

1 Rechnerarchitektur

In den folgenden Abschnitten sollen zunächst die Grundlagen von Rechnern kurz skizziert und die Einzelkomponenten näher vorgestellt werden. Dabei soll versucht werden, von deutschen Bezeichnungen ausgehend die englischen Begriffe einzuführen.

1.1 Rechneraufbau: die Hardware

Das Grundprinzip eines Rechners hat sich seit seiner Erfindung und Entwicklung in den dreißiger Jahren des vorigen Jahrhunderts fast kaum verändert. Auch die miniaturisierte Variante, die als „Personal Computer PC“ Mitte der achtziger Jahre populär wurde und damit den Siegeszug in die Arbeitswelt angetreten hatte, wird trotz interner Feinheiten immer noch von dem gleichen Schema beherrscht.

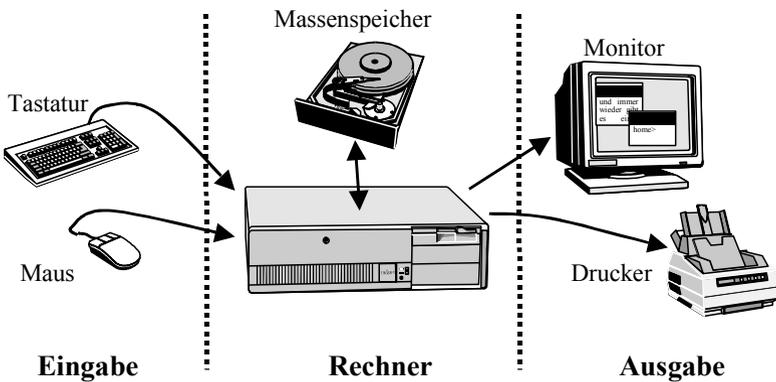


Abb. 1.1 Gliederung einer Computeranlage

Dieses Schema besteht aus Eingabegeräten, mit denen die Daten in den Rechner transportiert werden, aus Ausgabegeräten, mit denen die Ergebnisse angezeigt

werden können, und dem eigentlichen Rechner, dessen Speicher um einen Massenspeicher vergrößert wird. Betrachten wir dies genauer.

1.1.1 Die Eingabe

Die klassische Art und Weise, Texte und Befehle in einen Computer zu transferieren, ist die Tastatur. Üblicherweise existieren neben den Tasten für die reinen Buchstaben des Alphabets und den Tasten für Zahlen (Zehnertastatur) zusätzliche Tasten auf der Tastatur, die spezielle Kontrollfunktionen ausüben. Neben den Tasten \uparrow , \downarrow , \rightarrow , \leftarrow zum Verschieben des Aufmerksamkeitspunktes (Eingabemarkierung, *cursor*) auf dem Bildschirm gibt es noch Tasten für das Umschalten von Klein- auf Großbuchstaben (SHIFT), von einem Buchstabensatz auf einen anderen (ALTERNATE) und von Buchstabeneingabe auf Befehle (CONTROL).

Die Gesamtmenge der so erzeugten Tastenkombinationen (symbolische Eingaben) wird nun von einer unter der Tastatur befindlichen Elektronik (Tastaturprozessor) auf einen internen Zahlencode abgebildet, der im Rechner vom Betriebssystem in den offiziellen, international genormten Zahlencode für die Buchstaben umgesetzt wird. Bedingt durch die Herkunft der ersten Computer ist dies die amerikanische Alphabet, standardisiert in 128 Zeichen. Jedes Zeichen wird durch 7 Binärzeichen (**Bits**) aus einer Folge von „1“ und „0“ beschrieben gemäß dem *American Standard Code for Information Interchange ASCII*. In Abb. 1.2 sind zur Verdeutlichung einige Codes aufgeführt. Dabei ist der Binärcode, der Hexcode und der Dezimalwert der Codezahl neben dem eigentlichen Zeichen eingetragen. Man beachte, dass nur durch ein zusätzliches Bit (was einer Addition von $2^5 = 32$ zur Codezahl entspricht) aus einem großen A ein kleines wird. Ähnliches geschieht beim Halten der Control-Taste: hier wird ein Bit unterdrückt, so dass aus dem Code für S ein Control-S (DC3)-Zeichen wird.

Binärcode	Hexcode	Dezimalcode	Zeichen
0001 0001	1 1	1 7	Ctrl-Q (DC1)
0001 0010	1 3	1 9	Ctrl-S (DC3)
0011 0000	3 0	4 8	0
0011 0001	3 1	4 9	1
0100 0001	4 1	6 5	A
0100 0010	4 2	6 6	B
0110 0001	6 1	9 7	a
0110 0010	6 2	9 8	b

1010 0001	A 1	1 6 1	í
1010 0010	A 2	1 6 2	ó

Abb. 1.2 Einige ASCII-Buchstabenkodierungen

Der vollständige Code ist in Abb. 1.3 zu sehen.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0X	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	NL	VT	NP	CR	SO	SI
1X	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2X	SP	!	„	#	\$	%	&	'	()	*	+	,	-	.	/
3X	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4X	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5X	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6X	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7X	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Abb. 1.3 Die ASCII-Buchstabenkodierung

Außen an der Tabelle ist der Zahlencode in hexadezimaler Schreibweise notiert, bei der für 4 Bits jeweils eine von 16 Zahlen aus $\{0,1,\dots,8,9,A,B,C,D,E,F\}$ („Hexadezimalcode“) stehen. Der Wert der ersten vier binären Ziffern (Bits) bzw. ersten Hex-Zahl ist am linken Rand vertikal eingetragen, der der zweiten vier binären Ziffern horizontal am oberen Rand. Die beiden ersten Zeilen der Tabelle enthalten nur Kontrollzeichen wie Eingabeende EOT, Horizontaltabulator HT, Zeilenrücklauf CR usw. Nun besteht die Welt nicht nur aus Nordamerika, so dass sich bald die Notwendigkeit ergab, für eine verständliche Ausgabe auch europäische Zeichen wie ü, ä, æ und ê in den bestehenden Code aufzunehmen. Dies führte zu einem 8-Bit-ANSI-Code, der von der ISO (*International Standards Organization*) genormt wurde, beispielsweise *Latin-1* (ISO 8859-1) für Westeuropa und *Latin-2* für Osteuropa. Er besteht aus dem 7-Bit-ASCII-Code mit 128 zusätzlichen Zeichen, beispielsweise in Abb. 1.2 in den unteren, zusätzlichen Zeilen für die Zeichen mit dem Hexcode A1 und A2 notiert. In der Tabelle in Abb. 1.3 würden diese zusätzlichen 128 Zeichen durch 8 zusätzliche Zeilen beginnend mit 8X,9X,...,FX notiert werden.

Allerdings hört die Welt auch nicht bei Europa auf, so dass die Notwendigkeit, verschiedene Softwarepakete wie Texteditoren auch für den arabischen, chinesischen und indischen Markt zu schreiben, bald dazu führte, die Zeichenkodierung auf mehr als 8 Bit auszudehnen. Ein wichtiger Versuch in diese Richtung ist die Entwicklung eines „universellen“ Codes, des **Unicode**, in dem alle Schriftzeichen der Weltsprachen enthalten sind (UNI 1997). Zusätzlicher, freier Platz in der Codetabelle garantiert ihre Erweiterbarkeit. In Abb. 1.4 ist die Auslegung des Unicode, beginnend mit dem 16-Bit-Code 0000_H und endend mit $FFFF_H$, gezeigt. Man kann erkennen, dass der Unicode als Erweiterung des ASCII-Codes konzipiert

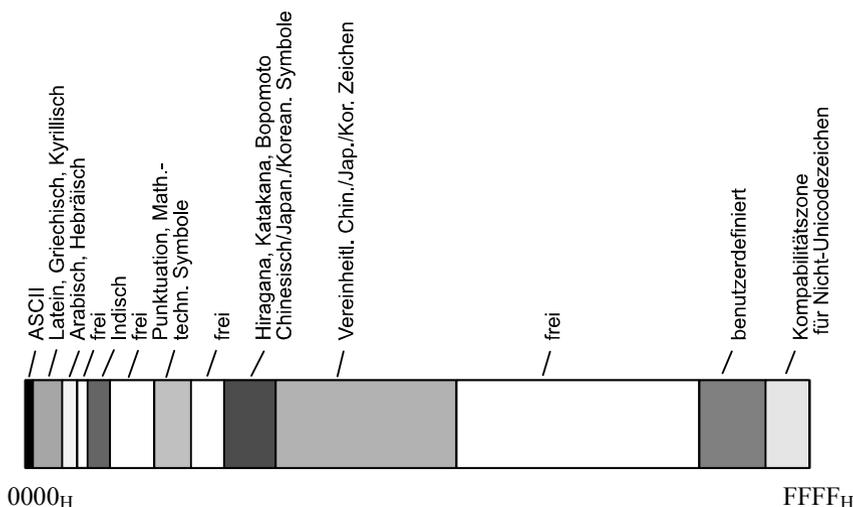


Abb. 1.4 Auslegung des Unicode

wurde, um Verträglichkeit mit den bestehenden Standardsystemen zu ermöglichen. Obwohl es sehr viele Schriftarten (*Fonts*) gibt, kann man beim Unicode mit einer 16-Bit-Kodierung auskommen, da die Information, *welches* Zeichen verwendet wird, streng von der Information getrennt wird, *wie* es verwendet wird. Alle Formatierungsinformationen (Schriftart, Darstellungsart fett/ schräg usw.) werden vom Zeichencode getrennt verwaltet und sind typisch für den jeweiligen Texteditor. Der Unicode ist genormt (ISO 10646) und wird ständig weiterentwickelt.

Ein weiterer wichtiger Code ist der 32-Bit-(4-Byte)-**Extended-UNIX-Code** EUC, der eine Multibyte-Erweiterung des UNIX(POSIX)-US-ASCII-Zeichensatzes darstellt. Asiatische und andere komplexe Schriftzeichen werden als Folge mehrerer ASCII-Buchstaben auf der normalen Tastatur eingegeben und als landesspezifischer EUC abgespeichert.

Beispiel UNIX Eingabecode

Seit dem Beginn wird in UNIX bzw. POSIX ausschließlich der US-ASCII-Zeichensatz verwendet. Erst in neueren Versionen wird auf nationale Besonderheiten eingegangen. Beispielsweise installiert die Firma Hewlett-Packard eine spezielle Erweiterung, das *Native Language Support* NLS. Es besteht zum einen aus einer Reihe von Umgebungsvariablen (LANG, LC_XX), die in der Prozessumgebung eines Benutzers, wenn sie korrekt gesetzt werden, den lokalen Teil der Bibliotheksroutinen und sprachspezifischen Kommandos (*ed*, *grep* usw.) des Betriebssystems steuern. Hierzu gehören nicht nur die sprachspezifischen Besonderheiten wie die richtige Verwendung von Multibyte-Codes bei

String-Vergleichen (z. B. ss = ß im Deutschen, LLL = LL im Spanischen) oder die Identifizierung von Buchstabencodes als Großbuchstaben, Zahlen, Kontrollzeichen usw., sondern auch die korrekte, landesübliche Anwendung von Punkt und Komma bei numerischen Ausdrücken, (z.B. 1.568,74 und dem Datumsformat wie 24.12.2004) sowie die Bezeichnung der Landeswährung.

Beispiel Windows NT *Eingabecode*

Im Gegensatz zu UNIX wurde in Windows NT bereits beim Design ein multinationaler Zeichencode für das Betriebssystem festgelegt, der Unicode. Dies bedeutet, dass alle Zeichenketten, Objektnamen, Pfade usw. nur Unicode-Zeichen verwenden. Es sind also auch chinesische Dateinamen in indischen Pfaden auf deutschen Computern möglich!

Die landestypischen Gegebenheiten wie Zeitzone, Währungsnotation, Sprache usw. wird unabhängig davon bei der Einrichtung des Systems festgelegt und zentral abgespeichert.

Neben der Tastatur für die Eingabe von Buchstaben gibt es heutzutage noch weitere, menschengerechtere Eingabemöglichkeiten.

- *Funktionstasten*

Eine einfache Erweiterung bietet die Möglichkeit, Tasten für besondere Funktionen bereitzustellen. Dies ist ein konzeptionell wichtiger Schritt, da hiermit neben der Eingabe alphanumerischer Daten auch das Ansprechen von Funktionen ermöglicht wird. Es ist deshalb logisch wichtig, die konzeptionell getrennten Daten und Funktionen im Programm auch beim Eingabemedium getrennt zu halten, um Verwechslungen zu vermeiden. Nur eine schlechte Benutzeroberfläche vermischt beide Funktionen, etwa das Anhalten der Ausgabe mit CTRL S oder, noch schlimmer, die Steuerung von Programmen durch einzelne Buchstabentasten. Beispielsweise kann eine harmlose Texteingabe DEF aus Versehen im Kommandomodus eingegeben als Abkürzung für „Delete“+„Erase“+„Format“ verheerende Auswirkungen haben.

- *Zeigegeräte*

Einen wichtigen Schritt zur analogen Eingabe bedeutete die Einführung von Zeigegeäten wie „Maus“ oder „Trackball“. Hier kann die Position eines Zeigers auf dem Bildschirm leichter kontrolliert werden als mit speziellen Funktionstasten (Cursortasten).

- *Grafische Tabletts*

Eine große Hilfe bedeutet auch die direkte Übermittlung der Position eines Stifts auf einem speziellen elektronischen Eingabebrett. Obwohl seine prinzipielle Funktion die eines Zeigegeäts ist, kann man damit auch Formen direkt in den Computer übertragen, z. B. Daten von biomedizinischen oder architektonischen Vorlagen, Funktionen und Kurven, oder es für Unterschriften zur Scheckverifikation verwenden.

- *Scanner*

Das grafische Tablett ist allerdings in den letzten Jahren bei den meisten Anwendungen durch hochauflösende, optoelektronische Abtastgeräte, die Scanner, ersetzt worden. Zusammen mit der Tendenz, Ergebnisse nicht mehr auf Papier zu speichern, sondern elektronisch und damit auch direkt verarbeitbar zu machen, haben sie sich auf diesem Gebiet durchgesetzt.

- *Spracheingabe*

Eine der benutzerfreundlichsten Eingabeschrittstellen ist zweifelsohne die Spracheingabe. Obwohl die benutzerabhängige Spracherkennung einzelner Worte in leiser Umgebung gut gelingt, lassen die Systeme für eine sprecherunabhängige, störste Spracherkennung allerdings noch viel zu wünschen übrig. Auch ihr Nutzen ist sehr umstritten: Zwar ist die Spracheingabe bei allen Menschen, die bei der Rechnerbedienung die Hände zur eigentlichen Arbeit frei haben müssen (z.B. Mikrochirurgen bei der Mikroskopsteuerung etc.) sehr beliebt, doch findet sie bei Büroangestellten mäßige Resonanz, da die Sprache zum einen nicht unbedingt genauer ist als ein Zeigeelement, und zum anderen dauerndes Reden beim Arbeiten störend sein kann.

1.1.1 Die Ausgabe: Rastergrafik und Skalierung

Neben der Standardausgabe von Texten auf Druckern wird für anspruchsvolle, grafisch ausgefeilte Texte und Grafiken oft ein spezielles Modell benutzt: das Modell einer **Rastergrafik**, das aus gleichmäßig in einem Raster angeordneten Bildpunkten (**Pixeln**) besteht. Es benutzt ein Koordinatensystem für Zeichnungen, wie es bei einem Bildschirmraster entsteht. Ausgangspunkt (0,0) der (x,y) Koordinate ist dabei die linke obere Ecke, wobei die Indizes der Bildpunkte (Pixel) ähnlich der einer transponierten Matrix angeordnet sind. Für einen Bildschirm mit 1024×768 Punkten ist dies in Abb. 1.5 zu sehen.

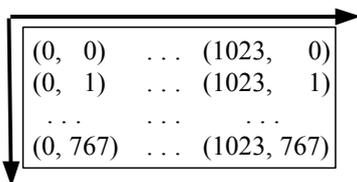


Abb. 1.5 Die Pixelkoordinaten der Rastergrafik

Das Rastergrafikmodell weist im Unterschied zu einer aus Linien bestehenden Grafik (**Vektorgrafik**), bei der die nur die Koordinaten (also z. B. als Anfangs- und Endpunkte der Linien) gespeichert werden, jedem Punkt einen definierten Farbwert zu, der als Zahl gespeichert wird. Der Bildwiederholpeicher der Raster-

grafik muss also, anders als bei der Vektorgrafik, für jeden Bildpunkt (Koordinate) des gesamten Bildschirms immer eine Speichereinheit vorsehen. Aus diesem Grund bevorzugte man früher Vektorgrafik, die sehr wenig Speicherplatz benötigt. Allerdings dauert das Neuzeichnen (*refresh*) komplizierter Grafiken durch das Neuzeichnen aller Grafikelemente in der Liste bei Vektorgrafiken sehr lange, was bei älteren Bildschirmen (phosphorbeschichtete Scheibe!) und komplexen Vektorgrafiken zu Flackern führte.

Die moderne Technologie der Rasterbildschirme (Fernseh- oder Flüssigkristallbildschirme mit Bildwiederholpeicher) ist stark mit dem Programmiermodell der Bildschirme verbunden und damit in die Konzeption der Grafiksoftware eingeflossen. Jeder Bildpunkt wird in seiner Farbe durch einen Farbwert (eine Zahl) aus n Bits (b_{n-1}, \dots, b_1, b_0) beschrieben. Vielfach können die Bits mit gleichem Index vom Displaycontroller hardwaremäßig extra zu einem Bild zusammengefasst werden. Ein solches Bild wird auch als **Ebene** bezeichnet.

Allerdings hat das Denken in Pixelflächen einen entscheidenden Nachteil: Die Auswirkungen der Funktionen sind sehr von der tatsächlich verfügbaren Hardware (Bildschirmauflösung etc.) abhängig. Beispielsweise können Vierecke und Fenster der aktuellen Bildschirmgröße leicht angepasst werden, nicht aber Schriften, deren Zeichen als feste Blöcke von Pixeln (*pixmaps*) definiert werden. Mischen wir nun Grafik (z. B. Fenster) und Text (z. B. Bezeichnungen), so ist die Lage und Größe des Textes in der Grafik von der Bildschirmauflösung abhängig, was zu sehr un schönen Effekten führen kann und eine hardwareunabhängige, auf viele Rechnerarten übertragbare Programme sehr erschwert.

Eine wichtige Alternative zu pixelorientierten Schriften sind deshalb die *skalierbaren* Schriften. Bei Ihnen wird ein Buchstabe nicht durch seine Bildpunkte (Pixel) sondern durch den Umriss und die Farbe bzw. Struktur (Textur) der darin enthaltenen Fläche charakterisiert. Dies ist zwar zunächst umständlich und ineffizienter bei der Speicherung und Darstellung, aber die Umrissbeschreibung lässt sich im Unterschied zu den Pixelfeldern beliebig vergrößern und verkleinern (Beispiel: *PostScript*, *TrueType*-Schriften). In Abb. 1.6 ist links eine pixelorientierte und rechts ein skalierbare Beschreibung des Buchstabens „A“ gezeigt. Die tatsächlich zu sehenden Bildpunkte sind jeweils grau schraffiert.

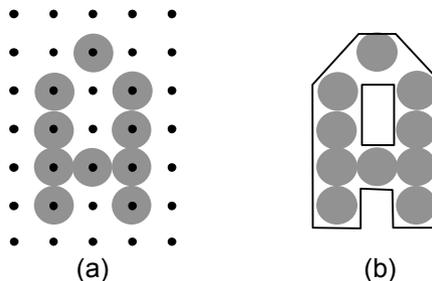


Abb. 1.6 Ein Buchstabe (a) im 5×7 -Raster und (b) als Umrandung

Eine solche Beschreibungsart aus Umriss und Textur ist zwar aufwendiger, aber sie lässt sich für alle wichtigen grafischen Elemente der Benutzeroberfläche (Balken, Kreise, Ikone usw.) einheitlich vorsehen und damit sehr effizient in den Grafikroutinen des Betriebssystems verankern sowie leicht Spezialprozessoren übertragen.

1.2 Die Architektur eines Rechners

Der Kasten „Rechner“ in Abb. 1.1 ist sehr allgemein. Möchten wir besser verstehen, wie ein Programm durch den Rechner ausgeführt wird, so müssen wir ihn in einzelne Funktionseinheiten aufgliedern. In Abb. 1.7 ist eine solche Gliederung vorgenommen. Unser einfaches Rechnermodell besteht nur aus einem Speicher, der in einzelnen Speichereinheiten die Befehle des Programms enthält, und einem Prozessor (*Central Processing Unit CPU*), der sie ausführt. Die Speichereinheiten können dadurch einzeln angesprochen (ausgelesen und gefüllt) werden, dass jeder Einheit eine eigene Zahl (Adresse) zugeordnet ist, ähnlich einer Hausnummer. Dabei haben wir aber verschiedene Implementierungsdetails vernachlässigt, die wichtig für das Verständnis der Programmiermöglichkeiten sind.

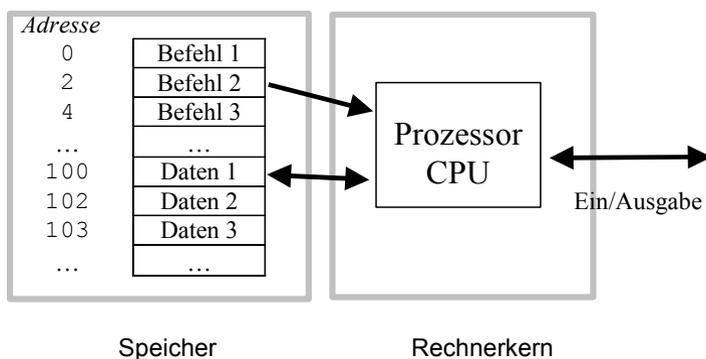


Abb. 1.7 Grobe funktionelle Aufgliederung der Recheneinheit

1.2.1 Busse

Die Adressen zum Ansprechen der Speichereinheiten werden über diskrete Leitungen zu der Speicherelektronik geschickt, ebenso die Daten und die Ein- und Ausgabe zu den Peripheriegeräten. Dies führte bei den ersten Rechnern zu einem ziemlichen Kabelwirrwarr zwischen den verschiedenen Einheiten. Die Abb. 1.8 gibt einen Eindruck davon.

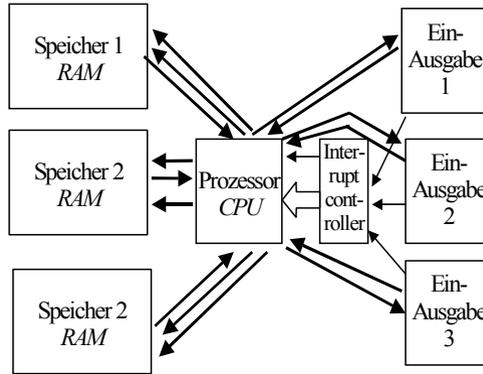


Abb. 1.8 Direkte Verbundarchitektur eines Rechners

Um die Anzahl der benötigten Leitungen und Stecker (und damit die Anzahl der Störungen!) zu verringern, fasste man nun alle Leitungen der gleichen Art zu einem gemeinsamen Transportweg (**Bus**) zusammen, an den alle Einheiten ohne Unterscheidung angeschlossen wurden, s. Abb. 1.9. Damit wurde es möglich, nicht nur den Speicher über den **Adressbus** anzusprechen, sondern auch die Ein- und Ausgabegeräte (**Peripheriegeräte**) fühlen sich angesprochen, wenn eine nur ihnen zugeordnete Adresse auf dem Adressbus erscheint. Dies wird als **memory-mapped input-output** bezeichnet und ist ein wichtiger Schritt zur Vereinheitlichung der Programmierung.

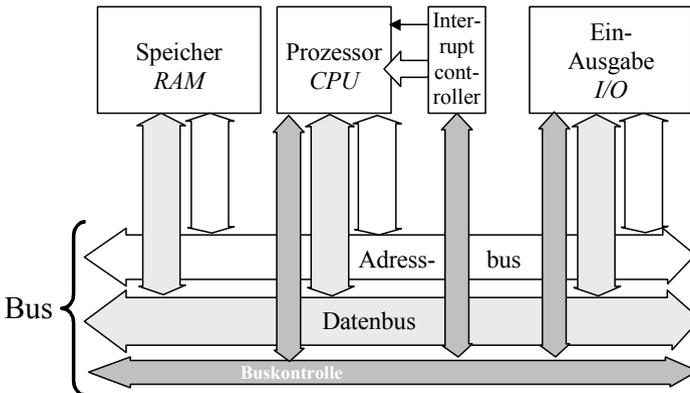


Abb. 1.9 Busarchitektur eines Rechners

Anstelle die Peripheriegeräte wie Magnetband, Konsole, Lochkartenstanzer, Bildschirm etc. durch besondere Befehle anzusprechen, lassen sich nun die üblichen Speicherbefehle verwenden. Schreibt man Daten auf eine Ausgabeadresse, so gibt sie das Gerät aus; die Eingabe besteht nur noch im Auslesen der Daten auf der Eingabeadresse.

Man sieht, dass eine modulare Erweiterung oder Modifikation des Rechners leicht möglich ist, ohne zusätzliche Anschlussprobleme. Jeder Bus besteht aus einzelnen Drähten, die meist zu einem Bandkabel zusammengefasst parallel von Einheit zu Einheit gehen. Jeder Draht führen ein elektrisches Signal, das entweder 0 Volt (bedeutet: binär *null*) oder 5 Volt (bedeutet: binär *eins*) ist. Bei einem 32 Drähte umfassenden Datenbus werden also zu einem Zeitpunkt ein Datenpaket aus 32 Bit oder 4 Byte transportiert.

1.2.2 Buskontrolle

Der selbe **Datenbus** dient zum Schreiben und zum Lesen; unterschieden wird nur durch ein Kontrollsignal, das extra parallel geführt wird und zur Buskontrolle (**bus control**) gehört. Gibt es zu wenige Leitungen, so dass man Adressen und Daten abwechselnd auf den selben Leitungen transportieren muss, so benötigt auch dieser **Multiplexbus** besondere Kontrollleitungen dafür.

Eine weitere Aufgabe der Buskontrolle besteht in der Koordination der unterschiedlichen Geräte mit dem Prozessor. Kann der Bus nicht nur zum Datentransport vom Prozessor genutzt werden, sondern auch von einem Gerät, etwa dem schnellen Massenspeicher aktiv angefordert und genutzt werden, so spricht man von einem **Multi-Master-Bus**. Die Abfolge, in der die Anfragen und Genehmigungen zur Buskontrolle ablaufen, wird als **Busprotokoll** bezeichnet. Sie sind für jeden Bus genau genormt und müssen von den Herstellern der Geräte bzw. der Einschubkarten, die Kontakt zum Bus haben, beachtet werden.

Interrupts und Prioritäten

Bisher haben wir vernachlässigt, dass die Operationen in den verschiedenen Einheiten nicht sequentiell ablaufen, sondern parallel: während ein Drucker arbeitet, kann prinzipiell der Prozessor weiter Befehle abarbeiten und z.B. Daten auf dem Massenspeicher lesen. Anstelle den Prozessor in einer Schleife aktiv warten zu lassen, bis der Drucker „fertig“ signalisiert (*polling, busy wait*), muss nur sichergestellt werden, dass der Prozessor durch ein Signal unterbrochen wird, wenn der Drucker fertig ist und neue Daten benötigt. Dieses Signal bezeichnet man als **Interrupt** (Unterbrechung), die dafür vorgesehenen Leitungen sind die **Interruptleitungen**.

Angenommen, ein Drucker ist fertig und wird gerade vom Prozessor bedient, als die Festplatte ebenfalls bedient werden will. Wer muss warten? Natürlich der Drucker, da er wesentlich langsamer ist; kommt die Festplatte nicht sofort dran, so

hat sie sich schon zu weit gedreht und das Schreiben/Lesen muss bis zur nächsten Umdrehung warten. Beim Drucker hingegen wirkt sich eine Verzögerung von einigen Millisekunden praktisch nicht aus. Aus diesem Grund ist jeder Interruptleitung zusätzlich noch eine **Interrupt-Priorität** (Wichtigkeit) zugeordnet nach der der Prozessor entscheidet. Sie ist vom Systemprogrammierer in Relation zu den anderen Geräten fest eingestellt.

Beispiel PC-Busse

In den handelsüblichen PCs gibt es verschiedene Bussysteme: Für die Massenspeicher (Festplatten, CD-ROM, DVD) wird meist der IDE (ATA)-Bus verwendet, der pro Buskabel nur zwei Geräte kennt, deren Rechte (*Master/Slave*) fest eingestellt werden müssen. Für die restliche Peripherie wurde für schnelle Geräte der SCSI-Bus, ein Multi-Master-Bus, verwendet; für langsamere Geräte und geringerem Aufwand wurde der ISA-Bus eingeführt.

Letzterer wurde dann Anfang der 90-Jahre durch den flexibleren Peripheral Component Interface (PCI)-Bus abgelöst. Ab der 66 MHz-Taktversion (PCI Version 2.2) besteht er aus einem kombinierten Adress/Datenbus, bei dem auf 64 Leitungen zuerst eine 64-Bit-Adresse und dann ein 64-Bit-Datenwort angelegt wird (Multiplexbus). Das abwechselnde Benutzen der selben Leitungen wird als „Zeitmultiplex“ bezeichnet.

- ◆ Der Kontrollbus besteht aus mehreren Komponenten: dem System Bus mit zwei Leitungen (*Clock/Reset*), dem Interface Control Bus mit sieben Leitungen (*Ready, Acknowledge, Stop,..*), dem Parity Bus mit zwei Leitungen, dem Fehler-Bus mit zwei Leitungen (*Parity, System*), dem Kommando-Bus, der 64 MHz-Kontrolle aus sechs Leitungen (*Enable, Running, Present, Acknowledge, Request*), der Cache-Kontrolle aus zwei Leitungen und dem Interrupt-Bus aus vier Leitungen (also max. $2^4 = 32$ Interruptgeräten). Dabei wurde kein fester Datentakt und Befehlstakt vorgegeben, sondern der Bus passt sich in seiner Geschwindigkeit (Taktrate) dem Gerät an, das ihn gerade benutzt. Die Notation „66 MHz“ gibt nur die maximale Taktrate an, die möglich ist.
- ◆ Außerdem gibt es noch einen Bus zur Spannungsversorgung mit +5, +3.3, +12, -12 Volt und GND (Masse)

Neuere Entwicklungen sehen einen Nachfolger des PCI-Busses, den PCI-Express, vor. Im Unterschied zu dem PCI-Bus, der in verschiedenen Versionen existiert und damit Probleme bei der Normierung der zahlreichen Anschluss-Pins hat, kommt die neue Version minimal mit nur zwei Anschlussdrähten aus. Die maximale Datengeschwindigkeit ist z. Z. 2,5 Gigabit pro Sekunde pro Richtung und wird prinzipiell nur noch durch die maximale Geschwindigkeit von 10 Gb/s pro Richtung begrenzt, die für eine Kupfer-Zweidrahtleitung gilt. Der Bus ist sehr flexibel: Bei der Initialisierung des Datenaustauschs zwischen zwei Geräten wird festgelegt, wie

viele parallele Drahtleitungen dafür verwendet werden. Über diese Leitungen werden dann alle Signale (Adressen, Daten, Kontrolle) im Zeitmultiplex-Verfahren gesendet. Natürlich erfordert eine solche Flexibilität entsprechend komplexe Chips (Kontroller), die den Bus bedienen können. Man spart also Leitungen durch erhöhte Anforderungen an die beteiligten Geräte. Da durch die moderne Chipfertigung die Chips stetig billiger werden, die Leitungsführung (Platzverbrauch, Energieverbrauch, Störungen, etc.) aber nicht, ist dieses Konzept für zukünftige Architekturen gut geeignet.

1.2.3 PC-Architektur

Für die heutige Architektur der Heimcomputer hat sich eine Architektur durchgesetzt, die eine Mischung aus einer diskreten Verbindungsstruktur wie in Abb. 1.8 und einer Busstruktur wie in Abb. 1.9 besteht. In Abb. 1.10 ist ein Überblick gezeigt.

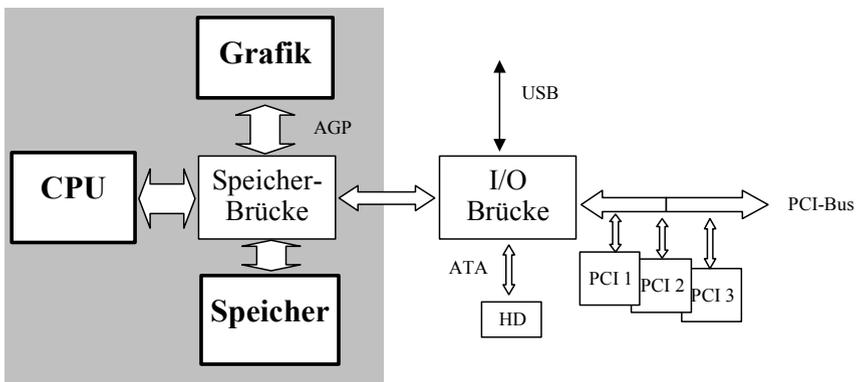


Abb. 1.10 Typische Architektur eines modernen PCs

Der Rechner gliedert sich dabei in zwei Teile: der Prozessor/Speicher-Konfiguration mit hohem Datenaustausch und schnellen Bussen, die sehr vom aktuellen Prozessortyp und Hersteller abhängig ist (schraffierter Bereich in Abb. 1.10), und der Peripheriekonfiguration, die geringeren Datentransfer und eine längere Systemlebensdauer hat. Dreht man obige Abbildung um 90° in Uhrzeigerichtung, so wird klar, warum die dann oben „im Norden“ gelegene Speicherbrücke als *Northbridge* und im Gegensatz dazu die I/O-Brücke als *Southbridge* bezeichnet werden.

Zukünftige Architekturen müssen noch die sehr datenintensiven Anwendungen wie Video und Netzwerke berücksichtigen, die einen parallelen Datentransfer vorsehen. In diesem Fall verschmelzen die Speicherbrücke und die I/O-Brücke mitein-

ander zu einer zentralen Brücke, die alle diskrete Datenpfade zwischen CPU, Speicher und I/O bereitstellt.

1.3 Maschinensprache und Prozessorstruktur

Die Computer gelten heutzutage zu Recht als „universelle Maschinen“. Im Unterschied zu allen anderen Maschinen werden sie nicht zu einem bestimmten Zweck gebaut, etwa ein Bankkonto zu verwalten oder eine Produktionssteuerung zu realisieren, sondern haben eine solch allgemeine, möglichst anwendungsunabhängige Struktur, dass sie in hohen Stückzahlen preiswert für (fast) alle Zwecke einsetzbar sind. Diese universelle Struktur hat sich in den letzten 50 Jahren fast nicht verändert, so dass es sich lohnt, einen tieferen Blick darauf zu werfen.

Ein Universalprozessor besteht im wesentlichen aus einem Speicher, in dem Befehle (Aktionen) aufgelistet sind, die der Prozessor in dieser Reihenfolge ausführen soll, und dem eigentlichen Prozessorkern, der die Befehle hintereinander ausliest und ausführt, siehe Abb. 1.7.

1.3.1 Ein einfaches Befehlsmodell

Die Befehle für den Prozessor werden als **Maschinenbefehle**, die Befehlssyntax und Semantik als **Maschinensprache** bezeichnet. Eine solche Maschinensprache können wir (in Anlehnung an existierende Prozessoren) uns zum besseren Verständnis selbst konstruieren. Das Aussehen einer solchen Anweisung soll einheitlich folgendes Format haben:

Befehl	Arg1	Arg2	..
--------	------	------	----

An Befehlen (**Instruktionen**) genügen uns anfangs nur folgende:

JUMP	A	Sprung zu einer Adresse A
JMPZ	A	jump_on_zero: Sprung zu Adresse A, wenn die Variable Z den Wert Z = TRUE hat.
TEST	A	Teste den Inhalt von A, ob er null ist. Wenn ja, setze die Variable Z = WAHR, sonst Z = FALSCH.
MOVE	A, B	Schiebe eine Kopie des Inhalts von B auf die Speicherstelle A und überschreibe damit den vorigen Inhalt von A. Dies entspricht der Zuordnung A := B .
ADD	A, B, C	Bilde B+C und speichere das Ergebnis auf A. Entspricht A := B+C. (Analog definiert für SUB, MUL, DIV)

Den Namen A einer Speicherstelle (*label*) notieren wir dabei mit „A:“ .

Die Anzahl der möglichen Befehle (Funktionen) dieser abstrakten Maschine, der Instruktionssatz, besteht also aus nur 8 Instruktionen (Es ist sogar möglich, mit weniger Instruktionen auszukommen: Wie? Übungsaufgabe!).

Für das Beispiel der Fakultätsberechnung (Bildung von $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$) für $n > 0$ sind im folgenden nun links die Blöcke der Hochsprache und rechts ihre Simulation aus den kleineren Blöcken aufgeschrieben. Übung: Was passiert, wenn $n = 0$ ist?

<code>read(n);</code>	<code>MOVE n,inp</code>	<code>n:=inp;</code>
<code>fac:=1;</code>	<code>MOVE fac,1</code>	<code>fac:=1;</code>
<code>FOR i:=2 TO n DO</code>	<code>MOVE i,2</code>	<code>i:=2;</code>
	<code>For:SUB R0,n,i</code>	<code>For: R0:=n-i;</code>
	<code>ADD R0,R0,1</code>	<code>R0:=R0+1;</code>
	<code>TEST R0</code>	<code>Z:=(R0=0);</code>
	<code>JMPZ EndFor</code>	<code>IF Z THEN GOTO EndFor</code>
<code>fac:=fac*i;</code>	<code>MUL fac,i, fac</code>	<code>fac:=fac*i;</code>
	<code>ADD i,1,i</code>	<code>i:=i+1;</code>
	<code>JUMP For</code>	<code>GOTO For</code>
	<code>EndFor:</code>	<code>EndFor:</code>
<code>write(fac);</code>	<code>MOVE out, fac</code>	<code>out:=fac;</code>

Abb. 1.11 Hochsprache, Assembler und Kommentar der Fakultätsberechnung

In diesem Beispiel sind Eingabe und Ausgabe nur symbolisch geschrieben worden: Um eine Zahl 42 einzugeben muss erst eine Konvertierung der Ziffernfolge (die Repräsentation der Zahl, etwa „4“, „2“), die über die alphanumerische Tastatur eingegeben wird, in eine Zahl (hier „42“) als Inhalt der Speicherzelle vorgenommen werden.

Die oben definierten Maschinenbefehle haben ein festes Format. Anstatt sie mit Buchstaben zu beschreiben (**Assemblercode**), kann man sie auch durchnummerieren, so dass im ersten Formatfeld auch eine Zahl stehen kann. Werden die Adressen der Speicherzellen ebenfalls mit einer Zahl („Hausnummer“) versehen, so besteht der gesamte Maschinenbefehl aus mehreren Zahlen, die durch Hintereinanderschreiben beispielsweise zu einer großen Zahl zusammengefasst werden können.

Ursprünglich war auf den ersten Rechnern nur die Programmierung als Liste von rein binären Zahlen (binärer Maschinencode) möglich, die man mittels Schalter eingeben (*eintogeln*) musste. Die Umstellung auf eine symbolische Schreibweise durch Buchstaben als menschenlesbares Programm wurde deshalb als große Erleichterung empfunden.

Der Assemblercode zeichnet sich also besonders dadurch aus, dass jedem Assemblerbefehl genau ein Maschinenbefehl entspricht, der von der Maschine direkt abgearbeitet wird. Je nach Maschine gibt es unterschiedlich komplexe Befehle, so dass Assemblercode abhängig von dem Prozessortyp ist. Beispielsweise lassen sich die arithmetischen Befehle mit dem Befehl TEST kombinieren, so dass bei

jeder Rechnung automatisch die 1-Bit Variable „Z“ gesetzt wird. Auch die Multiplikation bzw. Division wird meist von einem darauf spezialisierten Teil des Rechners (**Coprocessor**) unabhängig vom Hauptprozessor durchgeführt.

Wie wir sahen, ist es möglich, Befehle und Daten als Zahlen darzustellen und deshalb im selben (Zahlen-)Speicher unterzubringen. Dadurch ist es auch prinzipiell möglich, dass ein Rechner seinen eigenen Befehlscode als Daten ändert und damit ein neues Programm entsteht - eine wichtige Voraussetzung für die Funktion von Übersetzern (**Compiler**), die das Hochsprachenprogramm in Maschinencode transferieren!

1.3.2 Eine Prozessorgrundstruktur

Die Grundstruktur eines Prozessors sieht dazu folgendermaßen aus:

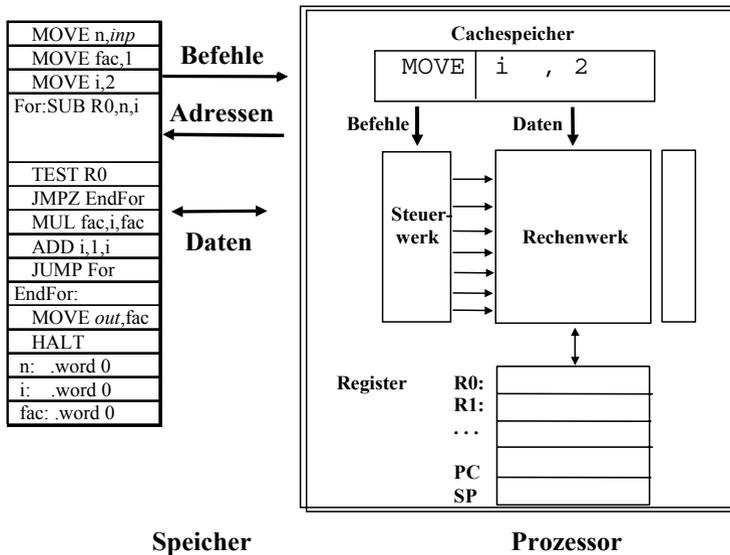


Abb. 1.12 Die Struktur eines von-Neumann-Prozessors

Die Befehle werden aus dem Speicher geholt und in einen besonderen, prozessorinternen Speicherplatz, den Cachespeicher. Dort wird der Befehl zerlegt in den Befehlscode(**Opcode**) und die Argumente (**Operanden**). Der Befehlscode selbst wird geladen, dekodiert und ausgeführt. Dabei ruft die eine Zahl des Befehls Aktivität auf einer Reihe von Steuerleitungen des **Steuerwerks** hervor, die je nach Befehl unterschiedlich kombiniert werden. Eine sequentielle Maschine, die wie die obige über die Adresse einen wahlfreien Zugriff (*random access*) auf jede Spei-

cherzelle hat, wird eine **Random Access Maschine (RAM)** genannt. Dies sollte aber nicht mit der für Speicher gebräuchlichen Abkürzung RAM (für Random Access Memory) verwechselt werden.

Typisch für einen Rechner mit **von-Neumann**(Princeton)-Architektur ist die Tatsache, dass Code und Daten im gleichen Speicher liegen. Im Unterschied dazu sind bei der **Harvard**-Architektur Befehle und Daten getrennt und können dadurch auch getrennt parallel eingelesen werden. Dieser Unterschied ist gerade für schnellere Rechner bedeutsam.

Im **Rechenwerk** werden die Daten bearbeitet; beispielsweise addiert oder getestet und die Variable Z gesetzt. Da dies eine logische Variable ist, kann sie durch ein Bit repräsentiert werden. Dieses Bit und noch weitere Bits für die Anzeige von Überlauf und anderen Zuständen, die in unserem einfachen Modell noch nicht aufgetaucht sind, werden in einem Bitfeld, dem **Statusregister**, zusammengefasst. Zusätzlich zu dem (prozessorexternen) Speicher gibt es noch einen Speicher für Hilfsvariable, in unserem obigen Beispiel „R0“, der sogar besonders schnell benutzt werden kann: die Register. Sie werden normalerweise in Adress- und Datenregister unterteilt, je nachdem, für was sie eingesetzt werden können.

Bei den **Adressregistern** ist eines ganz besonders wichtig: das des Programmzählers. Im Register des Programmzählers (**program counter PC**) wird die Speicheradresse des aktuellen Maschinenbefehls geführt. Dies dient nicht nur als „Le-sezeichen“, um sich die aktuelle Stelle im Programm zu merken, sondern kann auch als Referenz verwendet werden. Viele Adressen, die im Programm geführt werden, können damit dramatisch verkürzt werden: beispielsweise muss man für einen Sprung zu einem Befehl drei Adressen weiter als Zieladresse nur $PC+3$ angeben und nicht die volle Adresse. Dies nennt sich *relative* Adressierung.

Ein anderes wichtiges Register, das besonders bei Unterprogrammen nötig ist, ist der Stapelzeiger (**stack pointer SP**), der die aktuelle Adresse desjenigen Speichers enthält, auf dem die Parameter und die lokalen Variablen für Unterprogramme abgespeichert werden.

1.4 Prozessoren mit komplexem Befehlssatz (CISC)

Obwohl sich der prinzipielle Aufbau eines *von-Neumann*-Rechners in den letzten Jahrzehnten nicht geändert hat, gibt es doch einige Unterschiede in der Konzeption, bedingt durch neuere Hard- und Softwareerkenntnisse.

Eine wichtige Rolle spielt dabei die Verfügbarkeit preiswerter Speicher. In früheren Jahren waren die Speicher Bit für Bit noch durch die Magnetisierungsrichtung in Magnetkernen implementiert, die Stück für Stück in eine Drahtmatrix eingewebt werden mussten. Diese teure Bauweise bedeutete einen teuren und deshalb kleinen Hauptspeicher, so dass das Denken der Programmierer und der Prozessorhersteller darauf ausgerichtet war, möglichst wenig Speicher für ein Programm zu verbrauchen. Deshalb wurden die Prozessoren so gebaut, dass sie mächtige, kom-

plexe Befehle ausführen konnten (*Complex Instruction Set Computer, CISC*), von denen nur wenige benötigt wurden, um das Programm auszuführen. Damit konnte die semantische Lücke (*semantic gap*) zwischen Hochsprachen-Programm und Assembler-Programm verringert werden.

Die interne Architektur eines solchen CISC-Prozessors ist charakterisiert durch verschiedene Einheiten wie Rechenwerk, Steuerwerk, Adresswerk und Bussystem. In Abb. 1.13 ist dies als Erweiterung des Schemas von Abb. 1.12 gezeigt.

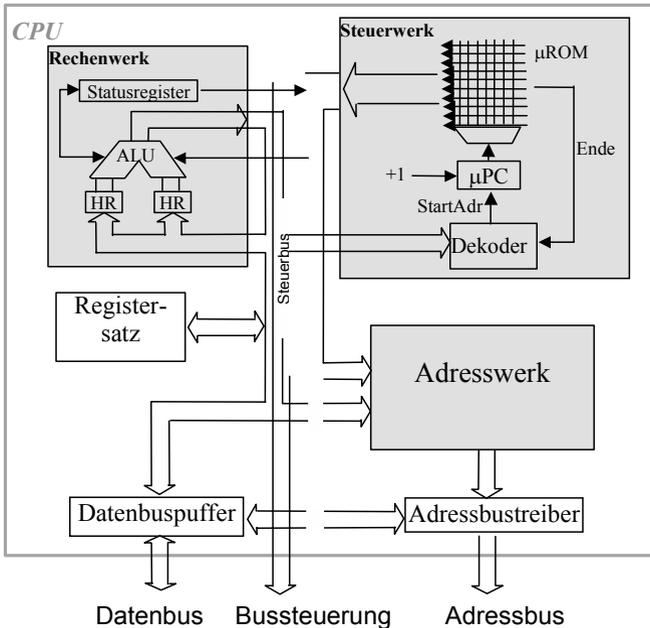


Abb. 1.13 CISC-Grundstruktur

Die Abarbeitung eines Befehls lässt sich in verschiedene Phasen unterteilen. Sie beginnen mit dem Holen (**instruction fetch IF**) des Befehls in einen Puffer im **Steuerwerk**. Dort wird der Befehl dann dekodiert (**instruction decode ID**). Durch die Dekodierung wird ein **Mikroprogramm** angestoßen, in dem der Programmzähler μPC auf die Adresse des ersten Mikro-Steuerworts (die Startadresse) für die Maschineninstruktion gesetzt wird. Alle Steuerworte der Mikroinstruktionen sind im **Mikro-ROM** abgespeichert. Alle weiteren Mikroinstruktionen werden dann durch Erhöhen des Zählers angesprochen bis zu einem Mikro-Steuerwort mit einem ENDE-Signal.

Die Steuerworte sind eine Zusammenfassung des Zustands aller benötigten Steuerleitungen beim jeweiligen Taktzeitpunkt; sie sind deshalb wesentlich breiter

(ca. 70-100 Bit) und ihr Zustand benötigt mehr Speicherplatz als die „normalen“ Speichereinheiten.

Die Hauptfunktionen der logischen und arithmetischen Funktionen werden im **Rechenwerk** durchgeführt. Hier ist ein Schaltnetz ALU (*Arithmetic Logical Unit*) implementiert, das sowohl zwei Binärzahlen addieren kann (Volladdierer!) als auch negieren oder logisch verknüpfen (z.B. bitweise UND, ODER) kann. Die jeweils gewünschte Teilfunktion des Netzes wird durch die Steuerleitungen aktiviert. Datenquelle und Datenziel ist immer ein bestimmtes Hilfsregister HR (der Akkumulator) sowie ein zweites Hilfsregister.

Die Adressrechnungen der Befehle (Addition von Basisadresse und *offset*, Verschiebung der Folgeadresse je nachdem, ob 8, 16 oder 32 Bit Einheiten angesprochen werden etc.) sowie der Speicherverwaltung, die in engem Verbund mit dem Betriebssystem durchgeführt wird, sind in einem besonderen Teil des Prozessors untergebracht, dem **Adresswerk**.

Die Befehle, Rechenoperanden und Adressdaten müssen zu und von den einzelnen Funktionseinheiten transportiert werden. Dies geschieht durch das prozessorinterne **Bussystem**. Dies vereinfacht zwar den Transport der Daten, schafft aber zusätzliche Engpässe (**Flaschenhals**), die nur über parallele, alternative Busse (*shortcuts*) ausgeglichen werden können, beispielsweise durch einen speziellen Befehlsbus, Ergebnisbus etc. Eine wichtige Maßnahme besteht dabei darin, den Daten- und Adressbus nicht auf den selben Leitungen mit einem Multiplexbus aus dem Chip zu führen, sondern sie wie in Abb. 1.13 physikalisch zu trennen (*Harvard-Architektur*). Die Steuerleitungen werden, obwohl sie als Bussystem gezeichnet sind, meist direkt zu den Einheiten geführt und bilden deshalb keinen Flaschenhals.

1.4.1 Charakteristik von CISC

Die bisher beschriebene Rechnerarchitektur lässt sich durch die Philosophie charakterisieren, in den Prozessoren umfangreiche, mächtige Maschinenbefehle zu implementieren, um die semantische Lücke zwischen den Hochsprachen (wie JAVA) und den einfachen Maschineninstruktionen zu schließen. Die Implementierung der Maschinenbefehle wird durch Mikroprogramme realisiert, wobei ein externer Takt, also ein Maschinenbefehl, durch viele interne Zyklen abgearbeitet wird. Der Hauptprozessor besteht also wie die russische „Puppe in der Puppe“ ebenfalls wieder aus einem Prozessor und Speicherplatz; ein Maschinenbefehl besteht aus mehreren Mikrobefehlen.

Die **Vorteile** dieses Konzepts sind:

- Verschiedene, binärkompatible Implementierungen (gleiche externe Architektur) in verschiedener Technologie und damit unterschiedlicher Ausführungsgeschwindigkeit (**Rechnerfamilien**) sind möglich

- Leichte Änderbarkeit des Befehlssatzes (Ergänzungen). Im Extremfall ist der Befehlssatz ladbar, so dass er auf die Applikationen des Benutzers zugeschnitten werden kann
- Kurze Programme resultiert in geringem Speicherbedarf
- Schnelle Befehlsabarbeitung bei langsamem, externen Speicher
- Eine vertikale Migration (Integration) von Betriebssystemprimitiven (Speicherverwaltung, Bitblock-, *string*-Befehle, etc.) ist durch die Änderung des Mikrocodes jederzeit leicht möglich.
- Durch die Ähnlichkeit der Hochsprache und des Maschinencodes sind einfache Compiler möglich.

Allerdings bringt dieses Konzept auch **Nachteile** mit sich, die nicht verschwiegen werden sollen:

- Eine große Anzahl von Mikroinstruktionen im Mikro-ROM (z.B. 51Kilobit beim 80386) bedeutet einen hohen Chip-Flächenbedarf des Steuerwerks (>50%), z.B. beim MC68020: 68%. Dies führte beim Intel iAPX432 (einem objekt-orientierten Prozessor) dazu, das Steuerwerk auf einen zweiten Chip auszulagern.
- Hohe Prozessor-Komplexität (Zahl der Transistorfunktionen) ist nötig. Dies bedeutet eine geringe Ausbeute bei der Chipproduktion, da bei gleicher Ausfallwahrscheinlichkeit pro Transistorfunktion mehr komplexe Chips defekt sind als einfache Chips. Da die defekten Chips weggeworfen werden müssen, ergeben sich insgesamt hohe Produktionskosten
- Die Anzahl möglicher Instruktionen ist sehr unübersichtlich und dabei einfach zu groß (beim MC68000 mehr als 1000), um vom Compilerbauer adäquat berücksichtigt werden zu können. Dies trifft besonders auf ladbare Mikroprogramme zu, die deshalb in den seltensten Fällen ausgenutzt werden.
- Die Komplexität der Befehle ist sehr verschieden, so dass die Abarbeitung unterschiedlicher und verschieden langer Befehle (iAPX432: 6-321 Bit!) sehr unterschiedliche Zeitspannen dauert. Dies erschwert die gleichmäßige Auslastung der Prozessoruntereinheiten erheblich.

1.4.2 Prozessoren mit reduziertem Befehlssatz (RISC)

Aus diesen Gründen wuchs in den 70-er Jahren an verschiedenen Orten die Erkenntnis, dass man so nicht weitermachen konnte. Am IBM T.J. Watson Forschungszentrum untersuchte dazu eine Gruppe um John Cocke ein Reihe von Programmen für das IBM 360 Modell. Sie fanden, dass von ca. 200 möglichen Befehlen nur 10 Befehle bereits 80% des Codes ausmachte; mit 30 Befehlen hatte man 99% erfasst. Außerdem bemerkten sie, dass man nicht nur einige komplexe In-

struktionen durch eine Folge von einfachen Instruktionen ersetzen konnte, sondern dass diese Folge auch schneller abgearbeitet wurde als der einzelne, komplexe Befehl. Dies führte zu der Idee, einen Rechner zu bauen, bei dem nicht viele komplexe Befehle, sondern nur wenige einfache Befehle vorhanden sind, wobei möglichst pro Takt ein Befehl abgearbeitet werden soll. Sie entwickelten darauf das IBM Modell 801, das nur 120 einfache Befehle kannte und erreichten mit einem darauf abgestimmten Compiler eine mittlere Rate von 1,1 Takten pro Befehl.

Diese Arbeit wurde von einer Gruppe um D.A. Patterson (Berkeley) und einer Gruppe um J. Hennessy (Stanford) aufgegriffen und systematisiert. Der Rechner von Berkeley wurde „**Reduced Instruction Set Computer**“ (**RISC**) genannt; die Modelle RISC I und RISC II (39 Befehle) führten viele Ideen dieses Prozessortypus ein.

Die Generation von RISC Rechnern lässt sich durch die Idee charakterisieren, in jedem Takt möglichst einen Befehl auszuführen. Folgerichtig enthält ein RISC Rechner im Steuerwerk auch keinen Mikrocode mehr, sondern ein festes Schaltnetzwerk, das weniger als 10% der Prozessorfläche belegt. Spezielle Compiler optimieren die Umsetzung der Hochsprachenkonstrukte auf Maschinencode. Der Maschinenbefehlssatz enthält deshalb nur wenige Befehle, die zum einen *unabhängig* voneinander sind (ein Befehl kann nicht durch eine Folge anderer ersetzt werden) und zum anderen *beliebig* mit den wenigen Adressierungsarten *kombiniert* werden können. Ein solcher Befehlssatz heißt **orthogonaler, regelmäßiger Befehlssatz**.

Es lassen sich einige Architekturmerkmale angeben, die typischerweise in RISC Prozessoren verwendet werden. In Klammern ist jeweils die Definition von Tabak (1987) angegeben.

Typisch RISC (Tabak, 1990)

- Es gibt nur 30-100 Maschinenbefehle (<50)
- Pro Takt wird ein Befehl abgearbeitet, außer bei den LOAD/STORE Befehlen, die auf den Hauptspeicher zugreifen.
- Es existieren nur wenige Adressierungsarten (*max.* 3)
- Meist ist das 3-Adressformat gegeben. Beispiel: `SUB R0, R1, R2`
- Es gibt nur wenige Befehlsformate (*max.* 3) (für 8, 16 und 32 Bit Operanden)
- Alle Befehle (außer LOAD/STORE) haben nur Register als Operanden, so dass ein Speicherzugriff nur mit den LOAD/STORE Befehlen möglich ist (**LOAD/STORE Architektur**)
- Es gibt sehr viele Register (>32), die meist zu Registerfeldern (**Registerfiles**) zusammengefasst werden.

Außerdem wurden meist 32 Bit (inzwischen 64 und 126 Bit) als Breite der internen und externen Datenwege gewählt. Zur weiteren Erhöhung des Durchsatzes

sind Adress- und Datenbus sowie Steuerleitungen getrennt herausgeführt: ein Merkmal der *Harvard-Architektur*.

Allerdings ist der Unterschied zwischen CISC und RISC in der Praxis nicht so stark wie bisher ausgeführt wurde. Viele Mechanismen, die zur Parallelisierung und Beschleunigung der Befehlsabarbeitung in RISC Rechnern dienen, sind bereits in CISC Architekturen eingeflossen. Die aktuellen Prozessoren sind häufig eine Mischform zwischen beiden Extremen (z.B. Schaltnetzwerk für Standardinstruktionen, Mikrocode für wenig benutzte Instruktionen); „reine“ Architekturen sind selten. Meistens versucht man, einen Kompromiss zwischen beiden Architekturen zu finden, bei dem möglichst viele Vorteile beider Konzepte erreicht werden.

1.5 Rechnerbetrieb: Die Software

In früheren Zeiten waren die Rechner zu jedem Zeitpunkt für nur eine Hauptaufgabe bestimmt. Alle Programme wurden zu einem Paket geschnürt und liefen nacheinander durch (**Stapelverarbeitung** oder *Batch-Betrieb*). Üblicherweise gibt es heutzutage aber nicht nur ein Programm auf einem Rechner, sondern mehrere (**Mehrprogrammbetrieb**, *multi-tasking*). Auch gibt es nicht nur einen Benutzer (*single user*), sondern mehrere (**Mehrbenutzerbetrieb**, *multi-user*).

Dazu ist der Speicher aufgeteilt zwischen allen Programmen und den allen gemeinsamen Systemprozeduren, dem **Betriebssystem**, siehe Abb. 1.14.

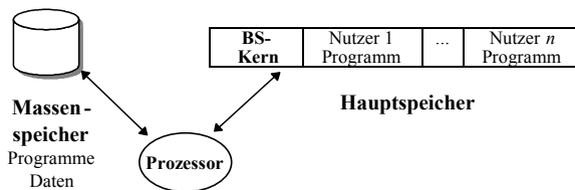


Abb. 1.14 Schema eines Computersystems

Hier dient der Massenspeicher nur dazu, als preiswertes, aber langsames Medium alle diejenigen Daten und Befehle bzw. Programmteile aufzunehmen, die nicht mehr in den Hauptspeicher hineinpassen und gerade nicht benötigt werden.

Um Konflikte zwischen ihnen bei der Benutzung des Rechners zu vermeiden, muss die Verteilung des Speichers und anderer Ressourcen („**Betriebsmittel**“), auf die Programme geregelt werden. Dies spart außerdem noch Rechnerzeit und erniedrigt damit die Bearbeitungszeiten. Beispielsweise kann die Zuteilung des Hauptprozessors für den Ausdruck von Text parallel zu einer Textverarbeitung so geregelt werden, dass die Textverarbeitung die CPU in der Zeit erhält, in der

Drucker ein Zeichen ausdruckt. Ist dies erledigt, schiebt der Prozessor ein neues Zeichen dem Drucker nach und arbeitet dann weiter an der Textverarbeitung.

Zusätzlich zu jedem Programm muss also gespeichert werden, welche Betriebsmittel es benötigt: Speicherplatz, CPU-Zeit, CPU-Inhalt etc. Die gesamte Zustandsinformation der Betriebsmittel für ein Programm wird als eine Einheit angesehen und als **Prozess** (*task*) bezeichnet (Abb. 1.15).

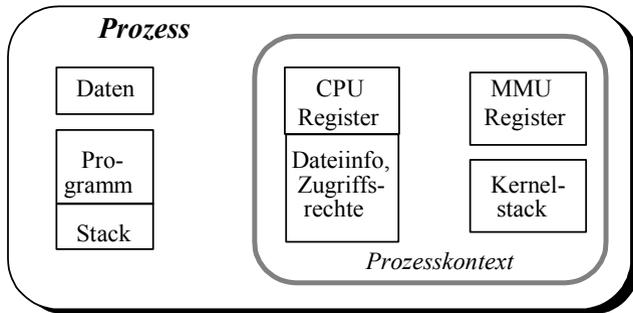


Abb. 1.15 Zusammensetzung der Prozessdaten

Ein Prozess kann auch einen anderen Prozess erzeugen, wobei der erzeugende Prozess als **Elternprozess** und der erzeugte Prozess als **Kindsprozess** bezeichnet wird.

Im Unterschied zu dem Maschinencode werden die Zustandsdaten der Hardware (CPU, FPU, MMU), mit denen der Prozess arbeitet, als **Prozesskontext** bezeichnet, s. Abb. 1.15. Der Teil der Daten, der bei einem blockierten Prozess den letzten Zustand der CPU enthält und damit wie ein Abbild der CPU ist, kann als *virtueller Prozessor* angesehen werden und muss bei einer Umschaltung zu einem anderen Prozess bzw. Kontext (*context switch*) neu geladen werden.

Ein Mehrprogrammssystem (*multiprogramming system*) erlaubt das „gleichzeitige“ Ausführen mehrerer Programme und damit mehrerer Prozesse (Mehrprozesssystem, *multi-tasking system*). Ein Programm (**Job**) kann dabei auch selbst mehrere Prozesse erzeugen.

Zusätzlich zu dem Zustand „aktiv“ (*running*) für den einen, aktuellen Prozess müssen wir noch unterscheiden, worauf die anderen Prozesse warten. Für jede der zahlreichen Ereignismöglichkeiten gibt es meist eine eigene Warteschlange, in der die Prozesse einsortiert werden.

Ein blockierter Prozess kann darauf warten,

- aktiv den Prozessor zu erhalten, ist aber sonst bereit (*ready*),
- eine Nachricht (*message*) von einem anderen Prozess zu erhalten,
- ein Signal von einem Zeitgeber (*timer*) zu erhalten,

- Daten eines Ein/Ausgabegeräts zu erhalten (*io*).

Üblicherweise ist der *bereit*-Zustand besonders ausgezeichnet: Alle Prozesse, die Ereignisse erhalten und so entblockt werden, werden zunächst in die *bereit*-Liste (*ready-queue*) verschoben und erhalten dann in der Reihenfolge den Prozessor. Die Zustände und ihre Übergänge sind in Abb. 1.16 skizziert.

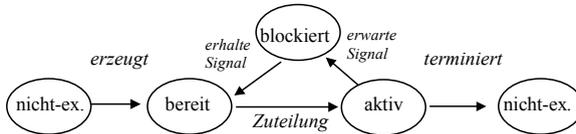


Abb. 1.16 Prozesszustände und Übergänge

Alle Zustände enthalten eine oder mehrere Warteschlangen (Listen), in die die Prozesse mit diesem Zustand eingetragen werden. Es ist klar, dass ein Prozess immer nur in einer Liste enthalten sein kann.

Die verschiedenen Betriebssysteme differieren in der Zahl der Ereignisse, auf die gewartet werden kann, und der Anzahl und Typen von Warteschlangen, in denen gewartet werden kann. Sie unterscheiden sich auch darin, welche Strategien sie für das Erzeugen und Terminieren von Prozessen sowie die Zuteilung und Einordnung in Wartelisten vorsehen.

Programme und damit die Prozesse existieren nicht ewig, sondern werden irgendwann erzeugt und auch beendet. Dabei verwalten die Prozesse aus Sicherheitsgründen sich nicht selbst, sondern das Einordnen in die Warteschlangen wird von einer besonderen Instanz des Betriebssystems, dem **Scheduler**, nach einer Strategie geplant. Bei einigen Betriebssystemen gibt es darüber hinaus eine eigene Instanz, den **Dispatcher**, der das eigentliche Überführen von einem Zustand in den nächsten bewirkt. Das Einordnen in eine Warteschlange, die Zustellung der Signale und das Abspeichern der Prozessdaten werden also von einer zentralen Instanz erledigt, die der Benutzer nicht direkt steuern kann. Statt dessen werden über die Betriebssystemaufrufe die Wünsche der Prozesse angemeldet, denen im Rahmen der Betriebsmittelverwaltung vom Scheduler mit Rücksicht auf andere Benutzer entsprochen wird.

Im einfachsten Fall können die Prozesse so lange laufen, bis sie von sich aus den Aktivzustand verlassen und auf ein Ereignis (I/O, Nachricht etc.) warten, die Kontrolle an andere Prozesse abgeben oder sich selbst beenden: Sie werden nicht vorzeitig unterbrochen (**non-preemptive scheduling**). Diese Art von Scheduling ist bei allen Systemen sinnvoll, bei denen man genau weiß, welche Prozesse existieren und welche Charakteristika sie haben. Ein Beispiel dafür ist das oben erwähnte Datenbankprogramm, das genau weiß, wie lange eine Transaktion normalerweise dauert.

In einem normalen Mehrbenutzersystem, bei dem die verschiedenen Jobs von verschiedenen Benutzern gestartet werden, gibt es aber zwangsläufig Ärger, wenn ein Job alle anderen blockiert. Hier ist ein anderes Scheduling erforderlich, bei dem jeder Job vorzeitig unterbrochen werden kann: das **preemptive Scheduling**.

Eine der wichtigsten Maßnahmen dieses Verfahrens besteht darin, die verfügbare Zeitspanne für das Betriebsmittel, meist die CPU, in einzelne, gleich große Zeitabschnitte (**Zeitscheiben**) aufzuteilen. Wird ein Prozess bereit, so wird er in die Warteschlange an einer Stelle einsortiert. Zu Beginn einer neuen Zeitscheibe wird der Dispatcher per Zeitinterrupt aufgerufen, der bisher laufende Prozess wird abgebrochen und wie ein neuer *bereit*-Prozess in die Warteschlange eingereiht. Dann wird der Prozess am Anfang der Schlange in den Aktivzustand versetzt. Dies ist in Abb. 1.17 dargestellt. Die senkrechten Striche symbolisieren die Prozesse, die von links in das „Gefäß“, die Warteschlange, hineingeschoben werden. Die Bearbeitungseinheit, hier der Prozessor, ist als Ellipse visualisiert. Nach einem Abbruch wird der Prozess an einen Platz zwischen die anderen Prozesse in der Warteschlange eingereiht.

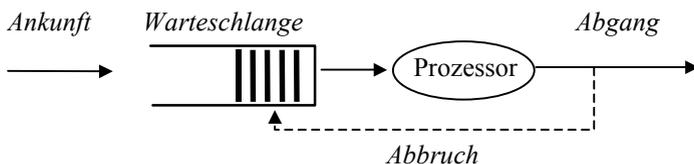


Abb. 1.17 Preemptives Scheduling

1.5.1 Das Betriebssystem

Historisch gesehen enthält ein Betriebssystem alle Programme und Programmteile, die nötig sind, einen Rechner für verschiedene Anwendungen zu betreiben. Die Meinungen, was alles in einem Betriebssystem enthalten sein sollte, gehen allerdings weit auseinander. Benutzt jemand einen Rechner nur zur Textverarbeitung, so erwartet er oder sie, dass der Rechner alle Funktionen der Anwendung „Textverarbeitung“ beherrscht.

Betrachten wir mehrere Anwendungsprogramme, so finden wir gemeinsame Aufgaben, die alle Programme mit den entsprechenden Funktionen abdecken müssen. Statt jedes mal „das Rad neu zu erfinden“, lassen sich diese gemeinsamen Funktionen auslagern und als „zum Rechner gehörige“ Standardsoftware ansehen, die bereits beim Kauf mitgeliefert wird. Dabei beziehen sich die Funktionen sowohl auf Hardwareeinheiten wie Prozessor, Speicher, Ein- und Ausgabegeräte, als auch auf logische (Software-) Einheiten wie Dateien, Benutzerprogramme usw. Das Betriebssystem ist also die Software (Programmteile), die für den Betrieb eines Rechners anwendungs-unabhängig notwendig ist.

Dabei ist allerdings die Interpretation von „anwendungs-unabhängig“ (es gibt keinen anwendungs-unabhängigen Rechnerbetrieb: Dies wäre ein nutzloser Rechner) und „notwendig“ sehr subjektiv (sind Fenster und Mäuse notwendig?) und lädt zu neuen Spekulationen ein.

Es gibt nicht *das* Betriebssystem schlechthin, sondern nur eine den Forderungen der Anwenderprogramme entsprechende Unterstützung, die von der benutzerdefinierten Konfiguration abhängig ist und sich im Laufe der Zeit stark gewandelt hat. Gehörte früher nur die Prozessor-, Speicher- und Ein-/Ausgabeverwaltung zum Betriebssystem, so werden heute auch eine grafische Benutzeroberfläche mit verschiedenen Schriftarten und -größen (Fonts) sowie Netzwerkfunktionen verlangt. Einen guten Hinweis auf den Umfang eines Betriebssystems bietet die Auslieferungsliste eines anwendungs-unabhängigen Rechnersystems mit allen darauf vermerkten Softwarekomponenten.

Die Beziehungen der Programmteile eines Rechners lassen sich durch das Diagramm in Abb. 1.18 visualisieren.

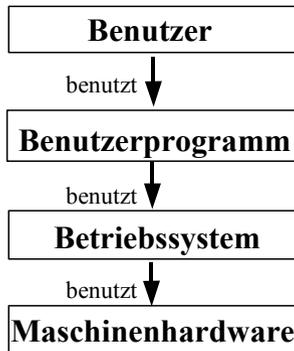


Abb. 1.18 Benutzungsrelationen von Programmteilen

Dies lässt sich auch kompakter zeichnen: in Abb. 1.19 links als **Schichtensystem** und rechts als System konzentrischer Kreise („Zwiebelschalen“).

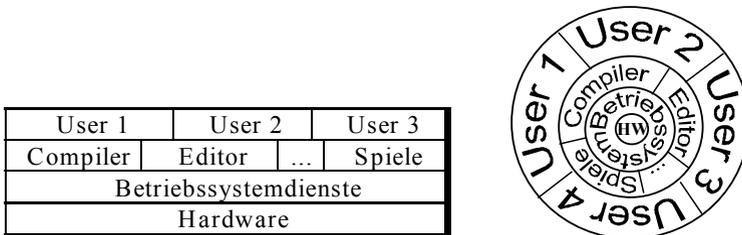


Abb. 1.19 Schichtenmodell und Zwiebelschalenmodell

Dabei betont die Darstellung als Schichtenmodell den Aspekt der Basis, auf der aufgebaut wird, während das Zwiebschalenmodell eher Aspekte wie „Abgeschlossenheit“ und „Sichtbarkeit“ von „inneren“ und „äußeren“ Schichten visualisieren.

Das Betriebssystem (*operating system*) als Gesamtheit aller Software, die für den anwendungsunabhängigen Betrieb des Rechners notwendig ist, enthält

- *Dienstprogramme, Werkzeuge*: oft benutzte Programme wie Editor, ...
- *Übersetzungsprogramme*: Interpreter, Compiler, Translator, ...
- *Organisationsprogramme*: Speicher-, Prozessor-, Geräte-, Netzverwaltung.
- *Benutzerschnittstelle*: textuelle und graphische Interaktion mit dem Benutzer.

Da ein vollständiges Betriebssystem inzwischen mehrere Gigabyte umfassen kann, werden aus diesem Reservoir nur die sehr oft benutzten Funktionen als **Betriebssystemkern** in den Hauptspeicher geladen. In Abb. 1.20 ist die Lage des Betriebssystemkerns innerhalb der Schichtung eines Gesamtsystems gezeigt.

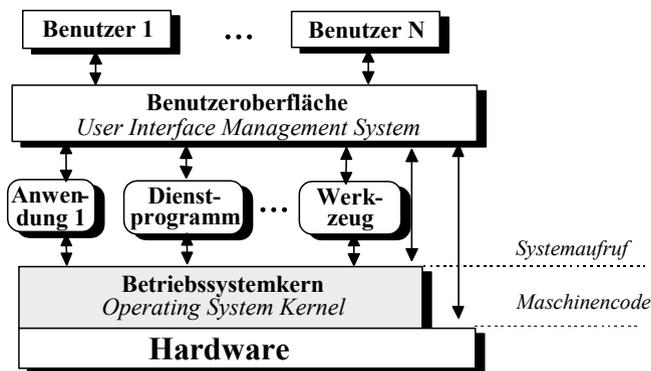


Abb. 1.20 Überblick über die Rechnersoftwarestruktur

Der Kern umfasst also alle Dienste, die immer präsent sein müssen, wie z. B. große Teile der Prozessor-, Speicher- und Geräteverwaltung (**Treiber**) und wichtige Teile der Netzverwaltung.

1.1.2 Beispiel UNIX

In UNIX gibt es traditionellerweise als Benutzeroberfläche einen Kommandointerpreter, die **Shell**. Von dort werden alle Benutzerprogramme sowie die Systemprogramme, die zum Betriebssystem gehören, gestartet. Die Kommunikation zwischen Benutzer und Betriebssystem geschieht durch Ein- und Ausgabekanäle; im

einfachsten Fall sind dies die Eingabe von Zeichen durch die Tastatur und die Ausgabe auf dem Terminal. In Abb. 1.21 ist das Grundscheema gezeigt.

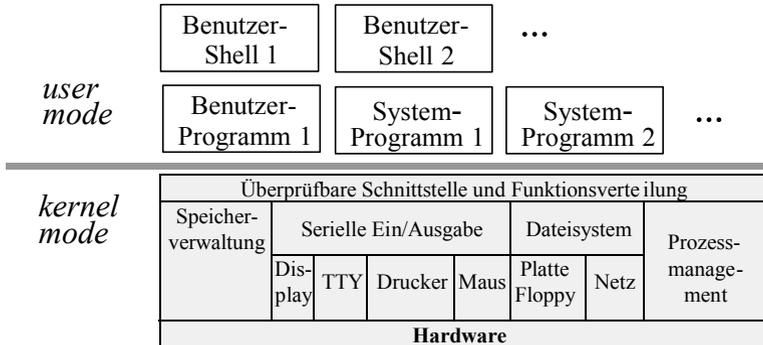


Abb. 1.21 UNIX-Schichten

Das UNIX-System wies gegenüber den damals verfügbaren Systemen verschiedene Vorteile auf. So ist es nicht nur ein Betriebssystem, das mehrere Benutzer (*Multi-user*) gleichzeitig unterstützt, sondern auch mehrere Programme (*Multi-programming*) gleichzeitig ausführen konnte. Zusammen mit der Tatsache, dass die Quellen den Universitäten fast kostenlos zur Verfügung gestellt wurden, wurde es bei allen Universitäten Standard und dort weiterentwickelt. Durch die überwiegende Implementierung mittels der „höheren“ Programmiersprache „C“ war es leicht veränderbar und konnte schnell auf eine andere Hardware übertragen (*portiert*) werden. Diese Faktoren machten es zum Betriebssystem der Wahl, als für die neuen RISC-Prozessoren schnell ein Betriebssystem benötigt wurde. Hatte man bei einem alten C-Compiler den Codegenerator für den neuen Instruktionssatz geändert, so war die Hauptarbeit zum Portieren des Betriebssystems bereits getan – alle Betriebssystemprogramme sowie große Teile des Kerns sind in C geschrieben und sind damit nach dem Kompilieren auf dem neuen Rechner lauffähig.

Allerdings gibt es trotzdem noch genügend Arbeit bei einer solchen Portierung. Die ersten Versionen von UNIX (Version 6 und 7) waren noch sehr abhängig von der Hardware, vor allem aber von der Wortbreite der CPU. In den folgenden Versionen (System IV und V sowie Berkeley UNIX) wurde zwar viel gelernt und korrigiert, aber bis heute ist die Portierbarkeit nicht problemlos.

Die Grundstruktur von UNIX differiert von Implementierung zu Implementierung. So sind Anzahl und Art der Systemaufrufe sehr variabel, was die Portierbarkeit der Benutzerprogramme zwischen den verschiedenen Versionen ziemlich behindert. Um dem abzuhelfen, wurden verschiedene Organisationen gegründet. Eine der bekanntesten ist die X/Open-Gruppe, ein Zusammenschluss verschiedener Firmen und Institutionen, die verschiedene Normen herausgab. Eine der ersten

Normen war die Definition der Anforderungen an ein portables UNIX-System (*Portable Operating System Interface based on UNIX: POSIX*), das als Menge verschiedener verfügbarer Systemdienste definiert wurde. Allerdings sind dabei nur Dienste, nicht die Systemaufrufe direkt definiert worden. Durch die auf X/Open übertragenen Rechte an dem Namen „UNIX“ konnte ein Zertifikationsprozess institutionalisiert werden, der eine bessere Normierung verspricht. Dabei wurde UNIX auch an das Client-Server-Modell angepasst: Es gibt eine Spezifikation für UNIX-98 Server und eine für UNIX-98 Client. Interessant ist, dass dabei die Bezeichnung „UNIX“ nur für eine Sammlung von verbindlichen Schnittstellen steht, nicht für eine Implementierung. Dies bedeutet, dass UNIX eigentlich als eine virtuelle und nicht als reelle Betriebssystemmaschine angesehen werden kann.

1.1.3 Beispiel Windows NT

Das Betriebssystem Windows NT der Firma Microsoft ist ein relativ modernes System; es wurde unter der Leitung von David Cutler, einem Betriebssystementwickler von VMS, RSX11-M und MicroVax der Fa. Digital, seit 1988 entwickelt. Das Projekt, mit dem Microsoft erstmals versuchte, ein professionelles Betriebssystem herzustellen, hatte verschiedenen Vorgaben zu genügen:

- Das Betriebssystem musste zu allen bisherigen Standards (MS-DOS, 16 Bit-Windows, UNIX, OS/2) kompatibel sein, um überhaupt am Markt akzeptiert zu werden.
- Es musste zuverlässig und robust sein, d. h. Programme dürfen weder sich gegenseitig noch das Betriebssystem schädigen können; auftretende Fehler dürfen nur begrenzte Auswirkungen haben.
- Es sollte auf verschiedene Hardwareplattformen leicht zu portieren sein.
- Es sollte nicht perfekt alles abdecken, sondern für die sich wandelnden Ansprüche leicht erweiterbar und änderbar sein.
- Sehr wichtig: Es sollte auch leistungsstark sein.

Betrachtet man diese Aussagen, so stellen sie eine Forderung nach der „eierlegenden Wollmilchsau“ dar. Die gleichzeitigen Forderungen von „kompatibel“, „zuverlässig“, „portabel“, „leistungsstark“ sind schon Widersprüche in sich: MS-DOS ist absolut nicht zuverlässig, portabel oder leistungsstark und überhaupt nicht kompatibel zu UNIX. Trotzdem erreichten die Entwickler ihr Ziel, indem sie Erfahrungen anderer Betriebssysteme nutzten und stark modularisierten. Die Schichtung und die Aufrufbeziehungen des Kerns in der ursprünglichen Konzeption sind in Abb. 1.22 gezeigt. Der gesamte Kern trägt den Namen *Windows NT Executive* und ist der Block unterhalb der *user mode/kernel mode*-Umschaltsschranke.

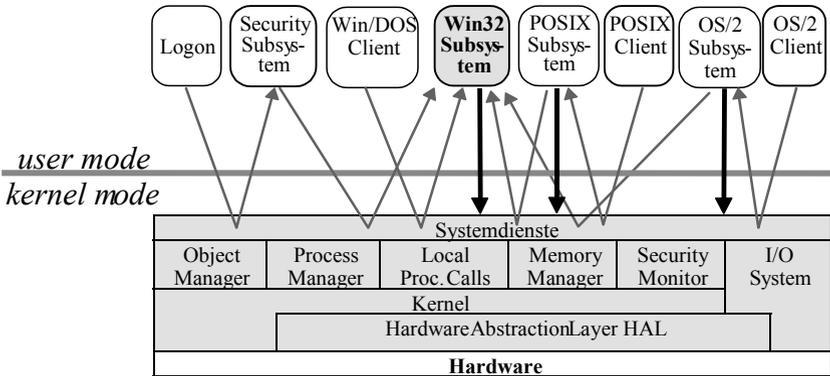


Abb. 1.22 Schichtung und Aufrufe bei Windows NT

Zur Lösung der Designproblematik seien hier einige Stichworte genannt:

- Die **Kompatibilität** wird erreicht, indem die Besonderheiten jedes der Betriebssysteme in ein eigenes Subsystem verlagert werden. Diese setzen als virtuelle Betriebssystemmaschinen auf den Dienstleistungen der NT Executive (Systemaufrufe, schwarze Pfeile in Abb. 1.22) auf.
Die zeichenorientierte Ein/Ausgabe wird an die Dienste des zentralen Win32-Moduls weitergeleitet. Die Dienste der Subsysteme werden durch Nachrichten (*local procedure calls* LPC, graue Pfeile in Abb. 1.22) von Benutzerprogrammen, ihren *Kunden (Clients)*, angefordert. Als Dienstleister (**Server**) haben sie also eine Client-Server-Beziehung.
- Die **Robustheit** wird durch rigorose Trennung der Programme voneinander und durch Bereitstellen spezieller Ablaufumgebungen („virtuelle DOS-Maschine VDM,“) für die MS-DOS/Windows Programme erreicht. Die Funktionen sind gleich, aber der direkte Zugriff auf Hardware wird unterbunden, so dass nur diejenigen alten Programme laufen können, die die Hardware-Ressourcen nicht aus Effizienzgründen direkt anzusprechen versuchen. Zusätzliche Maßnahmen wie ein fehlertolerantes Dateisystem und spezielle Sicherheitsmechanismen zur Zugriffskontrolle von Dateien, Netzwerken und Programmen unterstützen dieses Ziel.
- Die **Portierbarkeit**, Wartbarkeit und Erweiterbarkeit wird dadurch unterstützt, dass das gesamte Betriebssystem bis auf wenige Ausnahmen (z. B. in der Speicherverwaltung) in der Sprache C geschrieben, stark modularisiert und von Anfang an geschichtet ist. Eine spezielle Schicht HAL bildet als virtuelle Maschine allgemeine Hardware nach und reduziert bei der Portierung auf andere Prozessoren die notwendigen Änderungen auf wenige Module.

Die detaillierte Diskussion der oben geschilderten Lösungen würde zu weit führen; hier sei auf das Buch von Helen Custer (1993) verwiesen.

Obwohl in Windows NT einige wichtige Subsysteme wie z.B. das Sicherheitssystem nicht als Kernbestandteil, sondern als Prozess im *user mode* betrieben werden („Integrale Subsysteme“), ist interessanterweise seit Version 4.0 das Win32-Subsystem aus Effizienzgründen in den Kern verlagert worden, um die Zeit für die Systemaufrufe von Win32 zu NT Executive zu sparen. Auch die Unterstützung anderer Standards (z. B. des OS/2 HPFS Dateisystems ab Version 4.0) wurde mit fortschreitender Marktakzeptanz von Windows NT eingestellt. In Version 5, genannt Windows 2000, wurden zusätzlich viele Dienstprogramme zur Netzdateiverwaltung und Sicherheit integriert. Dies ließ den Umfang von 8 Mill. Codezeilen auf über 40 Mill. anschwellen, was an die Zuverlässigkeit und Testumgebung der Betriebssystemmodule besonders hohe Anforderungen stellt.

1.5.2 Schnittstellen und virtuelle Maschinen

Betrachten wir das Schichtenmodell etwas näher. Die Relation „A benutzt B“ lässt sich dadurch kennzeichnen, dass B für A **Dienstleistungen** erbringt. Dies ist beispielsweise bei der Benutzung einer Unterprozedur B in einem Programm A der Fall. Betrachten wir dazu das Zeichnen einer Figur, etwa eines Vierecks. Die Dienstleistung `DrawRectangle(x0, y0, x1, y1)` hat als Argumente die Koordinaten der linken unteren Ecke und die der rechten oberen. Wir können diesen Aufruf beispielsweise direkt an einen Grafikprozessor GPU richten, der dann auf unserem Bildschirm das Rechteck zeichnet, siehe Abb. 1.23 links. In diesem Fall benutzen wir eine echte Maschine, um die Dienstleistung ausführen zu lassen.

DrawRectangle(x0, y0, x1, y1)	DrawRectangle(x0, y0, x1, y1)	V1
Graphic Processor Unit (GPU)	DrawLine(x0, y0, x1, y0) DrawLine(x1, y0, x1, y1) DrawLine(x1, y1, x0, y1) DrawLine(x0, y1, x0, y0)	V2
	SetPoint(x0, y0, black) SetPoint(x0+dx, y0, black) ...	V3
	Display(RAM)	V4

Abb. 1.23 Echte und virtuelle Maschinen

In vielen Rechnern ist aber kein Grafikprozessor vorhanden. Stattdessen ruft das Programm, das diese Funktion ausführen soll, selbst wieder eine Folge von einfachen Befehlen (Dienstleistungen) auf, die diese Funktion implementieren sollen, etwa viermal das Zeichnen einer Linie. Die Dienstleistung `DrawRectangle(x0, y0, x1, y1)` wird also nicht wirklich, sondern mit `DrawLine()`

durch den Aufruf einer anderen Maschine erbracht; das aufgerufene Programm ist eine **virtuelle Maschine**, in Abb. 1.23 rechts V1 genannt.

Nun kann das Zeichnen einer Linie auch wieder entweder durch einen echten Grafikprozessor erledigt werden, oder aber die Funktion arbeitet selbst als virtuelle Maschine V2 und ruft für jedes Zeichnen einer Linie eine Befehlsfolge einfacher Befehle der darunter liegenden Schicht für das Setzen aller Bildpunkte zwischen den Anfangs- und Endkoordinaten auf. Diese Schicht ist ebenfalls eine virtuelle Maschine V3 und benutzt Befehle für die CPU, die im Video-RAM die Bildpunkte (Bits) setzt. Die endgültige Ausgabe auf den Bildschirm wird durch eine echte Maschine, die Displayhardware (Displayprozessor), durchgeführt.

Diesen Gedankengang können wir allgemein formulieren. Gehen wir davon aus, dass alle Dienstleistungen in einer bestimmten Reihenfolge (Sequenz) angefordert werden, so lassen sich die Anforderungen vom Anwender an das Anwenderprogramm, vom Anwenderprogramm an das Betriebssystem, vom Betriebssystem an die Hardware usw. im Schichtenmodell auf Zeitachsen darstellen, die untereinander angeordnet sind.

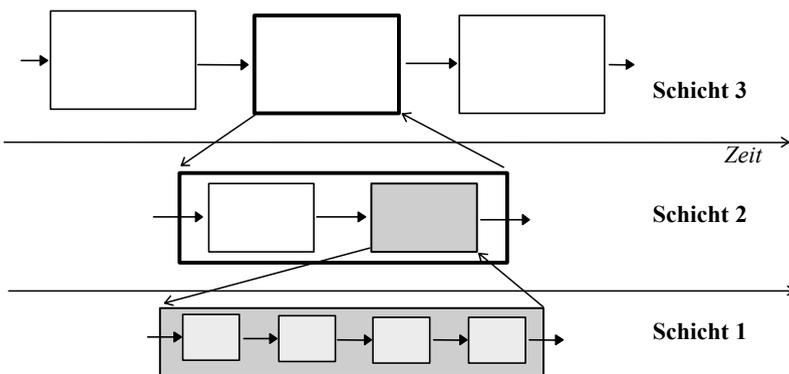


Abb. 1.24 Hierarchie virtueller Maschinen

Jede der so entstandenen Schichten bildet nicht nur eine Softwareeinheit wie in Abb. 1.19 links, sondern die Schichtenelemente sind hierarchisch als Untersequenzen oder Dienstleistungen angeordnet. Bei jeder Dienstleistung interessiert sich die anfordernde, darüber liegende Schicht nur dafür, dass sie überhaupt erbracht wird, und nicht, auf welche Weise. Die Dienstleistungsfunktionen einer Schicht, also die Prozeduren bzw. Methoden, Daten und ihre Benutzungsprotokolle, kann man zu einer **Schnittstelle** zusammenfassen. Das Programm, das diese Dienstleistungen erbringt, kann nun selbst wieder als Befehlssequenz aufgefasst werden, die darunter liegende, elementarere Dienstleistungen als eigene Leistungen benutzt. Die allerunterste Schicht, die die Arbeit nun tatsächlich auch ausführt, wird von der

„darunterliegenden“ Maschinenhardware gebildet. Da sich ihre Funktionen, so wie bei allen Schichten, über Schnittstellen ansteuern lassen, kann man die darüber liegende Einheit ebenfalls als eine Maschine auffassen; allerdings erbringt sie die Arbeit nicht selbst und ist deshalb eine virtuelle Maschine. Die Abb. 1.24 beschreibt also eine Hierarchie virtueller Maschinen. Das allgemeine Schichtenmodell aus Abb. 1.19 zeigt dies ebenfalls, aber ohne Zeitachsen und damit ohne Reihenfolge. Es ist auch als Übersicht über Aktivitäten auf parallel arbeitenden Maschinen geeignet.

Die Funktion der virtuellen Gesamtmaschine ergibt sich aus dem Zusammenwirken virtueller Einzelmaschinen. In diesem Zusammenhang ist es natürlich von außen ununterscheidbar, ob eine Prozedur oder eine Hardwareeinheit eine Funktion innerhalb einer Sequenz ausführt. Beispielsweise werden die Konstrukte höherer Programmiersprachen wie „lese“, „schreibe“, „dividiere“, usw. vom Compiler in Maschinenbefehle umgeformt. Diese Umformung entspricht einer Simulation der Funktionen (Anweisungen) einer virtuellen Maschine durch die Funktionen einer einfacheren (virtuellen) Maschine. Diese einfacheren Anweisungen, der Maschinencode, wird dann von der realen Maschine, dem Prozessor, ausgeführt.

Bisher haben wir zwischen physikalischen und virtuellen Maschinen unterschieden. Es gibt nun noch eine dritte Sorte: die **logischen** Maschinen. Für manche Leute sind sie als Abstraktion einer physikalischen Maschine mit den virtuellen Maschinen identisch, andere platzieren sie zwischen physikalische und virtuelle Maschinen.

Beispiel

Eine *virtuelle Festplatte* lässt sich als Feld von Speicherblöcken modellieren, die einheitlich mit einer sequentiellen Blocknummer angesprochen werden können. Im Gegensatz dazu modelliert die *logische Festplatte* alles etwas konkreter und berücksichtigt, dass eine Festplatte auch unterschiedlich groß sein kann sowie eine Verzögerungszeit und eine Priorität beim Datentransfer kennt. In Abb. 1.25 ist dies gezeigt. Mit diesen Angaben kann man eine Verwaltung der Speicherblöcke anlegen, in der der Speicherplatz mehrerer Festplatten unterschiedlicher Hersteller einheitlich verwaltet wird, ohne dass dies der Schnittstelle der virtuellen Festplatte bekannt sein muss. In der dritten Konkretisierungsstufe beim tatsächlichen Ansprechen der logischen Geräte müssen nun alle Feinheiten der Festplatten (Statusregister, Fehlerinformation, Schreib-/Lesebufferadressen, ...) bekannt sein. Das Verwalten dieser Informationen und Verbergen vor der nächsthöheren Schicht (der Verwaltung der logischen Geräte) besorgt dann der gerätespezifische Treiber. Die Definitionen bedeuten dann in diesem Fall:

virtuelles Gerät = logisches Gerät + Verwaltungstreiber,
logisches Gerät = physikalisches Gerät + HW-Treiber.

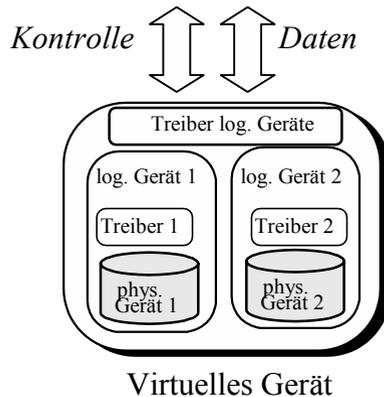


Abb. 1.25 Virtuelle, logische und physikalische Geräte

Die drei Gerätearten bilden also auch wieder drei Schichten für den Zugriff auf die Daten, vgl. Abb. 1.25.

1.5.3 Software-Hardware-Migration

Die Konstruktion von virtuellen Maschinen erlaubt es, analog zu Programmen die Schnittstelle beizubehalten, die Implementierung aber zu verändern. Damit ist es möglich, die Implementierung durch eine wechselnde Mischung aus Hardware und Software zu realisieren: Für die angeforderte Dienstleistung ist dies irrelevant. Da die Hardware meist schneller arbeitet, aber teuer ist, und die Software vergleichsweise langsam abgearbeitet wird, aber (als Kopie) billig ist und schneller geändert werden kann, versucht der Rechnerarchitekt, bei dem Entwurf eines Rechensystems eine Lösung zwischen diesen beiden Extremen anzusiedeln.

Ein Beispiel für diesen Gedankengang hatten wir schon in Abb. 1.23 kennen gelernt. Gibt es einen grafischen Prozessor, so erledigt er als reelle Maschine die Arbeit der virtuellen Maschinen V2 und V3; die Funktionen können statt durch Software auch durch Hardware erbracht werden; die Funktionen werden „migriert“.

Ein anderes Beispiel ist die Schichtung eines symbolischen Maschinencodes (hier: Java-Code, Abb. 1.26), der entweder softwaremäßig durch einen Compiler oder Interpreter in einen realen Maschinencode umgesetzt werden („*Java virtual machine*“) oder aber auch direkt als Maschineninstruktion hardwaremäßig ausgeführt werden kann. Im zweiten Fall wurde die mittlere Schicht in die Hardware (schraffiert in Abb. 1.26 links) migriert, indem für jeden Java-Code Befehl eine in Microcode programmierte Funktion in der CPU ausgeführt wird.

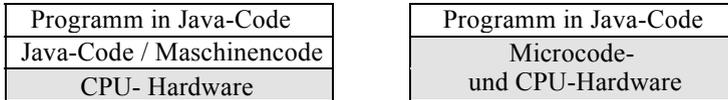


Abb. 1.26 *Software-Hardware-Migration des Java-Codes*

Wird für das Hardwaredesign statt Blaupausen eine formale Sprache verwendet (z. B. VHDL), so spielen die Unterschiede in der Änderbarkeit beider Implementierungen immer weniger eine Rolle. Entscheidend sind vielmehr andere Aspekte wie Kosten, Standards, Normen und Kundenwünsche.

1.6 Ein- und Ausgabegeräte

Mit der Einführung der Konzepte von „Schichten“ und „virtuellen Maschinen“ können wir nun auch leichter die Vielfalt und das Zusammenspiel der Peripheriegeräte wie Speicher und Drucker verstehen. In Betriebssystemen waren früher die Wechselbeziehungen zwischen Anwenderprogramm und Ein- und Ausgabegeräten sehr innig – jeder Anwenderprogrammierer setzte sein eigenes, „effizientes“ System auf, um den Datenfluss zwischen seiner Anwendung und dem Peripheriegerät zu steigern. Dieser Ansatz führte aber nicht nur dazu, dass jeder sein eigenes, geräteabhängiges Programm hatte, sondern auch zu Fehlern und Überschneidungen, wenn mehrere Programme auf dasselbe Gerät zugreifen wollten (z. B. MS-DOS). Da dies in *multi user*-Umgebungen nicht mehr tragbar war, wurden die gerätetypischen Programmteile abgetrennt und als eigene Module (**Treiber**) ins Betriebssystem integriert. Dies fördert nicht nur die Portabilität eines Programms über unterschiedliche Rechnerarchitekturen hinweg und hilft, Fehler zu vermeiden, sondern erspart auch dem Anwendungsprogrammierer Arbeit.

Die Grundaufgabe eines Treibers ist es also, alle gerätespezifischen Initialisierungsschritte und Datentransfermechanismen vor dem Anwenderprogramm hinter einer einheitlichen, betriebssystemspezifischen Schnittstelle zu verbergen. In unserer Notation aus Kapitel 1 ist der Treiber also eine *virtuelle Maschine*; er vermittelt zwischen Betriebssystem und physikalischem Gerät.

Die Aufgaben eines Treibers sind auf die Initialisierung der Datenstrukturen und des Geräts sowie Schreiben und Lesen von Daten begrenzt. Zusätzlich kommen aber noch weitere Aufgaben hinzu, die nur mit dem Betriebssystem zusammen durchgeführt werden können:

- Übersetzung vom logischen Programmiermodell zu gerätespezifischen Anforderungen
- Koordination der schreibenden und lesenden Prozesse für das Gerät
- Koordination verschiedener Geräte gleichen Typs
- Pufferung der Daten

- usw.

Diese zusätzlichen Aufgaben kann man in einer weiteren Softwareschicht zusammenfassen, so dass im allgemeinen mehrere Schichten virtueller Maschinen oder Treiber zwischen dem Anwenderprozess und dem physikalischen Gerät liegen. In Abb. 1.27 ist dies illustriert.



Abb. 1.27 Grundsichten der Geräteverwaltung

Die Einführung einer Schichtung erlaubt es, zusätzliche Aufgaben für die Datenbearbeitung in Form von Extraschichten in die Bearbeitungsreihenfolge einzufügen. Beispielsweise sieht ein Plattentreiber die Platte als ein Speichergerät an, dessen Speicheradressen durch eine Vielzahl verschiedener Parameter wie Laufwerks-, Sektor-, Plattennummer usw. bestimmt ist. Er übersetzt die Schreib- und Leseanforderungen, die für ein einfaches, lineares Modell von N Speicheradressen gelten, in die kompliziertere Adressierlogik des Plattenspeichers. Dieser Umsetzung von logischer zu physikalischer Adresse kann man nun einen weiteren Treiber vorschalten: Die Umsetzung von einer logischen, relativen Adresse innerhalb einer Datei zu der logischen, absoluten Adresse des Speichergeräts, auf dem sich die Datei befindet, wird ebenfalls von einem Dateitreiber durchgeführt.

Die Problematik, für ein neues oder existierendes Gerät einen passenden Treiber zu entwickeln, sollte nicht unterschätzt werden. Fast alle neueren Betriebssysteme leiden nicht nur unter dem Problem, zu wenige Anwendungen zu haben, sondern insbesondere darin, dass die Treiber neuer Geräte meist nur für die am meisten verkauften Betriebssysteme entwickelt werden und damit die Verbreitung des neuen Betriebssystems behindern. Es ist deshalb für jedes Betriebssystem wichtig, eine einfache Schnittstelle für Treiber (oder noch besser: ein Entwicklungssystem für den Treiber) öffentlich zur Verfügung zu stellen.

Eine interessante Initiative bildet in diesem Zusammenhang die seit 1994 im Unix-Bereich agierende *Uniform Driver Interface* UDI-Initiative aus mehreren großen Firmen. Sie versuchen, eine betriebssystemunabhängige, plattformneutrale Treiberschnittstelle zu entwickeln, um die Verwendung neuer Hardware auf mög-

lichst vielen Systemen zu beschleunigen. Das UDI-Konzept im Kontext ist in Abb. 1.28 visualisiert.

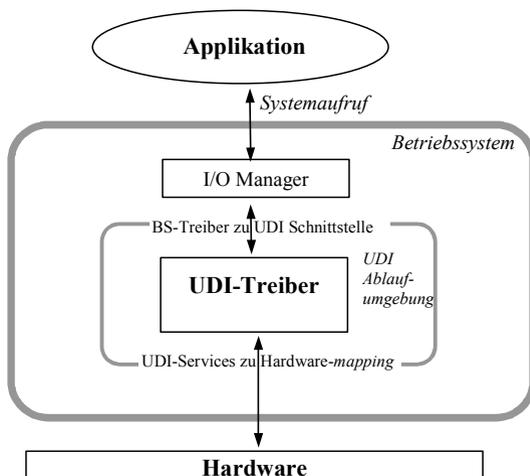


Abb. 1.28 Die Schichtung und Schnittstellen des UDI-Treibers

Die UDI-Spezifikation besteht dabei zum einen aus einer Programmierbeschreibung des UDI-Treibers und zum anderen aus einer plattformunabhängigen Ablaufumgebung, die nach oben eine Schnittstelle zum Betriebssystem hat und nach unten zur Hardwareplattform und damit dem eigentlichen Treiber eine einheitliche Umgebung anbietet. Zwar müssen sowohl Betriebssystemschnittstelle als auch Hardwareabhängigkeiten für die Ablaufumgebung programmiert werden, aber vom Implementierer des Betriebssystems nur einmal für alle Treiber.

1.6.1 Beispiel UNIX: I/O-Verarbeitungsschichten

Auch Unix ist nach verschiedenen Gesichtspunkten in Schichten eingeteilt. An dieser Stelle sollen zwei Schichtungskonzepte vorgestellt werden: Das Konzept des virtuellen Dateisystems und das der Datenströme und Filter (*streams*-Konzept).

Virtuelles Dateisystem

Unter Unix gibt es verschiedene Dateisysteme. Aus diesem Grund gab es schon früh das Bestreben, den Zugriff auf Dateien mit Standardfunktionen einer Dateisystemschnittstelle durchzuführen und die eigentliche Implementierung auf nachfolgende Schichten zu verlagern. Ein gutes Beispiel dafür ist die Schichtung in Linux, gezeigt in Abb. 1.29.

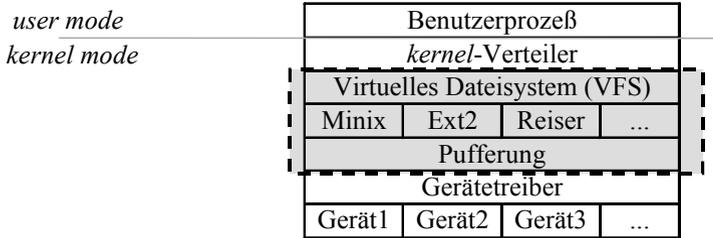


Abb. 1.29 Die Dateisystemsichtung unter Linux

Die Dateisystemsicht, grau schraffiert in der Zeichnung, besteht selbst wieder aus Unterschichten: dem virtuellen Dateisystem VFS, das die Verwaltung der Dateinformationen (i-nodes, i-node cache, Verzeichniscache) und die Anbindung an die Benutzerprozesse vornimmt, und den eigentlichen Dateisystemen wie das Minix FS, das Ext2-FS, das Reiser FS usw., die unterschiedliche Strategien der Implementierung von Dateioperationen (Lesen, Schreiben) und Verzeichnissen durch Allokation und Freigabe von Blöcken bereitstellen. Eine gemeinsame Pufferung dient allen Dateisystemen. Der allgemeine blockorientierte Gerätetreiber, der allen Dateisystemen zugrunde liegt, vermittelt deren Anforderungen an eine Vielzahl von speziellen Plattensystemen.

Das Stream-Konzept

Die grundsätzliche Schichtung des UNIX-Kerns wurde in Abb. 1.21 gezeigt. Zusätzlich zu den normalen Schichten ist es im *stream system*, einer UNIX-Erweiterung in System V (HP-UX, SUN-OS, ...), möglich, andere Bearbeitungsstufen („Treiber“) in den Bearbeitungsablauf einzufügen. In Abb. 1.30 (a) ist dies für ein einfaches, buchstabenorientiertes Terminalsystem gezeigt, in (b) für eine Festplatte.

Beim zeichenorientierten Gerät in (a) ist ein Treiber (*special char recognition*) in den Verarbeitungsweg eingeschoben, der besondere Buchstaben, die als Kommandos dienen (z. B. DEL zum Löschen des letzten Buchstabens, Control-C zum Abbruch des laufenden Prozesses usw.), sowie besondere Zeichenkonversion (z. B. 6 Leerzeichen für ein TAB-Zeichen usw.) erkennt und entsprechende Aktionen veranlasst.

Ein gesonderter Zugang (*raw interface*) erlaubt es, ohne eine „höhere“ Verarbeitung des Zeichenstroms (ohne lokales Echo, ohne Control-C usw.) direkt Zeichen zu senden und zu empfangen. Dieser Weg ist besonders für schnelle serielle Datenverbindungen zwischen Computern interessant zum Zwecke des Datenaustauschs, da hier alle Zeichen als Daten aufgefasst werden und nicht als Buchstaben mit besonderer Bedeutung interpretiert werden.

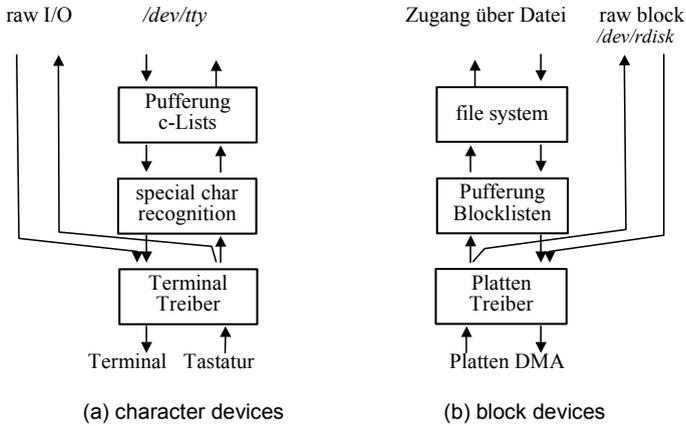


Abb. 1.30 Das Stream-System in UNIX

1.6.2 Beispiel Windows NT: I/O-Verarbeitungsschichten

Die Schichtung in Windows NT ist dynamisch und hängt vom jeweiligen Systemdienst ab. In Abb. 1.31 ist links die einfache Schichtung für serielle Geräte gezeigt, rechts die multiple Schichtung für Massenspeicher.

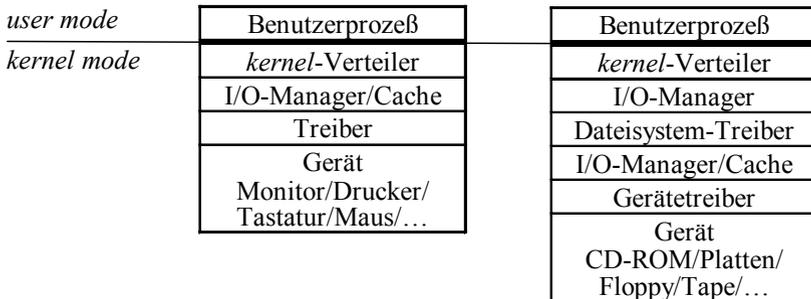


Abb. 1.31 Einfache und multiple Schichtung in Windows NT

Alle Betriebssystemaufrufe für Ein- und Ausgabe werden im zentralen I/O-Manager zu Aufträgen in Form eines *I/O Request Package* IRP gebündelt und weitergeleitet. Jedes dieser Auftragspakete enthält im Kopf die Folge der Adressaten sowie den Datenplatz, so dass ein Auftrag in der Schichtung von oben nach unten und mit den Ergebnissen aktualisiert wieder zurück an den Benutzerprozess wandert. Dabei werden – je nach Auftrag – unterschiedlich viele Stationen bzw.

Schichten durchlaufen. Zusätzlich ist auch die Existenz der Treiber dynamisch: Im Unterschied zu älteren UNIX-Versionen kann man während des Betriebs neue Treiber einklinken oder herausnehmen, um bestimmte Aufgaben zu erfüllen.

1.6.3 Der Zugriff auf Ein- und Ausgabe

Für den Zugriff von Programmen bzw. dem Betriebssystem auf Peripheriegeräte muss das Verhalten der physikalischen Geräten auf ein vom Betriebssystem erwartetes Verhalten (*Schnittstelle*) abgebildet werden. Dazu ist es nötig, dass der Systemprogrammierer etwas mehr von den Geräten weiß. Da viele Geräte ein sehr ähnliches Verhalten haben, lohnt es sich, die typischsten Modelle zu betrachten, um die charakteristischen Parameter zu verstehen. Wir können grob zwischen zwei Arten von Geräten unterscheiden: Geräte mit *wahlfreiem* Zugriff, die Adressierung benötigen, und Geräte mit *seriellem* Datentransfer ohne Adressinformation. Beide Gerätearten erhalten ihre Aufträge über eine spezielle Hardware-schnittstelle.

Die Geräteschnittstelle

Im Unterschied zu früher gibt es bei der Systembus-orientierten Rechnerarchitektur keine speziellen Hardwarekanäle und spezielle, dafür vorgesehene Prozessorbefehle mehr, sondern alle Geräte können einheitlich wie Speicher unter einer Adresse im Adressraum des Hauptspeichers angesprochen werden (*memory mapped I/O*).

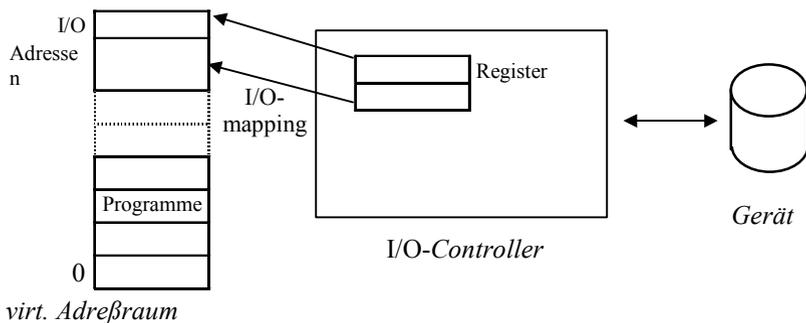


Abb. 1.32 Adressraum, Controller und Gerät

Dazu werden die Speicherzellen eines speziellen Chips (**Controller**) der Hauptplatine (*motherboard*) oder einer Einsteckplatine im Rechner und damit interne Speicherbereiche (Register, Puffer) zur Gerätekontrolle auf den Adressbereich des

Hauptspeichers abgebildet. In Abb. 1.33 sind die Reservierungen für die Peripheriegeräte im Adressraum eines Standard-PC gezeigt.

Adressraum (hexadezimal)	Gerät
000-00F	DMA Controller
020-021	Interrupt Controller
040-043	Timer
200-20F	Game controller
2FB-2FF	Serial ports (secondary)
320-32F	Hard disk controller
378-37F	Parallel port
3D0-3DF	Graphics controller
3F0-3F7	Floppy disk controller
3F8-3FF	Serial ports (primary)

Abb. 1.33 Adressreservierungen der Standardperipherie eines PC

Initialisierung der Geräteschnittstellen

Die Register der Geräte sind anfangs mit unspezifischen Daten belegt. Insbesondere die Information, welcher DMA-Kanal und welcher Interrupt von welchem Gerät benutzt werden soll, fehlen. Auch die Adressen der Befehls- und Kontrollregister können sich bei den Geräten überschneiden und müssen deshalb in einem Rechnersystem koordiniert werden. Diese Koordination wird bei jedem Neustart des Rechners durchgeführt und ist eine der frühesten Aufgaben des Betriebssystems.

Diese Aufgabe wird bei neueren Controllern vereinfacht durch vordefinierte Protokolle. Ein sehr bekanntes Protokoll ist das „**Plug-and-Play**“ (PnP) -Protokoll. Hierbei identifiziert das Betriebssystem zunächst alle beteiligten Geräte und fragt dann bei allen Controllern an, welche Ressourcen (DMA, IRQ, Registeradressen, etc.) benötigt werden. Bei PCI-Bus-Controllern ist dies durch eine Standardschnittstelle vereinfacht, bei der sowohl der Hersteller der Karte als auch der Chips, Geräte- und Versionsnummer sowie die Geräteklasse direkt als Zahlen aus der Karte gelesen werden können. In entsprechenden Geräte-Beschreibungsdateien (INF-Dateien in Windows NT) sind dann für diese Geräte alle wichtigen Daten (mögliche Registeradressen, Interrupts usw.) hinterlegt. Feste, nichtkonfigurierbare Geräte (*legacy devices*) müssen aber vorher manuell angegeben werden. Danach werden die Ressourcen nach einer Strategie verteilt und dies den Geräten mitgeteilt. Da dies ein langwieriger Prozess ist, wird meist die Tabelle verwendet, die beim letzten Start gültig und auf Platte gespeichert war (Windows 98, Windows NT) bzw. die vom Computer Mainboard Betriebssystem (BIOS) errechnet und in dessen nicht-flüchtigen ESCD-Bereich abgespeichert wurde (Linux).

Betrachten wir nun als typischen Vertreter aus der Gruppe der wahlfreien Geräte den Plattenspeicher.

1.6.4 Wahlfreier Zugriff: Plattenspeicher

Das Modell eines Plattenspeichers steht für eine ganze Modellgruppe aller Speichermedien, die eine sich drehende Scheibe verwenden. Dazu gehören neben den Festplattenspeichern auch CompactDisk (CD-R, CD-RW, DVD-R, DVD-RW), Disketten (3 1/2 Zoll, SuperDisk,...) und Wechselplattensysteme. Alle haben gewisse Modelleigenschaften gemeinsam. Betrachten wir als Beispiel den Festplattenspeicher genauer.

Der übliche magnetische Plattenspeicher besteht aus einer Aluminiumscheibe, die mit einer hauchdünnen Magnetschicht, z. B. Eisenoxid, überzogen ist. Wird nun auf einem langen Arm ein winziger Elektromagnet (**Schreib-/Lesekopf**) darauf gebracht und die Scheibe in Rotation versetzt, so kann der Magnetkopf auf einer kreisförmigen Bahn (**Spur**) die Eisenoxidpartikel verschieden magnetisieren („schreiben“). Umgekehrt rufen die kleinen, magnetischen Partikel beim Vorbeigleiten an dem Magnetkopf einen kleinen Induktionsimpuls in der Spule hervor. Bei diesem „Lesen“ der magnetisch fixierten Information wird jede Magnetisierungsänderung als Bit interpretiert, so dass auf jeder Spur eine Bitsequenz gespeichert werden kann. Da eine Spur sehr viel Information speichern kann, unterteilt man eine Spur nochmals in Untereinheiten gleicher Größe, genannt **Sektoren**.

Nun muss man nur noch den Arm kontrolliert über die Scheibe zwischen Mittelpunkt und Rand bewegen, um viele derartige Spuren schreiben und lesen zu können.

Das Grundmodell eines derartigen Plattenspeichers bleibt immer gleich, ganz egal ob man als Plattenbelag Eisenoxid, Chromdioxid oder ein optisch aktives Medium (CD-ROM, DVD) verwendet, und egal ob man die Platten flexibel aus Kunststoff mit Eisenoxidfüllung macht (*floppy disk*) oder mehrere davon mit jeweils einem Kopf von oben und von unten an der Scheibe übereinander anordnet (Festplatten), siehe Abb. 1.34. Sind mehrere Platten übereinander auf dieselbe Drehachse montiert, so werden alle Köpfe normalerweise von derselben Mechanik zusammen gleichartig bewegt. Alle Köpfe befinden sich also immer auf einer Spur mit der gleichen Nummer, aber verschiedenen Platten. Alle Spuren mit der gleichen Spurnummer liegen übereinander und bilden die Mantelfläche eines imaginären Zylinders, siehe Abb. 1.34. Die Menge der Spuren mit derselben Nummer wird deshalb auch als **Zylindergruppe** bezeichnet. Man beachte, dass diese Notation von gleichzeitig lesbaren Spuren an Bedeutung verliert, je mehr Spuren auf der Platte existieren und je ungenauer deshalb die Positionierung auf allen Spuren gleichzeitig wird. Bei den mehr als 2000 Zylindern moderner Laufwerke muss deshalb jede Spur unabhängig von den anderen Spuren eines

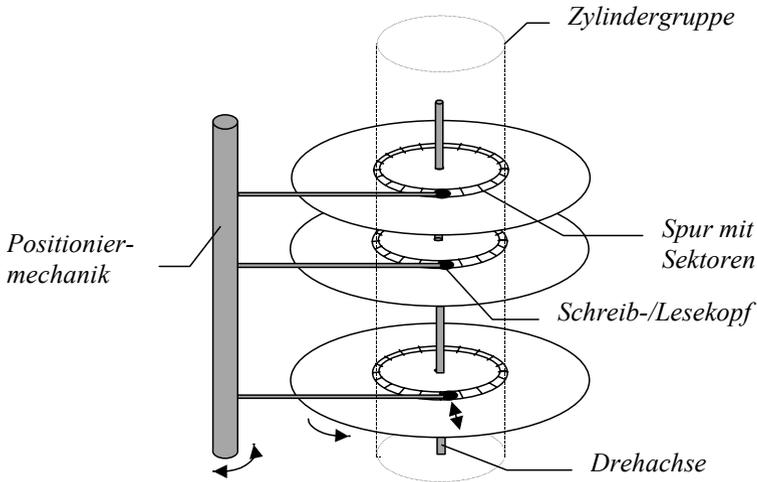


Abb. 1.34 Plattenspeicheraufbau

Zylinders betrachtet werden. Bei allen Konstruktionen muss ein Kopf mechanisch bewegt werden, um eine Spur zu lesen. Möchte man genau in der Spurmittle aufsetzen, den Spuranfang und die Abschnitte der Spur erkennen, so sind außerdem Hilfsinformationen wie regelmäßige Impulse zur Synchronisation (*Formatierung*) sowie spezielle Justierinformation (extra *Servospuren* an der Plattenunterseite) nötig.

Eine Zugriffsoptimierung für Platten sollte *nicht* im Betriebssystem stattfinden, sondern in dem eingebetteten System des Plattencontrollers. Stattdessen sollte vom Betriebssystem nur ein einfaches, klares Modell eines virtuellen Geräts ohne spezielle Zeiten berücksichtigt werden, etwa das eines großen, linearen Speicherfeldes; die eigentliche detailreiche, effiziente Modellierung sollte dagegen beim Hersteller erfolgen. Er allein kennt alle internen mechanischen Konstruktionsdetails und kann mit Hilfe des internen Controllers die Schnittstelle zum Betriebssystem auf die internen Einzelschnittstellen zu Motorelektronik und Cache effizient abbilden. Dabei sollte dieser interne Treiber (Kontrollprogramm) adaptiv seine Betriebsparameter an die jeweils aktuelle Lastverteilung (Datendurchsatz, Reihenfolge der Suchaufträge, ...) beim Betrieb anpassen.

Man sollte also die Geräteeigenschaften im gerätespezifischen Controller bzw. oder im Treiber kapseln und sie möglichst beim Betriebssystem im Normalbetrieb nicht berücksichtigen müssen.

Geräteschnittstelle

Üblicherweise gibt es bei einem wahlfreien Gerät wie einer Festplatte folgende Register:

- **Statusregister** (*read only*)
Der Speicherinhalt dieses Dateiwortes wird vom Controller aktualisiert und gelesen. Es dient als Statusregister, jedes Bit hat eine besondere Bedeutung. Beispiel: Bit 4 = 1 bedeutet, dass das Lesen beendet wurde und das Ergebnis im Datenpuffer gelandet ist. Bit 8 = 1 heißt, dass ein Lesefehler dabei aufgetreten ist.
- **Befehlsregister** (*write only*)
In dieses Register wird der Code für einen Befehl (Schreiben/ Lesen/ Formatieren/ Positionieren ...) geschrieben, der ausgeführt werden soll. Das Hineinschreiben des Befehlscodes in dieses Register lässt sich als Prozeduraufruf interpretieren, bei dem die Parameter aus dem Inhalt der anderen Register bestehen.
- **Adressregister**
Sie enthalten die Speicherzellen-Adresse auf dem Gerät (Gerät, Spur/Zylinder, Sektor, Platte, Kopf ...) und die Zahl der zu transferierenden Bytes. Die Angaben setzen dabei eine Information über die Geometrie der Festplatte voraus, also die Anzahl der verfügbaren Köpfe, Sektoren usw. Diese Information muss nicht echt sein, sondern kann auch nur das Modell widerspiegeln, das der Controller nach außen hin als Schnittstelle repräsentiert; die tatsächliche Kopfzahl, Spurzahl, Zylinder- und Sektorenzahl kann intern ganz anders sein.
- **Datenpuffer**
Alle Daten, die auf diese Adresse geschrieben werden, landen in einem internen Puffer in der Reihenfolge, in der sie geschrieben werden. Umgekehrt kann der Puffer hier sequentiell ausgelesen werden. Meist sind Ein- und Ausgabepuffer getrennt, so dass ein Puffer nur gelesen und der andere nur geschrieben werden kann.
- Ein **Interruptsystem**, das nach der Ausführung eines Befehls einen Interrupt auslösen kann.

Besitzt das Gerät sehr viele Parameter, die man einstellen kann, so benötigt man viele Register, und es geht sehr viel Adressraum für den Prozess dafür verloren. Es ist deshalb üblich, dass nur eine Adresse existiert, in die man den Zeiger (die Adresse) zum Auftragspaket oder auch sequentiell ganze Datenpakete (Aufträge) mit genau festgelegten Formaten schreiben muss.

Ein wichtiger Mechanismus für den Datenaustausch zwischen Rechner und dem Gerät ist das Verschieben von ganzen Datenblöcken zwischen dem Gerätepuffer des Controllers und dem Hauptspeicher des Rechners. Dies wird prozessorunabhängig mit speziellen Chips, (*Direct Memory Access DMA*) durchgeführt, die die Multi-Master-Busse ansteuern. Die DMA-Chips existieren dabei nicht nur im

Speichersystem, sondern auch auf den Controllerplatinen und können unabhängig voneinander arbeiten (*DMA-Kanäle*).

RAM-Disks

Für das Verhalten eines Massenspeichers im Betriebssystem ist nur der Treiber verantwortlich: Er bildet eine virtuelle Maschine. Man kann deshalb an die Stelle des realen Massenspeichers auch einen Bereich des Hauptspeichers als virtuelle Platte verwenden, ohne dass dies im Betriebssystem auffällt und anders behandelt werden müßte.

Eine solche **RAM-Disk** ist dadurch natürlich sehr schnell; für die darauf befindlichen Dateien gibt es keine mechanischen Verzögerungszeiten beim Schreiben und Lesen. Man muss nur vor dem Ausschalten des Rechners alle Dateien auf echten Massenspeicher kopieren, um sie dauerhaft zu erhalten.

Allerdings stellt sich hier prinzipiell die Frage, wozu man einen Teil des sowieso chronisch zu kleinen Hauptspeichers als Massenspeichergerät deklarieren soll. Effiziente Strategien für Cache und *paging*, verbunden mit der direkten Abbildung von Dateibereichen auf den virtuellen Adressraum wie sie heutzutage in modernen Betriebssystemen üblich sind, senken ebenfalls die Zugriffszeit und verwalten dynamisch den kleinen Hauptspeicher.

Die Antwort auf diese Frage hängt stark vom benutzten System ab.

- Beispielsweise ist eine RAM-Disk sinnvoll, wenn man mit fester, vorgegebener Software arbeiten muss, die temporäre Dateien erzeugt. Ein Beispiel dafür sind Compiler: Liegen die Zwischendateien des Kompiliervorgangs auf einer RAM-Disk, so erhöht sich die Kompiliergeschwindigkeit erheblich.

Allerdings kann man auch auf ein solches RAM-Dateisystem verzichten, wenn der *page-out-pool* groß genug ist, um alle Dateiblocke für das Schreiben zwischen zu lagern. Der Lesezugriff auf die temporäre Datei erfolgt dann über diese Blöcke, die nicht von einem Massenspeicher geholt werden müssen und deshalb genauso schnell sind wie eine RAM-Disk.

- Eine andere Situation liegt in Systemen vor, in denen das Betriebssystem keine größeren Adressbereiche ansprechen kann, beispielsweise in MS-DOS. Verfügt man über Hauptspeicher, der größer ist als die obligatorische 640-KB-Grenze, so kann man ihn über einen RAM-Disk-Treiber nutzen. Im RAM-Disk-Treiber kann man dann durch entsprechende sequentielle Adressierungskunstgriffe die Hürde überwinden.

1.6.5 Serielle Geräte

In Kapitel 4 bemerkten wir schon, dass auf fast alle Geräte sowohl sequentiell als auch wahlfrei zugegriffen werden kann, bis auf die „echten“ seriellen Geräte wie Tastatur und Drucker, bei denen dies normalerweise nicht möglich ist. Die Be-

zeichnung „serielle Geräte“ bezieht sich also auf die Art und Weise, wie Daten übergeben werden müssen: jedes Zeichen hintereinander, ohne Adressinformation für das Gerät.

Meist geschieht das mit geringer Geschwindigkeit (wenigen KB pro Sekunde). Dies sind die klassischen Randbedingungen von Terminals, langsamen Zeilendruckern und Tastaturen. Allerdings gibt es auch sehr schnelle serielle Verbindungen, die nicht zeichenweise vom Treiber bedient werden, sondern größere Datenblöcke zur Übertragung über DMA zur Verfügung gestellt bekommen. Ihre Ansteuerung wird dann mit Hilfe einer Mischung aus Techniken für Treiber von zeichen- und blockorientierten Geräten durchgeführt.

Geräteschnittstelle

Die Schnittstelle zum Controller von seriellen Geräten ist (ebenso wie die bei wahlfreien Geräten) mittels Speicheradressen ansprechbar:

- Ein **Kontrollregister** sorgt für Statusmeldungen, die Übertragungsgeschwindigkeit (z. B. 1200, 2400, 4800, 9600, 19200 Baud, gemessen in Daten- + Kontrollbits pro Sekunde), und den Übertragungsmodus „synchron“ oder „asynchron“. Bei *asynchroner* Übertragung werden in unregelmäßigen Abständen Daten (Zeichen) übermittelt; bei *synchroner* Übertragung folgen alle Zeichen in festem zeitlichen Abstand und Format.
- Ein **Eingaberegister** enthält das zuletzt empfangene Zeichen, das vom Treiber dort ausgelesen wird.
- In das **Ausgaberegister** wird das zu sendende Zeichen vom Treiber geschrieben; unmittelbar darauf wird es vom Controller gesendet.
- Ein **Interruptsystem** löst nach jedem gesendeten bzw. empfangenen Zeichen einen Interrupt aus.

Die meisten seriellen Geräte arbeiten außerdem mit einer **Flusssteuerung**, die im Kontrollregister gesetzt wird. Üblich ist entweder eine Hardwaresteuerung über extra Drähte (RS232-Norm), oder aber es werden spezielle Zeichen (Control-S, Control-Q, genannt XON und XOFF) zur Steuerung gesendet. Droht beim Empfänger der Empfangspuffer überzulaufen, so sendet er XOFF und veranlasst damit den Sender, mit dem Senden aufzuhören. Ist der Puffer leer, sendet der Empfänger XON und signalisiert so dem Sender, weiterzusenden. Allerdings funktioniert diese Mechanik nur dann, wenn der Empfänger sein Signal rechtzeitig sendet, so dass der Sender auch genügend Zeit hat, das Kontrollzeichen zu empfangen, zu interpretieren und im Treiber umzusetzen, bevor der Empfängerpuffer überläuft.

1.6.6 Multiple Plattenspeicher: RAIDs

Eine weitverbreitete Methode, die Plattenkapazität zu erhöhen, besteht darin, mehrere kleine Platten gemeinsam als eine große virtuelle Platte zu verwalten. Haben

wir für jede Platte ein extra Laufwerk, so muss man pro Laufwerk nur noch eine geringere Kapazität vorsehen und kann dafür kleine, preiswerte Laufwerke einsetzen. Ein solches System wird als RAID (**R**edundant **A**rray of **I**nexpensive **D**isks) bezeichnet und ist außerordentlich beliebt bei großen kommerziellen Datenbanken.

Für das Betriebssystem ist diese Hardwarevariante transparent: Der Treiber zeigt nur ein einziges, virtuelles, schnelles Laufwerk großer Kapazität; die Realisierung als Plattensammlung ist für die Schnittstelle zu den höheren Betriebssystemschichten unwichtig. Diese Basiseigenschaft wird als **RAID-linear** bezeichnet.

Ein weiterer Vorteil multipler Plattenspeicher kann genutzt werden, wenn mehrere unabhängige Prozesse oder *threads* im Betriebssystem existieren. Ihre Unabhängigkeit und damit die mögliche parallele, schnellere Abarbeitung wird behindert, wenn die an sich unabhängigen Daten alle auf einem gemeinsamen Massenspeicher versammelt sind, dessen Abschnitte nicht unabhängig voneinander gelesen werden können. Setzen wir nun mehrere Laufwerke anstelle eines einzigen mit fest gekoppelten Köpfen ein, so können die Daten der unabhängigen Prozesse auch auf unabhängig parallel agierenden Plattenspeichern abgelegt werden; der Datenzugriff wird für die Prozesse erheblich schneller. In Abb. 1.35 ist eine solche Aufteilung von Daten auf multiple Laufwerke (dargestellt als Säulen) zu sehen.

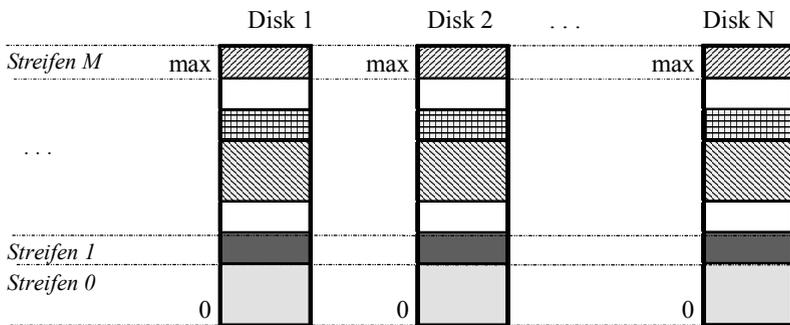


Abb. 1.35 Neugruppierung der Plattenbereiche in Streifen

Dazu wird der gesamte logische Speicherplatz in Abschnitte aufgeteilt. Korrespondierende, gleich große Abschnitte verschiedener Laufwerke werden vom Treiber als gemeinsamer Speicherbereich (**Streifen**) angesehen und bilden einen einheitlichen Speicherbereich für eine Prozessgruppe. Der Treiber im Betriebssystem hat dabei die Aufgabe, die Verteilung und Zusammenfassung der Daten vorzunehmen und dazu die Abbildung virtueller Speichereinheiten auf die logischen Einheiten unterschiedlicher Laufwerke durchzuführen.

Beispiel Windows NT *Plattenorganisation*

In diesem Betriebssystem kann bei der initialen Formatierung von logischen Plattenbereichen (*Partitionen*) die Option angegeben werden, die Plattenbereiche als *Streifen* zu organisieren.

Eine andere, ähnliche Option besteht darin, mehrere Partitionen beliebiger Größe von unterschiedlichen Platten zu einer neuen logischen Platte (*volume*) zusammenzufassen. Der virtuelle Plattenadressraum kann so unregelmäßig auf unterschiedliche Platten verteilt werden.

Machen wir den Streifen sehr schmal, z. B. 4 Byte oder auch einen Block groß, so wird der gesamte Datenpuffer, aufgeteilt in sequentielle Abschnitte der Länge 4 Byte bzw. ein Block, sehr schnell auf das Plattenfeld verteilt geschrieben oder gelesen. Eine solche schnelle, aber nicht ausfallsichere Konfiguration wird als **RAID-0**-System bezeichnet und gern für die großen Datenmengen in der Video- und Bildproduktion eingesetzt.

Systeme aus multiplen Laufwerken haben neben der höheren **Zugriffsschnelligkeit** und größeren **Speicherkapazität** noch einen weiteren wichtigen Vorteil: Mit ihnen kann man Ausfälle von Platten tolerieren. Für diese Art von Fehlertoleranz, der **Ausfalltoleranz**, sieht man für die Daten einer Platte eine exakte Kopie auf einer Platte eines anderen Laufwerks vor; die andere Platte ist bezüglich ihrer Daten wie ein Spiegel (*mirror*) der einen Platte aufgebaut. In Abb. 1.36 ist das Schema einer solchen **Spiegelplattenkonfiguration** gezeigt.

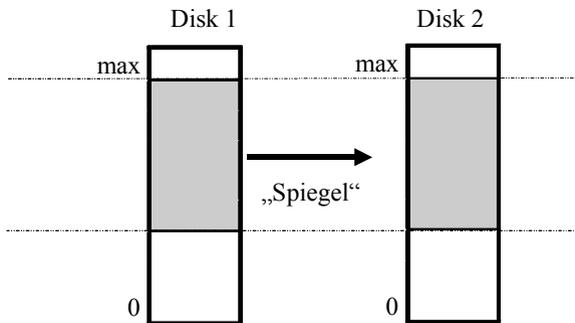


Abb. 1.36 Spiegelplattenkonfiguration zweier Platten

Allerdings muss diese Kopie ständig aktualisiert werden. Der Treiber, der eine solche Konfiguration bedienen kann, muss dazu bei jedem Schreibvorgang auf Disk 1 eine Kopie auf Disk 2 bewirken, was eine zusätzliche Belastung des Betriebssystems und der I/O-Hardware bewirkt. Eine solche ausfallsichere, aber ineffiziente Konfiguration von Spiegelplattenpaaren wird als **RAID-1**-System be-

zeichnet. Führen wir den Gedanken der Spiegelplatten in die Streifenkonfiguration von Abb. 1.35 ein, so erhalten wir ein schnelles, ausfallsicheres **RAID-0/1** System, das in Finanzanwendungen (Lohnbuchhaltung) eingesetzt wird.

Man kann versuchen, den Zeitverlust zum redundanten Schreiben bei den Lesevorgängen der Spiegelplatten wieder zu kompensieren. Verfügt der Treiber über den augenblicklichen Zustand der Platten, so kann er zum Lesen diejenige der beiden Platten auswählen, deren Lesekopf der gewünschten Spur am nächsten ist; der andere Plattenkopf kann bereits für den nächsten Auftrag positioniert werden oder in anderen, nicht gespiegelten Partitionen schreiben oder lesen.

Den zusätzlichen Zeitbedarf für ein schnelles, ausfallsicheres RAID-0/1-System kann man durch einen Trick drastisch senken. Dazu wird jedes Bit eines Datenwortes auf eine extra Platte geschrieben – für ein 32-Bit-Wort benötigt man also 32 Laufwerke. Dann wird jedes Wort mit einer speziellen Signatur versehen, beispielsweise einem 6-Bit-Zusatz, so dass im Endeffekt jedes Datenwort von einem Controller auf 38 Laufwerke verteilt wird. Dies bedeutet nicht nur einen enormen Datenfluss, sondern ermöglicht auch eine große Ausfalltoleranz. Da nicht jede Bitkombination der 38-Bit-Datenworte möglich ist, kann man auch bei mehreren ausgefallenen oder defekten Platten auf den korrekten 32-Bit-Code schließen, s. MacWilliams u. Sloane (1986).

Beispiel Fehlerkorrektur durch Paritätsbildung

Angenommen, wir wollen die Zahlen 10, 7, 3 und 12 abspeichern und garantieren, dass bei Verlust einer der Zahlen die fehlende wieder rekonstruiert werden kann. Wie können wir dies erreichen?

Dazu bilden wir zusätzlich zu den n Bits, die für eine Zahl unabhängig voneinander abgespeichert werden, als Kontrollinformation ein $(n+1)$ -tes Bit, das Paritätsbit p , nach der folgenden XOR-Funktion \oplus zweier Binärvariablen a und b :

a	b	XOR $p = a \oplus b$	$p \oplus b$	Es folgt direkt aus der Tabelle
1	1	0	1	$a \oplus 0 = a$
1	0	1	1	$a \oplus a = 0$
0	1	1	0	
0	0	0	0	und somit $p \oplus b = a \oplus b \oplus b = a \oplus 0 = a$

Mit $p \oplus b = a$ wissen wir nun, dass bei bekannter Parität p der Ausfall von einem Bit a toleriert werden kann, da wir a direkt aus p und b wiederherstellen können. Dies gilt natürlich auch, wenn b selbst wiederum aus vielen Termen besteht, beispielsweise

$$b = b_1 \oplus b_2 \oplus \dots \oplus b_{n-1}$$

Bei n Binärvariablen, aus denen wir die Parität bilden, kann also eine davon mit Hilfe der anderen und der Parität wieder erschlossen werden. Für unsere vier Zahlen von oben bedeutet dies bei Ausfall von Zahl 3:

Paritätsbildung	Rekonstruktion von Zahl 3
Zahl 1: 10 = 1 0 1 0	Zahl 1: 10 = 1 0 1 0
Zahl 2: 7 = 0 1 1 1	Zahl 2: 7 = 0 1 1 1
Zahl 3: 3 = 0 0 1 1	Parität: 2 = 0 0 1 0
<u>Zahl 4: 12 = 1 1 0 0</u>	<u>Zahl 4: 12 = 1 1 0 0</u>
Parität: 0 0 1 0 = 2	⊕ Ergebnis: 0 0 1 1 = 3

Behandeln wir nun 8 Binärvariablen (zusammengefasst in einem Byte) parallel, so gilt dies für jedes Bit in diesem Byte und damit für das ganze Byte. Speichern wir also jedes Bit – auch das Paritätsbit – auf einem anderen Massenspeicher, so können wir beim Ausfall eines Speichers aus den n verbliebenen Bits das fehlende direkt rekonstruieren und damit den Ausfall des einen Massenspeichers tolerieren. Bilden wir mehrere Fehlerkorrekturbits (ECC, MacWilliams u. Sloane 1986), so können wir auch den Ausfall mehrerer Laufwerke tolerieren.

Für ein fehlerkorrigierendes **RAID-2**-System werden die Bits der zu speichernden Datenworte extra behandelt und gespeichert. Die ist in Abb. 1.37 für ein Fehlertoleranzbit (Paritätsbit) pro Datenabschnitt (*Wort*) dargestellt.

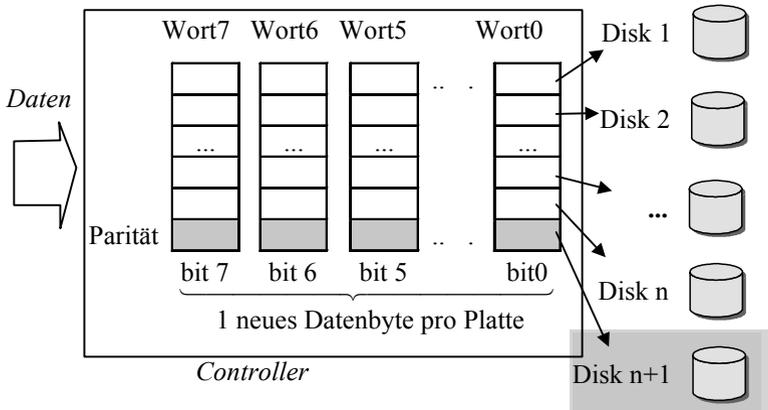


Abb. 1.37 Datenzuweisung beim RAID-2-System

Die Datenworte sind als Säulen gezeigt, die in ihre Bits aufgeteilt werden. Das i -te Bit aller Worte wird so auf der i -ten Platte gespeichert. Der Bitstrom für eine Platte lässt sich im Controller wieder in Bytes zusammenfassen, so dass als Speichersysteme handelsübliche Platten genommen werden können. Beim Raid-2-System wird von jedem Wort ein allgemeiner fehlerkorrigierender Code (*Error Correcting Bits* ECC) gebildet, dessen einzelne Bits jeweils auf einer eigenen Platte gespeichert werden. Würden wir die Spindeln der einzelnen Platten zwangs-synchronisieren, so wäre ein Gesamtsystem aus n Platten mit jeweils einem Kopf zwar nicht schneller als ein einzelnes Laufwerk mit n Platten und n Köpfen, aber dafür fehlertolerant. Natürlich können wir auch die Parität statt über die Säulen über die Zeilen, also über alle m Worte eines Puffers, bilden. Das sich ergebende Wort aus Paritäten (ECC) wird dann nach jeweils m Worten auf einer extra Platte gespeichert. In diesem Fall ist die Zahl der Platten nicht gleich der Zahl der Bits pro Wort, sondern m und damit beliebig; jedes Wort wird nicht in Bits aufgespalten, sondern mit anderen als Teilpuffer (etwa in einem Streifen) auf eine Platte gespeichert. Wir erhalten so ein **RAID-3**-System. Da hier genauso wie bei der reinen Parität von RAID-2 die gesamte Fehlertoleranzinformation aller Blöcke auf einem einzigen Laufwerk gespeichert ist, darf hier nur ein einziges Laufwerk ausfallen. Ein weiterer Nachteil ist die kurze Länge der Fehlertoleranzinformation: Hier sollten wie bei RAID-2 die Spindeln der Laufwerke synchronisiert sein, um die Zugriffszeit t_D möglichst klein zu halten.

Fassen wir weiter die Datenbits zu größeren Abschnitten zusammen (z.B. zu Blöcken) und speichern jeden Block wie bei RAID 0 jeweils auf einer separaten Platte, so wird dies als **RAID-4**-System bezeichnet. Da hier zusätzlich bei jedem Schreiben auch die Fehlertoleranzinformation aktualisiert werden muss, dauert das Schreiben länger: RAID-4 bietet keinen Geschwindigkeitsvorteil wie RAID-0. Die Lesegeschwindigkeit entspricht etwa der einer Einzelplatte im Block-Lesemodus. Ein Alternative dazu besteht darin, die häufige gelesene Fehlertoleranzinformation FI für jeden Block extra auf einem anderen Laufwerk zu sichern und so die I/O-Belastung der Aktualisierung auf alle Platten gleichmäßig zu verteilen. Dies ist ein **RAID-5**-System. Das Schema dafür ist in Abb. 1.38 gezeigt. Die Raid-5 Systeme werden gern für Datenbankserver und Internetserver eingesetzt.

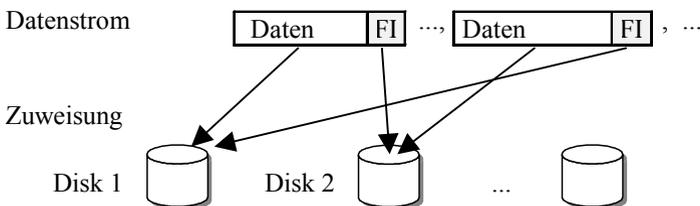


Abb. 1.38 Wechselseitige Abspeicherung der Fehlertoleranzinformation

Für den Betriebssystemarchitekten besteht die Hauptaufgabe darin, die Komplexität und das Fehlertoleranzverhalten modular in das RAID-Subsystem bzw. den Treiber zu verlagern und das Betriebssystem vom Speichersystem unabhängig zu machen: Für das Betriebssystem sollte nur ein einfaches, virtuelles Speichersystem sichtbar sein.

Bei der Wahl zwischen einer Softwarelösung und einer Hardwarelösung für RAID sollte man auf eine reine SW-Lösung verzichten: Sie ist sehr aufwändig und stellt viele Probleme, insbesondere das Wiederanlaufen nach einem Hardwarefehler.

Beispiel Windows NT *Fehlertoleranz durch Spiegelplatten und RAID*

Für eine fehlertolerante Dateiverwaltung gibt es in Windows NT einen speziellen Treiber: den FtDisk-Treiber. Dieser wird zwischen den NTFS-Treiber und den eigentlichen Gerätetreiber geschoben; er bildet als Zwischenschicht eine virtuelle Maschine, die alle auftretenden Fehler selbst abfangen soll, unsichtbar und unbemerkt von allen höheren Schichten. Tritt nun ein Fehler (*bad sector*) auf, so werden die Daten des fehlerhaften Blocks von der Spiegelplatte gelesen oder mit Hilfe der Fehlerkorrekturinformation (Paritäts-*stripes*) rekonstruiert. Dann wird für den fehlerhaften Sektor ein neuer Sektor vom Gerät angefordert (*bad sector mapping*) und die Daten darauf geschrieben. Damit existiert wieder die nötige Datenredundanz im System. Gibt es keine freien Sektoren (*spare sectors*) mehr oder kann das Gerät kein *sector mapping*, so wird der wiederhergestellte Datenblock zusammen mit einer Warnung an den Dateisystemtreiber weitergegeben.

Wurde keine fehlertolerante Plattenorganisation eingerichtet, so kann FtDisk den Fehler als „Lese-/Schreibfehler“ nur weitermelden. Es ist dann Sache des Dateisystems (oder des Benutzers), darauf sinnvoll zu reagieren.

Man beachte, dass die RAID-Konfigurationen zwar für Ausfalltoleranz sorgen, aber kein regelmäßiges Backup aller Daten der Platten ersetzen. Im Internet gibt es herzerreißende Geschichten von Systemadministratoren, die durch Softwarefehler ganze RAID-Datenkonfigurationen korrumpiert erhielten und so von der RAID-Fehlertoleranz bitter enttäuscht wurden.

1.6.7 Interleaving

Eine weitere Optimierungsmöglichkeit bietet sich an, wenn die Blöcke schneller von der Platte zum Controller gegeben werden, als sie weitertransportiert werden können. In diesem Fall eines langsamen Controllers lässt sich der Block mit der Nummer 4 nicht sofort nach dem Block 3 lesen – die Platte hat sich während der Übertragung von Block 3 schon weitergedreht, und es wird nun z. B. Block 5 statt Block 4 gelesen. Der Controller muss erst die nächste Umdrehung abwarten, um endlich Block 4 lesen zu können. Ändert man nun die Nummerierung der Blöcke

ab, so dass Block 5 die Nummer 4 bekommt, kann man statt dessen kontinuierlich ohne Wartezeit lesen. Wird also in diesem Fall nur jeder zweite physische Block für die logische Nummerierung gewählt, bleibt mehr Zeit für das Übermitteln der Blöcke, und der Datendurchsatz steigt an. In Abb. 1.39 (a) ist die physische Nummerierung außen an der Platte notiert; auf den Plattensegmenten innen ist die entsprechende logische Nummerierung vermerkt.

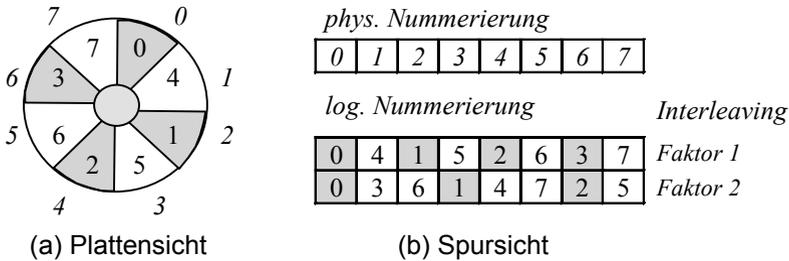


Abb. 1.39 Block interleaving

Diese Technik wird als **Interleaving** bezeichnet; die Zahl der bei der Nummerierung zunächst übersprungenen Blöcke ist der *Interleaving-Faktor*. In Abb. 1.39 (b) sind die Nummerierungen für die Interleaving-Faktoren 1 und 2 gegenübergestellt. Dieser Faktor muss vom Treiber für die Formatierung der Festplatte dem Controller mitgeteilt werden und hängt stark von der Übertragungsgeschwindigkeit des beteiligten I/O-Systems des Betriebssystems ab.

Man beachte, dass optimales Interleaving nur dann möglich ist, wenn auch die Zeit zum Spurwechsel mit einbezogen wird. Es ist deshalb sinnvoll, die Nummerierung der Sektoren bzw. Blöcke von Spur zu Spur um einen Betrag versetzt zu beginnen (*cylinder skew*), so dass es trotz Spurwechselzeit möglich ist, den logisch nächsten Block auf der nächsten Spur zu lesen. Allerdings gelingt dies nur in einer Richtung, so dass eine solche versetzte Nummerierung nur sinnvoll ist, wenn die Suchrichtung für Blöcke immer gleich ist.

Die bisher besprochenen Mechanismen der Leistungssteigerung im Treiber müssen allerdings alle im Kontext des Controllers gesehen werden. Prinzipiell kann leicht der Fall auftreten, dass alle Optimierungsmaßnahmen des Treibers vom Controller, der meist mit eigenem Mikroprozessor ausgestattet ist, durch eigene Optimierungsstrategien konterkariert werden. Beispielsweise benötigen moderne, schnelle mikroprozessorgesteuerte Controller kein Interleave mehr. Auch haben manche Controller die Möglichkeit, defekte Blöcke durch Ersatzblöcke auf speziellen, normal nicht zugänglichen Spuren zu ersetzen. Diese spezielle Abbildung tritt immer dann in Kraft, wenn ein defekter Block angesprochen wird, und ist transparent nach außen, kann also nicht vom Treiber bemerkt werden. Damit erweisen sich alle Bemühungen des Treibers, die Kopfbewegung zu minimieren, als

obsolet: Die dazwischenliegenden Bewegungen zur Ersatzspur lassen keine Optimierung von außen mehr zu. Es ist deshalb sehr sinnvoll, die Schnittstellen zwischen Treiber und Controller vom Hersteller aus systematisch zu durchdenken und vor auszuplanen. Beispielsweise kann man bestimmte geräteabhängige Optimierungen wie Kopfbewegung und Sektorsuche dem Controller zuweisen und dort kapseln; nach außen hin sind nur die Leistungen (Speicherung von Blöcken mit logischen Adressen etc.) sichtbar, die unabhängig von der implementierten Elektronik und Mechanik erbracht werden.

1.6.8 Pufferung

Eine wichtige Leistungsoptimierung bei Massenspeichern, die für starken Durchsatz benötigt werden (z. B. bei Datenbankanwendungen), kann man mit einem Puffer (Datencache) erreichen, der auf verschiedenen Ebenen eingerichtet werden kann.

Auf der Ebene des Dateisystemtreibers ist es üblich, die wichtigsten, am meisten benutzten Blöcke einer Datei zu puffern, wobei man ihn in zwei verschiedene Arten (Schreib- und Lese-puffer) unterteilen kann, jeden mit eigener Verwaltung.

Auf unterster Treiberebene kann man größere Schreib- und Lese-einheiten der Daten puffern, beispielsweise eine ganze Spur. Nachfolgende Schreib- und Leseoperationen beziehen sich meist auf aufeinanderfolgende Sektornummern und können mit der gepufferten Spur wesentlich schneller durchgeführt werden.

Allerdings hat die Pufferung auch hier die gleichen Probleme wie in Abschnitt 3.5 beschrieben. Bei der Verwaltung muss sichergestellt werden, dass beschriebene Blöcke und Sektoren auch für die Anfragen anderer Prozesse verwendet werden, um die Datenkonsistenz zu garantieren.

Wichtig ist dabei die Synchronisierung der Pufferinhalte mit dem Massenspeicher, bevor das Rechnersystem abgeschaltet wird. Bei Versorgungsspannungsausfall (*power failure*) muss dies sofort von den gepufferten Treibern eingeleitet werden.

Auch hier muss man sorgfältig die Schnittstelle zum Controller beachten. Manche Controller haben bereits einen Cache intern integriert; hier ist es sinnlos, auch auf Treiberebene einen Cache anzulegen.

Beispiel UNIX *Pufferung*

Die Pufferung der seriellen *character files* wird über Listen aus Zeilen (c-list) durchgeführt. Aus diesem Grund wird ein RETURN (Zeilenende/Neue Zeile)-Zeichen benötigt, um einen Text von der Tastatur einzulesen. Besondere Statusbits dieses *special file* erlauben es allerdings auch, für eine reine Zeicheneingabe (Cursor-Tasten etc.) sofort jedes Zeichen ohne Pufferung zu lesen oder das lokale Echo eines Zeichens auf dem Monitor zu unterdrücken.

Das Puffersystem für die *block devices* benutzt als Speichereinheiten die Blöcke. Für jeden Treiber gibt es eine Auftragsliste von Blöcken, die gelesen bzw. geschrieben werden sollen. Die Liste aller freien Blöcke ist doppelt verzeigert

und in einem Speicherpool zentral zusammengefasst. Außerdem gibt es zwei Zugangswege zu den *block devices*: zum einen über einen Dateinamen und damit über das Dateisystem und zum anderen über den *special file* als *raw device*. Da durch die Pufferung der *i-nodes* das gesamte Dateisystem bei einem Netzausfall hochgradig gefährdet ist, werden in regelmäßigem Takt (alle 30 Sek.) von der `sync()`-Prozedur alle Puffer auf Platte geschrieben und die Datenkonsistenz damit hergestellt. Dies wird auch beim Herunterfahren des Systems (*shut down*) durchgeführt.

Beispiel Windows NT Pufferung

Zur Verwaltung des Ein- und Ausgabecache gibt es einen speziellen *Cache Manager*. Dieser alloziert dynamisch Seiten im Hauptspeicher und stellt ein *memory mapping* zwischen den Hauptspeicherseiten und einer Datei her. Die Anzahl der so erzeugten *section objects* ist dynamisch: Sie hängt sowohl vom verfügbaren Hauptspeicher als auch von der Zugriffshäufigkeit auf die Dateiteile ab. Dies geschieht dadurch, dass diese Seiten des Cache-Managers wie die Speicherseiten eines normalen Prozesses verwaltet werden.

Für die Pufferung der seriellen Ein- und Ausgabe mussten spezielle Mechanismen entwickelt werden, um die Ergebnisse der Programme aus dem Multi-tasking non-preemptiven Windows 3.1 (16 Bit) auf dem Multi-tasking preemptive Windows NT zu erhalten. Für jedes Gerät gibt es in Windows 3.1 eine einzige I/O-Warteschlange, auch für serielle Geräte. Werden in verschiedene Fenster Eingaben (Zeichen, Mausklicks etc.) gegeben, so werden sie als jeweils eine Eingabeeinheit in den Eingabepuffer gestellt. Üblicherweise liest und schreibt ein Prozess unter Windows 3.1 beliebig lange (*non-preemptive*), bis er die für ihn bestimmte Eingabe abgearbeitet hat, und wird dann beim Lesen auf den Eingabepuffer blockiert, wenn die weitere Eingabe in einem anderen Fenster erfolgte und deshalb nicht für ihn bestimmt war. Der zum Fenster gehörende Prozess wird dann vom Windows-Manager aktiviert und liest seinen Pufferanteil ein.

Gehen wir nun zu einer preemptiven Umgebung über, bei der ein Prozess sofort deaktiviert werden kann, wenn seine Zeitscheibe abgelaufen ist, so führt dies bei nur einem Eingabepuffer zu Problemen – der neu aktivierte Prozess liest fehlerhaft die für den deaktivierten Prozess bestimmten Daten. Dies ist auch der Fall, wenn ein Prozess „abstürzt“, also fehlerhaft vorzeitig terminiert. Aus diesen Gründen hat in Windows NT jeder Prozess (hier: *thread* genannt) seine eigene Eingabewarteschlange – ungelesene Eingabe für einen *thread* verbleibt beim Prozess und wird nicht vom nächsten *thread* versehentlich gelesen; das System wird robust gegenüber fehlerhaften Prozessen.

Wie kann man zwei derart unterschiedliche Systeme wie Windows 3.1 und Windows NT miteinander integrieren? Die Logik der non-preemptiven Prozesse aus Windows 3.1 ähnelt sehr der Logik von non-preemptiven Prozessen. Die Idee ist nun, den Mechanismus der *threads* in Windows NT dafür zu nutzen. Dazu behandelten die Designer von Windows NT das Windows 16-Bit-

Subsystem (WOW), das die 16-Bit-Tasks als eigene *threads* gestartet hatte, wie einen einzigen Prozess. Diesem Prozess wird zwar regelmäßig der Prozessor entzogen (wie allen anderen NT Prozessen auch), bei der Prozessorrückgabe erhält aber der letzte laufende *thread* automatisch wieder die Kontrolle – als ob kein Prozesswechsel stattgefunden hätte. Somit wird genau das Verhalten erreicht, das im alten Windows/DOS System üblich war.

Allerdings ist es auch möglich, die 16-Bit-Applikationen in getrennten Adressräumen als eigene Prozesse ablaufen zu lassen. In diesem Fall verhalten sie sich natürlich nicht mehr zwangsläufig wie früher unter Windows 3.1, so dass dies nicht bei allen alten Applikationen sinnvoll ist.

1.6.9 Synchrone und asynchrone Ein- und Ausgabe

Bei der Ein- und Ausgabe ist es im Programm üblich, abzuwarten, bis ein Betriebssystemaufruf erfolgreich abgeschlossen wurde (**synchrone** Ein- und Ausgabe). Nun dauert es aber meistens eine gewisse Zeit, bis die Ein- oder Ausgabe durchgeführt wurde. Diese Zeit könnte das Programm besser mit anderen, ebenfalls wichtigen Arbeiten nutzen.

Eine Möglichkeit dafür bietet **asynchrone** Ein- und Ausgabe. Der Systemaufruf leitet dabei die Ein- und Ausgabeoperation nur ein; das Ergebnis muss von einem Prozess später mit einem speziellen Befehl abgeholt werden. Diese Art von Systemaufrufen stellt besondere Anforderungen an das Betriebssystem, da sowohl der Auftrag als auch das Ergebnis unabhängig vom beauftragenden Prozess zwischengespeichert und verwaltet werden muss.

Sowohl Unix als auch Windows NT bieten vielfältige Aufrufe für synchronen und asynchronen Datentransfer.

1.7 Die Energieverwaltung

In den letzten Jahren ist durch das stete Anwachsen der Rechnerzahl und der Verbreitung des Internet auch der Bedarf an Energie durch die Rechner gewachsen. Inzwischen verbrauchen die Rechner einen nicht unerheblichen Anteil am gesellschaftlichen Gesamtbedarf an Strom, so dass Maßnahmen in das Blickfeld des Interesses gerückt sind, um den Energiebedarf zu dämpfen. Dabei kann man einerseits so vorgehen, dass jedes Gerät wie Monitor, Drucker, Massenspeicher oder Grafikkarte für sich selbst so wenig wie möglich an Strom verbraucht, wenn es eine gewisse Zeit ungenutzt eingeschaltet ist, und in einen Energiesparmodus („Abwarten: *stand-by*“) übergeht.

Diese Lösung ist aber nicht optimal: Weiß man, dass der Rechner bald wieder mit voller Leistung arbeiten muss, so sollten die Einzelkomponenten nicht isoliert von einander in den Stromsparmodus übergehen, sondern eher abwarten. Ist aber klar, dass der Rechner absolut nicht gebraucht wird und nur irgendwann durch einen

externen Befehl „aufgeweckt“ werden soll, so kann auch eine fast vollständige Abschaltung durchgeführt werden („Winterschlaf: *hibernation*“).

Entscheidend ist also eine zentrale Kontrolle der Energiesparmaßnahmen durch eine gemeinsame, aufeinander abgestimmte Strategie. Dies ist formal in der Definition für ein „*Advanced Configuration and Power Management Interface ACPI*“ festgelegt (s. ACPI home), die von einem Firmenkonsortium (Compaq/HP, Intel, Microsoft, Phoenix, Toshiba) als Nachfolge des *Advanced Power Management APM* eingeführt wurde und muss sowohl von den Bauteilen bzw. Geräten als auch vom Betriebssystem (Treiber!) unterstützt werden. Erst das Zusammenspiel aller Komponenten ermöglicht ein unproblematisches Abschalten und wieder Anlaufen des Gesamtsystems. In Unix (Linux) wird dies z.Z. integriert (s. LINUX-ACPI); in Windows ist es bereits enthalten.

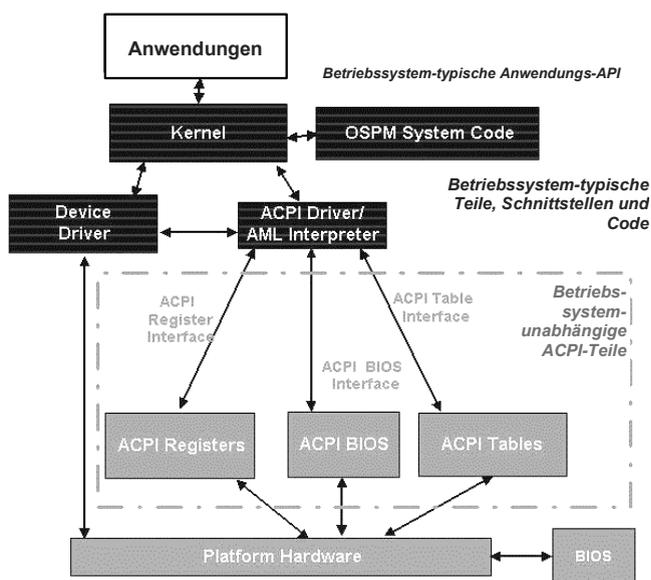


Abb. 1.40 Überblick über die ACPI-Energieverwaltung

In der Abb. 1.40 ist ein Gesamtüberblick über das System gezeigt. Wie man sieht, gliedert sich das Gesamtsystem in die von der ACPI-Spezifikation beschriebenen Teile wie Tabellen und Register, und in die betriebssystemtypischen Teile wie allgemeine Gerätetreiber, die mitspielen müssen, und spezielle ACPI-Treiber, die spezielle Geräte wie den Energiespar-Zeituhr usw. verwalten müssen.

Insgesamt verlangt die ACPI-Spezifikation verschiedene Funktionen wie

- eine Power/sleep-Taste, je eine Taste oder kombiniert in einer Taste (kurz drücken: sleep, dauernd drücken: aus)
- einen Zeitzähler (*timer*) mit einer Taktfrequenz von 3,579 MHz, der die Energiezufuhr kontrolliert. Er muss drei Register (2 Befehls-, 1 Status-) haben.
- mindestens vier Ereignis-Register, die frei belegt werden können, etwa durch „Aufwachen bei Netzaktivität“, „Aufwachen bei Tastendruck der Tastatur“, usw.
- Ein Interrupt-System, das bei Auftreten des Ereignisses das ACPI-System aktiviert
- Eine Stand-By-Stromversorgung (5V/500mA), die im Schlafmodus die nicht schlafenden Geräteabschnitte versorgt.

Das Gesamtsystem kann man als einen Automaten ansehen, der nur wenige diskrete Zustände annehmen kann. In Abb. 1.41 sind die Zustände und ihre möglichen Übergänge als Graf gezeigt.

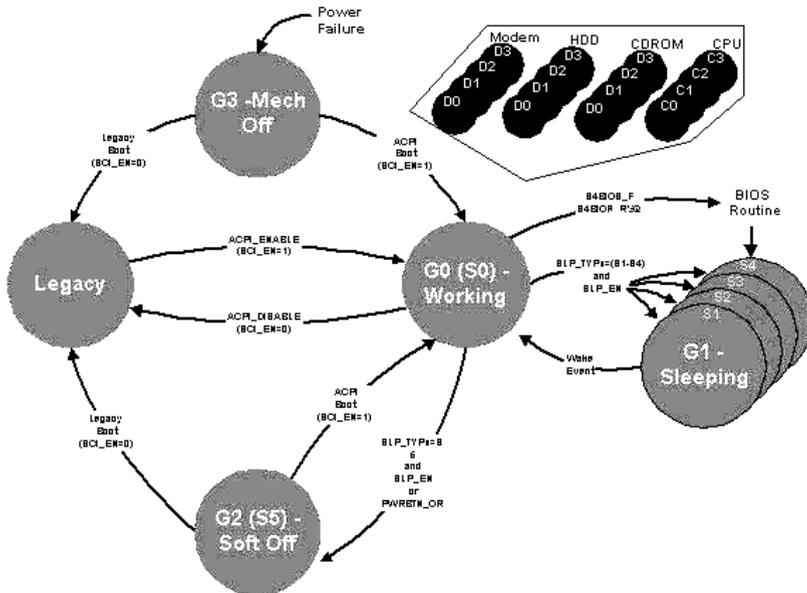


Abb. 1.41 Die Zustände des ACPI-Systems und ihre Übergänge

Ausgehend von einem Nicht-ACPI-Zustand („Legacy“) gelangt das System in einen von vier Zuständen: den Arbeitszustand G0, den Schlafzustand G1 mit seinen vier Stufen S1,S2,S3,S4, dem „beinahe-Aus“-Zustand G2 oder echten, elektrisch getrennten „Aus“-Zustand G3. Bis auf den G3-Zustand können alle Zustände

durch Signale wieder verlassen werden; bei G2 ist zusätzlich ein Neustart des Betriebssystems notwendig.

Das System kann je nach Anwendung alle Zusatzgeräte wie Modem, Netzkarte, Festplatte oder sogar die CPU in mehreren Stromsparstufen betreiben, wobei bei niedrigem Niveau mehr Teile vom Strom abgetrennt werden. Im Zustand D3 ist das Gerät komplett vom Strom getrennt und verliert alle noch verbliebenen Kontextinformationen, die beim Starten erst durch den Treiber wiederhergestellt werden müssen. Die ACPI-konformen Treiber haben deshalb von allen wichtigen Gerätedaten („Register“) Kopien in Form sog. „Schattenregister“.

Bei der CPU äußert sich die Energiesparfunktion nicht nur in einer geringeren Taktfrequenz, sondern auch in einer geringeren Betriebsspannung und damit geringeren Leistungsaufnahme.

Zusammenfassend können wir feststellen, dass die Energieverwaltung ACPI keine Software oder Hardware ist, sondern als Spezifikation auf vorhandenen Funktionen aufbaut. Fehlen diese Funktionen bei nur einem Gerät oder Gerätetreiber, so kann ACPI für das Gesamtsystem nicht funktionieren.

Literatur

- D. A. Patterson: *Reduced Instruction Set Computers*, Comm. of the ACM, 28(1), 1985
D. Tabak, *RISC Architecture* J. Wiley, 1987;
M. Slater, *A Guide to RISC Microprocessors* (1992)
UNI: siehe <http://www.unicode.org> im World Wide Web
Custer, H.: *Inside Windows NT*. Microsoft Press, Redmond, Washington 1993
Custer, H.: *Inside the Windows NT File System*. Microsoft Press, Redmond, Washington 1994
MacWilliams, F., Sloane, N.: *The Theory of Error-Correcting Codes*. North-Holland, Amsterdam 1986
ACPI-Home: <http://www.acpi.info>
ACPI-Linux: <http://acpi.sourceforge.net/>



<http://www.springer.com/978-3-540-20911-9>

Kompendium der Informationstechnologie
Hardware, Software, Client-Server-Systeme, Netzwerke,
Datenbanken

Brause, R.

2005, X, 262 S., Hardcover

ISBN: 978-3-540-20911-9