt auf das Byte, das nach dem Ende des übertragenen Blockes nachfolgt).

Öffnen (Open)

Es wird ein Dateisteuerblock (File Control Block FCB) angelegt. Der Datenträger wird nach dem Dateinamen durchsucht. Die gefundenen Zugriffsangaben werden in den Dateisteuerblock eingetragen. Das Anwendungsprogramm erhält eine Art laufende Nummer (Handle) für den Dateisteuerblock. Alle Zugriffsfunktionen beziehen sich nicht mehr auf den Dateinamen, sondern auf diese laufende Nummer. Der Dateisteuerblock enthält zudem einen Dateizeiger (File Pointer), der auf das jeweils erste zu adressierende Byte der Datei verweist.

Schreiben (Write)

Es wird eine bestimmte Anzahl an Bytes in die Datei geschrieben. Diese Bytes werden aus einem Pufferbereich entnommen. Verweist der Dateizeiger auf das letzte Byte der Datei, werden die weiteren zu schreibenden Bytes angehängt, die Datei wird also verlängert. Passt die Datei nicht mehr auf den Datenträger, wird der Schreibvorgang abgebrochen. Die Anzahl der tatsächlich geschriebenen Bytes wird an die Anwendung zurückgegeben.

Lesen (Read)

Es wird eine bestimmte Anzahl an Bytes aus der Datei gelesen und in einen Pufferbereich eingetragen. Wurde das letzte Byte der Datei gelesen, wird der Lesevorgang abgebrochen. Die Anzahl der tatsächlich gelesenen Bytes wird an die Anwendung zurückgegeben.

Positionieren (Seek)
Mit diese Funktion kann der Dateizeiger auf einen beliebigen Wert eingestellt werden.

Schließen (Close)
Diese Verwaltungsfunktion gibt den Dateisteuerblock wieder frei.

Sequentielle und wahfreie Zugriffe
Da nach jedem Lesen oder Schreiben der Dateizeiger auf das jeweils nachfolgende Byte verweist, ergeben sich durch Aneinanderreihen von Lese- oder Schreibfunktionen fortlaufende (sequentielle) Dateizugriffe. Für wahlfreie Zugriffe ist vor jedem Lesen oder Schreiben der Dateizeiger entsprechend zu positionieren.

Allgemeine Verwaltungsaufgaben
Wer ein Dateisystem entwickeln möchte, muss zunächst zwei Probleme lösen:

- Jede Datei unter ihrem Namen überhaupt zu finden (Verzeichnisverwaltung).
- Die Massenspeicher oder Datenträger mit dem Inhalt der Dateien zu belegen (Speicherverwaltung).

Verzeichnisverwaltung
Die Dateinamen aller Dateien werden in *Verzeichnissen* geführt. Ein Verzeichnis (Directory, Katalog, Ordner) ist eine Liste aller Dateinamen mit den zugehörigen Verweis- und Steuerangaben.

Speicherverwaltung
Ein Massenspeicher (z. B. eine Festplatte) ist im Grunde ähnlich zu verwalten wie ein Arbeitsspeicher. Wir können uns den Massenspeicher oder Datenträger als einen Behälter für Zuordungseinheiten (Sektoren, Cluster) vorstellen und die Zuordnungseinheiten wiederum als Behälter für die Dateien.

Zuordnungseinheiten
An sich könnte man auch die auf einem Datenträger gespeicherten Bytes einfach durchnummerieren. Es sind aber sehr vieleBytes. Man würde also lange Adressen brauchen, und es würde viel Zeit kosten, die Belegung des Datenträgers bis aufs einzelne Byte zu verwalten. Deshaln beschäftigt sich die Speicherverwaltung eines Dateisystems nicht mit einzelnen Bytes, sondern mit größeren Blöcken[63]. In den Dateisystemen sind diese Blöcke jeweils gleich lang. Der Microsoft-Fachbegriff: Zuordnungseinheiten (Allocation Units). Die elementare Zurdnungseinheit ist der Sektor. Der typische Sektor ist 512 Bytes lang. Diese Länge ist eine Art Industriestandard; sie hat sich in der Entwicklungsgeschichte der magnetischen

63: Es ist genau zu unterscheiden: die Zugriffsfunktionen der Anwendungen (Lesen, Schreiben, Positionieren) gehen bis aufs Byte, die Verwaltung der Datenträger befasst sich hingegen nur mit Zugriffseinheiten. Erforderlichenfalls wird großzügig aufgerundet.

Signalspeicherung als zweckmäßig herausgestellt. Mit der Vergrößerung der Speicherkapazitäten wurden es aber immer mehr Sektoren. Bald waren die üblichen Adresslängen nicht mehr ausreichend, um alle Sektoren eines Datenträgers zu adressieren. Deshab hat man Zuordnungseinheiten aus mehreren Sektoren gebildet. Der allgemeine Fachbegriff: Cluster[64]. Typische Clustergrößen ligen zwischen 1k und 4 kBytes. Der Kürze wegen wollen wir im den folgenden Erläuterungen nur von Sektoren sprechen, wo eigentlich Zuordnungseinheiten = Sektoren oder Cluster gemeint sind.

Probleme der Speicherverwaltung
Besondere Schwierigkeiten ergeben sich aus folgenden Tatsachen:

- Dateien können ihre Größe im Verlauf der Programmausführung beliebig ändern.
- Dateien können sehr groß werden.
- Die meisten Dateien müssen erhalten bleiben (im Gegensatz dazu können viele Bereiche im Arbeitsspeicher wieder freigegeben werden, wenn die Ausführung des Programms beendet ist).
- Massenspeicherzugriffe erfordern viel mehr Zeit als Arbeitsspeicherzugriffe. Trotzdem soll es schnell gehen.

Eine ganz einfache Lösung
Das Verzeichnis ist eine Liste von Einträgen. Jede Datei hat einen Eintrag. Dieser enthält den Dateinamen, die laufende Nummer des ersten Sektors sowie die Anzahl der Sektoren (Tabelle 2.7).

Dateiname	1. Sektor	Anzahl der Sektoren
Meiers Beschwerden	201	88
Stückliste	289	22

Tabelle 2.7 Eine einfache Verzeichnisstruktur. Beispiel mit zwei Einträgen.

Wollen wir z. B. auf die Stückliste zugreifen, so würde das Dateisystem das Verzeichnis durchsuchen, bis es auf diesen Dateinamen stößt. Aus dem gefundenen Verzeichniseintrag geht weiterhin hervor, dass die Datei auf Sektor 289 beginnt und 22 Sektoren lang ist (d. h., dass sie mit Sektor 310 endet). Das Prinzip funktioniert tatsächlich. Es funktioniert sogar recht ordentlich – wenn man voraussetzt, dass die Dateigröße feststeht. Wann ist damit zu rechnen?

- Wenn die Dateigröße von der Anwendung aus festgelegt wird. Der Programmierer oder der Nutzer müsste sich also beispielsweise überlegen, dass eine Sammlung von Beschwerdebriefen bestenfalls einige -zigtausend Bytes lang sein kann und auf Grund dessen

64: Wörtlich = Bündel.

zu dem Entschluss kommen, z. B. die in Tabelle 2.4 genannten 88 Sektoren (bei 512 Bytes je Sektor = 45 056 Bytes) vorzugeben. (Dies ist aber – wie bereits erwähnt – im PC-Bereich nicht üblich.)

- Wenn die Datei nicht mehr verändert wird und nur noch abgespeichert werden soll (z. B. bei der Datensicherung).

Dateien veränderlicher Länge

Auch hierfür gibt es eine naheliegende Lösung: wir führen im Verzeichnis eine Liste der Sektoren mit, welche die jeweilige Datei enthalten. Der offensichtliche Nachteil: die Liste kann lang werden. Z. B. entsprechen 5 MBytes rund 10 000 Sektoren. Ein weiterer Nachteil: die Verzeichniseinträge werden unhandlich – was tun wir z. B. mit den 10 000 Sektornummern, wenn die Datei gelöscht wird?

Trennung zwischen Verzeichnis und Datenträgerbeschreibung

Um die eben skizzierten Nachteile zu vermeiden, verwendet man, um die Belegung des Datenträgers zu beschreiben, besondere Informationsstrukturen. Die Verzeichniseinträge enthalten dann lediglich entsprechende Verweisangaben.

Eine weitere einfache Lösung

Es gibt gar keine Verzeichnisse. Dateien werden stets als Ganzes gespeichert (Abb. 2.40).

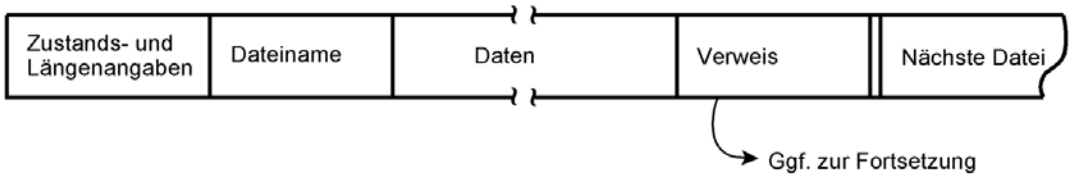


Abb. 2.46 Eine “verzeichnislose” Dateiorganisation.

Jede Datei wird als Folge von Zustands- und Längenangaben, dem Dateinamen sowie den eigentlichen Daten und einer Verweisangabe gespeichert. Eine solche Datei wird zunächst mit einer festen Länge oder mit einer Länge angelegt, die sich aus der Anwendung ergibt. Wird mehr Platz benötigt, so wird die Datei ggf. an einer anderen Position auf dem Datenträger fortgesetzt. Die Verweisangabe zeigt dann auf diese Stelle. Der wesentliche Nachteil: Dateien sind schwer zu finden, da es vorkommen kann, dass ein großer Teil der Oberfläche des Datenträgers nach dem Dateinamen abgesucht werden muss. Zudem ist dann, wenn sich Dateien in ihrer Größe laufend ändern, wenn immer wieder Dateien gelöscht und neue angelegt werden usw., mit Leistungseinbußsen zu rechnen, weil auch hierzu viele Positioniervorgänge erforderlich sind.

Eine Verbesserung

Wir sehen zunächst aufeinanderfolgende Blöcke fester Länge vor, die ähnlich Abb. 2.41 aufgebaut sind. Der Grundgedanke (Abb. 2.41): Ist die Datei kurz, so passt sie komplett in den

Block hinein[65]; ist sie länger, so speichern wir anstelle der Daten eine Verweisangabe auf eine andere Position auf dem Datenträger.

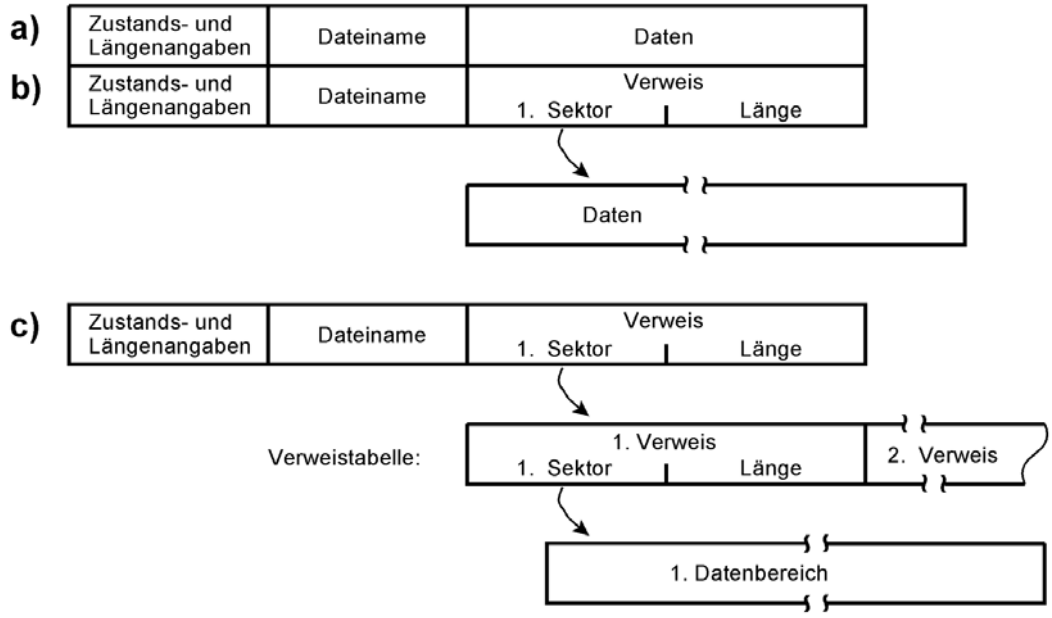


Abb. 2.47 Verbesserte Dateiorganisation.

Abb. 2.41 zeigt folgende Varianten der Speicherung von Dateien:

a) Kurze Datei. Passt komplett in den festen Block hinein.
b) Daten passen nicht in den Block. Dieser enthält statt dessen eine Verweisangabe auf den eigentlichen Datenbereich.
c) Es werden mehrere Datenbereiche benötigt. Hierzu bauen wir eine Hilfsstruktur auf, eine Verweistabelle. Die Verweisangabe im Block zeigt auf die Verweistabelle. Diese enthält ihrerseits die Verweise auf die Datenbereiche.

Dieses Prinzip – wir beginnen mit einer festen Struktur je Datei, falls diese nicht reicht, nehmen wir eine weitere Struktur hinzu, falls diese wiederum nicht reicht, noch eine usw. – liegt praktisch allen modernen Dateisystemen (Unix, NTFS usw.) zugrunde. In den Einzelheiten unterscheiden sich die Lösungen allerdings beträchtlich voneinander.

Wie kann man die Belegung eines Datenträgers beschreiben und verwalten?
Das Problem: wenn wir etwas Neues speichern wollen, müssen wir herausfinden, wo hierfür noch Platz ist. Es finden vor allem folgende Prinzipien Anwendung:

65: Die Folge dieser Blöcke ist also Verzeichnis und Daten-Behälter in Einem.

Sequentielle Speicherverwaltung
Der Datenträger wird vom Anfang bis zum Ende fortlaufend belegt. Hierfür genügt eine einzige Steuerangabe, nämlich die Nummer des ersten noch freien Sektors. Die Verzeichnisse können dann z. B. so gestaltet werden wie anhand von Tabelle 2.7 gezeigt. Anwendung: bei sequentiell und nur einmalig beschreibbaren Speichermedien (z. B. CD-R).

Speicherverwaltung über verkettete Listen
Wir sehen eine Liste vor, die für jeden Sektor des Datenträgers einen Eintrag aufweist. Ein solcher Eintrag enthält entweder eine Kennung dafür, dass der Sektor noch frei ist (z. B. den Wert Null) oder die Nummer des nächsten Sektors, der von der betreffende Datei belegt ist. Dies ist das klassische Prinzip der Datenträgerverwaltung in PCs. Die entsprechende Listenstruktur heißt FAT (= File Allocation Table). Weitere Einzelheiten in Abschnitt 2.6.4.

Freispeicherkennzeichnung mittels Bitfeld
Es wird eine Datenstruktur vorgesehen, in der jedem Sektor ein Bit zugeordnet ist. Ein gesetztes Bit kennzeichnet einen freien Sektor, ein gelöschtes einen belegten. Der Vorteil: das Auffinden freier Sektoren erfordert wesentlich weniger Zeit. (Rechenbeispiel: Speicherkapazität des Datenträgers: 10 GBytes = 20k Sektoren. Wenn wir jedem Sektor ein Bit zuordnen, so brauchen wir eine Bitliste von 20 kBits = 2560 Bytes.) Anwendung: z. B. im Dateisystem NTFS und in den Unix/Linux-Dateisystemen.

2.5.2 Dateien im Arbeitsspeicheradressraum

Der Fachbegriff: Memory Mapped Files. Damit ist nicht gemeint, dass sich die Dateien im physischen Arbeitsspeicher befinden (das wäre eine Art RAM Disk), sondern dass sie über den virtuellen Speicheradressraum zugänglich sind (Abb. 2.42). Es gibt Systemfunktionen, um Dateien zu öffnen und in den virtuellen Speicheradressraum abzubilden. Hierbei wird ein Adresszeiger zurückgegeben, mit dem sich das Anwendungsprogramm auf die einzelnen Bytes in der Datei beziehen kann. Das funktioniert so, dass die Seitentabellen so eingerichtet werden, dass der zugewiese Bereich der virtuellen Adressen auf die Zuordnungseinheiten des Datenträgers verweist, in denen die betreffende Datei untergebracht ist. Alles Weitere ist dann Sache der Verwaltung des virtuellen Speichers. Das ist dann vorteilhaft, wenn immer wieder wahlfreie Zugriffe auszuführen sind. Jeder herkömmliche wahlfreie Zugriff würde zwei Systemaufrufe erfordern (Positionierfunktion und Zugriffsfunktion). Dabei wäre zudem ein Pufferbereich zu verwalten. Für Zugriffe auf Bytes in einem Memory Mapped File genügen hingegen gewöhnliche Maschinenbefehle – es gibt also im Grunde gar keinen Verwaltungsaufwand. Befinden sich die betreffenden Bytes bereits im Arbeitsspeicher, so ist es gut, befinden sie sich auf dem Datenträger, tritt die Virtualspeicherverwaltung in Tätigkeit; das eigenliche Dateisystem hat damit gar nichts mehr zu tun.

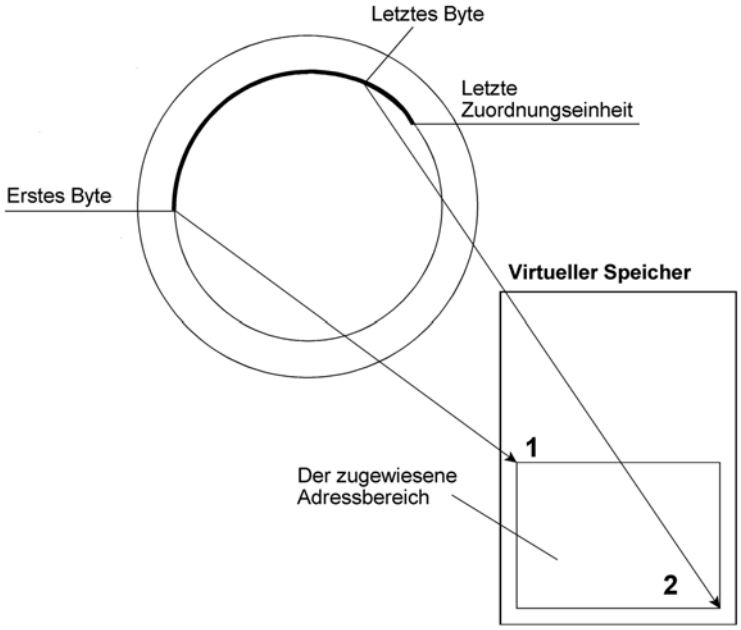


Abb. 2.48 Die Abbildung von Dateien in den Adressraum des virtuellen Speichers (Memory Mapped Files). Hierdurch werden Dateizugriffe (mit zeitaufwendigen Systemfunktionen) zu einfachen Speicherzugriffen (mit einzelnen Maschinenbefehlen)[66].

2.5.3 Dateien und Laufwerke

Das Problem: wir brauchen eine bestimmte Datei und wissen auch, wie sie heißt. Wo aber (auf welchem Laufwerk oder Datenträger) ist sie zu finden?

Die Mainframe-Lösung

Die Zuordnung ist veränderlich, und das System sorgt dafür, dass sich alles zusammenfindet. Die Anwendung kann z. B. eine Datei MAUSI anfordern (wir schenken uns hier die einschlägigen Spitzfindigkeiten) und es dem System überlassen, nachzusehen, auf welchem Datenträger sich diese Datei befindet. Hierzu werden zentrale Verzeichnisse (die sog. Kataloge) geführt. Zudem hat jeder Datenträger ein eigenes Inhaltsverzeichnis (Volume Table of Content VTOC).

Die Unix-Lösung

Kennzeichnend (und im Vergleich zu den herkömmlichen Mainframes neuartig) ist die baumförmige Verzeichnisstruktur (siehe weiterhin Abschnitt 2.5.4.). Es gibt aber auch hier keine unmittelbare Zuordnung zu den Laufwerken. Der Nutzer sieht nur seinen Verzeichnisbaum. Auf welchen Datenträgern oder Laufwerken sich die Dateien befinden, ist Sache der Plattform oder der Systemadministration. Es ist aber möglich, zusätzliche Laufwerke – die eigene Verzeichnisbäume enthalten – im System anzumelden.

66: Wenn alles günstig läuft, erfordern die Systemfunktionen viele Mikrosekunden, die Maschinenbefehe hingegen nur wenige Nanosekunden.

Die herkömmliche PC-Lösung

Es ist nicht vorgesehen, dass umfassende Kataloge verwaltet oder dass alle irgendwie in Betracht kommenden Speichereinrichtungen nach einer bestimmten Datei abgesucht werden. Vielmehr beziehen sich alle Dateiangaben stets auf ein bestimmtes Laufwerk. Jedes Laufwerk hat einen Kennbuchstaben. A und B kennzeichnen Diskettenlaufwerke, C, D usw. typischerweise Festplatten. Die Festplatten werden als logische Laufwerke verwaltet; man kann ohne weiteres auf einer physischen Festplatte mehrere logische Laufwerke einrichten (Partitionierung). Auch Zugriffe über (lokale) Netzwerke werden üblicherweise über besondere logische Laufwerke geführt (aus der Sicht des PC-Systems ist dann das Netzwerk nichts weiter als eine große und langsame Festplatte). Laufwerk-Kennbuchstaben werden durch Doppelpunkte abgeschlossen (Beispiele: A:, C: usw.). Diese Zentrierung auf das Laufwerk – mit anderen Worten - das Fehlen eines übergeordneten Katalog- oder Verzeichnissystems – ist aus der Entwicklungsgeschichte heraus zu verstehen (Kostenminimierung), ist aber im Grunde ein hässliches Relikt. Immerhin gibt man sich redlich Mühe (Tabelle 2.8).

Funktion	Bezeichnung
Unterstützung wechselbarer Datenträger	Removable Storage Manager (RSM)
Verteiltes Dateisystem (in Netzwerken). Ermöglicht Aufbau logischer Verzeichnisstrukturen und somit das Auffinden von Dateien, ohne wissen zu müssen, wo sie wirklich abgelegt sind.	Distributed File System (DFS)
Festplattenverwaltung auf logischer Ebene	Volume Manager
Zentrale Verzeichnisdienste (in Netzwerken)	Active Directory

*): Als Client im Netzwerk. DFS ist nicht im Einzel-PC nutzbar.

Tabelle 2.8 Microsoft-Lösungen zur Verbesserung der Dateiverwaltung (Auswahl).

Dateinamen

Der Dateiname sollte Klartext sein. Der Anwender möchte schließlich nicht auf Datei Nr. 32315 zugreifen, sondern auf die Beschwerdebriefe oder auf die Mahnungen. Um die Dateien finden zu können, müssen die Dateinamen mitgespeichert werden. Da ursprünglich der Speicherplatz auch auf Datenträgern knapp und die Verarbeitungsleistung der Prozessoren nicht allzu hoch war, sollten die Namen kurz sein und sich mit einfachen Suchabläufen finden lassen. Im PC-Bereich hat man deshalb das sog. 8.3-Schema gewählt. Ein Dateiname ist eine Zeichenkette aus maximal 11 Zeichen[67]. Dabei bilden bis zu acht Zeichen den eigentlichen Dateinamen. Dieser kann durch eine Erweiterung ergänzt werden, die aus einem Punkt als erstes Zeichen und maximal drei weiteren Zeichen besteht.

67: Der Punkt wird intern nicht mitgespeichert. Deshalb nur 11 Zeichen.

So kurze Dateinamen sind jedoch nicht besonders nutzerfreundlich. Deshalb unterstützen moderne Betriebssysteme auch extrem lange Dateinamen (z. B. von Windows 95 an mehr als 250 Zeichen[68]).

2.5.4 Partitions

Eine physische Festplatte lässt sich in mehrere kleinere logische unterteilen. Ein solches logisches Laufwerk heißt Partition. Aus Sicht der Anwendungen ist jede Partition ein selbständiges Laufwerk, das über einen eigenen Kennbuchstaben angesprochen wird (Abb. 2.43). Der Zweck besteht darin, den Speicherplatz auf dem Datenträger in überschaubare Bereiche aufzuteilen. Wichtig ist:

- Partitions sind zusammenhängende Bereiche auf dem Datenträger. Bei Zugriffen auf eine bestimmte Partition betreffen die zeitaufwendigen mechanischen Positioniervorgänge nur diesen Bereich, nicht aber die gesamte Oberfläche.
- Subtile Fehler im Dateisystem betreffen nur die jeweilige Partition, nicht aber die gesamte Oberfläche. Sinngemäß können übliche Sabotageprogramme nur in der jeweils adressierten Partition Schaden anrichten.
- Jede Partition lässt sich einzeln neu formatieren, wobei die Inhalter der anderen Partitions erhalten bleiben.
- Es ist es möglich, mehrere Betriebssysteme auf zu installieren. Beispielsweise lassen sich eine Windows-Partition, eine DOS-Partition (zum Testen und Fehlersuchen), eine Linux-Partition usw. einrichten. Es handelt sich allerdings – verglichen mit den virtuellen Maschinen – um eine Einfachlösung, denn man kann das jeweilige System nur während des Startablaufs auswählen.
- Um die Startpartition auszuwählen, braucht man ein Dienstprogramm, das noch vor dem Laden des Betriebssystems gestartet wird. Der Allgemeinbegriff für solche Programme: Bootmanager.

Die beschreibenden Angaben zu den Partitions sind im ersten Sektor des physischen Datenträgers untergebracht (Partition-Sektor oder Master Boot Record MBR)[69]. Der Partition-Sektor wird beim Kaltstart von einer Festplatte als erstes geladen. Ob es sich um einen gültigen Partition-Sektor handelt, erkennt das ROM-residente Kaltstartprogramm an den Erkennungsbytes (AAH, 55H), die ganz am Ende des Sektors erwartet werden. Der Sektor enthält neben den Partition-Tabellen, die Lage, Zustand usw. der einzelnen Partitions beschreiben, ein Partition-Startprogramm. Dieses wird nach dem Laden ausgeführt. Es muss die

68: Allzu lange Dateinamen (Richtwert: mehr als 50 Zeichen) sind aber auch unhandlich. Die meisten sinnvollen Begriffe haben weniger als 30 Zeichen.

69: Die Abbildung zeigt den herkömmlichen MBR der FAT16-Dateiorganisation. Neuere Systeme kommen mit einem einzigen Boot-Sektor nicht mehr aus. Deshalb werden beispielsweise bis zu 32 Sektoren für die Datenträgerbeschreibung und den Kaltstart reserviert.

Partition-Tabellen auswerten, die zu startende (aktive) Partition erkennen und das Holen des betreffenden Boot-Sektors veranlassen.

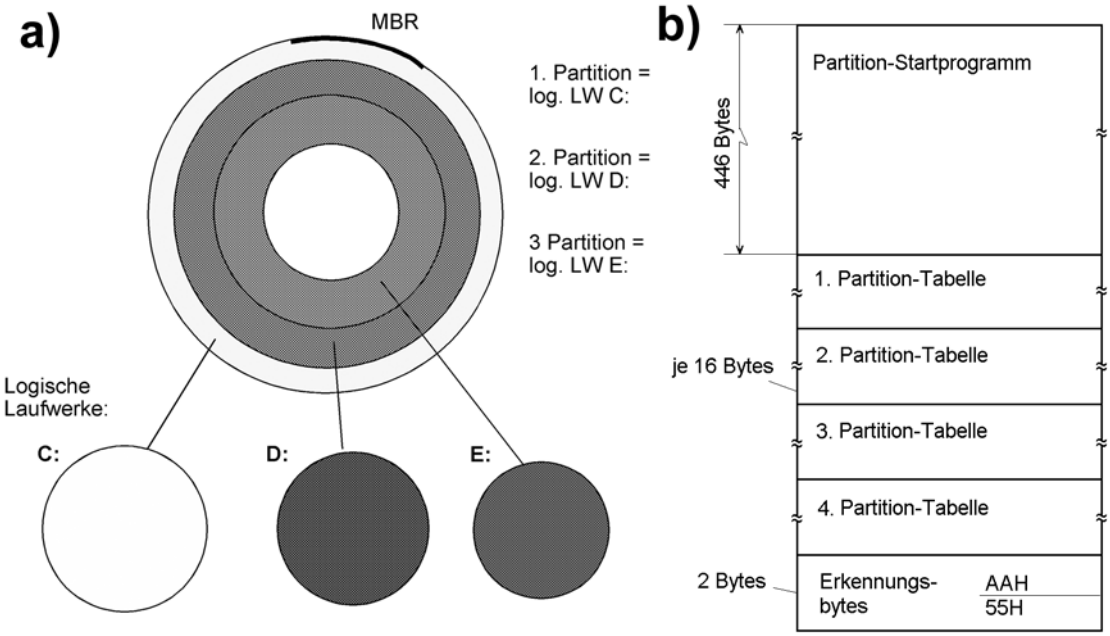


Abb. 2.49 Partitionierung. a) ein physisches Laufwerk mit drei Partitions, wobei jede ein logisches Laufwerk darstellt. b) die Struktur eines herkömmlichen Partition-Sektors (Master Boot Record MBR).

2.5.5 Installierbare Dateisysteme

Ein Plattenspeicher-Dateisystem (z. B. FAT oder NTFS) gehört zur Grundausstattung. Moderne Plattformen ermöglichen es darüber hinaus, weitere Dateisysteme zu installieren. Hiermit können den Anwendungsprogrammen vielfältige Speichereinrichtungen, aber auch weitere Dateisysteme zur Verfügung gestellt werden, die auf bestimmte Anforderungen hin optimiert sind[70]. Alle Dateisysteme sind dabei über einheitliche Anwendungsprogrammschnittstellen zugänglich.

2.5.6 Ein- und Ausgabe nach dem Dateiprinzip

Sind größere Informationsmengen ein- und auszugeben, so liegt es nahe, diese als (sequentielle) Dateien zu organisieren bzw. die an sich vorhandene Dateiorganisation für die Ein- und Ausgabe mitzunutzen. Das hat einen weiteren Vorteil: die Dateiorganisation erhält universellen Charakter; der Anwender sieht nur noch Dateien und gar keine Geräte mehr. Ein (logisches) Gerät ist vielmehr nur eine Quelle (zum Holen einer Datei) oder eine Bestimmung (um eine Datei dorthin

70: Manche Speichereinrichtungen können intern – infolge ihrer besonderen Wirkprinzipien – nicht wie Plattenspeicher (genauer: wie DASD-Einrichtungen)) organisiert werden. Das betrifft u. a. CD-ROM, CD-R, DVD usw. Andererseits versteht es sich von selbst, dass der Anwendungsprogrammierer alle Arten von Massenspeichern über eine einheitliche Schnittstelle ansprechen möchte.

zu senden) wie die anderen Ressourcen des Dateisystems (z. B. die Massenspeicher) auch. Mit anderen Worten: jeder Ein- oder Ausgabevorgang ist nichts weiter als der Transport einer Quelldatei zu einer Zieldatei (wobei es sich bei einer "Quelldatei" um einen Scanner und bei einer "Zieldatei" um einen Drucker handeln kann; die "Zieldatei" einer Eingabe bzw. die "Quelldatei" einer Ausgabe ist dann ein entsprechender Bereich im Arbeitsspeicher oder eine "echte" Datei auf der Festplatte). Um alle Einzelheiten kümmert sich das Betriebssystem.

Dateiverkettung (Pipes)
In Fortführung der Vereinheitlichung kann man Zwischendateien weglassen. Beispiel: Ein Programm A will einem Programm B Daten zukommen lassen. Herkömmlicherweise hat man dafür eine Zwischendatei d eingerichtet, wobei A Daten beispielsweise aus einer eigenen Datei a nach d schreibt und B Daten von d in seine eigene Datei b liest. Weshalb dann nicht gleich eine Verbindung von A und B herstellen, indem der Datei a die Datei b als Zieldatei zugewiesen wird? Dieses Prinzip ist mit dem Betriebssystem Unix zu weiter Verbreitung gelangt. Solche (vom Betriebssystem organisierten) Dateiverbindungen heißen Pipelines (oder kurz Pipes).

2.5.7 Die baumförmige Verzeichnisstruktur

Dieses Prinzip wurde durch das Betriebssystem Unix allgemein bekannt. Es wurde auch für die PC-Plattformen übernommen. Kennzeichnend ist die Möglichkeit, Verzeichnisse in Verzeichnissen (Unterverzeichnisse, Subdirectories) vorzusehen (Abb. 2.44). Jedes logische Laufwerk oder jeder Datenträger hat ein Wurzelverzeichnis (Root Directory). Dieses kann Unterverzeichnisse aufnehmen, die ihrerseits Unterverzeichnisse enthalten können usw.

2.5.8 Die FAT-Plattenspeicherorganisation

Die Aufgabe der Plattenspeicherorganisation besteht darin, die logische Dateiorganisation auf die physischen Sektoren der Datenträger abzubilden. Dafür sind besondere Datenstrukturen notwendig, die ihrerseits auf dem jeweiligen Datenträger mit untergebracht werden müssen. Die besondere Schwierigkeit besteht darin, mit Dateien zurechtzukommen, deren Größe sich fortlaufend ändert und trotzdem die verfügbare Speicherkapazität gut auszunutzen.

Die Aufteilung des Datenträgers
Ein Datenträger (Volume) ist entweder eine Diskette, eine ganze Festplatte oder ein Teilbereich (Partition) einer Festplatte. Es leuchtet ein, dass man für die beschreibenden Angaben, die zur Umsetzung von logischen Dateien in physische Sektoren benötigt werden, irgendeinen festen Punkt braucht, an dem die Systemsoftware ansetzen kann. Hierzu ist der Anfang eines Datenträgers fest aufgeteilt (Abb. 2.45).

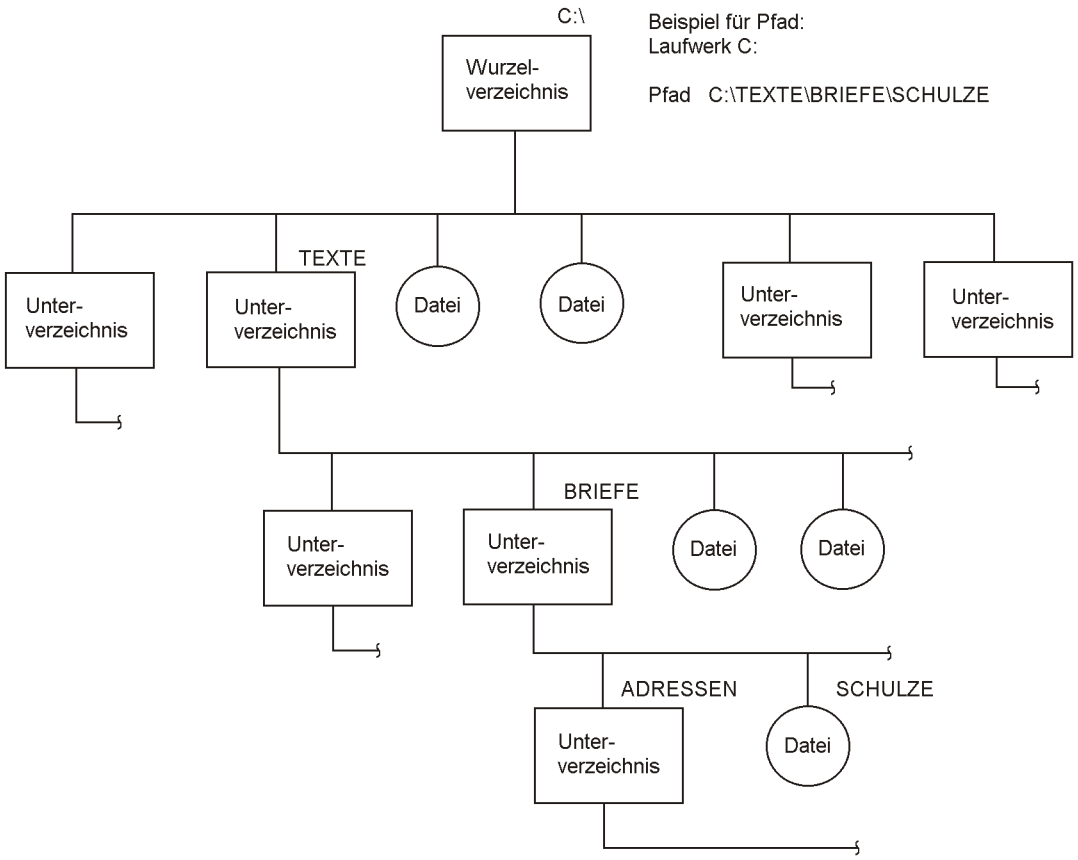


Abb. 2.50 Die baumförmige Verzeichnisstruktur (Beispiel).

Der Boot-Sektor

Das ist der erste Sektor eines jeden logischen Laufwerks[71)]. Der Sektor (Abb. 2.46) enthält am Anfang die grundlegenden beschreibenden Angaben der Plattenspeicherorganisation. Der Rest steht für die Kaltstart-Routine zur Verfügung.

Die Dateizuordnungstabelle (File Allocation Table FAT)

Diese Tabelle enthält die eigentliche Datei-Sektor-Abbildung. Machen wir uns die Wirkungsweise zunächst am einfachsten Fall klar. Insgesamt seien n Sektoren zur Datenspeicherung nutzbar. Demgemäß sehen wir in der FAT n Einträge vor. Jeder Eintrag hat eine feste Länge, beispielsweise von zwei Bytes. Die Einträge sind genau so hintereinander angeordnet, wie die n Sektoren durchnumeriert sind. Anfänglich, wenn noch nichts gespeichert ist, werden die FAT-Einträge gelöscht. Ist ein FAT-Eintrag gleich Null, so bedeutet dies, dass der betreffende Sektor frei ist. Wollen wir eine neue Datei speichern, so suchen wir in der FAT nach dem ersten freien Sektor. Dessen Nummer vermerken wir im betreffenden Verzeichnis, so

71: Der allererste Sektor einer physischen Festplatte ist der Partition-Sektor (vgl. Abb. 2.43). Er beschreibt die Einteilung des Datenträgers in logische Laufwerke.

dass wir, wenn wir später mit dem Dateinamen in das Verzeichnis gehen, dort den ersten Sektor und damit auch den ersten FAT-Eintrag finden. Damit können wir schon mal 512 Bytes der Datei unterbringen.

Was tun, wenn wir mehr brauchen? – Wir suchen einfach den nächsten freien Sektor, belegen ihn und speichern dessen Nummer im FAT-Eintrag des ersten zugeordneten Sektors. Anders ausgedrückt: der jeweilige FAT-Eintrag kennzeichnet den jeweils nachfolgenden, von der betreffenden Datei belegten Sektor. So sind alle Sektoren der Datei wie an einer Kette hängend untereinander verbunden (Abb. 2.47). Der FAT-Eintrag, der dem letzten Sektor der Datei entspricht, enthält einen Sonderwert (z. B. alle 16 Bits auf Eins gesetzt), der das Ende kennzeichnet. Es ist ersichtlich, dass man so eine Datei grundsätzlich über beliebige Sektoren verteilt speichern kann.

Längen von FAT-Einträgen
Herkömmliche Längen sind 12 Bits (für Disketten) und 16 Bits. Dieses System (FAT16) hat die Nutzung von Festplatten in PCs über viele Jahre maßgeblich bestimmt. Die Grenzen dieses Systems waren Anlass, ein neues einzuführen: FAT32 mit 32 Bits langen Einträgen.

Verzeichnisse (Directories)
Die Verzeichnisse enthalten alle Dateinamen eines Laufwerkes. Verzeichnisse können neben Dateinamen ihrerseits auch Namen weiterer Verzeichnisse enthalten. Abb. 2.48 veranschaulicht die Struktur eines Verzeichniseintrags.

Wurzelverzeichnis und Unterverzeichnisse
Das Wurzelverzeichnis (in der Microsoft-Literatur auch: Stammverzeichnis) hat herkömmlicherweise eine vom jeweiligen Datenträger-Typ abhängige feste Anzahl von Einträgen. Hingegen können Unterverzeichnisse (sie werden intern wie jede andere Datei angesehen) beliebig viele Einträge haben und auch während des Betriebs ihre Größe ändern.

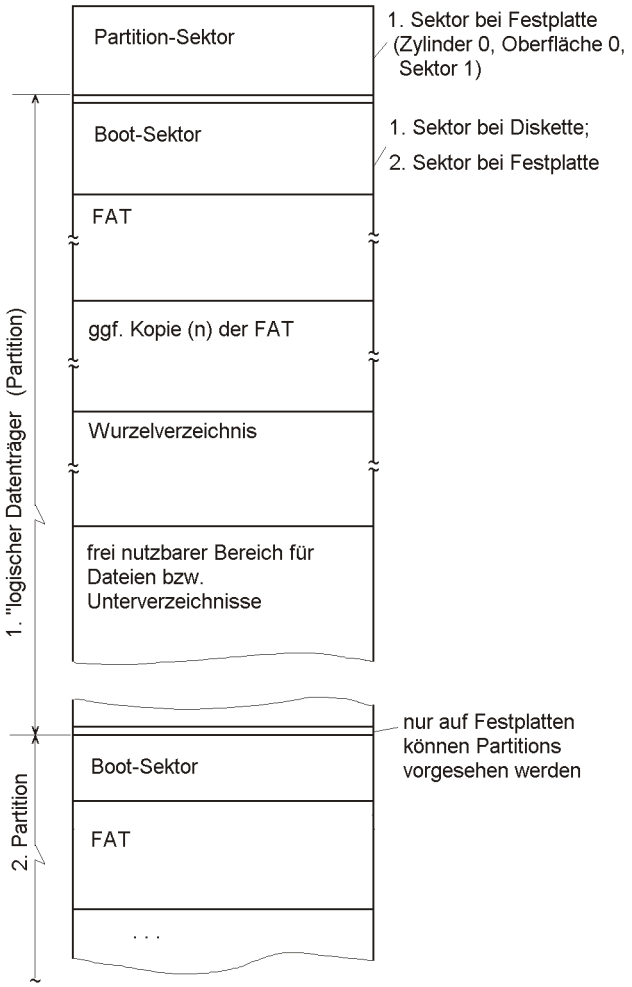


Abb. 2.51 Aufteilung eines (physischen) Datenträgers.

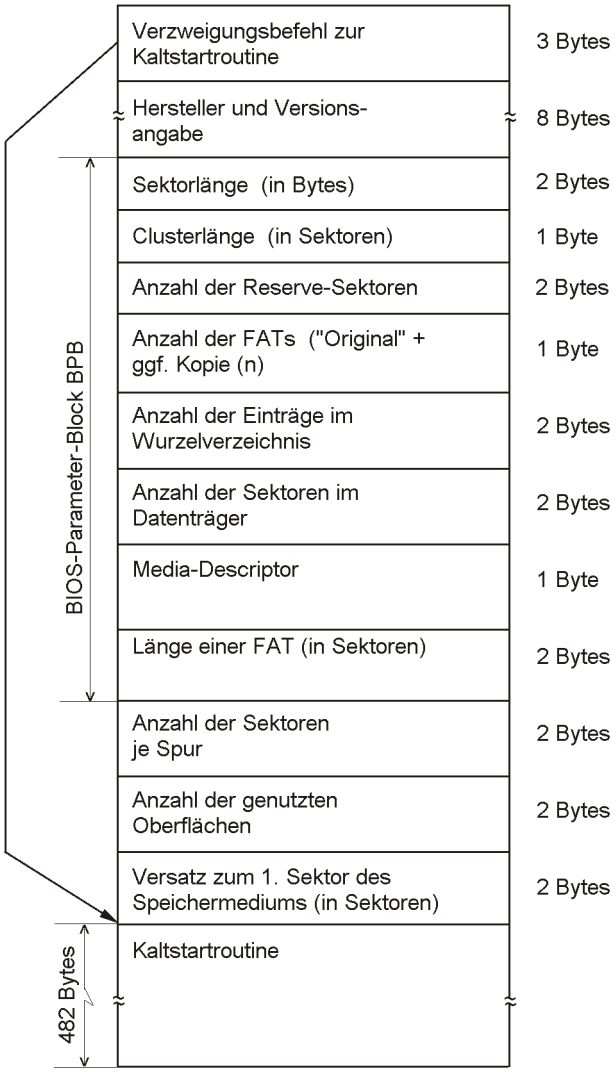


Abb. 2.52 Der Boot-Sektor.

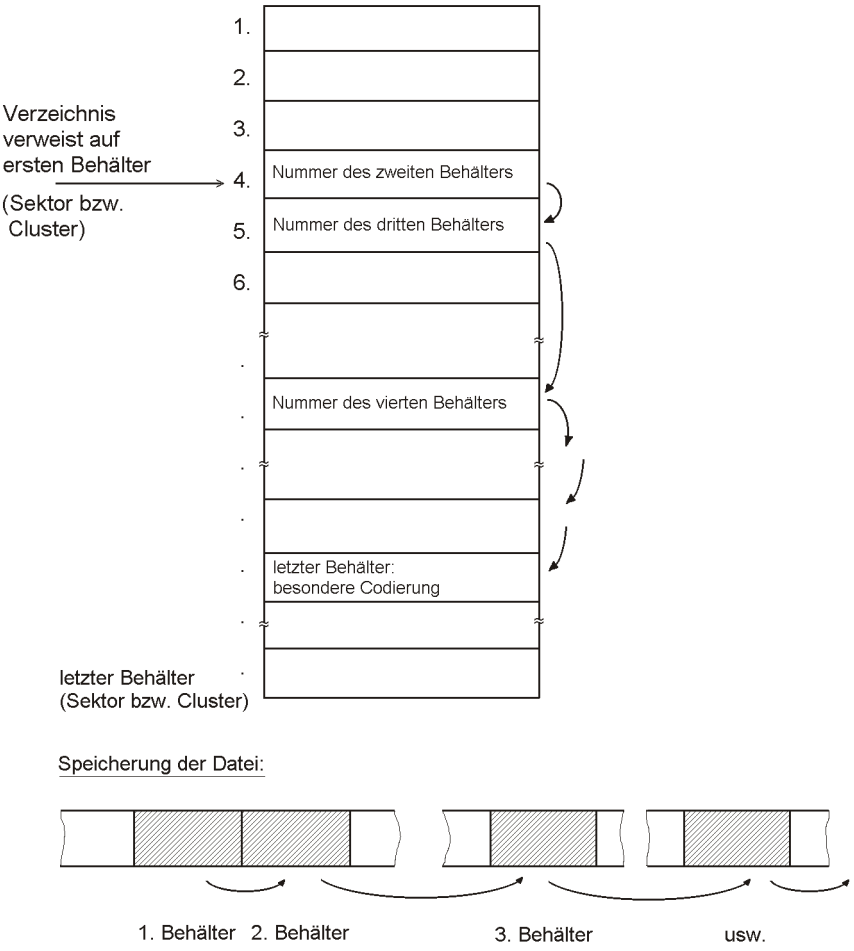


Abb. 2.53 Dateizuordnungstabelle (FAT): Organisations- und Wirkprinzip.

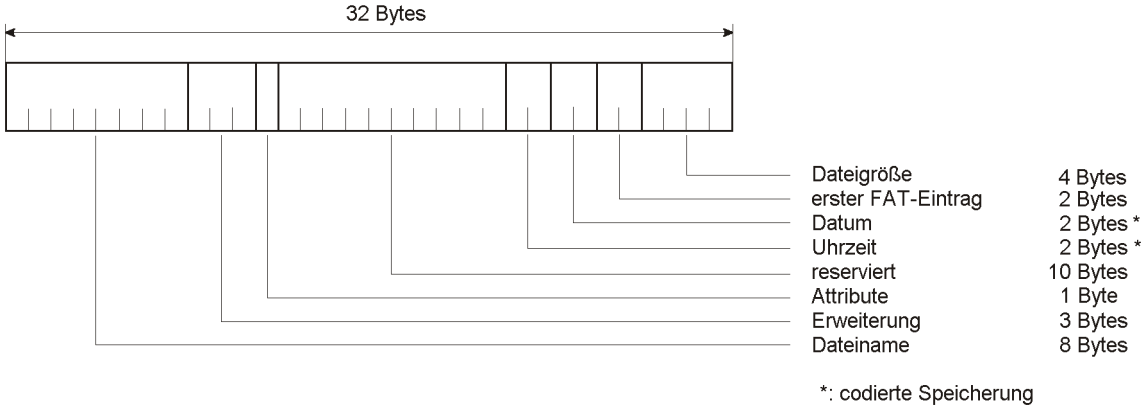


Abb. 2.54 FAT-Verzeichniseintrag.

2.5.9 NTFS

NTFS = New Technology File System (Dateisystem neuer Technologie). NTFS wurde als Teil von Windows NT entwickelt.

Dateibeschreibung

Alle Strukturen, die der Dateibeschreibung dienen, sind ihrerseits Dateien[72]. Solche Dateien heißen Metadata Files[73]. Die Dateinamen solcher Dateien beginnen mit einem Dollarzeichen ($).

MFT - die Datei aller Dateien

Am Anfang jedes logischen Datenträgers steht eine Datei, die MFT (Master File Table) genannt wird. Sie enthält für jede Datei einen Eintrag mit einer festen Länge von 1 kBytes (Abb. 2.49 bis 2.51).

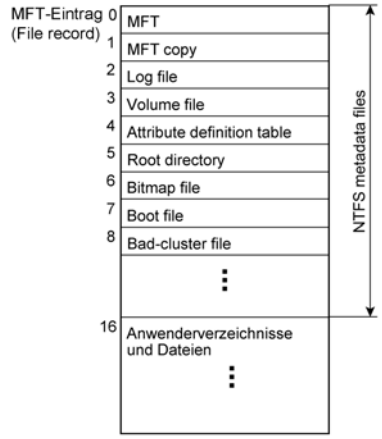


Abb. 2.55 Belegung der MFT (nach Microsoft).

File 0, 1, usw. bezeichnen MFT-Einträge (zu jeweils 1 kBytes). Jeder Eintrag kann zunächst als eine Art Verzeichniseintrag gesehen werden, der die Datei näher beschreibt (Dateiname, Position, Größe usw.). File 0 beschreibt die MFT selbst, File 1 ist eine zu Sicherungszwecken angelegte Kopie. File 6 (Bitmap File) enthält die Bitkette, die kennzeichnet, welche Cluster des Datenträgers belegt und welche frei sind. File 7 (Boot File) enthält das Kaltstartprogramm.

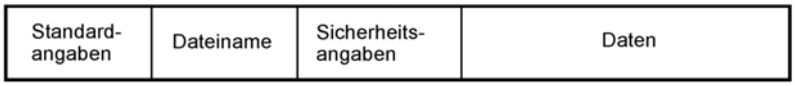


Abb. 2.56 Die Grundstruktur eines MFT-Eintrags.

72: In den FAT-Dateisystemen sind Verzeichnisse und FATs keine Dateien (wer ein Verzeichnis als Datei behandeln möchte, muss es sich vom System in eine Datei kopieren lassen).

73: Zur Etymologie: “Meta” (aus dem Griechischen) wird hier im Sinne von “über etwas” verwendet (eine Metasprache ist eine Sprache, in der eine andere Sprache beschrieben wird, Metadaten sind “Daten über Daten”, also Daten, die Daten beschreiben).

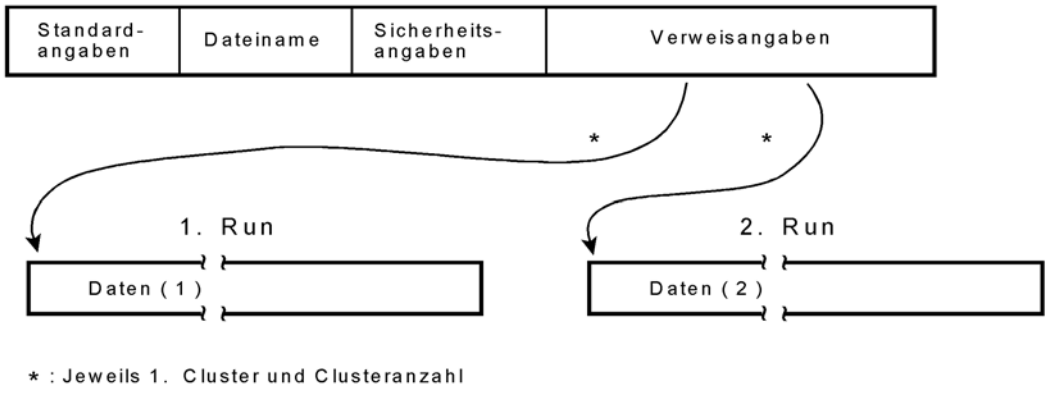


Abb. 2.57 Grundstruktur einer größeren Datei.

Die Struktur entspricht zunächst der eines Verzeichniseintrags mit Dateinamen und verschiedenen Zusatzangaben (Dateiart, Zugriffsberechtigungen, Zeitangaben usw. (Standard- und Sicherheitsangaben)). Sie enthält aber auch noch Daten. Hier hat man das anhand von Abb. 2.40 erläuterte Prinzip verwirklicht: passen alle Daten der Datei in diesen Bereich, so hat es sich gleichsam erledigt; der MFT-Eintrag *ist* die Datei. Passen sie nicht hinein, so belegt man weitere Bereiche auf dem Datenträger (die sog. Runs), welche die eigentlichen Daten enthalten und hinterlegt im Datenfeld des MFT-Eintrags entsprechende Zeiger- bzw. Verweisangaben.

Wenn es mehrere Runs sind
Sofern die Verweisangaben auf die Runs in den Datenbereich des MFT-Eintrags passen, hat es sich erledigt. Passen sie nicht hinein, so nutzt man einen weiteren MFT-Eintrag. (Das erspart die in Abb. 2.40 vorgesehene Verweistabelle.)

Verzeichnisse
Verzeichnisse sind Dateien, die Listen von Dateinamen und Verweise enthalten, um die Dateien schnell auffinden zu können. Sehr kurze Verzeichnisse passen komplett in den Datenbereich des jeweiligen MFT-Eintrags, längere werden in Runs gespeichert (wie in Abb. 2.51 veranschaulicht). Das System unterstützt die übliche baumförmige Verzeichnisstruktur.

2.6 Transaktionsorientierung

Eine Transaktion ist ein in sich abgeschlossener Informationsverarbeitungsvorgang, in dessen Verlauf Dateien verändert werden. Als Beispiel mag eine Abbuchung von einem Konto dienen. Der Kontostand ist aus der Datei zu lesen, um den abzubuchenden Betrag zu vermindern und wieder zurückzuschreiben, wobei zusätzlich die Umstände der Abbuchung (Datum, Uhrzeit, Kundennummer usw.) zu protokollieren sind.

Nun kann man hierfür ein Programm schreiben, das ohne viel Federlesen auf die betreffenden Dateien zugreift. Das Problem: was passiert, wenn etwas schiefgeht?

1. Beispiel: der abzubuchende Betrag wurde von Hand eingegeben und der Betreffende hat sich vertippt. Das wird aber erst später bemerkt. Wie will man jetzt wissen, wie hoch der bisherige Kontostand war?

2. Beispiel: mitten im Verarbeitungsvorgang ist etwas abgestürzt.

3. Beispiel: das Programm enthält einen tückischen Programmierfehler. Er tritt zwar nur sehr selten auf, aber wenn, dann werden Kundendaten verändert oder Beträge in astronomischer Höhe gebucht (alles schon vorgekommen...).

4. Beispiel: die Daten sind nur über ein Netzwerk zugänglich. Andere können sie mitlesen oder sogar verändern.

Die transaktionsorientierte Verarbeitung sieht nun Maßnahmen vor, die solche Probleme vermeiden oder die es wenigstens ermöglichen, im Fall des Falles einen Ausweg zu finden. Hierzu einige Stichworte:

- Kontrolle von Zugangsberechtigungen,
- Plausibilitätsprüfungen,
- kein pauschaler Zugriff auf ganze Dateien,
- Protokollierung aller Vorgänge (durch Führen eines sog. Journals), so dass es möglich ist, alle Vorgänge zurückzuverfolgen und vorhergehende Zustände wieder herzustellen.

Eine Transaktion gilt erst dann als abgeschlossen, wenn alles, was geprüft werden kann, auch tatsächlich geprüft und als in Ordnung befunden wurde. So ist eine Überweisung erst dann beendet, wenn das Geld tatsächlich auf dem Konto des Überweisungsempfängers verbucht werden konnte. Es darf schließlich nicht vorkommen, dass, wenn mittendrin einer der beteiligten Computer abstürzt oder wenn eine Datenübertragung zusammenbricht, das Geld einfach verschwindet.

Offensichtlich sind solche Vorkehrungen immer dann erforderlich, wenn es ums Geld geht (oder um andere Daten, bei denen die Gemütlichkeit aufhört, z. B. im Gesundheitswesen). Es ist leicht einzusehen, dass es keinen Sinn hat, transaktionsorientierte Software auf Plattformen laufen zu lassen, deren Sicherheitsvorkehrungen unzulänglich sind (mit anderen Worten, auf üblichen Arbeitsplatzsystemen).

2.7 Ablaufautomatisierung

Der Computer ist an sich als Apparat zum Automatisieren von Informationsverarbeitungsvorgängen gedacht, vergleichbar einer Maschine, die auf einen Knopfdruck hin vollkommen selbsttätig ein Werkstück nach dem anderen ausstößt. Nun leistet ein Programm, das wir zu einem bestimmten Zweck schreiben, durchaus das Gewünschte – nur ist eben das Programmieren eine Kunst für sich. Man hat deshalb schon vor Jahrzehnten

Vorkehrungen geschaffen, um den Computer zu veranlassen, einfache Abläufe selbsttätig auszuführen, ohne hierzu "richtige" Programme schreiben zu müssen. Als Beispiel möge folgender Vorgang dienen:

- die Abrechnungsdaten aus allen Abteilungen sind in eine einzige Datei zu kopieren,
- die Löhne der Beschäftigten sind zu berechnen,
- die Lohnzettel sind zu drucken,
- die Beträge sind zu überweisen,
- von den betreffenden Daten sind Sicherungskopien zu erstellen,
- für die Betriebsführung sind statistische Analysen durchzuführen,
- die Analysedaten sind grafisch darzustellen, und zwar so, dass auch Vorgesetzte und sonstige Laien damit etwas anfangen können.

Für jede dieser Aufgaben gibt es ein Programm. Es handelt sich nun darum, die einzelnen Programme nacheinander ablaufen zu lassen. Der Fachbegriff: Stapelverarbeitung (Batch Processing). Hierfür hat man sog. Script- oder Jobsteuersprachen geschaffen – an sich zwar einfache Programmiersprachen[74], aber teils mit abschreckendem Erscheinungsbild.

Ablaufautomatisierung im PC-Bereich (Auswahl)

DOS
DOS kann sog. Stapelverarbeitungsdateien (Batch-Dateien) auswerten. Diese haben das Dateikennzeichnen .BAT (die wohl wichtigste dieser Dateien: AUTOEXEC.BAT). Batch-Dateien sind ASCII-Textdateien, die Batch-Kommandos enthalten. Mehr als 100 verschiedene Anweisungen können in Batch-Dateien aufgenommen werden (sowohl Batch- als auch DOS-Kommandos). Damit kann man schon etwas anstellen (u. a. das obige Beispiel erledigen). Batch-Dateien sind ein einfaches Mittel, DOS-Anwendungsumgebungen für die heutzutage sinnvollen Einsatzbereiche einzurichten[75] (Datenwiederherstellung, Testen und Prüfen, maschinennahe Programmierung usw.).

Windows 3.x
Die Batch-Dateien des DOS wurden noch unterstützt, zu Windows selbst gab es aber nur einen einfachen sog. Macrorecorder, ein Programm, das Folgen von Bedienvorgängen aufzeichnen und wiederholt ablaufen lassen kann.

Windows 95
Es wurde gar nichts geliefert.

74: Es sind im Grunde nur simple Anweisungsfolgen zu definieren. Oftmals genügt eine einfache Aufzählung der Form "tue erstens das, dann zweitens jenes usw.".

75: So dass die DOS-Kommandozeile gar nicht erscheint. Es kann dann beispielsweise ein Auswahlmenü dargestellt oder die Anwendung sofort gestartet werden.

Windows NT
Nur für bestimmte Vorgänge werden Batch-Dateien unterstützt (z. B. zum Anmelden von Workstations (im Netzwerk)). Eine Jobsteuerung ist nicht vorgesehen.

Windows Scripting Host (WSH)
WSH ist anfänglich Windows 98 und Windows 2000 entwickelt worden, kann aber auch in Windows 95 und Windows NT 4.0 genutzt werden[76]. WSH unterstützt echte Scriptdateien, wobei zwei Script-Sprachen zur Wahl stehen:

- VBScript auf Grundlage von Visual Basic,
- JScript auf Grundlage von JavaScript.

Weitere Skriptsprachen (wie beispielsweise TCL, Perl oder REXX können bedarfsweise eingebunden werden. Alternativ zum WSH kann man Scripting-Software von Drittanbietern einsetzen.

Unix/Linux
Es gibt sog. Shell Scripts. Das sind Textdateien, die Unix-Kommandos enthalten. Die Shell Scripts entsprechen den Batch-Dateien des DOS. Zudem gibt es verschiedene höherentwickelte Script-Sprachen (von besonderer Bedeutung: die Sprache Perl).

2.8 Benutzeroberflächen

Benutzeroberfläche (User Interface) ist der Sammelbegriff für die Prinzipien der Bedienung und Anzeige (im Englischen bildhaft mit Look and Feel bezeichnet). In diesem allgemeinen Sinne haben auch Systeme wie DOS oder Unix Benutzeroberflächen, wenngleich nicht sonderlich einladende.

Eine Benutzeroberfläche sollte wenigstens folgendes gewährleisten:

- die Bedienung durch Funktionsauswahl,
- die Eingabe von Parametern, Daten, Texten usw.
- die Anzeige von Resultaten,
- die Bedienerführung und Unterstützung (Prompting, Bedienhilfe).

Die Benutzeroberfläche der Systemplattform ermöglicht den Zugang auf die vielfältigen Dienstfunktionen: Programmstart, Konfigurationssteuerung, Leistungsmessung, Systemverwaltung (Administration) usw. Am häufigsten werden aber die Funktionen der

76: Ein Einsatzfall in der heutigen Zeit: Ein älteres System soll mit einer speziellen Anwendung weitergenutzt werden. Mit einem Skript kann das Starten der Anwendung, die Datensicherung usw. so organisiert werden, dass der Nutzer sich nicht um die Eigentümlichkeiten der Bedienung des alten Systems kümmern muss.

Dateiverwaltung genutzt (Dateien anzeigen, kopieren, umbenennen, löschen usw.). In modernene Systemen bemüht man sich um einen nahtlosen Übergang zwischen System- und Anwendungsbedienung (einheitliches Look and Feel).

2.8.1 Kommandosprachen

Für jede Funktion wird eine bestimmte Zeichenkette vorgesehen, die man eintippen muss. Typischerweise schließt das Betätigen der Enter-Taste die Eingabe ab und veranlasst, dass die Zeichenkette vom Programm analysiert wird.

Die Vorteile:

- Die Anforderungen an die Hardware sind gering.
- Man kommt mit einfachsten Bedien- und Anzeigemittel aus (Bedienkonsole oder Terminal mit Textanzeige und Tastatur)[77]. In manche Fällen kann man sogar auf die Anzeige verzichten.
- Die Fernbedienung kostet nur wenig Aufwand (z. B. Eingabe der Kommandos über eine serielle Schnittstelle).
- Die Ablaufautomatisierung bereitet keine besonderen Schwierigkeiten – man braucht eigentlich nur ein Programm, dass das Eingeben der Zeichenketten emuliert (einfachste Form des Scripting).

Die Nachteile:

- Man muss die Vielzahl der Kommandos auswendig wissen oder immer wieder nachschlagen.
- Das Eintippen der vergleichsweise vielen Zeichen kostet Zeit.

Konsolen und Terminals

Aufgrund der genannten Vorteile ist das Prinzip der Kommandosprache keineswegs als grundsätzlich veraltet anzusehen. Zudem stellt in manchen Kreisen die Tatsache, dass man sich mit den Kommandos auskennen muss, überhaupt kein Problem dar. Im Gegenteil – mit solchen Kenntnissen kann man sich als ein Art Eingeweihter sehen, der dem Volk der gewöhnlichen Nutzer turmhoch überlegen ist. Deshalb ist die sog. Unix-Shell immer noch so beliebt wie vor 40 Jahren[78] ...

77: Vor dem Aufkommen der Bildschirmterminals – also in den 50er und 60er Jahren – hatte man als Bediengeräte elektrische Schreibmaschinen oder Fernschreiber eingesetzt.

78: Zu diesem überlegenen Wissen gehört u. a., dass cp Kopieren heißt und rm Löschen, während ls Verzeichnisse anzeigt (das ist aber längst noch nicht alles; es gibt auch noch awk, grep, fsck, nroff usw.). Würde sich die Automobilindustrie so verhalten wie die Unix/Linux-Szene, so könnten heute noch alle Autos mittels Handkurbel angeworfen werden, über das richtige Ankurbeln würden dicke Bücher geschrieben werden, und es gäbe Fanatiker, die für andere Verfahren, einen Verbrennungsmotor in Gang zu setzen, nur ein geringschätziges Lächeln übrig hätten.

Es gibt aber auch sachliche Gründe, gelegentlich Einfachlösungen der Bedienung und Anzeige zu bevorzugen. Man möchte eine moderne Systemplattform nutzen, dabei aber nicht den Ballast einer universellen grafischen Oberfläche mitschleppen[79].

Hierfür gibt es Programmentwicklungsumgebungen, die es ermöglichen, sog. Konsolanwendungen (Console Applications) zu entwickeln (z. B. Delphi, C++ Builder und Power Basic). Auf dem Windows-Bildschirm sieht eine Konsolanwendung zunächst[80] wie ein altmodisches DOS-Fenster aus, das dahinter stehende Programm ist aber eine richtige Windows-Anwendung (Abb. 2.52 und 2.53). Auch der Macintosh unterstützt Konsolen und Terminals (Abb. 2.54).

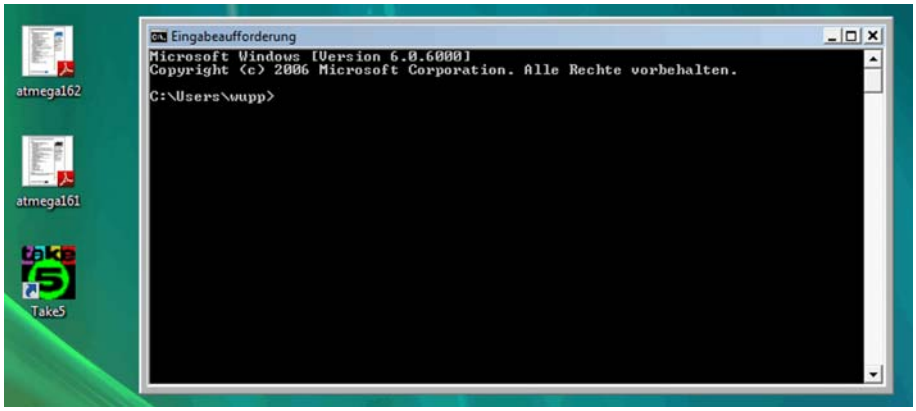


Abb. 2.58 Die Eingabeaufforderung von Windows. Hier können Kommandos eingegeben werden, die Systemfunktionen auslösen.

79: Hinsichtlich Programmieraufwand, Speicherplatzbedarf und Anforderungen an die Bedien- und Anzeigehardware. Typische Einsatzfälle: industrielle und messtechnische Anwendungen, Bearbeitung mathematischer Probleme (wo viel zu rechnen,aber wenig ein- und auszugeben ist), Testprogramme, Systemwiederherstellung.

80: Man kann es aber auch gefälliger gestalten (vgl. Abb. 2.55). Durch Nutzung der Windows-Anwendungsprogammschnittstellen (APIs) können zudem beliebige grafische Darstellungen ausgegeben werden. Die Entwicklungssysteme enthalten einschlägige Demonstrations-programme.

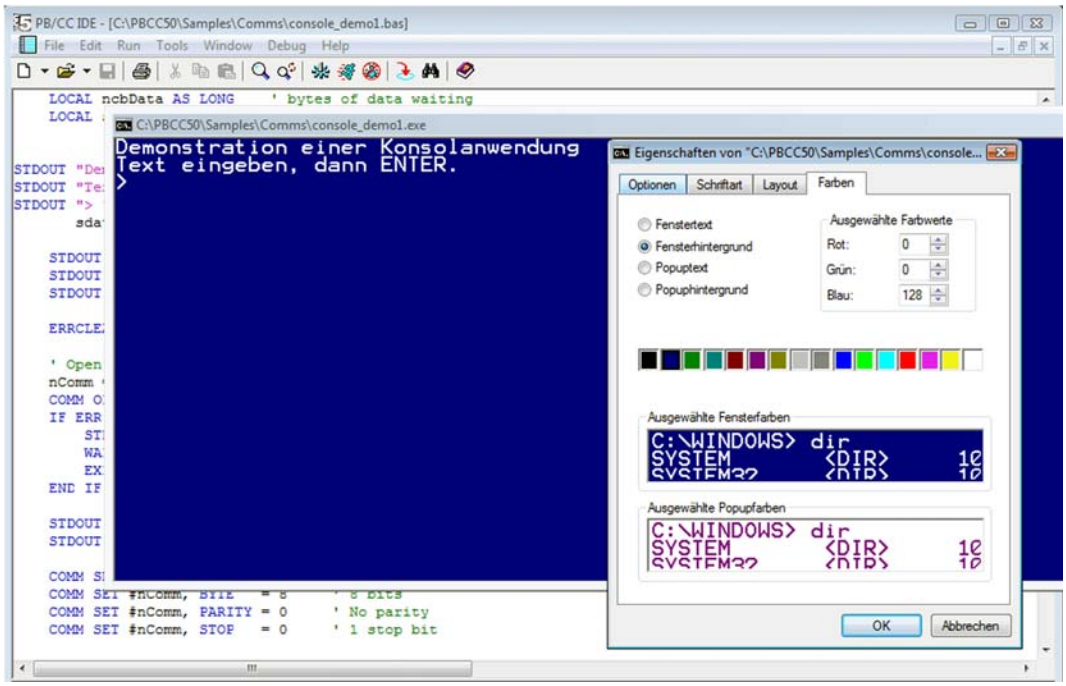


Abb. 2.59 Eine Konsolanwendung unter Windows. Hier während der Entwicklung[81]. Der Dialog "Eigenschaften" gehört zum System, nicht zur Anwendung. Offensichtlich ist es möglich, auch eine solche Darstellung so zu gestalten, dass sie nach etwas aussieht[82].

2.8.2 Auswahlbilder (Menüs)

Die Funktionsauswahl ist nach dem Prinzip der Speisekarte[83] organisiert. Die auswählbaren Funktionen werden auf dem Bildschirm dargestellt. Zur Auswahl werden sie entweder mittels Cursor oder Maus angefahren oder sie sind mit Kennzeichen versehen, z. B. mit einer laufenden Nummer, die man eintippen muss. Passt der gesamte Funktionsvorrat nicht auf einen Bildschirm, werden Untermenüs vorgesehen, die ihrerseits vom Hauptmenü aus anwählbar sind (vergleichbar einer mehrseitigen Speisekarte mit Inhaltsverzeichnis).

81: Im Beispiel mit dem Power Basic Console Compiler. Man kann sehr anspruchsvolle Anwendungen schreiben, ohne sich um die typischen Fenster und Steuerelemente kümmern zu müssen – und auch nicht um die Eigentümlichkeiten der Windows-Programmierung. Im Grunde kann man auf herkömmliche Weise drauflos programmieren. Da man alle Windows-Funktionen aufrufen kann, sind der grafischen Gestaltung keine Grenzen gesetzt.

82: Das gilt auch für die Eingabeaufforderung gemäß Abb. 2.52.

83: Engl. Menue.

Die Vorteile:

- Übersichtliche Darstellung.
- Einfache Bedienung durch direktes Auswählen.

Die Nachteile:

- Schlechter dokumentierbar. In eine Kommandosprache kann man sich mit dem Buch in der Hand einarbeiten, in ein rein Menü-orientiertes System praktisch nur am Computer.
- Bei ineinandergeschachtelten Menüs bringt Erfahrung keinen Vorteil, da man sich typischerweise stets vom Hauptmenü bis zum jeweiligen Untermenü vorarbeiten muss (nicht alle Systeme bieten für erfahrene Nutzer entsprechende Abkürzungen (Shortcuts) an).

Die Abb. 2.55 bis 2.57 zeigen Beispiele von Menüdarstellungen. Diese Form der Bedienung hat sich bei den meisten Softwareprodukten durchgesetzt.

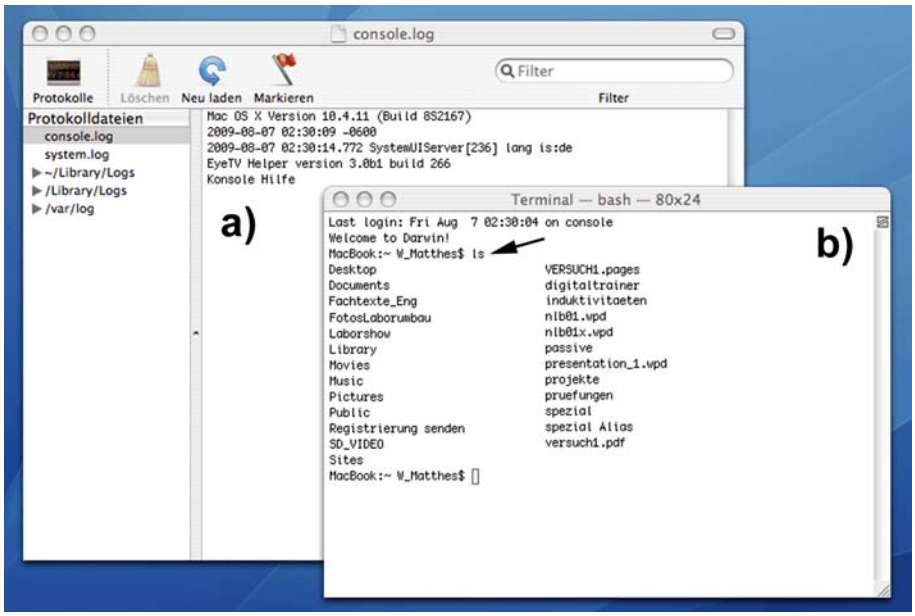


Abb. 2.60 Auf dem Bildschirm eines Macintosh. Die neueren Betriebssysteme sind im Grunde Unix-Varianten mit einer eigenen grafischen Bedienoberfläche. a) die Konsole dient zum Anzeigen von Protokolldateien. b) das Terminalfenster ist das Gegenstück zur Windows-Eingabeaufforderung. Es bildet ein herkömmliches Unix-Bildschirmterminal nach[84)]. Hier wurde probeweise das Kommando ls eingetippt (Pfeil). Im Ergebnis wurde das aktuelle Verzeichnis angezeigt.

84: 80x24 bedeutet, dass 24 Zeilen zu je 80 Zeichen dargestellt werden können – das ist das Anzeigevermögen eines typischen Bildschirmterminals der 70er Jahre. “bash” steht für die Unterstützung der sog. Bash-Shell.

```
MAIN                              AS/400 Main Menu
                                                          System:   NEPAS4
Select one of the following:

     1. User tasks
     2. Office tasks
     3. General System Tasks
     4. Files, libraries, and folders
     5. Programming
     6. Communications
     7. Define or change the system
     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. PC Support tasks

    90. Sign off

Selection or command
===> 75

F3=Exit    F4=Prompt    F9=Retrieve    F12=Cancel    F13=User support
F23=Set initial menu
Specified menu selection is not correct.
```

Abb. 2.61 Ein bescheiden aussehendes, aber sehr hoch entwickeltes und umfangreiches Menüsystem: die herkömmliche Bedienoberfläche des AS/400. Das Menüsystem wird durch eine Kommandosprache ergänzt.

Abb. 2.62 Einfache Oberflächen zum Starten von Testprogrammen. a) Funktionsauswahl über Funktionstasten und einzutippende Kennungen (eigentlich eine ganz einfache Kommandosprache; das Menü ist im Grunde nur eine Kurzerklärung der Kommandos). b) Funktionsauswahl durch Ansteuern der Menüpunkte mittels Cursortasten oder Maus.

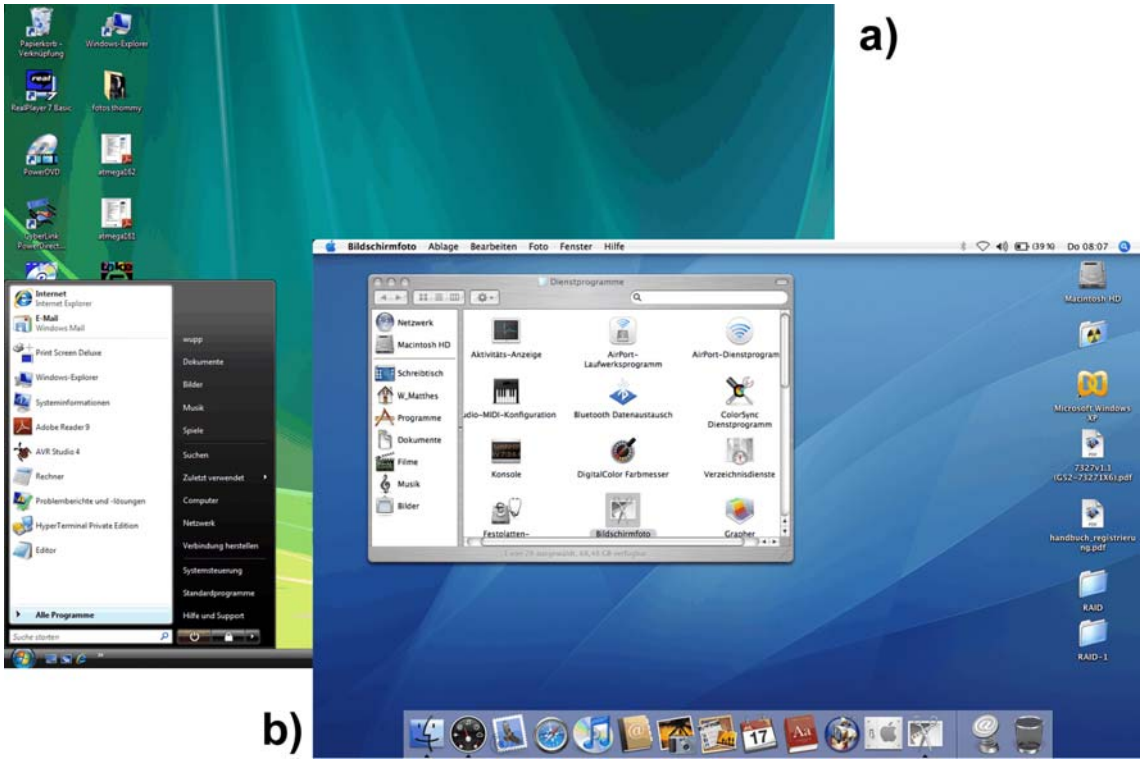


Abb. 2.63 Programmstart über grafische Bedienoberflächen. a) Windows; b) Macintosh. Diese bunten Darstellungen leisten im Grunde auch nichts Anderes als die in den Abb. 2.55 und 2.56 gezeigten Menüs ...

2.8.3 Grafische Benutzeroberflächen

Eine alphanumerische (zeichenorientierte) Benutzeroberfläche ist auf Zeichendarstellungen und auf die Eingabe von Zeichen oder auf die Funktionsauswahl über Funktions- und Cursortasten beschränkt. Hingegen können im Rahmen einer grafischen Benutzeroberfläche (Graphical User Interface GUI) beliebige Bildschirm-Darstellungen und zur Funktionsauswahl beliebige Einrichtungen zum Deuten (Maus, Rollkugel, Digitalisiertablett usw.) verwendet werden.

Standardbedienhandlungen, Standarddialoge

Dass es so etwas gibt, ist einer der wirklichen Vorteile der weit verbreiteten Benutzeroberflächen (z. B. von Windows): es erspart dem Nutzer, sich von Anwendung zu Anwendung an jedesmal neue Bedienhandlungen gewöhnen zu müssen. Die Systemanbieter (z. B. Microsoft) unterstützen dies, indem sie einschlägige Programmierrichtlinien herausgeben. Die Abbildungen 2.58 und 2.59 zeigen typische Standarddialoge und Windows-Steuerelemente. Wichtig ist, dass diese Elemente über die Windows-Anwendungsprogrammierschnittstelle (API) aufgerufen werden und dass Wirkung und Aussehen voneinander getrennt sind; ein Fenster mit Rollbalken bleibt ein Fenster mit Rollbalken, gleichgültig ob es spitze oder runde Ecken hat, ob der Rand grau oder blau eingefärbt ist oder ob er wie gebürstetes Aluminium aussieht. Mit integrierten Programmentwicklungsumgebungen kann man solche Dialoge und Steuerelemente nahezu mühelos in eigene Programme einbauen.

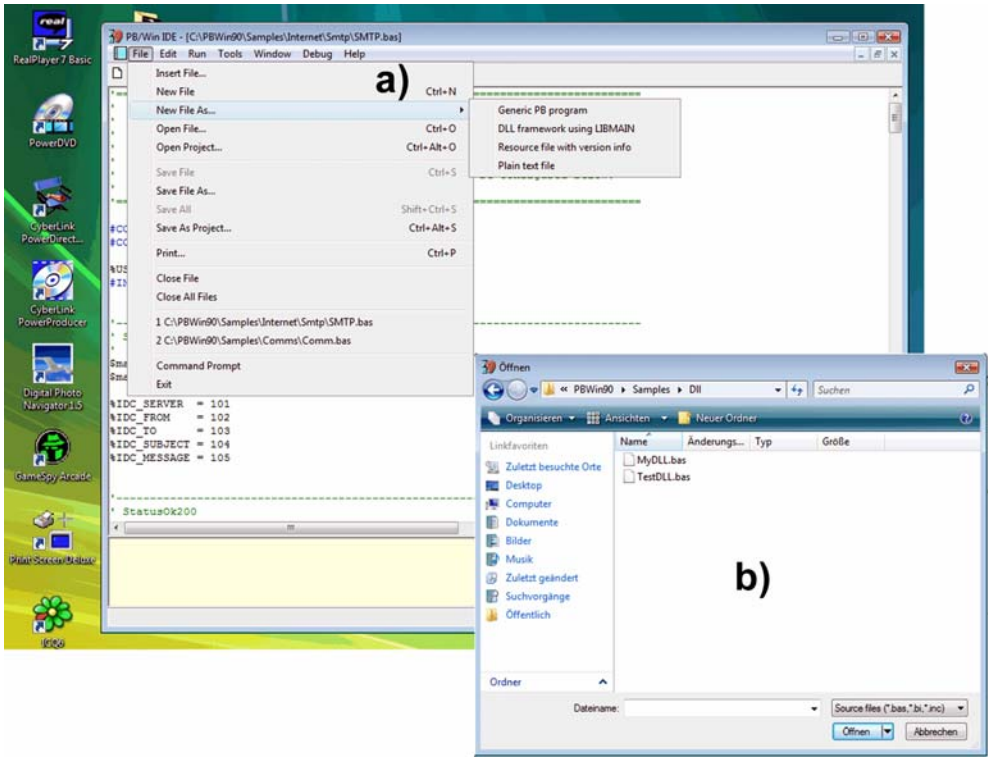


Abb. 2.64 Standarddialoge. a) Dateifunktionen; b) ein typischer Dateidialog.

Alternativen (1): X Window

X Window (auch X11 oder einfach X[85]) ist ein fensterorientiertes grafisches Subsystem, das auf dem Client-Server-Prinzip beruht (Abb. 2.60). Der Client ist hier die Anwendung, die vom X-Server Dienste anfordert. X ist im Grunde eine betriebssystemunabhängige Sammlung von Grafikroutinen. Es gibt X-Versionen für verschiedene Betriebssysteme. X bildet aber vor allem die Grundlage der grafischen Benutzeroberflächen von Unix und Linux. Es ist allerdings nicht einfach, ein komplettes System zu installieren und zu verwalten, denn es sind mehrere Bestandteile zusammenzubringen: das Betriebssystem an sich, ein X-Server und ein Fensterverwalter (Window Manager), wie beispielsweise KDE, Open Look oder Motif.

85: Sagen Sie bitte nie, unter keinen Umständen, X Windows. Ein solcher Fauxpas wird Sie aus dem Kreise ernstzunehmender Fachleute sofort ausschließen ...

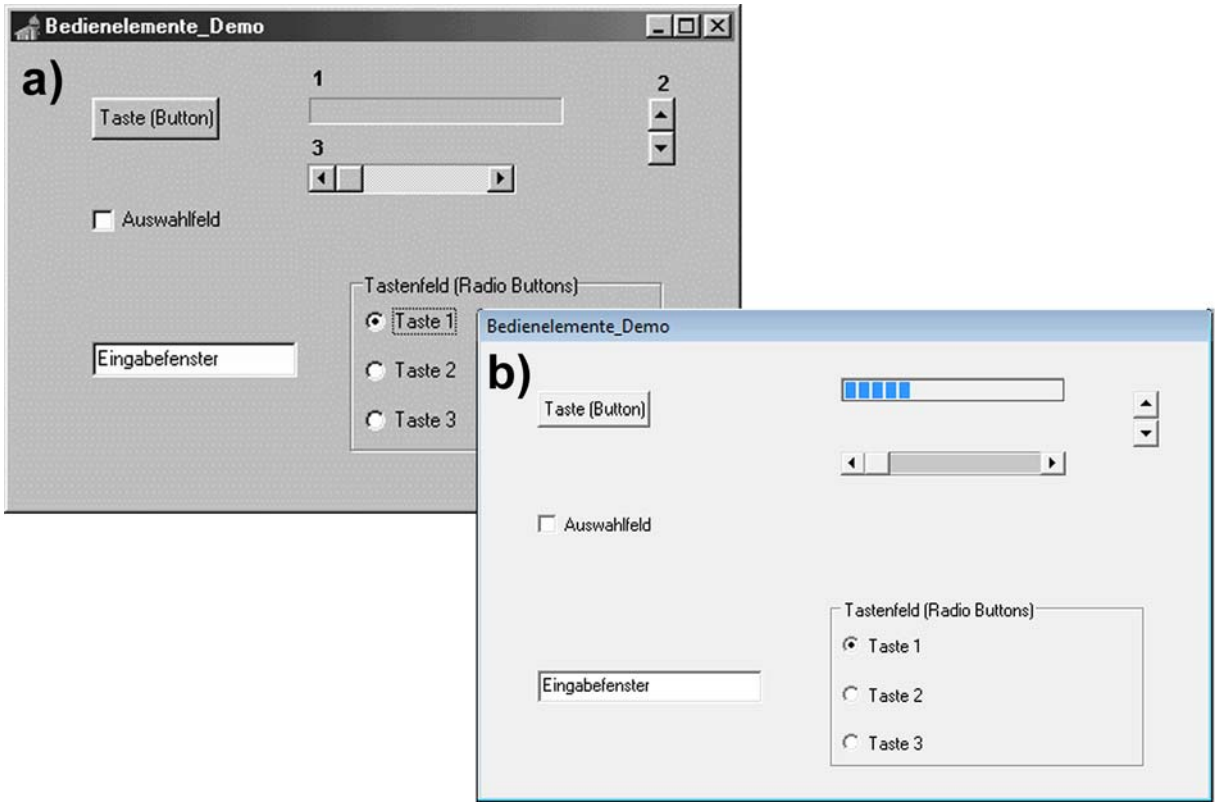


Abb. 2.65 Standardisierte Steuerelemente (Auswahl). 1 - Fortschrittsanzeige; 2 - Blättertasten (hoch-runter); 3 - Rollbalken (waagerecht). a) im Jahre 1999 erzeugt (mit Delphi 2); b) 10 Jahre später erzeugt (mit Power Basic 9). Die Funktionen sind gleich geblieben; das Äußere ist im Grunde nur schmückendes Beiwerk.

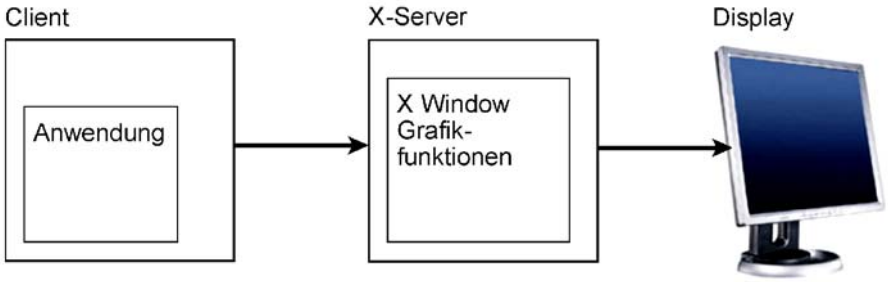


Abb. 2.66 X Window im Überblick. Alle drei Komponenten können auf verschiedene - untereinander vernetzte – Computer verteilt sein. Client und Server können aber auch in einem einzigen Computer laufen, an den ein Bildschirm angeschlossen ist.

X -Server und X-Terminals

Die Begriffe sind womöglich etwas verwirrend. Es ist aber eigentlich nicht schwierig. Der X-Server ist ein Programm, das grafische Objekte auf einem Bildschirm darstellen und dabei mit Bediengeräten wie Tastatur und Maus zusammenwirken kann. Ein X-Terminal ist ein –

vorzugsweise kostengünstiger – Computer mit angeschlossenem Bildschirm[86], auf dem nichts Anderes als ein X-Server läuft. Auch PCs können als X-Terminals betrieben werden, sogar solche, die unter DOS laufen. X verdankt seine Verbreitung zu einem beträchtlichen Teil dieser Flexiblität. Es war u. a. möglich, die eigenlichen Anwendungen auf einem Hochleistungsrechner auszuführen und für jeden Nutzer ein X-Terminal zur Verfügung zu stellen. Solche Konfigurationen erfordern allerdings einen beträchtlichen Installationsaufwand. Ein X-Server kann kommandiert werden, einzelne grafische Objekte darzustellen. Er kann sie aber nicht von sich aus vernünftig auf einer beschränkten Bildschirmfläche anordnen. Hierzu braucht man eine weitere Komponente, den sog. Fensterverwalter (Window Manager). Er läuft als gesonderter Prozess.

Praxistipp: Die einfachste Art, X einzusetzen, besteht darin, eine der komplett vorgefertigten Linux-Distributionen zu installieren, bei der man sich um die internen Mysterien nicht kümmern muss.

Alternativen (2): Das WWW

Der Grundgedanke: wenn sich die zu bedienende Anwendung wie ein Web-Server verhält, kann jedes hinreichend leisungsfähige Internet-Zugangsprogramm (Web-Browser) als Bedienprogramm verwendet werden (Abb. 2.61). Jeder, der mit einem Web-Browser umgehen kann, ist dann ohne Weiteres in der Lage, sich auf der Benutzeroberfläche einer solchen Anwendung zurechtzufinden. Zudem kann man nicht nur mit dem lokalen Browser, sondern auch übers Internet auf die Seite zugreifen. Ein weiterer Vorteil: man kann die Bedienoberfläche mit den verbreiteten Entwicklungswerkzeugen für Internetseiten erstellen. Dieses Prinzip ist vor allem im Bereich der Embedded Systems verbreitet (Abb. 2.62).

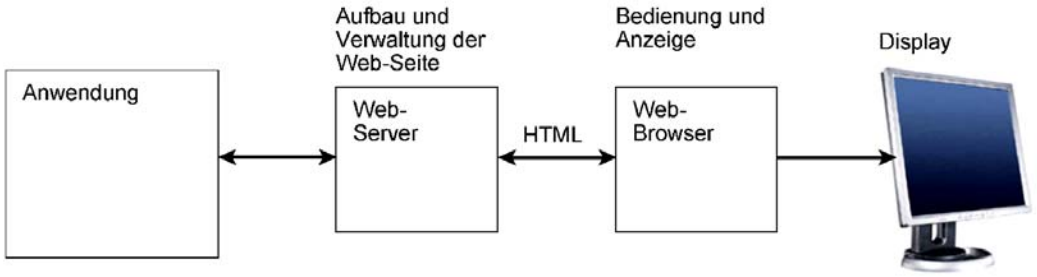


Abb. 2.67 Web-Seiten als GUI. Die Anwendung arbeitet mit einem Web-Server zusammen. Als Bedienprogramm kann ein gewöhnlicher Web-Browser eingesetzt werden.

86: Die Darstellung kann aber auch – wie bei Windows – auf mehrere Bildschirme aufgeteilt werden

Abb. 2.68 Die Web-Seite als Bedienoberfläche. Hier ein Anwendungsbeispiel aus der Gebäudeautomation (elka Elektronik, FH Dortmund).

2.9 Abfrage- und Ereignissteuerung

Auch diese Begriffe bezeichnen verschiedene Programmierphilosophien. Das Problem: wie erfährt das Anwendungsprogramm, was es zu tun hat? Mit anderen Worten: wie sieht das Programm seine Eingabeeinrichtungen?

Abfragesteuerung

Betrachten wir nochmals unser erstes Programmbeispiel, den Toaster. Eingabeeinrichtungen sind: der Endlagenkontakt 4, der Temperatursensor 5, die Stoptaste 9 sowie der Drehschalter 10. Alle diese Einrichtungen werden vom Programm abgefragt. Der Gedanke ist an sich einfach: wir schreiben ein Programm, das irgendwie anfängt und das die einzelnen Einrichtungen abfragt, um in Erfahrung zu bringen, was es als nächstes tun soll (Abb. 2.63).

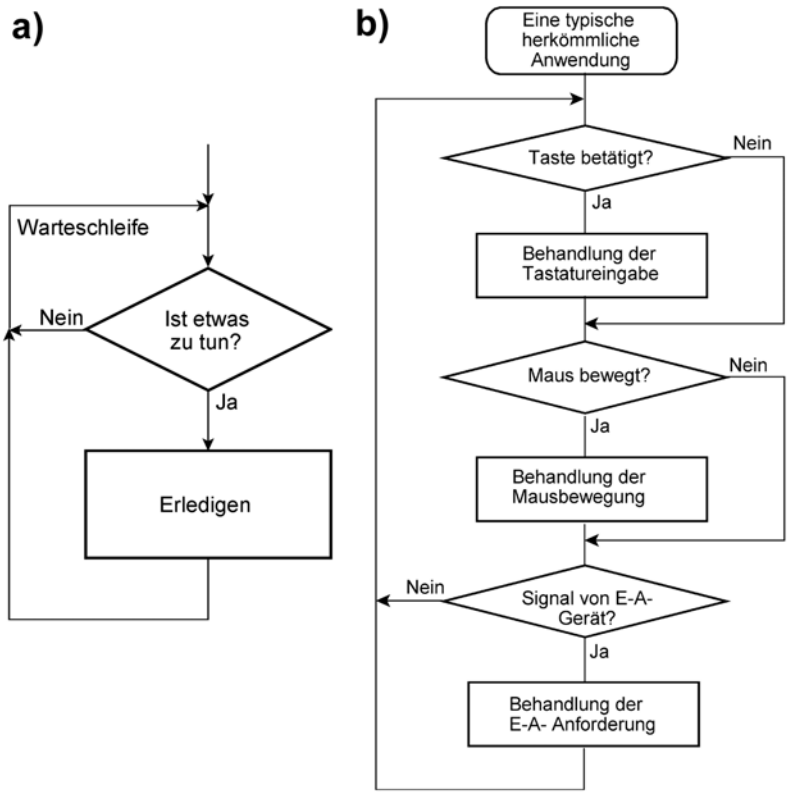


Abb. 2.69 Abfragesteuerung. a) das ganz allgemeine Prinzip; b) eine typische Abfrageschleife in einem Anwendungsprogramm.

Ereignissteuerung

Die Nachteile des Abfrageprinzips werden deutlich, wenn wir uns ein großes, kompliziertes Programm vorstellen (z. B. eine moderne Textverarbeitung mit ihren vielen Funktionen):

- Es ist viel abzufragen (Tasten, Mausbewegungen, Meldungen von Geräten usw.).
- Um auf irgend eine Eingabe überhaupt reagieren zu können, müssen wir auch im Programm "daran vorbeikommen" – ein Programm, das z. B. gerade mit einer Silbentrennung zu tun hat, kann währenddessen keine Tasten abfragen (und wenn wir – als Abhilfe – an jeder Ecke Abfragen einstreuen, wird das Programm zu groß, zu unübersichtlich und zu langsam).
- Je nach Funktionszustand ist auf gleiche Eingaben (z. B. Tastenbetätigungen) jeweils anders zu reagieren.
- In den Warteschleifen wird nicht mehr getan als Laufzeit zu verbrauchen.
- Wenn mehrere Anwendungen gleichzeitg laufen können (Multitasking), wird es noch komplizierter, denn dann muss die jeweilige Anwendung erst einmal sicherstellen, dass sie tatsächlich die Kontrolle über ihre E-A-Einrichtungen hat (Geräteverwaltung) – offensichtlich wird es nichts, wenn zwei Anwendungen gleichzeitig dieselbe Tastatur abfragen ...

Der Ausweg: eine grundsätzlich andere Gestaltung der Programme. Dabei geht man von Ereignissen (Events) aus, die außen ausgelöst werden. Ein Ereignis ist z. B. eine

Tastenbetätigung oder eine Mausbewegung. Der Programmierer schreibt dann kein großes Programm mehr, in dem alles an einem Faden hängt. Vielmehr löst er das Gesamtproblem in entsprechend viele sog. Ereignisbehandler (Event Handlers) auf. So liegt es nahe, in unserem Toaster-Beispiel folgende Ereignisse anzusetzen:

1. Das Gerät ist eingeschaltet worden.
2. Der Endlagenkontakt schließt (Korb unten).
3. Die Stoptaste wurde betätigt.

Der Programmablauf zerfällt somit in drei Ereignisbehandler (Abb. 2.64). Der Programmierer schreibt einfach die drei unabhängigen Programmstücke und überlässt es der Plattform, das jeweils passende aufzurufen. Nachdem das Ereignis behandelt wurde, kommt das Programm zu Ende[87] (vgl. die Ereignisse 1 und 3) oder es löst selbst ein weiteres Ereignis aus (vgl. Ereignis 2). Einem Ereignisbehandler ist es gleichgültig, von wem er aufgerufen wird. Es genügt also, jede Funktion nur einmal zu programmieren[88].

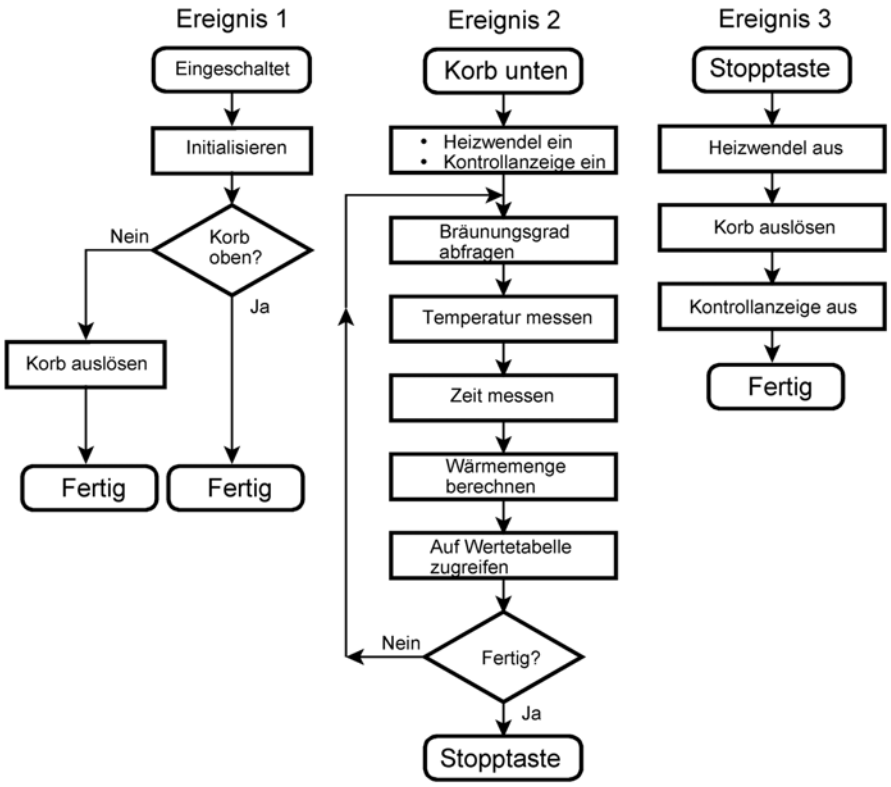


Abb. 2.70 Toaster-Steuerung mit Ereignisbehandlern.

87: Der Ablauf schließt mit einem Systemaufruf, der der Plattform das Ende anzeigt.

88: In unserem Toaster-Beispiel sind bei der weiteren Behandlung des Ereignisses 2 die gleichen Funktionen auszuführen wie beim Betätigen der Stopptaste. Der Programmierer macht es sich daher einfach und ruft den Behandler des Stopptasten-Ereignisses direkt auf.

Wie werden die Ereignisse wirksam?
Ein technisches Problem. Wir brauchen irgend etwas, das z. B. auf Grund einer Tastenbetätigung das Starten eines bestimmten Programms veranlasst. Eine Möglichkeit ist die Nutzung von Unterbrechungsvorkehrungen. Zudem muss eine wirkliche Software-Plattform vorhanden sein, die weiß, was der Prozessor tun soll, wenn ein Ereignisbehandler “fertig” geworden ist. Infolgedessen lohnt sich die Ereignissteuerung erst von einer gewissen Größe des Systems an.

Windows ist ein ereignisgesteuertes System
Jede Tastenbetätigung, jede Mausbewegung usw. wird als Ereignis wirksam. Windows-Anwendungen sind gleichsam riesige Sammlungen von Ereignisbehandlern. Integrierte Programmentwicklungsumgebungen liefern Muster, die der Anwendungsprogrammierer nur noch mit seinen Funktionen ausfüllen muss (Abb. 2.65).

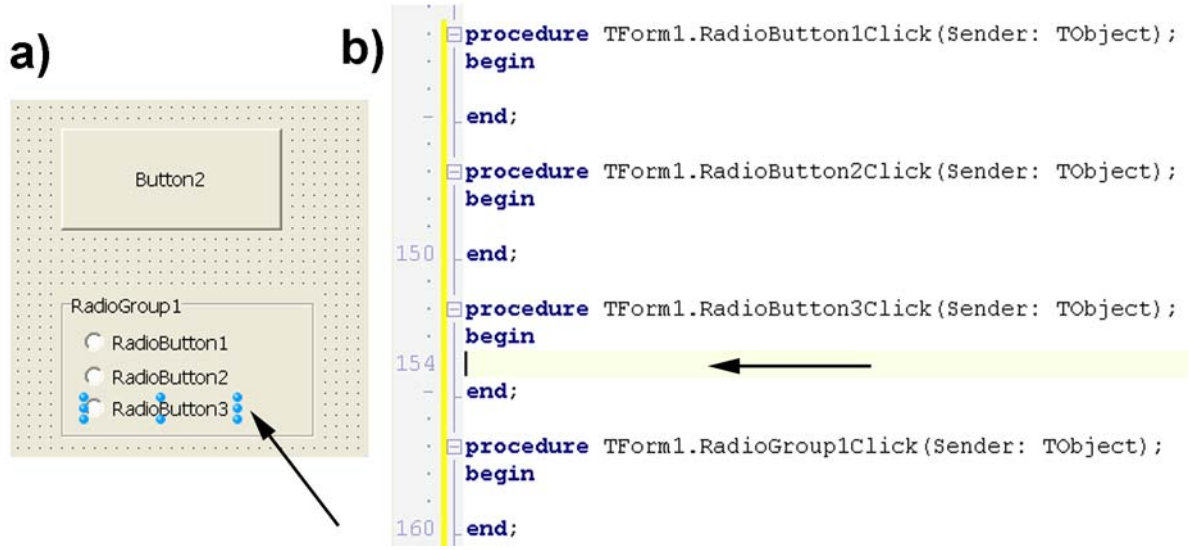


Abb. 2.71 Ereignisbehandlung in einer integrierten Entwicklungsumgebung (Beispiel: Delphi). a) der Entwurf der Bedienoberfläche; b) ein Ausschnitt aus dem automatisch erzeugten Programmcode. Jede Prozedur betrifft ein Ereignis. Was jeweils geschehen soll, wenn das Ereignis eintritt, ist zwischen begin und end einzutragen. Die Pfeile zeigen auf das Steuerelement “RadioButton3" und auf das Ereignis “RadioButton3 angeklickt".

Der Fokus
Ein wichtiger Begriff der Windows-Programmierung – den aber auch der Anwender kennen sollte. Windows kann verschiedene Fenster auf dem Bildschirm darstellen, aber nur eines ist jeweils Ziel aller Bedienhandlungen. Die übliche Redeweise: dieses Fenster besitzt den Fokus (es steht also gleichsam im Brennpunkt). Das bedeutet, dass das System alle Ereignisse, die von den Bedieneinrichtungen (Tastatur, Maus) signalisiert werden, auf die Ereignisbehandler dieses Fensters umleitet[89)].

Innen läuft es aber anders ab ...
Die elementare Windows-Systemprogrammierung beruht auf der Prorammiersprache C. Es gibt keine Sprachelemente – und auch keine Maschinenbefehle –, die eine 1:1-Abbildung von Ereignisauslösung und Ereignisbehandlung (wie in Abb. 2.64 und 2.65 gezeigt) unterstützen. Der Weg von der Auslösung zum Start der Behandlungsroutine muss also gleichsam zu Fuß ausprogrammiert werden.

Die Windows-Anwendungsprogrammschnittstelle (API) löst dieses Problem so, dass die Ereignisannahme aus Sicht des Anwendungsprogrammierers auf eine herkömmliche Abfrage zurückgeführt wird. Im Grunde kann er so vorgehen, als ob er eine herkömmliche C-Anwendung und die zugehörigen Bedienelemente (Tastatur, Maus usw.) für sich allein hätte (er schreibt also eigenlich eine Abfrageschleife ähnlich Abb. 2.63b, die zu den einzelnen Behandlungsroutinen verzweigt). Die E-A-Geräte werden aber nicht von den Anwendungen, sondern vom System behandelt. Ereignisse werden über Nachrichten bzw. Meldungen (Messages) signalisiert (Abb. 2.66). Für jedes laufende Programm verwaltet Windows eine Nachrichtenwarteschlange. Die Anwendung muss diese Warteschlange zyklisch abfragen. Diese Ereignisabfrage ist in die übliche C-Programmierung eingebunden. Auf den ersten Blick sieht es aus wie eine herkömmliche Abfrageschleife, die nach einem starren Schema aufzurufen ist (Abb. 2.67).

Die elementare Abfrageschleife besteht aus den Systemfunktionen GetMessage, TranslateMessage und DispatchMessage. GetMessage holt die Ereignisse aus der Warteschlange. Ist sie leer, so wird die Anwendung im Wartezustand gehalten. TranslateMessage wandelt den Nachrichtencode in eine einfache Binärzahl, und DispatchMessage ruft den zum betreffenden Fenster gehörenden Ereignisbehandler auf. Der Ereignisbehandler ist die zentrale Stelle im Anwendungsprogramm, an der die ankommenden Ereignisse ausgewertet werden; es ist im Grunde eine Art Verteiler, der zu den jeweiligen Behandlungsabläufen verzweigt.

89: Es kann durchaus sein, dass sich eine Bedienhandlung in einem anderen Fenster auswirkt. Dann muss dies der Anwendungsprogrammierer aber so gewollt haben (indem er das Ereignis in dem Fenster, das den Fokus hat, zwar annimmt, dort aber nichts anderes tut, als dem anderen Fenster eine Nachricht zu senden, die dort ein entsprechendes Ereignis auslöst).

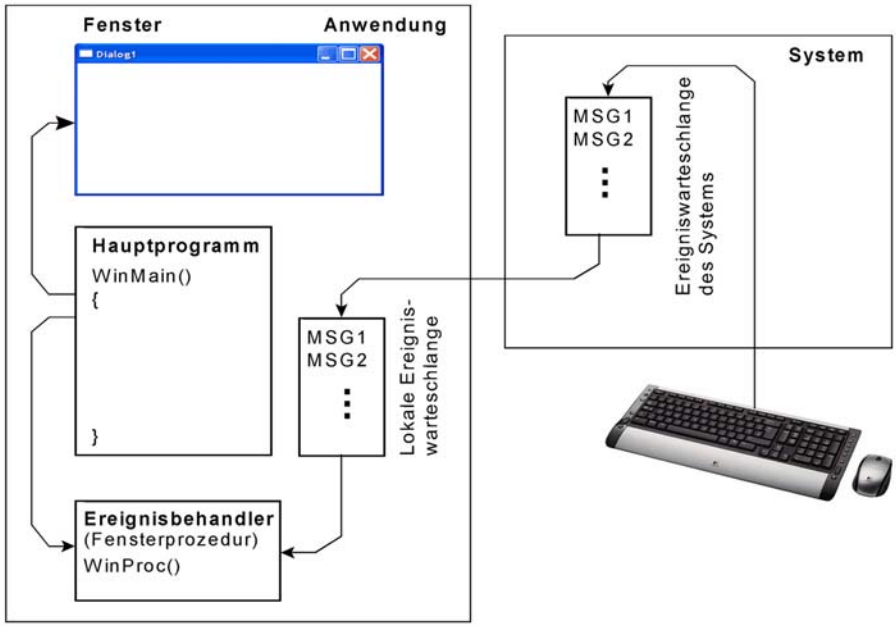


Abb. 2.72 Prinzip der Ereignissteuerung unter Windows. Die E-A-Geräte werden vom System verwaltet. Das jeweilge Hauptprogramm der Anwendung richte die Fenster und die zugehörigen Ereignisbehandler (Fensterprozeduren) ein. Der Ereignisbehandler wird vom System aufgerufen (Callback-Funktion).

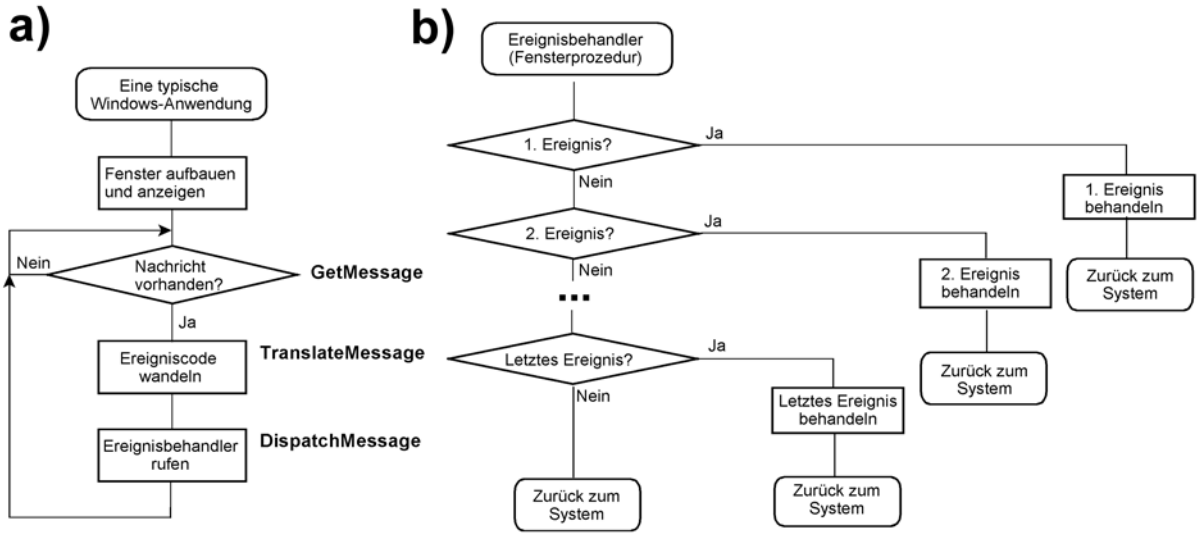
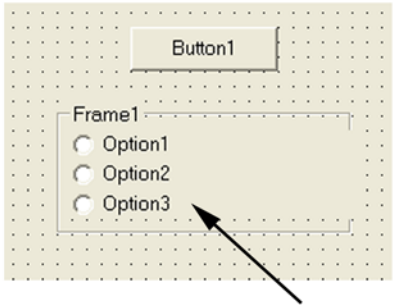


Abb. 2.73 Das Prinzip der Ereignisbehandlung in Windows-Anwendungen.

Manche Entwicklungsumgebungen – wie Delphi und Visual Basic (VB) – nehmen dem Anwendungsprogrammierer diese Einzelheiten ab; er muss sich nicht darum kümmern, wie seine Ereignisbehandler (vgl. Abb. 2.65) aktiviert werden. Den zugehörigen Programmcode (der Abläufe gemäß Abb. 2.66) bekommt er nie zu sehen. Abb. 2.67 veranschaulicht ein Gegenbeispiel – eine Entwicklungsumgebung, die die Callback-Funktionen auch im Quellcode des Anwendungsprogramms sichtbar werden lässt.




```
CALLBACK FUNCTION ShowDIALOG1Proc()

    SELECT CASE AS LONG CBMSG
        CASE %WM_INITDIALOG
            ' Initialization handler

        CASE %WM_NCACTIVATE
            STATIC hWndSaveFocus AS DWORD
            IF ISFALSE CBWPARAM THEN
                ' Save control focus
                hWndSaveFocus = GetFocus()
            ELSEIF hWndSaveFocus THEN
                ' Restore control focus
                SetFocus(hWndSaveFocus)
                hWndSaveFocus = 0
            END IF

        CASE %WM_COMMAND
            ' Process control notifications
            SELECT CASE AS LONG CBCTL
                CASE %IDC_BUTTON1
                    IF CBCTLMSG = %BN_CLICKED OR CBCTLMSG = 1 THEN
                        MSGBOX "%IDC_BUTTON1=" + FORMAT$(%IDC_BUTTON1), _
                            %MB_TASKMODAL
                    END IF

                CASE %IDC_OPTION1

                CASE %IDC_OPTION2

                CASE %IDC_OPTION3

                CASE %IDC_FRAME1

            END SELECT
    END SELECT
END FUNCTION
```

Abb. 2.74 Das Problem von Abb. 2.63 in einer anderen Entwicklungsumgebung (Power Basic). Die Callback-Funktion der Ereignisannahme ist als Quellcode dargestellt. Die einzelnen Ereignisse werden mit SELECT-CASE-Anweisungen abgefragt.

3. Schnittstellen der Software

3.1 Anwendungsprogrammschnittstellen

Eine Anwendungsprogrammschnittstelle (Application Programming Interface API) ermöglich es dem Programmierer, Funktionen, die von der Plattform bereitgestellt werden, in seinem Programm zu nutzen. Ein API ist praktisch eine Sammlung von Funktionsaufrufen. Moderne Programmentwicklungsumgebungen erleichtern es, solche Aufrufe zu formulieren.

Beispiel:
Der Anwendungsprogrammierer beabsichtigt, dass ein bestimmtes rechteckiges Fenster auf dem Bildschirm erscheint. Bei Nutzung einer modernen Entwicklungsumgebung (hier: der Borland[90] C++ Builder) wird dies etwa so hingeschrieben:

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    Canvas->Pen->Color = clRed;
    Canvas->Brush->Color = clWhite;
    Canvas->Brush->Style = bsSolid;
    Canvas->Pen->Width = 10;
    Canvas->Rectangle (120, 80, 900, 600);
}
```

Diese angegebenen Funktionsaufrufe werden – vom Compiler – in Aufrufe des Betriebssystems umgesetzt. Beispielsweise bewirkt unter den Windows-Betriebssystemen folgender Aufruf, dass auf dem Bildschirm ein Rechteck erscheint:

Rectangle (handle, x-links, y-oben, x-rechts, y-unten)

(Es handelt sich um einen Aufruf des GDI (vgl. Abschnitt ***). Hierbei sind die x- und y-Angaben die Koordinaten der oberen linken und der unteren rechten Ecke; *handle* ist ein Zeiger auf den jeweiligen “Gerätekontext”.)

Hinweise:

1. Nicht weiter nachdenken! Auch wirklich gestandene, erfahrene Programmierer wissen oftmals nur, was sie hinschreiben müssen (z. B. – s. oben – *Canvas ->Pen...*). Dass daraus ein Aufruf *Rectangle...* wird, wissen nur jene, die sich zudem in die Einzelheiten der Windows-Programmierung eingearbeitet haben.

90: So hieß der Anbieter ursprünglich. Dann hieß er Inprises. Neuerdings heißt er Embarcadero.

2. Es gibt verschiedene Anwendungsprogrammschnittstellen. Bestimmte APIs werden zur Plattform mitgeliefert, andere müssen gesondert beschafft werden.
3. Anwendungsprogrammschnittstellen werden typischerweise in Form von sog. dynamischen Programmbibliotheken (DLLs) bereitgestellt (Näheres in Abschnitt 3.3).
4. Benutzeroberfläche und Anwendungsprogrammschnittstelle sind nicht dasselbe. Eine Banalität, die man sich aber manchmal ins Gedächtnis rufen muss. Beispiel: wir kaufen ein Gerät, das nicht zur allgemeinen Grundausstattung der PCs gehört, z. B. einen Scanner, und wir installieren die mitgelieferte Software. Dann können wir das Gerät zwar bedienen (Benutzeroberfläche), aber noch lange nicht aus eigenen Programmen heraus ansprechen – denn wir haben keine passende Programmschnittstelle (API). Natürlich gibt es so etwas. Bei Scannern wäre dies z. B. die TWAIN-Schnittstelle. Um auf dieser Grundlage zu programmieren, brauchen wir aber eine passende Entwicklungssoftware (in diesem Fall: das TWAIN Developer's Toolkit). Manche Entwicklungsprogrammpakete sind ausgesprochen kostspielig. *Praxistipp:* Gelegentlich kann man sich mit Scripting-Vorkehrungen behelfen (z. B. ein Skript aufsetzen, das mehrere Anwendungen über Dateien miteinander verknüpft).

3.2 Bibliotheksfunktionen und Laufzeitsysteme

Von diesen Funktionen merkt der Nutzer praktisch nichts[91)]. Sie werden im Rahmen der Plattform oder der jeweiligen Programmentwicklungsumgebung bereitgestellt.

Eine Bibliothek (Library) enthält allgemein nützliche Programme, z. B. zur Ein- und Ausgabesteuerung, zum Berechnen mathematischer Ausdrücke usw. Solche Funktionen werden typischerweise vom Anwendungsprogramm aufgerufen (es handelt sich also um sozusagen gewöhnliche Unterprogramme, wie man sie als Programmierer auch selbst schreiben könnte).

Ein Laufzeitsystem ist ein Programmkomplex, der das Ausführen von Programmen unterstützt, die in einer höheren Programmiersprache geschrieben wurde (so gibt es C++-Laufzeitsysteme, Java-Laufzeitsysteme usw.). Als Anwendungsprogrammierer können wir diese Funktionen nicht aufrufen, sondern es ist der Compiler (Kapitel 4), der solche Aufrufe in das ausführbare Programm einbaut. Sie dienen vor allem zu Verwaltungs- und Kontrollzwecken (z. B. zum Aufrufen von Unterprogrammen oder zum Prüfen von Feldgrenzen).

3.3 Aufrufverfahren

Es kommt durchaus vor, dass ein Computer immer nur einziges Programm auszuführen hat. Wenn man es genau nimmt, tun dies sogar die weitaus meisten Computer, die heutzutage im Einsatz sind, nämlich die typischen Embedded Systems – vom Toaster bis zur Motorsteuerung im Auto. Im Bereich der PCs haben wir es aber mit Anwendungs- und Systemprogrammen zu tun, die immer wieder geändert werden. Auch kommt laufend Neues hinzu. Es müssen also stets mehrere Programme zusammenarbeiten, wobei sich die Einsatzbedingungen ständig ändern. Es

91: Es sei denn, die betreffenden Dateien sind gerade verschwunden oder fehlerhaft ...

leuchtet ein, dass es beim Zusammenwirken solcher Programme auf reguläre Software-Interfaces (Programmschnittstellen) ankommt. Im Folgenden sollen drei Grundsatzfragen solcher Schnittstellen angesprochen werden:

- das Laden von Programmen in beliebige Speicherbereiche (Verschieblichkeit),
- das Aufbauen von Anwendungslösungen aus mehreren Programmbestandteilen (modulare Programmorganisation),
- das Aufrufen von Systemfunktionen.

3.3.1 Verschiebliche Programme

Damit ist Folgendes gemeint: es muss möglich sein, ein Programm an eine x-beliebige Stelle im Speicher zu laden – nämlich dahin, wo gerade Platz ist – und es dort auch auszuführen[92].

Programmverschiebung während der Programmentwicklung
Geht es um vergleichsweise einfache Systeme (vgl. unser Toaster-Beispiel), so erledigt sich das Problem gleichsam von selbst. Die Programme stehen typischerweise in einem ROM (z. B. eines Mikrocontrollers). Damit liegen sie ohnehin im wörtlichen Sinne “fest”. Der Programmcode wird von einem Entwicklungssystem erzeugt (genauer: mittels eines Assemblers oder eines Compilers). Diese Entwicklungsprogramme übersehen alle Adressen und können deshalb die Zugriffs- und Verzweigungsbefehle mit jeweils passenden Adressangaben in den Code eintragen.

Programmverschiebung zur Laufzeit
Wir können uns leicht klarmachen, dass es in Systemen, die dazu vorgesehen sind, eine Vielzahl von Anwendungen zu unterstützen, nicht praktikabel ist, die Programmverschiebung der Entwicklungsumgebung zu überlassen – denn entweder müsste die Speicherbelegung eines jeden Programms von Anfang an festliegen, oder es müssten immer dann, wenn ein Programm zwecks Ausführung geladen werden soll, die Adressangaben neu angepasst werden.

Es gibt drei grundsätzliche Auslegungen:

1. Die Programmverschiebung wird ausschließlich von der Hardware erledigt. Praktisch sieht das so aus, dass man jedes Programm für sich so entwickeln kann, als ob es den Speicher von Adresse 0 an für sich hätte. Die Hardware sorgt dann dafür, dass die erforderlichen Adressumsetzungen und Adressrechnungen ausgeführt werden.
2. In der Hardware wird nichts vorgesehen. Die Adressumsetzung wird programmseitig beim Laden oder zur Laufzeit erledigt.
3. Kompromisslösungen zwischen 1. und 2.

92: Zur Fachsprache: Ein Programm, das diese Eigenschaft hat, heißt verschieblich oder relocatable, die Programmverschiebung selbst heißt Relocation.

Verschieblichkeit durch Befehlsmodifikation

Bei den weitaus meisten Befehle stellt sich das Problem gar nicht; sie haben immer dieselbe Wirkung, gleichgültig an welcher Adresse sie untergebracht sind. Bei manchen Befehlen, die Adressangaben enthalten, ist das aber nicht der Fall, da die adressierten Programmstücke (genauer: Programmsegmente[93]) bei jedem Aufruf immer wieder in andere Speicherbereiche geladen werden. Um dieses Problem zu lösen, werden den Programmen Adressumsetzungstabellen (Relocation Tables) beigegeben. Zu jedem Befehl, der sich auf ein anderes Segment bezieht, gibt es einen Tabelleneintrag. Er verweist zum Einen auf die Adresse im jeweiligen Zielsegment und zum Anderen auf die zu ändernde Adressangabe im betreffenden Befehl (Abb. 3.1)[94]. Beim Laden stehen die aktuellen Adressen aller zu ladenden Segmente fest. Demgemäß werden die Adressteile der betreffenden Befehle mit den jeweils gültigen Werten überschrieben.

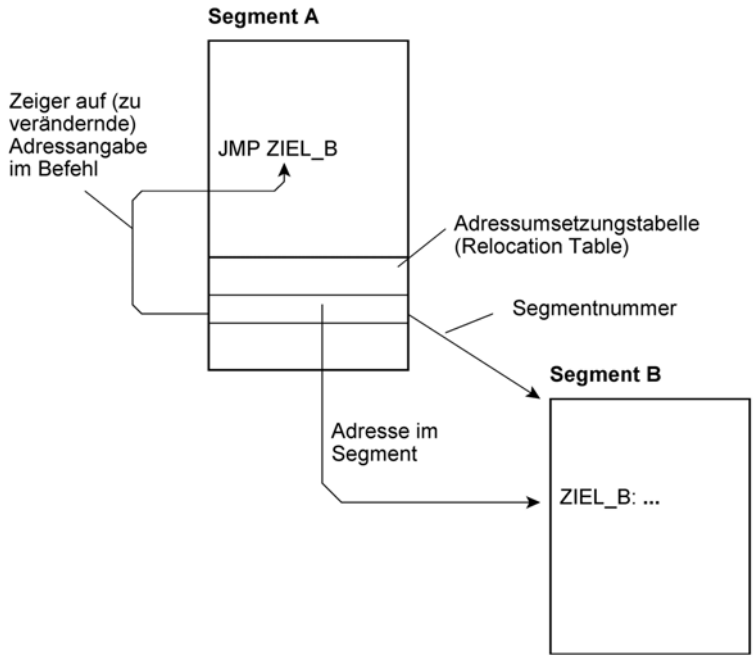


Abb. 3.1 Befehlsmodifikation über Adressumsetzungstabelle (Relocation Table). Beim Laden ist bekannt, an welchen Adressen die einzelnen Segmente untergebracht werden. Die Umsetzungstabelle des Segments verweist auf alle Adressangaben, die sich auf andere Segmente beziehen. Diese werden mit den aktuellen Adresswerten überschrieben.

93: Die Speicherverwaltung der x86- und IA32-Prozessoren beruht auf dem Prinzip der Segmentierung.

94: Abb. 3.1 ist eine sehr pauschale Darstellung, die nur das Prinzip veranschaulichen soll. Zum Aufbau der Tabellen muss auf die einschlägige Dokumentation verwiesen werden. In der Praxis braucht man diese Einzelheiten jedoch nicht (sie werden auch in den üblichen Hand- und Lehrbüchern der Programmierung nur selten beschrieben – und wenn, dann zumeist auch nur im Überblick ...).

3.3.2 Programme aus Modulen aufbauen

Es liegt nahe, umfangreiche Anwendungslösungen aus mehreren überschaubaren Programmen aufzubauen. Das betrifft sowohl die Zerlegung der selbst zu schreibenden Programmteile in handliche Abschnitte als auch die Nutzung fertiger Programmbausteine. Das Problem ist, das Zusammenwirken dieser Module zu organisieren. Der grundsätzliche Fachbegriff: Programmverbindung (Linking).

Programmverbindung während der Programmentwicklung
Während der Programmentwicklung (genauer: zur Compilierzeit) werden die Quelltexte aller Programm-Module übersetzt und zusammengeführt (Programmverbindung, Linking). Im einfachsten Fall entsteht aus mehreren Quelltexten eine einzige Datei, die den Maschinencode des Programms enthält. Es ist aber auch möglich, mehrere Objektdateien zu erzeugen. Wichtig ist, dass alle Adressbzüge untereinander festliegen (statische Programmverbindung).

Das ist die herkömmliche Vorgehensweise. So werden herkömmlicherweise Unix- oder Linux-Anwendungen in Form statisch miteinander verbundener Dateien (Statically Linked Images) ausgeliefert.

Der Vorteil liegt darin, dass solche Anwendungen in sich abgeschlossen sind und nicht von außen beeinflusst werden können. Der offensichtliche Nachteil liegt in der Speicherplatzverschwendung, die sich dann ergibt, wenn irgendwelche Dienst- oder Bibliotheksfunktionen von mehreren Anwendungen genutzt werden.

Beispiel: Es gibt eine Sammlung von Grafikfunktionen. Wir wollen sie LIB_2D nennen. Der Maschinencode umfasse 200 kBytes. Nun gibt es mehre Anwendungen, die diese Funktionen nutzen: ein Zeichenprogramm SUPER_CAD, ein Mathematikprogramm SUPER_MATH und ein Spiel SUPER_GAME. Wären dies alles herkömmliche Unix-Anwendungen, so würden die Grafikroutinen insgesamt 3 • 200 = 600 kBytes belegen. Das wäre der Nachteil. Ein einziges von allen Anwendungen gemeinsam nutzbares Programm wäre offensichtlich viel günstiger. Nun wissen wir aber aus der Praxis, dass komplizierte Programme nie so bleiben, wie sie ursprünglich geschrieben wurden. Sie enthalten (1) Fehler, und es gibt (2) immer wieder etwas zu verbessern.

Betrachten wir zunächst die Fehlerbeseitigung: auch hier wäre es von Vorteil, nur eine einzige Programmdatei zu haben. Wird sie durch eine neue Version ersetzt (Update), so kommt dies sofort allen Anwendungen zugute. Ansonsten müsste jede Anwendung neu übersetzt und gelinkt werden.

Bei den Verbesserungen ist die Sache jedoch nicht so eindeutig. Nehmen wir an, den Entwicklern des Spiels SUPER_GAME seien manche Grafikfunktionen einfach zu langsam, andere hingegen unnötig kompliziert (da es beim Spielen nicht so genau darauf ankommt ...). Sie wandeln also LIB_2D entsprechend ab. Gäbe es aber nur ein einziges Programm für alle Anwendungen, so könnte es sein, dass die anderen Anwendungen mit dieser Version nicht mehr zurecht kommen.

Dynamische Programmbibliotheken (DLLs)

DLL = Dynamic Link Library. Eine DLL ist eine Programmdatei, die ausführbaren Code enthält, aber nur vorgesehen ist, um von anderen Programmen aufgerufen zu werden – es ist im Grunde eine Sammlung von Unterprogrammen. Dies ist die Verwirklichung der soeben skizzierten Idee, gemeinsam nutzbare Funktionen nur einmal vorzusehen und sie allen Anwendungsprogrammen zur Verfügung zu stellen. Wichtig ist, dass alle Programme unabhängig voneinander übersetzt und geladen werden können, dass also nicht eigens ein Link-Durchlauf erforderlich ist, um Anwendungs- und Bibliotheksprogramme miteinander zu verbinden.

Windows und seine DLLs

Die Windows-Plattformen sind durch eine ausgiebige Nutzung von DLLs gekennzeichnet. Die Vorteile dieser Auslegung – Speicherplatzersparnis und sofortige Nutzbarkeit von Aktualisierungen und Verbesserungen – wurden bereits erläutert. Im Grunde ist es auch die einzige Auslegung, die für den Massenmarkt wirklich geeignet ist – im Gegensatz zum Unix-Administrator und zum Linux-Freak[95)] kann man es dem typischen PC-Nutzer wohl kaum zumuten, von Zeit zu Zeit die Quelltexte neu zu übersetzen, um die Updates einzubauen (ganz zu schweigen davon, dass dazu ja die Software-Anbieter ihre Quelltexte zur Verfügung stellen müssten).

Das DLL-Prinzip wurde auch für das Betriebssystems selbst ausgenutzt. Viele Systembestandteile sind als DLLs ausgelegt. Daraus ergibt sich der Vorteil, dass Windows zwar installiert, aber nicht generiert werden muss. Hierzu ein kurzer Blick in die Entwicklungsgeschichte: Auch vor Jahrzehnten schon war Systemsoftware modular aufgebaut. Nun braucht jedes System seine ganz bestimmte Ausstattung. Um diese zusammenzustellen, war ein sog. Generierungslauf erforderlich. Hierbei wurden die einzelnen Module ausgewählt, und alle Querbezüge wurden aufgelöst (ein dem Linken von Anwendungsprogrammen vergleichbarer Vorgang). Erst nach dem Generieren war ein System lauffähig. Waren größere Änderungen einzubringen, so war ein neuer Generierungslauf erforderlich. Bei Windows hingegen genügt es, eine DLL mit passendem Namen im passenden Verzeichnis zu installieren – es findet sich dann (zur Laufzeit) schon alles zueinander.

DLL-Funktionen aufrufen

Das Aufrufen von Funktionen in DLLs beruht darauf, dass die Programmdateien Umsetzungstabellen enthalten. Es gibt zwei Arten von Umsetzungstabellen[96)]:

- Exporttabellen verweisen auf alle Funktionen, die das Programm nach außen hin zur Verfügung stellt. Die exportierten Funktionen bilden praktisch das Interface des Programms zur Außenwelt.

95: Im Bereich Unix/Linux gibt es aber mitterweile auch etwas Ähnliches wie die DLLs, die sog. Shared Libraries.

96: Auch das ist nur ein ganz grober Überblick. Einzelheiten stehen in den Hand- und Lehrbüchern der Windows-Systemprogrammierung. In der Praxis brauchen wir sie nicht.

- Importtabellen enthalten Angaben zu allen Programmen, die das betreffende Programm seinerseits benötigt. So enthält die Importtabelle eines typischen Anwendungsprogramms die Bezeichner aller genutzten DLLs.

Es gibt zwei Arten des Funktionsaufrufs:

1. Über die laufende Nummer (Ordinalzahl) der Funktion.
2. Über den smybolischen Namen der Funktion (der in Textform – als Zeichenkette – übergeben wird).

Abb. 3.2 veranschaulicht das Prinzip.

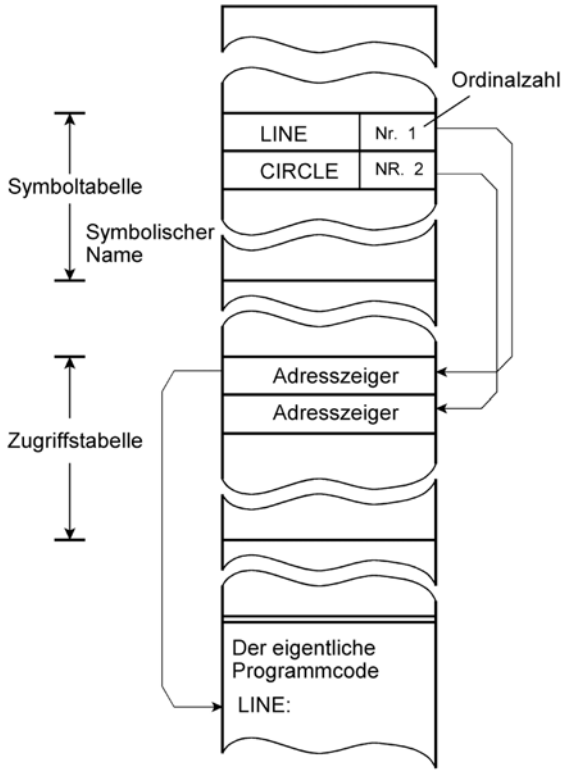


Abb. 3.2 Der Aufruf einer Funktion in einer DLL.

Schreibt ein Programmierer z. B. *LINE (...)* hin, so leistet das Laufzeitsystem folgendes:

1. Es durchsucht die Symboltabelle nach dem Namen LINE.
2. Es findet die zum Namen gehörende Ordinalzahl.
3. Es geht mit dieser Ordinalzahl in die Zugriffstabelle und findet dort den Adresszeiger. Diese Adressangabe wird verwendet, um die betreffenden Funktion aufzurufen (indirekte Adressierung).

Der Aufruf über Ordinalzahlen ist offensichtlich viel schneller als der über Namen. Deshalb haben viele Programmierer diesen Weg bevorzugt. Es ist aber schwierig, bei den Tausende von DLLs (und dem hektischen Änderungsgeschehen) den Überblick zu behalten. Es kann schon mal vorkommen, dass eine Fehlermeldung erscheint, die ähnlich aussieht wie die folgende[97]:

The ordinal 968 could not be located in the dynamic-link library MFC42.dll.

Deshalb empfiehlt Microsoft, DLL-Funktionen grundsätzlich über ihren Namen zu rufen. Den Aufwand (Speicherkapazität, Rechenzeit zum Durchsuchen) kann man sich mittlerweile leisten.

Wann wird eine DLL geladen?
Es gibt zwei Verfahrensweisen:

- Laden zur Ladezeit des rufenden Programms (Implicit Linking). Wird ein Programm geladen, so sucht das System in den Importtabellen die benötigten DLLs und lädt diese ebenfalls. Erst dann, wenn alle angeforderten DLLs im Speicheradressraum verfügbar sind, wird das Programm ausgeführt.
- Laden bei Bedarf (zur Laufzeit, Explicit Linking). Solche DLLs werden dann geladen, wenn sie benötigt werden. Sie müssen hierzu (über entsprechende Systemfunktionen) vom Anwendungsprogrammierer angefordert werden.

Wie werden DLLs geladen?
DLLs werden als Dateien in den Arbeitsspeicheradressraum abgebildet. Mit anderen Worten, DLLs sind Memory Mapped Files (Abschnitt 2.6.2 ****). Üblicherweise bringt Windows die DLLs in den globalen Heap (vgl. ***) und bildet den jeweiligen Adressbereich im virtuellen Speicheradressraum jedes Prozesses (= jeder Anwendung) ab, der die DLL verwendet.

Woher werden die DLLs geladen?
Ist der genaue Ort nicht angegeben, so sucht das System an folgenden Stellen:

- in dem Verzeichnis, das die rufende Programmdatei enthält,
- im aktuellen Verzeichnis der Anwendung,
- im Windows-Systemverzeichnis,
- in den Verzeichnissen, die in der Umgebungsvariablen des Zugriffsweges (PATH) angegeben sind.

Wurde nichts gefunden, so wird eine Fehlermeldung ausgegeben.

DLL-Konflikte
Das Durcheinander mit den DLLs (im Englischen bildhaft als DLL Hell bezeichnet) ist eines der größten Windows-Probleme. Hier einige typische Fehlererscheinungen:

97: Beispiel nach Microsoft.

- Mangelhafte Abwärtskompatibilität.
- Neue Versionen enhalten neue Fehler.
- DLLs wurden irrtümlich gelöscht.
- DLLs werden beim Installieren überschrieben. Hierbei kann es vorkommen, dass wir uns mit einem Programm, das wir später installieren, eine DLL eines älteren Standes in unser System holen und dass dann manches – das bisher immer funktioniert hat – plötzlich nicht mehr funktioniert. Andererseits kann es sein, dass, wenn der ältere – der zu installierenden Anwendung beigegebene – Stand nicht installiert wird, die Anwendung mit der bereits vorhandenen (neueren) DLL nicht richtig funktioniert.

Private DLLs
Spezifische, auf eine Anwendung zugeschnittene DLLs werden im Verzeichnis der Anwendung untergebracht[98]. Sie werden somit nur für diese Anwendung geladen. In unserem Beispiel hätte die Anwendung SUPER_GAME eine eigene DLL LIB_2D, während die anderen Anwendungen die gleichnamige DLL aus dem Systemverzeichnis nutzen würden.

Das Windows-Dateischutzsystem (Windows File Protection WFP)
Diese Vorkehrung wurde von Windows 2000 an eingeführt. Die hierüber geschützten DLLs des Systems können nur von Betriebsystem-Updates (z. B. Servie Packs) überschrieben werden. Es gelingt zwar, DLLs im Systemverzeicnis zu überschreiben (so werden entsprechende Kopiervorgänge ohne Fehlermeldung abgeschlossen). Das System überprüft aber die geschützten DLLs daraufhin, ob sie eine gültig digitale Signatur haben.Haben sie keine, werden sie mit der Original-DLL des Systems überschrieben.

Hinweise:

1. Viele DLLs kommen im Bündel. Es kann deshalb sein, dass es nicht viel nützt, nur eine einzelne DLL zu ändern (indem man sie mit einer besser passenden Version überschreibt).
2. Die Internetseite von Microsft bietet eine Vielzahl von Texten und Hilfsprogrammen zu diesem Problemkreis.
3. Mit dem sog. .NET-Framework soll alles viel besser werden ...

3.3.3 Betriebssystemfunktionen aufrufen

Der Betriebssystemaufruf ist ein Spezialfall des Aufrufs von Programm-Modulen:

- Betriebssystem und Anwendung sollen grundsätzlich voneinander unabhängig sein.
- Es ist zumeist ein Umschalten vom Anwender- in den Systemzustand erforderlich.
- Es soll schnell gehen.

98: Diese Empfehlung stand schon immer in den Programmierrichtlinien von Microsoft, nur haben sich etliche Anbieter nicht daran gehalten ...

Hierzu hat man spezielle Maschinenbefehle in den Prozessoren vorgesehen. Das "klassische" Beispiel ist der Befehl "Supervisorruf" (Supervisory Call, SVC) der IBM-Mainframes. Dieser Befehl hat zwei Wirkungen:

1. Er überführt die Maschine in den Systemzustand schaltet (es ist der einzige Befehl, mit dem dies aus dem Anwendungsprogramm heraus möglich ist).
2. Er übergibt der Systemsoftwaredie laufende Nummer der gewünschten Dienstleistung.

In den x86-Prozessoren ist der Befehl "Aufruf Interruptbehandler" (INTn) sinngemäß nutzbar. Im Speicher muss eine Tabelle (Interrupt Table) vorhanden sein, die für jede laufende Nummer n (von Null an) eine Adressangabe enthält. Der Prozessor adressiert die Tabelle gemäß der übergebenen Nummer und führt mit der entnommenen Adressangabe eine Verzweigung aus. Dieses Prinzip wurde auch in der 32-Bit-Architektur IA-32 erwirklicht. Nur hat man es um umfangreiche Speicherverwaltungs- und Schutzvorkehrungen erweitert. Diese Vorkehrungen sind allerdings von den Betriebssystementwicklern nicht voll ausgenutzt worden. Infolgedessen hat sich die – wie es sich in der Praxis herausgestellt hat, unnötige – Kompliziertheit als Bremse erwiesen (zu viel Overhead, zu lange Umschaltzeiten). Deshalb hat man neue Befehle eingeführt, die dem traditionellen Supervisorruf ähnlich sind – die Befehlspaare SYSENTER/SYSEXIT und SYSCALL/SYSRET (letztere für die 64-Bit-Erweiterung).

3.4 Schutzvorkehrungen

Beim Aufrufen von Systemprogrammen kann Einiges schiefgehen. Was könnte man dagegen tun? Stichwort: Schadensbegrenzung. Auch hierfür gibt es die uns bereits bekannten drei Ansätze

4. Schutzmaßnahmen (Protection Features) in der Hardware.
5. Maßnahmen in der Software (die an verschiedenen Stellen ansetzen können, z. B. in Kombination von (erzwungener) Programmierdisziplin und Kontrollfunktionen zur Entwicklungs- und Laufzeit.
6. Verbund- und Kompromisslösungen von 1. und 2.

Die einfachste Lösung: zwei Zustände

Die Hardware kann in jeweils einem von zwei Zuständen oder Privilegierungsstufen arbeiten. Diese Zustände werden üblicherweise als System- und als Anwenderzustand bezeichnet (System bzw. Supervisory Mode, User Mode). Programmen, die im Systemzustand laufen, stehen alle Ressourcen zur Verfügung (alle Maschinenbefehle, der gesamte Speicher, die E-A-Anschlüsse usw.). Im Anwenderzustand sind bestimmte Ressourcen von der Nutzung ausgeschlossen. Das gilt beispielsweise für Maschinenbefehle, die für die E-A-Steuerung und Speicherverwaltung vorgesehen sind (privilegierte Befehle). Der Wechsel zwischen beiden Zuständen ist nur über genau definierte Schnittstellen (z. B. mit einem besonderen Maschinenbefehl) möglich.

Dieses Schutzsystem, das im Wesentlichen Systemsteuer- und reine Verarbeitungsfunktionen voneinander isoliert, kann durch Speicherschutzmaßnahmen (Memory Protection Features) ergänzt werden, um auch innerhalb des Anwenderzustandes Fehler so zeitig wie möglich zu erkennen (das betrifft z.B. den Versuch, ein Programmstück mit Daten zu überschreiben oder den Zugriff eines Anwendungsprogramms auf Speicherbereiche, die einem anderen Programm zugewiesen sind).

Die ausgebaute Lösung: mehrere Privilegierungsstufen (ringförmiger Schutz)
In Weiterführung des Prinzips können durch mehr als zwei Zustände die Schutzmaßnahmen feiner dosiert werden. Dies ist z. B. in der IA-32-Architektur verwirklicht. Hier sind vier Zustände vorgesehen, die als Privilegebenen (Privilege Levels) bezeichnet werden. Sie sind von 0 bis 3 durchnummeriert. Privilegebene 0 entspricht dem eigentlichen Systemzustand (wo alles erlaubt ist), Privilegebene 3 dem eigentlichen Anwenderzustand. Den Privilegebenen liegen folgende Vorstellungen zur Nutzungsweise zugrunde:

- Privilegebene 0: der Kern des Betriebssystems (Betriebsmittelverwaltung, Speicherverwaltung, Starten und Beenden von Programmen, Fehlermaßnahmen usw.).
- Privilegebene 1: Dienstleistungen im Umfeld des Betriebssystems (bestimmte E-A-Funktionen, Dateiverwaltung usw.).
- Privilegebene 2: Dienstleistungen im Umfeld der Anwendung (allgemein nutzbare Bibliotheken mit grafischen, numerischen usw. Programmen sowie Tabellen, Zeichensätze, Wörterbücher usw.).
- Privilegebene 3: die eigentliche Anwendungsprogramme.

Um solche Schutzsysteme zu veranschaulichen, bedient man sich zumeist einer Symbolik mit konzentrischen Kreisen und spricht vom Ring- oder Schalenmodell der Software-Organisation (Abb. 3.3).

Die typischen Betriebssysteme der PCs nutzen aber nur die beiden äußeren Ringe (Ring 0 und Ring 3). Auch die Dienstfunktionen des Systems werden in Ring 3 ausgeführt. Mit anderen Worten, auch die ganz neumodischen Systeme sind im Grunde bei der herkömmlichen Lösung mit zwei Zuständen geblieben.

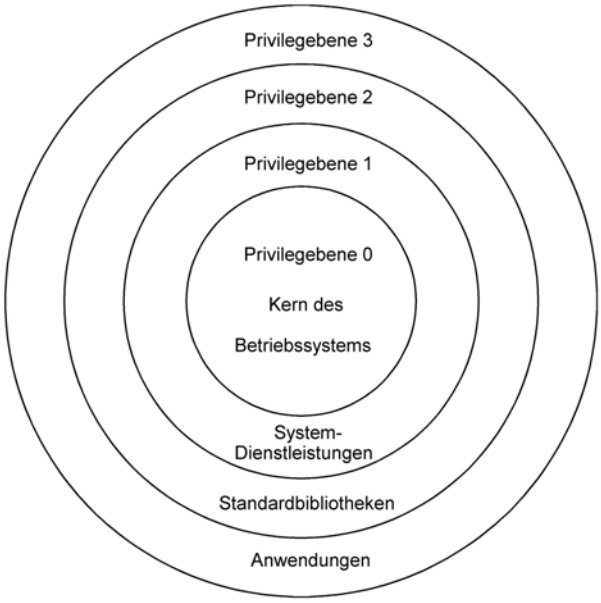


Abb. 3.3 Ring- oder Schalenmodell der Software-Organisation am Beispiel IA-32.