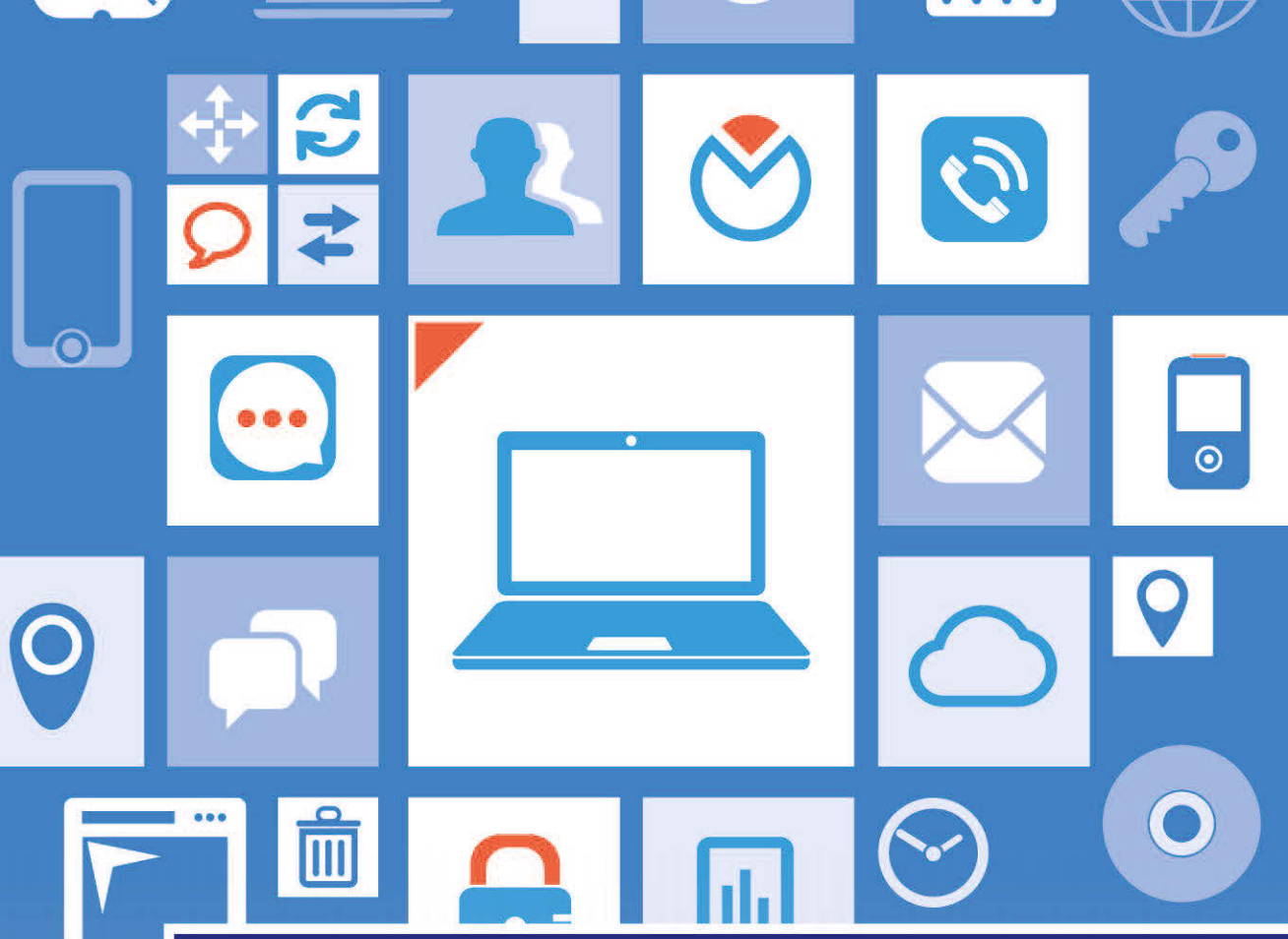

Moderne Betriebssysteme

4., aktualisierte Auflage

Andrew S. Tanenbaum
Herbert Bos

EXTRAS ONLINE

ALWAYS LEARNING

PEARSON

Moderne Betriebssysteme

Moderne Betriebssysteme

4., aktualisierte Auflage

Andrew S. Tanenbaum
Herbert Bos

ALWAYS LEARNING

PEARSON

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.dnb.de* abrufbar.

10 9 8 7 6 5 4 3 2 1

18 17 16

ISBN 978-3-86894-270-5 (Buch)
ISBN 978-3-86326-776-6 (E-Book)

Lilienthalstr. 2, D-85399 Hallbergmoos/Germany
www.pearson.de
A part of Pearson plc worldwide
Programmleitung: Birger Peil, bpeil@pearson.de
Übersetzung: Dipl.-Inf. Katharina Pieper
Fachlektorat: Professor Dr. Harald Görl
Coverabbildung: Shutterstock
Herstellung: Philipp Burkart, pburkart@pearson.de
Satz: Gerhard Alfes, mediaService, Siegen (www.mediaservice.tv)
Druck und Verarbeitung: Drukkerij Wilco, Amersfoort

Printed in the Netherlands

Inhaltsübersicht

Inhaltsverzeichnis

Für Suzanne, Barbara, Daniel, Aron, Nathan, Marvin, Matilde und Olivia.
Die Liste wächst weiter. (AST)

Für Mariele, Duko, Jip und Spot. Furchterregende Jedis – alle. (HB)

Vorwort

Die vierte Auflage dieses Buchs unterscheidet sich von der dritten Auflage auf vielerlei Weise. Es gibt durch das ganze Buch hinweg eine Reihe von kleinen Änderungen, um das Material auf den aktuellen Stand zu bringen, da Betriebssysteme nicht stillstehen. Das Kapitel über Multimedia-Betriebssysteme wurde ins Web verlegt, in erster Linie, um Platz für neuere Inhalte zu schaffen und um das Buch davor zu bewahren, auf eine völlig unhandliche Größe anzuwachsen. Das Kapitel über Windows Vista ist vollständig entfernt worden, da Vista nicht zu dem Erfolg wurde, den sich Microsoft erhofft hatte. Das Kapitel über Symbian wurde ebenfalls entfernt, da Symbian keine große Rolle mehr spielt. Die Inhalte zu Vista sind jedoch durch Windows 8 und die zu Symbian sind durch Android ersetzt worden. Außerdem wurde ein völlig neues Kapitel zum Thema Virtualisierung und der Cloud hinzugefügt.

Was bringt die vierte Auflage Neues?

- **Kapitel 1 Einführung** wurde stark verändert und an vielen Stellen aktualisiert, doch außer eines neuen Abschnitts zu mobilen Rechnern kamen keine größeren Teile hinzu bzw. wurden entfernt.
- **Kapitel 2 Prozesse und Threads** wurde auf den neuesten Stand gebracht: älteres Material verschwand und neues wurde hinzugefügt. Zum Beispiel behandeln wir jetzt die Synchronisationsprimitive „Futex“ und haben einen Abschnitt darüber hinzugenommen, wie mithilfe von Read-Copy-Update Sperren gänzlich vermieden werden können.
- **Kapitel 3 Speicherverwaltung** konzentriert sich nun mehr auf moderne Hardware anstelle der bisherigen Betonung auf Segmentierung und MULTICS.
- In **Kapitel 4 Dateisysteme** haben wir das Thema CD-ROM-Dateisysteme gekürzt, da diese nicht mehr sehr verbreitet sind, und durch modernere Lösungen (wie Flash-Speicher) ersetzt. Außerdem haben wir RAID-Ebene 6 im Abschnitt über RAID hinzugefügt.
- **Kapitel 5 Eingabe und Ausgabe** hat eine Reihe Änderungen erfahren. Ältere Geräte wie CRTs und CD-ROMs wurden gestrichen, während neue Technologien wie Touchscreens hinzugekommen sind.
- **Kapitel 6 Deadlocks** hat sich im Wesentlichen nicht verändert. Das Thema Deadlocks ist recht stabil, mit wenig neuen Resultaten.
- **Kapitel 7 Virtualisierung und die Cloud** ist komplett neu. Es behandelt die wichtigen Themen Virtualisierung und Cloud. Das Kapitel enthält auch einen Abschnitt über VMware als Fallstudie.
- **Kapitel 8 Multiprozessorsysteme** ist eine aktualisierte Version des bisherigen Materials zu Multiprozessorsystemen. Das Kapitel hat jetzt eine größere Betonung auf Mehrkern- und Vielkernsystemen, die in den letzten Jahren immer wichtiger geworden sind. Cache-Konsistenz ist jüngst zu einem größeren Thema geworden, es wird jetzt hier behandelt.

- **Kapitel 9 IT-Sicherheit** wurde stark überarbeitet und neu angeordnet. Hinzugekommen ist eine Reihe an neuem Material zum Ausnutzen von Codefehlern, Malware und den entsprechenden Abwehrmaßnahmen. Angriffe wie Null-Zeiger-Dereferenz und Pufferüberläufe werden jetzt ausführlicher behandelt. Auch Abwehrmaßnahmen, wie Canaries, das NX-Bit und Adressraum-Randomisierung, werden nun sehr detailliert dargestellt, ebenso wie die Wege, auf denen Angreifer versuchen, sie zu umgehen.
- **Kapitel 10 Fallstudie 1: Linux** hat eine größere Revision erfahren. Das Material zu UNIX und Linux wurde aktualisiert, aber die wesentliche Ergänzung hier ist ein neuer und umfangreicher Abschnitt zum Betriebssystem Android, das auf Smartphones und Tablets sehr verbreitet ist.
- In **Kapitel 11 Fallstudie 2: Windows 8** der dritten Auflage wurde Windows Vista diskutiert. Dieses Material ist von der Darstellung zu Windows 8, insbesondere Windows 8.1, abgelöst worden. Damit wurde die Besprechung von Windows vollständig auf den neuesten Stand gebracht.
- **Kapitel 12 Entwurf von Betriebssysteme** ist eine überarbeitete Version von Kapitel 13 der vorigen Auflage.
- **Kapitel 13 Bibliografie** ist eine gründlich aktualisierte Aufstellung der weiterführenden Literatur. Zusätzlich wurde die Literaturliste auf den neuesten Stand gebracht: sie enthält jetzt Einträge zu 223 neuen Arbeiten, die nach dem Erscheinen der dritten Auflage veröffentlicht wurden.
- **Kapitel 7 Multimedia-Betriebssysteme** aus der letzten Auflage wurde auf die Webseite zum Buch verlagert, um die Größe einigermaßen handhabbar zu behalten.
- Zusätzlich sind die Unterkapitel zu den Forschungsthemen im gesamten Buch von Grund auf überarbeitet worden, sodass sie nun die neueste Forschung im Bereich der Betriebssysteme reflektieren. Zu guter Letzt haben wir allen Kapiteln neue Aufgaben hinzugefügt.

Handhabung des Buchs

Für Dozenten Es gibt zu diesem Buch zahlreiche Lehrhilfen. Ergänzungen für Lehrende können unter *www.pearson-studium.de* gefunden werden. Hierzu gehören PowerPoint-Folien, Softwarehilfsprogramme zum Studium von Betriebssystemen, Labor-Experimente für Studenten, Simulatoren und weiteres Material, das in Seminaren über Betriebssysteme benutzt werden kann. Lehrende, die dieses Buch in einem Seminar einsetzen, sollten auf jeden Fall einen Blick darauf werfen.

Für Studenten Die Kapitel 1–6 decken den typischen Umfang einer einführenden Vorlesung zu Betriebssystemen ab und geben zusätzlich noch sehr wichtige Hintergrundinformationen. Die praktischen Umsetzungen der aufgeführten Konzepte werden in den Kapiteln 10 und 11 an zwei klassischen Vertretern moderner Betriebssysteme veranschaulicht.

Für Studenten stehen auf der Webseite zum Buch Experimente und Übungen sowie die Lösungshinweise zur Verfügung.

Webseite zum Buch

- Die Webseite dieses Buchs steht unter *www.pearson-studium.de*. Am schnellsten gelangen Sie von dort zur Buchseite, wenn Sie in das Feld „Schnellsuche“ die Buchnummer **4270** eingeben.
- PowerPoint-Folien für Dozenten: Auf der Webseite finden Sie die PowerPoint-Folien nur mit allen Abbildungen, die Sie an Ihre eigenen Lehrerfordernisse anpassen können.
- Programmieraufgaben: Die Webseite bietet eine Anzahl von detaillierten Programmier- und Simulationsaufgaben für verschiedene Betriebssysteme.
- Lösungshinweise für die Übungen: Enthalten sind die Lösungsansätze für die Übungsaufgaben im Text und die Programmieraufgaben.
- Außerdem finden Sie auf der Webseite ein kleines Wörterbuch der englischen Fachbegriffe mit der deutschen Übersetzung. Mit dem nebenstehenden QR-Code können Sie dieses Fachwörterverzeichnis auch bequem auf Ihr mobiles Endgerät herunterladen.

Danksagungen

Viele Menschen waren an dieser vierten Auflage beteiligt. Zuallererst ist Prof. Herbert Bos von der Freien Universität Amsterdam zu nennen, der als Mitautor hinzugekommen ist. Er ist ein Experte in den Bereichen Sicherheit, UNIX und Allroundsystemen und es ist großartig, ihn an Bord zu haben. Er hat vieles von dem neuen Material geschrieben, bis auf die Teile, die unten noch erwähnt wird.

Unsere Lektorin Tracy Johnson hat wie immer wundervolle Arbeit dabei geleistet, uns bei der Stange zu halten, alle Teile zusammenzufügen, Feuer zu löschen und das Projekt im Zeitplan fertigzustellen. Wir waren auch sehr glücklich, wieder mit unserer langjährigen Herstellungsleiterin Camille Trentacoste zusammenzuarbeiten. Ihre Kompetenzen in so vielen Bereichen haben uns bei mehr als einer Gelegenheit gerettet. Wir sind froh, dass sie nach mehreren Jahren der Abwesenheit wieder dabei ist. Carole Snyder hat sehr gute Arbeit geleistet, die vielen Personen zu koordinieren, die an diesem Buch beteiligt waren.

Das Material in Kapitel 7 zu VMware (Abschnitt 7.12) hat Edouard Bugnion vom EPFL in Lausanne, Schweiz, geschrieben. Ed war einer der Gründer des Unternehmens VMware und kennt diesen Stoff so gut wie niemand sonst auf der Welt. Wir sind ihm zu großen Dank verpflichtet, dass er ihn mit uns geteilt hat.

Ada Gavrilovska von Georgia Tech, eine Expertin für Linux-Interna, aktualisierte Kapitel 10 der dritten Auflage, das sie auch schon geschrieben hatte. Das Android-Material in Kapitel 10 stammt von Dianne Hackborn von Google, eine der Hauptentwicklerinnen des Android-Systems. Android ist das führende Betriebssystem auf Smartphones, deshalb sind wir sehr dankbar, dass Dianne uns geholfen hat. Kapitel 10 ist nun recht lang und ausführlich, doch die Anhänger von UNIX, Linux und Android

können viel daraus lernen. Es ist vielleicht bemerkenswert, dass die längsten und technischsten Kapitel in diesem Buch von zwei Frauen geschrieben wurden. Von uns Männern sind nur die leichten Themen.

Wir haben Windows jedoch nicht vernachlässigt. Dave Probert von Microsoft aktualisierte Kapitel 11 aus der vorigen Auflage dieses Buchs. Jetzt wird Windows 8.1 in diesem Kapitel ausführlich behandelt. Dave weiß sehr viel über Windows und hat genügend Weitblick, um zu unterscheiden, an welchen Stellen es Microsoft gut gemacht hat und an welchen es nicht gelungen ist. Windows-Anhänger werden dieses Kapitel sicher genießen.

Die Beiträge all dieser Experten haben das Buch erheblich verbessert. Wir möchten ihnen noch einmal für ihre unschätzbare Hilfe danken.

Wir waren zudem in der glücklichen Lage, dass mehrere Personen zu haben, die das Manuskript gelesen und begutachtet haben und die außerdem neue Aufgaben für den Übungsteil am Ende der einzelnen Kapitel vorgeschlagen haben. Dies waren Trude Levine, Shivakant Mishra, Krishna Sivalingam und Ken Wong. Steve Armstrong hat die PowerPoint-Folien für die Dozenten hergestellt, die einen Kurs auf Grundlage dieses Buch abhalten.

Normalerweise bekommen Desk-Editoren und Korrekturleser keine Danksagungen, aber Bob Lentz (Desk-Editor) und Joe Ruddick (Korrektorat) haben ungewöhnlich gründliche Arbeit geleistet. Insbesondere Joe kann den Unterschied zwischen einem Antiqua-Punkt und einem kursiven Punkt aus 20 Meter Entfernung ausmachen. Nichtsdestotrotz übernehmen die Autoren die volle Verantwortung für alle verbliebenen Fehler im Buch. Leser, die Fehler finden, werden gebeten, einen der Autoren zu kontaktieren.

Endlich, zu guter Letzt, Barbara und Marvin sind wie immer wundervoll, in ihrer ureigensten Art. Daniel und Matilde sind großartige Neuzugänge in unserer Familie. Aron und Nathan sind wundervolle kleine Jungs und Olivia ist ein Schatz. Und natürlich möchte ich Suzanne für ihre Liebe und Geduld danken, ganz abgesehen von all den *druiven*, *kersen* und *sinaasappelsap* und anderen landwirtschaftlichen Erzeugnissen. (AST)

Allen voran möchte ich Marieke, Duko und Jip danken. Marieke für ihre Liebe und für ihre Nachsicht in all den Nächten, in denen ich an diesem Buch gearbeitet habe, und Duko und Jip dafür, dass sie mich von dieser Arbeit weggezogen und mir gezeigt haben, dass es wichtigere Dinge im Leben gibt. Zum Beispiel Minecraft. (HB)

Andrew S. Tanenbaum

Herbert Bos

Über die Autoren

Andrew S. Tanenbaum hat einen S.B.-Abschluss des M.I.T. und den Doktortitel der Universität von Kalifornien in Berkeley. Er ist Professor Emeritus für Informatik an der Freien Universität Amsterdam in den Niederlanden. Er war früher Vorsitzender der Advanced School for Computing and Imaging, einer interuniversitären Forschungseinrichtung zu umfassenden parallelen verteilten und bildbearbeitenden Systemen. Außerdem war er ein Akademie-Professor der Königlich-Niederländischen Akademie der Wissenschaften, was ihn davor bewahrt hat, sich in einen Bürokraten zu verwandeln. Ihm wurde der prestigeträchtige European Research Council Advanced Grant zuteil.

In der Vergangenheit hat er über Compiler, Betriebssysteme, Netzwerke und verteilte Systeme geforscht. Sein hauptsächliches Forschungsinteresse gilt heute den zuverlässigen und sicheren Betriebssystemen. Aus all diesen Forschungsaktivitäten gingen mehr als 175 Beiträge in Journalen und auf Konferenzen hervor. Prof. Tanenbaum hat außerdem fünf Bücher verfasst bzw. als Co-Autor mitgewirkt, die in 20 Sprachen übersetzt wurden, von baskisch bis thailändisch. Sie werden an Universitäten auf der ganzen Welt eingesetzt. Alles in allem gibt es 163 Versionen (Kombinationen aus Sprache + Auflage) seiner Bücher.

Professor Tanenbaum hat auch eine beachtliche Zahl von Softwaresystemen entwickelt, allem voran MINIX, ein kleiner UNIX-Klon. MINIX war die direkte Inspiration für Linux und war die Plattform, auf der Linux anfangs entwickelt wurde. Die aktuelle Version von MINIX – MINIX 3 – hat jetzt seinen Schwerpunkt darauf gelegt, ein außerordentlich zuverlässiges und sicheres Betriebssystem zu sein. Prof. Tanenbaum wird seine Arbeit dann als getan ansehen, wenn kein Benutzer mehr weiß, was ein Betriebssystemabsturz ist. MINIX 3 ist ein laufendes Open-Source-Projekt, zu dem Sie eingeladen sind mitzuwirken. Besuchen Sie *www.minix3.org*, um eine kostenlose Kopie von MINIX 3 herunterzuladen, und probieren Sie es einfach einmal aus. Es ist sowohl eine x86- als auch eine ARM-Version verfügbar.

Prof. Tanenbaums Doktoranden erreichten nach ihren Promotionen einen hohen Bekanntheitsgrad. Er ist sehr stolz auf sie. In dieser Beziehung ähnelt er einer Glucke.

Prof. Tanenbaum ist Mitglied der ACM, Mitglied von IEEE und Mitglied der Königlich-Niederländischen Akademie der Wissenschaften. Er hat außerdem zahlreiche wissenschaftliche Preise von ACM, IEEE und USENIX verliehen bekommen. Wenn Sie sehr neugierig auf diese Preise sind, besuchen Sie seine Wikipedia-Seite. Er besitzt außerdem zwei Ehrendoktorwürden.

Herbert Bos erhielt seinen Masterabschluss von der Universität Twente und seinen Doktortitel vom Computer Laboratory der Universität Cambridge im Vereinigten Königreich. Seitdem hat er intensiv im Bereich zuverlässige und effiziente E/A-Architekturen für Betriebssysteme wie Linux aber ebenso für Forschungssysteme, die auf MINIX 3 basieren, gearbeitet. Aktuell bekleidet er eine Professur für System- und Netzwerksicherheit in der Informatikfakultät an der Freien Universität in Amsterdam,

Niederlande. Sein Hauptforschungsgebiet ist Systemsicherheit. Er arbeitet mit seinen Studenten an neuen Wegen, Angriffe zu entdecken und aufzuhalten, Malware zu analysieren und mittels Reverse Engineering zu untersuchen sowie Botnets aufzudecken (bösartige Infrastrukturen, die viele Millionen Rechner umfassen können). Im Jahr 2011 erhielt er einen ERC Starting Grant für seine Forschung zum Reverse Engineering. Drei seiner Studenten haben den Roger-Needham-Preis für beste europäische Doktorarbeiten zu Systemen gewonnen.

Vorwort zur deutschen Ausgabe

Die Betriebssystemforschung ist in den letzten Jahren aufgrund technischer Entwicklungen und dem spannenden Bereich der Sicherheit mobiler Endgeräte wieder stärker in den Vordergrund gerückt und damit hat sich auch das Wissen über Betriebssysteme weiterentwickelt. Diese Weiterentwicklung schlägt sich in der vorliegenden vierten Auflage des aktualisierten, in vielen Abschnitten angepassten und erweiterten Standardlehrbuchs *Modern Operating Systems* von Andrew S. Tanenbaum und um den Co-Autor erweiterten Herbert Bos nieder.

Die Übersetzung ist in vielen Teilen neu entstanden, da auch im Original viele Abschnitte ergänzt und Kapitel teils größeren, teils kleinen Veränderungen unterzogen wurden. Die Übertragung der Begrifflichkeit vom Englischen ins Deutsche wurde nochmals leicht modifiziert und an die Bedürfnisse angepasst, etablierte Begriffe wurden beibehalten. Übliche Bezeichnungen, wie die Namen von Schnittstellenoperationen, wurden bewusst nicht übersetzt, da diese Begriffe üblicherweise auch in deutschsprachigen Artikeln und Lehrveranstaltungen so verwendet werden. Gleiches gilt auch für die Übersetzung der neu enthaltenen Kapiteln zu Android, Virtualisierung und dem großen Kapitel zur IT-Sicherheit. Grundsätzlich wurde an entsprechenden Stellen von der wörtlichen Übersetzung etwas abgewichen und der Spielraum genutzt, um das bestmögliche Verständnis bei der Leserschaft zu erzielen. Die am Ende jedes Kapitels enthaltenen Aufgaben runden das Bild eines vollständigen Lehrbuchs zum Themenkomplex Betriebssysteme ab.

Trotz des in dieser vierten Auflage nochmals vergrößerten Umfangs des Buches lag die Übersetzung im Wesentlichen in einer Hand. An dieser Stelle sei Frau Katharina Pieper ganz herzlich für ihre kompetente und sehr verständnisvolle Art der Übersetzung gedankt. Auf der Basis einer einheitlichen Begriffsbildung konnte so eine Übersetzung in einheitlichem Stil entstehen. Ganz herzlich gedankt sei an dieser Stelle auch der Betreuung durch Herrn Birger Peil von Pearson Deutschland. Ohne seine Unterstützung wäre das äußerst umfangreiche Werk so nicht entstanden.

München *Harald Görl*

Einführung

1

» Ein moderner Rechner besteht aus einem oder mehreren Prozessoren, Arbeitsspeicher, Platten, Druckern, einer Tastatur, einer Maus, einem Bildschirm, Netzwerkschnittstellen und verschiedenen anderen Ein- und Ausgabegeräten – alles in allem ist er also ein komplexes System. Wenn jeder Anwendungsprogrammierer verstehen müsste, wie all diese Dinge im Detail funktionieren, dann würde niemals ein Programm zu Ende geschrieben werden. Zudem ist es eine äußerst anspruchsvolle Aufgabe, all diese Komponenten zu verwalten und sie optimal zu nutzen. Deshalb wurden Computer mit einer zusätzlichen Softwareschicht ausgestattet – dem **Betriebssystem**. Die Aufgabe des Betriebssystems ist es, den Benutzerprogrammen ein besseres, einfaches, klares Modell des Computers zur Verfügung zu stellen und außerdem die verfügbaren Ressourcen zu kontrollieren. Betriebssysteme sind der Gegenstand dieses Buchs.

Die meisten Leser haben vermutlich bereits einige Erfahrungen mit einem Betriebssystem wie z.B. Windows, Linux, FreeBSD oder OS X gemacht – aber der Schein kann trügen: Das Programm, mit dem die Benutzer üblicherweise interagieren, ist die Benutzungsschnittstelle – diese wird **Shell** genannt, wenn sie textbasiert ist, und **GUI** (**grafische Benutzungsoberfläche**, *Graphical User Interface*), wenn sie Icons benutzt. Sie ist in Wahrheit gar kein Teil des Betriebssystems, wenngleich die Benutzungsschnittstelle das Betriebssystem benötigt, um ihre Aufgaben zu erledigen.

Einen allgemeinen Überblick über die hier zur Diskussion stehenden Hauptkomponenten gibt ▶*Abbildung 1.1*. Die unterste Ebene ist die Hardware. Dazu gehören Halbleiterchips, Platinen, Festplatten, Tastatur, Monitor und ähnliche physische Geräte. Über der Hardware ist die Software angeordnet. Die meisten Rechner haben zwei Operationsmodi: den Kernmodus und den Benutzermodus. Das Betriebssystem als die grundlegende Software läuft im **Kernmodus** (*kernel mode*, auch **Supervisormodus** genannt). In diesem Modus hat das Betriebssystem vollständigen Zugang zur gesamten Hardware und kann jeden Befehl ausführen, zu dem die Maschine imstande ist. Der Rest der Software läuft im **Benutzermodus** (*user mode*), in dem nur eine Teilmenge der Maschinenbefehle verfügbar ist. Insbesondere sind jene Befehle, die die Kontrolle über die Maschine betreffen oder mit **Ein- und Ausgabe** (**I/O**, *Input/Output*) zu tun haben, für Programme im Benutzermodus tabu. Wir werden auf diesen Unterschied zwischen Kern- und Benutzermodus im Laufe des Buchs wiederholt zurückkommen. Diese Unterscheidung ist wichtig um zu verstehen, wie das Betriebssystem arbeitet.

Die Benutzungsschnittstelle (Shell oder GUI) ist die unterste Softwareebene im Benutzermodus. Sie erlaubt es dem Benutzer, andere Programme wie Webbrowser, E-Mail-Programm oder Musikspieler zu starten, die das Betriebssystem ebenfalls intensiv nutzen.

Die Einordnung des Betriebssystems ist in ▶*Abbildung 1.1* zu sehen: Es liegt direkt über der blanken Hardware und ist somit die Basis für die gesamte übrige Software.

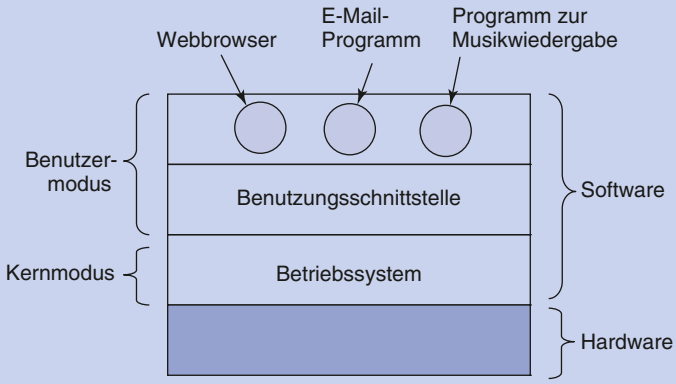

Abbildung 1.1: Die Einordnung des Betriebssystems.

Ein wichtiger Unterschied zwischen Betriebssystem und normaler Software (die im Benutzermodus läuft) ist: Falls ein Benutzer z.B. ein bestimmtes E-Mail-Programm nicht mag, steht es ihm frei, ein anderes zu benutzen oder auch sein eigenes zu schreiben. Es steht ihm aber nicht frei, seine eigene Timer-Unterbrechungsroutine zu schreiben, die Teil des Betriebssystems ist und von der Hardware gegen Versuche der Benutzer, diese zu modifizieren, geschützt ist.

Dieser Unterschied ist allerdings in manchen Systemen verwischt, beispielsweise in eingebetteten Systemen, die keinen Kernmodus haben, oder in interpretierten Systemen wie Java-basierten Systemen, die zur Trennung der Komponenten die Interpretation und nicht die Hardware benutzen.

Außerdem gibt es in vielen Systemen Programme, die zwar im Benutzermodus arbeiten, aber dennoch das Betriebssystem unterstützen oder privilegierte Funktionen ausführen. Beispielsweise existiert in gängigen Systemen meist ein Programm, das Benutzern die Möglichkeit gibt, ihre Passwörter zu ändern. Dieses Programm ist eigentlich kein Teil des Betriebssystems und läuft daher nicht im Kernmodus. Trotzdem führt es sicherheitskritische Funktionen aus und muss deshalb besonders geschützt werden. In manchen Systemen wurde dieser Ansatz bis zum Extrem getrieben und einige Teile, die traditionell zum Betriebssystem gehören (wie etwa das Dateisystem), laufen im Benutzermodus. Bei diesen Systemen ist es schwierig, eine klare Grenze zwischen Kern- und Benutzermodus zu ziehen. Alle Programme, die im Kernmodus arbeiten, sind sicherlich ein Teil des Betriebssystems, aber auch einige Programme im Benutzermodus können zum Betriebssystem gehören oder hängen zumindest eng damit zusammen.

Betriebssysteme unterscheiden sich von Benutzerprogrammen (d.h. Anwendungsprogrammen) aber noch auf andere Weise als nur dadurch, wie sie strukturiert sind. Insbesondere sind sie riesig, komplex und langlebig. Der Quellcode des Kernstücks eines Betriebssystems wie Linux oder Windows liegt im Bereich von 5 Millionen Codezeilen. Um eine Vorstellung von dieser Größe zu bekommen, denken Sie sich die 5 Millionen Zeilen ausgedruckt, und zwar in Form eines Buchs mit 50 Zeilen pro Seite und 1000 Seiten pro Band (so etwa in der Größenordnung dieses Buchs). Man bräuchte

dann 100 Bände, um ein Betriebssystem dieser Größe aufzulisten – also einen ganzen Bücherschrank. Stellen Sie sich nun vor, Ihr Job wäre es, ein Betriebssystem zu warten, und am ersten Tag würde Ihnen Ihr Chef solch einen Bücherschrank mit Quellcode bringen und sagen: „Lernen Sie das!“ Und dies ist nur der Lesestoff für den Teil, der im Kern läuft! Wenn wesentliche gemeinsam benutzte Bibliotheken hinzukommen, wächst Windows auf weit über 70 Millionen Codezeilen oder zehn bis zwanzig Bücherschränke an. Und hierbei sind grundlegende Anwendungsprogramme (wie z. B. der Windows Explorer, Windows Media Player und so weiter) noch nicht eingerechnet.

Nun ist klar, warum Betriebssysteme so langlebig sind: Sie sind sehr schwer zu schreiben – und sind sie erst einmal geschrieben, dann ist ihr Besitzer nicht gewillt, sie nach kurzer Zeit wieder auszumustern und von vorne anzufangen. Stattdessen entwickeln sich solche Systeme über einen langen Zeitraum hinweg weiter. So war Windows 95/98/Me ursprünglich ein eigenständiges Betriebssystem und Windows NT/2000/XP/Vista/Windows 7 ein anderes. Für den Benutzer sehen die beiden Systeme jedoch gleich aus, da Microsoft großen Wert darauf gelegt hat, die Benutzungsschnittstelle von Windows 2000/XP/Vista/Windows 7 ähnlich der Schnittstelle desjenigen Systems aussehen zu lassen, das jeweils ersetzt wurde (meistens war dies Windows 98). Dennoch hatte Microsoft sehr gute Gründe, sich von Windows 98 zu lösen. Dazu werden wir kommen, wenn wir Windows detailliert in ▶*Kapitel 11* untersuchen.

Das andere System, das wir (neben Windows) als Beispiel in diesem Buch immer wieder heranziehen werden, ist UNIX mit seinen Varianten und Klonen. Auch UNIX hat sich über Jahre weiterentwickelt. Dabei wurden Versionen wie System V, Solaris und FreeBSD vom Originalsystem abgeleitet. Linux hingegen stellt eine frische Codebasis dar, die allerdings sehr eng an UNIX modelliert und hoch kompatibel dazu ist. Wir werden Beispiele von UNIX durch das gesamte Buch hindurch benutzen und Linux ausführlich in ▶*Kapitel 10* betrachten.

In diesem ersten Kapitel werden wir viele Schlüsselaspekte von Betriebssystemen streifen. Wir gehen auf sie ein, behandeln ihre Geschichte, die unterschiedlichen Arten von Betriebssystemen, einige der Grundkonzepte und ihre Struktur. Wir werden viele dieser wichtigen Themen in späteren Kapiteln dann noch genauer beleuchten. «

1.1 Was ist ein Betriebssystem?

Es ist recht schwierig festzulegen, was ein Betriebssystem genau ist, außer dass es sich um Software handelt, die im Kernmodus läuft – und selbst das ist nicht immer richtig. Ein Teil des Problems besteht darin, dass Betriebssysteme im Wesentlichen zwei Aufgaben übernehmen, die in keinem Zusammenhang zueinander stehen: einerseits Anwendungsprogrammierern (und natürlich Anwendungsprogrammen) saubere Abstraktionen der Betriebsmittel anstelle der unschönen Hardware zur Verfügung zu stellen und andererseits diese Hardwareressourcen zu verwalten. Je nachdem, wer gerade über Betriebssysteme spricht, hört man eher die eine oder die andere Aufgabenstellung heraus. Betrachten wir jetzt beide etwas näher.

1.1.1 Das Betriebssystem als eine erweiterte Maschine

Die **Architektur** (Befehlssatz, Speicherorganisation, Ein-/Ausgabe und Busstruktur) der meisten Computer auf der Ebene der Maschinensprache ist sehr simpel, aber gleichermaßen komplex zu programmieren, insbesondere im Hinblick auf die Ein- und Ausgabe. Um dieses Problem etwas konkreter zu machen, wollen wir uns moderne **SATA**-Festplatten (**Serial ATA**) ansehen, die auf den meisten Rechnern verwendet werden. Eine frühe Version der Schnittstelle zur Platte wird von Anderson (2007) beschrieben: Hier kann man auf 450 Seiten lesen, was ein Programmierer eigentlich wissen müsste, um die Platte zu verwenden. Seitdem wurde die Schnittstelle mehrmals überarbeitet und ist heute noch komplizierter als im Jahr 2007. Zweifellos möchte kein (halbwegs normaler) Programmierer mit dieser Platte auf der Hardwareebene zu tun haben. Diese Aufgabe übernimmt ein Computerprogramm, der **Festplattentreiber** (*disk driver*). Dieser stellt eine Schnittstelle zum Lesen und Schreiben von Plattenblöcken zur Verfügung, ohne hier genauer in die Details zu gehen. Ein Betriebssystem enthält viele unterschiedliche Treiber zur Steuerung der Ein-/Ausgabegeräte.

Doch selbst diese Ebene ist für die meisten Anwendungen noch viel zu nah an der Hardware. Daher bieten alle Betriebssysteme noch eine weitere Abstraktionsebene an, um die Festplatten zu benutzen: Dateien. Mithilfe dieser Abstraktion können Programme Dateien erzeugen, schreiben und lesen, ohne sich um die lästigen Einzelheiten kümmern zu müssen, wie die Hardware tatsächlich arbeitet.

Diese Abstraktion ist der Schlüssel, um all diese Komplexität zu verwalten. Gute Abstraktionen verwandeln eine fast unmögliche Aufgabe in zwei handhabbare Aufgaben. Die erste besteht darin, Abstraktionen zu definieren und zu implementieren. Die zweite ist, diese Abstraktionen zu benutzen, um das vorliegende Problem zu lösen. Wie bereits erwähnt, ist eine Abstraktion, die fast jeder Computernutzer versteht, die Datei. Eine Datei ist eine nützliche Information, wie z.B. ein digitales Foto, eine gespeicherte E-Mail-Nachricht, ein Song oder eine Webseite. Es ist viel leichter, sich mit Fotos, E-Mails, Songs und Webseiten zu befassen, als mit den Details von SATA- oder anderen Plattenspeichern. Die Aufgabe des Betriebssystems ist es nun, gute Abstraktionen zu erzeugen und dann die so erzeugten abstrakten Objekte zu implementie-

ren und zu verwalten. In diesem Buch werden wir viel über Abstraktionen sprechen. Sie sind einer der Schlüssel zum Verständnis von Betriebssystemen.

Dieser Punkt ist so wichtig, dass es sich lohnt, das Ganze noch einmal mit anderen Worten auszudrücken. Bei allem nötigen Respekt vor den Leistungen der Wirtschaftsingenieure, die den Macintosh so sorgfältig gebaut haben – Hardware ist hässlich! Reale Prozessoren, Speicher, Platten und andere Geräte sind sehr kompliziert und bieten den Menschen, die Software dafür schreiben müssen, schwierige, unbeholfene, eigenartige und inkonsistente Schnittstellen an. Manchmal liegt dies an der Notwendigkeit der Rückwärtskompatibilität zu älterer Hardware. Ein anderes Mal ist es ein Versuch, Geld zu sparen. Aber häufig realisieren die Hardwareentwickler einfach nicht (oder es ist ihnen egal), wie viel Ärger sie der Software machen. Eine der Hauptaufgaben des Betriebssystems ist es, die Hardware zu verstecken und stattdessen Programmen (und ihren Programmierern) hübsche, saubere, elegante, konsistente Abstraktionen bereitzustellen. Betriebssysteme verwandeln also etwas Hässliches in etwas Schönes, wie man in ▶ *Abbildung 1.2* sehen kann.

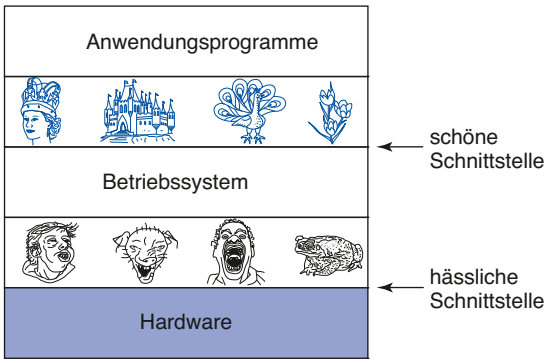

Abbildung 1.2: Betriebssysteme verwandeln die hässliche Hardware in wunderschöne Abstraktionen.

Es sollte noch angemerkt werden, dass die eigentlichen Kunden eines Betriebssystems die Benutzerprogramme sind (über die Anwendungsprogrammierer natürlich). Sie sind diejenigen, die direkt mit dem Betriebssystem und seinen Abstraktionen zu tun haben. Die Endbenutzer haben dagegen mit den Abstraktionen zu tun, die über die Benutzungsschnittstelle zur Verfügung gestellt werden, also entweder eine Kommandozeile oder eine grafische Schnittstelle. Auch wenn die Abstraktionen an der Benutzungsschnittstelle denen ähneln können, die vom Betriebssystem bereitgestellt werden, so ist dies nicht immer der Fall. Um diesen Punkt deutlicher zu machen, werfen wir einen Blick auf den normalen Windows-Desktop und die zeilenorientierte Befehlseingabeaufforderung: Beide Programme laufen unter dem Windows-Betriebssystem und benutzen die Abstraktionen von Windows, trotzdem bieten sie sehr unterschiedliche Benutzungsschnittstellen an. Ähnlich sieht ein Linux-Benutzer, der Gnome oder KDE benutzt, eine ganz andere Schnittstelle als ein Linux-Benutzer, der direkt auf dem X-Window-System arbeitet. Die zugrunde liegenden Betriebssystemabstraktionen sind jedoch in beiden Fällen dieselben.

In diesem Buch werden wir Abstraktionen für die Anwendungsprogramme noch detaillierter studieren. Dagegen werden wir uns eher weniger mit Benutzungsschnittstellen beschäftigen. Dies ist zwar ein großes und wichtiges Thema, hat aber nur am Rande mit Betriebssystemen zu tun.

1.1.2 Das Betriebssystem als Ressourcenverwalter

Das im vorigen Abschnitt dargestellte Konzept eines Betriebssystems, bei dem in erster Linie Abstraktionen für Anwendungsprogramme zur Verfügung gestellt werden, stellt eine Top-down-Sicht dar. Eine alternative Sichtweise ist die Bottom-up-Sicht, bei der das Betriebssystem die Verwaltung aller Bestandteile eines komplexen Systems übernimmt. Moderne Computersysteme bestehen aus Prozessoren, Speichern, Timern, Platten, Mäusen, Netzwerkschnittstellen, Druckern und einer großen Vielfalt weiterer Geräte. Aus dem Bottom-up-Blickwinkel besteht die Aufgabe eines Betriebssystems darin, eine geordnete und kontrollierte Zuteilung (*allocation*) der Prozessoren, Speicher und Ein-/Ausgabegeräte an die anfordernden Programme durchzuführen.

Auf einem modernen Betriebssystem können mehrere Programme zur selben Zeit im Speicher sein und ausgeführt werden. Man stelle sich nun vor, was passieren könnte, wenn drei Programme auf demselben Computer versuchten, gleichzeitig Daten über einen Drucker auszugeben. Die ersten Zeilen könnten von dem ersten Programm stammen, die nächsten vom zweiten, dann folgen Zeilen vom dritten Programm und so weiter – ein wildes Durcheinander entstünde. Das Betriebssystem bringt Ordnung in dieses potenzielle Chaos, indem beispielsweise alle Ausgaben für den Drucker zunächst auf der Platte gepuffert werden. Wenn das erste Programm fertig ist, kann das Betriebssystem dessen Ausgabe aus einer Datei, die auf der Festplatte gespeichert ist, an den Drucker geben. Währenddessen können andere Programme weiter ihre Ausgaben erzeugen, ungeachtet der Tatsache, dass die Ausgabe nicht unmittelbar zum Drucker gelangt.

Wenn ein Computer (oder ein Netzwerk) mehr als einen Anwender hat, dann gewinnen Maßnahmen zum Speicherschutz und auch zur Verwaltung der Ein-/Ausgabegeräte und anderen Betriebsmitteln noch mehr an Bedeutung. Die Benutzer würden sich sonst gegenseitig stören. Oft muss aber nicht nur die Hardware, sondern es müssen auch Informationen (Dateien, Datenbanken usw.) miteinander geteilt werden. Kurz gesagt, dieser Blick auf das Betriebssystem sieht dessen Hauptaufgabe darin, zu überwachen, welches Programm gerade welches Betriebsmittel benutzt, den Betriebsmittelanforderungen nachzukommen, deren Nutzung zu dokumentieren und zwischen konkurrierenden Anforderungen verschiedener Programme und Anwender zu vermitteln.

Die Ressourcenverwaltung beinhaltet **mehrfach genutzte** (*multiplexing*) Betriebsmittel auf zwei unterschiedliche Arten: zeitlich und räumlich. Wenn ein Betriebsmittel zeitlich mehrfach genutzt wird, dann wechseln sich die Programme bzw. Anwender bei der Benutzung ab: Einer nach dem anderen bekommt das Betriebsmittel für eine gewisse Zeit zugeteilt. Bei einem Rechner mit nur einem Prozessor, auf dem mehrere

Programme gleichzeitig laufen, teilt das Betriebssystem beispielsweise die CPU zuerst dem ersten Programm zu. Wenn dieses lange genug gelaufen ist, kann das nächste Programm die CPU benutzen und so fort, bis irgendwann das erste Programm wieder an der Reihe ist. Die Festlegung, wie das Betriebsmittel zeitlich verteilt wird – also wer es als Nächstes zugeteilt bekommt und wie lange –, ist die Aufgabe des Betriebssystems. Ein anderes Beispiel für die zeitliche Mehrfachnutzung ist die Belegung des Druckers. Liegen mehrere Druckaufträge für einen Drucker vor, so muss entschieden werden, welcher Auftrag als Nächster abgearbeitet werden soll.

Die zweite Art der Ressourcenverteilung ist die räumliche Mehrfachnutzung. Anstatt die Aufträge abwechselnd abzuarbeiten, bekommt jeder Nutzer einfach einen Teil der Ressource zugeteilt. Zum Beispiel wird der Arbeitsspeicher üblicherweise zwischen den laufenden Programmen aufgeteilt, sodass alle Programme gleichzeitig im Speicher gehalten werden können (um z.B. die CPU abwechselnd zu benutzen). Wenn genügend Speicherplatz vorhanden ist, so ist dies wesentlich effizienter, als einem Programm den gesamten Platz zur Verfügung zu stellen, vor allem wenn das Programm nur einen Bruchteil des Speichers benötigt. Natürlich treten durch die gemeinsame Nutzung von Ressourcen Probleme wie Fairness, Schutz usw. auf, die das Betriebssystem lösen muss. Eine weitere Ressource, die (räumlich) zwischen vielen Nutzern aufgeteilt wird, ist die Festplatte. In den meisten Systemen kann eine Platte gleichzeitig viele Dateien von vielen unterschiedlichen Benutzern speichern. Die Zuteilung des Speicherplatzes und die Überwachung, wer welchen Plattenblock benutzt, ist eine typische Aufgabe des Betriebssystems.

1.2 Geschichte der Betriebssysteme

Betriebssysteme haben sich über die Jahre hinweg ständig weiterentwickelt. In den folgenden Abschnitten werfen wir einen kurzen Blick auf einige wichtige Stationen dieser Entwicklung. Da Betriebssysteme historisch gesehen sehr eng mit der jeweiligen Rechnerarchitektur verbunden sind, auf der sie ausgeführt werden, wollen wir im Folgenden aufeinanderfolgende Generationen von Computern betrachten, um deren Betriebssysteme zu beleuchten. Diese Zuordnung von Betriebssystemgenerationen zu Rechnergenerationen ist zwar etwas grob, liefert uns aber ein Gerüst für unsere Darstellung.

Die unten dargestellte Entwicklung ist größtenteils chronologisch, aber eben nicht durchgehend, da eine neue Entwicklung nicht wartet, bis die vorherige vollständig abgeschlossen ist. Es gab eine Menge derartiger Überlappungen, nicht zu vergessen die vielen Fehlstarts und Sackgassen. Betrachten Sie die folgenden Ausführungen als eine Orientierungshilfe, nicht als der Weisheit letzter Schluss.

Der erste richtige Digitalrechner wurde von dem englischen Mathematiker Charles Babbage (1791–1871) entwickelt. Obwohl Babbage einen Großteil seines Lebens und seines Vermögens in den Bau seiner „Analytischen Maschine“ investierte, hat er sie doch nie richtig zum Laufen bringen können. Dies lag daran, dass es sich um einen rein mechanischen Entwurf handelte und die Technologie damals noch nicht in der Lage war, die

benötigten Räder, Gestänge und Zahnräder in der geforderten Präzision herzustellen. Es ist wohl unnötig zu sagen, dass die Analytische Maschine kein Betriebssystem besaß.

Im Zusammenhang mit der Analytischen Maschine gibt es eine interessante Geschichte am Rande: Als Babbage erkannte, dass er Software für seine Maschine brauchen würde, stellte er eine junge Dame mit dem Namen Ada Lovelace, die Tochter des bekannten britischen Dichters Lord Byron, als weltweit erste Programmiererin ein. Nach ihr wurde später die Programmiersprache Ada benannt.

1.2.1 Die erste Generation (1945–1955) – auf Basis von Elektronenröhren

Nach den erfolglosen Anstrengungen von Babbage gab es erst einmal wenig Fortschritt bei der Konstruktion digitaler Rechner – bis zur Zeit des Zweiten Weltkriegs, in der eine wahre Flut von Aktivitäten ausgelöst wurde. Professor John Atanasoff und sein Doktorand Clifford Berry bauten an der Universität von Iowa eine Maschine, die heute als der erste funktionierende Digitalcomputer angesehen wird. Sie bestand aus 300 Elektronenröhren. Etwa zur gleichen Zeit baute Konrad Zuse in Berlin seinen Z3 auf Basis von elektromechanischen Relais. 1944 entwarf und programmierte eine Gruppe Wissenschaftler (unter ihnen Alan Turing) in Bletchley Park in England den Colossus, Howard Aiken baute den Mark I an der Harvard-Universität und William Mauchley und sein Doktorand J. Presper Eckert bauten den ENIAC an der Universität von Pennsylvania. Manche dieser ersten Rechner waren binär, manche benutzten Elektronenröhren, manche waren programmierbar, aber alle waren sehr primitiv und brauchten Sekunden, um selbst die einfachste Berechnung durchzuführen.

In dieser Anfangszeit kümmerte sich ein und dieselbe Gruppe von Leuten (in der Regel Ingenieure) um den Entwurf, den Bau, die Programmierung, den Betrieb und die Wartung der jeweiligen Maschine. Die Programmierung erfolgte ausschließlich in Maschinensprache oder – schlimmer noch – durch Verdrahtung von Stromkreisen, indem Tausende von Kabeln auf Steckkarten verbunden wurden, um die Basisfunktionen der Maschine zu steuern. Programmiersprachen (selbst Assemblersprachen) waren damals noch unbekannt. Von Betriebssystemen hatte noch niemand etwas gehört. Der übliche Ablauf für die Programmierer bestand darin, sich an einem Aushang an der Wand für einen gewissen Zeitraum einzutragen, dann in den Maschinenraum zu gehen, seine Steckkarten in den Rechner zu schieben und die nächsten Stunden mit der Hoffnung zuzubringen, dass keine der ca. 20000 Röhren während der Ausführung durchbrennen würde. Nahezu alle Aufgaben waren lediglich unkomplizierte mathematische und numerische Berechnungen, wie die Ausgabe von Tabellen mit Sinus-, Kosinus- oder Logarithmuswerten.

In den frühen 1950er Jahren wurden die Routinearbeiten durch die Einführung von Lochkarten etwas verbessert. Jetzt war es möglich, Programme auf Lochkarten zu schreiben und diese statt der Steckkarten einzulesen. Ansonsten blieb die Prozedur dieselbe.

1.2.2 Die zweite Generation (1955–1965) – Transistoren und Stapelverarbeitungssysteme

Die Einführung der Transistoren Mitte der 1950er Jahre veränderte das Bild radikal. Die Rechner wurden zuverlässig, und zwar in dem Maße, dass sie in Serie hergestellt und an zahlende Kunden verkauft werden konnten – mit der Erwartung, dass sie lange genug funktionierten, um für den Kunden nützlich zu sein. Zum ersten Mal gab es eine klare Trennung zwischen Entwicklern, Herstellern, Operatoren, Programmierern und Wartungspersonal.

Diese Maschinen, jetzt **Großrechner** (*mainframe*) genannt, wurden in großen, klimatisierten Räumen aufbewahrt und von einem Stab professioneller Operatoren betrieben. Ausschließlich große Unternehmen, obere Behörden oder Universitäten konnten sich den Kaufpreis von mehreren Millionen US-Dollar leisten. Um einen **Job** (d.h. ein Programm oder eine Menge von Programmen) auszuführen, entwickelte ein Programmierer die Programme zunächst auf Papier (in FORTRAN oder Assembler) und stanzte diese danach auf Lochkarten. Anschließend brachte er den Lochkartenstapel in den Eingaberaum, wo er ihn einem der Operatoren übergab. Dann ging er Kaffee trinken, bis die Ausgabe fertig war.

Wenn der Rechner seinen aktuellen Job beendet hatte, ging einer der Operatoren zum Drucker, riss den entsprechenden Ausdruck ab und brachte ihn in den Ausgaberaum, wo der Programmierer ihn später abholen konnte. Anschließend nahm er sich den nächsten Lochkartenstapel, der in den Eingaberaum gebracht worden war, und las ihn ein. Wenn ein FORTRAN-Compiler benötigt wurde, so musste der Operator diesen aus einem Aktenschrank holen und ebenfalls einlesen. So wurde viel Rechenzeit durch das Hin- und Herlaufen der Operatoren im Maschinenraum verschwendet.

Angesichts der hohen Kosten für die Rechner wundert es nicht, dass man schnell nach Möglichkeiten zur Reduzierung der Zeitverschwendung suchte. Eine allgemein akzeptierte Lösung war das **Stapelverarbeitungssystem** (*batch system*). Die Idee dahinter war, die Jobs in einer Ablage im Eingaberaum zu sammeln und dann mittels eines kleinen, (relativ) billigen Rechners auf ein Magnetband einzulesen. Einer dieser Rechner war die IBM 1401, die sehr gut Lochkartenstapel einlesen, Bänder kopieren und Ausgaben drucken konnte, aber für numerische Berechnungen weniger gut geeignet war. Für die eigentlichen Berechnungen wurden andere, teurere Rechner wie die IBM 7094 verwendet. Diese Situation ist in ▶ *Abbildung 1.3* dargestellt.

Nachdem eine gute Stunde lang ein Stapel Jobs gesammelt worden war, wurden die Karten auf ein Magnetband geschrieben, das in den Rechnerraum gebracht wurde, wo es in ein Bandlaufwerk eingelegt wurde. Der Operator hat dann ein spezielles Programm geladen (quasi der Vorfahre der heutigen Betriebssysteme), das den ersten Job vom Band gelesen und ihn ausgeführt hat. Die Ausgabe wurde nicht direkt gedruckt, sondern zunächst auf ein zweites Band geschrieben. Nach der Beendigung eines Jobs hat das Betriebssystem automatisch den nächsten Job vom Band eingelesen und ausgeführt. Wenn der gesamte Stapel abgearbeitet war, nahm der Operator die Ein- und Ausgabebänder heraus: Das Eingabeband wurde durch ein Band mit dem nächsten

Stapel ersetzt, das Ausgabeband wurde zu einer 1401 gebracht und dort **offline** (d.h. ohne Verbindung zum Hauptrechner) ausgedruckt.

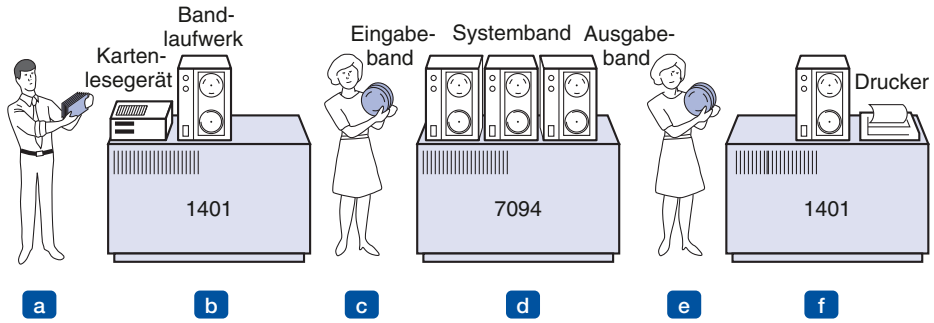

Abbildung 1.3: Ein frühes Stapelverarbeitungssystem. (a) Die Programmierer bringen die Stapel zur 1401. (b) Die 1401 liest den Stapel von Jobs auf ein Band. (c) Ein Operator trägt das Eingabeband zur 7094. (d) Die 7094 führt die Berechnung durch. (e) Ein Operator trägt das Ausgabeband zur 1401. (f) Die 1401 druckt die Ausgabe.

Die Struktur eines typischen Eingabejobs ist in ▶ *Abbildung 1.4* zu sehen. Ein Job begann mit einer $JOB-Karte, auf der die maximale Laufzeit des Jobs in Minuten, die Abrechnungsnummer und der Name des Programmierers angegeben war. Dann folgte eine $FORTRAN-Karte, die das Betriebssystem aufforderte, den FORTRAN-Compiler von einem Systemband zu laden. Direkt dahinter kam das zu übersetzende Programm sowie die $LOAD-Karte, mit der das Betriebssystem veranlasst wurde, das gerade übersetzte Objektprogramm zu laden. (Übersetzte Programme wurden häufig auf Arbeitsbänder geschrieben, die explizit geladen werden mussten.) Als Nächstes folgte die $RUN-Karte, die das Betriebssystem aufforderte, das Programm mit den nun folgenden Daten auszuführen. Schließlich markierte die $END-Karte das Ende des Jobs. Diese einfachen Kontrollkarten sind die Vorläufer der modernen Shells und der Kommandozeileninterpreter.

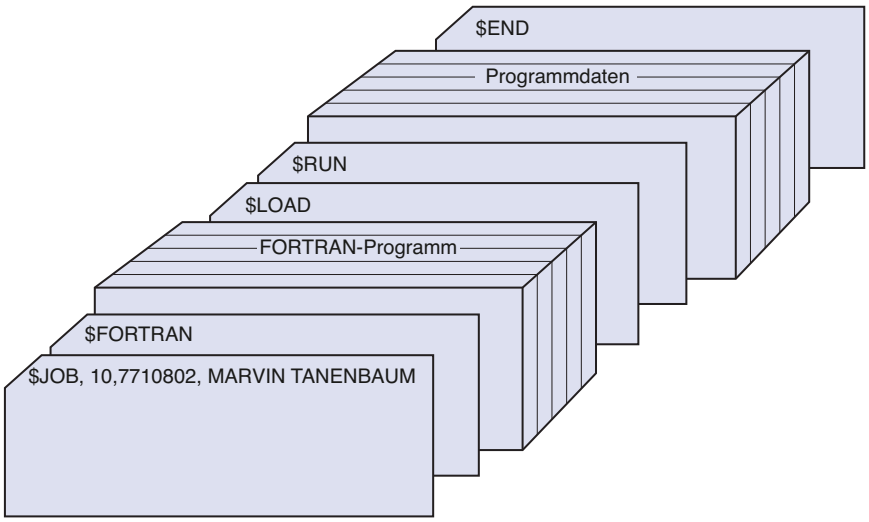

Abbildung 1.4: Struktur eines typischen FMS-Jobs.

Die großen Rechner der zweiten Generation wurden meistens für wissenschaftliche oder technische Berechnungen eingesetzt, wie z.B. das Lösen partieller Differenzialgleichungen, die häufig in der Physik oder in den Ingenieurswissenschaften vorkommen. Programmiert wurde überwiegend in FORTRAN oder in einer Assemblersprache. Typische Betriebssysteme waren das FMS (das FORTRAN Monitor System) und IBSYS, das Betriebssystem von IBM für die 7094.

1.2.3 Die dritte Generation (1965–1980) – integrierte Schaltkreise und Multiprogrammierung

In den frühen 1960er Jahren verfolgten die meisten Computerhersteller zwei miteinander konkurrierende Produktstrategien: Auf der einen Seite gab es die wortorientierten, großen wissenschaftlichen Rechner wie die 7094, die für industrietaugliche numerische Berechnungen in Wissenschaft und Technik verwendet wurden. Auf der anderen Seite gab es die zeichenorientierten, kommerziellen Rechner wie die 1401, die oft für das Sortieren und Ausdrucken von Bändern in Banken und Versicherungen eingesetzt wurden.

Die Entwicklung und Wartung zweier vollständig verschiedener Produktlinien war für die Hersteller eine teure Angelegenheit. Hinzu kam, dass viele der neuen Computerkunden anfangs kleine Rechner benötigten, diese aber später durch leistungsfähigere Maschinen ersetzen wollten, auf denen dann all die alten Programme weiterhin laufen sollten, nur eben schneller.

IBM versuchte durch Einführung des System/360 beide Probleme auf einmal zu lösen. Die 360 war eine Serie von softwarekompatiblen Rechnern, das Spektrum reichte von Modellen von der Größe einer 1401 bis hinauf zu großen Rechnern, die leistungsstärker als die mächtige 7094 waren. Die Maschinen unterschieden sich lediglich im Preis und in der Leistungsfähigkeit (maximaler Speicher, Prozessorgeschwindigkeit, Anzahl der erlaubten Ein-/Ausgabegeräte usw.). Da alle dieselbe Architektur und denselben Befehlssatz hatten, lief ein Programm, das für eine der Maschinen geschrieben war, auch auf allen anderen – zumindest der Theorie nach. (Doch wie Yogi Berra wiederholt feststellte: „In der Theorie sind Theorie und Praxis dasselbe – in der Praxis nicht.“) Da die 360 so konstruiert war, sowohl wissenschaftliche (also numerische) als auch kommerzielle Berechnungen durchzuführen, konnte eine einzige Rechnerfamilie die Bedürfnisse aller Kunden befriedigen. In den folgenden Jahren brachte IBM abwärtskompatible Nachfolger der 360er Linie heraus wie die 370, 4300, 3080 und die 3090, die eine modernere Technologie verwendeten. Die zSeries ist der jüngste Ableger dieser Linie, sie unterscheidet sich allerdings deutlich vom Originalmodell.

Die IBM 360 war die erste bedeutende Computerreihe, die (kleine) **integrierte Schaltkreise** (**IC**, *Integrated Circuit*) verwendete. Damit ergab sich ein wesentlich besseres Preis-Leistungs-Verhältnis als bei den Rechnern der zweiten Generation, die noch aus einzelnen Transistoren aufgebaut waren. Die Serie war ein unmittelbarer Erfolg und die Idee, Familien von kompatiblen Rechnern zu bauen, wurde bald von allen anderen bedeutenden Herstellern übernommen. Die Nachfolger dieser Rechner sind auch

heute noch in großen Rechenzentren im Einsatz, sie werden hauptsächlich als Datenbankserver (z. B. Reservierungssysteme von Fluglinien) oder als WWW-Server verwendet, wo Tausende von Anfragen in der Sekunde beantwortet werden müssen.

Die größte Stärke dieser Strategie von einer Rechnerfamilie war zugleich auch ihre größte Schwäche. Der Plan war eigentlich, dass die gesamte Software – und somit auch das Betriebssystem **OS/360** – auf allen Modellen arbeitet. Die Software sollte sowohl auf kleinen Rechnern laufen, die häufig lediglich die 1401 zum Kopieren von Karten auf Bänder ersetzten, als auch auf großen Systemen, die oft die 7094 ersetzten und für Wettervorhersagen oder andere aufwendige Berechnungen eingesetzt wurden. Sie musste sowohl Systeme mit nur wenigen als auch Systeme mit sehr vielen Peripheriegeräten gut handhaben können. Sie musste in kommerziellen wie in wissenschaftlichen Umgebungen einsatzfähig sein. Und darüber hinaus musste sie für all diese unterschiedlichen Anwendungen effizient arbeiten.

Eine Software zu entwickeln, die all diesen, teils widersprüchlichen Anforderungen gerecht wurde, war für IBM (und natürlich auch für jeden anderen) unmöglich. Das Resultat der Bemühungen war ein gewaltiges und außergewöhnlich komplexes Betriebssystem, das um mindestens zwei bis drei Größenordnungen umfangreicher war als FMS. Es bestand aus Millionen Zeilen Assemblercode, geschrieben von Tausenden von Programmierern – und enthielt Abertausende von Fehlern, deren Korrektur eine kontinuierliche Folge neuer Versionen notwendig machte. Jede neue Version beseitigte einige Fehler und produzierte gleichzeitig neue, sodass die Anzahl der Fehler vermutlich während der ganzen Zeit konstant blieb.

Einer der Entwickler von OS/360, Fred Brooks, schrieb später ein witziges und scharfsinniges Buch über seine Erfahrungen mit diesem Betriebssystem (Brooks, 1995). Natürlich ist es unmöglich, das Buch hier zusammenzufassen, aber vielleicht reicht es zu sagen, auf dem Bucheinband eine Herde von prähistorischen Tieren zu sehen ist, die in einer Teergrube festsitzen. Der Einband des Buchs von Silberschatz et al. (2012) spielt ebenfalls auf den Vergleich von Dinosauriern mit Betriebssystemen an.

Trotz der enormen Größe und der vielen Probleme stellten sowohl OS/360 als auch ähnliche Betriebssysteme der dritten Generation von anderen Computerherstellern die meisten Kunden einigermaßen zufrieden. Zudem wurden einige Schlüsseltechniken eingeführt, die in der zweiten Generation noch fehlten. Das wahrscheinlich bedeutendste dieser neuen Konzepte war die **Multiprogrammierung** (*multiprogramming*). Wenn auf der 7094 der aktuelle Job pausierte, um auf ein Band oder auf Beendigung anderer Ein-/Ausgabeoperationen zu warten, saß die CPU einfach ungenutzt herum. Solange hauptsächlich wissenschaftliche Berechnungen durchgeführt werden, treten wenig Ein-/Ausgabeoperationen auf, die verlorene Zeit spielt dann kaum eine Rolle. In der kommerziellen Datenverarbeitung dagegen kann die Ein-/Ausgabewartezeit oft 80 oder 90 % der Laufzeit betragen. Es musste also etwas gefunden werden, um zu verhindern, dass die (teure) CPU so oft unausgelastet ist.

Dies wurde gelöst, indem der Speicher in mehrere Bereiche aufgeteilt wurde, sodass jeder Job seine eigene Partition besaß, wie in ▶ *Abbildung 1.5* dargestellt. Während der

eine Job auf die Beendigung seiner Ein-/Ausgabe wartet, kann ein anderer Job die CPU nutzen. Wenn gleichzeitig genug Jobs im Arbeitsspeicher gehalten werden können, kann die CPU nahezu 100 % der Zeit beschäftigt werden. Dies erfordert allerdings spezielle Hardware, um die Jobs gegen das Ausspionieren und Beschädigen durch andere Jobs zu schützen. Die 360 und andere Systeme der dritten Generation waren bereits mit derartiger Hardware ausgestattet.

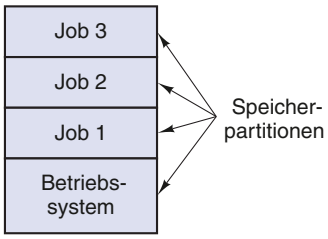

Abbildung 1.5: Ein Multiprogrammiersystem mit drei Jobs im Speicher

Eine weitere wichtige Eigenschaft, die mit den Betriebssystemen der dritten Generation aufkam, war die Fähigkeit, die Jobs von Karten einzulesen und auf Platten abzuspeichern, sobald sie in den Rechnerraum gebracht wurden. Immer wenn ein laufender Job beendet wurde, konnte das Betriebssystem einen neuen Job von der Platte in die soeben frei gewordene Partition laden und ihn dort ausführen. Diese Technik wird als **Spooling** bezeichnet (abgeleitet von **Simultaneous Peripheral Operation On Line**) und wurde auch für die Ausgabe verwendet. Mit der Einführung des Spooling wurde die 1401 nicht länger benötigt und das Herumtragen der Bänder entfiel größtenteils.

Obgleich die Betriebssysteme der dritten Generation gut für umfangreiche wissenschaftliche Berechnungen und riesige kommerzielle Datenverarbeitung geeignet waren, so waren sie doch im Kern noch Stapelverarbeitungssysteme. Viele Programmierer trauerten den Tagen der ersten Generation nach, als sie die Maschine einige Stunden, in denen sie ihre Programme schnell austesten und korrigieren konnten, für sich alleine zur Verfügung hatten. Bei Systemen der dritten Generation betrug die Zeit zwischen dem Abschicken eines Jobs und dem Eintreffen der Ausgabe häufig mehrere Stunden, sodass auch nur ein einziges falsch gesetztes Komma die Übersetzung eines Programms scheitern lassen konnte – und der Programmierer verlor schnell mal einen halben Tag. Man kann sich denken, wie sehr die Programmierer so etwas liebten.

Der Wunsch nach kurzen Antwortzeiten ebnete den Weg für das **Timesharing** als einer Variante der Multiprogrammierung, bei der jeder Benutzer online Zugang zum System über ein Terminal hatte. Wenn 20 Benutzer in einem Timesharing-System eingeloggt sind, von denen 17 aber nur nachdenken, sich unterhalten oder Kaffee trinken, kann die CPU abwechselnd den drei Jobs zugeteilt werden, die Rechenzeit benötigen. In der Regel erteilen Programmierer beim Korrigieren ihrer Programme eher schnell auszuführende Kommandos (z.B. „Übersetze eine fünf Seiten lange Funktion“[1]) als Kommandos, deren Ausführung lange dauert (z.B. „Sortiere ein Band mit einer Million

1 Wir verwenden die Begriffe „Funktion“, „Prozedur“ und „Unterroutine“ im Folgenden synonym.

Sätzen“). Deshalb kann der Computer vielen Benutzern gleichzeitig einen schnellen, interaktiven Dienst anbieten und eventuell zusätzlich im Hintergrund umfangreiche Stapeljobs verarbeiten, falls die CPU noch nicht ausgelastet ist. Das erste universale Timesharing-System **CTSS** (**Compatible Time Sharing System**) wurde am M.I.T. auf einer speziell modifizierten 7094 entwickelt (Corbató et al., 1962). Trotzdem wurde es so lange nicht eingesetzt, bis sich der benötigte Hardwareschutz in der dritten Generation durchsetzte, der die Jobs und deren Daten voreinander schützte.

Nach dem Erfolg des CTSS beschlossen M.I.T., Bell Labs und General Electric (zu der Zeit einer der großen Computerhersteller), mit der Entwicklung eines „Rechenwerkzeugs“ zu beginnen, also mit einer Maschine, die Hunderte von Timesharing-Benutzern gleichzeitig unterstützen sollte. Das Vorbild hierfür war die Elektrizitätsversorgung – wenn man Strom braucht, steckt man nur einen Stecker in die Wand und man bekommt (solange es im angemessenen Rahmen bleibt) so viel man will. Die Entwickler dieses Systems, das als **MULTICS** (**MULTiplexed Information and Computing Service**) bekannt wurde, stellten sich eine riesige Maschine vor, die genügend Rechenkapazität für alle Einwohner von Boston bereitstellen sollte. Die Vorstellung, dass Maschinen, die 10000-mal schneller als ihr GE-645 waren, nur 40 Jahre später (für weit unter 1000 US-Dollar) millionenfach verkauft würden, war zu dieser Zeit reine Science-Fiction.

MULTICS war nur teilweise ein Erfolg. Es war dafür ausgelegt, mehrere Hundert Benutzer gleichzeitig zu bedienen, mit einer Hardware, die etwa so leistungsstark wie ein Intel 386 PC war, jedoch über wesentlich mehr Ein-/Ausgabekapazität verfügte. Das ist nicht ganz so verrückt, wie es jetzt klingen mag, da die Leute damals noch wussten, wie man kleine, effiziente Programme schreibt – eine Fähigkeit, die heute kaum noch jemand beherrscht. Es gab viele Gründe, warum MULTICS kein weltweiter Erfolg wurde. Einer war sicherlich, dass das System in der Programmiersprache PL/I programmiert war, denn der PL/I-Compiler kam erst viel später, als von den Kunden erwartet wurde, und funktionierte dann kaum, als er endlich verfügbar war. Außerdem war MULTICS ein sehr ehrgeiziges Projekt für die damalige Zeit, in etwa vergleichbar mit der Analytischen Maschine von Charles Babbage im 19. Jahrhundert.

Kurz und gut, MULTICS brachte zwar viele wegweisende Ideen in die Fachliteratur ein, aber daraus ein solides Produkt und einen großen kommerziellen Erfolg zu machen, war viel schwerer, als alle dachten. Bell Labs stiegen aus dem Projekt aus und General Electric verließ die Computerbranche ganz. Das M.I.T. blieb bei MULTICS und konnte es auch zur Marktreife führen. Am Ende konnte MULTICS doch noch als kommerzielles Produkt vertrieben werden, und zwar von der Firma Honeywell, die den Computerzweig von General Electric gekauft hatte. Es wurde in über 80 großen Firmen und Universitäten weltweit installiert. Diese Zahl ist zugegebenermaßen etwas klein, dafür waren die MULTICS-Anwender aber äußerst treu. General Motors, Ford und die nationale Sicherheitsbehörde der USA (NSA) fuhren ihre MULTICS-Systeme beispielsweise erst Ende der 1990er Jahre herunter – 30 Jahre nach der Markteinführung von MULTICS, nachdem jahrelang versucht worden war, Honeywell zum Aufrüsten der Hardware zu bewegen.

Am Ende des 20. Jahrhunderts ist das Konzept des „Rechnerwerkzeugs“ etwas im Sande verlaufen, aber es könnte leicht z.B. in Form von **Cloud-Computing** zurückkehren. Beim Cloud-Computing sind relativ kleine Rechner (wie Smartphones, Tablets und Ähnliche) mit Servern in riesigen, entfernten Datenzentren verbunden, wo die gesamte Rechenleistung stattfindet. Die lokalen Computer sind nur für die Benutzungsschnittstelle zuständig. Die Motivation hierfür ist, dass die meisten Leute die Verwaltung der immer komplizierter und kniffliger werdenden Arbeitsstationen einem professionellen Team überlassen wollen, also zum Beispiel den Leuten, die im Datenzentrum arbeiten. E-Commerce ist ein gutes Beispiel dafür: Viele Unternehmen bieten ihren elektronischen Marktplatz auf Multiprozessorsystemen an, mit denen einfache Client-Maschinen verbunden sind – ganz im Sinn des MULTICS-Entwurfs.

Trotz des fehlenden wirtschaftlichen Erfolgs hatte MULTICS einen prägenden Einfluss auf nachfolgende Systeme (insbesondere auf UNIX und seine Derivate FreeBSD, Linux iOS und Android). Es wird in vielen Veröffentlichungen und einem Buch beschrieben (Corbató et al., 1972; Corbató und Vyssotsky, 1965; Daley und Dennis, 1968; Organick, 1972; Saltzer, 1974). Es gibt auch eine aktive Webseite zu MULTICS unter der Adresse *www.multicians.org*, mit vielen Informationen über das System, seine Entwickler und Benutzer.

Eine andere wichtige Entwicklung zur Zeit der dritten Computergeneration war die phänomenale Verbreitung der Minicomputer, die 1961 mit der DEC PDP-1 begann. Die PDP-1 hatte nur 4-KB-Wörter mit einer Breite von 18 Bit, dafür kostete sie aber lediglich 120000 Dollar (weniger als 5% des Preises einer 7094) und verkaufte sich wie warme Semmeln. Für einige Arten nicht numerischer Aufgaben war sie sogar genauso schnell wie die 7094. Mit der PDP-1 wurde ein völlig neuer Markt geschaffen. Es folgte schnell eine Serie anderer PDP-Rechner (die anders als in der IBM-Familie nicht miteinander kompatibel waren), die ihren Höhepunkt in der PDP-11 fand.

Einer der Informatiker an den Bell Labs, der auch am MULTICS-Projekt mitgearbeitet hatte, Ken Thompson, fand irgendwann einen kleinen PDP-7-Minicomputer, der nicht benutzt wurde, und machte sich daran, eine abgespeckte Einbenutzerversion von MULTICS zu schreiben. Diese Arbeit hat sich später zum Betriebssystem **UNIX** entwickelt, das in der akademischen Welt, in Behörden und vielen Unternehmen sehr beliebt wurde.

Die Geschichte von UNIX kann an vielen Stellen nachgelesen werden (z.B. Salus, 1994). Ein Teil der Geschichte wird in *Kapitel 10* erzählt. An dieser Stelle genügt es zu wissen, dass der Quellcode des Systems frei verfügbar war und deshalb verschiedene Organisationen jeweils eigene, zueinander inkompatible UNIX-Versionen entwickelten, was zu einem Chaos führte. Zwei der wichtigsten Entwicklungen waren **System V** von AT&T und **BSD-UNIX BSD-UNIX** (Berkeley Software Distribution) von der Universität Berkeley in Kalifornien. Diese beiden Versionen hatten zudem auch noch Unterversionen. Damit Programme gemeinsam für alle UNIX-Plattformen geschrieben werden konnten, wurde von der IEEE ein Standard entwickelt, **POSIX**, den heutzutage die meisten UNIX-Systeme unterstützen. POSIX definiert einen Teil der Systemschnittstelle, der von allen POSIX-konformen Systemen eingehalten werden muss. Mittlerweile unterstützen sogar andere Betriebssysteme den POSIX-Standard.

Nebenbei soll erwähnt werden, dass der Autor 1987 einen kleinen UNIX-Klon namens **MINIX** für Ausbildungszwecke entwickelt hat. MINIX ist funktional gesehen UNIX sehr ähnlich und unterstützt ebenfalls den POSIX-Standard. Mittlerweile hat sich das Original zu MINIX 3 weiterentwickelt, welches hochmodular und sehr zuverlässig ist. Es kann fehlerhafte oder auch einfach nur abgestürzte Module (wie z.B. Ein-/Ausgabegerätetreiber) sehr schnell entdecken und ersetzen, ohne einen Neustart erforderlich zu machen bzw. ohne die laufenden Programme zu unterbrechen. Der Fokus von MINIX 3 liegt darauf, extrem hohe Zuverlässigkeit und Verfügbarkeit zu bieten. In dem Buch Tanenbaum und Woodhull (2006) wird die Funktionsweise erklärt, der Quelltext ist im Anhang aufgelistet. Das MINIX-3-System einschließlich des gesamten Quelltextes ist im Internet unter *www.minix3.org* frei erhältlich.

Der Wunsch nach einem (nicht nur für Ausbildungszwecke) freien UNIX-System motivierte den finnischen Informatikstudenten Linus Torvalds dazu, **Linux** zu schreiben. Dieses System wurde mithilfe von MINIX entwickelt und bot anfangs auch einige MINIX-Mechanismen (z.B. das MINIX-Dateisystem). Es wurde seitdem von vielen Leuten in viele Richtungen weiterentwickelt, hat aber immer noch einige Basisstrukturen mit MINIX und UNIX gemeinsam. Dem interessierten Leser sei das Buch von Glyn Moody (2001) empfohlen, in dem die Geschichte von Linux und die Open-Source-Bewegung genauer beschrieben werden. Die meisten unserer Aussagen über UNIX gelten gleichermaßen für System V, BSD, MINIX, Linux sowie für andere Versionen und Klone von UNIX.

1.2.4 Die vierte Generation (1980 bis heute) – der PC

Mit der Entwicklung der LSI-Schaltungen (*Large Scale Integration*) – hochintegrierte Schaltkreise mit damals Tausenden von Transistoren pro Quadratzentimeter Silizium – brach das Zeitalter des PCs (*Personal Computer*) an. Was die Architektur betrifft, waren PCs (anfangs auch **Mikrocomputer** genannt) gar nicht so verschieden von den Minicomputern der PDP-11-Klasse, allerdings unterschieden sie sich deutlich im Preis. Während es Minicomputer einzelnen Abteilungen in Firmen und Universitäten möglich machten, eigene Rechner zu besitzen, ermöglichte es der Mikroprozessorchip sogar Privatpersonen, ihren eigenen PC zu besitzen.

Als Intel 1974 mit dem 8080 die erste Allzweck-8-Bit-CPU auf den Markt brachte, suchte die Firma ein Betriebssystem dafür, um den Prozessor zu testen. Gary Kildall, zu der Zeit ein Berater von Intel, wurde damit beauftragt. Kildall und ein Freund bauten zunächst einen Controller für eine neue Version eines 8-Zoll-Diskettenlaufwerkes von Shugart und schlossen diesen dann an den 8080 an. Damit hatten sie den ersten Mikrocomputer mit Diskettenlaufwerk produziert. Kildall schrieb dann auch ein plattenbasiertes Betriebssystem für das Gerät, das er **CP/M** (**Control Program for Microcomputers**) nannte. Da Intel wohl nicht mit einem Erfolg rechnete, überließen sie die Rechte daran Kildall, als er danach fragte. Gary Kildall gründete daraufhin eine Firma, Digital Research, um CP/M weiterzuentwickeln und zu vertreiben.

1977 wurde CP/M von Digital Research umgeschrieben, damit es auch auf weiteren Rechnern mit 8080-Prozessor, dem Z80 von Zilog und anderen Prozessoren lief. Damals wurden viele Anwendungen für CP/M entwickelt und das Betriebssystem konnte den Markt für Mikrocomputer etwa fünf Jahre lang dominieren.

In den frühen 1980er Jahren begann IBM mit dem Entwurf des IBM-PC und suchte Software, die darauf laufen sollte. So wurde Bill Gates gefragt, ob sein BASIC-Interpreter für den IBM-PC zu lizenzieren wäre. IBM fragte ihn außerdem, ob er ein Betriebssystem für den PC kennen würde. Gates schlug vor, Digital Research zu kontaktieren, damals die führende Firma für Betriebssysteme. Dann kam es zur wohl schlechtesten wirtschaftlichen Entscheidung in der Geschichtsschreibung: Kildall lehnte es ab, sich persönlich mit IBM zu treffen, und schickte stattdessen einen Angestellten. Zu allem Unglück weigerte sich sein Anwalt, eine Geheimhaltungsvereinbarung von IBM zu unterschreiben, die den noch unbekannten PC betraf. Folgerichtig fragte IBM wieder bei Bill Gates nach, ob er ein Betriebssystem liefern könne.

Als IBM auf ihn zurückkam, hatte Gates einen lokalen Computerhersteller ausfindig gemacht, Seattle Computer Products, der ein passendes Betriebssystem hatte: **DOS** (**Disk Operating System**). Er trat an die Firma heran und bot an, das System zu kaufen (für angeblich 75000 US-Dollar), was die Firma bereitwillig annahm. Gates bot IBM nun ein Paket aus DOS und BASIC an, das IBM schließlich kaufte. IBM wollte einige Änderungen, deshalb stellte Gates den Entwickler von DOS, Tim Paterson, in seiner jungen Firma Microsoft ein. Das überarbeitete System wurde in **MS-DOS** (**MicroSoft Disk Operating System**) umbenannt und konnte den IBM-PC-Markt schnell erobern. Eine sehr wichtige (und im Nachhinein auch sehr weise) Entscheidung von Gates war, MS-DOS an die Hardwareherstellern zu verkaufen, sodass diese Hardware und Software zusammen verkaufen konnten. Kildall dagegen versuchte (zumindest anfangs), CP/M den Endkunden einzeln zu verkaufen. Kildall starb später plötzlich und unerwartet unter bis heute ungeklärten Umständen.

Als 1983 der Nachfolger des IBM-PC, der IBM-PC/AT, mit dem Intel-80286-Prozessor herauskam, war MS-DOS bereits etabliert und CP/M lag in den letzten Zügen. MS-DOS wurde später auch noch viel auf den Nachfolgeprozessoren 80386 und 80486 eingesetzt. Die ersten Versionen von MS-DOS waren noch relativ primitiv, doch nach und nach kamen erweiterte Leistungsmerkmale hinzu, darunter viele, die ursprünglich von UNIX stammten. (Microsoft hatte UNIX sehr wohl zur Kenntnis genommen, in den Anfangsjahren verkauften sie sogar eine UNIX-Version namens XENIX für Mikrocomputer.)

CP/M, MS-DOS und die anderen Betriebssysteme für Mikrocomputer wurden alle nur mittels Tastatureingaben bedient. Aufgrund einer Forschungsarbeit von Doug Engelbart am Stanford Research Institute in den 1960er Jahren wurde das schließlich geändert. Engelbart erfand die grafische Benutzungsschnittstelle **GUI** (**Graphical User Interface**), komplett mit Fenstern, Icons, Menüs und Maus. Diese Ideen wurden von Forschern im Xerox PARC übernommen, die sie in ihren eigenen Maschinen einbauten.

Steve Jobs, der in seiner Garage den Apple mitentwickelt hatte, besuchte eines Tages die Forschungsstätte PARC, sah eine GUI und erkannte sofort deren potenziellen Wert – den das Management von Xerox bekanntermaßen nicht gesehen hatte. Dieser strategische Fehler von gewaltigem Ausmaß wird in dem Buch *Fumbling the Future*[2] beschrieben (Smith und Alexander, 1988). Jobs begann daraufhin, einen Apple mit einer GUI zu entwickeln. Dieses Projekt führte zu dem Produkt Lisa, das aber zu teuer und ein kommerzieller Fehlschlag war. Jobs zweiter Versuch, der Apple Macintosh, war dann ein riesiger Erfolg, nicht nur weil er viel billiger als Lisa war, sondern vor allem, weil er **benutzerfreundlich** (*user friendly*) war. Der Macintosh war damit für Leute geeignet, die nicht nur keine Ahnung von Computern hatten, sondern darüber hinaus auch nicht die Absicht hatten, etwas daran zu ändern. In der kreativen Welt des grafischen Designs, der professionellen digitalen Fotografie und der professionellen digitalen Videoproduktion ist der Macintosh immer noch sehr verbreitet und die Anwender sind nach wie vor begeistert. 1999 führte Apple einen Betriebssystemkern ein, der auf dem Mikrokern Mach der Carnigie-Mellon-Universität basierte, welcher wiederum ursprünglich entwickelt wurde, um den Kern von BSD-UNIX zu ersetzen. **Mac OS X** ist somit zwar ein UNIX-basiertes Betriebssystem, besitzt aber eine ganz eigene Schnittstelle.

Als Microsoft sich entschied, einen Nachfolger für MS-DOS zu entwickeln, war man sehr vom Erfolg des Macintosh beeinflusst. Microsoft entwickelte ein GUI-basiertes System namens Windows, welches zunächst auf MS-DOS aufsetzte (das bedeutet, dass es mehr eine Shell war als ein Betriebssystem). In den zehn Jahren von 1985 bis 1995 war Windows nur eine grafische Umgebung oberhalb von MS-DOS. Erst 1995 wurde eine eigenständige Version von Windows verkauft, Windows 95. Dieses System beinhaltete viele Betriebssystemelemente und MS-DOS wurde nur noch zum Hochfahren und als Laufzeitumgebung für ältere MS-DOS-Programme genutzt. 1998 wurde mit Windows 98 eine leicht veränderte Version dieses Systems herausgegeben. Aber sowohl Windows 95 als auch Windows 98 enthielten immer noch einen großen Anteil 16-Bit-Intel-Maschinencode.

Ein anderes Betriebssystem von Microsoft ist **Windows NT** (NT steht für **New Technology**). Es war bis zu einem gewissen Grad kompatibel zu Windows 95, wurde aber von Grund auf neu geschrieben und war ein reines 32-Bit-System. Der Chefdesigner von Windows NT ist Dave Cutler, einer der Entwickler des Betriebssystems VAX VMS, weshalb auch einige Ideen von VMS in NT eingeflossen sind. Tatsächlich waren dies so viele, dass DEC, der VMS gehört, Microsoft verklagt hat. Der Fall endete in einem Vergleich, bei dem eine Geldsumme gezahlt wurde, für die man viele Nullen bräuchte, um sie aufzuschreiben. Microsoft erwartete, dass schon die erste Version von NT sowohl MS-DOS als auch andere Windows-Systeme verdrängen würde, da es ein deutlich besseres System war – aber es scheiterte kläglich. Erst mit Windows NT 4.0 gelang der große Durchbruch, vor allem bei Unternehmensnetzwerken. Version 5 von Windows NT wurde zu Beginn des Jahres 1999 in Windows 2000 umbenannt und war als Nachfolger sowohl von Windows 98 als auch von Windows NT 4.0 geplant.

2 In Deutsch unter dem Titel *Das Milliardenspiel. Xerox' Kampf um den ersten PC* erschienen.

Dies gelang allerdings auch nicht richtig, deshalb brachte Microsoft noch eine neue Version von Windows 98 heraus, die **Windows Me** (**Millenium Edition**). 2001 erschien mit Windows XP eine leicht verbesserte Version von Windows 2000. Diese war bis zu ihrer Ablösung deutlich länger auf dem Markt (sechs Jahre) und ersetzte in diesem Zeitraum so gut wie alle vorherigen Windows-Versionen.

Doch die Veröffentlichung von neuen Versionen hielt unvermindert an. Nach Windows 2000 teilte Microsoft die Windows-Familie in eine Client- und eine Server-Linie auf. Die Client-Linie basierte auf XP und seinen Nachfolgern, während die Server-Linie die Systeme Windows Server 2003 und Windows 2008 enthielt. Eine dritte Produktlinie für die Welt der eingebetteten Systeme erschien ein wenig später. Für jede diese Versionen von Windows gab es eigene Änderungspakete, die sogenannten **Service Packs** – genug Stoff, um einige Administratoren (und Autoren von Lehrbüchern zu Betriebssystemen) in den Wahnsinn zu treiben.

Im Januar 2007 brachte Microsoft endlich den Nachfolger von XP, Windows Vista, heraus. Vista hatte eine neue grafische Schnittstelle, verbesserte Sicherheitsfunktionen und viele neue oder verbesserte Anwendungsprogramme. Microsoft hoffte, dass Vista Windows XP vollständig ersetzen würde, was aber nicht passierte. Im Gegenteil, Vista wurde viel kritisiert und hatte eine schlechte Presse, woran hauptsächlich die hohen Systemanforderungen und die restriktiven Lizenzbedingungen Schuld waren sowie die Unterstützung von **digitaler Rechteverwaltung** (**DRM**, *Digital Rights Management*) – Techniken, die es den Nutzern erschwerten, geschütztes Material zu kopieren.

Mit dem Erscheinen von Windows 7, einer neuen und deutlich weniger ressourcenhungrigen Version des Betriebssystems, beschlossen viele Leute, Vista ganz zu überspringen. Windows 7 hat zwar nicht allzu viele neue Funktionen mitgebracht, doch die Version war relativ klein und ziemlich stabil. In weniger als drei Wochen hatte Windows 7 mehr Marktanteile erreicht als Vista in sieben Monaten. Im Jahr 2012 gab Microsoft den Nachfolger heraus, Windows 8, ein Betriebssystem mit vollständig neuem „Look and Feel“, das ganz auf Touchscreens ausgerichtet ist. Microsoft hofft, dass das neue Design zum vorherrschenden Betriebssystem auf einer breiteren Palette an Geräten wird: Desktops, Laptops, Notebooks, Tablets, Telefonen und Heimkino-PCs. Bisher ist die Marktdurchdringung verglichen mit der von Windows 7 jedoch gering.

Der andere große Kontrahent im PC-Bereich ist UNIX (in all seinen zahlreichen Versionen). UNIX ist der Marktführer bei Netzwerk- und Unternehmensservern, ist aber immer häufiger auch auf Desktoprechnern, Notebooks, Tablets und Smartphones anzutreffen. Auf x86-basierten Rechnern wird Linux immer mehr zu einer Alternative zu Windows. Linux wird viel von Studenten und neuerdings vermehrt auch in Unternehmen eingesetzt.

Wir werden übrigens in diesem Buch den Begriff **x86** für alle Intel-Prozessoren verwenden, die zur Familie der Prozessoren mit der typischen Befehlssatzarchitektur gehören, angefangen beim 8086 in den 1970er Jahren. Es gibt eine ganze Reihe dieser Prozessoren, die von verschiedenen Unternehmen wie AMD und Intel hergestellt wer-

den und deren Innenleben sich häufig beträchtlich unterscheidet: Es kann sich um 32-Bit- oder 64-Bit-Prozessoren handeln, mit wenigen oder mit vielen Kernen sowie mit Pipelines, die lang oder kurz sein können, und so weiter. Aus Sicht des Programmierers ähneln sie sich dennoch und alle können noch den 8086-Code verwenden, der vor 35 Jahren geschrieben wurde. An den Stellen, an denen der Unterschied zum Tragen kommt, werden wir jeweils auf die expliziten Modelle verweisen – und verwenden die Bezeichnungen **x86-32** und **x86-64** für die 32-Bit- bzw. 64-Bit-Varianten.

FreeBSD ist ebenfalls ein bekannter UNIX-Ableger, der in dem BSD-Projekt von Berkeley seinen Ursprung hat. Auf allen modernen Macintosh-Rechnern läuft heute eine modifizierte Version von FreeBSD (OS X). UNIX ist außerdem der Standard im Workstation-Bereich mit leistungsstarken RISC-Prozessoren. UNIX-Derivate wie iOS oder Android werden häufig auf mobilen Geräten eingesetzt.

Viele UNIX-Benutzer, insbesondere erfahrene Programmierer, arbeiten lieber mit einer Kommandozeile als mit einer GUI, trotzdem unterstützen fast alle UNIX-Systeme auch ein Fenstersystem, das sogenannte **X-Window-System** (auch bekannt als **X11**), das am M.I.T. entwickelt wurde. Dieses System bietet die grundlegenden Funktionen zur Fensterverwaltung, die dem Benutzer das Erzeugen, Löschen, Verschieben und Verändern von Fenstern auch mit einer Maus ermöglichen. Es existieren auch komplette GUI-Umgebungen, die auf einem X11 aufsetzen, wie etwa **Gnome** oder **KDE**. Diese geben der grafischen UNIX-Umgebung ein einheitliches Erscheinungsbild wie auch unter dem Macintosh oder unter Microsoft Windows, wenn der Benutzer es wünscht.

Eine weitere interessante Entwicklung, die Mitte der 1980er Jahre ihren Anfang nahm, war die Verbreitung von PC-Netzwerken, die **Netzwerkbetriebssysteme** und **verteilte Betriebssysteme** nutzten (Tanenbaum und van Steen, 2007). Bei einem Netzwerkbetriebssystem sind sich die Benutzer des Vorhandenseins mehrerer Rechner bewusst und können sich auf entfernten Maschinen einloggen und Dateien untereinander austauschen. Auf jeder Maschine läuft ein eigenes lokales Betriebssystem, das seine eigenen lokalen Benutzer kennt.

Netzwerkbetriebssysteme unterscheiden sich nicht grundsätzlich von einfachen Betriebssystemen. Sie müssen natürlich über einen Netzwerkschnittstellen-Controller und Steuerungssoftware sowie über Programme für das entfernte Einloggen und für den Zugriff auf entfernte Dateien verfügen. Aber wegen dieser Ergänzungen ändert sich die grundlegende Struktur eines Betriebssystems nicht.

Anders ist das bei einem verteilten Betriebssystem: Für den Benutzer erscheint die Arbeitsumgebung immer wie die eines Einprozessorsystems, selbst wenn sie aus vielen Prozessoren besteht. Der Benutzer muss sich nicht darum kümmern, wo seine Programme ausgeführt werden oder wo seine Dateien abgelegt werden; das sollte alles automatisch und effizient durch das Betriebssystem geschehen.

Echte verteilte Betriebssysteme benötigen mehr, als dass ein bisschen Code zu einem Einprozessorsystem hinzugefügt wird, da sie sich in wesentlichen Punkten von zentralisierten Systemen unterscheiden. Zum Beispiel erlauben es verteilte Systeme oft,

Anwendungen auf mehreren Prozessoren gleichzeitig laufen zu lassen. Sie verfügen deshalb über kompliziertere Schedulingverfahren, um den Grad der Parallelisierung zu optimieren.

Verzögerungen bei der Nachrichtenübertragung bedeuten für diese (und andere) Verfahren oft, mit unvollständigen, veralteten oder sogar falschen Informationen zu arbeiten. Diese Situation unterscheidet sich grundlegend von der bei Einprozessorsystemen, wo das Betriebssystem die volle Kontrolle über den Systemzustand hat.

1.2.5 Die fünfte Generation (1990 bis heute) – mobile Computer

Von dem Tag an, als Dick Tracy aus dem Comicstrip der 1940er Jahre erstmals in sein Zweiwegefunkgerät am Handgelenk gesprochen hat, haben die Menschen ein Kommunikationsgerät herbeigesehnt, das sie überallhin mitnehmen können. Das erste echte Mobiltelefon gab es 1946 und es wog ungefähr 40 kg. Man konnte es überallhin mitnehmen – solange man ein Auto besaß, mit dem man es transportieren konnte.

Das erste richtige Handheld-Telefon erschien in den 1970er Jahren und war – mit ungefähr einem Kilogramm – ein erfreuliches Leichtgewicht. Es wurde liebevoll „the brick“ (Ziegelstein) genannt. Schon bald wollte jeder solch ein Telefon haben. Heute liegt die Handy-Durchdringung in der Weltbevölkerung nahe an 90 %. Wir können nicht nur mit unseren schnurlosen Telefonen und Armbanduhren Anrufe tätigen, sondern bald auch mit Brillen und anderen Kleidungsstücken. Außerdem ist das Telefonieren längst nicht mehr der interessanteste Teil: Wir empfangen E-Mails, surfen durch das Web, schreiben unseren Freunden eine SMS, spielen, navigieren um Verkehrstaus herum – und nehmen dies als selbstverständlich hin.

Obwohl es die Vorstellung, Telefonieren und mobile Computer in einem telefonähnlichen Gerät zu kombinieren seit den 1970er Jahren gibt, tauchte doch das erste echte Smartphone erst Mitte der 1990er Jahre auf, als Nokia das N9000 vorstellte, welches zwei im Grunde getrennte Geräte kombinierte: ein Telefon und ein **PDA** (*Personal Digital Assistant*). 1997 prägte Ericsson den Begriff *Smartphone* für seine GS88 „Penelope“.

Nun, da Smartphones allgegenwärtig sind, ist der Wettbewerb zwischen den unterschiedlichen Betriebssystemen hart und das Ergebnis sogar noch weniger eindeutig als in der PC-Welt. Zum jetzigen Zeitpunkt ist Android von Google das vorherrschende Betriebssystem und iOS von Apple liegt klar an zweiter Position, doch so war es nicht immer und alles kann in ein paar Jahren wieder anders sein. Wenn irgendetwas klar ist in der Welt der Smartphones, dann ist es dieses: Es ist nicht leicht, lange die Spitzenposition zu halten.

So benutzten die meisten Smartphones im ersten Jahrzehnt nach ihrer Einführung die **Symbian-Plattform**. Symbian war das Betriebssystem der Wahl für beliebte Marken wie Samsung, Sony Ericsson, Motorola und insbesondere Nokia. Doch andere Betriebssysteme wie Blackberry OS von RIM (2001 für Smartphones eingeführt) und iOS von Apple (das Betriebssystem für das erste **iPhone**, das im Jahr 2007 eingeführt wurde)

begannen bald, an den Marktanteilen von Symbian zu knabbern. Man erwartete, dass RIM den Markt der Geschäftskunden beherrschen werde, während iOS König der Endverbrauchergeräte werden würde. Der Marktanteil von Symbian fiel drastisch. Im Jahr 2011 beerdigte Nokia das Betriebssystem Symbian und verkündete, dass man sich auf Windows Phone als bevorzugte Plattform konzentrieren wolle. Eine Zeit lang waren Apple und RIM die Stars (auch wenn sie nicht annähernd so vorherrschend waren wie vormals Symbian), doch Android, ein Linux-basiertes Betriebssystem, das von Google im Jahr 2008 herausgegeben wurde, brauchte nicht lange, all seine Rivalen zu überholen.

Für Telefonhersteller hatte Android den Vorteil, dass es quelloffen und unter einer freien und quelloffenen Lizenz verfügbar war. Somit konnten alle Hersteller daran herumbasteln und es leicht an die eigene Hardware anpassen. Außerdem gibt es eine große Entwicklergemeinde für Android, in der Apps geschrieben werden, größtenteils in der gewohnten Programmiersprache Java. Und dennoch: die letzten Jahre haben gezeigt, dass diese Vorherrschaft möglicherweise nicht von langer Dauer ist – die Konkurrenten von Android sind darauf erpicht, sich ein Stück ihrer Marktanteile zurückzuerobern. Wir werden auf Android ausführlich in ▶ *Abschnitt 10.8* eingehen.

1.3 Überblick über die Computerhardware

Ein Betriebssystem ist sehr eng mit der Hardware des Computers verknüpft, auf dem es ausgeführt wird. Es erweitert den Befehlssatz des Rechners und verwaltet dessen Ressourcen. Damit das Betriebssystem arbeiten kann, muss es viel über die zugrunde liegende Hardware wissen, zumindest darüber, wie die Hardware vom Programmierer benutzt werden kann. Aus diesem Grund wollen wir einen kleinen Überblick über die Rechnerhardware geben, so wie sie in modernen PCs zu finden ist. Danach beschreiben wir im Detail, was Betriebssysteme machen und wie sie arbeiten.

Grundsätzlich ähnelt der Aufbau eines einfachen PCs dem Modell in ▶ *Abbildung 1.6*. Die CPU, der Speicher und die Ein-/Ausgabegeräte sind mit einem Systembus verbunden und können darüber miteinander kommunizieren. Modernere PCs haben eine kompliziertere Struktur und damit auch mehrere Busse, worauf wir später noch eingehen werden. Für den Moment ist dieses Modell ausreichend. In den nun folgenden Abschnitten werfen wir einen Blick auf die einzelnen Komponenten und untersuchen einige Hardwareaspekte, die wichtig für Betriebssystementwickler sind. Dies wird natürlich eine sehr kompakte Zusammenfassung werden. Über die Themen Rechnerhardware und Rechnerorganisation sind schon viele Bücher geschrieben worden, darunter die zwei bekannten von Tanenbaum und Austin (2012) und Patterson und Hennessy (2013).

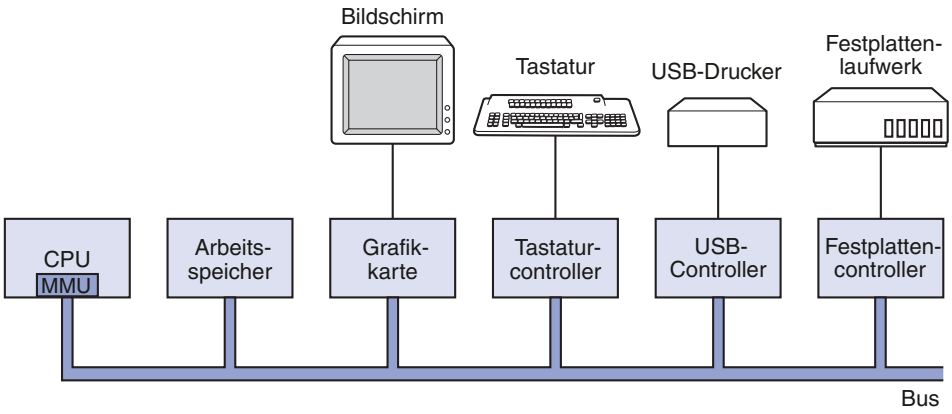

Abbildung 1.6: Einige Komponenten eines einfachen PCs.

1.3.1 Prozessoren

Die CPU ist quasi das „Gehirn" des Computers. Sie holt sich die Befehle aus dem Speicher und führt sie aus. Der grundsätzliche Arbeitsablauf einer CPU besteht darin, den ersten Befehl aus dem Speicher zu laden, diesen zu decodieren, die nötigen Operanden holen und den Befehl dann auszuführen. Anschließend wird der nächste Befehl geladen, decodiert und ausgeführt. Dieser Ablauf wird wiederholt, bis das Programm beendet ist. Auf diese Art werden Programme abgearbeitet.

Jede CPU besitzt eine bestimmte Menge von Befehlen, die sie ausführen kann. Ein x86 kann also kein ARM[3]-Programm und eine ARM kein x86-Programm ausführen. Da der Speicherzugriff zum Holen eines Befehls oder von Daten sehr viel länger dauert als die Ausführung der Anweisung, besitzen alle Prozessoren interne Register, damit wichtige Variablen und temporäre Ergebnisse innerhalb der CPU gespeichert werden können. Deshalb gibt es in der Regel Befehle, um ein Wort vom Speicher in ein Register zu laden und ein Wort vom Register wieder in den Speicher zu schreiben. Andere Befehle kombinieren zwei Operanden aus Registern, dem Speicher oder von beiden in einem einzigen Ergebnis, wie beispielsweise die Addition von zwei Wörtern und das Abspeichern des Ergebnisses in ein Register oder in den Speicher.

Zusätzlich zu den allgemeinen Registern, die Variablen und temporäre Ergebnisse speichern können, besitzen die meisten Rechner auch Spezialregister, die ein Programmierer nutzen kann. Eines dieser Spezialregister ist der **Befehlszähler** (*program counter*), der die Speicheradresse des nächsten Befehls enthält. Nachdem dieser Befehl geladen wurde, wird der Befehlszähler aktualisiert, sodass er jetzt auf die Adresse des nächsten Befehls zeigt.

3 Anm. d. Fachlektors: ARM (*Advanced RISC Machine*) ist eine Prozessorarchitektur der Firma ARM Ltd., die speziell bei mobilen Endgeräten marktführend ist.

Ein anderes Spezialregister ist der **Stackpointer**[4], der auf das Ende des aktuellen Kellers (auch Stack oder Stapel genannt) im Speicher zeigt. Der Stack enthält Rahmen (*frame*) für jede Prozedur, die angesprungen, aber noch nicht verlassen wurde. Dieser Rahmen enthält die Eingabeparameter, lokale Variablen und temporäre Variablen, die nicht in Registern gehalten werden.

Ein weiteres Register ist das **Programmstatuswort** (**PSW**, *Program Status Word*, auch Statusregister genannt). Dieses Register enthält die Statusbits, die bei Vergleichsoperationen gesetzt werden, die CPU-Priorität, den Ausführungsmodus (Benutzer- oder Kernmodus) und einige andere Kontrollbits. Benutzerprogramme können normalerweise das ganze Programmstatuswort lesen, sie dürfen aber nur einige Bits darin beschreiben. Das Programmstatuswort spielt bei Systemaufrufen und bei der Ein-/Ausgabe eine wichtige Rolle.

Das Betriebssystem muss all diese Register genau kennen. Wenn der Prozessor zeitlich mehrfach genutzt wird (Zeitmultiplexen), muss das Betriebssystem oft ein laufendes Programm anhalten, um einem anderen das Weiterarbeiten zu ermöglichen. Jedes Mal, wenn das Betriebssystem ein Programm stoppt, müssen alle Register gespeichert werden, um diese später wiederherstellen zu können, wenn das Programm weiterläuft.

Um die Effizienz zu steigern, haben die Prozessorentwickler schon lange das einfache Modell vom Holen, Decodieren und Ausführen von jeweils nur einem Befehl fallengelassen. Viele der heutigen Prozessoren können mehr als einen Befehl zur gleichen Zeit ausführen. Eine CPU kann beispielsweise über mehrere getrennte Hol-, Decodier- und Ausführungseinheiten verfügen, sodass während der Ausführung von Befehl n bereits der Befehl $n + 1$ decodiert und der Befehl $n + 2$ geholt werden kann. Diese Architektur nennt man **Pipeline**, sie wird in ▶ *Abbildung 1.7a* als Pipeline mit drei Stufen dargestellt, aber eigentlich sind längere Pipelines üblich. Bei den meisten Pipeline-Architekturen muss jeder Befehl, der einmal geladen wurde, auch ausgeführt werden, selbst dann, wenn durch den vorherigen Befehl ein bedingter Sprung ausgeführt wurde. Pipelines bereiten den Entwicklern von Compilern und Betriebssystemen große Kopfschmerzen, weil sich hier die ganze Komplexität der darunterliegenden Maschine offenbart, mit der die Entwickler umgehen müssen.

4 Anm. d. Fachlektors: In älteren wissenschaftlichen Veröffentlichungen wurde für den Begriff des Stackpointers oft „Kellerregister“ oder auch „Stapelregister“ verwendet, was allerdings in neueren Texten praktisch nicht mehr zu finden ist. Im Gegensatz zu früheren Übersetzungen benutzen wir daher den englischen Begriff.

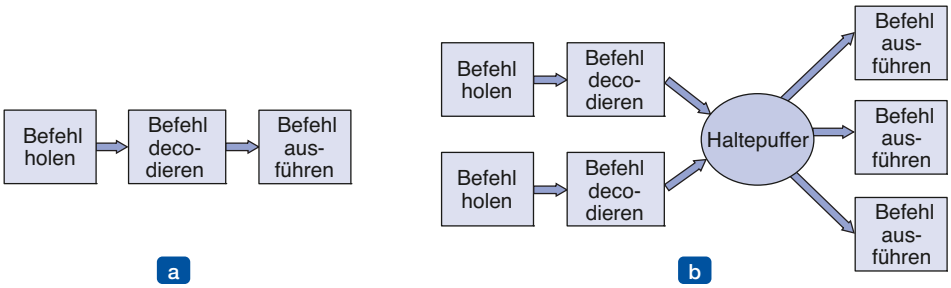

Abbildung 1.7: (a) Eine dreistufige Pipeline; (b) eine superskalare CPU.

Fortschrittlicher als eine Pipeline-Architektur ist die **superskalare** CPU, die in ▶*Abbildung 1.7b* dargestellt ist. Diese Konstruktion umfasst mehrere Ausführungseinheiten, beispielsweise eine für Festkommaarithmetik, eine für Gleitkommarithmetik und eine für boolesche Operationen. Zwei oder mehr Befehle werden auf einmal geholt, decodiert und in einem Puffer abgelegt, bis sie ausgeführt werden. Sobald eine Ausführungseinheit verfügbar wird, überprüft sie, ob in diesem Puffer ein neuer Befehl anliegt, den sie bearbeiten kann. Ist das der Fall, so nimmt sie den Befehl aus dem Puffer heraus und führt ihn aus. Diese Art der Architektur bringt es mit sich, dass Befehle oft in anderer Reihenfolge ausgeführt werden, als sie im Speicher vorliegen. Zum großen Teil ist es Aufgabe der Hardware sicherzustellen, dass das Ergebnis dasselbe ist, wie es bei einer sequenziellen Ausführung entstanden wäre. Aber ein ziemlich lästiger Teil dieser Aufgabe wird auf das Betriebssystem abgeschoben, wie wir noch sehen werden.

Wie bereits erwähnt besitzen die meisten CPUs außer den sehr einfachen Prozessoren in eingebetteten Systemen zwei Modi: den Kernmodus und den Benutzermodus. Normalerweise wird der Modus durch ein Bit im Programmstatuswort gesteuert. Wenn die CPU im Kernmodus läuft, kann jeder Befehl aus dem Befehlssatz ausgeführt und jede Eigenschaft der Hardware ausgenutzt werden. Auf Desktoprechnern und Servern läuft das Betriebssystem in der Regel im Kernmodus und hat damit Zugriff auf die gesamte Hardware. Im Großteil der eingebetteten Systeme läuft ein kleiner Teil im Kernmodus, der Rest des Betriebssystem wird im Benutzermodus ausgeführt.

Anwendungsprogramme arbeiten immer im Benutzermodus, in dem nur auf einen Teil der CPU-Befehle und auf einen Teil der Eigenschaften zugegriffen werden kann. Im Allgemeinen sind alle Befehle, die die Ein-/Ausgabe und den Speicherschutz betreffen, im Benutzermodus nicht erlaubt. Natürlich kann man im Benutzermodus ebenso wenig das entsprechende Bit im Programmstatuswort einfach auf Kernmodus umschalten.

Um einen Dienst vom Betriebssystem zu nutzen, muss ein Benutzerprogramm einen **Systemaufruf** (*system call*) ausführen, der in den Kern springt und das Betriebssystem einbindet. Der Befehl `TRAP` (deutsch: Unterbrechung) schaltet vom Benutzer- in den Kernmodus um und ruft eine Funktion des Betriebssystems auf. Wenn die Arbeit erledigt ist, wird die Kontrolle dem Benutzerprogramm zurückgegeben und der nächste Befehl wird ausgeführt. Wir werden später in diesem Kapitel noch Details der Mecha-

nismen beim Systemaufruf erklären. Vorläufig kann man ihn sich als einen speziellen Prozeduraufruf vorstellen, der die zusätzliche Eigenschaft hat, vom Benutzer- in den Kernmodus zu wechseln. Um Systemaufrufe auch typografisch unterscheidbar zu machen, werden sie im Text wie folgt dargestellt: `read`.

Es sollte noch erwähnt werden, dass viele Computer auch andere Unterbrechungen besitzen als solche, die einen Systemaufruf auslösen. Die meisten werden von der Hardware erzeugt und warnen vor Ausnahmesituationen wie der Division durch 0 oder einem arithmetischen Unterlauf. In allen Fällen erhält das Betriebssystem die Kontrolle und muss entscheiden, was zu tun ist. Manchmal muss das Programm mit einem Fehler abgebrochen werden. Ein anderes Mal kann der Fehler ignoriert werden (eine nicht mehr darstellbar kleine Zahl kann auf 0 gesetzt werden). Und dann gibt es noch die Fälle, in denen das Anwendungsprogramm bestimmte Probleme selbst behandeln möchte, dann kann die Kontrolle an das Programm zurückgegeben werden.

Multithread- und Mehrkernchips

Das Moore'sche Gesetz besagt, dass sich die Anzahl der Transistoren auf einem Chip alle 18 Monate verdoppelt. Dieses „Gesetz" ist kein Naturgesetz wie z. B. der Impulserhaltungssatz, sondern eine Beobachtung des Intel-Mitbegründers Gordon Moore, wie schnell die Verfahrenstechniker in der Halbleiterindustrie in der Lage sind, ihre Transistoren zu verkleinern. Das Moore'sche Gesetz ist nun schon seit über drei Jahrzehnten gültig und es wird erwartet, dass es auch noch mindestens ein weiteres Jahrzehnt gilt. Danach wird der Wert für die Anzahl der Atome pro Transistor zu klein sein und dann wird die Quantenmechanik eine große Rolle dabei spielen, der weiteren Verkleinerung bei der Transistorgröße entgegenzuwirken.

Die Fülle an Transistoren auf einem Chip wirft eine Frage auf: Was kann man mit all diesen Bauteilen machen? Wir haben oben bereits einen Ansatz gesehen: superskalare Architekturen mit mehreren Funktionseinheiten. Aber da die Anzahl der Transistoren weiter steigt, ist auch noch mehr möglich. Ein naheliegendes Vorgehen wäre, größere Cache-Speicher auf den CPU-Chip zu packen. Dies wird auch definitiv passieren, aber letztendlich wird man einen Punkt erreichen, an dem auch diese Möglichkeit ausgeschöpft ist.

Der nächstliegende Schritt wäre, nicht nur die Funktionseinheiten zu vervielfachen, sondern auch Teile der Steuerlogik. Im Intel Pentium 4 ist diese Eigenschaft, die **Multithreading** oder auch **Hyperthreading** (Intels Bezeichnung dafür) genannt wird, für den x86-Prozessor zum ersten Mal eingesetzt worden, mehrere andere CPU-Chips weisen sie ebenfalls auf, darunter SPARC, Power5, Intel Xeon und die Intel-Core-Familie. Als erste Annäherung an dieses Konzept reicht es zu wissen, dass der Prozessor in einem Zustand zwei verschiedene Threads verwalten und innerhalb von Nanosekunden zwischen diesen Threads hin- und herschalten kann. (Ein Thread ist eine Art leichtgewichtiger Prozess, ein Prozess wiederum ist ein in Ablauf befindliches Programm; wir werden uns diese Themen in *Kapitel 2* detaillierter ansehen.) Wenn beispielsweise einer der Prozesse ein Datenwort aus dem Speicher lesen muss (was mehrere Taktzyklen dauert), kann eine Multithread-CPU während der Wartezeit einfach zu einem

anderen Thread umschalten. Multithreading bietet keine echte Parallelität. Es läuft jeweils nur ein Prozess, aber die Zeit zwischen dem Umschalten der Threads wird auf die Größenordnung von Nanosekunden reduziert.

Multithreading hat natürlich Auswirkungen für das Betriebssystem, denn jeder Thread erscheint dem Betriebssystem wie eine separate CPU. Betrachten wir ein System mit zwei Prozessoren, von denen jeder zwei Threads behandeln kann. Dem Betriebssystem erscheint dies wie vier Prozessoren. Wenn es nun zu einem bestimmten Zeitpunkt lediglich Arbeit für zwei Prozessoren gibt, so könnte das Betriebssystem versehentlich zwei Threads demselben Prozessor zuteilen, während der andere völlig unbeschäftigt wäre. Diese Auswahl ist wesentlich ineffizienter, als jeden Prozessor nur einen Thread ausführen zu lassen.

Neben dem Konzept des Multithreading gibt es jetzt viele CPU-Chips mit vier, acht oder mehr vollständigen Prozessoren oder **Kernen** (*core*). Die Mehrkernchips in ▶*Abbildung 1.8* haben jeweils vier Minichips, jeder mit seinem eigenen, unabhängigen Prozessor. (Die Cache-Speicher werden weiter unten erklärt.) Einige Prozessoren wie Xeon Phi von Intel und TilePro von Tilera können mehr als 60 Kerne auf einem einzigen Chip aufweisen. Wenn man solche Mehrkernchips benutzen will, benötigt man auf jeden Fall ein Betriebssystem für Multiprozessoren.

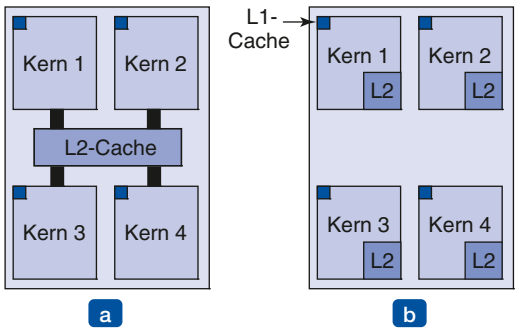

Abbildung 1.8: (a) Ein Vier-Kern-Chip mit einem gemeinsam benutzten L2-Cache; (b) ein Vier-Kern-Chip mit separaten L2-Caches.

1.3.2 Arbeitsspeicher

Die zweite Hauptkomponente eines Computers ist der Speicher. Idealerweise sollte der Speicher äußerst schnell (schneller als die Ausführung eines Befehls, um die CPU nicht aufzuhalten), unglaublich groß und spottbillig sein. Aktuell kann keine Technologie all diesen Ansprüchen gleichzeitig genügen, deshalb muss ein anderer Weg eingeschlagen werden: Das Speichersystem wird als Hierarchie von Schichten aufgebaut, wie in ▶*Abbildung 1.9* zu sehen ist. Die oberste Schicht arbeitet mit höherer Geschwindigkeit, hat eine kleinere Kapazität und erzeugt höhere Kosten pro Bit als die unteren, und zwar oft um einen Faktor von einer Milliarde und mehr.

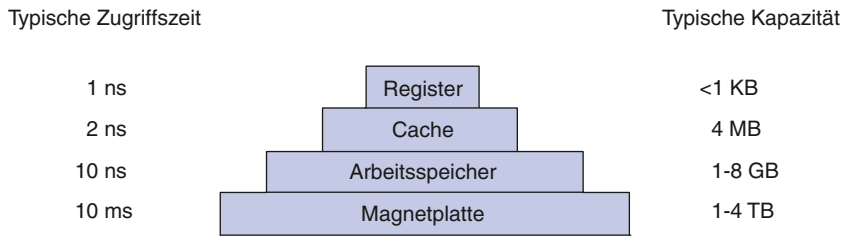

Abbildung 1.9: Eine typische Speicherhierarchie. Die Werte sind sehr grobe Annäherungen.

Die oberste Schicht beinhaltet die internen Register der CPU. Sie sind aus demselben Material hergestellt und deshalb genauso schnell wie die CPU. Folglich gibt es auch keine Verzögerung beim Zugriff. Die verfügbare Speicherkapazität ist typischerweise 32×32 Bit auf einem 32-Bit-Prozessor und 64×64 Bit auf einem 64-Bit-Prozessor, in beiden Fällen weniger als 1 KB. Programme müssen diese Register selbst verwalten (d.h. entscheiden, was darin gespeichert werden soll).

Als Nächstes folgt der Cache-Speicher oder kurz Cache[5], der hauptsächlich von der Hardware gesteuert wird. Der Arbeitsspeicher ist in Blöcke aufgeteilt, die **Cache-Lines** (*cache line*) genannt werden und die typischerweise jeweils 64 Byte groß sind. Die Adressen 0 bis 63 liegen damit in Cache-Line 0, die Adressen 64 bis 127 in Cache-Line 1 und so weiter. Die am häufigsten benutzten Cache-Lines werden in einem Hochgeschwindigkeitscache gehalten, der innerhalb oder sehr nahe der CPU platziert ist. Wenn ein Programm ein Speicherwort lesen will, überprüft die Puffer-Hardware, ob der entsprechende Block im Cache liegt. Falls der Block dort vorhanden ist, nennt man dies einen **Cache-Treffer** (*cache hit*). Die Anfrage ist damit erfüllt und es muss keine Speicheranforderung über den Bus an den Arbeitsspeicher gesendet werden. Cache-Treffer benötigen in der Regel zwei Taktzyklen. Wenn es keinen Treffer gibt, müssen die Daten erst aus dem Speicher geholt werden, was einen erheblichen zeitlichen Nachteil bedeutet. Die Größe des Cache-Speichers ist aufgrund seiner hohen Kosten begrenzt. Manche Maschinen besitzen zwei oder sogar drei Ebenen dieser Caches, jede Ebene jeweils größer und langsamer als die vorhergehende.

Das Konzept des Cachens spielt eine bedeutende Rolle in vielen Bereichen der Informatik, nicht nur beim Cachen von Blöcken im RAM-Speicher. Cachen wird gerne überall dort eingesetzt, wo eine Ressource in Teilstücke unterteilt werden kann, von denen einige stärker als andere benutzt werden, um die Performanz zu verbessern. Betriebssysteme benutzen dieses Prinzip ständig. Zum Beispiel halten viele Betriebssysteme häufig benutzte Dateien (bzw. Teile davon) im Arbeitsspeicher, anstatt sie immer wieder neu von der Platte zu laden. So kann Cachen eingesetzt werden, um lange Pfadnamen wie

/home/ast/projects/minix3/scr/kernel/clock.c

5 Im Deutschen auch Pufferspeicher genannt.

in die Adresse der Datei auf der Platte umzuwandeln und damit wiederholtes Nachschlagen zu vermeiden. Oder wenn die Adresse einer Webseite (URL) in eine Netzwerkadresse (IP-Adresse) umgewandelt wird, kann das Ergebnis für eine spätere Verwendung in einem Cache gespeichert werden. Es gibt noch viele weitere Einsatzmöglichkeiten.

In jedem System, das Caching benutzt, tauchen recht bald einige Fragen auf, zum Beispiel:

1. Wann soll ein neues Element in den Cache gebracht werden?
2. In welcher Cache-Line soll das neue Element gespeichert werden?
3. Welches Element soll aus dem Cache entfernt werden, wenn ein Platz gebraucht wird?
4. Wo soll ein gerade verdrängtes Element in dem größeren Speicher abgelegt werden?

Nicht alle diese Fragen sind bei jedem Caching relevant. Beim Caching von Blöcken des Arbeitsspeichers im CPU-Cache wird im Allgemeinen bei jedem Lesefehler ein neues Element eingebracht. Die zu benutzende Cache-Line wird mithilfe einiger der höherwertigen Bits der angesprochenen Speicheradresse berechnet. Nehmen wir als Beispiel an, dass 4096 Cache-Lines von jeweils 64 Byte und 32-Bit-Adressen zur Verfügung stehen. Dann könnten die Bits 6 bis 17 dazu genutzt werden, um die Cache-Line zu spezifizieren, und mit den Bits 0 bis 5 kann das Byte innerhalb der Cache-Lines adressiert werden. In diesem Fall wird das Element verdrängt, das an dem Speicherplatz steht, wo die neuen Daten hingeschrieben werden sollen. Wenn schließlich eine Cache-Line (deren Inhalt nach dem Laden in den Cache verändert wurde) in den Arbeitsspeicher zurückgeschrieben werden muss, dann wird die Stelle im Arbeitsspeicher, wohin diese Cache-Line zurückgeschrieben wird, einzig und allein durch die angefragte Arbeitsspeicheradresse bestimmt.

Cache-Speicher sind ein derartig gutes Konzept, dass moderne Prozessoren gleich zwei[6] von ihnen haben. Der Cache auf der ersten Ebene, Level-1-Cache oder **L1-Cache**, befindet sich immer innerhalb der CPU und gibt normalerweise decodierte Befehle an die Ausführungseinheit des Prozessors. Die meisten Chips haben einen zweiten L1-Cache für sehr häufig benutzte Datenwörter. Die L1-Caches haben typischerweise eine Größe von 16 KB. Zusätzlich gibt es oft einen zweiten Cache, den **L2-Cache**, der mehrere Megabyte der zuletzt benutzten Speicherwörter enthält. Der Unterschied zwischen L1- und L2-Cache liegt in der Geschwindigkeit. Der Zugriff auf den L1-Cache ist ohne Verzögerung möglich, dagegen bringt der Zugriff auf den L2-Cache eine Verzögerung von 1–2 Taktzyklen mit sich.

Bei Mehrkernchips müssen sich die Entwickler entscheiden, wo sie die Cache-Speicher platzieren. Bei dem Chip in ▶*Abbildung 1.8a* wird ein einziger L2-Cache von allen Kernen gemeinsam genutzt. Dieser Ansatz wird in den Mehrkernchips von Intel

6 Anm. d. Fachlektors: oder auch mehr als zwei.

verwendet. Bei dem Chip in ▶*Abbildung 1.8b* hat dagegen jeder Kern seinen eigenen L2-Cache. Diese Technik setzt AMD ein. Jede Strategie hat ihr Für und Wider: Die gemeinsamen L2-Caches von Intel erfordern zum Beispiel einen komplizierteren Cache-Controller, dagegen ist es bei der Methode von AMD schwieriger, die Inhalte der L2-Caches konsistent zu halten.

In der Hierarchie von ▶*Abbildung 1.9* folgt als Nächstes der Arbeitsspeicher, das Arbeitstier des Speichersystems. Der Arbeitsspeicher wird gewöhnlich **RAM** (**Random Access Memory,** zu Deutsch etwa „Speicher mit wahlfreiem Zugriff", Schreib-Lese-Speicher) genannt. Die Veteranen der Informatik nennen ihn manchmal auch **Kernspeicher** (*core memory*), weil bei Computern in den 1950er und 1960er Jahren kleine magnetisierbare Ferritkerne für den Arbeitsspeicher benutzt wurden. Es gibt sie schon seit Jahrzehnten nicht mehr, doch der Name ist geblieben. Momentan liegen die Speichergrößen im Bereich von Hunderten von Megabytes bis hin zu einigen Gigabytes und sie wachsen schnell weiter. Alle Anfragen des Prozessors, die nicht aus dem Cache beantwortet werden können, werden zum Arbeitsspeicher weitergeleitet.

Zusätzlich zum Arbeitsspeicher haben viele Computer noch eine kleine Menge an nicht flüchtigem Speicher (*nonvolatile RAM*). Diese Art von Speicher verliert anders als RAM seinen Inhalt nicht, wenn der Strom abgeschaltet wird. **ROM** (**Read Only Memory**, Nur-Lese-Speicher oder Festwertspeicher) wird bei der Herstellung schon beschrieben und kann später nicht mehr verändert werden. Er ist schnell und billig. Bei manchen Rechnern wird der Urlader bzw. Bootlader (*bootstrap loader*), ein Programm zur Steuerung des Bootvorgangs, auf dem ROM gespeichert. Auch einige Ein-/Ausgabesteckkarten besitzen ein ROM, um ihre Hardware ansteuern zu können.

EEPROM (**Electrically Erasable ROM**, elektrisch löschbares ROM) und **Flash-Speicher** sind ebenfalls nicht flüchtig, können aber im Gegensatz zum ROM gelöscht und wieder beschrieben werden. Allerdings dauert das Schreiben ein Vielfaches länger als das Beschreiben von RAM. Deshalb wird dieser Speicher benutzt wie ROM, mit der zusätzlichen Eigenschaft, dass nun Fehler in den Programmen dieser Speicher noch nachträglich ausgebessert werden können.

Flash-Speicher werden allgemein auch als Speichermedium in tragbaren elektronischen Geräten verwendet. Sie dienen als Film in Digitalkameras oder als Platte in tragbaren Musikspielern, um nur zwei mögliche Anwendungsbereiche zu nennen. Flash-Speicher liegen bezüglich Geschwindigkeit zwischen RAM und Festplatte. Aber anders als beim Festplattenspeicher nutzt sich Flash-Speicher durch häufige Schreib- bzw. Löschvorgänge viel schneller ab.

Noch eine andere Art von Speicher ist das flüchtige CMOS. Viele Computer nutzen CMOS-Speicher, um Datum und Zeit zu speichern. Der CMOS-Speicher und die für die Uhrzeit verantwortliche Schaltung, die die Zeit hochzählt, werden von einer kleinen Batterie gespeist, sodass die Uhr auch nach dem Abschalten des Rechners noch weiterläuft. Der CMOS-Speicher kann auch die Daten halten, die zur Konfiguration des Computers genutzt werden, beispielsweise von welcher Platte gebootet werden soll. CMOS verwendet man, weil es so wenig Strom verbraucht, dass die eingebaute

Batterie oft mehrere Jahre lang halten kann. Wenn die Batterie allerdings langsam leer wird, zeigt der Rechner Symptome wie bei der Alzheimer-Krankheit: Er vergisst Dinge, die er vorher jahrelang wusste, wie etwa von welcher Festplatte er starten soll.

1.3.3 Festplatten

Magnetische Festplatten sind die nächste Stufe in der Hierarchie. Plattenspeicher ist etwa um den Faktor 100 pro Bit billiger als RAM und besitzt zudem meist 100-mal mehr Kapazität. Das einzige Problem ist, dass der wahlfreie Zugriff etwa 1000-mal länger dauert. Dies liegt daran, dass die Platte ein mechanisches Gerät ist, wie man in ▶ *Abbildung 1.10* sehen kann.

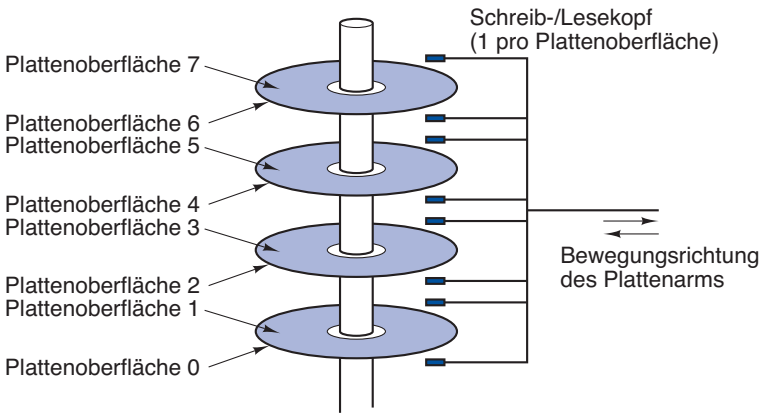

Abbildung 1.10: Struktur einer Festplatte.

Eine Festplatte besteht aus ein oder mehreren metallischen Scheiben, die mit 5400, 7200 oder 10800 Umdrehungen pro Minute rotieren. Ein mechanischer Arm schwenkt über die Scheiben, wie der Tonarm eines alten Plattenspielers für Vinylschallplatten. Daten werden in einer Folge von konzentrischen Kreisen auf die Scheiben geschrieben. In jeder Position des Arms können die Schreib-/Leseköpfe jeweils einen ringförmigen Bereich, die **Spur** (*track*), lesen. Zusammen bilden die Spuren auf allen Platten bei einer bestimmten Armposition einen **Zylinder** (*cylinder*).

Jede Spur wird in eine Anzahl von Sektoren eingeteilt, typischerweise 512 Byte pro Sektor. Bei modernen Festplatten besitzen die äußeren Zylinder mehr Sektoren als die inneren. Die Bewegung des Arms von einem Zylinder zum nächsten benötigt etwa 1 ms. Die Bewegung zu einem beliebigen Zylinder braucht je nach Laufwerk etwa 5 bis 10 ms. Sobald der Arm über der richtigen Spur steht, muss das Laufwerk warten, bis der gewünschte Sektor unter dem Schreib-/Lesekopf erscheint, was nochmals 5 bis 10 ms dauern kann, je nach der Umdrehungsgeschwindigkeit des Geräts. Wenn der Sektor dann unter dem richtigen Kopf ist, können Daten mit 50 MB pro Sekunde bei einfachen und mit 160 MB pro Sekunde bei schnelleren Platten gelesen und geschrieben werden.

Manchmal wird der Begriff „Festplatte“ auch für Speichermedien verwendet, die eigentlich überhaupt keine Platten besitzen, wie **SSDs** (**Solid State Disk**). SSDs haben keine beweglichen Teile, sie enthalten keine Platte in Scheibenform, sondern speichern die Daten in Flash-Speichern. Die einzige Eigenschaft, die sie mit Festplatten teilen, ist, dass auch SSDs eine Menge an Daten speichern, die nicht verloren gehen, wenn der Strom abgeschaltet wird.

Viele Computer unterstützen ein Modell, das als **virtueller Speicher** (*virtual memory*) bekannt ist. Dieses Thema werden wir noch in voller Länge in *Kapitel 3* besprechen. Das Konzept des virtuellen Speichers ermöglicht es, Programme laufen zu lassen, die größer als der physische Speicher sind. Die Programme befinden sich auf der Festplatte und der Arbeitsspeicher wird als eine Art Cache benutzt, in den die am häufigsten ausgeführten Teile eingelagert werden. Die Adressen, die das Programm erzeugt, müssen in die physische Adresse im RAM, wo die Datenwörter abgelegt sind, umgewandelt werden. Diese Zuordnung der Speicheradressen muss sehr schnell erfolgen. Sie wird von der **MMU** (**Memory Management Unit**) erledigt, die ein Teil der CPU ist (▶ *Abbildung 1.6*).

Die Verwendung von Caching-Methoden und der MMU kann bedeutenden Einfluss auf die Performanz haben: Wenn in einem System mit Multiprogrammierung von einem Programm zu einem anderen geschaltet wird – was manchmal als **Kontextwechsel** (*context switch*) bezeichnet wird –, könnte es notwendig sein, den Cache von allen modifizierten Blöcken zu leeren bzw. das Zuordnungsregister der MMU zu ändern. Beides sind teure Operationen und Programmierer geben sich in der Regel die größte Mühe, um sie zu vermeiden. Wir werden einige der Auswirkungen dieser Strategien später ansehen.

1.3.4 Ein-/Ausgabegeräte

Der Prozessor und der Speicher sind nicht die einzigen Ressourcen, die ein Betriebssystem verwalten muss. Auch zwischen Ein- und Ausgabegeräten und dem Betriebssystem gibt es ein intensives Zusammenspiel. Wie man in ▶ *Abbildung 1.6* schon sehen konnte, bestehen Ein-/Ausgabegeräte im Allgemeinen aus zwei Teilen: einem Controller und dem Gerät selbst. Der Controller ist ein Chip (oder auch mehrere Chips), der das Gerät auf der Hardwareebene steuert. Er bekommt Befehle vom Betriebssystem, wie das Lesen von Daten vom Gerät, und führt sie aus.

In vielen Fällen ist die eigentliche Steuerung des Geräts kompliziert und umständlich, deshalb ist es die Aufgabe des Controllers, dem Betriebssystem eine vereinfachte (aber immer noch sehr komplexe) Schnittstelle anzubieten. Ein Festplattencontroller könnte z.B. den Befehl erhalten, den Sektor 11 206 von der zweiten Platte zu lesen. Der Controller muss nun diese lineare Adresse in die Nummern für Zylinder, Spur und Sektor umrechnen. Diese Umrechnung verkompliziert sich möglicherweise dadurch, dass die äußeren Zylinder mehr Sektoren als die inneren haben oder dass einige fehlerhafte Sektoren auf andere ausgelagert wurden. Danach muss der Controller ermitteln, über welchem Zylinder der Schreib-/Lesekopf gerade steht, und einen Befehl geben, um ihn

nach innen oder außen zum richtigen Zylinder zu bewegen. Er muss warten, bis der entsprechende Sektor unter dem Kopf erscheint, und kann dann das Lesen starten und die gelesenen Bits abspeichern, so wie sie von der Platte geliefert werden. Dabei muss die Präambel entfernt und die Prüfsumme berechnet und verglichen werden. Schließlich müssen die ankommenden Bits zu Wörtern zusammengesetzt und im Arbeitsspeicher abgelegt werden. Um all dies zu leisten, besitzen Controller meistens kleine eingebettete Computer, die für genau diese Zwecke programmiert wurden.

Der andere Teil des Ein-/Ausgabegeräts ist das eigentliche Gerät selbst. Geräte haben recht einfache Schnittstellen, weil sie einerseits eigentlich nicht viel können und andererseits einem Standard entsprechen sollen. Letzteres ist notwendig, damit z.B. jeder SATA-Controller jede SATA-Festplatte steuern kann. **SATA** steht für **Serial ATA u**nd **ATA** wiederum für **AT Attachment**. Falls Sie sich fragen, wofür AT steht: so nannte IBM die zweite Generation seiner PC-Reihe: „Personal Computer Advanced Technology". Der PC war mit dem für die damalige Zeit extrem leistungsfähigen 6-MHz-80286-Prozessor ausgestattet und wurde im Jahr 1984 eingeführt. Was wir daraus lernen, ist, dass es in der Computerindustrie die Gepflogenheit gibt, existierende Akronyme ständig mit neuen Prä- und Suffixen zu erweitern. Wir lernen außerdem, dass ein Adjektiv wie „advanced" (fortschrittlich) sehr behutsam verwendet werden sollte – will man nicht dreißig Jahre später dumm aussehen.

SATA ist zurzeit der Standardplattentyp auf vielen Computern. Da die eigentliche Geräteschnittstelle durch den Controller verdeckt wird, sieht das Betriebssystem lediglich die Schnittstelle zum Controller, die sich deutlich von der Schnittstelle zum Gerät unterscheiden kann.

Da jeder Controllertyp anders ist, benötigt man auch für jeden Typ eine andere Software zur Steuerung. Die Software, die mit dem Controller kommuniziert, ihm Kommandos gibt und die Antworten empfängt, nennt man **Gerätetreiber** (*device driver*). Jeder Controllerhersteller muss einen Treiber für jeden Controller und für jedes Betriebssystem anbieten, das er unterstützen will. So werden bei einem Scanner z.B. Treiber für OS X, Windows 7, Windows 8 und Linux mitgeliefert.

Um den Treiber zu benutzen, muss er in das Betriebssystem integriert werden, damit er im Kernmodus arbeiten kann. Eigentlich könnte der Gerätetreiber auch außerhalb des Kerns im Benutzerraum arbeiten und Betriebssysteme wie Linux und Windows bieten heutzutage Unterstützung dafür an. Der Großteil der Treiber läuft immer noch unterhalb der Kerngrenzen. Nur bei sehr wenigen aktuellen Betriebssystemen wie beispielsweise MINIX 3 befinden sich alle Treiber im Benutzerraum. Treiber im Benutzerraum müssen kontrolliert auf das Gerät zugreifen dürfen – eine Eigenschaft, die nicht ganz einfach umzusetzen ist.

Ein Treiber kann auf drei Arten in den Kern integriert werden: Bei der ersten Art wird der neue Treiber in den Kern eingebunden und dann wird das System neu gestartet. Viele ältere UNIX-Systeme arbeiten nach diesem Prinzip. Die zweite Möglichkeit ist, in einer Datei des Betriebssystems einen Eintrag zu hinterlegen, der besagt, dass der Treiber benötigt wird, dann wird das System neu gestartet. Das Betriebssystem sucht

beim Hochfahren alle Treiber aus dieser Datei und lädt sie. Windows arbeitet nach diesem Prinzip. Bei der dritten Art wird zur Laufzeit ein neuer Treiber geladen und in das Betriebssystem eingebunden, das System muss dafür aber nicht neu gestartet werden. Diese Methode war bisher eher ungewöhnlich, wird aber heute immer häufiger eingesetzt. Hot-Plug-fähige Geräte (Geräte, die zur Laufzeit angeschlossen werden dürfen) wie USB-Geräte (die weiter unten besprochen werden) und Geräte nach dem IEEE-1394-Standard benötigen grundsätzlich dynamisch ladbare Gerätetreiber.

Jeder Controller hat eine kleine Anzahl an Registern, über die mit ihm kommuniziert werden kann. Zum Beispiel besitzt ein einfacher Festplattencontroller Register, um die Plattenadresse, die Speicheradresse, die Sektornummer und die Richtung (Lesen oder Schreiben) anzugeben. Um den Controller zu aktivieren, bekommt der Treiber ein Kommando vom Betriebssystem, übersetzt dieses in die entsprechenden Werte und schreibt sie dann in die Geräteregister. Alle Geräteregister zusammen bilden den **Namensraum der Ein-/Ausgabeports** (*I/O port space*), ein Thema, auf das wir in *Kapitel 5* noch zurückkommen werden.

Bei einigen Computern sind die Geräteregister in den Adressraum des Betriebssystems eingeblendet, sie können dann wie normaler Speicher gelesen und geschrieben werden. Bei diesen Systemen werden keine speziellen Ein-/Ausgabebefehle benötigt und der Schutz der Hardware gegen die Benutzerprogramme besteht darin, dass man den Speicherbereich der Register einfach nicht in den Adressbereich des Programms legt (z.B. durch die Verwendung von Basis- und Limit-Registern, siehe ▶ *Abschnitt 3.2*). Bei anderen Rechnern werden die Geräteregister auf spezielle Ein-/Ausgabeports gelegt, wobei jedes Register eine eigene Portadresse bekommt. Bei diesen Maschinen sind spezielle `IN`- und `OUT`-Befehle im Kernmodus verfügbar, die es dem Treiber erlauben, die Register zu lesen und zu beschreiben. Die erste Methode benötigt keine zusätzlichen Ein-/Ausgabebefehle, braucht aber etwas Speicherplatz. Die zweite benutzt den Adressraum nicht, benötigt aber ein paar spezielle Befehle. Beide Systeme werden heute viel genutzt.

Eingabe und Ausgabe können auf drei verschiedene Arten erfolgen: Bei der einfachsten Methode löst ein Benutzerprogramm einen Systemaufruf aus, den der Kern dann in einen Prozeduraufruf für den entsprechenden Treiber umsetzt. Der Treiber startet dann die Ein-/Ausgabe und fragt in einer Endlosschleife ständig den Zustand des Geräts ab (meistens gibt es ein eigenes Bit, das anzeigt, ob das Gerät noch beschäftigt ist). Wenn die Ein-/Ausgabe-Operation beendet ist, schreibt der Treiber die Daten (falls vorhanden) an die Stelle, wo sie benötigt werden, und springt zurück. Danach übergibt das Betriebssystem die Kontrolle wieder an das aufrufende Programm. Diese Methode bezeichnet man als **aktives Warten** (*busy waiting*). Sie hat den Nachteil, dass die CPU so lange durch die Abfrage des Geräts belegt ist, bis die Operation abgeschlossen ist.

Bei der zweiten Methode startet der Treiber das Gerät mit der Aufforderung, einen **Interrupt** zu senden, wenn es fertig ist. Sofort danach wird die Kontrolle zurückgegeben. Das Betriebssystem sperrt nötigenfalls das aufrufende Programm so lange und erledigt zwischenzeitlich andere Arbeiten. Sobald der Controller erkennt, dass die Arbeit des Geräts beendet ist, erzeugt er einen Interrupt, um diesen Zustand zu signalisieren.

Interrupts sind in Betriebssystemen sehr wichtig, deshalb wollen wir sie etwas näher betrachten. In ▶*Abbildung 1.11a* sieht man einen dreistufigen Vorgang bei der Ein-/Ausgabe. In Schritt 1 teilt der Treiber dem Controller durch das Beschreiben eines Geräteregisters mit, was er tun soll. Dann startet der Controller das Gerät. Sobald der Controller die erwartete Anzahl Bytes gelesen oder geschrieben hat, signalisiert er dies in Schritt 2 dem Interrupt-Controller über spezielle Busleitungen. Falls der Interrupt-Controller bereit ist, das Signal anzunehmen (was nicht unbedingt der Fall sein muss, weil er z.B. gerade ein Signal mit höherer Priorität bearbeitet), schickt er in Schritt 3 der CPU eine Nachricht über einen speziellen Pin. In Schritt 4 legt der Interrupt-Controller die Nummer des Geräts auf den Bus, damit der Prozessor feststellen kann, welches Gerät gerade mit seiner Arbeit fertig geworden ist (da viele Geräte gleichzeitig arbeiten können).

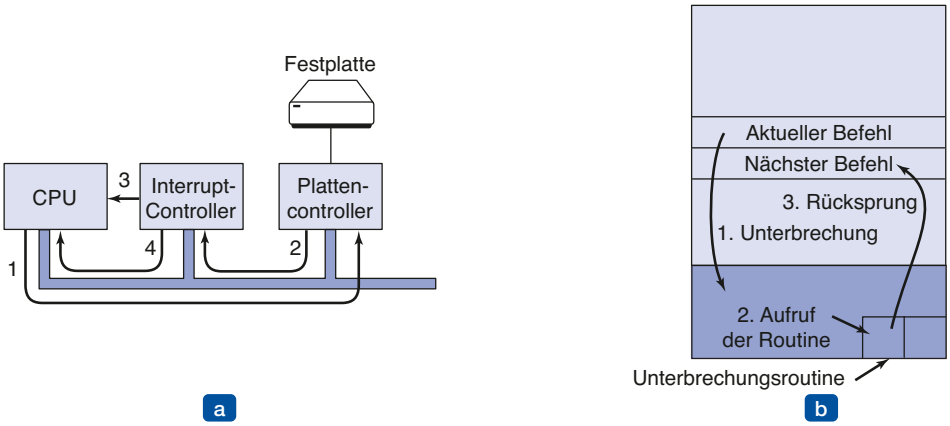

Abbildung 1.11: (a) Die Schritte beim Starten eines Ein-/Ausgabegeräts und Erzeugen einer Unterbrechung. (b) Interruptbehandlung besteht aus dem Abfangen des Interruptsignals, dem Ausführen der Unterbrechungsroutine und dem Rücksprung zum Benutzerprogramm.

Nachdem sich die CPU entschieden hat, den Interrupt zu behandeln, werden der Befehlszähler und das Programmstatuswort auf dem aktuellen Stack gespeichert und die CPU schaltet in den Kernmodus. Die Gerätenummer kann als Index genutzt werden, der auf eine Adresse im Speicher zeigt, wo sich die Unterbrechungsroutine (*interrupt handler*) für das Gerät befindet. Der Teil des Speichers, in dem sich die Liste mit allen Routinen befindet, wird **Unterbrechungsvektor** (*interrupt vector*) genannt. Sobald die Unterbrechungsroutine (dies ist der Teil des Treibers für das entsprechende Gerät) gestartet wurde, entfernt diese den Befehlszähler und das Programmstatuswort vom Stack, speichert sie zwischen und fragt dann das Gerät nach seinem Status ab. Sobald die Routine alles erledigt hat, kehrt sie zu dem zuletzt laufenden Programm zurück und der letzte noch nicht bearbeitete Befehl wird ausgeführt. Diese Schritte werden in ▶*Abbildung 1.11b* gezeigt.

Bei der dritten Methode, die Ein-/Ausgabe durchzuführen, benutzt man einen speziellen **DMA**-Chip (**Direct Memory Access**), der den Datenfluss zwischen dem Speicher und einem Controller regeln kann, ohne dass sich die CPU ständig einschalten muss. Der

Prozessor initialisiert den DMA-Chip und teilt ihm mit, wie viele Bytes zu übertragen sind, um welches Gerät es sich handelt, welches die Startadresse im Speicher ist und in welche Richtung (Lesen oder Schreiben) die Übertragung geht. Sobald der DMA-Chip die Aufgabe beendet hat, erzeugt er einen Interrupt, der wie oben beschrieben behandelt wird. DMA und Ein-/Ausgabehardware allgemein werden in *Kapitel 5* näher untersucht.

Interrupts treten oftmals in sehr unpassenden Momenten auf, beispielsweise während eine andere Unterbrechungsroutine läuft. Deshalb kann die CPU Interrupts blockieren und später wieder aktivieren. Solange die Interrupts abgeschaltet sind, sendet jedes Gerät, das eine Aufgabe beendet hat, sein Interruptsignal fortwährend. Diese Signale kann die CPU aber erst wieder erkennen, wenn Interrupts erneut aktiviert werden. Wenn mehrere Geräte ihre Aufgabe in dem Zeitraum beenden, in dem Interrupts noch deaktiviert sind, entscheidet der Interrupt-Controller, welcher Interrupt zuerst durchgelassen wird. Diese Entscheidung basiert meistens auf statisch vergebenen Prioritäten, die jedem Gerät zugewiesen sind. Das Gerät mit der höchsten Priorität macht das Rennen und wird zuerst behandelt. Die anderen müssen warten.

1.3.5 Bussysteme

Der in ▶ *Abbildung 1.6* gezeigte Aufbau wurde jahrelang auf Minicomputern und auch bei den PCs von IBM verwendet. Da die Prozessoren und Speicherbausteine jedoch immer schneller wurden, konnte ein einziger Bus (vor allem der des IBM-PC) nicht mehr den gesamten Verkehr beherrschen. Es musste etwas geschehen. Folglich wurden zusätzliche Busse integriert, sowohl für schnellere Ein-/Ausgabegeräte als auch für höheren Datendurchsatz zwischen CPU und dem Arbeitsspeicher. Damit sieht ein großes x86-System momentan so aus wie in ▶ *Abbildung 1.12*.

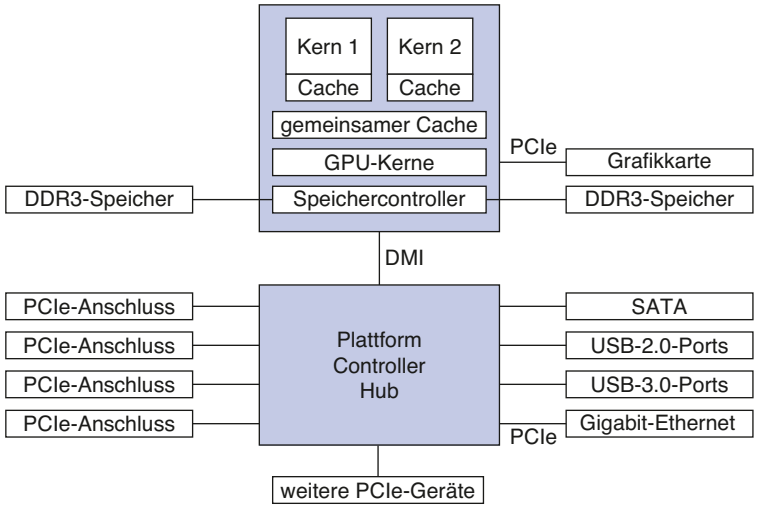

Abbildung 1.12: Der Aufbau eines ausgebauten x86-Systems.

Dieses System hat viele Busse (z.B. je einen für Cache, Speicher, PCIe, PCI, USB, SATA und DMI), wobei jedes dieser Bussysteme unterschiedliche Übertragungsraten und Funktionen besitzt. Das Betriebssystem muss zum Konfigurieren und Verwalten alles über diese Bussysteme wissen. Das wichtigste Bussystem ist der **PCIe**-Bus (**Peripheral Component Interconnect Express**).

Der PCIe-Bus wurde von Intel entwickelt und ist der Nachfolger des älteren **PCI**-Busses, der seinerseits den originalen **ISA**-Bus (**Industry Standard Architecture**) ersetzt hat. PCIe kann mehrere Gigabits pro Sekunde übertragen und ist damit viel schneller als seine Vorgänger. Außerdem ist seine Struktur eine ganz andere. Bevor der PCIe im Jahr 2004 herauskam, waren die meisten Busse parallel und wurden gemeinsam genutzt. Eine **gemeinsam genutzte Busarchitektur** bedeutet, dass mehrere Geräte dieselbe Leitung zur Datenübertragung verwenden. Wenn also mehrere Geräte Daten senden möchten, benötigt man einen Arbiter, um festzulegen, wer den Bus benutzen darf. Im Gegensatz dazu verwendet PCIe dedizierte Punkt-zu-Punkt-Verbindungen. Eine **parallele Busarchitektur**, wie sie beim traditionellen PCI eingesetzt wird, bedeutet, dass man jedes Datenwort über mehrere Leitungen sendet. Beispielsweise wird in normalen PCI-Bussen eine einzelne 32-Bit-Zahl über 32 parallele Leitungen geschickt. PCIe verwendet dagegen eine **serielle Busarchitektur** und sendet alle Bits in einer Nachricht durch eine einzelne Verbindung, die als Lane bezeichnet wird, ganz ähnlich wie bei Netzpaketen. Dies ist viel einfacher, weil man nicht sicherstellen muss, dass alle 32 Bits am Ziel zur exakt gleichen Zeit ankommen. Parallelismus wird dennoch verwendet, wenn man mehrere parallele Lanes hat. Zum Beispiel könnten wir 32 Lanes verwenden, um 32 Nachrichten parallel zu übertragen. Aufgrund der rasanten Steigerung der Geschwindigkeit von Peripheriegeräten wie Netzkarten und Grafikadaptern wird der PCIe-Standard alle 3–5 Jahre aktualisiert. PCIe 2.0 bietet beispielsweise 16 Lanes mit 64 Gigabit pro Sekunde an. Mit dem Upgrade auf PCIe 3.0 erreicht man schon das Doppelte an Geschwindigkeit und PCIe 4.0 wird diesen Wert noch einmal verdoppeln.

Doch einstweilen haben wir noch viele Altgeräte für den alten PCI-Standard. Wie man in ▶*Abbildung 1.12* sieht, sind diese Geräte an einen separaten Hub-Prozessor angeschlossen. Später, wenn wir PCI nicht mehr nur als *alt*, sondern als *uralt* ansehen, ist es möglich, dass alle PCI-Geräte an einen weiteren Hub angeschlossen sind, der wiederum mit dem Haupthub verbunden ist, wodurch ein Baum von Bussen erzeugt wird.

In der Konfiguration von ▶*Abbildung 1.12* kommuniziert die CPU mit dem Speicher über einen schnellen DDR3-Bus, mit einem externen Grafikgerät über PCIe und mit allen anderen Geräten mittels eines Hubs über einen **DMI**-Bus (**Direct Media Interface**). Der Hub wiederum verbindet all die anderen Geräte, dazu verwendet er den Universal Serial Bus zum Ansprechen der USB-Geräte, den SATA-Bus zur Interaktion mit den Festplatten und DVD-Laufwerken sowie PCIe zur Übertragung von Ethernet-Rahmen. Daneben gibt es, wie schon erwähnt, noch die älteren PCI-Geräte, die den traditionellen PCI-Bus benutzen.

Darüber hinaus hat jeder der Kerne einen eigenen Cache und es gibt einen größeren, gemeinsam genutzten Cache. Jeder dieser Caches bringt einen weiteren Bus mit sich.

Der **Universal Serial Bus** (**USB**) wurde entwickelt, um langsamere Geräte wie Tastatur oder Maus an den Rechner anzuschließen. Ein modernes USB-3.0-Gerät, das mit 5 Gbit/s vor sich hin summt, „langsam" zu nennen, mag nicht selbstverständlich für die Generation sein, die mit 8-Mbit/s-ISA als Hauptbus im ersten IBM-PC aufgewachsen ist. USB verwendet einen kleinen Stecker mit vier bis elf Anschlüssen (je nach Version), wovon einige für die Stromversorgung des USB-Geräts oder für die Erdung zuständig sind. USB ist ein zentralisiertes Bussystem, in dem es ein Hauptgerät (*root device*) gibt, das jede Millisekunde jedes Gerät am Bus abfragt, ob Nachrichten gesendet werden. USB 1.0 konnte insgesamt 12 Mbit/s pro Sekunde übertragen, USB 2.0 erhöhte die Geschwindigkeit auf 480 Mbit/s und USB 3.0 erreicht 5 Gbit/s. Jedes USB-Gerät, das an einen Rechner angeschlossen wird, kann sofort angesprochen werden, ohne dass ein Neustart erforderlich ist – was früher üblich war, sehr zum Ärger einer Generation von genervten Anwendern.

Der **SCSI**-Bus (**Small Computer System Interface**) ist als ein sehr schneller Bus dafür ausgelegt, Platten, Scanner und andere Geräte, die eine hohe Bandbreite benötigen, einfacher anzuschließen. Heutzutage finden wir sie hauptsächlich in Servern und Workstations. Sie können mit bis zu 640 MB pro Sekunde arbeiten.

Damit das Zusammenspiel der Komponenten bei einer Konfiguration wie in ▶ *Abbildung 1.12* klappt, muss das Betriebssystem wissen, welche Peripheriegeräte mit dem Computer verbunden sind, und diese außerdem konfigurieren. Dafür haben Intel und Microsoft ein System entwickelt, das sich **Plug-and-Play** nennt und auf einem ähnlichen Konzept basiert wie eine schon früher auf dem Apple Macintosh implementierte Eigenschaft. Bevor Plug-and-Play existierte, hatte jede Ein-/Ausgabekomponente eine fest zugeordnete Interruptnummer und eine feste Adresse für die Ein-/Ausgaberegister. Beispielsweise bekam die Tastatur Interrupt 1 und die Adressen 0×60 bis 0×64 zugewiesen, der Diskettencontroller erhielt Interrupt 6 und die Adressen 0×3F0 bis 0×3F7, der Drucker Interrupt 7 und den Adressbereich von 0×378 bis 0×37A und so weiter.

So weit, so gut. Die Probleme fingen an, wenn ein Nutzer eine Soundkarte und eine Modemkarte kaufte und beide Karten beispielsweise Interrupt 4 verwenden wollten. Es entstand ein Konflikt – die Karten konnten nicht gemeinsam funktionieren. Also hat man kleine Schalter, DIP-Switches, oder Stecker, sogenannte Jumper, auf allen Ein-/Ausgabekarten integriert. Die Benutzer wurden angewiesen, doch bitte die Interruptnummern und Ein-/Ausgabeadressen so einzustellen, dass diese von keiner anderen Karte benutzt wurden. Teenager, die ihr Leben der Komplexität von Hardware widmeten, haben dies ab und zu fehlerfrei geschafft. Unglücklicherweise konnte das sonst niemand, was zu einem Chaos führte.

Ein Plug-and-Play-System sammelt daher jede Information über die Einstellungen der Ein-/Ausgabegeräte automatisch, vergibt zentral die Interruptnummern und die Adressen für Ein-/Ausgabe und teilt dann jeder Karte mit, wie die Einstellungen sein müssen. Dieser Vorgang hängt eng mit dem Hochfahren des Computers zusammen, was wir uns als Nächstes ansehen werden. Es ist komplizierter, als man denkt.

1.3.6 Hochfahren des Computers

Kurz gefasst läuft der Prozess des Bootens wie folgt ab: Jeder PC enthält eine Hauptplatine (*motherboard*)[7], auf der ein Programm ist, das man **BIOS** (**Basic Input Output System**) nennt. Dieses BIOS enthält die grundlegendste Ein-/Ausgabesoftware, unter anderem Prozeduren zum Lesen der Tastatur, für die Ausgabe auf dem Bildschirm und für die Kommunikation mit den Festplattenlaufwerken. Heutzutage befindet sich das BIOS in einem Flash-RAM, das nicht flüchtig ist, aber vom Betriebssystem beschrieben werden kann, wenn Fehler darin gefunden werden.

Beim Einschalten des Computers wird das Programm im BIOS gestartet. Zunächst überprüft es die Größe des Arbeitsspeichers und ob die Tastatur und andere wichtige Geräte installiert sind und korrekt antworten. Danach werden die PCIe- und PCI-Bussysteme nach den angeschlossenen Geräten durchsucht. Wenn dabei Geräte auftauchen, die beim letzten Startvorgang noch nicht vorhanden waren, so werden diese konfiguriert.

Das BIOS bestimmt dann das Gerät, von dem das Betriebssystem geladen werden soll, indem es eine Liste mit Geräten durchprobiert, die im nicht flüchtigen CMOS-Speicher enthalten ist. Der Benutzer kann diese Einstellung ändern, indem er direkt nach dem Einschalten das Konfigurationsprogramm des BIOS startet. Normalerweise wird zuerst versucht, von einem CD-ROM-Laufwerk (oder manchmal USB-Laufwerk) zu laden, falls es eines gibt. Wenn das nicht funktioniert, wird das System von der Festplatte geladen. Der erste Sektor des Bootgeräts wird in den Speicher kopiert und ausgeführt. Normalerweise steht im ersten Sektor ein kleines Programm, das die Partitionstabelle am Ende des Bootsektors liest und die aktive Partition der Platte ermittelt. Danach wird ein weiteres Programm, der Bootlader, von dieser Partition gelesen und ausgeführt. Der Bootlader liest dann das Betriebssystem von der aktiven Partition ein und startet es.

Das Betriebssystem fragt das BIOS nach den Konfigurationseinstellungen. Für jedes gefundene Gerät sucht das System nach einem Gerätetreiber. Wenn keiner gefunden werden kann, wird der Benutzer gebeten, eine CD-ROM mit dem entsprechenden Treiber einzulegen (die in der Regel vom Hersteller mitgeliefert wird) oder den Treiber aus dem Internet zu laden. Nachdem alle Gerätetreiber gefunden wurden, lädt das Betriebssystem sie in den Kern. Dann werden interne Tabellen initialisiert, die nötigen Hintergrundprozesse eingerichtet und das Login-Programm oder die grafische Benutzungsoberfläche auf den angeschlossenen Terminals gestartet.

7 Anm. d. Übers.: Im englischen Original steht hier „parentboard“ und ein Hinweis darauf, dass es aus Gründen der politischen Korrektheit eigentlich Elternplatine heißen muss.

1.4 Die Betriebssystemfamilie

Betriebssysteme gibt es nun seit mehr als einem halben Jahrhundert. Während dieser Zeit wurden recht unterschiedliche Systeme entwickelt, einige sind allerdings eher unbekannt geblieben. In diesem Abschnitt wollen wir auf neun Betriebssystemarten kurz eingehen. In späteren Kapiteln des Buchs kommen wir auf einige dieser Systeme noch einmal zurück.

1.4.1 Betriebssysteme für Großrechner

Am oberen Ende der Skala stehen die Betriebssysteme für Großrechner. Diese raumgroßen Geräte finden sich heute noch in großen Rechenzentren von Unternehmen. Sie unterscheiden sich von PCs vor allem durch ihre sehr hohe Ein-/Ausgabeleistung. Ein Großrechner mit 1000 Festplatten und Millionen von Gigabyte an Daten ist nicht ungewöhnlich – um einen PC mit dieser Ausstattung würden einen die Freunde beneiden. Großrechner erleben gerade so etwas wie ein Comeback als hoch entwickelte Webserver, als Server für den E-Commerce oder als Server für Business-to-Business-Anwendungen.

Betriebssysteme für Großrechner sind stark darauf ausgelegt, viele Prozesse gleichzeitig auszuführen, von denen die meisten einen ungeheuren Bedarf an schneller Ein-/Ausgabe haben. Typischerweise bieten sie drei Arten der Prozessverwaltung an: Stapelverarbeitung, Dialogverarbeitung und Timesharing. Ein Stapelverarbeitungssystem verarbeitet Routineaufgaben, ohne dass ein interaktiver Benutzer anwesend sein muss. Schadensmeldungen einer Versicherung oder Verkaufsberichte einer Kaufhauskette werden normalerweise in Stapelverarbeitung durchgeführt. Systeme mit Dialogverarbeitung können eine große Anzahl kleinerer Aufgaben durchführen, zum Beispiel die Verarbeitung von Überweisungen einer Bank oder Flugbuchungen. Die einzelne Aufgabe dabei ist klein, aber es sind Hunderte oder sogar Tausende davon in der Sekunde zu erledigen. Timesharing-Systeme erlauben es Benutzern, viele Aufgaben praktisch gleichzeitig durchzuführen, wie etwa die Anfragen an eine große Datenbank. All diese Aufgaben hängen eng zusammen – Großrechner-Betriebssysteme führen meistens alle durch. Ein Beispiel für ein Großrechner-Betriebssystem ist OS/390, ein Nachfolger von OS/360. Dennoch werden diese Betriebssysteme nach und nach von UNIX-Varianten wie Linux verdrängt.

1.4.2 Betriebssysteme für Server

Eine Ebene tiefer angesiedelt sind die Server-Betriebssysteme. Sie laufen auf Servern, was entweder sehr große PCs, Workstations oder sogar Großrechner sein können. Sie bedienen gleichzeitig viele Benutzer über ein Netzwerk und verteilen Hardware- und Softwareressourcen an die Anwender. Diese Server können beispielsweise Druck-, Datei- oder Webdienste anbieten. Internetanbieter (Provider) haben viele Server, um ihre Kunden zu bedienen, und Websites benutzen Server, um ihre Webseiten zu speichern und eingehende Anfragen zu bearbeiten. Typische Server-Betriebssysteme sind Solaris, FreeBSD, Linux und Windows Server 201x.

1.4.3 Betriebssysteme für Multiprozessorsysteme

Um Rechenleistung der Extraklasse zu erhalten, werden zunehmend mehrere Prozessoren zu einem einzigen System zusammengeschaltet. Je nachdem, wie diese genau miteinander verschaltet sind, nennt man die Systeme Parallelcomputer, Multicomputer oder Multiprozessorsysteme. Es sind spezielle Betriebssysteme notwendig, oft sind dies allerdings abgeänderte Server-Betriebssysteme, die spezielle Eigenschaften für Kommunikation, Anschlussfähigkeit und Konsistenz mitbringen.

Mit dem Aufkommen von Mehrkernchips für PCs in jüngster Zeit stehen auch die konventionellen Betriebssysteme für Einzelrechner vor der Aufgabe, zumindest im kleinen Rahmen mit Multiprozessorsystemen umzugehen, und die Anzahl der Kerne wird langfristig wahrscheinlich noch wachsen. Glücklicherweise ist aus der Forschung der letzten Jahre schon ein wenig über Multiprozessor-Betriebssysteme bekannt, sodass es nicht so schwer sein sollte, dieses Wissen bei Mehrkernsystemen einzusetzen. Der schwierigste Teil dabei wird sein, die Anwendungsprogramme in die Lage zu versetzen, all diese Rechenleistung auch tatsächlich auszunutzen. Viele bekannte Betriebssysteme, einschließlich Windows und Linux, können auf Multiprozessorsystemen eingesetzt werden.

1.4.4 Betriebssysteme für PCs

Die nächste Kategorie sind die Betriebssysteme für PCs. Alle modernen PC-Betriebssysteme unterstützen heute Multiprogrammierung – oft werden direkt beim Hochfahren des Computers Dutzende von Programmen gestartet. Die Aufgabe dieser Betriebssysteme ist es, einen einzelnen Anwender gut zu unterstützen. Sie werden häufig für Textverarbeitung, Tabellenkalkulation, Spiele und Zugriff auf das Internet genutzt. Bekannte Beispiele sind Linux, FreeBSD, Windows 7, Windows 8 und OS X von Apple. Betriebssysteme für PCs sind so bekannt, dass eine Einführung kaum nötig ist. Tatsächlich wissen manche Leute überhaupt nicht, dass es noch andere Systeme gibt.

1.4.5 Betriebssysteme für Handheld-Computer

Geht man weiter in Richtung der kleineren Systeme, kommt man zu Tablets, Smartphones und anderen Handheld-Computern. Ein Handheld-Computer, anfangs als **PDA** (**Personal Digital Assistant**) bezeichnet, ist ein kleiner Computer, der während des Betriebs in der Hand gehalten werden kann. Smartphones und Tablets sind hierfür die bekanntesten Beispiele. Wie wir bereits gesehen haben, wird dieser Markt aktuell dominiert von Android (Google) und iOS (Apple), doch es gibt viele Konkurrenten. Die meisten dieser Geräte besitzen Mehrkern-CPUs, Kameras und andere Sensoren, große Mengen an Speicher und hochentwickelte Betriebssysteme. Darüber hinaus laufen auf allen unzählige Anwendungen von Drittanbietern (sogenannte **Apps**).

1.4.6 Betriebssysteme für eingebettete Systeme

Eingebettete Systeme sind Rechensysteme, die andere Geräte steuern, die man in der Regel überhaupt nicht als Computer wahrnimmt, und auf denen Benutzer keine eigene Software installieren dürfen. Typische Beispiele sind Mikrowellen, Fernsehgeräte, Autos, DVD-Recorder, traditionelle Telefone und MP3-Player. Die Haupteigenschaft, die eingebettete Systeme von Handheld-Computern unterscheidet, ist die Sicherheit, dass nur vertrauenswürdige Software auf ihnen ausgeführt wird. Man kann nicht einfach neue Anwendungsprogramme für seine Mikrowelle nachladen – die gesamte Software steht im ROM. Das heißt, dass man keinen Schutzmechanismus zwischen Anwendungen braucht, was einige Vereinfachungen im Entwurf mit sich bringt. Bekannte Systeme in diesem Bereich sind Embedded Linux, QNX und VxWorks.

1.4.7 Betriebssysteme für Sensorknoten

Netzwerke von winzigen Sensorknoten werden für unzählige Zwecke eingesetzt. Diese Knoten sind winzige Computer, die miteinander und mit einer Basisstation über Funk kommunizieren. Sensornetze werden sehr vielfältig eingesetzt, z. B. zum Gebäudeschutz, bei der Überwachung von Landesgrenzen, zur Entdeckung von Waldbränden, zur Temperatur- und Niederschlagsmessung für Wettervorhersagen oder zum Sammeln von Informationen über Feindesbewegungen auf Schlachtfeldern.

Die Sensoren sind kleine batteriebetriebene Computer mit eingebauten Funkschnittstellen. Sie haben nur eine begrenzte Leistung und müssen über einen längeren Zeitraum wartungsfrei funktionieren, und zwar in der Regel im Freien und oft unter rauen Umweltbedingungen. Das Netzwerk muss robust genug sein, um Ausfälle von einzelnen Knoten zu verkraften, was zunehmend häufiger vorkommt, wenn die Batterien langsam verbraucht sind.

Jeder Sensorknoten ist ein echter Computer, der mit einer CPU, RAM, ROM und einem oder mehreren Umweltsensoren ausgestattet ist. Auf jedem Knoten läuft ein kleines, aber echtes Betriebssystem, das in der Regel ereignisorientiert ist und auf externe Ereignisse antwortet oder periodische Messungen durchführt, die von einer internen Uhr gesteuert werden. Das Betriebssystem muss klein und einfach sein, weil die Knoten wenig RAM haben und die begrenzte Lebenszeit der Batterien ein wesentliches Problem darstellt. Wie bei den eingebetteten Systemen werden auch hier alle Programme fest vorinstalliert – es gibt keine Benutzer, die plötzlich Programme starten, die sie aus dem Internet heruntergeladen haben, was den Entwurf natürlich viel einfacher macht. TinyOS ist ein sehr bekanntes Betriebssystem für Sensorknoten.

1.4.8 Echtzeitbetriebssysteme

Eine weitere Betriebssystemklasse bilden die Echtzeitsysteme. Sie zeichnen sich dadurch aus, dass die Zeit ein sehr wichtiger Parameter bei Ressourcenvergaben ist. Zum Beispiel müssen in Steuerungssystemen bei der industriellen Fertigung Echtzeit-

computer Daten über den Produktionsprozess sammeln und aufgrund dieser Daten die Maschinen der Anlage steuern. Oft gibt es dabei strenge Deadlines, die unbedingt eingehalten werden müssen. Wenn beispielsweise während der Fertigung ein Auto das Montageband herunterläuft, müssen bestimmte Aktionen in einem ganz bestimmten Moment erfolgen: Ein Schweißroboter, der zu früh oder zu spät schweißt, kann das ganze Auto zerstören. Wenn eine Aktion unter allen Umständen zu einem bestimmten Zeitpunkt stattfinden *muss* (oder zumindest innerhalb eines Zeitintervalls), dann spricht man von einem **harten Echtzeitsystem** (*hard real-time system*). Man findet sie häufig in der industriellen Fertigungssteuerung, in der Avionik, beim Militär und ähnlichen Anwendungsgebieten. Diese Systeme müssen eine absolute Garantie bieten, dass eine bestimmte Aktion zu einer bestimmten Zeit ausgeführt wird.

Bei einem **weichen Echtzeitsystem** (*soft real-time system*) ist eine verpasste Deadline zwar nicht erwünscht, kann aber doch toleriert werden, ohne dass dadurch irgendein permanenter Schaden verursacht wird. Digitale Audio- oder Multimediasysteme fallen in diese Kategorie. Smartphones können ebenfalls weiche Echtzeitsysteme benutzen.

Da in (harten) Echtzeitsystemen das Einhalten von Deadlines entscheidend ist, ist das Betriebssystem manchmal einfach eine Bibliothek, die an die Anwendungsprogramme angebunden ist, wobei alles eng miteinander verflochten ist und es keinen Schutz zwischen den einzelnen Systemteilen gibt. Ein Beispiel für diesen Typ von Echtzeitsystemen ist das Produkt eCos.

Bei den Kategorien von Handheld-Computern, eingebetteten Systemen und Echtzeitsystemen gibt es beträchtliche Überlappungen. Fast alle haben mindestens einige weiche Echtzeitaspekte. Auf den eingebetteten Systemen und den Echtzeitsystemen läuft nur Software, die von den Systementwicklern eingegeben wurde – Benutzer können keine eigene Software hinzufügen, was den Schutz einfacher macht. Handheld-Computer und eingebettete Systeme sind für Endverbraucher vorgesehen, während Echtzeitsysteme eher für die industrielle Nutzung bestimmt sind. Dennoch haben sie einiges gemeinsam.

1.4.9 Betriebssysteme für Smartcards

Die kleinsten Betriebssysteme laufen auf Smartcards, die die Größe einer Kreditkarte haben und einen eigenen Prozessor besitzen. Sie sind bezüglich Rechenleistung und Speicherplatz sehr stark eingeschränkt. Einige Smartcards werden durch den Kontakt mit dem Lesegerät, in das sie eingeführt werden, mit Energie versorgt. Doch es gibt auch kontaktlose Smartcards, die mittels Induktion betrieben werden, was ihre Verwendbarkeit außerordentlich begrenzt. Manche beherrschen nur eine einzige Funktion wie das elektronische Bezahlen, andere können dagegen viele Aufgaben erledigen. Dies sind häufig proprietäre Systeme.

Einige Smartcards sind Java-orientiert. Das bedeutet, dass das ROM auf der Karte einen Interpreter für den Java-Bytecode besitzt. Java-Applets (kleine Programme) wer-

den auf die Karte geladen und von der Java Virtual Machine (JVM) interpretiert. Einige dieser Karten können viele Java-Applets gleichzeitig bedienen, was zur Multiprogrammierung führt und das Aufteilen der Rechenzeit notwendig macht. Ressourcenverwaltung und Schutzmechanismen werden auch wichtig, sobald zwei oder mehr Applets gleichzeitig im Speicher vorhanden sind. Mit diesen Problemen muss das (normalerweise äußerst primitive) Betriebssystem auf der Karte umgehen können.

1.5 Betriebssystemkonzepte

Die meisten Betriebssysteme stellen bestimmte grundlegende Konzepte und Abstraktionen wie Prozesse, Adressräume und Dateien bereit, die wesentlich sind, um die Arbeitsweise eines Betriebssystems zu verstehen. In den folgenden Abschnitten werfen wir als Einführung einen ganz kurzen Blick auf einige dieser Basiskonzepte. Später werden wir uns jeden einzelnen Punkt ausführlich ansehen. Um die Konzepte zu veranschaulichen, benutzen wir immer wieder Beispiele, die im Allgemeinen von UNIX stammen. Ähnliche Beispiele gibt es natürlich auch in anderen Betriebssystemen, von denen wir einige später ausführlich untersuchen werden.

1.5.1 Prozesse

Das Schlüsselkonzept ist in allen Betriebssystemen der **Prozess**. Im Prinzip ist ein Prozess ein Programm in Ausführung. Jedem Prozess wird ein **Adressraum** (*address space*) zugeordnet. Dieser besteht aus einer Liste von Speicherstellen von 0 bis zu einem maximalen Wert, in denen der Prozess lesen und schreiben darf. Der Adressraum beinhaltet das ausführbare Programm, die Programmdaten und den Stack. Außerdem ist jedem Prozess eine Ressourcenmenge zugeordnet. Diese enthält im Allgemeinen die Register (einschließlich Befehlszähler und Stackpointer), eine Liste von geöffneten Dateien, offene Fehlersignale, eine Liste von verbundenen Prozessen sowie alle weiteren Informationen, die zur Ausführung des Programms benötigt werden. Ein Prozess ist im Grunde ein Behälter, in dem alle Informationen aufbewahrt werden, die zur Ausführung eines Programms benötigt werden.

Auf das Prozesskonzept werden wir noch detaillierter in *Kapitel 2* eingehen. Vorläufig reicht es für das intuitive Verständnis von Prozessen, an ein System mit Multiprogrammierung zu denken. Stellen Sie sich beispielsweise folgendes Szenarium vor: Ein Benutzer startet ein Programm zur Videobearbeitung und lässt ein einstündiges Video in ein bestimmtes Format konvertieren (ein Vorgang, der Stunden dauern kann), verlässt dann das Programm und surft im Internet. In der Zwischenzeit ist ein Hintergrundprozess angesprungen, der regelmäßig aufwacht, um nach ankommenden E-Mails zu suchen. Damit gibt es (mindestens) drei aktive Prozesse: die Videobearbeitung, den Webbrowser und das E-Mail-Programm. Das Betriebssystem entscheidet in bestimmten Zeitabständen, einen laufenden Prozess zu stoppen und einen anderen Prozess zu starten – vielleicht weil der erste Prozess in den letzten ein bis zwei Sekunden seinen Anteil an CPU-Zeit verbraucht hat.

Wenn ein Prozess wie in diesem Fall kurzzeitig angehalten wird, so muss er später in genau demselben Zustand, in dem er gestoppt wurde, wieder gestartet werden. Das bedeutet, dass alle Informationen über den Prozess irgendwo für die gesamte Dauer der Ausführungspause gesichert werden müssen. Ein Prozess kann beispielsweise mehrere Dateien gleichzeitig zum Lesen geöffnet haben. Jede Datei hat einen Zeiger, der die aktuelle Position angibt (d.h. die Anzahl der Bytes oder Sätze, die als Nächstes gelesen werden sollen). Wenn ein Prozess zeitweilig angehalten wurde, müssen all diese Zeiger gesichert werden, sodass ein `read`-Aufruf, der nach der Wiederaufnahme des Prozesses ausgeführt wird, die richtigen Daten liest. In vielen Betriebssystemen werden alle Informationen der unterbrochenen Prozesse außer den Inhalten des jeweiligen Adressraums in einer Tabelle des Betriebssystems gespeichert, der **Prozesstabelle** (*process table*). Sie wird als Array realisiert, wobei für jeden Prozess eine eigene Tabelle existiert.

Ein (angehaltener) Prozess besteht also aus seinem Adressraum, der häufig als das **Speicherabbild** oder **Kernspeicherabbild** (*core image*) bezeichnet wird (gilt als Reminiszenz an den magnetischen Kernspeicher, der in alten Zeiten eingesetzt wurde), und seinem Prozesstabelleneintrag, der seine Registerbelegung und viele weitere Elemente enthält, die zur späteren Wiederaufnahme des Prozesses benötigt werden.

Die wichtigsten Systemaufrufe der Prozessverwaltung befassen sich mit der Erzeugung und der Beendigung von Prozessen. Sehen wir uns einen typischen Fall an: Ein Prozess, der als **Kommandozeileninterpreter** oder auch **Shell** bezeichnet wird, liest Kommandos von einem Terminal. Ein Benutzer hat gerade ein Kommando eingetippt, mit dem er die Übersetzung eines Programms starten will. Die Shell muss nun einen neuen Prozess erzeugen, der den Compiler ausführt. Wenn dieser Prozess mit der Übersetzung fertig ist, beendet er sich, indem er einen Systemaufruf ausführt.

Falls ein Prozess einen oder mehrere andere Prozesse (sogenannte **Kindprozesse**) erzeugt und diese Prozesse ihrerseits wieder Kindprozesse erzeugen, dann entsteht schnell ein Prozessbaum wie in ▶*Abbildung 1.13*. Prozesse, die in einer solchen Beziehung zueinander stehen und zusammen einen Auftrag ausführen, müssen oft miteinander kommunizieren und ihre Aktionen gegenseitig synchronisieren. Diese Art der Kommunikation nennt man **Interprozesskommunikation** (*interprocess communication*), sie wird in *Kapitel 2* genauer besprochen.

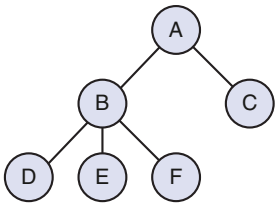

Abbildung 1.13: Ein Prozessbaum. *A* hat zwei Kindprozesse *B* und *C* erzeugt. Prozess *B* hat wiederum die Prozesse *D*, *E* und *F* erzeugt.

Für Prozesse stehen weitere Systemaufrufe zur Verfügung, etwa um mehr Speicher anzufordern (oder nicht benutzten Speicher freizugeben), um auf die Beendigung eines Kindprozesses zu warten und um sein Programm durch ein anderes zu ersetzen.

Gelegentlich müssen Informationen an einen laufenden Prozess übermittelt werden, der allerdings nicht ständig auf diese Information wartet. So kann zum Beispiel ein Prozess, der mit einem zweiten Prozess auf einem anderen Computer kommunizieren will, Nachrichten über ein Netzwerk verschicken. Es könnte jedoch passieren, dass eine Nachricht oder die Antwort darauf auf diesem Weg verloren geht. Deshalb fordert der Sender sein Betriebssystem auf, ihn nach einer vorgegebenen Anzahl von Sekunden zu benachrichtigen, sodass er die Nachricht noch einmal schicken kann, falls er nach Ablauf dieser Zeit keine Bestätigung bekommen hat. Nachdem das Programm den entsprechenden Timer gestellt hat, kann es mit anderen Arbeiten fortfahren.

Wenn die festgelegten Sekunden vergangen sind, sendet das Betriebssystem ein **Signal** an den Prozess. Dieses Signal bewirkt, dass der Prozess zeitweilig angehalten wird, unabhängig davon, welche Aufgabe der Prozess gerade ausführt. Seine Registerinhalte werden auf dem Stack gespeichert und es wird eine spezielle Behandlungsroutine für das Signal gestartet, um zum Beispiel eine vermutlich verloren gegangene Nachricht erneut zu übertragen. Ist die Behandlungsroutine abgeschlossen, so wird der Prozess genau in den Zustand gebracht, in dem er sich direkt vor dem Eintreffen des Signals befand. Signale auf der Softwareebene entsprechen den Interrupts auf der Hardwareebene. Neben dem Ablaufen eines Timers können Signale aus einer Vielzahl von Gründen generiert werden. Viele Unterbrechungen, die von der Hardware erkannt werden, wie die Ausführung eines illegalen Befehls oder die Benutzung einer illegalen Adresse, werden ebenfalls in Signale für den betreffenden Prozess umgesetzt.

Jede Person, die autorisiert ist, ein System zu benutzen, bekommt vom Systemadministrator eine **Benutzer-ID** (**UID**, *User IDentification*) zugewiesen. Alle Prozesse tragen die UID der Person, die sie gestartet hat. Ein Kindprozess erbt die UID von seinem Elternprozess. Benutzer können Mitglieder von Benutzergruppen sein, wobei jede Gruppe wiederum eine **Gruppen-ID** (**GID**, *Group IDentification*) besitzt.

Eine Benutzer-ID, die der **Superuser** (unter UNIX) oder **Administrator** (unter Windows) genannt wird, besitzt besondere Rechte und kann viele Schutzmechanismen umgehen. Bei großen Anlagen kennt nur der Systemadministrator das Passwort, um Superuser zu werden. Allerdings unternehmen auch viele normale Benutzer (vor allem Studenten) größte Anstrengungen, um Lücken im System zu finden und selbst Superuser zu werden, ohne das Passwort zu kennen.

Wir werden uns mit Prozessen, Interprozesskommunikation in *Kapitel 2* noch genauer beschäftigen.

1.5.2 Adressräume

Jeder Computer besitzt Arbeitsspeicher, um die auszuführenden Programme zu speichern. In einem sehr einfachen Betriebssystem befindet sich jeweils nur ein Programm im Speicher. Damit ein zweites Programm ausgeführt werden kann, muss das erste aus dem Speicher verdrängt und das zweite geladen werden.

Etwas anspruchsvollere Betriebssysteme erlauben es, mehrere Programme gleichzeitig im Speicher zu halten. Damit sie sich nicht gegenseitig (und auch nicht das Betriebssystem) behindern, werden Schutzmechanismen benötigt. Diese Mechanismen sind zwar in der Hardware realisiert, werden aber vom Betriebssystem verwaltet.

Der oben genannte Punkt betrifft die Verwaltung und den Schutz des Arbeitsspeichers. Eine andere, aber ebenso wichtige speicherbezogene Aufgabe ist die Verwaltung des Adressraums der Prozesse. Normalerweise hat jeder Prozess einen eigenen Adressbereich, in der Regel von 0 bis zu einer gewissen Obergrenze, die er nutzen darf. Im einfachsten Fall ist dieser Bereich kleiner als die Gesamtgröße des Arbeitsspeichers. Dann ist genügend Platz für den Prozess im Speicher vorhanden.

In vielen modernen Rechnern ist der Adressraum allerdings 32 oder 64 Bit groß, womit ein Adressraum von 2^{32} bzw. 2^{64} Byte gegeben ist. Was passiert nun, wenn ein Prozess einen größeren Adressraum hat, als der Rechner Arbeitsspeicher zur Verfügung hat, und der Prozess den Speicher auch komplett nutzen will? Bei den ersten Computern hatte der Prozess einfach Pech. Heute existiert, wie bereits erwähnt, eine Technik, die man virtuellen Speicher nennt. Dabei lädt das Betriebssystem einen Teil des Adressraums in den Arbeitsspeicher, ein anderer Teil bleibt auf der Festplatte.[8] Bei Bedarf werden Programmteile zwischen den beiden Speichern hin- und hergeschoben. Im Wesentlichen erzeugt das Betriebssystem die Abstraktion eines Adressraums als einer Menge von Adressen, auf die ein Prozess zugreifen kann. Der Adressraum ist entkoppelt vom physischen Speicher der Maschine, er kann sowohl größer als auch kleiner als dieser physische Speicher sein. Die Verwaltung der Adressräume und des physischen Speichers bildet einen wichtigen Teil der Betriebssystemaufgaben. Das gesamte *Kapitel 3* ist diesem Thema gewidmet.

1.5.3 Dateien

Ein anderes Schlüsselkonzept nahezu aller Betriebssysteme ist das Dateisystem. Eine der Hauptfunktionen eines PC-Betriebssystems besteht darin, die Eigenheiten von Platten und anderen Ein-/Ausgabegeräten zu verbergen und dem Programmierer stattdessen ein schönes und klares abstraktes Modell von geräteunabhängigen Dateien zu präsentieren. Offenbar werden Systemaufrufe benötigt, um Dateien zu erzeugen, zu verschieben, zu lesen und zu beschreiben. Bevor eine Datei gelesen werden kann, muss sie auf der Platte lokalisiert und geöffnet werden. Und nach dem Lesen sollte sie auch wieder geschlossen werden. Dafür werden Systemaufrufe bereitgestellt.

8 Anm. d. Fachlektors: Hier wird die Technik des virtuellen Speichers in Kombination mit der Technik des Swappings erklärt. Streng genommen sind dies zwei verschiedene Speicherverwaltungsarten.

Um die Möglichkeit zu schaffen, Dateien an einem Ort aufzubewahren, unterstützen die meisten Betriebssysteme das Konzept der **Verzeichnisse** (*directory*) zur Gruppierung von Dateien. Ein Student könnte zum Beispiel ein Verzeichnis für jedes Seminar, das er belegt (für die in diesem Seminar benötigten Programme), ein anderes Verzeichnis für elektronische Nachrichten und ein weiteres Verzeichnis für seine Homepage haben. Systemaufrufe werden jetzt benötigt, um Verzeichnisse zu erzeugen und zu verschieben. Ebenso werden Systemaufrufe bereitgestellt, um eine existierende Datei in ein Verzeichnis einzufügen und Dateien aus einem Verzeichnis zu entfernen. Einträge in einem Verzeichnis können entweder Dateien oder andere Verzeichnisse sein. Aus diesem Modell entsteht eine Hierarchie – das Dateisystem – wie es in ▶ *Abbildung 1.14* dargestellt ist.

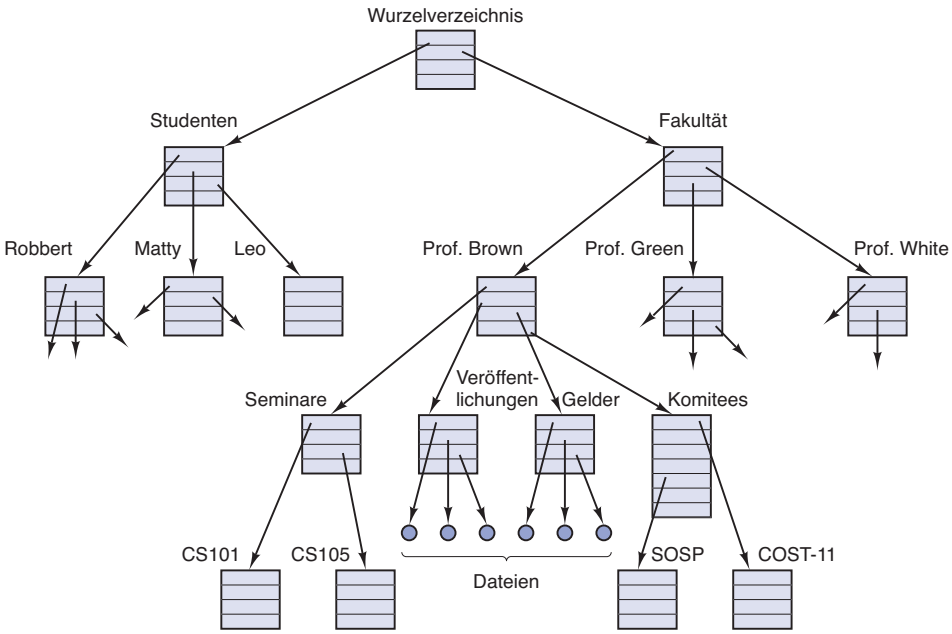

Abbildung 1.14: Das Dateisystem einer Fakultät in einer Universität.

Sowohl die Prozess- als auch die Dateihierarchie sind in Form eines Baums organisiert, damit enden allerdings die Ähnlichkeiten auch schon. Die Prozesshierarchien sind gewöhnlich nicht sehr tief (mehr als drei Ebenen ist unüblich), wohingegen Dateihierarchien im Normalfall vier, fünf oder noch mehr Stufen umfassen. Prozesshierarchien sind typischerweise kurzlebig, im Allgemeinen höchstens einige wenige Minuten, die Verzeichnishierarchie kann dagegen mehrere Jahre existieren. Bezüglich Eigentümerverhältnisse und Schutz unterscheiden sich Prozesse und Dateien ebenfalls deutlich voneinander: Typischerweise kann nur ein Elternprozess einen Kindprozess steuern oder auf ihn zugreifen, aber es existieren fast immer Mechanismen, um das Lesen von Dateien und Verzeichnissen von einer größeren Gruppe als nur dem Eigentümer zuzulassen.

Jede Datei innerhalb einer Verzeichnishierarchie kann durch Angabe eines **Pfadnamens** (*path name*) angesprochen werden, der an der Spitze der Verzeichnishierarchie, dem **Wurzelverzeichnis** (*root directory*), beginnt. Solche absoluten Pfadnamen bestehen aus einer Liste von Verzeichnissen, die von der Wurzel ausgehend durchlaufen werden müssen, um auf die Datei zuzugreifen. Die einzelnen Komponenten werden durch Schrägstriche getrennt. Der Pfad für die Datei *CS101* in ▶*Abbildung 1.14* lautet */Fakultät/Prof.Brown/Seminare/CS101*. Der führende Schrägstrich zeigt an, dass der Pfad absolut ist, das heißt, der Pfad startet im Wurzelverzeichnis. Nebenbei bemerkt wird unter Windows (aus historischen Gründen) der sogenannte Backslash (\) als Trennzeichen zwischen Pfadelementen anstelle des „normalen" Schrägstriches (/) benutzt. Der oben genannte Pfad würde also dann *\Fakultät\Prof.Brown\Seminare\CS101* lauten. In diesem Buch verwenden wir grundsätzlich die UNIX-Konvention (/) für Pfadnamen.

Jedem Prozess ist immer ein aktuelles **Arbeitsverzeichnis** (*working directory*) zugeordnet, von dem aus die Pfadnamen aufgelöst werden, die nicht mit einem Schrägstrich beginnen. Ist zum Beispiel in ▶*Abbildung 1.14 /Fakultät/Prof.Brown* das aktuelle Arbeitsverzeichnis, dann führt die Verwendung des Pfads *Seminare/CS101* zu derselben Datei wie die Angabe des obigen absoluten Pfadnamens. Prozesse können ihr jeweiliges Arbeitsverzeichnis durch Systemaufrufe verändern, wobei das neue Arbeitsverzeichnis als Argument anzugeben ist.

Bevor eine Datei gelesen oder beschrieben werden kann, muss sie geöffnet werden. Dabei werden die Zugriffsrechte überprüft. Wenn der Zugriff erlaubt ist, liefert das System eine kleine ganze Zahl, den sogenannten **Dateideskriptor** (*file descriptor*), der in den nachfolgenden Operationen verwendet wird. Wenn der Zugriff verweigert wurde, dann wird ein Fehlercode zurückgegeben.

Ein weiteres wichtiges Konzept unter UNIX sind die eingebundenen Dateisysteme (*mounted file system*). Die meisten Desktoprechner haben ein oder zwei optische Laufwerke, in die CD-ROMs, DVDs und Blu-Ray-Discs eingelegt werden können. Fast alle verfügen über USB-Ports, an denen USB-Sticks (bzw. eigentlich SSDs, *Solid State Disk*) angeschlossen werden können, und einige Computer haben Diskettenlaufwerke oder externe Festplatten. UNIX bietet einen eleganten Weg für den Umgang mit diesen Wechselmedien an, indem das Dateisystem eines optischen Laufwerks in die Dateisystemhierarchie eingebunden wird. Betrachten Sie beispielsweise die Struktur in ▶*Abbildung 1.15a*: Bevor der `mount`-Befehl ausgeführt wird, sind die beiden Dateisysteme, das **Wurzeldateisystem** auf der Festplatte und das Dateisystem der CD-ROM, völlig getrennt und ohne Beziehung zueinander.

Das Dateisystem auf der CD-ROM kann noch nicht genutzt werden, da man keine Pfadnamen innerhalb dieses Dateisystems angeben kann. UNIX erlaubt nämlich keine Pfadnamen, die mit einem Laufwerksbuchstaben oder einer Zahl beginnen – das wäre ja genau die Art Geräteabhängigkeit, die ein Betriebssystem beseitigen sollte. Stattdessen erlaubt der `mount`-Systemaufruf, das Dateisystem der CD-ROM in die Verzeichnishierarchie des Computers aufzunehmen, und zwar an jeder Stelle, die das Programm wünscht. In ▶*Abbildung 1.15b* wurde das Dateisystem der CD-ROM in das Verzeichnis *b* eingehängt. Nun kann man die Dateien */b/x* und */b/y* ansprechen. Falls Verzeich-

nis *b* vorher Dateien enthielt, sind diese nun so lange verdeckt, wie die CD-ROM eingehängt ist, da */b* ja jetzt auf das Wurzelverzeichnis der CD-ROM verweist. (Dass man nicht auf die verdeckten Dateien zugreifen kann, sieht auf den ersten Blick schlimmer aus, als es ist: Dateisysteme werden meist in leere Verzeichnisse eingehängt.) Verfügt ein System über mehrere Festplatten, können diese auch in einen einzigen Baum eingehängt werden.

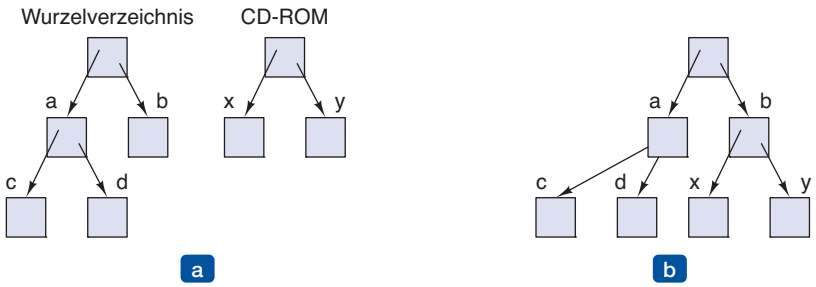

Abbildung 1.15: (a) Vor Ausführung des `mount`-Befehls konnte auf die Dateien der CD-ROM nicht zugegriffen werden. (b) Nach dem `mount`-Befehl sind sie Teil der Dateihierarchie.

Ein weiteres wichtiges Konzept unter UNIX ist die **Spezialdatei** (*special file*). Spezialdateien werden eingeführt, damit Ein-/Ausgabegeräte wie Dateien aussehen. Auf diese Weise können sie mit denselben Systemaufrufen wie Dateien gelesen und beschrieben werden. Es gibt zwei Arten von Spezialdateien: **Blockdateien** (*block special file*) und **Zeichendateien** (*character special file*). Blockdateien werden benutzt, um Geräte abzubilden, die aus einer Menge frei adressierbarer Blöcke bestehen, wie zum Beispiel Platten. Durch das Öffnen einer Blockdatei und das Lesen von beispielsweise Block 4 kann ein Programm unmittelbar auf den vierten Block des Geräts zugreifen, ohne sich um die Struktur des Dateisystems auf dem Gerät kümmern zu müssen. Auf dieselbe Art werden Zeichendateien benutzt, um Drucker, Modems und andere Geräte abzubilden, die zeichenorientiert arbeiten. Standardmäßig liegen die Spezialdateien im */dev*-Verzeichnis. Zum Beispiel wäre */dev/lp* der Drucker (früher Zeilendrucker (*line printer*) genannt).

Das letzte Merkmal, das wir in diesem Überblick besprechen, betrifft sowohl Prozesse als auch Dateien: Pipes. Eine **Pipe**[9] ist eine Art Pseudodatei, die verwendet werden kann, um zwei Prozesse miteinander zu verbinden, ▶ *Abbildung 1.16*. Bevor zwei Prozesse *A* und *B* mittels einer Pipe kommunizieren können, müssen sie zuvor diesen Kanal einrichten. Wenn Prozess *A* an Prozess *B* Daten senden möchte, schreibt er diese in die Pipe. Der Schreibvorgang wird analog zum Schreibvorgang in eine Ausgabedatei durchgeführt. Tatsächlich hat die Implementierung einer Pipe Ähnlichkeit mit der einer Datei. Prozess *B* kann die Daten lesen, indem er sie wie von einer Eingabedatei aus der Pipe liest. Auf diese Weise ist die Kommunikation zwischen Prozessen in UNIX dem gewöhnlichen Lesen und Schreiben einer Datei sehr ähnlich. Genauer gesagt gibt es nur einen einzigen Weg, wie ein Prozess feststellen kann, dass die Aus-

9 Wird auch als unidirektionaler Kanal bezeichnet.

gabedatei, die er gerade beschreibt, eigentlich keine Datei, sondern eine Pipe ist: Er muss einen speziellen Systemaufruf verwenden. Da Dateisysteme sehr wichtig sind, werden wir in *Kapitel 4* noch sehr viel mehr dazu sagen und uns auch in *Kapitel 10* und *11* näher damit beschäftigen.

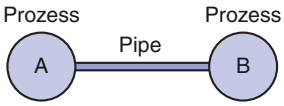

Abbildung 1.16: Zwei Prozesse, die durch eine Pipe verbunden sind.

1.5.4 Ein- und Ausgabe

Alle Computer besitzen physische Geräte, die Eingaben annehmen und Ausgaben produzieren. Was hätte man letzten Endes auch von einem Computer, wenn man ihm nicht mitteilen könnte, was er tun soll, und wenn man die Ergebnisse seiner Arbeit nicht bekommen könnte? Es gibt viele Arten von Ein-/Ausgabegeräten, einschließlich Tastaturen, Bildschirme, Drucker usw. Zu den Aufgaben des Betriebssystems gehört es, all diese Geräte zu verwalten.

Demzufolge hat jedes Betriebssystem ein Ein-/Ausgabe-Untersystem für die Geräteverwaltung. Manche Ein-/Ausgabesoftware ist geräteunabhängig, das heißt, sie passt für viele oder alle Geräte gleichermaßen. Andere Teile wiederum wie Gerätetreiber wurden speziell für ein bestimmtes Gerät programmiert. In *Kapitel 5* gehen wir auf Software für die Ein-/Ausgabe ein.

1.5.5 Datenschutz und Datensicherheit

Computer verwalten eine große Menge von Informationen, die der Benutzer oft schützen und vertraulich behandeln möchte. Das können beispielsweise E-Mails, Geschäftspläne, Steuererklärungen und andere sensiblen Daten sein. Es ist die Aufgabe des Betriebssystems, die Systemsicherheit zu wahren, sodass zum Beispiel Dateien nur von autorisierten Benutzern gelesen werden können.

Nur um eine Vorstellung davon zu bekommen, wie Sicherheit funktionieren kann, betrachten wir ein kleines Beispiel aus UNIX: Dateien unter UNIX werden mit einem 9-Bit-Code geschützt. Dieser Code besteht aus drei 3-Bit-Feldern: einem für den Eigentümer, einem für alle Benutzer der Gruppe, in der der Eigentümer eingetragen ist, und einem für alle anderen Benutzer. Die Einteilung der Benutzer in bestimmte Gruppen regelt der Systemadministrator. Jedes Feld wiederum besitzt ein Bit für den Lesezugriff, eines für den Schreibzugriff und eines für die Berechtigung, die Datei auszuführen. Diese Bits kennt man auch unter dem Namen **rwx-Bits** (von **r***ead*, **w***rite* und *e***x***ecute*). Die Vergabe der Rechte von *rwxr-x--x* bedeutet beispielsweise, dass der Eigentümer die Datei lesen, beschreiben und ausführen darf. Andere Mitglieder der Gruppe dürfen die Datei nur lesen und ausführen, nicht aber beschreiben. Ein Benutzer, der weder Eigentümer noch Mitglied der Gruppe ist, darf diese Datei nur ausfüh-

ren und weder beschreiben noch lesen. Bei einem Verzeichnis bedeutet das *x*-Bit die Erlaubnis zum Durchsuchen. Ein „-“ steht immer für ein fehlendes Recht.

Zum Schutz der Dateien kommen noch weitere Sicherheitsmechanismen hinzu. Das System muss beispielsweise vor Angreifern geschützt werden, die entweder Benutzer sein können oder automatisch arbeiten, wie zum Beispiel Computerviren. Das Thema Sicherheit (*security*) wird ausführlicher in *Kapitel 9* behandelt.

1.5.6 Die Shell

Das Betriebssystem ist für die Ausführung von Systemaufrufen zuständig. Editoren, Compiler, Assembler, Binder, Dienstprogramme und Kommandozeileninterpreter gehören sicher nicht zum Betriebssystem, auch wenn es sehr wichtige und nützliche Werkzeuge sind. Auf die Gefahr hin, die Dinge ein wenig durcheinanderzubringen, werden wir uns in diesem Abschnitt kurz den UNIX-Kommandozeileninterpreter, die **Shell**, ansehen. Die Shell ist zwar kein Teil des Betriebssystems, nutzt jedoch sehr stark dessen Eigenschaften und ist ein gutes Beispiel dafür, wie man Systemaufrufe nutzt. Sie ist zudem die wichtigste Schnittstelle zwischen einem Benutzer, der an seinem Terminal sitzt, und dem Betriebssystem, solange keine grafische Benutzungsoberfläche eingesetzt wird. Es gibt viele Shells wie die *sh*, *csh*, *ksh* und die *bash*. Alle unterschiedlichen Typen bieten die Funktionalität der ursprünglichen Shell, der *sh*.

Sobald sich ein Benutzer am System anmeldet, wird die Shell gestartet. Die Shell benutzt das Terminal als Standardeingabe und Standardausgabe. Sie beginnt mit der Ausgabe eines Zeichens zur Eingabeaufforderung, auch **Prompt** genannt, wie zum Beispiel dem $-Zeichen. Dadurch wird dem Benutzer angezeigt, dass die Shell auf Kommandos wartet. Wenn der Benutzer nun zum Beispiel

```
date
```

eingibt, erzeugt die Shell einen Kindprozess und lässt das Programm *date* als Kindprozess laufen. Während dieser Kindprozess ausgeführt wird, wartet die Shell auf dessen Terminierung. Ist der Kindprozess beendet, dann gibt die Shell erneut das Prompt-Zeichen aus und versucht, die nächste Eingabezeile zu lesen.

Der Benutzer kann die Standardausgabe auf eine Datei umlenken, indem er zum Beispiel Folgendes eingibt:

```
date >file
```

In ähnlicher Weise kann die Standardeingabe umgelenkt werden:

```
sort <file1 >file2
```

Dadurch wird ein Sortierprogramm aufgerufen, das seine Eingaben aus der Datei *file1* entnimmt und seine Ausgaben in die Datei *file2* schreibt.

Die Ausgabe eines Programms kann von einem anderen Programm als Eingabe benutzt werden, indem die beiden durch eine Pipe miteinander verbunden werden. So wird in

```
cat file1 file2 file3 | sort >/dev/lp
```

das Programm *cat* aufgerufen, um die drei Dateien aneinanderzuhängen (*con*cat*enate*). Die Ausgabe wird danach an *sort* übergeben, um die Zeilen in alphabetischer Reihenfolge zu sortieren. Die Ausgabe von *sort* wird auf die Datei */dev/lp* umgelenkt, die ein typischer Name für eine Zeichendatei zur Ansteuerung eines Druckers ist.

Wenn der Benutzer ein „&“ an ein Kommando anfügt, so wartet die Shell nicht auf dessen Terminierung. Stattdessen wird sofort das Prompt-Zeichen ausgegeben. So startet beispielsweise

```
cat file1 file2 file3 | sort >/dev/lp &
```

das Sortieren als Hintergrundjob. Dadurch kann der Benutzer seine Arbeit während des Sortierens normal fortsetzen. Die Shell hat viele interessante Fähigkeiten, auf die wir aus Platzgründen leider nicht weiter eingehen können. Die meisten UNIX-Bücher behandeln auch die Shell ausführlich (z.B. Kernighan und Pike, 1984; Quigley, 2004; Robbins, 2005).

Die meisten PCs benutzen heutzutage eine grafische Benutzungsoberfläche (GUI). Diese GUI ist eigentlich nur ein Programm, das über dem Betriebssystem läuft, genau wie eine Shell. In Linux-Systemen ist dies sehr offensichtlich, weil der Benutzer zwischen (mindestens) zwei GUIs wählen kann: Gnome oder KDE oder überhaupt keine (dann wird das Terminal-Fenster von X11 benutzt). Auch in Windows ist es möglich, die Standard-GUI (Windows Explorer) durch ein anderes Programm zu ersetzen. Dazu müssen einige Werte in der Registrierungsdatenbank (*registry*) verändert werden. Dies wird allerdings nur von sehr wenigen Leuten genutzt.

1.5.7 Die Ontogenese rekapituliert die Phylogenese

Nach Veröffentlichung des Buchs „Über die Entstehung der Arten“ von Charles Darwin postulierte der deutsche Zoologe Ernst Haeckel, dass „die Ontogenese die Phylogenese rekapituliere“. Damit meinte er, dass die Entwicklung eines Embryos (Ontogenese) die Evolution der Art (Phylogenese) wiederholt. Mit anderen Worten durchläuft eine menschliche Eizelle nach der Befruchtung verschiedene Stadien: Erst ist sie ein Fisch, dann ein Schwein und immer so weiter, bevor sie sich in ein menschliches Baby verwandelt. Obwohl diese Aussage heutzutage unter Biologen als grobe Vereinfachung gilt, so steckt doch ein Körnchen Wahrheit darin.

Etwas Vergleichbares ist auch in der Computerindustrie geschehen. Jede neue Art (Großrechner, Minicomputer, PC, Handheld-Computer, eingebettetes System, Smartcard usw.) scheint durch dieselben Entwicklungsstadien zu gehen wie ihre Vorgänger – sowohl was die Hardware als auch was die Software betrifft. Wir vergessen häufig, dass viel von dem, was in der Computerindustrie und in vielen anderen Geschäftsfeldern passiert, technologiegetrieben ist. Der Grund, warum die alten Römer keine Autos hatten, ist nicht, dass sie so gerne zu Fuß gingen – sie wussten nur einfach nicht, wie man Autos baut. PCs existieren *nicht*, weil Millionen von Menschen ein jahrhundertelang aufgestautes Verlangen hatten, einen Computer zu besitzen, sondern weil es jetzt möglich ist, sie billig herzustellen. Wir vergessen häufig, wie sehr die

Technologie unsere Sicht auf Systeme beeinflusst und es lohnt sich, dies von Zeit zu Zeit zu reflektieren.

Insbesondere kommt es häufig vor, dass durch technologische Veränderungen ein Konzept veraltet und schnell wieder verschwindet. Jedoch könnte schon die nächste technologische Änderung dieses Konzept wieder zu neuem Leben erwecken. Dies trifft vor allem dann zu, wenn sich die Veränderung nur auf einen Teil des Systems bezieht, das dadurch im Vergleich zu den anderen Teilen leistungsstärker wird. Als die Prozessoren beispielsweise bedeutend schneller als die Speicher wurden, kamen die Cache-Speicher auf, um den „langsamen" Speicher zu beschleunigen. Werden durch eine neue Technologie eines Tages die Speicher viel schneller als Prozessoren, so wird der Cache-Speicher verschwinden. Und wenn dann durch eine neue Technologie die Prozessoren wieder schneller als die Speicher werden, dann werden wohl auch die Cache-Speicher wieder auftauchen. In der Biologie stirbt etwas für immer aus, in der Informatik manchmal nur für ein paar Jahre.

Als eine Folge dieser Unbeständigkeit werden wir in diesem Buch von Zeit zu Zeit „veraltete" Konzepte ansehen, das heißt Ideen, die aus der Sicht der heutigen Technologie nicht mehr optimal sind. Gleichwohl könnten Änderungen in der Technologie einige dieser sogenannten „veralteten Konzepte" wiederbeleben. Deshalb sollte man verstehen, warum ein Konzept als veraltet gilt und welche Veränderungen es möglicherweise zurückbringen könnten.

Um dies noch deutlicher zu machen, lassen Sie uns ein Beispiel betrachten: Die ersten Computer hatten festverdrahtete Befehlssätze. Die Befehle wurden direkt von der Hardware ausgeführt und konnten nicht verändert werden. Dann kam die Mikroprogrammierung (in großem Umfang zuerst mit der IBM 360 eingeführt), bei der ein zugrunde liegender Interpreter die „Hardwarebefehle" auf Softwareebene ausführt. Festverdrahtete Befehle veralteten – sie waren nicht flexibel genug. Dann wurden RISC-Computer entwickelt und die Mikroprogrammierung (d.h. interpretierte Befehle) veralteten, weil direkte Ausführungen schneller waren. Im Moment beobachten wir die Wiederauferstehung der Interpretationen in Form von Java-Applets, die über das Internet verschickt und bei Ankunft interpretiert werden. Ausführungsgeschwindigkeit ist hier nicht so entscheidend, da die Verzögerungen durch das Netzwerk so groß sind, dass diese gewöhnlich dominieren. Das Pendel ist also zwischen direkter Ausführung und Interpretation schon mehrmals hin- und hergeschwungen und wird vielleicht in der Zukunft auch noch weiterschwingen.

Große Speicher

Wir wollen nun einige historische Entwicklungen in der Hardware untersuchen und welchen Einfluss diese immer wieder auf die Software hatten. Die ersten Großrechner verfügten nur über einen begrenzten Speicher. Eine vollständig geladene IBM 7090 oder 7094, von Ende 1959 bis 1964 die Königin unter den Rechnern, hatte kaum mehr als 128 KB an Speicherkapazität. Sie war hauptsächlich in Assembler programmiert, um kostbaren Speicherplatz zu sparen.

Im Laufe der Zeit wurden Compilersprachen wie FORTRAN und COBOL so gut, dass Assembler für tot erklärt wurde. Aber als der erste kommerzielle Minicomputer (die PDP-1) erschien, hatte diese nur 4096 18-Bit-Wörter an Speicher und Assembler feierte ein überraschendes Comeback. Schließlich bekamen die Minicomputer mehr Speicher und höhere Programmiersprachen hielten auch hier Einzug.

Als Mikrocomputer in den frühen 1980er Jahren aufkamen, hatten die ersten unter ihnen 4 KB an Speicherkapazität und die Assemblersprachen standen von den Toten auf. Eingebettete Systeme benutzen häufig dieselben CPU-Chips wie die Mikrocomputer (8080er, Z80s und spätere 8086er) und wurden anfangs ebenso in Assembler programmiert. Heute haben die Nachfahren der Mikrocomputer, die PCs, viel Speicher und sind in C, C++, Java und anderen höheren Programmiersprachen geschrieben. Smartcards machen eine ähnliche Entwicklung durch: Obwohl jenseits einer gewissen Größe, haben sie oft eine Java-VM (Virtuelle Maschine) und interpretieren Java-Programme, anstatt sie in übersetzter Form in der Maschinensprache der Smartcard auszuführen.

Hardwareschutz

Die ersten Großrechner wie die IBM 7090/7094 hatten keinen Hardwareschutz, es konnte also immer jeweils nur ein Programm laufen. Ein fehlerhaftes Programm konnte das Betriebssystem aushebeln und die Maschine schnell abstürzen lassen. Mit der Einführung der IBM 360 wurde eine primitive Form des Hardwareschutzes verfügbar. Diese Maschinen konnten nun mehrere Programme gleichzeitig im Speicher halten und sie abwechselnd laufen lassen (Multiprogrammierung). Monoprogrammierung wurde für veraltet erklärt, zumindest bis der erste Minicomputer auftauchte – ohne Hardwareschutz. Multiprogrammierung war also wieder nicht möglich. Dies traf für die PDP-1 und die PDP-8 zu, die PDP-11 verfügte endlich über die nötigen Schutzmechanismen und diese Eigenschaft führte zur Multiprogrammierung und schließlich zu UNIX.

Als die ersten Mikrocomputer gebaut wurden, benutzten sie den Intel-8080-CPU-Chip, der keinen Hardwareschutz hatte, womit wir wieder bei der Monoprogrammierung waren – jeweils nur ein Programm im Speicher. Erst der Intel-80286-Chip wurde um Schutzmechanismen ergänzt und Multiprogrammierung wurde möglich. Bis zum heutigen Tag haben viele eingebettete Systeme keinen Hardwareschutz und lassen nur ein einziges Programm laufen.

Werfen wir nun einen Blick auf die Betriebssysteme. Die ersten Großrechner besaßen anfangs keinen Schutzmechanismus und keine Unterstützung für Multiprogrammierung. Also lief auf ihnen ein einfaches Betriebssystem, das jeweils nur ein manuell geladenes Programm verwalten konnte. Später kam die nötige Hardware- und Betriebssystemunterstützung hinzu, um mehrere Programme gleichzeitig verwalten zu können. Damit besaßen sie alle Timesharing-Eigenschaften.

Als die Minicomputer zuerst auftauchten, hatten auch sie keinen Hardwareschutz und konnten jeweils nur ein manuell geladenes Programm ausführen, obwohl Multipro-

grammierung in der Welt der Großrechner schon längst etabliert war. Nach und nach kamen Schutzmechanismen hinzu und damit auch die Fähigkeit, mehrere Programme gleichzeitig laufen zu lassen. Auch die ersten Mikrocomputer konnten jeweils nur ein Programm ausführen und bekamen erst später die Fähigkeit zur Multiprogrammierung. Handheld-Computer und Smartcards durchliefen dieselben Stationen.

In allen Fällen wurde die Softwareentwicklung von der Technologie diktiert. Beispielsweise besaßen die ersten Mikrocomputer etwa 4 KB Speicher und keinerlei Hardwareschutz. Höhere Programmiersprachen und Multiprogrammierung waren für solche winzigen Computer einfach zu viel zu verwalten. Als sich die Mikrocomputer langsam in unsere moderne PCs verwandelten, erhielten sie die nötige Hardware und dann auch die notwendige Software für noch fortschrittlichere Eigenschaften. Wahrscheinlich wird diese Entwicklung in den kommenden Jahren noch andauern. Auch in anderen Bereichen gibt es dieses Rad der Wiedergeburten, doch in der Computerindustrie scheint es sich schneller zu drehen.

Platten

Die ersten Großrechner waren weitgehend magnetbandbasiert. Sie lasen ein Programm vom Band, übersetzten es, ließen es ablaufen und schrieben das Ergebnis zurück auf ein anderes Band. Es gab keine Platten und kein Konzept für Dateisysteme. Dies änderte sich, als IBM 1956 die erste Festplatte einführte – die RAMAC (RAndoM ACcess). Diese Platte nahm eine Fläche von 4 qm ein und konnte 5 Millionen 7-Bit-Zeichen speichern, genug für ein Digitalfoto in mittlerer Auflösung.

Charakteristisch für diese neuen Entwicklungen war der CDC 6600, der 1964 eingeführt wurde und über Jahre der mit Abstand schnellste Computer der Welt. Benutzer konnten sogenannte „permanente Dateien“ erzeugen, indem sie ihnen Namen gaben und hofften, dass kein anderer Benutzer ebenfalls beispielsweise „Daten“ für einen passenden Dateinamen hielt. Es gab insgesamt nur ein Verzeichnis. Schließlich entwickelten Großrechner komplexe hierarchische Dateisysteme, die vielleicht in dem MULTICS-Dateisystem ihren Höhepunkt fanden.

Die ersten Minicomputer hatten teilweise auch schon Festplatten. Als die PDP-11 1970 eingeführt wurde, war sie standardmäßig mit der RK05-Platte ausgerüstet, die mit 2,5 MB zwar nur ungefähr die Hälfte der Kapazität des RAMAC von IBM hatte, aber dafür auch nur einen Durchmesser von 40 cm und eine Höhe von 5 cm aufwies. Auch die RK05 hatte anfangs nur ein einstufiges Verzeichnis. Als dann die Mikrocomputer herauskamen, war CP/M anfangs das vorherrschende Betriebssystem, das auch nur ein Verzeichnis (auf dem Diskettenlaufwerk) unterstützte.

Virtueller Speicher

Das Konzept des virtuellen Speichers bietet die Möglichkeit, ein Programm ablaufen zu lassen, das größer als der physische Speicher ist, indem Teile zwischen RAM und Platte schnell hin- und hergeschoben werden. Dieses Konzept hat eine ähnliche Entwicklung durchlaufen, es tauchte zuerst bei Großrechnern auf und wanderte dann zu

den Mini- und den Mikrocomputern. Virtueller Speicher hat es auch möglich gemacht, ein Programm dynamisch zur Laufzeit mit einer Bibliothek zu verlinken, anstatt diese beim Übersetzen fest in das Programm einzubauen. MULTICS war das erste System, bei dem dies umgesetzt wurde. Diese Idee pflanzte sich schließlich nach unten fort und ist jetzt auf den meisten UNIX- und Windows-Systemen verbreitet.

Bei all diesen Entwicklungen sehen wir, dass Konzepte in einem bestimmten Kontext erfunden und dort später – wenn der Kontext sich änderte (Assemblersprachen, Monoprogrammierung, einstufige Verzeichnisse usw.) – verworfen wurden, nur um oft ein Jahrzehnt später in einem anderen Kontext wiederaufzutauchen. Deshalb werden wir in diesem Buch manchmal einen Blick auf Ideen und Algorithmen werfen, die zwar für die heutigen Gigabyte-PCs überholt sind, aber vielleicht bald für eingebettete Systeme oder Smartcards zurückkommen.

1.6 Systemaufrufe

Wir haben gesehen, dass Betriebssysteme zwei Hauptfunktionen haben: Abstraktionen für Benutzerprogramme zur Verfügung zu stellen und die Betriebsmittel des Computers zu verwalten. Größtenteils bezieht sich die Interaktion zwischen Benutzerprogrammen und dem Betriebssystem auf die erste Funktion, wie zum Beispiel das Erzeugen, Schreiben, Lesen und Löschen von Dateien. Der Betriebssystemteil, der für die Ressourcenverwaltung zuständig ist, stellt sich dem Nutzer weitgehend transparent dar und arbeitet automatisch. Das heißt, an der Schnittstelle zwischen Nutzerprogramm und dem Betriebssystem geht es hauptsächlich um die Abstraktionen. Um die Funktionsweise eines Betriebssystems richtig zu verstehen, müssen wir diese Schnittstelle genau untersuchen. Die zur Verfügung stehenden Systemaufrufe unterscheiden sich von einem Betriebssystem zum anderen (die zugrunde liegenden Konzepte sind jedoch ähnlich).

Wir sind daher gezwungen, uns zwischen (1) verschwommenen allgemeingültigen Aussagen („Betriebssysteme besitzen einen Systemaufruf zum Lesen von Dateien“) und (2) der Betrachtung eines speziellen Systems („UNIX besitzt einen `read`-Systemaufruf mit drei Parametern: einen, um die Datei auszuwählen; einen für den Zeiger auf den Zieldatenbereich; und einen, um die Anzahl der zu lesenden Zeichen festzulegen“) zu entscheiden.

Wir haben uns hier für die zweite Variante entschieden. Das bedeutet zwar mehr Aufwand, aber es ermöglicht einen besseren Einblick in die Funktionsweise eines Betriebssystems. Obwohl sich unsere Betrachtung hauptsächlich auf POSIX (International Standard 9945-1) und damit also auf UNIX, System V, BSD, Linux, MINIX 3 usw. bezieht, haben die meisten anderen Betriebssysteme Systemaufrufe für dieselben Funktionen, auch wenn diese im Detail etwas anders aussehen. Der Mechanismus, wie ein Systemaufruf gestartet wird, ist meistens sehr stark von der Hardware abhängig und muss oft in Assembler geschrieben werden. Daher gibt es auch eine Bibliotheksfunktion, die einen Systemaufruf von höheren Programmiersprachen wie C und anderen Sprachen zulässt.

Es ist nützlich, Folgendes im Kopf zu behalten: Jede Einprozessormaschine kann jeweils nur eine Anweisung bearbeiten. Wenn ein Prozess im Benutzermodus arbeitet und einen Systemaufruf wie etwa das Lesen einer Datei nutzen will, dann muss er einen Unterbrechungsbefehl ausführen und dadurch die Kontrolle an das Betriebssystem übergeben. Das Betriebssystem erkennt den Wunsch des Prozesses, indem es die Parameter untersucht. Danach wird der Systemaufruf ausgeführt und die Kontrolle an das Benutzerprogramm zurückgegeben, von dem der nächste Befehl nach dem Systemaufruf ausgeführt wird. Eigentlich ähnelt der Systemaufruf dem Aufruf einer speziellen Prozedur. Allerdings können nur Systemaufrufe in den Kern eintreten, eine gewöhnliche Prozedur darf das nicht.

Damit der Mechanismus eines Systemaufrufs klarer wird, werfen wir nun einen kurzen Blick auf den `read`-Aufruf. Wie bereits oben erwähnt, besitzt er drei Parameter: Der erste ist der Dateideskriptor, der zweite zeigt auf einen Datenpuffer und der dritte enthält die Anzahl der zu lesenden Zeichen. So wie fast alle Systemaufrufe wird `read` von einem C-Programm aufgerufen, indem eine Bibliotheksfunktion mit dem gleichen Namen wie der Systemaufruf angesprochen wird, in diesem Fall also *read*. Ein Aufruf eines C-Programms könnte also wie folgt aussehen:

```
count = read(fd, buffer, nbytes);
```

Der Systemaufruf (und die entsprechende Bibliotheksfunktion) liefert die Anzahl der Zeichen durch die Variable *count* zurück, die die tatsächlich gelesenen Zeichen angibt. Dieser Wert ist normalerweise gleich *nbytes*, kann aber auch kleiner sein, wenn etwa während des Lesens das Ende der Datei erreicht wurde.

Wenn der Systemaufruf wegen falscher Parameter oder wegen eines Lesefehlers nicht ausgeführt werden konnte, dann wird *count* auf −1 gesetzt. Die Fehlernummer wird in einer globalen Variablen abgelegt, die *errno* heißt. Jedes Programm sollte diese Variable nach einem Systemaufruf testen und prüfen, ob ein Fehler aufgetreten ist.

Systemaufrufe werden in einer Reihe von Schritten durchgeführt. Um dies noch deutlicher zu machen, wollen wir den `read`-Systemaufruf näher untersuchen. Bevor die Bibliotheksfunktion *read* aufgerufen wird, die dann wiederum den Systemaufruf `read` ausführt, legt das aufrufende Programm die Parameter auf den Stack, wie in den Schritten 1 bis 3 in ▶ *Abbildung 1.17* gezeigt wird.

C- und C++-Compiler legen die Parameter in umgekehrter Reihenfolge auf den Stack (aus historischen Gründen, damit der erste Parameter von *printf*, der Formatstring, ganz oben auf dem Stack erscheint). Der erste und der dritte Parameter werden als Wert übergeben, der zweite dagegen als Referenz. Das heißt, es wird nicht der Inhalt des Puffers, sondern nur ein Zeiger auf den Anfang des Puffers übergeben (was durch das & gekennzeichnet wird). Dann folgt der Sprung in die Bibliotheksfunktion (Schritt 4). Dieser Befehl ist ein normaler Befehl zum Aufruf von Funktionen.

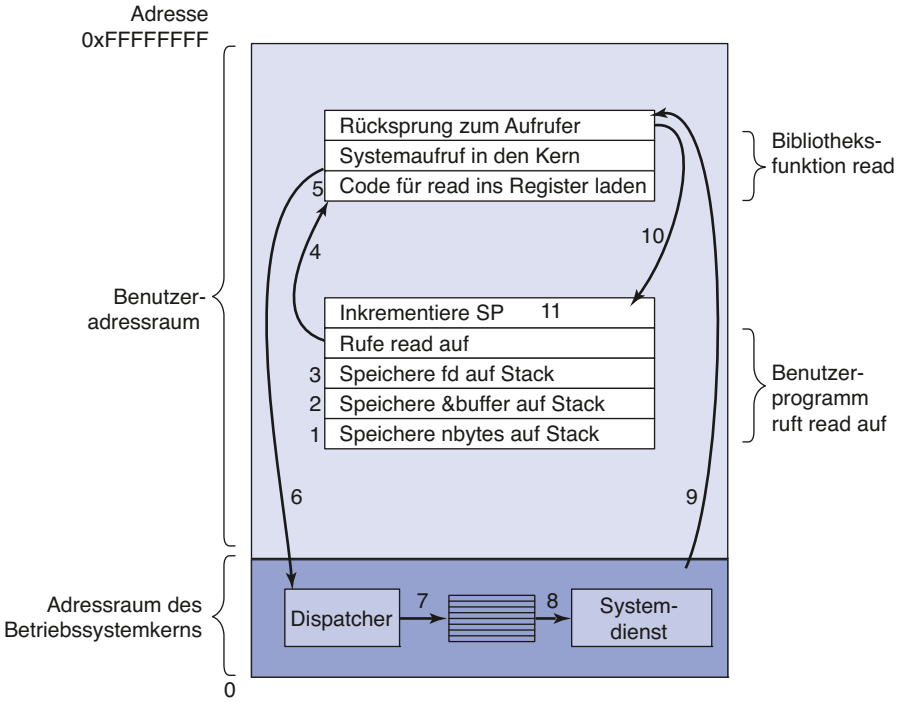

Abbildung 1.17: Die elf Schritte für den Systemaufruf `read(fd, buffer, nbytes)`.

Die Bibliotheksfunktion, die eventuell in Assembler geschrieben ist, speichert im Normalfall die Nummer des Systemaufrufs an einem Ort, an dem das Betriebssystem den Wert erwartet, beispielsweise in einem Register (Schritt 5). Dann wird der `TRAP`-Befehl ausgeführt, um vom Benutzermodus in den Kernmodus zu wechseln, und eine feste Adresse im Kern angesprungen (Schritt 6). Der `TRAP`-Befehl ist in der Tat dem Befehl zum Aufruf einer Prozedur recht ähnlich, und zwar in dem Sinne, dass der folgende Befehl von einem anderen Speicherort geholt wird und die Rücksprungadresse auf dem Stack zur späteren Verwendung gespeichert wird.

Dennoch unterscheidet sich der `TRAP`-Befehl vom Prozeduraufruf auf zwei grundsätzliche Arten: Erstens schaltet er als Nebeneffekt in den Kernmodus um, während der Prozeduraufruf den Modus beibehält. Zweitens wird nicht wie beim Prozeduraufruf eine relative oder absolute Adresse angegeben, an der die jeweilige Prozedur gespeichert ist, da der `TRAP`-Befehl nicht an eine beliebige Adresse springen kann. Je nach Architektur springt er entweder zu einer einzelnen festgelegten Stelle oder zu einer Adresse, die durch eine Tabelle mit Sprungadressen im Speicher und einem Index festgelegt ist, der durch ein 8-Bit-Feld im `TRAP`-Befehl bestimmt ist.

Der Programmcode im Kern, der dem `TRAP`-Befehl folgt, springt dann anhand der Systemaufrufnummer zu dem richtigen Systemaufruf, der gewöhnlich durch eine Tabelle aus Funktionszeigern mit allen Systemaufrufen gefunden wird. Die Systemaufrufnummer ist ein Index für den entsprechenden Eintrag in dieser Tabelle (Schritt 7). Zu diesem

Zeitpunkt startet der Systemaufruf (Schritt 8). Sobald der Systemaufruf abgeschlossen ist, kann die Kontrolle an die Bibliotheksfunktion im Benutzermodus zurückgegeben werden, die dem TRAP-Befehl folgt (Schritt 9). Diese Funktion kehrt danach zum Benutzerprogramm zurück, so wie es auch eine normale Prozedur getan hätte (Schritt 10).

Damit der Systemaufruf beendet werden kann, muss das Benutzerprogramm den Stack aufräumen, wie es nach jedem Funktionsaufruf getan wird (Schritt 11). Angenommen, der Stack wächst nach unten (was recht häufig der Fall ist), dann erhöht das übersetzte Programm den Stackpointer genau so weit, dass die Parameter, die vor dem *read*-Aufruf gespeichert wurden, wieder entfernt werden. Und das Benutzerprogramm darf jetzt wieder weiterarbeiten.

Im Schritt 9 haben wir aus gutem Grund gesagt: „... kann die Kontrolle an die Bibliotheksfunktion im Benutzermodus zurückgegeben werden“. Der Systemaufruf kann nämlich das aufrufende Programm auch blockieren und an der Weiterarbeit hindern. Wenn beispielsweise von der Tastatur gelesen werden soll, aber noch nichts eingetippt wurde, dann muss das aufrufende Programm blockiert werden. In diesem Fall sieht sich das Betriebssystem nach einem anderen Prozess um, der inzwischen weiterarbeiten kann. Wenn die erwartete Eingabe dann später vorhanden ist, bekommt der Prozess die Aufmerksamkeit des Betriebssystems zurück und die Schritte 9 bis 11 werden ausgeführt.

In den nächsten Abschnitten betrachten wir die am häufigsten benutzten POSIX-Systemaufrufe oder genauer gesagt die Bibliotheksfunktionen, die diese aufrufen. POSIX kennt etwa 100 Prozeduren. Einige der wichtigsten sind in ▶ *Abbildung 1.18* aufgeführt und werden der Einfachheit halber in vier Kategorien eingeteilt. In den folgenden Abschnitten wird die Arbeitsweise jedes Aufrufs kurz erläutert.

Die Dienste dieser Systemaufrufe bestimmen weitgehend die Hauptaufgaben eines Betriebssystems, da die Ressourcenverwaltung bei PCs meistens nicht sehr aufwendig ist (zumindest im Vergleich zu sehr großen Maschinen mit vielen Benutzern). Zu diesen Diensten zählen Aufgaben wie das Erzeugen und Beenden von Prozessen, das Erzeugen, Löschen, Lesen und Schreiben von Dateien, die Verzeichnisverwaltung und die Durchführung der Ein-/Ausgabe.

Es sollte an dieser Stelle noch darauf hingewiesen werden, dass die Zuordnung zwischen den POSIX-Funktionsaufrufen und den Systemaufrufen nicht eins zu eins erfolgt ist. Der POSIX-Standard definiert zwar einige Funktionen, die ein POSIX-konformes System anbieten muss, aber es wird nicht definiert, ob es sich dabei um einen Systemaufruf, eine Bibliotheksfunktion oder etwas anderes handelt. Wenn eine Prozedur ohne einen Systemaufruf ausgeführt werden kann (d.h., ohne in den Kernmodus zu wechseln), wird sie meistens im Benutzermodus realisiert, um Zeit zu sparen. Die meisten POSIX-Funktionen rufen allerdings Systemaufrufe auf. Gewöhnlich wird einer Funktion auch genau ein Systemaufruf zugeordnet. In einigen Fällen aber, insbesondere wenn mehrere angeforderte Prozeduren nur kleine Unterschiede aufweisen und von einem Systemaufruf abgedeckt werden können, werden mehrere Bibliotheksfunktionen für einen Systemaufruf zur Verfügung gestellt.

Prozessverwaltung

Aufruf	Beschreibung
pid = fork( )	Erzeugen eines neuen Kindprozesses
pid = waitpid(pid, &statloc, options)	Warten auf Beendigung eines Kindprozesses
s = execve(name, argv, environp)	Speicherabbild eines Prozesses ersetzen
exit(status)	Prozess beenden und Status zurückliefern

Dateiverwaltung

Aufruf	Beschreibung
fd = open(file, how, ...)	Datei zum Lesen, Schreiben oder für beides öffnen
s = close(fd)	Offene Datei schließen
n = read(fd, buffer, nbytes)	Daten aus Datei in Puffer lesen
n = write(fd, buffer, nbytes)	Daten vom Puffer in Datei schreiben
position = lseek(fd, offset, whence)	Dateipositionszeiger bewegen
s = stat(name, &buf)	Status einer Datei ermitteln

Verzeichnis- und Dateisystemverwaltung

Aufruf	Beschreibung
s = mkdir(name, mode)	Erzeugen eines neuen Verzeichnisses
s = rmdir(name)	Löschen eines leeren Verzeichnisses
s = link(name1, name2)	Erzeugen eines neuen Eintrags name2, der auf name1 zeigt
s = unlink(name)	Verzeichniseintrag löschen
s = mount(special, name, flag)	Dateisystem einhängen
s = umount(special)	Eingehängtes Dateisystem entfernen

Sonstige Systemaufrufe

Aufruf	Beschreibung
s = chdir(dirname)	Wechseln des Arbeitsverzeichnisses
s = chmod(name, mode)	Ändern der Dateirechte
s = kill(pid, signal)	Signal an einen Prozess senden
seconds = time(&seconds)	Abgelaufene Zeit seit dem 1. Januar 1970 erfragen

Abbildung 1.18: Einige der wichtigsten POSIX-Systemaufrufe. Der Rückgabewert ist −1, wenn ein Fehler aufgetreten ist. Die angegebenen Rückgabewerte bedeuten Folgendes: *pid* ist eine Prozess-ID, *fd* ist ein Dateideskriptor, *n* ist eine Anzahl von Zeichen, *position* ist die Position innerhalb einer Datei, *seconds* ist die abgelaufene Zeit. Die Parameter werden im Text erklärt.

1.6.1 Systemaufrufe zur Prozessverwaltung

Die erste Gruppe von Aufrufen in ▶*Abbildung 1.18* behandelt die Prozessverwaltung. Der Systemaufruf `fork` ist ein gutes Beispiel, um mit der Beschreibung zu beginnen. `fork` ist der einzige Weg, um unter POSIX einen neuen Prozess zu erzeugen. Er erzeugt eine exakte Kopie des aufrufenden Prozesses, einschließlich aller Dateideskriptoren, der Register etc. Nachdem `fork` beendet ist, laufen der ursprüngliche und der neu erzeugte Prozess (Eltern- und Kindprozess) getrennt voneinander weiter. Unmittelbar nach der Kopie besitzen zunächst beide Prozesse die gleichen Variableninhalte, aber nachdem die Daten des Elternprozesses zur Erzeugung des Kindprozesses kopiert wurden, wirken sich alle nachfolgenden Veränderungen nicht mehr auf den jeweils anderen Prozess aus. (Der unveränderbare bzw. konstante Programmtext wird allerdings von beiden gemeinsam genutzt.) Der Rückgabewert von `fork` ist entweder die Null, wenn er im neu erzeugten Kindprozess abgefragt wird, oder gleich der **Prozessnummer** (**Prozess-ID**, **PID**) des Kindprozesses im Elternprozess[10]. Aufgrund der PID kann jeder der beiden Prozesse erkennen, wer der Kind- und wer der Elternprozess ist.

Nach dem Aufruf von `fork` muss der Kindprozess meistens einen anderen Programmcode als der Elternprozess ausführen. Nehmen wir beispielsweise die Shell: Sie liest einen Befehl vom Terminal, spaltet einen neuen Kindprozess ab, wartet auf die Ausführung des Kommandos durch diesen Prozess und liest nach Beendigung des Kindprozesses das nächste Kommando ein. Um auf einen Kindprozess zu warten, führt der Elternprozess den Systemaufruf `waitpid` aus, der einfach auf die Beendigung eines oder mehrerer Kindprozesse wartet. Der Systemaufruf `waitpid` kann auf einen speziellen oder anderen beliebigen Kindprozess warten, indem ihm als erster Parameter -1 übergeben wird. Nach der Beendigung von `waitpid` zeigt die Adresse des zweiten Parameters *statloc* auf das Ergebnis des Kindprozesses (normale oder fehlerhafte Beendigung und den Rückgabewert). Einige spezielle Optionen können mittels des dritten Parameters an den Systemaufruf übergeben werden. Dazu gehört zum Beispiel der sofortige Rücksprung, wenn noch keiner der Kindprozesse beendet ist.

Betrachten wir nun, wie `fork` von der Shell benutzt wird. Wenn ein Kommando eingegeben wurde, erzeugt die Shell einen neuen Prozess. Dieser Kindprozess muss das Benutzerkommando ausführen. Dazu nutzt er den Systemaufruf `execve`, der einen kompletten Prozess im Speicher durch eine andere Datei ersetzt, wobei der Name der Datei als erster Parameter mit übergeben werden muss. (Der Systemaufruf heißt eigentlich `exec`, aber mehrere Bibliotheksfunktionen rufen ihn mit unterschiedlichen Parametern unter leicht verschiedenen Namen auf. Wir behandeln hier alle als Systemaufrufe.) In ▶*Abbildung 1.19* wird eine stark vereinfachte Shell dargestellt, die `fork`, `waitpid` und `execve` benutzt.

10 Anm. d. Fachlektors: Merkhilfe: Da jeder Kindprozess nur genau einen Elternprozess besitzt und diesen mit dem Systemaufruf `getppid` abfragen kann, erhält der Kindprozess die Null zurück, Elternprozesse können viele Kindprozesse erzeugen, daher müssen diese die PID des neuen Kinds zurückgeliefert bekommen.

```
#define TRUE 1
while (TRUE) {                          /* Endlosschleife */
   type_prompt( );                      /* Prompt ausgeben */
   read_command(command, parameters);   /* Befehl einlesen */

   if (fork( ) != 0) {                  /* Kindprozess erzeugen */
      /* Code des Elternprozesses */
      waitpid(-1, &status, 0);          /* auf Beendigung von Kindprozess warten */
   } else {
      /* Code des Kindprozesses */
      execve(command, parameters, 0);   /* Befehl ausführen */
   }
}
```

Abbildung 1.19: Eine kleine Version einer Shell. In diesem Buch wird *TRUE* durchgängig als Variable mit Inhalt 1 definiert.

Im allgemeinen Fall hat `execve` drei Parameter: den Namen der auszuführenden Datei, einen Zeiger auf eine Liste mit den zu übergebenden Argumenten und einen Zeiger auf die Umgebungsvariablen. Die Bedeutung der Parameter wird gleich noch beschrieben. Verschiedene Bibliotheksfunktionen wie *execv*, *execle* oder *execve* erlauben es, dass bestimmte Parameter weggelassen oder auf unterschiedliche Arten angegeben werden können. Hier benutzen wir den Namen `exec`, um den Systemaufruf darzustellen, der durch alle Varianten der Bibliotheksfunktion aufgerufen wird.

Betrachten wir nun folgendes Kommando zum Kopieren einer Datei *file1* nach *file2*:

```
cp file1 file2
```

Nachdem die Shell einen neuen Prozess erzeugt hat, führt der Kindprozess das Kommando *cp* aus und übergibt die Namen von Quell- und Zieldatei.

Die Hauptfunktion *main* des Programms *cp* (und die Hauptfunktion fast aller anderen C-Programme) enthält die Deklaration

```
main(argc, argv, envp)
```

Das Argument *argc* zeigt die Anzahl der Argumente auf der Kommandozeile an, wobei der Programmname mitgezählt wird. Im obigen Beispiel hat *argc* den Wert 3.

Der zweite Parameter *argv* ist ein Zeiger auf ein Array, wobei das Element *i* in diesem Array einen Zeiger auf die *i*-te Zeichenfolge der Kommandozeile darstellt. In unserem Beispiel bedeutet das, dass *argv*[0] auf die Zeichenfolge „cp“, *argv*[1] auf „file1“ und *argv*[2] auf „file2“ zeigt.

Der dritte Parameter in *main* ist *envp*, ein Zeiger auf die Umgebungsvariablen. Eine Umgebungsvariable ist ein Feld von Zeichenfolgen, das Zuweisungen der Form *name* = *value* enthält. Damit können Informationen wie die Art des Terminals oder der Name eines Benutzerverzeichnisses an das Programm übergeben werden. Einige Bibliotheksfunktionen können von Programmen aufgerufen werden, um die Inhalte der Umgebungsvariablen abzufragen. Diese werden oft benutzt, um die Durchführung bestimmter Aufgaben den Benutzerwünschen anzupassen (z.B. den Standarddrucker

festlegen). Im Beispiel der ▶*Abbildung 1.19* werden keine Umgebungsvariablen an den Kindprozess übergeben, deshalb ist der dritte Parameter von *execve* null.

Wenn `exec` jetzt sehr kompliziert aussieht, verzweifeln Sie bitte nicht, es ist wohl (auf der semantischen Ebene) der komplexeste aller POSIX-Systemaufrufe. Alle anderen sind wesentlich einfacher zu verstehen, wie beispielsweise `exit`, das zur Beendigung eines Prozesses verwendet wird. Der Systemaufruf `exit` besitzt genau einen Parameter, den Exit-Status (0 bis 255), der dem Elternprozess durch den Systemaufruf `waitpid` mittels *statloc* zurückgegeben wird.

Jeder Prozess unter UNIX teilt seinen Speicher in drei Segmente auf: das **Textsegment** (für den Programmcode), das **Datensegment** (für Variable) und das **Stacksegment** (auch **Stapelsegment** genannt). Das Datensegment wächst im Speicher nach oben und das Stacksegment nach unten, wie in ▶*Abbildung 1.20* zu sehen ist. Dazwischen befindet sich ein freier Bereich von ungenutzten Adressen. Der Stack wächst automatisch nach Bedarf in diese Lücke hinein, das Datensegment dagegen kann nur explizit durch den Systemaufruf `brk` erweitert werden. Dazu wird die neue Endadresse des Datensegmentes angegeben. Dieser Aufruf ist allerdings nicht durch den POSIX-Standard definiert, da Programmierer eigentlich die Bibliotheksfunktion *malloc* für die dynamische Speicherbelegung benutzen sollen. Die zugrunde liegende Implementierung von *malloc* wurde nicht als geeigneter Gegenstand für eine Standardisierung angesehen, da nur wenige Programmierer dies direkt verwenden. Heute weiß wahrscheinlich niemand mehr, dass `brk` kein POSIX-Standard ist.

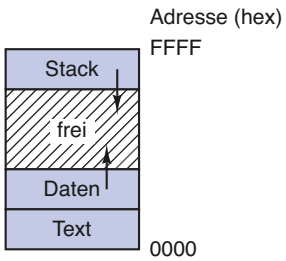

Abbildung 1.20: Prozesse besitzen drei Segmente: das Text-, das Daten- und das Stacksegment.

1.6.2 Systemaufrufe zur Dateiverwaltung

Viele Systemaufrufe betreffen das Dateisystem. In diesem Abschnitt betrachten wir Systemaufrufe, die einzelnen Dateien gelten, im nächsten Abschnitt untersuchen wir die Aufrufe, die mit Verzeichnissen oder dem Dateisystem als Ganzem zu tun haben.

Damit eine Datei gelesen oder geschrieben werden kann, muss sie zunächst geöffnet werden. Dieser Aufruf bekommt den Dateinamen übergeben, der entweder mit einem absoluten Pfad oder relativ zum aktuellen Verzeichnis angegeben werden kann. Zusätzlich wird noch ein Code übergeben – *O_RDONLY*, *O_WRONLY* oder *O_RDWR* –, der entweder das Öffnen zum Lesen, zum Schreiben oder beides erlaubt. Um eine neue Datei zu erzeugen, wird der Parameter *O_CREAT* angegeben. Der zurückgege-

bene Dateideskriptor lässt sich danach zum Lesen und Schreiben verwenden. Später kann die Datei mit `close` geschlossen werden, womit auch der Dateideskriptor wieder für kommende `open`-Aufrufe freigegeben wird.

Die am häufigsten verwendeten Systemaufrufe sind zweifellos `read` und `write`. Wir haben `read` bereits kennengelernt, `write` hat dieselben Parameter.

Auch wenn die meisten Programme Dateien sequenziell lesen und beschreiben, so müssen doch einige Programme auf einen beliebigen Teil innerhalb einer Datei zugreifen können. Jede Datei besitzt einen Zeiger, der die aktuelle Position innerhalb der Datei angibt. Beim sequenziellen Lesen bzw. Schreiben wird dieser Zeiger immer auf das nächste zu lesende bzw. zu beschreibende Byte gesetzt. Der `lseek`-Systemaufruf verändert den Wert dieses Positionszeigers, sodass folgende `read`- oder `write`-Aufrufe an einer beliebigen Stelle innerhalb der Datei beginnen können.

Der Systemaufruf `lseek` hat drei Parameter: Der erste ist der Dateideskriptor, der zweite die gewünschte Zielposition und der dritte gibt an, ob sich diese relativ zum Anfang der Datei, zur aktuellen Position oder zum Ende der Datei befindet. Der Rückgabewert von `lseek` ist die absolute Position in der Datei (in Byte angegeben), nachdem der Zeiger verändert wurde.

Für jede Datei speichert UNIX die Dateiart (reguläre Datei, Spezialdatei, Verzeichnis usw.), die Größe, den Zeitpunkt der letzten Änderung sowie weitere Informationen. Programme können diese Informationen mit dem `stat`-Systemaufruf abfragen. Der erste Parameter gibt die zu untersuchende Datei an, der zweite ist ein Zeiger auf eine Struktur, in die das Ergebnis abgespeichert werden soll. Der `fstat`-Aufruf macht dasselbe bei geöffneten Dateien.

1.6.3 Systemaufrufe zur Verzeichnisverwaltung

Hier beschäftigen wir uns nun mit Systemaufrufen, die Verzeichnisse oder das Dateisystem als Ganzes betreffen. Die ersten beiden Aufrufe dieser Gruppe (▶ *Abbildung 1.18*), `mkdir` und `rmdir`, erzeugen bzw. löschen leere Verzeichnisse. Der nächste Systemaufruf ist `link`, der es einer Datei erlaubt, unter verschiedenen Namen in unterschiedlichen Verzeichnissen vorzukommen. Eine typische Anwendung ist die gemeinsame Nutzung einer Datei von mehreren Mitgliedern einer Gruppe (z. B. eines Programmierteams), die die Datei in ihren eigenen Verzeichnissen haben, eventuell sogar unter verschiedenen Namen. Eine Datei gemeinsam zu nutzen ist nicht dasselbe, wie jedem Gruppenmitglied eine eigene Kopie zu geben: Bei einer gemeinsam genutzten Datei wird jede Änderung sofort für jedes Mitglied der Gruppe sichtbar, da ja nur eine Datei existiert. Wenn dagegen Kopien der Datei erzeugt wurden, wirken sich die nachfolgenden Änderungen, die in einer Kopie gemacht werden, nicht auf die anderen Kopien aus.

Damit man versteht, wie `link` funktioniert, betrachten wir die Situation in ▶ *Abbildung 1.21a*. Es gibt hier zwei Benutzer, *ast* und *jim*. Jeder besitzt sein eigenes Verzeichnis mit einigen Dateien. Wenn der Benutzer *ast* nun ein Programm ausführt, das den Aufruf

```
link("/usr/jim/memo", "/usr/ast/note");
```

enthält, dann erscheint die Datei *memo* aus dem Verzeichnis von *jim* im Verzeichnis von *ast* als Datei *note*. Danach beziehen sich */usr/jim/memo* und */usr/ast/note* auf dieselbe Datei. Nebenbei bemerkt ist es egal, ob die Benutzerverzeichnisse der Benutzer in */usr*, */user* oder */home* abgelegt werden, das ist nur eine Entscheidung des lokalen Systemadministrators.

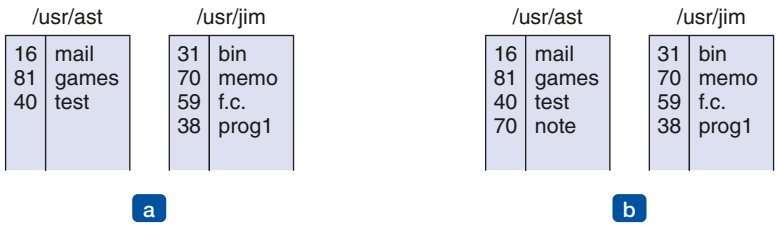

Abbildung 1.21: (a) Zwei Verzeichnisse, bevor */usr/jim/memo* in das Verzeichnis *ast* verlinkt wurde; (b) dieselben Verzeichnisse nach dem Aufruf von `link`.

Wenn man die Arbeitsweise von `link` versteht, wird wahrscheinlich auch klarer, was der Aufruf genau bewirkt. Jede Datei unter UNIX besitzt zur Identifizierung eine eindeutige Nummer, die sogenannte I-Nummer. Diese I-Nummer ist ein Index in einer Tabelle mit **I-Nodes**. Jeder I-Node gibt für jede Datei an, wem die Datei gehört, wo die Blöcke auf der Platte liegen und so weiter. Ein Verzeichnis ist somit lediglich eine Datei mit einer Menge von Paaren aus I-Nummer und ASCII-Name. In den ersten UNIX-Versionen bestand ein Verzeichniseintrag aus genau 16 Byte – 2 Byte für die I-Nummer und 14 Byte für den Namen. Mittlerweile ist eine etwas kompliziertere Struktur notwendig geworden, damit auch längere Namen vergeben werden können, aber im Prinzip ist ein Verzeichnis immer noch eine Menge von Paaren aus I-Nummer und ASCII-Name. In ▶ *Abbildung 1.21* hat *mail* die I-Nummer 16 und so weiter. Der Systemaufruf `link` erzeugt nun einfach einen völlig neuen Verzeichniseintrag mit einem (möglichst neuen) Namen, der die I-Nummer von einer existierenden Datei verwendet. In ▶ *Abbildung 1.21b* haben zwei Einträge dieselbe I-Nummer (70) und zeigen deshalb auf dieselbe Datei. Wenn später einer der beiden mit dem `unlink`-Systemaufruf gelöscht wird, bleibt der andere Eintrag bestehen. Wenn beide gelöscht werden, erkennt UNIX, dass keine Einträge für diese Datei mehr bestehen (ein Eintrag in der I-Node-Tabelle zählt die Anzahl der aktuellen Zeiger auf diese Datei mit), also wird die Datei gelöscht.

Wie bereits erwähnt, erlaubt es der `mount`-Systemaufruf, dass zwei Dateisysteme zu einem zusammengefasst werden. Normalerweise befindet sich das Wurzelverzeichnis mit den binären (ausführbaren) Versionen der gebräuchlichsten Kommandos und anderer häufig benutzter Dateien auf einer Partition der Festplatte und die Benutzerdateien auf einer anderen Partition. Der Anwender kann dann einen USB-Datenträger mit Dateien zum Lesen einlegen.

Durch das Ausführen des `mount`-Systemaufrufs kann das US-Dateisystem in ein Unterverzeichnis des Wurzelverzeichnisses eingehängt werden, wie in ▶ *Abbildung 1.22* zu sehen ist. Eine typische Anweisung dazu in C lautet:

```
mount("/dev/sdb0", "/mnt", 0);
```

Der erste Parameter ist dabei der Name der Blockdatei für das USB-Laufwerk 0, der zweite Parameter legt fest, an welcher Stelle im Verzeichnisbaum eingehängt werden soll, und der dritte Parameter gibt an, ob das Dateisystem nur zum Lesen oder sowohl zum Lesen als auch zum Schreiben geöffnet werden soll.

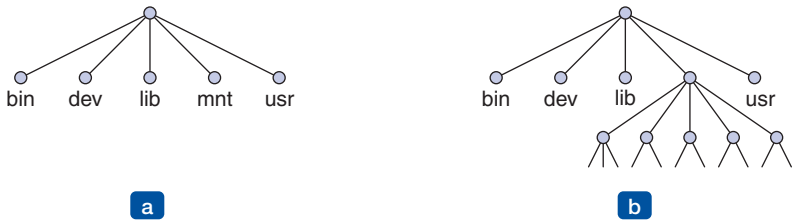

Abbildung 1.22: Dateisystem vor dem Aufruf von `mount`; (b) Dateisystem nach dem `mount`-Aufruf.

Nach dem Aufruf von `mount` kann jede Datei von Laufwerk 0 nur durch Angabe des Pfads vom Wurzelverzeichnis oder vom Arbeitsverzeichnis aus angesprochen werden, ohne dass man sich Gedanken machen muss, auf welchem Laufwerk die Datei liegt. Natürlich kann auch noch ein zweites, ein drittes und ein viertes Laufwerk irgendwo in den Baum eingehängt werden. Mit dem Aufruf von `mount` lassen sich auch Wechseldatenträger (*removable media*) in die Verzeichnisstruktur integrieren, ohne später überlegen zu müssen, auf welchem Gerät eine Datei liegt. Neben USB-Datenträgern wie in diesem Beispiel können auch logische Teile von Festplatten (oft **Partitionen**, aus Sicht der Gerätetreiber und im Englischen auch **minor device** genannt) ebenso wie externe Festplatten eingehängt werden. Wenn ein Dateisystem nicht mehr gebraucht wird, kann es einfach mit dem `umount`-Systemaufruf ausgehängt werden.[11]

1.6.4 Sonstige Systemaufrufe

Zusätzlich zu den bisher besprochenen Aufrufen existieren noch verschiedene Systemaufrufe, die sich keiner Gruppe zuordnen lassen. Von diesen werden wir uns hier noch vier ansehen. Der `chdir`-Aufruf wechselt das aktuelle Arbeitsverzeichnis eines Prozesses. Nach dem Systemaufruf

```
chdir("/usr/ast/test");
```

und dem Befehl zum Öffnen einer Datei *xyz* wird die Datei */usr/ast/test/xyz* geöffnet. Das Konzept der Arbeitsverzeichnisse verhindert, dass ständig (lange) absolute Pfadnamen eingetippt werden müssen.

11 Eigentlich „unmount“, aber das „n“ ist im Laufe der UNIX-Entwicklung irgendwann auf der Strecke geblieben.

Unter UNIX besitzt jede Datei einen Zugriffsmodus. Dieser Modus enthält die rwx-Bits jeweils für den Eigentümer der Datei, für die Gruppe und für alle anderen. Der `chmod`-Systemaufruf ermöglicht die Änderung dieses Modus für eine Datei. Um eine Datei *file* beispielsweise für alle zum Lesen zuzulassen und nur dem Besitzer zusätzlich das Schreiben zu erlauben, genügt der Aufruf

```
chmod("file", 0644);
```

Mit dem Systemaufruf `kill` können Benutzer und Prozesse Signale verschicken. Wenn ein Prozess ein Signal verarbeiten kann, dann wird die Signalverarbeitungsroutine aufgerufen, sobald das entsprechende Signal ankommt. Ist der Prozess allerdings nicht dazu eingerichtet, das Signal zu verarbeiten, so wird er durch das ankommende Signal „gekillt“ (daher auch der Name des Aufrufs).

Im POSIX-Standard werden mehrere Funktionen im Zusammenhang mit Zeit definiert. Beispielsweise liefert der Systemaufruf `time` die aktuelle Zeit in Sekunden zurück, wobei 0 die Zeit am 1.1.1970 um Mitternacht ist (zu Beginn des Tages). Bei Computern mit einer Wortbreite von 32 Bit ist der höchste Wert von `time` $2^{32}-1$ Sekunden (wenn eine vorzeichenlose ganze Zahl zur Darstellung verwendet wird). Dieser Wert entspricht einem Zeitraum von etwas mehr als 136 Jahren. Demnach werden 32-Bit-UNIX-Systeme im Jahr 2106 quasi durchdrehen – nicht unähnlich dem Jahr-2000-Problem, das ein Chaos unter den Computern dieser Welt angerichtet hätte, wenn die IT-Branche nicht einen gewaltigen Aufwand betrieben hätte, dieses Problem zu lösen. Sollten Sie also zurzeit noch ein 32-Bit-System verwenden, sind Sie gut beraten, es vor 2106 gegen ein 64-Bit-System auszutauschen.

1.6.5 Die Win32-Programmierschnittstelle (API) unter Windows

Bisher haben wir uns hauptsächlich auf UNIX konzentriert. Jetzt ist es an der Zeit, auch einen kurzen Blick auf Windows werfen. Windows und UNIX unterscheiden sich ganz fundamental in ihren Programmiermodellen. Ein UNIX-Programm besteht aus Code, der irgendeine Aktion durchführt und Systemaufrufe ausführt, um bestimmte Dienste zu nutzen. Ein Windows-Programm ist dagegen normalerweise ereignisgesteuert[12]. Das Hauptprogramm wartet auf ein Ereignis und ruft danach eine Funktion auf, die dieses Ereignis behandelt. Solch ein Ereignis kann ein Tastendruck, eine Mausbewegung, das Drücken einer Maustaste oder das Anschließen eines USB-Laufwerks sein. Dabei werden Behandlungsroutinen aufgerufen, um das jeweilige Ereignis zu bearbeiten, den Bildschirminhalt abzugleichen oder den internen Zustand des Programms zu ändern. Insgesamt führt dies zu einer anderen Art der Programmierung als unter UNIX. Da sich dieses Buch aber mit Betriebssystemfunktionen und -strukturen befasst, werden wir diese unterschiedlichen Programmiermodelle nicht mehr weiter betrachten.

12 Anm. d. Fachlektors: Zumindest dann, wenn es sich um ein Programm mit grafischer Bedienoberfläche handelt. Selbstverständlich kann man unter Windows auch das genannte UNIX-Programmiermodell nutzen.

Natürlich hat Windows auch Systemaufrufe. Unter UNIX existiert praktisch eine Eins-zu-eins-Relation zwischen den Systemaufrufen (z.B. `read`) und den entsprechenden Bibliotheksfunktionen (z.B. *read*), die die Systemaufrufe dann durchführen. Mit anderen Worten, für jeden Systemaufruf existiert ziemlich genau eine Bibliotheksfunktion, die aufgerufen wird, wie in ▶*Abbildung 1.17* zu sehen ist. Außerdem kennt POSIX nur etwa 100 Prozeduraufrufe.

Unter Windows sieht das grundlegend anders aus. Zunächst einmal sind die Bibliotheksfunktionen und die Systemaufrufe im Prinzip voneinander entkoppelt. Microsoft hat eine Menge von Funktionen definiert, die **Win32-API** (**Application Program Interface**) genannt wird und die Programmierer benutzen sollen, wenn sie Dienste des Betriebssystems in Anspruch nehmen wollen. Diese Schnittstelle wird (zumindest teilweise) von allen Windows-Betriebssystemen seit der Version Windows 95 angeboten. Da die Schnittstelle von den Systemaufrufen entkoppelt ist, hat Microsoft immer die Möglichkeit, die Systemaufrufe zu ändern, ohne damit die Funktionsfähigkeit existierender Programme zu gefährden. Woraus sich die Win32-API wirklich zusammensetzt, ist nicht ganz klar, da mit aktuellen Windows-Versionen jeweils viele neue Systemaufrufe eingeführt wurden, die es vorher noch nicht gab. In diesem Abschnitt bezeichnet Win32 immer die Schnittstelle, die von allen Windows-Versionen unterstützt wird. Win32 garantiert Kompatibilität unter allen Versionen.

Die Anzahl der Win32-API-Funktionen ist extrem groß und liegt bei einigen Tausend. Zwar führen viele dieser Funktionen Systemaufrufe aus, aber eine beträchtliche Anzahl wird auch vollständig im Benutzermodus abgearbeitet. Deshalb kann man unter Windows unmöglich entscheiden, was ein echter Systemaufruf ist (der vom Kern bearbeitet wird) und was einfach nur eine Bibliotheksfunktion ist (die im Benutzermodus ausgeführt wird). Zusätzlich gilt: was in der einen Windows-Version ein Systemaufruf ist, kann in einer anderen Version im Benutzeradressraum bearbeitet werden und umgekehrt. Wenn wir in diesem Buch die Windows-Systemaufrufe besprechen, beziehen wir uns, wo immer möglich, auf die Win32-Funktionen. Für diese Funktionen garantiert Microsoft, dass sie über lange Zeit hinweg gleich bleiben. Aber man sollte immer daran denken, dass einige davon keine echten Systemaufrufe sind und nicht in den Kernmodus eintreten.

Die Win32-API besitzt eine riesige Anzahl an Funktionen zum Verwalten von Fenstern, geometrischen Figuren, Text, Schriftarten, Scrollbalken, Dialogboxen, Menüs und anderer Elemente der GUI. Soweit das grafische Teilsystem im Kern abläuft (was aber nicht für alle Windows-Versionen gilt), handelt es sich um Systemaufrufe, ansonsten sind es nur Bibliotheksfunktionen. Sollten wir diese Funktionen dann hier beschreiben? Da sie eigentlich nichts mit der Funktionsweise eines Betriebssystems zu tun haben, entschieden wir uns dagegen, auch wenn sie möglicherweise im Kern ausgeführt werden. Leser, die sich mit der Win32-API näher beschäftigen wollen, sollten eines der zahlreichen Bücher zu diesem Thema lesen (beispielsweise Hart, 1997; Rector und Newcomer, 1997; Simon, 1997).

Da selbst das Aufzählen aller Win32-API-Funktionen hier unmöglich ist, werden wir uns auf die Einführung der Aufrufe beschränken, die in etwa eine gleiche Funktionali-

tät besitzen wie die UNIX-Systemaufrufe in ▶*Abbildung 1.18*. Diese werden in ▶*Abbildung 1.23* gegenübergestellt.

UNIX	Win32	Beschreibung
fork	CreateProcess	Erzeugen eines neuen Prozesses
waitpid	WaitForSingleObject	Warten auf das Ende eines Prozesses
execve	(nicht vorhanden)	CreateProcess = fork + execve
exit	ExitProcess	Ausführung beenden
open	CreateFile	Erzeugen einer Datei oder Öffnen einer existierenden Datei
close	CloseHandle	Datei schließen
read	ReadFile	Daten aus einer Datei lesen
write	WriteFile	Daten in eine Datei schreiben
lseek	SetFilePointer	Dateizeiger bewegen
stat	GetFileAttributesEx	Dateiattribute erfragen
mkdir	CreateDirectory	Erzeugen eines neuen Verzeichnisses
rmdir	RemoveDirectory	Löschen eines leeren Verzeichnisses
link	(nicht vorhanden)	Win32 unterstützt keine Links
unlink	DeleteFile	Löschen einer existierenden Datei
mount	(nicht vorhanden)	Win32 unterstützt kein Einhängen
umount	(nicht vorhanden)	Win32 unterstützt kein Einhängen, somit gibt es kein umount
chdir	SetCurrentDirectory	Ändern des aktuellen Arbeitsverzeichnisses
chmod	(nicht vorhanden)	Win32 unterstützt Security nicht (NT schon)
kill	(nicht vorhanden)	Win32 unterstützt keine Signale
time	GetLocalTime	Aktuelle Zeit erfragen

Abbildung 1.23: Die Win32-API-Aufrufe, die in etwa mit den UNIX-Systemaufrufen aus ▶*Abbildung 1.18* übereinstimmen. Es soll nicht unerwähnt bleiben, dass Windows noch sehr viele weitere Systemaufrufe besitzt, die aber kein Pendant in UNIX haben.

Im Folgenden gehen wir kurz die Liste aus ▶*Abbildung 1.23* durch. `CreateProcess` erzeugt einen neuen Prozess, der die Arbeit von `fork` und `execve` unter UNIX miteinander kombiniert. Die Funktion hat viele Parameter, die die Eigenschaften des neuen Prozesses genauer bestimmen. Windows besitzt keine Prozesshierarchie wie UNIX, deshalb gibt es auch keine Eltern- oder Kindprozesse. Nachdem ein Prozess erzeugt

wurde, sind der erzeugende Prozess und der erzeugte Prozess gleichwertig. Die Funktion `WaitForSingleObject` wird benutzt, um auf ein Ereignis zu warten. Es gibt viele Ereignisse, auf die man warten kann. Wenn der Parameter einen Prozess spezifiziert, dann wartet der Aufrufer auf die Beendigung dieses Prozesses, was mit `ExitProcess` geschieht.

Die nächsten sechs Aufrufe behandeln Dateien. Ihre Funktionalität entspricht in etwa der unter UNIX, auch wenn Parameter und einige Details unterschiedlich sind. Schließlich werden auch unter Windows Dateien geöffnet, geschlossen, gelesen und beschrieben, genau wie unter UNIX. Die Funktionen `SetFilePointer` und `GetFileAttributes` setzen die Leseposition innerhalb der Datei und liefern einige Dateiattribute.

Windows kennt auch Verzeichnisse, die mit den API-Aufrufen `CreateDirectory` und `RemoveDirectory` angelegt bzw. gelöscht werden können. Es gibt ein aktuelles Verzeichnis, das mit `SetCurrentDirectory` gesetzt wird. Die aktuelle Tageszeit lässt sich mit `GetLocalTime` erfragen.

Die Win32-Schnittstelle hat keine Funktionen für Links auf Dateien, für das Einhängen von Dateisystemen und auch keine Sicherheitsfunktionen (die aber in letzter Zeit entwickelt werden) oder ein Signalkonzept. Deshalb existieren hier keine den UNIX-Funktionen entsprechenden Aufrufe. Natürlich hat die Win32-API eine große Zahl von Aufrufen, die es unter UNIX nicht gibt und die hauptsächlich die GUI betreffen. Windows Vista verfügt über ein ausgefeiltes Sicherheitskonzept und unterstützt Links auf Dateien. Mit Windows 7 und Windows 8 kommen weitere Funktionen und Systemaufrufe hinzu.

Eine letzte Bemerkung zu Win32: Win32 ist keine besonders einheitliche oder konsistente Schnittstelle. Hauptsächlich ist dies der Notwendigkeit geschuldet, die Kompatibilität mit den früheren 16-Bit-Windows-Versionen aus Windows 3.x zu erhalten.

1.7 Betriebssystemstrukturen

Nachdem wir gesehen haben, wie Betriebssysteme von außen aussehen (d.h., wie die Schnittstelle zum Programmierer aussieht), wird es nun Zeit für einen Blick ins Innere. In den folgenden Abschnitten betrachten wir sechs unterschiedliche Strukturen, mit denen das Spektrum aller Möglichkeiten, ein Betriebssystem zu konstruieren, beleuchtet werden soll. Sie sind in keiner Weise vollständig, geben aber eine Vorstellung von den Entwürfen, die in der Praxis anzutreffen sind. Diese sechs Entwurfsmuster sind monolithische Systeme, geschichtete Systeme, Mikrokerne, Client-Server-Systeme, virtuelle Maschinen und Exokerne.

1.7.1 Monolithische Systeme

Die bei Weitem häufigste Organisationsform ist der monolithische Ansatz. Hier läuft das gesamte Betriebssystem als ein einziges Programm im Kernmodus. Das Betriebssystem ist als eine Menge von Prozeduren realisiert, die alle zu einem einzigen großen, ausführbaren binären Programm zusammengefügt sind. Wird diese Methode eingesetzt, dann darf jede Prozedur jede andere des Systems aufrufen, die eine nützliche Funktionalität bietet. Diese Möglichkeit, jede beliebige Prozedur aufrufen zu können, ist sehr effizient, doch wenn man Tausende von Prozeduren hat, die sich gegenseitig ohne Einschränkungen aufrufen können, dann kann dies zu einem schwerfälligen und schwer verständlichen System führen. Außerdem kann ein Fehler in einer dieser Prozeduren den Absturz des gesamten Betriebssystems nach sich ziehen.

Bei den monolithischen Systemen wird das aktuelle Objektprogramm eines Betriebssystems erzeugt, indem zuerst die einzelnen Prozeduren (oder die Dateien, die diese Prozeduren enthalten) übersetzt werden und dann vom Systembinder zu einer einzigen ausführbaren Datei verknüpft werden. Dabei sind alle Informationen und jede Funktion des Betriebssystems für jede andere Funktion sichtbar (im Gegensatz zu einer Struktur, die aus Modulen oder Einheiten besteht, in der große Teile der Informationen nur innerhalb einer Funktion verwendet werden können und bei der nur die offiziell festgelegten Einstiegspunkte von außerhalb des Moduls aufgerufen werden können).

Aber selbst in monolithischen Systemen ist es möglich, etwas Struktur einzubringen. Die Dienste (Systemaufrufe), die das Betriebssystem bereitstellt, werden angefordert, indem Parameter an wohldefinierten Stellen platziert werden, wie zum Beispiel auf dem Stack. Der eigentliche Aufruf findet dann durch einen speziellen Unterbrechungsbefehl (Trap) statt, der auch als Kernaufruf bekannt ist. Dieser Befehl schaltet die Maschine vom Benutzermodus in den Kernmodus und übergibt die Kontrolle an das Betriebssystem, wie der Schritt 6 in ▶ *Abbildung 1.17* darstellt. Das Betriebssystem überprüft dann die Parameter des Aufrufs, um festzustellen, welcher Systemaufruf ausgeführt werden soll. Über einen Index schaut das Betriebssystem danach in eine Tabelle, die im Eintrag *k* einen Zeiger auf die Prozedur enthält, die den Systemaufruf *k* ausführt (Schritt 7 in ▶ *Abbildung 1.17*).

Diese Mechanismen legen folgende Basisstruktur für das Betriebssystem nahe:

1. ein Hauptprogramm, das die angeforderte Dienstprozedur aufruft;
2. eine Menge von Dienstprozeduren, die die Systemaufrufe ausführen;
3. eine Menge von Hilfsfunktionen, die die Dienstprozeduren unterstützen.

In diesem Modell existiert für jeden Systemaufruf eine Dienstprozedur, die diesen Aufruf überwacht und ausführt. Die Hilfsfunktionen stellen Mechanismen bereit, die von verschiedenen Dienstprozeduren benötigt werden, wie das Kopieren von Daten aus einem Benutzerprogramm. Diese Aufteilung der Prozeduren in drei Ebenen ist in ▶ *Abbildung 1.24* dargestellt.

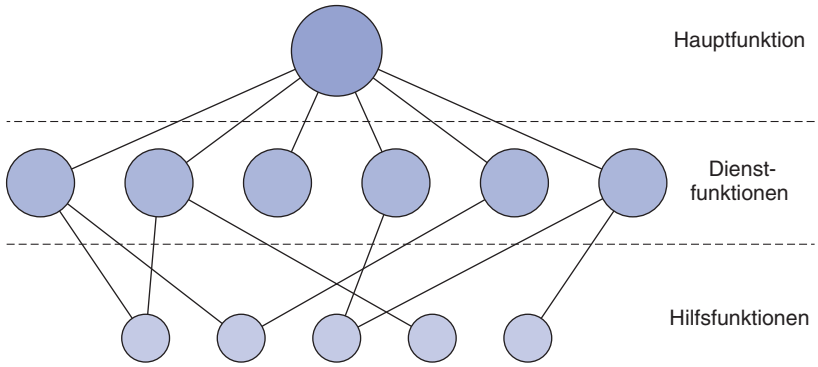

Abbildung 1.24: Ein einfaches Strukturmodell für ein monolithisches System.

Zusätzlich zu dem Kernbetriebssystem, das geladen wird, wenn der Computer hochgefahren wird, unterstützen viele Betriebssysteme ladbare Erweiterungen wie Treiber für Ein-/Ausgabegeräte und Dateisysteme. Diese Komponenten werden nur auf Anfrage geladen. Unter UNIX wird solch eine Komponente **Shared Library** (gemeinsam genutzte Bibliothek) genannt, unter Windows **DLL** (**Dynamic-Link Library**, dynamisch-gebundene Bibliothek). Letztere besitzen die Dateiendung *.dll* und in Windows-Systemen findet man im Verzeichnis *C:\Windows\system32* weit über 1000 davon.

1.7.2 Geschichtete Systeme

Eine Verallgemeinerung des Ansatzes aus ▶*Abbildung 1.24* besteht darin, das Betriebssystem als eine Hierarchie von Schichten zu organisieren, von denen jede auf der darunterliegenden Schicht aufbaut. Das erste System, das auf diese Weise konstruiert wurde, war das THE-System, das an der Technischen Hochschule Eindhoven in den Niederlanden von E. W. Dijkstra (1968) und seinen Studenten entwickelt wurde. Das THE-System war ein einfaches Stapelverarbeitungssystem für einen holländischen Computer, die Electrologica X8, die 32 K 27-Bit-Wörter hatte (Bits waren damals sehr teuer).

Das System umfasste sechs Schichten, so wie es in ▶*Abbildung 1.25* dargestellt ist. Schicht 0 kümmerte sich um die Zuteilung des Prozessors und das Umschalten zwischen Prozessen beim Auftreten von Unterbrechungen oder beim Ablaufen von Timern. Oberhalb von Schicht 0 bestand das System aus sequenziellen Prozessen, die so programmiert werden konnten, dass sie sich nicht darum kümmern mussten, wenn mehrere Prozesse auf einem einzigen Prozessor ausgeführt wurden. Mit anderen Worten: Schicht 0 legte die Basis für die Multiprogrammierung der CPU.

Schicht	Funktion
5	Der Operator
4	Benutzerprogramme
3	Ein-/Ausgabeverwaltung
2	Operator-Prozess-Kommunikation
1	Speicherverwaltung
0	Prozessorzuteilung und Multiprogrammierung

Abbildung 1.25: Die Struktur des THE-Betriebssystems.

Schicht 1 übernahm die Speicherverwaltung. Sie belegte den Platz für die Prozesse im Arbeitsspeicher und in einem 512-K-Trommelspeicher, der zur Speicherung von Teilen der Prozesse (Seiten) benutzt wurde, für die kein Platz im Arbeitsspeicher war. Oberhalb von Schicht 1 brauchten sich die Prozesse nicht darum zu kümmern, ob sie im Arbeitsspeicher oder im Trommelspeicher abgelegt waren. Die Software der Schicht 1 stellte sicher, dass die Seiten in den Arbeitsspeicher transportiert wurden, sobald sie benötigt wurden, und wieder entfernt, wenn sie nicht mehr gebraucht wurden.

Schicht 2 behandelte die Kommunikation zwischen den Prozessen einerseits und der Bedienkonsole (d.h. dem Benutzer) andererseits. Oberhalb dieser Schicht hatte jeder Prozess seine eigene Bedienkonsole. Schicht 3 übernahm die Aufgabe, die Ein-/Ausgabegeräte zu verwalten und die Informationsströme von und zu diesen zu puffern. Oberhalb von Schicht 3 konnte jeder Prozess mit abstrakten Ein-/Ausgabegeräten arbeiten, die einfacher zu benutzen sind als die echten Geräte mit ihren vielen Eigenarten. In Schicht 4 waren die Benutzerprogramme zu finden. Sie brauchten sich nicht um die Verwaltung der Prozesse, des Speichers, der Konsole oder die Ein-/Ausgabeverwaltung zu kümmern. Die Systemverwaltung war in Schicht 5 angesiedelt.

Eine weitere Verallgemeinerung dieses Schichtenkonzepts war im MULTICS-System zu finden. Anstelle von Schichten war MULTICS in einer Folge konzentrischer Ringe organisiert, von denen die inneren privilegierter waren als die äußeren (was praktisch auf dasselbe hinausläuft). Wenn eine Prozedur in einem äußeren Ring eine Prozedur in einem inneren Ring aufrufen wollte, musste sie die Entsprechung zu einem Systemaufruf durchführen, das heißt einen `TRAP`-Befehl, dessen Parameter sorgfältig auf ihre Gültigkeit überprüft wurden, bevor der Aufruf fortgesetzt werden durfte. Obwohl in MULTICS das gesamte Betriebssystem Teil des Adressraums eines jeden Benutzerprozesses war, machte es die Hardware möglich, einzelne Prozeduren (konkret Speichersegmente) als geschützt gegen Lesen, Schreiben oder Ausführen zu kennzeichnen.

War die Schichtenbildung in THE eigentlich nur eine Entwurfshilfe, weil alle Teile des Systems letztlich zu einem ausführbaren Programm verbunden wurden, so stand der Ringmechanismus in MULTICS auch noch zur Laufzeit mit Unterstützung durch die Hardware zur Verfügung. Der Vorteil des Ringmechanismus bestand darin, dass er

leicht dahingehend erweitert werden konnte, Benutzeruntersysteme zu strukturieren. So könnte zum Beispiel ein Professor ein Programm zum Testen und Benoten von studentischen Programmen schreiben und dieses dann in Ring *n* laufen lassen. Die studentischen Programme würden im Ring *n* + 1 ausgeführt und konnten trotzdem niemals ihre Noten verändern.

1.7.3 Mikrokerne

Bei dem Schichtenmodell können die Entwickler wählen, wo die Grenze zwischen Kern- und Benutzermodus gezogen werden soll. Traditionell waren alle Schichten im Kern, aber das ist nicht unbedingt notwendig. Vielmehr gibt es starke Argumente dafür, so wenig wie möglich im Kernmodus auszuführen, weil Fehler im Kern das System auf der Stelle zu Fall bringen können. Benutzerprozesse können dagegen so eingerichtet werden, dass ein Fehler keine besonders große Wirkung hat.

Verschiedene Wissenschaftler haben immer wieder die Anzahl der Fehler pro 1000 Codezeilen analysiert (z.B. Basilli und Perricone, 1984; Ostrand und Weyuker, 2002). Die Fehlerdichte hängt von der Modulgröße, dem Modulalter und weiteren Parametern ab, aber eine Richtzahl für solide industrielle Systeme ist zwischen zwei und zehn Fehler pro 1000 Codezeilen. Das heißt, dass ein monolithisches Betriebssystem mit 5 Millionen Codezeilen wahrscheinlich 10000 bis 50000 Kernfehler enthält. Natürlich sind nicht alle schwerwiegend, einige Fehler betreffen möglicherweise nur das Übermitteln von inkorrekten Fehlernachrichten in Situationen, die selten auftreten. Dennoch sind Betriebssysteme genügend fehlerbehaftet, dass Computerhersteller einen Reset-Knopf auf ihnen angebracht haben (oft an der Frontseite) – etwas, was die Hersteller von Fernsehgeräten, Stereoanlagen und Autos nicht tun, trotz der großen Menge an Software in diesen Geräten.

Die Grundidee des Mikrokernentwurfes ist es, eine hohe Ausfallsicherheit zu erreichen, indem das Betriebssystem in kleine, wohldefinierte Module aufgespalten wird, von denen nur eines – nämlich der Mikrokern – im Kernmodus ausgeführt wird, während der Rest als relativ wirkungsarmer gewöhnlicher Benutzerprozess läuft. Insbesondere kann durch die Ausführung jedes Gerätetreibers und jedes Dateisystems als jeweils ein separater Benutzerprozess ein Fehler in einem von diesen zwar die jeweilige Komponente zum Absturz bringen, aber eben nicht das gesamte System. So kann beispielsweise ein Fehler im Audiotreiber einen verstümmelten oder unterbrochenen Sound verursachen, aber er kann nicht den Computer abstürzen lassen. In einem monolithischen System dagegen, bei dem alle Treiber im Kern laufen, kann ein fehlerhafter Audiotreiber leicht eine ungültige Speicheradresse referenzieren und das System auf der Stelle zum völligen Stillstand bringen.

Es werden seit Jahrzehnten viele Mikrokerne implementiert und eingesetzt (Haertig et al., 1997; Heiser et al., 2006; Herder et al., 2006; Hildebrand, 1992; Kirsch et al., 2005; Liedtke, 1993, 1995, 1996; Pike et al., 1992; Zuberi et al., 1999). Mit Ausnahme von OS X, das auf dem Mach-Mikrokern (Accetta et al., 1986) basiert, werden Mikrokerne in den gängigen Desktop-Betriebssystemen nicht eingesetzt. Mikrokerne sind in Echt-

zeit-, industriellen, avionischen und militärischen Anwendungen vorherrschend, die auftragsentscheidend sind und sehr hohe Anforderungen an die Betriebssicherheit haben. Zu den besser bekannten Mikrokernen zählen Integrity, K42, L4, PikeOS, QNX, Symbian und MINIX 3. Wir geben jetzt einen kurzen Überblick über MINIX 3, bei dem das Konzept der Modularität komplett ausgereizt wurde, indem der größte Teil des Betriebssystems in eine Reihe unabhängiger Benutzermodusprozesse aufgebrochen wurde. MINIX 3 ist vollständig POSIX-konform und als Open-Source-System frei erhältlich unter *www.minix3.org* (Giuffrida et al., 2012; Giuffrida et al., 2013; Herder et al., 2006; Herder et al., 2009; Hruby et al., 2013).

Der Mikrokern von MINIX 3 hat nur ca. 12000 Zeilen C-Code und 1400 Zeilen Assemblercode für viele grundlegende Funktionen wie das Abfangen von Interrupts und das Wechseln von Prozessen. Der C-Code verwaltet und teilt die Prozesse ein, organisiert die Interprozesskommunikation (durch das Austauschen von Nachrichten zwischen Prozessen) und stellt ungefähr 40 Kernaufrufe zur Verfügung, die es dem Rest des Betriebssystems ermöglichen, ordentlich zu funktionieren. Diese Aufrufe führen Funktionen aus, wie zum Beispiel Behandlungsroutinen in Interrupts einzuhaken, Daten zwischen Adressräumen zu verschieben und Speicherabbildungen für neu erzeugte Prozesse zu installieren. Die Prozessstruktur von MINIX 3 ist in ▶ *Abbildung 1.26* zu sehen, wobei die Behandlungsroutine für die Kernaufrufe mit *Sys* beschriftet ist. Der Gerätetreiber für die Uhr befindet sich ebenfalls im Kern, da der Scheduler sehr eng mit ihm interagiert. Alle anderen Gerätetreiber laufen als separate Benutzerprozesse.

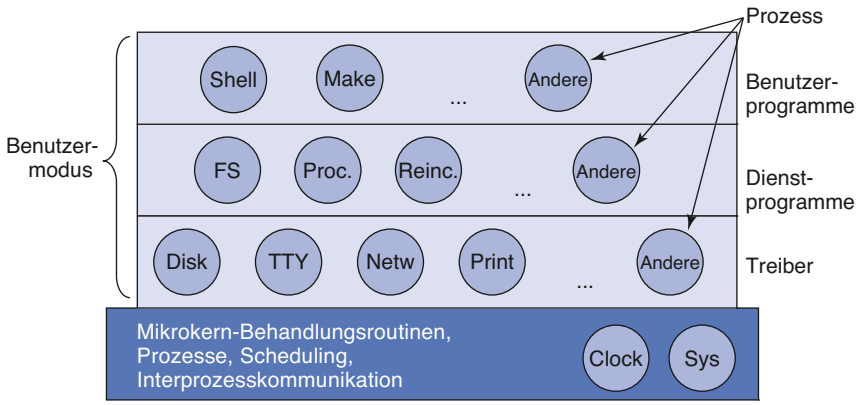

Abbildung 1.26: Vereinfachte Struktur des MINIX-Systems.

Außerhalb des Kerns ist das System in drei Schichten strukturiert, alle Prozesse dieser Schichten laufen im Benutzermodus. Die unterste Schicht enthält die Gerätetreiber. Da diese im Benutzermodus laufen, haben sie keine physische Adresse im Ein-/Ausgabeport-Namensraum und können Kommandos nicht direkt ausgeben. Um ein Ein-/Ausgabegerät zu programmieren, bildet der Treiber stattdessen eine Struktur, indem er mitteilt, welche Werte auf welchen Ein-/Ausgabeport geschrieben werden sollen. Dann führt er einen Kernaufruf aus, um das Schreiben durch den Kern auszulösen.

Durch dieses Vorgehen hat der Kern die volle Kontrolle darüber, ob der Treiber, der gerade schreibt (oder liest), dazu auch berechtigt ist. Folglich kann ein fehlerhafter Audiotreiber (anders als beim monolithischen Entwurf) nicht versehentlich etwas auf die Platte schreiben.

Oberhalb der Treiber gibt es eine weitere Schicht im Benutzermodus. Hier befinden sich die Dienstprogramme (*server*), die einen Großteil der Aufgaben eines Betriebssystems erledigen. Ein oder mehrere Dateiserver verwalten das Dateisystem (bzw. die Dateisysteme), der Prozessverwalter erzeugt, löscht und verwaltet Prozesse usw. Benutzerprogramme können die Betriebssystemdienste durch das Senden von kurzen Nachrichten an diese Dienstprogramme abrufen, indem sie die POSIX-Systemaufrufe anfordern. Zum Beispiel schickt ein Prozess, der einen `read`-Aufruf benötigt, eine Nachricht zu einem der Dateiserver und teilt ihm mit, was zu lesen ist.

Ein interessantes Dienstprogramm ist der **Reincarnation-Server**, dessen Aufgabe es ist zu überprüfen, ob die anderen Dienstprogramme und Treiber korrekt funktionieren. Falls ein defektes Element entdeckt wird, ersetzt es der Reincarnation-Server automatisch ohne irgendeine Benutzerintervention. Auf diese Art kann sich das System selbst regulieren und erzielt so eine hohe Zuverlässigkeit.

Das System bietet viele Kontrollmechanismen, um die Befugnisse jedes Prozesses zu begrenzen. Wie erwähnt können Treiber nur autorisiert auf Ein-/Ausgabeports zugreifen, aber auch die Benutzung der Kernaufrufe wird prozessweise gesteuert, genauso wie das Senden von Nachrichten zu anderen Prozessen. Prozesse können wiederum anderen Prozessen die begrenzte Erlaubnis erteilen, dass der Kern auf ihren Adressraum zugreifen kann. Beispielsweise erlaubt es ein Dateisystem dem Plattentreiber, dass der Kern einen neu eingelesenen Plattenblock an einer speziellen Adresse innerhalb des Dateisystemadressraums ablegt. Die Gesamtheit all dieser Kontrollmechanismen bewirkt, dass jeder Treiber und jedes Dienstprogramm genau die Berechtigungen hat, um seine Aufgaben zu erledigen, und nichts darüber hinaus. Damit lässt sich der Schaden, den eine fehlerhafte Komponente anrichten kann, außerordentlich begrenzen.

Ein Konzept, das ein wenig mit der Idee des minimalen Kerns verwandt ist, bringt den **Mechanismus**, wie etwas gemacht wird, im Kern unter, nicht aber die **Strategie** (*policy*). Um diesen Ansatz noch verständlicher zu machen, betrachten wir kurz das Scheduling von Prozessen: Eine relativ einfache Schedulingstrategie ist es, jedem Prozess eine numerische Priorität zuzuweisen und dann den Kern den (ablauffähigen) Prozess mit der höchsten Priorität ausführen zu lassen. Der Mechanismus – im Kern – umfasst das Suchen nach dem Prozess mit der höchsten Priorität und dessen Ausführung. Die Strategie – den Prozessen Prioritäten zuzuweisen – kann von einem Prozess im Benutzermodus übernommen werden. So können Strategie und Mechanismus entkoppelt werden und der Kern lässt sich klein halten.

1.7.4 Das Client-Server-Modell

Eine leichte Variation der Mikrokern-Idee ist die Einteilung der Prozesse in zwei Klassen: die **Server**, von denen jeder einige Dienste zur Verfügung stellt, und die **Clients**, die diese Dienste nutzen. Dieses Modell ist als **Client-Server**-Modell bekannt. Oft ist die unterste Schicht ein Mikrokern, aber das ist keine notwendige Bedingung. Das Wesentliche ist das Vorhandensein von Client- und Server-Prozessen.

Kommunikation zwischen Clients und Servern geschieht oft durch Nachrichtenaustausch (*message passing*). Um einen Dienst zu erhalten, erstellt ein Client-Prozess eine Nachricht mit seinen Anforderungen und sendet diese zu dem passenden Dienst. Dieser Dienst erledigt den Auftrag und schickt eine Antwort zurück. Wenn Client und Server zufällig auf derselben Maschine laufen, sind noch gewisse Verbesserungen möglich, aber wir sprechen hier dennoch über Nachrichtenaustausch.

Eine offensichtliche Verallgemeinerung dieser Idee besteht darin, Clients und Server auf unterschiedlichen Computern auszuführen, die durch ein lokales oder ein Fernnetz miteinander verbunden sind, wie in ▶*Abbildung 1.27* dargestellt. Da die Clients mit Servern durch das Senden von Nachrichten kommunizieren, müssen sie nicht wissen, ob die Nachrichten lokal auf ihrer eigenen Maschine bearbeitet werden oder ob sie über ein Netzwerk zu Servern auf entfernten Maschinen geschickt werden. Soweit es den Client betrifft, passiert schließlich in beiden Fällen dasselbe: Anforderungen werden verschickt und Antworten kommen zurück. Somit ist das Client-Server-Modell eine Abstraktion, die sowohl für eine einzelne Maschine als auch für ein Netzwerk von Maschinen genutzt werden kann.

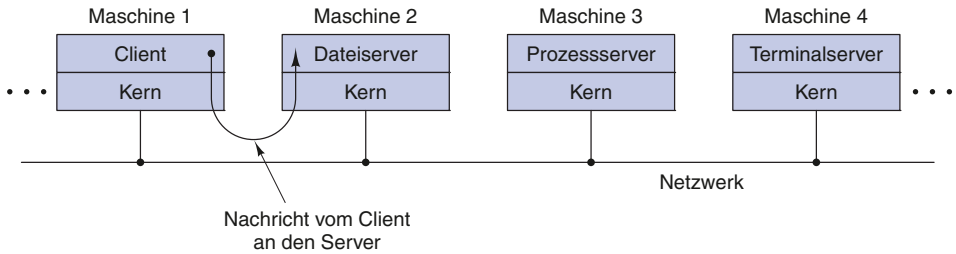

Abbildung 1.27: Das Client-Server-Modell über einem Netzwerk.

Immer mehr Systeme beziehen PC-Benutzer als Clients ein, wobei große Maschinen andernorts als Server fungieren. Genau genommen funktioniert ein Großteil des Webs so. Ein PC sendet eine Anfrage für eine Webseite zu einem Server und die Webseite kommt als Antwort zurück. Dies ist eine typische Nutzung des Client-Server-Modells in einem Netzwerk.

1.7.5 Virtuelle Maschinen

Die erste Version von OS/360 war ein reines Stapelverarbeitungssystem. Doch viele Benutzer der 360-Systeme wollten interaktiv an einem Terminal arbeiten. Deshalb begannen verschiedene Programmierteams innerhalb wie außerhalb von IBM, Time-

sharing-Systeme für OS/360 zu entwickeln. Das offizielle Timesharing-System von IBM, TSS/360, wurde erst spät ausgeliefert und als es schließlich zum Einsatz kam, war es so groß und langsam, dass kaum darauf umgestellt wurde. TSS/360 wurde schließlich aufgegeben, nachdem seine Entwicklung ungefähr 50 Millionen US-Dollar gekostet hatte (Graham, 1970). Aber eine andere Gruppe am Scientific Center von IBM in Cambridge, Massachusetts, brachte ein grundlegend anderes System heraus, das IBM letzten Endes als Produkt akzeptierte. Ein direkter Nachfolger davon ist **z/VM**, das heute auf den aktuellen Großrechnern von IBM, den zSeries, weit verbreitet ist. Die IBM zSeries werden viel in großen Datenzentren von Unternehmen genutzt, zum Beispiel als Server im Bereich des E-Commerce, wo viele Tausend Transaktionen pro Sekunde anfallen und Datenbanken eingesetzt werden, die bis zu Millionen von Gigabyte groß sind.

VM/370

Dieses System, das ursprünglich CP/CMS genannt wurde und nun als VM/370 bezeichnet wird (Seawright und MacKinnon, 1979), basierte auf einer scharfsinnigen Beobachtung: Ein Timesharing-System stellt erstens Mehrprogrammbetrieb und zweitens eine erweiterte Maschine mit einer praktischeren Schnittstelle als die reine Hardware zur Verfügung. Der Grundgedanke beim VM/370 ist es, beide Funktionen vollständig voneinander zu trennen.

Das Herz des Systems, das auch als **Virtual Machine Monitor** bekannt ist, kommt auf der blanken Hardware zur Ausführung und stellt die Multiprogrammierung zur Verfügung, indem nicht eine, sondern mehrere virtuelle Maschinen auf der nächsthöheren Schicht bereitgestellt werden (▶ *Abbildung 1.28*). Im Unterschied zu allen anderen Betriebssystemen sind diese virtuellen Maschinen aber keine erweiterten Maschinen mit Dateien oder anderen netten Eigenschaften. Stattdessen sind sie *exakte* Kopien der blanken Hardware einschließlich Kern- und Benutzermodus, Ein-/Ausgabe, Interrupts und allen anderen Eigenschaften der realen Maschine.

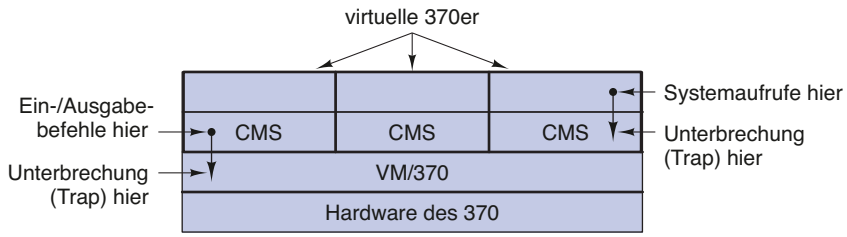

Abbildung 1.28: Die Struktur des VM/370-Systems mit CMS.

Da jede virtuelle Maschine identisch mit der zugrunde liegenden Hardware ist, kann auf jeder Maschine jedes Betriebssystem arbeiten, das auch unmittelbar auf der Hardware läuft. Auf verschiedenen virtuellen Maschinen können somit unterschiedliche Betriebssysteme arbeiten, was auch häufig vorkommt. Auf den virtuellen Maschinen des ursprünglichen VM/370-Systems von IBM lief OS/360 oder eines der anderen großen Stapel- oder Dialogverarbeitungssysteme, während auf anderen das interaktive

Einbenutzersystem **CMS** (**Conversational Monitor System**) im Mehrprogrammbetrieb lief. Letzteres war unter Programmierern sehr beliebt.

Wenn das CMS-Programm einen Systemaufruf ausführte, wurde der Aufruf an das Betriebssystem der eigenen virtuellen Maschine (via Trap) und nicht an das VM/370 weitergeleitet – gerade so, als würde die reale Maschine anstelle der virtuellen Maschine benutzt. CMS verwendete dann die normalen Ein-/Ausgabebefehle der Hardware zum Lesen seiner virtuellen Platte oder was sonst durch den Systemaufruf veranlasst wurde. Diese Ein-/Ausgabebefehle wurden vom VM/370 abgefangen, welches die Befehle dann als Bestandteil seiner Simulation der realen Hardware ausführte. Durch die vollständige Separation der Funktionen der Multiprogrammierung und der Bereitstellung einer erweiterten Maschine wurde jeder der Teile deutlich einfacher, wesentlich flexibler und leichter zu warten.

In seiner modernen Ausführung wird z/VM in der Regel eingesetzt, um mehrere vollständige Betriebssysteme anstelle von reduzierten Einbenutzersystemen wie CMS laufen zu lassen. Zum Beispiel können auf der zSeries ein oder mehrere virtuelle Linux-Maschinen zusammen mit traditionellen IBM-Betriebssystemen ausgeführt werden.

Wiederentdeckt: die virtuellen Maschinen

Während IBM seit vier Jahrzehnten virtuelle Maschinen auf seinen Produkten einsetzt und einige Firmen, unter ihnen Oracle und Hewlett-Packard, unlängst ihre High-End-Unternehmensserver mit virtuellen Maschinen ausgestattet haben, wurde bis vor Kurzem das Konzept der Virtualisierung im Bereich der PCs weitgehend ignoriert. In den letzten Jahren jedoch entwickelte es sich durch eine Kombination aus neuen Bedürfnissen, neuer Software und neuer Technologie wieder zu einem viel diskutierten Thema.

Zunächst zu den Bedürfnissen: Viele Unternehmen lassen traditionell ihre Mailserver, Webserver, FTP-Server und andere Server auf separaten Computern laufen, manchmal mit unterschiedlichen Betriebssystemen. Hier bietet sich Virtualisierung als eine Möglichkeit an, all diese Server auf der gleichen Maschine laufen zu lassen, ohne dass der Absturz eines Servers alle anderen ebenfalls zu Fall bringt.

Virtualisierung ist auch in der Welt des Webhostings verbreitet. Ohne Virtualisierung müssen die Webhosting-Kunden wählen zwischen **Shared Hosting** (welches nur einen Login-Zugang auf einem Webserver zuweist, aber keine Kontrolle über die Software des Servers) und **Dedicated Hosting** (welches jedem Kunden seine eigenen Maschinen zuweist, was zwar sehr flexibel, aber für kleine bis mittelgroße Websites nicht kosteneffizient ist). Wenn ein Webhosting-Unternehmen virtuelle Maschinen zur Miete anbietet, können auf einer einzigen physischen Maschine viele virtuelle Maschinen laufen, wobei jede wie eine vollständige Maschine erscheint. Kunden, die solch eine virtuelle Maschine mieten, können jedes beliebige Betriebssystem und jede Software darauf ausführen, und das zu einem Bruchteil der Kosten eines dedizierten Servers (da dieselbe physische Maschine viele virtuelle Maschinen gleichzeitig unterstützt).

Aber auch Endbenutzer können Virtualisierung gut einsetzen, wenn sie zwei oder mehr Betriebssysteme gleichzeitig laufen lassen wollen, etwa Windows und Linux, weil einige ihrer Lieblingsanwendungen unter dem einen und einige unter dem anderen Betriebssystem laufen. Diese Situation ist in ▶*Abbildung 1.29a* dargestellt, wobei der Ausdruck „Virtual Machine Monitor" in **Typ-1-Hypervisor** umbenannt wurde. Diese Bezeichnung ist jetzt geläufiger, weil „Virtual Machine Monitor" mehr Tastenanschläge benötigt, als man heute zu tippen bereit ist. Beachten Sie, dass viele Autoren die Begriffe dennoch synonym verwenden.

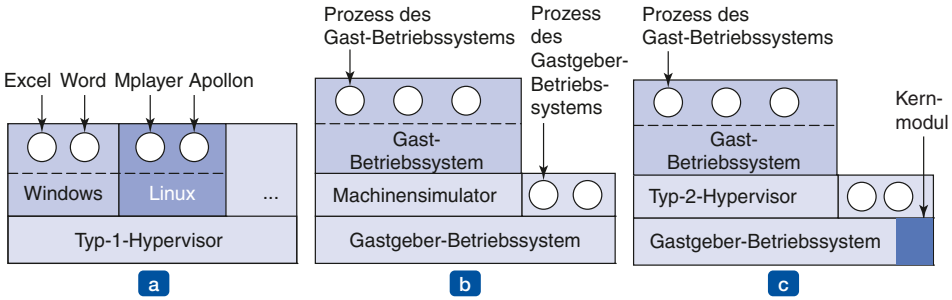

Abbildung 1.29: (a) Typ-1-Hypervisor; (b) reiner Typ-2-Hypervisor; (c) Typ-2-Hypervisor in der Praxis.

Während heute niemand die Attraktivität von virtuellen Maschinen infrage stellt, war das Problem die Implementierung. Um eine virtuelle Maschine auf einem Computer einzusetzen, musste die CPU virtualisierbar sein (Popek und Goldberg, 1974). Kurz gesagt, hier genau liegt das Problem. Wenn ein Betriebssystem auf einer virtuellen Maschine (im Benutzermodus) einen privilegierten Befehl ausführen will, wie z.B. das Verändern des Programmstatusworts oder die Durchführung von Ein-/Ausgabe, dann muss die Hardware den Virtual Machine Monitor aufrufen, damit der Befehl durch die Software nachgebildet werden kann. Auf einigen CPUs – namentlich dem Pentium, seinen Vorgängern und Klonen – werden solche Versuche, privilegierte Befehle im Benutzermodus auszuführen, einfach ignoriert. Dadurch ist es unmöglich, virtuelle Maschinen mit dieser Hardware zu kombinieren – was das mangelnde Interesse in der x86-Welt erklärt. Natürlich gab es Interpreter für den Pentium wie *Bochs*, diese brachten aber normalerweise einen Performanzverlust um ein bis zwei Größenordnungen mit sich und waren damit für ernsthaftes Arbeiten nicht geeignet.

In den 1990er Jahren und den ersten Jahren dieses Jahrhunderts änderte sich diese Situation durch eine Reihe von akademischen Forschungsprojekten, insbesondere Disco an der Universität von Stanford (Bugnion et al., 1997) und Xen an der Universität von Cambridge (Barham et al., 2003). Diese Forschungsarbeiten mündeten in mehreren kommerziellen Softwareprodukten (z.B. VMware Workstation und Xen) und ließen das Interesse an virtuellen Maschinen wiederaufleben. Neben VMware und Xen sind heute KVM (für den Linux-Kern), VirtualBox (von Oracle) und Hyper-V (von Microsoft) bekannte Hypervisors.

Einige dieser frühen Forschungsprojekte verbesserten die Leistung über Interpretern wie *Bochs*, indem Codeblöcke direkt übersetzt, in einem internen Cache gespeichert

und dann bei erneuter Ausführung wiederverwendet wurden. Diese deutliche Performanzsteigerung führte zu dem, was wir **Maschinensimulator** nennen wollen (▶ *Abbildung 1.29b*). Obwohl diese Technik, die als **binäre Übersetzung** (*binary translation*) bekannt ist, eine Verbesserung darstellte, waren die resultierenden Systeme zwar gut genug, um auf akademischen Konferenzen vorgestellt zu werden, aber nicht schnell genug, um sie in kommerziellen Umgebungen einzusetzen, in denen Geschwindigkeit entscheidend ist.

Der nächste Schritt in Richtung Leistungssteigerung war das Hinzufügen eines Kernmoduls, das einen Teil der Arbeit übernimmt, wie in ▶ *Abbildung 1.29c* zu sehen ist. In der Praxis verwenden heutzutage alle kommerziell verfügbaren Hypervisors wie VMware Workstation diese hybride Strategie (zusätzlich zu vielen weiteren Verbesserungen). Diese Systeme werden allgemein als **Typ-2-Hypervisor** bezeichnet. Dem schließen wir uns (ein wenig widerwillig) an und werden diesen Namen im Rest des Buchs benutzen, auch wenn wir die Bezeichnung „Typ-1.7-Hypervisor“ vorziehen würden, um dem Umstand Rechnung zu tragen, dass diese Programme nicht gänzlich im Benutzermodus laufen. In *Kapitel 7* werden wir ausführlich beschreiben, wie VMware Workstation arbeitet und was die einzelnen Teile tun.

Der wirkliche Unterschied zwischen einem Typ-1-Hypervisor und einem Typ-2-Hypervisor besteht in der Praxis darin, dass ein Typ-2-Hypervisor auf einem **Gastgeber-Betriebssystem** (*host operating system*) aufsetzt und dessen Dateisystem verwendet, um Prozesse zu erzeugen, Dateien zu speichern und so weiter. Ein Typ-1-Hypervisor besitzt solch ein zugrunde liegendes System nicht und muss all diese Funktionen selbst durchführen.

Nach dem Start des Typ-2-Hypervisors liest dieser die Installations-CD (oder das Speicherabbild der CD-ROM) für das gewählte **Gast-Betriebssystem** (*guest operating system*) und installiert es auf einer virtuellen Platte, die nichts anderes ist als eine große Datei im Dateisystem des Gastgeber-Betriebssystems. Typ-1-Hypervisors können so nicht vorgehen, weil es kein Gastgeber-Betriebssystem gibt, auf dem Dateien gespeichert werden könnten. Daher müssen sie ihren eigenen Speicher auf einer rohen Plattenpartition verwalten.

Wenn nun das Gast-Betriebssystem hochgefahren wird, verhält es sich so wie auf der echten Hardware. Es startet also in der Regel ein paar Hintergrundprogramme und dann eine GUI. Für den Anwender/Aus Sicht des Anwenders verhält sich das Gast-Betriebssystem genauso, als liefe es auf echter Hardware, auch wenn das hier nicht der Fall ist.

Bei einem anderen Ansatz wird das Betriebssystem dahingehend verändert, dass die Steuerbefehle entfernt werden. Dieses Vorgehen ist keine echte Virtualisierung, sondern eine **Paravirtualisierung**. Auf diese und andere Aspekte der Virtualisierung werden wir in *Kapitel 8* noch genauer eingehen.

Die Java Virtual Machine (JVM)

Ein anderer Bereich, in dem virtuelle Maschinen zum Einsatz kommen, allerdings in etwas anderer Form, ist die Ausführung von Java-Programmen. Als Sun Microsystems die Programmiersprache Java erfand, wurde auch eine virtuelle Maschine (d.h. eine Computerarchitektur) dafür entwickelt, die sogenannte **JVM** (**Java Virtual Machine**). Der Java-Compiler erzeugt Code für die JVM und dieser Code wird dann typischerweise von einem JVM-Interpreter ausgeführt. Der Vorteil dieser Architektur ist, dass der JVM-Code über das Internet übertragen und anschließend auf jedem Computer mit einem JVM-Interpreter ausgeführt werden kann. Wenn der Compiler stattdessen Code für die SPARC- oder x86-Architektur übersetzt hätte, könnte der Code nicht so einfach übertragen und ausgeführt werden. (Natürlich hätte Sun einen Compiler entwickeln können, der für SPARC übersetzt, und dann einen SPARC-Interpreter verteilen können, aber die JVM ist eine viel einfachere Architektur zur Interpretation.) Ein weiterer Vorteil der JVM ist, dass bei korrekter Implementierung des Interpreters (was nicht ganz einfach ist) ablaufende Programme auf Sicherheitseigenschaften überprüft werden können. Dadurch können sie in einer sicheren Umgebung keine Daten stehlen oder irgendeinen Schaden anrichten.

1.7.6 Exokerne

Anstatt die eigentliche Maschine nachzubilden, wie es bei den virtuellen Maschinen geschieht, besteht eine andere Strategie darin, die Maschine zu teilen: Jedem Benutzer wird dabei eine Teilmenge der Betriebsmittel zugeteilt. Auf diese Art bekommt eine virtuelle Maschine zum Beispiel die Blöcke der Festplatte von 0 bis 1023, die nächste die Blöcke von 1024 bis 2047 und so weiter.

Auf der untersten Ebene im Kernmodus läuft ein Programm, das man **Exokern** nennt (Engler et al., 1995). Seine Aufgabe ist es, Ressourcen für die einzelnen virtuellen Maschinen zu belegen und sicherzustellen, dass keine Maschine die Ressourcen von jemand anderem benutzt. Jede virtuelle Maschine im Benutzermodus kann ihr eigenes Betriebssystem verwenden, wie auf einem VM/370-System und dem Virtual-8086-Modus bei einem Pentium. Der Unterschied ist, dass jede virtuelle Maschine nur die Ressourcen verwenden kann, die sie verlangt und belegt hat.

Der Vorteil des Exokern-Konzepts ist die Einsparung einer Zwischenschicht. Bei anderen Modellen hat jede Maschine ihre eigene Festplatte mit den Blöcken von 0 bis zum Maximum, sodass der Virtual Machine Monitor Tabellen verwalten muss, um die Plattenadressen (und alle anderen Ressourcen) zuzuordnen. Beim Exokern ist diese Zuordnung nicht nötig. Der Exokern muss sich nur merken, welcher virtuellen Maschine welches Betriebsmittel zugeteilt wurde. Diese Methode hat den Vorteil, dass die Multiprogrammierung (im Exokern) vom Benutzerbetriebssystem (im Benutzermodus) getrennt ist. Dafür ist aber weniger Verwaltungsaufwand nötig: Die einzige Aufgabe des Exokerns ist es, die virtuellen Maschinen davon abzuhalten, sich in die Haare zu kriegen.

1.8 Die Welt aus der Sicht von C

Betriebssysteme sind normalerweise große C-Programme (oder manchmal C++-Programme), die aus vielen Teilen bestehen und von vielen Programmierern geschrieben werden. Die Entwicklungsumgebung für Betriebssysteme ist komplett anders als das, was manche (etwa Studenten) vom Schreiben kleiner Java-Programme kennen. Dieser Abschnitt ist ein Versuch, eine sehr kurze Einführung in die Welt der Programmierung von Betriebssystemen für Java- oder Python-Programmierer zu geben.

1.8.1 Die Programmiersprache C

Wir wollen hier keine Einführung in C geben, sondern nur einige der Hauptunterschiede zwischen C und Sprachen wie **Python** und speziell Java kurz darstellen. Da Java auf C basiert, gibt es natürlich viele Gemeinsamkeiten. Python ist etwas unterschiedlich, aber hat immer noch viele Gemeinsamkeiten. Der Einfachheit halber konzentrieren wir uns auf Java. Java, Python und C sind imperative Sprachen mit Datentypen, Variablen und Steueranweisungen. Die elementaren Datentypen in C sind ganze Zahlen (*integer*, einschließlich *short integer* und *long integer*), Zeichen (*character*) und Gleitkommazahlen (*floating-point number*). Zusammengesetzte Datentypen können durch die Benutzung von Feldern (*array*), Strukturen (*structure*) und Variantenrecords (*union*) konstruiert werden. Die Steueranweisungen in C sind denen in Java recht ähnlich, einschließlich der if-, switch-, for- und while-Anweisungen. Funktionen und Parameter sind in beiden Sprachen in etwa dasselbe.

Ein Konzept in C, das Java nicht kennt, sind die expliziten Zeiger. Ein **Zeiger** (*pointer*) ist eine Variable, die auf eine andere Variable oder Datenstruktur verweist (d.h., sie enthält deren Adresse). Betrachten wir als Beispiel die folgenden Anweisungen:

```
char c1, c2, *p;
c1 = 'c';
p = &c1;
c2 = *p;
```

Hier werden *c1* und *c2* als Zeichenvariablen deklariert und *p* als eine Variable, die auf ein Zeichen zeigt (d.h., die Adresse eines Zeichens enthält). Die erste Zuweisung speichert den ASCII-Code für das Zeichen „c“ in der Variablen *c1* ab. Die zweite weist die Adresse von *c1* der Zeigervariablen *p* zu und die dritte den Inhalt der Variable, auf die *p* zeigt, der Variablen *c2*. Nachdem alle Anweisungen ausgeführt sind, enthält also *c2* ebenfalls den ASCII-Code für „c“. Theoretisch sind Zeiger typisiert, d.h., ein Zeiger besitzt durchaus auch einen ihm zugeordneten Datentyp. Man sollte beispielsweise daher die Adresse einer Gleitkommazahl nicht einem Zeichenzeiger zuweisen können. Doch in der Praxis akzeptieren Compiler solche Zuweisungen, manchmal jedoch mit einer Warnung. Zeiger sind ein sehr mächtiges Konstrukt, stellen aber auch eine große Fehlerquelle dar, vor allem wenn sie nachlässig eingesetzt werden.

Zu dem, was es in C nicht gibt, zählen Datentypen für Zeichenketten bzw. Strings, Threads, Packages, Klassen, Objekte, Typsicherheit und automatische Speicherbereinigung (*garbage collection*). Letztere wäre ein K.O.-Kriterium für Betriebssysteme. Die

gesamte Speicherung in C ist entweder statisch oder der Speicher wird explizit vom Programmierer belegt und wieder freigegeben, in der Regel mit den Bibliotheksfunktionen *malloc* und *free*. Diese Eigenschaft – totale Kontrolle des Programmierers über den Speicher – ist es, die zusammen mit dem Konzept der expliziten Zeiger C so attraktiv für die Programmierung von Betriebssystemen macht. Betriebssysteme sind bis zu einem gewissen Grad im Prinzip Echtzeitsysteme, eigentlich sogar Allzwecksysteme. Wenn ein Interrupt auftritt, hat das Betriebssystem möglicherweise nur ein paar Mikrosekunden Zeit, um noch eine Aktion durchzuführen, bevor kritische Informationen verloren gehen. Eine automatische Speicherbereinigung beispielsweise, die zu einem beliebigen Zeitpunkt losläuft, ist in Betriebssystemen völlig unbrauchbar.

1.8.2 Header-Dateien

Ein Betriebssystemprojekt besteht im Allgemeinen aus einer Reihe von Verzeichnissen mit vielen *.c*-Dateien, die den Code für einige Systemteile beschreiben, und mit einigen *.h*-Header-Dateien, die Deklarationen und Definitionen für eine oder mehrere Codedateien enthalten. Header-Dateien können auch einfache **Makros** einschließen, wie

```
#define BUFFER_SIZE 4096
```

die es dem Programmierer erlauben, Konstanten zu benennen. So wird *BUFFER_SIZE*, wenn es in einem Code auftritt, während der Übersetzung durch die Zahl 4096 ersetzt. Als gute C-Programmierpraxis gilt, jede Konstante außer 0, 1 und −1 zu benennen, und manchmal sogar diese. Makros können Parameter haben, zum Beispiel

```
#define max(a, b) (a > b ? a : b)
```

Damit hat die Zuweisung

```
i = max(j, k+1)
```

die gleiche Bedeutung wie

```
i = (j > k+1 ? j : k+1)
```

d.h., der größere Wert von *j* und *k* + 1 wird in *i* gespeichert. Header können auch bedingte Übersetzungen enthalten wie

```
#ifdef X86
  intel_int_ack();
#endif
```

Hier wird nur dann in einen Aufruf der Funktion *intel_int_ack* übersetzt, falls das Makro *X86* definiert ist, andernfalls passiert nichts. Bedingte Übersetzung wird häufig benutzt, um architekturabhängigen Code zu isolieren: Ein bestimmter Code wird nur benutzt, wenn das System auf einem x86 übersetzt wird, ein anderer Code wird verwendet, wenn das System auf einem SPARC übersetzt wird, usw. Eine *.c*-Datei kann keine oder mehrere Header-Dateien einschließen, indem die Direktive *#include* benutzt wird. Es gibt auch viele Header-Dateien, die in fast jeder *.c*-Datei benutzt und in einem zentralen Verzeichnis gespeichert werden.

1.8.3 Große Programmierprojekte

Um ein Betriebssystem zu erstellen, wird jede *.c*-Datei durch den C-Compiler in eine **Objektdatei** übersetzt. Objektdateien, zu erkennen an der Endung *.o*, enthalten binäre Befehle bzw. Maschinencode für die Zielmaschine. Sie werden später direkt von der CPU ausgeführt. So etwas wie einen Java-Bytecode gibt es in der Welt von C nicht.

Der erste Durchlauf des C-Compilers heißt **C-Präprozessor**. Er liest jede *.c*-Datei ein und holt immer, sobald er auf eine *#include*-Direktive trifft, die dort genannte Header-Datei und verarbeitet sie. Außerdem expandiert er Makros, behandelt bedingte Übersetzungen (und einige andere Dinge) und übergibt schließlich die Ergebnisse dem nächsten Durchlauf des Compilers, als ob sie in der ursprünglichen *.c*-Datei enthalten wären.

Da Betriebssysteme sehr groß sind (fünf Millionen Codezeilen sind nicht ungewöhnlich), wäre die vollständige Neuübersetzung nach jeder Änderung in einer Datei untragbar. Auf der anderen Seite verlangt die Veränderung in einer zentralen Header-Datei, die in tausend anderen Dateien eingeschlossen ist, die Neuübersetzung all dieser Dateien. Den Überblick darüber zu behalten, welche Objektdatei von welchen Header-Dateien abhängt, ist ohne Hilfe nicht machbar.

Glücklicherweise können Computer genau das ziemlich gut. Auf UNIX-Systemen gibt es ein Programm namens *make* (mit zahlreichen Varianten wie *gmake*, *pmake* usw.), das den sogenannten *Makefile* einliest. Mit dessen Hilfe kann ermittelt werden, welche Dateien von welchen anderen Dateien abhängen. Die Aufgabe von *make* ist es nun herauszufinden, welche Objektdateien erforderlich sind, um die Binärdatei des Betriebssystems zu erstellen. Dann prüft *make* für jede dieser Dateien, ob eine der Dateien, von denen die Binärdatei abhängt (der Code oder Header), seit der letzten Übersetzung verändert wurde. Falls ja, dann muss diese Objektdatei neu übersetzt werden. Wenn *make* festgelegt hat, welche *.c*-Dateien neu übersetzt werden müssen, ruft es anschließend den C-Compiler auf. Somit reduziert sich die Anzahl der Übersetzungen auf das absolute Minimum. Bei großen Projekten ist das Erzeugen von *Makefile* fehleranfällig, deshalb gibt es Hilfsprogramme, die dies automatisch erledigen.

Wenn alle *.o*-Dateien bereitstehen, werden sie dem **Binder** (*linker*) übergeben, einem Programm, das alle *.o*-Dateien zu einer einzigen ausführbaren Binärdatei kombiniert. Zu diesem Zeitpunkt werden auch alle aufgerufenen Bibliotheksfunktionen hinzugenommen, die Verweise zwischen Funktionen werden aufgelöst und die Maschinenadressen werden nötigenfalls neu berechnet. Sobald der Binder seine Aufgabe beendet hat, ist das Ergebnis ein ausführbares Programm, das in UNIX-Systemen traditionell mit *a.out* bezeichnet wird. Die verschiedenen Phasen dieses Prozesses bei einem Programm mit drei C-Dateien und zwei Header-Dateien sind in ▶*Abbildung 1.30* zu sehen. Auch wenn wir unsere Ausführungen auf Betriebssysteme beschränkt haben, so gilt das Gesagte ebenso für die Entwicklung eines jeden großen Programms.

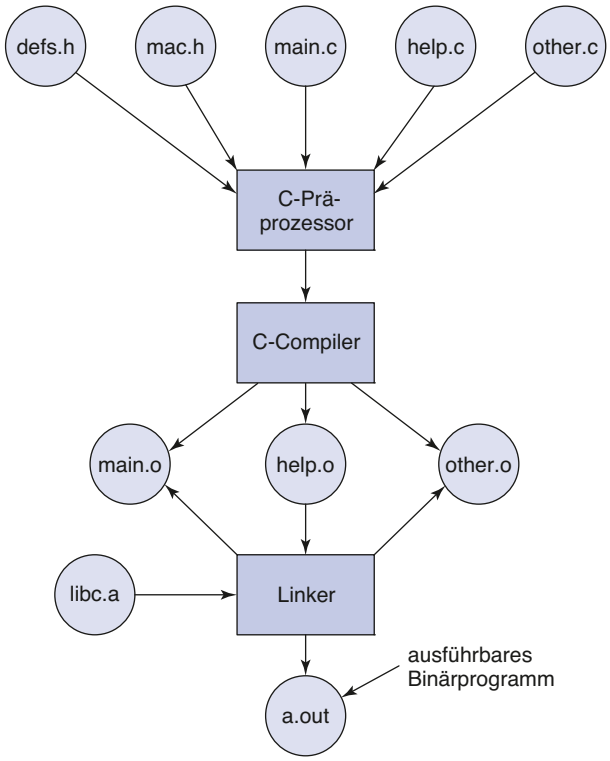

Abbildung 1.30: Der Übersetzungsprozess von C- und Header-Dateien, um eine ausführbare Datei zu erstellen.

1.8.4 Das Laufzeitmodell

Sobald die Binärdatei des Betriebssystems gebunden ist, kann der Computer neu hochgefahren und das neue Betriebssystem gestartet werden. Wenn dies läuft, wird es eventuell Teile laden, die nicht statisch in der Binärdatei enthalten sind, wie Gerätetreiber und Dateisysteme. Zur Laufzeit besteht das Betriebssystem aus mehreren Segmenten für Text (dem Programmcode), für die Daten und für den Stack. Das Textsegment ist normalerweise unveränderbar, es wird während der Ausführung nicht geändert. Das Datensegment hat am Anfang eine bestimmte Größe und ist mit bestimmten Werten initialisiert, aber es kann sich bei Bedarf ändern und wachsen. Der Stack ist anfangs leer, wächst und schrumpft aber mit dem Aufruf von Funktionen und den Rücksprüngen. Oft ist das Textsegment im unteren Bereich des Speichers platziert, das nach oben wachsende Datensegment befindet sich direkt darüber und das nach unten wachsende Stacksegment liegt an einer höheren virtuellen Adresse. Das kann jedoch von System zu System anders sein.

In allen Fällen wird der Code des Betriebssystems direkt von der Hardware ausgeführt, ohne Interpreter und ohne Just-in-time-Übersetzung, wie es bei Java normal ist.

1.9 Forschung im Bereich der Betriebssysteme

Informatik ist eine Wissenschaft, die sich permanent weiterentwickelt, und es ist sehr schwer vorauszusagen, in welche Richtung dies führt. Forscher an Universitäten und an Forschungslaboren in der Industrie denken fortlaufend über neue Entwicklungen nach. Einige davon landen im Papierkorb, andere werden zu Schlüsseltechnologien von zukünftigen Produkten und beeinflussen Industrie wie Benutzer gleichermaßen. Die Spreu vom Weizen zu trennen, ist deshalb so schwer, weil es manchmal 20 bis 30 Jahre dauert, ehe sich eine Idee durchsetzt.

Als beispielsweise Präsident Eisenhower 1958 die zum Verteidigungsministerium gehörende Advanced Research Projects Agency (ARPA) gründete, wollte er die Army daran hindern, über das Forschungsbudget des Pentagon, Navy und Air Force lahmzulegen. Er hatte sicher nicht die Erfindung des Internets im Sinn. Doch zu den Projekten, die die ARPA finanzierte, gehörte eine universitäre Forschung im Bereich des damals obskuren Konzepts eines Paketvermittlungsnetzes, was im ersten experimentellen Paketvermittlungsnetz, dem ARPANET, mündete. Es wurde 1969 ins Leben gerufen. Binnen kürzester Zeit wurden andere von der ARPA finanzierten Forschungsnetzwerke zum ARPANET dazugeschaltet – und das Internet war geboren. Das Internet wurde für etwa 20 Jahre munter von Forschern in der ganzen Welt zum Austausch von E-Mails genutzt. In den frühen 1990er Jahren erfand Tim Berners-Lee das World Wide Web am CERN in Genf und Marc Andreesen schrieb dafür den ersten grafischen Browser an der Universität Illinois. Und mit einem Mal war das Internet voll von twitternden Teenagern. Präsident Eisenhower würde sich angesichts dessen wahrscheinlich im Grabe herumdrehen.

Die Forschung im Bereich Betriebssysteme führte auch zu dramatischen Veränderungen in verwandten Systemen. Wie bereits besprochen, waren die ersten Computer durchweg stapelverarbeitende Systeme, bevor am M.I.T. das erste Allzweck-Timesharing-System in den frühen 1960ern erfunden wurde. Alle Computer waren textbasiert, bis Doug Engelbart die Maus und die grafische Benutzungsoberfläche am Stanford Research Institute in den späten 1960ern erfand. Wer weiß, was als Nächstes kommen wird?

In diesem und ähnlichen Abschnitten in den weiteren Kapiteln werfen wir einen kurzen Blick auf die Forschungsarbeit der letzten fünf bis zehn Jahre im Bereich der Betriebssysteme – nur um einen Eindruck davon zu vermitteln, was sich am Horizont abzeichnet. Dieser Leitfaden ist aber sicherlich nicht allumfassend und basiert hauptsächlich auf Veröffentlichungen auf den wichtigen Konferenzen. Diese Ideen haben zumindest die harte Prüfung überlebt, die einer Veröffentlichung vorausgeht. Beachten Sie, dass in der Informatik – im Gegensatz zu anderen wissenschaftlichen Gebieten – der Großteil der Forschungsarbeiten auf Konferenzen, nicht in Journalen veröffentlicht wird. Die meisten der zitierten Artikel kommen entweder von der ACM, von der IEEE Computer Society oder von USENIX und sind für (studentische) Mitglieder

dieser Organisationen über das Internet erhältlich. Nähere Informationen zu den Organisationen und deren Bibliotheken findet man unter den folgenden URLs:

ACM	*http://www.acm.org*
IEEE Computer Society	*http://www.computer.org*
USENIX	*http://www.usenix.org*

So ziemlich alle Forscher im Bereich der Betriebssysteme erkennen, dass bestehende Betriebssysteme riesengroß, unflexibel und unsicher sowie mit unzähligen Fehlern behaftet sind, einige mehr als andere (*wir wollen hier keine Namen nennen*). Folglich wird viel geforscht, wie man bessere Betriebssysteme entwickelt. In der näheren Vergangenheit erschienen Arbeiten zu Fehlern und Fehlerbeseitigung (Renzelmann et al., 2012; Zhou et al., 2012), Wiederherstellung nach Abstürzen (Correia et al., 2012; Ma et al., 2013; Ongaro et al., 2011; Yeh und Chang, 2012), Energieverwaltung (Pathak et al., 2012; Petrucci und Loques, 2012; Shen et al., 2013), Datei- und Speichersysteme (Elnably und Wang, 2012; Nightingale et al., 2012; Zhang et al., 2013a), leistungsstarke Ein-/Ausgabe (De Bruijn et al., 2011; Li et al., 2013a; Rizzo, 2012), Hyperthreading und Multithreading (Liu et al., 2011), Live-Update (Giuffrida et al., 2013), GPU-Verwaltung (Rossbach et al., 2011), Speicherverwaltung (Jantz et al., 2013; Jeong et al., 2013), Mehrkernbetriebssysteme (Baumann et al., 2009; Kapritsos, 2012; Lachaize et al., 2012; Wentzlaff et al., 2012), Korrektheit von Betriebssystemen (Elphinstone et al., 2007; Yang et al., 2006; Klein et al., 2009), Zuverlässigkeit von Betriebssystemen (Hruby et al., 2012; Ryzhyk et al., 2009, 2011; Zheng et al., 2012), Datenschutz und Sicherheit (Dunn et al., 2012; Giuffrida et al., 2012; Li et al., 2013b; Lorch et al., 2013; Ortolani und Crispo, 2012; Slowinska et al., 2012; Ur et al., 2012), Auslastungs- und Leistungskontrolle (Harter et al., 2012; Ravindranath et al., 2012) und Virtualisierung (Agesen et al., 2012; Ben-Yehuda et al., 2010; Colp et al., 2011; Dai et al., 2013; Tarasov et al., 2013; Williams et al., 2012) neben vielen weiteren Themen.

1.10 Überblick über das Buch

Wir haben nun unsere Einführung zu den Betriebssystemen abgeschlossen und verlassen jetzt die Vogelperspektive. Es ist an der Zeit, in die Details zu gehen. Wie bereits gesagt besteht aus Sicht des Programmierers die Hauptaufgabe eines Betriebssystems darin, Schlüsselabstraktionen zur Verfügung zu stellen. Die wichtigsten Abstraktionen sind Prozesse und Threads, Adressräume und Dateien. Demzufolge sind die nächsten drei Kapitel diesen entscheidenden Themen gewidmet.

In ▶*Kapitel 2* geht es um Prozesse und Threads. Es werden deren Eigenschaften erklärt und wie sie untereinander kommunizieren können. Außerdem wird anhand einiger ausführlicher Beispiele gezeigt, wie Interprozesskommunikation funktioniert und wie man die eine oder andere Falle vermeiden kann.

In ▶*Kapitel 3* untersuchen wir Adressräume und die damit zusammenhängende Speicherverwaltung detailliert. Virtueller Speicher wird als wichtiger Punkt neben eng verwandten Konzepten wie Paging und Segmentierung behandelt.

Danach widmen wir uns in ▶*Kapitel 4* dem überaus wichtigen Thema der Dateisysteme. Mit ihnen haben Benutzer am weitaus häufigsten zu tun. Wir werden sowohl die Schnittstelle zum Dateisystem als auch Realisierungen von Dateisystemen betrachten.

Ein-/Ausgabe ist das Thema von ▶*Kapitel 5*. Das Konzept der Geräteunabhängigkeit bzw. der Geräteabhängigkeit wird näher beleuchtet. Einige wichtige Geräte dienen dabei als Beispiele, wie etwa Plattenspeicher, Tastaturen und Bildschirme.

Das ▶*Kapitel 6* handelt von Deadlocks. Dabei werden auch Verfahren besprochen, wie man sie verhindern und vermeiden kann.

An dieser Stelle sind die wichtigsten Ausführungen über Einprozessorsysteme abgeschlossen. Trotzdem gibt es noch viel zu weiterführenden Themen zu sagen. In ▶*Kapitel 7* wird Virtualisierung untersucht. Wir besprechen sowohl die Prinzipien als auch einige der bestehenden Virtualisierungslösungen ausführlich. Da Virtualisierung im Bereich Cloud-Computing viel eingesetzt wird, werden wir uns außerdem verschiedene Clouds ansehen. Ein weiterer wichtiger Punkt sind Mehrprozessorsysteme. Dazu gehören Mehrkernprozessoren, Parallelrechner und verteilte Systeme. Diese Themen werden in ▶*Kapitel 8* behandelt.

Ein weiteres sehr wichtiges Thema ist die Sicherheit von Betriebssystemen, auf das wir in ▶*Kapitel 9* eingehen werden. Neben den bereits in diesem Kapitel angesprochenen Punkten werden dann noch Bedrohungen (z.B. durch Viren und Würmer), Schutzmechanismen und Sicherheitsmodelle besprochen.

Danach folgen einige Fallbeispiele von gängigen Betriebssystemen. Dies sind UNIX, Linux und Android (▶*Kapitel 10*) sowie Windows 8 (▶*Kapitel 11*). Das Buch schließt mit einigen Weisheiten und Gedanken zum Entwurf von Betriebssystemen in ▶*Kapitel 12*.

1.11 Metrische Einheiten

Um Missverständnisse zu vermeiden, ist es an dieser Stelle sinnvoll zu erwähnen, dass in diesem Buch – wie auch in der Informatik im Allgemeinen – das metrische System für Einheiten verwendet wird, anstelle des traditionellen englischen Systems. Die Präfixe des metrischen Systems sind in ▶*Abbildung 1.31* aufgeführt, sie werden meistens mit dem ersten Buchstaben abgekürzt, bei Einheiten größer als eins werden sie als Großbuchstaben geschrieben. Eine 1-TB-Datenbank belegt demnach 10^{12} Byte im Speicher und ein Zeitgeber mit der Auflösung 100 psek (oder auch 100 ps) tickt alle 10^{-10} Sekunden. Da milli und mikro beide mit einem „m“ beginnen, musste man festlegen, dass das „m“ für milli und das „µ“ (das griechische Zeichen für „m“) für mikro steht.

Exp.	Ausgeschrieben	Präfix	Exp.	Ausgeschrieben	Präfix
10^{-3}	0,001	Milli	10^{3}	1.000	Kilo
10^{-6}	0,000 001	Mikro	10^{6}	1.000.000	Mega
10^{-9}	0,000 000 001	Nano	10^{9}	1.000.000.000	Giga
10^{-12}	0,000 000 000 001	Pico	10^{12}	1.000.000.000.000	Tera
10^{-15}	0,000 000 000 000 001	Femto	10^{15}	1.000.000.000.000.000	Peta
10^{-18}	0,000 000 000 000 000 001	Atto	10^{18}	1.000.000.000.000.000.000	Exa
10^{-21}	0,000 000 000 000 000 000 001	Zepto	10^{21}	1.000.000.000.000.000.000.000	Zetta
10^{-24}	0,000 000 000 000 000 000 000 001	Yokto	10^{24}	1.000.000.000.000.000.000.000.000	Yotta

Abbildung 1.31: Die wichtigsten metrischen Präfixe.

Es sollte noch erwähnt werden, dass die Einheiten für Speichergrößen im Allgemeinen eine leicht unterschiedliche Bedeutung haben. Kilo bedeutet dann 2^{10} (1024) im Gegensatz zu 10^3 (1000), weil Speichergrößen aus physischen Gründen immer eine Potenz von 2 sind. Ein 1 KB großer Speicher enthält also 1024 Byte, nicht 1000 Byte. Ebenso enthält ein Speicher von 1 MB 2^{20} (1048576) Byte und ein 1-GB-Speicher damit 2^{30} (1073741824) Byte. Trotzdem überträgt eine 1-kbit/s-Leitung genau 1000 Bit in der Sekunde und ein 10-Mbit/s-LAN arbeitet mit 10000000 Bit/s, weil diese Geschwindigkeiten keine Zweierpotenzen sind. Leider werden diese beiden Faktoren gerne vermischt, vor allem wenn es um Größen von Festplatten geht. Um in diesem Buch Verwechslungen zu vermeiden, verwenden wir die Symbole KB, MB und GB für jeweils 2^{10}, 2^{20} bzw. 2^{30} Byte und die Symbole kbit/s, Mbit/s und Gbit/s für 10^3, 10^6 und 10^9 Bit/s.

ZUSAMMENFASSUNG

Betriebssysteme können unter zwei Gesichtspunkten betrachtet werden: als Ressourcenverwalter oder als erweiterte Maschinen. Wenn man von einem Ressourcenverwalter spricht, meint man die Aufgabe, verschiedene Teile des Systems effizient zu verwalten. Ist die Rede von einer erweiterten Maschine, dann geht es um die Aufgabe, den Benutzern Abstraktionen zur Verfügung zu stellen, deren Benutzung komfortabler als die direkte Verwendung der Hardware ist. Dies schließt Prozesse, Adressräume und Dateien mit ein.

Betriebssysteme haben eine lange Geschichte, die ihren Anfang nahm, als sie die Operatoren ersetzten, und mit modernen multiprogrammierbaren Systemen ihre Fortsetzung findet. Wichtige Stationen waren die frühen Stapelverarbeitungssysteme, die multiprogrammierbaren Systeme und die PCs.

Da Betriebssysteme sehr eng mit der Hardware zusammenarbeiten, benötigt man Kenntnisse über die Hardware, um Betriebssysteme besser verstehen zu können. Computer sind aus Prozessoren, Speicher und Ein-/Ausgabegeräten aufgebaut, die alle durch Busse miteinander verbunden sind.

Die Grundkonzepte, auf denen alle Betriebssysteme basieren, sind Prozesse, Speicherverwaltung, Ein-/Ausgabeverwaltung, das Dateisystem und Sicherheit. Jeder dieser Aspekte wird in einem der folgenden Kapitel behandelt.

Das Herzstück jedes Betriebssystems bilden die angebotenen Systemaufrufe. Sie zeigen, was das Betriebssystem wirklich leistet. Wir haben für UNIX vier Gruppen dieser Systemaufrufe betrachtet. Die erste betraf die Erzeugung und das Beenden von Prozessen, die zweite das Lesen und Schreiben von Dateien. Die dritte Gruppe enthielt Funktionen für die Verzeichnisverwaltung, die vierte umfasste verschiedene andere Funktionen.

Betriebssysteme können auf unterschiedliche Art und Weise strukturiert werden. Die bekanntesten sind monolithische Systeme, hierarchisch geschichtete Systeme, Mikrokerne, Client-Server-Modelle, virtuelle Maschinen und Exokerne.

Übungen

1. Was sind die zwei Hauptfunktionen eines Betriebssystems?
2. In ▶*Abschnitt 1.4* wurden neun unterschiedliche Betriebssystemarten beschrieben. Geben Sie Anwendungsbeispiele für jedes dieser Systeme an (eines für jede Betriebssystemart).
3. Worin besteht der Unterschied zwischen Timesharing- und Multiprogrammiersystemen?
4. Um Cache-Speicher zu benutzen, wird der Arbeitsspeicher in Cache-Lines aufgeteilt, die in der Regel 32 oder 64 Byte lang sind. Eine Cache-Line wird an einem Stück in den Cache geladen. Worin liegt der Vorteil, eine gesamte Cache-Line anstelle von einzelnen Bytes oder Wörtern nacheinander zu laden?
5. Bei den ersten Computern wurde das Lesen und Schreiben jedes einzelnen Bytes vom Prozessor durchgeführt (d.h., es gab noch keine DMA). Was für Auswirkungen hat das für die Multiprogrammierung?
6. Befehle, die den Zugriff auf Ein-/Ausgabegeräte betreffen, sind in der Regel privilegierte Anweisungen, das heißt, sie können im Kernmodus ausgeführt werden, aber nicht im Benutzermodus. Geben Sie einen Grund dafür an, warum diese Anweisungen privilegiert sind.
7. Die Idee, ganze Familien von Rechnern zu bauen, wurde in den 1960er Jahren mit dem System/360 von IBM für Großrechner eingeführt. Ist diese Idee heute gestorben oder existiert sie immer noch?
8. Ein Grund für die zurückhaltende Annahme grafischer Benutzungsoberflächen war, dass die nötige Hardware anfangs noch sehr teuer war. Wie viel Videospeicher braucht man für die Darstellung von 80 Zeichen auf 25 Zeilen Textmodus in Schwarz-Weiß? Und wie viel braucht man für die Darstellung von 1200×900 Bildpunkten mit 24-Bit-Farbtiefe? Was hat der nötige Speicher 1980 gekostet, als ein KB etwa 5 US-Dollar kostete? Und wie viel kostet er heute?
9. Es gibt mehrere Entwurfsziele bei der Entwicklung eines Betriebssystems, z.B. Betriebsmittelausnutzung, Rechtzeitigkeit, Robustheit usw. Geben Sie ein Beispiel für zwei Entwurfsziele, die sich möglicherweise gegenseitig widersprechen.
10. Worin besteht der Unterschied zwischen Kern- und Benutzermodus? Erläutern Sie, inwiefern die Tatsache, zwei unterschiedliche Modi zu haben, beim Entwurf eines Betriebssystems hilfreich ist.

12. Eine 256-GB-Platte hat 65536 Zylinder mit 255 Sektoren pro Track und 512 Byte pro Sektor. Wie viele Scheiben und Köpfe hat diese Platte? Gehen Sie von einer durchschnittlichen Zylindersuchzeit von 11 ms, einer durchschnittlichen Rotationsverzögerung von 7 ms und einer Lesegeschwindigkeit von 100 MB/s aus. Berechnen Sie die durchschnittliche Zeit, die benötigt wird, um 400 KB aus einem Sektor zu lesen.

13. Welche der folgenden Befehle sollten nur im Kernmodus erlaubt sein?

a. Sperren aller Unterbrechungen

b. Lesen der aktuellen Uhrzeit

c. Setzen der aktuellen Uhrzeit

d. Ändern der Speicherzuordnungstabellen

14. Betrachten Sie ein System mit zwei Prozessoren, wobei jeder diese Prozessoren zwei Threads (Hyperthreading) hat. Nehmen Sie an, dass drei Programme, *P0*, *P1* und *P2*, mit Laufzeiten von 5, 10 bzw. 20 ms gestartet werden. Wie lange wird es dauern, bis die Ausführung dieser Programme vollständig abgeschlossen ist? Nehmen Sie dazu an, dass alle drei Programme zu 100 % CPU-gebunden sind, sich während der Ausführung nicht gegenseitig blockieren und die einmal zugewiesene CPU nicht getauscht wird.

15. Ein Computer hat zur Befehlsabarbeitung eine vierstufige Pipeline eingebaut. Jede Stufe benötigt für die Bearbeitung dieselbe Zeit von 1 ns. Wie viele Befehle kann dieser Rechner pro Sekunde abarbeiten?

16. Betrachten Sie ein Computersystem, dass über Cache-Speicher, Arbeitsspeicher (RAM) und Plattenspeicher und ein Betriebssystem verfügt, das virtuellen Speicher benutzt. Man benötigt 1 ns für den Zugriff auf ein Datenwort im Cache, 10 ns für ein Wort im RAM und 10 ms für ein Wort von der Platte. Wenn die Cache-Trefferrate 95 % und die Trefferrate im Arbeitsspeicher (nach einem Fehlschlag im Cache) 99 % beträgt, wie lang ist dann die durchschnittliche Zugriffszeit auf ein Wort?

17. Wenn ein Benutzerprogramm einen Systemaufruf macht, um eine Datei von der Festplatte zu lesen oder darauf zu schreiben, übergibt es einen Identifikator für die Datei, einen Zeiger auf einen Datenpuffer und einen Zähler. Danach wird die Kontrolle an das Betriebssystem abgegeben, das den entsprechenden Treiber aufruft. Stellen Sie sich vor, dass der Gerätetreiber die Aktion startet und sich beendet, bis ein Interrupt auftritt. Für den Fall, dass von der Platte gelesen wird, muss der Aufrufer blockiert werden, weil er auf die zu lesenden Daten wartet. Aber was ist, wenn auf die Platte geschrieben wird? Muss der Aufrufer dann auch auf die Beendigung der Datenspeicherung warten?

18. Wozu braucht man eine Prozesstabelle in einem Timesharing-System? Wird die Tabelle auch bei PCs benötigt, auf denen UNIX oder Windows als Einbenutzersystem läuft?

19. Gibt es einen Grund, warum Sie ein Dateisystem in ein nicht leeres Verzeichnis einhängen wollen? Falls ja, welchen?

20. Geben Sie für jeden der folgenden Systemaufrufe einen Grund an, weshalb er nicht funktionieren könnte: `fork`, `exec` und `unlink`.

21. Welche Art von Multiplexing (zeitlich, räumlich oder beides) kann verwendet werden, um die folgenden Ressourcen gemeinsam zu nutzen: CPU, Speicher, Platte, Netzkarte, Drucker, Tastatur und Bildschirm?

22. Kann der Aufruf von

```
count = write(fd, buffer, nbytes);
```

einen anderen Wert als *nbytes* in *count* zurückliefern? Wenn ja, warum?

23. Eine Datei mit dem Dateideskriptor *fd* enthält die folgende Sequenz von Zeichen: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. Folgende Systemaufrufe werden durchgeführt:

```
lseek(fd, 3, SEEK_SET);
read(fd, &buffer, 4);
```

Dabei wird die aktuelle Leseposition in der Datei durch `lseek` auf das dritte Byte gesetzt. Was enthält *buffer* nach dem Ende des Lesens?

24. Nehmen Sie an, dass eine 10 MB große Datei auf einer Platte auf derselben Spur (Spur 50) in aufeinanderfolgenden Sektoren gespeichert wird. Der Plattenarm befindet sich aktuell über der Spurnummer 100. Wie lange wird es dauern, diese Datei von der Platte abzurufen? Nehmen Sie dazu an, dass die Bewegung des Arms von einem Zylinder zum nächsten ungefähr 1 ms dauert und dass es 5 ms dauert, bis der Sektor mit dem Dateianfang unter dem Kopf platziert ist. Nehmen Sie weiter an, dass das Lesen mit 200 MB/s stattfindet.

25. Was ist der wesentliche Unterschied zwischen einer Blockdatei und einer Zeichendatei?

26. Im Beispiel der ▶*Abbildung 1.17* hat die Bibliotheksfunktion den Namen *read* und der Systemaufruf heißt auch `read`. Ist es nötig, dass beide Funktionen denselben Namen haben? Falls nicht, welcher ist wichtiger?

27. Moderne Betriebssysteme entkoppeln den Adressraum eines Prozesses von der physischen Speicheradresse. Geben Sie zwei Vorteile dieses Vorgehens an.

28. Für einen Programmierer sieht ein Systemaufruf genauso aus wie der Aufruf einer Bibliotheksfunktion. Ist es für den Programmierer wichtig zu wissen, wann eine Bibliotheksfunktion einen Systemaufruf ausführt? Falls ja, unter welchen Umständen und warum?

29. In ▶*Abbildung 1.23* sieht man einige Systemaufrufe von UNIX, für die keine entsprechende Funktion in der Win32-API existiert. Überlegen Sie sich für jede dieser Funktionen, welche Folgen das für einen Programmierer hat, der ein UNIX-Programm konvertieren will, um es unter Windows laufen zu lassen.

30. Ein Betriebssystem heißt portabel, wenn es von einer Systemarchitektur zu einer anderen ohne Modifikationen übertragen werden kann. Erklären Sie, warum es nicht machbar ist, ein Betriebssystem zu entwickeln, das vollständig portabel ist. Beschreiben Sie zwei obere Schichten, die man bei der Entwicklung eines hochportablen Betriebssystems erhält.

31. Erklären Sie, wie die Trennung von Strategie und Mechanismus bei der Entwicklung von mikrokernbasierten Betriebssystemen hilft.

32. Virtuelle Maschinen sind aus unterschiedlichen Gründen sehr beliebt geworden. Dennoch haben sie auch einige Nachteile. Nennen Sie einen.

33. Nun ein paar Aufgaben, die das Rechnen mit Einheiten einüben sollen:

a. Wie lange dauert ein Nanojahr in Sekunden?

b. Mikrometer werden manchmal als Mikron (im Deutschen oft 1 μ) bezeichnet. Wie lang ist ein Megamikron?

c. Wie viele Bytes enthält ein 1-PB-Speicher?

d. Die Masse der Erde beträgt 6000 Yottagramm. Wie viel ist das in Kilogramm?

34. Programmieren Sie eine Shell, die ähnlich wie die in ▶*Abbildung 1.19* ist. Erweitern Sie diese so, dass Sie sie auch testen können. Außerdem sollte Ihre Shell die Möglichkeit besitzen, Ein- und Ausgabe umzuleiten, Pipes anbieten und Jobs in den Hintergrund stellen können.

35. Wenn Sie einen Rechner mit einem UNIX-System zur Verfügung haben (Linux, MINIX, FreeBSD oder Ähnliches), den Sie ohne jemanden zu stören einfach abstürzen lassen und neu starten können, dann schreiben Sie ein kleines Programm, das versucht, eine unbegrenzte Anzahl von Kindprozessen zu erzeugen. Beobachten Sie, was dann passiert. Bevor Sie dieses Experiment starten, sollten die Datenpuffer mit dem Shell-Befehl `sync` noch auf die Platte geschrieben werden, damit das Dateisystem keinen Schaden nimmt. **Achtung:** Versuchen Sie das bitte nicht ohne Erlaubnis des Systemadministrators auf einem System, an dem andere Benutzer arbeiten. Die Folgen sind sehr schnell zu spüren und Sie könnten dafür zur Rechenschaft gezogen werden!

36. Untersuchen Sie den Inhalt eines UNIX- oder Windows-Verzeichnisses und versuchen Sie, diese zu interpretieren. Nutzen Sie dazu Hilfsmittel wie das UNIX-Programm *od*. (*Hinweis:* Wie Sie dabei vorgehen, hängt davon ab, was das verwendete Betriebssystem erlaubt. Ein Trick wäre, das Verzeichnis auf einem USB-Stick anzulegen und dann die Daten unter einem anderen Betriebssystem direkt einzulesen, das diesen Zugriff erlaubt.)

Prozesse und Threads

2

ÜBERBLICK

» Wir beginnen nun eine detaillierte Studie über den Entwurf und die Konstruktion von Betriebssystemen. Das zentrale Konzept in jedem Betriebssystem ist der *Prozess*: eine Abstraktion eines laufenden Programms. Alles andere hängt von diesem Konzept ab, deshalb sollten Betriebssystementwickler (sowie Studierende) so früh wie möglich eine fundierte Vorstellung davon bekommen, was ein Prozess ist.

Prozesse sind eine der ältesten und wichtigsten Abstraktionen, die von Betriebssystemen angeboten werden. Sie unterstützen die Fähigkeit der (Quasi-)Parallelität, auch wenn es nur eine CPU im System gibt – eine einzelne CPU wird in viele virtuelle CPUs verwandelt. Ohne die Prozessabstraktion gäbe es die moderne EDV nicht. In diesem Kapitel werden wir uns Prozesse und ihre „Verwandten", die Threads, sehr detailliert ansehen. «

2.1 Prozesse

Moderne Computer können oft mehrere Dinge gleichzeitig erledigen. Der normale PC-Nutzer ist sich dieser Tatsache möglicherweise nicht völlig bewusst, also könnten hier ein paar Fakten für Aufhellung sorgen. Betrachten wir zunächst einen Webserver. Von überall her kommen Anfragen nach Webseiten. Wenn solch eine Anfrage eintrifft, überprüft der Server zunächst, ob die benötigte Seite im Cache ist. Falls ja, schickt er die angefragte Seite zurück. Falls nicht, wird eine Plattenanfrage gestartet, um die Seite zu holen. Aus der Perspektive der CPU benötigt diese Plattenanfrage eine Ewigkeit. Während dieser Wartezeit kommen möglicherweise viele weitere Anfragen herein. Wenn es mehrere Platten im System gibt, werden alle oder einige der neueren Anfragen vielleicht schon zu anderen Platten umgeleitet, lange bevor die erste Anfrage erfüllt ist. Offensichtlich wird irgendeine Methode benötigt, um diese Nebenläufigkeit zu modellieren und zu steuern. Prozesse (und insbesondere Threads) können hier helfen.

Werfen wir nun einen Blick auf den PC. Wenn das System hochgefahren wird, werden viele Prozesse heimlich gestartet, was der Benutzer oft nicht bemerkt. Zum Beispiel könnte ein Prozess gestartet werden, der auf ankommende E-Mails wartet. Ein weiterer Prozess könnte im Auftrag des Antivirenprogramms in bestimmten Abständen überprüfen, ob neue Virendefinitionen erhältlich sind. Zusätzlich könnten explizite Benutzerprogramme laufen, die zum Beispiel Dateien drucken oder eine Sicherheitskopie der Fotos auf einem USB-Stick herstellen – und dies alles, während der Benutzer im Web surft. All diese Aktivitäten müssen verwaltet werden und da kommt ein Multiprogrammiersystem, das mehrere Prozesse unterstützt, jetzt ganz gelegen.

In einem Multiprogrammiersystem wechselt die CPU schnell von Programm zu Programm, wobei jedes Programm im Bereich von ungefähr zehn bis hundert Millisekunden rechnet. Genau betrachtet läuft zu jedem Zeitpunkt immer nur ein Programm auf der CPU. In einem Zeitraum von einer Sekunde jedoch können mehrere Programme bearbeitet werden, was beim Benutzer die Illusion von Parallelität erzeugt. Manchmal spricht man in diesem Zusammenhang von **Quasiparallelität** (*pseudoparallelism*), um

den Gegensatz zur echten Hardwareparallelität von **Multiprozessorsystemen** (in denen sich zwei oder mehr CPUs den gleichen physischen Speicher teilen) hervorzuheben. Für einen Menschen ist es schwierig, den Überblick über mehrere parallele Vorgänge zu behalten. Deshalb erarbeiteten Betriebssystementwickler im Laufe der Jahre ein konzeptionelles Modell (sequenzielle Prozesse), das den Umgang mit Parallelität vereinfacht. Dieses Modell, seine Anwendungen und einige seiner Auswirkungen sind der Inhalt dieses Kapitels.

2.1.1 Das Prozessmodell

In diesem Modell ist die gesamte auf dem Computer ausführbare Software und manchmal auch das Betriebssystem als eine Menge von **sequenziellen Prozessen** oder kurz **Prozessen** organisiert. Ein Prozess ist nichts anderes als die Instanz eines Programms in Ausführung, inklusive des aktuellen Wertes des Befehlszählers, der Registerinhalte und der Belegungen der Variablen. Konzeptionell besitzt jeder Prozess seine eigene virtuelle CPU. In der Realität schaltet natürlich die CPU zwischen den Prozessen hin und her. Zum Verständnis ist es jedoch viel einfacher, sich eine Menge von (quasi-)parallel laufenden Prozessen vorzustellen, als zu versuchen, die Übersicht darüber zu behalten, wie die CPU zwischen den Programmen wechselt. Dieses schnelle Hin- und Herschalten wird als **Multiprogrammierung** bezeichnet, wie schon in *Kapitel 1* erwähnt.

▶*Abbildung 2.1a* zeigt den Speicher eines Computers, der vier Programme parallel berechnet. In ▶*Abbildung 2.1b* sehen wir vier Prozesse, von denen jeder seine eigene Ablaufsteuerung (d.h. seinen eigenen logischen Befehlszähler) besitzt und unabhängig von den anderen läuft. Natürlich existiert in der Hardware nur ein einziger Befehlszähler. Daher wird der logische Befehlszähler des jeweils laufenden Prozesses in den realen Befehlszähler geladen. Sobald die Zeitscheibe des Prozesses abgelaufen ist, wird der reale Befehlszähler im abgespeicherten logischen Befehlszähler des Prozesses im Speicher gesichert. In ▶*Abbildung 2.1c* ist zu sehen, dass über ein längeres Zeitintervall alle Prozesse in ihrer Ausführung fortgeschritten sind, obwohl zu einem beliebigen Zeitpunkt nur ein Prozess tatsächlich läuft.

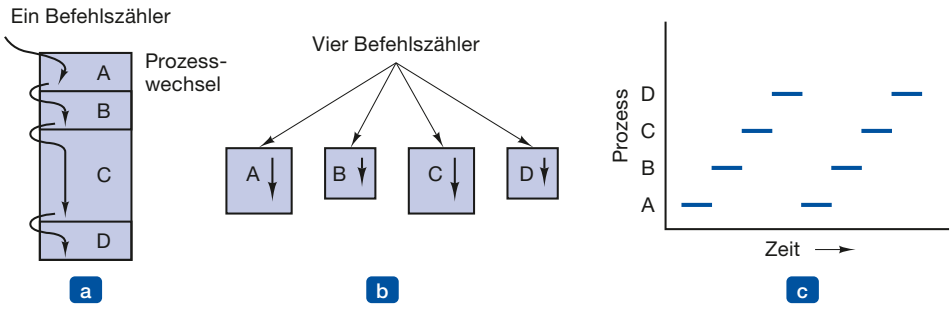

Abbildung 2.1: (a) Multiprogrammierung von vier Programmen. (b) Konzeptionelles Modell von vier individuellen sequenziellen Prozessen. (c) Zu jedem Zeitpunkt ist immer nur ein Programm aktiv.

In diesem Kapitel gehen wir zunächst davon aus, dass es im betrachteten System nur eine CPU gibt. Diese Annahme verliert zunehmend an Richtigkeit, da neue Chips oft Mehrkernchips mit zwei, vier oder mehr Kernen sind. Wir werden uns die Mehrkernchips und Multiprozessoren allgemein in *Kapitel 8* ansehen, doch vorläufig ist es einfacher, nur jeweils eine CPU zu betrachten. Wenn wir also sagen, dass eine CPU wirklich nur jeweils einen Prozess laufen lassen kann, dann heißt das, dass bei mehreren Kernen (oder CPUs) jede von ihnen jeweils nur einen Prozess laufen lassen kann.

Durch das Wechseln der CPU zwischen den Prozessen ist die Geschwindigkeit, mit der ein Prozess seine Berechnung durchführt, nicht einheitlich und wahrscheinlich nicht einmal reproduzierbar, wenn derselbe Prozess noch einmal ausgeführt wird. Daher dürfen keine Annahmen über den Zeitablauf in einen Prozess einprogrammiert werden. Stellen wir uns beispielsweise einen Audioprozess vor, der die Musik zu einem Video hoher Qualität beisteuert, welches von einem anderen Gerät abgespielt wird. Weil die Audiowiedergabe ein wenig später als das Video starten sollten, signalisiert der Audioprozess dem Videoserver, mit dem Abspielen zu beginnen, und durchläuft dann 10000-mal eine Warteschleife, bevor die Audiowiedergabe gestartet wird. Dies funktioniert gut, wenn die Schleife ein zuverlässiger Timer ist. Falls sich aber die CPU entscheidet, während der Warteschleife zu einem anderen Prozess zu wechseln, kann es passieren, dass der Audioprozess erst dann wieder zur Ausführung kommt, wenn die dazu gehörigen Videorahmen bereits abgespielt sind – Video- und Audiosignal sind dann nicht mehr synchron. Weist ein Prozess kritische Echtzeitanforderungen dieser Art auf, wenn also bestimmte Ereignisse innerhalb einer festgelegten Anzahl von Millisekunden stattfinden *müssen*, dann muss durch spezielle Maßnahmen sichergestellt werden, dass diese Ereignisse auch wirklich eintreten. Im Allgemeinen werden die meisten Prozesse jedoch nicht von der Multiprogrammierung der CPU oder von den relativen Geschwindigkeiten verschiedener Prozesse beeinflusst.

Der Unterschied zwischen einem Prozess und einem Programm ist subtil, aber (zum Verständnis) äußerst wichtig. Eine Analogie kann uns hierbei helfen: Stellen Sie sich einen kulinarisch interessierten Informatiker vor, der einen Geburtstagskuchen für seine kleine Tochter bäckt. Er hat ein Geburtstagskuchenrezept und eine gut ausgestattete Küche mit allen Zutaten: Mehl, Eier, Zucker, Vanillearoma und so weiter. In dieser Analogie ist das Rezept das Programm, also ein in einer passenden Notation geschriebener Algorithmus, der Informatiker ist der Prozessor (CPU) und die Zutaten für den Kuchen sind die Eingabedaten. Der Prozess ist die Aktivität, die daraus besteht, dass unser Bäcker das Rezept liest, die Zutaten herbeiholt und den Kuchen bäckt.

Nun stellen Sie sich vor, dass der Sohn des Informatikers wie am Spieß schreiend hereingelaufen kommt, weil er von einer Biene gestochen wurde. Der Informatiker notiert sich, an welcher Stelle des Rezepts er sich befindet (der Zustand des aktuellen Prozesses wird gespeichert), holt ein Erste-Hilfe-Buch und beginnt den Anordnungen darin zu folgen. Hier sehen wir, wie der Prozessor von einem Prozess (Backen) zu einem Prozess mit höherer Priorität (medizinische Hilfe leisten) wechselt. Jedem der Pro-

zesse liegt ein unterschiedliches Programm zugrunde (das Rezept bzw. das Erste-Hilfe-Buch). Nachdem der Bienenstich behandelt worden ist, kann der Informatiker zu seinem Kuchen zurückkehren und an dem Punkt fortfahren, an dem er unterbrochen wurde.

Der entscheidende Gedanke ist hier, dass ein Prozess eine Aktivität jedweder Art ist. Er umfasst ein Programm, Eingaben, Ausgaben und einen Zustand. Mehrere Prozesse können sich einen einzelnen Prozessor teilen. Eine Schedulingstrategie entscheidet, wann die Arbeit an einem Prozess unterbrochen und ein anderer Prozess bedient wird. Im Gegensatz dazu ist ein Programm etwas, das man auf einem Datenträger speichern kann und das nichts tut.

Es sollte noch bemerkt werden, dass ein Programm, das zweimal läuft, wie zwei Prozesse gezählt wird. Zum Beispiel ist es häufig möglich, ein Textverarbeitungsprogramm zweimal zu starten oder zwei Dateien gleichzeitig zu drucken, falls zwei Drucker verfügbar sind. Die Tatsache, dass zwei Prozesse zufällig das gleiche Programm ausführen, spielt keine Rolle – es sind trotzdem verschiedene Prozesse. Das Betriebssystem kann möglicherweise den Code zwischen ihnen aufteilen, sodass nur eine Kopie im Speicher ist, aber dies ist lediglich ein technisches Detail, das die grundsätzliche Situation nicht ändert.

2.1.2 Prozesserzeugung

Betriebssysteme benötigen ein Verfahren zum Erzeugen von Prozessen. In sehr einfachen Systemen oder in Systemen, deren Entwurf nur eine einzige Anwendung vorsieht (z.B. die Steuerung eines Mikrowellenherds), können alle benötigten Prozesse bereits beim Systemstart vorhanden sein. In allgemein gebräuchlichen Systemen benötigt man jedoch ein Verfahren, um Prozesse im laufenden Betrieb je nach Bedarf zu erzeugen und zu beenden. Wir werden nun einige der Punkte betrachten.

Es gibt prinzipiell vier Ereignisse, die das Erzeugen eines Prozesses verursachen:

1. Initialisierung des Systems
2. Systemaufruf zum Erzeugen eines Prozesses durch einen anderen Prozess
3. Benutzeranfrage, einen neuen Prozess zu erzeugen
4. Initiierung einer Stapelverarbeitung (Stapeljob)

Beim Hochfahren eines Betriebssystems werden normalerweise viele Prozesse erzeugt. Einige davon sind Prozesse, die im Vordergrund laufen, das heißt Prozesse, die mit (menschlichen) Benutzern interagieren und für diese Aufgaben ausführen. Andere laufen im Hintergrund und lassen sich nicht bestimmten Benutzern zuordnen lassen, sondern erfüllen spezielle Funktionen. Beispielsweise kann für das Empfangen von E-Mails ein Hintergrundprozess bestimmt sein, der den größten Teil des Tages schläft, aber plötzlich quicklebendig wird, wenn eine E-Mail eintrifft. Ein anderer Hintergrundprozess bearbeitet beispielsweise Anfragen für Webseiten, die auf diesem Rechner abgelegt sind. Er erwacht beim Eintreffen einer Anfrage, um diese dann zu

bearbeiten. Prozesse, die im Hintergrund bleiben, um Aktivitäten wie E-Mails, Webseiten, Druckaufträge und so weiter zu bearbeiten, heißen **Daemons**. Große Systeme haben üblicherweise Dutzende davon. In UNIX[1] kann man sich mithilfe des Programms *ps* die laufenden Prozesse anzeigen lassen. Unter Windows wird dazu der Taskmanager benutzt.

Zusätzlich zu den beim Systemstart erzeugten Prozessen können auch später Prozesse erzeugt werden. Oft führen laufende Prozesse Systemaufrufe aus, um einen oder mehrere neue Prozesse zu erzeugen, die ihnen bei der Verarbeitung helfen. Das Erzeugen von Prozessen ist insbesondere dann hilfreich, wenn die zu erledigende Arbeit leicht auf mehrere in Verbindung stehende, aber unabhängig voneinander interagierende Prozesse aufteilbar ist. Wenn beispielsweise große Datenmengen über ein Netzwerk für die weitere Verarbeitung geladen werden müssen, dann kann es praktisch sein, einen ersten Prozess zu erzeugen, der die Daten lädt und in einen gemeinsamen Puffer schreibt, während ein zweiter Prozess die Daten aus dem Puffer liest und verarbeitet. In einem Multiprozessorsystem ließe sich die Aufgabe noch weiter beschleunigen, indem jeder Prozess auf einem eigenen Prozessor läuft.

In interaktiven Systemen können Benutzer beliebige Programme durch Eingeben eines Kommandos oder das (Doppel-)Klicken eines Icons starten. Jede dieser Aktionen startet einen neuen Prozess und lässt darin das gewählte Programm ablaufen. In kommandobasierten UNIX-Systemen, auf denen X-Window läuft, übernimmt der Prozess das Fenster, in dem er gestartet wurde. Bei Microsoft Windows hat ein gestarteter Prozess kein Fenster, er kann aber eines (oder mehrere) erzeugen, was meistens auch geschieht. In beiden Systemen können Benutzer mehrere Fenster, in denen je ein Prozess läuft, gleichzeitig geöffnet haben. Mithilfe der Maus kann der Benutzer ein Fenster auswählen und dann mit dem Prozess interagieren, um beispielsweise erforderliche Eingaben zu machen.

Die letzte Situation, in der Prozesse erzeugt werden, entsteht nur in Stapelverarbeitungssystemen von Großrechnern. Denken Sie beispielsweise an die Bestandsverwaltung am Ende eines Tages bei einer Ladenkette. Benutzer können hier Stapeljobs an das System übertragen (möglicherweise von entfernten Rechnern aus). Sobald das Betriebssystem entscheidet, dass genügend Betriebsmittel zur Verarbeitung einer neuen Aufgabe zur Verfügung stehen, erzeugt es einen neuen Prozess und verarbeitet darin die nächste Aufgabe aus der Warteschlange.

Technisch gesehen wird in all diesen Fällen ein neuer Prozess erzeugt, indem ein bestehender Prozess einen Systemaufruf zur Prozesserzeugung ausführt. Dieser aufrufende Prozess kann ein laufender Benutzerprozess, ein durch die Tastatur oder Maus aufgerufener Systemprozess oder auch ein Verwaltungsprozess zur Stapelverarbeitung sein. Der Systemaufruf teilt dem Betriebssystem mit, dass ein neuer Prozess zu erzeugen ist, und gibt direkt oder indirekt an, welches Programm darin ausgeführt werden soll.

1 In diesem Kapitel verstehen wir unter „UNIX“ alle POSIX-basierten Systeme, einschließlich Linux, FreeBSD, OS X, Solaris usw. sowie teilweise auch Android und iOS.