Pearson Education

ADDISON-WESLEY

Linux-Kernelprogrammierung

Linux Specials

Pearson Education

ADDISON-WESLEY

Michael Beck, Harald Böhme,
Mirko Dziadzka, Ulrich Kunitz,
Robert Magnus, Claus Schröter,
Dirk Verworner

Linux-Kernel-programmierung

Algorithmen und Strukturen der Version 2.4

6., aktualisierte und erweiterte Auflage

Bitte beachten Sie: Der originalen Printversion liegt eine CD-ROM bei.
In der vorliegenden elektronischen Version ist die Lieferung einer CD-ROM nicht enthalten.
Alle Hinweise und alle Verweise auf die CD-ROM sind ungültig.

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek — CIP-Einheitsaufnahme

Ein Titelsatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich

10 9 8 7 6 5 4 3 2 1

04 03 02 01

ISBN 3-8273-1659-6

Lektorat: Susanne Spitzer, susanne.spitzer@gmx.net
Korrektorat: Friederike Daenecke, Zülpich
Produktion: TYPisch Müller, Arcevia, Italien, typmy@freefast.it
Satz: Hilmar Schlegel, Berlin
Umschlaggestaltung: Hommer DesignProduction, Haar bei München
Druck und Verarbeitung: Kösel, Kempten

Printed in Germany

Pearson Education

ADDISON-WESLEY

Inhaltsverzeichnis

Pearson Education

ADDISON-WESLEY

Vorwort zur 6. Auflage

Ein Preisausschreiben an einem Linux-Stand auf der diesjährigen CeBIT fragte, welche Linux-Version der Version 0.95a vorausging. Ich muss zugeben — ich weiß es nicht mehr — es war jedenfalls nicht 0.94.

Das erinnert uns wieder an die Anfänge und das sich schnell verändernde, kreative Chaos, das Linux zu dieser Zeit umgab. Zu dieser Zeit war es für viele Mitentwickler von Linux eine Herausforderung, die Quellen eines Betriebsystems verstehen und modifizieren zu können.

Inzwischen hat Linux nicht nur den magischen Versionsmeilenstein 2.4 erreicht, sondern auch in der hart umkämpften Softwarebranche einen festen Platz erobert. An der Herausforderung, einen Betriebsystemkern zu verstehen, hat sich nichts geändert, sie ist nur größer geworden.

Viele der seit der letzten Meilensteinversion hinzugefügten Features dienen heute nicht mehr nur der banalen Funktion eines Betriebssystems, sondern es kommen mehr und mehr Funktionalitäten hinzu, die Kompatibilitäten zu großen Softwareprodukten ermöglichen, der Unterstützung neuer Hardware oder der Verbesserung der Performance des Systems dienen. Aber auch die 2.4 Version bietet wieder spannende neue Konzepte wie die IP-Tables oder verbessertes Plug-and-Play.

Wie mit jedem neuen Meilenstein des Linux-Kernels, muss auch ein Kernel-Buch grundlegend überarbeitet werden, um auf dem neuesten Stand zu sein. Trotz massiver Änderungen in Schnittstellen und Konzepten soll es wieder einen Einblick in das Getriebe von Linux geben. Das Ergebnis dieser Überarbeitung liegt nun vor Ihnen. Wir wünschen Ihnen nicht nur beim Lesen des Buches, sondern auch beim Experimentieren mit dem Linux-Kern viel Spaß.

Berlin/Frankfurt/Furtwangen, den 24. 4. 2001

Michael Beck
Harald Böhme
Mirko Dziadzka
Ulrich Kunitz
Robert Magnus
Claus Schröter

Vorwort der Autoren zur 1. Auflage

Linux gibt es seit etwa zwei Jahren. Was einst als Programmierübung des Informatikstudenten Linus Torvalds begann, ist heute eines der erfolgreichsten Free-Software-Projekte und macht kommerziellen Systemen ernsthaft Konkurrenz. Dies ist das Ergebnis der freiwilligen Arbeit einer weltweiten Programmierergemeinde, die durch ein effektives Kommunikationsmedium, das Internet, verbunden sind. Die freie Verfügbarkeit von Linux hat zu seiner raschen Verbreitung beigetragen. Sicher ist es schwer, die Zahl der Linux-Nutzer zu schätzen. In Deutschland sind es mit Sicherheit schon mehrere zehntausend.

Vor circa eineinhalb Jahren haben wir, die Autoren, das LINUX-System für uns entdeckt. Ein Grund dafür besteht sicherlich darin, dass wir jetzt für unsere heimischen PCs ein „richtiges" UNIX-System haben, ohne dafür gleich Tausende von Mark, die man als Student sowieso nicht hat, auf den Tisch legen zu müssen.

Der andere, vielleicht wichtigere Grund besteht für uns, und sicherlich auch für einen Großteil der LINUX-Gemeinde in der Welt, in der Verfügbarkeit der Quelltexte des LINUX-Systems. Es macht einfach Spaß, in den Interna eines Betriebssystems zu wühlen, eigene Ideen auszuprobieren und das System in allen Belangen an seine eigenen Wünsche anzupassen. Dieses Buch wendet sich an alle, die genauso denken, aber auch an die, die einfach nur entdecken wollen, wie ein 32-Bit-Betriebssystem funktioniert.

Der LINUX-Kern hat im Laufe der Zeit an Umfang zugenommen. Einen wirklich guten Überblick kann man sich nicht mehr allein verschaffen. Da Dokumentationen dünn gesät sind (die einzige Dokumentation, die wir kennen, ist der Entwurf des *Linux Kernel Hackers Guide* [Joh95]), haben wir im Sommersemester 1993 ein LINUX-Seminar begonnen. Jeder, der sich bei uns mit LINUX beschäftigte, gab einen Einblick in sein Interessengebiet, in sein Wissen und seine Erfahrungen beim „Kernel Hacking". Im Seminar kam es häufig zu spannenden Diskussionen um Modellierungskonzepte, Implementierungsvarianten und Details, die unterschiedlich aufgefaßt wurden. Wir haben im Rahmen dieses Seminars begonnen, unser Wissen über das LINUX-System aufzuschreiben, um anderen einen einfacheren Einstieg zu ermöglichen. Dieses Wissen liegt nun — überarbeitet — in diesem Buch vor.

Da die Entwicklung von LINUX sehr schnell vorwärtsschreitet, konnten wir uns für das Schreiben des Buches nicht allzuviel Zeit lassen. Wir teilten deswegen die einzelnen Kapitel des Buches entsprechend den Interessengebieten der Autoren auf. Ulrich Kunitz schrieb die Einleitung, das Kapitel über die Speicherverwaltung und das Kapitel über die Interprozesskommunikation. Mirko Dziadzka zeichnet für die Einführung in den Kern verantwortlich. Harald Böhme, unser Netzexperte, hätte sicherlich ein ganzes Buch schreiben müssen, um die Netzwerkimplementation umfassend zu erläutern. Hier konnte er nur in die Materie einführen. Robert Magnus fiel die undankbare Aufgabe zu, die Referenz der Systemrufe auszuarbeiten und die systemnahen Kommandos zu erläutern. Die weiteren Kapitel teilten sich die anderen Autoren auf.

Beim Schreiben eines deutschen Buches über ein Betriebssystem ist man immer wieder mit dem Problem der korrekten Übersetzung englischer Fachbegriffe konfrontiert. Im Buch ist bei der Einführung eines Begriffs die englische Originalbezeichnung und deren deutsche Übersetzung angegeben. Oft wurde, wo es dem Sprachgefühl nicht widersprach, die englische Bezeichnung weiterverwendet.

Im Text sind Bezeichner aus Quelltexten in der Schriftart `Courier` gesetzt. Parameter, die sich aus einem speziellen Kontext ergeben, sind in einem kursiven Font gesetzt. Zum Beispiel:

`% make` *Argument*

Da nicht alle Leser dieses Buches Zugang zum Internet haben, sind auf der beiliegenden CD die Slackware-Distribution 1.2.0 und die deutsche LST-Distribution 1.7 enthalten. Sie

lassen sich, nachdem mit Hilfe der MS-DOS-Programme `GZIP.EXE` und `RAWRITE.EXE` entsprechende Startdisketten erzeugt worden sind, direkt von der CD installieren. Die Autoren möchten sich ausdrücklich bei Patrick J. Volkerding und dem Linux-Support-Team Erlangen, namentlich Ralf Flaxa und Stefan Probst, für die gewiß sehr umfangreiche Arbeit an diesen Distributionen bedanken.

Die CD enthält darüber hinaus den Linux-Kernel Version 1.0.9, die Quellen der im Anhang B erläuterten Programme sowie die Quellen der GNU-C-Bibliothek und der G++-Bibliothek. Darüber hinaus sind Texte aus dem Linux-Documentation-Project und die Internet-RFCs enthalten. Die Dateien sind nicht komprimiert und können unter Linux mit dem `mount`-Kommando in die Verzeichnisstruktur eingebunden werden.

Der Inhalt des Buches entspricht unserem heutigen Wissen über den Linux-Kern 1.0, und dieses Wissen ist mit Sicherheit nicht vollständig. Wir sind für alle Korrekturen, Anregungen, Hinweise und Kommentare dankbar.

Über E-Mail sind wir mit der Adresse `linux@informatik.hu-berlin.de` zu erreichen. Wer keinen E-Mail-Zugang besitzt, kann uns auch schreiben:

Linux-Team
Humboldt-Universität zu Berlin
Institut für Informatik
10099 Berlin

Vorwort von Linus Torvalds zur 1. Auflage

Creating an operating system has been (and still is) an exciting project, and has been made even more rewarding through the extensive (and almost uniformly positive) feedback from users and developers alike.

One of the problems for people wanting to get to know the kernel internals better has been the lack of documentation, and fledgling kernel hackers have had to resort to reading the actual source code of the system for most of the details. While I think that is still a good idea, I'm happy that there now exists more documentation like this explaining about Linux use and internals.

Hope you have a good time with Linux and this book,

Helsinki, 28. 4. 1994 *Linus Torvalds*

Danksagung

Dieses Buch wäre ohne die Arbeit vieler anderer Menschen nicht möglich gewesen. An erster Stelle möchten wir uns bei den Linux-Hackern in der ganzen Welt und natürlich bei Linus Torvalds bedanken. Ein weiterer Dank geht an die *Free Software Foundation* (auch unter dem Namen GNU bekannt). Ohne GNU-Software wäre Linux nicht das, was es ist.

Danken wollen wir auch den Mitarbeitern und Studenten am Institut für Informatik der Humboldt-Universität zu Berlin und am Fachbereich Allgemeine Informatik der Fachhochschule Furtwangen, die uns bei unserer Arbeit unterstützt haben.

Zuletzt noch einen Dank an die unzähligen Korrekturleser, allen voran Ralf Kühnel, deren akribische Korrekturen uns eine große Hilfe waren.

Eine besondere Erwähnung verdient hier auch Martin von Löwis, der uns mit konstruktiven Diskussionen und der Implementierung des WindowsNT-Dateisystem für Linux unterstützte.

Viel Spaß beim Lesen und der Beschäftigung mit Linux!

Berlin/Furtwangen, den 1. 5. 94

Michael Beck, Ulrich Kunitz
Harald Böhme, Robert Magnus
Mirko Dziadzka, Dirk Verworner

Pearson Education

ADDISON-WESLEY

1 Linux — Das Betriebssystem

Linux is obsolete!
Andrew S. Tanenbaum

Linux ist ein frei verfügbares Unix-artiges Betriebssystem. Ursprünglich nur für den PC entwickelt, läuft es heute auch auf Digital-Alpha und Sparc Workstations. Weitere Portierungen z. B. auf Pocket-PCs sind in der Entwicklung und laufen schon relativ stabil.

Linux ist kompatibel zum POSIX-1003.1-Standard und umfasst große Teile der Funktionalität von Unix System V und BSD. Wesentliche Teile des Linux-Kerns, um den es in diesem Buch gehen soll, wurden von Linus Torvalds, einem finnischen Informatikstudenten, entwickelt. Er stellte die Programmquellen des Kerns unter die *GNU Public License*. Damit hat jedermann das Recht, die Programme kostenlos zu benutzen, zu kopieren und zu modifizieren.

Die erste Version des Linux-Kerns war 1991 im Internet verfügbar. Es bildete sich schnell eine Gruppe von Linux-Aktivisten, die die Entwicklung dieses Betriebssystems vorantrieben. Zahlreiche Nutzer testen neue Versionen und tragen dazu bei, die Software fehlerfrei zu machen.

Die Linux-Software wird unter offenen und verteilten Bedingungen entwickelt. Mit „offen" ist gemeint, dass jeder, der dazu in der Lage ist, sich an der Entwicklung beteiligen kann. Das bedeutet, dass die Linux-Aktivisten schnell, effektiv und vor allem weltweit kommunizieren müssen. Das Medium dafür ist das Internet. So verwundert es nicht, dass ein großer Teil der Entwicklungen von begabten Studenten stammen, die an ihren Universitäten und Colleges auf das Internet zugreifen können. Diesen Studenten standen anfangs Entwicklungssysteme mit eher bescheidener Ausstattung zur Verfügung. Aus diesem Grund ist Linux immer noch das 32-Bit-Betriebssystem, das die wenigsten Ressourcen verbraucht, ohne an Funktionalität einzubüßen.

Da Linux unter den Bedingungen der GNU Public License [GPL] verbreitet wird, hat man Zugriff auf den vollständigen Quellcode. Damit kann jeder selbst die Funktionsweisen des Systems erkunden sowie Fehler aufspüren und beseitigen. Der eigentliche Reiz für die Autoren des Buches besteht aber im „Herumexperimentieren" am System.

Linux hat natürlich auch Nachteile. Es ist genauso ein „Programmierersystem" wie Unix. Kryptische Kommandos, schwer überschaubare Konfigurationen und eine nicht immer durchgängige Dokumentation erschweren nicht nur Anfängern die Nutzung. Es scheint aber so, als ob diese Nachteile von Vielen in Kauf genommen werden, um manch anderer Beschränkung (technologischer oder finanzieller Art) proprietärer Systeme wie MS-DOS, Windows oder auch kommerzieller Unix-Derivate für den PC zu entkommen. Mittlerweile gibt es neben dem *Linux Document Project* [LDP] auch viele andere

für Einsteiger brauchbare Bücher zu Linux. Insbesondere auf dem deutschsprachigen Markt hat sich hier vieles getan. Hier sei auf das Literaturverzeichnis verwiesen.

Linux-Systeme werden längst in Softwarehäusern, bei Internet-Providern, in Schulen und Universitäten sowie privat eingesetzt. Mittlerweile gibt es kaum eine Computerzeitschrift mehr, die nicht regelmäßig über dieses Betriebssystem berichtet. Allein auf dem deutschen Linux-Markt werden mehrere Millionen DM pro Jahr umgesetzt. Linux als reines Hackerspielzeug zu bezeichnen, wird der Realität nicht mehr gerecht.

Obwohl es inzwischen Portierungen auf andere Hardwarearchitekturen gibt, benutzen die meisten Nutzer Linux noch auf Intel 386ern oder kompatiblen Systemen. Durch die weite Verbreitung dieser Intel-Systeme hat man unter Linux auch kaum Probleme mit Treibern für Peripheriehardware. Sobald eine neue Steckkarte für den PC auf dem Markt ist, findet sich ein Linux-Nutzer, der dafür einen Treiber implementiert.[1] Seit der Version 2.0 werden auch Mehrprozessorsysteme unterstützt.

Für ein vernünftiges Arbeiten mit Linux sollte ein PC mindestens acht, bei der Benutzung des X-Window-Systems als grafischer Oberfläche mindestens 16 MByte Hauptspeicher enthalten. Mit jeweils der doppelten Menge macht das Arbeiten auch dann noch Spaß, wenn man im Hintergrund mehrere Compiler gleichzeitig laufen lässt und im Vordergrund einen Text bearbeiten will. Für Spezialanwendungen wie Modem/Fax-Server oder Firewalls reichen aber auch schon vier MByte vollkommen aus.

Linux unterstützt im Prinzip jede frei verfügbare Unix-Software. So kann man mit GNU-C++ objektorientiert programmieren oder unter dem X-Window-System Grafiken erstellen. Spiele wie Tetris stehen genauso zur Verfügung wie Entwicklungssysteme für grafische Oberflächen. Durch die Netzwerkunterstützung können Linux-Rechner problemlos in bestehende Netze eingebunden werden. Dieses Buch ist (natürlich) mit LaTeX unter Linux gesetzt worden.

1.1 Wesentliche Eigenschaften

Linux erfüllt alle Anforderungen, die heute an ein modernes, Unix-ähnliches Betriebssystem gestellt werden.

Multitasking Linux unterstützt echtes präemptives Multitasking. Alle Prozesse laufen völlig unabhängig voneinander. Damit braucht kein Prozess dafür Sorge zu tragen, anderen Prozessen Rechenzeit abzugeben.

Multiuser Linux erlaubt es mehreren Nutzern gleichzeitig, mit dem System zu arbeiten.

1 Ausnahmen sind Karten von Herstellern, die Informationen über die Funktionsweise ihrer Hardware geheim halten.

Multiprozessing Linux arbeitet seit der Version 2.0 auch auf Multiprozessor-Architekturen. Das heißt, dass das Betriebssystem mehrere Anwendungen (echt parallel) auf mehrere Prozessoren verteilen kann.

Architekturunabhängig Linux läuft inzwischen auf fast allem, was Bits und Bytes verarbeiten kann. Die unterstützte Hardware reicht von Embedded-Systemen bis zu IBM-Mainframes. Diese Hardwareunabhängigkeit wird von keinem anderen Betriebssystem erreicht.

Demand Load Executables Es werden nur die Teile eines Programms in den Speicher geladen, die auch wirklich zur Ausführung benötigt werden. Bei der Erzeugung eines neuen Prozesses mittels `fork()` wird nicht sofort Speicher für Daten angefordert, sondern der Datenspeicher des Elternprozesses wird von beiden Prozessen gemeinsam genutzt. Greift dann der neue Prozess irgendwann schreibend auf einen Teil des Datenspeichers zu, muss dieser Teil vor der Modifizierung erst kopiert werden. Dieses Konzept wird *Copy-On-Write* genannt.

Paging Trotz aller Maßnahmen, um den physischen Speicher effektiv zu verwenden, kann es vorkommen, dass dieser vollständig belegt ist. Linux sucht dann 4 KByte große Speicherseiten, so genannte *Pages*, die freigemacht werden können. Seiten, deren Inhalt auf Festplatte gespeichert ist (z. B. Code aus Programmdateien), werden verworfen. Alle anderen Seiten werden auf die Festplatte ausgelagert. Wird auf eine dieser Speicherseiten wieder zugegriffen, muss sie wieder zurückgeladen werden. Dieses Verfahren wird *Paging* genannt. Es unterscheidet sich vom *Swapping* älterer Unix-Varianten, bei denen der gesamte Speicher eines Prozesses auf die Festplatte geschrieben wird, was ohne Zweifel wesentlich ineffektiver ist.

Dynamischer Cache für Festplatten MS-DOS-Nutzer kennen das Problem, dass man für ein Festplattencache-Programm wie SMARTDRIVE Speicher mit einer festen Größe reservieren muss. Linux passt die Größe des verwendeten Cache dynamisch an die aktuelle Speicherauslastungssituation an. Ist momentan kein Speicher mehr frei, wird die Größe des Cache reduziert und damit freier Speicher zur Verfügung gestellt. Wird wieder Speicher freigegeben, wird der Cachebereich vergrößert.

Shared Libraries Bibliotheken sind eine Sammlung von Routinen, die ein Programm zur Abarbeitung benötigt. Es gibt eine Reihe von Standardbibliotheken, die von mehreren Prozessen gleichzeitig benutzen werden. Es ist daher naheliegend, den Programmcode für diese Bibliotheken nur einmal in den Speicher zu laden und nicht für jeden Prozess extra. Genau das ist mit *Shared Libraries* möglich. Da diese Bibliotheken erst zur Laufzeit des Programms zu dessen Code hinzugeladen werden, spricht man auch von dynamisch gebundenen Bibliotheken. So ist es kein Wunder, dass in anderen Betriebssytemwelten dieses Konzept als *Dynamic Link Libraries* bekannt ist.

Unterstützung des POSIX-1003.1-Standards, teilweise System V und BSD POSIX 1003.1 definiert eine minimale Schnittstelle zu einem UNIX-ähnlichen Betriebssystem. Mittlerweile wird dieser Standard von allen neueren und anspruchsvollen Betriebssystemen unterstützt. LINUX (ab Version 1.2) unterstützt POSIX 1003.1 vollständig. Mittlerweile gibt es sogar LINUX-Distributionen, die den offiziellen Zertifizierungsprozess durchlaufen haben und sich deshalb auch offiziell „POSIX-kompatibel" nennen dürfen. Zusätzliche Systemschnittstellen der UNIX-Entwicklungslinien System V und BSD wurden auch implementiert. Software, die für UNIX geschrieben wurde, lässt sich deswegen in der Regel ohne weiteres auf LINUX übersetzen.

Verschiedene Formate von ausführbaren Dateien Es ist sicher wünschenswert, Programme, die in anderen Systemumgebungen laufen, unter LINUX auszuführen. Aus diesem Grund werden zur Zeit Emulatoren für MS-DOS und MS-Windows entwickelt. Des Weiteren ist LINUX in der Lage, Programme anderer UNIX-Systeme, die dem iBCS2-Standard entsprechen, auszuführen. Dies trifft zum Beispiel für viele unter SCO-UNIX eingesetzte kommerzielle Programme zu. Auch bei den Portierungen für andere Hardwarearchitekturen (z. B. Sparc und Alpha) wird darauf geachtet, die jeweiligen „native-Binaries" ausführen zu können. So steht dem LINUX-Anwender eine Fülle von kommerzieller Software zur Verfügung, ohne dass diese speziell auf LINUX portiert wurde.

Speicherschutz LINUX benutzt die Speicherschutzmechanismen der Prozessoren, um den Zugriff eines Prozesses auf den Speicher des Systemkerns oder anderer Prozesse zu verhindern. Dies trägt entscheidend zur Sicherheit und Stabilität des Systems bei. Ein fehlerhaftes Programm kann deswegen (theoretisch) das System nicht zum Absturz bringen.

Unterstützung von nationalen Tastaturen und Fonts Unter LINUX kann man mit den unterschiedlichsten nationalen Tastaturen und Zeichensätzen arbeiten. Da der von der internationalen Standardisierungsorganisation (ISO) definierte Zeichensatz *Latin1* auch deutsche Umlaute enthält, ist die Verwendung anderer Zeichensätze in Deutschland nicht unbedingt notwendig.

Verschiedene Dateisysteme LINUX unterstützt verschiedenste Dateisysteme. Das zur Zeit gebräuchlichste Dateisystem ist das zweite erweiterte Dateisystem (*Ext2*-Dateisystem). Es unterstützt Dateinamen mit bis zu 255 Zeichen und hat eine Reihe von Merkmalen, die es gegenüber herkömmlichen UNIX-Dateisystemen sicherer machen. Weitere implementierte Dateisysteme sind das MS-DOS-Dateisystem und das VFAT-Dateisystem für den Zugriff auf MS-DOS bzw. Windows-95-Partitionen, das ISO-Dateisystem für den Zugriff auf CD-ROMs und das NFS für den transparenten Zugriff auf Dateisysteme anderer im Netzwerk befindlicher UNIX-Rechner. Weniger gebräuchlich sind das AFF-Dateisystem für den Zugriff auf das Amiga-Fast-Filesystem, das UFS-Dateisystem und das SysV-Dateisystem für den Zugriff auf UNIX-Filesysteme anderer Hersteller, das HPFS für den Zugriff auf OS/2-Partitionen oder das Samba-Dateisystem für den Zugriff auf exportierte Filesysteme von Windows-Rechnern.

Andere Filesysteme, wie das unter Windows-NT benutzte WindowsNT-Dateisystem sind in Arbeit und als Beta-Versionen verfügbar. Große Verbreitung gewinnen auch Journaling Filesysteme wie z. B. das Reiser-FS, die sich durch ihre kurze Recovery-Zeit für kommerzielle System empfehlen. Welches kommerzielle Betriebssystem kann schon mit einer solchen Vielfalt aufwarten?

TCP/IP, SLIP und PPP-Unterstützung Linux kann in lokale Unix-Netze integriert werden. Im Prinzip können alle Netzwerkdienste, wie das *Network File System* und Remote Login, benutzt werden. SLIP und PPP unterstützen die Nutzung des TCP/IP-Protokolls über serielle Leitungen. Damit ist mit einem Hochgeschwindigkeitsmodem die Einbindung in das Internet über das öffentliche Telefonnetz möglich.

Embedded Linux In letzter Zeit wird Linux mehr und mehr auch für Aufgaben eingesetzt, bei denen es nicht auf die Bedienerfreundlichkeit eines Desktop-Systems ankommt, sondern auf schonenden Umgang mit den verfügbaren Ressourcen. Das ist zum Beispiel in so genannten Embedded-Anwendungen, wie in Industriesteuerungen, Routern, Unterhaltungselektronik und Palmtops der Fall. In den Kern der Version 2.4 sind einige Änderungen eingeflossen, die diesen Anwendungsbereich möglich machen. So kann beispielsweise die Konsole abgeschaltet werden, und es gibt eine Unterstützung für handelsübliche Flash-Memory Hardware (Disk-On-Chip).

1.2 Linux-Distributionen

Zur Installation von Linux ist eine Distribution notwendig. Sie besteht aus einer Boot-Diskette und weiteren Disketten oder einer CD-ROM. Installationsskripten ermöglichen es auch unerfahrenen Benutzern, lauffähige Systeme zu installieren. Vorteilhaft ist, dass viele Softwarepakete schon an Linux angepasst und entsprechend konfiguriert sind, was dem Anwender eine Menge Arbeit erspart. In der Linux-Gemeinde gibt es immer wieder Diskussionen über die Qualität der einzelnen Distributionen. Dabei wird sehr oft übersehen, dass das Zusammenstellen einer solchen Distribution eine sehr umfangreiche und komplexe Aufgabe ist.

Sehr verbreitet sind international die RedHat-, die S.u.S.E.-, die Debian- und die Slackware-Distribution. Welche dieser Distributionen Sie verwenden, ist Geschmackssache. Erhalten können Sie die angesprochenen Distributionen auf FTP-Servern, in Mailboxen, bei Public-Domain-Vertrieben und in einigen Buchhandlungen. Bezugsquellen finden Sie in einschlägigen Fachzeitschriften oder in den Linux-Newsgruppen des Usenet.

Pearson Education

ADDISON-WESLEY

Pearson Education

ADDISON-WESLEY

2 Die Übersetzung des Kerns

Ein System ist alles, was keines hat,
Hardware das, was beim Runterfallen klappert,
und Software das, wovon man logisch erklären kann,
warum es nicht funktioniert.

Johannes Leckebusch

Bevor wir uns in den folgenden Kapiteln genauer mit dem Innenleben des Linux-Kerns beschäftigen, werden wir hier einen Überblick über den Sourcecode und die Übersetzung des Kerns geben.

2.1 Wo finde ich was?

Da die Quelltexte schon einen recht großen Umfang angenommen haben, sind die einzelnen Teile des Kerns in unterschiedlichen Verzeichnissen zu finden.

Die Quellen finden Sie normalerweise unter `/usr/src/linux`. In den folgenden Kapiteln sind die Pfadangaben deshalb immer relativ zu diesem Verzeichnis.

Durch die laufenden Portierungen auf andere Architekturen hat sich die Verzeichnisstruktur in den einzelnen Versionen des Kerns immer wieder geändert. Architekturabhängiger Code befindet sich in den Unterverzeichnissen von **`arch/`**. Dort finden Sie momentan die Verzeichnisse

`arch/alpha/` für die DEC-Alpha-Architektur,

`arch/arm/` für die ARM-Architektur,

`arch/i386/` für die IA-32-Architektur,

`arch/ia64/` für die IA-64-Architektur (Itanium Prozessor),

`arch/m68k/` für die 68000-Architektur und kompatible Prozessoren,

`arch/mips/` und

`arch/mips64` für die MIPS-Architektur,

`arch/parisc/` für die PA-RISC-Architektur,

`arch/ppc/` für die Power-PC-Architektur und

`arch/s390/` für die IBM S390-Architektur,

`arch/sh/` für die SuperH-Architekturs sowie

arch/sparc/ und

arch/sparc64/ für die Portierung auf Sparc Workstations.

Da Linux hauptsächlich auf PCs verwendet wird, werden wir im folgenden nur auf diese Architektur eingehen. Der Kern von Linux ist im Grunde auch nur ein „normales" C-Programm. Es gibt eigentlich nur zwei wesentliche Unterschiede: Die normale Eintrittsfunktion, in C-Programmen als `main(int argc,char *argv[])` bekannt, heißt bei Linux `start_kernel(void)` und bekommt keine Argumente übergeben. Außerdem existiert die Umgebung des „Programms" noch nicht. Aus diesem Grund ist vor dem Aufruf der ersten C-Funktion einiges an Vorarbeit zu leisten. Die Assembler-Quellen, die diese Aufgabe wahrnehmen, befinden sich im Verzeichnis **arch/i386/boot/**. Außerdem konfigurieren sie die Hardware, weshalb dieser Teil sehr maschinenspezifisch ist.

Von einer entsprechenden Assembler-Routine wird der Kern geladen. Dann erfolgt die Installation der Interruptservice-Routinen, der globalen Deskriptortabellen und der Interrupt-Deskriptortabellen, die nur während der Initialisierungsphase benutzt werden. Die Adressleitung A20 wird zugelassen, und der Prozessor schaltet in den Protected Mode.

Das Verzeichnis **init/** enthält alle zum Start des Kerns nötigen Funktionen. Unter anderem befindet sich dort die bereits erwähnte Funktion `start_kernel()`. Ihre Aufgabe ist es, den Kern korrekt zu initialisieren, wobei die übergebenen Boot-Parameter Berücksichtigung finden. Des Weiteren wird der erste Prozess ohne den Systemruf *fork*, sozusagen „von Hand", erzeugt.

In den Verzeichnissen **kernel/** und **arch/i386/kernel/** befindet sich, wie der Name vermuten lässt, der zentrale Teil des Kerns. Dort sind die wichtigsten Systemrufe implementiert. Außerdem wird der Mechanismus definiert, der von allen Systemrufen zum Übergang in den Systemmodus benutzt wird. Weitere wichtige Teile sind das Zeitmanagement (Systemzeit, Timer usw.), der Scheduler, die DMA- und Interruptrequest-Verwaltung sowie die Signalbehandlung.

Die Speicherverwaltung des Kerns befindet sich in den Verzeichnissen **mm/** und **arch/i386/mm/**. Sie umfasst das Anfordern und Freigeben von Speicher im Kern, das Auslagern momentan nicht benutzter Speicherseiten (Paging), das Einblenden von Datei- und Speicherbereichen an bestimmten Adressen (siehe Systemruf *mmap*, Seite 390) und die virtuelle Speicherschnittstelle.

Die virtuelle Dateisystemschnittstelle befindet sich im Verzeichnis **fs/**. In dessen Unterverzeichnissen sind die Implementierungen der konkreten Dateisysteme enthalten, die Linux unterstützt. Die beiden wichtigsten Dateisysteme sind das *Proc*-Dateisystem und das *Ext2*-Dateisystem. Das erste wird für das Systemmanagement benötigt. Das andere ist zur Zeit „das" Standarddateisystem für Linux.

Jedes Betriebssystem benötigt Treiber für die Hardwarekomponenten. Diese befinden sich im Verzeichnis `drivers/` und lassen sich, den Unterverzeichnissen entsprechend, in Gruppen einteilen. Im Einzelnen sind dies unter anderem:

drivers/block/ die Gerätetreiber für blockorientierte Geräte (z. B. Festplatten und CD-ROMs),

drivers/cdrom/ die Gerätetreiber für proprietäre CD-ROM-Laufwerke (keine SCSI- oder IDE-Laufwerke),

drivers/char/ die zeichenorientierten Gerätetreiber,

drivers/i2c/ ein generischer I^2C-Treiber,

drivers/isdn/ die ISDN-Treiber,

drivers/net/ die Treiber für verschiedene Netzwerkkarten,

drivers/pci/ die Ansteuerung des PCI-Busses,

drivers/pnp/ die Ansteuerung von ISA-PNP-Karten,

drivers/sbus/ die Ansteuerung des S-Busses von Sparc-Maschinen,

drivers/scsi/ das SCSI-Interface sowie

drivers/sound/ die Soundkartentreiber und

drivers/usb/ die Treiber für das USB-Subsystem.

Es gibt noch weitere Subsysteme, deren Treiber hier zu finden sind. Einige der hier befindlichen Treiber sind architekturabhängig und gehören eigentlich in das Verzeichnis `arch/*/`, wo sich unter anderem bereits die Emulation der FPU (in **arch/i386/math-emu/**) befindet. Diese kommt auf PCs dann zum Einsatz, wenn keine FPU vorhanden ist.

In **ipc/** sind die Quellen für die klassische Interprozesskommunikation (IPC) nach System V zu finden. Dazu gehören *Semaphore*, *Shared Memory* und *Message Queues* (altdeutsch: Nachrichtenwarteschlangen).

Die Implementierungen verschiedener Netzwerkprotokolle (TCP, ARP usw.) sowie der Code für Sockets der UNIX- und Internet-Domain wurden in das Verzeichnis **net/** eingeordnet. Der Nutzer kann, wie es auch in anderen UNIX-Systemen üblich ist, auf die unteren Protokollschichten (z. B. IP und ARP) zugreifen. Dieser Teil ist wegen seiner Komplexität noch nicht abgeschlossen.

Teile der Funktionalität der Standard-C-Bibliothek sind in **lib/** implementiert, so dass man selbst im Kern so programmieren kann wie in C.

Das wohl wichtigste Verzeichnis für die kernnahe Programmierung ist das Verzeichnis **include/**. Es enthält die kernspezifischen Headerdateien. Hier befinden sich im Verzeichnis **include/asm-i386/** die architekturabhängigen Include-Dateien für Intel-PCs. Für den einfacheren Zugriff verweist der symbolische Link **include/asm/** auf das Verzeichnis der aktuellen Architektur.

Da sich die Headerdateien von Version zu Version ändern können, ist es einfacher, im Verzeichnis **/usr/include/** symbolische Links auf die beiden Unterverzeichnisse **include/**

linux/ und **include/asm/** anzulegen. Durch den Austausch der LINUX-Kernquellen werden somit auch die Headerdateien aktualisiert.

2.2 Die Übersetzung

Das Generieren eines neuen Kerns erfolgt im Wesentlichen in drei Schritten. Zu Beginn wird der Kern mit:

```
# make config
```

konfiguriert. Dabei wird das Bash-Skript `scripts/Configure` gestartet. Es liest die im Architekturverzeichnis befindliche Datei `arch/i386/config.in` ein, in der die Konfigurationsoptionen des Kerns mit ihren Standardbelegungen definiert sind und fragt ab, welche Komponenten in den Kern aufzunehmen sind. Dabei greift die Datei `arch/i386/config.in` auf die Dateien `Config.in` in den Verzeichnissen der einzelnen Subsysteme des Kerns zurück. Komfortablere Konfigurationsskripten kann man mit

```
# make menuconfig
```

für eine menügesteuerte Installation auf der Konsole bzw. mit

```
# make xconfig
```

für eine menügesteuerte Installation unter X-Windows aufrufen.

Während der Konfiguration werden die Dateien `<linux/autoconf.h>` und `.config` erstellt. Mit Hilfe von `.config` wird der Ablauf der Übersetzung gesteuert, während `<linux/autoconf.h>` für die bedingte Kompilierung innerhalb der Kern-Quellen sorgt. Die Datei `.config` wird bei einem erneuten Aufruf von `Configure` benutzt, um die Standardantworten auf die einzelnen Fragen zu bestimmen. Eine erneute Konfiguration liefert also die letzten Werte als Standard zurück. Mittels

```
# make oldconfig
```

werden die Standardwerte ohne weitere Rückfrage übernommen. Somit ist es möglich, eine `.config`-Datei in eine neue LINUX-Version zu übernehmen und den Kern mit derselben Konfiguration zu übersetzen.

Erweiterungspakete für den Kern sollten die Datei `config.in` bzw. `Config.in` um Einträge der Form

```
bool 'PC-Speaker and DAC driver support' CONFIG_PCSP
```

ergänzen, damit sie bei der Konfiguration hinzugefügt oder entfernt werden können. Weitere Möglichkeiten der Konfigurierung des LINUX-Kerns werden im nächsten Abschnitt beschrieben, sind aber im Normalfall nicht erforderlich.

Im zweiten Schritt werden die Abhängigkeiten der Quelltexte neu berechnet. Dies geschieht mit

```
# make depend
```

und stellt einen rein technischen Vorgang dar. Dabei wird die Eigenschaft des GNU-C-Compilers genutzt, Abhängigkeiten für die *Makefile*s erstellen zu können. Diese Abhängigkeiten werden in den `.depend`-Dateien der einzelnen Unterverzeichnisse gesammelt und später in die Makefiles eingefügt. Die eigentliche Übersetzung des Kerns beginnt nun mit einem schlichten Aufruf:

```
# make
```

Danach sollte man die Datei `vmlinux` im obersten Verzeichnis der Quellen vorfinden. Um einen bootbaren Linux-Kern zu erzeugen, muss

```
# make boot
```

aufgerufen werden. Da PCs im Real-Mode booten, kann nur ein komprimierter Kern geladen werden. Daher ist das Ergebnis dieses Kommandos der komprimierte, bootbare Linux-Kern `arch/i386/boot/zImage`. Falls beim Erstellen von zImage eine Fehlermeldung angezeigt wird, dass der Kern zu groß für ein zImage ist, ruft man

```
# make bzImage
```

auf. Dann kommt ein anderer Mechanismus zum Tragen, der es erlaubt, den Linux-Kern sogleich in den Speicher jenseits der 1MByte-Grenze des Real-Modes zu laden. Die erstellte Datei findet sich unter `arch/i386/boot/bzImage`.

Mit Hilfe von `make` können aber auch noch weitere Aktionen ausgeführt werden. So erzeugt das Target `zdisk` nicht nur einen Kern, sondern schreibt diesen auch auf eine Diskette. Das Target `zlilo`[1] kopiert den generierten Kern nach `/vmlinuz`, der alte Kern wird in `/vmlinuz.old` umbenannt. Dann erfolgt die Installation des Linux-Kerns durch den Aufruf des Linux-Loaders (LILO), der jedoch auch vorher konfiguriert werden sollte (siehe Abschnitt D.2.4).

Für Arbeiten an Teilen des Linux-Kerns (zum Beispiel beim Schreiben eines neuen Treibers) ist es nicht nötig, den kompletten Kern neu zu übersetzen bzw. die Abhängigkeiten zu überprüfen. Stattdessen kann man mit

```
# make drivers
```

nur die Quellen im Unterverzeichnis `drivers/`, d.h. die Treiber, übersetzen lassen. Dabei wird kein neuer Kern erstellt. Wollen Sie zusätzlich den Kern neu binden, so sollten Sie

```
# make SUBDIRS=drivers
```

aufrufen. Diese Vorgehensweise ist auch für die anderen Unterverzeichnisse möglich.

Viele nicht in den Kern eingebundene Gerätetreiber und Dateisysteme lassen sich als Module erstellen. Dies geschieht mittels:

1 Es gibt natürlich auch `bzlilo` und `bzdisk` für große Linux-Kerne.

```
# make modules
```

Die dabei erstellten Module können mittels

```
# make modules_install
```

installiert werden. Die Installation erfolgt in den Unterverzeichnissen `drivers/`, `fs/` und `net/` des Verzeichnisses `/lib/modules/`*KernelVersion*`/kernel/`.

2.3 Zusätzliche Konfigurationsmöglichkeiten

Unter besonderen Umständen kann es notwendig sein, innerhalb der Quellen Einstellungen zu ändern. Normalerweise sollten Sie jedoch versuchen, die Konfiguration zur Laufzeit nicht in den Kernelquellen zu ändern.

Im Folgenden werden die Dateien des Linux-Kerns beschrieben, in denen man ab und zu Änderungen vornehmen muss.

`Makefile` Dies ist die einzige Datei, an der sich Änderungen nicht vermeiden lassen, wenn man keinen „Standard-PC" hat. Die Variable

```
ARCH := …
```

wird auf die Hardwarearchitektur gesetzt, auf der der Kern laufen soll. Dies geschieht normalerweise durch ein Skript, das die Architektur aus den Daten des aktuellen Systems übernimmt.

Mögliche Werte für `ARCH` sind im Moment `alpha`, `sparc`, `m68k`, `arm`, `mips` und `ppc`.

Wenn Sie einen Linux-Kern für eine andere Zielarchitektur übersetzen wollen, kann über die Variable `CROSS_COMPILE` der Pfad für den entsprechenden Compiler gesetzt werden.

`drivers/char/serial.c` Im Normalfall hat man mit den seriellen Schnittstellen keine Probleme, da die meisten PCs nur zwei davon besitzen und diese standardmäßig die IRQs 4 (COM1) und 3 (COM2) nutzen. Hat man aufgrund einer speziellen Hardware (z. B. internes Modem oder Faxkarte) mehr serielle Schnittstellen, können optional die automatische IRQ-Erkennung sowie die Unterstützung von diversen Spezialkarten (AST-Fourport-Karte u.a.) zugeschaltet werden. Dort befindet sich auch eine Erklärung dieser und weiterer Makros. Die meisten Optionen stehen auch in der Datei `drivers/char/Config.in` und können bei `make config` eingestellt werden.

`drivers/net/Space.c` Diese Datei enthält die anfängliche Konfiguration der Netzwerkgeräte. So können die konstant definierten `device`-Strukturen `eth1_dev`, … geändert werden.

```
static struct device eth1_dev = {
/* NAME RECVMEM MEM I/O-BASE IRQ FLAGS NEXT_DEV   INIT       */
  "eth1", 0,0,  0,0, ETH_NOPROBE_ADDR , 0, 0,0,0, &eth2_dev,
  ethif_probe};
```

Die I/O-Adresse `ETH_NOPROBE_ADDR` bedeutet hier, dass dieses Gerät nicht auf Vorhandensein getestet wird. Sie können dies vermeiden, indem Sie eine Null für den automatischen Test oder die entsprechende I/O-Adresse angeben. Über den Bootparameter

`ether=`*irq,port,mem_start,mem_end,name*

lassen sich die Einstellungen beim Hochfahren des Systems nachträglich beeinflussen.

`include/linux/fs.h` Für LINUX-Rechner in sehr großen Netzwerken kann es notwendig sein, mehr als 256 Dateisysteme zu verwalten. Die Anzahl der Dateisysteme wird aber durch das Präprozessormakro `NR_SUPER` auf 256 eingeschränkt. Diese Einschränkung kann hier geändert werden.

Die Konfigurationsmöglichkeiten des LINUX-Kerns sind damit sicher nicht erschöpft. In den folgenden Kapiteln werden an der einen oder anderen Stelle weitere Möglichkeiten beschrieben.

Abschließend muss nochmals darauf hingewiesen werden, dass die in diesem Abschnitt beschriebenen Änderungen der Kern-Quellen meist nicht notwendig sind und nur im Bedarfsfall durchgeführt werden sollten.

Pearson Education

ADDISON-WESLEY

3 Einführung in den Kern

Dijkstra probably hates me.
Linus Torvalds

In diesem Kapitel soll der grundlegende Aufbau des Systemkerns und das Zusammenspiel der wichtigsten Komponenten im Mittelpunkt stehen. Es ist Grundlage für das Verständnis der weiteren Kapitel. Bevor es jedoch so richtig losgeht, noch einige allgemeine Bemerkungen zum Linux-Kern.

Linux ist nicht auf dem Reißbrett entstanden, sondern hat sich evolutionär entwickelt und entwickelt sich noch weiter. Jede Funktion des Kerns wurde mehrfach geändert und erweitert, um Fehler zu beheben und neue Features einzubauen. Wer selbst schon an soeinem großen Projekt gearbeitet hat, der weiß, wie schnell Programmcode unübersichtlich und fehlerhaft werden kann. Linus Torvalds hat es als Koordinator des Linux-Projektes geschafft, den Kern übersichtlich zu gestalten und immer wieder von alten Überbleibseln zu säubern.

Trotzdem ist der Linux-Kern sicherlich nicht in allen Punkten ein gutes Beispiel für strukturiertes Programmieren. Es gibt „Magic-Numbers" im Programmtext statt Konstantendeklarationen in Headerfiles, inline expandierte Funktionen statt Funktionsaufrufen, `goto`-Anweisungen statt eines einfachen `break`, Assembleranweisungen statt C-Code und viele andere Unschönheiten mehr. Viele dieser Merkmale unstrukturierten Programmierens wurden jedoch bewusst eingearbeitet. Große Teile des Systemkerns sind zeitkritisch, deswegen wurde der Programmcode auf gutes Laufzeitverhalten und nicht auf gute Lesbarkeit optimiert. Das unterscheidet Linux zum Beispiel von Minix (siehe [Tan90]), welches als „Lehrbetriebssystem" geschrieben wurde und nie für den täglichen Einsatz gedacht war. Im Gegensatz dazu ist Linux jedoch ein „richtiges" Betriebssystem, und für ein solches ist der Kern bemerkenswert gut strukturiert.

Ziel unseres Buches ist es, die prinzipielle Funktionsweise des Linux-Kerns zu erläutern. Deswegen stellen die in diesem und in den nächsten Kapiteln vorgestellten Algorithmen einen Kompromiss zwischen den Original-Quelltexten und einem gut lesbaren Programmcode dar, wobei darauf geachtet wurde, dass die Veränderungen leicht nachvollziehbar sind.

Allgemeine Architektur Seit den Anfängen von Unix hat sich die interne Struktur von Betriebssystemen stark geändert. Damals war es revolutionär, dass der größte Teil des Kerns in einer höheren Programmiersprache, C, geschrieben wurde. Heute ist so etwas selbstverständlich. Der aktuelle Trend geht in Richtung einer Mikrokernel-Architektur, wie zum Beispiel dem Mach-Kern (vgl. [Tan86]) oder auch dem Kern von Windows-NT. Auch das Experimental-Unix Minix (vgl. [Tan90]) und das sich in Entwicklung befindliche Hurd-System sind Mikrokernel-basiert. Der eigentliche Kern stellt dabei nur das

absolut notwendige Minimum an Funktionalität (Interprozesskommunikation und Speicherverwaltung) zur Verfügung und kann deswegen klein und kompakt implementiert werden. Auf diesen Mikrokernel aufbauend, wird die restliche Funktionalität des Betriebssystems in eigenständige Prozesse ausgelagert, die mit dem Mikrokernel über eine wohldefinierte Schnittstelle kommunizieren. Der große Vorteil dieser Architekturen ist (neben einer gewissen Eleganz) eine auf den ersten Blick wartungsfreundlichere Struktur des Systems. Einzelne Komponenten arbeiten unabhängig voneinander, können sich nicht ungewollt beinflussen und sind leichter austauschbar. Die Entwicklung neuer Komponenten wird vereinfacht.

Daraus ergibt sich auch ein Nachteil dieser Architekturen. Mikrokernel-Architekturen erzwingen die Einhaltung der definierten Schnittstellen zwischen den einzelnen Komponenten und erschweren damit trickreiche Optimierungen. Außerdem ist die im Mikrokernel benötigte Interprozesskommunikation auf heutigen Hardware-Architekturen aufwendiger als einfache Funktionsaufrufe. Das System wird dadurch etwas langsamer als traditionelle monolithische Kerne. Dieser leichte Geschwindigkeitsnachteil wird gern in Kauf genommen, da die heutige Hardware in der Regel schnell genug ist und da der Vorteil der einfacheren Wartbarkeit des Systems die Entwicklungskosten senkt. Erst in den letzten Jahren wurden Mikrokernel Systeme gebaut, deren Performance es mit monolitischen Systemen aufnehmen kann. Dies ist aber noch ein Bereich aktiver Grundlagenforschung.

Mikrokernel-Architekturen repräsentieren sicherlich die Zukunft der Betriebssystementwicklung. Linux hingegen entstand auf der „langsamen“ 386-Architektur, der unteren Grenze für ein vernünftiges Unix-System. Gutes Laufzeitverhalten durch Ausreizen aller Optimierungsmöglichkeiten stand bei der Entwicklung mit im Vordergrund. Das ist ein Grund dafür, warum Linux in der klassischen monolithischen Kernarchitektur realisiert wurde. Ein weiterer Grund ist sicherlich, dass eine Mikrokernel-Architektur ein sorgfältiges Systemdesign bedingt. Da Linux evolutionär, aus Spaß am Systementwickeln, entstanden ist, war dies einfach nicht möglich.

Trotz des monolithischen Ansatzes ist Linux keine chaotische Ansammlung von Programmcode. Die meisten Komponenten des Kerns werden nur über sauber definierte Schnittstellen angesprochen. Ein gutes Beispiel hierfür ist das Virtuelle Dateisystem (VFS), welches eine abstrakte Schnittstelle zu allen dateiorientierten Operationen darstellt. Auf das VFS gehen wir im Kapitel 6 näher ein. Das Chaos findet sich eher im Detail. An zeitkritischen Stellen sind Programmteile oftmals in „handoptimiertem“ C-Code oder gar in Assembler geschrieben und damit schwer zu verstehen. Zum Glück sind diese Programmfragmente selten und in der Regel recht gut kommentiert.

Wenn man sich die Codegrößen der einzelnen Komponenten des Linux-Kernels ansieht, stelt man fest, dass der überwiegende Teil auf Gerätetreiber und Ähnliches entfällt. Die zentralen Routinen zur Prozess- und Speicherverwaltung (also der eigentliche Kern im Sinne einer Mikrokernel-Architektur) sind dagegen mit jeweils 13.000 Zeilen C-Code relativ klein und überschaubar.

Inzwischen ist es möglich, einen Großteil der Treiber aus dem Kern auszulagern. Sie können als eigenständige, unabhängige Module (siehe Kapitel 9) bei Bedarf zur Laufzeit

nachgeladen werden. Linux versucht damit erfolgreich, die Vorteile einer Mikrokernelarchitektur zu nutzen, ohne den monolithischen Entwurf aufzugeben.

Prozesse und Tasks Aus der Sicht eines unter Linux ablaufenden Prozesses stellt sich der Kern als Anbieter von Dienstleistungen dar. Einzelne Prozesse existieren unabhängig nebeneinander und können sich nicht direkt beeinflussen. Der eigene Speicherbereich ist vor dem Zugriff fremder Prozesse geschützt.

Die interne Sicht auf ein laufendes Linux-System ist etwas anders. Auf dem Rechner läuft nur ein Programm — das Betriebssystem —, welches auf alle Ressourcen zugreifen kann. Die verschiedenen Tasks werden durch Coroutinen realisiert, d.h., jede Task entscheidet selbst, ob und wann sie die Steuerung an eine andere Task abgibt.[1] Eine Konsequenz daraus ist, dass ein Fehler in der Kernprogrammierung das ganze System blockieren kann. Jede Task kann auf alle Ressourcen anderer Tasks zugreifen und diese modifizieren.

Bestimmte Teile einer Task laufen in einem weniger privilegierten Nutzermodus des Prozessors ab. Diese Teile der Task erscheinen nach außen hin (in der externen Sicht auf den Kern) als Prozesse. Aus Sicht dieser Prozesse findet ein echtes Multitasking statt. Die Abbildung 3.1 soll das verdeutlichen.

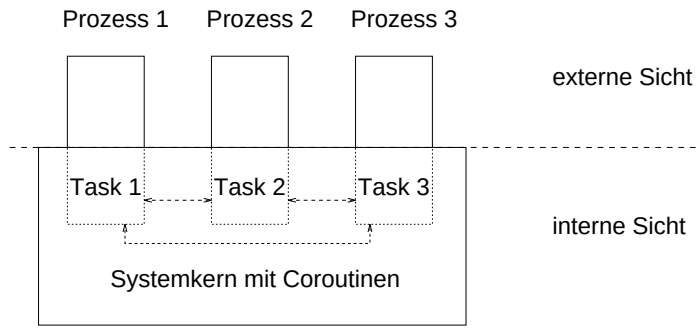


Abbildung 3.1: Verhältnis von interner und externer Sicht auf die Prozesse

Im Folgenden wollen wir allerdings auf eine exakte Unterscheidung der Begriffe *Task* und *Prozess* verzichten und diese Begriffe synonym gebrauchen. Dieses einfache Prozessmodell wird allerdings insofern erweitert, dass es auch Threads geben kann, die nur im Kernelmodus existieren. Wenn sich eine Task im privilegierten Systemmodus befindet, kann sie verschiedene Zustände annehmen. Abbildung 3.2 zeigt die wichtigsten dieser Zustände. Die Pfeile geben die möglichen Zustandsübergänge in diesem Diagramm an.

Die folgenden Zustände sind möglich:

In Ausführung Die Task ist aktiv und befindet sich im nichtprivilegierten Nutzermodus. In dem Fall arbeitet der Prozess ganz normal das Programm ab. Dieser Zustand kann nur durch einen Interrupt oder einen Systemruf verlassen werden. In Abschnitt

1 Dies wird auch als kooperatives Multitasking bezeichnet.

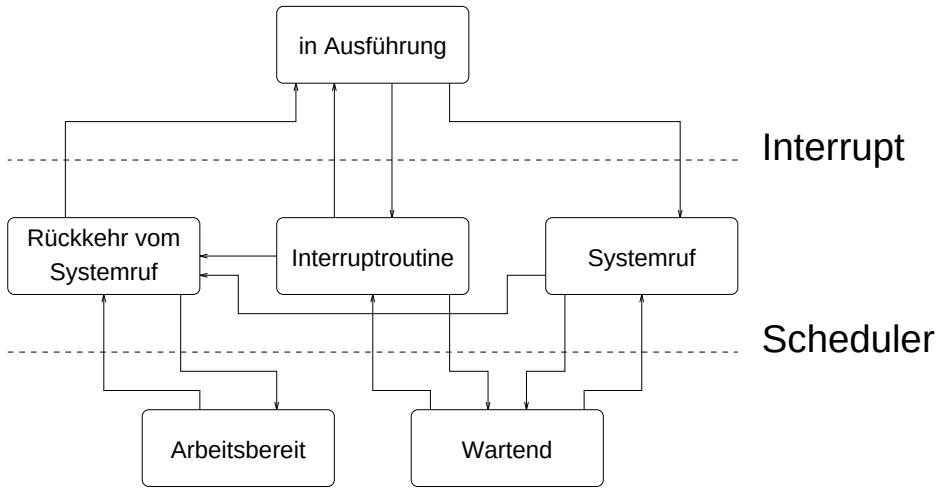


Abbildung 3.2: Zustandsgraph eines Prozesses

3.3 werden wir sehen, dass Systemrufe auch nur Spezialfälle von Interrupts sind. In jedem Fall wird der Prozessor in den privilegierten Systemmodus geschaltet und die entsprechende Interruptroutine aktiviert.

Interruptroutine Die Interruptroutinen werden aktiv, wenn die Hardware eine Ausnahmebedingung signalisiert, sei es, dass neue Zeichen an der Tastatur anliegen oder dass der Zeitgeberbaustein alle 10 Millisekunden ein Signal sendet. Weitere Informationen zu Interruptroutinen sind in Abschnitt 3.2.2 enthalten.

Systemruf Systemrufe werden durch softwaremäßig ausgelöste Interrupts eingeleitet. Nähere Informationen dazu finden Sie in Abschnitt 3.3. Ein Systemruf hat die Möglichkeit, seine Arbeit explizit zu unterbrechen, um auf ein Ereignis zu warten.

Wartend Der Prozess wartet auf ein externes Ereignis. Erst nachdem dieses eingetreten ist, setzt der Prozess seine Arbeit fort.

Rückkehr vom Systemruf Dieser Zustand wird automatisch nach jedem Systemaufruf und nach einigen Interrupts erreicht. Hier wird geprüft, ob der Scheduler aufgerufen werden muss und ob Signale zu behandeln sind. Der Scheduler kann den Prozess in den Zustand „arbeitsbereit" überführen und einen anderen Prozess aktivieren.

Arbeitsbereit Der Prozess bewirbt sich um den Prozessor, dieser ist aber im Moment von einem anderen Prozess belegt.

Prozesse und Threads In vielen modernen Betriebssystemen gibt es die Unterscheidung von Prozessen und Threads. Ein Thread ist dabei ein unabhängiger „Faden" im Ablauf eines Programmes, der parallel zu anderen Threads abgearbeitet werden kann. Im Unterschied zu Prozessen arbeiten die Threads über demselben Hauptspeicher und können sich so gegenseitig beeinflussen.

Linux macht diese Unterscheidung nicht. Im Kern gibt es nur den Begriff der Task, diese kann sich mit anderen Tasks Ressourcen (wie zum Beispiel denselben Speicher) teilen. Damit ist die Task eine Verallgemeinerung des sonst üblichen Thread-Konzeptes. Näheres dazu finden Sie im Abschnitt 3.3.3.

Multiprozessorsysteme Seit der Version 2.0 unterstützt Linux SMP (*Symmetric Multi Prozessing*). Während in der Version 2.0 die Implementation anfänglich noch trivial war — es konnte zu einem Zeitpunkt nur ein Prozessor Kernelcode abarbeiten —, ist sie mittlerweile ziemlich komplex geworden. Mehrere Prozessoren können jetzt gleichzeitig Kernelcode abarbeiten. Deswegen muss der Zugriff auf alle globalen Datenstrukturen des Kernels synchronisiert werden. Die dadurch auftretenden Probleme haben wir in diesem Kapitel größtenteils ignoriert, um die Beschreibung verständlich zu halten.

3.1 Wichtige Datenstrukturen

In diesem Kapitel werden wichtige Datenstrukturen des Linux-Kerns beschrieben. Das Verständnis dieser Strukturen und ihres Zusammenspiels ist Voraussetzung für das Verständnis der weiteren Kapitel.

3.1.1 Die Taskstruktur

Einer der wichtigsten Begriffe in einem Multitaskingsystem wie Linux ist die *Task*. Die Datenstrukturen und Algorithmen zur Prozessverwaltung sind der zentrale Kern von Linux.

Die Beschreibung der Eigenschaften eines Prozesses erfolgt in der Struktur `task_struct`, welche im Folgenden erläutert wird. Auf die ersten Komponenten der Struktur wird auch aus Assemblerroutinen heraus zugegriffen. Dieser Zugriff erfolgt nicht wie in C üblich über die Namen der Komponenten, sondern über ihren Offset relativ zum Anfang der Struktur. Deswegen darf man den Anfang der Taskstruktur auch nicht modifizieren, ohne dass vorher alle Assemblerroutinen überprüft und gegebenenfalls angepasst werden.

```
struct task_struct
{
        volatile long state;
```

`state` enthält eine Codierung für den aktuellen Zustand des Prozesses. Wenn der Prozess auf die Zuteilung der CPU wartet oder gerade läuft, hat `state` den Wert `TASK_RUNNING`. Wartet der Prozess dagegen auf bestimmte externe Ereignisse und ist deswegen im Moment stillgelegt, hat `state` den Wert `TASK_INTERRUPTIBLE` oder `TASK_UNINTERRUPTIBLE`. Der Unterschied zwischen diesen Werten besteht darin, dass im Zustand `TASK_INTERRUPTIBLE` ein Prozess durch Signale wieder aktiviert werden kann, während er im Zustand `TASK_UNINTERRUPTIBLE` in der Regel direkt oder indirekt auf eine Hardwarebedingung wartet und damit keine Signale akzeptiert. `TASK_STOPPED` beschreibt einen Prozess, dessen Ausführung angehalten worden ist.

Dies ist entweder nach dem Empfang eines entsprechenden Signales (SIGSTOP, SIGSTP, SIGTTIN oder SIGTTOU) der Fall, oder wenn der Prozess von einem anderen Prozess durch den Systemruf *ptrace* überwacht wird und die Steuerung an den überwachenden Prozess übergeben hat. TASK_ZOMBIE beschreibt einen Prozess, der beendet wurde, dessen Taskstruktur sich aber noch in der Prozesstabelle befinden muss (vgl. Systemrufe *_exit* und *wait* in Abschnitt 3.3.3). Das Schlüsselwort volatile deutet an, dass diese Komponente auch asynchron aus Interruptroutinen heraus geändert werden kann.

```
unsigned long flags;
```

flags enthält die Kombination der Statusflags PF_ALIGNWARN, PF_STARTING, PF_EXITING, PF_FORKNOEXEC, PF_SUPERPRIV, PF_DUMPCORE, PF_SIGNALED, PF_MEMALLOC, PF_VFORK und PF_USEDFPU.

Diese Flags werden im Wesentlichen für das Accounting von Prozessen benutzt und haben auf die Arbeitsweise des Systems keinen weiteren Einfluss.

Das in älteren Kerneln vorhandene Statusflag PF_TRACED wurde in die Komponente

```
unsigned long ptrace;
```

ausgelagert. Die Werte PF_PTRACED und PF_TRACESYS zeigen hier an, dass der Prozess von einem anderen Prozess mit Hilfe des Systemrufes *ptrace* überwacht wird. Nähere Informationen zu diesem Systemruf findet der interessierte Leser in Abschnitt 5.4 und Anhang A.

```
int sigpending;
```

Das Flag sigpending ist gesetzt, wenn an diesen Prozess Signale ausgeliefert werden müssen. Näheres dazu findet sich im Abschnitt 3.2.1.

```
mm_segement_t addr_limit;
```

Im Gegensatz zu älteren Kerneln kann es seit Version 2.4 auch Tasks (Threads) innerhalb des Kernels geben. Diese dürfen selbstverständlich auf einen größeren Adressraum zugreifen als Tasks im Userspace. addr_limit beschreibt den Adressraum, auf dem der Kernel der Tasks Zugriff ermöglicht.

```
struct exec_domain *exec_domain;
```

Linux kann Programme anderer UNIX-Systeme auf i386-Basis, die dem iBCS2-Standard entsprechen, abarbeiten. Da sich die verschiedenen iBCS2-Systeme leicht unterscheiden, wird für jeden Prozess in der Komponente exec_domain eine Beschreibung mitgeführt, welches UNIX für diesen Prozess emuliert werden soll.

```
long need_resched;
```

need_resched ist ein Flag, welches anzeigt, dass eine Scheduling durchgeführt werden muss. In der Kernelversion 2.0 war dies noch eine globale Variable, aus Effizienzgründen ist diese jetzt in der Taskstruktur der aktuellen Task abgelegt.

Für den Betrieb auf Multiprozessorsystemen muss die ganze Struktur vor dem gleichzeitigen Zugriff geschützt werden. Das Locking wird über die Komponente

```
int lock_depth;
```

realisiert.

Damit endet der hartcodierte Teil der Taskstruktur. Die folgenden Komponenten der Taskstruktur sind der Übersicht halber zu Gruppen zusammengefasst.

```
long counter;
long nice;
unsigned long policy; /* SCHED_FIFO, SCHED_RR, *
                       * SCHED_OTHER           */
unsigned long rt_priority;
```

`counter` enthält die Zeit in „Ticks“ (siehe Abschnitt 3.2.5), die der Prozess noch laufen kann, ehe zwangsweise ein Scheduling durchgeführt wird. Der Scheduler benutzt den Wert in `counter`, um den nächsten Prozess auszuwählen. Damit stellt `counter` so etwas wie die dynamische Priorität eines Prozesses dar. `nice` enthält die statische Priorität des Prozesses. Bis zur Kernelversion 2.2 hatte diese Komponente den Namen `priority` Der Schedulingalgorithmus (siehe Abschnitt 3.2.6) verwendet `nice`, um im Bedarfsfall einen neuen Wert für `counter` zu ermitteln.

Linux unterstützt inzwischen mehrere Scheduling-Algorithmen. Neben dem klassischen Scheduling (`SCHED_OTHER`) gibt es jetzt auch noch zwei in POSIX.4 beschriebene Real-Time-Scheduling-Algorithmen (`SCHED_RR` und `SCHED_FIFO`). Jeder Prozess kann in eine dieser Schedulingklassen eingeordnet werden. Diese wird, zusammen mit der Real-Time-Priorität, in den Komponenten `policy` und `rt_priority` vermerkt. Näheres dazu im Abschnitt 3.2.6.

Signale

```
/* int sigpending */
sigset_t blocked;
struct signal_struct *sig;

struct sigpending pending;
```

`sigpending` enthält, wie oben bereits beschrieben, eine Bitmaske der für den Prozess eingetroffenen Signale und `blocked` eine Bitmaske aller Signale, die der Prozess erst später bearbeiten möchte, d.h. deren Bearbeitung im Moment blockiert ist. Die Komponente `sig` enthält Verweise auf die entsprechenden Signalbehandlungsroutinen.

Linux unterstützt sogenannte „verlässliche Signale“ oder auch „Realtime-Signale“ nach POSIX.4. Diese können nicht wie normale Unix-Signale in einem Bitfeld verwaltet werden; der Kernel muss dafür sorgen, dass ein mehrfach gesendetes Signal den Empfänger auch mehrfach erreicht. Linux versucht dieses Verhalten auch für traditionelle Signale zu implementieren. Jedes an den Prozess gesendete Signal wird deswegen in einer Liste `pending` vermerkt. Die Auswertung dieser Signalinformation ist im Abschnitt 3.2.1 beschrieben.

Prozessrelationen Alle Prozesse sind mit Hilfe der folgenden beiden Komponenten in eine doppelt verkettete Liste eingetragen.

```
struct task_struct *next_task;
struct task_struct *prev_task;
```

Den Anfang und das Ende dieser Liste enthält die globale Variable `init_task`.

Prozesse existieren in einem UNIX-System nicht unabhängig voneinander. Jeder Prozess (außer dem Prozess `init_task`) hat einen Elternprozess, der ihn mit Hilfe des Systemrufs *fork()* (siehe Abschnitt 3.3.3 und Anhang A) erzeugt hat. Daraus ergeben sich Verwandtschaftsbeziehungen zwischen den Prozessen, die durch die folgenden Komponenten repräsentiert werden:

```
struct task_struct *p_opptr; /* original parent */
struct task_struct *p_pptr;  /* parent */
struct task_struct *p_cptr;  /* youngest child */
struct task_struct *p_ysptr; /* younger sibling */
struct task_struct *p_osptr; /* older sibling */
```

`p_pptr` ist ein Zeiger auf die Taskstruktur des Elternprozesses. Damit ein Prozess auf alle seine Kinderprozesse zugreifen kann, enthält die Taskstruktur den Eintrag für den zuletzt erzeugten Kindprozess — das „jüngste" Kind (*youngest child*). Die Kindprozesse desselben Elternprozesses sind untereinander wiederum durch `p_ysptr` (*younger sibling* = nächstjüngeres Kind) und `p_osptr` (*older sibling* = nächstälteres Kind) als doppelt verkettete Liste verbunden. Die Abbildung 3.3 versucht, die Verwandtschaftsbeziehungen zwischen Prozessen etwas zu verdeutlichen. Der Scheduler benutzt eine Liste aller Prozesse, die sich um den Prozessor bewerben. Diese wird über die Komponente

```
struct list_head run_list;
```

realisiert.

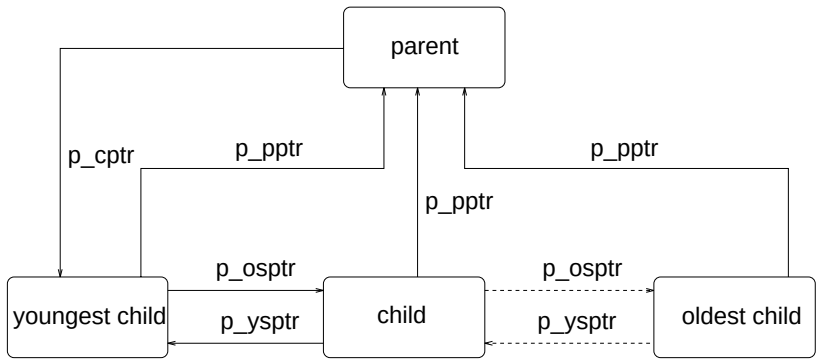


Abbildung 3.3: Verwandtschaftsbeziehungen von Prozessen

Speicherverwaltung Die für jeden Prozess notwendigen Daten zur Speicherverwaltung werden aus Gründen der Übersichtlichkeit in einer eigenen Unterstruktur

```
struct mm_struct *mm;
```

zusammengefasst. Diese hat die Komponenten:

```
unsigned long start_code, end_code, start_data, end_data;
unsigned long start_brk, brk,
unsigned long start_stack,start_mmap;
unsigned long arg_start, arg_end, env_start, env_end;
…
```

welche den Beginn und die Größe der Code- und Datensegmente für das aktuell laufende Programm beschreiben. Weitergehende Informationen sind in Kapitel 4 enthalten.

Prozessidentifikation Jeder Prozess besitzt seine eindeutige Prozessidentifikationsnummer `pid`, ist einer Prozessgruppe `pgrp` und einer Sitzung `session` zugeordnet. Jede Sitzung hat einen führenden Prozess (`leader`). Da unter Linux auch Threads durch eine eigene Task realisiert werden, wurde mit `tgid` die Threadgroup-Identifikationsnummer eingeführt. Es ist dies in der Regel die `pid` des Prozesses, aus dem heraus neue Threads gestartet werden. Im klassischen Sinne ist dies also die wahre PID.

```
pid_t pid, pgrp, session, tgid;
int leader;
```

Zur Realisierung von Zugriffsrechten besitzt jeder Prozess eine Nutzeridentifikation (*User Identification*) `uid` und die Gruppenidentifikation (*Group Identification*) `gid`. Diese werden beim Erzeugen eines neuen Prozesses durch den Systemruf *fork* (siehe Abschnitt 3.3.3 und Anhang A) vom Elternprozess an den Kindprozess vererbt. Für die eigentliche Zugriffskontrolle werden allerdings die effektive Nutzeridentifikation `euid` und die effektive Gruppenidentifikation `egid` benutzt. Eine Neuerung in Linux ist die Komponente `fsuid`. Diese wird bei allen Identifikationen gegenüber dem Dateisystem benutzt. Normalerweise gilt `(uid==euid)&&(gid==egid)` und `(fsuid==euid)&&(fsgid==egid)`.

Ausnahmen ergeben sich bei sogenannten Set-UID-Programmen, bei denen die Werte `euid` und `fsuid` bzw. `egid` und `fsgid` auf die Nutzer-ID bzw. die Gruppen-ID des Eigentümers der ausführbaren Datei gesetzt wird. Dadurch ist eine kontrollierte Vergabe von Privilegien möglich.

Normalerweise hat `fsuid` immer den Wert von `euid`, und in anderen Unix-Systemen oder in älteren Linux-Versionen wurde anstelle von `fsuid` auch immer die effektive Nutzeridentifikation `euid` benutzt. Linux erlaubt hingegen durch den Systemruf *setfsuid* das Ändern der `fsuid` ohne Änderung der `euid`. Damit können Dämonen mit *setfsuid* ihre Rechte in Bezug auf Dateisystemzugriffe einschränken (auf die Rechte des Nutzers, für den Dienstleistungen erbracht werden), behalten aber ihre Privilegien bei. Analoges gilt für die Komponente `fsgid` und den Systemruf *setfsgid*.

```
uid_t uid,euid,suid,fsuid;
gid_t gid,egid,sgid,fguid;
```

Linux erlaubt, wie die meisten modernen Unix-Derivate, die gleichzeitige Zuordnung eines Prozesses zu mehreren Nutzergruppen. Diese Gruppen werden bei der Kontrolle der Zugriffsrechte auf Dateien berücksichtigt. Jeder Prozess kann maximal `NGROUPS` Gruppen angehören, die in der Komponente `groups` der Taskstruktur abgespeichert werden.

```
gid_t groups[NGROUPS];
int ngroups;
```

Traditionell sind in einem Unix-System viele Aktionen dem Superuser vorbehalten. Dieser wird daran erkannt, dass seine effektive UID 0 ist. Nur mit dieser EUID kann ein Prozess zum Beispiel eine Netzwerkverbindung auf einem priviligierten Port eröffnen, ein Signal an einen fremden Prozess senden oder das System rebooten. Dieses Konzept der Privilegien ist relativ grob und führte in der Vergangenheit zu vielen Sicherheitsproblemen in Unix-Systemen. Linux hat zusätzlich zum Konzept „Superuser" noch das Konzept der „Capabilities" eingeführt. Damit ist es möglich, einem Prozess zum Beispiel gezielt das Recht einzuräumen, einen reservierten Netzwerkport zu eröffnen, ohne ihm gleich Superuserprivilegien zu gewähren. Diese Rechte werden in den Komponenten

```
kernel_cap_t cap_effective;
kernel_cap_t cap_inheritable;
kernel_cap_t cap_permitted;

int keep_capabilities:1;
```

verwaltet und im Kern bereits Stellen benutzt. Eine Liste der möglichen Werte für diese Komponenten ist in `include/linux/capability.h` zu finden. Obwohl der Linux-Kern inzwischen durchgängig auf Capabilities umgestellt ist, werden diese von den Verwaltungsprogrammen und den üblichen Dateisystemen noch nicht unterstützt. Für die Zukunft zeichnet sich hier aber eine interessante Entwicklung zu sichereren Systemen ab.

Files Die dateisystemspezifischen Informationen sind in der Unterstruktur

```
struct fs_struct *fs;
```

abgelegt. Diese enthält unter anderem die Komponenten

```
atomic_t count;
int umask;
struct dentry * root, * pwd;
```

Ein Prozess kann über den Systemruf *umask* den Zugriffsmodus von neu zu erzeugenden Dateien beeinflussen. Die mit dem Systemruf *umask* gesetzten Werte werden dazu in der Komponente `umask` abgelegt. Unter Unix besitzt jeder Prozess ein aktuelles Verzeichnis `pwd`[2], welches bei der Auflösung relativer Pfadnamen benötigt wird und mit dem Systemruf *chdir* geändert werden kann. Jeder Prozess verfügt weiterhin über ein

2 Die Abkürzung `pwd` steht hier höchstwahrscheinlich in Anlehnung an das Unix-Kommando `pwd` (*print working directory*), welches den Namen des aktuellen Verzeichnisses ausgibt.

eigenes Wurzelverzeichnis `root`, das zum Auflösen absoluter Pfadnamen benutzt wird. Dieses Root-Verzeichnis kann nur vom Superuser geändert werden (Systemruf *chroot*). Da dies nur in wenigen Fällen genutzt wird (z. B. anonymous FTP), ist diese Tatsache weniger bekannt. `count` ist ein Referenzzähler, da diese Struktur von mehreren Tasks benutzt werden kann.

Ein Prozess, der eine Datei mit `open()` oder `creat()` eröffnet, erhält vom Kern einen Dateideskriptor für die weitere Referenzierung dieser Datei. Dateideskriptoren sind kleine Integerzahlen. Die Zuordnung der Dateideskriptoren zu den Dateien erfolgt unter Linux über das Feld `fd` in der Unterstruktur

```
struct files_struct *files;
```

Sie hat unter anderem folgende Komponenten

```
atomic_t count;
int max_fds;
struct file ** fd;
fd_set *close_on_exec;
fd_set *open_fds;
```

Dateideskriptoren werden als Index im Feld `fd` benutzt. Man findet auf diese Weise den dem Dateideskriptor zugeordneten Filepointer, mit dessen Hilfe man dann auf die Datei zugreifen kann. `open_fds` ist eine Bitmaske aller benutzten Filedeskriptoren.

Die Komponente `close_on_exec` in der Unterstruktur `files` enthält eine Bitmaske aller benutzten Filedeskriptoren, die beim Systemruf *exec* geschlossen werden sollen. `count` dient wieder als Referenzzähler; `max_fds` ist die Maximalanzahl von offenen Filedescriptoren für den Prozess.

Zeitmessung Für jeden Prozess werden mehrere unterschiedliche Zeiten gemessen. Die Zeitmessung wird unter Linux grundsätzlich in Ticks vorgenommen. Diese Ticks werden von einem Zeitgeberbaustein der Hardware alle 10 Millisekunden erzeugt und vom Timerinterrupt gezählt. In den Abschnitten 3.1.6 und 3.2.5 gehen wir genauer auf die Zeitmessung unter Linux ein.

`per_cpu_utime[]` und `per_cpu_stime[]` enthalten die Zeit, die der Prozess im Nutzermodus bzw. im Systemmodus verbracht hat. Diese Daten werden in einem Multiprozessorsystem für jede CPU einzeln erhoben. Die Summe dieser Werte über alle CPUs sowie die Summe der entsprechenden Zeiten aller Kindprozesse sind in der Komponente `times` abgelegt. Die Werte können mit Hilfe des Systemrufs *times* abgefragt werden.

```
long per_cpu_utime[NR_OF_CPUS];
long per_cpu_stime[NR_OF_CPUS];
struct tms times;
unsigned long start_time;
```

`start_time` enthält den Zeitpunkt, zu dem der aktuelle Prozess erzeugt wurde.

Unix unterstützt verschiedene prozessspezifische Timer. Zum einem gibt es den Systemruf *alarm*, der dafür sorgt, dass dem Prozess nach einer bestimmten Zeit das Signal `SIGALRM` gesandt wird. Neuere Unix-Systeme unterstützen zusätzlich Intervalltimer (siehe Systemrufe *setitimer* und *getitimer* auf Seite 331).

```
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
```

Die Komponenten `it_real_value`, `it_prof_value` und `it_virt_value` enthalten die Zeitspanne in Ticks, nach der der Timer abgelaufen ist. In den Komponenten `it_real_incr`, `it_prof_incr` und `it_virt_incr` befinden sich die notwendigen Werte, um den Timer nach Ablauf wieder zu initialisieren. `real_timer` dient der Realisierung des Realzeit-Intervalltimers. Näheres dazu findet sich auch bei der Beschreibung des Timerinterrupts in Abschnitt 3.2.5.

Interprozesskommunikation Im Linux-Kern ist eine zum System V kompatible Interprozesskommunikation implementiert, die unter anderem Semaphore zur Verfügung stellt. Ein Prozess kann einen Semaphor belegen und damit sperren. Wenn andere Prozesse diesen Semaphor auch belegen wollen, werden sie so lange gestoppt, bis der Semaphor freigegeben wird. Dazu dient die Komponente

```
struct sem_queue *semsleeping;
```

Wenn der Prozess beendet wird, muss das Betriebssystem alle vom Prozess belegten Semaphore freigeben. Die Komponente

```
struct sem_undo *semundo;
```

enthält die dazu notwendigen Informationen.

Verschiedenes Die folgenden Komponenten lassen sich nicht in die obigen Gruppen einordnen.

```
wait_queue_head_t wait_chldexit;
```

Ein Prozess, der den Systemruf *wait4* ausführt, soll bis zur Beendigung eines Kindprozesses unterbrochen werden. Dazu trägt er sich in die Warteschlange `wait_chldexit` seiner eigenen Taskstruktur ein, setzt sein Statusflag auf den Wert `TASK_INTERRUPTIBLE` und gibt die Steuerung an den Scheduler ab. Wenn ein Prozess beendet wird, signalisiert er dies seinem Elternprozess über diese Warteschlange. Näheres findet sich im Abschnitt über Warteschlangen (Abschnitt 3.1.5), im Abschnitt zu den Systemrufen *_exit* und *wait* (Abschnitt 3.3.3) sowie in den Quelltexten zur Kernelfunktion `sys_wait4()` (`kernel/exit.c`).

```
struct semaphore *vfork_sem;
```

Die Semantik des Systemaufrufes *vfork* verlangt, dass der Elternprozess mit der Weiterarbeit solange warten muss, bis der Kindprozess entweder beendet wurde oder über *exec*

ein anderes Programm geladen hat. Dieses Warten wird über die Semaphore `vfork_sem` realisiert. Näheres findet sich im Abschnitt 3.3.3

```
struct rlimit rlim[RLIM_NLIMITS];
```

Jeder Prozess kann mit Hilfe der Systemrufe *setrlimit* und *getrlimit* (siehe Seite 332) seine Limits für die Verwendung von Ressourcen kontrollieren. Sie werden in der Struktur `rlim` abgespeichert.

```
int exit_code, exit_signal;
```

Dies sind der Return-Code des Programms und das Signal, mit dem das Programm abgebrochen wurde. Diese Informationen kann ein Elternprozess nach dem Ende eines Kindprozesses abfragen.

```
char comm[16];
```

Der Name des vom Prozess ausgeführten Programms ist in der Komponente `comm` gespeichert. Dieser Name wird für das Debugging benutzt.

```
unsigned long personality;
```

Wie schon erwähnt, unterstützt Linux über das iBCS-Interface das Abarbeiten von Programmen anderer Unix-Systeme. Zusammen mit der oben beschriebenen Komponente `exec_domain` dient `personality` der genauen Beschreibung der Eigenarten dieser Unix-Version. Für normale Linux-Programme hat `personality` den Wert `PER_LINUX` (definiert in `<linux/personality.h>` als 0)

```
int dumpable:1;
int did_exec:1;
```

Das Flag `dumpable` gibt an, ob vom laufenden Prozess beim Eintreffen bestimmter Signale ein Speicherabzug erzeugt werden soll.

Eine ziemlich obskure Semantik im POSIX-Standard erfordert beim Systemruf *setpgid* die Unterscheidung, ob ein Prozess noch das ursprüngliche Programm ausführt oder schon mit dem Systemruf *execve* ein neues Programm geladen hat. Diese Information wird im Flag `did_exec` verwaltet.

Eine weitere wichtige Komponente in der Taskstruktur ist `binfmt`. Sie beschreibt die Funktionen, die für das Laden des Programms zuständig sind.

```
struct linux_binfmt *binfmt;
struct thread_struct thread;
```

Die `thread_struct`-Struktur enthält alle Informationen über den aktuellen Prozessorstatus zum Zeitpunkt des letzten Übergangs vom Nutzermodus in den Systemmodus. Hier sind alle Prozessorregister gerettet, damit sie bei der Rückkehr in den Nutzermodus wieder restauriert werden können.

3.1.2 Die Prozesstabelle

Jeder Prozess belegt genau einen Eintrag in der Prozesstabelle. Sie war bis Linux 2.2 statisch angelegt und in der Größe auf `NR_TASKS` (512) Tasks beschränkt. In der aktuellen Version ist die Prozesstabelle nur noch eine Abstraktion. Die einzelnen Tasks sind über die in der Struktur `task_struct` vorhandenen Verkettungen `next_task` und `prev_task` erreichbar.

Das Makro `init_task` zeigt auf die erste Task im System. Sie wird beim Starten des Systems (in Abschnitt 3.2.4 beschrieben) mit Hilfe des Makros `INIT_TASK` initialisiert. Diese ist nach dem Booten des Systems nur noch für den Verbrauch nichtbenötigter Systemzeit verantwortlich (Idle-Prozess). Sie fällt deswegen etwas aus dem Rahmen und sollte nicht als normale Task angesehen werden.

Obwohl die Prozesstabelle eine rein dynamische Struktur hat, wird die Anzahl der Tasks im System auf `max_threads` begrenzt.

```
int max_threads;
```

Dieser Wert kann jedoch zur Laufzeit über das *sysctl* Interface geändert werden.

Viele Algorithmen im Kern müssen jede einzelne Task berücksichtigen. Um dies zu erleichtern, wurde das Makro `for_each_task()` wie folgt definiert:

```
#define for_each_task(p)        \
   for( p = &init_task ; ( p = p->next_task) != &init_task ; )
```

Wie man sieht, wird die `init_task` übergangen. Der Eintrag für die aktuell laufende Task war in Version 1 noch über die globale Variable

```
struct task_struct *current;
```

zu erreichen. Da seit der Version 2.0 Multiprocessing (SMP) unterstützt wird, musste dies erweitert werden — es gibt jetzt für jeden Prozessor eine aktuelle Task.

```
#define current get_current()
inline struct task_struct * get_current(void)
{
    struct task_struct *current;
    __asm__("andl %%esp,%; ":"=r" (current) : "0" (~8191UL));
    return current;
}
```

Die Funktion `get_current()` ist etwas magisch, die Task-Struktur ist aus Effizienzgründen im Stacksegment der aktuellen Task untergebracht.

3.1.3 Files und Inodes

Unix-Systeme unterscheiden traditionell zwischen einer File-Struktur und einer Inode-Struktur. Die Inode-Struktur beschreibt eine Datei. Dabei ist der Begriff *Inode* mehrfach

belegt. Sowohl die Datenstruktur im Kern als auch die Datenstrukturen auf der Festplatte beschreiben (jede aus ihrer Sicht) eine Datei und werden deswegen Inodes genannt. Wir reden im Folgenden immer von der im Hauptspeicher liegenden Datenstruktur. Inodes enthalten Informationen, wie etwa den Eigentümer und die Zugriffsrechte der Datei. Zu jeder im System benutzten Datei gibt es *genau einen* Inode-Eintrag im Kern.

File-Strukturen (die Datenstrukturen vom Typ `struct file`) enthalten dagegen die Sicht eines Prozesses auf diese (durch eine Inode repräsentierte) Datei. Zu dieser Sicht auf die Datei gehören Attribute, wie etwa der Modus, in dem die Datei benutzt werden kann (read, write, read+write), oder die aktuelle Position der nächsten I/O-Operation.

File-Struktur Die Struktur `file` ist in `include/linux/fs.h` definiert.

```
struct file
{
        mode_t f_mode;
        loff_t f_pos;

        atomic_t f_count;
        unsigned int f_flags;

        struct dentry *fs_dentry;
        struct file_operations * f_op;

        …
};
```

`f_mode` beschreibt den Zugriffsmodus, in dem die Datei eröffnet wurde (nur Lesen, Lesen und Schreiben oder nur Schreiben). `f_pos` enthält die Position des Schreib-Lese-Zeigers, an der die nächste I/O-Operation vorgenommen wird. Dieser Wert wird durch jede I/O-Operation sowie durch die Systemrufe *lseek* und *llseek* aktualisiert. Man beachte, dass der Offset im Kern als 64-Bit-Wert vom Typ `loff_t` gespeichert wird. Damit ist Linux in der Lage, Dateien größer als 2 Gigabyte (2^{31} Byte) korrekt zu behandeln.

`f_flags` enthält zusätzliche Flags, die den Zugriff auf diese Datei steuern. Diese können entweder beim Eröffnen einer Datei mit dem Systemruf *open* gesetzt und später mit dem Systemruf *fcntl* gelesen und modifiziert werden. `f_count` ist ein einfacher Referenzzähler. Mehrere Dateideskriptoren können auf dieselbe File-Struktur verweisen. Da diese durch den Systemruf *fork* vererbt werden, kann auch aus verschiedenen Prozessen auf dieselbe File-Struktur verwiesen werden. Beim Eröffnen einer Datei wird `f_count` mit 1 initialisiert. Jedes Kopieren des Dateideskriptors (durch die Systemrufe *dup*, *dup2* oder *fork*) erhöht den Referenzzähler um 1, während er bei jedem Schließen einer Datei (durch die Systemrufe *close*, *_exit* oder *exec*) um 1 dekrementiert wird. Die File-Struktur wird erst freigegeben, wenn kein Prozess mehr auf sie verweist.

`f_dentry` ist ein Verweis auf einen Eintrag im Verzeichniscache, welcher alle offenen Dateien enthält. Über diesen Eintrag kann auch auf die Inode (die eigentliche Beschreibung der Datei) zugegriffen werden. `f_op` verweist auf eine Struktur von Funktions-

pointern, die alle File-Operationen referenzieren. Linux unterstützt im Vergleich zu anderen Unix-Systemen sehr viele Dateisystemtypen. Jedes Dateisystem realisiert die Zugriffe auf eine andere Art. Deswegen ist in Linux ein „Virtuelles Dateisystem“ (VFS) realisiert worden. Die Idee besteht darin, dass die auf dem Dateisystem operierenden Funktionen nicht direkt, sondern über eine datei(system)spezifische Funktion aufgerufen werden. Diese dateisystemspezifischen Operationen sind Teil der Struktur `file` bzw. `inode`. Das entspricht dem Prinzip virtueller Funktionen in objektorientierten Programmiersprachen. Ausführlichere Informationen zum VFS finden sich in Abschnitt 6.2.

Inodes Die Inode-Struktur

```
struct inode
{
```

ist ebenfalls in `include/linux/fs.h` definiert. Viele Komponenten dieser Struktur können über den Systemruf *stat* abgefragt werden.

```
        kdev_t i_dev;
        unsigned long i_ino;
```

`i_dev` ist eine Beschreibung des Geräts (die Plattenpartition), auf der sich die Datei befindet. `i_ino`[3] identifiziert die Datei innerhalb des Geräts. Das Paar (`i_dev`, `i_ino`) beschreibt die Datei damit systemweit eindeutig.

```
        umode_t i_mode;
        uid_t i_uid;
        gid_t i_gid;
        off_t i_size;
        time_t i_mtime;
        time_t i_atime;
        time_t i_ctime;
```

Diese Komponenten beschreiben die Zugriffsrechte der Datei, ihre Eigentümer (Nutzer und Gruppe), die Größe `i_size` in Byte, die Zeiten der letzten Änderung `i_mtime`, des letzten Zugriffs `i_atime` sowie der letzten Änderung der Inode `i_ctime`.

```
        struct inode_operations * i_op;
        …
```

Wie die File-Struktur besitzt auch die Inode einen Verweis auf eine Struktur, die Zeiger auf Funktionen enthält, die auf Inodes anwendbar sind (siehe Abschnitt 6.2.7). Weitere Informationen zur Inode befinden sich in Abschnitt 6.2.

3.1.4 Dynamische Speicherverwaltung

Unter Linux wird der Speicher seitenweise verwaltet. Eine Seite umfasst 2^{12} Bytes. Grundoperationen zum Anfordern neuer Seiten sind die Funktionen

3 `i_ino` steht hier auch für die Inode. Damit ist in diesem Fall die Blocknummer der Datenstruktur auf der Festplatte gemeint, welche die Datei auf dem externen Speicher beschreibt.

```
struct page * __alloc_pages(int gfp_mask, unsigned long order);
unsigned long __get_free_pages(int gfp_mask
                unsigned long order);
```

welche in der Datei `mm/page_alloc.c` definiert sind. `gfp_mask` gibt Aukunft darüber, wer zu welchem Zweck die Seiten anfordert und steuert darüber das Verhalten der Funktionen, wenn im Hauptspeicher zuwenig Seiten frei sind. Ein Userprozess kann dann zum Beispiel warten, bis wieder Speicher frei wird, dies ist einer Interruptroutine nicht möglich. Für `gfp_mask` sind dabei die Werte `GFP_BUFFER`, `GFP_ATOMIC`, `GFP_USER`, `GFP_KERNEL`, `GFP_KSWAPD` und `GFP_NFS` zulässig.

`order` beschreibt die Anzahl der zu reservierenden Seiten. Es werden 2^{order} viele Seiten reserviert.

Angeforderte Seiten können mit den Funktionen

```
void __free_pages(struct page *page, unsigned long order);
void free_pages(unsigned long addr, unsigned long order);
```

wieder freigegeben werden. Dadurch werden die Seiten wieder in die Freispeicherliste eingetragen.

Obwohl dies die Grundoperation für die Anforderung einer Seite darstellt, sollte man sie in dieser Form nicht benutzen. Besser geeignet ist die Funktion

```
unsigned long get_zeroed_page(int gfp_mask);
```

Sie initialisiert den angeforderten Speicher zusätzlich mit 0. Das ist aus zwei Gründen wichtig. Erstens erwarten einige Teile des Kerns, dass neu angeforderter Speicher mit 0 initialisiert ist (z. B. der Systemruf *exec*). Andererseits handelt es sich um eine Sicherheitsmaßnahme. Wenn die Seite vorher schon benutzt wurde, enthält sie vielleicht Daten eines anderen Nutzers (z. B. Passwörter), die dem aktuellen Prozess nicht zugänglich gemacht werden sollen.

Normalerweise ist der C-Programmierer an den Umgang mit `malloc()` und `free()` zur Speicherverwaltung gewohnt. Etwas Ähnliches gibt es auch im Linux-Kern. Die Funktion

```
void *kmalloc(size_t size, int flags);
```

arbeitet analog zu `malloc()`. Das Argument `flags` gibt, ähnlich wie bei `get_zeroed_page()` an, wie `kmalloc()` neue Speicherseiten anfordern soll. Das Gegenstück zu `kmalloc()` ist die Funktion

```
void kfree( const void * objp);
```

welche einen mit `kmalloc()` angeforderten Speicherbereich wieder freigibt.

Weitere Informationen zur Funktionsweise der Speicherverwaltung unter Linux findet der Leser in Kapitel 4.

3.1.5 Warteschlangen und Semaphore

Oftmals ist ein Prozess vom Eintreten bestimmter Bedingungen abhängig. Sei es, dass der Systemruf *read* darauf warten muss, dass die Daten von der Festplatte in den Speicherbereich des Prozesses geladen werden, oder dass ein Elternprozess mit *wait* auf das Ende eines Kindprozesses wartet. In jedem dieser Fälle ist nicht bekannt, wie lange ein Prozess warten muss.

Dieses „Warten bis eine Bedingung erfüllt ist" wird in Linux mit Hilfe der Warteschlangen (*Waitqueues*) realisiert. Eine Warteschlange ist im Prinzip nichts anderes als eine zyklische Liste, welche als Elemente Zeiger auf Tasks enthält.

```
struct list_head {
        struct list_head *next, *prev;
};

typedef struct __wait_queue {
        struct task_struct * task;
        struct list_head task_list;
} wait_queue_t;

typedef struct __wait_queue_head {
        struct list_head task_list;
} wait_queue_head_t;
```

Zyklische Listen sind eine grundlegende Datenstruktur, deswegen wurde mit der Struktur `list_head` und den dazugehörigen Funktionen, eine einheitliche Implementation für sie geschaffen.

Der Datentyp `wait_queue_t` beschreibt ein Element einer Warteschlange und `wait_queue_head_t` die Warteschlange selbst.

Warteschlangen sollten nur mit Hilfe der folgenden beiden Funktionen modifiziert werden. Diese sorgen durch entsprechenden Locking dafür, dass der Zugriff auf die Warteschlangen synchronisiert erfolgt.

```
void add_wait_queue(wait_queue_head_t *q,
                wait_queue_t *wait);

void remove_wait_queue(wait_queue_head_t *q,
                wait_queue_t *wait);
```

`q` enthält jeweils die zu modifizierende Warteschlange und `wait` den hinzuzufügenden oder zu entfernenden Eintrag.

Ein Prozess, der auf ein bestimmtes Ereignis warten will, trägt sich nun in eine solche Warteschlange ein und gibt die Steuerung ab. Zu jedem möglichen Ereignis gibt es eine Warteschlange. Wenn das entsprechende Ereignis eintritt, werden alle Prozesse in dieser Warteschlange wieder aktiviert und können weiterarbeiten. Diese Semantik wird durch die Funktionen

```
void sleep_on(wait_queue_head_t *q);
void sleep_on_timeout(wait_queue_head_t *q, long timeout);
void interruptible_sleep_on(wait_queue_head_t *q);
void interruptible_sleep_on_timeout(wait_queue_head_t *q,
                                    long timeout);
```

realisiert. Sie setzen den Status des Prozesses (`current->state`) auf den Wert `TASK_UNINTERRUPTIBLE` beziehungsweise `TASK_INTERRUPTIBLE`, tragen den aktuellen Prozess (`current`) in die Warteschlange ein und rufen den Scheduler auf. Damit gibt der Prozess die Steuerung freiwillig ab.

Er wird erst wieder aktiviert, wenn der Prozessstatus auf `TASK_RUNNING` gesetzt wurde. Das geschieht in der Regel dadurch, dass ein anderer Prozess eines der Makros

```
void wake_up(struct wait_queue **p);
void wake_up_interruptible(struct wait_queue **p);
```

aufruft, um alle in der Warteschlange eingetragenen Prozesse zu wecken.

Mit Hilfe von Warteschlangen stellt Linux auch Semaphore bereit. Diese dienen der Synchronisation der Zugriffe verschiedener Routinen des Kerns auf gemeinsam benutzte Datenstrukturen. Diese Semaphore sind nicht mit den für Anwenderprogramme bereitgestellten Semaphoren von Unix System V zu verwechseln.

```
struct semaphore {
        atomic_t count;
        wait_queue_head_t wait;
        …
};
```

Ein Semaphor gilt als belegt, wenn `count` einen Wert kleiner oder gleich 0 hat. In die Warteschlange tragen sich alle Prozesse ein, die den Semaphor belegen wollen. Sie werden dann benachrichtigt, wenn er von einem anderen Prozess freigegeben wird. Zum Belegen oder Freigeben von Semaphoren gibt es zwei Hilfsfunktionen:

```
void down( struct semaphore * sem )
{
    while( sem -> count <= 0 )
        sleep_on( & sem->wait );
    sem -> count -= 1;
}

void up( struct semaphore * sem )
{
    sem -> count += 1;
    wake_up( & sem -> wait );
}
```

Die reale Implementation dieser Funktionen ist aus Effizienzgründen in Assemblercode ausgeführt und erheblich komplizierter.

3.1.6 Systemzeit und Zeitgeber (Timer)

Im Linux-System gibt es genau eine interne Zeitbasis. Sie wird in vergangenen Ticks seit dem Starten des Systems gemessen. Ein Tick entspricht dabei 10 Millisekunden. Diese Ticks werden von einem Zeitgeberbaustein der Hardware generiert und vom Timerinterrupt (siehe Abschnitt 3.2.5) in der globalen Variablen `jiffies` gezählt. Alle im Folgenden genannten Systemzeiten beziehen sich immer auf diese Zeitbasis.

Wofür braucht man Timer? Viele Gerätetreiber möchten eine Meldung erhalten, wenn das Gerät nicht bereit ist. Andererseits muss bei der Bedienung eines langsamen Gerätes vielleicht etwas gewartet werden, ehe die nächsten Daten gesendet werden können.

Um dies zu unterstützen, bietet Linux die Möglichkeit, Funktionen zu einem definierten zukünftigen Zeitpunkt zu starten. Dafür gibt es das Interface der Form:

```
struct timer_list {
        struct list_head list;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
};
```

Der Eintrag `list` in dieser Struktur dient der internen Verwaltung der Timer in einer doppelt verketteten Liste. Die Komponente `expires` gibt den Zeitpunkt an, zu dem die Funktion `function` mit dem Argument `data` aufgerufen werden soll. Die Funktionen

```
extern void add_timer(struct timer_list * timer);
extern int  del_timer(struct timer_list * timer);
extern int  mod_timer(struct timer_list * timer,
                      unsigned long expires);
```

dienen der Verwaltung einer globalen Timerliste. `add_timer()` aktiviert einen Timer durch Eintrag in die globale Timerliste; `del_timer()` entfernt ihn wieder, und `mod_timer()` ändert den `expire`-Zeitpunkt eines aktivierten Timers.

Der Timerinterrupt (siehe Abschnitt 3.2.5) ruft regelmäßig die Funktion

```
static inline void run_timer_list(void);
```

auf, die nach abgelaufenen Timern sucht und die zugehörigen Funktionen aufruft.

3.2 Zentrale Algorithmen

In diesem Kapitel werden zentrale Algorithmen der Prozessverwaltung beschrieben.

3.2.1 Signale

Eine der ältesten Möglichkeiten der Interprozesskommunikation unter Unix sind Signale. Der Kern benutzt Signale, um Prozesse über bestimmte Ereignisse zu informieren.

Der Anwender benutzt Signale in der Regel, um Prozesse abzubrechen oder interaktive Programme in einen definierten Zustand zu überführen. Prozesse ihrerseits benutzen Signale, um sich mit anderen Prozessen zu synchronisieren.

Normalerweise werden Signale durch die Funktion

```
int send_sig_info(int sig,
                  struct siginfo *info,
                  struct task_struct *t);
```

versandt.

Sie erhält als Argumente neben der Signalnummer `sig` und einem Pointer auf die Task, die das Signal empfangen soll, noch einen Parameter `info`, welcher den Sender identifiziert. Der Kern darf jedem Prozess ein Signal senden, ein normaler Nutzerprozess darf dies nur unter bestimmten Bedingungen. Er muss dafür entweder Superuser-Rechte besitzen oder dieselbe UID und GID wie der Empfängerprozess haben. Eine Ausnahme bildet das Signal `SIGCONT`. Dieses darf auch von einem beliebigen Prozess derselben Sitzung (*Session*) gesendet werden.

Wenn die Berechtigung zum Senden des Signals vorliegt, und der Prozess dieses Signal nicht ignorieren möchte, wird es dem Prozess geschickt. Dies geschieht, indem das Signal in der Komponente `pending` der Taskstruktur des empfangenden Prozesses eingetragen wird. Damit einfach zu überprüfen ist, ob für einen Prozess Signale vorliegen, wird ebenfalls die Komponente `sigpending` der Taskstruktur gesetzt.

Damit ist das Signal gesendet. Es erfolgt noch keine Behandlung des Signals durch den empfangenden Prozess. Dies geschieht erst, wenn der Scheduler den Prozess wieder in den Zustand `TASK_RUNNING` versetzt (vgl. Abschnitt 3.2.6).

Wenn der Prozess vom Scheduler wieder aktiviert wird, wird vor dem Umschalten in den Nutzermodus die Routine `ret_from_sys_call` (Abschnitt 3.3.1) abgearbeitet. Diese prüft, ob Signale für den aktuellen Prozess vorliegen und bearbeitet werden müssen. Dies ist der Fall, wenn das Flag `sigpending` in der Taskstruktur des Prozesses gesetzt ist. Wenn dies der Fall ist, wird die Funktion `do_signal()` aufgerufen, welche die eigentliche Signalbehandlung übernimmt. Für den Fall, dass beim Eintreffen eines Signals eine nutzerdefinierte Funktion aufgerufen werden soll, ruft `do_signal` die Funktion `handle_signal()` auf.

Offen ist noch die Frage, wie diese Funktion für den Aufruf der vom Prozess definierten Signalbehandlungsroutine sorgt. Das wurde trickreich gelöst. Die Funktion `handle_signal()` manipuliert den Stack und die Register des Prozesses. Der Instruction-Pointer des Prozesses wird auf den ersten Befehl der Signalbehandlungsroutine gesetzt. Des Weiteren wird der Stack um die Parameter der Signalbehandlungsroutine ergänzt. Wenn nun der Prozess seine Arbeit fortsetzt, sieht es für ihn so aus, als ob die Signalbehandlungsroutine wie eine normale Funktion aufgerufen wurde.

Das ist die prinzipielle Vorgehensweise, die allerdings in der realen Implementierung um zwei Punkte erweitert wird.

Zum einen erhebt Linux den Anspruch, POSIX-kompatibel zu sein. Der Prozess kann angeben, welche Signale während der Abarbeitung einer Signalbehandlungsroutine blockiert werden sollen. Das wird dadurch realisiert, dass der Kern vor dem Aufruf der nutzerdefinierten Signalbehandlungsroutine die Signalmaske `current->blocked` um weitere Signale ergänzt. Ein Problem besteht nun darin, dass die Signalmaske nach der Abarbeitung der Signalbehandlungsroutine wieder restauriert werden muss. Deswegen wird als Return-Adresse der Signalbehandlungsroutine auf dem Stack ein Befehl eingetragen, der den Systemruf *sigreturn* aufruft. Dieser übernimmt dann die Aufräumarbeiten nach dem Beenden der nutzerdefinierten Signalbehandlungsroutine.

Die zweite Erweiterung ist eine Optimierung. Müssen mehrere Signalbehandlungsroutinen aufgerufen werden, so werden auch mehrere Stackframes angelegt. Damit werden dann die Signalbehandlungsroutinen direkt hintereinander ausgeführt.

3.2.2 Hardwareinterrupts

Zur Kommunikation der Hardware mit dem Betriebssystem werden Interrupts verwendet. Auf die Programmierung von Interruptroutinen werden wir in Abschnitt 7.3.2 näher eingehen. Hier interessieren wir uns mehr für den prinzipiellen Ablauf beim Aufruf eines Interrupts.

Beim Entwurf einer Interruptroutine steht der Programmierer vor einem Problem. Zum einen muss die eigentliche Interruptroutine die Hardware so schnell wie möglich bedienen und dann den Prozessor für andere Aufgaben, zum Beispiel für die Bearbeitung weiterer Interrupts, wieder freigeben. Zum anderen kann so ein Interrupt aber auch die Verarbeitung einer größeren Datenmenge auslösen.

Um dieses Problem zu lösen, erfolgt die Interruptbehandlung unter Linux in zwei Phasen. Zuerst wird die zeitkritische Kommunikation mit der Hardware durchgeführt, hierbei sind eventuell andere Interrupts gesperrt. Die eigentliche Verarbeitung der Daten erfolgt asynchron zum eigentlichen Interrupt. Dafür werden entweder „Softwareinterrupts“, „Tasklets“ oder „Bottom-Halfs“ verwendet. Dies sind Funktionen, die zu einem späteren Zeitpunkt aufgerufen werden. Sie können ihrerseits auch wieder von anderen Interrupts unterbrochen werden.

Die zentrale Behandlungsroutine für Hardwareinterrupts sieht vereinfacht so aus:

```
unsigned int do_IRQ(struct pt_regs regs) {
    int irq;
    struct irqaction * action;

    /* irq nummer aus den registern entnehmen */
    irq = regs.orig_eax & 0xff;

    /* entsprechende handler finden */
    action = irq_desc[irq].action;

    /* und die Aktionen ausführen */
```

```
    while ( action )
    {
        action->handler(irq, regs)
        action = action->next;
    }

    /* hier ist der eigentliche Hardwareinterrupt beendet. */

    if( softirq_active & softirq_mask)
        do_softirq();
}
```

3.2.3 Softwareinterrupts

Linux 2.4 führt das Konzept der Softwareinterrupts ein. Ein Softwareinterrupt ist wie ein Hardwareinterrupt ein Ereignis, das ausgelöst werden kann und das zum Abarbeiten von Interruptbehandlungsroutinen führt. Nur werden diese nicht wie Hardwareinterrupts sofort gestartet, sondern nur zu bestimmten Zeitpunkten. Konkret passiert dies nach jedem Hardwareinterrupt und nach jedem Systemcall.

Wie auch bei Hardwareinterrupts ist die Anzahl von Softwareinterrupts begrenzt

```
enum { HI_SOFTIRQ=0,
       NET_TX_SOFTIRQ, NET_RX_SOFTIRQ,
       TASKLET_SOFTIRQ
};
static struct softirq_action softirq_vec[32];
```

`HI_SOFTIRQ` ist dabei der Softwareinterrupt mit der höchsten Priorität, `NET_TX_SOFTIRQ` und `NET_RX_SOFTIRQ` werden vom Netzwerkcode benutzt, und `TASKLET_SOFTIRQ` ist der Softwareinterrupt, über den die Abarbeitung der Tasklets realisiert wird.

Die Registrierung eines Interrupthandler erfolgt über die Funktion `open_softirq()`. Ein Aufruf von `raise_softirq()` sorgt dafür, dass die registrierte Behandlungsroutine beim nächsten Aufruf von `do_softirq()` ausgeführt wird.

```
void open_softirq(int nr,
                  void (*action)(struct softirq_action*),
                  void *data);
raise_softirq(int nr);
void do_softirq();
```

Zu beachten ist, dass in einem Multiprozessorsystem durchaus derselbe Interrupthandler gleichzeitig auf verschiedenen Prozessoren abgearbeitet werden kann, die Funktionen müssen also reentrant sein oder explizit eine Serialisierung beim Zugriff auf gemeinsame Ressourcen implementieren.

Etwas einfacher als Softwareinterrupts kann mit Tasklets gearbeitet werden. Hier wird vom System garantiert, dass ein bestimmtes Tasklet zu einem Zeitpunkt immer nur einmal ausgeführt wird, verschiedene Tasklets können allerdings auch parallel abgearbeitet werden.

Die Registrierung eines Tasklets erfolgt durch die Funktion `tasklet_init()`. Mit `tasklet_schedule()` wird ein Tasklet zur Abarbeitung markiert und der Softwareinterrupt `TASKLET_SOFTIRQ` ausgelöst. Dessen Interruptroutine arbeitet dann die Tasklets ab.

```
struct tasklet_struct;

void tasklet_init(struct tasklet_struct *t,
                  void (*func)(unsigned long),
                  unsigned long data);

void tasklet_schedule(struct tasklet_struct *t);
```

Softwareinterrupts und Tasklets sind neu in Linux 2.4, sehr lange gibt es dagegen schon die Bottom-Halfs. Diese waren bisher ähnlich den Softwareinterrupts implementiert, inzwischen setzen sie dagegen auf die Tasklets auf. Die Bottom-Half-Funktionalität sollte in Neuentwicklungen nicht mehr benutzt werden, deswegen wird sie hier auch nicht mehr detailliert beschrieben. Der wesentliche Unterschied zu den Tasklets besteht darin, dass auch auf einem Multiprozessorsystem zu einem Zeitpunkt immer nur ein Bottom-Half-Handler abgearbeitet wird.

3.2.4 Booten des Systems

Das Booten eines Unix-Systems (eigentlich jedes Betriebssystems) hat etwas Magisches an sich. Das soll in diesem Abschnitt etwas transparenter gemacht werden.

In Anhang D wird erklärt, wie LILO (der *LI*nux *LO*ader) den Linux-Kern findet und in den Speicher lädt. Er beginnt dann seine Arbeit am Eintrittspunkt `start:`, der sich in der Datei `arch/i386/boot/setup.S` befindet. Wie der Name schon sagt, handelt es sich hierbei um Assemblercode, welcher die Initialisierung der Hardware übernimmt. Nachdem wichtige Hardwareparameter ermittelt wurden, erfolgt die Umschaltung des Prozessors in den *Protected Mode* durch Setzen des Protected-Mode-Bit im *Maschinen Status Wort*.

Anschließend bewirkt der Assemblerbefehl

```
jmpi 0x100000 , __KERNEL_CS
```

den Sprung zur Startadresse des 32-Bit-Codes des eigentlichen Betriebssystemkerns, und es geht bei `startup_32:` in der Datei `arch/i386/kernel/head.S` weiter. Hier werden weitere Teile der Hardware initialisiert (insbesondere die MMU (Page-Tabelle), der Coprozessor und die Interruptdeskriptortabelle) sowie die Umgebung (Stack, Environment usw.), welche benötigt wird, um die C-Funktionen des Kerns auszuführen. Nach

der vollständigen Initialisierung wird dann die erste C-Funktion, `start_kernel()`, aus `init/main.c` aufgerufen.

Hier erfolgt zunächst in der Funktion `setup_arch()` die Sicherung aller Daten, die der bisherige Assemblercode über die Hardware ermittelt hat sowie die Initialisierung weiterer architekturabhängiger Komponenten. Anschließend werden die hardwareunabhängigen Teile des Kerns initialisiert.

```
asmlinkage void __init start_kernel(void)
{
    char * command_line;

    printk(linux_banner);
    setup_arch(&command_line);
    parse_options(command_line);

    trap_init();
    init_IRQ();
    sched_init();
    time_init();
    softirq_init();

    console_init();

    init_modules();
    …
```

Der jetzt laufende Prozess ist der Prozess 0. Dieser erzeugt dann einen Kernel-Thread, welcher die Funktion `init()` ausführt.

```
    kernel_thread(init,NULL,…)
```

Der Prozess 0 ist anschließend nur noch damit beschäftigt, nicht benötigte Rechenzeit zu verbrauchen.

```
    cpu_idle(NULL);
```

Die Funktion `init()` erledigt die restliche Initialisierung. Unter anderem werden hier von der Funktion `do_basic_setup{}` sämtliche Treiber für die Hardware initialisiert.

```
static int init()
{
    do_basic_setup();
```

Jetzt kann versucht werden, eine Verbindung zur Konsole herzustellen und die Dateideskriptoren 0, 1 und 2 zu eröffnen.

```
    if (open("/dev/console", O_RDWR, 0) < 0)
        printk("Warning: unable to open an initial console.\n");

    (void) dup(0);
    (void) dup(0);
```

Danach wird versucht, ein vom User beim Booten spezifiziertes Programm oder eines der Programme `/sbin/init`, `/etc/init` oder `/bin/init` auszuführen. Diese starten dann normalerweise die unter Linux laufenden Hintergrundprozesse und sorgen dafür, dass auf jedem angeschlossenen Terminal das Programm `getty` läuft — ein Nutzer kann sich beim System anmelden.

```
if (execute_command)
    execve(execute_command,argv_init,envp_init);

execve("/sbin/init",argv_init,envp_init);
execve("/etc/init",argv_init,envp_init);
execve("/bin/init",argv_init,envp_init);
```

Für den Fall, dass keines der oben genannten Programme existiert, wird versucht, eine Shell zu starten, so dass der Superuser das System reparieren kann. Wenn dies auch nicht möglich ist, wird das System angehalten.

```
execve("/bin/sh",argv_init,envp_init);
panic("No init found…");
```

Die hier beschriebene Vorgehensweise sollte nur einen Überblick über die beim Starten des Systems ablaufenden Aktionen geben. Die Realität ist, bedingt durch Hardwareinitialisierung (MMU, SMP) und die Behandlung von Sonderfällen, wie zum Beispiel dem Benutzen einer Initialen Ramdisk (INITRD), komplizierter.

3.2.5 Timerinterrupt

Jedes Betriebssystem braucht eine Zeitmessung und eine Systemzeit. Realisiert wird die Systemzeit in der Regel dadurch, dass die Hardware in bestimmten Abständen einen Interrupt auslöst. Die so angestoßene Interruptroutine übernimmt das „Zählen“ der Zeit. Die Systemzeit wird unter Linux in Ticks seit dem Start des Systems gemessen. Ein Tick entspricht 10 Millisekunden, der Timerinterrupt wird also 100-mal in der Sekunde ausgelöst. Die Zeit wird in der Variablen

```
unsigned long volatile jiffies;
```

gespeichert, welche nur vom Timerinterrupt modifiziert werden darf. Dieser Mechanismus stellt jedoch nur die interne Zeitbasis zur Verfügung.

Anwendungen interessieren sich aber bevorzugt für die „reale Zeit“. Diese wird in der Variablen

```
volatile struct timeval xtime;
```

mitgeführt und ebenfalls vom Timerinterrupt aktualisiert.

Der Timerinterrupt wird relativ häufig aufgerufen und ist deswegen etwas zeitkritisch. Deswegen ist auch hier die Implementierung zweigeteilt.

Die eigentliche Interruptroutine `do_timer()` aktualisiert nur die Variable `jiffies` und kennzeichnet die *Bottom-Half*-Routine (vgl. die Abschnitte 3.2.2 und 7.3.5) des Timerinterrupts als aktiv. Diese wird vom System zu einem späteren Zeitpunkt aufgerufen und erledigt den Rest der Arbeit.

```
void do_timer(struct pt_regs * regs)
{
    (*(unsigned long *)&jiffies)++;

    update_process_times(user_mode(regs));

    mark_bh(TIMER_BH);
    if (TQ_ACTIVE(tq_timer))
        mark_bh(TQUEUE_BH);
}
```

`update_process_times()` wird weiter unten beschrieben. Wir wollen uns aber zuerst die Botton-Half-Routine des Timerinterrupts anschauen.

```
void timer_bh(void)
{
    update_times();
    run_timer_list();
}
```

`run_timer_list()` sorgt dabei für das Abarbeiten der im Abschnitt 3.1.6 beschriebenen Funktionen zur Aktualisierung systemweiter Timer. Darunter fallen auch die Real-Zeit-Timer der aktuellen Task. `update_times()` ist für das Aktualisieren der Zeiten verantwortlich.

```
static inline void update_times(void)
{
    unsigned long ticks;

    ticks = jiffies - wall_jiffies;

    if (ticks) {
        wall_jiffies += ticks;
        update_wall_time(ticks);
    }
    calc_load(ticks);
}
```

`update_wall_time()` widmet sich nun dem Aktualisieren der *realen Zeit* `xtime` und wird aufgerufen, wenn seit dem letztem Aufruf dieser Funktion Zeit vergangen ist.

Die Funktion `update_process_time` sammelt die Daten für den Scheduler und entscheidet, ob dieser aufgerufen werden muss.

```
static void update_process_times(int user_ticks);
{
```

```
    struct task_struct * p = current;
    int cpu = smp_processort_id();

    unsigned long user = ticks - system;
```

Zuerst wird die Komponente `counter` der Task-Struktur aktualisiert. Wenn `counter` gleich Null wird, ist die Zeitscheibe für den aktuellen Prozess abgelaufen, und der Scheduler wird bei der nächsten Gelegenheit aktiviert.

```
    update_one_process(p, ticks, user, system, 0);
    if(p->pid)
    {
        p->counter -= 1;
        if (p->counter <= 0) {
            p->counter = 0;
            p->need_resched = 1;
        }
```

Danach werden für Statistikzwecke die Komponenten `per_cpu_user` Task-Struktur aktualisiert.

```
        p->per_cpu_user[cpu] += user_ticks;
    }
```

Unter Linux ist es möglich, die Ressource „CPU-Verbrauch“ eines Prozesses zu beschränken. Das geschieht durch den Systemruf *setrlimit*, mit welchem auch andere Ressourcen eines Prozesses beschränkt werden können. Das Überschreiten des Zeitlimits wird im Timerinterrupt geprüft, und der Prozess wird durch Senden des Signals `SIGXCPU` informiert bzw. durch das Signal `SIGKILL` abgebrochen. Anschließend müssen noch die Intervalltimer für die laufende Task aktualisiert werden. Wenn diese abgelaufen sind, wird die Task durch ein entsprechendes Signal informiert.

```
void update_one_process(p,user,system,cpu)
{
    p->per_cpu_utime[cpu] += user;
    p->per_cpu_stime[cpu] += system;
    do_process_times(p, user, system);

    do_it_virt(p, user);
    do_it_prof(p);
}

void do_process_times(p,user,system)
{
    psecs = (p->times.tms_utime += user);
    psecs += (p->times.tms_stime += system);
    if (psecs / HZ > p->rlim[RLIMIT_CPU].rlim_cur) {
        /* Send SIGXCPU every second.. */
        if (!(psecs % HZ))
            send_sig(SIGXCPU, p, 1);
        /* and SIGKILL when we go over max.. */
```

```
        if (psecs / HZ > p->rlim[RLIMIT_CPU].rlim_max)
            send_sig(SIGKILL, p, 1);
    }
}

void do_it_virt(p, user) {
    unsigned long it_virt = p->it_virt_value;

    if (it_virt)
    {
        it_virt -= user;
        if (it_virt <= 0)
        {
            it_virt =  p->it_virt_incr;
            send_sig(SIGVTALRM, p, 1);
        }
        p->it_virt_value = it_virt - user;
    }
}
```

3.2.6 Scheduler

Der Scheduler ist für die Zuteilung der Ressource „Prozessor“ (also der Rechenzeit) an die einzelnen Prozesse verantwortlich. Nach welchen Kriterien das geschieht, ist von Betriebssystem zu Betriebssystem verschieden. UNIX-Systeme bevorzugen traditionell interaktive Prozesse, um kurze Antwortzeiten zu ermöglichen und dem Benutzer dadurch das System subjektiv schneller erscheinen zu lassen. Linux unterstützt, dem Posix-Standard 1003.4 folgend, verschiedene Schedulingklassen, welche mit dem Systemruf *sched_setscheduler()* ausgewählt werden können.

Zum einen gibt es Echtzeitprozesse in den Schedulingklassen `SCHED_FIFO` und `SCHED_RR`. Echtzeit heißt hier nicht „harte Echtzeit“ mit garantierten Prozessumschalt- und Reaktionszeiten sondern eine „weiche Echtzeit“. Wenn ein Prozess mit höherer Echtzeitpriorität (beschrieben in der Komponente `rt_priority` der Task-Struktur) laufen möchte, so werden alle Prozesse mit niedrigerer Echtzeitpriorität verdrängt.

Der Unterschied zwischen `SCHED_FIFO` und `SCHED_RR` besteht darin, dass ein Prozess der Klasse `SCHED_FIFO` so lange laufen kann, bis er die Steuerung freiwillig abgibt oder ein Prozess mit höherer Echtzeitpriorität laufen möchte. Im Gegensatz dazu wird ein Prozess in der Klasse `PROG_RR` auch unterbrochen, wenn seine Zeitscheibe abgelaufen ist und es Prozesse mit derselben Echtzeitpriorität gibt. Dadurch wird unter den Prozessen derselben Priorität ein klassisches *Round-Robin*-Verfahren realisiert.

Zum anderen gibt es die Schedulingklasse `SCHED_OTHER`, die einen klassischen UNIX-Schedulingalgorithmus implementiert. Laut Posix 1003.4 hat jeder Echtzeitprozess eine höhere Priorität als ein Prozess der Schedulingklasse `SCHED_OTHER`.

Der Scheduling-Algorithmus von LINUX ist in der Funktion `schedule()` (`kernel/sched.c`) implementiert. Sie wird an zwei verschiedenen Stellen aufgerufen. Zum einen

gibt es Systemrufe, die die Funktion `schedule()` aufrufen, in der Regel indirekt durch den Aufruf von `sleep_on()` (vgl. Abschnitt 3.1.5). Zum anderen wird nach jedem Systemruf und nach jedem Interrupt von der Routine `ret_from_sys_call()` das Flag `need_resched` in der aktuelle Taskstruktur geprüft. Wenn es gesetzt ist, wird der Scheduler ebenfalls aufgerufen. Da zumindest der Timerinterrupt regelmäßig aufgerufen wird und dabei bei Bedarf auch das Flag `need_resched` setzt, wird der Scheduler regelmäßig aktiviert.

Die Funktion `schedule()` besteht aus drei Teilen. Zuerst werden alle anstehenden Softwareinterrupts abgearbeitet. Danach wird der Prozess mit der höchsten Priorität bestimmt. Dabei haben *Real-Time*-Prozesse immer Vorrang vor „normalen". Als drittes wird der neue Prozess zum aktuellen Prozess, und der Scheduler hat seine Arbeit erledigt.

Leider ist der eigentliche Quellcode des Schedulers seit der Kernelversion 2.0 relativ unübersichtlich geworden. Zum Teil liegt das an den aus Effizienzgründen erfolgten Umstrukturierungen in der Version 2.0, zum wesentlichen Teil aber auch an dem neu vorhanden Multiprozessor-Support.

Deswegen werden wir hier eine stark vereinfachte Version der Funktion `schedule()` vorstellen. Unter anderem wurde dabei auf die für den SMP-Support benötigten Details verzichtet.

```
asmlinkage void schedule(void)
{
    struct task_struct * prev, * next, *p;

    prev = current;
    prev->need_resched = 0
```

Zuerst werden die Softwareinterrupts aufgerufen. Dabei handelt es sich um zeitunkritische Routinen, die aus Effiziensgründen aus den Interrupthandlern ausgelagert worden sind. Da diese aber eventuell Informationen manipulieren, die das Scheduling beeinflussen (zum Beispiel kann durch so eine Routine eine Task wieder in den Zustand `TASK_RUNNING` gebracht werden), müssen sie spätestens hier abgearbeitet werden.

```
    if (softirq_active(this_cpu) & softirq_mask(this_cpu))
        do_softirq();
```

Falls `schedule()` aufgerufen wurde, weil der aktuelle Prozess auf ein Ereignis warten muss, so wird er aus der Run-Queue entfernt. Falls die aktuelle Task der Schedulingklasse `SCHED_RR` angehört und das Zeitfenster für diese Task abgelaufen ist, wird sie in der Run-Queue an letzter Stelle und damit hinter allen anderen lauffähigen Tasks der Schedulingklasse `SCHED_RR` gestellt.

Die Run-Queue ist eine durch die Komponente `run_list` der Task-Struktur doppelt verkettete Liste der Prozesse, die sich um den Prozessor bewerben.

```
    if (!prev->counter && prev->policy == SCHED_RR)
    {
```

```
        prev->counter = prev->priority;
        move_last_runqueue(prev);
    }
    if( prev->state != TASK_RUNNING )
    {
        del_from_runqueue(prev);
    }
```

Als nächstes wird der eigentliche Schedulingalgorithmus durchgeführt, das heißt, es wird der Prozess in der Run-Queue gesucht, der die höchste Priorität hat. Die Funktion `goodness()` berechnet dabei für jeden Prozess seine Priorität; Realtime Prozesse werden bevorzugt.

```
    next = idle_task;       /* nachster Prozess */
    next_p = -1000;         /* und dessen Prioritaet */

    list_for_each(p,&runqueue_head)
    {
        if( ! can_schedule(p) )
            continue;
        weight = goodness(p,prev,this_cpu);
        if( weight > next_p)
        {
            next_p = weight; next   = p;
        }
    }
```

Wenn jetzt `next_p` größer 0 ist, so haben wir einen geeigneten Kandidaten gefunden. Ist `next_p` kleiner 0, so gibt es keinen laufbereiten Prozess, und wir müssen die Idle-Task aktivieren. In beiden Fällen zeigt `next` auf die nächste zu aktivierende Task. Wenn jedoch `next_p` gleich 0 ist, so gibt es zwar laufbereite Prozesse, wir müssen aber deren dynamische Prioritäten (den Wert von `counter`) neu berechnen. Hierbei werden dann auch noch die `counter`-Werte aller anderen Prozesse neu berechnet. Danach sollten wir den Scheduler nochmals, diesmal erfolgreich, starten.

```
    if( next_p == 0 )
    {
        for_each_task(p)
        {
            p->counter = (p->counter / 2) + p->priority;
        }
    }
```

Jetzt wird die Task, auf die `next` zeigt, als nächste aktiviert.

```
    if( prev != next )
        switch_to(prev,next);
} /* schedule() */
```

Damit ist die Beschreibung des Schedulers abgeschlossen. Es sei nochmals darauf verwiesen, dass es sich bei diesem Quelltext um eine sehr vereinfachte Version des Schedulers

handelt, die jedoch unserer Meinung nach vollständig genug ist, um die Arbeitsweise des Schedulers zu verstehen.

3.3 Implementierung von Systemrufen

Die Funktionalität des Betriebssystems wird den Prozessen mit Hilfe von Systemrufen zur Verfügung gestellt. In diesem Kapitel wollen wir uns mit der Implementierung von Systemrufen unter Linux beschäftigen.

3.3.1 Wie funktionieren Systemrufe eigentlich?

Ein Systemruf setzt einen definierten Übergang vom Nutzermodus in den Systemmodus voraus. Dies ist in Linux nur durch Interrupts möglich. Für die Systemrufe wurde deshalb der Interrupt 0x80 reserviert.[4]

Normalerweise ruft man als Nutzer immer eine Bibliotheksfunktion (wie etwa `fork()`) auf, um bestimmte Aufgaben auszuführen. Diese Bibliotheksfunktion (in der Regel aus den `_syscall`-Makros in `<asm/unistd.h>` generiert) schreibt ihre Argumente und die Nummer des Systemrufs in definierte Übergaberegister und löst anschließend den Interrupt 0x80 aus. Wenn die zugehörige Interruptserviceroutine zurückkehrt, wird der Rückgabewert aus dem entsprechenden Übergaberegister gelesen, und die Bibliotheksfunktion ist beendet.

Die eigentliche Arbeit der Systemrufe wird von der Interruptroutine erledigt. Diese beginnt bei der Einsprungadresse `system_call()`, die in der Datei `arch/i386/kernel/entry.S` zu finden ist.

Leider ist diese Routine vollständig in Assembler geschrieben. Hier wird der besseren Lesbarkeit halber ein C-Äquivalent dieser Funktion beschrieben. Wo immer im Assemblertext symbolische Marken vorkommen, haben wir sie auch in den C-Text als Marken übernommen.

`sys_call_num` und `sys_call_args` entsprechen der Nummer des Systemrufs (vgl. `<asm/unistd.h>`) und dessen Argumenten.

```
PSEUDO_CODE system_call( int sys_call_num , sys_call_args )
{
system_call:
```

Zuerst werden alle Register des Prozesses gerettet.

```
    SAVE_ALL; /* Makro aus entry.S */
```

Wenn `sys_call_num` einen legalen Wert repräsentiert, wird die der Nummer des Systemrufs zugeordnete Behandlungsroutine aufgerufen. Diese ist im Feld `sys_call_`

4 Das gilt für die Linux-Systemrufe auf dem PC. Schon die von Linux auf dem PC unterstützte iBCS-Emulation verwendet ein anderes Verfahren — das sogenannte `lcall7`-Gate.

table[] (in der Datei arch/i386/kernel/entry.S definiert) eingetragen. Falls für den Prozess das Flag PF_TRACESYS gesetzt ist, wird dieser von seinem Elternprozess überwacht. Die dazu notwendigen Arbeiten erledigt die Funktion syscall_trace() (arch/i386/kernel/ptrace.c). Sie ändert den Zustand des aktuellen Prozesses auf TASK_STOPPED, sendet dem Elternprozess das Signal SIGTRAP und ruft den Scheduler auf. Der aktuelle Prozess wird unterbrochen, bis der Elternprozess ihn wieder aktiviert. Da das vor und nach jedem Systemruf erfolgt, hat der Elternprozess vollständige Kontrolle über das Verhalten des Kindprozesses.

```
    if (sys_call_num >= NR_syscalls)
    {
badsys:
            errno = -ENOSYS;
    }
    else
    {
        if (current->ptrace)
            syscall_trace();

        errno=(*sys_call_table[sys_call_num])(sys_call_args);

        if (current->ptrace)
            syscall_trace();
   }
```

Die eigentliche Arbeit des Systemrufs ist jetzt erledigt. Bevor der Prozess weiterarbeiten kann, gibt es aber eventuell noch einige administrative Aufgaben zu erledigen.

```
ret_from_sys_call:
    if (softirq_active & softirq_mask)
handle_softirq:
        do_softirq()
```

Falls ein Scheduling angefordert wurde (need_resched != 0), wird der Scheduler aufgerufen. Dadurch wird ein anderer Prozess aktiv. schedule() kehrt erst wieder zurück, wenn der Prozess vom Scheduler neu aktiviert wurde.

```
ret_with_reschedule:
    if (current->need_resched) {
reschedule:
        schedule();
        goto ret_from_sys_call;
    }
```

Falls für den aktuellen Prozess Signale gesendet wurden, und der Prozess ihre Annahme nicht blockiert hat, werden sie jetzt abgearbeitet. Die Funktion do_signal() ist in Abschnitt 3.2.1 näher beschrieben.

```
    if (current->sigpending)
signal_return:
        do_signal();
```

Damit ist alles erledigt, und der Systemruf kehrt zurück. Dazu werden zuerst alle Register restauriert, und anschließend wird mit dem Assemblerbefehl `iret` die Interruptroutine beendet.

```
restore_all:
    RESTORE_ALL;
} /* PSEUDO_CODE system_call */
```

3.3.2 Beispiele für einfache Systemrufe

Im Folgenden wollen wir uns die Implementierung einiger Systemrufe genauer ansehen. Dabei soll die Benutzung der in diesem Kapitel erläuterten Algorithmen und Datenstrukturen demonstriert werden.

Der Systemruf getuid

getuid ist ein sehr einfacher Systemruf – er liest lediglich einen Wert aus der Taskstruktur und liefert diesen zurück:

```
asmlinkage int sys_getuid(void)
{
    return current->uid;
}
```

Der Systemruf nice

Der Systemruf *nice* ist etwas komplizierter. *nice* erwartet als Argument eine Zahl, um die die statische Priorität des aktuellen Prozesses modifiziert werden soll.

Alle Systemrufe, die Argumente verarbeiten, müssen diese auf ihre Plausibilität hin überprüfen.

```
asmlinkage int sys_nice(int increment)
{
    int newpriority;
```

Man beachte, dass ein größeres Argument von `sys_nice()` eine geringere Priorität bedeutet. Deswegen ist der Name `increment` für das Argument von *nice* etwas irreführend.

```
    if (increment < 0 && !capable(CAP_SYS_NICE))
        return -EPERM;
```

`capable()` überprüft, ob der aktuelle Prozess die Berechtigung hat, seine Priorität zu erhöhen. In klassischen Unix-Systemen ist dies der Fall, wenn der Prozess Superuserrechte besitzt. Linux stellt mittlerweile ein Konzept zur Verfügung, dieses Superuserrecht feiner zu unterteilen.

Jetzt kann die neue Priorität des Prozesses ermittelt werden. Dabei wird unter anderem sichergestellt, dass sich die neue Priorität des Prozesses in einem sinvollen Bereich bewegt.

```
    newpriotity = …

    if (newpriority < -20)
        newpriority = -20;
    if (newpriority > 19)
        newpriority = 19;

    current->nice = newpriority;
    return 0;
} /* sys_nice */
```

Der Systemruf pause

pause unterbricht die Programmausführung so lange, bis der Prozess durch ein Signal wieder aktiviert wird. Dazu wird lediglich der Status des aktuellen Prozesses auf `TASK_INTERRUPTIBLE` gesetzt und anschließend der Scheduler aufgerufen. Dadurch wird eine andere Task aktiv.

Der Prozess kann nur wieder aktiv werden, wenn der Status des Prozesses wieder auf `TASK_RUNNING` gesetzt wird. Dies geschieht beim Eintreffen eines Signals (vgl. Abschnitt 3.2.6). Danach kehrt der Systemruf *pause* mit dem Fehler `ERESTARTNOHAND` zurück und führt (wie in Abschnitt 3.2.1 beschrieben) die notwendigen Aktionen zur Signalbehandlung durch.

```
asmlinkage int sys_pause(void)
{
    current->state = TASK_INTERRUPTIBLE;
    schedule();
    return -ERESTARTNOHAND;
}
```

3.3.3 Beispiele für komplexere Systemrufe

Jetzt wollen wir uns etwas komplexeren Systemrufen zuwenden. In diesem Abschnitt werden wir die Systemrufe zur Prozessverwaltung (*fork*, *execve*, *_exit* und *wait*) beschreiben.

Der Systemruf fork

Der Systemruf *fork* ist die einzige Möglichkeit, einen neuen Prozess zu starten. Das geschieht, indem eine (fast) identische Kopie des Prozesses erzeugt wird, welcher *fork* aufgerufen hat.

fork ist eigentlich ein recht aufwendiger Systemruf. Es müssen alle Daten des Prozesses kopiert werden. Das können durchaus einige Megabyte sein. Im Laufe der Entwicklung von UNIX wurden verschiedene Wege eingeschlagen, um den Aufwand für *fork* so gering wie möglich zu halten. In dem häufig vorkommenden Fall, dass nach *fork* direkt *exec* aufgerufen wird, ist das Kopieren der Daten nicht notwendig. Sie werden nicht benötigt. In den UNIX-Systemen der BSD-Reihe wurde deswegen der Systemruf *vfork* etabliert. Er erzeugt wie *fork* einen neuen Prozess, teilt das Datensegment aber zwischen beiden Prozessen auf. Dies ist eine recht unsaubere Lösung, da ein Prozess die Daten des anderen Prozesses beinflussen kann. Um diese Beeinflussung so gering wie möglich zu halten, wird die weitere Ausführung des Elternprozesses so lange gestoppt, bis der Kindprozess entweder durch *_exit* beendet wurde oder durch *exec* ein neues Programm gestartet hat.

Neuere UNIX-Systeme wie zum Beispiel LINUX schlagen einen anderen Weg ein. Sie benutzen die *Copy-On-Write*-Technik. Hintergedanke dabei ist, dass mehrere Prozesse durchaus gleichzeitig auf denselben Speicher zugreifen dürfen — solange sie die Daten nicht modifizieren.

Die Speicherseiten werden also unter LINUX beim Systemruf *fork* nicht kopiert, sondern vom alten und vom neuen Prozess gleichberechtigt benutzt. Allerdings werden die von beiden Prozessen benutzten Speicherseiten als schreibgeschützt markiert — sie können von den Prozessen also nicht modifiziert werden. Wenn nun einer der Prozesse eine Schreiboperation auf diesen Speicherseiten ausführen will, wird von der Speicherverwaltungshardware (MMU) ein Seitenfehler (*page fault*) ausgelöst, der Prozess unterbrochen und der Kern benachrichtigt. Erst jetzt kopiert der Kern die mehrfach benutzte Speicherseite und teilt dem schreibenden Prozess seine eigene Kopie zu. Dieses Verfahren erfolgt vollständig transparent — d.h. die Prozesse merken nichts davon. Der große Vorteil dieses Copy-On-Write-Verfahrens besteht darin, dass aufwendiges Kopieren von Speicherseiten nur bei Bedarf stattfindet.

Neuere Betriebssystemkonzepte kennen außer dem Begriff des Prozesses noch den Begriff des Threads. Darunter versteht man einen unabhängigen „Faden" im Programmablauf eines Prozesses. Mehrere Threads können innerhalb eines Prozesses parallel und unabhängig voneinander abgearbeitet werden. Der wesentliche Unterschied zum Konzept eines Prozesses besteht darin, dass die verschiedenen Threads innerhalb eines Prozesses auf demselben Speicher operieren und sich damit gegenseitig beinflussen können. Es gibt verschiedene Konzepte zur Implementierung von Threads. Einfache Varianten, wie die weitverbreitete Pthread-Bibliothek, kommen ohne Unterstützung durch den Betriebssystemkern aus. Nachteile dieser Konzeptionen sind, dass das Scheduling der einzelnen Threads vom Nutzerprogramm vorgenommen werden muss. Für den Betriebssystemkern handelt es sich um einen normalen Prozess. Das führt dazu, dass ein blockierender Systemruf (z. B. ein *read* vom Terminal) den ganzen Prozess und damit alle Threads blockiert. Ideal wäre es aber, wenn nur der Thread, der den Systemruf verwendet hat, blockiert. Dies setzt aber eine Unterstützung des Thread-Konzepts durch den Kern voraus. Neuere UNIX-Versionen (z. B. Solaris 2.x) bieten diese Unterstützung.

LINUX unterstützt Threads durch die Bereitstellung des (linuxspezifischen) Systemrufs *clone*, welcher die nötige Kernelunterstützung zur Implementierung von Threads liefert.

clone arbeitet ähnlich wie *fork*, erzeugt also eine neue Task. Der wesentliche Unterschied zu *fork* besteht darin, dass nach dem Systemruf *clone* beide Tasks auf gemeinsamem Speicher arbeiten können.

Da *fork* und *clone* im Wesentlichen dasselbe tun, werden sie auch durch eine gemeinsame Funktion realisiert. Diese wird je nach Systemruf nur auf unterschiedliche Weise aufgerufen.

```
asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs,0);
}

asmlinkage int sys_clone(struct pt_regs regs)
{
    unsigned long clone_flags;
    unsigned long newsp;

    clone_flags = regs.ebx;
    newsp = regs.ecx;
    if (!newsp)
        newsp = regs.esp;
    return do_fork(clone_flags, newsp, &regs,0);
}
```

Die eigentliche Arbeit erledigt die Funktion `do_fork()`:

```
int do_fork(unsigned long clone_flags,
            unsigned long start_stack,
            struct pt_regs *regs,
            unsigned long stack_size)
{
    int error = -ENOMEM;
    unsigned long new_stack;
    struct task_struct *p;
```

Als Erstes wird der notwendige Speicherplatz für die neue Taskstruktur und den Kernel-Stack alloziert.

```
    p = alloc_task_struct();
    if (!p)
        goto fork_out;
```

Falls der Nutzer sein Limit an Prozessen überschritten hat, wird die Funktion abgebrochen. Dasselbe passiert, wenn es im System bereits zu viele Tasks gibt.

```
    if (current->user->count >=
                current->rlim[RLIMIT_NPROC].rlim_cur)
        goto bad_fork_free;
    if (nr_threads >= max_threads)
        goto bad_fork_cleanup_count;
```

Der Kindprozess p erbt alle Einträge des Elternprozesses.

```
*p = *current;
```

Einige der Einträge müssen für einen neuen Prozess jedoch auch neu initialisiert werden.

```
p->state = TASK_UNINTERUPTIBLE;
p->did_exec = 0;
p->swappable = 0;
p->pid = get_pid(clone_flags);
…
p->run_list.next = NULL;
p->run_list.prev = NULL;
…
p->start_time = jiffies;
…
```

Jetzt werden die Unterstrukturen der Taskstruktur kopiert. Je nach Wert der `clone_flags` werden hier Datenstrukturen entweder kopiert oder gemeinsam benutzt. Damit werden die Unterschiede zwischen den Systemrufen *fork* und *clone* realisiert.

```
if (copy_files(clone_flags, p))
    goto bad_fork_cleanup;
if (copy_fs(clone_flags, p))
    goto bad_fork_cleanup_files;
if (copy_sighand(clone_flags, p))
    goto bad_fork_cleanup_fs;
if (copy_mm(clone_flags, p))
    goto bad_fork_cleanup_sighand;
copy_thread(0, clone_flags, start_stack, stack_size, p, regs);
```

Abschließend wird der Zustand der neuen Task auf `TASK_RUNNING` gesetzt, damit er vom Scheduler aktiviert werden kann. Die alte Task (der Elternprozess) kehrt vom Systemruf mit der Prozessidentifikationsnummer (PID) des neuen Prozesses zurück.

```
++nr_threads;
wake_up_process(p);
return p->pid;
```

Wenn irgend etwas schiefgelaufen sein sollte, so müssen bis dahin angeforderte Datenstrukturen wieder freigegeben werden.

```
bad_fork:
    …
    return error;
}
```

Die oben aufgerufene Funktion `copy_thread()` ist auch für die Initialisierung der Register des neuen Prozesses verantwortlich. Unter anderem wird der Befehlszeiger `p->tss.eip` auf die Assemblerroutine `ret_from_sys_call()` gelegt, so dass der neue Prozess seine Abarbeitung so beginnt, als ob auch er den Systemruf *fork* ausgeführt hätte. Gleichzeitig wird der Rückgabewert auf Null gesetzt, damit das Programm

Eltern- und Kindprozess anhand des unterschiedlichen Rückgabewertes unterscheiden kann.

Der Systemruf execve

Der Systemruf *execve* erlaubt es einem Prozess, sein ausführendes Programm zu wechseln. Linux erlaubt mehrere Formate für ausführbare Dateien. Sie werden, wie in Unix üblich, an den sogenannten „Magic-Numbers" — den ersten Bytes einer ausführbaren Datei erkannt. Traditionell verwendet jedes Unix-System sein eigenes Format für ausführbare Dateien. In den letzten Jahren hat sich ein Standard herausgebildet — das ELF-Format.[5] Dieses hat sich durchgesetzt, da sich hier die Behandlung von dynamischen Bibliotheken drastisch vereinfacht. Weitere Informationen zum ELF-Format findet der interessierte Leser in [ELF].

Des Weiteren unterstützt Linux die aus der BSD-Welt stammenden Scriptdateien. Wenn eine Datei mit den beiden Zeichen „`#!`" beginnt, wird sie nicht direkt geladen, sondern einem in der ersten Zeile der Datei spezifizierten Interpreterprogramm zur Bearbeitung übergeben. Die bekannte Version davon ist eine Zeile der Form

```
#!/bin/sh
```

am Anfang von Shell-Skripten. Ein Ausführen dieser Datei (d.h. ein *execve*) ist äquivalent zum Ausführen der Datei `/bin/sh` mit der ursprünglichen Datei als Argument. Hier folgt nun der kommentierte und stark gekürzte Quelltext von `do_execve()`.

```
static int do_execve(char *filename, char **argv, char **envp,
                     struct pt_regs * regs)
{
```

Zuerst wird versucht, aus dem Namen des auszuführenden Programms die zugehörige Datei (ihre Inode) zu finden. Die Struktur `bprm` wird benutzt, um alle Informationen über die Datei zu speichern.

```
    struct linux_binprm bprm;
    struct file *file;

    file = open_exec(filename);o

    if ( IS_ERR(file))
        return PTR_ERR(file);

    brpm.file = file;
    bprm.filename = filename;
    bprm.argc = count(argv);
    bprm.envc = count(envp);
    …
```

5 ELF steht für *Executable and Linkable Format*.

Die Funktion `prepare_binrpm` prüft die Zugriffsrechte und liest die ersten 128 Byte der Datei ein.

```
    retval = prepare_binrpm(&brpm);
    if (retval <= 0)
        goto error;
```

Jetzt kann anhand der ersten Bytes der Datei geprüft werden, wie diese geladen werden soll. Linux verwendet für jedes ihm bekannte Dateiformat eine eigene Funktion zum Laden der Datei. Sie werden nacheinander aufgerufen und „gefragt", ob sie die Datei laden können. Wenn die Datei geladen werden kann, endet `execve()` erfolgreich, wenn nicht, liefert es `ENOEXEC` zurück.

```
    return seach_binary_handler(&bprm.regs);
} /* do_execve() */

int search_binary_handler(struct linux_binprm *bprm,
                          struct pt_regs *regs)
{
    for( fmt = formats; fmt ; fmt = fmt->next )
    {
        if (!fmt->load_binary)
            continue;
        retval = (fmt->load_binary)(bprm, regs);
        if (retval >= 0) {
            current->did_exec = 1;
            return retval;
        }
        if( retval != -ENOEXEC )
            break;
    }
    return(retval);
}
```

Die eigentliche Arbeit wird also von der Funktion `fmt->load_binary()` erledigt. Betrachten wir eine solche Funktion einmal genauer:

```
int load_aout_binary(struct linux_binprm *bprm,
                     struct pt_regs *regs)
{
```

`bprm->buf` enthält die ersten 128 Bytes der zu ladenden Datei. Zuerst wird anhand dieses Dateianfangs geprüft, ob es sich um das richtige Dateiformat handelt. Wenn dies nicht der Fall ist, liefert diese Funktion den Fehler `ENOEXEC`. Daraufhin kann `search_binary_handler()` weitere Formate überprüfen. Bei den Überprüfungen werden auch gleich einige später benötigte Informationen aus dem Header extrahiert.

```
    struct exec ex;

    ex = *((struct exec *) bprm->buf);
    if ((N_MAGIC(ex) != ZMAGIC && N_MAGIC(ex) != OMAGIC &&
```

```
            N_MAGIC(ex) != QMAGIC && N_MAGIC(ex) != NMAGIC) ||
            N_TRSIZE(ex) || N_DRSIZE(ex) ||
            bprm->dentry->inode->i_size < ex.a_text+ … )
    {
        return -ENOEXEC;
    }
    …
    fd_offset = N_TXTOFF(ex);
    …
```

Wenn diese Prüfungen erfolgreich abgeschlossen wurden, wird das neue Programm geladen. Dazu wird als erstes der Speicher des Prozesses freigegeben, er enthält ja noch das alte Programm. Nach dieser Freigabe kann `execve()` nicht mehr in das alte Programm zurückkehren. Wenn jetzt ein Fehler beim Laden der Datei auftritt, muss der Prozess abgebrochen werden.

```
    flush_old_exec(bprm);
```

Jetzt kann die Taskstruktur aktualisiert werden. Dabei wird in der Komponente `personality` registriert, dass es sich um ein Programm im Linux-eigenen Format handelt.

```
    set_personality(PER_LINUX);


    current->mm->end_code = ex.a_text +
                (current->mm->start_code = N_TXTADDR(ex));
    …
```

Nun kann das Text- und das Datensegment des Programmes mit `do_mmap()` in den Speicher eingeblendet werden. Man beachte, dass `do_mmap()` die Datei hier nicht lädt, sondern nur die Page-Tabellen aktualisiert und dem Paging-Algorithmus damit angibt, woher er die Speicherseiten bei Bedarf laden kann. Das Paging ist in Abschnitt 4.4 beschrieben.

```
    do_mmap(bprm->file, N_TXTADDR(ex), ex.a_text,
                 PROT_READ | PROT_EXEC,
                 MAP_FIXED | MAP_PRIVATE | …, fd_offset);
    …
```

Jetzt wird das BSS-Segment geladen. Es enthält unter Unix die uninitialisierten Daten eines Prozesses. Die Funktion `set_brk()` erledigt das. Anschließend werden die Register und speziell der Instruction-Pointer für das neue Programm initialisiert. Dies wird durch die Funktion `start_thread()` erledigt. Wenn der Systemruf *execve* jetzt seine Arbeit beendet, wird die Programmausführung des Prozesses an der neuen Adresse fortgesetzt.

```
    set_brk(current->mm->start_brk, current->mm->brk);
    current->mm->start_stack = …;
    start_thread(regs, ex.a_entry, current->mm->start_stack);
    return 0;
} /* load_aout_binary */
```

Im Kern sind die Funktionen `do_execve()` und `load_aout_binary()` erheblich komplizierter. Das liegt zum einen an der notwendigen Fehler- und Ausnahmebehandlung[6]. Andererseits haben wir in der Beschreibung auch viele „unwichtige" Details weggelassen; „unwichtig" hier in dem Sinne, dass sie zum Verständnis der Grundprinzipien von `do_execve()` nicht notwendig sind. Wer sich ernsthaft damit beschäftigen und etwa ein neues Dateiformat implementieren will, wird um das Studium der Originalquellen nicht herumkommen.

Der Systemruf exit

Ein Prozess wird immer durch den Aufruf der Kernfunktion `do_exit()` beendet. Dies geschieht entweder direkt durch den Systemruf *_exit* oder indirekt beim Auftreten eines Signals, welches nicht abgefangen werden kann.

Eigentlich hat `do_exit()` nicht viel zu tun. Es müssen nur die vom Prozess belegten Ressourcen freigegeben und eventuell andere Prozesse benachrichtigt werden. Hier steckt jedoch viel Aufwand im Detail, weswegen die folgende Beschreibung der Funktion `do_exit()` auch wieder stark gekürzt ist. Wir betrachten zum Beispiel nicht die Aktionen, die zur sauberen Verwaltung der Prozessgruppen und Threads notwendig sind.

```
NORET_TYPE void do_exit(long code)
{
```

Als erstes gibt der Prozess alle von ihm belegten Ressourcen frei.

```
    del_timer(&current->real_timer);
    sem_exit();
    __exit_mm(current);
    __exit_files(current);
    __exit_fs(current);
    __exit_sighand(current);
    exit_thread();
```

Der Elternprozess wird vom Ableben des Kindprozesses informiert. Eventuell wartet dieser ja schon mit dem Systemruf *wait* auf das Ende des Kindprozesses. Wenn ein Prozess seine Arbeit beendet, müssen auch alle Kindprozesse einen neuen Elternprozess bekommen. Standardmäßig erbt der Prozess 1 alle Kindprozesse. Falls er nicht mehr existieren sollte, werden sie an den Prozess 0 vererbt. Dieses erledigt alles die Funktion `exit_notify()`. Desweiteren setzt sie auch den Status des aktuellen Prozesses von `TASK_RUNNING` auf `TASK_ZOMBIE`

```
    exit_notify();
```

Alle Aufräumarbeiten sind erledigt. Für den Prozess wird jetzt (außer für die Taskstruktur) kein Speicherplatz mehr benötigt. Er wird zu einem Zombie-Prozess. Der Prozess bleibt so lange ein Zombie-Prozess, bis der Elternprozess den Systemruf *wait* ausführt.

6 Es kann zum Beispiel vorkommen, dass ältere Linux-Binaries nicht mit `do_mmap()` geladen werden können. Dann muss `load_aout_binary()` den Programmcode und die Daten vollständig laden und kann sich nicht auf das *Demand-Loading* verlassen.

```
    current->exit_code = code;
```

Schließlich ruft `do_exit()` den Scheduler auf und erlaubt anderen Prozessen die Weiterarbeit. Da der Status des aktuellen Prozesses `TASK_ZOMBIE` ist, kehrt die Funktion `schedule()` an dieser Stelle nie mehr zurück.

```
    schedule();
   /* NOTREACHED */
} /* do_exit */
```

Der Systemruf wait

Der Systemruf *wait4* ermöglicht das Warten auf das Ende eines Kindprozesses und die Abfrage des vom Kindprozess gelieferten Exit-Codes. *wait4* wartet dabei, je nach übergebenem Argument, entweder auf einen bestimmten Kindprozess, einen Kindprozess aus einer bestimmten Prozessgruppe oder auf jeden Kindprozess. Genauso lässt sich steuern, ob *wait4* wirklich auf das Ende eines Kindprozesses warten oder ob er nur bereits beendete Kindprozesse beachten soll. Da all diese Fallunterscheidungen ziemlich langweilig sind, beschreiben wir im Folgenden eine modifizierte Version von *wait4*, deren Semantik ungefähr der von *wait* entspricht. (Normalerweise ist *wait* eine Bibliotheksfunktion, die *wait4* mit passenden Argumenten aufruft.)

```
int sys_wait( … )
{
repeat:
```

`sys_wait()` besteht aus zwei Teilen. Zuerst wird geprüft, ob es bereits einen Kindprozess im Zustand `TASK_ZOMBIE` gibt. Wenn ja, haben wir den gesuchten Prozess gefunden und `sys_wait()` kann erfolgreich zurückkehren. Vorher werden noch statistische Daten aus der Prozesstabelle des Kindprozesses übernommen (verbrauchte Systemzeit, Exit-Code usw.) und anschließend die Taskstruktur des Kindprozesses freigegeben. Dies ist die einzige Möglichkeit, einen Prozesseintrag wieder aus der Prozesstabelle zu entfernen.

```
    nr_of_childs = 0;
    for (p = current->p_cptr ; p ; p = p->p_osptr)
    {
        ++nr_of_childs;

        if(p->state == TASK_ZOMBIE)
        {
            current->times.tms_cutime += p->times.tms_utime +
                                         p->times.tms_cutime;
            current->times.tms_cstime += p->times.tms_stime +
                                         p->times.tms_cstime;
            if (stat_addr)
                put_user(p->exit_code, stat_addr);

            release(p);
```

```
            return p;
        }
    }
```

Falls es keine Kindprozesse gibt, kehrt `sys_wait()` sofort zurück.

```
    if (nr_of_childs == 0)
        return 0;
```

Falls es doch Kindprozesse gibt, wird auf das Ende eines der Kindprozesse gewartet. Dazu trägt sich der Elternprozess in die dafür bestimmte Warteschlange seiner eigenen Taskstruktur ein. Wie wir oben gesehen haben, weckt jeder Prozess beim Systemruf *_exit* alle in dieser Warteschlange wartenden Prozesse mit Hilfe der Funktion `wake_up()` auf. Dadurch ist garantiert, dass der Elternprozess über das Ende eines Kindprozesses informiert wird.

```
    interruptible_sleep_on(&current->wait_chldexit);
```

Das beim Beenden des Kindprozesses von `do_exit()` gesendete Signal `SIGCHLD` wird ignoriert. Falls zwischendurch ein Signal empfangen wurde (immerhin kann `interruptible_sleep_on()` auch von einem Signal unterbrochen worden sein), wird der Systemruf mit einer Fehlermeldung beendet. Ansonsten wissen wir, dass es jetzt einen Kindprozess gibt, der sich im Zustand `TASK_ZOMBIE` befindet, und können wieder von vorne anfangen, ihn zu suchen.

```
    current->signal &= ~(1<<(SIGCHLD-1));

    if (current->signal & ~current->blocked)
        return -EINTR;

    goto repeat;

} /* sys_wait */
```

Pearson Education

ADDISON-WESLEY

4 Die Speicherverwaltung

Datenspeicher bestehen aus Tausenden von Speicherzellen. Die einzelnen Zellen müssen daher nach einem sinnreichen und möglichst einfachen System geordnet sein.

John S. Murphy

Obiges Zitat stammt aus einem Buch, das 1958 im Original erschienen ist. Seitdem sind die Anforderungen an ein Speicherverwaltungssystem stark gewachsen. Einige tausend Speicherzellen reichen heute für die wenigsten Anwendungen aus. Das Verlangen nach Einfachheit und Zweckdienlichkeit ist heute genauso aktuell wie damals.

Ein Multitaskingsystem wie Linux stellt besondere Anforderungen an die Speicherverwaltung. Der Speicher eines Prozesses und der vom Kernel genutzte Speicher muss vor dem Zugriff anderer Prozesse geschützt werden. Dieser Schutz ist entscheidend für die Stabilität eines multitaskingfähigen Betriebssystems. Er verhindert, dass ein Prozess wahllos in den Speicher anderer Prozesse hineinschreibt und diese zum Absturz bringt. Dies kann zum Beispiel schon dann passieren, wenn in einem C-Programm die Grenzen einer Feldvariablen überschritten werden.

Speicherschutz verhindert Systemabstürze durch Fehler in Anwendungsprogrammen und verhindert auch Manipulationen durch böswillige Programme wie Viren und Trojaner. Darüber hinaus zwingt der Speicherschutz Programmierer dazu, die Kommunikation zwischen Anwendungsprogramm und Betriebssystem über definierte Schnittstellen stattfinden zu lassen, anstatt den Speicher des Betriebssystems direkt zu manipulieren.

Arbeitsspeicher (RAM) war lange Zeit eine sehr knappe Ressource. Der Speicherbedarf von Anwendungen und der üblicherweise verfügbare Arbeitsspeicher scheinen auch weiterhin wechselseitig in die Höhe getrieben zu werden. Zu der früher als Speicherfresser bekannten Software für Linux, wie dem GNU-C-Compiler oder dem X-Window-System, gesellen sich heute Datenbanksysteme, Desktop-Shells wie GNOME und KDE sowie Java. Java ist ein Beispiel dafür, wie höhere Abstraktionsgrade bei der Programmierung mit höherem Speicherbedarf einhergehen.

Da ein Multitaskingsystem wie Linux mehrere Prozesse quasiparallel abarbeitet, ist es möglich, dass der Speicherbedarf aller auszuführenden Prozesse die Größe des Arbeitsspeichers übersteigt. Durch eine Auslagerung von Bereichen aus dem primären Arbeitsspeicher in den sekundären Speicher (z. B. in Bereiche auf Festplatten) lösen Betriebssysteme dieses Problem. Es ist auch möglich, dass ein Prozess mehr Speicherbedarf hat, als im primären Speicher vorhanden ist. Das Betriebssystem sollte also auch Auslagerungen von Teilen des Speichers eines Prozesses erlauben.

Wenn zwei Prozessinstanzen eines Programms quasiparallel abgearbeitet werden, müssen zumindest die Daten der beiden Prozesse in unterschiedlichen physischen Speicherbereichen abgelegt sein. Das bedeutet, dass Daten derselben Variablen für jeden Prozess an unterschiedlichen physischen Adressen gespeichert sind. Die bei weitem eleganteste Methode, mit diesem Problem umzugehen, ist die Einführung eines virtuellen Adressraums für jeden Prozess. Der Programmierer kann sein Programm unabhängig von der realen Verteilung des Codes und der Daten im physischen Adressraum gestalten. Für die Abbildung der virtuellen Adressen, die auch lineare Adressen genannt werden, auf die physischen Adressen ist das Speicherverwaltungssystem des Betriebssystems verantwortlich.

Speicherschutz verhindert den Austausch von Daten zwischen zwei Prozessen durch das gegenseitige Verändern von Speicherbereichen. Die Interprozesskommunikation muss in diesem Fall über Systemrufe erfolgen. Der Aufruf eines Systemrufs ist aber mit einer Reihe von Operationen verbunden, z. B. mit der mehrmaligen Sicherung von Registern auf den Stack, dem Kopieren von Speicherbereichen etc. Wenn Prozesse Speicherbereiche kontrolliert teilen können, kann die Interprozesskommunikation effizienter erfolgen.

Dieses Konzept, *Shared Memory*, ist nicht nur auf die Kommunikation mit Prozessen beschränkt. So ließen sich zum Beispiel auch Bereiche aus Dateien in den Speicher eines Prozesses einblenden. Anwendungen können so viele wiederholte Systemrufe zum Lesen und Schreiben einsparen.

Der Programmcode eines Prozesses kann in mehreren Threads gleichzeitig abgearbeitet werden. Diese Threads teilen sich grundsätzlich einen gemeinsamen virtuellen Adressraum.

Eine effiziente Implementierung einer modernen Speicherverwaltung ist ohne Hardwareunterstützung nicht denkbar. Da Linux auch auf Systemen läuft, die nicht auf der Intel-Architektur basieren, war es notwendig, ein *architekturunabhängiges Speichermodell* zu entwickeln. Solch ein Speichermodell muss so universell sein, dass es zusammen mit den Speicherarchitekturen unterschiedlichster Prozessortypen angewendet werden kann. In diesem Kapitel wird zunächst dieses architekturunabhängige Speichermodell vorgestellt. Die Umsetzung dieses Speichermodells wird für die zum i386-Prozessor kompatible Prozessorfamilie vorgestellt. CPUs aus dieser Familie werden im Folgenden x86-Prozessoren genannt. Intel hat mit der Einführung der eigenen 64-Bit-Architektur die zugrunde liegende Architekur als IA-32 bezeichnet. Wir bleiben bei der alten Bezeichnung, da sie definitiv auch die Prozessoren anderer Hersteller einschließt.

Um die Flexibilität des Speichermodells zu demonstrieren, werden hier aber auch die Umsetzungen für andere Prozessorarchitekturen dargestellt.

Die Kernelentwickler haben das Speicherverwaltungssystem in den letzten Jahren immer wieder optimiert. Insbesondere stand hier die Optimierung für Multiprozessorsysteme im Vordergrund. Eine detaillierte Darstellung dieser Optimierung ist hier nicht möglich. So unterstützt Linux in der Version 2.4 auch das Konzept *Nonuniform Memory Access* (NUMA) für die *Intel Architecture 64* (IA-64) und MIPS64. Das NUMA-Konzept kommt zum Tragen, wenn jeder CPU eines Multiprocessingsystems ein eigener Speicherbereich

zugeordnet ist, auf den der Prozessor schneller zugreifen kann als auf Speicher von anderen Prozessoren. Das Betriebssystem sollte also die Zahl der Zugriffe des Prozessors auf fremden Speicher minimieren.

Im zweiten Teil dieses Kapitels wird erläutert, wie das architekturunabhängige Speichermodell verwendet wird, um die Speicherverwaltung zu implementieren. Es werden dabei die von Linux verwendeten Algorithmen zur Speicherverwaltung vorgestellt. Es sei darauf hingewiesen, dass von anderen Systemen andere Algorithmen und Konzepte für die Speicherverwaltung verwendet wurden und werden. Interessierten Lesern sei [Bac86] zur Lektüre empfohlen.

4.1 Das architekturunabhängige Speichermodell von Linux

Ein typischer Rechner verfügt heute über mehrere Speicherstufen mit unterschiedlichen Zugriffszeiten. Die erste Stufe besteht meist aus prozessorinternem Cachespeicher. In den neuesten Prozessoren ist sogar die zweite Cachestufe im Prozessor integriert, um die Zugriffszeiten noch weiter zu minimieren. Der eigentliche Arbeitsspeicher besteht in fast allen Fällen aus preiswerten DRAM-Bausteinen mit hohen Zugriffszeiten bis zu 70 ns. Diese hohen Zugriffszeiten sind heute der Flaschenhals bei der Steigerung der Rechnerleistung. Hier setzen auch Technologien wie RAMBUS oder DDR (*Double Data Rate*) an.

Für eine Reihe von Systemen sind die Cachestufen nach deren Initialisierung beim Start des Rechners transparent. So zum Beispiel bei der PowerPC-Plattform, Alpha- oder x86-Systemen. Bei anderen Systemen muss das Betriebssystem eine Reihe von Aufgaben bei der Steuerung des Caches übernehmen, so zum Beispiel bei der Sparc-Architektur.

4.1.1 Speicherseiten

Der physische Speicher wird in *Pages* (Speicherseiten) strukturiert. Die Größe der Speicherseite ist durch das Makro `PAGE_SIZE` in der Datei `asm/page.h` definiert. Für 32-Bit-Architekturen wie x86 beträgt diese Größe meist 4 Kbyte, bei 64-Bit-Architekturen meist 8 Kbyte, wobei beim IA-64-Prozessor bis zu 64 Kbyte möglich sind. Tabelle 4.1 gibt die Größen der Speicherseiten für alle von Linux 2.4 unterstützten Prozessorarchitekturen an.

Mittlerweile unterstützen die *Memory Management Units* (MMU) unterschiedlicher Architekturen auch Speicherseiten von bis zu 4 Mbyte. Solche überdimensionalen Seiten sind besonders dann sinnvoll, wenn man aufeinander folgenden physischen Speicher in den virtuellen Adressraum einblenden will.

4.1.2 Virtueller Adressraum

Ein Prozess besitzt einen *virtuellen Adressraum*. Daten und Code des Prozesses sind über den virtuellen Adressraum verstreut. Die Größe des gesamten virtuellen Adressraums

hängt von der Anzahl der Bits ab, die in einer Speicheradresse verwendet werden. Die heutzutage üblichen Größen sind 32 Bit oder 64 Bit. Bei 32 Bit steht ein virtueller Adressraum von 4 Gbyte zur Verfügung, bei 64 Bit sind dies 2^{24} Terabyte beziehungsweise 2^{14} Petabyte oder 16 Exabyte.

Dieses Speichermodell vereinfacht das Programmieren. Der Programmierer kann sehr große Strukturen in aufeinander folgenden Adressen anlegen und muss keine Segment- oder Speicherseitengrenzen beachten; es gibt auch nur Zeiger von einer Größe. Die WIN16-Programmierung mit Near- und Far-Zeigern zeigt, wie umständlich es auch gehen kann.

Ein Prozess kann sich unter Linux im Nutzermodus oder im Systemmodus befinden. Jedes Programm, dass in einem UNIX-System gestartet wird, befindet sich erst einmal im niedrig privilegierten Nutzermodus. Wenn das Programm auf Systemressourcen zugreifen will, muss es über Systemrufe in den Systemmodus wechseln. Der Prozess führt dann Code im Kernel aus. Eine Besonderheit stellen Prozesse dar, die innerhalb des Kerns gestartet werden. Diese befinden sich ständig im Kernmodus. In der x86-Welt wird der Systemmodus als Ring 0 und der Nutzermodus als Ring 3 bezeichnet. Das Standardverfahren, um unter Linux vom Usermodus in den Systemmodus zu wechseln, ist der Aufruf des Softwareinterrupts 0x80. Da Hardwareinterrupts auch im Systemmodus ausgeführt werden, kann dafür dann das gleiche Verfahren verwendet werden.

Selbstverständlich darf ein Prozess im Nutzermodus nicht auf Speicher des Kerns zugreifen. Dies verhindern die Schutzmechanismen der Prozessorarchitektur. Einer Speicherseite kann der Zugriff vom System- und/oder dem Nutzermodus gewährt werden.

Linux blendet im Allgemeinen den physischen Speicher ab einem Offset (`PAGE_OFFSET`) in den virtuellen Speicherbereich ein. Der Zugriff auf diesen Speicher ist aber nur im Systemmodus erlaubt. Adressen im Kernel zeigen immer auf diesen Bereich. Allerdings gibt es keine Regel ohne Ausnahme: Die Implementation für die Sparc64-Architektur nutzt einen separaten virtuellen Adressraum für den Speicher des Systemmodus.

Der im Nutzermodus zur Verfügung stehende Adressraum ist damit kleiner als das theoretische Maximum und ist durch das Makro `TASK_SIZE` definiert. Die Werte für die verschiedenen Architekturen sind in Tabelle 4.1 angegeben. Den Bereich des virtuellen Speichers, der nur im Systemmodus zugreifbar ist, bezeichnen die Kernelprogrammierer als Kernelsegment und den im Nutzermodus zur Verfügung stehenden Bereich als Nutzersegment. Der Segmentbegriff ist sicherlich aus der x86-Architektur abgeleitet.

Segmente in der x86-Architektur lassen noch zusätzliche Zugriffsbeschränkungen auf Bereiche des virtuellen Adressraums zu. Unter der Version 2.0 gab es jeweils ein Daten- und ein Codesegment für den System- und Nutzermodus, die den Zugriff auf die entsprechenden Bereiche im virtuellen Adressraum einschränkten. Dies führte dazu, dass im Systemmodus unterschiedliche Segmentselektoren benutzt werden mussten, um auf die entsprechenden Daten zuzugreifen. Die Verwaltung der entsprechenden Segmentselektoren bringt natürlich mehr Aufwand mit sich. Seit der Version 2.2 sind zwar noch die Segmente definiert, allerdings sind sie jetzt für den ganzen virtuellen Adressraum definiert.

Auch nach dieser Vereinfachung kann der Kernel auf den Nutzerbereich nicht ohne zusätzliche Prüfungen zugreifen. Es ist mindestens zu überprüfen, ob Zeiger auch tatsächlich in den Benutzerbereich verweisen. Würde man auf diese Überprüfung verzichten, wäre es für ein Programm möglich, Daten im Kernel zu überschreiben. Man denke nur an einen `read`-Ruf, bei dem der Inhalt aus einer zuvor angelegten Datei an eine beliebige Stelle im Kernel geschrieben würde.

Bei älteren Intel-Prozessoren wurden im Systemmodus Schreibschutzbits für Speicherseiten nicht beachtet, die aber gerade im Benutzerbereich für effizientes Paging benutzt werden. Bei diesen Prozessoren muss vor dem Speicherzugriff dieses Bit explizit durch die Software überprüft werden. Da die Sparc64-Architektur für den System- und den Nutzermodus zwei unterschiedliche Adressräume unterstützt, kann der Zugriff auf Daten im Nutzersegment nie direkt, sondern nur über Makros oder Funktionen erfolgen.

Die Include-Datei `asm/uaccess.h` definiert die Makros und Funktionen für den Zugriff aus dem Kernel.

Das wichtigste Makro in dieser Datei ist `access_ok()`. Es überprüft, ob lesend (`VERIFY_READ`) beziehungsweise schreibend (`VERIFY_WRITE`) auf einen Nutzeradressbereich zugegriffen werden kann. Das Schreibschutzproblem bei älteren Prozessoren wird mit diesem Makro behoben. Im Fehlerfall wird 0 zurückgegeben. Für die Kompatibilität mit Code für Linux 2.0 existiert die Inline-Funktion `verify_area()`. Diese gibt im Fehlerfall `-EFAULT` zurück.

Die folgende Liste gibt einen Überblick über alle Funktionen beziehungsweise Makros, die für das Lesen und Schreiben von Daten aus dem Nutzersegment verwendet werden können.

```
/* Lesen eines skalaren Wertes im Nutzersegment */
get_user(val, ptr);

/* Schreiben eines skalaren Wertes in das Nutzersegment */
put_user(val, ptr);

/* Kopieren von Daten aus dem Nutzersegment */
copy_from_user(to, from, n);

/* Kopieren von Daten in das Nutzersegment */
copy_to_user(to, from, n);

/* Kopieren einer Zeichenkette aus dem Nutzersegment */
strncpy_from_user(to, from, n);

/* Ermitteln der Länge einer Zeichenkette im Nutzersegment */
strnlen_user(str, n);
strlen_user(str);       /* sollte nicht mehr benutzt werden */

/* Löschen von Speicher im Nutzersegment */
clear_user(mem, len);
```

Bei den Get- und Put-Funktionen gibt der Typ des Zeigers an, wie groß die Daten sind, die gelesen beziehungsweise geschrieben werden sollen. Für alle Funktionen außer `strnlen_user()` und `strlen_user()` gibt es ein Pendant, das mit zwei Unterstrichen (z. B. `__get_user()`) anfängt, bei dem das Makro `access_ok()` nicht aufgerufen wird. Dem Aufruf dieser Funktionen muss dann ein für die zugegriffenen Daten entsprechender Aufruf von `access_ok()` vorausgegangen sein. Die Makros und Funktionen dürfen nicht in einem Interruptkontext aufgerufen werden, da es notwendig sein kann, die Speicherseite erst von einem externen Blockgerät einzulesen.

4.1.3 Übersetzung der linearen Adresse

Die linearen Adressen müssen vom Prozessor oder durch eine separate MMU in eine physische Adresse umgesetzt werden. Im architekturunabhängigen Speichermodell betrachtet man die *Pageübersetzung* als einen dreistufigen Vorgang. Die lineare Adresse wird dabei in vier Teile zerlegt. Der erste Teil wird als Index in das Pagedirectory verwendet. Der Eintrag im Pagedirectory verweist in Linux auf das so genannte *Page Middle Directory*, das mittlere Pagedirectory. Der zweite Teil der Adresse stellt einen Index in einem mittleren Pagedirectory dar. Der so referenzierte Eintrag verweist auf eine Pagetabelle. Der dritte Teil dient als Index in dieser Pagetabelle. Der referenzierte Eintrag sollte dabei möglichst auf eine Speicherseite im physischen Speicher zeigen. Der vierte Teil der Adresse gibt das Offset innerhalb der selektierten Speicherseite an. Abbildung 4.1 stellt die Verhältnisse schematisch dar.

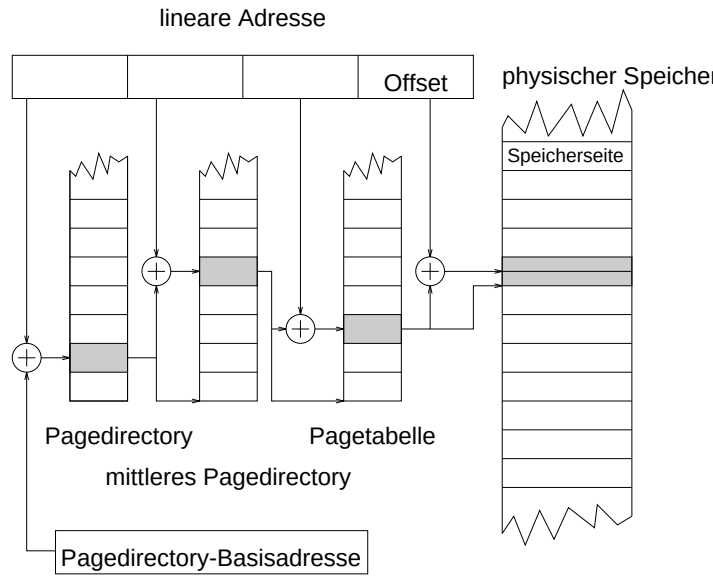


Abbildung 4.1: Übersetzung der linearen Adresse im architekturunabhängigen Speichermodell

Die x86-Prozessoren unterstützten ursprünglich nur eine zweistufige Übersetzung der linearen Adresse. Bei der Umsetzung des architekturunabhängigen Speichermodells hilft

ein Trick. Ein Eintrag im Pagedirectory wird als mittleres Pagedirectory mit nur einem Eintrag betrachtet. Die Operationen zum Zugriff auf Pageübersetzungstabellen müssen diese Festlegung natürlich berücksichtigen. Wenn der x86-Prozessor die *Page Size Extension* (PSE) unterstützt, wird der physische Speicher als 4-Mbyte-Speicherseiten in das Kernelsegment eingeblendet. Das spart Speicherplatz und ist schneller als der Zugriff über 4-Kbyte-Speicherseiten. In diesem Fall ist die Adressübersetzung nur noch einstufig, da der Verweis auf die 4-Mbyte-Page in das Pagedirectory eingetragen ist.

x86-Rechner mit mehr als 896 Mbyte stellen Linux-2.2-Systeme vor ein Problem. Der virtuelle Adressraum des Kernels, das Kernelsegment, ist ein Gigabyte groß. Für `vmalloc()` und für die *Advanced Programmable Interrupt Controller* (APICs) von Multiprozessorsystemen werden 128 Mbyte Adressraum reserviert. Linux kann also nicht mehr den gesamten physischen Speicher in das Kernelsegment einblenden. Linux legt eine Pagetabelle an, um den restlichen physischen Speicher temporär in das Kernelsegment einblenden zu können. Der so reservierte virtuelle Adressbereich beginnt ab `PKMAP_BASE` (0xfe000000) und hat die Größe von 4 Mbyte. Die Kernelprogrammierer haben dieses Feature wohl in Erinnerung an den *x86 Real Mode* „Highmem“ getauft.

Dieser Highmem-Support hilft bis zu einer physischen Speichergröße von 4 Gbyte; so groß ist der physische Adressraum bei 32-Bit-Adressen. Intel spendierte aber in weiser Voraussicht dem Pentium-Pro vier zusätzliche Adresspins und kreierte die *Physical Address Extension* (PAE). Sie war notwendig, da die bisherige zweistufige Adressübersetzung nur 32 Bit große physische Adressen unterstützte. Die PAE beherrscht auch eine dreistufige Adressübersetzung wie im abstrakten Modell von Linux. Die Größe des virtuellen Adressraums bleibt unverändert bei 4 Gbyte. Der zusätzliche physische Speicher kann über das Highmem-Feature in das Kernelsegment eingeblendet werden.

64-Bit-Architekturen wie der Alpha-Prozessor benutzen genauso wie die PAE der x86-Prozessoren eine dreistufige Adressübersetzung. Eine nur zweistufige Adressübersetzung würde zu überdimensionalen Pagedirectories und Pagetabellen führen, wollte man die Nutzung der linearen Adresse nicht auf 32 Bit reduzieren. Beim Alpha-Prozessor ist eine Speicherseite für Linux 8 Kbyte groß. Da die Speicherseitenverzeichnisse und die -tabellen jeweils eine Speicherseite belegen und ein Eintrag jeweils 64 Bit groß ist, können pro Ebene nur 1.024 Einträge verwaltet werden. Da im Pagedirectory die Basisadresse des Pagedirectories mitverwaltet wird, ist die Größe des virtuellen Adressraums auf $1.023 * 1.024 * 1.024 * 8.192$ Byte $= 8.184$ Gbyte beschränkt. Das sind knapp 8 Tbyte. Für ein Nutzersegment werden davon 4 Tbyte $= 2^{42}$ Byte zur Verfügung gestellt. Tabelle 4.1 gibt einen Überblick über die Übersetzungsstufen und Nutzersegmentgrößen, für alle von Linux 2.4 unterstützten Architekturen. Die Anzahl der Stufen bezieht sich auf Speicher im Nutzersegment.

4.1.4 Pagedirectories

Der Kernel verwaltet die Pagedirectories und -tabellen für das Kernelsegment und die Nutzersegmente der Prozesse. Die dafür notwendigen Datentypen, Makros und Funktionen sind in den Dateien `asm/page.h`, `asm/pgtable.h` und `asm/pgalloc.h` definiert.

Prozessor	Größe einer Speicherseite PAGE_SIZE	Stufen der Page-übersetzung	Größe des Benutzer-adressraums TASK_SIZE
Alpha	8 Kbyte	3	4 Tbyte
ARM 26 Bit	16 oder 32 Kbyte	2	bis zu 26 Mbyte
ARM 32 Bit	4 Kbyte	2	3 Gbyte
IA-64	4, 8, 16 oder 64 Kbyte	3	10 Exabyte
M68000	4 Kbyte	3	224 Mbyte
M68000 SUN3	8 Kbyte	2	3,75 Mbyte
MIPS	4 Kbyte	2	2 Gbyte
MIPS64	4 Kbyte	3	1 Petabyte
PARISC	4 Kbyte	2 oder 3	3 Gbyte
PowerPC	4 Kbyte	2	2 Gbyte
S390	4 Kbyte	2	2 Gbyte
SH	4 Kbyte	2	1.984 Mbyte
Sparc	4 Kbyte	3	3.75 Gbyte
Sparc SUN4	8 Kbyte	2	3.75 Gbyte
Sparc 64	8 Kbyte	3	16 Exabyte – 16 Gbyte
x86	4 Kbyte	2 oder 3	3 Gbyte

Tabelle 4.1: Parameter der Speicherseitenverwaltung für die von Linux *unterstützten Architekturen*

Der Kurzbezeichner für Pagedirectories der ersten Stufe ist `pgd` und für die der zweiten Stufe `pmd`. Ein Eintrag in den Verzeichnissen hat den Datentyp `pgd_t` beziehungsweise den Datentyp `pmd_t`. Beide Typen sind als Strukturen definiert, um versehentliche Konversionen in Integerdatentypen zu vermeiden. Der Kernel referenziert die Pagedirectories als Zeiger auf den ersten Eintrag im jeweiligen Directory: `pgd_t*` und `pmd_t*`.

`pgd_val()`, `pmd_val()` Diese Makros erlauben den Zugriff auf den eigentlichen Wert des Verzeichniseintrags. Abhängig von der Prozessorarchitektur kann dies ein 64-Bit- beziehungsweise ein 32-Bit-Wert sein.

`pgd_alloc()`, `pmd_alloc()` Diese Funktionen stellen eine Speicherseite für das jeweilige Verzeichnis bereit. Alle Einträge sind so initialisiert, dass sie kein mittleres Pagedirectory beziehungsweise keine Pagetabelle referenzieren. Im Kernel 2.4 können sich die Makros aus einer so genannten *Quicklist* bedienen, in der schon entsprechend initialisierte Speicherseiten eingetragen sind. Der Kernel definiert Varianten dieses Makros, die Directories speziell für den Kernel allozieren (`pgd_alloc_kernel()` und

`pmd_alloc_kernel()`). Diese Makros sind aber bei den meisten Prozessorarchitekturen mit `pgd_alloc()` beziehungsweise `pmd_alloc()` identisch.

`pgd_free()`, `pmd_free()` Die Pagedirectories werden wieder freigegeben. Im Kernel 2.4 können sie in die oben erwähnte Quicklist eingetragen werden.

`pgd_clear()`, `pmd_clear()` Dieses Makro löscht den Eintrag im Pagedirectory. Er verweist danach nicht mehr auf ein mittleres Pagedirectory beziehungsweise auf eine Pagetabelle.

`pgd_none()`, `pmd_none()` Beide Makros testen, ob der Directoryeintrag auf kein mittleres Pagedirectory oder eine Pagetabelle verweist.

`pgd_present()`, `pmd_present()` Wenn ein Verzeichniseintrag doch eine Referenz auf ein mittleres Pagedirectory beziehungsweise eine Pagetabelle enthält, geben diese Makros ein positives Ergebnis zurück. Das Ergebnis ist die Negation des Ergebnisses von `pgd_none()` oder entsprechend `pmd_none()`.

`pgd_bad()`, `pmd_bad()` Diese Einträge überprüfen die Korrektheit von Einträgen in den Directories, die ein mittleres Pagedirectory oder eine Pagetabelle referenzieren.

`pgd_page()`, `pmd_page()` Diese Makros geben den Zeiger auf das referenzierte mittlere Pagedirectory oder die referenzierte Pagetabelle zurück.

`__pgd_offset()`, `__pmd_offset()` Diese Funktion gibt zu einer linearen Adresse den Index in dem jeweiligen Pagedirectory zurück. Für das oberste Pagedirectory ist das Makro `pgd_index()` identisch mit `__pgd_offset()`.

`pgd_offset()`, `pgd_offset_k()` Das Makro `pgd_offset()` ermittelt aus der Speicherverwaltungsstruktur (`struct mm_struct`) und der linearen Adresse den Zeiger des zugehörigen Eintrags im obersten Pagedirectory. `pgd_offset_k()` benutzt die Speicherverwaltungsstruktur des Kerns, um den Zeiger auf den Pagedirectoryeintrag zu ermitteln.

`pmd_offset()` Um aus einer linearen Adresse einen Zeiger auf einen Eintrag im mittleren Pagedirectory zu ermitteln, benutzen die Kernelprogrammierer dieses Makro. Dazu muss aber das zugehörige obere Pagedirectory bekannt sein.

`set_pgd()`, `set_pmd()` Der Kernel muss irgendwann die Pagedirectories mit Einträgen füllen. Dazu rufen die Kernelprogrammierer diese Makros auf.

4.1.5 Die Pagetabelle

Der Datentyp `pte_t` definiert einen Pagetabelleneintrag. Der Kernel verweist auf eine Pagetabelle mit einem Zeiger auf den ersten Eintrag der Tabelle. Die wichtigste Aufgabe des Pagetabelleneintrags ist die Adressierung einer Seite im physischen Speicher.

`pte_val()` Dieses Makro gibt den Wert eines Pagetabelleneintrags zurück.

`pte_alloc(), pte_alloc_kernel()` Anhand eines Eintrags des mittleren Pagedirectories und der linearen Adresse wird der Pagetabelleneintrag für die übergebene lineare Adresse zurückgegeben. Existiert die zugehörige Pagetabelle noch nicht, wird sie alloziert. Für Adressen im Kernel ist `pte_alloc_kernel()` zu benutzen. Eine Quicklist kann zur Anwendung kommen, um die Pagetabellen nicht neu initialisieren zu müssen.

`pte_free(), pte_free_kernel()` Diese Funktionen geben eine Pagetabelle wieder frei. Die Tabelle sollte keine initialisierten Einträge mehr aufweisen, da die Tabelle unter Umständen wieder in die Quicklist zurückgeschrieben wird.

`pte_page()` Diese Funktion ermittelt anhand des Pagetabelleneintrags einen Zeiger auf einen Eintrag in die `mem_map`, in der alle physischen Speicherseiten des Rechners eingetragen sind.

`pte_offset()` Dieses Makro berechnet anhand eines Eintrags im mittleren Pagedirectory und der linearen Adresse den zugehörigen Pagetabelleneintrag.

`set_pte()` Der Wert des Pagetabelleneintrags wird gesetzt, aber der gesetzte Eintrag darf entweder nicht präsent sein oder durch die Hardware nicht aktualisiert werden. Ansonsten sollte `ptep_get_and_clear()` eingesetzt werden.

`ptep_get_and_clear()` Diese Funktion gibt den Pagetabelleneintrag zurück und löscht ihn.

`pte_same()` Diese Funktion vergleicht zwei Pagetabelleneinträge.

Flags im Pagetabelleneintrag beschreiben die erlaubten Zugriffsarten auf die Speicherseite und ihren Zustand. Linux bildet die Flags der architekturabhängigen Flags auf Attribute ab, die für alle Architekturen gelten. Das Präsenzattribut zeigt der MMU an, ob auf die Speicherseite zugegriffen werden kann. Für die Berechtigungen zum Lesen, Schreiben und Ausführen des Inhalts der Speicherseite gibt es jeweils ein eigenes Attribut. Ein Attribut zeigt an, ob auf die Speicherseite zugegriffen worden ist. Das Attribut sagt also etwas über das Alter der Speicherseite aus. Das Dirty-Attribut zeigt an, wenn der Inhalt der Speicherseite modifiziert worden ist.

Der Kernel definiert eine Reihe von Attributkombinationen als Makros vom Typ `pgprot_t`. Für einzelne Architekturen sind zusätzliche spezielle Attributkombinationen definiert.

`PAGE_NONE` Durch den Pagetabelleneintrag wird keine physische Speicherseite referenziert.

`PAGE_SHARED` Alle Zugriffe auf die Speicherseite sind erlaubt.

`PAGE_READONLY` Auf diese Speicherseite ist nur lesender oder ausführender Zugriff erlaubt. Bei schreibendem Zugriff wird eine Exception generiert, die es erlaubt, den Fehler zu behandeln. Dabei kann die Speicherseite kopiert werden und der Pagetabelleneintrag auf die physische Adresse der neuen Seite gesetzt werden und seine Attribute auf `PAGE_SHARED`. Genau das bedeutet Copy-On-Write.

`PAGE_COPY` Dieses Makro entspricht `PAGE_READONLY`.

`PAGE_KERNEL` Der Zugriff auf diese Speicherseite ist nur im Kernelsegment erlaubt.

`PAGE_KERNEL_RO` Der Zugriff auf diese Speicherseite ist nur lesend im Kernelsegment erlaubt.

`asm/pgtable.h` definiert zusätzlich die Makros `__P000` bis `__P111` und `__S000` bis `__S111`, die zusammen mit dem Makro `_PAGE_NORMAL()` die Definitionen beliebiger Kombinationen der Schutzattribute erlauben. Die Bitpositionen in den Makronamen sind als „`xwr`“ zu interpretieren. Bei den mit `__P` beginnenden Makros wird die Stelle des Schreibattributs als Copy-On-Write-Attribut interpretiert. Die mit `__S` beginnenden Makros beschreiben Seiten, die als `PAGE_SHARED` gekennzeichnet sind; sie sind also zum schreibenden Zugriff freigegeben.

Die x86-Architektur unterstützt nicht alle Kombinationen der Attribute für Lesen, Schreiben und Ausführen. Tabelle 4.2 zeigt die Semantik aller möglichen Attributkombinationen unter Verwendung der klassischen Unix-Schreibweise „`rwx`“ für solche Attribute.

Attributkombination	x86-Semantik
`---`	`---`
`--x`	`r-x`
`-w-`	`rwx`
`-wx`	`rwx`
`r--`	`r-x`
`r-x`	`r-x`
`rw-`	`rwx`
`rwx`	`rwx`

Tabelle 4.2: Semantik der Schutzattributkombinationen bei x86-Prozessoren

Der Kernel verwaltet die Pagetabelleneinträge mit einer Reihe von Makros.

`mk_pte()`, `mk_pte_phys()` Die Funktion bzw. das Makro `mk_pte()` erzeugt aus einem Zeiger in das Verzeichnis der physischen Speicherseiten `mem_map` und den Schutzattributen (Typ `pgprot_t`) einen Pagetabelleneintrag. Die Funktion `mk_pte_phys()` macht das Gleiche, benutzt aber die physische Speicherseitenadresse als Input.

`pte_modify()` Diese Funktion verändert die Attribute eines Speicherseiteneintrags.

`pte_none()`, `pte_clear()` Das Makro `pte_none()` überprüft, ob der Pagetabelleneintrag leer ist. Mit `pte_clear()` wird der Eintrag gelöscht.

`pte_present()` Diese Funktion testet, ob das Präsenzattribut des Pagetabelleneintrags gesetzt ist.

`pte_dirty()`, `pte_mkdirty()`, `pte_mkclean()` Diese Funktionen verwalten das Dirty-Attribut eines Pagetabelleneintrags. Die Funktion `pte_mkdirty()` setzt das Dirty-Attribut, und `pte_mkclean()` löscht es.

`ptep_test_and_clear_dirty()` Diese Funktion testet das Dirty-Flag eines Pagetabelleneintrags und löscht es.

`pte_exec()`, `pte_mkexec()`, `pte_exprotect()` Diese Funktionen verwalten das Ausführungsberechtigungsattribut. Die Funktion `pte_exec()` testet es, `pte_mkexec()` setzt es und `pte_exprotect()` löscht es.

`pte_young()`, `pte_mkyoung()`, `pte_mkold()` Diese Funktionen verwalten ein Attribut, das Aussagen über den Zugriff auf die Speicherseite macht. Die Speicherverwaltungseinheit des Prozessors setzt bei einem Speicherzugriff das Attribut auf *jung*. Mit diesem Attribut kann der Kernel feststellen, ob ein Zugriff auf die Speicherseite erfolgt ist, seitdem das Attribut auf *alt* gesetzt wurde.

`ptep_test_and_clear_young()` Wenn der Kernel das Alterattribut überprüfen soll und dabei das Attribut löschen soll, wird diese Inline-Funktion verwendet.

`pte_read()`, `pte_mkread()`, `pte_exprotect()` Wie bei dem Attribut zur Ausführungsberechtigung werden diese Funktionen zur Manipulation der Leseberechtigung benutzt.

`pte_read()`, `pte_mkread()`, `pte_exprotect()` Der Kernel benutzt diese Funktionen, um das Leseberechtigungsattribut zu verwalten.

4.2 Der virtuelle Adressraum eines Prozesses

Wie schon im vorigen Abschnitt erwähnt, ist der virtuelle Adressraum eines Linux-Prozesses segmentiert. Es wird zwischem dem Kernelsegment und dem Nutzersegment unterschieden.

4.2.1 Das Nutzersegment

Im Nutzermodus, der Privilegierungsstufe 3 bei x86-Prozessoren, kann ein Prozess nur auf das Nutzersegment zugreifen. Da das Nutzersegment die Daten und den Code des Prozesses enthält, muss sich der Inhalt dieses Segments von dem anderer Prozesse unterscheiden. Dies wird durch andere Pagedirectories oder zumindest durch einzelne unterschiedliche Pagetabellen erreicht. Im Systemruf *fork* werden die Pagedirectories und die Pagetabellen des Elternprozesses für den Kindprozess kopiert. Eine Ausnahme bilden die Pagetabellen des Kernelsegments. Diese werden von allen Prozessen geteilt.

Zum Systemruf *fork* gibt es die Alternative *clone*. Beide Systemrufe erzeugen einen neuen Thread, der aber bei *clone* seinen Speicher mit dem aufrufenden Thread teilen kann. Threads werden demnach in Linux als Tasks betrachtet, die sich mit anderen Prozessen ihren Adressraum teilen. Über Parameter des Systemrufs *clone* kann die Behandlung weiterer taskspezifischer Ressourcen, z. B. des Stacks, gesteuert werden. Die unter Linux zur Verfügung gestellte POSIX-Thread-Bibliothek benutzt diesen Systemruf, um Threads zu erzeugen. Threads werden immer dann eingesetzt, wenn es schwierig ist, Programmfunktionen in mehrere Prozesse aufzuteilen, um sie zu parallelisieren. Dieser Performancevorteil bedeutet aber auch, dass man Wettbewerbsbedingungen (*race conditions*) riskiert, die bei Prozessen weitgehend durch den Kernel behandelt werden.

Die Struktur des Nutzersegments während der Ausführung im ELF-Format ist in Abbildung 4.2 dargestellt. Das Nutzersegment eines jeden Prozesses, außer dem Idle-Prozess (Prozessnummer 0), ist durch das Laden beziehungsweise Einblenden einer Binärdatei durch den Systemruf *execve* initialisiert worden. Ein durch *fork* erzeugter Prozess übernimmt diese Struktur vom Vaterprozess.

Die im Nutzersegment abgebildeten Shared Libraries bedürfen einer Erläuterung. Ursprünglich wurde unter Linux der gesamte Code eines Programms statisch in ein Binary gebunden. Das führte dazu, dass mit dem Wachsen der Bibliotheken die Binaries immer größer wurden. Um das zu verhindern, wurden die Bibliotheken in separaten Library-Dateien gespeichert und beim Starten des Programms geladen. Allerdings mussten durch Beschränkungen im a.out-Format die Shared Libraries an fixe Adressen gebunden werden. Alle Shared Libraries benötigten eigene voneinander verschiedene feste Adressbereiche. Mit dem Objektdateiformat ELF wurden eine Dateistruktur sowie Methoden definiert, mit dem dies überflüssig wurde und Shared Libraries während der Programmausführung nachgeladen werden konnten. Bei entsprechend flexibler Auslegung können in ein Programm nun Shared Libraries eingebunden werden, die zur Kompilation des Programms noch gar nicht bekannt waren. Die automatischen Module der Scriptsprache Perl sind dafür ein sehr gutes Beispiel. Die Shared Libraries werden an dynamisch

ermittelten Adressen eingeblendet. Allerdings setzt das voraus, dass die Bibliotheken als positionsabhängiger Code (PIC) erzeugt worden sind. Das heißt, es dürfen keine absoluten Adressreferenzen im übersetzten Code vorhanden sein, und ein Register (x86: EBX) wird immer für die Referenzierung globaler Adressen belegt.

Linux unterstützt auch heute noch das klassische a.out-Format. Das strukturiert allerdings das Nutzersegment anders. Der Programmtext startet an der virtuellen Adresse 0, und die dynamischen Bibliotheken werden zwischen Heap und Stack an festen Adressen eingeblendet. Die fixe Adressvergabe und die wesentlich umständlichere Erstellung von Shared Libraries im a.out-Format haben zur Ablösung dieses Binärformats geführt.

Zusätzlich kann Linux Skripte wie richtige Binaries behandeln. Beim Aufruf eines Skripts wird der in der ersten Zeile nach den Zeichen `#!` angegebene Interpreter mit dem Skript als Argument gestartet. Für binäre Formate, die `#!` nicht unterstützen können, gibt es seit Linux 2.2 das `binfmt_misc`-Feature, das es zulässt, anhand der Dateiextension oder anhand von spezifischen Bytefolgen, den *Magics* am Anfang der Datei, bestimmte Programme aufzurufen, die als Interpreter für diese Dateien dienen. In Linux 2.0 gab es ein solches Feature nur für Java-Dateien, obwohl Emulatoren wie DOSEMU davon auch profitieren können.

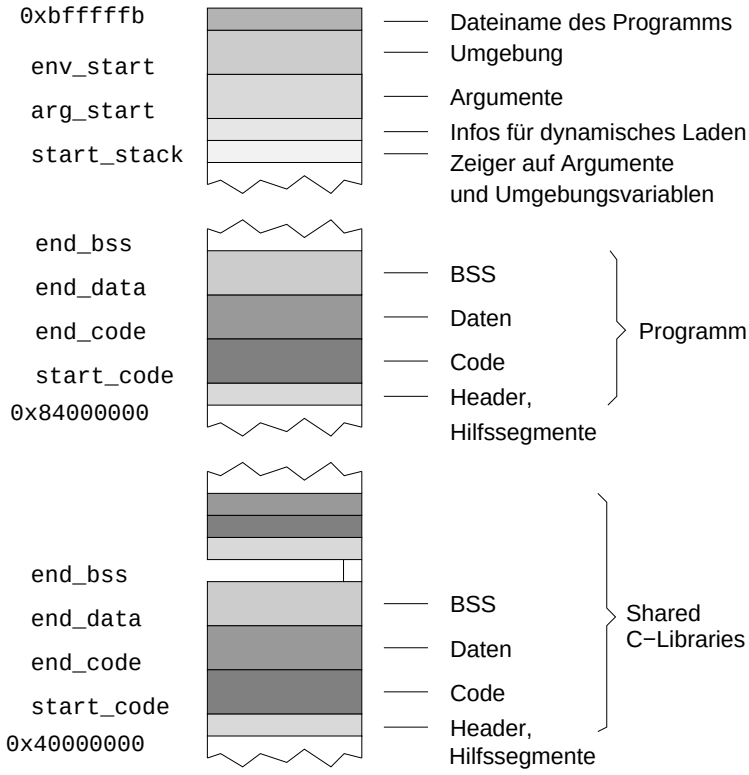


Abbildung 4.2: Struktur des Nutzersegments eines Prozesses, dessen Binärdatei im ELF-Format vorliegt

Am oberen Ende des Nutzersegments sind der Dateiname des Programms, die Umgebungsvariablen und die Argumente des Prozesses als Folge von nullterminierten Zeichenketten gespeichert. Darunter befinden sich Zeigertabellen für die Argumente und die Umgebungsvariablen, auf die in einem C-Programm mit `argv` beziehungsweise `environ` verwiesen wird. Erst daran anschließend beginnt der Stack.

4.2.2 Virtuelle Speicherbereiche

Da eine Shared Library sehr groß sein kann, wäre es von Nachteil, wenn ihr gesamter Code ständig im physischen Speicher gehalten würde. Die laufenden Prozesse werden sicherlich nicht gleichzeitig alle in einer Bibliothek enthaltenen Funktionen nutzen. Das Laden des Codes nicht benutzter Funktionen verbraucht Speicherressourcen und ist überflüssig. Auch bei großen Programmen gibt es sicherlich Codeabschnitte, die von einem Prozess nie durchlaufen werden, da zum Beispiel bestimmte Programmfunktionen nicht genutzt werden. Das Laden dieser Programmteile ist genauso wenig notwendig wie das Laden von nicht benutzten Abschnitten einer Bibliothek.

Virtuelle Speicherbereiche werden häufig von Hardwaregeräten genutzt, die Speicher in den Adressraum einblenden. Die Kommunikation zwischen einer Anwendung und der Hardware erfolgt dann wesentlich schneller über virtuelle Speicherbereiche als über Systemrufe. Ein Beispiel dafür sind die Framebuffergeräte, mit denen der Speicher der Grafikkarte in einen Speicherbereich eingeblendet werden kann.

Sind mit einer ausführbaren Datei zwei Prozesse gestartet worden, muss der Programmcode nicht zweimal in den Hauptspeicher geladen werden. Beide Prozesse können gemeinsam denselben Code im primären Speicher ausführen. Es ist auch möglich, dass große Teile des Datensegments dieser Prozesse übereinstimmen. Diese könnten ebenfalls zwischen den Prozessen geteilt werden, solange keiner der beiden Prozesse diese Daten modifiziert. Erst wenn ein Prozess eine Speicherseite modifiziert, sollte sie entsprechend der Methode Copy-On-Write kopiert werden.

Reserviert ein Prozess sehr große Mengen an Speicher, so wäre die Allozierung von freien physischen Speicherseiten Verschwendung. Der Prozess wird diesen Speicher erst später vollständig nutzen, möglicherweise auch nie. Ein Ansatz zur Lösung dieses Problems ist die Anwendung von Copy-On-Write, wobei bei einem Lesezugriff eine systemweit einzige leere Speicherseite in die Pagetabellen des Prozesses eingetragen wird. Erst bei Modifikationen an einer bestimmten Adresse im Nutzersegment muss diese Speicherseite kopiert und an der entsprechenden Stelle in den linearen Adressraum eingeblendet werden.

Damit wird deutlich, dass sich die einzelnen Bereiche im Nutzersegment durch die Attribute für die Pagetabelleneinträge der Speicherseite, die Behandlungsroutinen für Zugriffsfehler und die Strategien für die Auslagerung auf sekundären Speicher unterscheiden. Bei der Entwicklung von Linux wurde deshalb die Abstraktion *virtueller Speicherbereich* eingeführt. Ein virtueller Speicherbereich wird durch die Datenstruktur `vm_area_struct` in der Datei `linux/mm.h` definiert:

```
struct vm_area_struct {
        /* Parameter für den virtuellen Speicherbereich */
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;

        /* verbundene VM-Bereiche einer Task, sortiert nach */
         * Adressen */
        struct vm_area_struct *vm_next;

        pgprot_t vm_page_prot;
        unsigned long vm_flags;

        /* AVL-Baum der VM-Bereich einer Task, sortiert nach */
         * Adressen */
        short vm_avl_height;
        struct vm_area_struct * vm_avl_left;
        struct vm_area_struct * vm_avl_right;

        /* Für Bereiche mit einem Adressraum und Hintergrundspeicher
         * eine der address_space->i_mmap{,shared}-Listen,
         * für SHM-Bereiche die Liste der Attachments; ansonsten
         * unbenutzt.
         */
        struct vm_area_struct *vm_next_share;
        struct vm_area_struct **vm_pprev_share;

        struct vm_operations_struct * vm_ops;
        /* Offset in PAGE_SIZE Einheiten */
        unsigned long vm_pgoff;
        struct file * vm_file;
        unsigned long vm_raend;
        void * vm_private_data;
};
```

Die Komponenten `vm_start` und `vm_end` legen die Anfangs- und Endadresse des durch die Struktur verwalteten virtuellen Speicherbereichs fest.

`vm_mm` ist ein Zeiger auf den Speichermanagementbereich in der Prozesstabelle. In `vm_page_prot` stehen die Schutzattribute für Speicherseiten aus diesem Bereich und in `vm_flags` die Informationen über den Typ des Speicherbereichs. Dazu gehören unter anderem die aktuellen Zugriffsrechte auf den Speicherbereich und Festlegungen, welche Schutzattribute gesetzt werden dürfen.

Die virtuellen Speicherbereiche eines Prozesses werden in einer doppelt verketteten Liste, nach Adressen sortiert, verwaltet. Zusätzlich dazu gibt es einen AVL-Baum. Für spezielle Zwecke, z. B. dem Einblenden einer Datei oder die Benutzung des Shared Memory von System V, sind zusätzlich Felder für eine doppelt verkettete Ringliste definiert.

Der Inode-Zeiger `vm_file` verweist auf die Datei oder das Gerät, deren bzw. dessen Inhalt ab dem Offset `vm_pgoff` in den virtuellen Speicherbereich eingeblendet ist. Ist

dieser Zeiger auf NULL gesetzt, wird von einem anonymen Mapping gesprochen. Der Integerwert vm_raend dient für das Read-Ahead von eingeblendeten Dateien. Das Feld vm_private_data findet bei der Implementierung des Shared Memory von System V Verwendung.

Da die virtuellen Speicherbereiche nur reserviert werden, kommt es bei einem Zugriff auf Speicher eines solchen Bereichs zu einem Seitenfehler. Entweder existiert für die Seite noch kein Eintrag im Pagedirectory, oder auf die referenzierte Speicherseite ist kein schreibender Zugriff erlaubt. Der Prozessor generiert eine Seitenfehler-Ausnahmeunterbrechung, und der Prozessor aktiviert die entsprechende Behandlungsroutine. Diese Routine ruft dann die Operationen auf, die die benötigten Speicherseiten bereitstellen. Zeiger auf solche Operationen gibt es in vm_ops. Darüber hinaus enthält vm_ops Zeiger für zusätzliche Operationen, die das Kopieren und Freigeben eines virtuellen Speicherbereichs organisieren. Die Struktur vm_operations_struct definiert die möglichen Funktionszeiger, mit denen unterschiedlichen Bereichen verschiedene Operationen zugeordnet werden können. vm_ops wird nicht für anonyme Mappings benutzt; diese werden durch globale Funktionen behandelt.

```
struct vm_operations_struct {
  void (*open)(struct vm_area_struct * area);
  void (*close)(struct vm_area_struct * area);
  struct page * (*nopage)(struct vm_area_struct * area,
    unsigned long address,
    int write_access);
};
```

Die Funktion open() wird aufgerufen, wenn ein virtueller Speicherbereich in das Nutzersegment eingeblendet wird. Wird ein virtueller Speicherbereich aus einem Prozess entfernt, wird close() aufgerufen.

Mit nopage() werden Fehler beim Zugriff auf eine nicht im physischen Speicher vorhandene und nicht ausgelagerte Speicherseite behandelt. Die Funktion hat nur die Aufgabe, die Speicherseite an der übergebenen Adresse einzulesen und einzublenden. Der Parameter write_access legt fest, ob die Datei privat oder als geteilte Seite eingeblendet wird.

Die Funktion do_mmap_pgoff() fügt dem Prozess virtuelle Speicherbereiche hinzu.

```
int do_mmap_pgoff(struct file * file, unsigned long addr,
            unsigned long len, unsigned long prot,
            unsigned long flags, unsigned long pgoff)
```

Sie blendet die mit file referenzierte Datei in den virtuellen Adressraum ein. Der Offset in der Datei (pgoff) wird in Speicherseiten angegeben. Der Parameter prot legt den Zugriffsschutz für den virtuellen Speicherbereich fest; Tabelle 4.3 zeigt eine Übersicht.

Stellt do_mmap_pgoff() fest, dass file Null ist und damit keine Datei referenziert, legt die Funktion einen anonymen Speicherbereich an. Aus Sicht des Prozesses sind die anonymen virtuellen Speicherbereiche mit Null-Bytes gefüllt. Greift der Prozess erstmalig auf eine Speicherseite im virtuellen Adressraum zu, referenziert der Kernel immer

Wert	Erläuterung
PROT_READ	Bereich kann gelesen werden.
PROT_WRITE	Bereich kann beschrieben werden.
PROT_EXEC	Bereich kann ausgeführt werden.
PROT_NONE	Auf den Bereich kann nicht zugegriffen werden. Dies wird zur Zeit nicht unterstützt.

Tabelle 4.3: Werte für das Argument `prot` *der Funktion* `do_mmap()`

dieselbe globale leere Speicherseite in der entsprechenden Pagetabelle. Die globale leere Speicherseite wird durch das Makro `ZERO_PAGE()` referenziert. Schreibt der Prozess erstmalig in einer Speicherseite, schreibt der Kernel die Adresse einer neuen, frisch allozierten, leeren Speicherseite in den zugehörigen Pagetabelleneintrag.

Der Parameter `flags` gibt Attribute für den virtuellen Speicherbereich an. `MAP_FIXED` legt fest, dass der Kernel den Speicherbereich exakt an der angegebenen Adresse einblendet. Der Aufrufer hat zu beachten, dass im angegebenen Adressbereich keine virtuellen Speicherbereiche eingeblendet sind. Die Flags `MAP_SHARED` und `MAP_PRIVATE` steuern die Behandlung von Speicheroperationen im virtuellen Speicherbereich. Bei `MAP_SHARED` werden alle Schreiboperationen in denselben Speicherseiten ausgeführt. Bei `MAP_PRIVATE` werden bei einem Schreibzugriff die Speicherseiten dupliziert. Das Setzen von `MAP_PRIVATE` führt zu in einem Copy-On-Write für den jeweiligen Speicherbereich. Wegen der Beschränkungen der x86-Architektur — die Rechte für das Lesen und Ausführen werden in einem Flag behandelt —, ist eine hundertprozentig korrekte Implementierung von `do_mmap_pgoff()` auf x86-Prozessoren nicht möglich.

4.2.3 Der Systemruf brk

Der Wert des Feldes `brk` eines Prozesstabelleneintrags (siehe auch Abschnitt 3.1.1) zeigt beim Start des Prozesses auf das Ende des BSS-Segments für nicht statisch initialisierte Daten. Durch Veränderung dieses Zeigers kann der Prozess dynamisch Speicher allozieren und wieder freigeben. Die Standard-C-Funktion `malloc()` verändert den `brk`-Zeiger, um den Heap bei entsprechendem Bedarf zu vergrößern. Das Allozieren von Speicher führt natürlich zu Änderungen im Pagedirectory des Prozesses. Das Pagedirectory kann natürlich nur der Kernel ändern. Zu diesem Zweck stellt er für das Lesen und das Modifizieren des `brk`-Zeigers den Systemruf *brk* zur Verfügung. Der Systemruf überprüft die Gültigkeit eines neu gesetzten Wertes. Ein Test prüft, ob überhaupt so viel Speicher im primären und sekundären Speicher zur Verfügung steht. Linux 2.4 erlaubt es, über den `sysctl`-Eintrag `VM_OVERCOMMIT_MEMORY` diesen Test ein- beziehungsweise auszuschalten. Der Systemruf verifiziert aber auch die aktuellen vom Nutzer vorgegebenen Grenzen für den Speicherverbrauch des Prozesses. Der Systemruf schließt auch ein Überschneiden mit einem bereits eingeblendeten Speicherbereich aus.

Bei einer Erhöhung des `brk`-Zeigers wird Speicher im virtuellen Adressraum des Prozesses alloziert. Der Systemruf *brk* blendet dabei einen anonymen Speicherbereich ein

oder erweitert einen bereits vorhandenen anonymen Speicherbereich. Der Kernel reserviert also erst Speicher, wenn auf ihn schreibend zugegriffen wird. Erfolgt das Lesen vor dem Schreiben, trägt der Kernel die globale leere Speicherseite (`ZERO_PAGE()`) in das Pagedirectory des Prozesses ein.

4.2.4 Funktionen für das Mapping

Die C-Bibliothek stellt in der Headerdatei `sys/mman.h` unter anderem folgende drei Funktionen bereit:

```
#include <sys/mman.h>

extern __ptr_t mmap(__ptr_t addr, size_t len,
                    int prot, int flags, int fd, __off_t off);
extern int munmap(__ptr_t addr, size_t len);
extern int mprotect(__ptr_t addr, size_t len, int prot);
```

`mmap()` blendet ab dem Offset `off` die Datei oder das Gerät, auf das der Dateideskriptor `fd` verweist, als virtuellen Speicherbereich ein. Das Flag `MAP_ANONYMOUS` ist für anonymes Einblenden zu verwenden.

Die Funktion `munmap()` greift auf den Systemruf *munmap* zu. Dieser Systemruf blendet Speicherbereiche wieder aus dem Nutzersegment aus.

Die Bibliotheksfunktion `mprotect()` setzt für einen Speicherbereich im Nutzersegment die Schutzattribute um, wobei die schon bei `do_mmap_pgoff()` erläuterten Makros `PROT_NONE`, `PROT_READ`, `PROT_WRITE` und `PROT_EXEC` Verwendung finden. Die Implementierung dieser Funktion basiert auf dem Systemruf *mprotect*. Dieser Systemruf testet natürlich, ob überhaupt ein Speicherbereich an dieser Stelle eingeblendet ist und ob die neuen Schutzattribute für diesen Bereich gesetzt werden dürfen.

Für die x86-Architektur schließt das Setzen der Attribute `PROT_WRITE` und `PROT_EXEC` das Setzen des `PROT_READ`-Attributs ein. Wird das `PROT_READ`-Attribut gesetzt, ist das `PROT_EXEC`-Attribut implizit gesetzt.

Weitere Funktionen für den virtuellen Speicherbereich unterstützen das Synchronisieren des Arbeitsspeichers mit dem Platteninhalt (`msync()`), das Sperren des Auslagerns von eingeblendeten Speicherbereichen in den sekundären Speicher (`mlock()`) sowie das Verschieben des Speicherbereichs (`mremap()`). Seit der Version 2.0 kann man `mmap()` und Freunde unter Linux uneingeschränkt nutzen.

4.2.5 Das Kernelsegment

Beim Aufruf einer Systemfunktion geht ein Prozess in den Systemmodus über. Linux löst im Normalfall den Softwareinterrupt 128 (0x80) aus. Der Prozessor liest aus der Interruptdeskriptortabelle den Gatedeskriptor aus. In diesem Fall handelt es sich um einen Trapgatedeskriptor, der auf die Assemblerroutine `system_call` aus `arch/i386/`

`kernel/entry.S` zeigt. Der Prozessor springt zu dieser Adresse, wobei der Segmentselektor im Register CS auf das Kernelsegment zeigt. Dann setzt die Assemblerroutine die Segmentselektoren in den Registern DS und ES, so dass bei Speicherzugriffen Daten im Kernelsegment gelesen und geschrieben werden.

Da die Pagetabellen aller Prozesse für das Kernelsegment identisch sind, ist sichergestellt, dass jeder Prozess im Systemmodus dasselbe Kernelsegment vorfindet. Der physische Speicher ist ab der durch `PAGE_OFFSET` festgelegten Startadresse in den virtuellen Kerneladressraum eingeblendet. Zusätzlich können auch noch mit `vmalloc()` beziehungsweise `ioremap()` virtuelle Adressbereiche in den Kerneladressraum eingeblendet werden.

4.2.6 Speicherreservierung im Kernelsegment während des Bootens

Beim Starten des Kernels werden vor der Erzeugung des ersten Prozesses Initialisierungsroutinen für eine Reihe von Kernelkomponenten aufgerufen. Diese müssen unter Umständen zu einem Zeitpunkt Speicher allozieren, an dem die normale Speicherverwaltung des Kerns noch nicht initialisiert ist. Das nutzt vor allem der Konsolentreiber, der für das Debuggen der Speicherverwaltungsinitialisierung von Nutzen ist. Den so allozierten Speicher bezeichnen die Kernelprogrammierer als *Boot Memory*; die Routinen für das Reservieren und Freigeben von Boot Memory sind in `linux/bootmem.h` zu finden. Der Kern gibt nach dem erfolgreichen Starten der Speicherverwaltung den Boot Memory wieder frei. Der Speicher wird seitenweise belegt, wobei nicht genutzter Platz einer vorhergehenden Speicherseite bei aneinander grenzenden Allozierungen genutzt wird.

4.2.7 Dynamische Speicherreservierung im Kernelsegment

Im Systemkern ist es häufig notwendig, Speicher zum Beispiel für temporäre Puffer zu allozieren. Ab Linux 2.2 wird dafür ein neues Allozierungsverfahren eingesetzt. Es wird aufgrund der Benennung der bei diesem Verfahren verwendeten Einheiten als *Slab-Allokation* bezeichnet.

Grundsätzlich kann sowohl mit `kmalloc()` als auch mit `kfree()`+ Speicher dynamisch alloziert und wieder freigegeben werden.

```
void * kmalloc (size_t size, int priority);
void kfree (const void *obj);
```

Die Funktion `kmalloc()` versucht, Speicher mit der Größe `size` zu reservieren.

Mit der Funktion `kfree()` kann der reservierte Speicher wieder freigegeben werden. Die Version 1.0 von Linux ließ nur die Reservierung von Speicher bis zur Größe von 4.072 Byte zu. Nach einer abermaligen Reimplementierung ist nun die Reservierung

von Speicher bis zu 128 Kbyte möglich. Der Kernel reserviert den Speicher aus dem physischen Speicherbereich.

`kmalloc()` löscht den reservierten Speicher nicht. Es kann sein, dass `kmalloc()` den Prozess unterbricht, da im Kernelsegment keine freien Speicherseiten vorhanden sind und der Kernel erst Speicherseiten freigeben oder auslagern muss.

Die Slab-Allokation implementiert `kmalloc()` und `kfree()`. Sie benutzt Kernelspeichercaches, die jeweils Speicherabschnitte einer bestimmten Größe bereitstellen. Diese Kernelspeichercaches können auch direkt eingesetzt werden, wenn sehr viele Objekte der gleichen Größe alloziert und wieder freigegeben werden müssen. So gibt es einen speziellen Cache für Verzeichniseinträge. Die Datei `/proc/slabinfo` gibt die aktiven Caches sowie die Anzahl der allozierten und freien Objekte im Cache an.

```
kmem_cache_t* kmem_cache_create(const char *name,
  size_t size, size_t offset, unsigned long flags,
  void (*ctor)(void *, kmem_cache_t *, unsigned long),
  void (*dtor)(void *, kmem_cache_t *, unsigned long));

int kmem_cache_destroy(kmem_cache_t *cachep);
int kmem_cache_shrink(kmem_cache_t *cachep);

void* kmem_cache_alloc(kmem_cache_t * cachep, int flags);
void  kmem_cache_free(kmem_cache_t * cachep, void *);
```

`kmem_cache_create()` legt einen neuen Cache für Objekte der Größe `size` an. Soll die Funktion ein spezielles Alignment für die Objekte benutzen, sollte der Aufrufer den Parameter `offset` ungleich Null setzen. Der Parameter darf die Größe des Objekts nicht überschreiten. Die Funktionen `ctor()` und `dtor()` werden vom Kernel beim Anlegen beziehungsweise Löschen der Objekte aufgerufen. Wichtig dabei ist, dass das Objekt in der Zwischenzeit beliebig oft alloziert und freigegeben werden kann. Dem Konstruktor können verschiedene Flags übergeben werden. So muss beim Aufruf des Konstruktors das Flag `SLAB_CTOR_CONSTRUCTOR` gesetzt sein, ansonsten wird er als Destruktor aufgerufen. Beim Aufruf mit dem Flag `SLAB_CTOR_ATOMIC` ist dem Konstruktor verboten zu blockieren, und bei `SLAB_CTOR_VERIFY` muss der Konstruktor das übergebende Objekt prüfen.

Mit dem Flag `SLAB_HWCACHE_ALIGN` wird `kmem_cache_create()` mitgeteilt, dass alle Objekte an der Größe der Level-1-Cachelines des Prozessors auszurichten sind. Dies dient der Performance, insbesondere bei sehr kleinen Objekten. Weitere Flags dienen zum Debugging; so kann der Konstruktor bei der Freigabe der Objekte zum Überprüfen des Objekts eingesetzt werden. Rote Zonen zwischen den einzelnen Objekten helfen, das Überschreiten von Grenzen eines allozierten Bereichs zu erkennen. Sucht ein Kernelprogrammierer das Auftreten von nicht initialisiertem Speicher, kann er `SLAB_POISON` benutzen, das den Slab-Speicher mit dem Byte 0xa5 beschreibt. Die Funktion `kmem_cache_destroy()` schließt den Cache und gibt den vom Cache reservierten Speicher vollständig frei. Diese Funktion erlaubt es, Caches in Modulen zu verwenden; sie gibt es erst seit Linux 2.4. Muss der Kernel den Cache verkleinern, ruft er `kmem_cache_shrink()` auf.

`kmem_cache_alloc()` und `kmem_cache_free()` reservieren ein Objekt beziehungsweise geben ein Objekt frei.

Ein Cache verwaltet mehrere Slabs, in denen sich die Objekte befinden. Ein Slab ist ein Vielfaches einer Speicherseite groß. Das Vielfache ist immer eine Zweierpotenz und ist normalerweise nicht größer als 32. Das größte Slab-Objekt kann also 128 Kbyte groß werden. Je nachdem was Speicherplatz-sparender ist, verwaltet der Slab-Allokator die Slab-Managementstrukturen im Slab oder außerhalb des Slabs. Wird beim Allozieren eines Objekts zusätzlich das Flag `SLAB_NO_GROW` gesetzt, wird darauf verzichtet, einen neuen Slab anzulegen, wenn alle anderen Slabs des Caches keine freien Objekte enthalten.

Die Funktion `kmalloc()` benutzt zwei Tabellen von Caches. Die eine Tabelle wird für ISA-DMA-Speicher und die andere für den restlichen Speicher benutzt. In beiden Tabellen gibt es jeweils einen Cache für jede Zweierpotenz ab 64 Byte, 32 bei 4-Kbyte-Speicherseiten, bis 128 Kbyte. Fordert `kmalloc()` einen Speicherbereich an, sucht der Kernel den Cache mit der kleinsten Größe, in den der Speicherbereich hineinpasst. Bei Objekten, die nicht der Länge einer Zweierpotenz entsprechen, verschenkt `kmalloc()` Speicher. Sind sehr viele Objekte mit der gleichen Größe anzulegen, sollte ein eigener Slab-Cache benutzt werden.

Mit dem `flags`-Parameter kann die Allokation des Speichers gesteuert werden. So kann man mit `GFP_DMA` ISA-DMA-Speicher anfordern oder mit `GFP_ATOMIC` verhindern, dass sich der Prozess schlafen legt.

In sehr alten Kernelversionen (0.x) war `kmalloc()` die einzige Möglichkeit, Speicher im Kernel dynamisch zu allozieren. Außerdem war die Größe der zu reservierenden Speicherbereiche auf die Größe einer Speicherseite beschränkt.

Die Funktion `vmalloc()` und das zugehörige `vfree()` schufen Abhilfe. Mit ihnen kann Speicher in Vielfachen einer Speicherseite reserviert werden. Beide Funktionen sind in `mm/vmalloc.c` definiert.

```
void * vmalloc(unsigned long size);
void * vfree(void * addr);
```

Für `size` kann auch eine nicht durch 4.096 teilbare Größe angegeben werden; sie wird dann aufgerundet. Sind kleinere Bereiche als 4.072 Byte zu reservieren, ist die Verwendung von `kmalloc()` sinnvoller. Der maximale Wert für `size` ist durch den zur Verfügung stehenden freien physischen Speicher beschränkt. Weil der von `vmalloc()` reservierte Speicher nicht ausgelagert wird, sollten Kernelprogrammierer nicht allzu großzügig damit umgehen. Da `vmalloc()` die Funktion `__get_free_page()` aufruft, ist es möglich, dass der Prozess blockiert wird, um Speicherseiten auszulagern. Der reservierte Speicher ist nicht initialisiert.

Nach dem Aufrunden von `size` sucht `vmalloc()`, genauer die Hilfsfunktion `__vmalloc()`, eine freie Adresse, an die der zu allozierende Bereich in das Kernelsegment eingeblendet werden kann. Wie schon erläutert, wird im Kernelsegment

der gesamte physische Speicher ab dessen Beginn eingeblendet, so dass die virtuellen Adressen den physischen Adressen plus ein architekturabhängiges Offset entsprechen.

`__vmalloc()` muss nun oberhalb des Endes des physischen Speichers den zu allozierenden Speicher einblenden. Die Suche inach einer freien Adresse beginnt bei der x86-Architektur ab der nächsten auf einer 8-Mbyte-Grenze (`VMALLOC_OFFSET`) hinter dem physischen Speicher liegenden Adresse. Die dortigen Adressen könnten durch vorhergehende `vmalloc`-Aufrufe schon belegt worden sein. Zwischen den einzelnen reservierten Bereichen wird jeweils eine Speicherseite freigelassen, um Zugriffe abzufangen, die über den allozierten Speicherbereich hinausgehen. `__vmalloc()` verändert das Pagedirectory des Kernelsegments entsprechend.

Linux verwaltet die so erzeugten virtuellen Kerneladressbereiche auf einfache Art und Weise mit Hilfe einer linearen Liste. Die zugehörige Datenstruktur `vm_struct` beinhaltet die virtuelle Adresse des Bereichs und dessen Größe, die auch die nicht in die Pagetabelle eingetragene Seite umfasst. Die Komponente `flags` unterscheidet zwischen den von `vmalloc()` beziehungsweise von `ioremap()` genutzten Strukturen.

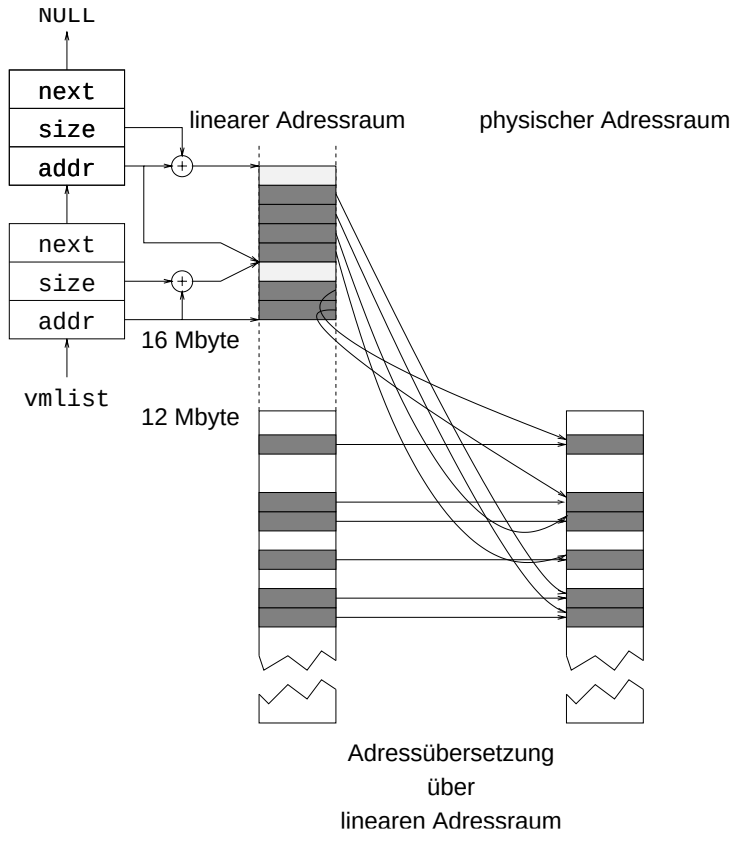


Abbildung 4.3: Funktionsweise von `vmalloc()`

Der Vorteil der `vmalloc()`-Funktion besteht sicherlich darin, dass die Größe des angeforderten Speicherbereichs besser an den Bedarf angepasst wird als bei `kmalloc()`. `kmalloc()` benötigt für die Reservierung von etwas mehr als 64 Kbyte Speicher 128 Kbyte aufeinander folgenden physischen Speicher. Außerdem ist `vmalloc()` nur durch die Größe des freien physischen Speichers beschränkt und nicht durch dessen Segmentierung wie `kmalloc()`. Da `vmalloc()` keine physischen Adressen zurückgibt und die reservierten Speicherbereiche über nicht aufeinander folgende Speicherseiten verstreut sein können, ist diese Funktion für die Reservierung von Speicher für DMA ungeeignet. Linux 2.4 kennt zwar die Funktion `vmalloc_dma()`, die Speicher in ISA-DMA-fähigen Bereichen reserviert, allerdings wird sie im Kernel nicht ein einziges Mal aufgerufen.

PCI-Geräte werden auf Adressen am oberen Ende des physischen Adressraums eingeblendet. Da diese Adressen nicht in den virtuellen Adressraum des Prozesses eingeblendet sind, ist zunächst ein Zugriff auf diesen Speicher nicht möglich. Mit der Funktion `ioremap()` sollten solche physischen Adressen in das Pagedirectory des Kernelsegments eingetragen werden. Der dabei verwendete virtuelle Adressbereich ist der von `vmalloc()` vorgesehene. Mit `iounmap()` wird der virtuelle Adressraum wieder freigegeben. Mit `ioremap_nocache()` wird verhindert, dass der entsprechende Speicherbereich im Prozessorcache zwischengespeichert wird.

4.3 Das Caching der Blockgeräte

Bei der Beurteilung der Leistungsfähigkeit eines Computersystems ist die Zugriffsgeschwindigkeit auf Blockgeräte ein entscheidender Faktor. Linux benutzt ein dynamisches Cachesystem, das den von Kernel und Prozessen ungenutzten Hauptspeicher als Puffer für die Blockgeräte verwendet. Steigt der Bedarf an Hauptspeicher an, wird der Platz für den Puffer wieder verkleinert.

Das Caching der Blockgeräte wird seit Version 2.0 durch das weiter unten erläuterte dateiorientierte Speicherseitencaching ergänzt.

4.3.1 Blockpuffer

Blockgeräte können Anforderungen zum Lesen und Schreiben von Datenblöcken bearbeiten. Dabei kann die Blockgröße für ein Gerät ein Vielfaches von 512 bis zur Größe einer Speicherseite (`PAGE_SIZE`) betragen. Diese Blöcke müssen durch ein Puffersystem im Speicher gehalten werden.

Ein Zugriff auf das Gerät sollte nur in zwei Fällen erfolgen. Ein Block ist zu laden, wenn er noch nicht im Puffer vorhanden ist. Er ist zu schreiben, wenn der Pufferinhalt des Blocks nicht mehr mit dem Inhalt auf dem externen Medium übereinstimmt. Dazu wird nach einer Schreiboperation der betroffene Block im Puffer als „dirty“ gekennzeichnet. Allerdings kann dieses Schreiben verzögert erfolgen, da der gültige Inhalt des Blocks im Puffercache vorhanden ist. Einen Sonderfall bilden Blöcke von Dateien, die mit dem

Flag O_SYNC geöffnet worden sind. Sie werden nach jeder Modifikation ihres Inhalts auf Platte übertragen.

Ursprünglich basierte die Implementierung des Linux-Puffercaches, wenn auch leicht modifiziert, auf dem Konzept, das in „The Design of the UNIX (R) Operating System“ von Maurice J. Bach [Bac86] beschrieben wurde. Mittlerweile rechtfertigen es die Änderungen, von einem eigenen Puffercache-Konzept von Linux zu sprechen.

Wie weiter oben schon festgestellt worden ist, verwaltet der Puffercache einzelne Blockpuffer unterschiedlicher Größe. Dazu wird jedem Block eine buffer_head-Datenstruktur zugeordnet. Der Einfachheit halber wird sie im Folgenden als *Pufferkopf* bezeichnet. Die Definition des Pufferkopfes findet man in linux/fs.h:

```
struct buffer_head {
  /* Erste Cache-Zeile: */
  struct buffer_head * b_next;      /* Hashliste */
  unsigned long b_blocknr;          /* Blocknummer */
  unsigned short b_size;            /* Blockgröße */
  unsigned short b_list;            /* Pufferliste */
  kdev_t b_dev;                     /* Gerät (B_FREE = frei) */
  atomic_t b_count;                 /* Anzahl der Nutzer für
                                     * diesen Block */
  kdev_t b_rdev;                    /* reales Gerät */
  unsigned long b_state;            /* Pufferstatusbitmap */
  unsigned long b_flushtime;        /* Zeit, wenn der Puffer
                                     * geschrieben werden sollte */
  struct buffer_head *b_next_free;  /* LRU bzw. freie Liste */
  struct buffer_head *b_prev_free;
  struct buffer_head *b_this_page;  /* Liste der Puffer einer
                                     * Speicherseite */
  struct buffer_head *b_reqnext;    /* Liste von IO-Anforderungen */
  struct buffer_head **b_pprev;     /* Hash-Listen */
  char * b_data;                    /* Zeiger auf den Datenblock */
  struct page *b_page;              /* Seite, auf der der Block,
                                     * eingebettet ist. */
  void (*b_end_io)(struct buffer_head *bh, int uptodate);
                                    /* Aufgerufen nach dem Ende
                                     * der Ein-/Ausgabe */
  void *b_private;                  /* reserviert für b_end_io */

  unsigned long b_rsector;          /* realer Ort auf der */
                                    /* Festplatte */
  wait_queue_head_t b_wait;

  struct inode *       b_inode;
  struct list_head     b_inode_buffers;
                                    /* Liste von veränderten
                                     * Puffern einer Inode */
};
```

Die Datenstruktur ist so organisiert, dass oft abgefragte Daten sehr nah beieinander liegen und möglichst im Prozessorcache gehalten werden können.

Der Zeiger `b_data` zeigt auf die Blockdaten in einem extra reservierten Bereich des physischen Speichers. Die Größe dieses Bereichs entspricht genau der Blockgröße `b_size`. Zusammen mit diesem Datenbereich bildet der Pufferkopf den Blockpuffer. Der Wert von `b_dev` gibt das Gerät an, auf dem der zu diesem Blockpuffer gehörige Block gespeichert ist, und `b_blocknr` enthält die Nummer dieses Blocks auf dem zum Gerät gehörigen Speichermedium. Da es möglich ist, dass es sich bei dem referenzierten Gerät um ein Pseudogerät handelt, das mehrere Blockgeräte zusammenfasst (mehrere Partitionen einer Festplatte) oder um ein logisches Volume, gibt es noch `b_rdev` und `b_rsector`, die den realen Sektor auf einem realen Gerät referenzieren.

Die Zahl der Prozesse, die den Blockpuffer gerade benutzen, ist in `b_count` zu finden. Die Bitmapvariable `b_state` enthält eine Reihe von Zustandsflags. Der Blockpuffer entspricht dem Inhalt auf der Platte, wenn das Flag `BH_Uptodate` gesetzt ist. Der Blockpuffer muss auf das Medium zurückgeschrieben werden, wenn `BH_Dirty` gesetzt ist. Der Zugriff auf den Blockpuffer ist gesperrt, wenn `BH_Lock` gesetzt ist. In einem solchen Fall müssen sich Prozesse in die Warteschlange `b_wait` einreihen. Das Flag `BH_Req` zeigt an, ob der zum Puffer gehörige Block von einem Gerät angefordert wurde. Mit `BH_Protected` markierte Puffer können nie mehr freigegeben werden. Dieses Feature wird für Ramdisks verwendet. Früher musste der Speicherplatz für Ramdisks statisch alloziert werden. `BH_New` zeigt an, dass der Pufferinhalt neu ist, und noch nicht auf die Festplatte geschrieben ist. Ist der Puffer `BH_Mapped`, so ist dem Puffer ein Block auf dem Gerät zugewiesen.

Für einen als „dirty" markierten Blockpuffer zeigt `b_flushtime` den Zeitpunkt in `jiffies` an, ab dem der Blockpuffer auf das Gerät zurückgeschrieben werden sollte. Wenn der Block als „dirty" markiert wird, wird die `b_flushtime` auf die aktuelle Zeit plus einem Verzögerungsparameter gesetzt. Damit wird der Puffer erst dann auf die Platte zurückgeschrieben, wenn längere Zeit kein Schreibzugriff erfolgte.

Der Handler `b_end_io()` wird aufgerufen, wenn die IO-Operation für diesen Puffer erfolgreich ausgeführt worden ist.

4.3.2 Bdflush und Kupdate

`Bdflush` und `kupdate` sind Kernelthreads, die Puffer zurück auf die Festplatte schreiben. `Kupdate` schreibt alte modifizierte Puffer auf die Platte zurück, inklusive der Superblock- und der Inode-Informationen.

`kupdate` schreibt alle modifizierten Pufferblöcke, die seit einem gewissen Zeitraum nicht mehr benutzt wurden, auf die Platte zurück, ebenso alle Superblock- und Inode-Informationen. Das standardmäßig unter Linux verwendete `kupdate`-Intervall beträgt fünf Sekunden. Die Zeit, die `kupdate` wartet, um einen modifizierten Puffer auf die Platte zu schreiben, beträgt standardmäßig 30 Sekunden.

`Bdflush` schreibt in einer Endlosschleife die durch den `bdflush`-Parameter vorgegebene Anzahl von Blöcken (Standard 64) auf die Festplatte. Ist die Gesamtzahl der modifizierten Blöcke größer als eine Prozentzahl (Standard 30) werden die Puffer weiter auf die Platte geschrieben.

Die Parameter für beide Kernelthreads setzt der Systemruf *bdflush* während des laufenden Betriebs. Tabelle 4.4 erklärt die einzelnen Parameter.

Parameter	Standard-wert	Erläuterung
`nfract`	30	Prozentsatz für veränderte Pufferblöcke, bei dessen Überschreitung der `bdflush`-Prozess aktiviert wird
`ndirty`	64	maximale Anzahl von Pufferblöcken, die bei jeder Aktivierung von `bdflush` geschrieben werden
`interval`	500	Ticks, nach denen `kupdate` wieder gestartet wird (fünf Sekunden)
`age_buffer`	3.000	Ticks, um die das Ausschreiben eines modifizierten Blockpuffers verzögert wird (30 Sekunden)
`nfract_sync`	60	Prozentsatz, bei dem veränderte Pufferblöcke ausgeschrieben werden, ohne auf die Aktivierung des `bdflush`-Kernelthreads zu warten

Tabelle 4.4: Parameter für den `bdflush`-Prozess

Der Vorteil der Kombination von `bdflush` und `kupdate` liegt auf der Hand: Die Anzahl der sich im Puffercache befindlichen modifizierten Blockpuffer wird minimiert.

4.3.3 Die Listenstrukturen des Puffercaches

Linux verwaltet die Blockpuffer in einer Vielzahl von verschiedenen Listen. Freie Blockpuffer werden in doppelt verketteten Ringlisten verwaltet. Die Tabelle `free_list[]` enthält eine solchen Liste für die in Linux unterstützten Blockgrößen. Mögliche Blockgrößen sind 1, 2, 4, 8, 16, und 32 Kbyte sowie 512 Byte. Blöcke, die in der `free_list[]` stehen, sind so markiert, dass im `b_dev`-Feld ihres Pufferkopfes `B_FREE` (0xffff) eingetragen ist.

Blockpuffer, die in Benutzung sind, werden in einer Reihe von speziellen LRU-Listen verwaltet. Mit LRU wird der englische Ausdruck *least recently used* (zuletzt verwendet) abgekürzt. Die einzelnen LRU-Listen sind in der Tabelle `lru_list[]` zusammengefasst. Die Indizes in dieser Tabelle bestimmen die Art der in die einzelnen LRU-Listen eingetragenen Blockpuffer. Tabelle 4.5 erläutert die möglichen Indizes und damit auch die verschiedenen Arten von LRU-Listen. Die Listen mit freien Blöcken als auch die LRU-Listen sind doppelt verkettete Ringlisten, die über die Zeiger `prev_free_list` und `next_free_list` verkettet sind. Ein Blockpuffer wird von der Funktion `refile_buffer()` in die für den Puffer richtige LRU-Liste einsortiert.

LRU-Liste (Index)	Erläuterung
BUF_CLEAN	Blockpuffer, die nicht in den restlichen Listen verwaltet werden — ihr Inhalt stimmt mit zugehörigem Block auf der Festplatte überein.
BUF_LOCKED	Blockpuffer, die gesperrt sind (b_lock != 0)
BUF_DIRTY	Blockpuffer, deren Inhalt nicht mit den zugehörigen Blöcken auf der Festplatte übereinstimmen
BUF_PROTECTED	Blockpuffer in einer RAMDISK

Tabelle 4.5: Die verschiedenen LRU-Listen

Die benutzten Pufferblöcke sind in der Tabelle hash_table[] referenziert. Sie dient dazu, Blockpuffer anhand der Gerätenummer und der Blocknummer aufzufinden. Das dabei angewendete Verfahren ist offenes Hashing. Die Hashlisten sind als doppelt verkettete lineare Listen realisiert. Dazu werden die Zeiger b_next und b_prev des Pufferkopfes benutzt.

4.3.4 Verwendung des Puffercaches

Um einen Block zu lesen, ruft eine Systemroutine die Funktion bread() auf. Ihre Definition steht in der Datei fs/buffer.c:

```
struct buffer_head * bread(kdev_t dev, int block, int size)
```

Zuerst wird überprüft, ob für den Block block zum Gerät dev nicht schon ein Blockpuffer vorhanden ist. Dabei wird auf die Blockpuffer-Hashtabelle zugegriffen. Wird der Puffer gefunden und ist das Flag BH_Uptodate gesetzt, wird bread() mit der Rückgabe des Blockpuffers beendet. Wenn das Flag nicht gesetzt ist, muss der Puffer durch das Lesen des externen Mediums aktualisiert werden, und die Routine kann zurückkehren.

Zum Lesen des Blocks wird die Funktion ll_rw_block() verwendet, die die entsprechende Anforderung für den Gerätetreiber erzeugt. Sie ist in der Datei ll_rw_blk.c im Verzeichnis drivers/block/ implementiert. Nach dem Absetzen der Gerätetreiberanforderung muss der aktuelle Prozess allerdings noch auf die Abarbeitung dieser Anforderung warten. Mit brelse() sollte der von bread() zurückgegebene Speicherblock freigegeben werden, wenn er nicht mehr benötigt wird.

Für das Lesen und Schreiben von Speicherseiten in den Arbeitsspeicher gibt es die Funktion brw_page():

```
int brw_page(int rw, struct page *page,
                kdev_t dev, int b[], int size)
```

Diese Funktion schreibt oder liest abhängig vom Parameter rw die Blöcke mit den Nummern aus dem Vektor b[] aus der Speicherseite page.

Linux stellt die klassischen Systemrufe *sync* und *fsync* zur Verfügung. *sync* schreibt alle modifizierten Pufferblöcke im Puffercache inklusive Inodes und Superblöcke zurück,

wobei nicht auf das Beenden der Schreibanforderungen gewartet wird. Die Funktion stützt sich auf `sync_buffers()`.

```
static int sync_buffers(kdev_t dev, int wait);
```

Der Parameter `dev` kann auf 0 gesetzt werden, um dadurch alle Blockgeräte zu aktualisieren. `wait` steuert, ob die Routine auf die Ausführung der Schreibanforderungen durch die Gerätetreiber warten soll. Wenn nicht, wird der gesamte Puffercache nach modifizierten Blockpuffern durchsucht. Findet `sync_buffers()` solche Blockpuffer, generiert es durch den Aufruf von `ll_rw_block()` die notwendigen Schreibanforderungen an die Gerätetreiber.

Komplizierter ist der Fall, wenn auf die erfolgreiche Ausführung der Schreiboperationen gewartet werden soll. Dazu wird der gesamte Puffercache ingesamt dreimal durchlaufen. Im ersten Durchgang werden für alle modifizierten und nicht gesperrten Blöcke entsprechende Anforderungen generiert. Im zweiten Durchgang wird auf die Beendigung aller blockierten Operationen gewartet. Allerdings kann es jetzt möglich sein, dass ein im ersten Durchgang durch eine Leseoperation blockierter Puffer während des Wartens durch einen anderen Prozess modifiziert wurde. Darum werden auch in diesem Durchgang für modifizierte Puffer Schreibanforderungen generiert. Im dritten Durchgang wird nur noch auf das Beenden aller Operationen gewartet, die Puffer blockieren. Hier zeigt sich ein besonderer Vorteil der asynchronen Ansteuerung der Gerätetreiber: Während noch Blockpuffer auf die Blockgeräte geschrieben werden, kann Linux schon die nächsten modifizierten Blockpuffer suchen.

4.4 Paging unter Linux

Der RAM-Speicher in einem Rechner ist immer begrenzt und im Vergleich zu Festplatten relativ teuer. Besonders Multitasking-Betriebssysteme lasten den Arbeitsspeichers sehr schnell aus. So kam man schon recht früh auf die Idee, momentan nicht genutzte Bereiche aus dem primären Speicher (dem RAM) in sekundäre Speicher (zum Beispiel auf eine Festplatte) auszulagern.

Das traditionelle Auslagerungsverfahren war das Swapping, das ganze Prozesse aus dem Hauptspeicher auf das sekundäre Medium auslagert und wieder einliest. Dieses Verfahren löste nicht das Problem, Prozesse mit größerem Speicherbedarf als dem vorhandenen primären Speicher auszuführen. Außerdem ist das Auslagern und Einlesen eines gesamten Prozesses ineffektiv.

Neue Hardwarearchitekturen (VAX) führten das Konzept des *Demand Paging* ein. Dieses Konzept teilt den Speicher in Speicherseiten. Einzelne Speicherseiten können mit Hilfe einer *Memory Management Unit* auf Anforderung eingelesen und ausgelagert werden. Da alle modernen Prozessorarchitekturen, inklusive der x86-Architektur, die Verwaltung von Speicherseiten unterstützen, wird von Linux das Demand Paging verwendet. Dabei werden Speicherseiten, die ohne Schreiberlaubnis direkt mit `do_mmap()` in den virtuellen Adressbereich eines Prozesses eingeblendet worden sind, nicht ausgelagert,

sondern verworfen. Ihr Inhalt kann wieder aus den eingeblendeten Dateien gelesen werden. Modifizierte Speicherseiten müssen im Gegensatz dazu in Auslagerungsbereiche geschrieben werden.

Speicherseiten des Kernelsegments dürfen aus einem einfachen Grund nicht ausgelagert werden: Routinen und Datenstrukturen, die Speicherseiten aus dem sekundären Speicher zurücklesen, müssen immer im primären Speicher vorhanden sein. Die einfachste Methode, das sicherzustellen, besteht darin, den vom Kernel direkt genutzten Speicher für die Auslagerung zu sperren.

Linux kann zwei Arten von Auslagerungsbereichen auf externen Medien bereitstellen. Bei der ersten wird ein ganzes Blockgerät als Auslagerungsbereich benutzt. In der Regel ist einem solchen Blockgerät eine Partition auf einer Festplatte zugeordnet. Die zweite Art von Auslagerungsbereichen sind Dateien mit einer festen Größe in einem Dateisystem. Dem in Linux üblichen, eher lockeren Umgang mit Begriffen ist es zu verdanken, dass diese Bereiche irreführend als *Swapgeräte* beziehungsweise *Swapdateien* bezeichnet werden. Korrekterweise müsste von *Paginggeräten* und *-dateien* gesprochen werden. Da sich aber nun die beiden nicht ganz korrekten Begriffe eingebürgert haben, werden sie hier übernommen. Mit dem Begriff *Auslagerungsbereich* kann im Folgenden sowohl ein Swapgerät als auch eine Swapdatei bezeichnet werden. Für Swapgerät und Swapdatei sind zwei verschiedene Formate definiert.

Die neuere Version 2 wird seit Linux 2.2 unterstützt. Beim alten Format ist in den ersten 4.096 Byte eine Bitmap enthalten. Gesetzte Bits zeigen an, dass die Speicherseite, deren Nummer im Auslagerungsbereich mit dem Offset des Bits zum Anfang des Bereichs übereinstimmt, für Auslagerungen zur Verfügung steht. Ab dem Byte 4.086 ist noch die Zeichenkette `"SWAP-SPACE"` als Kennung abgelegt. Demnach kann das alte Format nur `4.086 * 8 - 1 = 32.687` Speicherseiten (130.784 Kbyte bei 4-Kbyte-Pages) in einem Swapgerät oder einer Swapdatei unterstützen. Im Verhältnis zu den heute üblichen Plattengrößen ist das nicht allzu viel. Deshalb wurde ein neues Format mit der Kennung `"SWAPSPACE2"` eingeführt. Dieses Format unterstützt Unterversionen, wobei derzeit nur die Unterversion 1 bekannt ist. Dieses Format benutzt die Bitmap nicht. Mit diesem Format können derzeit bis etwa 2 Gigabyte für den Auslagerungsbereich benutzt werden. Diese Grenze hängt mit dem früher maximal möglichen Dateioffset zusammen, Linux 2.4 unterstützt zwar jetzt 64-Bit-Offsets, die Behandlung der Auslagerungsbereiche ist aber noch nicht angepasst worden.

Außerdem gibt es die Möglichkeit, mehrere Swapdateien und -geräte parallel zu benutzen. Linux legt diese Anzahl mit `MAX_SWAPFILES` auf acht fest. Dieser Wert kann höher gesetzt werden, allerdings dürfte es kaum Anwendungen geben, die $2*63 = 126$ Gigabyte Swapspace benötigen.

Die Verwendung eines Swapgeräts ist effektiver als die einer Swapdatei. In einem Swapgerät ist eine Seite immer in aufeinander folgenden Blöcken abgespeichert. In einer Swapdatei können dagegen die einzelnen Blöcke — abhängig von der Fragmentierung des benutzten Dateisystems beim Einrichten der Datei — unterschiedliche Blocknummern besitzen. Diese Blöcke müssen noch über die Inode der Swapdatei ermittelt werden. Bei einem Swapgerät ergibt sich der erste Block schon aus dem Offset der auszulagernden

beziehungsweise einzulesenden Speicherseite. Die restlichen Blöcke folgen auf den ersten Block. Bei einem Swapgerät muss für eine Speicherseite nur eine Lese- beziehungsweise Schreibanforderung abgesetzt werden, bei einer Swapdatei entsprechend dem Quotienten von Speicherseitengröße und Blockgröße mehrere. Im typischen Fall (bei der Verwendung einer Blockgröße von 1.024 Byte) sind das vier separate Anforderungen, die nicht unbedingt hintereinander liegende Bereiche des externen Mediums lesen müssen. Bei einer Festplatte werden dadurch Bewegungen des Schreib-Lese-Kopfes verursacht, die dann auch auf die Lesegeschwindigkeit Einfluss haben. Durch den Systemruf *swapon* wird ein Swapgerät oder eine Swapdatei im Kernel angemeldet:

```
int sys_swapon(const char * specialfile, int swap_flags);
```

Der Parameter `specialfile` ist der Name des Geräts beziehungsweise der Datei. Mit den Flags `swap_flags` kann die Priorität des Auslagerungsbereichs festgelegt werden. Dabei ist das Flag `SWAP_FLAG_PREFER` zu setzen; und die Bits in der `SWAP_FLAG_PRIO_MASK` geben die positive Priorität des Auslagerungsbereichs an. Ist keine Priorität angegeben, wird den Auslagerungsbereichen automatisch eine negative Priorität zugeordnet. Wobei die Priorität je Aufruf von *swapon* absteigt. In der Systemroutine wird für den Auslagerungsbereich ein Eintrag in der Tabelle `swap_info` gefüllt. Er hat den Typ `swap_info_struct`:

```
struct swap_info_struct {
  unsigned int flags;
  kdev_t swap_device;
  spinlock_t sdev_lock;
  struct dentry * swap_file;
  struct vfsmount *swap_vfsmnt;
  unsigned short * swap_map;
  unsigned int lowest_bit;
  unsigned int highest_bit;
  unsigned int cluster_next;
  unsigned int cluster_nr;
  int prio;
  int pages;
  unsigned long max;
  int next;
};
```

Ist in `flags` das Bit `SWP_USED` gesetzt, wird in der Tabelle `swap_info` der Eintrag vom Kernel schon für einen anderen Auslagerungsbereich genutzt. Der Kernel setzt `flags` auf `SWP_WRITEOK`, wenn alle Initialisierungsschritte für den Auslagerungsbereich abgeschlossen sind. Verweist die Struktur auf eine Swapdatei, ist der Verzeichniseintrag-Zeiger `swap_file` gesetzt, andernfalls ist zusätzlich das Swapgerät in `swap_device` eingetragen. Sed weiteren ist mit `swap_vfsmnt` der Mount-Punkt des Dateisystems referenziert. Mit `sdev_lock` werden Zugriffe auf den Auslagerungsbereich und seine Datenstrukturen synchronisiert. Der Zeiger `swap_map` zeigt auf eine mit `vmalloc()` allozierte Tabelle, in der jeder Speicherseite des Auslagerungsbereichs ein *Short* zugeordnet ist. In diesem *Short* wird gezählt, wie viele Prozesse auf diese Speicherseite verweisen. Kann die Speicherseite nicht benutzt werden, ist der Wert in `swap_map` auf

SWAP_MAP_BAD (0x8000) gesetzt. In der Integer-Komponente pages ist die Anzahl der Speicherseiten abgelegt, die in diesen Auslagerungsbereich geschrieben werden dürfen. Die Werte von lowest_bit und highest_bit geben das minimale beziehungsweise maximale Offset einer freien Speicherseite im Auslagerungsbereich an. In max ist der um 1 erhöhte initiale Wert von highest_bit gespeichert, da dieser häufig benötigt wird. In prio ist die dem Auslagerungsbereich zugeordnete Priorität gespeichert.

Neu auszulagernde Speicherseiten werden sequenziell in Gruppen (Clustern) im Auslagerungsbereich gespeichert. Das soll übermäßige Kopfbewegungen der Festplatte beim aufeinander folgenden Auslagern und Einlesen von Speicherseiten verhindern. In der Variablen cluster_nr ist gespeichert, wieviel freie Speicherseiten sich im aktuellen Cluster noch befinden sollen, und cluster_next speichert den Offset der letzten allozierten Speicherseite.

Der Index next der Struktur swap_info_struct bildet eine Liste der Auslagerungsbereiche entsprechend ihrer Priorität.

Mit dem Systemruf *swapoff* kann versucht werden, eine Swapdatei oder ein Swapgerät wieder beim Kernel abzumelden. Dazu muss aber im Hauptspeicher oder in den anderen Auslagerungsbereichen genügend Speicher vorhanden sein, um die Speicherseiten aufzunehmen, die sich im abzumeldenden Auslagerungsbereich befinden:

```
int sys_swapoff(const char * specialfile);
```

4.4.1 Speicherseitenverwaltung und -cache

Im Kernel wird für jede Speicherseite in einer Tabelle, auf die der Zeiger mem_map zeigt, eine Datenstruktur struct page beziehungsweise mem_map_t verwaltet. Die Daten sind so organisiert, dass zusammengehörende Daten in einer Cachezeile (16 Bytes) gespeichert werden.

```
typedef struct page
{
  struct list_head list;
  struct address_space *mapping;
  unsigned long index;
  struct page *next_hash;
  atomic_t count;
  unsigned long flags;
  struct list_head lru;
  unsigned long age;
  wait_queue_head_t wait;
  struct page **pprev_hash;
  struct buffer_head * buffers;
  void *virtual; /* nicht NULL wenn mit kmap eingeblendet */
  struct zone_struct *zone;
} mem_map_t;
```

Die Struktur `list` dient zur Verwaltung der Speicherseiten in doppelt verketteten Ringlisten. Der Zeiger `mapping` verweist auf eine Struktur, die Informationen zu allen Speicherseiten innerhalb einer Datei oder Blockgeräts hält, unter anderem die Inode. Das Feld `index` gibt den Offset in die Datei oder dem Blockgerät an, von der die Speicherseite eingeblendet ist.

Ist die Speicherseite einmal in den Arbeitsspeicher geladen, referenziert sie der Kernel auch in der Hashtabelle `page_hash_table` mit `next_hash` und `pprev_hash`. Bei Leseanforderungen an eine Speicherseite wird zuerst die Hashtabelle befragt, ob die Speicherseite existiert. Kann sie dort gefunden werden, braucht sie nicht mit Hilfe des Filesystems gelesen zu werden. Damit ist ein dateiorientiertes Caching gegeben, das beliebige Filesysteme (insbesondere NFS) unterstützt. Auch normale Filesystem-Leseoperationen wie `read()` greifen über den Speicherseitencache zu.

Zurück zur Speicherseitenstruktur: Die Anzahl der Nutzer einer Speicherseite ist in `count` gespeichert. Der Zeiger `buffers` referenziert den Blockpuffer, wenn die Speicherseite Bestandteil eines solchen ist. In die Warteschlange `wait` tragen sich die Tasks ein, die auf das Entsperren der Speicherseite warten. Tabelle 4.6 erklärt die Bedeutung der einzelnen in der Variable `flags` gespeicherten Flags.

Das Feld `zone` verweist auf die Speicherzone, in der die Speicherseite liegt. Der Kernel 2.4 kennt drei Zonen: `ZONE_DMA`, `ZONE_NORMAL` und `ZONE_HIGHMEM`. Die DMA-Zone enthält Speicherseiten, die für *Direct Memory Access* (DMA) der Hardware geeignet sind. Die normale Zone enthält Speicherseiten, die im Kernel direkt adressierbar sind. Die HIGHMEM-Zone ist gefüllt, wenn es Speicherseiten gibt, die der Kernel nur temporär in das Kernelsegment einblenden kann. Das ist dann der Fall, wenn die Hardware mehr physischen Speicher besitzt als Adressraum im Kernelsegment, und der HIGHMEM-Support eingeschaltet ist. Für HIGHMEM-Speicherseiten ist der Zeiger `virtual` initialisiert, wenn die Speicherseite mit `kmap()` in das Kernelsegment eingeblendet ist.

Flag	Erläuterung
PG_locked	Speicherseite ist gesperrt. Das Flag wird gesetzt, wenn die Speicherseite gerade eingelesen oder ausgegeben wird.
PG_error	Dieses Flag zeigt einen Fehler bei der Ein- oder Ausgabe an.
PG_referenced	Die Speicherseite ist durch einen Pufferkopf referenziert.
PG_uptodate	Die Speicherseite ist erfolgreich geladen worden.
PG_dirty	Die Speicherseite ist modifiziert und entspricht nicht mehr dem Festplatteninhalt.
PG_decr_after	Der Zähler nr_async_pages für die momentan asynchron ein- beziehungsweise auszulagernden Seiten wird nach dem Lesen bzw. Schreiben der Seite dekrementiert.
PG_active	Die Speicherseite ist in Benutzung, dass heißt, auf sie wurde zugegriffen oder mehr als zwei Nutzer teilen sich die Seite.
PG_inactive_dirty	Die Speicherseite ist beschrieben oder gesperrt worden, seit längerem ist aber kein Zugriff mehr erfolgt.
PG_slab	Die Speicherseite wird für die Slab-Allozierung benutzt.
PG_swap_cache	Die Speicherseite ist im Swap-Cache eingetragen.
PG_skip	Die Speicherseite markiert Lücken im physischen Speicher (z. B. zwischen einzelnen Speicherbänken in der Sparc-Architektur) — in 2.4.2 nicht genutzt.
PG_inactive_clean	Die Speicherseite entspricht dem Inhalt auf Festplatte und seit längerer Zeit ist kein Zugriff mehr auf sie erfolgt.
PG_highmem	Die Speicherseite liegt im High Memory; ein Speicher, der im Kernelsegment nicht direkt adressierbar ist.
PG_arch_1	Dieses Flag ist ein architekturspezifisches Speicherseitenflag. Es wird für die Sparc64-Architektur und von der IA-64 für die Behandlung des Datencaches benutzt.
PG_reserved	Auf die Speicherseite sollte nicht zugegriffen werden, da dort entweder keine Speicher vorhanden ist oder Hardware eingeblendet ist.

Tabelle 4.6: Die Speicherseitenflags

4.4.2 Speicherseitenreservierung

Der Kernel ruft zum Reservieren von physischen Speicherseiten die Funktion `__get_free_pages()` auf. Sie ist in der Datei `mm/page_alloc.c` definiert:

```
unsigned long __get_free_pages(int gfp_mask,
                unsigned long order)
```

Der Parameter `gfp_mask` steuert die Abarbeitung der Funktion. Die zulässigen Werte sind in Tabelle 4.7 zusammengefasst. Diese Prioritäten sind auch für die weiter oben erläuterten Funktionen `kmalloc()` und `kmem_cache_alloc()` zu verwenden, wobei für die letztere Funktion eine Reihe von Prioritäten mit dem Präfix `SLAB_` anstatt `GFP_` definiert sind.

`GFP_ATOMIC` ist für den Aufruf von `__get_free_pages` aus Interruptbehandlungsroutinen heraus gedacht, es erfolgt keine Unterbrechung der Task, um auf das Bereitstellen von Speicherseiten oder gar dem Auslagern von Speicherseiten zu warten. Ein Aufruf `GFP_BUFFER` kann zwar unterbrochen werden, aber nicht für das Auslagern von Speicherseiten. Bei allen anderen Prioritäten kann der Kernel die aktuelle Task unterbrechen und auf das Auslagern von Speicherseiten gewartet werden.

Der zweite Parameter `order` bestimmt die Ordnung des zu reservierenden Speicherblocks von aufeinander folgenden physischen Seiten. Ein Block mit der Ordnung x ist 2^x Speicherseiten groß. Im Linux-Kernel sind nur Ordnungen zugelassen, die kleiner als das Makro `MAX_ORDER` (Standardwert 10) sind. In der x86-Architektur mit 4 Kilobyte Größe für eine Speicherseite kann minimal $2^0 = 4$ und maximal $2^9 = 2.048$ Kilobyte reserviert werden.

Wenn `__get_free_pages()` einen entsprechenden Block reservieren konnte, gibt es die Adresse dieses Blocks zurück, wobei die jetzige Implementierung sicherstellt, dass der Block an einer Adresse beginnt, die durch seine Größe in Byte teilbar ist.

Der Kernel unterscheidet drei Speicherzonen, die erste Speicherzone `ZONE_DMA` enthält Speicherseiten, die für *Direct Memory Access* von Peripheriegeräten geeignet sind. Speicherseiten, die nicht in das Kernelsegment eingeblendet werden können, werden in der Zone `ZONE_HIGHMEM` verwaltet. Alle anderen Speicherseiten sind der „normalen" Zone `ZONE_NORMAL` zugeordnet.

Jeder möglichen Priorität für `__get_free_pages()` ist eine Liste von Zonen zugeordnet. `__get_free_pages()` versucht den Speicherbereich entsprechend der Reihenfolge der Zonen in der Liste zu befriedigen. Damit wird sichergestellt, dass der DMA-Speicher erst für normale Speicheranforderungen benutzt wird, wenn kein Speicher in den anderen Speicherzonen verfügbar ist. Sind zu wenig freie Speicherseiten vorhanden, werden Kernelthreads aktiviert, die die freien Speicherseiten wieder auffüllen. Unter Umständen wird auch auf das Auslagern von Speicherseiten gewartet. Kann keine Speicherseite gefunden werden, kehrt die Funktion mit 0 zurück.

Die Zonen und Zonenlisten werden in der Struktur `pglist_data` verwaltet. Der Kernel greift auf diese Struktur über die globale Variable `contig_page_data` zu, wenn NUMA nicht unterstützt wird.

Priorität	Erläuterung
GFP_ATOMIC	Beim Allozieren einer solchen Speicherseite darf der Kernel-Thread nicht unterbrochen werden.
GFP_BUFFER	Beim Allozieren der Speicherseite kann der laufende Prozess unterbrochen werden, allerdings nicht für das Auslagern von Speicherseiten.
GFP_KERNEL	Für das Allozieren der Speicherseiten dürfen Speicherseiten ausgelagert und der Prozess unterbrochen werden.
GFP_NFS	hat dieselbe Priorität wie GFP_KERNEL.
GFP_RPC	Auch hier wird dieselbe Priorität wie GFP_KERNEL verwendet.
GFP_USER	entspricht derzeit praktisch GFP_KERNEL, sollte aber so angewendet werden, dass die Allozierung niedrigere Priorität hat.
GFP_HIGHUSER	Gleiche Priorität wie GFP_USER, die Speicherseite kann aber aus dem High Memory kommen. Das heißt, sie ist nicht im Kernelsegment direkt adressierbar.
GFP_KSWAPD	Diese Priorität wird vom Kernelthread GFP_KSWAPD benutzt. Allerdings nicht um Speicherseiten zu allozieren, sondern bei Speicherknappheit die Speicherseiten auszulagern.
GFP_DMA	Flag, das eine Speicherseite aus dem architekturabhängigen DMA-Bereich anfordert. Der ISA-DMA-Controller kann z. B. nur einen geringen Adressbereich bedienen.
GFP_HIGHMEM	Flag, das eine Speicherseite im High Memory anfordert. Diese Speicherseiten werden nicht permanent in das Kernelsegment eingeblendet.

Tabelle 4.7: Prioritäten für die Funktion __get_free_pages()

Für jede Zone verwaltet der Kernel die Tabelle free_area[], um die verschieden großen Speicherbereiche zu unterstützen. In einem Tabelleneintrag ist eine doppelt verkettete Ringliste von Speicherseiten der entsprechenden Größe eingetragen. Der Zeiger map des Eintrags referenziert eine Bitmap. In dieser Bitmap ist jeweils ein Bit für zwei aufeinander folgende Speicherblöcke der jeweiligen Ordnung reserviert. Das Bit ist gesetzt, wenn einer der beiden Speicherblöcke frei ist und der andere, wenn auch nur teilweise, reserviert ist. Dieses Verfahren wird von Donald E. Knuth in [Knu98] als *Buddy System* bezeichnet. Abbildung 4.4 stellt eine mögliche Belegung von Maps für die ersten drei Ordnungen dar.

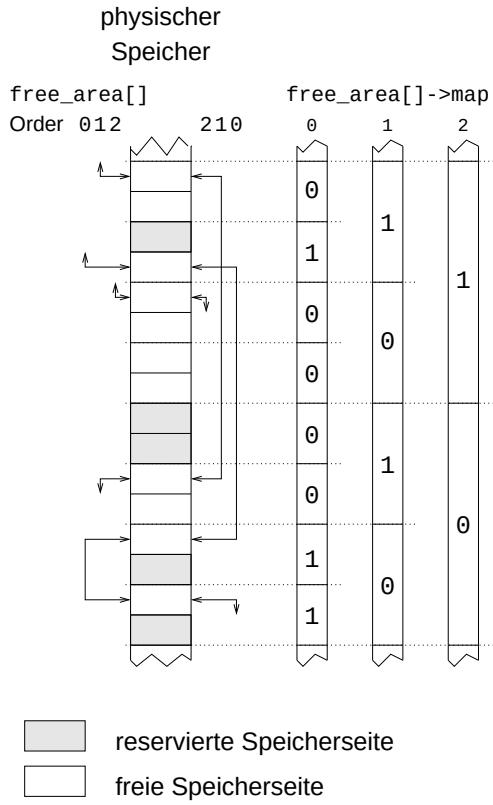


Abbildung 4.4: Beispiel der Belegung von `free_area`-Maps

Die Linux-Implementierung stellt sicher, dass nirgends zwei aufeinander folgende Speicherblöcke frei sind, die zu einem größeren Speicherblock zusammengefasst werden können. Das kann unter Umständen dazu führen, dass keine Speicherblöcke für die unteren Ordnungen frei sind. Bei einer entsprechenden Anforderung müssen Speicherblöcke mit höheren Ordnungen geteilt werden. Die Funktion `expand()` aus `mm/page_alloc.c` aktualisiert die `free_area`-Datenstrukturen entsprechend.

Die Anzahl der Speicherblöcke für die einzelnen Zonen und Speichergrößen, kann der Linux-Nutzer durch Betätigen einer spezifischen Tastenkombination auf die Konsole ausgeben. Für Suse-Linux sind dabei die Tasten Umschaltung (*Shift*) und Rollen gleichzeitig zu drücken. Hier ein Auszug aus der Ausgabe, der die Freispeicherlisten für die DMA und die normale Zone zusammenfasst:

```
1*4kB 1*8kB 1*16kB 0*32kB 0*64kB 0*128kB 0*256kB 1*512kB
0*1024kB 0*2048kB = 540kB)
23*4kB 6*8kB 1*16kB 2*32kB 1*64kB 0*128kB 1*256kB 1*512kB
0*1024kB 0*2048kB = 1052kB)
```

Die Funktion `get_free_pages()` ruft über die Inline-Funktion `alloc_pages()` die Funktion `__alloc_pages()` auf. Sie weckt den Kernelthread kswapd auf, wenn zu

wenig freie und inaktive Speicherseiten vorhanden sind. Der Kernelthread sucht nach inaktiven Speicherseiten und stellt sie in die entsprechende Liste. Sind genug inaktive Speicherseiten vorhanden, wird der Kernelthread bdflush aktiviert, der Speicherseiten auslagert. Dann wird in den einzelnen Zonen nach einem freien Speicherbereich gesucht. Sind in einer Zone nur noch wenige freie Speicherseiten aktiviert und der Kernelthread kreclaimd ist nicht aktiv, wird er geweckt. Der kreclaimd-Kernelthread trägt inaktive, saubere Speicherseite wieder in die `free_area`-Maps ein.

4.4.3 Optimierung der Speicherseitenverwaltung durch Kernelthreads

Der Kernel könnte erst mit dem Suchen und Auslagern von Speicherseiten beginnen, wenn er beim Allozieren von Speicherseiten keine freien Speicherseiten mehr finden kann. Das wurde auch in früheren Linux-Versionen getan. Dieser Ansatz ist jedoch problematisch, da die CPU-Last gerade dann ansteigt, wenn viel Speicher benötigt wird.

Linux 2.4 optimiert die Speicherseitenverwaltung. Dazu werden globale Listen für die aktiven Speicherseiten und „schmutzige" (*dirty*) Speicherseiten verwaltet. Jede Speicherzone verfügt über eine eigene Liste von inaktiven sauberen Speicherseiten.

Auf inaktive Speicherseiten ist seit längerem nicht mehr zugegriffen worden; saubere inaktive Speicherseiten stimmen mit dem Inhalt auf der Platte überein, schmutzige inaktive Speicherseiten müssen entweder in den Auslagerungsbereich oder auf Platte geschrieben werden. Werden freie Speicherseiten benötigt, sind die inaktiven Speicherseiten Kandidaten für die Speicherseiten, die verworfen werden und neu genutzt werden.

Kernelthreads sorgen dafür, dass die `free_area`-Maps und die Listen mit den inaktiven Speicherseiten gefüllt sind. Der Kernelthread kswapd wird, solange der Speicher nicht besonders knapp ist, einmal pro Sekunde aktiviert. Mit der Funktion `do_try_to_free_pages()` füllt kswapd die Liste der inaktiven sauberen Speicherseiten. Der Kernelthread `kswapd` wird sonst immer dann reserviert, wenn zu wenig inaktive Speicherseiten im System vorhanden sind.

Der Kernelthread kreclaimd trägt inaktive saubere Speicherseiten in die `free_area`-Maps ein. Er wird nur aktiviert, wenn beim Allozieren von Speicherseiten festgestellt wird, dass in einer Speicherzone zu wenig Speicherseiten frei sind.

Sind beim Speicherseiten-Allozieren zwar genug inaktive Speicherseiten aktiv, aber zu wenig freie Speicherseiten, wird der Kernelthread bdflush aufgeweckt. Der Kernelthread schreibt Blöcke auf die Festplatte, so dass beim nächsten Aufruf von `do_try_to_free_pages()` Speicherseiten in die Listen der inaktiven sauberen Speicherseiten eingetragen werden können.

4.4.4 Seitenfehler und das Zurückladen einer Speicherseite

Kann die MMU des Prozessors nicht auf eine Speicherseite zugreifen, generiert er eine Seitenfehler-Ausnahmeunterbrechung (*Page Fault*). Bei einem x86-Prozessor wird ein Fehlercode auf den Stack geschrieben, und die lineare Adresse, für die die Unterbrechung verursacht wurde, wird im Register `CR2` gespeichert. Bei anderen Architekturen gibt es ähnliche Mechanismen.

Linux ruft in diesem Fall die architekturspezifische Funktion `do_page_fault()` auf:

```
void do_page_fault(struct pt_regs *regs,
                                unsigned long error_code);
```

Dieser Routine werden die Werte der Register beim Auftreten der Unterbrechung und die Fehlernummer übergeben. Die Routine sucht nach dem virtuellen Speicherbereich des gerade aktiven Prozesses, in dem die Adresse, an der die MMU keine Speicherseite zuordnen konnte, liegt.

Befindet sich die Adresse nicht innerhalb eines virtuellen Speicherbereichs, überprüft `do_page_fault()`, ob das Flag `VM_GROWSDOWN` für den nächsten virtuellen Speicherbereich gesetzt ist. Ein solcher Bereich stellt Speicher für den Stack bereit und kann nach unten wachsen. `do_page_fault()` nimmt die notwendige Erweiterung vor.

Konnte die Funktion einen virtuellen Speicherbereich finden, wird die Funktion `handle_mm_fault()` aufgerufen. Kann `handle_mm_fault()` keine Speicherseite zuordnen, wird das Signal `SIGBUS` generiert. Ist nicht genug Speicher vorhanden, wird der aktuelle Prozess mit SIGKILL`SIGKILL` beendet. War `handle_mm_fault()` erfolgreich, kehrt `do_page_fault()` zurück.

Konnte kein virtueller Speicherbereich zugeordnet werden, und handelt es sich um eine Adresse im Nutzersegment, wird das Signal `SIGSEGV` generiert. Mit diesem Signal *Segmentation Violation* wird sich sicherlich jeder ernsthafte Unix-Programmierer auseinander gesetzt haben.

Trat der Fehler innerhalb des Systemmodus auf, wird überprüft, ob es sich um den F00F-Bug bei einigen Pentium-Prozessoren handelt. Er wird wie ein ungültiger Maschinencodefehler behandelt.

Bei einem Kernelfehler wird eine Ausnahmetabelle durchsucht und der Befehlszeiger des Registersatz wird entsprechend manipuliert.

Wenn all diese Prüfungen nicht erfolgreich waren, handelt es sich definitiv um einen Fehler im Kernel, und entsprechende Informationen werden auf die Konsole ausgegeben.

Die erwähnte Funktion `handle_mm_fault()` ist architekturunabhängig. Sie ermittelt den zur Speicherseite gehörenden Pagetabelleneintrag. Ist die Speicherseite nicht präsent und ist das entsprechende Bit nicht gesetzt, wird `do_swap_page()` aufgerufen, falls ein Swapindex im Pagetabelleneintrag zu finden ist. Ansonsten wird `do_no_page()`

aufgerufen. Ist die Speicherseite vorhanden, aber schreibgeschützt, wird `do_wp_page()` gerufen.

```
static int do_no_page(struct mm_struct * mm,
        struct vm_area_struct * vma,
        unsigned long address, int write_access,
        pte_t *page_table);

static int do_swap_page(struct mm_struct * mm,
        struct vm_area_struct * vma,
        unsigned long address, pte_t * page_table,
        swp_entry_t entry, int write_access);

static int do_wp_page(struct mm_struct *mm,
        struct vm_area_struct * vma,
        unsigned long address, pte_t *page_table,
        pte_t pte);
```

Die Funtion `do_no_page()` überprüft, ob es sich um eine anonym eingeblendete Speicherseite handelt. Wenn ja, wird `do_anonymous_page()` aufgerufen, die die entprechenden Copy-On-Write-Operationen übernimmt. Das heißt, bei lesendem Zugriff wird die `ZERO_PAGE` eingeblendet, bei schreibendem Zugriff eine vollkommen neue Speicherseite.

Ansonsten wird die entsprechende Behandlungsroutine aus dem virtuellen Speicherbereich aufgerufen, und der Pagetabelleneintrag wird je nach schreibendem oder lesendem Zugriff entsprechend gesetzt.

`do_swap_page()` prüft, ob die Speicherseite schon im Swapcache vorhanden ist. Ist das nicht der Fall wird die Speicherseite mit `swapin_readahead()` eingelesen. Diese Funktion liest zusätzlich zu aktuellen Seite noch möglicherweise im Auslagerungsbereich vorhandene auf die aktuelle Seite folgende Seiten asynchron ein.

Die Funktion `do_wp_page()` überprüft, ob eine schreibgeschützte Speicherseite überhaupt unter der angegebenen Adresse zu finden ist. Ist sie nur einmal referenziert, wird der Schreibschutz einfach aufgehoben. Wird sie mehrmals referenziert, wird eine Kopie der Speicherseite erzeugt und diese nicht schreibgeschützt in die entsprechende Pagetabelle des Prozesses eingetragen, der den Fehler ausgelöst hat.

5 Interprozesskommunikation

Is simplicity best
Or simply the easiest
Martin L. Gore

Es gibt viele Anwendungen, in denen es notwendig ist, dass Prozesse kooperieren. Dies ist zum Beispiel immer dann der Fall, wenn Prozesse sich eine Ressource (z. B. einen Drucker) teilen müssen. Dabei ist auszuschließen, dass mehrere Prozesse gleichzeitig auf die Ressource zugreifen, also Daten an den Drucker senden. Diese Situation wird Wettbewerbsbedingung (*race condition*) genannt. Kommunikation zwischen den Prozessen muss solche Situationen verhindern. Der Ausschluss von Wettbewerbsbedingungen ist aber nur eine Einsatzmöglichkeit von Interprozesskommunikation, die von uns einfach nur als Austausch von Informationen zwischen Prozessen eines oder mehrerer Rechner verstanden wird.

Es gibt viele verschiedene Arten der Interprozesskommunikation. Sie unterscheiden sich unter anderem durch ihre Effektivität. Die Übermittlung einer kleineren natürlichen Zahl könnte mit Hilfe von zwei Prozessen realisiert werden, wobei der eine die entsprechende Anzahl an Kindprozessen erzeugt und der andere diese zählt.

Dieses nicht ganz ernst gemeinte Beispiel ist sicherlich sehr aufwendig, langsam und indiskutabel. Gemeinsam genutzter Speicher, *shared memory*, kann dieses Problem effektiver und schneller lösen.

In der letzten Zeit sind *Threads* populärer geworden. Threads ermöglichen es, Code parallel im selben Adressraum auszuführen. Der Wechsel zwischen Threads ist weniger aufwendig als zwischen Prozessen, da die Pagedirectories nicht ausgetauscht werden müssen. In Multiprozessor-Umgebungen ermöglichen es Threads zum Beispiel, zwei Methoden einer Objektinstanz parallel auszuführen, dies ist insbesondere für CORBA-Applikationen interessant. Allerdings werden die Vorteile mit vielen Wettbewerbsbedingungen erkauft, da die Threads eines Programms auf den gesamten Adressraum Zugriff haben. Auch hier muss zwischen den einzelnen Threads kommuniziert werden, um diese Wettbewerbsbedingungen zu vermeiden.

Diese Kommunikation wird durch Programmierschnittstellen möglich, die als POSIX-Threads bekannt sind. Wir beschäftigen uns hier nicht mit dieser Programmierschnittstelle, da sie in Linux noch nicht im Kern realisiert sind. Linus Torvalds vertrat bei seiner Implementation von Threads die Auffassung, dass Threads Prozesse sind, die sich nur den Adressraum mit anderen Prozessen teilen. Eine darüber hinausgehende Unterstützung von Threads ist im Kern noch nicht zu finden. Daraus resultieren Inkompatibilitäten mit dem POSIX-Standard, die vor allem die Signalbehandlung betreffen. Linus Torvalds hat aber nach jahrelangen Debatten — die Linux-Threads-Implementierung stammt aus dem

Jahr 1996 —, einem Design für die POSIX-kompatible Unterstützung von Threads zugestimmt. Die Kernelentwickler werden dieses Design in den 2.5-Entwicklungskerneln umsetzen.

Unter Linux kann man zahlreiche Formen von Interprozesskommunikation verwenden. Sie unterstützen Ressourcenteilung, Synchronisation, verbindungslosen beziehungsweise verbindungsorientierten Datenaustausch oder Kombinationen davon. Synchronisationsmechanismen dienen dem Ausschluss der oben erwähnten Wettbewerbsbedingungen.

Verbindungsloser und verbindungsorientierter Datenaustausch unterscheiden sich von den ersten beiden Varianten durch andere semantische Modelle. In diesen Modellen sendet ein Prozess Nachrichten an einen Prozess oder eine bestimmte Gruppe von Prozessen.

Beim verbindungsorientierten Datenaustausch müssen die Kommunikationspartner erst eine Verbindung aufbauen, bevor die Kommunikation erfolgen kann. Bei einem verbindungslosen Datenaustausch versendet ein Prozess nur Datenpakete, die mit einer Zieladresse oder mit einem Nachrichtentyp versehen sein können, und überlässt es der Infrastruktur, die Pakete zuzustellen. Dem Leser sind diese Modelle aus dem täglichen Leben bestens vertraut. Beim Telefonieren benutzt er das Modell vom verbindungsorientierten Datenaustausch und beim Versenden von Briefen das verbindungslose Modell.

Es ist möglich, basierend auf dem einen Konzept (z. B. verbindungslosem Nachrichtenaustausch) andere Konzepte (möglicherweise Semaphore) zu implementieren. Linux realisiert alle Möglichkeiten der Interprozesskommunikation zwischen Prozessen desselben Systems mit geteilten Ressourcen, Kerndatenstrukturen und dem Synchronisationsmechanismus „gegenseitiger Ausschluss“, *Mutex*. Die Kernelprogrammierer verwenden den Begriff *Spin-Lock*, der mehr die Art und Weise und weniger den Effekt des Synchronisationsmechanismus beschreibt.

Linux-Prozesse können Speicher mit System-V-Shared-Memory teilen. Das Dateisystem ist von vornherein so implementiert, dass Dateien und Geräte gleichzeitig von mehreren Prozessen genutzt werden können. Um beim Zugriff auf Dateien Wettbewerbsbedingungen zu vermeiden, können verschiedene Dateisperrmechanismen verwendet werden. System-V-Semaphore können als Synchronisationsmechanismus zwischen Prozessen eines Rechners genutzt werden.

Es gibt neuerdings auch eine POSIX-Spezifikation dieser oben genannten IPC-Mechanismen. Linux 2.4 ermöglicht mit dem Shared-Memory-Filesystems die Implementation dieser Mechanismen. Die GNU-C-Library Version 2.2 unterstützt POSIX Semaphore und Shared Memory. Da die POSIX Messagequeues auf Basis von Semaphoren und Shared Memory realisiert werden können, stellt der Kernel der POSIX-Kompatibilität nichts mehr in den Weg. Interessierten Lesern sei die Lektüre des Buchs von W. Richard Stevens [Ste98] zum Thema Interprozesskommunikation empfohlen, in dem die neuen POSIX-Funktionen ausführlich dargestellt werden. Eine andere Darstellung enthält das Buch [Gal95] von Bill O. Gallmeister.

Die einfachste Variante eines verbindungslosen Datenaustauschs sind Signale. Signale kann man als sehr kurze Nachrichten auffassen, die an einen bestimmten Prozess oder

eine Prozessgruppe gesendet werden (siehe Kapitel 3). In dieser Kategorie stehen unter Linux noch System-V-Messagequeues (Nachrichtenwarteschlangen) und die Datagramm-Sockets der INET-Adressfamilie zur Verfügung. Die Datagramm-Sockets nutzen den UDP-Teil des TCP/IP-Codes. Sie können auch im Netzwerk verwendet werden (siehe Kapitel 8).

Für den verbindungsorientierten Datenaustausch stehen *Pipes*, *Named Pipes*, die in der deutschen Literatur auch als FIFOs[1] bezeichnet werden, Unix-Domain-Sockets und Stream-Sockets der INET-Adressfamilie zur Verfügung. Die Stream-Sockets sind die Schnittstelle zum TCP-Teil des Netzwerks und werden unter anderem für die Realisierung von Diensten wie FTP und TELNET verwendet. Auch sie werden im Kapitel 8 erläutert. Die Verwendung des Socket-Programmierinterfaces stellt streng betrachtet nicht in jedem Fall eine Interprozesskommunikation dar. Der gegenüberliegende Kommunikationsendpunkt in einem Netz muss kein Prozess sein. Es kann sich zum Beispiel um ein Programm eines Betriebssystems handeln, das keine Prozesse kennt.

In [Bac86] wird der Systemruf *ptrace* als eine Variante der Interprozesskommunikation aufgeführt. Mit ihm kann ein Prozess die Abarbeitung eines anderen Prozesses bis hin zur Einzelschrittausführung steuern und sowohl Speicher als auch Register des Prozesses modifizieren. Er wird vor allem für Debugging-Zwecke benutzt. Dieses Kapitel diskutiert die Implementation.

Tabelle 5.1 fasst noch einmal die von Linux unterstützten Arten der Interprozesskommunikation zusammen. Da NFS auf Datagramm-Sockets basiert, ist die Möglichkeit, Dateien über ein NFS-Dateisystem zu teilen, nicht aufgeführt. Der Systemruf *mmap* ist seit der Version 2.0 des Kerns vollständig implementiert. Es kann also Shared Memory über anonymes Einblenden wie bei BSD-Systemen realisiert werden. Kernel 2.4 erlaubt auch das Einblenden des `/dev/zero`-Gerätes als gleichzeitig benutzter Speicher ohne Probleme. Das System V Transport Library Interface wird nicht unterstützt.

5.1 Synchronisation im Kern

Da der Kern die Systemressourcen verwaltet, muss der Zugriff von Prozessen auf diese Ressourcen synchronisiert werden. Ein Prozess wird, solange er einen Systemruf ausführt, nicht durch den Scheduler unterbrochen. Dies geschieht nur, wenn er blockiert oder selber `schedule()` aufruft, um anderen Prozessen die Ausführung zu ermöglichen. Bei der Kernelprogrammierung sollte darauf geachtet werden, dass Funktionen wie `__get_free_pages()` und `down()` blockieren können. Außerdem können Prozesse im Kern durch Interruptbehandlungsroutinen unterbrochen werden. So kann es auch zu Wettbewerbsbedingungen kommen, wenn der Prozess keine Funktionen ausführt, die blockieren können.

In einem Multiprozessorsystem ist die Situation zusätzlich dadurch erschwert, dass mehrere Prozesse gleichzeitig auf verschiedenen Prozessoren ausgeführt werden können. Wettbewerbsbedingungen treten damit auch zwischen laufenden Prozessen auf.

1 FIFO ist ein Anglizismus und kürzt First-In/First-Out ab, was das Verhalten einer Pipe sehr gut beschreibt.

	im Kern	zwischen Prozessen	im Netz
Ressourcen-teilung	Datenstrukturen, Puffer	SysV-Shared-Memory, Dateien Anonymes `mmap` `/dev/zero mmap`	
Synchronisations-methode	Warteschlangen, Semaphore	SysV-Semaphore, File Locking, Lock-Datei	
verbindungsloser Datenaustausch	Signale	Signale, SysV-Messagequeues UNIX-Domain-Sockets im Datagramm-Modus	Datagramm-Sockets (UDP)
verbindungs-orientierter Datenaustausch		Pipes, Named Pipes, UNIX-Domain-Sockets im Stream-Modus	Stream-Sockets (TCP)

Tabelle 5.1: Von Linux *unterstützte Arten der Interprozesskommunikation*

Ursprünglich wurden die Wettbewerbsbedingungen im Kernel durch Löschen des Interruptflags beim Eintritt in den kritischen Abschnitt und Zurücksetzen des Flags beim Austritt ausgeschlossen. Bei gelöschtem Interruptflag werden vom Prozessor bis auf den nichtmaskierbaren Interrupt (NMI), der bei der PC-Architektur zur Anzeige von RAM-Fehlern dient, keine Hardwareinterrupts zugelassen. Der NMI sollte im normalen Betrieb nicht auftreten. Diese Methode hat den Vorteil, dass sie sehr einfach ist, aber den Nachteil, dass eine zu freizügige Verwendung das System verlangsamt. Da bei Multiprozessoren Wettbewerbsbedingungen zwischen Prozessen der unterschiedlichen Prozessoren auftreten können, hilft diese Methode nicht weiter. Daten, die von Interruptbehandlungsroutinen ignoriert werden, müssen nicht durch das Sperren von Interrupts geschützt werden.

Der auch von anderen Betriebssystemen genutzte Basismechanismus der Synchronisation in Multiprozessorsystemen sind *Spin-Locks*. Sie realisieren den gegenseitigen Ausschluss von Prozessen im Kernel. Der kritische Abschnitt kann nur von dem Prozess ausgeführt werden, der sich gerade im Besitz des Spin-Locks befindet. Im englischen Sprachraum ist dieses Konzept auch unter dem Begriff *Mutex* bekannt. Die Implementation ist sehr stark von von der darunter liegenden Rechnerarchitektur abhängig. Für ein x86-Multiprozessorsystem ist ein Spin-Lock durch den C-Datentyp `spinlock_t` definiert:

```
typedef struct { volatile unsigned int lock; } spinlock_t;
```

Im ungesetzten Zustand hat die Variable `lock` den Wert 1. Der Prozess, der den Spin-Lock sperren möchte, versucht `lock` auf 0 zu setzen, falls dies noch nicht der Fall ist. Dies ist mit einem x86-Maschinenbefehl möglich, wobei die Aktivitäten der anderen Prozessoren während der Abarbeitung dieses Befehls gesperrt werden. Das Testen und Setzen der `lock`-Variable ist damit atomar. Kann der Spin-Lock nicht gesetzt werden, wartet der Prozessor in einer Schleife darauf, dass die `lock`-Variable wieder freigegeben wird. Dabei werden die anderen Prozessoren aber nicht blockiert. Ist die `lock`-Variable wieder frei, wird erneut versucht den Spin-Lock zu sperren. Das Entsperren, das Zurücksetzen der `lock`-Variable, erfolgt auch atomar.

Beim Warten auf die Freigabe der `lock`-Variable handelt es sich um *Busy Waiting*. Ein Wechsel des Prozesses ist nicht möglich, da dieser für den Zugriff auf die globale Prozesstabelle wieder eine Sperre benötigt. Spin-Locks sind also der atomare Synchronisationsmechanismus im Linux-Kernel. Zusätzlich haben die Spin-Locks den entscheidenden Vorteil, dass sie in Interruptbehandlungsroutinen benutzt werden können. Spin-Locks sollten daher nur dann eingesetzt werden, wenn die Dauer der Sperrung voraussichtlich nur sehr kurz ist und den Aufwand des Blockierens des Prozesses nicht lohnt, oder die Sperren für den Prozesswechsel selbst notwendig sind. Spin-Locks verhindern das Unterbrechen durch Interrupts nicht. Single-Prozessorsysteme benötigen keine Spin Locks, deshalb sind die entsprechenden Operationen standardmäßig leer definiert

Eine Alternative zu den Spin-Locks sind die *Read-Write-Locks*. Hier wird unterschieden, ob ein Prozess eine Ressource nur lesen oder aber schreiben will. Prozesse, die nur lesend auf die Ressource zugreifen, schließen sich gegenseitig nicht aus. Ein schreibender Prozess schließt aber alle anderen Prozesse aus. Ein Read-Write-Lock wird im Linux-Kernel für x86-Multiprossorsysteme durch die Datenstruktur `rwlock_t` wie folgt repräsentiert.

```
typedef struct {
  volatile unsigned int lock;
} rwlock_t;
```

Bei x86-Prozessoren hat die `lock`-Variable im ungesperrten Zustand den Wert `RW_LOCK_BIAS` (0x01000000) gesetzt. Eine Schreibsperre versucht durch Subtraktion mit `RW_LOCK_BIAS` diesen Wert auf 0 zu setzen, und wartet in einer Schleife bis sie erfolgreich ist. Ein Lesesperre versucht 1 zu subtrahieren, ohne dass das Ergebnis negativ wird. Das Warten erfolgt auch hier in einer Busy Loop. Das Setzen und Sperren von Read-Write-Locks ist aufwendiger als von Spin Locks. Sie sollten dann zum Einsatz kommen, wenn die Häufigkeit von Lesezugriffen wesentlich höher ist als die von Schreibzugriffen.

Ingo Molnar und David S. Miller haben zusätzlich für Linux 2.4 *Big-Reader-Locks* implementiert. Sie haben die selbe Semantik wie Read-Write-Locks, setzt aber Lesesperren nur pro Prozessor. Das verhindert, dass für eine Lesesperre die prozessorinternen Caches der Prozessoren abgeglichen werden müssen. Diese Spin-Locks benötigen mehr Speicher als Read-Write-Locks können aber Read-Write-Locks mit einem hohen Anteil an Lesesperren beschleunigen. Sie werden derzeit für das globale Interrupthandling und vom Netzwerkcode genutzt.

Spin- und Read-Write-Locks sind durch handoptimierte Assemblerroutinen realisiert, die vor allem die Anzahl der Operationen bei einem erfolgreichen Setzen der Sperre minimieren. Die Initialisierung von Spin Locks ist mit dem Makro `SPIN_LOCK_UNLOCKED` möglich, für Read Write Locks ist `RW_LOCK_UNLOCKED` zu verwenden. Das einfache Setzen und Entsperren von Spin Locks ist mit `spin_lock()` beziehungsweise mit `spin_unlock()` möglich. Die Makros `spin_lock_irq()` und `spin_unlock_irq()` sperren und erlauben zusätzlich Interrupts für den aktuellen Prozessor. Diese Makros sind problematisch, falls die Interrupts vor ihrer Verwendung gesperrt waren, da durch `spin_unlock_irq()` die Interrupts wieder erlaubt werden. Hier sind `spin_lock_irqsave()` und `spin_unlock_irqrestore()` anzuwenden. Sie sichern vor dem Sperren der Interrupts den Interruptstatus des lokalen Prozessors und setzen ihn beim Entsperren wieder zurück.

Das Sperren der Interrupts ist in Interruptbehandlungsroutinen für die jeweilige CPU notwendig. Werden die Interrupts nicht gesperrt, kann es sein, dass dieselbe Interruptbehandlungsroutine durch einen weiteren Interrupt aufgerufen wird. Diese Routine würde dann auf demselben Spin Lock blockieren, der bei der vorherigen Behandlung des Interrupts gesperrt worden ist. Da das Blockieren in einer Schleife läuft, wird der Spin Lock nie mehr freigegeben.

Auch Softwareinterrupts können sich selbst unterbrechen, da sie nach der Behandlung von Hardwareinterrupts aufgerufen werden. Um das zu verhindern, muss für den Spin-Lock die Ausführung von weiteren Softwareinterrupts auf der aktuellen CPU verhindert werden. Der Kernel definiert dafür `spin_lock_bh()` und `spin_unlock_bh()`.

Die Makros `read_lock()` und `write_lock()` setzen eine Lese- beziehungsweise eine Schreibsperre. Das Entsperren erfolgt mit `read_unlock()` oder entsprechend mit `write_unlock()`. Die Varianten für die Interruptbehandlung werden analog zu den Spin-Lock-Makros in `include/linux/spinlock.h` bereitgestellt. Die Varianten für die Big-Reader-Locks haben das Prefix `br_` und die Locks werden über einen Index selektiert und müssen in `include/linux/brlock.h` zur Compilezeit als Werte einer Enumeration deklariert werden.

Im Kern kann es häufig vorkommen, dass auf bestimmte Ereignisse, zum Beispiel das Schreiben eines Blocks auf die Festplatte, gewartet werden muss. Der aktuelle Prozess sollte blockieren, um anderen Prozessen die Ausführung zu ermöglichen. Die Alternative dazu wäre *Busy Waiting*, bei dem der Prozess so lange eine Schleife durchläuft, bis das Ereignis eintrifft. Allerdings könnte die entsprechende Rechenzeit auch durch andere Prozesse genutzt werden.

Das Blockieren von Prozessen und das Warten auf ein bestimmtes Ereignis wird im Linux-Kernel über Warteschlangen realisiert. Ein Prozess kann sich auf eine Warteschlange setzen und ist dabei so lange unterbrochen, bis die Prozesse in der Warteschlange durch eine Interruptbehandlungsroutine oder einen anderen Prozess wieder geweckt werden.

Für Waitqueues gibt es die Datentypen `wait_queue_head_t` und `wait_queue_t`.

```
struct __wait_queue_head {
    wq_lock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;

struct __wait_queue {
   unsigned int flags;
   struct task_struct * task;
   struct list_head task_list;
};
typedef struct __wait_queue wait_queue_t;
```

Die Warteschlange ist eine doppelt verkettete Ringliste von Zeigern in die Prozesstabelle und einem eigenen Spin-Lock. Linux 2.4 stellt generische doppelt verkettete Ringlisten in dem Header `include/linux/list.h` zur Verfügung. Diese überaus elegante Lösung ermöglicht es, eine Datenstruktur in mehreren Ringlisten zu verwalten, in dem pro Ringliste ein Feld mit dem Typ `struct list_head` in die Datenstruktur aufgenommen wird. Der Typ `wq_lock_t` kann als Spin-Lock oder als Read-Write-Lock definiert werden. Derzeit ist er auf Spin-Lock gesetzt. In Linux 2.2 gab es noch eine globale Sperre für alle Wartelisten, dies ist jetzt fein granulierter.

Will der Kernelprogrammierer Waitqueues verwenden, nimmt er eine Feld mit dem Typ `wait_queue_head_t` in die Datenstruktur mit auf, und initialisiert sie mit mit der Funktion `init_waitqueue_head()`. Nach der Initialisierung ist die Waitqueue leer. Der Kernel trägt einen Prozess, von den Kernelprogrammierern auch Task genannt, in die Waitqueue mit zwei Schritten ein. Im ersten Schritt deklariert und initialisiert das Makro `DECLARE_WAITQUEUE()` eine Datenstruktur `wait_queue_t` mit dem Zeiger auf die Taskstruktur. Im zweiten Schritt wird die Datenstruktur mit `add_wait_queue()` in die Datenstruktur eingetragen. Ob Einträge in der Liste stehen, lässt sich mit `waitqueue_active()` ermitteln. Mit `remove_wait_queue()` kann die Task wieder aus der Taskliste gelöscht werden. Beim Modifizieren der Warteschlange sind die Interrupts gesperrt, so dass auch Interruptroutinen auf die Warteschlangen zugreifen können.

Mit diesen Funktionen können Prozesse aber noch nicht blockiert werden. `sleep_on()` versetzt den Prozess in den Zustand `TASK_UNINTERRUPTIBLE`. Der Prozess kann in diesem Zustand nicht durch Signale unterbrochen werden. Die Funktion `interruptible_sleep_on()` hält den Prozess im Zustand `TASK_INTERRUPTIBLE` an, so dass Signale zum Wecken des Prozesses führen können. Das Blockieren des Prozesses für einen gewissen Zeitraum kann man durch `sleep_on_timeout()` und `interruptible_sleep_on_timeout()` erreichen.

Als Beispiel sei hier die Implementation von `sleep_on()` in Pseudocode dargestellt:

```
__pseudo__ sleep_on(struct wait_queue **p)
{
```

```
  struct wait_queue wait;

  current->state = TASK_UNINTERRUPTIBLE;
  wait.task = current;

  add_wait_queue(p, &wait);
  schedule();
  remove_wait_queue(p, &wait);
}
```

Der Prozess ist also selber dafür verantwortlich, sich in die Warteschlange einzutragen und wieder daraus zu löschen. Die richtige `sleep_on`-Funktionen verwendet Makros, um diese im Kern häufig verwendeten Routinen zu beschleunigen. Es ist möglich, unterbrechbare und nicht unterbrechbare Prozesse in die selbe Warteschlange einzutragen.

Schlafenden Prozessen, die noch dazu nicht durch Signale unterbrochen werden können, sind fast wie im Märchen Dornröschen zum ewigen Schlaf verdammt. Allerdings gibt es Makros, mit denen man Prozesse wachküssen kann.

Das Makro `wake_up()` weckt einen Prozess in der Warteschlange, `wake_up_nr()` maximal eine bestimmte Anzahl und `wake_up_all()` alle Prozesse in der Warteschlange. Man kann das Aufwecken mit `wake_up_interruptible()`, `wake_up_interruptible_nr()` und `wake_up_interruptible_all()` auch auf Prozesse beschränken, die sich unterbrechbar in Warteschlangen schlafen gelegt haben. Es gibt auch sogenannte synchrone Varianten, zum Beispiel `wake_up_sync()`, die sich nur dadurch unterscheiden, dass sie nicht zum nächstmöglichen Zeitpunkt zur Rescheduling führen, sondern erst, wenn der Scheduler aus einem anderen Grund aktiviert wird.

Warteschlangen werden verwendet, um Kernel-Semaphore zu implementieren. Semaphore sind Zählervariablen, die in jedem Fall inkrementiert werden, aber nur dann dekrementiert werden können, wenn ihr Wert größer Null ist. Im anderen Fall wird der dekrementierende Prozess blockiert. Er wird dabei in eine Warteliste für die Semaphore eingetragen. Semaphore können als Mutexe zum gegenseitigen Ausschluss verwendet werden, wenn man Situationen mit einem Zähler von 0 als gesperrt interpretiert. Das Inkrementieren des Zählers ist dann eine Freigabe des Zählers und das Dekrementieren des Zählers ein Sperren des Mutexes. Allerdings muss man darauf achten, dass Dekrementieren und Inkrementieren genau paarweise erfolgt. Linux 2.4 wählt eine etwas komplexere Implementation als der naive Ansatz:

```
struct semaphore {
  atomic_t count;
  int sleepers;
  wait_queue_head_t * wait;
};
```

Die Variable `count` ist vom Type `atomic_t`. Auf sie kann nur mit atomaren Operationen zugegriffen werden. Atomare Operationen sehen aus Kernelprogrammierersicht so aus, dass sie keine Wettbewerbsbedingungen produzieren. Die Multiprozessorhardware stellt diese Operationen bereit, wobei sie komplexe Cachekonsistenzprotokolle nutzt.

Die `up()`-Funktion inkrementiert `count` und weckt alle schlafenden Prozesse, wenn der Wert von `count` kleiner gleich 0 ist. Würde `count` den richtigen Wert des Semaphors repräsentieren, müsste `up()` einen schlafenden Prozesse wecken, wenn `count` gleich 1 ist. Für das Inkrementieren und den Test auf 1 gibt es aber keine atomare Operation. Deshalb ist `count` so korrigiert, dass er kleiner 0 ist, wenn Prozesse auf die Semaphore warten und der richtige Wert eigentlich 0 sein müsste.

Das erlaubt der Funktion `down()` `count` einfach zu dekrementieren und wenn der Wert kleiner 0 ist, sich in die Warteschlange der Semaphore schlafen zu legen. Dabei modifiziert `down()` `sleepers` so, dass — wenn alle Prozesse in der Semaphore-Warteschlange schlafen oder Code außerhalb der Semaphorroutinen abarbeiten — die Summe von `sleepers` und `count` gleich dem richtigen Wert des Semaphors ist. Das Ganze ist so implementiert, dass `sleepers` möglichst 1 und `count` möglichst -1 ist. Im Ergebnis muss bei mehreren aufeinanderfolgenden `up()`-Aufrufen nur der erste Aufruf einen Prozess in der Warteschlange wecken. Der erste aufgewachte Prozess weckt dann den nächsten Prozess, der versucht, `count` wieder kleiner 0 zu setzen und sich schlafen zu legen. War der zweite `up`-Aufruf schneller, gelingt das nicht und der nächste blockierte Prozess wird geweckt, solange bis keine schlafenden Prozesse mehr da sind oder `count` wieder kleiner 0 ist. Für `down()` gibt es zwei weitere Varianten: `down_interruptible()` und `down_trylock()`. Bei der ersten wird der Prozess unterbrechbar schlafen gelegt und bei der zweiten blockiert der Prozess nicht, wenn die Semaphore nicht dekrementiert werden kann.

5.2 Kommunikation über Dateien

Die Kommunikation über Dateien ist die älteste Variante, zwischen Programmen Daten auszutauschen. Programm A schreibt Daten in eine Datei, und Programm B liest diese Daten wieder aus. In einem System, wo zu jedem Zeitpunkt nur ein Programm abgearbeitet werden kann, ist das völlig unproblematisch.

In einem Multitaskingsystem können jedoch beide Programme als Prozesse zumindest quasiparallel abgearbeitet werden. Wettbewerbsbedingungen führen dabei meist zu Inkonsistenzen der Daten in der Datei, die daraus resultieren, dass ein Prozess einen Datenbereich liest, bevor der andere dessen Modifikation abgeschlossen hat oder beide Prozesse gleichzeitig denselben Speicherbereich modifizieren.

Es werden also Sperrmechanismen benötigt. Die einfachste Methode ist natürlich das Sperren der ganzen Datei. Hier bietet Linux wie andere Unix-Derivate eine Reihe von Möglichkeiten. Allgemeiner und effizienter ist die Sperrung von Dateibereichen. Es wird zwischen absoluter (*mandatory*) und kooperativer (*advisory*) Dateizugriffssperrung unterschieden[2]. Bei kooperativer Zugriffssperrung ist nach dem Setzen einer Sperre das Lesen und Schreiben der Datei weiterhin möglich.

Allerdings schließen sich Sperren entsprechend der durch ihre Typen bestimmten Semantik aus. Bei absoluter Sperrung werden Lese- bzw. Schreiboperationen im gesperrten

2 Die deutsche Ausgabe von W. Richard Stevens „Programmieren von Unix-Netzen“ [Ste92b] verwendet die Begriffe „absoluter und relativer Dateizugriffsschutz“.

Bereich blockiert. Bei kooperativer Sperrung müssen also alle auf die Datei zugreifenden Prozesse vor Lese- und Schreiboperationen die notwendigen Sperren setzen und wieder freigeben. Hält sich ein Prozess nicht an diese Regel, sind Inkonsistenzen möglich. Absolute Zugriffssperrung schützt jedoch genausowenig vor dem Fehlverhalten von Prozessen. Prozesse können, falls sie Schreibzugriffsrechte auf die Datei besitzen, durch das Schreiben auf ungesperrte Bereiche Inkonsistenzen erzeugen. Die Probleme, die bei fehlerhaften Programmen auftreten, wenn absolute Sperrung eingesetzt wird, sind sehr kritisch, weil die gesperrten Dateien nicht modifiziert werden können, solange der entsprechende Prozess noch läuft. Seit der Version 2.0 unterstützt Linux absolutes Sperren. Der entsprechende Kernelkonfigurationsparameter ist aber standardmäßig ausgeschaltet. Aus den zuvor genannten Gründen und der Tatsache, dass POSIX 1003.1 absolutes Sperren nicht fordert, ist das durchaus akzeptabel.

Wird durch den generierten Linux-Kernel absolutes Sperren unterstützt, muss für jede Datei, die absolutes Sperren unterstützen soll, das Gruppen-Ausführungsbit ausgeschaltet und das SGID-Bit gesetzt werden. Es ist nicht möglich, gleichzeitig absolute Sperren für eine Datei gesetzt zu haben und diese mit `mmap()` und dem Flag `MAP_SHARED` einzublenden.

5.2.1 Das Sperren ganzer Dateien

Zum Sperren ganzer Dateien gibt es zwei Methoden:

1. Zusätzlich zu der zu sperrenden Datei gibt es eine Hilfsdatei, die, wenn sie angelegt ist, den Zugriff verbietet. Im Folgenden wird sie analog zur englischen Bezeichnung *Lock File* als Sperrdatei bezeichnet. In W. Richard Stevens „Programmieren von Unix-Netzen“ [Ste92b] und der Neuauflage [Ste98] sind folgende Verfahren aufgeführt:

   - Es wird ausgenutzt, dass der Systemruf *link* fehlschlägt, wenn der einzurichtende Verweis auf die Datei schon vorhanden ist. Es wird eine Datei mit der Prozessnummer als Name eingerichtet und anschließend versucht, einen Link auf den Namen der Sperrdatei einzurichten, der nur dann erfolgreich ist, wenn der Link noch nicht existiert. Der Link mit der Prozessnummer als Name kann danach gelöscht werden. Bei Fehlschlägen kann der Prozess (allerdings nur für eine gewisse Zeit) mittels der Bibliotheksfunktion `sleep()` pausieren und dann erneut versuchen, den Link anzulegen.
   - Hier wird die Eigenschaft des Systemrufs *creat* genutzt, mit einem Fehlercode abzubrechen, wenn der aufrufende Prozess nicht die entsprechenden Zugriffsrechte besitzt. Beim Erzeugen der Sperrdatei werden alle Schreibzugriffsbits gelöscht. Diese Variante ist allerdings wiederum mit aktivem Warten verbunden und kann nicht für Prozesse, die mit den Rechten des Superusers laufen, verwendet werden.
   - Die unter Linux empfehlenswerte Variante ist die Benutzung der Kombination der `O_CREAT`- und `O_EXCL`-Flags beim Systemruf *open*. Die Sperrdatei kann nur dann geöffnet werden, wenn sie noch nicht existiert. Im anderen Fall erhält man eine Fehlermeldung.

- Man kann die Sperrdatei auch durch `open()` mit `O_CREAT | O_WRONLY | O_TRUNC` und einem Modus 0 ohne Berechtigungen erzeugen. Wenn die Datei existiert, kommt es zu einer Fehlermeldung. Allerdings funktioniert dieser Trick nicht als Superuser.

Der Nachteil aller vier Varianten ist jedoch, dass der Prozess nach dem Fehlschlagen den Versuch wiederholen muss, eine Sperrdatei einzurichten. Meistens warten die Prozesse mit Hilfe von `sleep()` eine Sekunde und versuchen es erneut. Es kann aber passieren, dass der Prozess, der die Sperrdatei angelegt hat, durch ein `SIGKILL`-Signal beendet wird, so dass die Sperrdatei nicht mehr gelöscht wird. Sie muss nun explizit gelöscht werden. Viele Programme, zum Beispiel der Mail-Reader `elm`, beschränken daher die Anzahl der Versuche, eine Sperrdatei anzulegen, und brechen beim Überschreiten dieser Anzahl dann mit einer Fehlermeldung ab, um den Nutzer auf eine solche Situation aufmerksam zu machen.

2. Die andere Methode ist das Sperren der gesamten Datei mit Hilfe des Systemrufs *fcntl*. Dieser ist auch zum Sperren von Dateibereichen geeignet; das wird im nächsten Abschnitt erläutert. Dies ist die für das Sperren ganzer Dateien empfehlenswerte Variante. Der aus BSD 4.3 stammende Systemruf `flock()` zum Sperren ganzer Dateien ist seit Version 2.0 als eigener Systemruf implementiert. `flock()` unterstützt nur kooperatives Sperren und basiert im Kernel auf denselben Datenstrukturen wie das Sperren mit `fcntl()`. Da `flock()` nicht vom POSIX-Standard definiert wird, sollten Programmierer auf dessen Anwendung verzichten.

5.2.2 Sperren von Dateibereichen

Das Sperren von Dateibereichen wird im Englischen üblicherweise als *Record Locking* bezeichnet. Man müsste dies eigentlich mit Datensatzsperrung übersetzen. Diese Bezeichnung ist aber für Unix-Systeme wenig sinnvoll, da sie Dateien nicht in Datensätzen organisieren.

Unter Linux ist das Sperren von Dateibereichen mit dem Systemruf *fcntl* möglich. Im Kernel wird jetzt auch die 64-Bit-Variante für größere Offsets unterstützt.

```
int sys_fcntl(unsigned int fd, unsigned int cmd,
                          unsigned long arg);
int sys_fcntl64(unsigned int fd, unsigned int cmd,
                          unsigned long arg);
```

Mit `fd` wird ein Dateideskriptor übergeben. Für Sperrungen sind die Kommandos `F_GETLK`, `F_SETLK` und `F_SETLKW` interessant. Wird eines dieser Kommandos benutzt, muss für `arg` ein Zeiger auf eine `flock`-Struktur angegeben sein. Das Kommando `F_GETLK` testet, ob die mit `flock` angegebene Sperrung möglich wäre; wenn nicht, wird die verhindernde Sperre zurückgegeben. `F_SETLK` setzt die Sperre. Ist das nicht möglich, kehrt die Funktion zurück. `F_SETLKW` blockiert, wenn die Sperre nicht gesetzt werden kann. Die letzteren beiden Kommandos können eine Sperre wieder freigeben, wenn der Typ der Sperrung `l_type` auf `F_UNLCK` gesetzt ist. Die 64-Bit-Varianten der Makros sind mit `64` markiert.

```
struct flock {
   short l_type;   /* F_RDLCK, F_WRLCK, F_UNLCK          */
   short l_whence; /* SEEK_SET, SEEK_CUR, SEEK_END       */
   off_t l_start;  /* Offset relativ zu l_whence         */
   off_t l_len;    /* Länge des zu sperrenden Bereichs   */
   pid_t l_pid;    /* wird bei F_GETLK zurückgegeben     */
};
```

Mit dem Typ `F_RDLCK` wird eine Lesesperre des Dateibereichs gesetzt, mit dem Typ `F_WRLCK` eine Schreibsperre. Tabelle 5.2 zeigt, wie die Sperren sich gegenseitig ausschließen. Der Zugriffsmodus der teilweise zu sperrenden Datei muss dem Prozess lesenden beziehungsweise schreibenden Zugriff erlauben. Die 64-Bit-Variante verwendet anstatt dem Typ `off_t` den Typ `loff_t`.

Eine Linux-Besonderheit war, dass für `l_type` auch `F_SHLCK` und `F_EXLCK` möglich waren. Sie wurden von einer älteren Implementierung der erwähnten Funktion `flock()` benutzt. Die genannten Sperrtypen wurden unter Linux auf `F_RDLCK` bzw. `F_WRLCK` abgebildet, mit dem Unterschied, dass die zu sperrende Datei zum Lesen und Schreiben geöffnet sein muss. Wenn also ein *shared lock* als Lesesperre und ein *exclusive lock* als Schreibsperre interpretiert wird, haben wir dieselbe Semantik wie bei `F_RDLCK` und `F_WRLCK` (siehe Tabelle 5.2). Allerdings unterscheidet sich die Semantik zwischen den `fcntl`- und den `flock`-Sperren darin, dass `flock`-Sperren keinen Prozessen zugeordnet sind. Aus diesem Grund ist diese Adhoc-Implementierung fehlerhaft.

bestehende Sperren	Setzen einer Lesesperre	Setzen einer Schreibsperre
keine	möglich	möglich
mehr als eine Lesesperre	möglich	nicht erlaubt
eine Schreibsperre	nicht erlaubt	nicht erlaubt

Tabelle 5.2: Semantik der `fcntl`-Sperren

Die neuen `flock`-Sperren werden zwar im Kern in denselben Datenstrukturen wie die `fcntl()`Sperren verwaltet, sind aber entsprechend markiert, so dass die Sperren unterschiedlicher Art nicht gemischt werden können. Bei einem Setzen einer Sperre auf eine Datei, in der schon Sperren der anderen Art gesetzt sind, wird der Fehler `EBUSY` zurückgegeben. Für jede der beiden Sperrtypen gibt es unterschiedliche Behandlungsroutinen.

Mit `F_UNLCK` können Sperren wieder aufgehoben werden. Die Anfangsposition wird mit `l_whence` und `l_start` angegeben. Für den Parameter `l_whence` können die von der Funktion `lseek()` bekannten Parameter `SEEK_SET` für den Anfang der Datei, `SEEK_CUR` für die aktuelle Position in der Datei und `SEEK_END` für das Ende der Datei verwendet werden. Zu diesen Werten wird dann noch `l_start` addiert. Linux rechnet `SEEK_END` in das aktuelle Ende der Datei um, so dass die Sperre nicht relativ zum Ende

der Datei gesetzt ist. Es ist zum Beispiel nicht möglich, mit einer einzigen Sperre unabhängig von Schreiboperationen am Ende der Datei immer die letzten zwei Byte zu sperren.

Damit zeigt Linux dasselbe Verhalten wie SVR4, das sich darin von BSD unterscheidet. Der Parameter `l_len` bestimmt die Länge des zu sperrenden Bereichs. Ein `l_len` mit dem Wert von 0 ist so zu interpretieren, dass sich der Bereich bis zum aktuellen und jedem zukünftigen Ende der Datei erstreckt. Dies ist das durch die POSIX-Spezifikation abgesegnete Verhalten.

Wird bei `F_GETLK` eine schon gesetzte Sperre gefunden, die das Sperren des angegebenen Bereichs ausschließt, wird die Prozessnummer des Prozesses, der diese Sperre gesetzt hat, in `l_pid` zurückgegeben. Die Implementierung dieser Funktionalität stützt sich dabei auf die doppelt verkettete Liste `file_lock_table` (mit Einträgen `flock`-ähnlicher `file_lock`-Datenstrukturen).

```
struct file_lock {
  struct file_lock *fl_next;
  struct list_head fl_link;
  struct list_head fl_block;
  fl_owner_t fl_owner;
  unsigned int fl_pid;
  wait_queue_head_t fl_wait;
  struct file *fl_file;
  unsigned char fl_flags;
  unsigned char fl_type;
  loff_t fl_start;
  loff_t fl_end;

  void (*fl_notify)(struct file_lock *);
  void (*fl_insert)(struct file_lock *);
  void (*fl_remove)(struct file_lock *);

  struct fasync_struct * fl_fasync;

  union {
          struct nfs_lock_info  nfs_fl;
  } fl_u;
};
extern struct list_head file_lock_list;
```

Der Zeiger `fl_next` wird zum Aufbau einer linearen Liste verwendet, die alle Sperren auf eine Datei (`inode->i_flock`) verkettet.

Die Komponente `fl_owner` speichert die Dateideskriptoren des Prozesses, der die Sperre gesetzt hat, `fl_pid` den entsprechenden Prozessidentifier. Beide Informationen werden für das Kommando `F_GETLK` verwendet. Die gesperrte Datei wird über `fl_file` identifiziert.

Dieser Parameter dient dazu, die `fcntl`-Sperren (`FL_POSIX`), die neuen `flock`-Sperren (`FL_FLOCK`), die alten `flock`-Sperren (`FL_BROKEN`), absolute Sperren

(FL_ACCESS), lockd-Sperren (FL_LOCKD) für NFS und FL_LEASE für das Lease-Konzept. Netzwerks-Clients können mit dem Lease-Konzept die Konsistenz der Daten im eigenen Dateicache und auf dem Server sicherzustellen. SAMBA und das experimentelle SODA-Dateisystem nutzen dieses Feature.

Mit fl_type ist die Art der Sperre angegeben. Die Parameter fl_start und fl_end geben den gesperrten Bereich der Datei an. Sie sind in absoluten Offsets angegeben. Daraus resultiert auch die POSIX-konforme Behandlung von SEEK_END.

Die weiteren Parameter unterstützen Callbacks für Sperroperationen und werden für den NFS-lockd-Daemon und das Lease-Konzept benutzt.

Diese Strukturen bestimmen die Implementierung der Kommandos GET_LK, SET_LK und SET_LKW. GET_LK wird durch die Funktion fcntl_getlk() aus fs/locks.c ausgeführt. Sie testet, ob der Dateideskriptor geöffnet ist und die Werte der flock-Struktur gültig sind. Dann wird die flock-Struktur in eine file_lock-Struktur kopiert und die Lock-Funktion der Datei aufgerufen, wenn sie definiert ist. Das gilt zum Beispiel für das ext2-Dateisystem nicht. In diesem Fall spürt fcntl_getlk() mögliche Konflikte mit posix_test_lock() auf. Die Funktion posix_test_lock() ruft in einer Schleife für alle POSIX-Dateisperren die Funktion posix_locks_conflict() auf. Ist das der Fall, wird die blockierende Sperre in flock eingetragen und die Funktion kehrt zurück.

Die Kommandos SET_LK und SET_LKW werden durch fcntl_setlk() ausgeführt. Nachdem die Gültigkeit der Parameter überprüft worden ist, testet diese Funktion, ob die Datei im richtigen Modus geöffnet ist. Mit Hilfe der Funktion posix_lock_file() wird die Sperre gesetzt. Dabei werden alle Sperren nach Konflikten mit posix_locks_conflict() durchsucht.

Wird ein solcher Konflikt gefunden, kehrt die Funktion bei SET_LK mit EAGAIN zurück oder blockiert bei SET_LKW. Beim Blockieren wird der aktuelle Prozess in die Warteschlange der Sperre eingetragen. Beim Freigeben dieser Sperre werden alle Prozesse in der Warteschlange geweckt und prüfen die gesetzten Sperren wiederum auf Konflikte. Kann kein Konflikt gefunden werden, wird die Sperre in die Liste der Dateisperren eingetragen.

In Abbildung 5.1 hat der Prozess 1 das erste Byte der Datei und der Prozess 2 das zweite Byte zum Lesen gesperrt. Dann versucht Prozess 1, das zweite Byte zum Schreiben zu sperren, wobei Prozess 1 blockiert. Danach versucht Prozess 2, das erste Byte zu sperren und blockiert dabei. Beide Prozesse würden nun darauf warten, dass der andere seine Sperre freigibt. Solche Situationen werden als Deadlock bezeichnet. Die Szenarios für Deadlocks sind im Allgemeinen komplexer, da mehrere Prozesse daran beteiligt sein können. Linux spürt beim blockierenden Setzen einer Sperre solche Situationen mit posix_locks_deadlock() auf, und der Systemruf *fcntl* kehrt mit dem Fehler EDEADLK zurück.

Die fcntl-Sperren werden bei *fork* nicht auf den Kindprozess übertragen, bleiben aber bei einem *execve* bestehen. Dieses Verhalten ist POSIX-konform, aber sehr einfach zu implementieren.

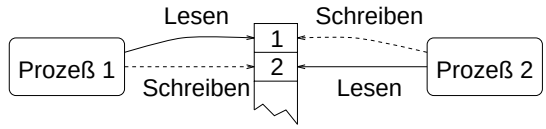


Abbildung 5.1: Ein Deadlockszenario beim Sperren von Dateien

Die `flock`-Sperren sind nicht einzelnen Prozessen zugeordnet, so dass Sperren so lange bestehen bleiben, wie die Datei geöffnet ist. Das ist aber auch nicht schwerer zu implementieren.

5.3 Pipes

Pipes sind die klassische Methode zur Interprozesskommunikation unter Unix. Einem Unix-Nutzer sollte eine Befehlszeile wie

```
% ls -l | more
```

nicht unbekannt vorkommen. Die Shell startet dann die Prozesse `ls` und `more`, die über eine Pipe miteinander verbunden sind. `ls` schreibt Daten in die Pipe, und `more` liest daraus.

Eine andere Variante von Pipes sind benannte Pipes (*named pipes*). Sie werden auch als FIFOs bezeichnet (Pipes funktionieren ebenfalls nach dem Prinzip „first in — first out“). Im folgenden Text werden wir die Begriffe Named Pipe und FIFO synonym verwenden. Im Gegensatz zu Pipes sind FIFOs keine temporären Objekte, die so lange existieren, wie noch ein Dateideskriptor für sie geöffnet ist. Sie können mit dem Kommando

`mkfifo` *Pfadname*

beziehungsweise

`mknod` *Pfadname* `p`

in einem Dateisystem erzeugt werden.

```
% mkfifo fifo
% ls -l fifo
prw-r--r--   1 kunitz   users           0 Feb 27 22:47 fifo|
```

Das Verknüpfen der Standardein- und -ausgabe zweier Prozesse ist mit FIFOs etwas umständlicher.

```
% ls -l >fifo & more <fifo
```

Offensichtlich gibt es zwischen FIFOs und Pipes viele Ähnlichkeiten. Das wird auch von der Implementierung unter Linux ausgenutzt. Die Inodes haben für Pipes und FIFOs dieselben spezifischen Komponenten.

```
struct pipe_inode_info {
   wait_queue_head_t wait;    /* Warteschlange */
   char *base;                /* Adresse des FIFO-Puffers */
   unsigned int start;        /* Offset des aktuellen Bereichs */
   unsigned int readers;      /* Anzahl der aktuell lesenden
                               * Prozesse */
   unsigned int writers;      /* Anzahl der aktuell schreibenden
                               * Prozesse */
   unsigned int waiting_readers; /* Anzahl der lesend
                                  * blockierten Prozesse */
   unsigned int waiting_writers; /* Anzahl der schreibend
                                  * blockierten Prozesse */
   unsigned int r_counter;    /* Zähler für Prozesse, die lesend
                               * geöffnet haben. */
   unsigned int w_counter;    /* Zähler für Prozesse, die schreibend
                               * geöffnet haben. */
};
```

Die Länge des gerade aktuellen Bereichs der Pipe wird im Feld `i_size` verwaltet. Der Systemruf *pipe* erzeugt eine Pipe. Er richtet eine temporäre Inode ein und weist `base` eine Speicherseite zugewiesen. Die architekturabhängige Speicherseitengröße definierte also die Größe einer Pipe. Der Systemruf gibt jeweils einen Dateideskriptor zum Lesen und zum Schreiben zurück.

Bei FIFOs gibt es eine `open`-Funktion, die die Speicherseite zuweist und einen Dateideskriptor zurückgibt. Diesem Dateideskriptor wurde ein Operationsvektor mit Lese- und Schreiboperationen zugewiesen. Das Verhalten ist in Tabelle 5.3 zusammengefasst.

		blockierend	nicht blockierend
zum Lesen	keine schreibenden Prozesse	blockieren	FIFO öffnen
	schreibende Prozesse	FIFO öffnen	FIFO öffnen
zum Schreiben	keine lesenden Prozesse	blockieren	Fehler ENXIO
	lesende Prozesse	FIFO öffnen	FIFO öffnen
zum Lesen und Schreiben		FIFO öffnen	FIFO öffnen

Tabelle 5.3: Das Öffnen einer FIFO

FIFOs und Pipes verwenden dieselben Lese- und Schreiboperationen. Diese Operationen interpretieren den zur Pipe beziehungsweise dem FIFO gehörigen Speicher als Ringpuffer. Daten, die noch nicht ausgelesen worden, sind ab `start` mit einer Länge von `i_size` Bytes gespeichert. Das `O_NONBLOCK`-Flag steuert, ob die Schreib- beziehungsweise Leseoperationen blockieren, wenn es nichts zu tun gibt. Wenn die Anzahl der zu schreibenden Bytes nicht die interne Puffergröße der Pipe übersteigt, muss die Schreiboperation atomar ausgeführt werden. Das heißt, wenn mehrere Prozesse in die Pipe/FIFO schreiben, werden die Bytefolgen der einzelnen Schreiboperationen nicht zerstört. Die in

Linux implementierte Semantik wird in Tabelle 5.4 und Tabelle 5.5 zusammengefasst. Da Prozesse beim Zugriff auf Pipes beziehungsweise FIFOs sehr häufig blockieren, müssen die Schreib- und Leseoperationen dementsprechend oft Prozesse der zur Inode gehörigen Warteschlange wecken. Dabei werden alle Prozesse in einer einzigen Warteschlange verwaltet, obwohl sie auf unterschiedliche Ereignisse warten.

	blockierend	nicht blockierend
leere Pipe	blockieren des aufrufenden Prozesses, wenn schreibende Prozesse vorhanden sind, sonst 0 zurückgeben	Fehler `EAGAIN`, wenn schreibende Prozesse existieren, sonst 0
ansonsten	maximale Anzahl von Zeichen bis zur angeforderten Länge lesen, dabei auf blockierte Schreibprozesse warten	wie bei der blockierenden Operation, aber kein Warten auf Schreibprozesse

Tabelle 5.4: Semantik der Pipe/FIFO-Leseoperation

	blockierend	nicht blockierend
kein lesender Prozess	Signal `SIGPIPE` wird an den schreibenden Prozess gesendet und Rückkehr mit Fehler `EPIPE`	ebenso
atomares Schreiben	blockieren des aufrufenden Prozesses, so lange nicht genug Platz da ist	Fehler EAGAIN, wenn nicht genug Platz da ist
ansonsten	immer wieder blockieren, bis die geforderte Zahl von Bytes geschrieben ist	maximal mögliche Zahl an Bytes schreiben

Tabelle 5.5: Semantik der Pipe/FIFO-Schreiboperation

5.4 Debugging mit ptrace

Es gibt keinen Programmierer, der auf Anhieb fehlerfreie Programme schreiben kann. Werkzeuge, mit denen man Fehler aufspüren kann, sind also gefragt. UNIX stellt den Systemruf *ptrace* bereit, der einem Prozess die Kontrolle über einen anderen Prozess ermöglicht. Der kontrollierte Prozess kann schrittweise abgearbeitet, sein Speicher kann ausgelesen und modifiziert werden. Es können auch Informationen aus der Prozesstabelle gelesen werden. Auf dem Systemruf `ptrace()` basieren Debugger wie etwa `gdb`.

Dieser Systemruf ist wegen seiner Abhängigkeit von der Prozessorarchitektur in der Datei `arch/i386/kernel/ptrace.c` definiert.

```
int sys_ptrace(long request, long pid, long addr,
                            long data);
```

Die Funktion bearbeitet verschiedene Anforderungen, die im Parameter `request` angegeben sind. Der Parameter `pid` gibt die Prozessnummer des zu steuernden Prozesses an.

Mit dem Request `PTRACE_TRACEME` kann ein Prozess festlegen, dass sein Elternprozess ihn per `ptrace()` kontrolliert. Durch `sys_ptrace()` wird das Trace-Flag (`PT_PTRACED`) für den Prozess gesetzt.

Durch `PTRACE_ATTACH` kann der rufende Prozess einen beliebigen Prozess zu seinem Kindprozess machen und dessen `PT_PTRACED`-Flag setzen. Allerdings müssen die Nutzer- und Gruppennummer des rufenden Prozesses mit der effektiven Nutzer- und Gruppennummer des zu steuernden Prozesses übereinstimmen. Dem neuen Kindprozess wird ein `SIGSTOP`-Signal gesendet, so dass dieser im Normalfall seine Abarbeitung unterbricht. Nach dieser Anforderung hat der neue Elternprozess diesen Prozess unter Kontrolle.

Bis auf die Anforderung `PTRACE_KILL` werden die nachfolgenden Anforderungen nur von *ptrace* bearbeitet, wenn der Kindprozess gestoppt ist. Mit den Anforderungen `PTRACE_PEEKTEXT` und `PTRACE_PEEKDATA` können 32-Bit-Werte aus dem Nutzerspeicherbereich des kontrollierten Prozesses gelesen werden. LINUX unterscheidet die beiden Anforderungen nicht. Mit `PTRACE_PEEKTEXT` kann normalerweise Code, mit `PTRACE_PEEKDATA` können Daten gelesen werden. Mit der Anforderung `PTRACE_PEEKUSR` wird ein Long-Wert aus der Struktur `user` für den Prozess gelesen. Dort sind Informationen für das Debugging gespeichert, so zum Beispiel die Debugregister des Prozesses. Sie werden nach einem Debugging-Trap vom Prozessor aktualisiert und durch die entsprechende Behandlungsroutine in die Prozesstabelle geschrieben. Die `user`-Struktur ist virtuell. `sys_ptrace()` schließt aus der zu lesenden Adresse, welche Information zurückgegeben werden muss, und ermittelt diese. So werden die Register vom Stack des Kindprozesses und die Debugregister aus der Prozesstabelle gelesen.

Mit den Anforderungen PTRACE_POKEDATA oder PTRACE_POKETEXT kann der Nutzerbereich des zu kontrollierenden Prozesses modifiziert werden. Ist der zu modifizierende Bereich schreibgeschützt, wird die entsprechende Speicherseite per Copy-On-Write kopiert. Die Speicherseite behält aber ihre Zugriffsattribute. Das wird zum Beispiel dazu genutzt, an eine bestimmte Stelle des Codes einen speziellen Maschinenbefehl zu schreiben, der einen Debugging-Trap auslöst. Auf diese Art und Weise werden von Debuggern Breakpoints gesetzt. Der Code wird so lange ausgeführt, bis der den Trap auslösende Maschinenbefehl (beim x86-Prozessor `int3`) abgearbeitet wird und die Behandlungsroutine des Debugging-Traps die Ausführung des Prozesses unterbricht und den Elternprozess darüber informiert. Es ist auch möglich, mit PTRACE_POKEUSR die virtuelle `user`-Struktur zu modifizieren. Die wichtigste Anwendung dieser Anforderung ist das Modifizieren der Register des Prozesses.

Der durch ein Signal (im Allgemeinen SIGSTOP) unterbrochene Kindprozess kann durch die Anforderung PTRACE_CONT fortgesetzt werden. Mit dem Argument `data` kann bestimmt werden, welches Signal der Prozess behandeln soll, wenn er die Ausführung wieder startet. Der Kindprozess informiert beim Empfang eines Signals den Elternprozess und stoppt. Der Elternprozess kann nun den Kindprozess fortsetzen und entscheiden, ob er das Signal behandeln soll. Ist das `data`-Argument Null, wird der Kindprozess kein Signal bearbeiten.

Die Anforderung PTRACE_SYSCALL setzt den Kindprozess wie PTRACE_CONT fort, allerdings nur bis zum nächsten Systemruf. Die Funktion `sys_ptrace()` setzt dazu das PT_TRACESYS-Flag. Erreicht der Kindprozess den Systemruf, stoppt er und erhält das SIGTRAP-Signal. Der Elternprozess könnte jetzt zum Beispiel die Argumente des Systemrufs untersuchen. Wird der Prozess mit einer erneuten PTRACE_SYSCALL-Anforderung fortgesetzt, stoppt der Prozess bei der Beendigung des Systemrufs. Das Resultat und (gegebenenfalls) die Fehlervariable können vom Elternprozess gelesen werden.

PTRACE_SINGLESTEP unterscheidet sich von PTRACE_CONT dadurch, dass bei PTRACE_SINGLESTEP das Trapflag des Prozessors gesetzt wird. Der Prozess führt also nur einen Maschinenbefehl aus und generiert ein Debuginterrupt (Nummer 1), dieser setzt das SIGTRAP-Signal für den Prozess und ist wieder unterbrochen. Mit der PTRACE_SINGLESTEP-Anforderung kann man also den Maschinencode Befehl für Befehl abarbeiten. Die Anforderung PTRACE_KILL setzt den Kindprozess mit dem gesetzten Signal SIGKILL fort. Er wird abgebrochen.

Durch PTRACE_DETACH wird der mit PTRACE_ATTACH unter Kontrolle genommene Prozess wieder vom kontrollierenden Prozess getrennt. Der kontrollierte Prozess erhält seinen alten Elternprozess zurück, und die Flags PT_PTRACED, PT_TRACESYS sowie das Trapflag des Prozessors werden gelöscht.

In der Version 2.2 von Linux sind nun auch Requests zum Auslesen und Setzen aller allgemeinen und aller Floatingpointregister hinzugekommen. Das Auslesen und Modifizieren über die virtuelle `user`-Struktur mit einzelnen 32-Bit-Werten ist bei modernen Prozessoren zu langsam. Die Requests für allgemeine Register sind PTRACE_GETREGS und PTRACE_SETREGS. Für die Gleitkommaregister werden die Requests

PTRACE_GETFPREGS und PTRACE_SETFPREGS angeboten. Linux 2.4 ergänzte die Kommandos PTRACE_GETFPXREGS und PTRACE_SETFPXREGS, um die zusätzlichen MMX-Register der neueren X86-CPU für das Debugging zu unterstützen.

Ein weiterer neuer Request ist PTRACE_SETOPTIONS. Er setzt, beziehungsweise löscht das Flag PT_TRACESYSGOOD. Es erlaubt dem Vaterprozess zu erkennen, ob ein SIGTRAP-Signal, von einer Unterbrechung nach einem Systemruf oder einem normalen SIGTRAP-Signal herrührt.

Ein Debugger nutzt *ptrace* folgendermaßen: Er führt den Systemruf *fork* aus und ruft im Kindprozess die Funktion mit PTRACE_TRACEME auf. Dort wird dann das zu inspizierende Programm mit *execve* gestartet. Da das PT_PTRACED-Flag gesetzt ist, sendet der Systemruf *execve* ein SIGTRAP-Signal an sich selbst. Der Systemruf gestattet nicht, mit *ptrace* Programme, bei denen ein S-Bit gesetzt ist, zu bearbeiten. Man kann sich sicherlich vorstellen, welche Möglichkeiten sich für Hacker bieten würden, wenn dies nicht so wäre. Bei der Rückkehr von *execve* wird das SIGTRAP-Signal bearbeitet, der Prozess wird angehalten, und der Elternprozess wird informiert, indem ihm ein SIGCHLD-Signal gesendet wird. Der Debugger wird mit einem Aufruf des Systemrufs *wait* darauf warten. Er kann dann den Speicher des Kindes inspizieren, ihn modifizieren und Breakpoints setzen. Die einfachste Methode, das beim x86-Prozessor zu tun, ist, an die entsprechende Adresse im Code einen int3-Maschinenbefehl zu schreiben. Dieser ist nur ein Byte lang.

Ruft der Debugger ptrace() mit der Anforderung PTRACE_CONT auf, läuft der Kindprozess so lange weiter, bis der int3-Maschinenbefehl abgearbeitet wird. Die entsprechende Interruptbehandlungsroutine sendet das SIGTRAP-Signal an den Kindprozess, dieser wird unterbrochen, und der Debugger wird wieder benachrichtigt. Er könnte dann zum Beispiel das zu inspizierende Programm einfach abbrechen.

Es gibt allerdings noch andere Möglichkeiten, diesen Systemruf zu nutzen. Das Programm strace gibt einen Bericht (*Trace*) über alle erfolgten Systemrufe ab. Als Beispiel sei hier die Ausgabe von strace cat motd gezeigt. Selbstverständlich nutzt strace PTRACE_SYSCALL.

```
%strace cat motd
uselib("/lib/ld.so")                    = 0
getuid()                                = 15211
geteuid()                               = 15211
getgid()                                = 15200
getegid()                               = 15200
stat("/etc/ld.so.cache", {st_mode=S_IFREG|0644, st_size=3653,
…}) = 0
open("/etc/ld.so.cache", O_RDONLY)      = 3
mmap(0, 3653, PROT_READ, MAP_SHARED, 3, 0) = 0x40000000
close(3)                                = 0
uselib("/lib/libc.so.4.6.27")           = 0
munmap(0x40000000, 3653)                = 0
munmap(0x62f00000, 24576)               = 0
brk(0)                                  = 0x3000
```

```
brk(0x6000)                              = 0x6000
brk(0x7000)                              = 0x7000
stat("/etc/locale/C/libc.cat", 0xbffff1b0) = -1 ENOENT (No such
file or directory)
stat("/usr/lib/locale/C/libc.cat", 0xbffff1b0) = -1 ENOENT (No
such fil e or directory)
stat("/usr/lib/locale/libc/C/usr/share/locale/C/libc.cat", 0x
bffff1b0) = -1 ENOENT (No such file or directory)
stat("/usr/local/share/locale/C/libc.cat", 0xbffff1b0) = -1 ENOENT
(No such file or directory)
fstat(1, {st_mode=S_IFCHR|0622, st_rdev=makedev(4, 195), …}
) = 0
open("motd", O_RDONLY)                   = -1 ENOENT (No such
file or di rectory)
write(2, "cat: ", 5cat: )                = 5
write(2, "motd", 4motd)                  = 4
write(2, ": No such file or directory", 27: No such file or
directory) = 27
write(2, "\n", 1)                        = 1
close(1)                                 = 0
_exit(1)                                 = ?
```

`ptrace()` bietet ausreichende Funktionalität, um Programme in Multitasking-Umgebungen zu debuggen. Als Kritikpunkt wäre zu vermerken, dass es sehr ineffektiv ist, für das Lesen und Schreiben im Adressraum für je einen 32-Bit-Wert einen Systemruf zu benötigen.

5.5 System V IPC

Schon 1970 wurden in einer speziellen UNIX-Variante die klassischen Varianten der Interprozesskommunikation, Semaphore, Messagequeues und Shared Memory, realisiert. Sie wurden später in System V integriert und werden heute als System-V-IPC bezeichnet. LINUX unterstützt diese Varianten, obwohl sie kein Bestandteil von POSIX sind. Die ursprüngliche LINUX-Implementierung stammt von KRISHNA BALASUBRAMANIAN; sie ist von einer Reihe von Entwicklern immer wieder verbessert worden.

Für die Interprozesskommunikation gibt es jetzt auch POSIX-Spezifikationen, die sich von den Schnittstellen der System-V-IPC stark unterscheiden. Der Kernel stellt mit `mmap()` bereit, diese Schnittstellen im Nutzersegment zu realisieren. In der GNU-C-Library Version 2.2 sind die Schnittstellen für Shared Memory und Semaphore nach POSIX enthalten. Auf dieser Basis lassen sich POSIX-Messagequeues auch realisieren.

5.5.1 Zugriffsrechte, Nummern und Schlüssel

Bei System-V-IPC werden im Kernel Objekte erzeugt. Diesen müssen eindeutige Bezeichner zugewiesen werden, um sicherzustellen, dass von den Nutzerprozessen aktivierte Operationen auch für die richtigen Objekte ausgeführt werden. Die einfachste Form eines Bezeichners ist eine Nummer. Diese Nummern werden dynamisch generiert

und dem Prozess zurückgegeben, der das Objekt erzeugt. Möchte ein vom Erzeugerprozess völlig unabhängiger Prozess auf dieses Objekt zugreifen, kann er das nicht, da ihm die Nummer nicht bekannt ist. Beide Prozesse müssen in diesem Fall einen statischen Schlüssel vereinbaren, mit dem sie das IPC-Objekt referenzieren. Die C-Bibliothek bietet die `ftok`-Funktion, die aus einem Dateinamen und einem Zeichen einen Schlüssel generiert, der mit hoher Wahrscheinlichkeit eindeutig ist. Ein besonderer Schlüssel ist `IPC_PRIVATE`, der garantiert, dass kein existierendes IPC-Objekt referenziert wird. Zugriffe auf mit `IPC_PRIVATE` erzeugte Objekte sind nur über deren Objektnummer möglich.

Die Zugriffsrechte werden vom Kern analog zu Unix System V in der Struktur `kern_ipc_perm` verwaltet.

```
struct kern_ipc_perm
{
        key_t           key;
        uid_t           uid;  /* Eigentümer       */
        gid_t           gid;  /* Eigentümer       */
        uid_t           cuid; /* Erzeuger         */
        gid_t           cgid; /* Erzeuger         */
        mode_t          mode; /* Zugriffsmodi     */
        unsigned long   seq;  /* Zähler für die
                               * Berechnung des
                               * Bezeichners      */
};
```

Da Linux 2.4 jetzt Nutzer- und Gruppennummer mit 32 Bit unterstützt, muss die Systemrufschnittstelle zwei Versionen der Schnittstelle unterstützen: `IPC_OLD` und `IPC_64`. Im Kern werden die Datenstrukturen unabhängig von der beim Systemruf benutzten Schnittstelle behandelt. Die Schnittstellen für die System-V-IPC-Systemrufe werden im Folgenden für die `IPC_64`-Version beschrieben.

Greift ein Prozess auf ein Objekt zu, wird die Routine `ipcperms()` aufgerufen, wobei hier wiederum die üblichen Unix-Zugriffsflags für den Nutzer, die Gruppe und andere verwendet werden. Stimmt die effektive Nutzernummer des zugreifenden Prozesses mit der des Eigentümers oder des Erzeugers überein, werden die Nutzerzugriffsrechte überprüft. Gleiches gilt für die Überprüfung der Gruppenzugriffsrechte.

5.5.2 Semaphore

Das Semaphor-Konzept unter System V erweitert das klassische Semaphor-Modell. Ein Array von Semaphoren kann durch einen Systemruf erzeugt werden. Es ist möglich, in einer einzigen Operation mehrere Semaphore eines Arrays zu modifizieren. Ein Prozess kann Semaphore auf beliebige Werte setzen. Das Inkrementieren und Dekrementieren von Semaphoren kann in Schritten größer als 1 erfolgen. Es ist möglich, dass der Programmierer für Operationen festlegt, dass sie bei Beenden des Prozesses rückgängig gemacht werden.

In Linux gibt es zu jedem reservierten Semaphorarray folgende Datenstruktur:

```
struct sem_array {
  struct kern_ipc_perm sem_perm; /* Berechtigungen           */
  time_t           sem_otime;    /* Zeit der letzten
                                  * Semaphoroperation        */
  time_t           sem_ctime;    /* Zeit der letzten
                                  * Änderung                 */
  struct sem       *sem_base;    /* Zeiger zum ersten
                                  * Semaphor                 */
  struct sem_queue *sem_pending; /* Operationen, die noch
                                  * abgearbeitet werden
                                  * müssen                   */
  struct sem_queue **sem_pending_last;
                                 /* letzte abzuarbeitende
                                  * Operation                */
  struct sem_undo  *undo;        /* Undo-Operationen, die bei
                                  * Löschung des Sema-
                                  * phores abzuarbeiten sind */
  unsigned long    sem_nsems;    /* Anzahl der Semaphoren in
                                  * diesem Array             */
};
```

Der Zugriff auf einzelne Semaphore erfolgt über einen Offset auf `sem_base`. In der Struktur `sem_queue` befindet sich unter anderem eine Warteschlange, in der Prozesse blockieren, deren Operation nicht ausgeführt werden kann. Die Struktur `sem` verwaltet ein einziges Semaphor:

```
struct sem {
   int  semval;   /* Aktueller Wert            */
   int  sempid;   /* Prozessnummer der letzten *
                   * Operation                 */
};
```

Komplexer ist die Aufgabe, einzelne Semaphor-Operationen eines Prozesses bei dessen Beendigung zurückzusetzen. Für jeden Aufruf einer Semaphor-Operation kann der Prozess das Zurücksetzen bei der Beendigung anfordern. Bei solchen Aufrufen werden `sem_undo`-Strukturen dynamisch erzeugt.

```
struct sem_undo {
   struct sem_undo *proc_next; /* Liste aller UNDO-Struktu-
                                * ren eines Prozesses       */
   struct sem_undo *id_next;   /* Liste aller UNDO-Struktu-
                                * ren eines Semaphorarrays  */
   int    semid;               /* Semaphorarraynummer       */
   short * semadj;             /* Werte, auf die die Semaphore
                                * zurückgesetzt werden      */
};
```

In einer `sem_undo`-Struktur werden alle rückgängig zu machenden Operationen eines Prozesses auf einem Semaphor gespeichert. Der Kern legt für ein Semaphor maximal

eine sem_undo-Struktur je Prozess an. Wird der Prozess beendet, versucht der Systemruf *exit*, die Semaphore entsprechend den semadj-Werten zurückzusetzen. Der Prozess blockiert in *exit* nicht, wenn der Wert dadurch kleiner als 0 würde. Der Wert des Semaphors wird einfach auf 0 gesetzt. Dieses Feature wird im Englischen häufig als *Adjust On Exit* bezeichnet. Mit den erläuterten Strukturen sind die Semaphoroperationen realisiert.

```
asmlinkage long sys_semget(key_t key, int nsems, int semflg);
asmlinkage long sys_semop(int semid, struct sembuf *sops,
                          unsigned nsops);
asmlinkage long sys_semctl(int semid, int semnum, int cmd,
                          union semun arg);
```

Sie werden zusammen mit den anderen Operationen von System V IPC über den Systemruf *ipc* aufgerufen. Dieser wiederum ruft anhand seines ersten Arguments die entsprechenden Funktionen auf. Die C-Bibliothek muss alle zugehörigen Bibliotheksrufe in einen Systemruf umsetzen. Man kann hier von Systemrufmultiplexing sprechen.

Mit sys_semget() wird die Nummer eines Semaphorarrays mit nsems-Semaphoren ermittelt. Für semflg können die in Tabelle 5.6 aufgelisteten Werte verwendet werden.

Flag	
0400	Leserechte für Erzeuger
0200	Schreibrechte für Erzeuger
0040	Leserechte für Erzeugergruppe
0020	Schreibrechte für Erzeugergruppe
0004	Leserechte für alle
0002	Schreibrechte für alle
IPC_CREAT	Ein neues Objekt soll erstellt werden, wenn es noch nicht vorhanden ist.
IPC_EXCL	Ist IPC_CREAT gesetzt, und es existiert schon ein solches Objekt, soll die Funktion mit Fehler EEXIST zurückkehren.

Tabelle 5.6: Flags für semget()

semop() führt die durch nsops festgelegte Anzahl von Operationen aus der Tabelle sops aus. Eine Operation wird durch die Struktur sembuf beschrieben:

```
struct sembuf {
   unsigned short sem_num; /* Index des Semaphors im Array */
   short          sem_op;  /* Operation                    */
   short          sem_flg; /* Flags                        */
};
```

Der Wert in sem_op wird zum Semaphor addiert. Die Operation blockiert, wenn die Summe einen negativen Wert ergeben würde. Der Prozess wartet dann darauf, dass der

Semaphor wieder addiert wird. Wenn `sem_op` gleich 0 ist, wartet der Prozess darauf, dass das Semaphore 0 wird. Er blockiert nie, wenn `sem_op` größer als 0 ist. Erhöht sich der Wert, werden alle Prozesse geweckt, die für dieses Semaphorarray auf ein solches Ereignis warten. Analog werden die Prozesse geweckt, die darauf warten, dass ein Semaphor aus dem Array den Wert 0 erreicht. Für `sem_flg` sind zwei Werte möglich, `IPC_NOWAIT` und `SEM_UNDO`. Bei `IPC_NOWAIT` blockiert der Prozess nie. `SEM_UNDO` bewirkt, dass für alle Operationen in diesem Funktionsruf eine `sem_undo`-Struktur eingerichtet beziehungsweise aktualisiert wird. Der negative Operationswert wird in die `sem_undo`-Struktur eingetragen oder bei einer Aktualisierung zum alten Adjust-Wert addiert.

Mit `sys_semctl()` können die verschiedensten Kommandos, die als Parameter angegeben werden müssen, ausgeführt werden. Ein weiterer Parameter dieser Funktion ist die Union `semun`.

```
union semun {
   int val;               /* Wert für SETVAL               */
   struct semid_ds *buf;  /* Puffer für IPC_STAT & IPC_SET */
   unsigned short *array; /* Feld für GETALL & SETALL      */
   struct seminfo *__buf; /* Puffer für IPC_INFO           */
   void *__pad;
};
```

`IPC_INFO` trägt Werte in die `seminfo`-Struktur ein (siehe Tabelle 5.7). Alle Werte sind durch entsprechende Makrodefinitionen fest vorgegeben.

Komponente	Wert	Erläuterung
`semmni`	128	maximale Anzahl von Arrays
`semmns`	32.000	maximale Anzahl von Semaphoren im System
`semmsl`	250	maximale Anzahl von Semaphoren je Array
`semopm`	32	maximale Zahl von Operationen je `semop`-Ruf
`semvmx`	32.767	maximaler Wert eines Semaphors
`semmap`	32.000	wird von Linux ignoriert — Anzahl der Einträge einer „Semaphore map“
`semmnu`	32.000	wird von Linux ignoriert — die maximale Anzahl von `sem_undo`-Strukturen im System
`semume`	32	wird von Linux ignoriert — maximale Anzahl der `sem_undo`-Einträge für einen Prozess
`semusz`	20	wird von Linux ignoriert — Größe der `sem_undo`-Struktur (Wert ist zu groß)
`semaem`	16.383	wird von Linux ignoriert — der maximaler Wert für eine `sem_undo`-Struktur

Tabelle 5.7: Komponenten der `seminfo`-Struktur

Für das Programm `ipcs`, das Informationen über IPC-Objekte anzeigt, gibt es die `SEM_INFO`-Variante dieses Kommandos. Sie gibt in `semusz` die Zahl der eingerichteten Semaphorarrays an und in `semaem` die Gesamtzahl der Semaphore im System.

`IPC_STAT` gibt die `semid64_ds`-Struktur für ein Semaphorarray zurück. Für `ipcs` gibt es hier wiederum die `SEM_STAT`-Variante, bei der man nicht die Nummer des Semaphorarrays, sondern den Index in der Tabelle der Arrays angibt. Das Programm `ipcs` kann damit Informationen über alle Arrays ermitteln, indem es in einer Schleife von 0 bis `seminfo.semmni` zählt und `semctl()` mit `SEM_STAT` und dem Zähler als Argument aufruft.

Durch `IPC_SET` ist es möglich, den Eigentümer und den Modus des Semaphorarrays neu zu setzen. Als Parameter benötigt `IPC_SET` die `sem_setbuf`-Struktur. `IPC_RMID` löscht ein Semaphorarray, wenn der Aufrufer der Inhaber oder Erzeuger des Arrays ist oder wenn der Superuser `semctl()` aufgerufen hat. Die restlichen Kommandos von `sys_semctl()` sind in Tabelle 5.8 zusammengefasst.

Kommando	Rückgabe und Funktion
`GETVAL`	Wert des Semaphors
`GETPID`	Prozessnummer des Prozesses, der das Semaphor zuletzt modifiziert hat
`GETNCNT`	Anzahl der Prozesse, die auf das Inkrementieren des Semaphors warten
`GETZCNT`	Anzahl der Prozesse, die auf den Wert 0 warten
`GETALL`	Werte aller Semaphore des Arrays im Feld des Parameters `semun.array`
`SETVAL`	Wert des Semaphors wird gesetzt.
`SETALL`	Werte der Semaphore werden gesetzt.

Tabelle 5.8: Kommandos für sys_semctl()

5.5.3 Messagequeues

Messages, Nachrichten, bestehen aus einer Folge von Bytes. Zusätzlich enthalten IPC-Messages von System V eine Typkennzeichnung. Prozesse senden Nachrichten in die Messagequeues und können Nachrichten empfangen. Der Empfang kann auf Nachrichten eines bestimmten Typs eingeschränkt werden. Nachrichten werden in der Folge empfangen, in der sie in die Messagequeue eingetragen wurden. Die Grundlage der Implementierung unter Linux ist die Struktur `msg_queue`.

```
struct msg_queue {
  struct kern_ipc_perm q_perm;  /* Zugriffsrechte */
  time_t q_stime;               /* Zeit des letzten Sendens    */
```

```
  time_t q_rtime;              /* Zeit des letzten Empfangs   */
  time_t q_ctime;              /* Zeit der letzten Änderung   */
  unsigned long q_cbytes;      /* aktuelle Anzahl von Bytes in
                                * der Queue                   */
  unsigned long q_qnum;        /* Anzahl von Nachrichten in der
                                * Messagequeue                */
  unsigned long q_qbytes;      /* Fassungsvermögen der
                                * Warteschlange in Bytes      */
  pid_t q_lspid;               /* Prozessnummer des letzten
                                * Senders                     */
  pid_t q_lrpid;               /* Prozessnummer des letzten
                                * Empf"angers                 */

  struct list_head q_messages; /* Liste der Nachrichten       */
  struct list_head q_receivers; /* Liste der blockierten
                                * Empfänger                   */
  struct list_head q_senders;  /* Liste der blockierten
                                * Sender                      */
};
```

Neben Verwaltungsinformationen enthält diese Struktur auch zwei Warteschlangen: `q_senders` und `q_receivers`. Ein Prozess trägt sich in `q_senders` ein, wenn die Messagequeue gefüllt ist. Das heißt, ein Senden der Nachricht ist nicht mehr möglich, ohne die maximal erlaubte Anzahl von Bytes in einer Messagequeue zu überschreiten. Die Warteschlange `q_receivers` enthält Prozesse, die darauf warten, dass Nachrichten in die Warteschlange geschrieben werden.

Die doppelt verkettete Ringliste `q_messages` enthält die Nachrichten der Messagequeue. Einzelne Nachrichten werden im Kernel in der Struktur `msg_msg` gespeichert.

```
struct msg_msgseg {
  struct msg_msgseg *next;
  /* das Nachrichtensegment folgt hier */
};

struct msg_msg {
  struct list_head m_list; /* Liste der Nachrichten in der
                            * Messagequeue                */
  long  m_type;            /* Typ der Nachricht           */
  int m_ts;                /* Länge der Nachricht         */
  struct msg_msgseg* next; /* n"achstes Segment der
                            * Nachricht                   */
  /* die Nachricht beziehungsweise das erste Nachrichtensegment
   * folgt hier */
}
```

Linux speichert die Nachricht direkt hinter dieser Struktur. Sind Struktur und Nachricht länger als eine Speicherseite, wird die Nachricht segmentiert, wobei die folgenden Segmente nur die Struktur `msg_msgseg` enthalten. Damit kann der Freispeicher sehr effizient genutzt werden.

Wie bei den Semaphoren gibt es Funktionen zur Initialisierung, zum Senden und Empfangen von Nachrichten, zur Rückgabe von Informationen und zur Freigabe von Messagequeues. Die auszuführenden Operationen sind zwar relativ einfach, werden aber durch Zugriffsschutz und die Aktualisierung von Statistikdaten komplexer. Die zugehörigen Bibliotheksfunktionen rufen den Systemruf *ipc* auf, das den Aufruf an die entsprechenden Kernfunktionen weiterreicht. Die Funktion `sys_msgget()` erzeugt eine Messagequeue. Sie verwendet die üblichen Parameter der IPC-Get-Funktionen.

```
int sys_msgget (key_t key, int msgflg);
```

`key` ist der geforderte Schlüssel, und `msgflg` entspricht den Flags bei `semget()` (siehe Tabelle 5.6). Mit der Funktion `sys_msgsnd()` werden Nachrichten versendet.

```
struct msgbuf {
   long mtype;         /* Typ der Nachricht  */
   char mtext[1];      /* Text der Nachricht */
};

int sys_msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz,
                int msgflg);
```

Der Parameter `msgsz` ist die Länge des Textes in `mtext` und darf nicht größer als `MSGMAX` sein. Der Prozess blockiert, wenn die neue Anzahl der Bytes in der Messagequeue den Wert in der Komponente `msg_qbytes`, das erlaubte Maximum, übersteigt. Erst wenn andere Prozesse Nachrichten aus der Warteschlange gelesen haben oder nichtblockierte Signale an den Prozess gesendet werden, setzt er die Abarbeitung fort. Das Blockieren kann mit dem Setzen des Flags `IPC_NOWAIT` verhindert werden.

Eine Nachricht kann mit Hilfe von `sys_msgrcv()` wieder aus der Warteschlange gelesen werden.

```
int sys_msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz,
                long msgtyp, int msgflg);
```

Welche Nachrichten empfangen werden, wird in `msgtyp` festgelegt. Beim Wert 0 wird die erste Nachricht aus der Warteschlange selektiert. Bei einem Wert größer als Null wird die erste Nachricht dieses Typs aus der Messagequeue gelesen. Ist allerdings das Flag `MSG_EXCEPT` gesetzt, wird die erste Nachricht empfangen, die nicht dem Messagetyp entspricht. Ist `msgtyp` kleiner als Null, wird die erste Nachricht des Typs mit dem kleinsten Integerwert gelesen, der kleiner oder gleich dem absoluten Wert von `msgtyp` ist. Die Länge der Nachricht muss kleiner als `msgsz` sein. Ist allerdings `MSG_NOERROR` gesetzt, werden nur die ersten `msgsz` Byte der Nachricht gelesen. Wurde keine entsprechende Nachricht gefunden, blockiert der Prozess. Das kann man durch Setzen des `IPC_NOWAIT`-Flags verhindern.

`sys_msgctl()` dient zur Steuerung der Messagequeues. Diese Funktion ähnelt sehr stark `sys_semctl()`.

```
int sys_msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

Das Kommando IPC_INFO gibt wieder die Maxima der für Messagequeues relevanten Werte in der Struktur msginfo aus. Sie sind in der Tabelle 5.9 zusammengefasst. Linux verwendet wiederum nur einen kleinen Teil dieser Werte. Für alle Komponenten

Komponente	Wert	Erläuterung
msgmni	16	maximale Anzahl von Messagequeues
msgmax	8.192	maximale Größe einer Nachricht in Bytes
msgmnb	16.384	Standardwert für die maximale Größe einer Messagequeue in Bytes
msgmap	16.384	nicht verwendet — Anzahl der Einträge in einer „Message Map“
msgpool	256	nicht verwendet — Größe des „Messagepools“
msgtql	16.384	nicht verwendet — Anzahl von „System Message Headers“
msgssz	16	nicht verwendet — Nachrichtensegmentgröße
msgseg	0x4000	nicht verwendet — maximale Anzahl von Segmenten

Tabelle 5.9: Komponenten der msginfo-Struktur

sind in msg.h die entsprechenden Makros definiert. Das Kommando MSG_INFO ist die von IPC_INFO abgewandelte Variante für das Programm ipcs. Es gibt in msgpool die Anzahl der benutzten Warteschlangen, in msgmap die Anzahl der Nachrichten und in msgtql die Gesamtzahl der Bytes der vom System gespeicherten Nachrichten zurück.

IPC_STAT kopiert die msqid_ds-Struktur der referenzierten Messagequeue in den Nutzerspeicherbereich. Die MSG_STAT-Variante lässt als Parameter analog zu SEM_STAT den Index in die systeminterne Tabelle der Messagequeue zu. Auch dieses Linux-Feature wird vom Befehl ipcs benutzt.

Durch IPC_SET können der Eigentümer, der Modus und die maximal mögliche Zahl von Bytes in der Messagequeue modifiziert werden. Ein Prozess, der keine Superuser-Rechte besitzt, darf diesen Wert nicht größer als MSGMNB (16.384) setzen. Wäre das nicht so, hätte ein normaler Prozess die Chance, durch Hochsetzen dieses Wertes und Senden von Nachrichten in die Messagequeue Kernelspeicher zu allozieren, der nicht aus dem primären Speicher ausgelagert werden kann.

Der Besitzer oder Erzeuger der Messagequeue sowie der Superuser können mit Hilfe von IPC_RMID die Messagequeue wieder löschen.

Den Messagequeues kam unter Linux 2.0 die Funktion zu, die Kommunikation mit dem kerneld-Dämon zu übernehmen. Dieser Dämon war dafür zuständig, auf Anforderung vom Kernel, Kernelmodule mit modprobe zu laden. Der Linux 2.2 Kernelthread kmod erfüllt diese Aufgabe jetzt durch direkten Aufruf dieses Programms. Damit wurden eine ganze Reihe von spezifischen Modifikationen am Messagequeue-Code überflüssig und konnten gelöscht werden.

5.5.4 Shared Memory

Shared Memory ist die schnellste Form der Interprozesskommunikation. Prozesse, die einen gemeinsamen Speicherabschnitt nutzen, können durch normale Maschinenbefehle zum Lesen und Schreiben Daten austauschen. Bei allen anderen Verfahren ist das nur über Systemrufe möglich, indem die Daten vom Speicherbereich des einen Prozesses in den Speicherbereich des anderen Prozesses kopiert werden. Der Nachteil von Shared Memory ist, dass die Prozesse zusätzliche Synchronisationsmechanismen verwenden müssen, um Wettbewerbsbedingungen auszuschließen. Die schnellere Kommunikation bedingt einen höheren Programmieraufwand. Die Synchronisation über andere Systemrufe auszuführen ist zwar portabel, verringert aber den Geschwindigkeitsvorteil. Eine andere Möglichkeit wäre, die Maschinenbefehle zum bedingten Setzen eines Bits der Prozessoren verschiedener Architekturen auszunutzen. Solche Befehle setzen ein Bit abhängig von seinem Wert. Da dies innerhalb eines Maschinenbefehls geschieht, kann diese Operation nicht durch Interrupts unterbrochen werden. Der gegenseitige Ausschluss lässt sich mit diesen Befehlen sehr einfach und sehr schnell realisieren. In Abschnitt 4.2.2 wurde schon dargestellt, wie komplex die gemeinsame Benutzung von Speicherbereichen ist. Da ab der Version 2.0 mit `mmap()` auch Speicherbereiche schreibbar von mehreren Prozessen eingeblendet werden können, können auch über diesen Mechanismus Shared-Memory-Anwendungen realisiert werden.

Wie bei den anderen IPC-Varianten von System V wird ein gemeinsam benutztes Speichersegment durch eine Nummer identifiziert. Diese Nummer verweist auf eine `shmid_kernel`-Datenstruktur, die öffentliche und nur im Kern benutzte Informationen enthält. Ein Prozess kann ein solches Speichersegment mit Hilfe einer Attach-Operation in das Nutzersegment des virtuellen Adressraums einblenden. Mit einer Detach-Operation wird diese Einblendung wieder rückgängig gemacht. Der Einfachheit halber werden wir den durch die `shmid_kernel`-Struktur verwalteten Speicher als Segment bezeichnen, obwohl dieser Begriff auch schon für die Segmente des virtuellen Adressraums von x86-Prozessoren verwendet wird.

```
struct shmid_kernel
{
  struct kern_ipc_perm shm_perm;   /* Zugriffsrechte          */
  struct file *        shm_file;   /* Datei im SHM-Datei-
                                    * system                  */
  int                  id;         /* Id                      */
  unsigned long        shm_nattch; /* Anzahl der Attachments  */
  unsigned long        shm_segsz;  /* Größe des Segments      */
  time_t               shm_atim;   /* Zeit des letzten Attach */
  time_t               shm_dtim;   /* Zeit des letzten Detach */
  time_t               shm_ctim;   /* Zeit der Erstellung     */
  pid_t                shm_cprid;  /* PID des Erzeugers       */
  pid_t                shm_lprid;  /* PID der letzten
                                    * Operation               */
};
```

Die Moduskomponente der `kern_ipc_perm`-Struktur wird zur Speicherung zweier zusätzlicher Flags verwendet. Das Flag `SHM_LOCKED` verhindert, dass Seiten des gemein-

sam benutzten Speichersegments auf sekundäre Geräte ausgelagert werden. SHM_DEST bestimmt, dass das Segment bei der letzten Detach-Operation freigegeben wird.

Das Feld shm_file verweist auf eine Datei im SHM-Dateisystem, die beim Anlegen des Segments erzeugt wird. Das SHM-Dateisystem ist keinem Gerät zugeordnet und benutzt nur den Speicherseitencache. Diese Datei kann dann mit do_mmap() in den Speicherbereich eines Prozesses eingeblendet werden. Das heißt, System-V-Shared-Memory wird von der Linux-Speicherverwaltung wie mit mmap()-eingeblendeter Speicher behandelt. Dies ist neu für Linux 2.4 und hat eine Menge Code für die Sonderbehandlung von System-V-Shared-Memory eingespart. Der Kernel mountet das SHM-Dateisystem automatisch an einem internen Mount-Punkt, der außerhalb des Kernels nicht sichtbar ist. Man kann zwar ein weiteres Shared-Memory-Dateisystem zum Beispiel unter /dev/shm mounten, die Shared-Memory-Segmente des System-V-IPCs sind dort logischerweise nicht sichtbar. Das unter /dev/shmem gemountete Dateisystem kann aber für die Umsetzung von POSIX Shared Memory genutzt werden. Es gibt Kernel-Patches mit denen man die im Shared-Memory-Dateisystem angelegten Dateien auch mit read() und write() lesen kann. Dieses Dateisystem könnte zum Beispiel als Dateisystem für temporäre Dateien genutzt werden. Solaris nutzt ein solches Dateisystem für das /tmp-Verzeichnis.

Mit sys_shmget() kann ein Prozess ein Segment erzeugen beziehungsweise darauf verweisen.

```
asmlinkage long sys_shmget(key_t key, size_t size, int shmflg);
```

Der Parameter size gibt die Größe des Segments an. Ist das Segment schon eingerichtet, kann der Parameter kleiner als die eigentliche Größe sein. Im Parameter shmflg können wieder die in Tabelle 5.6 aufgeführten Flags gesetzt sein.

Die Funktion initialisiert nur die shmid_kernel-Datenstruktur. Zwar wird schon der Eintrag im Kernel-internen SHM-Dateisystem erzeugt, allerdings werden für die Datei noch keine Speicherseiten reserviert.

Die Hauptfunktion für die Nutzung von Shared Memory ist sys_shmat(). Sie blendet das Segment in das Nutzersegment des Prozesses ein.

```
int sys_shmat (int shmid, char *shmaddr, int shmflg,
               unsigned long *addr);
```

Der Parameter shmaddr kann vom Prozess benutzt werden, um die Adresse, an die das Segment eingeblendet werden soll, festzulegen. Ist er Null, sucht sich die Funktion selbst einen freien Speicherbereich. Die gewählte Adresse wird über addr zurückgegeben. Dieses etwas umständliche Verfahren über den Parameter musste benutzt werden, da sonst bei der Rückkehr in den Nutzerprozess Adressen über 2 Gigabyte als Fehlerwerte interpretiert würden.

In shmflg sind als Flags SHM_RND und SHM_RDONLY zugelassen. Ist SHM_RND gesetzt, wird die übergebene Adresse auf eine Speicherseitengrenze abgerundet. Linux erlaubt nur die Einblendung eines Segments auf eine Speicherseitengrenze. SHM_RDONLY gibt an, dass das Segment nur lesbar eingeblendet wird.

Für das Einblenden des Speicherbereichs wird `do_mmap()` aufgerufen. Auch hier wird der Shared-Memory-Bereich noch nicht alloziert, das passiert erst, wenn der Zugriff auf eine Speicherseite im eingeblendeten Shared-Memory-Segment fehlschlägt. `shmem_nopage()` sorgt dann dafür, dass eine entsprechende Speicherseite eingeblendet wird. Sie überprüft, ob die Speicherseite vielleicht doch schon in den Auslagerungsbereich geschrieben worden ist. Wenn ja wird die Speicherseite eingelesen, ansonsten wird eine neue Speicherseite alloziert.

Die Funktion `sys_shmdt()` blendet ein gemeinsam benutztes Segment aus dem Nutzersegment eines Prozesses aus.

```
int sys_shmdt (char *shmaddr);
```

Die Funktion `sys_shmctl()` ist das Pendant zu den bereits erläuterten Funktionen `sys_semctl()` und `sys_msgctl()`.

```
int sys_shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

Ein Aufruf dieser Funktion mit dem `IPC_INFO`-Kommando gibt die Maximalwerte für die Benutzung der Shared-Memory-Implementierung von Linux zurück. Die Tabelle 5.10 fasst die Komponenten der dazu genutzten `shminfo`-Struktur zusammen.

Komponente	Wert für x86	Erläuterung
`shmmni`	4096	die maximale Anzahl von Shared-Memory-Segmenten
`shmmax`	33.554.432	maximale Größe eines Segments in Bytes
`shmmin`	1	minimale Größe eines Segments
`shmall`	2.097.152	maximale Anzahl von geteilten Speicherseiten im gesamten System
`shmseg`	4096	erlaubte Zahl von Segmenten je Prozess

Tabelle 5.10: Komponenten der `shminfo`-Struktur für das `IPC_INFO`-Kommando

Obwohl für die Größe des Segments der Wert 1 erlaubt ist, wird von Linux immer mindestens eine Speicherseite (4.096 Byte) für die gemeinsame Benutzung alloziert. Das `SHM_INFO`-Kommando füllt die `shm_info`-Struktur. Tabelle 5.11 fasst sie zusammen.

Das Kommando `IPC_STAT` kann zum Auslesen der Segmentdatenstruktur `shmid_ds` aufgerufen werden. Die `SHM_STAT`-Variante dieses Kommandos erfüllt dieselbe Aufgabe, erwartet aber als Parameter nicht die Segmentnummer, sondern einen Index in die Tabelle der Segmentdatenstrukturen.

Wird `sys_shmctl()` mit dem Kommando `IPC_SET` aufgerufen, können der Eigentümer und der Zugriffsmodus für ein Segment vom alten Eigentümer oder dem Prozess, der das Shared-Memory-Segment initialisiert hat, modifiziert werden.

Im Gegensatz zu den Funktionen `sys_semctl()` und `sys_msgctl()` kann bei dem Kommando `IPC_RMID` die IPC-Datenstruktur nicht in jedem Fall freigegeben werden, da

Komponente	Erläuterung
`used_ids`	Anzahl der benutzten Segmente
`shm_tot`	Gesamtzahl der gemeinsam genutzten Seiten
`shm_rss`	Zahl der gemeinsam genutzten Speicherseiten, die im Hauptspeicher alloziert sind
`shm_swp`	Anzahl der momentan ausgelagerten Seiten
`swap_attempts`	Versuche, gemeinsam genutzte Seiten auszulagern
`swap_successes`	Zahl der ausgelagerten gemeinsam genutzten Seiten seit dem Start des Systems

Tabelle 5.11: Komponenten der `shm_info`-Struktur für das `SHM_INFO`-Kommando

es möglicherweise noch Prozesse gibt, die das Segment eingeblendet haben. Um die Segmentstruktur als gelöscht zu markieren, wird im Modusfeld der `ipc_perm`-Komponente das Flag `SHM_DEST` gesetzt.

Die Kommandos `SHM_LOCK` und `SHM_UNLOCK` erlauben es dem Superuser, das Auslagern der Seiten eines Segments zu verbieten beziehungsweise wieder zu erlauben.

5.5.5 Die Befehle ipcs und ipcrm

Ein Nachteil der System-V-IPC ist, dass das Testen und Entwickeln von Programmen, die sie benutzen, sehr leicht zu folgendem Problem führen kann: Nach Beendigung von Testprogrammen bleiben IPC-Ressourcen bestehen, ohne dass dies beabsichtigt war. Mit dem Befehl `ipcs` kann man die Situation untersuchen und mit `ipcrm` die entsprechenden Ressourcen löschen.

So könnten zum Beispiel durch ein Programm drei Semaphorarrays erzeugt worden sein. `ipcs` gibt Informationen über die Shared-Memory-Segmente, Semphorarrays und Messagequeues aus, auf die der Nutzer Zugriffsrechte hat.

```
% ipcs

—— Shared Memory Segments ————
shmid      owner      perms      bytes      nattch     status



—— Semaphore Arrays ————
semid      owner      perms      nsems      status
1152       kunitz     666        1
1153       kunitz     666        1
1154       kunitz     666        1

—— Message Queues ————
msqid      owner      perms      used-bytes  messages
```

Mit `ipcrm` kann nun jeweils eines dieser Semaphorarrays gelöscht werden. Für Messagequeues und Shared-Memory-Segmente kann das Kommando analog angewendet werden.

```
% ipcrm sem 1153

resource deleted
% ipcs

—— Shared Memory Segments ————
shmid      owner      perms      bytes      nattch     status

—— Semaphore Arrays ————
semid      owner      perms      nsems      status
1152       kunitz     666        1
1154       kunitz     666        1

—— Message Queues ————
msqid      owner      perms      used-bytes  messages
```

In Linux 2.4 werden Informationen über System-V-IPC-Ressourcen unter `/proc/sysvipc` angezeigt. In `/proc/sys/kernel` können die IPC-Parameter modifiziert werden, ohne dass eine Kernel-Neukompilation notwendig ist.

5.6 IPC mit Sockets

Bis jetzt haben wir nur Formen der Interprozesskommunikation kennengelernt, die die Kommunikation von Prozessen eines Rechners unterstützen. Die Socket-Programmierschnittstelle ermöglicht die Kommunikation über ein Netz sowie lokal auf einem Rechner. Der Vorteil der Socketschnittstelle liegt darin, dass sie die Programmierung von Netzwerkanwendungen unter der Verwendung des unter Unix schon lange etablierten Konzepts des Dateideskriptors gestattet. Ein besonders gutes Beispiel dafür ist der INET-Dämon. Der Dämon wartet auf ankommende Netzdienstanforderungen und ruft dann die entsprechenden Dienstprogramme mit dem Socket-Dateideskriptor als Standardeingabe und -ausgabe auf. Bei sehr einfachen Diensten braucht das aufgerufene Programm keine einzige Zeile netzwerkrelevanten Codes zu enthalten.

In diesem Kapitel werden wir uns auf die Verwendung und Implementierung von Unix-Domain-Sockets beschränken. Sockets der INET-Domain werden im Kapitel 8 erklärt.

5.6.1 Ein einfaches Beispiel

Mit Unix-Domain-Sockets können ähnlich wie bei FIFOs Programme verbindungsorientiert Daten austauschen. Im folgenden Beispiel wird gezeigt, wie das funktioniert. Für Client- und Server-Programme werden dieselben Include-Dateien verwendet.

```
/* sc.h */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define SERVER "/tmp/server"
```

Die Aufgabe des Clients besteht darin, eine Nachricht mit seiner Prozessnummer an den Server zu senden und die Antwort des Servers auf die Standardausgabe zu schreiben.

```
/* cli.c - Client verbindungsorientiertes Modell */
#include "sc.h"

int main(void)
{
   int sock_fd;
   struct sockaddr_un unix_addr;
   char buf[2048];
   int n;

   if ((sock_fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
   {
      perror("cli: socket()");
      exit(1);
   }

   unix_addr.sun_family = AF_UNIX;
   strcpy(unix_addr.sun_path, SERVER);

   if (connect(sock_fd, (struct sockaddr*) &unix_addr,
               sizeof(unix_addr.sun_family) +
               strlen(unix_addr.sun_path)) < 0)
   {
      perror("cli: connect()");
      exit(1);
   }

   sprintf(buf, "Hallo Server! Ich bin %d.\n", getpid());
   n = strlen(buf) + 1;

   if (write(sock_fd, buf, n) != n)
   {
      perror("cli: write()");
      exit(1);
   }

   printf("Client sent: %s", buf);
```

```
   if ((n = read(sock_fd, buf, 2047)) < 0)
   {
      perror("cli: read()");
      exit(1);
   }

   buf[n] = 0;
   while (buf[n] == 0) { n--; }

   if (buf[n] == '\n')
      buf[n] = '\0';
   printf("Client received: %s\n", buf);
   exit(0);
}
```

Zuerst wird mit `socket()` ein Socket-Dateideskriptor erzeugt. Danach wird die Adresse des Servers generiert, die bei UNIX-Domain-Sockets aus einem Dateinamen besteht, in unserem Falle `/tmp/server`. Der Client versucht dann, mit `connect()` eine Verbindung zum Server herzustellen. Gelingt das, ist es möglich, durch ganz normale `read`- und `write`-Funktionen Daten an den Server zu senden. Genau dies macht der Client, indem die Nachricht

`Hallo Server! Ich bin` *Prozessnummer des Clients*`.`

gesendet wird. Damit der Server antworten kann, sind einige Zeilen C-Programm mehr notwendig.

```
/* srv.c - Server verbindungsorientiertes Modell */

#include <signal.h>
#include "sc.h"

static void stop(int n)
{
   unlink(SERVER);
   exit(0);
}

static void server(void)
{
   int sock_fd, cli_sock_fd;
   struct sockaddr_un unix_addr;
   char buf[2048];
   int n, addr_len;
   pid_t pid;
   char *pc;

   signal(SIGINT, stop);
   signal(SIGQUIT, stop);
   signal(SIGTERM, stop);
```

```
if ((sock_fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
{
   perror("srv: socket()");
   exit(1);
}

unix_addr.sun_family = AF_UNIX;
strcpy(unix_addr.sun_path, SERVER);
addr_len = sizeof(unix_addr.sun_family) +
           strlen(unix_addr.sun_path);
unlink(SERVER);

if (bind(sock_fd, (struct sockaddr *) &unix_addr,
         addr_len) < 0)
{
   perror("srv: bind()");
   exit(1);
}

if (listen(sock_fd, 5) < 0)
{
   perror("srv: client()");
   unlink(SERVER); exit(1);
}

while ((cli_sock_fd =
        accept(sock_fd, (struct sockaddr*) &unix_addr,
               &addr_len)) >= 0)
{
   if ((n = read(cli_sock_fd, buf, 2047)) < 0)
   {
      perror("srv: read()");
      close(cli_sock_fd);
      continue;
   }

   buf[n] = '\0';
   for (pc = buf; *pc != '\0' &&
                    (*pc < '0' || *pc > '9'); pc++);

   pid = atol(pc);

   if (pid != 0)
   {
      sprintf(buf, "Hallo Client %d! Hier ist der Server.\n",
              pid);
      n = strlen(buf) + 1;

      if (write(cli_sock_fd, buf, n) != n)
         perror("srv: write()");
   }
```

```
      close(cli_sock_fd);
   }

   perror("srv: accept()");
   unlink(SERVER); exit(1);
}

int main(void)
{
   int r;

   if ((r = fork()) == 0) server();

   if (r < 0)
   {
      perror("srv: fork()");
      exit(1);
   }
   exit(0);
}
```

Der Server ruft `fork()` auf und beendet die Ausführung. Der Kindprozess läuft im Hintergrund weiter und installiert eine Behandlungsroutine für Unterbrechungssignale. Nachdem ein Socket-Dateideskriptor geöffnet worden ist, wird die eigene Adresse an diesen Socket gebunden. Dabei wird eine Datei unter dem in der Adresse angegebenen Pfadnamen erzeugt. Durch Einschränken der Zugriffsrechte dieser Datei könnte der Server die Menge der Nutzer, die mit ihm kommunizieren können, verkleinern. Der `connect`-Aufruf eines Clients ist nur erfolgreich, wenn diese Datei existiert und er die notwendigen Zugriffsrechte besitzt. Der Aufruf von `listen()` ist notwendig, um den Kern darüber zu informieren, dass der Prozess nun bereit ist, Verbindungen auf diesem Socket zu akzeptieren. Mit `accept()` wird dann auf Verbindungen gewartet. Wird durch einen Client mit `connect()` eine Verbindung aufgebaut, kehrt `accept()` mit einem neuen Socket-Dateideskriptor zurück. Dieser wird dann benutzt, um Nachrichten vom Client zu empfangen und diese zu beantworten. Der Server schreibt einfach zurück:

`Hallo Client` *Prozessnummer des Clients*`! Hier ist der Server.`

Der Server schließt dann den Dateideskriptor für diese Verbindung und ruft wiederum `accept()` auf, um dem nächsten Client seinen Dienst anzubieten.

Die Schreib- und Leseoperationen blockieren normalerweise auf dem Socketdeskripor, wenn entweder keine Daten vorhanden sind oder kein Platz mehr im Puffer vorhanden ist. Wurde mit Hilfe von `fcntl()` das `O_NONBLOCK`-Flag gesetzt, blockieren diese Funktionen nicht.

Ab Version 2.0 ist es möglich, mit den Funktionen `sendto()` und `recvfrom()` UNIX-Domain-Sockets im verbindungslosen Modus zu verwenden.

5.6.2 Die Implementierung von Unix-Domain-Sockets

Ein Socket wird im Kern durch die Datenstruktur `socket` repräsentiert. Die in den Sockets enthaltenen Daten werden in `sk_buff`-Strukturen gespeichert. Sie werden in Kapitel 8 erläutert.

Es gibt eine Reihe socketspezifischer Funktionen, wie etwa `socket()` und `setsockopt()`. Sie werden alle durch einen einzigen Systemruf realisiert. Er heißt *socketcall* und ruft anhand des ersten Parameters alle benötigten Funktionen auf. Die Dateioperationen `read()`, `write()`, `poll()`, `ioctl()`, `lseek()`, `close()` und `fasync()` werden direkt über die Dateioperationen des Dateideskriptors aufgerufen.

Die Socketfunktionen im Nutzerbereich haben zugehörige Funktionen im Kernel, die mit dem Präfix `sys_` versehen sind. Diese Funktionen unterstützen verschiedene Protokolle und Adressfamilien. Die Funktion `sys_socket()` ermittelt anhand einer Adressfamilientabelle, welche Funktion den Socket initialisieren soll. Alle anderen Socketoperationen stützen sich auf die protokollspezifischen Funktionen, die im Operationsvektor `proto_ops` enthalten sind. `proto_ops` ist in der Socketstruktur enthalten. Im Folgenden wird die Semantik der Socketoperationen für das Unix-Domain-Sockets kurz erläutert.

```
long sys_socket(int family, int type, int protocol);
```

Ein Socket-Dateideskriptor wird eingerichtet. Die Funktion ruft die Protokolloperation `unix_create()` auf. Diese Funktion kann blockieren. Der Status des Sockets ist nach dem Beenden dieser Operation `SS_UNCONNECTED`.

```
long sys_bind(int fd, struct sockaddr *umyaddr, int addrlen);
```

Die Adresse `umyaddr` wird an den Socket gebunden. Die Protokolloperation testet natürlich, ob die Adresse der Unix-Adressfamilie angehört, und versucht, die Socket-Adressdatei einzurichten. Der Aufruf `sys_bind()` ist nur erfolgreich, wenn die Socket-Adressdatei noch nicht von einem anderen Programm gebunden wurde.

```
long sys_connect(int fd, struct sockaddr *uservaddr,
                 int addrlen);
```

Diese Operation versucht, den Socket mit der Adresse `uservaddr` zu verbinden. Natürlich muss die Adresse eine Unix-Domain-Adresse sein. Es wird versucht, die Socket-Adressdatei des Servers zu öffnen. Für Datagramm-Sockets reicht das auch völlig aus. Bei Stream-Sockets testet die Protokolloperation `unix_connect()`, ob an der Serveradresse überhaupt Verbindungen akzeptiert werden. War die Operation erfolgreich, ist der Socketstatus danach `SS_CONNECTED`.

```
long sys_listen(int fd, int backlog);
```

Der Server teilt dem Kern mit dieser Operation mit, dass er ab jetzt Verbindungen akzeptiert. Der Status in der `sock`-Struktur wird auf `TCP_LISTEN` gesetzt und `max_ack_backlog` bekommt den Wert des Parameters `backlog`.

```
long sys_accept(int fd, struct sockaddr *upeer_sockaddr,
                int *upeer_addrlen);
```

Ein Prozess kann diese Operation nur aufrufen, wenn vorher `listen()` für diesen Socket aufgerufen wurde. Der Prozess blockiert, wenn es keine Prozesse gibt, die ein `connect()` für die Adresse unseres Sockets aufgerufen haben.

```
long sys_getsockname(int fd, struct sockaddr *usockaddr,
                     int *usockaddr_len);
```

Die Protokolloperation `unix_getname()` ist die Basis für diese Funktion. Die an den Socket gebundene Adresse wird zurückgegeben.

```
int sys_getpeername(int fd, struct sockaddr *usockaddr,
                     int *usockaddr_len);
```

Die Operation basiert ebenfalls auf der Protokolloperation `unix_getname()` des Sockets. Ein Parameter dieser Funktion bestimmt aber, dass die Adresse des verbundenen Sockets (*Peer*) zurückgegeben werden soll.

```
long sys_socketpair(int family, int type, int protocol,
                    int usockvec[2]);
```

Es werden zwei Socketdeskriptoren erzeugt und miteinander verbunden.

```
long sys_send(int fd, void * buff, int len, unsigned flags);
long sys_sendto(int fd, void * buff, int len, unsigned flags,
        struct sockaddr *addr, int addr_len);
long sys_sendmsg(int fd, struct msghdr *msg, unsigned int flags);
```

Dies sind die verschiedenen Socketoperationen zum Versenden von Nachrichten. Abhängig davon ob das Socket ein Datagramm- oder ein Streamsocket ist, wird `unix_dgram_sendmsg()` beziehungsweise `unix_stream_sendmsg()` aufgerufen. `unix_sendmsg()`. Dort werden die Nachrichten möglicherweise auf mehrere `sk_buff`-Strukturen aufgeteilt und in die Empfangsliste des Peersockets geschrieben.

```
long sys_recv(int fd, void * buff, int len, unsigned flags);
long sys_recvfrom(int fd, void * ubuf, int size, unsigned flags,
        struct sockaddr *addr, int *addr_len);
long sys_recvmsg(int fd, struct msghdr *msg, unsigned int flags);
```

Diese Funktionen rufen abhängig davon, ob es sich um ein Datagramm- oder ein Streamsocket handelt, `unix_dgram_recvmsg()` oder `unix_stream_recvmsg()` auf. Dabei kann blockiert werden, wenn der Peer keine Daten geschrieben hat.

```
long sys_shutdown(int fd, int how)
```

Diese Socketoperation wird durch `unix_shutdown()` realisiert. Im Socketstatus wird markiert, ob noch empfangen und gesendet werden darf. Ein eventuell vorhandener Peersocket wird entsprechend markiert.

```
long sys_getsockopt(int fd, int level, int optname, char *optval,
```

```
        int *optlen)
long sys_setsockopt(int fd, int level, int optname, char *optval,
        int optlen)
```

Da es für UNIX-Sockets keine spezifischen Optionen gibt, werden hier nur die allgemeinen Socket-Optionen auf dem `SOL_SOCKET`-Layer unterstützt.

Da es Prozessen möglich sein sollte, Sockets wie normale Dateideskriptoren zu benutzen, muss die Funktionalität fast aller Dateioperationen unterstützt werden. Nur die Operationen `readdir()` und `fsync()` werden gar nicht unterstützt. Sockets haben eine Spezialbehandlung `sock_no_open()` des Open-Calls, UNIX-Sockets haben für `mmap()` `sock_no_mmap()` definiert. Die `lseek()`-Behandlungsroutine gibt auch nur -ESPIPE zurück.

Die meisten Dateioperation werden schon allgemein für alle Sockets auf der Socket-Ebene behandelt. UNIX-Sockets behandeln `poll()` und `ioctl()` speziell, der `close()`-Ruf wird auch mit `unix_release()` behandelt. Die Funktion `unix_ioctl()` ermöglicht es dem Nutzerprozess, mit `SIOCOUTQ` die Anzahl der Bytes in der Empfangsqueue abzufragen. Das IOCTL-Kommando `SIOCINQ` fragt die Anzahl der Bytes in der Sendequeue ab.

Es sei noch darauf hingewiesen, dass durch Setzen des `O_NONBLOCK`-Flags des Dateideskriptors das Blockieren des Prozesses bei der Ausführung dieser Operation verhindert werden kann. Die Datei, die von `bind()` angelegt wird, kann nur geöffnet und geschlossen werden.

In der Inode-Struktur der Datei ist das Flag `S_IFSOCK` gesetzt, so dass eine solche Datei als spezielle Socket-Adressdatei gekennzeichnet ist. Ein `ls -lF` für die Socket-Adressdatei aus dem Beispiel gibt die Meldung aus.

```
% ls -l server
srwxr-xr-x  1 kunitz   mi89       0 Mar  7 00:09 server=
```

Pearson Education

ADDISON-WESLEY

Pearson Education
ADDISON-WESLEY

6 Das Linux-Dateisystem

»Mein lieber Watson, gute Informationen zu bekommen, ist nicht schwer. Viel schwieriger ist es, sie wiederzufinden.«

Conan Doyle, Die Abenteuer des Sherlock Holmes

Im PC-Bereich sind unterschiedliche Betriebssysteme nicht gerade eine Seltenheit. Fast jedes Betriebssystem bringt dabei auch sein eigenes Dateisystem mit. Jedes erhebt natürlich für sich den Anspruch, „schneller, besser und sicherer" zu sein als seine Vorgänger.

Gerade die große Anzahl der von Linux unterstützten Dateisysteme ist sicher einer der hauptsächlichen Gründe dafür, dass Linux innerhalb seiner kurzen Geschichte so schnell an Akzeptanz gewann. Nicht jeder Nutzer kann für ein neues Betriebssystem seine alten Daten mit großem Aufwand in ein anderes Dateisystem konvertieren.

Diese Vielfalt unterstützter Dateisysteme wird durch die einheitliche Schnittstelle zum Linux-Kern ermöglicht. Dabei handelt es sich um den *Virtual Filesystem Switch* (VFS), der im Weiteren einfach als „Virtuelles Dateisystem" bezeichnet wird, obwohl es sich dabei eigentlich nicht um ein Dateisystem handelt, sondern um eine Schnittstelle, die einen klar definierten Übergang zwischen dem Betriebssystemkern und den unterschiedlichen Dateisystemen zur Verfügung stellt (Abbildung 6.1 illustriert dies).

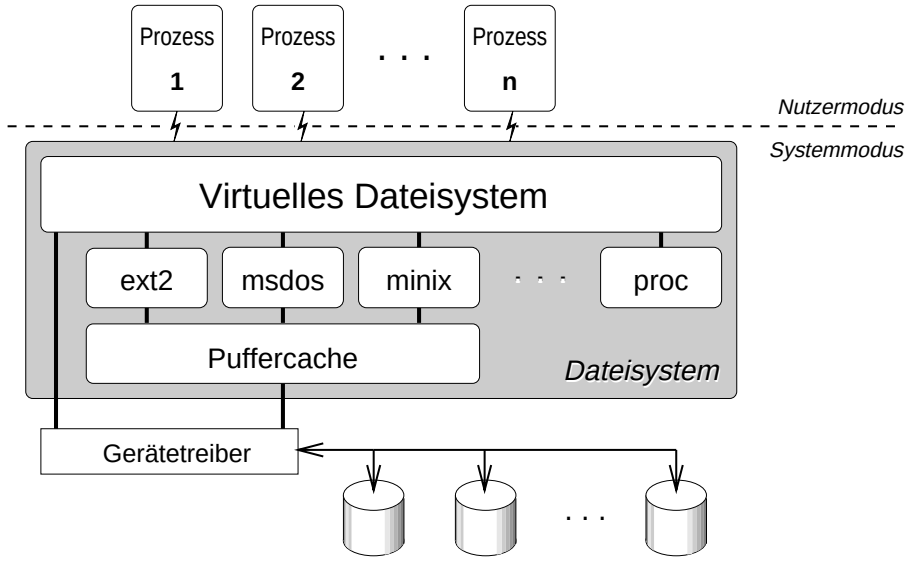


Abbildung 6.1: Die Schichten des Dateisystems

Das Virtuelle Dateisystem stellt den Applikationen die Systemrufe zur Dateiverwaltung (siehe Abschnitt A.2) zur Verfügung, führt die Verwaltung interner Strukturen durch

und leitet die Arbeit an das entsprechende eigentliche Dateisystem weiter. Eine weitere wichtige Aufgabe des VFS ist es, Standardaktionen durchzuführen. So stellt in der Regel keine Dateisystemimplementation eine Funktion `lseek()` bereit, da die Standardaktion des VFS die Funktionalität von `lseek()` realisiert. Es ist somit doch gerechtfertigt, beim VFS von einem Dateisystem zu reden.

Im Folgenden sollen die Funktionsweise des VFS sowie das Zusammenspiel zwischen VFS und konkreten Dateisystemimplementationen genauer betrachtet werden. Als einfaches Beispiel soll dazu die Implementierung des *Proc*-Dateisystems genauer betrachtet werden. Des Weiteren wollen wir auf das Design und den Aufbau des *Ext2*-Dateisystems als das Linux-Dateisystem eingehen.

6.1 Grundlagen

Die Bedeutung eines guten Datenverwaltungssystems wird sehr oft unterschätzt. Während der Mensch dazu sein Gedächtnis oder Notizbuch verwenden kann, muss ein Computer zu anderen Mitteln greifen.

Eine sehr wichtige Forderung an ein Dateisystem ist die *sinnvolle Strukturierung* von Daten. Bei der Wahl einer sinnvollen Strukturierung darf aber wiederum die *Geschwindigkeit* des Datenzugriffs sowie die Möglichkeit eines *wahlfreien Zugriffs* nicht vernachlässigt werden.

Wahlfreier Zugriff wird durch Blockgeräte ermöglicht, die in eine bestimmte Anzahl gleich großer Blöcke unterteilt sind. In Linux können wir uns dabei auf eine andere Instanz, nämlich den in Abschnitt 4.3 beschriebenen Puffercache, verlassen. Es ist also möglich, mit Hilfe der Puffercache-Funktionen auf einen beliebigen der sequenziell durchnummerierten Blöcke eines bestimmten Geräts zuzugreifen. Das Dateisystem muss nun in der Lage sein, eine eindeutige Zuordnung der Daten auf die Geräteblöcke zu gewährleisten.

In Unix sind die Daten in einem hierarchischen Dateisystem untergebracht, das Dateien unterschiedlichen Typs enthält. Dieses umfaßt nicht nur reguläre Dateien und Verzeichnisse, sondern auch Gerätedateien, FIFOs (*named pipes*), symbolische Links und Sockets. Es kann dadurch auf alle Ressourcen des Systems über Dateien zugegriffen werden.

Dateien sind aus Programmierersicht nur Datenströme beliebigen Inhalts, die keine weitere Strukturierung enthalten. Das Dateisystem übernimmt dabei die Aufgabe, diese „Datenströme“ effizient zu verwalten sowie die *Darstellung unterschiedlicher Dateiarten* (also auch Pseudodateien) zu ermöglichen.

Die zur Verwaltung nötigen Informationen werden in Unix strikt von den Daten getrennt. Für jede Datei sind diese Informationen in einer separaten Inode-Struktur zusammengefasst. Abbildung 6.2 zeigt den Aufbau einer typischen Unix-Inode. In ihr befinden sich unter anderem Zugriffszeiten, Zugriffsrechte sowie die Zuordnung der Daten zu den Blöcken auf dem physischen Speichermedium. Wie Sie sehen, enthält die Inode bereits einige Datenblocknummern, um den effizienten Zugriff auf kleine Dateien (die

unter Unix häufig auftreten) zu gewährleisten. Der Zugriff auf größere Dateien erfolgt über indirekte Blöcke, die ihrerseits wieder Blocknummern enthalten. Jede Datei wird durch genau eine Inode repräsentiert. Innerhalb eines Dateisystems besitzt jede Inode deshalb eine eindeutige Nummer. Somit lässt sich auch die Datei selbst über diese Nummer ansprechen.

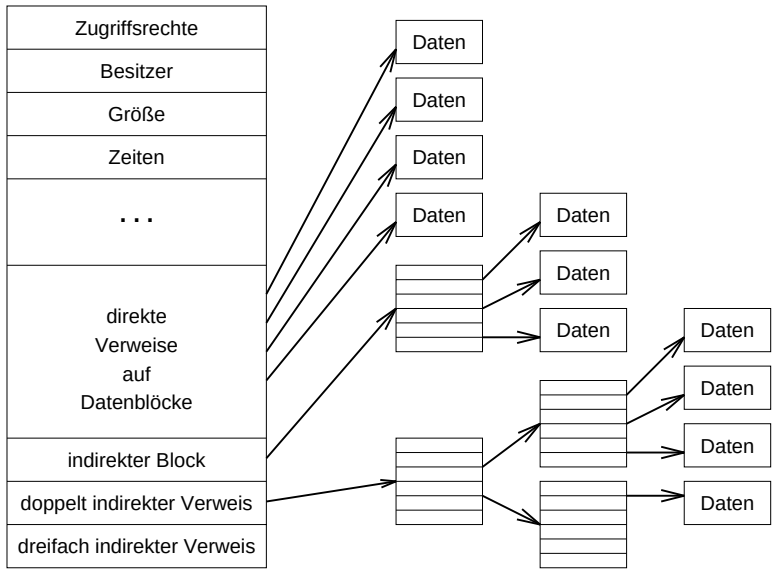


Abbildung 6.2: Aufbau einer Unix*-Inode*

Verzeichnisse sorgen für den hierarchischen Aufbau des Dateisystems. Sie sind ebenfalls als Dateien implementiert, wobei der Kern jedoch voraussetzt, dass sie Paare aus dem Dateinamen und der dazugehörigen Inode-Nummer enthalten. Eine Datei kann durchaus durch mehrere Namen, die sogar in unterschiedlichen Verzeichnissen liegen können, angesprochen werden (*Hardlink*). Bei älteren Unix-Versionen war das Ändern der Verzeichnisdateien noch mit Hilfe eines einfachen Editors möglich; aus Konsistenzgründen ist das bei neueren Versionen nicht mehr erlaubt. So lassen die Linux-Dateisysteme nicht einmal das Auslesen dieser Dateien zu.

Der prinzipielle Aufbau der unterschiedlichen Unix-Dateisysteme ist von der Struktur her gleich (siehe Abbildung 6.3). Jedes Dateisystem beginnt mit einem *Boot-Block*. Dieser Block ist reserviert, um den zum Booten des Betriebssystems notwendigen Code aufzunehmen (siehe Anhang D). Da Dateisysteme meist auf jedem blockorientierten Gerät existieren sollen und dort vom Prinzip her (aus Gründen der Einheitlichkeit) denselben Aufbau besitzen, existiert der Boot-Block auch, wenn der Computer nicht von dem Gerät gebootet wird.

Alle für die Verwaltung des Dateisystems wichtigen Informationen sind im *Superblock* untergebracht. Danach folgen mehrere *Inode-Blöcke*, die die Inode-Strukturen des Dateisystems enthalten. Die verbleibenden Blöcke des Gerätes dienen zur Aufnahme der

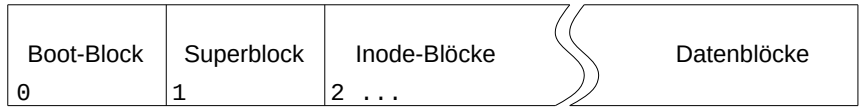


Abbildung 6.3: Schematischer Aufbau eines Unix*-Dateisystems*

Daten. Diese *Datenblöcke* enthalten demnach sowohl den Inhalt regulärer Dateien als auch die Verzeichniseinträge und die indirekten Blöcke.

Da Dateisysteme auf unterschiedlichen Geräten beheimatet sind, müssen die Dateisystemimplementationen sich auch an unterschiedliche Gerätecharakteristika wie Blockgrößen usw. anpassen. Dabei streben alle Betriebssysteme nach *Geräteunabhängigkeit*, so dass es unerheblich ist, auf welchem Medium die Daten abgelegt werden. In Linux übernimmt die entsprechende Dateisystemimplementation diese Aufgabe, so dass das Virtuelle Dateisystem mit geräteunabhängigen Strukturen arbeiten kann.

In Unix werden einzelne Dateisysteme nicht wie bei anderen Betriebssystemen durch einzelne Bezeichner (zum Beispiel Laufwerksbezeichner) angesprochen, sondern sind in einem hierarchischen Verzeichnisbaum vereinigt.

Das Vereinigen geschieht durch das *Mounten*[1] eines Dateisystems. Dabei wird dem bestehenden Verzeichnisbaum ein weiteres (beliebiges) Dateisystem hinzugefügt. Man spricht auch davon, dass das neue Dateisystem in den Verzeichnisbaum „eingehängt" bzw. „eingebunden", dem Verzeichnisbaum „aufgesetzt" oder am Verzeichnisbaum „befestigt" wird. Da unserer Meinung nach keines dieser deutschen Wörter den Sachverhalt des „Mountens" richtig beschreibt, werden wir im Folgenden die englischen Begriffe *mounten* und *unmounten* gebrauchen.

Ein neues Dateisystem kann auf ein beliebiges Verzeichnis gemountet werden. Dieses originale Verzeichnis heißt dann *Mount-Point* und wird mitsamt seinen Unterverzeichnissen und Dateien durch das Wurzelverzeichnis des neuen Dateisystems verdeckt. Ein Unmounten des Dateisystems gibt die verdeckte Verzeichnisstruktur dann wieder frei.

Ein nicht zu unterschätzender Aspekt für die Güte eines Dateisystems ist außerdem die *Datensicherheit*. Dazu gehören einerseits Möglichkeiten der Konsistenzerhaltung und Mechanismen zur Gewährleistung des Datenschutzes. Andererseits sollte das Dateisystem robust gegenüber Systemfehlern, Verletzungen der Datenintegrität sowie Programmabstürzen sein. Das Paradebeispiel dafür ist das neue XFS von SGI ...

6.2 Die Repräsentation von Dateisystemen im Kern

Die Darstellung der Daten auf Diskette oder Festplatte kann sich von Fall zu Fall stark unterscheiden. Im Endeffekt aber ist die eigentliche Darstellung von Dateien bei Linux im

1 *Engl.* mount: aufkleben, -setzen, befestigen, einbinden, einhängen, montieren etc.

Speicher gleich. Auch hier hält sich Linux eng an sein „Vorbild" Unix, denn die Verwaltungsstrukturen der Dateisysteme sind dem logischen Aufbau eines Unix-Dateisystems sehr ähnlich.

Verwaltet werden diese vom VFS, das die dateisystemspezifischen Funktionen der einzelnen Implementierungen aufruft, um die Strukturen zu füllen. Diese Funktionen sind in jeder konkreten Implementierung enthalten. Neue Implementierungen werden dem VFS mit Hilfe der Funktion `register_filesystem()` bekannt gemacht, indem sie an die Liste der bekannten Dateisysteme angehängt werden.

```
int register_filesystem(struct file_system_type * fs)
{
        int res = 0;
        struct file_system_type ** p;
        …
        write_lock(&file_systems_lock);
        p = find_filesystem(fs->name);
        if (*p)
                res = -EBUSY;
        else
                *p = fs;
        write_unlock(&file_systems_lock);
        return res;
}
```

Im Beispiel des *Ext2*-Dateisystems geschieht das mit `init_ext2_fs()`, welche wiederum die `register`-Funktion aufruft[2]:

```
static DECLARE_FSTYPE_DEV(ext2_fs_type, "ext2", ext2_read_super);

static int __init init_ext2_fs(void)
{
        return register_filesystem(&ext2_fs_type);
}
```

Das VFS erhält somit den Namen des Dateisystems (`"ext2"`), eine Funktion zum Mounten sowie (über die Makro-Expansion) ein Flag, das anzeigt, ob ein Gerät zum Mounten unbedingt nötig ist. Die dabei übergebene Funktion `read_super()` bildet die Mount-Schnittstelle. Erst durch sie werden dem Virtuellen Dateisystem weitere Funktionen der Dateisystemimplementation bekannt gegeben.

Die Funktion legt die Struktur `file_system_type`, die ihr übergeben wurde, in einer einfach verketteten Liste ab. `file_systems` zeigt auf den Anfang dieser Liste.

```
struct file_system_type {
   const char *name;
   int fs_flags;
   struct super_block *(*read_super) (struct super_block *,
                                      void *, int);
```

2 Die Umsetzung geschieht in der Datei `super.c` des jeweiligen Dateisystems.

```
    struct module *owner;
    struct vfsmount *kern_mnt;
    struct file_system_type *next;
}
static struct file_system_type *file_systems;
```

In älteren Linux-Kernen (vor Version 1.1.8) wurden die Strukturen noch in einer statischen Tabelle verwaltet, da alle Dateisystemimplementationen zur Übersetzungszeit des Linux-Kerns eingebunden wurden. Erst durch die Einführung von Modulen ergab sich die Möglichkeit und Notwendigkeit, Dateisysteme auch zur Laufzeit des Linux-Systems nachträglich einzubinden.

Beim Registrieren eines Dateisystems können verschiedene Flags angegeben werden. Diese steuern das Verhalten des Kernels, wenn später ein Dateisystem mit diesem Typ gemountet wird. In der Version 2.4 definiert der Linux-Kern die folgenden Registrierungsflags:

FS_REQUIRES_DEV Dieses Dateisystem benötigt ein Gerät zum Mounten.

FS_NO_DCACHE Anweisung an das VFS, den Verzeichniscache (siehe Abschnitt 6.2.4) für dieses Dateisystem nicht zu benutzen. Derzeit nicht implementiert.

FS_NO_PRELIM Derzeit nicht implementiert.

FS_SINGLE Das Dateisystem hat nur einen Superblock.

FS_NOMOUNT Auf diesem Dateisystem können später keine Mount-Operationen ausgeführt werden.

FS_LITTER Wird ein Dateisystem mit diesem Flag abgemountet, so werden alle DCache-Einträge für dieses Dateisystem gelöscht.

Nachdem eine Dateisystemimplementation beim VFS registriert wurde, ist es möglich, Dateisysteme dieses Typs zu mounten.

6.2.1 Das Mounten

Bevor auf eine Datei zugegriffen werden kann, muss das Dateisystem, auf dem sich die Datei befindet, erst einmal gemountet werden. Dies geschieht entweder durch den Systemruf *mount* oder die Funktion `mount_root()`.

Die Funktion `mount_root()` ist dabei für das Mounten des ersten Dateisystems (*Root-Dateisystem*) zuständig. Sie wird beim Starten des Systems nach der Registrierung der Geräte und Dateisystemimplementationen von der Funktion `do_basic_setup()` aufgerufen.

Jedes gemountete Dateisystem wird durch eine Struktur `super_block` repräsentiert. Diese Strukturen werden in der dynamischen Liste `super_blocks` vom Typ `struct list_head` gehalten. Allerdings ist die maximale Länge dieser Liste durch die

Variable `max_super_blocks` begrenzt, die mit `NR_SUPER` initialisiert wird. Sie kann jedoch über das Sysctl-Interface zur Laufzeit verändert werden (siehe A.1).

Zur Initialisierung des Superblocks dient die Funktion `read_super()` des Virtuellen Dateisystems. Sie erzeugt einen leeren Superblock, hängt ihn in die Superblockliste ein und ruft die von jeder Dateisystemimplementation bereitgestellte Funktion zur Erzeugung des Superblocks auf. Diese dateisystemspezifische Funktion ist diejenige, die bei der Registrierung der Implementierung dem VFS übergeben wurde. Sie erhält beim Aufruf

- die mittlerweile mit den allgemeinen Informationen gefüllte Superblockstruktur (das Gerät und die Flags, letztere sind entsprechend Tabelle 6.1 gefüllt),
- einen Zeiger (`void *`), auf weitere Mount-Optionen für das Dateisystem sowie
- ein Flag `silent`, das anzeigt, ob fehlgeschlagenes Mounten durch Meldungen anzuzeigen ist.

  Dieses Flag wird nur durch die Kernfunktion `mount_root()` genutzt, da diese der Reihe nach alle vorhandenen Funktionen `read_super()` der einzelnen Dateisystemimplementationen zum Mounten des Root-Dateisystems aufruft und die dabei entstehenden Fehlermeldungen beim Hochfahren des Systems stören würden.

Makro	Wert	Bemerkung
`MS_RDONLY`	1	Dateisystem ist nur lesbar
`MS_NOSUID`	2	ignoriert S-Bits
`MS_NODEV`	4	verbietet den Zugriff auf Gerätedateien
`MS_NOEXEC`	8	verbietet die Programmausführung
`MS_SYNCHRONOUS`	16	unmittelbares Schreiben auf Platte
`MS_REMOUNT`	32	Änderung der Flags
`MS_MANDLOCK`	64	erlaubt absolute Dateizugriffssperrung
`MS_NOATIME`	1024	Zeitpunkt des letzten Zugriffs wird nicht aktualisiert
`MS_NODIRATIME`	2048	Zeitpunkt des letzten Zugriffs auf Verzeichnisse wird nicht aktualisiert

Tabelle 6.1: Die dateisystemunabhängigen Mount-Flags des Superblocks

6.2.2 Der Superblock

Die dateisystemspezifische Funktion `read_super()` liest, falls nötig, ihre Informationen mit den in Abschnitt 4.3 vorgestellten Funktionen des Linux-Caches von dem entsprechenden Blockgerät. Dies ist auch der Grund, warum zum Mounten von Dateisys-

temen ein Prozess benötigt wird[3]. Dieser kann nämlich, da der Zugriff auf das entsprechende Gerät Zeit braucht, durch den Gerätetreiber angehalten werden. Dazu verwendet er den Sleep-WakeUp-Mechanismus (siehe Abschnitt 3.1.5), der mit Prozessen arbeitet. Der Linux-Superblock hat Folgendes Aussehen:

```
struct super_block {
   struct list_head s_list;        /* dient zur Verkettung   */
                                   /* in Liste super_blocks  */
   kdev_t s_dev;                   /* Gerät des Dateisystems */
   unsigned long s_blocksize;      /* Blockgröße             */
   unsigned char s_blocksize_bits; /* ld (Blockgröße)        */
   unsigned char s_lock;           /* Superblocksperre       */
   unsigned char s_rd_only;        /* nicht genutzt (= 0)    */
   unsigned char s_dirt;           /* Superblock geändert    */
   struct file_system_type *s_type;/* Dateisystemtyp         */
   struct super_operations *s_op;  /* Superblock-Operationen */
   struct dquot_operations *dq_op; /* Quota-Operationen      */
   unsigned long s_flags;          /* Flags                  */
   unsigned long s_magic;          /* Dateisystemkennung     */
   struct dentry *s_root;          /* DEntry von '/'         */
   wait_queue_head_t s_wait;       /* s_lock -Warteschlange  */
   struct list_head s_dirty;       /* Liste aller Dirty Inodes */
   struct list_head s_files;
   struct block_device *s_bdev;    /* Blockgerät-Struktur    */
   struct list_head s_mounts;      /* Liste der Mounts       */

   union {
      struct minix_sb_info minix_sb;
                  …
      void *generic_sdp;
   } u;            /* dateisystemspezifische Informationen  */

   struct semaphore s_vfs_rename_sem; /* Semaphor für Um-    */
                   /* benennungen von Verzeichnissen        */

   struct semaphore s_nfsd_free_path_sem; /* Semaphor für    */
                   /* den Zugriff auf Unterverzeichnisse    */
};
```

Der Superblock enthält Informationen über das gesamte Dateisystem, wie etwa beispielsweise die Blockgröße, die Zugriffsrechte und den Dateisystem-Typ. Des Weiteren enthält die Union `u` am Ende der Struktur spezielle Informationen über die entsprechenden Dateisysteme. Für spezielle Dateisystem-Module existiert dabei der Zeiger `generic_sdp`.

Die Komponenten `s_lock` und `s_wait` gewährleisten die Synchronisation des Zugriffs auf den Superblock. Dazu dienen die Funktionen `lock_super()` und `unlock_super()`, die in der Datei `<linux/locks.h>` definiert sind.

3 Wenn das Root-Dateisystem gemountet wird, existiert zwar noch kein User-Prozess, jedoch der Kernel-Thread `init()`, der `do_basic_setup()` aufruft.

```
extern inline void lock_super(struct super_block * sb)
{
   if (sb->s_lock)
      __wait_on_super(sb);
   sb->s_lock = 1;
}

extern inline void unlock_super(struct super_block * sb)
{
   sb->s_lock = 0;
   wake_up(&sb->s_wait);
}
```

Außerdem befindet sich im Superblock ein Verweis auf den `dentry` (*Directory-Entry*) des Dateisystems `s_root`.

Eine weitere Aufgabe der Funktion `read_super()` der konkreten Dateisystemimplementation ist demnach, die Root-Inode des Dateisystems zur Verfügung zu stellen, damit sie in einen `dentry` umgewandelt und in den Superblock eingetragen werden kann. Dies kann durch die Funktionen des VFS, wie etwa die später beschriebene Funktion `iget()`, erfolgen, sofern die Komponenten `s_dev` und `s_op` richtig gesetzt sind.

6.2.3 Superblock-Operationen

Die Superblockstruktur stellt Funktionen zum Zugriff auf das Dateisystem in dem Funktionsvektor `s_op` bereit, die die Grundlage für die weitere Arbeit mit dem Dateisystem bilden.

```
struct super_operations {
   void (*read_inode) (struct inode *);
   void (*write_inode) (struct inode *);
   void (*put_inode) (struct inode *);
   void (*delete_inode) (struct inode *);
   int  (*notify_change) (struct dentry * dentry,
                struct iattr * attr);
   void (*put_super) (struct super_block *);
   void (*write_super) (struct super_block *);
   void (*statfs) (struct super_block *, struct statfs *, int);
   int  (*remount_fs) (struct super_block *, int *, char *);
   void (*clear_inode) (struct inode *);
   void (*umount_begin) (struct super_block *);
};
```

Die Funktionen der Struktur `super_operations` dienen zum Lesen und Schreiben der einzelnen Inodes, zum Schreiben des Superblocks sowie zum Auslesen der Dateisysteminformationen. Die Superblock-Operationen enthalten also Funktionen, die die konkrete Darstellung des Superblocks und der Inodes auf dem Datenträger in die allgemeine Form im Speicher überführen und umgekehrt. Diese Schicht verbirgt somit die konkreten Darstellungen vollkommen. Genau genommen müssen die Inodes und der Superblock nicht

einmal existieren. Ein Beispiel dafür ist das MS-DOS-Dateisystem, bei dem die FAT und die Informationen im Boot-Block in die Unix-interne Sicht von Superblock und Inodes überführt werden. Manche Superblock-Operationen sind optional. Sind sie nicht implementiert, d.h. ist der Zeiger der Operation `NULL`, so findet keine weitere Aktion statt. Eine Übersicht darüber, welches Dateisystem welche Operation implementiert, ist in Tabelle 6.2 aufgelistet. Die mit [x] gekennzeichneten Einträge sind dann aktiv, wenn der Schreibzugriff für das Dateisystem konfiguriert wurde.

`read_inode(inode)` Diese Funktion muss implementiert werden und ist für das Füllen der ihr übergebenen Struktur `inode` zuständig. Sie wird von der schon erwähnten Funktion `get_new_inode()` aufgerufen, die schon folgende Einträge belegt:

```
inode->i_sb = sb;
inode->i_dev = sb->s_dev;
inode->i_ino = ino;
inode->i_flags = 0;
inode->i_count = 1;
inode->i_state = I_LOCK;
```

`read_inode()` sorgt hauptsächlich für die Unterscheidung der verschiedenen Dateiarten, indem sie in Abhängigkeit von Dateisystem und Dateiart die Inode-Operationen in die Inode einträgt. So enthält fast jede `read_inode`-Funktion, wie zum Beispiel die des *Ext2*-Dateisystems, die folgenden Zeilen:

```
if (S_ISREG(inode->i_mode))
   inode->i_op = &ext2_file_inode_operations;
else if (S_ISDIR(inode->i_mode))
   inode->i_op = &ext2_dir_inode_operations;
else if (S_ISLNK(inode->i_mode))
   inode->i_op = &ext2_symlink_inode_operations;
```

`write_inode(inode)` Die Funktion `write_inode()` dient zum Sichern der Informationen der Inode-Struktur. Sie ist das Gegenstück zu `read_inode()` und muss dafür sorgen, dass alle Informationen der VFS-Inode in die Dateisystemdarstellung zurückübertragen werden. Diese Operation ist optional, muss jedoch von allen Dateisystemen implementiert werden, die den schreibenden Zugriff erlauben.

`put_inode(inode)` Sie wird von `iput()` aufgerufen, wenn die Inode nicht mehr gebraucht wird, weil z. B. die zugehörige Datei geschlossen wurde. Ihre Hauptaufgabe ist alle von dieser Inode momentan belegten Blöcke und andere Ressorcen freizugeben. Diese Operation ist optional.

`delete_inode(inode)` Diese Funktion wird ebenfalls von `iput()` aufgerufen, wenn die Komponente `i_nlink` null ist, somit diese Inode auf dem Dateisystem nicht mehr referenziert wird. Sie muss diese Inode löschen. Diese Operation ist optional, muss jedoch von allen Dateisystemen implementiert werden, die den schreibenden Zugriff erlauben.

Dateisystem	read_inode	write_inode	put_inode	delete_inode	notify_change	put_super	write_super	statfs	remount_fs	clear_inode	umount_begin
adfs	x					x		x	x		
affs	x	x	x	x	x	x	x	x	x		
autofs	x	x	x	x		x		x			
coda	x		x	x	x	x		x			
devpts	x	x				x		x			
efs	x					x		x			
ext2	x	x	x	x		x	x	x	x		
fat		x		x	x	x		x		x	
hfs	x		x		x	x	x	x			
hpfs	x			x	x	x		x	x		
isofs	x					x		x			
minix	x	x		x		x	x	x	x		
msdos	*wie fat*										
ncpfs	x		x	x	x	x		x			
nfs	x		x	x	x	x		x			x
ntfs	x	[x]				x		x	x	x	
proc	x	x	x	x		x		x			
qnx4	x	[x]	[x]	[x]		x	[x]	x	x		
romfs	x					x		x			
smbfs	x		x	x	x	x		x			
sysv	x	x		x	x	x	x	x			
ufs	x	x	x	x		x	x	x	x		
umsdos		x	x	x	x	x		x		x	
vfat	*wie fat*										

Tabelle 6.2: Die Implementierungen der Superblock-Operationen

`notify_change(inode, attr)` Die Änderung der Inode durch Systemrufe wird mit dem Aufruf der Operation `notify_change()` quittiert. Dies geschieht mit Hilfe der Struktur `iattr`:

```
struct iattr {
   unsigned int   ia_valid; /* Flags der geänderten Komponenten */
```

```
    umode_t         ia_mode;  /* neue Zugriffsrechte */
    uid_t           ia_uid;   /* neuer Nutzer */
    gid_t           ia_gid;   /* neue Gruppe */
    off_t           ia_size;  /* neue Größe */
    time_t          ia_atime; /* Zeitpunkt des letzten Zugriffs */
    time_t          ia_mtime; /* Zeitpunkt der letzten Änderung */
    time_t          ia_ctime; /* Zeitpunkt der Erzeugung */
    unsigned int    ia_attr_flags; /* Inode-Flags   */
};
```

Die Funktionen, die `notify_change()` aufrufen, sowie die in der Komponente `ia_valid` übergebenen Flags sind in Tabelle 6.3 aufgelistet. In `ia_attr_flags` stehen die dateisystemspezifischen Flags. Für diese Operation stellt das VFS eine Standardimplementierung bereit, auf die alle anderen Dateisystemimplementierungen zurückgreifen können.

Kernfunktion	ATTR_MODE	ATTR_UID	ATTR_GID	ATTR_SIZE	ATTR_ATIME	ATTR_MTIME	ATTR_CTIME	ATTR_ATIME_SET	ATTR_MTIME_SET	ATTR_FORCE	ATTR_ATTR_FLAG
`sys_chmod()`	x						x				
`sys_fchmod()`	x						x				
`sys_chown()`	x	x	x				x				
`sys_lchown()`	x	x	x				x				
`sys_fchown()`	x	x	x				x				
`sys_truncate()`				x			x				
`sys_ftruncate()`				x			x				
`sys_utime()`					x	x	x	x	x		
`sys_utimes()`					x	x	x	x	x		

Tabelle 6.3: Die Flags von `notify_change()`

`put_super(sb)` Das Virtuelle Dateisystem ruft diese Funktion beim Unmounten von Dateisystemen auf. Dabei sollte sie den Superblock und weitere Informationspuffer (z. B. Inode-Bitmap, Freiblockliste usw.) wieder freigeben (siehe `brelse()` in Abschnitt 4.3) bzw. die Konsistenz des Dateisystems wiederherstellen. Diese Operation ist optional.

`write_super(sb)` Die Funktion `write_super()` dient zum Zurückschreiben der Informationen des Superblocks. Die Konsistenz des Dateisystems muss dabei nicht un-

bedingt gewährleistet sein[4]. Unterstützt das jeweilige Dateisystem ein auf Inkonsistenz hinweisendes Flag, so wird es (das Flag) gelöscht. Im Normalfall wird in der Funktion der Cache veranlasst, die Puffer des Superblocks zurückzuschreiben. Dafür sorgt einfach das Setzen des `s_dirt`-Flags des Puffers. Die optionale Funktion wird beim Synchronisieren des Geräts verwendet und hat für Nur-Lese-Dateisysteme wie z. B. das *Isofs* keinen Sinn.

`statfs(sb, statfsbuf, int)` Die beiden Systemrufe *statfs* sowie *fstatfs* (siehe Seite 379) rufen die Superblock-Operation auf, die eigentlich nur zum Füllen der Struktur `statfs` dient. Diese Struktur liefert Informationen über das Dateisystem, die Anzahl der freien Blöcke und die bevorzugte Blockgröße. Beim Fehlen der Operation liefert das VFS den Fehler `ENODEV`.

`remount_fs(sb, flags, options)` Die Funktion `remount_fs()` ändert den Zustand eines Dateisystems (siehe Tabelle 6.1). Dabei werden meist nur die neuen Attribute des Dateisystems in den Superblock eingetragen sowie die Konsistenz des Dateisystems wiederhergestellt.

`clear_inode(inode)` Diese Funktion wird unter anderem von `iput()` aufgerufen und löscht die Informationen einer Inode. Die Implementierung des VFS gibt außerdem die Quotas frei und setzt den Status der Inode auf 0.

`umount_begin(sb)` Diese Funktion wird beim Unmounten eines Geräts aufgerufen, falls die Option `MNT_FORCE` gesetzt ist. Sie ist optional und ist derzeit nur bei NFS implementiert. Dabei werden alle Aufrufe, die auf das Dateisystem zugreifen, abgebrochen und liefern die Fehlermeldung `-EIO` zurück.

6.2.4 Der Verzeichniscache

Der Verzeichniscache geht ursprünglich auf das *Ext2*-Dateisystem zurück. Seit der Linux-Version 1.1.37 gehört er zum VFS und kann von allen Dateisystemimplementationen genutzt werden. In diesem Cache werden Verzeichniseinträge gehalten, um Zugriffe beim Auslesen von Verzeichnissen, wie sie beim Öffnen von Dateien notwendig sind, zu beschleunigen. Er soll das alte Problem lösen, dass der Nutzer mit Dateinamen arbeitet, der Kern aber mit Inodes. Also muss der Kern zu einem Namen die Inode ermittlen — und bei erneuten Zugriffen immer wieder. Im Gegensatz zu den Inodes, die persistent auf Festplatten existieren, sind die Einträge des Verzeichniscaches (im Weiteren DEntry genannt) rein RAM-basiert. Die Einträge in diesem Cache haben folgende Struktur:

4 Die Daten- und Inode-Blöcke sowie Freiblocklisten oder -bitmaps müssen nicht zurückgeschrieben werden; somit ist das Dateisystem vielleicht nicht konsistent.

```
struct dentry {
   int d_count;                /* Nutzungszähler             */
   unsigned int d_flags;       /* Flags                      */
   struct inode  * d_inode;    /* zugehörige Inode           */
   struct dentry * d_parent;   /* übergeordnetes Verzeichnis */
   struct list_head d_vfsmnt;  /* Mountinformationen         */
   struct list_head d_hash;    /* Eintrag in die Hashliste   */
   struct list_head d_lru;     /* unbenutzte Einträge        */
   struct list_head d_child;   /* Liste der Kinder           */
   struct list_head d_subdirs; /* Unterverzeichnisse         */
   struct list_head d_alias;   /* Inode-Aliasliste           */
   struct qstr d_name;         /* Name der Datei             */
   unsigned long d_time;       /* Zeitstempel (Netzwerk-DS)  */
   struct dentry_operations  *d_op; /* Operationen           */
   struct super_block * d_sb;  /* zugehöriger Superblock     */
   unsigned long d_reftime;    /* letzte Zugriffszeit        */
   void * d_fsdata;            /* DS-spezifische Daten       */
   unsigned char d_iname[DNAME_INLINE_LEN]; /* Kurznamen     */
};
```

Der Name der Datei ist zusammen mit seiner Länge und seinem Hashwert in einer extra Struktur abgespeichert:

```
struct qstr {
   const unsigned char * name; /* der Name */
   unsigned int len;           /* Länge    */
   unsigned int hash;          /* Hashwert */
};
```

Der Verzeichniscache ist eine globale (Hash-)Liste, in der doppelt verkettete Listen eingetragen sind. Die Position der Unterliste in der globalen Hashliste bestimmt sich aus dem Hashwert des Namens und der Adresse des DEntry-Eintrags des übergeordneten Verzeichnisses.

```
static struct list_head *dentry_hashtable;
```

Ein neuer DEntry wird mit `d_alloc()` erzeugt. Die Funktion

```
struct dentry * d_alloc(struct dentry * parent,
                        const struct qstr *name);
```

bekommt den DEntry des übergeordneten Verzeichnisses und den Namen der aktuellen Datei übergeben. Der Kern alloziert Speicher für den neuen DEntry, trägt den übergebenen DEntry als `parent` ein, übernimmt den Superblock und trägt seine Child-Liste in die Subdirs-Liste des Vaters ein. Der Name wird eingetragen und der DEntry zurückgegeben. An dieser Stelle ist der DEntry noch „negativ", denn noch trägt er keine Inode-Informationen.

Den Rest übernimmt `d_add()`, die erst durch den Aufruf von

```
void d_instantiate(struct dentry *entry,
                   struct inode * inode);
```

den DEntry gültig macht, indem sie den Inode in den DEntry und den `i_dentry`-Eintrag des Inodes in die Alias-Liste einträgt und dann den DEntry mittels `d_rehash()` in seine Hashliste einhängt.

Die wichtigste Funktion ist `d_lookup()`. Sie wird benutzt, um Namen aufzulösen. Sie bekommt das Ausgangsverzeichnis als DEntry und den Namen in `qstr`-Form als Parameter. Sie durchsucht aber nur den bisher existierenden Cache; das Neuanlegen eines DEntry geschieht an anderer Stelle (siehe Seite 169).

```
struct dentry * d_lookup(struct dentry * parent,
                         struct qstr * name);
```

Die Funktion bestimmt die Liste in der globalen Hashtabelle, durchläuft sie und vergleicht zuerst den Hashwert und den Parent-Eintrag des gefundenen DEntrys mit dem aktuellen. Sollte der Parent eine Operation `d_compare` definieren, so wird der Name mit ihr verglichen, ansonsten ein schlichtes `memcmp()` bemüht. Ein Grund für die Definition einer eigenen Vergleichsoperation ist, das ein Dateisystem möglicherweise nicht zwischen Groß- und Kleinschreibung unterscheidet. Wurde der richtige Eintrag gefunden, werden die Statistiken aktualisiert (eigener Nutzungszähler, Anzahl der ungenutzten DEntrys bei Erstbenutzung) und der Eintrag wird zurückgegeben, ansonsten wird null zurückgeliefert.

6.2.5 DEntry-Operationen

Wie (mehr oder weniger) jede wichtige Struktur eines Dateisystems hat auch der DEntry seine eigenen Operationen. Mit ihnen können neue DEntries angelegt, verwaltet und auch wieder gelöscht werden.

```
struct dentry_operations {
   int (*d_revalidate)(struct dentry *, int);
   int (*d_hash) (struct dentry *, struct qstr *);
   int (*d_compare) (struct dentry *, struct qstr *,
                     struct qstr *);
   void (*d_delete)(struct dentry *);
   void (*d_release)(struct dentry *);
   void (*d_iput)(struct dentry *, struct inode *);
};
```

`d_revalidate(dentry, int)` dient bei Netzwerk-Dateisystemen dazu, die Information der lokalen DEntry-Kopie zu aktualisieren. `dentry` ist der aktuelle DEntry, in `int` stehen Flags, mit denen z. B. das Timeout-Verhalten gesteuert werden kann.

`d_hash(dentry, qstr)` berechnet aus der Adresse von `dentry` und dem Hashwert aus `qstr` die Position der Liste in der Hashtabelle. DEntry ist hier das übergeordnete Verzeichnis, da diese Funktion auch von Stellen aufgerufen wird, an denen der aktuelle DEntry negativ ist.

d_compare(dentry, qstr, qstr) vergleicht die zwei `qstr`-Einträge miteinander. Da hierbei Feinheiten wie die Groß-/Kleinschreibung berücksichtigt werden müssen, ist diese Funktion eine Angelegenheit des speziellen Dateisystems. DEntry ist auch hier das übergeordnete Verzeichnis, da diese Funktion auch von Stellen aufgerufen wird, an denen der aktuelle DEntry negativ ist.

d_delete(dentry, qstr) testet als Erstes den Nutzungszähler des DEntry. Ist `d_count` gleich eins, ist der Prozess der einzige Nutzer des DEntry und kann ihn abräumen. Dazu gehören das Freigeben der Inode-Struktur mittels `d_iput()` (falls definiert, ansonsten wird das allgemeine `iput` verwendet) und das Austragen aus der Alias-Liste. Damit ist der DEntry als negativ markiert. Dann wird mittels `d_drop()` der DEntry aus der Hashliste ausgetragen, so dass er bei `d_lookup()` nicht mehr gefunden wird.

d_release(dentry, qstr) gibt den Speicher eines DEntrys frei. Diese Funktion ist nur bei Dateisystemen implementiert, die in `d_fsdata` Daten ablegen.

d_iput(dentry, qstr) gibt die Informationen einer Inode frei. Diese Funktion ist nur bei Dateisystemen implementiert, bei denen die Inode noch in speziellen Listen verwaltet wird (HFS). Ist diese Funktion nicht implementiert, wird `iput()` aufgerufen.

Der Verzeichniscache dient also dazu, die dateisystemspezifische `lookup`-Funktion zu beschleunigen. Abschließend bleibt zu erwähnen, dass der Verzeichniscache gerade bei Systemen mit wenig Speicher eine Beschleunigung des Dateizugriffs erreicht. Auf Systemen mit viel Speicher wird dieser ohnehin für das Caching der Blockgeräte benutzt und hält demzufolge auch Verzeichnisse im Speicher.

6.2.6 Die Inode

Die Struktur `inode` hat dabei folgendes Aussehen:

```
struct inode {
   struct list_head i_hash;  /* Inode-Verkettungen          */
   struct list_head i_list;
   struct list_head i_dentry;

   unsigned long  i_ino;     /* Inode-Nummer                */
   unsigned int   i_count;   /* Referenzzähler              */
   kdev_t         i_dev;     /* Gerätenummer der Datei      */
   umode_t        i_mode;    /* Dateiart und Zugriffsrechte */
   nlink_t        i_nlink;   /* Anzahl der Hardlinks        */
   uid_t          i_uid;     /* Eigentümer                  */
   gid_t          i_gid;     /* Eigentümer                  */
   kdev_t         i_rdev;    /* Gerät bei Gerätedateien     */
   off_t          i_size;    /* Größe                       */
   time_t         i_atime;   /* Zeit des letzten Zugriffs   */
```

```
   time_t         i_mtime;   /* Zeit der letzten Änderung   */
   time_t         i_ctime;   /* Zeit der Erzeugung          */
   unsigned long  i_blksize; /* Blockgröße                  */
   unsigned long  i_blocks;  /* Blockanzahl                 */
   unsigned long  i_version; /* DCache-Versionsverwaltung   */
   struct semaphore i_sem;          /* Zugriffssteuerung    */
   struct semaphore i_zombie;       /* Zugriffssteuerung    */
   struct inode_operations *i_op;   /* Inode-Operationen    */
   struct file_operations  *i_fop;  /* Datei-Operationen    */
   struct super_block *i_sb;        /* Superblock           */
   struct wait_queue *i_wait;       /* Warteschlange        */
   struct file_lock *i_flock;       /* Dateisperren         */
   struct address_space *i_mapping; /* Speicherbereiche     */
   struct address_space i_data;
   struct dquot *i_dquot[MAXQUOTAS];
   struct pipe_inode_info *i_pipe;
   struct block_device *i_bdev;     /* zugeh. Blockgerät    */
   unsigned long i_state;      /* Status (DIRTY, …)        */
   unsigned int i_flags;       /* Flags                    */
   unsigned char i_sock;       /* Inode repräsentiert Socket*/

   atomic_t i_writecount;      /* Flag für Schreibzugriffe  */
   unsigned int i_attr_flags;
   __u32 i_generation;

   union {
      struct minix_inode_info minix_i;
               …
      void *generic_ip;
   } u;             /* dateisystemspezifische Informationen */
};
```

Diese Struktur enthält im Wesentlichen Informationen über die Datei. Zusätzlich befinden sich in dieser Struktur Verwaltungsinformationen sowie die dateisystemabhängige Union `u`.

Im Kern gibt es drei Aufbewahrungsorte, in denen die Inode gespeichert wird. Zum einen sind das zwei (doppelt verkettete) Listen — die eine speichert alle benutzen, die andere alle unbenutzen Inodes. Außerdem existiert eine Hashtabelle (`inode_hashtable`), die Listen mit allen genutzten Inode enthält. Mit einem Hashwert aus der Superblockadresse und Inode-Nummer kann die Liste adressiert werden, die die gesuchte Inode enthält.

Zu Statistikzwecken braucht der Kern natürlich ständig die Anzahl aller Inodes und aller freien Inodes. Dazu gibt es die Struktur `inodes_stat`:

```
struct {
        int nr_inodes;
        int nr_free_inodes;
        int dummy[5];
} inodes_stat = {0, 0,};
```

Für die Arbeit mit Inodes gibt es die Funktionen `iget()` und `iput()`. Sie dienen zum Erzeugen bzw. zum Freigeben von Inode-Strukturen.

```
static inline struct inode *iget(struct super_block *sb,
                                 unsigned long ino)
{
        return iget4(sb, ino, NULL, NULL);
}
```

Die Funktion `iget()` liefert die durch den Superblock `sb` und die Inode-Nummer `nr` angegebene Inode. Sie ist allerdings nur eine Kapsel für die Funktion `iget4()`. Dieser können noch eine Funktion und ein Parameter übergeben werden, die das Suchen der Inode zusätzlich steuern. Benutzt wird das beim NFS, um mit 64-Bit-Inodes umgehen zu können.

Die Funktion `iget4()` ruft ihrerseits `find_inode()` auf, die aus der übergebenen Liste (`head`) die Inode mit der richtigen Nummer heraussucht. Ist die gesuchte Inode in der Liste enthalten, wird einfach der Referenzzähler `i_count` erhöht.

Wurde sie nicht gefunden, wird mittels (`get_new_inode()`) eine „freie" Inode ausgewählt. Sie wird in beide Listen (alle und benutzte) eingetragen, und die Implementierung des entsprechenden Dateisystems veranlasst über die Superblockoperation `read_inode()`, die Inode mit Informationen zu füllen.

Eine mit `iget()` erhaltene Inode muss mit der Funktion `iput()` wieder „freigegeben" werden. Sie ruft die Funktion `put_inode` des Dateisystems auf und verringert den Referenzzähler um 1. Ist dieser anschließend null und gibt es keine Verweise auf die Inode, wird sie gelöscht. Gibt es noch Verweise, wird sie in die Liste der unbenutzten Inodes verschoben.

Die Verknüpfung eines Dateinamens mit seiner Inode erfolgt über den DEntry. DEntry und der aus ihnen bestehende Verzeichnis-Cache sind in Abschnitt 6.2.4 beschrieben.

6.2.7 Inode-Operationen

Auch die Inode-Struktur verfügt über ihre eigenen Operationen, die in der Struktur `inode_operations` untergebracht sind. Sie dienen hauptsächlich zur Verwaltung von Dateien. Diese Funktionen werden meist direkt in der Implementierung der entsprechenden Systemrufe aufgerufen. Fehlt eine der Inode-Operationen, führt die aufrufende Funktion Standardaktionen aus; häufig wird jedoch nur ein Fehler zurückgeliefert.

Da nicht alle Operationen für jeden Dateityp sinnvoll sind, haben die meisten Dateisystem-Implementierungen unterschiedliche Operationen definiert, z. B. gibt es spezielle für einfache Dateien oder für Verzeichnisse.

Funktionen, die früher für das Mappen von Dateien in den Speicher benutzt wurden, sind eine extra Unterstruktur der in der Inode verwalteten Adressräume (Speicherbereiche):

```
struct inode_operations {
   int (*create) (struct inode *,struct dentry *,int);
   struct dentry * (*lookup) (struct inode *,struct dentry *);
   int (*link) (struct dentry *,struct inode *,struct dentry *);
   int (*unlink) (struct inode *,struct dentry *);
   int (*symlink) (struct inode *,struct dentry *,const char *);
   int (*mkdir) (struct inode *,struct dentry *,int);
   int (*rmdir) (struct inode *,struct dentry *);
   int (*mknod) (struct inode *,struct dentry *,int,int);
   int (*rename) (struct inode *, struct dentry *,
                  struct inode *, struct dentry *);
   int (*readlink) (struct dentry *, char *,int);
   struct dentry * (*follow_link) (struct dentry *,
                                   struct nameidata *);
   void (*truncate) (struct inode *);
   int (*permission) (struct inode *, int);
   int (*revalidate) (struct dentry *);
   int (*setattr) (struct dentry *, struct iattr *);
   int (*getattr) (struct dentry *, struct iattr *);
};
```

`create(inode, dentry, int)` wird von der Funktion `open_namei()` des VFS aufgerufen. Diese Funktion erfüllt mehrere Aufgaben. Zuerst entnimmt sie mit Hilfe der Funktion `get_empty_inode()` der Liste aller Inodes eine freie Inode. Die Inode-Struktur muss jetzt dateisystemspezifisch gefüllt werden. Dazu wird zum Beispiel eine freie Inode auf dem Medium gesucht. Zusätzlich trägt `create()` den Dateinamen aus `dentry` in das Verzeichnis ein und füllt das Attribut `mode`. Fehlt `create()` in einer Dateisystemimplementation, wird vom VFS der Fehler `EACCES` zurückgegeben.

`lookup(inode, dentry)` sucht in dem Verzeichnis `inode` die Inode der Datei, deren Name in `dentry` steht. Wurde die Inode gefunden, wird sie mittels `d_add()` an den DEntry gebunden und null wird zurückgegeben.

`link((dentry, inode, dentry)` dient zum Anlegen eines Hardlinks. Im ersten DEntry steht die alte Datei, der Name der neuen Datei im zweiten, und in `inode` steht die Inode des Vaterverzeichnisses des zweiten DEntrys. Fehlt diese Funktion, liefert die aufrufende Funktion im VFS den Fehler `EPERM` zurück.

`unlink(inode, dentry)` löscht die in `dentry` angegebene Datei im durch die Inode `inode` angegebenen Verzeichnis. In der aufrufenden Funktion wird vorher sichergestellt, dass diese Operation die entsprechenden Berechtigungen besitzt. Das VFS liefert den Fehler `EPERM` zurück, falls `unlink()` nicht implementiert ist.

`symlink(inode,dentry, char)` richtet im Verzeichnis `inode` den symbolischen Link `char` ein. Bevor diese Funktion vom VFS aufgerufen wird, wurden bereits durch einen Aufruf von `permission()` die Zugriffsrechte überprüft. Fehlt `symlink()` in einer konkreten Implementierung, gibt das VFS den Fehler `EPERM` zurück.

mkdir(inode, dentry, int) entspricht dem Systemruf `mkdir()`, wobei `dentry` das Verzeichnis enthält, `inode` die Inode des übergeordneten Verzeichnisses und `int` die Zugriffsrechte. Die Funktion muss zunächst überprüfen, ob in dem Verzeichnis noch Unterverzeichnisse angelegt werden dürfen, danach eine freie Inode auf dem Datenträger sowie einen freien Block allozieren, in den dann das Verzeichnis mit den Standardeinträgen „`.`“ und „`..`“ geschrieben wird. Die Zugriffsrechte wurden bereits in der aufrufenden Funktion des VFS überprüft. Ist die Funktion `mkdir()` nicht implementiert, wird der Fehler `EPERM` zurückgegeben.

rmdir(inode, dentry) löscht das Unterverzeichnis `dentry` aus dem Verzeichnis `inode`. Die Funktion muss überprüfen, ob das zu löschende Verzeichnis leer ist und ob es momentan von einem Prozess benutzt wird bzw. ob der Prozess Eigentümer des Unterverzeichnisses ist, falls im Verzeichnis `dir` das Sticky-Bit gesetzt ist. Wie bei den zuvor beschriebenen Funktionen wurden vor dem Aufruf durch eine Funktion des VFS bereits die Zugriffsrechte überprüft. Ein fehlendes `rmdir()` wird vom VFS mit der Fehlermeldung `EPERM` quittiert.

mknod(inode, dentry, int, int) legt eine neue Inode mit dem Modus des ersten `int`-Parameters an. Diese Inode erhält den Namen aus `dentry`; in `inode` steht (wie üblich) das übergeordnete Verzeichnis. Falls es sich bei der Inode um eine Gerätedatei handelt (dann gilt `S_ISBLK(mode)` oder `S_ISCHR(mode)`), enthält der zweite `int`-Parameter die Nummer des Geräts. Fehlt diese Funktion, wird der Fehler `EPERM` zurückgeliefert.

rename(inode, dentry, inode, dentry) verschiebt eine Datei bzw. ändert ihren Namen. Die ersten beiden Parameter defineren die Quelle, die zweiten beiden das Ziel. Die aufrufende Funktion des VFS prüft zuvor die jeweiligen Zugriffsrechte in den Verzeichnissen. Fehlt diese Funktion, wird vom VFS der Fehler `EPERM` zurückgegeben.

readlink(dentry, buf, size) liest den symbolischen Links `dentry` aus und kopiert den Pfad der Datei, auf den der Link zeigt, in den Puffer. Bei eigenen Implementierungen ist zu beachten, das `buf` im Nutzeradreßraum liegt! Ist der Puffer zu klein, sollte der Pfadname einfach abgeschnitten werden. Handelt es sich bei der Inode nicht um einen symbolischen Link, sollte `EINVAL` zurückgegeben werden. Diese Funktion wird von `sys_readlink()` aufgerufen, nachdem die Zulässigkeit des Schreibzugriffs auf den Puffer `buf` überprüft und der DEntry mit `lnamei()` ermittelt wurde. Fehlt die Implementierung, liefert der Systemruf den Fehler `EINVAL` zurück.

follow_link(dentry, nameidata) löst einen symbolischen Link auf, indem die Funktion `nameidata` mit den Werten der Datei füllt, auf die der im ersten DEntry stehende Link zeigt. Um Endlosschleifen zu vermeiden[5], ist bei der Ausführung der Funktion `lookup_dentry()` die Anzahl der maximal aufzulösenden Links in Linux auf

5 Schließlich kann ein symbolischer Link auch auf einen weiteren symbolischen Link zeigen.

acht gesetzt. Diese Zahl ist in der Funktion `do_follow_link()`, die in `path_walk()` aufgerufen wird, „fest verdrahtet". Fehlt `follow_link()`, gibt die aufrufende Funktion gleichen Namens im VFS einfach `inode` zurück, als ob der Link auf sich selbst zeigen würde. Durch dieses Verhalten kann die VFS-Funktion stets aufgerufen werden, ohne dass getestet werden muss, ob die aktuelle Inode eine Datei oder einen symbolischen Link beschreibt.

`truncate(inode)` dient eigentlich zum „Verkürzen einer Datei", kann aber auch die Datei auf eine beliebige Länge vergrößern, falls dies von der konkreten Implementierung unterstützt wird. `truncate()` erhält als einzigen Parameter die Inode der zu ändernden Datei. Das Feld `i_size` wurde vor dem Aufruf der Funktion auf die neue Länge gesetzt. Die `truncate`-Funktion wird an mehreren Stellen im Kern verwendet, sowohl vom Systemruf `sys_truncate()` als auch beim Öffnen einer Datei. Sie muss auch die nicht mehr verwendeten Blöcke einer Datei freigeben.

`permission(inode, int)` überprüft anhand der Inode, ob die durch eine Maske angegebenen Zugriffsrechte für den aktuellen Prozess vorliegen. Fehlt die Funktion, überprüft die aufrufende Funktion des virtuellen Dateisystems die üblichen Unix-Rechte, wodurch eine Implementierung eigentlich überflüssig ist. Diese ist nur dann notwendig, wenn zusätzliche Zugriffsmechanismen implementiert werden sollen.

`revalidate(dentry)` frischt die Informationen einer Inode wieder auf. Diese Funktion wird von verteilten Dateisystemen (z. B. NFS) benutzt, denn dort ist es notwendig (z. B. bei einem `notify_change`), die Inode-Informationen auf einem konsistenten Stand zu bringen.

`setattr(dentry, iattr)` überträgt die Werte aus `iattr` in die zum DEntry gehörende Inode. Durch eine eigene Implementierung dieser Funktion kann jedes Dateisystem seine eigenen Attribute nutzen. Das VFS stellt eine Default-Implementation zur Verfügung.

`getattr(dentry, iattr)` überträgt die Werte aus der zum DEntry gehörenden Inode in `iattr`.

6.2.8 Die File-Struktur

In einem Multitaskingsystem tritt häufig das Problem auf, dass mehrere Prozesse gleichzeitig, sowohl lesend als auch schreibend, auf eine Datei zugreifen wollen. Aber auch ein einziger Prozess kann an unterschiedlichen Stellen einer Datei Daten lesen *und* schreiben. Um Probleme bei der Synchronisation zu verhindern und gemeinsame Zugriffe auf Dateien durch verschiedene Prozesse zu ermöglichen, wurde in Unix einfach eine weitere Struktur eingeführt.

Diese relativ einfache Struktur `file` enthält jeweils Informationen über die Zugriffsrechte `f_mode`, die aktuelle Dateiposition `f_pos`, die Art des Zugriffs `f_flags` und die Anzahl der Zugriffe `f_count`.

```
struct file {
   struct list_head f_list;      /* Verkettung                */
   struct dentry    *f_dentry;   /* DEntry-Eintrag            */
   struct vfsmount  *f_vfsmnt;   /* Mount-Daten               */
   struct file_operations *f_op; /* File-Operationen          */
   atomic_t         f_count;     /* Referenzzähler            */
   unsigned short   f_flags;     /* open()-Flags              */
   mode_t           f_mode;      /* Zugriffsart               */
   loff_t           f_pos;       /* Dateiposition             */
   unsigned long    f_reada, f_ramax, f_raend, f_ralen, f_rawin;
                                 /* Steuerungsinformationen   */
                                 /* für den Cache-Zugriff     */
   struct fown_struct f_owner;   /* Daten über den Besitzer   */
   unsigned int    f_uid, f_gid; /* Besitzer                  */
   int f_error;

   unsigned long   f_version;    /* DCache-Versionsverwaltung */

   void          *private_data;  /* Daten für u.a. Terminal-  */
                                 /* Treiber                   */
};
```

6.2.9 File-Operationen

Die Struktur `file_operations` ist die allgemeine Schnittstelle für die Arbeit mit Dateien. Sie enthält die Funktionen zum Öffnen, Schließen, Lesen und Schreiben von Dateien. Der Grund dafür, dass diese Funktionen nicht auch noch in den `inode_operations`, sondern in einer separaten Struktur gehalten werden, besteht darin, dass Sie an der Struktur `file` Änderungen vornehmen müssen. In der Struktur `inode_operations` enthält jede Inode zudem die Komponente `default_file_ops`, in der die Standard-Operationen für Dateien bereits festgelegt sind.

```
struct file_operations {
   struct module *owner;
   loff_t (*llseek) (struct file *, loff_t, int);
   ssize_t (*read) (struct file *, char *, size_t, loff_t *);
   ssize_t (*write) (struct file *, const char *, size_t,
                     loff_t *);
   int (*readdir) (struct file *, void *, filldir_t);
   unsigned int (*poll) (struct file *,
                         struct poll_table_struct *);
   int (*ioctl) (struct inode *, struct file *, unsigned int,
                 unsigned long);
   int (*mmap) (struct file *, struct vm_area_struct *);
   int (*open) (struct inode *, struct file *);
   int (*flush) (struct file *);
```

```
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *,
                      unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *,
                       unsigned long, loff_t *);
};
```

Diese Funktionen sind auch für Sockets und Gerätetreiber von Interesse, da sie die eigentliche Funktionalität der Sockets und Geräte beinhalten. Die Inode-Operationen dagegen arbeiten nur mit der Repräsentation des Sockets oder Geräts in dem entsprechenden Dateisystem bzw. dessen Darstellung im Speicher.

Die alten Funktionen, die für Media-Änderungen verantwortlich waren, sind zu den Operatioen für Blockgeräte verschoben worden.

`llseek(file, offset, origin)` Die Aufgabe der `llseek`-Funktion ist es, Positionierungen innerhalb der Datei vorzunehmen. Ist diese Funktion nicht implementiert, setzt die Standardimplementation (`default_llseek()`) einfach die Dateiposition `f_pos` der File-Struktur um.

`read(file, buf, count, ppos)` Diese Funktion liest `count` Bytes aus der Datei und kopiert sie in den Puffer `buf` im Nutzeradressraum. Vorher testet das VFS, ob der Puffer `buf` vollständig im Nutzeradressraum liegt und beschrieben werden kann sowie ob der Dateizeiger gültig ist und die Datei zum Lesen geöffnet war. `ppos` zeigt dabei auf die aktuelle Dateiposition. Die `read`-Funktion sollte die Dateiposition anpassen, wenn die Arbeit mit Dateipositionen unterstützt wird. Gerätetreiber ignorieren diese meist, so dass die Dateiposition hier stets null ist. Ist keine Read-Funktion implementiert, wird der Fehler `EINVAL` zurückgegeben.

`write(file, buf, count, ppos)` Die `write`-Funktion arbeitet äquivalent zu `read()` und kopiert Daten aus dem Nutzeradressraum in die Datei.

`readdir(file, buf, callback)` Diese Funktion liefert den nächsten Verzeichniseintrag in der Struktur `dirent` zurück — oder die Fehler `ENOTDIR` bzw. `EBADF`. Ist diese Funktion nicht implementiert, liefert das virtuelle Dateisystem `ENOTDIR` zurück. Die Implementation ist nur für Verzeichniseinträge notwendig. Der Callback ist notwendig, da die Funktion sowohl vom Systemruf *readdir* als auch von *getdents* benutzt wird, die unterschiedliche Ausgabeformate besitzen.

```
struct old_linux_dirent {
    unsigned long  d_ino;
    unsigned long  d_offset;
    unsigned short d_namlen;
    char           d_name[1];
```

```
};

struct readdir_callback {
        struct old_linux_dirent * dirent;
        int count;
};

 /* __buf wird als readdir_callback interpretiert */
int fillonedir(void * __buf, const char * name, int namlen,
               off_t offset, ino_t ino);

int old_readdir(…)
{
   struct readdir_callback buf;
   …
   error = file->f_op->readdir(file, &buf, fillonedir);
   …
}
```

Die zweite Variante sieht so aus:

```
struct linux_dirent {
   unsigned long  d_ino;
   unsigned long  d_off;
   unsigned short d_reclen;
   char           d_name[1];
};

struct getdents_callback {
   struct linux_dirent * current_dir;
   struct linux_dirent * previous;
   int count;
   int error;
};

 /* __buf wird als getdents_callback interpretiert */
int filldir(void * __buf, const char * name, int namlen,
            off_t offset, ino_t ino);

int sys_getdents(…)
{
   struct getdents_callback buf;
   …
   error = file->f_op->readdir(file, &buf, filldir);
   …
}
```

In beiden Fällen muss sich die `read_dir`-Funktion nicht für den Parameter `buf` interessieren; er wird lediglich an die Callback-Funktion `callback` weitergereicht. `readdir()` muss vielmehr dafür sorgen, dass der Name, die Länge des Namens, die Dateiposition

des aktuellen Verzeichniseintrags sowie die Inode-Nummer des Eintrags an die Callback-Funktion weitergereicht werden.

`poll(file, poll_tbl)` Diese Funktion überprüft, ob Daten von einer Datei gelesen oder in eine Datei geschrieben werden können. Zusätzlich kann man noch testen, ob Ausnahmebedingungen vorliegen. Diese Funktion ist nur für Gerätetreiber und Sockets sinnvoll. Eine tiefergehende Betrachtung der `poll`-Funktion finden Sie in Abschnitt 7.4.8.

`ioctl(inode, file, cmd, arg)` Die Funktion `ioctl()` dient im eigentlichen Sinne zur Einstellung von gerätespezifischen Parametern. Der Systemruf *ioctl* fragt vor der Weiterleitung an die Funktion noch die folgenden Standardargumente ab:

`FIONCLEX` löscht das Close-on-exec-Bit.

`FIOCLEX` setzt das Close-on-exec-Bit.

`FIONBIO` Falls das zusätzliche Argument `arg` auf einen Wert ungleich 0 verweist, wird das Flag `O_NONBLOCK` gesetzt, ansonsten gelöscht.

`FIOASYNC` Analog zu `FIONBIO` wird das Flag `O_SYNC` gelöscht oder gesetzt. Außerdem wird die Funtion `fasync()` aufgerufen.

Befand sich `cmd` nicht unter diesen Werten, wird getestet, ob `file` auf eine reguläre Datei verweist. Falls ja, wird die Funktion `file_ioctl()` aufgerufen und der Systemruf ist beendet. Für nichtreguläre Dateien testet das VFS ab, ob eine `ioctl`-Funktion vorhanden ist. Fehlt sie, wird der Fehler `ENOTTY` zurückgegeben; ansonsten wird die dateispezifische `ioctl`-Funktion aufgerufen. Die Funktion `file_ioctl()` kennt die folgenden Kommandos:

`FIBMAP` erwartet im Argument `arg` einen Zeiger auf eine Blocknummer und liefert die logische Blocknummer dieses Blocks der Datei auf dem Gerät zurück, falls die zur Datei gehörende Inode eine `get_block`-Funktion besitzt. Diese logische Nummer wird auf die Adresse `arg` zurückgeschrieben. Fehlen die Inode-Operationen, wird ein Fehler `EBADF` generiert, fehlt `get_block()`, wird `EINVAL` zurückgeliefert.

`FIGETBSZ` liefert die Blockgröße des Dateisystems zurück, in dem sich die Datei befindet. Sie wird in die Adresse `arg` geschrieben, falls der Datei ein Superblock zugeordnet ist. Sonst wird der Fehler `EBADF` zurückgeliefert.

`FIONREAD` schreibt die Anzahl der noch nicht gelesenen Bytes innerhalb der Datei in die Adresse `arg`.

Da alle diese Kommandos Daten in den Nutzeradressraum schreiben, wird die Erlaubnis dazu stets mit Hilfe der Funktion `verify_area()` eingeholt und möglicherweise ein Zugriffsfehler zurückgeliefert. Befand sich das Kommando `cmd` nicht unter den beschriebenen Werten, wird auch von `file_ioctl()` die dateispezifische `ioctl`-Funktion aufgerufen. Existiert diese nicht, wird der Fehler `ENOTTY` zurückgeliefert.

`mmap(file, vm_area)` Diese Funktion bildet einen Teil einer Datei in den Nutzeradressraum des aktuellen Prozesses ab. Die übergebene Struktur `vm_area` beschreibt alle Eigenschaften des einzublendenden Speicherbereichs. Die Komponenten `vm_start` und `vm_end` beschreiben die Start- und die Endadresse des Speicherbereichs, in den die Datei abgebildet werden soll, `vm_offset` die Position der Datei, ab der die Abbildung erfolgen soll. Eine weitergehende Beschreibung des Mmap-Mechanismus finden Sie in Abschnitt 4.2.2.

`open(inode, file)` Diese Funktion ist nur in zwei Fällen sinnvoll, da die Standardfunktion des Virtuellen Dateisystems alle notwendigen Aktionen für Dateien wie beispielsweise die Allozierung der File-Struktur vornimmt. Der eine Fall betrifft Gerätetreiber und der zweite das Öffnen von Dateien auf 32-Bit-Systemen.

`flush(file)` Die `flush`-Funktion wird vom VFS aufgerufen, wenn mit Hilfe des Systemrufes *close* eine Datei geschlossen wird. Damit wird gewährleistet, dass eventuell noch gepufferte Daten geschrieben werden. Fehlt dieser Funktion, übernimmt das VFS keine weitere Aktion, allerdings kann `flush` einen Fehlercode zurückliefern, der dann als Fehlercode des *close*-Systemrufs zurückgeliefert wird.

`release(inode, file)` Diese Funktion wird bei der Freigabe der File-Struktur aufgerufen, d.h. wenn ihr Referenzzähler `f_count` gleich 0 ist. Diese Funktion ist primär für Gerätetreiber gedacht, ein Fehlen wird vom virtuellen Dateisystem ignoriert. Die Aktualisierung der Inode wird ebenfalls automatisch vom Virtuellen Dateisystem vorgenommen.

`fsync(file, dentry)` Die Funktion `fsync()` muss dafür sorgen, dass alle Puffer der Datei aktualisiert und auf das Gerät zurückgeschrieben werden, deshalb ist diese Funktion nur für Dateisysteme relevant. Implementiert ein Dateisystem keine `fsync`-Funktion, wird `EINVAL` zurückgegeben.

`fasync(fd, file, on)` Diese Funktion ist in keinem Dateisystem implementiert, sondern wird von Gerätetreibern (siehe Abschnitt 7.4.11) und Sockets benutzt. Mit ihr kann sich ein Prozess über asynchron eintreffene Daten benachrichtigen lassen.

`lock(file, op, file_lock)` `lock` wird aufgerufen, wenn mittels des Systemrufs *fnctl* Dateisperren gesetzt oder gelesen werden. Ist `lock` nicht implementiert, führt das VFS die Standardaktionen `posix_test_lock()` und `posix_lock_file()` aus (siehe Abschnitt 5.2.1).

`readv(file, iovec, count, offset)` Diese Funktion liest Daten aus einer Datei. Im Unterschied zur normalen `read`-Funktion landen die Daten nicht in einem Puffer, sondern in einer Anzahl von I/O-Vektoren. Beim Füllen der Vektoren wird nacheinander aus der Datei gelesen.

```
struct iovec
{
   void *iov_base;          /* Adresse  */
   __kernel_size_t iov_len; /* Größe    */
};
```

`writev(file, iovec, count, offset)` Diese Funktion schreibt den Inhalt der übergebenen Vektoren nacheinander in die Datei `file`.

6.2.10 Das Öffnen einer Datei

Eine der wichtigsten Operationen beim Zugriff auf Daten ist das Öffnen von Dateien mit dem Systemruf *open*. Dabei hat das System nicht nur entsprechende Vorbereitungen zu treffen, die einen reibungslosen Zugriff auf Daten gewährleisten, sondern auch die Rechte des Prozesses zu überprüfen. Hier ist außerdem die eigentliche Schaltfunktion des Virtuellen Dateisystems implementiert, das zwischen den konkreten Dateisystemimplementationen und speziellen Geräten vermittelt.

Wurde die Funktion `sys_open()` aufgerufen, holt sie sich mit `get_unused_fd()` einen freien Dateideskriptor und ruft als Nächstes `filp_open()` mit den übergebenen Parametern auf. Diese Funktion erzeugt mit der Funktion `get_empty_filp()` eine neue File-Struktur und trägt sie in die Dateideskriptortabelle des Prozesses ein. In dieser Struktur werden die Felder `f_flags` und `f_mode` besetzt und die Funktion `open_namei()` wird aufgerufen, um die Inode der zu öffnenden Datei zu erhalten. Vor dem Aufruf dieser Funktion werden noch die `open()`-Flags geändert, so dass die beiden unteren Bits nun die Funktion von Zugriffsrechten besitzen. Bit 0 steht für das Lesen und Bit 1 für das Schreiben der Datei. Der Vorteil dieser Darstellung des Zugriffs auf die Datei liegt klar auf der Hand: Das Abfragen der Rechte ist damit durch einfache Befehle möglich. Die Funktion `open_namei()` überlagert die übergebenen Rechte mit der prozesseigenen Umask und nimmt die Namensauflösung vor. Dazu werden die Struktur `nameidata` sowie die Funktionen `path_init()` und `path_walk()` benutzt. Die Struktur hat folgendes Aussehen:

`open()`-Flags	Wert	Bit 1 & 0	des `open_namei()`-Flags
		00	keine Rechte erforderlich (symbolische Links)
`O_RDONLY`	0	01	Leserecht erforderlich
`O_WRONLY`	1	10	Schreibrechte erforderlich
`O_RDWR`	2	11	Schreib- und Leserechte erforderlich
`O_CREAT`	1000	1*	Schreibrechte erforderlich
`O_TRUNC`	2000	1*	Schreibrechte erforderlich

Tabelle 6.4: Die Umrechnung der `open()`-Flags

```
struct nameidata {
   struct dentry *dentry; /* DEntry der Datei         */
   struct vfsmount *mnt;  /* Mount-Daten              */
   struct qstr last;      /* qstr-Struktur des Namens */
   unsigned int flags;    /* Flags                    */
   int last_type;         /* Typ des letzten Parents  */
};
```

Die Funktion `path_init()` hat folgende Definition:

```
int path_init(const char *name,unsigned int flags,
        struct nameidata *nd);
```

Sie trägt die übergebenen Flags in `nd` ein. Mögliche Flags sind:

`LOOKUP_FOLLOW` — Links werden aufgelöst.

`LOOKUP_CONTINUE` — Flag für den internen Gebrauch. Es wird beim NFS verwendet, um kleinere Zeitunterschiede zu ignorieren.

`LOOKUP_POSITIVE` — Wenn bei der stufenweisen Auswertung keine Inode für den DEntry gefunden wird und dieses Flag gesetzt ist, wird ein Fehler (anstatt 0) zurückgegeben.

`LOOKUP_DIRECTORY` — Die aktuelle Pfadkomponente ist ein Verzeichnis.

`LOOKUP_PARENT` — Der Typ des Vaterverzeichnisses wird in die `nd`-Struktur eingetragen. Es sind fünf Typen möglich; bei der Pfadauswertung werden nur die ersten drei verwendet.

- **`LAST_DOT`** — **`.`** ist das Vaterverzeichnis.
- **`LAST_DOTDOT`** — **`..`** ist das Vaterverzeichnis.
- **`LAST_NORM`** — **`.`** ist das Vaterverzeichnis.
- **`LAST_ROOT`** — wird als Standard bei `path_init()` eingetragen.
- **`LAST_BIND`** — wird vom *Proc*-Dateisystem eingetragen und bewirkt die korrekte Auflösung der dortigen Links.

`LOOKUP_NOALT` — Wenn das Flag gesetzt ist, wird bei `path_init()` nicht `fs->root`, sondern `fs->altroot` verwendet (wenn es ungleich `NULL` ist).

Außerdem belegt die Funktion `path_init()` die Werte `mnt` und `dentry`. Ist der übergebene Name ein absoluter Pfadname, werden die Root-Werte des aktuellen Prozesses verwendet (`current->fs->rootmnt` und `current->fs->root`), ansonsten die des aktuellen Verzeichnisses (`current->fs->pwdmnt` und `current->fs->pwd`). Jetzt haben wir einen Startpunkt für die Namensauflösung.

Die Funktion `path_walk()` übernimmt die eigentliche Namensauflösung. Sie ersetzt die alten Funktionen `dir_namei()`, `_namei()`, `d follow_link()` und `lookup_dentry()`. In ihr wird der DEntry für `name` aufgebaut, falls er noch nicht existiert, und zurückgeliefert. Sie hat folgende Aufrufkonvention:

```
int path_walk(const char * name, struct nameidata *nd);
```

Dabei ist `name` der Name der Datei und `nd` die aus `path_init()` initialisierte Struktur.

Jetzt beginnt eine Endlosschleife, in der Stück für Stück versucht wird, den Namen aufzulösen. Der Name wird verzeichnisweise zerlegt, nämlich in durch `'/'` getrennte Stücke. Zuerst wird für das nächste Stück eine `qstr`-Struktur aufgebaut.

Wenn wir schon am letzten Teil des Namens angekommen sind, dann wird in einen extra Zweig gesprungen, der das gleiche wie die Anweisungen in der Schleife leistet, aber nur einmal durchlaufen wird. Endet der Name mit einem `'/'`, so wird noch `do_follow_link()` aufgerufen, und es wird der Fehler `-ENOTDIR` zurückgegeben, falls der gefundene DEntry eine Datei ist.

Jetzt beginnt die eigentliche Suche. Ist das aktuelle Namensstück gleich **..**, dann wird das Parent-Verzeichnis ermittelt und mit dem nächsten Namensstück fortgefahren.

Ist für `dentry` eine eigene DEntry-Operation `d_hash` implementiert, wird mit dieser jetzt ein neuer Hashwert berechnet.

Jetzt wird mittels `cached_lookup()` der Verzeichniscache durchsucht. Liefert diese Funktion kein gültiges Ergebnis, wird `real_lookup()` aufgerufen, und die Arbeit muss zu Fuß von der Implementation des jeweiligen Dateisystems erledigt werden. Das aktuelle Verzeichnis wird gesperrt, es wird noch einmal in den Cache geschaut (für das Warten auf die Sperre), mit `d_alloc()` wird ein neuer DEntry angelegt und mit der Inode-Operation `lookup()` gefüllt (diese ruft auch `d_add()` auf).

Nun haben wir einen gültigen DEntry, allerdings sind noch zwei Dinge zu tun: Wir müssen uns um Mount-Points und um Links kümmern. Mountpoints werden rekursiv aufgelöst, solange der DEntry ein Mountpoint ist und die Funktion `__follow_down()` einen Wert ungleich 1 liefert. Links werden verfolgt, indem entweder (wenn das Flag `LOOKUP_FOLLOW` gesetzt ist und es eine Inode-Operation `follow_link()` gibt) in der Funktion `do_follow_link()` die Inode-Operation aufgerufen und das Ergebnis zurückgegeben wird oder (ansonsten) der aktuelle DEntry zurückgeliefert wird. Unabhängig davon, welcher der beiden Fälle zutrifft, das Ergebnis wird auf jeden Fall `base` zugewiesen, und wir sind damit einen (Verzeichnis-)Schritt weiter. Jetzt wird die Schleife erneut durchlaufen.

Jetzt sind wir wieder auf der Ebene von `open_namei()` und führen eine Reihe von Tests durch. Zum Schluss werden noch folgende Aktionen durchgeführt:

- Hat der DEntry jetzt eine Inode? Wenn nicht, liefert der Aufruf den Fehler `ENOENT`.
- Ist die Inode ein symbolischer Link? Wenn ja, dann haben wir eine Link-Schleife und liefern `ELOOP`.
- Gehört die Inode zu einem Verzeichnis, und haben wir in den Flags den Schreibzugriff gesetzt? Wenn ja, gibt den Fehler `EISDIR` zurück.
- Überprüfe den übergebenen Mode.

- Ist die Datei ein FIFO, ein Socket? Wenn ja, dann erlaube Schreibzugriff (lösche das Flag `O_TRUNC`), auch wenn es ein Read-Only-Dateisystem ist, ansonsten gib `EROFS` zurück.
- Ist `O_TRUNC` (noch) gesetzt? Wenn ja, kürze die Datei, es sei denn, wir haben keinen Schreibzugriff (`get_write_access()`) oder es existieren Sperren (`locks_verify_locked()`) auf der Datei.

Ist all dies überstanden, wird der DEntry zurückgegeben, und wir sind wieder in `filp_open()`. Jetzt wird die File-Struktur in `dentry_open()` gefüllt: Der DEntry wird eingetragen, `f_pos` mit 0 initialisiert, und die Standard-File-Operationen der Inode werden als File-Operation eingetragen. Ist dort eine `open()`-Funktion definiert, wird diese noch aufgerufen.

Handelt es sich bei der geöffneten Datei um die Datei eines zeichenorientierten Geräts, wird an dieser Stelle die Funktion `chrdev_open()` aufgerufen, die wiederum die File-Operationen in Abhängigkeit von der Major- und Minor-Nummer des Geräts ändert:

```
int chrdev_open(struct inode * inode, struct file * filp)
{
   int ret = -ENODEV;

   filp->f_op = get_chrfops(MAJOR(inode->i_rdev),
                            MINOR(inode->i_rdev));
   if (filp->f_op != NULL){
      ret = 0;
      if (filp->f_op->open != NULL)
         ret = filp->f_op->open(inode,filp);
   }
   return ret;
}
```

Die File-Operationen der Gerätetreiber befinden sich in der Tabelle `chrdevs[]` und wurden dort mit der Funktion `register_chrdev()` (siehe Kapitel 7) bei der Initialisierung des Treibers eingetragen. Die Minor-Nummer wird an dieser Stelle benötigt, um bei der Nachfrage nach seriellen Geräten das Modul nachladen zu können, vorausgesetzt natürlich, bei der Übersetzung des Kernes wurde der Kernel-Modullader konfiguriert. Die Funktion `open()` des Gerätetreibers wird sicherlich in Abhängigkeit von der Minor-Nummer des Geräts weitere File-Operationen eintragen. Sie wird im nächsten Kapitel beschrieben.

Der File-Pointer wird zurückgegeben, und wir sind wieder in `sys_open()`. Nun wird noch mittels `fd_install()` der File-Pointer mit dem Deskriptor verknüpft. War auch das erfolgreich, ist der Aufruf beendet und liefert den Deskriptor.

6.3 Das *Ext2*-Dateisystem

Die Enwicklung von Linux fand in den Anfängen unter Minix statt, so verwundert es nicht, dass das erste Linux-Dateisystem das Minix-Dateisystem war. Durch seine Be-

schränkung auf Partitionen mit maximal 64 MByte und eine Dateinamenlänge von maximal 14 Zeichen entstand die Suche nach einem besseren Dateisystem. So gab es ab April 1992 mit dem *Ext*-Dateisystem das erste eigens für Linux entwickelte Dateisystem. Es ließ zwar nun Partitionen bis 2 GByte und Dateinamen bis zu 255 Zeichen zu, befriedigte aber bei weitem nicht die Linux-Gemeinde, da es langsamer als das Minix-Dateisystem war und die einfache Freiblockverwaltung zu einer hohen Fragmentierung des Dateisystems führte. Ein heute weniger verbreitetes Linux-Dateisystem wurde im Januar 1993 von Frank Xia vorgestellt — das *Xia*-Dateisystem. Es basiert ebenfalls auf dem Minix-Dateisystem und lässt Partitionsgrößen bis 2 GByte sowie Dateinamen bis 248 Zeichen zu. Durch die Verwaltung der freien Blöcke in Bitmaps und optimierender Blockallokierungsfunktionen ist es zudem schneller und robuster als das *Ext*-Dateisystem.

Ungefähr zeitgleich stellten Rémy Card, Wayne Davidson und andere das *Ext2*-Dateisystem als die Weiterentwicklung des *Ext*-Dateisystems vor. Es kann mittlerweile als das Linux-Dateisystem schlechthin betrachtet werden, da es bei den meisten Linux-Systemen und -Distributionen Verwendung findet.

6.3.1 Der Aufbau des *Ext2*-Dateisystems

Das Design des *Ext2*-Dateisystems wurde wesentlich durch das *Fast Filesystem* von BSD (BSD-FFS) beeinflusst. So ist eine Partition ähnlich wie den Zylindergruppen des FFS in mehrere *Blockgruppen* unterteilt. Dabei enthält jede Blockgruppe, wie in Abbildung 6.4 dargestellt ist, eine Kopie des Superblocks sowie Inode- und Datenblöcke. Durch die Blockgruppen wird nun versucht,

- Datenblöcke in der Nähe ihrer Inode und
- Datei-Inodes in der Nähe ihrer Verzeichnis-Inode

zu halten, den Positionierungsaufwand auf dem Medium zu verringern und somit den Zugriff auf Daten zu beschleunigen. Zudem enthält jede Blockgruppe den Superblock sowie die Informationen über alle Blockgruppen, mit denen im Notfall das Dateisystem restauriert werden kann.

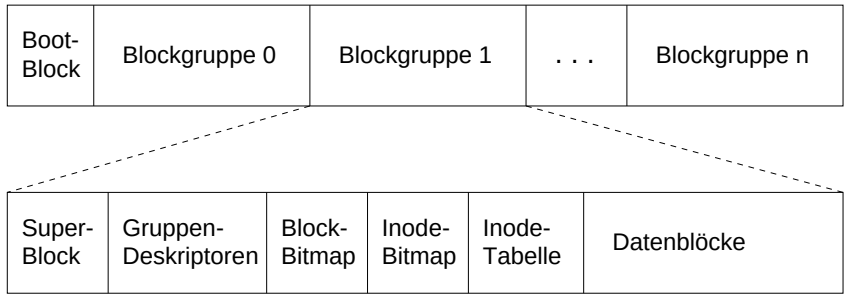


Abbildung 6.4: Der Aufbau des Ext2*-Dateisystems.*

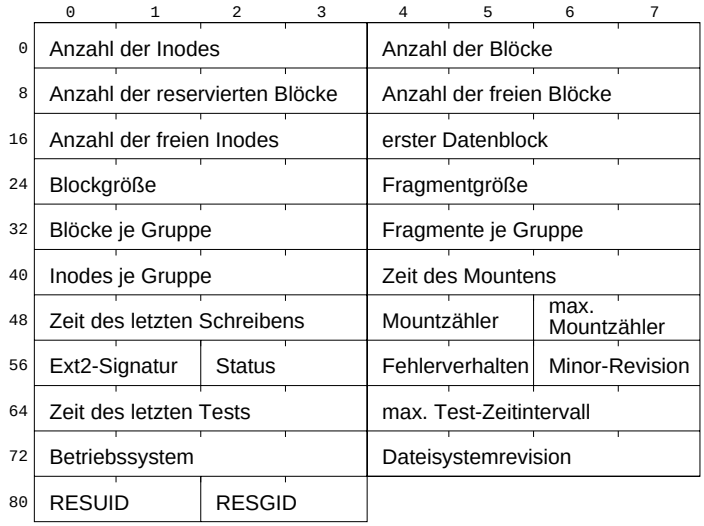


Abbildung 6.5: Der Superblock des Ext2*-Dateisystems. Er wird durch Füllbytes auf die Größe von 1.024 Byte gebracht.*

Den physischen Superblock — definiert als Struktur `ext2_super_block` — stellt Abbildung 6.5 dar. Er enthält die Informationen über das Dateisystem, z. B. die Anzahl der Inodes und Blöcke. Die verwendete Blockgröße ist nicht direkt, sondern als dualer Logarithmus der Blockgröße minus den der minimalen, durch das *Ext2*-Dateisystem unterstützten Blockgröße — im Normalfall also 0 — abgelegt. Zur Benutzung muss die minimale Blockgröße `EXT2_MIN_BLOCK_SIZE` nur um den angegebenen Wert „geshiftet" werden. Des Weiteren sind im Superblock die Informationen über die Anzahl der Inodes und Blöcke je Blockgruppe sowie die Zeiten des letzten Mountens, Schreibens des Superblocks und des letzten Dateisystemtests untergebracht. Der Superblock hält auch die Informationen über das Verhalten des Dateisystems beim Auftreten von Fehlern, das maximale Zeitintervall bis zum nächsten Dateisystemtest, einen Mount-Zähler und die maximale Mount-Anzahl, die darüber Aufschluss gibt, wann auf jeden Fall ein Dateisystemtest durchgeführt werden soll. Die Werte *resuid* und *resgid* geben über den Nutzer bzw. die Gruppenzugehörigkeit Aufschluss, die außer dem Superuser die reservierten Blöcke nutzen dürfen.

Der Superblock wird mit Füllbytes auf eine Größe von 1.024 Byte — die minimale Blockgröße `EXT2_MIN_BLOCK_SIZE` — gebracht. Es ist somit einfach möglich, diesen Raum für Erweiterungen zu nutzen und den Superblock mit `bread()` einzulesen.

Nach dem Superblock folgen, in jeder Blockgruppe, die *Blockgruppendeskriptoren*. Sie enthalten die Informationen über die Blockgruppen. Jede Blockgruppe wird durch einen 32 Byte großen Blockgruppendeskriptor beschrieben (siehe Abbildung 6.6). Er enthält die Blocknummern der Inode-Bitmap, der Blockbitmap und der Inode-Tabelle, die Anzahl der freien Inodes und Blöcke sowie die Anzahl der Verzeichnisse in dieser Blockgruppe. Die Anzahl der Verzeichnisse dient dem Algorithmus zur Inode-Allokation von Verzeichnissen. Dabei wird versucht, Verzeichnisse möglichst gleichmäßig über die Blockgruppen

zu verteilen, d.h. ein neues Verzeichnis wird in der Blockgruppe mit der kleinsten Verzeichnisanzahl angelegt.

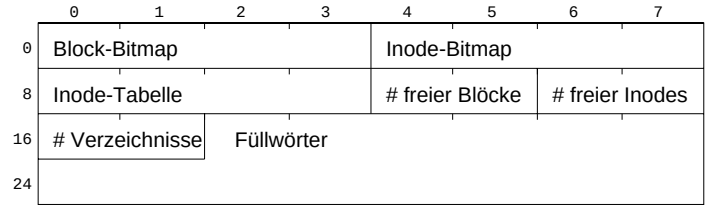


Abbildung 6.6: Die Blockgruppendeskriptoren des Ext2*-Dateisystems.*

Die Bitmaps haben jeweils die Größe eines Blocks. Die Größe der Blockgruppe ist, bei einer Blockgröße von 1.024 Byte, demnach auf 8.192 Blöcke beschränkt.

Die Inode-Tabelle einer Blockgruppe belegt aufeinander folgende Blöcke, beginnend mit dem angegebenen. Dort befinden sich die 128 Byte großen Inodes (siehe Abbildung 6.7). Sie enthalten außer den bekannten Informationen, die Zeit des Löschens der Datei (dient zur Implementierung der Restaurierung gelöschter Dateien), Einträge für ACLs (*Access Control Lists* für eine feinere Differenzierung von Zugriffsberechtigungen) sowie betriebssystemabhängige Informationen. Momentan sind ACLs noch nicht implementiert, so dass die Funktion `ext2_permission()` nur die UNIX-Rechte sowie das Flag `S_IMMUTABLE` testet.

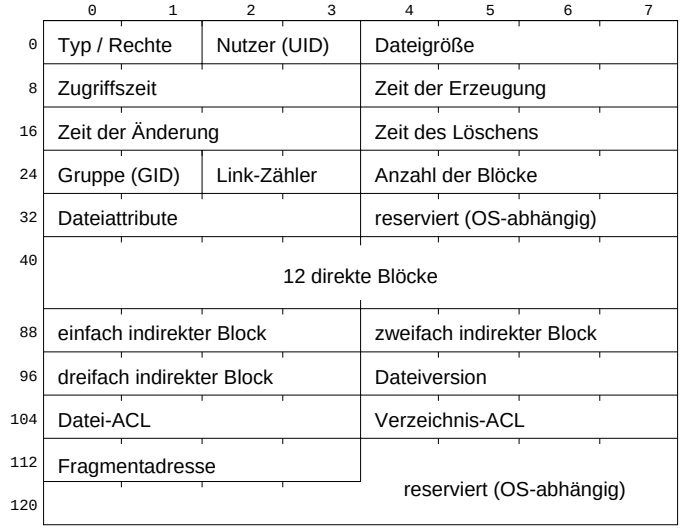


Abbildung 6.7: Die Inode des Ext2*-Dateisystems*

Handelt es sich bei der Inode um eine Gerätedatei, d.h. ist `S_IFCHR` oder `S_IFBLK` in `i_mode` gesetzt, so entspricht die erste Blocknummer (`i_block[0]`) der Gerätenummer. Bei einem kurzen symbolischen Link (`S_IFLNK`) enthalten die Blocknummern

den Pfad, so dass kein zusätzlicher Datenblock benötigt wird. In diesem Fall enthält die Blockanzahl, also `i_blocks`, den Wert 0. Ist der symbolische Link länger als

```
EXT2_N_BLOCKS * sizeof (long)
```

wird er im ersten Block abgelegt. Auf diese Weise ist die maximale Länge eines Verweises auf die Blockgröße beschränkt.

6.3.2 Verzeichnisse im *Ext2*-Dateisystem

Verzeichnisse werden im *Ext2*-Dateisystem mit Hilfe einer einfach verketteten Liste verwaltet. Jeder Eintrag hat dabei folgende Struktur:

```
struct ext2_dir_entry_2 {
   __u32 inode;    /* Inode-Nummer                    */
   __u16 rec_len;  /* Länge des Verzeichniseintrags   */
   __u8  name_len; /* Länge des Dateinamens           */
   __u8  file_type; /* Dateityp                       */
   char  name[EXT2_NAME_LEN];  /* Dateiname           */

};
```

Das Feld `rec_len` enthält die Länge des aktuellen Eintrags. Sie wird stets auf ein Vielfaches von 4 aufgerundet. Mit seiner Hilfe lässt sich also der Beginn des nächsten Eintrags berechnen. Das Feld `name_len` enthält die Länge des Dateinamens. Ein Verzeichniseintrag kann durchaus länger sein, als erforderlich ist, um den Dateinamen zu speichern. In `file_type` ist der Dateityp registiert, d.h. es ist festgelegt, ob es sich um eine Gerätedatei oder eine Pipe handelt. Ein möglicher Aufbau wird durch die Abbildung 6.8 illustriert.

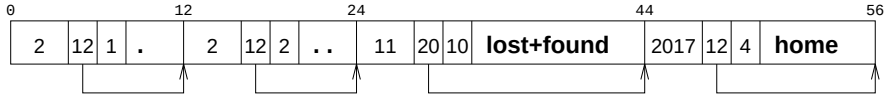


Abbildung 6.8: Ein Verzeichnis des Ext2*-Dateisystems*

Ein Eintrag wird gelöscht, indem die Inode-Nummer auf null gesetzt wird und aus der Kette ausgetragen wird, d.h. der vorherige Verzeichniseintrag verlängert sich einfach. Dadurch sind keine Verschiebeoperationen innerhalb des Verzeichnisses notwendig, die sich sonst auch über Puffergrenzen hinweg auswirken könnten. Der „verschenkte Platz" geht jedoch nicht verloren, sondern wird beim Anlegen eines neuen Eintrags wiederverwertet, entweder durch Überschreiben eines Eintrags mit der Inode-Nummer 0 oder durch Verwendung des zusätzlichen Platzes, der durch das Auslinken entstanden ist.

6.3.3 Blockallokation im *Ext2*-Dateisystem

Ein geläufiges Problem aller Dateisysteme ist die Fragmentierung von Dateien. Darunter versteht man die „Zersplitterung" der Datenblöcke der Dateien, die durch das ständige

Löschen und Erzeugen von neuen Dateien entsteht. Zur Lösung dieses Problems dienen meist „Defragmentierprogramme“, wie `defrag` für Linux. Einige Dateisysteme versuchen durch eine geschickte Blockallokation die Fragmentierung von Dateien weitgehend zu unterbinden. Auch das *Ext2*-Dateisystem benutzt zwei spezielle Algorithmen, um die Fragmentierung von Dateien einzuschränken.

Zielorientierte Allokation Neue Datenblöcke werden stets in der Nähe eines „Zielblocks“ gesucht. Falls dieser Block frei ist, wird er alloziert. Sonst wird versucht, innerhalb eines Bereichs von 32 Blöcken einen freien Block zu finden und zu allozieren. Schlägt auch dies fehl, versucht die Blockallokationsroutine wenigstens einen Block in der Blockgruppe des Zielblockes zu finden. Erst danach werden alle anderen Blockgruppen durchsucht.

Preallokation Wurde ein freier Block gefunden, wird eine Anzahl folgender Blocks vorgemerkt (falls sie frei sind). Die Anzahl kann im Ext2-Superblock eingetragen sein, ansonsten werden `EXT2_DEFAULT_PREALLOC_BLOCKS` (8) Blöcke reserviert. Wird die Datei geschlossen, werden die restlichen noch vorgemerkten Blöcke wieder freigegeben. Dies sorgt zusätzlich dafür, dass möglichst viele Datenblöcke zusammen in einem Cluster liegen. Die Preallokation von Blöcken kann abgeschaltet werden, wenn die Definition von `EXT2_PREALLOCATE` aus der Datei `<linux/ext2_fs.h>` entfernt wird.

Wie wird nun der Zielblock bestimmt? Bezeichne `n` die relative Nummer des zu allozierenden Blocks in der Datei. Der Blockallokationsalgorithmus verwendet die folgenden Heuristiken in der angegebenen Reihenfolge:

- In `u.ext2_i.i_next_alloc_goal` in der Inode der Datei ist eine Zielblocknummer eingetragen.
- Alle bisher vorhandenen Blöcke der Datei, angefangen beim Block Nummer `n-1`, werden danach durchsucht, ob ihnen logische Blöcke zugewiesen sind (ob der Block also kein „Loch“ ist). Der Zielblock ergibt sich aus der Nummer des ersten gefundenen logischen Blocks.
- Der Zielblock ist der erste Block der Blockgruppe, in der die Inode der Datei liegt.

6.3.4 Erweiterungen des *Ext2*-Dateisystems

Das *Ext2*-Dateisystem kennt im Vergleich zu den normalen Unix-Dateisystemen zusätzliche Dateiattribute (siehe Tabelle 6.5). In der derzeit aktuellen Version 0.5a sind dies:

EXT2_SECRM_FL Besitzt eine Datei dieses Attribut, so werden ihre Datenblöcke zunächst mit zufälligen Bytes überschrieben, bevor sie in der `truncate`-Funktion freigegeben werden. Dies sorgt dafür, dass nach dem Löschen der Datei ihr Inhalt auf keinen Fall wieder restauriert werden kann. Diese Eigenschaft führt jedoch bei gemappten Dateien zu Problemen und ist deswegen deaktiviert.

Makro	Wert	Bemerkung
EXT2_SECRM_FL	1	sicheres Löschen (n.i.)
EXT2_UNRM_FL	2	Undelete (n.i.)
EXT2_COMPR_FL	4	komprimierte Datei (n.i.)
EXT2_SYNC_FL	8	synchrones Schreiben
EXT2_IMMUTABLE_FL	16	unveränderliche Datei
EXT2_APPEND_FL	32	„Append-only"-Datei
EXT2_NODUMP_FL	64	Datei nicht archivieren
EXT2_NOATIME_FL	128	Zugriffszeit nicht aktualisieren

Tabelle 6.5: Die Dateiattribute des Ext2*-Dateisystems (n.i. = noch nicht implementiert)*

EXT2_UNRM_FL Dieses Attribut soll in Zukunft zur Implementierung der Restauration von gelöschten Dateien dienen. Derzeit ist diese Funktion jedoch noch nicht implementiert.

EXT2_COMPR_FL Dieses Attribut soll später anzeigen, dass die Datei komprimiert gespeichert ist. Derzeit ist die Online-Komprimierung noch nicht implementiert.

EXT2_SYNC_FL Besitzt eine Datei dieses Attribut, werden alle Schreibanforderungen synchron ausgeführt, d.h. nicht durch den Puffercache verzögert.

EXT2_IMMUTABLE_FL Dateien mit diesem Attribut können weder gelöscht noch verändert werden. Auch die Umbenennung und das Anlegen neuer Hardlinks sind nicht möglich. Selbst der Superuser kann die Dateien nicht ändern, solange sie dieses Attribut besitzen. Verzeichnisse, die dieses Attribut besitzen, können nicht verändert werden, d.h. es können keine neuen Dateien erstellt bzw. gelöscht werden. Bereits vorhandene Dateien oder Unterverzeichnisse können jedoch nach Belieben verändert werden.

EXT2_APPEND_FL Dateien mit diesem Attribut können ebenso wie beim vorherigen Attribut weder gelöscht noch umbenannt oder neu gelinkt werden. Dieses Attribut erlaubt jedoch zusätzlich ein anfügendes Schreiben in die Datei. In Verzeichnissen, die dieses Attribut tragen, können nur neue Dateien angelegt werden. Diese erben das `EXT2_APPEND_FL`-Attribut bei ihrer Erstellung.

EXT2_NODUMP_FL Dieses Attribut wird vom Kern nicht verwendet. Es sollte benutzt werden, um Dateien zu kennzeichnen, die bei einem Backup nicht gesichert werden müssen. Dieses Flag ist derzeit nicht implementiert.

EXT2_NOATIME_FL Ist dieses Flag gesetzt, wird bei einem Zugriff auf eine Datei die Zugriffszeit nicht aktualisiert.

Diese Attribute können jedoch nur mit Hilfe des `chattr`-Programms geändert werden. Das Programm `lsattr` zeigt sie an.

Die Entwicklung des *Ext2*-Dateisystems ist noch nicht beendet. Zur Liste der geplanten Erweiterungen gehören:

- Restauration gelöschter Dateien
- ACLs
- Automatische Dateikomprimierung
- Fragmente

6.4 Das *Proc*-Dateisystem

Als Beispiel für das Zusammenspiel des Virtuellen Dateisystems mit einer Dateisystemimplementation soll im Folgenden das *Proc*-Dateisystem näher betrachtet werden. Das *Proc*-Dateisystem in dieser Form ist eine Besonderheit von Linux. Es stellt auf portable Art und Weise Informationen über den aktuellen Zustand des Linux-Kerns sowie über die laufenden Prozesse bereit. Zusätzlich erlaubt es auf einfache Weise die Änderung von Kernelparametern zur Laufzeit.

Für jeden laufenden Prozess des Systems existiert ein Prozess-Verzeichnis `/proc/`*pid*, wobei *pid* die Prozessidentifikationsnummer des entsprechenden Prozesses ist. In diesem Verzeichnis befinden sich Dateien, die Informationen über bestimmte Eigenschaften des Prozesses enthalten. Einen genauen Überblick über diese Dateien und ihren Inhalt bietet Anhang C. Außerdem gibt es noch Dateien und Verzeichnisse für die prozessunabhängigen Informationen wie geladene Module, verwendete Bussysteme usw.

Allerdings gibt es auch Nachteile. Es gibt keine feste Schnittstelle zu den einzelnen Dateien, jeder Nutzer muss sozusagen selbst herausfinden, wo und wie die gewünschten Informationen in der Datei versteckt sind. Ein zweiter Nachteil ist, dass alle Informationen als Strings ausgegeben werden, dass also für eine Weiterverarbeitung jedes Mal eine Umwandlung notwendig ist.

Was seine Ideen betrifft, ähnelt es dem Prozessdateisystem von System V Release 4 sowie in Ansätzen dem Experimentalsystem Plan 9.[6]

Betrachten wir nun, wie dieses Dateisystem realisiert ist. Die vollständige Implementierung befindet sich im Verzeichnis `fs/proc`.

6.4.1 Strukturen des *Proc*-Dateisystems

Da das Dateisystem nicht auf Inodes im herkömmlichen Sinne zurückgreifen kann und die Strukturen sich zur Laufzeit ändern können, gibt es den Proc-Dir-Entry:

6 Plan 9 wurde von so bekannten Leuten wie Rob Pike und Ken Thompson bei den Bell Labs von AT&T entwickelt. Es zeigt, was die Entwickler von Unix heute machen. Einen guten Überblick über Plan 9 finden Sie in [PT+91]

```
struct proc_dir_entry {
   unsigned short low_ino;  /* (untere) Inode-Nummer */
   unsigned short namelen;  /* Länge des Namens      */
   const char *name;        /* Namen des Eintrags    */
   mode_t mode;             /* Mode                  */
   nlink_t nlink;           /* Link-Zähler           */
   uid_t uid;               /* UID                   */
   gid_t gid;               /* GID                   */
   unsigned long size;      /* Größe der Datei       */
   struct inode_operations * proc_iops; /* Inode-Op. */
   struct file_operations  * proc_fops; /* Datei-Op. */
   get_info_t *get_info;
   struct module *owner;
   struct proc_dir_entry *next, *parent, *subdir;
                            /* Verknüpfung           */
   void *data;
   read_proc_t *read_proc;   /* Read-Funktion        */
   write_proc_t *write_proc; /* Write-Funktion       */
   unsigned int count;       /* Nutzungszähler       */
   int deleted;              /* Lösch-Flag           */
   kdev_t rdev;              /*                      */
};
```

Im Weiteren wird diese Struktur auch als PD-Entry bezeichnet. Viele Einträge entsprechen denen einer Inode. Die Zeiger `next`, `parent` und `subdir` dienen zur Verknüpfung: `next` zeigt auf den nächsten Eintrag im aktuellen Verzeichnis, `parent` auf das übergeordnete Verzeichnis (beim Root-Verzeichnis auf sich selbst) und `subdir` auf ein Unterverzeichnis (falls vorhanden). Damit kann man alle PD-Entrys in einem Verzeichnis, ausgehend vom Parent-PD-Entry, mit folgender Schleife durchlaufen:

```
for (de = de->subdir; de ; de = de->next) {
  …
}
```

Es gibt eine Reihe von festen PD-Entries, die gleich beim Starten des Systems initialisiert werden, wobei die meisten nur die Werte bis zu den Datei-Operationen definieren. Diese Entries sind die Dateien und Verzeichnisse, die direkt unter `/proc/` liegen. Da es fast 30 sind und da sie außerdem von der Konfiguration des Kerns abhängen, spart sich der Autor an dieser Stelle eine Aufzählung und verweist auf die Funktion `proc_root_init()` in der Datei `fs/proc/root.c`. Wichtig ist, dass beim Erzeugen dieser Einträge zum Abschluss `proc_register()` aufgerufen wird, worin die PD-Entrys miteinander verkettet und die für den Dateityp richtigen Inode- und Datei-Operationen eingetragen werden.

Aus den PD-Entries werden dann Inodes erzeugt. Einige Bestandteile der Inode haben im *Proc*-Dateisystem eine besondere Bedeutung: In der Inode-Nummer ist (um 16 Bits verschoben) die PID des Prozesses, der sie erzeugt und der Typ der Datei gespeichert. Der Typ sagt aus, ob es sich um eine Datei im Verzeichnis `/proc` handelt, um ein Prozess-Verzeichnis usw. Dafür ist eine Reihe von festen(!) Aufzählungstypen definiert, die unter `pid_directory_inos` zu finden sind.

6.4.2 Implementierung des *Proc*-Dateisystems

Nachdem die Strukturen des *Proc*-Dateisystems erläutert wurden, soll als Erstes das Mounten des *Proc*-Dateisystems erläutert werden: Wie schon in Abschnitt 6.2.1 beschrieben, wird beim Mounten durch `do_mount()` die Funktion `read_super()` des VFS aufgerufen. Diese sucht sich aus der Liste `file_systems` den zum *Proc*-Dateisystem gehörenden Eintrag und ruft dessen Funktion `proc_read_super()` auf. Der an dieser Stelle übergebene Superblock ist bis auf die beim Mounten übergebenen Flags und die Gerätenummer noch leer.

```
struct super_block *proc_read_super(struct super_block *s,
                                    void *data, int silent)
{
   struct inode * root_inode;

   lock_super(s);
   s->s_blocksize = 1024;
   s->s_blocksize_bits = 10;
   s->s_magic = PROC_SUPER_MAGIC;
   s->s_op = &proc_sops;
   root_inode = proc_get_inode(s, PROC_ROOT_INO, &proc_root);
   if (!root_inode)
      goto out_no_root;
   /*
    * Prozesse als Unterverzeichnisse …
    */
   read_lock(&tasklist_lock);
   for_each_task(p) if (p->pid) root_inode->i_nlink++;
   read_unlock(&tasklist_lock);

   s->s_root = d_alloc_root(root_inode, NULL);
   if (!s->s_root)
      goto out_no_root;
   parse_options(data, &root_inode->i_uid, &root_inode->i_gid);
   s->u.generic_sbp = (void*) proc_super_blocks;
   proc_super_blocks = s;
   unlock_super(s);
   return s;

out_no_root:
   printk("proc_read_super: get root inode failed\n");
   iput(root_inode);
   s->s_dev = 0;
   unlock_super(s);
   return NULL;
}
```

Die Funktion initialisiert unter anderem die Superblock-Operationen (`s_ops`) mit der speziellen Struktur `proc_sops`:

```
static struct super_operations proc_sops = {
   read_inode:   proc_read_inode,
   put_inode:    proc_put_inode,
   delete_inode: proc_delete_inode,
   put_super:    proc_put_super,
   statfs:       proc_statfs,
};
```

Außerdem wird mit der Funktion `proc_get_inode()` die Inode-Struktur für den Superblock erzeugt. Darin wird über `iget()`-Schnittstelle des VFS die Funktion `proc_read_inode()` aufgerufen, ihre Arbeitsweise ist weiter unten beschrieben.

Beim Erzeugen der Root-Inode wird natürlich auf den entsprechenden PD-Entry zurückgegriffen. Dieser ist fest definiert:

```
struct proc_dir_entry proc_root = {
   PROC_ROOT_INO, 5, "/proc",
   S_IFDIR | S_IRUGO | S_IXUGO, 2, 0, 0,
   0, &proc_root_inode_operations, &proc_root_operations,
   NULL, NULL,
   NULL,
   &proc_root, NULL
};
```

Der PD-Entry wird in die Inode eingetragen (`u.generic_ip`) und außerdem werden folgende Informationen in die Inode übertragen: Mode, UID, GID, Größe und die Operationen.

Nach dem Erzeugen der Inode wird die Anzahl der Verweise auf das Verzeichnis berechnet. Dies wäre im Normalfall 2 + die Anzahl der Unterverzeichnisse, da jedes Unterverzeichnis mit „..“ einen solchen Verweis besitzt. Hier verbirgt sich ein Problem. Da die Funktion `proc_get_inode()` nur einmal für die „Lebenszeit“ der Speicher-Inode aufgerufen wird und die Anzahl der Prozesse sich während der Laufzeit eines Linux-Systems sich sicher ändert, muss dieser Wert später neu berechnet werden. Das geschieht in der weiter unten beschriebenen Funktion `proc_root_lookup()`.

Die Funktion `d_alloc_root()` erzeugt mit Hilfe der Inode daraufhin den PD-Entry des *proc*-Wurzelverzeichnisses, der in den Superblock eingetragen wird.

Danach sucht die Funktion `parse_options()` in den übergebenen Mount-Optionen `data` Angaben von UID und GID (z. B. „`uid=1701,gid=42`“) und setzt den Eigentümer der Root-Inode. Sind keine Werte angegeben, werden die des aktuellen Prozesses eingetragen. Der fertige Superblock wird zurückgeben und die Arbeit der Funktion ist beendet.

Die Operationen für die Inode- und Dateibehandlung sind sehr spärlich implementiert: die Struktur `proc_root_inode_operations` stellt nur die Komponente `lookup` als Funktion `proc_root_lookup()` zur Verfügung.

```
static struct dentry *proc_root_lookup(struct inode * dir,
                                       struct dentry * dentry);
```

Diese Funktion dient der Namensauflösung für die PID-Verzeichnisse (für die anderen Verzeichnisse ist `proc_dir_inode_operations` als Inode-Operationen eingetragen). Zuerst aktualisert sie die Anzahl der laufenden Prozesse (minus der Anzahl der Idle-Tasks) in `nlinks`. Da das lange dauern kann (relativ gesehen), wird dieser Code-Teil nur durchlaufen, wenn sich der Wert in `total_forks` seit dem letzen Mal geändert hat. Dann ruft sie `proc_lookup()` auf, um den PD-Entry zu ermitteln. Als letztes ruft sie `proc_pid_lookup()` auf. Diese Funktion überprüft, ob der Prozess, dessen PID den Namen des gewünschten Verzeichnisses bildet, noch existiert und erzeugt eine spezielle Inode, bei der PID, GID und UID nicht aus dem PD-Entry, sondern aus dem Prozess übernommen werden. Diese neue Inode wird dann in den PD-Entry eingetragen und damit die alte, bei `proc_lookup()` erzeugt überschrieben.

Sehen wir uns nun an, was beim Zugriff auf dieses Dateisystem passiert. Interessant ist, dass die entsprechenden Daten immer erst dann generiert werden, wenn sie benötigt werden. Nehmen wir an, wir wollen durch den Systemruf *open* auf eine Datei zugreifen, dann landen wir irgendwann in der proc-spezifischen `lookup()`-Funktion:

```
struct dentry *proc_lookup(struct inode * dir,
                           struct dentry *dentry);
```

Ohne Beschränkung der Allgemeinheit sei ein absoluter Pfadnahme beim Öffnen angegeben, in `dir` befindet sich dann die Root-Inode. Daraus holt sich die Funktion den PD-Entry und durchläuft die `next`-Liste. Sollte der Namen eines Eintrages mit dem des PD-Entry übereinstimmen wird mit `proc_get_inode()` eine neue Inode erzeugt. Als Inode-Nummer wird der Wert `low_ino` aus dem PD-Entry verwendet.

In dieser wird über die Funktion `iget()` des VFS und `get_new_inode()` die in den Superblock-Operationen eingetragene Funktion zum Lesen von Inodes `proc_read_inode()` aufgerufen. Die übergebene Inode ist eine bereits allozierte, aber leere Inode. Diese Funktion ist schön kurz setzt nur die Zeiten der Inode.

```
inode->i_mtime = inode->i_atime =
            inode->i_ctime = CURRENT_TIME;
```

Nach dem Aufruf von `iget()` werden aus dem PD-Entry der Modus, UID, GID, Größe, Links und die Operationen in die Inode übernommen. Anschließend wird aus der Inode-Nummer die darin gespeicherte PID extrahiert und der Prozess gesucht. Öffnen wir ein Prozess-Verzeichnis und das `dumpable`-Flag des Prozesses ist gesetzt, werden EUID und EGID des Prozesses als UID und GID der Inode übernommen. Nun ist die Inode gefüllt und die Arbeit der Funktion beendet.

Als drittes Beispiel soll erläutert werden, was passiert, wenn eine Datei ausgelesen wird. Der Einsprung sei der Aufruf der `read()`-Funktion des VFS. Im *Proc*-Dateisystem wird bei Dateien beim Registrieren (siehe Abschnitt 6.4.1) dafür die funktion `proc_file_read` eingetragen.

```
#define PROC_BLOCK_SIZE (3*1024)

static ssize_t proc_file_read(struct file * file, char * buf,
                              size_t nbytes, loff_t *ppos)
{
   struct inode * inode = file->f_dentry->d_inode;
   int   eof=0;
   ssize_t   n, count;
   char   *start;
   struct proc_dir_entry * dp;

   dp = (struct proc_dir_entry *) inode->u.generic_ip;
   if (!(page = (char*) __get_free_page(GFP_KERNEL)))
      return -ENOMEM;

   while ((nbytes > 0) && !eof)
   {
      count = MIN(PROC_BLOCK_SIZE, nbytes);

      if (dp->get_info) {
         /*
          * Kompatibilitaet mit alten Netzwerk-Routinen …
          */
         n = dp->get_info(page, &start, *ppos, count);
         if (n < count)
            eof = 1;
      } else if (dp->read_proc) {
         n = dp->read_proc(page, &start, *ppos,
                  count, &eof, dp->data);
      } else
         break;
      …
   }
   …
}
```

Aufgrund von Speicher-Beschränkungen können nicht mehr als 4 KB auf einmal ausgelesen werden, deswegen die `while`-Schleife.

Wie man sieht, geschieht das Auslesen mit Hilfe der Funktion, die im PD-Entry unter `read_proc` eingetragen ist. Daraus folgt, das jede Datei, die im *Proc*-Dateisystem registriert werden soll, eine solche Funktion definieren muss!

Will ein Prozess zum Beispiel die Datei `/proc/loadavg` auslesen, so wird die Funktion `loadavg_read_proc()` aufgerufen, die beim Initialisieren der Datei dort eingetragen wurde. Die Funktion generiert den Inhalt der Datei, indem sie die nötigen Werte in die Speicherseite einträgt und die Größe des Puffers (also der Datei) zurückgibt. Dies hat in den Quellen folgendes Aussehen:

```
static int loadavg_read_proc(char *page, char **start, off_t off,
                             int count, int *eof, void *data)
```

```
{
   int a, b, c;
   int len;

   a = avenrun[0] + (FIXED_1/200);
   b = avenrun[1] + (FIXED_1/200);
   c = avenrun[2] + (FIXED_1/200);
   len = sprintf(page,"%d.%2d %d.%2d %d.%2d %d/%d %d\n",
      LOAD_INT(a), LOAD_FRAC(a),
      LOAD_INT(b), LOAD_FRAC(b),
      LOAD_INT(c), LOAD_FRAC(c),
      nr_running, nr_threads, last_pid);
   …
   return len;
}
```

Die Funktionen der einzelnen Dateien sind in `fs/proc/proc_misc.c` oder bei den speziellen Quellen implementiert. So ist die Funktion `get_module_list()` für die Datei `/proc/module` in der Datei `kernel/module.c` der Module-Implementation zu finden.

Pearson Education

ADDISON-WESLEY

7 Gerätetreiber unter Linux

Ein Computerterminal ist kein klobiger alter Fernseher,
vor dem eine Schreibmaschinentastatur liegt.
Es ist eine Schnittstelle, an der sich Körper und Geist mit dem
Universum zusammenschalten
und Teile davon durch die Gegend bewegen können.

Douglas Adams

Im Steinzeitalter der Computer wurde der hardwarespezifische Teil eines Programmes einfach als Bibliothek an das ausführbare Programm gelinkt. Früher gab es jedoch weder mehrere Benutzer noch mehrere Programme, die sich eine CPU und die dazugehörige Hardware teilen mussten. Sollte ein Betriebssystem über diese Eigenschaften verfügen, musste man darüber nachdenken, wie man die Ansteuerung von Hardware implementieren konnte, ohne dass sich verschiedene Prozesse oder Benutzer gegenseitig störten. So kam es zu dem Konzept der Gerätetreiber.

Gerätetreiber sind genau genommen eine Sammlung von Routinen, die magische Zahlen an magische Plätze in der Hardware schreiben. Damit sich diese Routinen aber vernünftig in ein Betriebssystem einpassen lassen, ist eine Schnittstelle implementiert, die das Betriebssystem bei bestimmten Aktionen auf der Hardware aufrufen kann. Das Betriebssystem übernimmt dabei die Aufgabe, sowohl den Hardwarezugriff selbst als auch die Ressourcenverteilung zu koordinieren. Da Gerätetreiber im Speicherbereich des Kerns laufen, haben Sie auch dieselben Berechtigungen wie der Kern. Bei der Implementation eines Gerätetreibers ist daher besondere Vorsicht und Sorgfalt geboten, da ein Griff in das falsche Register verheerende Folgen haben kann. Außerdem sollte ein Gerätetreiber immer so sparsam wie möglich mit den Ressourcen umgehen, vor allem dann, wenn er fest in den Kern eingefügt wurde.

In Unix-Systemen ist die Treiberschnittstelle, wie bei vielen anderen Betriebssystemen auch, über Zugriffe auf das Dateisystem implementiert. Spezielle Eintrittspunkte der Treiber werden unter Unix als spezielle virtuelle Dateien im Dateisystem dargestellt. So kann der Benutzer mit ganz normalen Dateioperationen wie z. B. *open*, *read*, *write* und *close* auf Geräte zugreifen.

Damit das Betriebssystem die Gerätetreiber unterscheiden kann, wird den virtuellen Dateien über das Dateisystem eine eindeutige Kennung zugeordnet. Über die sogenannte *Major-Nummer* wird der Zugriff auf eine Treiberdatei auf die richtigen Routinen im Kernel umgeleitet.

Einige der zur Zeit fest vergebenen Major-Nummern können Tabelle 7.1 entnommen werden, die Datei `Documentation/devices.txt` enthält eine jeweils vollständige Liste.

Major	Zeichengeräte	Blockgeräte
0	*unnamed* für NFS, Netzwerk usw.	
1	Speichergeräte (`mem`)	RAM-Disk
2	Pseudo-TTY Master (`pty*`)	Disketten (`fd*`)
3	Pseudo-TTY Slaves (`ttyp*`)	IDE–Festplatten (`hd*`)
4	Terminals	
5	Terminals & AUX	
6	Parallele Schnittstellen	
7	Virtuelle Konsolen (`vcs*`)	Loopback-Geräte
8		SCSI-Festplatten (`sd*`)
9	SCSI-Tapes (`st*`)	Metadisk-Geräte (RAID)
10	Busmäuse (`bm, psaux`)	
11	Keyboard Rohdevice	SCSI-CD-ROM (`sr*`)
12	QIC02-Tape	MSCDEX CD-ROM callback support
13	PC-Speaker-Treiber	XT-8-Bit-Festplatten (`xd*`)
14	Soundkarten	BIOS-Festplatten-Unterstützung
15	Joystick	Cdu31a/33a CD-ROM
16	Nicht-SCSI Scanner	GoldStar CD-ROM
17	Chase serial card — Alternativen	Optics Storage CD-ROM
18	Chase serial card — Alternativen	Sanyo CD-ROM
19	Cyclades-Treiber	Double — komprimierender Treiber
20	Cyclades-Treiber	Hitachi CD-ROM
21	SCSI Generic	Acorn MFM hard drive interface
22	Digiboard serial card	2. IDE-Schnittstellen-Treiber
23	Digiboard serial card — Alternativen	Mitsumi CD-ROM (`mcd*`)
24	Stallion serial card	Sony535 CD-ROM
25	Stallion serial card — Alternativen	Matsushita CD-ROM 1
26	Quanta WinVision frame grabber	Matsushita CD-ROM 2
27	QIC117-Tape	Matsushita CD-ROM 3
28	Stallion serial card — die Programmierung	Matsushita CD-ROM 4
29	Framebuffer-Treiber	weitere CD-ROMs
30	iCBS2	Philips LMS-205 CD-ROM

Tabelle 7.1: Einige der zur Zeit vergebenen Major-Nummern

Häufig können Geräte mehrere logische Untereinheiten haben (zum Beispiel Festplatten oder serielle Geräte), deshalb hat man eine weitere Kennung, die *Minor-Nummer* vorgesehen, die dem Treiber über das Dateisystem übergeben wird. Diese Zahl zwischen 0 und 255 kann auch dazu benutzt werden, zwischen mehreren Betriebsarten eines Gerätes umzuschalten.

7.1 Zeichen- und Blockgeräte

In UNIX-Systemen wird zwischen zeichenorientierten und blockorientierten Geräten unterschieden.

Zeichenorientierte Geräte stellen die Daten mit sequentiellem Zugriff zeichenweise zur Verfügung bzw. tauschen die Daten in sequentiellen Datenströmen aus. Zum Beispiel geschieht der Datentransfer bei seriellen oder parallelen Schnittstellen oder Bandlaufwerken zeichenweise. Da diese Zugriffsart sehr einfach ist, werden sehr viele Treiber als zeichenorientierte Gerätetreiber implementiert.

Für Festplatten wäre diese Zugriffsart sehr ineffizient, da beim Lesen eines Datenbytes im asynchronen Betrieb immer eine ganze Umdrehung der Platte abgewartet werden müsste, bis das nächste Byte gelesen werden könnte. Um diesem Manko zu entgehen, werden solche Geräte immer blockweise gelesen, d.h. das System fordert von der Hardware immer eine Anzahl ganzer Blöcke einer bestimmten Größe an und legt diese Blöcke in einem Zwischenpuffer ab. Wird beim Schreiben oder Lesen eine Blockgrenze erreicht, so wird der alte Block mit der Hardware synchronisiert und dann ein neuer Block bearbeitet. Treiber für diese Art von Geräten stellen meist nur die Funktionen zur Verfügung, die zum Transfer der Blöcke oder zur Verwaltung der Blockgrößen nötig sind. Sie werden deshalb Blockgerätetreiber genannt.

Im Dateisystem von LINUX werden Zeichen- und Blockgeräte anhand eines Flags unterschieden, das beim Anlegen einer virtuellen Treiberdatei durch `mknod` als Parameter übergeben wird.

```
# mknod /dev/name type major minor
```

Zeichenorientierte Treiber implementieren die benötigten Funktionen direkt und registrieren diese beim System mit der `file_operations`-Struktur.

```
struct file_operations busmouse_fops=
{
        owner:          THIS_MODULE,
        read:           busmouse_read,
        write:          busmouse_write,
        poll:           busmouse_poll,
        open:           busmouse_open,
        release:        busmouse_release,
        fasync:         busmouse_fasync,
};
```

Bei einem Aufruf von *read* und *write* werden also direkt die entsprechenden Routinen im Treiber aufgerufen.

In der Version 2.4 wurde die `file_operations`-Struktur aufgeteilt. In einem blockorientiertem Treiber werden die notwendigen Routinen nun in der `block_device_operations`-Struktur registriert — wie hier beim Floppy-Treiber:

```
static struct block_device_operations floppy_fops = {
        open:                   floppy_open,
        release:                floppy_release,
        ioctl:                  fd_ioctl,
        check_media_change:     check_floppy_change,
        revalidate:             floppy_revalidate,
};
```

Analog zu den zeichenorientierten Geräten existiert auch für Blockgeräte eine Registrierungsfunktion `register_blkdev()`, mit der die `block_device_operations`-Struktur beim System angemeldet wird.

7.2 Hardware

Eigentlich sollte in einem Kernelbuch nichts über Hardware stehen. Dennoch ist die Kenntnis der Hardware die wichtigste Voraussetzung, zum Schreiben eines Gerätetreibers. Wir wollen deswegen kurz auf die Besonderheiten der gängigen PC Hardware eingehen.

7.2.1 Port I/O

Der einfachste Zugriff auf Hardware ist der Port-Zugriff, d. h. auf Adressen im Speicherbereich der CPU — die so genannten I/O-Ports `0x0000-0xffff`. In `include/asm/io.h` sind mit Hilfe von Makros für diese Zugriffe entsprechende Funktionen definiert.

Für Bytezugriffe dienen die Funktionen `outb()` bzw. `outb_p()` und `inb()` bzw. `inb_p()`. Analog dazu dienen die Funktionen `outw()` und `inw()` für Wortzugriffe und `outl()` sowie `inl()` für Doppelwortzugriffe.

Wie vermutet, dienen die `outX`-Funktionen zum Schreiben und die `inX` zum Lesen des angegebenen Ports. Die Funktionen mit der Endung `_p` fügen eine kleine Pause an die I/O-Operation an. Das kann in manchen Fällen, vor allem bei alter ISA-Hardware nötig sein. Für uralte Hardware und für problematische Timingsituationen kann das Verhalten der `*_p()`-Funktionen mit Hilfe des Makros `REALLY_SLOW_IO` verändert werden. Dieses Makro muss vor dem Include der Datei `include/asm/io.h` definiert werden, um nach einem I/O-Zugriff dem Bus etwas mehr Zeit zu geben.

Linux verwendet für diese Verzögerung auf x86-Systemen normalerweise einen Lesezugriff auf den Port 0x80. Dieser Port wird vom BIOS benutzt, um während des Bootens Zustandsmeldungen auszugeben; das Lesen dieses Ports sollte deshalb stets ungefährlich sein. In früheren Zeiten wurden zur Verlangsamung von I/O-Zugriffen unbedingte

Sprünge eingesetzt, diese dauerten auf Prozessoren der 386-Klasse hinreichend lange. Heutige Prozessoren führen unbedingte Sprünge jedoch so schnell aus, dass diese Methode ungeeignet ist[1]. Trotzdem ist es möglich, auf x86-Systemen mit Hilfe des Makros `SLOW_IO_BY_JUMPING` die Verzögerung durch zwei unbedingte Sprünge durchzuführen.

Damit nicht verschiedene Teile des Kerns, insbesondere nicht mehrere Gerätetreiber auf dieselben I/O-Ports schreiben und so die Hardware in einen undefinierten Zustand bringen, lassen sich unter Linux I/O-Ports für den Zugriff sperren.

Dazu kann dem Kern beim Start ein Bootparameter übergeben werden, der alle gesperrten Bereiche enthält. Startet das System nach dem Einbau einer neuen Karte nicht mehr, sollte man zunächst versuchen, den Adressraum dieser Karte auszublenden. Ein fiktives Beispiel soll dies verdeutlichen:

Eine Scannerkarte belege die Adressen 0x300–0x30f (dort könnte sich auch eine Netzwerkkarte befinden). Mit Hilfe des Bootparameters

```
reserve=0x300,0x10
```

wird dieser Bereich ausgeschlossen.

Innerhalb eines Treibers dienen die folgenden Makros zum I/O-Port-Management:

```
check_region(from, num)
request_region(from, num, name)
release_region(from, num)
```

Das Makro `check_region()` fragt beim Kern an, ob `num` I/O-Ports beginned ab der I/O-Adresse `from` frei sind. Ist bereits mindestens ein I/O-Port in diesem Bereich gesperrt, liefert es den Fehler `EBUSY` zurück, sonst NULL.

Das Makro `request_region()` sperrt `num` I/O-Ports beginned ab der I/O-Adresse `from`. Der dritte Parameter ist der Name des Treibers, der den Port sperren will. Der Name des Treibers wird nur vom *Proc*-Dateisystem verwendet, man kann dadurch feststellen, welche I/O-Ports von welchem Treiber belegt sind. Auch dieses Makro liefert im Fehlerfall den Fehler `EBUSY` zurück. Das Makro `release_region()` gibt gesperrte I/O-Ports schließlich wieder frei. Dies ist z. B. bei Modulen notwendig, sowie sie aus dem Kern entfernt werden.

7.2.2 Der PCI-Bus

Die PCI[2]-Busarchitektur ist nicht nur bei Intel-PCs verbreitet, sie hat sich auch in anderen Architekturen durchgesetzt. Die Besonderheit am PCI-Bus ist die vollständige Trennung zwischen dem Bus-Subsystem und dem CPU-Subsystem. Ein spezieller Chipsatz,

1 Die schnellere Ausführung von unbedingten Sprüngen sorgte bei der Einführung der ersten 486-Computer zu ernsten Problemen, da Diskettenlaufwerke plötzlich nicht mehr stabil funktionierten.

2 Peripheral Component Interconnect

die sogenannte *PCI-Bridge*, übernimmt dabei die meisten Aufgaben des Transfers. Da beim PCI-Bus alle Busleitungen sowohl als Daten- als auch als Adressleitungen benutzt werden können, müssen abwechselnd immer Adress- und Datenpakete über den Bus gesendet werden. Bei einem 16-bit-Zugriff können zudem die verbleibenden Leitungen zugleich als Datenbus benutzt werden.

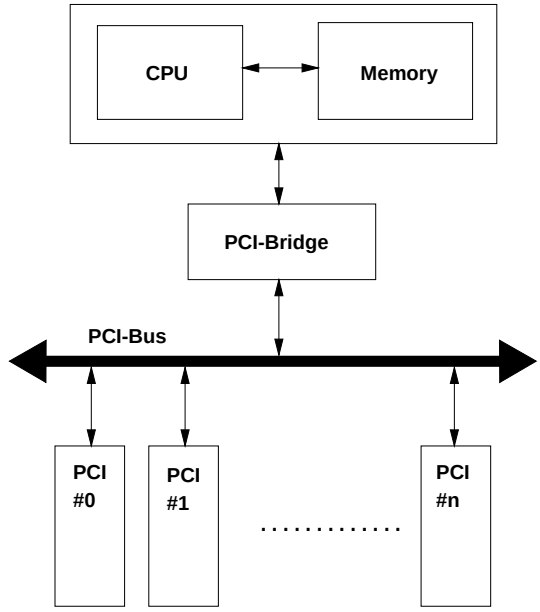


Abbildung 7.1: Schematischer Aufbau des PCI-Subsystems

Bei Übertragung eines ganzen Datenblocks kann die Bridge den Transfer dadurch optimieren, dass sie nur einmal die Startadresse über den Bus sendet und danach alle Datenpakete im Block hinterherschiebt. Die Adressen werden sowohl in der Bridge als auch im Adapter (Steckkarte) automatisch inkrementiert. Dieses Verfahren wird auch *PCI-Burst-Zyklus* genannt. Mit einem 64-bit-Datentransfer können damit Geschwindigkeiten von bis zu 266 MByte/s erreicht werden[3]. Da die PCI-Bridge selbstständig den schnellsten Transfermodus wählt, braucht sich der Treiberprogrammierer darum nicht zu kümmern.

Ein PCI-Adapter kann seine Daten auch ohne Beteiligung der CPU in den Speicherbereich des PCs schicken. Dazu versetzt sich der Controller auf dem Adapter in den sogenannten *Busmaster*-Zustand und kann dann den Datentransfer zur Bridge und damit in den Hauptspeicher des PCs oder in den Speicherbereich eines anderen PCI-Adapters selbstständig vornehmen. Da dieses Verfahren ähnlich dem ISA-DMA realisiert ist, wird es auch häufig *Busmaster-DMA* genannt.

Für jedes Gerät am PCI-Bus ist ein Konfigurationsadressraum von 256 Bytes vorgesehen, in dem die Parameter für dieses Gerät abgelegt sind. Die ersten 64 Bytes sind fest in der

3 Allerdings erfüllen derzeit nur Serverboards diese Spezifikation.

PCI-Spezifikation vorgegeben, der Rest ist herstellerabhängig. Beim Booten bestimmt das PCI-BIOS die benötigten Resourcen aller Geräte und weist jedem von ihnen Parameter wie Basisadresse und IRQ zu, indem sie in diesem Header eingetragen werden. Der Treiber liest die zugewiesenen Parameter später wieder aus.

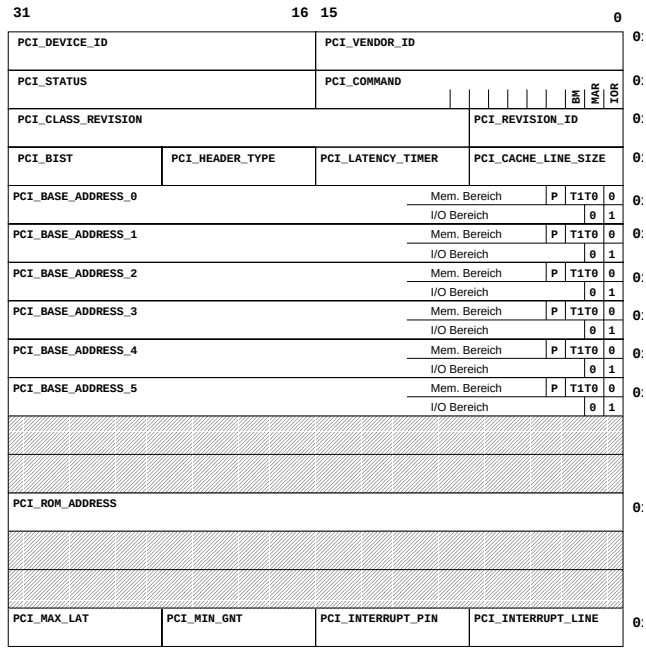


Abbildung 7.2: Der PCI-Header und die dazugehörigen MAKROS aus `pci.h`

Weiterhin besitzt jedes Gerät am PCI-Bus eine eindeutige Adresse. Im Gegensatz zum ISA-Bus, bei dem die Slots von der Treiberseite nicht unterscheidbar sind, kann die Position eines jeden PCI-Gerätes bestimmt werden. Diese Position ergibt sich aus der Bus-Nummer, einer Geräte- und einer Funktionsnummer. Die Bus-Nummer gibt Auskunft, auf welchem PCI-Bus sich das Gerät befindet, es können sich durchaus mehrere PCI-Busse in einem Computer befinden (der AGP-Port erscheint auch als ein eigenständiger Bus). Die Gerätenummer bestimmt eindeutig den PCI-Slot, in dem sich ein Gerät befindet, während die Funktionsnummer es erlaubt, in einem PCI-Gerät mehrere Untereinheiten, wie z. B. 2 SCSI-Controller, unterzubringen. Steckt man also ein PCI-Gerät in einen anderen Slot, ändert sich seine Adresse, was dazu führen kann, dass das PCI-BIOS andere Resourcen zuweist[4].

Linux stellt eine Reihe nützlicher Funktionen und Makros zur Verfügung, mit denen der PCI-Konfigurationsraum behandelt werden kann. Dem Treiberprogrammierer reichen meistens die Funktionen, welche zur Erkennung der Karte und zum Ermitteln der Basisadressen bzw. der Interruptnummer nötig sind, dennoch soll hier kurz auf

4 Dies ist auch der Grund, dass andere Betriebssysteme plötzlich neue Hardware „entdecken“.

die wichtigsten Konfigurationsmerkmale eingegangen werden. Die zugehörigen Makrodefinitionen finden sich in `include/linux/pci.h`. Für eine eindeutige Identifizierung der Karte sind im Header ein Hersteller-Code (`PCI_VENDOR_ID`), ein Geräte-Code (`PCI_DEVICE_ID`), ein Klassen- und ein herstellerspezifischer Revisionscode abgelegt, die in der Regel nicht überschrieben werden können. Falls die PCI-Frontends anderer Hersteller verwendet werden, so geben der Unterhersteller- (`PCI_SUB_VID`) und der Untergeräte-Code (`PCI_SUB_ID`) weitere Informationen über den Hersteller des Subsystems. Anhand des Klassencodes kann festgestellt werden, mit welcher Geräteklasse (z. B. Massenspeicher-Controller oder Netzwerkkarte) man es zu tun hat. Jedes PCI-Gerät kann bis zu 6 I/O-Bereiche belegen, die über das PCI-Bios im Konfigurationsraum eingetragen werden (`PCI_BASE_ADDRESS_0-5`). Das letzte Bit des abgelegten Wertes markiert, ob es sich bei der Adresse um einen I/O-Adressbereich handelt, oder ob der Bereich in den Speicherbereich eingeblendet wird. Bei Speicheradressen im Memory-Bereich wird zusätzlich eingetragen, ob der Zugriff auf die Speicheradresse als 32 Bit, 64 Bit oder im Speicherbereich unterhalb von 1 MByte möglich ist.

Das PCI-Treibermodell

Mit der Version 2.4 hält ein neues PCI-Treibermodell Einzug. Der Vorteil des neuen Ansatzes besteht in der Möglichkeit, auf *Hot-Plug*-Geräte (zur Laufzeit des Systems hinzugefügte Geräte) reagieren zu können. Während Hot-Plug-PCI-Geräte noch ausgesprochen selten sind, erlaubt dieser Ansatz jedoch, PC-Card-Geräte wie PCI-Geräte anzusprechen. Wenn im Folgenden also von PCI-Geräten gesprochen wird, sind auch PC-Card Geräte gemeint.

Zunächst muss im Treiber eine Tabelle aller unterstützten Geräte angelegt werden. Dazu dient die Struktur `pci_device_id`:

```
struct pci_device_id {
  unsigned int vendor, device;    /* Hersteller- und Geräte-Code */
  unsigned int subvendor, subdevice; /* Subsystem-Codes          */
  unsigned int class, class_mask;    /* Class und Subclass-Maske */
  unsigned long driver_data;         /* beliebige Treiberdaten   */
};
```

Diese Struktur beschreibt die Maske aller Geräte, für die sich der Treiber interessiert. Die Komponenten `vendor` und `device` nehmen den Hersteller- und Gerätecode des PCI-Gerätes. `subvendor` und `subdevice` nehmen die Subsystem-Codes auf. Die Konstante `PCI_ANY_ID` muss benutzt werden, falls ein entsprechender Code nicht existiert oder bei der Auswahl nicht beachtet werden soll. So kann sich z. B. ein Treiber für alle Geräte eines Herstellers registrieren, indem der Herstellercode eingetragen wird, während der Gerätecode of `PCI_ANY_ID` gesetzt wird. Die Komponenten `class` und `class_mask` erlauben es einem Treiber, sich für eine Geräteklasse zu registrieren. Hier ist der Wert 0 einzutragen, falls die Geräteklasse nicht benutzt werden soll. Die letzte Komponente erlaubt es, beliebige Treiberdaten in der Struktur einzutragen.

Ein Treiber kann nun ein Feld von `pci_device_id`-Einträgen registrieren (ID-Tabelle). Das folgende Beispiel stammt aus dem PCnet32 Treiber:

```
static struct pci_device_id pcnet32_pci_tbl[] __devinitdata = {
  { PCI_VENDOR_ID_AMD, PCI_DEVICE_ID_AMD_PCNETHOME,
    PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0 },
  { PCI_VENDOR_ID_AMD, PCI_DEVICE_ID_AMD_LANCE,
    PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0 },
  { PCI_VENDOR_ID_AMD, PCI_DEVICE_ID_AMD_LANCE,
    0x1014, 0x2000, 0, 0, 0 },
  { 0, }
};
```

Der Modifier `__devinitdata` sollte stets für die Registrierungsfelder verwendet werden. Er sorgt dafür, dass die Tabelle vom Kern automatisch entfernt werden kann, falls das Hot-Plugin deaktiviert wurde.

Das Feld `pcnet32_pci_tbl` enthält 3 Einträge. Der erste Eintrag erklärt, dass sich der Treiber für AMD-HomePNA-Netzwerkadapter interessiert. Der zweite Eintrag betrifft PCnet-LANCE-Netzwerkkarten. Der dritte Eintrag spezifiziert einen Subsystem-Hersteller- und -Geräte-Code. Der letzte Eintrag dient als Endekennzeichen.

Die zweite zur Registrierung notwendige Struktur ist `pci_driver`:

```
struct pci_driver {
  struct list_head node;
  char *name;
  /* Zeiger auf die ID-Tabelle */
  const struct pci_device_id *id_table;
  /* neues Gerät wurde gefunden */
  int (*probe)(struct pci_dev *dev,
               const struct pci_device_id *id);
  /* Geät wurde entfernt */
  void (*remove)(struct pci_dev *dev);
  /* Power-Management: Gerät wird suspendiert */
  void (*suspend)(struct pci_dev *dev);
  /* Power-Management; Geät wird aufgeweckt */
  void (*resume)(struct pci_dev *dev);
};
```

Die Komponente `node` dient zur Verkettung aller Einträge im PCI-Bustreiber. `name` enthält den Namen des Treibers. Die Funktion `probe()` wird vom PCI-Bustreiber gerufen, wenn ein neues Gerät gefunden wurde. Analog dazu ruft der PCI-Bustreiber `remove()`, sowie das Gerät entfernt wurde. Die Funktionen `suspend()` und `resume()` dienen dem Power-Management. Schauen wir wiederum in den PCnet32-Treiber:

```
static struct pci_driver pcnet32_driver = {
  name:     "pcnet32",
  probe:    pcnet32_probe_pci,
  remove:   NULL,
  id_table: pcnet32_pci_tbl,
};
```

Der PCnet32-Treiber registriert eine `probe()`-Funktion, sowie die bereits angelegte ID-Tabelle. Solange der Treiber kein Hot-Plugin unterstützt, darf die `remove()`-Funktion NULL sein. Unser Beispiel unterstützt auch kein Power-Management.

```
static int __init pcnet32_init_module(void)
{
  int err;

  …

  /* find the PCI devices */
  if ((err = pci_module_init(&pcnet32_driver)) < 0 )
     return err;
  return 0;
}
```

`pcnet32_init_module()` registriert nun den Treiber beim PCI-Bustreiber. Dies kann über zwei Funktionen geschehen:

```
int pci_register_driver(struct pci_driver *drv);
int pci_module_init(struct pci_driver *drv);
```

`pci_register_driver()` registriert den Treiber beim Bustreiber und testet sofort, ob sich bereits Geräte im System befinden, für die sich der Treiber registrieren möchte und für die sich noch kein anderer Treiber registriert hat. Falls ja, so wird zu diesem Zeitpunkt die `probe()`-Funktion des Treibers aufgerufen. `pci_register_driver()` gibt die Anzahl der Geräte zurück, für die die `probe()`-Funktion gerufen wurde. Der Wert 0 zeigt an, dass sich noch kein Gerät im System befindet, für das sich der Treiber registriert hat. Zu beachten ist, dass in diesem Falle der Treiber trotzdem registriert wird.

Die Funktion `pci_module_init()` bildet eine dünne Schicht um `pci_register_driver()` und ist als Hilfsfunktion für Module gedacht.

```
static inline int pci_module_init(struct pci_driver *drv)
{
  int rc = pci_register_driver (drv);

  if (rc > 0)
    return 0;

  /* Falls der Treiber statisch in der Kern eingebunden wurde
   * und die Option CONFIG_HOTPLUG aktiviert ist, sollte der
   * Treiber aktiv bleiben, um auf das Einstecken von
   * Hardware reagieren zu können.
   * Wurde der Treiber andererseits als Modul compiliert,
   * so sollte ein Dämon dafür sorgen, dass der Treiber
   * nachgeladen wird, falls eine Hardware dem System
   * hinzugefügt wird. */
#if defined(CONFIG_HOTPLUG) && !defined(MODULE)
   if (rc == 0)
     return 0;
```

```
#endif

  /* Falls der Kontrolfluss zu dieser Stelle gelangs, so
   * muss der Treiber wieder entfernt sowie ein passender
   * Fehler generiert werden */
  pci_unregister_driver (drv);

  return -ENODEV;
}
```

Falls sich noch kein Gerät im System befindet und der Treiber als Module compiliert wurde, so entfernt `pci_module_init()` den Treiber wieder und liefert einen Fehler zurück. Auf diese Weise kann verhindert werden, dass ein Modul geladen wird, bevor sich mindestens ein Gerät im System befindet.

Die Funktion `pci_unregister_driver()` entfernt einen Treiber. Falls sich noch Geräte im System befinden, so wird die `remove()`-Funktion des Treibers für jedes Gerät aufgerufen. Diese Funktion muss zumindest in der Exit-Funktion eines Moduls aufgerufen werden.

Die Probe-Funktion

Befindet sich zur Zeit der Registrierung ein passendes Gerät im System bzw. wird ein Gerät nach der Registrierung hinzugefügt, so wird die `probe()`-Funktion des PCI-Treibers gerufen.

```
int probe(struct pci_dev *dev, const struct pci_device_id *id);
```

Das erste Argument der Funktion ist ein Zeiger auf die `pci_dev`-Struktur. Das zweite Argument zeigt auf den Eintrag der ID-Tabelle, der mit dem Gerätecode übereingestimmt hat. War die ID-Tabelle gleich NULL, so ist auch dieses Argument gleich NULL.

Die `pci_dev`-Struktur enthält bereits alle Ressourcen des PCI-Gerätes. Im Gegensatz zu älteren Linux-Versionen sollten Ressourcen in der Version 2.4 nicht mehr direkt aus dem PCI-Konfigurationsraum gelesen werden. Insbesondere kann der Interrupt-Eintrag im Konfigurationsraum unterschiedlich sein.

```
struct pci_dev {
   struct list_head global_list;   /* Verkettung in der Liste aller *
                                    * PCI-Geräte                    */
   struct list_head bus_list;      /* Verkettung in der Liste des   *
                                    * zugehörigen Busses            */
   struct pci_bus  *bus;           /* Bus, auf dem das              *
                                    * Gerät sitzt                   */
   struct pci_bus  *subordinate;   /* Bus, zu dem das Gerät eine    *
                                    * Brücke bildet                 */
   void            *sysdata;       /* Zeiger für mögliche           *
                                    * Zusatzdaten                   */
   struct proc_dir_entry *procent; /* Geräteeintrag in              *
                                    * /proc/bus/pci                 */
```

```
    unsigned int    devfn;          /* Geräte- und Funktions-        *
                                     * index                         */
    unsigned short  vendor;         /* Hersteller-Code des Gerätes   */
    unsigned short  device;         /* Geräte-Code des Gerätes       */
    unsigned short  subsystem_vendor; /* Hersteller-Code des Subsystems */
    unsigned short  subsystem_device; /* Geräte-Code des Subsystems  */
    unsigned int    class;          /* Geräteklasse                  */
    u8              hdr_type;       /* PCI-Header-Typ                */
    u8              rom_base_reg;   /* Kontrollregister des ROM      */
    struct pci_driver *driver;   /* Zeiger auf zugehörigen Treiber   */
    void         *driver_data;   /* Zeiger auf mögliche Zusatzdaten  *
                                  * des Treibers                     */
    dma_addr_t      dma_mask;    /* Bitmaske der für den DMA-Transfer *
                                  * gültigen Adressbits, üblicherweise *
                                  * 0xFFFFFFFF                       */
    /* Liste der kompatiblen Geräte */
    unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
    unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];

    unsigned int    irq;            /* Interruptnummer               */
    /* I/O-, Speicher- und ROM-Ressourcen */
    struct resource resource[DEVICE_COUNT_RESOURCE];
    struct resource dma_resource[DEVICE_COUNT_DMA];
    struct resource irq_resource[DEVICE_COUNT_IRQ];

    char         name[80];       /* Name des Gerätes */
    char         slot_name[8];   /* Name des Slots, in dem sich das *
                                  * Gerät befindet */
    int          active;         /* ISAPnP: Gerät ist aktiv */
    int          ro;             /* ISAPnP: Ressources sind Read-Only */
    unsigned short regs;         /* ISAPnP: unterstützte Register */

    /* ISAPnP Funktionen */
    int (*prepare)(struct pci_dev *dev);
    int (*activate)(struct pci_dev *dev);
    int (*deactivate)(struct pci_dev *dev);
};
```

Der Inhalt dieser zurückgelieferten Struktur sollte nicht verändert werden, lediglich die Komponente `driver_data` könnte von einem Treiber benutzt werden, um spezifische Daten an ein Gerät zu hängen. Dazu dienen die folgenden Hilfsfunktionen:

```
void *pci_get_drvdata(struct pci_dev *pdev);
void pci_set_drvdata(struct pci_dev *pdev, void *data);
```

Die Ressourcen-Informationen sind in der Struktur `resource` gespeichert:

```
struct resource {
  const char *name;         /* Name der Ressource */
  unsigned long start, end; /* Start und Ende der Ressource */
  unsigned long flags;      /* Flags, die den Typ der Ressource erklären */
  /* dient zur Verkettung von Ressourcen in Baumstrukturen */
```

```
  struct resource *parent, *sibling, *child;
};
```

Um die Komponente `resource` auszulesen, sollten jedoch die folgenden Makros verwendet werden:

```
pci_resource_start(dev,bar)
pci_resource_end(dev,bar)
pci_resource_flags(dev,bar)
pci_resource_len(dev,bar)
```

Wenden wir uns wieder unserem Beispielcode zu:

```
static int __init
pcnet32_probe_pci(struct pci_dev *pdev, const struct pci_device_id *ent)
{
  static int card_idx;
  long ioaddr;
  int err = 0;

  printk(KERN_INFO "pcnet32_probe_pci: found device %#08x.%#08x\n",
    ent->vendor, ent->device);

  ioaddr = pci_resource_start (pdev, 0);
  printk(KERN_INFO "    ioaddr=%#08lx  resource_flags=%#08lx\n",
    ioaddr, pci_resource_flags (pdev, 0));
  if (!ioaddr) {
    printk (KERN_ERR "no PCI IO resources, aborting\n");
    return -ENODEV;
  }

  if (!pci_dma_supported(pdev, PCNET32_DMA_MASK)) {
    printk(KERN_ERR "pcnet32.c: architecture does not support"
                    " 32bit PCI busmaster DMA\n");
    return -ENODEV;
  }

  if ((err = pci_enable_device(pdev)) < 0) {
    printk(KERN_ERR "pcnet32.c: failed to enable device"
                    " -- err=%d\n", err);
    return err;
  }

  pci_set_master(pdev);

  return pcnet32_probe1(ioaddr, pdev->irq, 1, card_idx, pdev);
}
```

Zunächst wird die I/O-Adresse des Karte gelesen. Da der Treiber die PCnet32-Netzwerkkarte kennt, weiß er, dass es sich bei der ersten Ressource um eine I/O-Adresse handelt.

Da die PCnet32-Netzwerkkarte mittels Busmaster-DMA arbeitet, sollte zunächst überprüft werden, ob dies auf der aktuellen Architektur möglich ist. Dazu dient die folgende Funktion:

```
int pci_dma_supported(struct pci_dev *hwdev, dma_addr_t mask);
```

Als zweiten Parameter muss der Funktion die Maske aller möglichen Adressbits, die das PCI-Device generieren kann, übergeben werden. Im Falle der PCnet32-Netzwerkkarte ist dies die Konstante `PCNET32_DMA_MASK`, die als `0xFFFFFFFF` definiert ist, da diese Netzwerkkarte jede mögliche 32-Bit-Adresse generieren kann. Es existieren jedoch einige PCI-Karten, die z. B. nur 24-Bit-Adressen erzeugen können. Für diese Geräte müsste die Konstante `0x00FFFFFF` übergeben werden. Liefert die Funktion `pci_dma_supported()` den Wert 0 zurück, kann auf der aktuellen Architektur dieser DMA-Zugriff nicht durchgeführt werden.

Wird nun der DMA-Zugriff unterstützt, so aktiviert der Treiber das Gerät. Dies geschieht mit Hilfe der folgenden Funktion:

```
int pci_enable_device(struct pci_dev *dev);
```

Diese Funktion aktiviert die Ressourcen der Karte, falls sie noch nicht aktiviert wurden. Diese Funktion kann fehlschlagen, wenn es zu Ressourcekonflikten kommt. Dies sollte jedoch nicht geschehen, wenn das PCI-BIOS fehlerfrei ist.

Bei busmasterfähigen Karten erlaubt es die PCI-Spezifikation, dass der Busmaster-Mode zunächst ausgeschaltet ist. Wenn das Gerät im Busmaster-Betrieb genutzt werden soll, muss also das Busmaster-Bit im PCI-Commandregister gesetzt sein. Dazu dient die folgende Funktion:

```
void pci_set_master(struct pci_dev *dev);
```

Diese Funktion setzt das Master-Bit und erhöht zusätzlich den PCI-Latency-Counter auf 64, falls er vorher auf einen Wert kleiner 16 eingestellt war.

Zu beachten ist, dass die Belegung der Interruptleitungen nicht notwendigerweise exklusiv ist; die PCI-Spezifikation schreibt sogar vor, dass ein Treiber mit gemeinsamen Interrupts von verschiedenen Geräten klarkommen muss (siehe auch Abschnitt 7.3.3).

Kompatibilitätsmodus

Linux 2.4 unterstützt auch noch weiterhin die älteren, nicht Hot-Plugin fähigen PCI-Zugriffsfunktionen. Diese sollten jedoch nicht mehr für neuere Treiberimplementationen genutzt werden. Deshalb folgt hier nur ein kurzer Überblick:

```
struct pci_dev *pci_find_device(unsigned int vendor, unsigned int device,
  const struct pci_dev *from);
struct pci_dev *pci_find_subsys(unsigned int vendor, unsigned int device,
  unsigned int ss_vendor, unsigned int ss_device,
  const struct pci_dev *from);
```

```
struct pci_dev *pci_find_class(unsigned int class,
  const struct pci_dev *from);
```

`pci_find_device()` erlaubt es, nach einem Gerät im PCI-Adressraum zu suchen. Notwendig dazu sind der Herstellercode `vendor` und der Gerätecode `device`. Falls ein Gerät gefunden wird, so liefert diese Funktion einen Zeiger auf die `pci_dev`-Struktur des Gerätes zurück. Um nach weiteren Geräten zu suchen, kann `pci_find_device` erneut aufgerufen werden. Dazu muss als letzter Parameter der Zeiger auf die `pci_dev`-Struktur des zuletzt gefundenen Gerätes angegeben werden, NULL beginnt die Suche von vorn. `pci_find_subsys` erlaubt zusätzlich einen Subsystem-Hersteller und -Gerätecode zu spezifizieren. `pci_find_class` schließlich sucht nach einer Geräteklasse im PCI-Adressraum.

Zusätzlich besteht die Möglichkeit, durch die Liste aller PCI-Geräte des Systems zu gehen. Dazu dient der folgende Code:

```
struct pci_dev *dev;

pci_for_each_dev(dev) {
  /* Schleife, wird für jedes Gerät durchlaufen wird */
}
```

Zusätzlich zum Makro `pci_for_each_dev` existiert noch eine Version `pci_for_each_dev_reverse`, die die Liste von hinten nach vorn durchläuft. Dies war die Reihenfolge in älteren Linux-Versionen.

Zugriff auf den Konfigurationsraum

Um auf den Konfigurationsraum zugreifen zu können, stellt Linux die folgenden Funktionen bereit:

```
int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *val);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val);
int pci_write_config_byte(struct pci_dev *dev, int where, u8 val);
int pci_write_config_word(struct pci_dev *dev, int where, u16 val);
int pci_write_config_dword(struct pci_dev *dev, int where, u32 val);
```

Diese Funktionen sollten jedoch nicht benutzt werden, um die Ressourcen eines PCI-Gerätes abzufragen, sondern nur zum Lesen und Schreiben weiterer Konfigurationsregister.

Zugriff auf den Geräteadressraum

Bevor auf den I/O-Adressraum eines PCI-Gerätes zugegriffen werden kann, sollte dieser mittels `request_region()` alloziert werden. Speicherbereiche sind analog dazu mittels des Makros

```
request_mem_region(from, num, name)
```

zu allozieren. Auf diese Weise kann sichergestellt werden, dass ausschließlich der Treiber auf die Ressourcen des PCI-Gerätes zugreift.

Um nun auf den Speicher eines PCI-Gerätes zugreifen zu können, muss dieser zunächst in den virtuellen Adressraum der CPU eingeblendet werden. Dies geschieht mit Hilfe der Funktionen `ioremap()` und `iounmap()`. `ioremap()` liefert dabei eine virtuelle Adresse zurück, über die direkt auf den Speicher zugegriffen werden kann.

Weiterhin kann es notwendig werden, die Speicheradressen in die jeweilige Sicht „zu übersetzen“. Das Problem besteht darin, dass in einem heutigen PC zumindest drei Arten von Speicheradressen existieren:

Physische Adressen Dies sind die Adressen, wie sie außerhalb der CPU auf dem Speicherbus existieren.

Virtuelle Adressen Diese Adressen existieren nur innerhalb der CPU. Beim Zugriff auf virtuelle Adressen führt die CPU automatisch eine Umwandlung in physische Adressen durch.

Busadressen Diese Adressen entsprechen physischen Adressen „von der anderen Seite“ des PCI-Busses. Die Busadresse ist also die, welche ein PCI-Gerät generieren muss, um eine Speicherzelle auslesen zu können.

Während in der x86-Architektur die Busadresse identisch der physischen Adresse ist, gilt dies leider nicht für alle von Linux unterstützten Architekturen. Deshalb muss bei der Adressumrechnung besondere Sorgfalt verwandt werden, um den Treiber kompatibel mit allen Architekturen schreiben zu können. Diesem Zwecke dienen die folgenden Funktionen:

```
unsigned long virt_to_phys(volatile void * address);
unsigned long phys_to_virt(volatile void * address);
unsigned long virt_to_bus(volatile void * address);
unsigned long bus_to_virt(volatile void * address);
```

Virtuelle Adressen werden benötigt, um vom Kern aus auf Speicherbereiche zugreifen zu können. Busadressen werden benötigt, um ein PCI-Gerät auf den Speicher zugreifen lassen zu können. Physische Adressen werden niemals direkt benötigt, sondern nur als Parameter für Funktionen des Speichermanagements.

7.2.3 Der Dinosaurier — ISA Bus

Automatische Hardwareerkennung

Der vielbeschimpfte ISA-Bus ist wegen seiner historischen Bedeutung immer noch auf vielen PC-Boards vorhanden, heute sind jedoch nur noch wenige Karten für diesen Bus im Umlauf. Bei neueren Boards ist der ISA über eine Bridge am PCI-Bus realisiert, was viele Probleme beseitigen konnte, leider aber auch neue schafft. Da außerdem durch das

Design des ISA-Busses für die möglichen Portadressen Grenzen gesetzt sind[5], kommt es häufig zu Adressüberschneidungen.

Das wohl geläufigste Beispiel war die Belegung der I/O-Adresse der COM4-Schnittstelle durch ISA-Karten mit S3-Chip.

Zudem hatte die Marktentwicklung dazu geführt, dass durch verschiedenste Hardware die gleichen I/O-Adressräume belegt wurden. Meist konnte man noch mittels *Jumpern* verschiedene Basisadressen auswählen. Dies war zwar oft nötig, verwirrte aber unbedarfte Nutzer, da sich in Dokumentationen meist nur der Hinweis befand, man sollte „die Standardbelegung beibehalten und im Falle eines Nichtfunktionierens Jumper XX auf Stellung YY“ setzen.

Bei der Entwicklung eines Treibers hat man also zunächst die Möglichkeit des „sicheren“ Weges. Sämtliche Parameter werden vor dem Compilieren fest eingestellt. Das ist zwar sehr sicher, aber nicht sehr komfortabel. Wer will schon jedesmal den Kern neu übersetzen, wenn er einen Jumper umgesteckt hat?

Es sind also Algorithmen gesucht, die Hardware „erkennen“. Im Idealfall müsste eine solche Erkennung allein durch Auslesen von I/O-Ports möglich sein, aber leider war das bei der Entwicklung von Hardware oft keine Option. Man ist also gezwungen, ins Blaue hinein Werte zu schreiben, I/O-Ports auszulesen und davon abhängig seine Entscheidung zu treffen. Meist werden dabei gewisse Besonderheiten einzelner Chips ausgenutzt (sprich Bugs bzw. „unbenutzte Features“), die dann dazu führen können, dass die kompatible Hardware eines anderen Herstellers nicht erkannt wird.

Das bei weitem unangenehmste Problem ist aber, dass das „Probeschreiben“ die Funktionsweise anderer Hardware hemmen bzw. das System zum Absturz bringen kann. Der zweite Fall tritt häufig bei der Entwicklung eines Treibers auf, denn meist bemerkt man das Nichtfunktionieren eines anderen Geräts erst viel später.

Will ein Gerätetreiber also I/O-Ports testen, sollte zunächst mit Hilfe des Makros `check_region()` die Erlaubnis dazu eingeholt werden. Dazu wollen wir ein Fragment des Skeleton für ISA-Netzwerktreiber betrachten.

```
#include <linux/ioport.h>

netcard_probe(struct device *dev)
{
   …
   for (i = 0; netcard_portlist[i]; i++) {
      int ioaddr = netcard_portlist[i];
      if (check_region(ioaddr, NETCARD_IO_EXTENT))
         continue;
      if (netcard_probe1(dev, ioaddr) == 0)
         return 0;
   }
```

5 Die meiste PC-Hardware decodiert nur die ersten 10 Bit einer Portadresse. Das bedeutet, dass alle 65.536 möglichen Portadressen auf den Bereich 0-0x3ff abgebildet werden.

```
   return -ENODEV;
}
```

Liefert `check_region()` also einen Wert ungleich 0, darf auf mindestens einen Port in diesem Bereich nicht zugegriffen werden, und ein Test ist zu unterlassen. Hat ein Treiber seine Hardware eindeutig identifiziert, sollte er die zugehörigen I/O-Ports mit Hilfe des Makros `request_region()` sperren.

Automatische Interrupterkennung

Auf vielen ISA-Erweiterungskarten muss der verwendete IRQ mittels Jumpern gesetzt werden. Erst Erweiterungen wie PCI oder *Plug-and-Play* erlauben das Einstellen und Auslesen der Konfiguration von Erweiterungskarten. Es stellt sich somit das Problem, die verwendeten IRQs während der Initialisierung des Kerns zu bestimmen. Da eine automatische Interrupterkennung jedoch einen Unsicherheitsfaktor darstellt und zum Absturz des Systems führen kann, sollte sie beim Laden von Modulen unterbleiben.

Die Vorgehensweise bei der Ermittlung verwendeter IRQs ist dabei eigentlich immer gleich. Es werden einfach alle möglichen IRQs belegt, das anzusprechende Gerät bzw. die Erweiterungskarte zur Auslösung eines IRQs „genötigt“ und — falls nur ein IRQ von den zuvor allozierten angesprochen wurde —, haben wir ihn mit hoher Wahrscheinlichkeit gefunden. Alle anderen IRQs müssen nun nur noch freigegeben werden.

Linux stellt jedoch Funktionen bereit, die diese Erkennung vereinfachen. Sehen wir uns zunächst als Beispiel ein Fragment aus dem WaveFront-Soundtreiber an:

```
unsigned long irq_mask;
short reported_irq;

irq_mask = probe_irq_on ();

outb (0x0, dev.control_port);
outb (0x80 | 0x40 | bits, dev.data_port);
wavefront_should_cause_interrupt(0x80|0x40|0x10|0x1,
                        dev.control_port,
                        (reset_time*HZ)/100);

reported_irq = probe_irq_off (irq_mask);

if (reported_irq != dev.irq) {
        if (reported_irq == 0) {
           printk(KERN_ERR LOGNAME
                   "No unassigned interrupts detected "
                   "after h/w reset\n");
        } else if (reported_irq < 0) {
           printk(KERN_ERR LOGNAME
                   "Multiple unassigned interrupts detected "
                   "after h/w reset\n");
        } else {
           printk(KERN_ERR LOGNAME "autodetected IRQ %d not the "
```

```
                    "value provided (%d)\n", reported_irq,
                    dev.irq);
        }
        dev.irq = -1;
        return 1;
} else {
        printk (KERN_INFO LOGNAME "autodetected IRQ at %d\n",
           reported_irq);
}
```

Die IRQ-Erkennung wird mit einem Aufruf der Funktion `probe_irq_on()` eingeleitet. Sie liefert als Rückgabewert `irqs` eine Bitmaske zurück, in der alle derzeit freien und zur Erkennung genutzen IRQ-Nummern codiert sind. Danach wird eine Funktion aufgerufen, die einen IRQ durch die Soundkarte auslöst. Diese Funktion sollte auch eine kurze Zeit warten, um die Zeit bis zur Auslösung des Interrupts durch die Hardware zu überbrücken. Danach beendet der Aufruf von `probe_irq_off()` die Erkennung. Als Argument muss diese Funktion die von `probe_irq_on()` gelieferte Bitmaske erhalten und liefert als Rückgabewert die Nummer des aufgetretenen IRQs. Ist diese Nummer kleiner 0, ist mehr als ein IRQ aufgetreten. Dies kann ein Zeichen für eine falsch konfigurierte Karte oder einen anderen Hardwarekonflikt sein. Man könnte jetzt die Erkennung noch einmal versuchen, oder wie im Beispiel, aufgeben. Der Wert 0 zeugt davon, dass kein IRQ aufgetreten ist, zum Beispiel weil überhaupt kein IRQ-Jumper gesetzt wurde. Auch in diesem Fall muss der Nutzer eingreifen. Nur ein positiver Rückgabewert zeigt den eindeutig erkannten IRQ an.

Die Erkennung von DMA-Kanälen ist schwieriger. Glücklicherweise unterstützen die meisten Karten nur wenige DMA-Kanäle, oder diese sind durch Konfigurationsregister auswählbar. Hat man diese Möglichkeiten nicht, sollte der DMA-Kanal durch Setup-Parameter eingestellt werden. Man kann auch versuchen, einfach alle noch möglichen Kanäle zu allozieren und einen DMA-Transfer auszulösen. Dies geht jedoch nur, wenn die Hardware eine Möglichkeit bietet, um festzustellen, ob der Transfer geglückt ist.

DMA-Betrieb

Sollen besonders viele Daten *kontinuierlich* von bzw. zu einem Gerät transportiert werden, bietet sich der DMA-Betrieb an.

In dieser Betriebsart transferiert der *DMA-Controller* Daten ohne Mithilfe des Prozessors direkt aus dem Speicher zu einem Gerät. Normalerweise löst das Gerät danach einen IRQ aus, so dass in der behandelnden ISR der nächste DMA-Transfer vorbereitet werden kann. Diese Arbeitsweise ist ideal für das Multitasking, da die CPU während des Datentransfers andere Aufgaben übernehmen kann. Leider gibt es auch Beispiele für DMA-fähige Geräte, die keinen IRQ unterstützen. Einige Handscanner fallen in diese Kategorie. Will man einen Gerätetreiber für diese Klasse schreiben, muss man den DMA-Controller pollen, um das Ende eines Transfers festzustellen.

Außerdem muss man beim DMA-Betrieb von Geräten mit Problemen ganz anderer Art kämpfen, die zum Teil aus der Kompatibilität zum Ur-PC stammen.

- Da der DMA-Controller unabhängig vom Prozessor arbeitet, kennt er nur physische Adressen.
- Das Basisadressregister des DMA-Controllers ist nur 16 Bit breit. Dadurch kann kein DMA-Transfer über eine 64-KB-Grenze hinweg durchgeführt werden. Da der erste im AT vorhandene Controller einen 8-Bit-Transfer durchführt, können mit Hilfe der ersten vier DMA-Kanäle nicht mehr als 64 KB auf einmal transferiert werden. Der zweite im AT vorhandene DMA-Controller führt einen 16-Bit-Transfer durch, d.h., in einem Zyklus werden zwei Bytes übertragen. Da das Basisadressregister auch hier nur 16 Bit breit ist, hängt der zweite Controller eine 0 an, der Transfer muss also stets auf geraden Adressen beginnen (der Registerinhalt wird also mit 2 multipliziert). Dadurch kann der zweite Controller maximal 128 KB transferieren, aber keine 128-KB-Grenze überschreiten.
- Zusätzlich zum Basisadressregister existiert ein DMA-Pageregister, welches die Adressbits ab A15 aufnimmt. Da dieses Register im AT nur 8 Bit breit ist, kann der DMA-Transfer nur innerhalb der ersten 16 MB durchgeführt werden. Obwohl diese Beschränkung durch den EISA-Bus und viele Chipsätze (dort leider nicht kompatibel) aufgehoben wurde, unterstützt Linux dies nicht.

Um diese Probleme zu lösen, belegte z. B. der Soundtreiber in früheren Linux-Versionen mit Hilfe einer speziellen Funktion Puffer für den DMA-Transfer zur Soundkarte.

Da im Protected Mode das DMA-Konzept durch die notwendigen physischen Adressen gestört wird, kann nur das Betriebssystem bzw. Gerätetreiber DMA nutzen. So kopiert der Soundtreiber die Daten erst mit Hilfe des Prozessors in die DMA-Puffer und transferiert sie dann mittels DMA zur Soundkarte. Obwohl dieses Vorgehen eigentlich der Idee widerspricht, Daten ohne Hilfe des Prozessors zu übertragen, ist es trotzdem sinnvoll, da man sich nicht um ein Timing bei der Datenübertragung zur Soundkarte bzw. anderen Geräten kümmern muss. Wir werden im Folgenden noch näher auf das DMA-Konzept eingehen.

Ein Beispiel für den DMA-Betrieb

Um näher auf das DMA-Konzept einzugehen, müssen wir uns zunächst mit der Programmierung des DMA-Controllers beschäftigen. Es soll hier jedoch nur eine kurze Einleitung gegeben werden, für weitergehende Informationen sei [Mes93] empfohlen.

Wie bereits erwähnt, besitzt der DMA-Controller ein Basisadressregister, welches die unteren 16 Bit der Adresse des zu übertragenden Speicherbereiches enthalten. Ein zweites 16-Bit-Register, das Basiszählwertregister, enthält die Anzahl der durchzuführenden Datentransfers. Dieses Register wird bei jedem Datentransfer dekrementiert, das Erreichen des Wertes 0xFFFF wird als *Terminal Count* (TC) bezeichnet. Jeder DMA-Controller besitzt vier Kanäle, jedem Kanal ist sowohl ein Basisadressregister als auch ein Basiszählwertregister zugeordnet. Jedem Kanal ist ein Eingabesignal DREQx sowie ein Ausgabesignal DACKx zugeordnet. Ein Gerät fordert eine DMA-Übertragung an, indem es das DREQ-Signal aktiviert. Hat der DMA-Controller die Kontrolle über den Bus erlangt,

zeigt er dies durch das DACK-Signal an. Zu einem gegebenen Zeitpunkt kann jedoch nur maximal ein DACK aktiv sein, die einzelnen DREQ-Signale besitzen deshalb verschiedene Prioritäten. Üblicherweise besitzt DREQ0 die höchste und DREQ3 die niedrigste Priorität. Mit Hilfe des Request-Registers lässt sich außerdem noch der DMA-Transfer „von Hand" aktivieren, als ob ein entsprechendes DREQ-Signal eingetroffen wäre. Diese Möglichkeit wird jedoch normalerweise nicht ausgenutzt. Sie diente u.a. im PC/XT dazu, einen Speicher-zu-Speicher-Transfer zu ermöglichen, der jedoch seit dem PC-AT nicht möglich ist, da der DMA-Kanal 0 des Master-Controllers, der für diesen Modus notwendig ist, zur Kaskadierung des Slave-Controllers verwendet wird.

Insgesamt besitzt jeder DMA-Controller 12 verschiedene Register, die die Arbeitsweise regeln. Die Funktionen im Linux-Kern kapseln diese Register jedoch völlig, so dass wir hier auf eine weitere Erklärung verzichten.

Der DMA-Controller unterstützt zudem noch verschiedene Transfermodi, die über das Modusregister eines jeden Kanals eingestellt werden müssen. Dazu zählen die folgenden Betriebsarten.

Demandtransfer In dieser Betriebsart überträgt der DMA-Controller so lange Daten, bis der Terminal Count erreicht oder das Gerät DREQ deaktiviert. Danach wird der Transfer unterbrochen, bis das Gerät DREQ wiederum aktiviert.

Einzeltransfer In dieser Betriebsart überträgt der DMA-Controller nur jeweils einen Wert und übergibt den Bus wieder dem Prozessor. Jeder weitere Transfer muss durch ein DREQ-Signal oder einen Zugriff aus das Request-Register angefordert werden. Diese Betriebsart wird für langsame Geräte wie Floppy oder Scanner verwendet.

Blocktransfer In dieser Betriebsart führt der DMA-Controller eine Blockübertragung durch, ohne den Bus abzugeben. Die Übertragung wird mit einem DREQ eingeleitet.

Cascade Kaskadierung eines weiteren DMA-Controllers. In dieser Betriebsart leitet der DMA-Controller eintreffende DMA-Requests durch und erlaubt so die Benutzung von mehr als einem Controller. Standardmäßig befindet sich der DMA-Kanal 0 des zweiten Controllers (bzw. DMA-Kanal 4 bei durchgehender Zählung) in dieser Betriebsart, da er der Master im AT ist.

Diese grundsätzlichen Modi können sowohl im Lese- als auch im Schreibtransfer zum Einsatz kommen. Der DMA-Controller kann dabei die Speicheradressen sowohl inkrementieren als auch dekrementieren, so dass auch ein Transfer beginnend mit der höchsten Adresse möglich ist. Zusätzlich kann die Autoinitialisierung ein- und ausgeschaltet werden. Ist sie eingeschaltet, wird der entsprechende DMA-Kanal nach Erreichen des Terminal Counts wieder automatisch mit den Anfangswerten initialisiert. Dadurch kann man immer gleich bleibende Datenmengen in einen bzw. aus einem festen Puffer im Speicher übertragen.

Betrachten wir als Beispiel für den DMA-Betrieb die Implementierung eines Treibers für einen Handscanner. Ebenso wie zu verwendende IRQs muss zunächst der zu verwendende DMA-Kanal alloziert werden.

```
if ( (err = request_dma(AC4096_DMA, AC4096_SCANNER_NAME)) ) {
   printk("AC 4096: unable to get DMA%d\n", AC4096_DMA);
   return err;
}
```

Die Funktionen `request_dma()` und `free_dma()` arbeiten anlog zu den bereits beschriebenen `request_irq()` und `free_irq()`. `request_dma()` erwartet die Nummer des DMA-Kanals sowie den Namen des Treibers, der diesen Kanal benutzen möchte. Dieser Name wird jedoch nur vom *Proc*-Dateisystem ausgewertet. DMA-Kanäle sollten ebenso wie IRQs nur dann alloziert werden, wenn sie demnächst benutzt werden; normalerweise geschieht dies in der `open`-Funktion eines Gerätetreibers. Benutzt ein Treiber sowohl IRQ- als auch DMA-Kanäle, so sollte zunächst der Interrupt und dann der DMA-Kanal alloziert werden.

Das Belegen von Puffern kann ebenfalls in der `open`-Funktion, aber auch erst in `read()` oder `write()` erfolgen, da der Speicher eine weit weniger kritische Ressource ist. Seit der Linux-Version 1.2 ist es nicht mehr notwendig, Puffer für DMA-Transfer während des Hochfahrens des Systems dauerhaft zu belegen. Damit lassen sich nun auch Gerätetreiber als Module realisieren, die DMA-Transfer benutzen. Die Linux-Speicherverwaltung sorgt selbstständig dafür, dass für DMA-Puffer allozierter Speicher unterhalb der 16-MB-Grenze liegt und keine 64-KB-Grenze überschritten wird. Dazu muss bei der Allokation von Speicher die Funktion `kmalloc()` verwendet und ihr zusätzlich das Flag `GFP_DMA` übergeben werden.

```
tmp = kmalloc(blksize + HEADERSIZE, GFP_DMA | GFP_KERNEL);
```

Jetzt kann der DMA-Transfer eingeleitet werden. Wie bereits erwähnt, kapseln die dazu vorhandenen Funktionen die Hardware weitgehend ein, so dass der DMA-Transfer einfach zu programmieren ist. Normalerweise wird man sie sogar stets in der im folgenden Beispiel zu sehenden Reihenfolge benutzen.

```
static void start_dma_xfer(char *buf)
{
   unsigned long flags;

   flags = claim_dma_lock(void);
   disable_dma(AC4096_DMA);
   clear_dma_ff(AC4096_DMA);
   set_dma_mode(AC4096_DMA, DMA_MODE_READ);
   set_dma_addr(AC4096_DMA, (unsigned int) buf);
   set_dma_count(AC4096_DMA, hw_modeinfo.bpl);
   enable_dma(AC4096_DMA);
   release_dma_lock(flags);
}
```

Die Funktion `disable_dma()` schaltet den DMA-Transfer auf dem ihr als Argument übergebenen Kanal ab. Danach kann die Programmierung des DMA-Controllers erfolgen. `clear_dma_ff()` löscht das *DMA Pointer Flip Flop*. Da der DMA-Controller nur 8-Bit-Datenports hat, müssen Zugriffe auf interne 16-Bit-Register zerlegt werden. Das

DMA Pointer Flip Flop gibt Auskunft, ob der nächste Wert als LSB oder MSB interpretiert werden soll. Nach jedem Löschen erwartet der DMA-Controller zunächst das LSB. Da die Aufrufe von `set_dma_addr()` und `set_dma_count()` darauf vertrauen, sollte `clear_dma_ff()` einmal vor der Benutzung dieser Funktionen aufgerufen werden. `set_dma_mode()` setzt die Betriebsart des DMA-Kanals. Die von Linux mit Hilfe von bereits definierten Makros unterstützten Betriebsarten sind:

DMA_MODE_READ Einzeltransfer ohne Autoinitialisierung vom Gerät in den Speicher, Adressen werden inkrementiert

DMA_MODE_WRITE Einzeltransfer ohne Autoinitialisierung aus dem Speicher zum Gerät, Adressen werden inkrementiert

DMA_MODE_CASCADE Kaskadierung eines anderen Controllers

Diese Modi reichen jedoch für die meisten Fälle bereits aus.

Bleibt noch, mittels der Funktion `set_dma_addr()` die Adresse des Pufferbereichs und mittels `set_dma_count()` die Anzahl der zu übertragenden Bytes zu setzen. Beide Funktionen sorgen selbstständig für die passende Umsetzung der ihnen übergebenen Werte für den DMA-Controller und erwarten deshalb gerade Adressen bzw. eine gerade Byteanzahl, falls ein DMA-Kanal des zweiten Controllers verwendet wird. Die Funktionen `claim_dma_lock()` und `release_dma_lock()` sorgen dafür, dass die Programmierung des DMA-Controllers weder von einem Interrupt, noch von einem anderen Kernelthread unterbrochen wird.

Erzeugt das Gerät einen Interrupt nach beendeter Übertragung, so ist eine ISR zu implementieren, die der für den reinen Interruptbetrieb gleicht. Nach einem eventuellen Test, ob der Interrupt auch wirklich vom betreffenden Gerät ausgelöst wurde, muss der wartende Prozess mit Hilfe von `wake_up_interruptible()` geweckt werden und, falls noch Daten zu transferieren sind, der nächste DMA-Transfer eingeleitet werden.

Erzeugt das Gerät wie in unserem Fall keinen Interrupt, muss der DMA-Controller abgefragt werden, ob das Ende des DMA-Transfers erreicht ist. Dazu dient das Statusregister des entsprechenden DMA-Controllers. Die unteren 4 Bit des Registers zeigen an, ob der entsprechende Kanal einen Terminal Count erreicht hat. Ist das Bit gesetzt, wurde der TC erreicht und der Transfer ist beendet. Jedes Auslesen des Statusregisters löscht jedoch diese Bits wieder. Die folgende Funktion kann zur Abfrage benutzt werden.

```
int dma_tc_reached(int channel)
{
   if (channel < 4)
      return ( inb(DMA1_STAT_REG) & (1 << channel) );
   else
      return ( inb(DMA2_STAT_REG) & (1 << (channel & 3)) );
}
```

Sie kann in einer Pollingroutine verwendet werden, beispielsweise so:

```
int dma_polled(void)
{
   unsigned long count  = 0;

   do {
      count ++;
      if (current->need_resched)
         schedule();
   } while (!dma_tc_reached(dma_channel) && count < TIMEOUT );
   …
}
```

Abhängig vom entsprechenden Gerät, kann es auf diese Weise jedoch zu einem Datenverlust kommen, da die Zeit zu der nächsten Aktivierung des Prozesses (d.h. bis zur Rückkehr aus `schedule()`) nicht vorhersagbar ist. Bei einem Scanner kann es dadurch zum Verlust von Scanzeilen kommen, falls dieser keinen oder nur einen sehr kleinen Puffer besitzt. Deshalb wurde in unserem Beispiel ein anderer Weg beschritten. Der DMA-Controller wird in einer Timerroutine abgefragt, die 50 mal pro Sekunde aufgerufen wird. Diese Routine arbeitet genauso wie eine entsprechende ISR, nur statt des Tests, ob das Gerät den Interrupt ausgelöst hat, wird getestet, ob der DMA-Transfer beendet wurde.

```
static inline void start_snooping(void)
{
   timer.expires  = jiffies + 2;
   timer.function = test_dma_rdy;
   add_timer(&timer);
}

static void test_dma_rdy(unsigned long dummy)
{
   static int needed_bytes;
   char cmd;

   if (! xfer_going) return;
   start_snooping(); /* Timer erneut starten */

   if ( dma_tc_reached(AC4096_DMA) ) {
      …

      stop_scanner(); /* Scanner anhalten */

      /* Falls noch ein genügend großer Puffer frei ist */
      if (WR_BUFSPC >= hw_modeinfo.bpl) {
         …
         /* nächsten DMA-Transfer einleiten */
         start_dma_xfer(WR_ADDR);
      }
      else xfer_going = 0;
   }
}
```

7.2.4 ISA-PnP

Die wohl übelste Geißel, die dem ISA-Bus (und Linux) angetan wurde, ist ISA-Plug and Play (PnP)[6]. Die Grundidee an PnP ist, die Karten beim ersten Zugriff zu initialisieren und die Parameter wie Basisadresse, IRQ und DMA-Kanal während des Bootvorgangs einzustellen. Dazu müssen besondere Adressen reserviert sein, auf denen die Karte detektiert und initialisiert wird. Genau hier liegt das Problem: Beim ISA Bus besteht keine Möglichkeit, Hardwarekonflikte zu erkennen und auszuschließen, deswegen kann auch nicht ausgeschlossen werden, dass sich auf der soeben geprobten Adresse nicht doch der Harddiskcontroller XY befand und nun fleißig die Festplatte formatiert (das ist zwahr sehr unwahrscheinlich …aber …).

Linux 2.4 unterstützt nun ISA-PnP bereits im Kernel. Somit ist es möglich, Treiber für ISA-PnP Karten fest in den Kernel einzubinden. Vor der Version 2.4 musste das `isapnptools`-Paket von Peter Fox benutzt werden, um PnP-Karten zu initialisieren; die passenden Treiber mussten dann als Modul geladen werden, wobei die Initialisierungsparameter als Parameter angegeben wurden. Die ISA-PnP-Handhabung wurde an die PCI-Bus-Handhabung angelehnt, es wird sogar dieselbe Struktur `pci_dev` benutzt, um die allozierten Ressourcen aufzunehmen.

ISA-PnP-Geräte suchen

In der PnP-Welt werden die Begriffe *Gerät* und *Funktion* benutzt, um Hardware-Einheiten zu kennzeichnen. Dem Begriff Gerät entspricht dabei eine komplette Funktionseinheit, z. B. eine PnP-Einsteckkarte. Jedes Gerät besitzt mindestens eine oder mehrere Funktionen. Funktionen sind dabei die kleinsten Hardwareeinheiten. So kann eine Soundkarte z. B. die Funktionen Audio-DSP und Gameport beinhalten.

Bevor nun ein Gerät angesprochen werden kann, muss der Treiber zunächst feststellen, ob ein solches Gerät überhaupt vorhanden ist. Dazu dient die folgende Funktion:

```
struct pci_bus *isapnp_find_card(unsigned short vendor,
                                 unsigned short device,
                                 struct pci_bus *from);
```

Die Funktion `isapnp_find_card` durchsucht den ISA-PnP-Bus nach einer PnP-Karte. Die Karte wird dabei über einen Hersteller- und einen Gerätecode erkannt. Diese beiden Codes sind jeweils 16 Bit breit. Der Herstellercode ist dabei ein String aus 3 Buchstaben, die jedoch mit jeweils 5 Bit bzw. 6 Bit codiert sind. Das Makro `ISAPNP_VENDOR` erzeugt diese Darstellung aus dem Herstellercode. Die Codierung des Gerätecodes wird vom Makro `ISAPNP_DEVICE` erzeugt. Der dritte Parameter gibt den Startpunkt der Suche an; `NULL` beginnt die Suche mit der ersten PnP-Karte. Ist der Parameter `from` nicht `NULL`, so beginnt die Suche nach dem in `from` angegebenen Gerät.

Ist das Gerät gefunden, kann nach Funktionen gesucht werden. Dazu dient die folgende Funktion:

6 Wegen der häufigen Probleme auch „Plug and Pray“ genannt.

```
struct pci_dev *isapnp_find_dev(struct pci_bus *card,
                                unsigned short vendor,
                                unsigned short function,
                                struct pci_dev *from);
```

Der Parameter `card` beschreibt das Gerät, auf dem nach der Funktion gesucht werden soll. Dabei handelt es sich üblicherweise um den Returnwert der `isapnp_find_card`-Funktion. Ist dieser Parameter gleich `NULL`, wird auf dem gesamten ISA-PnP Bus über Gerätegrenzen hinweg nach der Funktion gesucht. Die Funktion wiederum wird durch die zwei 16-Bit-Codes `vendor` und `function` beschrieben, wobei das Makro `ISAPNP_FUNCTION` zur Erzeugung des Funktionscodes zur Anwendung kommt.

Alternativ ist es auch möglich, ein Gerät ähnlich wie beim PCI-Bustreiber per Callback-Funktion zu suchen. Diesem Zweck dienen die folgenden Strukturen und Funktionen:

```
struct isapnp_card_id {
  /* beliebige Treiberdaten */
  unsigned long driver_data;
  /* Hersteller- und Gerätecode */
  unsigned short card_vendor, card_device;
  struct {
    /* Funktionscode */
    unsigned short vendor, function;
  } devs[ISAPNP_CARD_DEVS];
};

struct isapnp_device_id {
  /* Hersteller- und Gerätecode */
  unsigned short card_vendor, card_device;
  /* Funktionscode */
  unsigned short vendor, function;
  /* beliebige Treiberdaten */
  unsigned long driver_data;
};

int isapnp_probe_cards(const struct isapnp_card_id *ids,
  int (*probe)(struct pci_bus *_card,
               const struct isapnp_card_id *_id));

int isapnp_probe_devs(const struct isapnp_device_id *ids,
  int (*probe)(struct pci_dev *dev,
               const struct isapnp_device_id *id));
```

Die Funktion `isapnp_probe_cards` durchsucht die Liste aller PnP-Geräte nach einem bestimmten Gerät mit einer oder mehreren bestimmten Funktionen und ruft für jede matchende Kombination die Funktion `probe()` auf, die als Parameter übergeben wird. Soll dabei ein Hersteller oder ein Gerätecode nicht gematched werden, so ist der Wert `ISAPNP_ANY_ID` zu verwenden. Die Liste der zu suchenden Funktionen wird mit den Werten 0 für die Komponenten `vendor` und `function` beendet. Die `probe()`-Funktion sollte einen Wert größer oder gleich NULL zurückliefern, falls die Initialisierung

des Gerätes positiv war. `isapnp_probe_cards` liefert als Rückgabewert die Anzahl der gematchen und positiv „probierten“ Geräte zurück.

Die Funktion `isapnp_probe_devs` arbeitet analog, sucht jedoch nur nach einem Gerät mit einer Funktion.

Da derzeit noch kein Treiber die letztgenannten Funktionen benutzt, müssen wir uns mit dem Beispiel aus der ISA-PnP-Dokumentation (`Documentation/isapnp.txt`) begnügen:

```
static struct isapnp_card_id card_ids[] __devinitdata = {
  {
    ISAPNP_CARD_ID('A','D','V', 0x550a),
    devs: {
      ISAPNP_DEVICE_ID('A', 'D', 'V', 0x0010),
      ISAPNP_DEVICE_ID('A', 'D', 'V', 0x0011)
    },
    driver_data: 0x1234,
  },
  {
    ISAPNP_CARD_END,
  }
};
ISAPNP_CARD_TABLE(card_ids);
```

Das Makro `ISAPNP_CARD_ID` generiert den Hersteller- sowie den Gerätecode. Weiterhin wird nur nach Geräten gesucht, die mindestens 2 Funktionen aufweisen. Die Hersteller- und Funktionscodes erzeugt das Makro `ISAPNP_DEVICE_ID`. Das Makro `ISAPNP_CARD_END` schließlich erzeugt einen Eintrag mit Hersteller- und Gerätecode Null und sorgt so für den Abschluss der Suchliste. Das Makro `ISAPNP_CARD_TABLE` sorgt schließlich dafür, dass die Tabelle exportiert wird, falls der Treiber als Modul kompiliert wird. Auf diese Weise könnte ein im Usermode arbeitender Daemon feststellen, welche Module zu laden sind.

Das zweite Beispiel zeigt die Verwendung der Funktion `isapnp_probe_devs`:

```
static struct isapnp_device_id device_ids[] __devinitdata = {
  { ISAPNP_DEVICE_SINGLE('E','S','S', 0x0968, 'E','S','S', 0x0968), },
  { ISAPNP_DEVICE_SINGLE_END, }
};
MODULE_DEVICE_TABLE(isapnp, device_ids);
```

Das Makro `ISAPNP_DEVICE_SINGLE` erzeugt eine komplette Funktionsdefinition bestehend aus Hersteller- und Gerätecode sowie Funktionscode. Das Makro `ISAPNP_DEVICE_SINGLE_END` beendet wiederum die Tabelle. Mit Hilfe des Makros `MODULE_DEVICE_TABLE` wird die Tabelle wiederum exportiert, falls der Treiber als Modul kompiliert wurde.

ISA-PnP-Geräte konfigurieren

Ist nun auch die Funktion gefunden, die der Treiber ansprechen soll, kann die PnP-Karte konfiguriert werden. Dazu ist es notwendig, die `prepare`-Funktion der zurückgelieferten Struktur `pci_dev` aufzurufen. Diese Funktion initialisiert alle Ressourcen-Einträge in der `pci_dev`-Struktur. Dabei werden jedoch noch keine konkreten Werte zugewiesen, sondern lediglich die Typen der Ressourcen und ihre Eigenschaften (I/O-Port, Speicher, IRQ) zugewiesen. War die Funktion bereits konfiguriert (z. B. durch das BIOS oder andere Mechanismen), liefert sie `-EBUSY` zurück, ohne jedoch die Konfiguration zu ändern.

Alle Ressourcen, die das Attribut `IORESOURCE_AUTO` tragen, können nun noch explizit zugewiesen werden. Dazu dient die folgende Funktion:

```
void isapnp_resource_change(struct resource *resource,
                            unsigned long start,
                            unsigned long size);
```

Beachtenswert ist, dass diese Funktion nicht zwischen den Typen der Ressourcen unterscheidet; es ist somit möglich, sowohl I/O-Ports als auch Speicher zuzuweisen. Die Komponente `flags` in der Struktur `resource` gibt über die Art der Ressource Auskunft.

Die Funktion `activate()` der Struktur `pci_dev` übernimmt nun die automatische Zuweisung aller noch nicht zugewiesenen Ressourcen und aktiviert die Funktion. War sie bereits aktiviert, liefert `activate()` die aktuellen Einstellungen zurück.

Schauen wir uns die PnP-Erkennung am Beispiel des Soundblaster-Treibers einmal an:

```
static struct {
  char *name;
  unsigned short  card_vendor, card_device,
                  audio_vendor, audio_function,
                  mpu_vendor, mpu_function,
                  opl_vendor, opl_function;
  short           dma, dma2, mpu_io, mpu_irq;
} sb_isapnp_list[] = {
  {"Sound Blaster 16",
  ISAPNP_VENDOR('C','T','L'), ISAPNP_DEVICE(0x0024),
  ISAPNP_VENDOR('C','T','L'), ISAPNP_FUNCTION(0x0031),
  0,0,0,0,
  0,1,1,-1},

  …

  {0}
};
```

Das Feld `sb_isapnp_list[]` enthält die Hersteller- und Gerätecodes aller Soundblaster-Karten sowie deren Klone. Zusätzlich enthält es die Codes aller möglicher Funktionen auf den Karten, wie *Audio-DSP*, *MPU* und *OPL*. Die Erkennung beginnt mit der Funktion `sb_isapnp_probe`:

```
int sb_isapnp_probe(struct address_info *hw_config,
  struct address_info *mpu_config, int card)
{
  …
  while ((bus = isapnp_find_card(
                  sb_isapnp_list[i].card_vendor,
                  sb_isapnp_list[i].card_device,
                  bus))) {
    if(sb_isapnp_init(hw_config, mpu_config, bus, i, card)) {
      /* gefunden */
      return 0;
    }
  …
  }
  …
}
```

Diese Funktion ruft `isapnp_find_card()` für alle Karten in dem Feld `sb_isapnp_list[]`. Wird das Gerät gefunden, so wird die Funktion `sb_isapnp_init()` aufgerufen.

```
int sb_isapnp_init(struct address_info *hw_config,
  struct address_info *mpu_config, struct pci_bus *bus,
  int slot, int card)
{
  …
  if(sb_init(bus, hw_config, mpu_config, slot, card)) {
    /* gefunden */
    return 1;
  }
  …
}
```

Diese Funktion wiederum ruft zunächst `sb_init()` auf, um alle Funktionen auf der Soundkarte zu aktivieren:

```
struct pci_dev *sb_init(struct pci_bus *bus,
  struct address_info *hw_config,
  struct address_info *mpu_config,
  int slot, int card)
{
  /* Audio konfigurieren */
  if((sb_dev[card] = isapnp_find_dev(bus,
                        sb_isapnp_list[slot].audio_vendor,
                        sb_isapnp_list[slot].audio_function,
                        NULL))) {
    int ret;
    ret = sb_dev[card]->prepare(sb_dev[card]);
    /* falls die Audio Funktion bereits konfiguriert sein
     * sollte, Konfiguration abbrechen und weiter gehen */
     * and use anyway. Some other way to check this? */
    if(ret && ret != -EBUSY) {
```

```
      printk(KERN_ERR "sb: ISAPnP found device that could not"
                      " be autoconfigured.\n");
      return(NULL);
    }
    if(ret == -EBUSY)
      audio_activated[card] = 1;

    if((sb_dev[card] = activate_dev(
                         sb_isapnp_list[slot].name,
                         "sb", sb_dev[card]))) {
      hw_config->io_base = sb_dev[card]->resource[0].start;
      hw_config->irq     = sb_dev[card]->irq_resource[0].start;
      …
    } else
      return(NULL);
  } else
    return(NULL);

  …

  return(sb_dev[card]);
}
```

In dieser Funktion nun werden alle ISA-PnP Funktionen, die zur gefundenen Karte gehören, aktiviert. Danach wird die zurückgelieferte Struktur ausgelesen, um die zugewiesenen Werte wie I/O-Adresse, Interrupt-Kanal usw. zu initialisieren. Im Feld `audio_activated[]` wird noch vermerkt, ob die Funktion bereits vorher aktiviert war.

Ist ein Treiber als Modul realisiert und hat er die PnP-Hardware aktiviert, sollte er sie wieder deaktivieren, bevor er entladen wird. Dazu dient die Funktion `deactivate()` der Struktur `pci_dev`. In unserem Beispiel wurde mithilfe des Feldes `audio_activated[]` festgestellt, ob die Funktion deaktiviert werden muss.

7.3 Polling, Interrupts und Wait Queues

Im Vergleich zu der CPU ist die meiste Hardware sehr langsam. Im Multiprocessingbetrieb ist es daher nicht wünschenswert, die CPU solange warten zu lassen, bis eine Hardwareoperation beendet ist, d.h., man möchte die übrige Zeit gern für andere Tasks verwenden. Aus diesem Umstand ergibt sich, dass keine simplen Abfrageschleifen verwendet werden dürfen, um den Hardwarestatus abzufragen. Es sei denn, man gibt Linux zwischenzeitlich wieder die Kontrolle zurück. Für diese Aufgaben stehen mehrere Methoden zur Verfügung.

7.3.1 Polling

Eine dieser Methoden ist der Aufruf von `schedule()`, der den Linux-Scheduler dazu veranlasst, einem neuen Prozess die Kontrolle über die CPU zuzuordnen.

Der „Zufallszahlengenerator“ (`/dev/random`) arbeitet standardmäßig im Pollingbetrieb. So fragt er den „Entropie-Pool“ so lange ab, bis sich genügend Zufallswerte angesammelt haben. Dieses Vorgehen hat in den Quellen folgendes Aussehen:

```
static ssize_t
random_read(struct file * file, char * buf, size_t nbytes,
            loff_t *ppos)
{
  …
  ssize_t                       n, retval = 0, count = 0;

  if (nbytes == 0)
    return 0;

  …
  while (nbytes > 0) {
    set_current_state(TASK_INTERRUPTIBLE);

    n = nbytes;
    if (n > SEC_XFER_SIZE)
      n = SEC_XFER_SIZE;
    if (n > random_state->entropy_count / 8)
      n = random_state->entropy_count / 8;
    if (n == 0) {
      if (file->f_flags & O_NONBLOCK) {
        retval = -EAGAIN;
        break;
      }
      if (signal_pending(current)) {
        retval = -ERESTARTSYS;
        break;
      }
      schedule();
      continue;
    }
    n = extract_entropy(sec_random_state, buf, n,
        EXTRACT_ENTROPY_USER | EXTRACT_ENTROPY_SECONDARY);
    if (n < 0) {
      retval = n;
      break;
    }
  …
}
```

Ist die Entropiemenge leer, so wird zunächst getestet, ob das Gerät nichtblockierend geöffnet wurde. Falls ja, wird der Fehler `EAGAIN` zurückgegeben. Sonst muss überprüft werden, ob Signale für den Prozess vorliegen. Dies testet die Funktion `signal_pending()`.

7.3.2 Interruptbetrieb

Im *Interruptbetrieb* benachrichtigt das Gerät die CPU über einen Interruptkanal (IRQ), wenn es eine Operation beendet hat. Voraussetzung dafür ist, dass die Hardware die Auslösung von Interrupts unterstützt.

Diese unterbricht den laufenden Betrieb und führt eine Interruptserviceroutine (ISR) aus. Innerhalb der ISR erfolgt dann die weitere Kommunikation mit dem Gerät.

So wird ein Prozess, der auf die serielle Schnittstelle im Interruptbetrieb schreiben will, vom Gerätetreiber im Interruptbetrieb nach dem Schreiben eines Zeichens mit der Funktion

```
interruptible_sleep_on(&lp->lp_wait_q);
```

angehalten. Kann die serielle Schnittstelle weitere Zeichen entgegennehmen, löst sie einen IRQ aus. Die behandelnde ISR weckt den Prozess daraufhin wieder, und der Vorgang wiederholt sich.

Ein weiteres Beispiel ist die serielle Maus, die bei jeder Bewegung Daten an den seriellen Port, der einen IRQ auslöst, überträgt. Erst die behandelnde ISR liest die Daten aus dem seriellen Port aus und stellt sie dem Anwendungsprogramm zur Verfügung.

IRQs werden mit Hilfe der Funktion

```
int request_irq(unsigned int irq,
                void (*handler)(int, struct pt_regs *),
                unsigned long irqflags, const char * devname,
                void *dev_id)
```

installiert. Unter Linux gibt es zumindest zwei Möglichkeiten der IRQ-Bearbeitung. Das Argument `irqflags` gibt darüber Auskunft, welche Art von Interrupt verwendet werden soll. In älteren Linux-Versionen wurde zwischen langsamen Interrupts und und schnellen Interrupts unterschieden. Langsame Interrupts konnten durch andere Interrupts unterbrochen werden, schnelle nicht. Außerdem wurde nur am Ende von langsamen Interrupts der Bottom-Half-Händler gestartet. Linux 2.4 kennt diese Unterscheidung nicht mehr, lediglich die Wahl zwischen unterbrechbar und nichtunterbrechbar ist geblieben.

Die Installation von unterbrechbaren IRQs erfolgt ohne das Flag `SA_INTERRUPT` im Argument `irqflags`, die Installation nichtunterbrechbaren IRQs mit dem `SA_INTERRUPT` Flag. Das Argument `name` hat keine weitere Bedeutung für den Kern, es wird jedoch vom *Proc*-Dateisystem benutzt, um den Eigentümer eines IRQs anzuzeigen. Es sollte deshalb auf den Namen des Treibers zeigen, der den IRQ benutzt. Das Argument `dev_id` wird der Interruptroutine unverändert mitgegeben, kann also frei benutzt werden, um zusätzliche Daten zu übergeben. Falls der IRQ frei war und belegt werden konnte, liefert `request_irq()` 0 zurück.

Die Behandlungsroutine eines IRQs hat also folgendes Aussehen:

```
void do_irq(int irq,  void *dev_id, struct pt_regs * regs);
```

Jeder ISR wird als erstes Argument die Nummer des aufrufenden IRQs mitgegeben. Somit kann man also theoretisch eine ISR für mehrere IRQs benutzen. Das zweite Argument ist der bereits beschriebene Zeiger `dev_id`, das letzte Argument schließlich ist ein Zeiger auf die Struktur `pt_regs` und enthält alle Register des Prozesses, der durch den IRQ unterbrochen wurde. Auf diese Weise kann zum Beispiel der Timerinterrupt feststellen, ob ein Prozess im Kern- oder im Nutzermodus unterbrochen wurde und die jeweilige Zeit für die Abrechnung hochzählen.

Ein Beispiel soll die Installation eines nichtunterbrechbaren Interrupts zeigen:

```
if (request_irq(rtc_irq, rtc_interrupt, SA_INTERRUPT,
                "rtc", (void *)&rtc_port)) {
  printk(KERN_ERR "rtc: cannot register IRQ %d\n", rtc_irq);
  return -EIO;
}
```

Normalerweise wird man also für die Kommunikation mit der Hardware nichtunterbrechbare Interrupts verwenden.

7.3.3 Interrupt Sharing

Die Anzahl freier IRQs in einem PC ist begrenzt. Somit kann es sinnvoll sein, dass sich verschiedene Hardware Interrupts teilt. Bei PCI-Steckkarten ist dies sogar zwingend.

Die Voraussetzungen für ein solches *Interrupt Sharing* genanntes Vorgehen ist die Möglichkeit, die Hardware abzufragen, ob dieser Interrupt wirklich von ihr generiert wurde, und die Fähigkeit der ISR, einen nicht von ihrer Hardware ausgelösten Interrupt weiterzuleiten.

Die Linux-Version 2.4 unterstützt Interrupt Sharing, indem sie Ketten von Interruptbehandlungsroutinen aufbaut. Tritt ein Interrupt auf, wird jede ISR innerhalb der Kette von der Funktion `handle_IRQ_event()` aufgerufen.

```
int handle_IRQ_event(unsigned int irq,
  struct pt_regs * regs, struct irqaction * action)
{
  int status;

  …

  if (!(action->flags & SA_INTERRUPT))
    __sti();

  do {
    status |= action->flags;
    action->handler(irq, action->dev_id, regs);
```

```
    action = action->next;
  } while (action);
  if (status & SA_SAMPLE_RANDOM)
    add_interrupt_randomness(irq);
  __cli();


  …
}
```

Falls eine ISR installiert wird, die Interrupt Sharing beherrscht, muss dies der `request_irq`-Funktion durch Setzen des `SA_SHIRQ`-Flags mitgeteilt werden. War auf dieser IRQ-Nummer bereits eine andere ISR installiert, die ebenfalls Interrupt Sharing beherrscht, so wird eine Kette aufgebaut. Es ist jedoch nicht möglich, nichtunterbrechbare und unterbrechbare Interrupts zu mischen, d. h., alle Behandlungsroutinen eines IRQs müssen von derselben Art sein. Als Beispiel sei ein Fragment des DE4x5-Ethernet-Treibers gezeigt:

```
request_irq(dev->irq, (void *)de4x5_interrupt, SA_SHIRQ,
            lp->adapter_name, dev)
…

static void de4x5_interrupt(int irq, void *dev_id,
                struct pt_regs *regs)
{
   …
   sts = inl(DE4X5_STS);         /* IRQ-Status-Register lesen  */
   outl(sts, DE4X5_STS);         /* Reset der Board Interrupts */
   if (!(sts & lp->irq_mask)) break;/* Nicht vom Board, fertig */
   …
}
```

7.3.4 Softwareinterrupts

Es tritt jedoch häufig der Fall auf, dass nach Auftreten eines Interrupts nicht alle Funktionen sofort ausgeführt werden müssen: „Wichtige“ Aktionen müssen sofort erledigen werden, andere können auch später noch erledigt werden bzw. würden vergleichsweise lange dauern und man will den Interrupt nicht blockieren. Für diesen Fall wurden ursprünglich die *Bottom Halfs* (untere Hälften) geschaffen. In Linux 2.4 wurde dieser Mechanismus durch das Konzept von *Softwareinterrupts* ersetzt (siehe auch Abschnitt 3.2.3). Nach jedem Sprung durch `ret_from_syscall` und auch nach jedem Interrupt wird eine Menge von maximal 32 Softwareinterrupts aufgerufen, wenn gleichzeitig kein weiterer Hardwareinterrupt auf dem aktuellen Prozessor läuft[7].

7 Das kann durchaus geschehen, wenn z. B. ein unterbrechbarer Interrupt unterbrochen wird.

7.3.5 Bottom Halfs — die unteren Interrupthälften

Bottom Halfs sind also die Vorgänger von Softwareinterrupts. Sind sie als aktiv markiert, werden sie der Reihe nach einmal ausgeführt und dann automatisch wieder als inaktiv gekennzeichnet. Dabei sind alle Bottom-Halfs zueinander atomar, d. h. , solange ein Bottom-Half aktiv ist, kann kein anderer ausgeführt werden. Man muss sich also nicht vor Unterbrechungen schützen.

Für die Installation eines Bottom Halfs dient die Funktion `init_bh()`, die den Bottom Half in die Tabelle von Funktionspointern `bh_base` einträgt.

```
void init_bh(int nr, void (*routine)(void));

enum {
  TIMER_BH = 0,
  TQUEUE_BH,
  DIGI_BH,
  SERIAL_BH,
  RISCOM8_BH,
  SPECIALIX_BH,
  AURORA_BH,
  ESP_BH,
  SCSI_BH,
  IMMEDIATE_BH,
  CYCLADES_BH,
  CM206_BH,
  JS_BH,
  MACSERIAL_BH,
  ISICOM_BH

};
```

Standardmäßig sind alle Bottom Halfs zugelassen, sie können aber auch mit den Funktionen

```
void disable_bh(int nr);
void enable_bh(int nr);
```

ab- und wieder zugeschaltet werden. Die Funktion

```
void mark_bh(int nr);
```

dient zum Markieren eines Bottom Halfs, d. h., dieser Bottom Half wird zum nächstmöglichen Zeitpunkt abgearbeitet.

Betrachten wir nun die Verwendung eines Bottom Halfs. Als Beispiel sei hier der Timer-Interrupt gezeigt:

```
void do_timer(struct pt_regs *regs)
{
  (*(unsigned long *)&jiffies)++;
```

```
  …
  mark_bh(TIMER_BH);
  …
}

void timer_bh(void)
{
  update_times();
  run_timer_list();
}

void __init sched_init(void)
{
   …
  init_bh(TIMER_BH, timer_bh);
  …
}
```

Die Init-Funktion des Schedulers installiert `timer_bh()` als Bottom Half. Bei jedem Aufruf des Timer-Interrupts wird `mark_bh(TIMER_BH)` aufgerufen, d. h., der Bottom Half läuft zum ersten Zeitpunkt nach Beendigung des Timer-Interrupts – im Idealfall gleich danach.

Bottom-Halfs sind in der Version 2.4 mit Hilfe von Softwareinterrupts implementiert. Der höchstpriorisierte Softwareinterrupt `HI_SOFTIRQ` wird benutzt, um die Bottom-Half Händler auszuführen.

7.3.6 Task Queues

Wie der vorherige Abschnitt zeigt, ist jedoch die direkte Benutzung der Bottom Halfs ein wenig schwierig, da es nur 32 gibt und einige Aufgaben bereits an feste Nummern gebunden sind. Seit der Version 2.0 bietet Linux deshalb mit den *Task Queues* eine Erweiterung der Bottom Halfs, die es erlaubt, das Konzept der Bottom Halfs dynamisch zu erweitern.

Die Task Queues erlauben es, beliebig viele Funktionen in eine Warteschlage einzureihen und sie dann später zu einer geeigneten Zeit nacheinander abzuarbeiten. Die Verkettung der auszuführenden Funktionen geschieht mit Hilfe der Struktur `tq_struct`.

```
struct tq_struct {
  struct list_head list ;  /* Verkettung mit dem nächsten      */
                           /* Eintrag                          */
  unsigned long sync;      /* Synchronisationsflag             */
  void (*routine)(void *); /* aufzurufende Funktion            */
  void *data;              /* beliebiges Argument der Funktion */
};

typedef struct list_head task_queue;
```

Bevor nun eine Funktion in eine Task Queue aufgenommen werden kann, muss eine `tq_struct`-Struktur angelegt und initialisiert werden. Die Komponente `routine` erhält die Adresse der aufzurufenden Funktion, `data` ein beliebiges Argument, das der Funktion beim Aufruf übermittelt werden soll. Die Komponente `sync` muss mit 0 initialisiert werden. Die Aufnahme in eine Task Queue geschieht mit Hilfe der folgenden Funktion:

```
int queue_task(struct tq_struct *bh_pointer, task_queue *bh_list)
{
  int ret = 0;
  if (!test_and_set_bit(0,&bh_pointer->sync)) {
    unsigned long flags;
    spin_lock_irqsave(&tqueue_lock, flags);
    list_add_tail(&bh_pointer->list, bh_list);
    spin_unlock_irqrestore(&tqueue_lock, flags);
    ret = 1;
  }
  return ret;
}
```

Die speziellen Versionen `queue_task_irq()` und `queue_task_irq_off()` werden in der Version 2.4 nicht mehr unterstützt.

Von der Funktion `run_task_queue()` wird die Abarbeitung einer Task Queue übernommen.

```
void run_task_queue(task_queue *list)
```

Sie übernimmt eine Task Queue als Argument und arbeitet alle in der Queue eingehängten `tq_struct`-Strukturen durch Aufruf ihrer Funktionen ab. Das `sync`-Flag wird dabei gelöscht, bevor die Funktion aufgerufen wird, so dass es innerhalb dieser Funktion bereits wieder möglich wäre, die `tq_struct`-Struktur in eine beliebige Task Queue einzuhängen.

In der Linux-Version 2.4 sind unter anderem die folgenden Task Queues definiert:

`tq_timer` wird nach jedem Timerinterrupt bzw. zum nächstmöglichen Zeitpunkt nach einem Timerinterrupt abgearbeitet.

`tq_immediate` wird zum nächstmöglichen Zeitpunkt nach Aufruf von der Funktion `mark_bh(IMMEDIATE_BH)` aufgerufen und entspricht somit den Bottom Halfs der Version 1.x.

`tq_disk` wird von Blockgeräten benutzt und an verschiedenen Stellen, an denen das VFS auf eintreffende Puffer oder ähnliches warten muss, aufgerufen.

`tq_disk` zeigt, dass Task Queues nicht nur an Bottom Halfs gebunden sein müssen. Task Queues sind nur als ein Zeiger auf eine `tq_struct`-Struktur implementiert und mittels des Makros `DECLARE_TASK_QUEUE()` zu deklarieren. Durch einen Aufruf der Funktion `run_task_queue()` können sie an beliebigen Stellen abgearbeitet werden. Das Abarbeiten von Task Queues innerhalb von Interruptserviceroutinen sollte jedoch vermieden werden, um Interrupts nicht unnötig lange zu blockieren.

7.3.7 Timer

In manchen Fällen muss man gezielt den wartenden Prozess nach einer gewissen Laufzeit aufwecken. Zum Beispiel, wenn der Prozess auf einen Interrupt wartet, dieser aber wegen eines Hardware-Fehlers oder eines anderen Problems niemals kommt.

Für diese Fälle bietet Linux die Möglichkeit, Timer zu programmieren, die den Prozess nach Beendigung einer einstellbaren Zeit wieder zum Leben erwecken. Im folgenden Beispiel ruft der Timer nach einer Zeit eine Timer-Interrupt-Routine auf, die die verlorenen Interrupts registriert.

In der Init-Routine des Treibers wird zunächst der Timer initialisiert und auf die Timer-Interrupt Routine gesetzt.

```
…
static struct timer_list rtc_irq_timer;
…
static int __init rtc_init(void)
{
  …
  init_timer(&rtc_irq_timer);
  rtc_irq_timer.function = rtc_dropped_irq;
  …
}
```

Nachdem die Hardware für die Auslösung des Interrupts programmiert wurde, wird ein Timer auf eine bestimmte Zeit (Anzahl der Timer-Ticks in jiffies) programmiert und mit `add_timer()` gestartet:

```
  if (!(rtc_status & RTC_TIMER_ON)) {
    spin_lock_irq (&rtc_lock);
    rtc_irq_timer.expires = jiffies + HZ/rtc_freq + 2*HZ/100;
    add_timer(&rtc_irq_timer);
    rtc_status |= RTC_TIMER_ON;
    spin_unlock_irq (&rtc_lock);
  }
```

Kommt der gewünschte Interrupt nicht in der programmierten Zeit, verfällt der Timer, und die Routine `rtc_dropped_irq()` wird aufgerufen. Diese reprogrammiert den entsprechenden Baustein und startet den Timer erneut. Dazu wird die Funktion `mod_timer()` verwendet.

```
void rtc_dropped_irq(unsigned long data)
{
  …
  if (rtc_status & RTC_TIMER_ON)
    mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);
  …
}
```

```
static void rtc_interrupt(int irq, void *dev_id,
                          struct pt_regs *regs)
{
  /*
   * Can be an alarm interrupt, update complete interrupt,
   * or a periodic interrupt. We store the status in the
   * low byte and the number of interrupts received since
   * the last read in the remainder of rtc_irq_data.
   */
  …
  if (rtc_status & RTC_TIMER_ON)
    mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);
  …
}
```

Die „echte“ Interrupt-Routine muss den Timer ebenfalls neu starten, da sonst ein Timer-Interrupt nachfolgen würde.

Schließlich existiert noch die Funktion `del_timer()`, mit deren Hilfe ein Timer gelöscht werden kann.

7.4 Die Implementierung eines Treibers

7.4.1 Beispiel — PC Lautsprechertreiber

Wollen wir nun einen Gerätetreiber für den internen Lautsprecher schreiben, kommen wir nicht umhin, uns genauer mit dieser Hardware und ihrer Steuerung zu beschäftigen.

Seit den Urzeiten des PCs schon vorhanden, ist der PC-Lautsprecher aufgrund seines Designs nicht gerade gut zur Ausgabe von Samples geeignet. Wie Abbildung 7.3 zeigt, ist sowohl der Aufbau als auch die Programmierung des Lautsprechers sehr einfach.

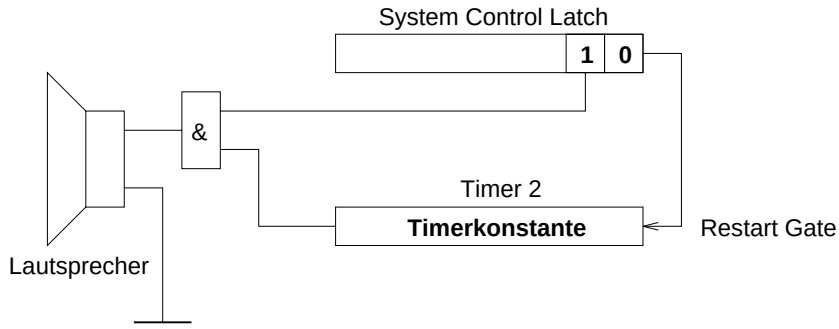


Abbildung 7.3: Schematischer Anschluss des PC-Lautsprechers

Der Timerbaustein 8253 besitzt drei interne Timer. Timer 2 ist zur Verwendung mit dem PC-Lautsprecher bestimmt. Dazu ist der Ausgang des Timers 2 über ein AND-Gatter mit dem Bit 1 des System Control Latches auf I/O-Adresse 0x61 verbunden. Bit 0 dient zum

Start bzw. Neustart des Timers 2. Der Lautsprecher kann also nur entweder voll ein- oder ausgeschaltet sein. Normalerweise wird der Timer 2 als Frequenzteiler programmiert (d.h. beide Bits sind gesetzt). Dadurch werden Rechteckwellen erzeugt, die den „typischen" Klang des internen Lautsprechers ausmachen. Die Frequenz entsteht durch Teilung der Timergrundfrequenz von 1,193 MHz (= 4,77 MHz / 4) durch die eingestellte Timerkonstante.

Um mit dem Lautsprecher ein analoges Signal zu übertragen, wird die Pulslängenmodulation benutzt. Durch sehr schnelles Umschalten zwischen verschieden langen Ein- und Aus-Phasen, deren Verhältnis gerade dem auszugebenden Analogwert entspricht, wird durch die mechanische Trägheit des Lautsprechers eine analoge Ausgabe erzeugt. Leider ist die Pulslängenmodulation auch sehr empfindlich. Bereits das Fehlen eines Samples äußert sich in einem störenden Knacken im Lautsprecher[8].

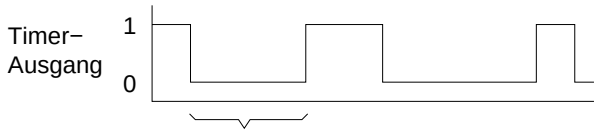
Als das eigentliche Problem bei der Verwendung der Pulslängenmodulation stellt sich das Bestimmen der benötigten Zeitintervalle und ihre Erzeugung heraus. Die erste Möglichkeit besteht darin, Timer 2 nicht zu benutzen und die Ausgabe völlig mit Hilfe des Bit 1 des System Control Latches zu steuern. Die Zeitintervalle können durch Warteschleifen erzeugt werden. Dieses Vorgehen ist am einfachsten zu realisieren, bietet aber zwei entscheidende Nachteile:

- Die Zeitschleifen sind abhängig vom Prozessortakt.
- Die meiste Zeit während der Ausgabe wird mit *Busy Waiting* verbraucht; dies ist in einem Multitasking-Betriebssystem nicht akzeptabel.

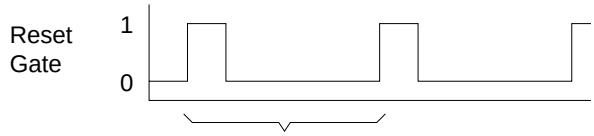
Die zweite Möglichkeit besteht darin, den Timer 2 als *Retriggerable Oneshot* zu programmieren. Durch Anlegen einer 1 am Restart Gate wird der Timer gestartet und gibt eine 0 aus. Nach dem Herunterzählen der Timerkonstante wird wieder eine 1 ausgegeben. Nach einer gewissen Zeit, die dem maximalen Samplewert entspricht, wird eine neue Konstante in den Timer 2 übertragen und dieser wieder gestartet. Diese konstante Zeit kann wiederum mit Hilfe einer Zeitschleife oder mit dem Timer 0 erzeugt werden. Timer 0 läuft normalerweise im Teilermodus und generiert nach jedem Herunterzählen der Timerkonstante den IRQ 0. Diese vom Timer 0 erzeugte Frequenz ist gleichzeitig die Samplerate, mit der die Samples abgespielt werden können. Sie wird im Folgenden *Reale Samplerate* genannt. In der Interruptbehandlungsroutine muss Timer 2 dann neu initialisiert werden. Dieses Vorgehen zeigt Abbildung 7.4.

Der Timerbaustein besitzt 4 I/O-Ports. Port 0x43 ist das Mode Control Register. Die Datenports 0x40 bis 0x42 sind den Timern 0 bis 2 zugeordnet. Um einen Timer zu programmieren, muss also ein Kommando nach 0x43 und die Timerkonstante in den entsprechenden Datenport geschrieben werden. Ein Kommando ist sehr einfach aufgebaut. Die Bits 7 und 6 enthalten die Nummer des zu programmierenden Timers, 5 und 4 eine der in Tabelle 7.2 aufgezeigten Zugriffsarten und die Bits 3 bis 1 den Timermodus.

8 Dies ist der Grund für die Nebengeräusche bei Diskettenzugriffen oder sogar bei Mausbewegungen. Das Nichtbehandeln eines einzigen Interrupts lässt die Dynamik des Lautsprechers zusammenbrechen.



Länge des Intervalls wird durch Timerkonstante 2 festgelegt.



Konstantes Intervall wird durch Timer 0 erzeugt.

$$\text{Intervallänge} = \frac{\text{Timerkonstante}}{1193180} \text{ sec}$$

Abbildung 7.4: Pulslängenmodulation mit Hilfe der Timer 0 und 2

Bits 54	Mode	Erklärung
00	Latch	Der Zähler wird in ein internes Register übertragen und kann danach ausgelesen werden.
01	LSB only	Nur die unteren 8 Bit des Zählers werden übertragen.
10	MSB only	Nur die oberen 8 Bit des Zählers werden übertragen.
11	LSB/MSB	Zunächst werden die unteren, danach die oberen 8 Bit übertragen.

Tabelle 7.2: Bits 4 und 5 des Timer-Kommandos

Um z. B. einen Ton mit 10.000 Hz zu erzeugen, sind folgende Schritte notwendig.

```
outb_p (inb_p (0x61) | 3, 0x61);
   /* öffnet das AND-Gate und                                  */
   /* setzt das Restart-Gate auf aktiv                         */
tc = 1193180 / 10000;
   /* berechnet die nötige Timerkonstante                      */
outb_p (0xb6, 0x43);
   /* entspricht dem Befehl:
   /*    Timer 2, Read/Write LSB dann MSB, Timermode 3         */
outb_p (tc & 0xff, 0x42); outb ((tc >> 8) & 0xff, 0x42);
   /* schreibt die Zeitkonstante in den Timer 2;               */
   /* von jetzt an "tönt" es aus dem internen Lautsprecher     */
```

Das „Stilllegen“ erfolgt einfach durch

```
outb(inb_p(0x61) & 0xfc, 0x61);
```

Dadurch wird sowohl der Lautsprecher abgeschaltet als auch der Timer gestoppt. Leider ist im Standard-PC nur der Timer 0 interruptfähig, so dass die beschriebene zweite Möglichkeit nicht ganz ungefährlich ist, wird doch der für Linux so wichtige Timerinterrupt IRQ 0 verändert. Die neue Interruptroutine muss dafür sorgen, dass die ursprüngliche Prozedur in genau denselben Zeitintervallen wieder aufgerufen wird. Außerdem benötigt die Interruptbehandlung im Protected Mode weitaus mehr Zeit als im Real Mode, so dass durch die größere Anzahl ausgelöster Interrupts die Rechenzeit merklich verbraucht wird.

Kommen wir zurück zur Pulslängenmodulation. Wie bereits erwähnt, ist die Wahl der Zeitintervalle sehr wichtig. Versuche haben gezeigt, dass für eine reale Samplerate zwischen 16.000 Hz und 18.000 Hz die besten Resultate erzielt werden. Je höher die reale Samplerate, desto besser, da diese als Eigenfrequenz (Pfeifen) zu hören ist[9]. Diese Frequenzen ergeben bei Benutzung des Timers zwei mögliche Timerkonstanten zwischen 1 und 74 (eine 0 würde 65.536 bedeuten und ist deshalb unzulässig). Da die Konstanten direkt mit den Samples zusammenhängen, kann man also nur 6 Bit (1 bis 65) ausgeben.

Als Maximalwert für die reale Samplerate sind also 18.357 Hz möglich (dies entspricht 1,193 MHz / 65). Dieser Wert ist allerdings nicht sehr gebräuchlich, deshalb werden mit Hilfe zusätzlich generierter Samples (*Oversampling*) auch andere Sampleraten unterstützt. Aus Zeitgründen sorgt ein einfacher Algorithmus dafür, dass durch die Wiederholung[10] einzelner Samples die Daten „auseinandergezogen“ werden. Soll die Ausgabe z. B. mit 10.000 Hz erfolgen, muss jedes Sample im Durchschnitt 1,8-mal abgespielt werden.

Die Ausgabe über Digital-Analog-Wandler (DAC) hingegen ist sehr einfach. Diese werden einfach an einen Parallelport angeflanscht und wandeln die eingehenden 8 Bit in ein Analogsignal um. Da der Parallelport die eingehenden Werte zwischenpuffert, kann der Aufbau eines solchen DACs sehr einfach sein; im genügsamsten Falle handelt es sich einfach um ein Widerstandsnetz. Außerdem kann der Parallelport die Daten in fast beliebiger Geschwindigkeit ausgeben, Timer 0 kann also mit der wahren Samplerate programmiert werden.

Ebenso entfällt die Transformation der Samples in eine 6-Bit-Darstellung; die Ausgabe über DACs benötigt deshalb weniger Prozessorzeit als die über den internen Lautsprecher. Letzter Pluspunkt für diese Lösung: Fehlende Interrupts äußern sich nur durch eine Verlangsamung des abgespielten Sounds und sind innerhalb gewisser Grenzen fast unhörbar.

9 Ab welcher Eigenfrequenz dieses Pfeifen hörbar ist, ist subjektiv unterschiedlich. Ich höre erst ab 14.500 Hz etwas, andere hören auch 17.000 Hz noch.

10 Normalerweise würden die neuen Samples durch Interpolation berechnet. Bei der Ausgabe durch den internen Lautsprecher ist damit jedoch keine Qualitätsverbesserung zu erzielen.

7.4.2 Ein einfacher Treiber

Nach detaillierter Erklärung der Hardware des internen Lautsprechers drängt sich nun die Frage auf, warum extra ein Gerätetreiber gebraucht wird, um das Schreiben und Lesen einiger I/O-Ports zu erledigen.

Zur Erzeugung von „Geräuschen" könnte auch ein Programm `auplay`[11] geschrieben werden, welches mit Hilfe des Systemrufes *ioperm* die entsprechenden Ports freigibt

```
if (ioperm(0x61,1,1) || ioperm(0x42,1,1) || ioperm(0x43,1,1)) {
        printf("can't get I/O permissions for internal speaker\n");
        exit(-1);
}
```

und die Ausgabe der Samples danach selbst übernimmt. Dies würde aber zu mehreren Nachteilen führen:

- Der Systemruf *ioperm* führt nur mit privilegierten Rechten zum Erfolg. Somit benötigt das Programm Set-UID-Rechte von `root`. Im Allgemeinen sollten in UNIX-Systemen keine Programme mit Set-UID-Rechten von `root` existieren, da sie ein großes Sicherheitsproblem darstellen. Dies kann im Normalfall durch das Einrichten spezieller Nutzer und Gruppen (z. B. die Gruppe `kmem` zur Nutzung des Geräts `/dev/kmem`) gewährleistet werden, lässt sich aber in unserem Fall schlecht umgehen.

  Ein Gerätetreiber dagegen arbeitet mit Kern-Berechtigung und kann somit über alle Ressourcen frei verfügen. Dies sollte aber bei der Implementierung eines Treibers nicht vergessen werden, da Fehler im Treiber sich möglicherweise schlimmer auswirken können als Fehler in einem Programm.[12]

- Das Hauptproblem ist wohl die genaue Zeitabstimmung eines Programms in einem Multitaskingsystem. Die einzige Möglichkeit besteht in Warteschleifen der Art:

  ```
  for ( j = 1; j < DELAY; j++);
  ```

  Dieses *Busy Waiting* ist nicht akzeptabel, da eine genaue Abstimmung der Samplerate nicht möglich ist. Die Nutzung des Timerinterrupts ist dafür eine wesentlich elegantere Variante, kann aber nur im Kern geschehen.

- Ein weiteres Problem ist die Kontrolle über den PC-Lautsprecher.

  Wer garantiert, dass nicht ein anderer Prozess zur gleichen Zeit auf die I/O-Ports zugreift und es so zu Sample-Schrott kommt? Die Nutzung von SYSTEM V-IPC (in diesem Fall Semaphoren) ist, als würde man mit Kanonen auf Spatzen schießen, zumal nicht geklärt ist, ob andere Programme nicht auch auf dieselben Ports zugreifen.

  Die Zugangsbeschränkung für Geräte ist demgegenüber relativ einfach und wird im Folgenden noch erklärt.

11 Das Programm `auplay` von RICK MILLER war der Anstoß für den PC-Speaker-Treiber bzw. der Implementierung.

12 Dies stimmt nur bedingt, da man beim Programm `auplay` über die falsche Benutzung des Mode Control Registers auf I/O-Adresse 0x43 den Timerinterrupt durcheinander und den Rechner zum Absturz bringen kann.

Das Schreiben eines „Audio-Dämons“, der aus einer *named pipe* die Sampledaten liest und beim Hochfahren des Systems über die Datei `rc.local` gestartet wird, hilft dabei nur bedingt. Das Problem der Zeitabstimmung bleibt auf jeden Fall erhalten.

Ein Gerätetreiber wäre also doch ratsam. Die eigentliche Implementierung des PC-Speaker-Treibers läuft dabei auf das Ausfüllen der im vorigen Kapitel beschriebenen Struktur `file_operations` hinaus. Der Programmierer muss dabei je nach Art des Geräts nicht alle Funktionen belegen. Zusätzlich muss er eine weitere Prozedur zur Initialisierung des Treibers bereitstellen.

Die Namen dieser C-Funktionen sollten alle nach demselben Schema gebildet werden, um Konflikte mit existierenden Funktionen zu vermeiden. Die sicherste Variante ist, einen kurzen Namen des Treibers den eigentlichen Funktionsnamen voranzustellen. So werden für den PC-Speaker-Treiber, oder kurz „`pcsp`“, die Funktionen `pcsp_init()`, `pcsp_read()` usw. im Folgenden genauer erklärt. Das gleiche Vorgehen sollte auch für externe und statische C-Variablen Anwendung finden.

7.4.3 Die Setup-Funktion

Manchmal möchte man einem statisch gelinktem Gerätetreiber oder allgemein dem Linux-Kern Parameter übergeben. Dies kann notwendig sein, wenn eine automatische Erkennung von Hardware nicht möglich ist oder zum Konflikt mit anderer Hardware führt. Dazu bietet sich die Nutzung der Linux-Bootparameter an, die dem Kern während des Bootvorgangs übergeben werden können. Im Allgemeinen werden diese Parameter in Form einer Kommandozeile zum Beispiel vom Linux-Lader LILO kommen (siehe Abschnitt D.2.5).

Diese Kommandozeile wird von der Funktion `parse_options()`, die sich in `init/main.c` befindet, in ihre einzelnen Bestandteile zerlegt. Für jeden dieser Parameter wird die Funktion `checksetup()` aufgerufen. Diese Funktion vergleicht den Anfang des Parameters mit den im Feld `__setup_start` gespeicherten Strings und ruft bei Übereinstimmung die zugehörige `setup`-Funktion auf. Gibt diese einen Wert ungleich Null zurück, wird die Bearbeitung dieses Parameters beendet, ansonsten nach weiteren Matches gesucht. Ein Parameter sollte dabei in der Version 2.4 den folgenden Aufbau haben:

name=Parameterstring

Falls im Parameterstring Leerzeichen vorkommen, muss der gesamte String mit Hilfe von Gänsefüßchen gequotet werden.

Der Parameterstring wird einfach an die `setup`-Funktion weitergereicht, die den folgenden Aufbau haben muss:

```
int setup_func(char *);
```

Als Beispiel soll hier die `setup`-Funktion des PC-Speaker-Treibers dienen.

```
static int __init pcsp_setup(char *str)
{
  if (!strcmp(str, "off")) {
    pcsp_enabled = 0;
    return 1;
  }
  pcsp.maxrate = simple_strtol(str, NULL, 0);
  pcsp_enabled = 1;
  return 1;
}

__setup("pcsp=", pcsp_setup);
```

Wie zu sehen ist, testet diese Funktion zunächst das Vorhandensein des Wortes „off". Der Boot-Parameter „pcsp=off" schaltet den PC-Speaker-Treiber also ab. Ansonsten wird angenommen, dass es sich bei dem übergebenen String um einen numerischen Parameter handelt, der zur Initialisierung einer globalen Variablen des PC-Speaker-Treibers benutzt wird.

Diese Funktion muss jedoch registriert werden. In Linux-Versionen vor 2.4 musste die Funktion dazu in das globale Feld `bootsetups[]` eingetragen werden. Die aktuelle Version implementiert einen eleganteren Mechanismus, der jedoch etwas „Linker-Magie" benutzt. Dazu wird das folgende Makro benutzt:

```
struct kernel_param {
  const char *str;
  int (*setup_func)(char *);
};

#define __setup(str, fn)                                \
  static char __setup_str_##fn[] __initdata = str;  \
  static struct kernel_param __setup_##fn           \
   __attribute__((unused)) __initsetup =            \
    { __setup_str_##fn, fn }
```

Dieses Makro erzeugt also einen statischen Eintrag vom Typ `kernel_param`. Das spezielle Attribut `unused` sorgt dafür, dass der Compiler keine Warnungen wegen unbenutzter Variablen erzeugt. Das Makro `__initsetup` schließlich sorgt dafür, dass diese erzeugte Variable in einer speziellen Sektion des Linux-Kerns angelegt wird. Ein Linkler-Script sorgt dafür, dass die Variablen `__setup_start` und `__setup_end` auf den Anfang und das Ende des auf diese Weise erzeugten Feldes von Variablen zeigen.

Bei der Verwendung einer `setup`-Funktion sollte man beachten, dass sie vor der Initialisierung der Gerätetreiber durch ihre `init`-Funktion aufgerufen wird. Man sollte in der `setup`-Funktion also nur globale Variablen setzen, die dann von der `init`-Funktion ausgewertet werden können.

7.4.4 Init

`init()` wird bei statisch gelinkten Gerätetriebern während der Kerninitialisierung aufgerufen, erfüllt jedoch wichtige Aufgaben. Diese Funktion dient der Überprüfung des Vorhandenseins eines Geräts, dem Aufbau interner Strukturen des Gerätetreibers sowie der Anmeldung des Geräts.

Während in älteren Versionen der Aufruf der `init`-Funktion noch in andere Funktionen[13] eingetragen werden musste, wird in der Version 2.4 wiederum der Linker benutzt, um die Liste aller aufzurufenden `init`-Funktionen dynamisch zu erstellen. Dazu dient das Makro `__initcall()`:

```
typedef int (*initcall_t)(void);
typedef void (*exitcall_t)(void);

#define __initcall(fn) \
  static initcall_t __initcall_##fn __init_call = fn

#define __exitcall(fn) \
        static exitcall_t __exitcall_##fn __exit_call = fn
```

Wie man sieht, ist auch ein Makro `__exitcall()` definiert, für statische Treiber wird es aber nicht benutzt. Das Makro `__initcall()` wird jedoch üblicherweise nicht direkt benutzt, stattdessen gilt für statisch gelinkte Treiber:

```
#define module_init(x)  __initcall(x);
#define module_exit(x)  __exitcall(x);
```

Somit kann der gleiche Code für die Initialisierung von statischen und dynamischen Treibern benutzt werden (siehe 9.4).

Der folgende Code sorgt also dafür, dass die `init`-Funktion des PC-Speaker-Treibers automatisch von der Funktion `do_initcalls()` aufgerufen wird:

```
int __init pcsp_init(void)
{
  …
}

module_init(pcsp_init);
```

Damit Linux mit dem Treiber überhaupt etwas anfangen kann, muss dieser registriert werden. Dazu dient die Funktion `register_chrdrv()`, die

- die Major-Nummer des Gerätetreibers,
- den symbolischen Namen des Gerätetreibers und
- die Adresse der `file_operations`-Struktur (hier `pcsp_fops`) erhält.

13 Dazu zählten insbesondere `chr_dev_init()` und `blk_dev_init()`.

Eine zurückgelieferte 0 signalisiert, dass der neue Treiber registriert ist. Ist die Major-Nummer schon durch einen anderen Gerätetreiber belegt worden, so liefert `register_chrdrv()` den Fehler `EBUSY` zurück.

```
if (register_chrdev(PCSP_MAJOR, "pcsp", &pcsp_fops))
   printk("unable to get major %d for pcsp devices\n",PCSP_MAJOR);
else {
   printk("PCSP-device 1.0 init:\n");
   …
}
```

In diesem Fall kann man versuchen, eine freie Major-Nummer zu allozieren. Dazu muss der Funktion `register_chrdrv()` eine 0 als Major-Nummer übergeben werden. `register_chrdrv()` sucht dann die Liste aller Major-Nummern — angefangen bei `MAX_CHRDEV-1` — durch und registriert den Treiber unter der ersten freien Nummer. Diese Nummer wird zurückgeliefert. Konnte keine freie Nummer gefunden werden, liefert `register_chrdrv()` den Fehler `EBUSY` zurück.

```
if (!register_chrdev(DEFAULT_MAJOR, "device", &device_ops))
   printk("Device registered.\n");
else {
   major = register_chrdev(0, "device", &device_ops));
   if (major > 0)
      printk("Device registered using major %d.\n", major);
   else {
      printk("Cannot register device!\n");
      …
   }
}
```

`init()` ist auch der richtige Platz, um zu testen, ob überhaupt ein Gerät, welches vom Treiber unterstützt wird, vorhanden ist. Dies gilt besonders für Geräte, die nicht während des Betriebs gewechselt bzw. angeschlossen werden können, wie etwa Festplatten. Kann kein Gerät gefunden werden, sollte *jetzt* der Treiber eine Meldung ausgeben (das Nichterkennen des Geräts könnte ja auch ein Hardwarefehler sein) und sicherstellen, dass das Gerät später auch nicht angesprochen wird.

Findet z. B. ein CD-ROM-Treiber kein CD-Laufwerk, hat es keinen Sinn, dass der Treiber Speicher für Puffer belegt — das Laufwerk kann nicht während des Betriebes hinzukommen. Anders ist dies bei Geräten, die später zugeschaltet werden können. Wenn der PC-Speaker-Treiber keinen Stereo-on-One[14] erkennt, lässt der Treiber diesen auch später noch zu.

Wurden ein oder mehrere Geräte erkannt, sollten diese innerhalb der `init`-Funktion initialisiert werden, wenn dies notwendig ist.

14 Ein Stereo-on-One ist ein von Mark J. Cox entworfener einfacher Stereo-Digital-Analog-Wandler, der aber nur einen Parallelport belegt und softwaremäßig erkannt werden kann.

7.4.5 Open und Release

Die `open`-Funktion ist verantwortlich für die Verwaltung aller Geräte. `open()` wird aufgerufen, sobald ein Prozess eine Gerätedatei öffnet. Falls, wie in unserem Beispiel, nur ein Prozess mit einem Gerät arbeiten kann, muss `open()` als Rückgabewert `-EBUSY` zurückgeben. Kann ein Gerät von mehreren Prozessen gleichzeitig benutzt werden, sollte `open()` die dafür notwendigen Warteschlangen einrichten, falls diese nicht in `read()` oder `write()` eingerichtet werden können. Falls kein Gerät existiert (z. B. wenn ein Treiber mehrere Geräte unterstützt, aber nur eines vorhanden ist), sollte er `-ENODEV` zurückgeben. Ansonsten ist `open()` der richtige Platz, um für den Treiber notwendige Standardeinstellungen zu initialisieren. Ein gelungenes Öffnen ist durch eine 0 als Rückgabewert anzuzeigen.

```
static int pcsp_open(struct inode *inode, struct file *file)
{
  if (pcsp_active)
    return -EBUSY;

  switch (minor & 0x0f) {
    case 3:         /* DSP device /dev/dsp* */
      if (pcsp_set_format(AFMT_S16_LE) != AFMT_S16_LE)
        pcsp_set_format(AFMT_U8);
      break;
    case 4:         /* Sun Audio device /dev/audio* */
      pcsp_set_format(AFMT_MU_LAW);   /* ULAW-Format */
      break;
    …
  }

  if (! (pcsp.buf[0] = vmalloc(pcsp.ablk_size)))
    return -ENOMEM;
  if (! (pcsp.buf[1] = vmalloc(pcsp.ablk_size))) {
    vfree(pcsp.buf[0]);
    return -ENOMEM;
  }

  pcsp.buffer   = pcsp.end   = pcsp.buf[0];
  pcsp.in[0]    = pcsp.in[1] = 0;
  pcsp.timer_on  = pcsp.frag_size = pcsp.frag_cnt  = 0;
  …
  pcsp_active   = 1;
  return 0;
}
```

Die `release`-Funktion wird aufgerufen, wenn der Dateideskriptor auf das Gerät freigegeben wird (siehe Abschnitt 6.2.9). Ihre Aufgabe sind Aufräumaktionen globaler Natur, u.a. das Leeren von Warteschlangen. Bei bestimmten Geräten kann es auch sinnvoll sein, zunächst alle Daten, die sich noch in Puffern befinden, an das Gerät weiterzuleiten. Im Falle des PC-Speaker-Treibers bedeutet das, dass die Gerätedatei schon geschlossen werden kann, bevor alle Daten in den Ausgabepuffern abgespielt sind. Die Funktion

`pcsp_sync()` wartet deshalb darauf, dass beide Puffer geleert wurden und gibt sie danach wieder frei.

```
static int pcsp_release(struct inode *inode,
                        struct file *file)
{
  pcsp_sync();
  pcsp_stop_timer();
  outb_p(0xb6,0x43);    /* binary, mode 2, LSB/MSB, ch 2 */

  vfree(pcsp.buf[0]);
  vfree(pcsp.buf[1]);

  pcsp_active = 0;
  return 0;
}
```

Die `release`-Funktion ist optional; allerdings ist so eine Konstellation schwerlich vorstellbar.

7.4.6 Read und Write

`read()` und `write()` sind vom Prinzip her symmetrische Funktionen. Da man vom internen Lautsprecher keine Daten lesen kann, ist im PC-Speaker-Treiber nur `write()` implementiert. Nachdem bereits in Abschnitt 7.3 der Aufbau einer `read`-Funktion für Treiber im Pollingbetrieb betrachtet wurde, betrachten wir nun die vereinfachte `write`-Funktion des PC-Speaker-Treibers als Beispiel für den Interruptbetrieb.

```
static int pcsp_write(struct inode *inode, struct file *file,
                      char *buffer, int count)
{
  unsigned long copy_size;
  unsigned long max_copy_size;
  unsigned long total_bytes_written = 0;
  unsigned bytes_written;
  int i;

  …

  max_copy_size = pcsp.frag_size \
                ? pcsp.frag_size : pcsp.ablk_size;
  do {
    bytes_written = 0;
    copy_size = (count <= max_copy_size) \
                  ? count : max_copy_size;
    i = pcsp.in[0] ? 1 : 0;
    if (copy_size && !pcsp.in[i]) {
      copy_from_user(pcsp.buf[i], buffer, copy_size);
      pcsp.in[i] = copy_size;
      if (! pcsp.timer_on) pcsp_start_timer();
```

```
      bytes_written += copy_size;
      buffer += copy_size;
    }

    if (pcsp.in[0] && pcsp.in[1]) {
       interruptible_sleep_on(&pcsp_sleep);
       if (signal_pending(current)) {
         if (total_bytes_written + bytes_written)
           return total_bytes_written + bytes_written;
         else return -EINTR;
       }
    }
    total_bytes_written += bytes_written;
    count -= bytes_written;

  } while (count > 0);
  return total_bytes_written;
}
```

In den ersten freien Puffer werden zunächst mittels `copy_from_user()` Daten aus dem Nutzerbereich übertragen. Dies ist unbedingt notwendig, da der Interrupt unabhängig vom aktuellen Prozess auftreten kann und man somit die Daten nicht während des Interrupts aus dem Nutzerbereich holen kann. Der Zeiger `buffer` würde dann ja in den Nutzeradressraum des jeweils aktuellen Prozesses zeigen. Sollte der entsprechende Interrupt noch nicht initialisiert sein, wird er jetzt eingeschaltet (`pcsp_start_timer()`). Da die Übertragung der Daten zum Gerät in der ISR erfolgt, kann `write()` den nächsten Puffer füllen.

Sind alle Puffer voll, muss der Prozess angehalten werden, bis zumindest ein Puffer wieder frei ist. Dazu wird die Funktion `interruptible_sleep_on()` verwendet (siehe Abschnitt 3.1.5). Wurde der Prozess durch ein Signal aufgeweckt, so endet `write()`, sonst geht der Transfer weiterer Daten in den freigewordenen Puffer weiter. Betrachten wir nun den prinzipiellen Aufbau der ISR.

```
static int pcsp_do_timer(void)
{
  if (pcsp.index < pcsp.in[pcsp.actual]) {
    /* Ausgabe eines Bytes */
    …
  }
  if (pcsp.index >= pcsp.in[pcsp.actual]) {
    pcsp.xfer = pcsp.index = 0;
    pcsp.in[pcsp.actual] = 0;
    pcsp.actual ^= 1;
    pcsp.buffer = pcsp.buf[pcsp.actual];
    if (pcsp_sleep)
      wake_up_interruptible(&pcsp_sleep);
    if (pcsp.in[pcsp.actual] == 0)
      pcsp_stop_timer();
  }
```

```
  ...
}
```

Solange sich im aktuellen Puffer noch Daten befinden, werden diese ausgegeben. Ist der Puffer leer, wird auf den zweiten Puffer umgeschaltet und mittels `wake_up_interruptible()` der Prozess wieder aufgeweckt. Ist auch der zweite Puffer leer, wird der Interrupt wieder abgeschaltet. Das `if` vor dem Aufruf der Funktion ist eigentlich unnötig, da `wake_up_interruptible()` diesen Test selbst vornimmt. Er geschieht an dieser Stelle lediglich aus Zeitgründen.

Wie man sieht, passt diese ISR nicht in das zuvor erklärte Schema von langsamen und schnellen Interrupts. Das liegt daran, dass der Timerinterrupt in Linux ein langsamer Interrupt ist, aber für den PC-Speaker-Treiber aus Geschwindigkeitsgründen ein schneller Interrupt benötigt wird. Darum enthält der PC-Speaker-Treiber eine „dritte" Art, die gewissermaßen schnelle und langsame Interrupts enthält. Die Routine `pcsp_do_timer()` wird wie ein schneller Interrupt aufgerufen (allerdings mit gesetztem Interruptflag, d.h. unterbrechbar); gibt sie 0 zurück, wird der Interrupt beendet. Anderenfalls wird der ursprüngliche Timerinterrupt als langsamer Interrupt gestartet. Da der ursprüngliche Timerinterrupt viel seltener aufgerufen werden muss, bringt dieses Vorgehen einen großen Geschwindigkeitsvorteil.

7.4.7 IOCTL

Obwohl ein Gerätetreiber versucht, die Bedienung von Geräten nach außen hin möglichst zu abstrahieren, hat doch jedes Gerät seine speziellen Eigenschaften. Dazu können verschiedene Operationsmodi ebenso wie gewisse Grundeinstellungen gehören. Auch eine Einstellung von Geräteparametern zur Laufzeit, wie IRQs, I/O-Adresse usw., ist denkbar.

Die `ioctl`-Funktion erhält als Parameter ein Kommando sowie ein Argument. Da unter Linux

```
sizeof(unsigned long) == sizeof(void *)
```

gilt, kann als Argument auch ein Zeiger auf Daten im Nutzeradressraum übergeben werden. Normalerweise besteht die `ioctl`-Funktion deshalb aus einer großen `switch`-Anweisung, in der für das Argument eine entsprechende Typumwandlung stattfindet. `ioctl`-Aufrufe verändern zumeist nur Treiber-globale Variablen oder globale Geräteeinstellungen.

Betrachten wir ein Fragment der `ioctl`-Funktion des PC-Speaker-Treibers:

```
static int pcsp_ioctl(struct inode *inode, struct file *file,
        unsigned int cmd, unsigned long arg)
{
  unsigned long ret;
  unsigned long *ptr = (unsigned long *)arg;
  int i, error;
```

```
  switch (cmd) {
    case SNDCTL_DSP_SPEED:
      if (get_user(arg, ptr))
      return -EFAULT;
      arg = pcsp_set_speed(get_user(ptr));
      arg = pcsp_calc_srate(arg);
      return pcsp_ioctl_out(ptr, arg);
    …
    case SNDCTL_DSP_SYNC:
      pcsp_sync();
      pcsp_stop_timer();
      return (0);
    …
  }
}
```

Das Kommando `SNDCTL_DSP_SPEED` formt das Argument `arg` in einen Zeiger um und liest mit seiner Hilfe die neue Samplerate. Danach berechnet die Funktion `pcsp_calc_srate()` lediglich einige Zeitkonstanten in Abhängigkeit von der neuen Samplerate. `SNDCTL_DSP_SYNC` hingegen ignoriert das Argument völlig und ruft die Funktion `pcsp_sync()` auf. Diese Funktion hält den Prozess so lange an, bis alle Daten, die sich noch in Puffern befinden, abgespielt sind. Diese Synchronisation ist z. B. nötig, wenn während des Abspielens von Audiodaten die Samplerate oder der Abspielmodus (Mono oder Stereo) geändert wird oder die Ausgabe von Audiodaten mit anderen Ereignissen im Prozess synchronisiert werden soll.

Somit lässt sich die `ioctl`-Funktion auch dazu verwenden, andere Funktionen innerhalb des Treibers, die nicht vom Virtuellen Dateisystem erfasst werden, auszuführen. Ein weiteres Beispiel für dieses Verhalten ist im Treiber für die serielle Schnittstelle enthalten. Das Kommando `TIOCSERCONFIG` startet die automatische Erkennung des UART-Bausteins sowie der benutzten IRQs für die Schnittstellen.

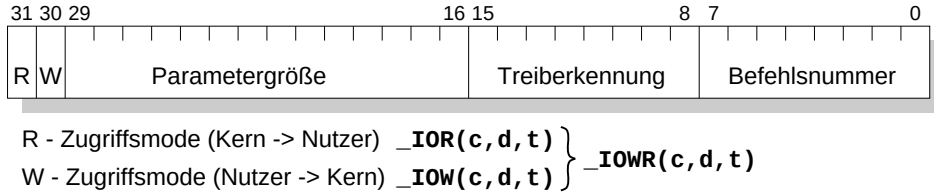


Abbildung 7.5: Kodierung der Ioctl-Kommandos

Bei der Entwicklung eines eigenen Treibers sollte die Codierung der IOCTL-Kommandos nicht willkürlich gewählt werden. Die Datei ‹linux/ioctl.h› enthält Makros, mit deren Hilfe die einzelnen Kommandos codiert werden sollten. Werden diese Makros verwendet, lassen sich die einzelnen IOCTL-Kommandos leicht decodieren. Die Bits 8–15 des Kommandos enthalten eine eindeutige Kennung des Gerätetreibers. Auf diese Weise kann sichergestellt werden, dass bei einer fälschlicherweisen Verwendung von IOCTL-Kommandos auf ein falsches Gerät ein Fehler zurückgeliefert wird, anstatt diesen Ge-

rätetreiber möglicherweise falsch zu konfigurieren. Es wird empfohlen, als eindeutige Treiberkennung die Major-Nummer des Gerätetreibers zu verwenden.

Den Makros zur Codierung der IOCTL-Kommandos wird als erstes Argument die Treiberkennung, als zweites die Kommandonummer übergeben.

`_IO(c,d)` für Kommandos, die kein Argument besitzen

`_IOW(c,d,t)` für Kommandos, die einen Wert des C-Typs `t` in den Nutzeradressraum zurückschreiben

`_IOR(c,d,t)` für Kommandos, die einen Wert des C-Typs `t` aus dem Nutzeradressraum lesen

`_IOWR(c,d,t)` für Kommandos, die sowohl lesen als auch zurückschreiben

Betrachten wir zum Abschluss als Beispiel die Definition einiger IOCTLs des Soundtreibers:

```
#define SNDCTL_DSP_RESET     _IO  ('P', 0)
#define SNDCTL_DSP_SYNC      _IO  ('P', 1)
#define SNDCTL_DSP_SPEED     _IOWR('P', 2, int)
#define SNDCTL_DSP_STEREO    _IOWR('P', 3, int)
```

Während also z. B. das Kommando `SNDCTL_DSP_RESET` ohne Argumente auskommt, liest `SNDCTL_DSP_SPEED` ein Argument vom Type `int` aus dem Nutzeradressraum und schreibt auch eines wieder zurück. Natürlich enthält die Datei `<linux/ioctl.h>` auch Makros, die die Decodierung der IOCTL-Kommandos erleichtern.

`_IOC_DIR(cmd)` liefert zurück, ob es sich um einen Ein- oder Ausgabebefehl handelt.

`_IOC_TYPE(cmd)` liefert die Treiberkennung zurück.

`_IOC_NR(cmd)` liefert das Kommando ohne Typinformation zurück.

`_IOC_SIZE(cmd)` liefert die Größe des übergebenen Argumentes in Bytes zurück.

Die Datei `Documentation/ioctl-number.txt` gibt Auskunft über bereits benutzte Gerätekennungen.

7.4.8 Poll

Seit der Version 2.1.23 unterstützt Linux zusätlich zum Systemruf `select` den System V Systemruf `poll`. Aus diesem Grunde war es notwendig, die ehemalige `select`-Funktion der File-Operationen zu ändern, um beide Systemrufe zu unterstützen. Sie wurde dabei in `poll` umbenannt. Obwohl `poll()` in unserem Beispiel nicht implementiert ist, soll hier die Funktionsweise beschrieben werden, da diese Funktion insbesondere für Zeichengeräte sinnvoll ist.

Die Aufgabe der `poll`-Funktion besteht in einer Überprüfung, ob vom Gerät gelesen oder Daten an das Gerät geschrieben werden können, ohne dass der lesende bzw. schreibende Prozess blockiert. Zusätzlich wird überprüft, ob eine Ausnahmebedingung vorliegt.

Da fast die gesamte Komplexität dieser Aufgabe vom Virtuellen Dateisystem übernommen wird, ist die Aufgabe der `poll`-Funktion einfach zu beschreiben.

Als Beispiel sei die `poll`-Implementierung des PS/2-Maustreibers betrachtet:

```
static unsigned int aux_poll(struct file *file, poll_table * wait)
{
  poll_wait(file, &queue->proc_list, wait);
  if (!queue_empty())
    return POLLIN | POLLRDNORM;
  return 0;
}
```

Die `poll`-Routine liefert als Rückgabewert einen Bitvektor, in dem entsprechende Bits gesetzt sind, je nachdem ob das Gerät zum Lesen (`POLLIN`) oder zum Schreiben (`POLLOUT`) bereit ist. Manche Geräte liefern besondere Daten wie beispielsweise Fehlercodes über die Kernelschnittstelle oder veranlassen die Behandlung einer Ausnahmesituation. Für diese Unterscheidung werden in `<asm/poll.h>` zusätzliche Makros definiert, die von `poll()` gesetzt werden können:

`POLLIN` Das Gerät hat Daten geliefert.

`POLLOUT` Das Gerät kann jetzt Daten entgegennehmen.

`POLLRDNORM` Es handelt sich um normale lesbare Daten. Dieses Bit wird üblicherweise zusammen mit `POLLIN` gesetzt.

`POLLRDBAND` Es handelt sich um „höherpriorisierte“ Daten.

`POLLWRNORM` Normale Daten können geschrieben werden. Dieses Bit wird üblicherweise zusammen mit `POLLOUT` gesetzt.

`POLLWRBAND` Es handelt sich um „höherpriorisierte“ Daten.

`POLLHUP` Beim Lesen wird mit diesem Bit markiert, dass der Treiber ein Ende des Datenstroms erkannt hat. Im TTY-Treiber wird dieses Bit beispielsweise gesetzt, wenn die Gegenstelle aufgelegt hat (z. B. das Modem ein Hangup signalisiert).

`POLLERR` Dieses Bit teilt einen aufgetretenen Fehler mit. Die FIFO-Implementation setzt z. B. dieses Bit, wenn der lesende Prozess terminiert (und `POLLHUP` wenn der schreibende Prozess terminiert ist).

`POLLPRI` Mit diesem Bit kann dem Prozess mitgeteilt werden, dass hochpriorisierte Daten zum Lesen bereitstehen. Ein gesetztes Bit veranlasst den Systemruf `select()`, eine Ausnahmesituation an den Prozess weiterzuleiten.

`POLLNVAL` Der übergebene Dateidescriptor ist invalid. Dieses Bit wird vom VFS automatisch gesetzt und `poll` nicht aufgerufen.

`POLLMSG` Derzeit definiert, aber nicht benutzt.

Für `wait` ungleich `NULL` muss der Prozess bis zur Verfügbarkeit des Geräts angehalten werden. Dazu wird jedoch nicht `sleep_on()` verwendet; diese Aufgabe erledigt die folgende Funktion:

```
void poll_wait(struct file * filp,
  wait_queue_head_t * wait_address, poll_table *p);
```

Als Argumente erwartet sie eine Warteschlage sowie das letzte der `poll`-Funktion übergebene Argument. Da `poll_wait()` sofort zurückkehrt, falls dieses Argument gleich `NULL` ist, kann man sich die Abfrage sparen und bekommt einen Funktionsaufbau wie in der oben gezeigten Beispielfunktion.

Falls das Gerät verfügbar wird (im Allgemeinen durch einen Interrupt angezeigt), weckt ein `wake_up_interruptible(wait_address)` den Prozess wieder auf. Dies zeigt der Maus-Interrupt des Treibers.

```
static void keyboard_interrupt(int irq, void *dev_id,
                               struct pt_regs *regs)
{
        spin_lock_irq(&kbd_controller_lock);
        handle_kbd_event();
        spin_unlock_irq(&kbd_controller_lock);
}

static unsigned char handle_kbd_event(void)
{
  …
      if (status & KBD_STAT_MOUSE_OBF)
        handle_mouse_event(scancode);
      else
        handle_keyboard_event(scancode);
  …
}

static inline void handle_mouse_event(unsigned char scancode)
{
  …

  add_mouse_randomness(scancode);
  if (aux_count) {
    int head = queue->head;

    queue->buf[head] = scancode;
    head = (head + 1) & (AUX_BUF_SIZE-1);
    if (head != queue->tail) {
      queue->head = head;
      kill_fasync(&queue->fasync, SIGIO, POLL_IN);
      wake_up_interruptible(&queue->proc_list);
    }
  }
}
```

7.4.9 Llseek

Diese Funktion ist im PC-Speaker-Treiber-Beispiel nicht implementiert. Sie ist für Zeichengeräte auch nur bedingt sinnvoll, da diese nicht positionieren können. Da jedoch die Standardfunktion `llseek()` im Virtuellen Dateisystem keine Fehlermeldung zurückgibt, muss man explizit eine `llseek`-Funktion definieren, falls der Treiber auf `llseek()` mit einer Fehlermeldung reagieren soll.

7.4.10 MMap

Diese Funktion ist im PC-Speaker-Treiber-Beispiel zwar nicht implementiert, dennoch ist Memory Mapping sehr nützlich und für manche Treiberimplementierungen unabdingbar, deshalb soll hier kurz darauf eingegangen werden.

Beim Datenaustausch zwischen Kernel- und Prozessbereich mit `read()` und `write()` müssen jedesmal die Daten hin- und herkopiert werden. Für einige Geräte ist das jedoch zu langsam. Bei einem Videotreiber beispielsweise möchte man kein flackerndes Bild, deswegen ist es effizienter, direkt in den Speicherbereich der Videokarte zu schreiben. Linux und andere Unix-artige Betriebssysteme stellen deshalb eine Technik bereit, die man Memory Mapping nennt. Ein physischer Speicherbereich wird dabei in den Adressraum eines Prozesses eingeblendet, so dass der Benutzerprozess direkt darauf zugreifen kann, ohne die Daten kopieren zu müssen.

Treiberseitig muss hierzu eine `mmap()`-Routine implementiert werden, die das eigentliche Mapping durchführt, benutzerseitig gibt es ebenfalls ein entsprechendes Gegenstück.

Der X-Server benutzt beispielsweise das `/dev/kmem`-Device, um sich Zugriff auf den Speicherbereich der Videokarte zu beschaffen. Der entsprechende Treiber im Kernel implementiert eine `mmap()`-Funktion, die diesen Speicherbereich mit `remap_page_range()` auf den Speicherbereich des Prozesses abbildet. Die entsprechenden Parameter werden in der Struktur `vm_area_str` übergeben.

```
static int mmap_mem(struct file * file, struct vm_area_struct * vma)
{
  unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

  /*
   * das Caching für diese Speicherseiten muß abgeschaltet
   * werden, wenn die zu mappenden Adressen über der
   * höchsten Speicheradresse liegen oder das O_SYNC
   * Flag gesetzt war
   */
  if (noncached_address(offset) || (file->f_flags & O_SYNC))
    vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);

  /*
   * Eingemappte Speicherseiten dürfen nicht ausgeswapt werden
   */
```

```
  vma->vm_flags |= VM_RESERVED;

  /*
   * nur echte Speicherseiten dürfen in die core-Datei
   */
  if (offset >= __pa(high_memory) || (file->f_flags & O_SYNC))
    vma->vm_flags |= VM_IO;

  if (remap_page_range(vma->vm_start, offset,
                       vma->vm_end-vma->vm_start,
                       vma->vm_page_prot))
    return -EAGAIN;
  return 0;
}
```

Im Beispiel wird der Offset für die physische Adresse der Videokarte benutzt, die über die Adresse im Prozessraum `vma->vm_start` gemappt werden soll. Für eigene Treiber können hier aber eigene Werte übergeben werden, die dann innerhalb des Kernels auf physische Adressen umgerechnet werden.

Nach dem Aufruf von `remap_page_range()` wird der vom Device benutzte Inode an die `vma`-Struktur gebunden; so wird markiert, dass sowohl für den Inode als auch für den entsprechenden Speicherbereich ein Mapping existiert. Das Beispiel zeigt auch eine Besonderheit, die z. B. bei Intel-Architekturen beachtet werden muss. Da bei der Intel-Architektur Hardwareadressen im normalen Speicherbereich liegen können, kann es unter Umständen möglich sein, dass der Prozessor den Zugriff auf diesen Bereich in seinem Cache hält, was bei echten Hardwareadressen nicht wünschenswert ist, da diese sofort reagieren soll. Daher wird die benutzte Speicherseite mit Hilfe der Funktion `pgprot_noncached()` markiert und somit aus dem Cache-Bereich ausgeblendet.

Eine andere Besonderheit ist zu beachten, wenn ein Speicherbereich gemappt werden soll, der im normalen Bereich des Kerns liegt, wie z. B. eine zuvor allozierte Speicherseite für einen DMA-Puffer, denn `remap_page_range()` funktioniert nur auf als reserviert markierten Speicherseiten. Mit `mem_map_reserve()` kann diese Markierung nachträglich vorgenommen werden; `mem_map_unreserve()` hebt sie wieder auf. Im Beispiel ist `page_ptr` eine zuvor allozierte Adresse für eine Speicherseite.

```
for (i = MAP_NR (page_ptr);
     i <= MAP_NR (page_ptr+PAGE_SIZE-1); i++)
{
   mem_map_reserve (i);
}
```

7.4.11 Fasync

Die Funktion `fasync()` bietet die Möglichkeit, einen Prozess über eintreffende Daten[15] asynchron zu informieren. Im Gegensatz zur Funktion `poll()`, mit der ein Prozess auf

15 Üblicherweise handelt es sich dabei um eintreffende Daten; es spricht aber nichts dagegen, auch über andere Ereignisse zu informieren.

Ereignisse wartet, kann mit Hilfe von `fasync()` das Signal `SIGIO` ausgelöst werden, auf das der Prozess dann reagieren kann.

Die gesammte Funktionalität dieser Funktion wird bereits vom Linux-Kern in Form von zwei Funktionen bereitgestellt.

Die Funktion `fasync_helper()` sorgt dafür, dass der sich für eintreffende Ereignisse interessierende Prozess abhängig vom Flag `on` auf die spezielle Warteliste vom Typ `struct fasync_struct` gesetzt oder von ihr entfernt wird.

```
int fasync_helper(int fd, struct file * filp, int on,
                  struct fasync_struct **fapp)
{
  struct fasync_struct *fa, **fp;
  struct fasync_struct *new = NULL;
  int result = 0;

  /*
   * Speicher Allokation kann schedulen, deshalb
   * wird sie vor den gesperrten Bereich vorgezogen
   */
  if (on) {
    new = kmem_cache_alloc(fasync_cache, SLAB_KERNEL);
    if (!new)
      return -ENOMEM;
  }

  /* suche zun"achst das Ende der Liste */
  for (fp = fapp; (fa = *fp) != NULL; fp = &fa->fa_next) {
    if (fa->fa_file == filp) {
      /* Eintrag existiert bereits */
      if (on) {
        fa->fa_fd = fd;
        kmem_cache_free(fasync_cache, new);
      } else {
        *fp = fa->fa_next;
        kmem_cache_free(fasync_cache, fa);
        result = 1;
      }
      goto out;
    }
  }

  /*
   * setze Dateidestriptor-Nummer auf die Liste
   * und trage alle Daten ein
   */
  if (on) {
    new->magic = FASYNC_MAGIC;
    new->fa_file = filp;
    new->fa_fd = fd;
```

```
    new->fa_next = *fapp;
    *fapp = new;
    result = 1;
  }
out:
  write_unlock_irq(&fasync_lock);
  return result;
}
```

Das Senden eines Signals an alle sie interessierenden Prozesse übernimmt die folgende Funktion.

```
void kill_fasync(struct fasync_struct **fp, int sig, int band)
{
  read_lock(&fasync_lock);
  __kill_fasync(*fp, sig, band);
  read_unlock(&fasync_lock);
}

void __kill_fasync(struct fasync_struct *fa, int sig, int band)
{
  while (fa) {
    struct fown_struct * fown;
    if (fa->magic != FASYNC_MAGIC) {
      printk(KERN_ERR "kill_fasync: bad magic number in "
             "fasync_struct!\n");
      return;
    }
    fown = &fa->fa_file->f_owner;
    /* Don't send SIGURG to processes which have not set a
       queued signum: SIGURG has its own default signalling
       mechanism. */
    if (fown->pid && !(sig == SIGURG && fown->signum == 0))
      send_sigio(fown, fa->fa_fd, band);
    fa = fa->fa_next;
  }
}
```

Wie man sieht, wird mit Hilfe der Funktion `send_sigio()` ein Signal `SIGIO` verschickt, falls sich ein Prozess als „Besitzer" eines Dateideskriptors eingetragen hat. Dazu dient der Systemruf *fcntl* (siehe A.2).

Die Implementation einer +prog+fasync+-Funktion im Treiber ist demzufolge einfach, wie wiederum das folgende Beispiel aus dem PS/2-Maustreiber zeigt:

```
static int aux_fasync(int fd, struct file *filp, int on)
{
  int retval;

  retval = fasync_helper(fd, filp, on, &queue->fasync);
  if (retval < 0)
    return retval;
```

```
  return 0;
}
```

Bei dieser einfachen Implementation bleibt noch die Frage, warum das Virtuelle Dateisystem dazu die Hilfe eines Treibers braucht und nicht alle Aufgaben selbst erledigt. Dazu müsste es jedoch die `fasync_struct` des Treibers kennen …

Sind nun Daten eingetroffen, müssen alle sich interessierenden Prozesse informiert werden. Dies übernimmt die `kill_fasync`-Funktion, in unserem Beispiel geschieht dies im Maus-Interrupt, nachdem festgestellt wurde, dass neue Mausdaten anliegen.

```
static inline void handle_mouse_event(unsigned char scancode)
{
  …
  kill_fasync(&queue->fasync, SIGIO, POLL_IN);
  …
}
```

`kill_fasync` benutzt übrigens in der aktuellen Implementation (siehe oben) den zweiten Parameter nur für einen Vergleich, es ist also nicht möglich, ein anderes Signal als `SIGIO` zu erzeugen.

Zusätzlich ist es nötig, einen Prozess von der `fasync_struct`-Liste zu nehmen, wenn er die Datei schließt. Üblicherweise ruft deshalb die `release`-Funktion des Treibers einfach die `fasync`-Funktion auf.

```
static int aux_release(struct inode * inode, struct file * file)
{
   aux_fasync(-1, file, 0);
   …
}
```

Da zum Entfernen eines Prozesses die Dateideskriptor-Nummer nicht benötigt wird, übergibt `release_mouse()` hier eine `-1`. Jeder andere Wert hätte denselben Effekt.

Wie im Linux-Kern üblich, lässt sich diese Funktion auch dann aufrufen, wenn der aktuelle Prozess nicht auf der Liste war, so dass nicht geprüft werden muss, ob sich der Prozess für asynchrone Ereignisse interessiert hat.

7.4.12 Readdir, Fsync

Diese Funktionen sind primär für Dateisysteme gedacht und im PC-Speaker-Treiber nicht implementiert.

Da die Struktur `file_operations` nicht nur für Geräte — im besonderen Zeichengeräte — verwendet wird, enthält sie von Gerätetreibern nicht genutzte Funktionen. So ist `readdir()` und `fasync()` für Geräte sinnlos.

7.5 Dynamische und statische Treiber

In früheren Linux-Versionen waren alle Gerätetreiber statisch in den Kernel eingebunden, heute können Treiber zu Laufzeit des Kerns dynamisch nachgeladen werden.

Das dynamische Nachladen von Treibern hat viele Vorteile, zum einen kann man erheblich Speicherplatz sparen, wenn das Modul nicht benötigt wird, zum anderen kann man über den Modul-Lademechanismus einige Parameter für das Treibermodul direkt beim Laden angeben (z. B. die Major-Nummer oder Puffergrößen). Der größte Vorteil ist aber die wesentlich kürzere Testzeit bei der Treiberentwicklung, da nicht jedesmal der ganze Kernel neu kompiliert und gebootet werden muss.

Der Aufwand, ein dynamisch ladbares Treibermodul zu entwickeln, ist minimal, man muss aber einige Regeln beachten. Die wesentlichen Bestandteile eines dynamisch ladbaren Treibers sind:

```
#include <linux/module.h>

…

int driver_init(void) {
  …
  if (register_chrdev(MY_MAJOR,"mydriver",&my_fops)) {
      printk("mydriver: unable to get major %d\n", MY_MAJOR);
      return -EIO;
  }
  …
}

void driver_term(void)
{
  …
  unregister_chrdev(MY_MAJOR,"mydriver");
  …
}

module_init(driver_init);
module_exit(driver_term);
```

Wird ein Treiber als Modul übersetzt, ändert sich die Bedeutung der Makros `module_init()` und `module_exit()`. Beim Laden des Moduls mit `insmod` wird die mittels `module_init()` markierte Initialisierungsroutine aufgerufen. Der Treiber muss sich nun beim System anmelden, indem er seine Fops-Struktur beim System registriert. Nun steht die angemeldete Funktionalität zur Verfügung, bis der Treiber mit `rmmod` wieder aus dem System entfernt wird. Bevor das Modul aus dem Speicher gelöscht wird, wird die mittels `module_exit()` markierte Routine aufgerufen, damit der Treiber vom System angeforderte Ressourcen wieder freigeben kann.

Hierbei ist darauf zu achten, dass alle Prozesse den Treiber wieder geschlossen haben, denn ein noch benötigtes Modul zu entfernen, könnte verheerende Folgen haben.

Pearson Education

ADDISON-WESLEY

8 Netzwerkimplementierung

Da sagte eine Stimme aus dem Chaos:
»Sei still, es könnte schlimmer kommen!«
und ich war stille, und es kam schlimmer.
Unbekannter Netzwerkadministrator

In der heutigen Zeit gehört es zu den Grundanforderungen an ein Betriebssystem, Netzwerkkommunikation zu unterstützen. Für Linux bestand diese Notwendigkeit von Anfang an. Diese Kommunikation bildet die Grundlage für eine Vielzahl von Netzdiensten. Hier wären etwa die den meisten Nutzern bekannten Dienste wie `http` (WWW), `ftp` (Dateitransfer), `telnet` und `rlogin` (*remote login*) zu nennen. Darüber hinaus gibt es auch die Möglichkeit, Dateisysteme von anderen Rechnern zu benutzen (NFS), *E-Mail* und *NetNews* zu empfangen und vieles mehr. Der Typ des dabei benutzten Netzwerks (OSI, IPX, UUCP usw.) ist für den Nutzer von zweitrangiger Bedeutung.

In der Unix-Welt dominieren die unter der Bezeichnung TCP/IP zusammengefassten Protokolle. Linux ist ein Unix-ähnliches Betriebssystem, deshalb stellt es natürlich eine TCP/IP-Implementierung zur Verfügung. Sie konzentriert sich hauptsächlich auf die Kommunikation mittels Ethernet. Doch Linux kann noch mehr. So ist es durch SLIP (*serial line interface protocol*), PLIP (*parallel line interface protocol*) bzw. PPP (*point to point protocol*) möglich, Rechner über die serielle bzw. parallele Schnittstelle und Modem zu verbinden. Besonders die Fähigkeiten des PPP-Protokolls sind beeindruckend, da es mit Hilfe von Modems und Telefonleitungen Netzwerkverbindungen in die ganze Welt herstellen kann.

Mit dem Protokoll AX.25 ist in Linux auch eine Möglichkeit zur Rechnerkommunikation über Funk vorhanden. Auch die Kommunikation mit IPX, einem von Novell entwickelten Protokoll, wurde realisiert. Die Datenwelt von Apple ist über eine Umsetzung des AppleTalk-Protokolls erreichbar. Für AppleTalk und IPX gibt es auch Software-Pakete, die das Drucken und den Dateizugriff ermöglichen.

In diesem Kapitel wollen wir uns mit den Besonderheiten der TCP/IP-Umsetzung in Linux beschäftigen. Es ist nicht das Anliegen der Autoren, eine Beschreibung der Funktionsweise von TCP/IP zu geben,[1] sondern vielmehr auf das Implementierungsdesign unter Linux einzugehen. Deshalb sind für das Verständnis Kenntnisse in der Programmiersprache C sowie ein Verständnis für die grundsätzlichen Zusammenhänge im TCP/IP notwendig.

1 Als weiterführende Literatur zum Thema TCP/IP seien die Bücher [Com91], [CS91], [Ste94] und [WE94] empfohlen.

8.1 Einführung und Überblick

Für den „normalen“ Programmierer ist der Zugriff auf Netzwerkdienste über Sockets möglich. Ihre Funktionalität wurde unter Linux erweitert. Die Schnittstelle besteht aus den folgenden C-Bibliotheksroutinen:

```
int socket(int addr_family,int type,int protocol);
int bind(int s,struct sockaddr *address,int address_len);
int connect(int s,struct sockaddr *address,int address_len);
int listen(int s,int backlog);
int accept(int s,struct sockaddr *address,int *address_len);
int getsockname(int s,struct sockaddr *address,
                int *address_len);
int getpeername(int s,struct sockaddr *address,
                int *address_len);
int socketpaire(int addr_family,int type,int protocol,
                int fds[2]);
int send(int s,char *msg,int len,int flags);
int sendto(int s,char *msg,int len,int flags,
           struct sockaddr *to, int tolen);
int recv(int s,char *buf,int len,int flags);
int recvfrom(int s,char *buf,int len,int flags,
             struct sockaddr *from,int *fromlen);
int shutdown(int s,int how);
int setsockopt(int s,int level,int oname,char *ovalue,
               int olen);
int getsockopt(int s,int level,int oname,char *ovalue,
               int *olen);
int sendmsg(int s,struct msghdr *msg,int flags);
int recvmsg(int s,struct msghdr *msg,int flags);
```

Alle diese Funktionen basieren auf dem Systemruf *socketcall* (siehe Seite 387). Darüber hinaus können mittels des Systemrufs *ioctl* für Socketdateideskriptoren netzwerkspezifische Konfigurationen verändert werden.

Da die C-Bibliotheksroutine `socket()` einen Dateideskriptor zurückgibt, sind natürlich auch die normalen I/O-Systemrufe wie *read* und *write* anwendbar.

Ein Rechner kann über unterschiedliche Hardware mit einem Netzwerk verbunden werden. Das können zum Beispiel Ethernetkarten und D-Link-Adapter sein. Die Unterschiede werden von einer einheitlichen Schnittstelle, den Netzwerkgeräten, verborgen. Die den Ethernetkarten zugeordneten Netzwerkgeräte heißen `eth0`, `eth1` usw. Die Namen für die Geräte von SLIP-Verbindungen lauten `sl0`, `sl1`, … und analog für PLIP `plip0`, `plip1`, …

Für diese Netzwerkgeräte existiert keine Repräsentation im Dateisystem. Sie können nicht wie „normale“ Geräte durch das Kommando `mknod` in dem Verzeichnis `/dev/` angelegt werden. Ein Netzwerkgerät kann nur angesprochen werden, wenn die Initialisierungsfunktion die entsprechende Hardware gefunden hat.

8.1.1 Das Schichtenmodell der Netzwerkimplementation

Da die Auseinandersetzung mit Netzwerkspezifika eine ausreichend komplizierte Aufgabe darstellt, ist auch hier, wie beim Dateisystem, eine Schichtenstruktur (*Layering*) zu finden. Die einzelnen Schichten entsprechen unterschiedlichen Abstraktionsniveaus. Dabei steigt das Abstraktionsniveau von Schicht zu Schicht, ausgehend von der Hardware.

Wenn ein Prozess über das Netzwerk kommuniziert, benutzt er dazu die Funktionen der BSD-Socketschicht. Sie übernimmt ähnliche Aufgaben wie das Virtuelle Dateisystem. Diese Schicht verwaltet eine allgemeine Datenstruktur für Sockets, die wir als BSD-Sockets bezeichnen. Die Wahl der BSD-Socketschnittstelle ist durch ihre weite Verbreitung und die daraus resultierende einfache Portierung der ohnehin meist komplexen Netzwerkapplikationen begründet.

Unter dieser Schicht ist die INET-Socketschicht angeordnet. Sie verwaltet die Kommunikationsendpunkte für die auf IP basierenden Protokolle TCP und UDP. Sie werden durch die Datenstruktur `sock` repräsentiert, die wir als INET-Sockets bezeichnen wollen. In den bisher erwähnten Schichten wird noch keine Unterscheidung der Sockets der Adressfamilie `AF_INET` anhand ihrer Typen vorgenommen. Die Schicht, auf der die INET-Socketschicht aufsetzt, wird durch den Typ des Sockets bestimmt. Das können die Schichten UDP, TCP oder direkt IP sein. Durch die UDP-Schicht wird das *User Datagram Protocol* auf der Basis von IP realisiert. Analog implementiert die TCP-Schicht das *Transmission Control Protocol* für zuverlässige Kommunikationsverbindungen. In der IP-Schicht ist das *Internet Protocol Version 4* implementiert. Dort treffen alle Kommunikationsströme der oberen Schichten zusammen. Die Sockets von anderen Typen bleiben bei diesen Betrachtungen jedoch unberücksichtigt.[2] Unter der IP-Schicht befinden sich die Netzwerkgeräte. An sie übergibt das IP die fertigen Pakete. Die Netzwerkgeräte sorgen dann für den physischen Transport der Informationen.

Eine echte Kommunikation findet immer zwischen zwei Seiten statt, wobei es einen wechselseitigen Informationsfluss gibt. Aus diesem Grund sind die entsprechenden Schichten auch in der umgekehrten Richtung miteinander verbunden. Das heißt, dass IP-Pakete beim Empfänger von seinen Netzwerkgeräten an die IP-Schicht übergeben und von ihr weiterbearbeitet werden. Das Zusammenwirken der einzelnen Schichten wird in Abbildung 8.1 verdeutlicht.

8.1.2 Die Reise der Daten

Um einen Einblick in das Zusammenspiel der einzelnen Teile der Netzwerkimplementation zu bekommen, werden wir die Daten begleiten, die von Prozess A über das Netzwerk zu Prozess B geschickt werden.

Wir setzen voraus, dass beide Prozesse bereits einen Socket erzeugt haben und mittels `connect()` bzw. `accept()` miteinander verbunden sind. Dabei beschränken wir uns auf die Betrachtung einer TCP-Verbindung unter Linux. Es sollen Daten vom Prozess A

2 Es gibt weitere Typen für Sockets wie ATM, IPv6, AX.25 uvm.

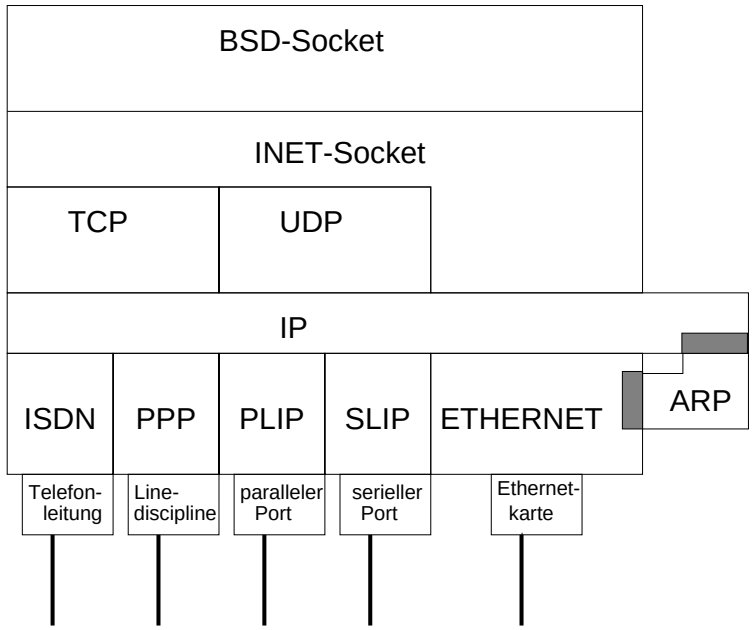


Abbildung 8.1: Die Layerstruktur des Netzwerks

zu Prozess B gesendet werden. Sie sind in einem Puffer der Länge `length` gespeichert; der Zeiger `data` verweist auf die Daten. Prozess A enthält das folgende Codefragment:

```
write(socket,data,length);
```

Damit ruft er die Kernfunktion `sys_write()` (siehe Abschnitt 6.2.9 und Seite 373) auf, die ein Bestandteil des Virtuellen Dateisystems ist. Sie testet einige Bedingungen, unter anderem, ob eine Write-Operation in den Dateioperationsvektor des Deskriptors eingetragen ist. Um das Virtuelle Dateisystem zu benutzen, stellt ein Socket die klassischen Dateioperationen in einem Vektor bereit.

Die Schreiboperation der BSD-Sockets heißt `sock_write()`. Sie erfüllt nur Verwaltungsfunktionen. Sie sucht die mit der Inode-Struktur assoziierte Socketstruktur. Danach werden die Parameter der Schreiboperation in eine Message-Struktur übertragen. `sock_write()` ruft dann die Sendefunktion `sock_sendmsg()` auf. Ihr werden als Parameter der Zeiger auf die BSD-Socket-Datenstruktur, der Zeiger auf die Message-Struktur und die Länge der Daten übergeben.

Die Funktion `inet_sendmsg()` entnimmt dem übergebenen BSD-Socket einen Zeiger (`sk`) auf die INET-Socket-Struktur `sock`. Diese Struktur beinhaltet in unserem Beispiel die wesentlichen in den TCP- und IP-Ebenen verwendeten Daten. Der Zeiger `prot` dieser Struktur verweist auf den Operationsvektor der TCP-Implementierung. Die `inet_sendmsg`-Funktion ruft die Sendeoperation dieses Vektors, `tcp_sendmsg()`, auf. Sie bekommt als Parameter den Zeiger auf den INET-Socket, den Zeiger auf die Message-Struktur und die Länge der Daten.

Bis hierher wurden die Daten nur durch die verschiedenen Abstraktionsniveaus weitergereicht. In `tcp_sendmsg()` beginnt die eigentliche Behandlung der kommunikationsrelevanten Aspekte. Zuerst wird auf einige Fehlerbedingungen getestet, zum Beispiel darauf, ob sich der Socket nicht im sendebereiten Zustand befindet. Nun wird mit der Funktion `tcp_alloc_skb` Speicher angefordert, der eine `sk_buff`-Struktur, den Header und das TCP-Segment aufnehmen wird. Die Funktion `tcp_sendmsg()` initialisiert die `sk_buff`-Struktur. Nun werden die Daten aus dem Adressraum des Prozesses in das Paket kopiert (siehe `copy_from_user()` in Abschnitt 4.1.2). Normalerweise wird danach die Prüfsumme berechnet. Zur Optimierung gibt es die Funktion `csum_and_copy_from_user()`, die diese beiden Aktionen in einem Schritt durchführt. Übersteigt die Länge der Daten die maximale Segmentgröße (MSS), werden sie auf mehrere Pakete verteilt. Es ist aber auch möglich, dass kurze Datenblöcke mehrerer Sendeoperationen zu einem Paket zusammengefasst werden. Abschließend wird die Funktion `tcp_send_skb()` aufgerufen. Hier werden die Sequenznummern für das TCP-Protokoll weitergezählt. Anschließend wird mit Hilfe eines Timers indirekt oder direkt die Funktion `tcp_transmit_skb()` aufgerufen. Hier wird dann der TCP-Header in das Paket eingefügt. Anschließend wird die TCP-Prüfsumme berechnet und die Funktion `ip_queue_xmit()`) aufgerufen. In dieser Funktion nun erfolgt das IP-Routing und die Erstellung des IP-Headers. Weiter geht es mit den Funktionen `ip_queue_xmit2()`, `ip_output()`, `ip_finish_output()` und `ip_finish_output2()`. Letztere kopiert den MAC-Header aus dem Header-Cache in das Paket und ruft schließlich `dev_queue_xmit()` auf.

Eine Besonderheit von Linux ist, dass alle Header linear aufeinander folgend in den Speicher geschrieben werden. In anderen TCP/IP-Implementationen wird das Paket als Vektor einzelner Fragmente gespeichert.

Die `dev_queue_xmit`-Funktion ruft schließlich die `hard_start_xmit()`-Funktion des Gerätes auf. Für eine 3COM-Karte zeigt sie auf die Funktion `boomerang_start_xmit()`. Diese Funktion übergibt die Daten dem Netzwerkadapter, der sie dann in das Ethernet sendet.

Die Hälfte des Weges ist jetzt geschafft. Die Daten — eingebettet in ein Ethernetpaket — werden von einer Netzwerkkarte am Zielrechner empfangen. Gehen wir z. B. davon aus, dass es sich bei dem Adapter um eine WD8013-Karte handelt.

Nachdem die Netzwerkkarte das Ethernetpakets empfangen hat, löst einen Interrupt aus. Dieser wird von der Funktion `vortex_interrupt()` behandelt. Gab es bei der Übertragung des Pakets über das Ethernet keine Fehler, wird die Funktion `vortex_rx()` mit einem Verweis auf das Netzwerkgerät aufgerufen. Sie schreibt mittels DMA-Transfer das Paket in einen mit `dev_alloc_skb()`+ neu angelegten Puffer. In diesem Puffer ist wie beim Versenden Platz für die Struktur `sk_buff` enthalten. Diese wird nach dem Transfer mit Hilfe der Funktion `eth_type_trans()` entsprechend initialisiert. Ist dies geschehen, wird die Funktion `netif_rx()` mit dem Paket als Argument aufgerufen. Sie fügt es an die `input_pkt_queue`-Liste für die aktuelle CPU an. Alle bisher erläuterten Funktionen zum Empfangen von Paketen werden innerhalb des Interrupts ausgeführt. Durch `netif_rx()` wird dann der *soft interrupt* für den Empfang von Paketen ausgelöst.

Bei gesetzter Markierung in der Maske ruft `do_softirq()` die Funktion `net_rx_action()` auf. Die Funktion `do_softirq()` wird nach Systemrufen und langsamen (normalen) Interrupts aufgerufen. Der Aufruf erfolgt nicht, wenn ein Interrupt einen anderen oder `do_softirq()` selbst unterbrochen hat. Weitere Informationen über den Soft-IRQ-Mechanismus findet der Leser im Abschnitt 3.2.3.

Die Funktion `net_rx_action()` setzt den `raw`-Zeiger der Unions `h` und `nh` der `sk_buff`-Struktur auf den Anfang des Protokollpakets hinter dem Ethernetheader. Der Pakettyp im Ethernetheader bestimmt dann, welche Protokollempfangsfunktion aufgerufen wird. Bei den SLIP- und PLIP-Protokollen ist der Typ nicht im Header des Pakets enthalten; er wird beim Empfang im `protocol`-Feld der `sk_buf`-Struktur direkt eingetragen.

Im betrachteten Fall ist ein IP-Paket empfangen worden, und die Empfangsfunktion `ip_rcv()` wird aufgerufen. In ihr zeigt sich der Vorteil der Union `nh`. In der *soft interrupt*-Routine ist der `raw`-Zeiger auf den Header des Protokollpakets gesetzt worden. Nun kann auf den IP-Header über den Zeiger `iph` der `nh`-Union zugegriffen werden, ohne dass er extra initialisiert werden muss, da er mit dem `raw`-Zeiger identisch ist. Hier wird auch die Checksumme des IP-Paketes überprüft. Anschließend wird `ip_rcv_finish()` aufgerufen.

In `ip_rcv_finish()` wird als Erstes das genaue Ziel des Pakets bestimmt. Dazu wird die Funktion `ip_route_input()` aufgerufen, die ihr Ergebnis im `dst`-Feld des Buffers einträgt. Wenn vorhanden, werden nun die IP-Optionen des Paketes bearbeitet. Dann ruft man die `input`-Funktion aus dem `dst`-Feld auf. In unserem Fall[3] ist das `ip_local_deliver()`. Hier wird nun, falls es sich bei dem Paket um ein Fragment handelt, mit Hilfe der Funktion `ip_defrag()` die Defragmentierung vollzogen.

Weiter geht es in der Funktion `ip_local_deliver_finish()`. Der `raw`-Zeiger in `h` und `nh` der `sk_buff`-Struktur wird an das Ende des IP-Headers gesetzt und zeigt damit wieder auf den Anfang des Headers des nächsten Protokolls. Dieses Protokoll ist im Protokollfeld des IP-Headers festgelegt. In unserem Fall ist das TCP. Abhängig von diesem Protokoll wird nun die entsprechende Protokollempfangsfunktion aufgerufen; im Falle von TCP ist das `tcp_v4_rcv()`. Dort wird mit der Funktion `__tcp_v4_lookup()` anhand der Ziel- und Absenderadressen sowie der Ziel- und Quellportnummern der INET-Socket ermittelt, für den das TCP-Segment bestimmt ist. Nach einigen Konsistenztests kopiert `tcp_rcv_established()` die Daten mit Hilfe der Funktion `tcp_copy_to_iovec()` aus dem `sk_buff` in den Speicher des empfangenden Prozesses. Bei `tcp_rcv_established()` handelt es sich um eine optimierte Funktion für den am häufigsten benutzten Ausführungsweg. Das heißt, dass sich der TCP-Socket im *established*-Zustand befindet und schon den Systemruf `read()` bzw. `rcvmsg()` aufgerufen hat und noch der aktive Prozess ist. Falls eine der Vorbedingungen für `tcp_rcv_established()` nicht erfüllt ist, wird `tcp_data()` aufgerufen. Dort werden ganz konventionell alle Protokolloptionen überprüft und die Daten am Socket in die Liste für die empfangenen Daten eingetragen. Sind neue Daten in der

3 Für IP-Pakete, die weitergeleitet werden sollen, wird im `dst`-Feld `ip_forward()` eingetragen.

Reihenfolge des Datenflusses eingegangen, werden entsprechende Acknowledge-Pakete verzögert versandt, und die `data_ready`-Operation des INET-Sockets wird aufgerufen. Sie weckt alle Prozesse, die auf ein Ereignis am Socket warten. Die Verzögerung der Acknowledgements ist notwendig, um nicht unnötig Pakete über das Netz zu senden. Bis jetzt sind alle Aktionen für das Empfangen eines Pakets im Kern außerhalb des Programmflusses eines Prozesses abgelaufen. Die dafür verbrauchte Rechenzeit lässt sich keinem Prozess zuordnen.

Der Prozess B möchte die vom Prozess A gesendeten Daten empfangen. Dazu führt er eine Leseoperation mit dem Socket-Dateideskriptor aus:

```
read(socket,data,length);
```

Über mehrere Abstraktionsstufen wird dieser Aufruf einer C-Bibliotheksfunktion über `sys_read()`, `sock_read()`, `sock_recvmsg()`, `inet_recvmsg()` und `tcp_recvmsg()` weitergereicht. Ist der Empfangspuffer des INET-Sockets leer, muss der Prozess blockieren. Es besteht aber die Möglichkeit, das Blockieren zu verhindern, indem mit *fcntl* das Flag `O_NONBLOCK` gesetzt wird. Wie im letzten Absatz schon erläutert, wird der Prozess wieder geweckt, wenn Daten eingetroffen sind. Nach dem Wecken oder wenn beim Systemruf *read* schon Daten im Puffer vorhanden sind, werden diese an die Adresse `data` des Nutzerspeicherbereichs des Prozesses kopiert.

Dort endet der Weg der Daten vom Prozess A zum Prozess B. Er hat uns über verschiedene Ebenen des Betriebssystems geführt. Die Daten sind nur viermal kopiert worden, vom Nutzerbereich des Prozesses A in den Kernelspeicher, von dort auf die Netzwerkkarte, von der Netzwerkkarte im anderen Rechner in den Kernelspeicher und von dort in den Nutzerbereich des Prozesses B. Bei der Linux-Implementierung des TCP/IP-Codes hat man sich sehr darum bemüht, unnötige Kopieroperationen zu vermeiden.

Die Netzwerkimplementation ist sehr eng in sich verwoben. Es gibt eine Vielzahl von aufeinander basierenden Funktionen, deren Zugehörigkeit zu bestimmten Schichten nicht immer klar ist. Ein Blick auf den Quellcode zeigt, dass viele Funktionen sehr groß (mehr als 200 Quelltextzeilen) und damit kaum überschaubar sind. Sicherlich ist die Komplexität der C-Quellen von der Materie abhängig. Sie ist aber ein deutlicher Hinweis darauf, wie wichtig ein gutes Design der Netzwerkimplementation ist. In der Linux-Version 1.2 waren die Schnittstellen zwischen den Schichten auf IP zugeschnitten. Doch beim Übergang zur Version 2.0 wurden schon einige Verbesserungen integriert, und die Versionen 2.2 und 2.4 haben weitere Generalisierungen vorgenommen. Diese Entwicklung hat jedoch die Komplexität des Codes weiter erhöht.

Es ist eine weit verbreitete Meinung, dass eine Netzwerkimplementation zwischen hoher Geschwindigkeit und sauberer Strukturierung balancieren muss. Die Autoren sind aber der Meinung, dass sich beide Punkte nicht unbedingt ausschließen. Andere Bereiche des Kerns (z. B. das Virtuelle Dateisystem) beweisen das.

8.2 Wichtige Strukturen

Ein Schritt auf dem Weg zu einer sauberen Strukturierung ist die richtige Definition der Datenstrukturen, die die Grundlage jeder Funktionalität eines Netzwerks bilden. Deshalb sollen in diesem Abschnitt in die vielen unterschiedlichen Datenstrukturen der Netzwerkimplementation von Linux eingeführt werden.

8.2.1 Die `socket`-Struktur

Die `socket`-Struktur ist die Basis für die Implementierung der BSD-Socketschnittstelle. Sie wird mit dem Systemruf *socket* angelegt und initialisiert. In diesem Abschnitt werden nur die Besonderheiten von Sockets der Adressfamilie `AF_INET` behandelt.

```
struct socket {
   socket_state        state;
```

In `state` wird der aktuelle Zustand des Sockets vermerkt. Die wichtigsten Zustände sind `SS_CONNECTED` und `SS_UNCONNECTED`.

```
   long                flags;
   struct proto_ops    *ops;
   struct inode        *inode;
```

Der `ops`-Zeiger eines Sockets der INET-Adressfamilie zeigt auf den Operationsvektor `inet_stream_ops` oder `inet_dgram_ops`. Dort sind die spezifischen Operationen für diese Adressfamilie eingetragen. `inode` ist ein Rückverweis auf die Inode, die zu diesem Socket gehört.

```
   struct fasync_struct *fasunc_list;
   struct file         *file;
```

In `file` befindet sich ein Verweis auf die primäre File-Struktur, die mit diesem Socket assoziiert ist.

Daraus ergeben sich aber auch gewisse Probleme bei der asynchronen Behandlung von Dateien. Verschiedene File-Strukturen können auf ein und dieselbe Inode und damit auf denselben BSD-Socket verweisen. Wenn von den Prozessen die asynchrone Benutzungsart für diese Datei ausgewählt worden ist, müssen auch alle Prozesse bei Ereignissen benachrichtigt werden. Deshalb sind sie in der `fasync_list` vermerkt. In Abschnitt 3.1.1 und in Abschnitt 6.2.9 wird die Beziehung zwischen Inodes und File-Struktur genauer erklärt.

```
   struct sock         *sk;
```

Der `sk`-Zeiger weist auf die der Adressfamilie entsprechende Unterstruktur des Sockets. Bei `AF_INET` ist das der INET-Socket (siehe Abbildung 8.2). In Linux wird jede Datei durch eine Inode beschrieben. Auch für jeden BSD-Socket gibt es eine Inode, so dass eine Eins-zu-eins-Abbildung zwischen den BSD-Sockets und den zugehörigen Inodes existiert. Da sich gegenüber den vorherigen Versionen von Linux die `sock`-Struktur

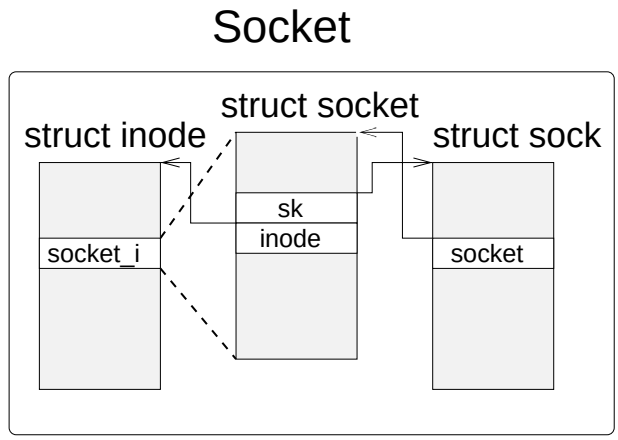


Abbildung 8.2: Der Socket und das Verhältnis zu seiner Unterstruktur

schon in der zugehörigen Inode befindet, ist kein separater Zeiger mehr notwendig, d.h. auf die Inode für einen BSD-Socket kann mit Hilfe des `sk`-Zeigers zugegriffen werden.

```
   struct wait_queue  *wait;
   short               type;
   unsigned char       passcred;
};
```

Für `type` sind `SOCK_STREAM`, `SOCK_DGRAM` und `SOCK_RAW` gültige Einträge. Sockets vom Typ `SOCK_STREAM` werden für TCP-Verbindungen, Sockets vom Typ `SOCK_DGRAM` für das UDP-Protokoll und Sockets vom Typ `SOCK_RAW` für das Senden und Empfangen von IP-Paketen verwendet.

8.2.2 Die Struktur `sk_buff` — Pufferverwaltung im Netzwerk

`sk_buff`-Puffer dienen zur Verwaltung einzelner Kommunikationspakete.

```
struct sk_buff {
   struct sk_buff          *next, *prev;
   struct sk_buff_head     *list;
```

Dieser Zeiger ist, genau wie die ersten beiden Zeiger in der Struktur, zur Verkettung in einer Ringliste und verschiedenen anderen Listen notwendig.

```
   struct sock             *sk;
```

`sk` zeigt auf den Socket, zu dem der Puffer gehört (siehe Abbildung 8.4).

```
   struct timeval           stamp;
```

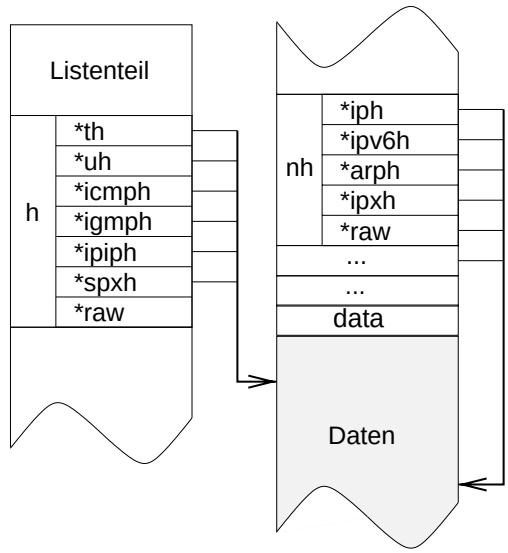


Abbildung 8.3: Die normale Benutzung der Struktur `sk_buff`

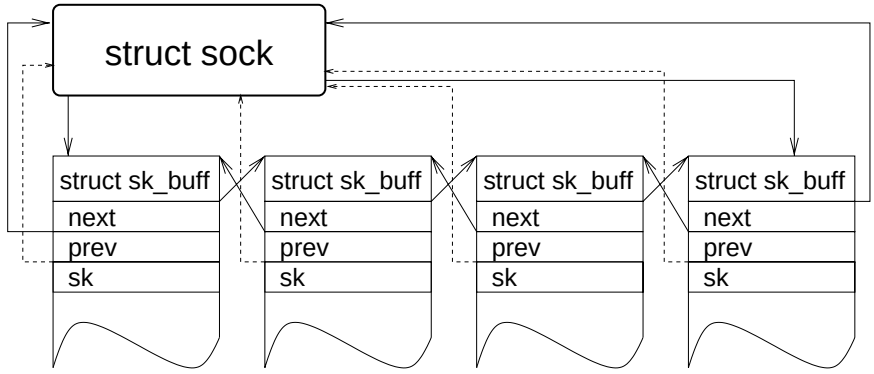


Abbildung 8.4: Die normale Lokalisation der `sk_buff`-Strukturen

`stamp` gibt an, wann das Paket zum letzten Mal übertragen worden ist. Die Puffer werden jedoch nicht nur zum Verschicken von Paketen benutzt, sondern auch für den Empfang. Wenn ein Paket von den Netzwerkgeräten an die höheren Schichten der Netzwerkimplementation weitergeleitet wird, vermerkt die Funktion `netif_rx()` die aktuelle Zeit in der Struktur `stamp`. Dazu benutzt sie die Kernelvariable `xtime`, die auch vom Timerinterrupt aktualisiert wird (siehe Abschnitt 3.2.5).

```
struct net_device       *dev;
```

Für die Verwaltung von Netzwerkpuffern ist es wichtig, von welchem bzw. über welches Netzwerkgerät ein Paket gesendet bzw. empfangen wird. Deshalb wird unter `dev` ein Zeiger auf das Gerät eingetragen.

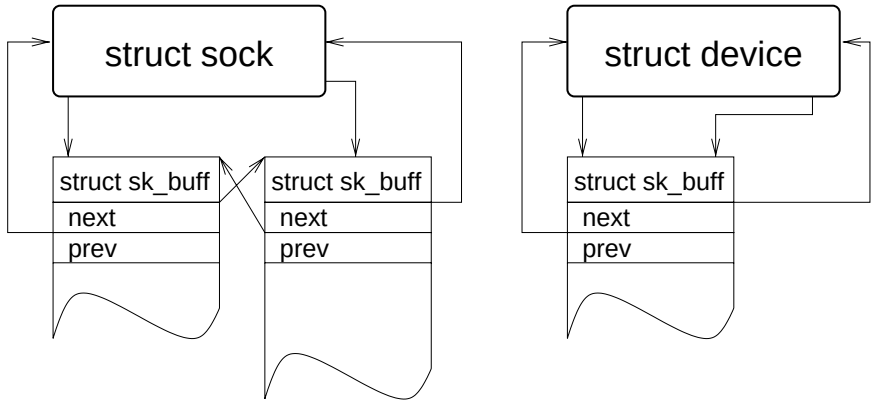


Abbildung 8.5: Übertragung eines Pakets vor dem Aufruf der `xmit`-Funktion (der Puffer ist beim Socket)

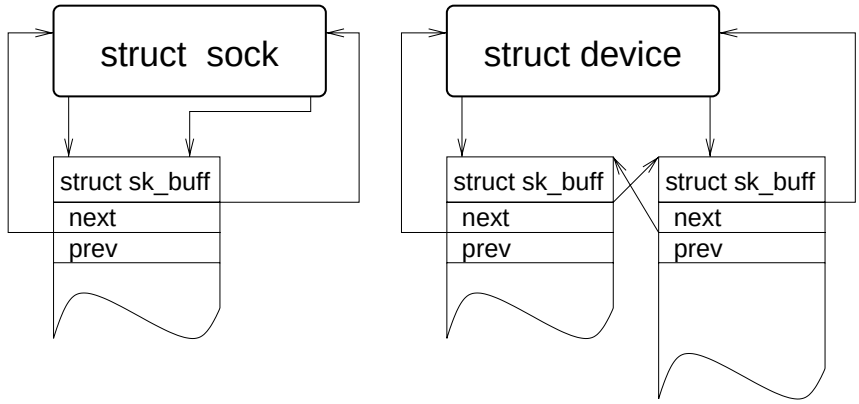


Abbildung 8.6: Übertragung eines Pakets nach dem Aufruf der `xmit`-Funktion (der Puffer ist jetzt beim Gerät, z. B. `eth0`)

```
union {
  struct tcphdr         *th;
  struct udphdr         *uh;
  struct icmphdr        *icmph;
  struct igmphdr        *igmph;
  struct iphdr          *ipiph;
  struct spxhdr         *spxh;
  unsigned char         *raw;
} h;
union {
  struct iphdr          *iph;
  struct ipv6hdr        *ipv6h;
  struct arphdr         *arph;
  struct ipxhdr         *ipxh;
  unsigend char         *raw;
```

```
  } nh;
  union {
    struct ethhdr         *ethernet;
    unsigend char         *raw;
  } mac;
```

Diese bereits erwähnte Union dient im Allgemeinen als Zeiger auf verschiedene Headerstrukturen innerhalb des Pakets.

```
  struct dst_entry        *dst;
```

Der `dst_entry` spielt eine zentrale Rolle bei der Verarbeitung von Paketen in der Netzwerkimplementation. Wenn die Route für ein Paket bestimmt wurde, wird hier ein entsprechender Eintrag hinterlassen. Dieser enthält u.a. auch Funktionszeiger für die weitere Bearbeitung des Paketes. So kann für eine häufig benutzte TCP-Verbindung gleich der Eintrag aus dem INET-Socket benutzt werden. Dieser enthält dann auch schon die Zeiger der Funktionen, die eine schnellstmögliche Auslieferung für das Paket ermöglichen. Eine weitere aufwendige Bestimmung der Route entfällt dadurch.

```
  char                    cb[48];
  unsigned int            len, csum;
```

`len` gibt die Länge des Paketes an, und `csum` enthält die Checksumme, falls eine berechnet wurde.

```
  volatile char           used;
  unsigned char           cloned,pkt_type,ip_summed;
  __u32                   priority;
  atomic_t                users;
  unsigned short          protocol,security;
  unsigned int            truesize;
  unsigned char           *head,*data,*tail,*end;

  void (*destructor)(struct sk_buff *);
};
```

Diese Elemente befassen sich mit der Verwaltung des Puffers und des zur Struktur gehörenden Speichers. Die Daten befinden sich nicht mehr direkt hinter der Struktur wie in den Implementierungen bis zu Version 1.2, sondern in einem eigenen Speicherblock. Dieser enthält einen Referenzzähler, der angibt, wie viele Puffer auf ihn verweisen. Dadurch funktioniert die Klone-Funktion sehr schnell. Sie kopiert die `sk_buff`-Struktur, erhöht den Referenzzähler der Daten und setzt in der `sk_buff`-Struktur `cloned` auf 1.

Hier noch einige Bemerkungen zu den Zeigern: `head` zeigt immer auf den Anfang des zum Puffer gehörenden Speicherblocks und `end` auf das Ende. `data` verweist auf den Anfang der eigentlich Nutzdaten des Pakets. Der Bereich zwischen `head` und `data` enthält den Header des Pakets. Die Nutzdaten befinden sich dann also in dem Bereich von `data` bis `tail`.

Bei der Benutzung von Net-Filtern enthält die Struktur noch einige weitere Elemente, die uns hier aber nicht interessieren.

Für die Verwaltung der `sk_buff`-Strukturen werden normalerweise doppelt verkettete Listen benutzt. Deshalb gibt es auch eine Struktur, die einen Listenkopf realisiert:

```
struct sk_buff_head {
   struct sk_buff          *next, *prev;
   __u32                    qlen;
   spinlock_t               lock;
};
```

Dabei zeigt `next` auf den Anfang der Liste und `prev` auf das Ende. `qlen` gibt die Anzahl der Listenelemente an. Das `lock`-Element dient zur Synchronisierung im Mehrprozessorbetrieb.

8.2.3 Der INET-Socket — spezieller Teil eines Sockets

In der INET-Socketstruktur werden die netzwerkspezifischen Teile der Sockets verwaltet. Sie wird für TCP-, UDP- und RAW-Sockets benötigt.

```
struct sock {
   struct sock           *next;
   struct sock          **pprev;
   struct sock           *bind_next;
   struct sock          **bind_pprev;
   struct sock           *prev;
   int                    hashent;
```

Um den INET-Socket für einen bestimmten Port schneller zu finden, werden die INET-Sockets außerdem noch in verschiedenen Hashtabellen verwaltet. Für diesen Zweck sind die folgenden Zeiger vorgesehen:

```
   __u32                  daddr;
   __u32                  saddr;
   __u32                  rcv_saddr;
   __u16                  dport;
   __u16                  sport;
   unsigned short         num;
   struct dst_entry      *dst_cache;
   int                    bound_dev_if;
```

Diese Felder enthalten die IP-Adressen und Portnumern für den INET-Socket. `rcv_saddr` gibt die Adresse an, an die der INET-Socket gebunden worden ist. `dst_cache` verweist auf eine Datenstruktur, die eine schnellere Verarbeitung der Pakete zulässt. Falls der INET-Socket an ein bestimmtes IP-Interface gebunden ist, enthält `bound_dev_if` den Index des Interface.

```
   rwlock_t               dst_lock,
                          callback_lock;
   atomic_t               refcnt;
   socket_lock_t          lock;
   wait_queue_head_t     *sleep;
```

`sleep` zeigt auf den Kopf einer Warteschlange, in der die Prozesse gespeichert sind, die bei Aktionen auf diesem INET-Socket blockierten.

```
atomic_t                rmem_alloc;
atomic_t                wmem_alloc;
atomic_t                omem_alloc;
unsigned int            allocation;
int                     rcvbuf;
int                     sndbuf;
int                     wmem_queued;
```

Die beiden Variablen `wmem_alloc` und `rmem_alloc` geben an, wie viel Speicher schon von diesem INET-Socket angefordert wurde. Die erste Variable gilt für das Schreiben und die zweite für das Lesen auf dem INET-Socket. `omem_alloc` zählt den Speicher, der nicht direkt zum Schreiben oder Lesen bnötigt wird (z. B. für die Optionsverarbeitung des IP). In `allocation` wird bei der Erzeugung des INET-Sockets vermerkt, mit welcher Priorität Speicher für diesen INET-Socket angefordert wird (siehe Abbildung 4.3). Die Werte in `sndbuf` und `rcvbuf` sind die oberen Schranken für den Speicherverbrauch des INET-Sockets beim Schreiben bzw. Lesen.

```
struct sk_buff_head     write_queue;
struct sk_buff_head     receive_queue;
```

`write_queue` enthält eine Liste der noch zu verschickenden Pakete. Die Pakete in der Liste `receive_queue` wurden vom Protokoll schon empfangen, aber noch nicht an den Prozess geliefert.

```
unsigned char           reuse;

volatile char           dead,done,urginline,
                        keepopen,linger,destroy,
                        no_check,broadcast,
                        bsdism;
unsigned char           debug;
unsigned char           rcvtstamp;
unsigned char           userlocks;
unsigned long           lingertime;
```

Diese Variablen enthalten verschiedene Flags und Werte, die für einen INET-Socket gesetzt werden können.

```
int                     proc;
```

In `proc` ist ein Prozess oder eine Prozessgruppe abgelegt, die bei der Ankunft von Out-of-band-Daten ein Signal erhalten sollen.

```
struct sock            *pair;
```

Bei der Protokolloperation `accept()` der INET-Sockets wird eine neue `sock`-Struktur angelegt. Auf die neu erzeugte Struktur zeigt dann `pair`.

```
struct {
  struct sk_buff *head;
  struct sk_buff *tail;
}                       back_log;
struct sk_buff_head     error_queue;
```

In `back_log` werden Pakete eingetragen, die für den INET-Socket bestimmt sind, aber momentan nicht in die richtige Liste eingefügt werden können, weil ein Prozess gerade Daten vom Socket holt.

```
struct proto           *prot;
```

Hier ist der Operationsvektor für das Protokoll vermerkt, mit dem der INET-Socket assoziiert ist. Im Normalfall wird das die Adresse einer der folgenden Strukturen sein: `tcp_prot`, `udp_prot` oder `raw_prot`.

```
unsigned char           shutdown;

union {
  struct tcp_opt        af_tcp;
  struct raw_opt        tp_raw4;
} tp_pinfo;

union {
  void                 *destruct_hook;
  struct unix_opt       af_unix;
  struct inet_opt       af_inet;
  struct packet_opt    *af_packet;
} protinfo;
```

Es gibt folgende private Daten für jede Adressfamilie:

```
int                     err,err_soft;
volatile unsigned char  state,zapped;
```

Bei `err` handelt es sich um einen Fehlerhinweis, der der `errno`-Variable unter C ähnelt. `state` gibt den Zustand des INET-Sockets an.

```
unsigned short          ack_backlog,
                        max_ack_backlog;
__u32                   priority;
unsigned short          type,
                        family;
unsigned char           localroute,
                        protocol;
struct ucred            peercred;
int                     rcvlowat;
```

`type` und `family` werden von der zugehörigen BSD-Socket-Struktur `socket` übernommen und bestimmen den Typ und die Familie des INET-Sockets. Mit Hilfe von `localroute` wird angezeigt, dass die Pakete nur lokal geroutet werden sollen.

```
    spinlock_t              timer_lock;
    struct timer_list       timer;
    struct timeval          stamp;
    long                    rcvtimeo,
                            sndtimeo;
```

Diese beiden Strukturkomponenten dienen zur Verwaltung von Timern, die für die Implementierung des TCP notwendig sind. `stamp` wird bei jedem neuankommenden Paket aktualisiert. Hier kann also genau festgestellt werden, wann das letzte Paket empfangen worden ist.

```
    struct socket           *socket;
```

Dieser Zeiger verweist auf den zugehörigen BSD-Socket.

```
    void                    *user_data;
    void                   (*state_change)(struct sock *sk);
    void                   (*data_ready)(struct sock *sk,
                             int bytes;
    void                   (*write_space)(struct sock *sk);
    void                   (*error_report)(struct sock *sk);
    void                   (*backlog_rcv)(struct sock *sk,
                             struct sk_buff *skb);
};
```

Die Funktion `state_change()` wird jedes Mal gestartet, wenn sich der Status des Sockets geändert hat. Analog dazu wird `data_ready()` aufgerufen, wenn Daten angekommen sind, `write_space()`, wenn sich der freie Speicher zum Schreiben vergrößert hat, und `error_report()` im Fehlerfall.

Im weiteren Text wird mit der Bezeichnung „Socket“ die Kombination eines BSD-Sockets mit einem INET-Socket bezeichnet (siehe Abbildung 8.2).

Beim Übergang zur Version 2.4 von Linux wurden weitere Elemente der `sock`-Struktur, die spezifisch für IP waren, in die protokollspezifische Datenstruktur `inet_opt` verschoben:

```
struct inet_opt
{
        int                     ttl,
                                tos;
        unsigned                cmsg_flags;
```

Diese Werte werden beim Erstellen eines IP-Headers benutzt, um die entsprechenden Felder des Protokollkopfes zu füllen.

```
        struct ip_options      *opt;
```

`opt` ist ein Zeiger auf eine Struktur, die die einzelnen für diesen INET-Socket zu verwendenden IP-Optionen beinhaltet. Sie sind bei der Bildung eines IP-Protokollkopfes zu berücksichtigen.

```
    unsigned char           hdrincl;
    __u8                    mc_ttl;
    __u8                    mc_loop;
    unsigned                recverr:1,freebind:1;
    __u8                    pmtudisc;
    int                     mc_index;
    __u32                   mc_addr;
    struct ip_mc_socklist *mc_list;
};
```

Die Elemente der Struktur, die mit dem Präfix `mc_` beginnen, dienen zur Realisierung des IP-Multicast-Protokolls.

8.2.4 Protokolloperationen in der `proto`-Struktur

Protokolle wie TCP und UDP werden unter Linux über eine abstrakte Schnittstelle angesprochen. Diese besteht aus einer Reihe von Operationen und ermöglicht es, Funktionen, deren Aktionen für alle Protokolle dieselben sind, nur einmal programmieren zu müssen. Dies hilft, Implementierungsfehler zu vermeiden und den Code so kompakt wie möglich zu halten.

```
struct proto {
    void            (*close)(struct sock *sk, long timeout);
```

`close()` leitet die zum Schließen eines Sockets notwendigen Maßnahmen ein. Für einen TCP-Socket wird zum Beispiel ein Paket mit dem ausstehenden ACK und einem FIN gesendet.

```
    int             (*connect)(struct sock *sk,
                               struct sockaddr *uaddr,
                               int addr_len);
    int             (*disconnect)(struct sock *sk, int flags);
    struct sock *   (*accept) (struct sock *sk, int flags,
                               int *err);
```

`connect()` ist für alle Protokolle zu implementieren, wohingegen `accept()` für verbindungslose Protokolle nicht notwendig ist. Die Semantik von `connect()` unterscheidet sich für verbindungslose und verbindungsorientierte Protokolle. Bei verbindungslosen Protokollen wird sie dazu benutzt, eine Adresse festzulegen, die für `write`-Aufrufe als Zieladresse benutzt wird. Für TCP hingegen baut `connect()` die Verbindung auf.

```
    int             (*ioctl)(struct sock *sk, int cmd,
                             unsigned long arg);
```

Mit Hilfe der `ioctl`-Funktion lassen sich unter anderem die Menge der noch nicht gelesenen oder übertragenen Daten eines TCP- oder UDP-Sockets bestimmen und die Debugging-Ausgaben ein- bzw. ausschalten.

```
    int             (*init)(struct sock *sk);
    int             (*destroy)(struct sock *sk);
```

Durch die `init`-Funktion des jeweiligen Protokolls werden die für diese Protokollimplementation notwendigen Felder des INET-Sockets initialisiert. `destroy` führt die Aufräumarbeiten durch, die beim Freigeben eines INET-Sockets notwendig werden, wie z. B. die Freigabe von Speicher.

```
    void            (*shutdown)(struct sock *sk, int how);
    int             (*setsockopt)(struct sock *sk, int level,
                                  int optname, char *optval,
                                  int optlen);
    int             (*getsockopt)(struct sock *sk, int level,
                                  int optname, char *optval,
                                  int *option);
```

Die erste Funktion wird zur Zeit nur von TCP-Verbindungen genutzt. Mit ihr kann eine TCP-Verbindung abgebrochen werden. Die beiden anderen Funktionen implementieren `setsockopt()` bzw. `getsockopt()` für das zugehörige Protokoll.

```
    int             (*sendmsg)(struct sock *sk, struct msghdr *msg,
                               int len);
    int             (*recvmsg)(struct sock *sk, struct msghdr *msg,
                               int len, int noblock, int flags,
                               int *addr_len);
    int             (*bind)(struct sock *sk,
                            struct sockaddr *uaddr, int addr_len);
```

Bei der Weiterentwicklung von Linux 1.2 haben sich erhebliche Änderungen an den Protokollinterfaces vollzogen. So wurden alle Sende- und Empfangsfunktionen durch `sendmsg` und `recvmsg` ersetzt. Die spezifischen Übergabeparameter werden in der Struktur `msghdr` übergeben. Außerdem sind ab der Version 2.0 die Funktionen `readv()` und `writev()` auf Sockets möglich. Mit `bind()` wird ein Socket an eine bestimmte Adresse gebunden. Dieser Zeiger wird bisher von keinem Protokoll benutzt.

```
    int             (*backlog_rcv)(struct sock *sk,
                                   struct sk_buff *skb);

    void            (*hash)(struct sock *sk);
    void            (*unhash)(struct sock *sk);
    int             (*get_port)(struct sock *sk,
                                unsigned short snum);
```

Die Hashfunktonen realisieren die Hashtabellen der einzelnen Protokolle, d. h. sie tragen sie ein und aus.

```
    char             name[32];
    int              inuse[NR_CPUS];
};
```

`name` gibt für die Fehlersuche den Namen des zugehörigen Protokolls an (z. B. TCP). Die anderen Werte haben statistischen Charakter und werden für SNMP benötigt.

Die gerade vorgestellte Struktur `proto` kann als Interface für die Protokolle der Familie `AF_INET` betrachtet werden. Eine ganz ähnliche Struktur beschreibt die Schnittstelle zu

der darüber liegenden BSD-Socketschicht. Der Name dieser Struktur ist `proto_ops`. Diese gibt es für jede implementierte Protokollfamilie. Seit der Version 2.2 von Linux sind das `AF_INET`, `AF_INET6`, `AF_IPX`, `AF_X25`, `AF_UNIX` und einige weitere.

8.2.5 Die allgemeine Struktur einer Socketadresse

Da Sockets für verschiedene Adressfamilien unterschiedliche Adressformate unterstützen müssen, gibt es eine allgemeine Adressstruktur, in der die Adressfamilie, die Portnummer und ein Feld für verschieden große Adressen enthalten sind. Für Internetadressen ist eine spezielle Struktur `sockaddr_in` definiert, die mit der allgemeinen Struktur `sockaddr` übereinstimmt.

```
struct sockaddr {
   unsigned short sa_family;   /* Adressfamilie AF_xxx */
   char           sa_data[14]; /* Anfang der Protokolladresse */
};

struct sockaddr_in {
   short int    sin_family;     /* Adressfamilie */
   unsigned short int sin_port; /* Portnummer */
   struct in_addr  sin_addr;    /* Internetadresse */

   /* Füllbytes zur sockaddr-Struktur */
   unsigned char __pad[__SOCK_SIZE__ - sizeof(short int)
                       - sizeof(unsigned short int)
                       - sizeof(struct in_addr)];
};
```

8.3 Netzwerkgeräte unter Linux

Wie wir bereits gesehen haben, gibt es unterschiedliche Hardware, die zur Vernetzung von Rechnern eingesetzt werden kann. Die Ansteuerung dieser Hardware variiert stark. Um dies vor den oberen Schichten zu verbergen, wurde eine abstrakte Schnittstelle zur Netzwerkhardware eingeführt. Damit wird es möglich, die oberen Netzwerkschichten unabhängig von der verwendeten Hardware zu implementieren. Offensichtlich wird hier das Konzept der Polymorphie bei der Betriebssystemprogrammierung angewendet.

In der Datenstruktur `net_device` wird ein abstraktes Netzwerkgerät verwaltet. Im Englischen wird auch oft von einem *Network Interface* gesprochen, wobei mit dieser Bezeichnung mehr die Schnittstelle zum Netzwerk als zur Hardware betont wird.

```
struct net_device {
   char                    name[IFNAMSIZ];
```

Jedes Netzwerkgerät hat bei Linux einen eindeutigen Namen, der sich in `name` befindet.

```
   unsigned long           rmem_end;
   unsigned long           rmem_start;
```

```
unsigned long           mem_end;
unsigned long           mem_start;
unsigned long           base_addr;
unsigned int            irq;
```

Mit diesen Elementen wird die Hardware des Geräts beschrieben. In `base_addr` und `irq` sind die für die PC-Architektur wichtige IO-Adresse und die Nummer des Interrupts, der zu dem Gerät gehört, vermerkt. `rmem_start` bis `rmem_end` und `mem_start` bis `mem_end` beschreiben den Empfangsspeicher bzw. den Sendespeicher des Geräts. Diese Parameter sind allerdings auf Ethernetkarten zugeschnitten. Für andere Geräte wird ein Teil dieser Strukturfelder mit einer anderen Semantik benutzt. Bei einem SLIP-Gerät ist in `base_addr` der Index der entsprechenden SLIP-Struktur vermerkt.

```
unsigned char           if_port;
unsigned char           dma;
```

Die Hardware einiger Netzwerkgeräte benutzt DMA für die Ein- und Ausgabe und für die Kommunikation mit IO-Bereichen. So musste die Struktur `net_device` für die Version 1.2 von Linux um entsprechende Felder in der Struktur erweitert werden. In `if_port` wird vermerkt, welcher Port (AUI, TP, …) der Karte benutzt wird.

```
unsigned long           state;
struct net_device      *next;
```

`state` gibt den internen Zustand von Netzwerkgeräten an. Die Netzwerkgeräte werden unter Linux in einer Liste verwaltet. Die Kernel-Variable `dev_base` zeigt auf das erste Element dieser Liste. Zur Verkettung wird `next` benutzt. Mit Hilfe des eindeutigen Namens und der Funktion `dev_get()` kann im Kernel auf ein Netzwerkgerät zugegriffen werden. Diese Funktion stellt fest, ob die für dieses Gerät notwendige Hardware vorhanden ist, und initialisiert die `net_device`-Struktur.

```
int     (*init)(struct net_device *dev);
int     (*uninit)(struct net_device *dev);
void    (*destructor)(struct net_device *dev);
```

Wie in Abschnitt 2.3 beschrieben wurde, kann vor dem Übersetzen eines Kerns festgelegt werden, welche Geräte er überhaupt auf Vorhandensein testen soll. Das gilt auch für Netzwerkgeräte. So gibt es in `drivers/net/Space.c` eine statische Liste von Strukturen des Typs `net_device`. In ihr befinden sich Elemente, die nur aus dem öffentlichen Teil der `net_device`-Struktur bestehen. Der öffentliche Teil erstreckt sich vom Anfang bis zur Komponente `init` der Struktur. Der Zeiger `dev_base` zeigt auf den Anfang dieser Liste. Durch Modifikation dieser Liste kann nun festgelegt werden, welche Geräte beim Booten auf Vorhandensein geprüft und welche Initialisierungsfunktionen benutzt werden. Besonders für die verschiedenen Typen von Ethernetkarten ist das wichtig, da in der `ethif_probe`-Funktion die einzelnen Kartentypen nacheinander getestet werden.

Wenn wir uns noch einmal die Abfolge der Aktionen beim Starten des Kerns aus Abschnitt 3.2.4 in Erinnerung rufen, sehen wir, dass dort `sock_init()` aufgerufen wird. Ihre Aufgabe ist es, den gesamten Netzwerkteil des Kernels zu initialisieren. Dabei werden die Voreinstellungen für die BSD-Sockets gesetzt, und im weiteren Ablauf wird die

Funktion `dev_init()` aufgerufen. Durch diese `init`-Funktion werden alle konfigurierten Netzwerkgeräte initialisiert. Dazu iteriert die Funktion über die Liste der Netzwerkgeräte, auf die `dev_base` zeigt. Für jeden Eintrag wird die `init`-Funktion aufgerufen. Sie füllt die gesamte `net_device`-Struktur mit korrekten Werten, d.h. auch die Funktionszeiger.

```
struct net_device       *next_sched;
struct net_device       *master;
int                      ifindex,iflink;
unsigned long            pkt_queue;
struct net_device       *slave;
```

Die letzten beiden Felder werden für das „load balancing“ benötigt. `pkt_queue` gibt dabei die Anzahl der noch zu bearbeitenden Pakete an, und `slave` zeigt auf eine weiteres Netzwerkgerät, das als Slave-Gerät arbeitet.

```
struct net_device_stats* (*get_stats)(struct net_device *dev);
struct iw_statistics* (*get_wireless_stats)
                              (struct net_device *dev);
```

`net_device_stats` ist die jeweilige Statistikfunktion für das Gerät. Sie kommt immer dann zum Einsatz, wenn statistische Informationen über das Netzwerkgerät von einem anderen Teil des Kerns erfragt werden. Mit `get_wireless_stats` bekommt man spezifische Informationen für Netzgeräte, die nicht auf kabelgebundenen Medien arbeiten.

```
unsigned long            trans_start, last_rx;
```

In diesen beiden Feldern wird vermerkt, wann das letzte Mal etwas gesendet (`trans_start`) bzw. empfangen (`last_rx`) wurde. Die Zeiteinheiten sind dabei Hundertstelsekunden, die der Variablen `jiffies` entnommen werden.

```
unsigned short           flags;
unsigned short           gflags;
unsigned                 mtu;
```

Diese Variablen werden vom IP-Protokoll benutzt. Sie können mit dem Systemkommando `ifconfig` beeinflusst werden (siehe Anhang B.7). Dabei gibt `mtu` die maximale Größe eines Pakets an, das von diesem Netzwerkgerät übertragen werden kann. Der Wert ist abzüglich des Hardware-Headers (z. B. der Header für Ethernetpakete) zu verstehen. Durch `flags` lässt sich das Verhalten beeinflussen. Die möglichen Werte sind in Tabelle 8.1 zusammengefasst.

```
unsigned short           type;
unsigned short           hard_header_len;
void                    *priv;
```

In `type` wird der Gerätetyp verzeichnet. Dabei ist eigentlich die Hardware gemeint. Doch bis jetzt benutzen alle den Typ `ARPHRD_ETHER`, auch SLIP und PLIP. Durch `hard_header_len` wird die Länge des Protokollkopfes auf der Hardwareebene festgelegt. Unter `priv` kann ein Zeiger auf eine speziell an den Gerätetyp angepasste Struktur abgelegt werden.

Flag	Erläuterung
IFF_UP	Das Netzwerkgerät kann Pakete empfangen und senden.
IFF_BROADCAST	Die Rundrufadresse in `struct device` ist gültig und kann benutzt werden.
IFF_DEBUG	Das Debugging ist eingeschaltet (momentan nicht genutzt).
IFF_LOOPBACK	Dieses Gerät schickt alle übergebenen Pakete an den eigenen Rechner zurück.
IFF_POINTOPOINT	Punkt-zu-Punkt-Verbindung (SLIP, PLIP); `pa_dstaddr` enthält die Protokolladresse des Kommunikationspartners.
IFF_NOTRAILERS	Ist immer ausgeschaltet, diente aber in alten BSD-Systemen zu einer alternativen Anordnung der Header am Ende eines Pakets.
IFF_RUNNING	Die Ressourcen für den Betrieb sind belegt.
IFF_NOARP	ARP wird von diesem Netzwerkgerät nicht benutzt.
IFF_PROMISC	Das Netzwerkgerät empfängt alle Pakete auf dem Netzwerk, selbst die, die an andere Geräte adressiert sind.
IFF_ALLMULTI	Das Netzwerkgerät empfängt alle IP-Multicast-Pakete.
IFF_MASTER	Der Master-Slave-Modus wurde aktiviert, es gibt einen Slave für das Gerät.
IFF_SLAVE	Dieses Gerät wird als Slave für ein anderes Netzwerkgerät benutzt.
IFF_MULTICAST	Die Hardware ist in der Lage, IP-Multicast-Pakete zu empfangen.
IFF_PORTSEL	Die Hardware unterstützt das Setzen des Kartenausgangsports.
IFF_AUTOMEDIA	Die Hardware sucht sich automatisch das angeschlossene Medium aus.
IFF_DYNAMIC	Das Netzwerkgerät kann seine Adresse ändern (für Dialup).

Tabelle 8.1: Flags der Netzwerkgeräte

```
unsigned char           broadcast[MAX_ADDR_LEN],pad;
unsigned char           dev_addr[MAX_ADDR_LEN];
unsigned char           addr_len;
```

`dev_addr[]` enthält die Hardwareadresse des Geräts. Auch in `broadcast[]` befindet sich eine Adresse, die man als Rundrufadresse bezeichnen könnte. Pakete mit dieser Zieladresse werden von allen Rechnern im angeschlossenen Netz empfangen. Die Adressen sind, da als Bytefelder implementiert, typunabhängig. `addr_len` gibt die Länge der Adressen an und ist natürlich durch `MAX_ADDR_LEN` begrenzt. Die Werte dieser Felder werden bei der Geräteinitialisierung belegt und können nicht geändert werden.

```
struct dev_mc_list      *mc_list;
int                      mc_count;
int                      promiscuity;
int                      allmulti;
```

Diese Elemente wurden in den Versionen 1.2 und 2.0 von Linux hinzugefügt. `mc_list` und `mc_count` dienen zur Realisierung des Multicastings auf Hardwareebene. Viele Ethernetkarten unterstützen Multicasting bereits durch ihre Hardware. Die Anzahl der Einträge steht in `mc_count`. Jedes Element der Liste beschreibt genau eine Multicast-Adresse. `promiscuity` und `allmulti` sind zählen, wie häufig das Netzwerkgerät in den jeweiligen Modus gesetzt wurde.

```
void                    *atalk_ptr,*ip_ptr,*dn_ptr,…;
```

Gegenüber den vorherigen Versionen von Linux wurden die protokollspezifischen Daten nun in eigene Unterstrukturen verlagert. Einzig verwunderlich bleibt hier die Benutzung von typlosen Zeigern.

```
struct Qdisc            *qdisc;
struct Qdisc            *qdisc_sleeping;
struct Qdisc            *qdisc_list;
struct Qdisc            *qdisc_ingress;
unsigned long            tx_queue_len;
```

Diese Felder, die zur Implementation der Paketwarteschlangen gehören, sind in der Version 2.2 von Linux dazugekommen. Für einfache Ethernetgeräte verweisen sie initial auf `noop_qdisc` und werden bei der Aktivierung des Gerätes auf die Standardimplementation gesetzt. Diese ist eine einfache FIFO-Realisierung. `tx_queue_len` gibt die maximale Länge der Warteschlangen an, für Ethernetgeräte beträgt sie z. B. 100.

```
spinlock_t               xmit_lock;
int                      xmit_lock_owner;
spinlock_t               queue_lock;
atomic_t                 refcnt;
int                      deadbeaf;
int                      features;
```

Der `xmit_lock` wird zur Synchronisation für die `xmit`-Funktionen benutzt; er ersetzt den `tbusy`-Mechanismus. In `xmit_lock_owner` wird die CPU vermerkt, die den Lock hält.

```
int       (*open)(struct net_device *dev);
int       (*stop)(struct net_device *dev);
```

Eigentlich müsste die Operation `stop()` den Namen `close()` oder die Operation `open()` den Namen `start()` haben. Dadurch würde ihr Zusammenspiel genauer beschrieben werden. Nach dem Aufruf von `open()` können Pakete über das Netzwerkgerät ausgegeben werden. Die Funktion initialisiert jedoch nicht die Adressen. `stop()` beendet die Übertragung von Paketen und setzt die Adressen auf `NULL`.

```
   int  (*hard_start_xmit) (struct sk_buff *skb,
                            struct net_device *dev);
   int  (*hard_header)     (struct sk_buff *skb,
                            struct net_device *dev,
                            unsigned short type,
                            void *daddr, void *saddr,
                            unsigned len);
   int  (*rebuild_header)  (struct sk_buff *skb);
```

`hard_start_xmit()` ist eine hardwareabhängige Funktion. Sie wird beim Erkennen einer bestimmten Karte entsprechend gesetzt. Ihre Aufgabe ist es, das Paket im übergebenen Puffer zu senden. Die globale Funktion `dev_queue_xmit()` kann man als die gepufferte Variante von `hard_start_xmit()` bezeichnen. Wenn das Gerät nicht beschäftigt ist, versucht die Funktion `dev_queue_xmit()` mittels `hard_start_xmit()` das Paket sofort zu übertragen. Andernfalls wird das Paket entsprechend der Priorität in eine der Listen für noch zu sendende Pakete eingereiht. `hard_header()` schreibt den Hardware-Protokollkopf in den übergebenen Puffer. `rebuild_header()` bringt diesen auf den aktuellen Stand, entsprechend der Daten in der Struktur `net_device`, auf die `dev` aus dem Puffer zeigt.

```
#define HAVE_MULTICAST
   void (*set_multicast_list)(struct net_device *dev);
```

`set_multicast_list()` ist eine Funktion, die die neueren Entwicklungen im Internet unterstützt. Sie erlaubt es einem Netzwerkgerät, Pakete zu empfangen, die nicht an das Gerät direkt oder an die Rundrufadresse gerichtet sind. Die Implementierung für die Ethernetkarten erfolgt häufig mit Hilfe des Promiscuous-Modus der Karten, bei dem sie alle Pakete empfangen, die in das Netz gesendet werden.

```
#define HAVE_SET_MAC_ADDR
   int  (*set_mac_address) (struct net_device *dev,
                            void *addr);
#define HAVE_PRIVATE_IOCTL
   int  (*do_ioctl)        (struct net_device *dev,
                            void *addr);
#define HAVE_SET_CONFIG
   int  (*set_config)      (struct net_device *dev,
                            struct ifmap *map);
```

Die Funktion `set_mac_address()` ermöglicht es, die Netzwerkadresse der Hardware zu setzen. Mit Hilfe der `do_ioctl`-Funktion können Netzwerkgeräte spezielle Einstellungen von außen ermöglichen. Bei PLIP kann zum Beispiel der Wert für das Timeout gesetzt bzw. gelesen werden. Allgemeinere Einstellungen der Hardware sind mit der Funktion `set_config()` möglich. Mit dieser Funktion kann zum Beispiel die Nummer des Interrupts eingestellt werden.

```
#define HAVE_HEADER_CACHE
   int  (*hard_header_cache)(struct neighbour *neigh,
                             struct hh_cache *hh);
   void (*header_cache_update)(struct hh_cache *hh,
```

```
                              struct net_device *dev,
                              unsigned char *  haddr);
#define HAVE_CHANGE_MTU
   void (*change_mtu)(struct net_device *dev, int new_mtu);
```

Die beiden `cache`-Funktionen sind für die Implementierung des Routencache notwendig. Dieser wurde in der Version 2.0 von Linux hinzugefügt, um einen besseren Netzwerkdurchsatz zu erzielen. Wenn sich durch ein Nutzerprogramm die MTU eines Netzwerkgerätes ändert, wird `change_mtu` gerufen.

```
   int                        watchdog_timeo;
   struct timer_list          watchdog_timer;
   int  (*tx_timeout)(struct net_device *dev);
```

Mit Hilfe dieser drei Elemente wird ein Übertragungstimer für Netzwerkgeräte realisiert. Dabei gibt `watchdog_timeo` die Zeitspanne an, nach der die Funktion in `tx_timeout` aufgerufen werden soll.

```
   int  (*hard_header_parse)(struct sk_buff *skb,
                              unsigned char *haddr);
   int  (*neigh_setup)(struct net_device *dev,
                       struct neigh_parms *);
   int  (*accept_fastpath)(struct net_device *,
                           struct dst_entry*);
};
```

Die `hard_header_parse`-Funktion wurde eingeführt, um es dem Netzwerkgerät zu ermöglichen, die Details der Low-Level-Pakete zu verbergen.

```
   struct module              *owner;
```

Durch die Einführung eines Module-Zeigers in diese Struktur wurde die Verwaltung von Modulen vereinfacht, die Netzwerkgeräte enthalten. Nun muss man nicht optional den Referenzzähler des Moduls erhöhen, sondern benutzt das Makro `SET_MODULE_OWNER(X)` mit einem Zeiger auf die eigene `net_device`-Struktur. Dadurch wird der Referenzzähler automatisch behandelt. Der Trick dabei ist die leere Definition des Makros, falls es sich nicht um eine Modulübersetzung handelt.

```
struct in_device
{
        struct net_device            *dev;
        struct in_ifaddr             *ifa_list;
        struct ip_mc_list            *mc_list;
        unsigned long                 mr_v1_seen;
        unsigned                      flags;
        struct neigh_parms           *arp_parms;
        struct ipv4_devconf           cnf;
};
```

8.3.1 Ethernet

Linux unterstützt für Ethernet zwei Gruppen von Adaptern. Das sind einerseits die klassischen Ethernetkarten, die an den PC-Bus angeschlossen sind, und andererseits Adapter, die über die parallele Schnittstelle oder den PCMCIA-Bus mit dem PC verbunden sind.

Die Netzwerkgeräte für Ethernetkarten heißen „`eth0`“, …, „`eth3`“. Das gilt auch für die über den PCMCIA-Bus betriebenen Pocket-Adapter, die als Modul eingebunden werden. Linux ordnet die Karten den Geräten in der Reihenfolge zu, in der die Hardware erkannt wurde. Der Kernel gibt beim Start eine Meldung über die erkannten Karten und ihre Zuordnung zu den Netzwerkgeräten aus. Bei den Modulen erfolgt die Ausgabe natürlich erst zum Zeitpunkt des Ladens. Welche Karten bzw. Adapter Linux schon unterstützt, kann dem „Ethernet-HOWTO“[4] entnommen werden. Da WD8013- und NE2000-kompatible Karten unterstützt werden, ist eine große Anzahl von preiswerten Ethernetadaptern verfügbar.

Befassen wir uns weiter mit den Ethernetnetzwerkgeräten. Die Ethernetadresse der zugehörigen Netzwerkkarte ist im Feld `dev_addr[]` gespeichert. Jeder Ethernetadapter hat eine weltweit eindeutige Adresse. Diese Adressen sind sechs Byte lang. Ihre textuelle Darstellung ist zum Beispiel `0:0:c0:9b:13:29`. Nach der Konfiguration des Netzwerkgeräts mit einer IP-Adresse wird beim Zuschalten der Karte ein Eintrag in der ARP-Tabelle erzeugt (vgl. `ifconfig` in Anhang B.7).

Durch ein Feld im Hardware-Header eines Ethernetpaketes werden verschiedene Typen von Ethernetpaketen unterschieden. So gibt es Typen für IP, ARP, IPX und andere Protokolle. Durch den Typ wird festgelegt, welcher Empfangsfunktion das Paket zu übergeben ist.

Die Zuordnung der Pakettypen erfolgt mit Hilfe einer Liste. So besteht die Möglichkeit, die bekannten Pakettypen dynamisch zu ändern. Für das IP gibt es zum Beispiel ein Listenelement, das wie folgt aussieht:

```
static struct packet_type ip_packet_type = {
  htons(ETH_P_IP),
  NULL,
  ip_rcv,
  (void*)1,
  NULL
};
```

Dieser Eintrag enthält sowohl den Ethernetpakettyp als auch die zugehörige Empfangsfunktion. Die erste Null besagt, dass Pakete dieses Typs nur von diesem Netzwerkgerät empfangen werden sollen. Dort, wo der `(void*)1`-Zeiger steht, kann ein Zeiger auf spezielle Daten angegeben werden, der nächste Zieger dient zur Verkettung in der Liste aller Pakettypen.

4 Das „Ethernet-HOWTO“ befindet sich in der Datei `docs/HOWTO/Ethernet-HOWTO` auf der dem Buch beiliegenden CD.

Diese Liste ist also die Schnittstelle zwischen den Netzwerkgeräten und den einzelnen Protokollen aus der Sicht der Geräte. Pakete, die keinem der in der Liste registrierten Typen entsprechen, werden verworfen.

8.3.2 SLIP und PLIP

Kommen wir zu etwas „exotisch“ anmutenden Geräten. Der einzige wesentliche Unterschied zwischen SLIP und PLIP ist, dass das eine Protokoll die serielle und das andere die parallele Schnittstelle des Rechners zur Datenübertragung nutzt. Mit der parallelen Schnittstelle sind hier keine Ethernetpocketadapter gemeint, sondern vielmehr die „nackte“ Schnittstelle.

Mit PLIP kann eine recht leistungsfähige Verbindung zwischen zwei Rechnern aufgebaut werden. SLIP ist die einfachste Möglichkeit, einen Rechner oder ein lokales Netz über eine serielle Leitung (Modemverbindung im Telefonnetz) an das Internet anzuschließen. SLIP und PLIP unterscheiden sich vom Ethernet dahin gehend, dass sie nur IP-Pakete übertragen können. SLIP benutzt der Einfachheit halber nicht einmal einen Hardware-Header. Auch PLIP betreibt dabei keinen großen Aufwand. Es setzt die Hardwareadresse einfach auf „`fd:fd`“ + IP-Adresse und benutzt dann die Ethernetfunktionen für den Protokollkopf (siehe Abbildung 8.7).

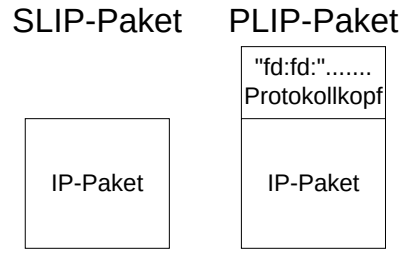


Abbildung 8.7: Verhältnis von SLIP- bzw. PLIP-Paketen zu IP-Paketen

8.3.3 Das Loopback-Gerät

Das Loopback-Gerät dient zur Kommunikation von Anwendungen auf dem lokalen Rechner, die Sockets der INET-Adressfamilie verwenden. Es lässt sich mit minimalem Aufwand implementieren, da es die zu sendenden Pakete sofort an die oberen Schichten zurückgibt. Mit ihm ist auch der Test von Netzwerkanwendungen auf einem Rechner möglich. Fehler der Netzwerkhardware sind dabei ausgeschlossen. Dem Loopback-Gerät „`lo`“ wird meist die IP-Adresse `127.0.0.1` zugeordnet.

8.3.4 Das Dummy-Gerät

Mit dem Dummy-Gerät treffen wir auf einen etwas exotischen Vertreter der Netzwerkgeräte. Es verhält sich eigentlich wie jedes andere Gerät, aber es erfolgt keine echte Datenübertragung.

Jetzt stellt sich die Frage, wozu dieses Netzwerkgerät denn überhaupt gut ist. Es wird normalerweise dazu benutzt, den weiter oben liegenden Teilen der Netzwerkimplementation ein funktionierendes Netzwerkgerät zu präsentieren, auch wenn ein solches nicht vorhanden ist. Mit den höheren Teilen der Netzwerkimplementation sind hier auch Nutzerprozesse gemeint.

8.3.5 Ein Beispielgerät

In diesem Abschnitt soll anhand eines Ethernetgerätes gezeigt werden, wie man eigene Netzwerkgeräte implementiert. Dabei wird von der wirklichen Hardware abstrahiert und nur auf die wirklich notwendigen Teile eingegangen. An den Stellen, wo eine Hardwarebehandlung notwendig wäre, wird dann darauf hingewiesen.

Die Funktionsweise eines Netzwerkgerätes lässt sich wie die meisten Teile der Netzwerkimplementation am besten mit einem *stream*-Modell beschreiben. Höhere Protokollschichten schicken Daten an das Gerät, das diese an die Hardware weiterleiten soll. Außerdem empfängt die Hardware Daten, die an die höheren Schichten weitergeleitet werden müssen. Die Funktionen in der `net_device`-Struktur stellen die Sichtweise der höheren Schichten auf das Netzwerkgerät dar.

Für die Ethernetgeräte, die nicht als Module geladen werden, gibt es eine zentrale Setup-Funktion (`eth_setup()`). Mit ihrer Hilfe ist es möglich, den Interrupt, den Anfang der I/O-Adressen sowie den Speicheranfang und das Speicherende für ein bestimmtes `ethX` zu setzen. Die Werte werden für den späteren Zugriff in die `net_device`-Struktur geschrieben. Bei der Implementierung als Module können die Parameter natürlich frei gewählt werden; man unterliegt dann jedoch den allgemeinen Beschränkungen eines Moduls (s. Kapitel 9).

Bevor ein Gerätetreiber seine Funktion aufnimmt, sollte immer überprüft werden, ob die entsprechende Hardware vorhanden und funktionstüchtig ist. Bei den Netzwerkgerätetreibern hat es sich eingebürgert, eine eigenständige Funktion (`probe`) für diese Aufgabe zu benutzen. Diese kann dann entweder fest bei der Initialisierung der Kerns eingebunden oder in der Initialisierungsfunktion des Moduls aufgerufen werden. Dazu hat man in Linux 2.4 das Makro `module_init()` eingeführt. Dieses bekommt als Parameter eine *Init*-Funktion, die jedoch nicht mehr `init_modul` heißen muss. Wenn nun der Treiber direkt im Kernel eingebunden ist, wird diese Funktion während des Startens aufgerufen. Falls es sich aber um ein Modul handelt, wird diese Funktion als `module_init` deklariert (siehe auch Kapitel 9).

Down stream

Die Funktionen, die für den Transport der Daten zum Netzwerk benötigt werden, sind in der `net_device`-Struktur enthalten und werden nachfolgend im Einzelnen behandelt. Die Aufgabe der `probe`-Funktion ist es nicht nur, die Hardware in einen funktionstüchtigen Zustand zu bringen, sondern auch, die Funktionszeiger in der `net_device`-Struktur zu belegen. Sie sollte diese komplett initialisiert zurückgeben.

`open(dev)` Vor der Benutzung eines Netzwerkgerätes wird die `open`-Funktion aufgerufen, meist durch `dev_open()`. Von `open` sollten die Hardware und der Treiber in einen betriebsbereiten Zustand gebracht werden, das heißt, unter anderem sollten die für den Betrieb notwendigen Ressourcen (Interrupts[5], …) angefordert und belegt werden. Außerdem ist es eine gute Strategie, erst hier die Interrupterzeugung auf der Hardware einzuschalten und nicht gleich bei der Hardware-Erkennung. Unter Umständen können hier auch noch die Funktionszeiger in der `net_device`-Struktur modifiziert werden. Der Ergebnistyp der Funktion ist ein Integerwert. Eine Null bedeutet dabei die fehlerfreie Ausführung. Ansonsten werden negative Fehlernummern wie `-EBUSY` zurückgegeben.

`stop(dev)` In dieser Funktion sollten die inversen Aktionen zu `open` durchgeführt werden. Die Ressourcen werden freigegeben, die Interrupterzeugung wird ausgeschaltet. Nach dem Aufruf von `stop` sollten vom Netzwerkgerät keine weiteren Daten mehr an die höheren Protokollschichten weitergeleitet werden. Der Ergebnistyp der Funktion ist ein Integerwert. Eine Null bedeutet dabei die fehlerfreie Ausführung. Ansonsten werden negative Fehlernummern wie `-EBUSY` zurückgegeben. Beim Schließen sollten aber normalerweise keine Fehler auftreten.

`hard_start_xmit(skb, dev)` Die Funktion `hard_start_xmit()` ist das Herzstück des Netzwerkgerätes. Sie wird für jedes zu übertragende Paket aufgerufen, das sich in `skb` befindet. Änderungen am Paket sind nicht mehr vorzunehmen. Die Paketdaten sollen einfach an die Hardware bzw. das Netz übergeben werden. Welcher Mechanismus dazu benutzt wird, bleibt dem Implementator überlassen. Gängige Möglichkeiten sind DMA-Transfer, Shared Memory der Karte oder I/O-Ports. Weiterhin wird im `trans_start` der aktuelle Wert aus `jiffies` eingetragen, der den Anfang der Übertragung angibt und mit dessen Hilfe man später feststellen kann, ob es zu einem Timeout bei der Übertragung gekommen ist. Für weitere Variablen benutzt man die private Struktur des Netzwerkgerätetreibers, die in `priv` vermerkt ist und für jeden Treiber selbst definiert wird. Nach der erfolgreichen Übertragung eines Paketes sollte mit

```
netif_wake_queue(dev);
```

das Senden zwischengespeicherter Pakete angestoßen werden. Bei einer DMA-Übertragung sollte das dann beim abschließenden Interrupt vom DMA erfolgen.

`set_multicast_list(dev)` Die Aufgabe dieser Funktion ist es, einen Abgleich der `mc_list` der Treiberstruktur mit der Hardware vorzunehmen. Für Karten, die keine Multicastimplementierung haben, sollte hier der Promiscuous-Modus ein- bzw. ausgeschaltet werden. Bei unterstütztem Multicast müssen die Adressen aus der Liste mit den Adressen auf der Hardware abgeglichen werden. Unter Umständen muss dann der Multicast-Empfang ein- bzw. ausgeschaltet werden. An dieser Stelle ist genau wie beim Öffnen des Netzwerkgerätes auch das `IFF_ALLMULTI`- und `IFF_PROMISC`-Flag der Treiberstruktur zu berücksichtigen und die Karte in den entsprechenden Empfangsmodus zu setzen.

5 Bei der Registierung der ISR geben wir am besten den Zeiger auf unsere `net_device`-Struktur als Datenzeiger an.

`get_stats(dev)` Diese Funktion wird insbesondere vom *Proc*-Dateisystem benutzt, um die Statistik für das Netzwerkgerät auszugeben. Auf dieses Dateisystem wird für die Implementierung der MIB (SNMP) für Linuxrechner zugegriffen.

Da man, um die Werte für die `net_device_stats`-Struktur zu berechnen, während des Betriebs bereits Daten sammeln muss, bietet es sich an, die Struktur als Feld der privaten Daten des Netzwerkgerätetreibers zu definieren. An den geeigneten Stellen der anderen Funktionen müssen die Werte in der Statistikstruktur weitergezählt werden.

`do_ioctl(dev, addr)` Diese Funktion implementiert die `ioctl`-Rufe an den Treiber. Sie wird für Aufgaben benutzt, die nicht im normalen Funktionsumfang des Kerns vorgesehen sind, wie z. B. das Setzen der MTU auf nicht standardkonforme Werte, das Einschalten von Fehlermeldungen usw.

Up Stream

Für den Datenverkehr vom Netzwerk zu den oberen Protokollschichten gibt keine große Schnittstelle. Die einzige Funktion, die dort benutzt wird, ist `netif_rx()`. Sie bekommt als Parameter eine initialisierte `sk_buff`-Struktur, die das empfangene Paket enthält.

Da es ja die Aufgabe des Netzwerkgerätetreibers ist, die Unterschiede der Hardware für den Rest des Kerns zu verbergen, gibt es natürlich keine einheitliche Schnittstelle zur Hardware. Hier gelten die normalen Regeln für Gerätetreiber (siehe Kapitel 7).

Die Anbindung der Hardware mit Hilfe von Interrupts ist jedoch der Normalfall. Der ISR wird das Feld `interrupt` der `net_device`-Struktur bereitgestellt. Es sollte auf einen Wert ungleich Null gesetzt werden, wenn die ISR aktiv ist, um den mehrfachen Aufruf der ISR zu verhindern. Dieses sollte mit der `test_and_set_bit`-Funktion erfolgen, da es sich dabei um einen Mutex handelt, der für SMP funktioniert. Der weitere Funktionsverlauf hängt nun wiederum stark von der Hardware ab. Unter Umständen sind Teile für die DMA-Behandlung hier implementiert. Wenn die ISR jedoch festgestellt hat, dass ein Paket von der Hardware empfangen worden ist, muss in etwa der folgende Ablauf implementiert werden: Eine `sk_buff`-Struktur wird mit der Funktion `dev_alloc_skb()` angefordert. Die dort anzugebende Größe entspricht der Paketgröße. Der Zeiger auf unsere `net_device`-Struktur muss in das Feld `dev` des Puffers eingetragen werden. Anschließend müssen die Paketdaten in den Puffer kopiert werden, was wieder in hohem Maße von der Hardware (DMA, I/O, …) abhängig ist. Jetzt wird das Feld `protocol` des Puffers gesetzt - für Ethernetgeräte auf den Rückgabewert von `eth_type_trans()` mit dem Puffer und unserem `net_device`-Zeiger als Parameter. Nun ist der Puffer fertig, und wir rufen die Funktion `netif_rx` mit dem Puffer als Parameter auf. Damit ist der Datenverkehr geregelt, und es folgen noch einige statistische Aufgaben. `last_rx` aus der `net_device`-Struktur wird auf den aktuellen Wert von `jiffies` gesetzt. Außerdem sind einige Felder der `net_device_stats`-Struktur auf den aktuellen Stand zu bringen. Diese Felder sind natürlich auch im Fehlerfall auf neue Werte zu setzen.

9 Module und Debugging

»Viel Glück, Jungs«, zwitscherte der Computer,
»Einschlag in dreißig Sekunden …«
Douglas Adams

Von Version zu Version nimmt der Linux-Kern an Umfang zu. Dies geschieht sowohl durch die kontinuierliche Verbesserung und Erweiterung der Kernfunktionalität als auch durch das Hinzufügen neuer Gerätetreiber, Dateisysteme oder Emulationen wie das iBCS2. Da Linux ein monolithisches System ist, sind jedoch alle verwendeten Gerätetreiber bzw. Dateisysteme fest in den Kern eingebunden. Das bedeutet einerseits, dass bei einer Änderung der Konfiguration der Kern neu übersetzt werden muss, und andererseits, dass auch selten benötigte Treiber oder Dateisysteme permanenten Speicher belegen. Ein weiterer Nachteil offenbart sich Entwicklern von neuem Kernelcode: Nach jeder noch so kleinen Änderung muss der neue Kern erstellt, der Kern neu installiert und der Rechner neu gebootet werden. Dieser und viele andere Gründe führten zur Entwicklung von Modulen. Dabei stellt sich erst einmal die Frage: Was sind Module?

9.1 Was sind Module?

Aus der Sicht des Kerns sind Module zur Laufzeit link- und entfernbarer Objektcode, normalerweise eine Menge von (mindestens zwei) Funktionen. Dieser Objektcode wird gleichberechtigt in den bereits laufenden Kern integriert, läuft also im Systemmodus. Die monolithische Struktur des Kerns wird nicht verändert; im Gegensatz zum Microkernel laufen die neu angefügten Funktionen nicht als eigener Prozess. Die Implementierung von Gerätetreibern bzw. Dateisystemen als Module hat zudem den Vorteil, dass nur die dokumentierten bzw. exportierten Schnittstellen verwendet werden können.

Aus Sicht der Benutzer erlauben es Module, einen kleinen und kompakten Kern zu verwenden und nur bei Bedarf die benötigte Funktionalität einzufügen. Die Version 2.4 des Kernels enthält bereits einen automatischen Mechanismus für das Laden von Modulen, so dass sich der Benutzer nicht mehr darum kümmern muss. Als weiteres Beispiel für das automatische Laden von Modulen sei auch der *PCMCIA*-Kartenmanager genannt.

In Abschnitt 2.1 wird der Aufbau der Quelltexte des Linux-Kerns beschrieben. Die C-Dateien werden in verschiedenen funktional zusammengehörenden Verzeichnissen organisiert. Bei der Übersetzung werden die funktionalen Untereinheiten zu einer Objektdatei zusammengefasst, so dass beim späteren Zusammenlinken des Kerns nicht auf jede Objektdatei einzeln zugegriffen werden muss. Diese funktionalen Einheiten können auch oft als Modul verwendet werden.

9.2 Implementierung im Kernel

Nachdem die Vorteile der Benutzung von Modulen aufgezeigt worden sind, werden wir uns nun mit deren Implementierung beschäftigen. Für die Benutzung stellt Linux drei Systemrufe zur Verfügung; *create_module*, *init_module* und *delete_module*. Ein weiterer Systemruf wird vom Benutzerprozess angewendet, um eine Kopie der Symboltabelle des Kerns zu erhalten.

Zur Verwaltung der Module unter Linux wird eine Liste benutzt, in der alle geladenen Module verzeichnet sind. Die Form der Einträge ist auf Seite 315 dargestellt. Diese Liste verwaltet auch die Symboltabellen und Referenzen der Module.

Das Laden eines Moduls erfolgt aus der Sicht des Kerns in zwei Schritten: mit den beiden Systemrufen *create_module* und *init_module*. Für den Nutzerprozess gliedert sich dieser Vorgang in vier Phasen:

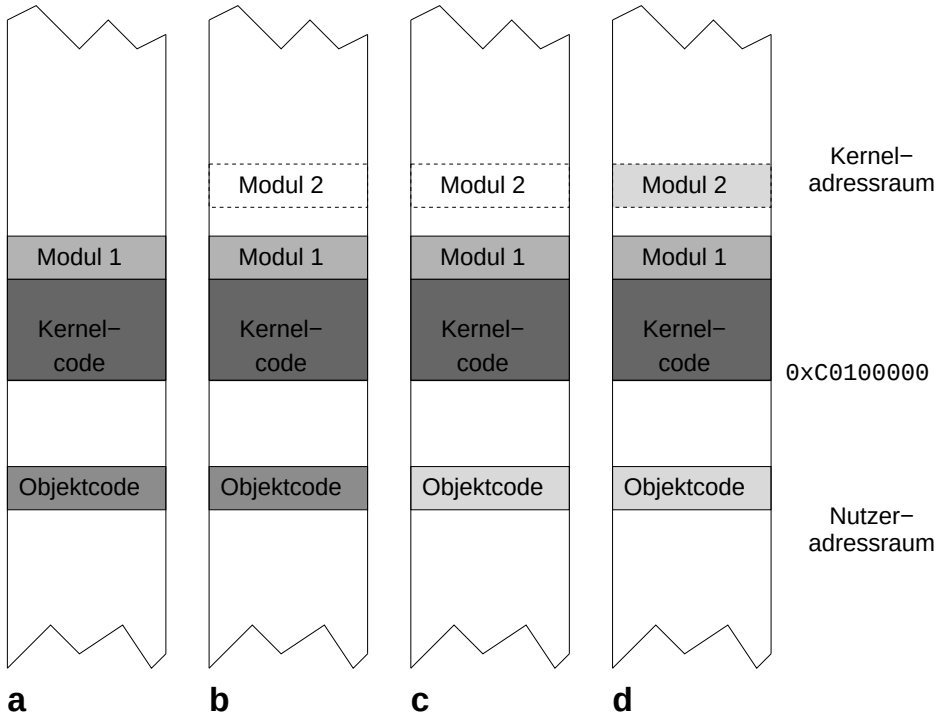


Abbildung 9.1: Der Adressraum beim Laden eines Modules

- Der Prozess holt sich den Inhalt der Objekdatei in seinen Adressraum. In einer normalen Objektdatei werden der Code und die Daten so abgelegt, als ob sich diese beim nachfolgenden Laden an Adresse 0 befinden. Um den Code und die Daten in eine Form zu bringen, in der sie wirklich ausgeführt werden können, muss an bestimmten Stellen die wirkliche Ladeadresse addiert werden. Diesen Vorgang bezeichnen

wir als *Relokieren*. Verweise auf die entsprechenden Stellen sind in der Objektdatei vorhanden. Außerdem können sich auch noch nicht aufgelöste Referenzen in der Objektdatei befinden. Bei der Analyse der Objektdatei ergibt sich auch die Größe des Objektmoduls (siehe Abbildung 9.1).[1]

- Nun wird der Systemruf *create_module* benutzt, um einerseits die letztendliche Adresse des Objektmoduls zu erhalten und andererseits den Speicher und den Namen für das Modul zu reservieren. Dabei wird im Kernel eine Struktur `module` für das Modul in die Liste der Module eingetragen und der Speicher ausgefasst. Als Rückgabewert erhalten wir die Adresse, an die das Modul später kopiert werden soll (siehe Abbildung 9.1 b).

- Die von *create_module* erhaltene Ladeadresse des Moduls wird für die Relokation der Objektdatei benutzt. Dieser Vorgang findet in einem Speicherbereich des Prozesses statt, d. h., das Objektmodul befindet sich zu dieser Zeit noch nicht an der richtigen Adresse, wird aber für die Ladeadresse des Moduls im Kernsegment relokiert.

  Unaufgelöste Referenzen können mit Hilfe der Symbole des Kerns bzw. schon geladener Module behoben werden. Linux stellt dazu den Systemruf *query_module* zur Verfügung. Mit seiner Hilfe es möglich, die Symboltabelle für jedes geladene Modul und den Kern zu bekommen. Sie müssen dabei beachten, dass sich in der Symboltabelle keinerlei Typinformationen befinden, sondern nur Adressen. Sie müssen deshalb bei der Entwicklung von Modulen darauf achten, dass stets die richtigen Headerdateien eingebunden werden.

  Um eine möglichst große Flexibilität zu erreichen, können die Module eigene Symbole in die Symboltabelle des Kerns exportieren. So kann ein Modul Funktionen eines vorher geladenen Moduls benutzen. Dieser Mechanismus wird als *Stacked Module* bezeichnet. Alle von einem Modul exportierten Symbole werden in einer eigenen Symboltabelle zusammengetragen (siehe Abbildung 9.1).

- Nachdem die Vorarbeiten abgeschlossen sind, kommen wir zum Laden des Objektmoduls. Dazu wird der Systemruf *init_module* verwendet. Dieser bekommt als Parameter einen Zeiger auf die Struktur `module` übergeben. Die Verwaltungsfunktionen des Moduls werden in diese Struktur unter `init` und `cleanup` eingetragen. Von Linux wird jetzt das Objektmodul in den Adressraum des Kerns kopiert. Die Verwaltungsfunktion `init()` wird nach der Installation des Codes und der Daten aufgerufen. In ihr sollte auch die entsprechende Registrierfunktion aufgerufen werden.

  Der Rückgabewert von `init()` entscheidet, ob die Installation als geglückt oder gescheitert betrachtet wird. Die zweite Verwaltungsfunktion `cleanup()` wird bei der Deinstallation des Moduls aufgerufen. Sie muss die zugehörige Abmeldefunktion starten.

Die Symboltabelle des Kerns ist hauptsächlich in den Dateien `kernel/ksyms.c` und `arch/<arch>/kernel/<arch>_ksyms.c` definiert. Alle Funktionen, die exportiert werden, besitzen einen Eintrag in der Tabelle `symbol_table`. Die Einträge werden mit

1 Weitere Einzelheiten über den Aufbau von und den Umgang mit Objektdateien finden Sie in [Gir90] und [ELF].

Hilfe des Makros `EXPORT_SYMBOL()` erzeugt, das den Namen der Funktion oder Variablen in die Struktur `module_symbol` überführt.

In der eigenen Symboltabelle befinden sich nicht nur die zu exportierenden Symbole, sondern auch die Referenzen auf Symbole im Kern, die vom Modul benutzt worden sind. Dadurch kann die Abhängigkeit der Module untereinander bestimmt werden. So kann festgelegt werden, dass ein Modul, das noch von anderen Modulen genutzt wird, nicht deinstalliert werden darf.

Ein weiteres Hilfsmittel, um Probleme bei der Deinstallation von benutzten Modulen zu vermeiden, ist der `USE_COUNT`-Mechanismus. Wenn wir zum Beispiel einen Gerätetreiber als Modul implementiert haben und es geladen worden ist, wird der Benutzungszähler bei jedem Öffnen erhöht und beim Schließen heruntergezählt. Somit kann bei der Deinstallation festgestellt werden, ob sich das Modul noch in Benutzung befindet. Beachten Sie jedoch, dass sich die Stellen für die Änderungen des Benutzungszählers teilweise nur schwer oder gar nicht finden lassen.

Als letzte Möglichkeit für besonders schwere Fälle kann man sich natürlich auf den Standpunkt stellen, dass der Benutzungszähler während der `init`-Funktion einmalig hochgezählt wird. Somit kann das Modul nicht entfernt werden.

Die Flexibilität der Module besteht nicht nur darin, dass sie dynamisch geladen werden können. Durch die Benutzung des Systemrufs *delete_module* kann ein geladenes Modul wieder entfernt werden. Dazu sind zwei Vorbedingungen zu erfüllen. Es darf keine Referenzen auf das Modul geben, und der Benutzungszähler des Modulss muss den Wert 0 haben. Vor der Freigabe des Modules wird die bei der Installation registrierte `cleanup`-Funktion aufgerufen. In ihr können dynamische Ressourcen freigegeben werden, die während des Betriebes und der `init`-Funktion angefordert worden sind. Für einen Gerätetreiber, der Hardware bedient, heißt das, die Hardware in einen Zustand zu versetzen, die ohne die Unterstützung des Gerätetreibers auskommt. Insbesondere sollte die Hardware keine Interrupts mehr auslösen, die vom Gerätetreiber behandelt wurden, da dieser nun die Interruptbehandlungsroutine deregistriert.

9.2.1 Signatur von Symbolen

Ein geläufiges Problem bei der Implementierung von Modulen ist die Abhängigkeit der Module von der Kernversion. Aufgrund der rapiden Weiterentwicklung des Linux-Kerns ändern sich auch exportierte Strukturen und Funktionen. Bei jeder neuen Kernversion sollte man deshalb alle Module neu übersetzen, so dass sichergestellt ist, dass Symbole entsprechend ihrer Definition benutzt werden. Einen Ausweg aus diesem Dilemma bilden Symbolnamen, die eine Signatur des zugehörigen C-Objektes (der Funktion oder Strukturelemente) enthalten. Ein ähnlicher Mechanismus wird zum Beispiel in C++ für Funktionen und Namensräume benutzt. Er ist dort eindeutig.

Im Linux-Kern wird ein anderes Modell benutzt. Dabei werden zu exportierende Symbole zu ihrer vollen Definition expandiert, und über diese wird eine 32-Bit-Prüfsumme gebildet. Sie wird schließlich an das ursprüngliche Symbol angefügt. Dieses Verfahren

ist zwar nicht eindeutig, die Wahrscheinlichkeit einer Überschneidung ist jedoch hinreichend gering. Dieser Mechanismus muss jedoch bei der Konfiguration des Kerns eingeschaltet werden. Dazu ist die Frage

```
Set version information on all symbols for modules
```

zu bejahen. Die Erzeugung dieser speziellen Symbolinformationen wird von dem Programm `genksyms` übernommen, das Teil der Modulwerkzeuge ist.

Das dazugehörige Programm `insmod` prüft beim Laden eines mit Signaturinformationen übersetzten Moduls automatisch die Übereinstimmung der Prüfsummen. Dadurch wird vermieden, dass ein Modul geladen wird, das eine Funktion mit falschen Parametern aufruft oder auf eine Struktur zugreift, deren Definition sich geändert hat.

9.3 Bedeutung der Objektsektionen für Module und Kern

Im Folgenden werden die einzelnen Objektsektionen (siehe auch [ELF]) und ihre Funktion aufgeführt.

.text Hier befindet sich normaler ausführbarer Code.

.data Hier befinden sich initialisierte Daten für den Code.

.bss Hier befinden sich uninitialisierte Daten.

.rodata Hier befinden sich initialisierte Daten, auf die nur lesend zugegriffen wird.

.text.lock Da beim Laden der Instruktionspipe meist ein optimistischer Ansatz gewählt wird, wird die Behandlung der negativen Fälle aus dem normalen Instruktionsfluss entfernt. Sie landen in der Sektion `.text.lock`. Das gilt für Semaphore und die „spin locks“ bei SMP.

__ksymtab Der Inhalt dieser Sektion besteht aus einem Feld von Einträgen für die exportierten Symbole des Moduls. Die Einträge haben folgende Struktur:

```
struct module_symbol {
   unsigend long value;
   const char *name;
};
```

Das Element `value` enthält die Adresse des Symbols, und `name` verweist auf den Namen. Die Strings selbst sind in der folgenden Sektion untergebracht.

.kstrtab Hier stehen die Strings für die exportierten Symbole. Diese entsprechen dem Namen des exportierten Symbols oder, falls die Signatur aktiviert ist, dem signierten Namen des Symbols.

.fixup Diese Sektion enthält die Behandlungsroutinen für unzulässige Zugriffe aus dem Kernmodus auf den User Space. Sie werden implizit bei der Verwendung der `copy_...` -Makros aus `<asm/uaccess.h>` erzeugt.

__ex_table Um das meist unnötige Überprüfen[2] des richtigen Zugriffs im Kernmodus auf den User Space zu vermeiden, wurde ein optimistischer Ansatz gewählt. Jede Stelle im Code, die potenziell einen falschen Zugriff erzeugen kann, wird in die Exception-Tabelle eingetragen. Weiterhin wird eine Fehlerbehandlungsroutine speziell für diese Code-Stelle dort eingetragen (implizit durch die `copy_...`-Makros). Bei fehlerhaften Zugriffen gibt es einen Speicherzugriffsfehler; die allgemeine Behandlungsroutine durchsucht die Exception-Tabelle und springt in die spezielle Fehlerbehandlung, die in der Tabelle vermerkt wurde.

.modinfo Hier haben wir ein wirklich innovatives Konzept für die Module unter Linux. Allgemeine Informationen über das Modul werden in Form von Zeichenketten in dieser Sektion untergebracht. Dabei haben die Zeichenketten das Format `<name>=<value>` und werden mit einem 0-Byte terminiert. Es gibt die fest vordefinierten Namen `author` (Autor des Moduls), `description` (Beschreibung) und `device` (implementiertes Gerät). Weitere Einträge beschäftigen sich mit den erlaubten Übergabeparametern, die dem Modul beim Laden mitgegeben werden können. Sie haben dann das folgende Format: `parm_<name>=<type>` und `parm_desc_<name>=<desc>`. Der Name bezieht sich dabei auf den unterstützten Modulparameter. Mit Hilfe des ersten Formats wird der Parametertyp festgelegt. Das allgemeine Format ist dabei `[<min>[-<max>]]{b,h,i,l,s}[p]`. Die minimalen und maximalen Werte sind optional und nur für Felder notwendig. Der eigentliche Typ wird durch einen Buchstaben repräsentiert (siehe auch Tabelle 9.1). Das optionale p am Ende kennzeichnet persistente Modulparameter. Diese werden beim Entfernen des Moduls durch `rmmod` gespeichert und dann beim Laden durch `insmod` wieder gesetzt. Mit Hilfe des zweiten Formats kann eine textuelle Beschreibung des Modulparameters eingefügt werden.

b	byte
h	short
i	int
l	long
s	string

Tabelle 9.1: Typcodes für Modulparameter

.text.init Diese Sektion enthält Code, der nur während der Initialisierung benötigt wird. Beim normalen Betrieb des Moduls wird dieser Code nicht mehr gebraucht und kann deshalb entfernt werden. Aus diesem Grund gibt es diese zusätzliche Code-Sektion, die nach der Initialisierung wieder entfernt werden kann[3].

.data.init Analog zur vorherigen Sektion liegen hier Daten, die nur während der Initialisierung benötigt werden.

2 Messungen ergaben ein 99:1-Verhältnis zwischen korrekten und unkorrekten Zugriffen.

3 In der Version 2.4 von Linux wird das Freigeben des initialen Codes zwar nur für den Kern selbst unterstützt, ist jedoch auch für Module geplant.

.setup.init Ein Feld aus `kernel_param`-Strukturen befindet sich in dieser Sektion des Linux-Kernels. Diese werden durch die Benutzung des Makros `__setup()` erzeugt. Als Parameter erhält das Makro einem String, wie z. B. `"root="` und ein Funktion für die Auswertungen der Kernelparamter.

.initcall.init Auch diese Sektion ist ein Feld und enthält Zeiger auf Initialisierungsfunktionen, die beim Start des Linux Kernels aufgerufen werden sollen. Sie werden mit Hilfe des Makros `__initcall()` angelegt. Das Makro bekommt die zu rufende Funktion als Parameter (siehe auch Tabelle 9.2) übergeben. Falls eine Datei nicht als Modul übersetzt wird, wird aus dem Makro `module_init()` ein `__initcall()`.

.data.init_task Hier befindet sich die Union `init_task_union` für die i386-Architektur. Eine eigene Sektion wurde gewählt, um das Alignment auf 8 Kbyte zu sichern.

.data.cacheline_aligned In diese Sektion werden Daten gepackt, die ein passendes Alignment entsprechend dem CPU-Cache haben müssen. Das ist besonders für SMP-Rechner wichtig.

.data.page_aligned entspricht `.data.cacheline_aligned`, nur diesmal ein Alignment auf Speicherseitengrenzen.

Die Sektionen **.comment** und **.note** sind für die Funktion von Modulen und des Kernels nicht entscheidend.

9.4 Parameterübergabe und Beispiel

Um es für den Modulprogrammierer einfach zu gestalten, werden in den Include-Dateien von Linux C-Makros angeboten, die die Parameterübergabe unterstützen und die entsprechenden Informationen für die `.modinfo`-Sektion generieren. Wie diese benutzt werden, ist aus dem folgenden Beispiel ersichtlich. Zusätzlich wird das Exportieren von Symbolen demonstriert:

```
#include <linux/module.h>

int register_hbsubmod(void *x)
{
  return 0;
}

/*
 * zunächst die Variablen für die Parameter anlegen, sie
 * können bereits vorinitialisiert werden
 */

static int hbdebug = 0;
static int io_addr[4] = { 0x300, 0, 0, 0 };
static char *init_string = NULL;

/*
```

```
 * Falls keinerlei Funktionen dieses Moduls von anderen Modulen
 * benötigt werden, bitte auskommentieren.
 */

/* EXPORT_NO_SYMBOLS;*/

/*
 * Dieses Modul exportiert jedoch die
 * Funktion register_hbsubmod().
 */

EXPORT_SYMBOL(register_hbsubmod);

MODULE_AUTHOR("Harald Boehme");
MODULE_DESCRIPTION("Demonstration module of the usage of \
                    the MODULE-macros");
MODULE_SUPPORTED_DEVICE("hbdev");

MODULE_PARM(hbdebug, "i");
MODULE_PARM_DESC(hbdebug, "Debug level");

MODULE_PARM(io_addr, "1-4i");
MODULE_PARM_DESC(io_addr, "Up to four I/O-addresses");

MODULE_PARM(init_string, "s");
MODULE_PARM_DESC(init_string, "optional init-string");


int hb_init(void)
{
   /* Initialisierung, register-Funktionen aufrufen. */
}

void hb_cleanup(void)
{
   /* Ressourcen freigeben, unregister-Funktionen
      aufrufen. */
}

module_init(hb_init)
module_exit(hb_cleanup)
```

Der Aufruf `insmod module io_addr=0x300,0x308 init_string="foo"` initialisiert das Feld `io_addr` mit den Werten 0x300 und 0x308 sowie den Zeiger `init_string` mit dem übergebenen String.

9.5 Was kann als Modul implementiert werden?

Eine Faustregel für die Implementation eines Moduls sollte sein, möglichst wenige Symbole bzw. Funktionen aus dem Kern oder anderen Modulen zu benutzen. Außerdem sollte

Makro	Funktion des Makros
`MODULE_AUTHOR(name)`	Autor des Moduls
`MODULE_DESCRIPTION(desc)`	kurze Beschreibung des Moduls
`MODULE_SUPPORTED_DEVICE(dev)`	Device, das von dem Modul implementiert wird
`MODULE_PARM(var,type)`	ein Modulparameter
`MODULE_PARM_DESC(var,desc)`	kurze Beschreibung für den Modulparameter
`EXPORT_SYMBOL(var)`	Exportieren der Variablen oder Funktion
`module_init(func)`	definiert die Funktion `func` für dieses Module als `init`-Funktion.
`module_exit(func)`	definiert die Funktion `func` für dieses Module als `cleanup`-Funktion.

Tabelle 9.2: Makros für Module

die Möglichkeit bestehen, das Modul dynamisch an- und abzumelden. Als grobe Richtlinie kann gelten, dass es für die vom Modul realisierte Funktionalität eine Registrier- und Abmeldefunktion gibt. Diese Voraussetzung ist bei einer Anzahl von Kernelelementen gegeben. Das bekannteste Beispiel dafür sind Dateisystemimplementationen, für die es die Funktionen `register_filesystem()` und `unregister_filesystem()` gibt. Sie erfüllen alle Voraussetzungen, wozu auch geeignete Punkte für die Verwaltung des Benutzungszählers gehören. Wie schon in Abschnitt 9.2 beschrieben, muss verhindert werden, dass ein Modul entfernt wird, während es noch in Benutzung ist. Bei Dateisystemimplementationen ist das relativ einfach. Eine Dateisystemimplementierung wird benutzt, wenn ein Dateisystem dieses Typs gemountet ist. Während des Mount-Vorgangs wird die `read_super`-Funktion der Dateisystemimplementierung aufgerufen. In ihr kann der Zähler bei einer erfolgreichen Beendigung erhöht werden. Das Gegenstück zum Mount-Vorgang ist das Unmounten und somit die Funktion `put_super()`.

In Tabelle 9.3 sind die wichtigsten funktionalen Einheiten zusammengefasst, die als Modul implementiert werden können. Zu jeder Einheit werden die Registrier- und Abmeldefunktionen angegeben.

9.6 Der Kernel-Dämon

Gegenüber der Version 2.0 wurde der Kernel-Dämon abgeschafft. Einer der Gründe dafür ist, dass man SysV-IPC und Unix Domain Sockets selbst nicht als Modul realisieren konnte. Diese werden nämlich für die Funktion des Kernel-Dämons schon benötigt.

Von den Funktionen im Kern, die sich mit der Anforderung von Modulen beschäftigen, ist nur `request_module()` übrig geblieben. Ihre Implementation erzeugt einen neuen Prozess, der dann das Programm `modprobe` mit dem entsprechenden Modulnamen ausführt.

Einheit	Funktionen
Dateisystem	`register_filesystem()`
	`unregister_filesystem()`
Blockgerätetreiber	`register_blkdev()`
	`unregister_blkdev()`
Zeichengerätetreiber	`register_chrdev()`
	`unregister_chrdev()`
Netzwerkgerätetreiber	`register_netdev()`
	`unregister_netdev()`
PCMCIA-Geräte	`register_pccard_driver()`
	`unregister_pccard_driver()`
Exec-Domain	`register_exec_domain()`
	`unregister_exec_domain()`
Binärformat	`register_binfmt()`
	`unregister_binfmt()`
Netzprotokolle	`sock_register()`, `dev_add_pack()` und
	`register_netdevice_notifier()`
	`sock_unregister()`, `dev_remove_pack()`
	und `unregister_netdevice_notifier()`

Tabelle 9.3: Funktionale Einheiten, die als Module realisiert werden können

9.7 Einfacher Datenaustausch zwischen Modulen

Bei der Entwicklungsarbeit für Module stellt man manchmal fest, dass für einen Teil des Moduls auch andere Alternativen in Betracht gezogen werden müssen. Dann teilt man den Code in mehrere Module auf (*Stacked Module*). Teilweise ist ein breites Funktionsinterface nicht notwendig. Da man aber Daten zwischen den Modulen austauschen muss, benötigt man eine Art Interface. Für diesen Fall bietet Linux 2.4 einen neuen Mechanismus an. Mit Hilfe der Funktionen `inter_module_register()` und `inter_module_unregister()` kann man als Modul einen Speicherbereich unter einem Namen registieren bzw. deregistieren. Auf dieser Basis kann ein Modul also Daten für andere Module bereitstellen. Damit reduziert sich die Interfacedefinition zwischen Modulen auf die Festlegung einer Datenstruktur.

Module können nun mit Hilfe der Funktionen `inter_module_get()` und `inter_module_get_request()` unter Angabe des Names nach einem exportierten Speicherbereich suchen und sich diesen zurückgeben lassen. In der zweiten Variante wird für den Fall, dass sich der Name nicht finden lässt, versucht, ein Modul mit dem als Para-

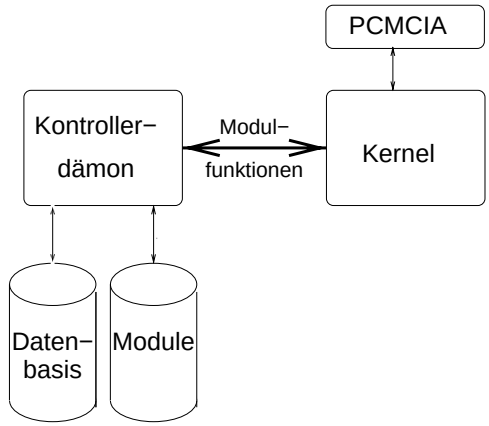


Abbildung 9.2: Dämon zum dynamischen Laden und Entfernen von Modulen

meter übergebenen Modulnamen zu laden. Beide Funktionen erhöhen den `USE_COUNT` des exportierenden Moduls. Wenn man den Zugriff auf den Speicherbereich beendet hat, benutzt man die Funktion `inter_module_put()`, um den Speicherbereich zu dereferenzieren und dem exportierenden Modul die Möglichkeit zum Beenden zu geben.

9.8 Ein Modulbeispiel

Eine interessante Modulanwendung ist der PCMCIA-Kartenmanager. Er verbindet die dynamischen Eigenschaften der Module mit denen des PCMCIA-Systems. Genauso wie eine PCMCIA-Karte nur in den Rechner geschoben wird, wenn ihre Dienste benötigt werden, sorgt der PCMCIA-Kartenmanager dafür, dass die Module für die Karte geladen werden.

Als Basis für diesen Dienst wird ein PCMCIA-Gerät realisiert. Mit dessen Hilfe wird der PCMCIA-Kartenmanager über jede Statusänderung an der PCMCIA-Hardware informiert. Außerdem ermöglicht dieses Gerät das Auslesen der Kartenkennung. Anhand dieser Kennung und der Informationen aus seiner Datenbasis ist der PCMCIA-Kartenmanager nun in der Lage, Module zu laden bzw. zu entfernen.

Um die notwendige Grundfunktionalität im Kern zu realisieren, wurden auch hier Module gewählt. Das Basispaket gliedert sich in drei Module. Es gibt ein zentrales Modul, das den allgemeinen Standard für PCMCIA enthält. Ein zweites Modul steuert den Controller-Chip des PCMCIA an. Da es von diesen Chips zwei unterschiedliche Typen gibt, existieren auch zwei unterschiedliche Module für diese Aufgabe. Zum Schluss kommt noch ein Modul, in dem die Schnittstellen realisiert sind. Dazu gehören der Zeichengerätetreiber für das PCMCIA-Gerät und die Funktionen für die auf diesem System aufbauenden Gerätetreiber.

Die Funktion des PCMCIA-Kartenmanagers lässt sich nun relativ einfach beschreiben. Er öffnet die mit den einzelnen Sockets (PCMCIA-Einschüben) assoziierten Zeichengeräte.

Mit Hilfe der Geräte kann sich der Manager über Statusänderungen der Sockets informieren lassen. Weiterhin kann er auch detaillierte Informationen zu den eingeschobenen Karten erhalten.

Die für sein Verhalten wichtigen Informationen entnimmt er der Datenbasis. Diese befindet sich normalerweise in der Datei `/etc/pcmcia/config`. In der Datenbasis werden unterschiedliche Geräte definiert. Zur Definition gehören die Module, die geladen sein müssen, und die Programme, die beim Hinzufügen und Entfernen von Karten ausgeführt werden.

```
device "de650_cs"
   module "net/8390", "de650_cs"
   start "/etc/pcmcia/network start %d%"
   stop  "/etc/pcmcia/network stop %d%"
```

Der andere Teil der Daten befasst sich mit dem Erkennen verschiedener Karten. Jede PCMCIA-Karte enthält eine ASCII-Zeichenkette mit ihrem Namen. Anhand dieser Informationen werden die verschiedenen Karten den Geräten zugeordnet.

```
card "Accton EN2212 EtherCard"
   version "ACCTON", "EN2212", "ETHERNET", "*"
   bind "de650_cs"

card "D-Link DE-650 Ethernet Card"
   version "D-Link", "DE-650", "*", "*"
   bind "de650_cs"

card "GVC NIC-2000P Ethernet Card"
   version "GVC", "NIC-2000p", "*", "*"
   bind "de650_cs"
```

Die bis zur Version 2.5.0 erstellten Treiber umfassen Ethernetkarten, Speicherkarten, serielle Karten, Modemkarten, SCSI-Karten und viele mehr.

Mit all diesen Informationen ausgestattet, muss der PCMCIA-Kartenmanager nicht mehr viel tun. Er wartet mit Hilfe des Systemrufes `select()` an den Geräten auf eine Änderung. Tritt eine solche auf, holt er sich die entsprechenden Daten vom Gerät. Dann bestimmt er mit Hilfe der Datenbasis, welche Aktionen durchzuführen sind. Anschließend werden diese ausgeführt. Jetzt sind alle Ereignisse bearbeitet, und der Manager kann wiederum mittels `select()` warten, bis seine Dienste erneut gebraucht werden.

9.9 Debugging

In den wenigsten Fällen wird ein Stück Programmcode sofort fehlerfrei sein. Normalerweise lädt man sein Programm in einen Debugger wie den `gdb` und arbeitet es schrittweise ab, bis man den Fehler gefunden hat. Leider gibt es Software, die sich nicht auf diese einfache Art *debuggen* lässt. Darunter fallen Echtzeitanwendungen, (quasi)parallele Prozesse und Software, die ohne Host-Betriebssystem läuft. Leider treffen auf den Linux-Kern (wie auf alle Betriebssystemkerne) alle diese Eigenschaften zu. Dass man auch und

gerade bei Änderungen an einem Betriebssystemkern nicht vor Fehlern geschützt ist, sollte klar sein. Dieser Abschnitt versucht, Auswege aus diesem Dilemma aufzuzeigen.

9.9.1 Änderungen sind der Anfang vom Ende

Als allgemeine Empfehlung kann gelten: „Versuche den Kern nicht zu ändern, denn wenn ich nichts am Linux-Kern ändere, dann muss ich auch nicht debuggen und spare mir viele Probleme." So einfach diese Aussage ist, sie hat doch für den Kernprogrammierer einige Bedeutung.

Wir wollen niemanden davon abhalten, kreativ am Linux-Kern tätig zu werden. Jeder sollte sich jedoch ernsthaft die Frage stellen, ob die geplante Erweiterung wirklich in den Kern gehört. Oft ist es auch möglich, diese ganz oder teilweise als externes Programm zu implementieren. Zumindest das teilweise Auslagern der Funktionalität in einen externen Prozess ist in vielen Fällen möglich. Ein privilegierter Prozess (d.h. einer mit der UID 0) kann fast alles, was ein Treiber im Kern auch kann. Häufig erfolgt die Kommunikation mit der Hardware nur über I/O-Ports, und das kann ein privilegierter Prozess auch. Diese Vorgehensweise findet man in der Bibliothek `svgalib`, die die Ansteuerung der Grafikmodi verschiedener SVGA-Karten übernimmt. Es gibt natürlich auch Fälle, in denen man mit diesem Vorgehen nicht zum Ziel kommt. Gerätetreiber, die mit der Hardware über Interrupts kommunizieren, benötigen zumindest gezielte Unterstützung durch den Linux-Kern, da nur dieser Interrupts behandeln kann. Nur die wirklich notwendige Funktionalität wird im Kern realisiert, die eigentliche Arbeit sollte ein normaler Prozess übernehmen. Das *User*-Dateisystem ist ein recht gutes Beispiel für dieses Konzept. Es ist zwar kein Teil des Standardkerns, aber auf der dem Buch beiliegenden CD im Verzeichnis `src/extensions/userfs-0.8.1` zu finden.

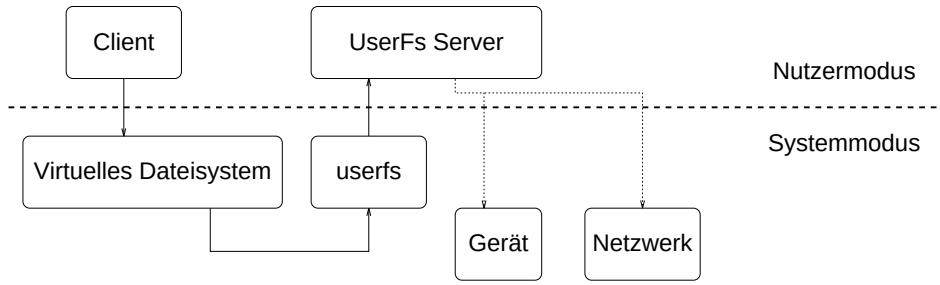


Abbildung 9.3: Funktion des User-*Dateisystems*

Das logische Gruppieren von Daten auf einem physischen Gerät wird traditionell im Betriebssystemkern realisiert, obwohl es genaugenommen da nicht hingehört. Das *User*-Dateisystem erlaubt es, diese Funktionalität in einen normalen Prozess auszulagern. Im Kern ist lediglich eine Schnittstelle für die Anfragen enthalten, die dann an den Prozess weitergeleitet werden.Ein Vorteil ist klar ersichtlich: Der größte Teil des Codes liegt in einem normalen Prozess. Somit kann er mit den normalen Werkzeugen debuggt werden. Ein weiterer Vorteil ergibt sich, wenn man fragt, warum die Dateisystemimplementation denn immer auf eine Festplatte oder Ähnliches zugreifen muss. Der Prozess

kann beliebige Daten als Dateisystem zugänglich machen. So findet sich in der aktuellen Implementierung des *User*-Dateisystems auch ein FTP-Dateisystem. Es greift über das FTP-Protokoll auf die Daten zu. Dadurch können beliebige FTP-Server dem Nutzer zugänglich gemacht werden, als würde er über NFS zugreifen. Einen Nachteil der Architektur des *User*-Dateisystems darf man allerdings nicht verschweigen: Der Zugriff auf die Daten ist nicht besonders schnell.

9.9.2 Der beste Debugger — `printk()`

Ein Test-Log an einer strategischen Stelle kann stundenlanges Debuggen ersparen. Leider braucht man etwas Erfahrung, um die richtigen Stellen zu finden.

Aus diesem Grund sollte man schon beim Entwurf eines Treibers Test-Logs mit einplanen, die sowohl kurz als auch aussagekräftig sind. So ist beim Debuggen des Kerns das Setzen von Breakpoints nur mit großen Änderungen im Kern möglich. Deshalb sollte man sich an diesen Stellen mit passenden Test-Logs begnügen, z. B. mit Hilfe der `printk`-Funktion (siehe Anhang E).

Weiterhin lässt sich hervorragend eine Erweiterung des GNU-C-Compilers verwenden. Er erlaubt C-Präprozessormakros mit einer variablen Anzahl von Argumenten. Man kann also ein Debug-Makro nach dem folgenden Muster definieren:

```
#ifdef DEBUG
#define MY_PRINTK(format, a…)   printk(format, ## a)
#else
#define MY_PRINTK(format, a…)
#endif /* DEBUG */
```

Das so definierte `MY_PRINTK` kann jetzt genauso verwendet werden wie die Funktion `printk()`. Man kann jedoch zur Compile-Zeit festlegen, ob Test-Logs erfolgen sollen. Ein weiterer Vorteil ist, dass man sich gegenüber normalen C-Makros viel Schreibarbeit erspart. Wenn man die Logs zusätzlich noch von einem Flag im Kern abhängig macht, wird dieser Prozess noch dynamischer. Das Flag muss dann durch ein externes Ereignis gesetzt werden. Dazu kann auf das `sysctl`-Interface (siehe A.1) oder ein Ioctl-Kommando zurückgegriffen werden.

Wie in Anhang E beschrieben, werden nur die Logs mit einem Level kleiner der Kernvariablen `console_loglevel` auch auf die Konsole ausgeschrieben. Da man diese Logs manchmal auch dort haben möchte, weil der Kern danach abstürzt oder aus ähnlichen Gründen, muss der Wert von `console_loglevel` entsprechend geändert werden. Dazu bieten sich mehrere Möglichkeiten an:

- mit Hilfe des Systemrufes *syslog*
- direkte Änderung der Variablen

Nach schwerwiegenden Problemen (Traps) schaltet der Kern den Level automatisch auf den höchsten Wert, so dass alle Meldungen auf der Konsole erscheinen.

Für die Benutzer von Modulen sei noch erwähnt, dass eine direkte Manipulation der Variablen `console_loglevel` nur auf Umwegen möglich ist, da sich das zugehörige Symbol nicht in der globalen Symboltabelle befindet. Deshalb muss diese externe Referenz mit Hilfe der Map-Datei `System.map` des Kerns aufgelöst oder das Symbol in der Datei `kernel/ksyms.c` eingetragen werden.

9.9.3 Debuggen mit GDB

Schließlich lässt sich der Linux-Kern auch recht komfortabel mit Hilfe des GNU-Debuggers `gdb` debuggen. Dazu sind zunächst jedoch einige Voraussetzungen zu erfüllen. Der Kern oder zumindest der zu debuggende Bereich des Kerns muss mit Debuginformationen übersetzt werden. Dazu genügt es, im zentralen Makefile des Kerns die Zeile

```
HOSTCFLAGS = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer
```

gegen

```
HOSTCFLAGS = -Wall -Wstrict-prototypes -O2 -g
```

auszutauschen. Anschließend ist der interessante Bereich zu übersetzen und der Kern neu zu linken.

Nun können wir den Debugger mittels

```
# gdb /usr/src/linux/vmlinux /proc/kcore
```

starten. Wie Sie bereits an der Kommandozeile sehen, wird `/proc/kcore` vom Debugger als Core-Datei des Kerns eingelesen. Damit ist es möglich, alle Strukturen des Kerns auszulesen, aber keine lokalen Variablen. Leider kann man weder Werte ändern noch Kernfunktionen aufrufen; die Funktionalität beschränkt sich auf das reine Auslesen von Werten. Trotzdem lassen sich auf diese Weise viele Fehlerfälle finden. Im Unterschied zur Benutzung normaler Core-Dateien liest der `gdb` die Werte aus dem Speicher. Deshalb erhält man immer die aktuellen Werte.

Bei der Benutzung des `gdb` ist weiterhin zu beachten, dass man für die geladenen Module auch die Symboltabelle für diese in den `gdb` laden muss. Dazu müssen auch die Module mit Debuginformationen übersetzt werden. Wenn man nun auf die Werte aus einem geladenen Modul zugreifen möchte, muss man die Symbole aus der .o-Datei des Moduls in den `gdb` laden. Dazu benötigt man die Ladeadresse des Moduls im laufenden Kernel. Diese wird von `insmod` beim Laden in der Symboltabelle des Kernels vermerkt. Sie ist unter `__insmod_<modul name>_S.text_L...` zu finden. Außerdem findet man dort auch `..._S.data_...` und `..._S.bss_...`, diese Werte werden im `gdb` an die Funktion `add-symbol-file` übergeben.[4] Nun kann man ganz normal auf die Variablen und Funktionen aus dem Modul zugreifen.

4 Ein Beispiel für das Nachladen eines Moduls in den `gdb` ist:
`add-symbol-file awe_wave.o 0xc8820060 0xc8828000 0xc882a760`.

Pearson Education

ADDISON-WESLEY

10 Multiprocessing

Auch wenn immer neuere und schnellere Prozessoren auf den Markt kommen, so gibt es doch stets Anwendungen, die mehr Prozessorleistung verlangen. In Multitaskingsystemen bietet es sich jedoch an, auch mehrere Prozessoren einzusetzen, um so echte Parallelverarbeitung von Tasks zu erreichen. Wie in allen echten parallel arbeitenden Systemen steigt jedoch die Leistung nicht linear mit der Anzahl der eingesetzten Prozessoren. Vielmehr trägt das Betriebssystem eine erhöhte Verantwortung, um alle Tasks so unter den Prozessoren aufzuteilen, dass sich möglichst wenig Prozessoren gegenseitig behindern. Dieses Kapitel beschäftigt sich deshalb mit dem Symmetric Multi Processing (SMP), das seit der Linux-Version 2.0 unterstützt wird.

10.1 Die Intel-Mehrprozessorspezifikation

Die meisten der heute verfügbaren Mehrprozessor-Mainboards für PCs verwenden PentiumII- oder PentiumIII-Prozessoren. Diese Prozessoren besitzen intern schon verschiedene Funktionen, die den Mehrprozessorbetrieb unterstützen. Dazu gehören die Cache-Synchronisation, die Interprozessor-Interruptbehandlung sowie atomare Operationen zum Testen, Setzen und Austauschen von Werten im Hauptspeicher. Insbesonders die Cache-Synchronisation erleichtert die SMP-Implementierung im Kern erheblich.

Die Mehrprozessorspezifikation MP 1.4 von Intel [SMP] definiert die Zusammenarbeit von Hardware und Software, um sowohl die Entwicklung von SMP-tauglichen Betriebssystemen zu vereinfachen als auch die Möglichkeit zu schaffen, dass diese auf neuer Hardware laufen. Das Ziel der Spezifikation ist es, eine Multiprozessor-Plattform zu schaffen, die 100%ig kompatibel zum PC/AT bleibt. Sie definiert eine hochgradig symmetrische Architektur in Bezug auf:

Speichersymmetrie Alle Prozessoren teilen sich denselben Hauptspeicher, insbesondere sind auch alle physischen Speicheradressen gleich. Das bedeutet, alle Prozessoren führen dasselbe Betriebssystem aus, alle Daten und Anwendungen sind für alle Prozessoren sichtbar und können auf jedem Prozessor benutzt bzw. ausgeführt werden.

I/O-Symmetrie Alle Prozessoren teilen sich dasselbe I/O-Subsystem (einschließlich der I/O-Ports und des Interrupt-Controllers). Die I/O-Symmetrie erlaubt eine Verringerung eines möglichen I/O-Flaschenhalses. Allerdings gibt es MP-Systeme, bei denen alle auftretenden Interrupts an einen einzigen Prozessor delegiert werden.

Abbildung 10.1 zeigt den Hardwareüberblick über ein typisches SMP-System mit zwei Prozessoren. Beide sind über den ICC-Bus (*Interrupt Controller Communications*) mit einem oder mehreren I/O-APICs (*Advanced Programmable Interrupt Controller*), erweiterten Interruptcontrollern verbunden. Pentium-Prozessoren besitzen jeweils einen eigenen lokalen APIC. Die lokalen APICs bilden zusammen mit den I/O-APICs eine Einheit, die für die Verteilung eintreffender Interrupts sorgt.

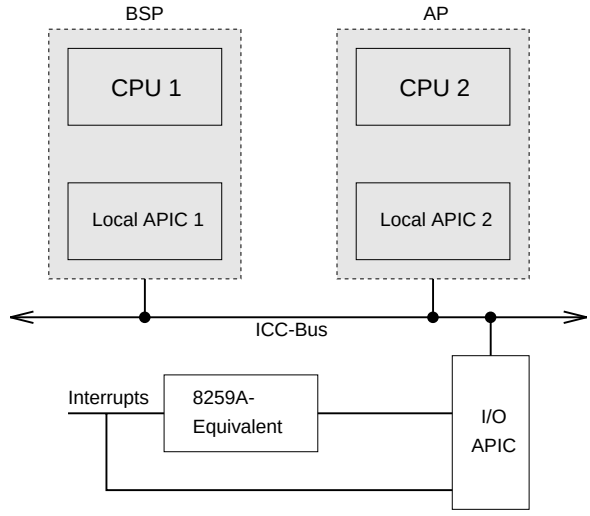


Abbildung 10.1: Ein typisches SMP-System mit zwei Prozessoren

Ein Prozessor wird vom BIOS ausgewählt, er heißt Bootprozessor (BSP) und dient zur Initialisierung des Systems. Alle anderen Prozessoren heißen Applikationsprozessoren (AP) und werden vom BIOS zunächst angehalten. Die MP-Spezifikation definiert eine Konfigurationsstruktur, welche vom BIOS gefüllt wird und die dem Betriebssystem Auskunft über das vorhandene MP-System gibt. Alle Interrupts werden vom BIOS zunächst nur an den Bootprozessor weitergereicht, so dass Einzelprozessorbetriebssysteme keinen Unterschied vorfinden und nur auf dem BSP laufen.

10.2 Probleme bei Mehrprozessorsystemen

Für die korrekte Funktion eines Multitaskingsystems ist es wichtig, dass Daten, die im Kernel sind, immer nur von einem Prozess verändert werden können, um gleiche Resourcen nicht doppelt zu vergeben. In unixartigen Systemen gibt es zwei Herangehensweisen zur Lösung dieser Aufgabe. Traditionelle UNIX-Systeme benutzen ein relativ grobkörniges Locking; manchmal wird auch der gesamte Kern gesperrt, so dass sich nur ein Prozess im Kern befinden kann. Einige modernere Systeme implementieren ein feinkörnigeres Locking, welches jedoch einen großen Mehraufwand mit sich bringt und normalerweise nur für Mehrprozessor- und Echtzeit-Betriebssysteme benutzt wird. In letzteren verringert das feinkörnige Locking die Zeit, die ein Lock gehalten werden muss und erlaubt so eine Verringerung der besonders kritischen Latenzzeit.

Bei der Implementierung des Einzelprozessor-LINUX-Kernels wurden verschiedene Festlegungen getroffen. Eine davon lautet, dass kein Prozess im Kernmodus von einem anderen Prozess im Kernmodus unterbrochen wird, es sei denn, er gibt selbst die Steuerung ab und schläft. Diese Festlegung sichert uns, dass große Bereiche im Kern-Code atomar bezüglich anderer Prozesse sind und vereinfacht damit viele Funktionen im LINUX-Kern.

Eine weitere Festlegung ist, dass die Interruptbehandlung zwar einen Prozess im Kernmodus unterbrechen kann, aber die Steuerung letztendlich wieder an diesen Prozess zurückgegeben wird. Ein Prozess kann Interrupts sperren und somit sicherstellen, dass er nicht unterbrochen wird.

Die letzte für uns wichtige Festlegung lautet, dass die Interruptbehandlung nicht von einem Prozess im Kernmodus unterbrochen werden kann. Das heißt, die Interruptbehandlung wird komplett durchlaufen oder höchstens von einem anderen Interrupt höherer Priorität unterbrochen.

Bei der Entwicklung des Mehrprozessor-Linux-Kerns wurde zunächst die Entscheidung getroffen, diese drei grundlegenden Festlegungen beizubehalten, um sowohl die erste Implementierung zu vereinfachen als auch eine einfache Integration bereits bestehenden Codes zu ermöglichen. Eine einzige Semaphore wurde von allen Prozessoren benutzt, um den Übergang in den Kernmodus zu überwachen. Jeder Prozessor, der diesen Lock hielt, konnte den Kernmodus immer betreten, beispielsweise für die Interruptbehandlung. Sobald der Prozessor den Lock nicht mehr besaß, durfte er nicht mehr in den Kernmodus wechseln.

Durch diese Semaphore wurde sichergestellt, dass kein Prozess im Kernmodus von einem anderen unterbrochen werden konnte. Außerdem wird damit garantiert, dass nur ein Prozess im Kernmodus die Interrupts sperren kann, ohne dass ein anderer Prozessor die Interruptbehandlung übernimmt.

Aus dieser Designentscheidung resultierte jedoch eine schlechte Performance für Ein-/Ausgabe-intensive Anwendungen, da die Rechenzeit im Kernmodus zum Flaschenhals wurde. In der Linux-Version 2.2 wurde deshalb ein feinkörnigeres Locking eingeführt. Nur dieses sichert eine größere Parallelität und damit eine bessere Systemleistung. Mit der aktuellen Mehrprozessor-Implementierung für Linux wird sowohl für rechenzeitintensive Prozesse, die sich meist im Nutzermodus befinden, als auch für Prozesse mit großem Anteil an Ein-/Ausgabe, eine gute Performance erreicht.

10.3 Änderungen am Kern

Um SMP im Linux-Kern zu implementieren, sind sowohl Änderungen im portablen Teil als auch an den prozessorspezifischen Implementierungen notwendig.

10.3.1 Initialisierung des Kerns

Das erste Problem bei der Implementierung des Multiprozessorbetriebes stellt sich beim Starten des Kerns ein. Wir beschreiben hier deshalb den Bootvorgang auf Intel-Prozessoren. Alle Prozessoren müssen gestartet werden, da das BIOS alle APs angehalten hat und zunächst nur der Bootprozessor läuft. Nur dieser betritt zunächst die Startfunktion des Kerns `start_kernel()`. Nachdem er die normale Linux-Initialisierung durchgeführt hat, wird `smp_init()` aufgerufen. Diese Funktion sorgt durch den Aufruf von `smp_boot_cpus()` für die Aktivierung aller anderen Prozessoren.

Nachdem die SMP-Informationen und das notwendige Hardwaresetup überprüft wurde, wird jeder Prozessor gestartet, indem die Funktion `do_boot_cpu()` aufgerufen wird.

Wie wird ein angehalterer Prozessor jedoch gestartet? Diesem Zweck dient der APIC. Er ermöglicht jedem Prozessor, einen so genannten Interprozessorinterrupt (IPI) an andere Prozessoren zu schicken. Weiterhin ist es möglich, jedem Prozessor ein INIT zu schicken (INIT IPI). Ein INIT-Signal wirkt auf einen Pentium-Prozessor wie ein Reset, außer dass Cache, FPU und Write-Buffer gelöscht werden. Der Prozessor springt daraufhin über seinen Reset-Vektor in das BIOS. Wird vorher im CMOS das Warmstart-Flag gesetzt und der Warmstart-Vektor (0040:0067) auf eine Real-Mode-Routine gesetzt, springt der Prozessor diese dann an. Weiterhin ist es möglich, an Pentium-Prozessoren ein STARTUP IPI zu senden. Dadurch beginnt der Prozessor, eine Real-Mode-Routine auf der Adresse VV00:0000 abzuarbeiten.[1]

Kehren wir zurück zur Funktion `smp_boot_cpus()`. Nachdem alle anderen Prozessoren gestartet wurden, enthält die Variable `smp_num_cpus` die Anzahl aller jetzt laufenden Prozessoren. Für jeden Prozessor wurde bereits ein eigener Idle-Task erzeugt.

Als letzte Aktion innerhalb von `smp_init()` ruft der Bootprozessor schließlich `smp_commence()` auf. Diese Funktion setzt das Flag `smp_commenced`, woraufhin alle APs frei laufen können und nun jeweils ihre Idle-Tasks abarbeiten.

10.3.2 Scheduling

Der Linux-Scheduler enthält nur kleine Änderungen. Zunächst besitzt die Task-Struktur nun eine Komponente `processor`, welche die Nummer des bearbeitenden Prozessors enthält oder die Konstante `NO_PROC_ID`, falls noch kein Prozessor zugewiesen wurde. Die Komponente `last_processor` enthält die Nummer des Prozessors, der die Task zuletzt bearbeitete.

Jeder Prozessor arbeitet den Scheduler ab und bekommt eine neue Task zugewiesen, welche abarbeitbar und noch nicht an einen anderen Prozessor zugewiesen ist. Weiterhin werden Tasks, welche zuletzt auf dem gerade verfügbaren Prozessor liefen, bevorzugt. Dies kann zur Verbesserung der Systemperformance führen, wenn die internen Prozessorcaches noch die für den ausgewählten Prozess gültigen Daten enthalten.

10.3.3 Interruptbehandlung

Interrupts werden vom I/O-APIC an die Prozessoren verteilt. Bei Systemstart werden jedoch alle Interrupts nur an den BSP weitergeleitet. Jedes SMP-Betriebssystem muss deshalb den APIC in den SMP-Modus schalten, so dass auch andere Prozessoren Interrupts behandeln können (Ausnahme: IPI). Linux unterstützt diese Betriebsart, so dass Interrupts an alle verfügbaren Prozessoren verteilt werden können.

1 Die MP-Spezifikation definiert den genauen Algorithmus, wie APs zu starten sind. Unter anderem werden dabei an Pentiums eine INIT-IPI und zwei STARTUP-IPIs gesendet.

10.4 Atomare Operationen

Oft genügt es sicherzustellen, dass gewisse einfache Operationen, wie Increment, Decrement und Test von Variablen atomar bezüglich aller Prozessoren durchführbar sind. Einige Probleme, die durch die gleichzeitige Ausführung von Code im Kern-Modus durch mehrere Prozessoren entstehen, lassen sich auf diese Weise lösen. Leider bietet ANSI-C keine Möglichkeit, dies portabel auszudrücken. In der Headerdatei `asm/atomic.h` sind deshalb für die jeweiligen Architekturen atomare Operationen in Assembler codiert. Weiterhin haben diese Funktionen die Eigenschaft, auch auf Einzelprozessoren atomar zu sein, d. h. sie werden nicht von Interrupts unterbrochen.

10.4.1 Der atomare Datentyp

Zunächst definiert `asm/atomic.h` den „atomaren Datentyp“:

```
typedef struct { volatile int counter; } atomic_t;
```

Variablen des Types `atomic_t` werden im Kern also nicht in Registern gehalten, ansonsten sind sie dem skalaren Datentyp `int` äquivalent.[2] Diese Definition allein reicht jedoch noch nicht, um atomare Operationen auf diesem Datentyp zu garantieren, da in der Zeit zwischen Lesen eines Wertes aus dem Speicher und Zurückschreiben des Wertes andere Prozessoren den Wert verändern können.

10.4.2 Zugriffe auf den atomaren Datentyp

Um auf Variablen vom atomaren Datentyp zugreifen zu können, sind die folgenden Funktionen bzw. Makros definiert:

```
atomic_t ATOMIC_INIT(int i);
int atomic_read(atomic_t *v);
int atomic_set(atomic_t *v, int i);
```

Obwohl hier als Funktionen dargestellt, sind sie auf fast allen Architekturen als Makros definiert. `ATOMIC_INIT` dient der initialen Wertzuweisung einer atomaren Variable. Mittels `atomic_read()` kann der Wert einer Variable gelesen werden. `atomic_set()` weist einer Variable schließlich einen neuen Wert zu.

10.4.3 Ändern und Testen von atomaren Variablen

Kommen wir nun zu den komplizierteren Funktionen:

```
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
```

2 Auf der Sparc-Architektur ist jedoch aufgrund der Implementation nicht der gesamte Definitionsbereich nutzbar, sondern nur 24 Bit.

```
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
```

Diese Funktionen erlauben es, den Wert einer atomaren Variable durch Addition oder Subtraktion eines Wertes zu ändern. Die Funktionen `atomic_sub_and_test()` und `atomic_dec_and_test` geben zusätzlich noch zurück, ob durch das Subtrahieren von der atomaren Variable der Wert Null entstanden ist. Zusätzlich unterstützen manche Architekturen noch weitere Funktionen, von denen hier einige aufgezählt sind:

```
/* x86 */
void atomic_clear_mask(int mask, atomic_t *v);
void atomic_set_mask(int mask, atomic_t *v);

/* ALPHA, SPARC */
int atomic_add_return(int i, atomic_t * v);
int atomic_sub_return(int i, atomic_t * v);
int atomic_inc_return(atomic_t * v);
int atomic_dec_return(atomic_t * v);
```

10.5 Spinlocks

Spinlocks haben im Linux-Kern die Aufgabe, sicherzustellen, dass kritische Code-Bereiche zu jedem Zeitpunkt von höchstens einem Prozessor ausgeführt werden. Sie schützen somit den Kernel-Code vor „ungewollter" Parallelausführung durch mehrere Prozessoren. Ein Betrachtung der Spinlocks unter dem Gesichtspunkt der Kommunikation findet sich in Abschnitt 5.1.

Spinlocks werden durch den Datentyp `spinlock_t` repräsentiert. Für jeden zu schützenden Bereich muss ein eigener Spinlock definiert sein. Dieser Spinlock sollte mit Hilfe der Makros `SPIN_LOCK_UNLOCKED` oder `spin_lock_init()` initialisiert werden, wie die folgenden Beispiele zeigen:

```
spinlock_t inode_lock = SPIN_LOCK_UNLOCKED;

/* oder in einer Funktion */
spinlock_t lock;

void f()
{
  spin_lock_init(lock);
}
```

10.5.1 Zutrittsfunktionen

Die folgenden Funktionen und Makros dienen dazu, einen Bereich mit Hilfe eines Spinlocks zu schützen, zum einfacheren Verständnis sind sie alle in Form von Funktionsdefinitionen aufgezeigt:

```
/* ohne Änderung des Interrupt-Zustandes */
void spin_lock(spinlock_t *lock);
void spin_unlock(spinlock_t *lock);
int spin_trylock(spinlock_t *lock);

/* lokale Interrupts werden ab- und wieder eingeschaltet */
void spin_lock_irq(spinlock_t *lock);
void spin_unlock_irq(spinlock_t *lock);

/* lokale Interrupts werden abgeschaltet, der Zustand gesichert */
void spin_lock_irqsave(spinlock_t *lock, int *flags);
void spin_unlock_irqrestore(spinlock_t *lock, int *flags);

/* Softwareinterruptswerden ab- und wieder eingeschaltet */
void spin_lock_bh(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

Um einen kritischen Bereich zu betreten, muss `spin_lock()` aufgerufen werden. Falls sich ein anderer Prozessor bereits in diesem Bereich befindet, bleibt der aktuelle Prozessor solange im Spinlock[3] gefangen, bis der andere Prozessor den kritischen Bereich verlassen hat. In der Linux-Version 2.4 sind Spinlocks übrigens nicht „fair" implementiert, d. h., falls 2 Prozessoren darauf warten, einen kritischen Bereich zu betreten, erhält nicht automatisch derjenige Prozessor den Zutritt, welcher am längsten wartet.

Verlässt ein Prozessor einen kritischen Bereich, muss `spin_unlock()` aufgerufen werden. Vergisst man diesen Aufruf, kann schnell der gesamte Rechner einfrieren, da danach kein Prozessor mehr den Bereich betreten kann.

Mit Hilfe von `spin_trylock()` kann abgefragt werden, ob ein Bereich betreten werden kann. Liefert diese Funktion den Wert 0, ist der Bereich momentan belegt. Ansonsten ist er frei. Ein weiteres `spin_lock` ist nicht mehr erforderlich, da `spin_trylock()` den Spinlock belegt.

Die bisher beschriebenen Funktionen ändern nicht das Interrupt-Flag des lokalen Prozessors, d. h., Interrupts werden weiterhin behandelt, wenn sie erlaubt waren, auch wenn sich der Prozessor im Spinlock befindet. Zusätzlich existieren Variationen dieser Funktionen, die die Abarbeitung von Interrupts auf dem lokalen Prozessor während der Zeit im Spinlock bis zum Verlassen des kritischen Bereiches verbieten.

10.5.2 Read-Write-Spinlocks

Read-Write-Spinblocks sind eine spezielle Art von Spinlocks, die es einer beliebigen Anzahl von lesenden Prozessoren erlauben, sich in einem kritischen Bereich aufzuhalten. Schreibende Prozessoren erhalten jedoch exklusiven Zutritt zu diesem Bereich.

Read-Write-Spinlocks werden durch den Datentyp `rwlock_t` repräsentiert und müssen mit Hilfe der Makros `RW_LOCK_UNLOCKED` oder `rwlock_init()` initialisiert werden. Für lesende Zugriffe dienen die Funktionen bzw. Makros:

3 Der „eingefangene" Prozessor muss dann endlos „Runden drehen". Daher der Name Spinlock.

```
void read_lock(rwlock_t *lock);
void read_unlock(rwlock_t *lock);
```

während für schreibende Zugriffe die folgenden Funktionen und Makros zu benutzen sind:

```
void write_lock(rwlock_t *lock);
void write_unlock(rwlock_t *lock);
```

Analog zu den einfachen Spinlocks sind `*_irq()`, `*_irqsave()` und `*_bh()` Versionen verfügbar.

A Systemrufe

Rufe mich an, dann will ich Dir antworten und will Dir Großes und Unfassbares mitteilen, das Du nicht kennst.

Jeremiah 33,3

Dieses Kapitel beschreibt die Implementierung der Systemrufe in Linux. Bei den architekturabhängigen Implementierungen liegt der Schwerpunkt[1] auf dem auf Intel-PCs laufenden Linux-System. Eine Beschreibung der anderen Architekturen ist aus mehreren Gründen (Zeit und Dokumentationsmaterial) nicht möglich. Grundlegende Kenntnisse erhalten Sie in den vorhergehenden Kapiteln. Sehr zu empfehlen ist außerdem ein Blick in die entsprechenden Quelldateien des Kerns.

Sie müssen genau zwischen dem Systemruf[2] und der entsprechenden Kernelfunktion unterscheiden. Ein Systemruf ist der Übergang eines Prozesses vom Nutzer- zum Systemmodus. In Linux geschieht das durch das Aufrufen des Interrupts 0x80 zusammen mit der konkreten Registerbelegung. Der Kern ruft (im Systemmodus) eine in der Tabelle `_sys_call_table` stehende Kernfunktion auf.

Diese Funktionen, sie beginnen im Quelltext mit „`sys_`", werden hier beschrieben. Mittlerweile sind die Rückkehrwerte der Systemrufe auf den Typ `long` umgestellt. Zum einen werden in den höherwertigen Bits Fehlerinformationen übertragen, zum anderen wurde dies durch die Portierung auf den Merced-Prozessor notwendig.

Die Umsetzung von einer Funktion, die ein Programm verwendet, auf den Systemruf geschieht in der C-Bibliothek. Dadurch ist es zum Beispiel möglich, wie es `sys_socketcall()` gut zeigt, mehrere Funktionen mit einer Kernelfunktion zu bearbeiten. Solche Funktionen haben eine typische Eigenschaft: Parameter, deren Struktur unterschiedlich sein kann, werden der Kernelfunktion als `unsigned long` übergeben. Dieser Parameter wird dann als Adresse verwendet. In Linux ist es üblich, normalerweise bekannte Systemrufe als Bibliotheksfunktionen zur Verfügung zu stellen — was den Unterschied zwischen einem Systemruf und einer C-Bibliotheksfunktion verwischt.

Die Kernelfunktionen sind in sechs Gruppen unterteilt: die Prozessverwaltung, das Dateisystem, die Interprozesskommunikation, die Speicherverwaltung, die Initialisierung und den Rest (nicht oder noch nicht implementierte Systemrufe). Eine grobe Einteilung ist folgende: Systemrufe, deren Quelldateien sich in einem Unterverzeichnis befinden, werden auch in einer Gruppe beschrieben.

---

1 Sollte ein Systemruf für eine andere Architektur nicht verfügbar sein, wird das allerdings explizit erwähnt.

2 Die Diskussion *„Was ist ein Systemruf?"* beschäftigte uns recht lange!

Die Beschreibung einer Kernelfunktion ist ähnlich wie eine UNIX-Manualseite aufgebaut: Links oben steht der Name der Kernelfunktion, rechts oben die Herkunft des entsprechenden Systemrufs (POSIX, BSD, SVR4). Darunter steht der Name der Datei, in der die Kernelfunktion implementiert ist. Sind bei der Verwendung des dazugehörigen Systemrufs spezielle Headerdateien erforderlich, sind diese ebenfalls angegeben. Es folgt der Prototyp der Funktion und die Beschreibung. Die Schnittstelle, die die C-Bibliothek anbietet, und eventuell dabei auftretende Besonderheiten sind im Abschnitt *Implementierung* aufgeführt. Den Schluss bildet eine Aufzählung von Fehlernummern, die bei der Ausführung der Kernfunktion auftreten können.

A.1 Die Prozessverwaltung

Die folgenden Aufrufe greifen auf den Kern eines jeden UNIX-Systems zu, den Scheduler und die Prozessverwaltung. Die Grundlagen dafür werden in dem Kapitel 3 und dem Kapitel 4 beschrieben.

Systemruf adjtimex 4.3+BSD

Datei: kernel/time.c

```
#include <linux/timex.h>

long sys_adjtimex(struct timex *txc_p);
```

Der Aufruf `sys_adjtimex()` ermöglicht das Lesen und Setzen der Zeitstrukturen des Kerns, genauer gesagt, der Variablen, die mit „`time_`“ beginnen. Da diese den Ablauf des Timers kontrollieren, kann so das Zeitverhalten des Systems gesteuert werden[3]. Die Struktur `timex` ist eine Erweiterung der Struktur `timeval`:

```
struct timex {
  unsigned int modes;     /* Funktion                          */
  long offset;            /* Zeitoffset (usec)                 */
  long freq;              /* Frequenzoffset (scaled ppm)       */
  long maxerror;          /* maximaler Fehler (usec)           */
  long esterror;          /* geschätzter Fehler (usec)         */
  int status;             /* Status der Uhr                    */
  long constant;          /* PLL-Zeitkonstante                 */
  long precision;         /* Genauigkeit der Uhr (usec) (ro)   */
  long tolerance;         /* Frequenzschwankungen der Uhr      */
                          /*  (ppm) (read only)                */
  struct timeval time;    /* Systemzeit (read only)            */
  long tick;              /* Mikrosekunden zwischen zwei Ticks */
  long ppsfreq;           /* PPS-Frequenz (scaled ppm) (ro)    */
```

3 Der kommentierte Quelltext spricht von: „to discipline kernel clock oscillator“.

```
  long jitter;            /* PPS-Jitter (us) (ro)             */
  int shift;              /* Interval-Dauer (s) (shift) (ro)  */
  long stabil;            /* PPS-Stabilität (scaled ppm) (ro) */
  long jitcnt;            /* Jittergrenze überschritten       */
                          /* (Zähler) (ro)                    */
  long calcnt;            /* Kalibrierungs-Interval (ro)      */
  long errcnt;            /* Kalibrierungs-Fehler (ro)        */
  long stbcnt;            /* Stabilitatsgrenze überschritten  */
                          /* (Zähler) (ro)                    */
  int  :32; int  :32; int  :32; int  :32; int  :32; int  :32;
  int  :32; int  :32; int  :32; int  :32; int  :32; int  :32;
};
```

Wenn `modes` 0 ist, werden die Werte gelesen, sonst geschrieben. Dazu muss das Recht `CAP_SYS_TIME` gesetzt sein. Für das Setzen von Werten können in `modes` folgende Werte eingetragen werden:

`ADJ_STATUS` — `time_status` wird gesetzt.

`ADJ_FREQUENCY` — `time_freq` ergibt sich aus `txc.frequency`.

`ADJ_MAXERROR` — `time_maxerror` wird gesetzt.

`ADJ_ESTERROR` — `time_esterror` wird gesetzt.

`ADJ_TIMECONST` — `time_constant` wird gesetzt.

`ADJ_OFFSET` — Ist außerdem `ADJ_OFFSET_SINGLESHOT` gesetzt oder sind PLL-Änderungen erlaubt, ergibt sich der Wert `time_adjust` aus `txc.offset`. Sonst wird `time_offset` auf den Wert `txc.offset << SHIFT_UPDATE` sowie `time_reftime` auf `xtime.tv.sec` gesetzt. `time_freq` wird dann neu berechnet.

`ADJ_TICK` — `tick` wird auf `txc.tick` gesetzt. Aus Stetigkeitsgründen darf der Wert `txc.tick` nicht mehr als zehn Prozent vom Normalwert (1000) abweichen.

`ADJ_OFFSET_SINGLESHOT` — ermöglicht zusammen mit ADJ_OFFSET die Emulierung des bekannten Systemrufs *adjtime*.

Da der Timerinterrupt die Einstellungen stören würde, werden während des Kopierens Interrupts ausgeschaltet. Nach dem Kopieren wird die Struktur `txc` mit den jetzt gültigen `time_`-Werten gefüllt (`offset` enthält den vorher abgespeicherten `time_adjust`-Wert) und zurückgegeben.

Implementierung

Der Systemruf wird mit dem Syscall-Makro umgesetzt. Außerdem baut der bekannte Systemruf *adjtime* auf der Funktion `adjtimex()` auf, wie der folgende (gekürzte) Quelltext zeigt:

```
int adjtime(struct timeval * itv, struct timeval * otv)
{ struct timex tntx;
```

```
  if (itv) {
    struct timeval tmp;
    tmp.tv_sec = itv->tv_sec + itv->tv_usec / 1000000L;
    tmp.tv_usec = itv->tv_usec % 1000000L;
    tntx.offset = tmp.tv_usec + tmp.tv_sec * 1000000L;
    tntx.mode = ADJ_OFFSET_SINGLESHOT;
  }
  else tntx.mode = 0;
  if (adjtimex(&tntx) < 0) return -1;
  return 0;
}
```

Fehler

EPERM — Ein Schreibzugriff erfolgte ohne Superuserrechte.

EINVAL — Ein Wert der Struktur `txc` ist ungültig.

Systemruf alarm POSIX

Datei: kernel/sched.c kernel/itimer.c

```
unsigned long sys_alarm(unsigned int seconds);
```

`sys_alarm()` setzt einen Timer (`itimerval`) auf den Wert `seconds`. Nach dem Ablauf des Timers wird das Signal SIGALRM ausgelöst. Wenn `seconds` gleich null ist, wird der Timer zurückgesetzt.

Läuft noch ein alter Alarm, wird dessen restliche Zeit zurückgegeben. Die Restzeit sind die Sekunden und (wenn nur noch Millisekunden laufen) der Wert 1. Die Abarbeitung des Alarms ist in Abschnitt 3.2.1 beschrieben.

Implementierung

Die Umsetzung geschieht über das Syscall-Makro.

Systemruf brk

Datei: mm/mmap.c

```
unsigned long sys_brk(unsigned long brk);
```

`sys_brk()` ändert die Größe des nicht verwendeten Bereichs des Datensegmentes. Er setzt den Wert `mm->brk` der Taskstruktur auf `new_brk`. Vorher wird `brk` auf den Beginn der nächsten Speicherseite aufgerundet.

Die Funktion `do_mmap()` (siehe Abschnitt 4.2.2) organisiert den nötigen Speicher (`vma`-Bereiche) und setzt die Flags `PROT_READ`, `PROT_WRITE` und `PROT_EXEC` sowie `MAP_FIXED` und `MAP_PRIVATE`. Der neue `brk`-Wert wird anschliessend zurückgegeben.

Implementierung

Der Systemruf verwendet bei Intel-Systemen nicht das Syscall-Makro, sondern springt direkt über Assemblercode in den Interrupt 0x80.

Dieser Systemruf wird bei `malloc()` verwendet, um Speicher zu allozieren. Der von `malloc()` geforderte Speicher wird zum aktuellen `brk`-Wert addiert und angefordert.[4]

Fehler

ENOMEM — Es ist kein Speicher für einen größeren `brk`-Wert verfügbar.

Systemruf	**capget** **capset**	POSIX

Datei: kernel/capability.c

```
#include <linux/capability.h>

long sys_capget(cap_user_header_t header,
        cap_user_data_t dataptr);
long sys_capset(cap_user_header_t header,
        const cap_user_data_t data);
```

Die Funktionen `sys_capget()` und `sys_capset()` ermöglichen den Zugriff auf die Rechte (*capabilities*) eines Prozesses. Der Parameter `header` hat folgende Struktur:

```
typedef struct __user_cap_header_struct {
        __u32 version; /* interne Versionsnummer        */
        int pid;       /* PID des gewünschten Prozesses */
} *cap_user_header_t;
```

Wenn die übergebene Version (ist in `capability.h` definiert) nicht mit der des Kerns übereinstimmt, wird die Version des Kerns eingetragen und ein Fehler zurückgegeben. Der zweite Parameter (`dataptr`) hat folgende Struktur:

```
typedef struct __user_cap_data_struct {
        __u32 effective;   /* effektive  Rechte */
        __u32 permitted;   /* erlaubte Rechte   */
```

4 Die dafür verwendete Funktion heißt bezeichnenderweise `morecore()`, was nur ein Zeiger auf `brk()` ist.

```
        __u32 inheritable; /* geerbte Rechte    */
} *cap_user_data_t;
```

Diese Struktur enthält die Daten für das Setzen der neuen Rechte.

In `sys_capget()` werden die Rechte des Prozesses `pid` in `dataptr` gespeichert. Ist `pid` gleich 0, werden die Rechte des aktuellen Prozesses verwendet.

Die Rechte eines Prozesses kann man mit `sys_capset()` setzen. Dabei ist Folgendes zu beachten: Die Rechte eines Prozesses können nur eingeschränkt werden und Rechte fremder Prozesse können verändert werden, wenn der Prozess selbst das Recht `CAP_SETCAP` hat. Bei den Einschänkungen gilt, dass das neue Recht eine Teilmenge (kleiner oder gleich) sein muss und zwar bei

geerbten Rechten von den bitweise mit ODER verknüpften geerbten Rechten des Zielprozesses und den erlaubten Rechten des aktuellen Prozesses,

erlaubten Rechten von den bitweise mit ODER verknüpften erlaubten Rechten des Zielprozesses und den erlaubten Rechten des aktuellen Prozesses sowie

effektiven Rechten von den neuen erlaubten Rechten.

Wird als `pid` ein Wert kleiner 0 benutzt, hat das besondere Folgen: Ist `pid` gleich -1, werden die Rechte aller Prozesse (außer `init`) gesetzt, ansonsten die Rechte aller Prozesse mit der Prozessgruppe `-pid`. Für die obigen Tests wird dabei als Zielprozess der aktuelle Prozess eingesetzt.

Implementierung

Beide Systemrufe werden über das Syscall-Makro umgesetzt.

Fehler

EINVAL — wenn die falsche Version in `header` eingetragen ist.

EPERM — wenn ein unberechtigter Prozess fremde Rechte ändern will.

ESRCH — wenn der angegebene Prozess nicht existiert.

Systemruf exit POSIX

Datei: kernel/exit.c

```
long sys_exit(int error_code);
```

Die Kernelfunktion verschiebt die zwei untersten Bytes um acht Stellen nach links und übergibt den Wert an `do_exit()`. Diese Funktion gibt im Kern alle vom Prozess benutzten Ressourcen frei und benachrichtigt betroffene Prozesse.

An den Elternprozess wird der Wert `status` zurückgegeben. Der Ablauf wird in Abschnitt 3.3.3 beschrieben.

Implementierung

Der Systemruf wird ohne Änderung der Parameter in die Kernelfunktion umgesetzt.

Systemruf	**clone** **fork** **vfork**	POSIX LINUX

Datei: arch/i386/kernel/process.c
kernel/fork.c

```
long sys_clone(struct pt_regs regs);
long sys_fork(struct pt_regs regs);
long sys_vfork(struct pt_regs regs);
```

`sys_fork()` erzeugt einen neuen Prozess (Kindprozess) als Kopie des aktuellen Prozesses (Elternprozess). Um zwischen Eltern- und Kindprozess zu unterscheiden, wird im Elternprozess die PID des Kindprozesses und im Kindprozess eine 0 zurückgegeben. In LINUX wird *Copy-On-Write* benutzt, so dass nur die Pagetabellen und die Taskstruktur dupliziert werden. Ein Kindprozess kann nur erzeugt werden, wenn die Ressourcen des Vaterprozesses (`rlim[RLIMIT_NPROC]`) und des Systems nicht überschritten werden. Die maximale Anzahl der Prozesse (genauer: der Threads) wird beim Booten mit folgendem Wert initialisiert:

```
max_threads = num_physpages / (THREAD_SIZE/PAGE_SIZE) / 2;
```

Der Kindprozess sendet bei seinem `exit()` ein `SIGCHLD` an den Vaterprozess.

Um die Semantik des Systemrufs *fork* zu erweitern, bietet LINUX einen Systemruf *clone* an. Mit den Registern `regs` werden zwei Parameter übergeben. Im Register `regs.ebx` befinden sich die Flags und das Signal. Das Signal steht in den unteren zwei Bytes und wird bei der Beendigung des Kindprozesses an den Elternprozess geliefert. Interessanterweise wird das Signal später mit `0xff` ausgeblendet, was sogar 255 Signale möglich macht. Die Flags steuern die „Kinderstube“ des neuen Prozesses:

`CLONE_VM` — Der Eltern- und der Kindprozess teilen sich dieselben Speicherseiten. Ist dieses Flag nicht angegeben, werden mittels *Copy-On-Write* die Speicherseiten des Kindes erzeugt.

`CLONE_FS` — Der Eltern- und Kindprozess benutzen dieselbe Dateisystem-Struktur (wobei der Zähler inkrementiert wird). Ansonsten wird sie kopiert.

CLONE_FILES — Der Eltern- und der Kindprozess benutzen dieselben Deskriptoren. Ansonsten werden die Dateideskriptoren kopiert.

CLONE_SIGHAND — Der Eltern- und der Kindprozess teilen sich die Signalbehandlungsroutinen. Ansonsten werden diese Strukturen kopiert.

CLONE_PID — Der Eltern- und der Kindprozess teilen sich die PID.

CLONE_PTRACE — Der Eltern- und der Kindprozess teilen sich die `ptrace()`-Flags.

CLONE_VFORK — Bei einem `mm_release()` des Elternprozesses wird der Kindprozess geweckt.

CLONE_PARENT — Der Eltern- und der Kindprozess bekommen denselben Vater.

Im Register `regs.ecx` befindet sich ein Zeiger, der als Stackpointer des Kindeprozesses verwendet wird. Steht darin 0, wird der Stackpointer des Elternprozesses verwendet. Die Implementierung der Systemrufe ist in Abschnitt 3.3.3 beschrieben.

Der Systemruf *vfork* ist im Prinzip ein Aufruf von *clone*, bei dem intern die Flags `CLONE_VFORK`, `CLONE_VM` und `SIGCHLD` gesetzt werden.

Implementierung

Die Umsetzung des Systemrufs *fork* geschieht über das Syscall-Makro. Die Struktur `pt_regs` aus `<asm/ptrace.h>` enthält genau die Register in ihrer Reihenfolge, die ein Systemruf auf den Stack legt. Dadurch hat die Kernfunktion darauf Zugriff, obwohl der Systemruf parameterlos ist. Der Ruf `clone()` wird ebenfalls durch das Syscall-Makro umgesetzt.

Fehler

ENOMEM — wenn `sys_fork()` keinen Speicher für die Pagetabelle und die Taskstruktur allozieren kann.

EAGAIN — wenn kein freier Prozess mehr vorhanden war.

Systemruf				POSIX 4.3+BSD
	getpid	**getuid**	**geteuid**	
	getgid	**geteuid**	**getegid**	
	getppid	**getpgid**	**getpgrp**	
	getsid	**setuid**	**setgid**	
	setreuid	**setregid**	**setsid**	
	setfsuid	**setfsgid**		

Datei: kernel/sched.c
kernel/sys.c

```
long sys_getpid(void);          long sys_getuid(void);
long sys_geteuid(void);         long sys_getgid(void);
long sys_getegid(void);         long sys_getppid(void);
long sys_getpgid(pid_t pid);    long sys_getpgrp(void);
long sys_getsid(pid_t pid);     long sys_setuid(uid_t uid);
long sys_setgid(gid_t gid);     long sys_setsid(void);
long sys_setfsuid(uid_t uid); long sys_setfsgid(gid_t gid);
long sys_setreuid(uid_t ruid, uid_t euid);
long sys_setregid(uid_t rgid, uid_t egid);
long sys_setpgid(pid_t pid,pid_t pgid);
```

All diese Rufe lesen (bzw. setzen) Identifikationsnummern des Prozesses. Mittlerweile haben die Nummern für die Identifizierung von Nutzern und Gruppen eine Größe von 32 Bit. Aus Kompatibilitätsgründen gibt es die dafür verantwortlichen Rufe noch einmal mit dem Suffix 16 (z.B. `sys_getuid16()`), die dann intern den Wert auf 16 Bit herunterrechnen.

`sys_getpid()` und `sys_getpgrp()` ermitteln die Prozessidentifikation (PID) bzw. die Prozessgruppe (PGRP[5]) des aktuellen Prozesses. Die Prozessgruppe eines beliebigen Prozesses `pid` gibt `sys_getpgid()` zurück. Ist `pid` gleich 0, gibt sie die eigene zurück. Die Funktion `sys_getppid()` gibt die PID des Elternprozesses (PPID) zurück. Die Funktion `sys_getsid()` gibt die Sitzung des Prozesses `pid` zurück.

Die Funktion `sys_getuid()` gibt die ID des Nutzers (UID) und die Funktion `sys_getgid()` gibt die ID der Gruppen (GID) des rufenden Prozesses zurück. Die effektive Nutzer- (EUID) und Gruppenidentifikation (EGID) ermitteln die Kernfunktionen `sys_geteuid()` und `sys_getegid()`.

All diese Funktionen lesen einfach die Taskstruktur des rufenden Prozesses aus, wie folgendes Beispiel zeigt:

```
asmlinkage int sys_getpid(void)
{
        return  current->pid;
}
```

`sys_setpgid()` setzt die Prozessgruppe des aktuellen Prozesses oder eines seiner Kinder auf `pgid`. Wenn `pid` gleich 0 ist, wird der Wert des aufrufenden Prozesses verwendet, und wenn `pgid` 0 ist, der Wert `pid`. Die PGRP kann nur für den eigenen Prozess oder für einen Kindprozess geändert werden — im letzteren Fall auch nur, wenn der Kindprozess sich in derselben Sitzung befindet und `did_exec` gesetzt ist (siehe Abschnitt 3.1.1).

Außerdem muss in beiden Fällen für den Prozess ein `leader` definiert sein, und es muss einen Prozess geben, der schon die angegebene PGRP besitzt. Er darf sich aber nicht in einer anderen Sitzung befinden.

Die Funktionen `sys_setreuid()` und `sys_setregid()` manipulieren die UID und EUID bzw. die GID und EGID eines Prozesses. Ist `ruid` größer −1, wird die UID auf

5 Es gibt auch die Bezeichnung PGID für die Prozessgruppe (z.B. bei *ps*).

diesen Wert gesetzt. Dazu wird geprüft, ob eine der folgenden Bedingungen zutrifft: Der rufende Prozess hat das Recht CAP_SETUID und die UID oder EUID ist gleich ruid. Dann wird ruid als neue UID gespeichert (aber noch nicht gesetzt), ansonsten wird ein Fehler zurückgegeben. Ist euid größer als −1, wird die EUID auf diesen Wert gesetzt. Für das Setzen der EUID auf euid ist eine der folgenden Bedingungen Voraussetzung das Recht CAP_SETUID; UID, EUID oder SUID gleich euid. Dann wird die EUID gesetzt, ansonsten wird ein Fehler zurückgegeben. Ist ruid oder euid gleich −1 und ungleich der UID, wird die SUID des Prozesses auf den Wert der EUID gesetzt. Gab es bis hier keinen Fehler, setzt die Funktion zum Schluss noch FSUID auf EUID unddie UID auf den gemerkten Wert. Beim Setzen der UID wird das Limit RLIMIT_NPROC überprüft. Wenn außerdem im Kern in den Secure Bits SECURE_NO_SETUID_FIXUP gesetzt ist, werden die Rechte des Prozesses angepasst (siehe Anhang E).

So kann ein normaler Benutzer nur effektive und reale IDs vertauschen. setreuid(geteuid(),getuid()) führt den Tausch durch; nach einem erneuten Aufruf sind die ursprünglichen Werte wiederhergestellt.

Für den Aufruf von setregid() gilt dasselbe, nur dass er die Gruppen- statt der User-ID betrifft und die letzten beiden Schritte (Limit und Rechte) wegfallen.

sys_setuid() setzt die UIDs eines Prozesses auf uid. Falls der Prozess das Recht CAP_SETUID besitzt, sind das UID, EUID, SUID und FSUID. Ansonsten werden nur FSUID und EUID gesetzt, vorausgesetzt, uid ist gleich der UID oder der SUID. Bei einer Änderung der UID wird wieder RLIMIT_NPROC getest, und die Rechte werden aktualisiert. Als Äquivalent für das Setzen der Prozess-GIDs gibt es die Funktion sys_setgid(). Die Funktionen sind das SVR4-Pendant zu den obigen set-Aufrufen, die aus dem BSD-Universum stammen. Beachten Sie, dass es hier keine Möglichkeit gibt, die einmal geänderte EUID zurückzusetzen, wie es bei sys_setreuid() möglich ist. Zurückgegeben wird 0 und im Fehlerfall ein negativer Wert.

Die Funktionen sys_setfsuid() und sys_setfsgid() setzen die FSUID bzw. FS-GID. Das ist (für das Setzen der UIDs) nur erlaubt, wenn entweder UID, EUID, SUID oder FSUID gleich uid ist oder das Recht CAP_SETUID gesetzt ist. Für das Setzen der GID muss eine der ersten vier Bedingungen zutreffen. Beim Setzen der FSUID von 0 auf einen anderen Wert wird das Recht CAP_MASK, beim Setzen auf 0 werden alle Rechte außer CAP_MASK gelöscht. Zurückgegeben wird jeweils die alte ID.

sys_setsid() macht den aufrufenden Prozess zum Sitzungsführer. Er setzt SESSION und PGRP auf PID, die Komponente leader der Taskstruktur auf 1 und löscht sein Controlling-Terminal. Wenn es schon einen Prozess gibt, der Sitzungsführer ist (PGRP gleich PID des aktuellen Prozesses), ist das ein Fehlerfall. Der Rückgabewert ist die neue PGRP.

Implementierung

Unterstützt durch die Einfachheit der Funktionen, erfolgt die Umsetzung über das Syscall-Makro.

Die bekannten Systemrufe *seteuid*, *setegid* und *setpgrp* stellt LINUX als Bibliotheksfunktionen zur Verfügung. Die Umsetzung wird hier am Beispiel von `seteuid()` gezeigt:

```
int seteuid(uid_t uid)
{
    return setreuid(-1, uid);
}
```

Fehler

EINVAL — wenn eine ungültige PID, PGID usw. an eine Funktion übergeben wird.

EPERM — wenn die benutzte Funktion nicht erlaubt ist. Im Allgemeinen dürfen nur berechtigte Prozesse alle Prozessdaten ändern. Normale Benutzer können nur ihre Gruppen- und Nutzer-IDs ändern.

ESRCH — wenn bei `sys_setpgid()` kein Prozess gefunden wird.

Systemruf	**getpriority** **setpriority**	4.3+BSD

Datei: kernel/sys.c

```
#include <linux/time.h>
#include <linux/resource.h>

long sys_getpriority(int which, int who);
long sys_setpriority(int which, int who, int niceval);
```

Die Kernelfunktionen `sys_getpriority()` und `sys_setpriority()` verwalten die Prioritäten für das Scheduling.

Zum Abfragen dient `sys_getpriority()`. Der Parameter `which` gibt an, ob man die Priorität eines Prozesses, einer Prozessgruppe oder eines Nutzers abfragen will. In `who` wird der Wert angegeben. Es sind folgende Werte für `which` möglich:

`PRIO_PROCESS` — Der Wert in `who` gibt eine PID an.

`PRIO_PGRP` — Der Wert in `who` gibt eine PGRP an.

`PRIO_USER` — Der Wert in `who` gibt eine UID an.

Wenn für `who` 0 angegeben wird, verwendet der Kern den Wert des aktuellen Prozesses.

Alle Prozesse werden darauf durchsucht, ob sie zu den angegebenen Werten passen (`proc_sel()`). Zurückgegeben wird der höchste gefundene Wert, falls mehrere Einträge gefunden werden (Prozessgruppe).

Die gefundene Priorität wird noch auf das Intervall $[0, 40]$ skaliert und dann zurückgegeben. Dadurch wird das Manko früherer Versionen beseitigt, in denen es gültige negative Rückkehrwerte gab.

Die Funktion `sys_setpriority()` setzt die Priorität für den mit `which` und `who` ausgewählten Prozess, wobei `niceval` zwischen $[-20, 20]$ liegen muss. Die Priorität wird wieder auf Zeitscheiben-Einheiten skaliert und allen gefundenen Prozessen zugewiesen. Für die Zuweisung muss entweder die UID des gefundenen Prozesses gleich der UID oder EUID des aktuellen sein, oder es muss das Recht `CAP_SYS_NICE` gesetzt sein.

Implementierung

Während `setpriority()` einfach das Syscall-Makro verwendet, baut der Ruf `getpriority()` den Interrupt 0x80 von Hand zusammen und rechnet die Spiegelung an `PZERO` wieder zurück, damit der Rückgabewert von `getpriority()` mit dem `sys_setpriority()` übergebenen Wert übereinstimmt.

Dadurch gibt es hier eine *gefährliche Ausnahme*! Es ist möglich, dass die Bibliotheksfunktion `getpriority()` -1 zurückgibt und trotzdem kein Fehler aufgetreten ist. Hier sollte also für die Fehlerkontrolle nicht der Rückgabewert, wie normalerweise in UNIX üblich, sondern auch `errno` abgefragt werden.

Fehler

ESRCH — wenn zu `which` und `who` kein passender Prozess gefunden wurde.

EINVAL — wenn für `which` ein ungültiger Wert angegeben wird.

EPERM — wenn bei `sys_setpriority()` die EUID des angegebenen Prozesses ungleich der EUID des aufrufenden Prozesses ist.

EACCES — wenn ein nichtprivilegierter Benutzer die Priorität höher setzen will.

Systemruf ioperm iopl LINUX

Datei: arch/i386/kernel/ioport.c

```
int sys_ioperm(unsigned long from,
        unsigned long num, int turn_on);
int sys_iopl(unsigned long unused);
```

Diese Aufrufe können nur mit dem Recht `CPA_SYS_RAWIO` benutzt werden. Die Bits der Portzugriffsrechte setzt `sys_ioperm()`. Es werden `num` Bits ab der Adresse `from` auf den Wert `turn_on` gesetzt. Dabei bedeutet der Wert 1 vollen Zugriff auf den Port (lesen und schreiben) und der Wert 0 keinen Zugriff. Es können nur die ersten 1.023 (32*`IO_BITMAP_SIZE`) Ports gesetzt werden.

Um unter Linux zum Beispiel für den X-Server alle 65.536 Ports ansprechen zu können, wird der Systemruf *iopl* zur Verfügung gestellt. Die zugehörige Kernelfunktion `sys_iopl()` betrachtet `unused` als Zeiger auf eine `pt_regs`-Struktur und trägt den Wert `regs->ebx` als I/O-Privilegierungsstufe des Prozesses ein. Es werden normalerweise nur zwei Level der vier möglichen Stufen verwendet, Level 0 und Level 3.

Implementierung

Beide Systemrufe arbeiten über das Syscall-Makro.

Fehler

EINVAL — wenn für `num` ein negativer Wert angegeben wurde, `from+num` größer als 1023 oder `level` größer als 3 ist.

EPERM — wenn der rufende Prozess nicht berechtigt ist.

Systemruf kill POSIX

Datei: kernel/exit.c

```
#include <signal.h>

long sys_kill(int pid, int sig);
```

`sys_kill()` sendet das Signal `sig` an einen Prozess oder eine Prozessgruppe, genauer an die Struktur `siginfo`. Die gesamte Struktur ist auf Seite 325 zu finden.

```
typedef struct siginfo {
  int si_signo;
  int si_errno;
  int si_code;

  union { …  } _sifields;
} siginfo_t;
```

Die Funktion setzt `si_sino` auf `sig`, `si_errno` auf 0 und `si_code` auf den Wert `SI_USER`. Die Struktur `_kill` der Union wird mit den Werten des aktuellen Prozesses gefüllt.

Wenn `pid` eine Zahl größer als 0 ist, wird das Signal an den Prozess mit der PID `pid` gesendet. Ist `pid` gleich 0, wird das Signal an alle Mitglieder der Prozessgruppe des aktuellen Prozesses gesandt. Ist `pid` kleiner als −1, geht das Signal an alle Prozesse der Prozessgruppe `-pid`.

Das Verhalten von `kill(-1,sig)` ist in POSIX nicht definiert. In Linux wird das Signal an jeden Prozess mit einer PID größer als eins (außer an den aktuellen) gesendet.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EINVAL — wenn `sig` ungültig ist.

ESRCH — wenn der Prozess bzw. die Prozessgruppe `pid` nicht existiert.

EPERM — wenn die Rechte des rufenden Prozesses das Senden des Signals nicht erlauben.

Systemruf modify_ldt LINUX

Datei: arch/i386/kernel/ldt.c

```
#include <linux/ldt.h>

int sys_modify_ldt(int func, void *ptr, unsigned long bytecount);
```

Im Zuge der Implementierung von WINE wurde es notwendig, die internen Funktionen von MS-Windows nachzubilden. Dazu gehört auch die Manipulation der lokalen Deskriptortabelle. Genau dazu dient der Systemruf *modify_ldt*. Die Tabelle ist als Bestandteil der Taskstruktur relativ einfach zu manipulieren.

Bei `func` gleich 0 liest die Funktion die lokale Deskriptortabelle des aktuellen Prozesses. Die gewünschte Größe kann mit dem Parameter `bytecount` eingestellt werden. Sollte er noch keine Tabelle besitzen, wird die Defaulttabelle `{0, 0}` geliefert. Ist die Tabelle kleiner, wird nur die Tabelle (mit der Größe `LDT_ENTRIES*LDT_ENTRY_SIZE`) ausgelesen. Zurückgegeben wird die tatsächliche Größe der Tabelle. `ptr` ist hier ein Zeiger auf die Struktur `desc_struct`:

```
struct desc_struct {
        unsigned long a,b;
}
```

Um einen Eintrag in der Tabelle zu ändern, muss `func` 1 oder `0x11` sein. Dann ist der Zeiger `ptr` ein Zeiger auf die Struktur `modify_ldt_ldt_s`:

```
struct modify_ldt_ldt_s {
    unsigned int  entry_number; /* Index des gesuchten Eintrags */
    unsigned long base_addr;
    unsigned int  limit;
    unsigned int  seg_32bit:1;
    unsigned int  contents:2;
    unsigned int  read_exec_only:1;
    unsigned int  limit_in_pages:1;
```

```
    unsigned int  seg_not_present:1;
    unsigned int  useable:1;
};
```

`bytecount` muss genau die Größe der Struktur angeben. Dabei wird die angegebene Struktur in die Tabelle des aktuellen Prozesses geschrieben. Wenn dieser noch keine lokale Deskriptortabelle besitzt, wird eine solche initialisiert. Es ist auch möglich, einen Eintrag zu löschen (0 einzutragen).

Implementierung

Die C-Bibliothek stellt keine Schnittstelle für diesen Systemruf zur Verfügung. Nutzer müssen wie bei `sys_sysinfo()` (siehe Seite 342) vorgehen.

Fehler

ENOSYS — `func` ist ungültig.

EINVAL — `ptr` ist 0 (für das Lesen) oder falsch belegt (für das Schreiben).

Systemruf	create_module	delete_module	Linux
	init_module	query_module	
	get_kernel_syms		

Datei: kernel/module.c

```
unsigned long sys_create_module(char *name, unsigned long size);
long sys_init_module(const char *name_user,
                struct module *mod_user);
long sys_delete_module(char *name);
long sys_query_module(const char *name_user, int which,
                char *buf, size_t bufsize, size_t *ret)
long sys_get_kernel_syms(struct kernel_sym *table);
```

Die Funktion `sys_create_module()` stellt Speicher für ein Modul bereit. `size` ist die Größe des gewünschten Speichers und `name` der Name des Moduls. Der Prozess muss dazu das Recht `CAP_SYS_MODULE` besitzen. Der Ruf erzeugt eine Instanz der Struktur `module` und trägt folgende Werte ein: den Namen, die Größe (Anzahl der Seiten), die Startadresse des freien Speichers und den Status (mit `MOD_UNINITIALIZED`). Alle anderen Werte werden mit `NULL` initialisiert.

```
struct module {
   unsigned long size_of_struct; /* Größe des Moduls          */
   struct module *next;          /* das nächste Modul         */
   const char *name;             /* Name des Moduls           */
   unsigned long size;           /* Modulgröße                */
```

```
   union
   {
      atomic_t usecount;          /* Nutzungszähler            */
      long pad;
   } uc;                          /* Strukturgröße             */

   unsigned long flags;           /* Flags (AUTOCLEAN etc.)    */

   unsigned nsyms;                /* Anzahl der Symbole        */
   unsigned ndeps;                /* Anzahl der Abhängigkeiten */

   struct module_ref *deps;       /* Liste von Modulen, die    */
   struct module_ref *ref;        /*   auf mich zeigen         */
   struct module_symbol *syms;    /* Symboltabelle             */
   init (*init)(void);            /* Initfunktion              */
   void (*cleanup)(void);         /* Löschfunktion             */
   const struct exception_table_entry *ex_table_start;
   const struct exception_table_entry *ex_table_end;
   const struct module_persist *persist_start;
   const struct module_persist *persist_end;
   int (*can_unload)(void);
};
```

Wenn schon ein Modul mit gleichem Namen existiert, wird ein Fehler zurückgegeben, ansonsten ist der Rückgabewert die Adresse des allozierten Speichers im Kernel-Adressraum. `sys_init_module()` lädt das Modul und aktiviert es. `code` ist die Adresse, an die das Modul geladen wird, und `codesize` ist die Größe in Bytes. Diese darf nicht größer sein als der in `module->size` gespeicherte Wert. Wenn das geladene Modul nicht auf einer Seitenadresse endet, wird der Rest mit 0 initialisiert. Der Prozess muss dazu das Recht `CAP_SYS_MODULE` besitzen.

Nachdem das Modul geladen ist, wird `init()`, die eigene Initialisierungsroutine, aufgerufen und der Status auf `MOD_RUNNING` gesetzt; das Modul ist aktiviert.

Die Funktion `sys_delete_module()` entfernt Module wieder. Ist `name` angegeben, wird dieses Modul freigegeben. Der Prozess muss dazu das Recht `CAP_SYS_MODULE` besitzen. Auf das Modul dürfen keine Verweise eingetragen und der Nutzungszähler muss 0 sein. Wenn es in Funktion ist (`MOD_RUNNING`) wird die eigene `cleanup`-Funktion gerufen. Zum Schluß wird der Status auf `MOD_DELETED` gesetzt. Damit kann das Modul durch das danach gerufene `free_modules()` abgeräumt werden. Ist kein Name angegeben, wird die Liste aller Module durchgegangen, und es wird versucht, alle nicht genutzten Module freizugeben.

Um an Informationen über ein (oder alle) Module zu gelangen, gibt es die Funktion `sys_query_module()`. In `name` steht das zu untersuchende Modul, ist dieser NULL, wird der (statische) Beginn der Modulliste (`&kernel_module`) benutzt. Die genaue Funktion hängt von `which` ab:

0 — es wird nur überprüft, ob `name` ein gültiger Modulname ist.

QM_MODULES — Die Namen aller Module werden nacheinander an der Adresse `buf` abgespeichert. In `bufsize` muss die Größe von `buf` enthalten sein. Sollte der Platz nicht für alle Namen ausreichen, wird so weit wie möglich kopiert und in `ret` der fehlende Platz eingetragen.

QM_DEPS — Die Namen aller Module, auf die dieses Modul verweist, werden in `buf` abgelegt. Bei fehlendem Platz gilt das oben Gesagte.

QM_REFS — Die Namen aller Module, die auf dieses Modul verweisen, werden in `buf` abgelegt. Bei fehlendem Platz gilt das oben Gesagte.

QM_SYMBOLS — kopiert die Symbole des Moduls in `buf`. Dabei gilt für das Symbol Nummer `n`: An der Stelle `buf+n` steht die Adresse des Symbols und an `buf+(buf+n+1)` der Name des Symbols.

QM_INFO — Die wichtigsten Informationen des Moduls werden als Struktur `module_info` an die Adresse `buf` geschrieben.

```
struct module_info
{
  unsigned long addr;     /* Adresse des Moduls */
  unsigned long size;     /* Größe des Moduls   */
  unsigned long flags;    /* Flags des Moduls   */
  long usecount;          /* Nutzungszähler     */
}
```

`sys_get_kernel_syms()` ermöglicht den Zugriff auf die Symboltabelle. Er kopiert die Symboltabelle an die durch `table` referenzierte Stelle und gibt die Anzahl der bekannten Symbole zurück. Der Aufruf fragt vorher ab, ob an der Adresse genügend Speicher zum Schreiben frei ist. Deshalb wird normalerweise mit Hilfe des Aufrufs `get_kernel_syms(0)` erst die Größe der Tabelle ermittelt, der benötigte Speicher alloziert und dann noch einmal `get_kernel_syms()` aufgerufen.

Fehler

EBUSY — wenn die Initialisierungsroutine fehlschlägt oder wenn versucht wird, ein noch benutztes Modul freizugeben.

EEXIST — wenn das Modul `name` schon existiert. `sys_create_modul()` kann diesen Fehler liefern.

ENOENT — wenn das Modul `name` nicht existiert. Diese Fehlermeldung ist bei `sys_init_module()` und `sys_delete_modul()` möglich.

ENOMEM — wenn bei `sys_create_modul()` nicht genügend Speicher frei ist.

EPERM — wenn ein nicht berechtigter Prozess einen dieser Systemrufe benutzt.

ENOSPC — wenn die fehlende Größe nicht zurückgeben werden kann.

Systemruf nanosleep LINUX

Datei: kernel/sched.c

```
long sys_nanosleep(struct timespec *rqtp, struct timespec *rmtp);
```

Die gestiegene Taktfrequenz heutiger CPUs ermöglicht (bzw. erfordert) präzisere Zeitstrukturen. Hiermit ist das Anhalten des aktuellen Prozesses auf Nanosekunden-Niveau möglich. Die gewünschte Zeit wird in `rqtp` angegeben:

```
struct timespec {
       long    tv_sec;    /* Sekunden     */
       long    tv_nsec;   /* Nanosekunden */
};
```

Eine Zeit bis 2 ms wird vom Prozess in einer kurzen `for`-Schleife selbst verzögert, sofern nicht `SCHED_OTHER` gesetzt ist. Ansonsten wird die Pause in `jiffies` umgerechnet, als `timeout` des Prozesses eingetragen und der Scheduler aufgerufen. Sollte der Timeout nach dem Wiedereintritt in die Funktion noch nicht abgelaufen sein, wird die restliche Zeit in `rmtp` zurückgegeben.

Fehler

EINVAL — wenn eine negative Zeit oder mehr als 1.000.000.000 Nanosekunden angegeben wurden.

EINTR — wenn eine Restzeit übrig bleibt.

Systemruf nice 4.3+BSD

Datei: kernel/sched.c

```
long sys_nice(long inc);
```

`sys_nice()` legt die Priorität des aktuellen Prozesses fest. Da die Prioritäten in Zeitscheiben gemessen werden, sind einige Umrechnungen nötig. Die neue Priorität ergibt sich (ungefähr) aus der alten Priorität minus `inc`. Das bedeutet: Je größer der Wert `inc` ist, desto kleiner ist die Priorität des Prozesses nach Ausführung des Aufrufs. Nur Prozesse mit dem Recht `CAP_SYS_NICE` sind berechtigt, negative Werte für `inc` anzugeben und damit die Priorität zu erhöhen. Eine gleichzeitige Änderung des Wertes durch `setpriority()` wird im Moment noch nicht berücksichtigt.

Bei der Abarbeitung wird zuerst der absolute Betrag von `inc` auf den maximalen Wert von 40 begrenzt und auf das Interval $[0, 2{*}$`DEF_PRIORITY`$]$ skaliert. Dann ergibt sich die neue Priorität aus der alten minus dem skalierten Wert (bzw. plus, falls `inc`< 0). Sie wird

nach unten mit 1, nach oben mit DEF_PRIORITY*2 begrenzt und dem Prozess dann zugewiesen. sys_nice() verwendet nicht sys_setpriority(). Der Grund dafür besteht vermutlich darin, dass sys_nice() einfach früher implementiert wurde.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EPERM — wenn ein nicht privilegierter Prozess einen negativen Wert für inc angibt.

Systemruf pause POSIX

Datei: arch/i386/kernel/sys_i386.c

```
int sys_pause(void);
```

Die Funktion sys_pause() ist ein sehr einfacher Systemruf. Er setzt den Status des aktuellen Prozesses auf den Wert TASK_INTERRUPTIBLE und ruft den Scheduler auf. Dadurch gibt der Prozess die Steuerung freiwillig ab. Er kann seine Arbeit nur wieder fortsetzen, wenn er durch ein Signal aufgeweckt wird.

Die Funktion gibt -ERESTARTNOHAND zurück. Diese Fehlermeldung wird in der Routine ret_from_sys_call in ein -EINTR umgewandelt.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Systemruf personality Linux

Datei: kernel/exec_domain.c

```
#include <personality.h>

long sys_personality(unsigned long personality);
```

Der Linux-Kern unterstützt mehrere Ausführungsumgebungen, *Exec-Domain* genannt. Der Kern legt beim Hochfahren die erste Exec-Domain (gefüllt mit den Linux-spezifischen Daten) an; alle anderen können per Modul nachgeladen werden. Eine Domain hat folgende Struktur:

```
struct exec_domain {
  char *name;
  lcall7_func handler;
  unsigned char pers_low, pers_high;
  unsigned long * signal_map;
  unsigned long * signal_invmap;
  struct module * module;
  struct exec_domain *next;
};
```

Die Werte `pers_low` und `pers_high` sind nicht der höher- und niederbytige Wert der `personality`, sondern stellen eine (numerische) Unter- und Obergrenze für das in `personality` liegende Betriebssystem dar. Der Parameter `personality` gliedert sich in zwei Bereiche. Im oberen Wort stehen Flags für bekannte Bugs[6] und im unteren Wort das Betriebssystem. Die Werte dafür sind in der Headerdatei `<linux/personality.h>` zu finden.

Mittels `sys_personality()` kann gezielt eine bestimmte Domain eingestellt oder die aktuelle abgefragt werden. Sind im Aufruf alle Bits in `personality` gesetzt (`0xffffffff`), wird nur der aktuelle Wert zurückgegeben, ansonsten wird eine passende Domain gesucht. Das ist die Domain, bei der die unteren zwei Bytes der `personality` zwischen dem `low`- und dem `high`-Wert der Domain liegen. Wird keine Domain gefunden, gibt der Kern eine Fehlermeldung mit dem Level `KERN_ERR` aus.

Die Domain wird, zusammen mit `personality` in die Taskstruktur des aktuellen Prozesses eingetragen. Ist eine der Domains als Modul implementiert, wird der Zähler des Moduls (`usecount`) bei der alten Domain dekrementiert und bei der neuen inkrementiert. Zurückgegeben wird der alte Wert von `personality`.

Fehler

EINVAL — Es gibt keine zu `personality` passende Domain.

Systemruf prctl LINUX

Datei: kernel/sys.c

```
int sys_prctl(int option, unsigned long arg2, unsigned long arg3,
                    unsigned long arg4, unsigned long arg5)
```

Dieser Ruf ist zur Manipulation eines Prozesses vorgesehen. Die genaue Funktion hängt von `option` ab:

`PR_GET_PDEATHSIG` — gibt das `pdeath`-Signal des Prozesses zurück.

6 Korrekter ist es, nicht von Bugs, sondern von Eigenschaften der jeweiligen Betriebssysteme zu sprechen. Ein Beipiel ist das Flag `STICKY_TIMEOUTS`, siehe auch Seite 377.

PR_SET_PDEATHSIG — arg2 wird als pdeath-Signal des Prozesses eingetragen. Wie der Name schon sagt, bekommt ein Prozess dieses Signal, wenn sein Vater stirbt.

PR_GET_DUMPABLE — liefert das dumpable-Flag des Prozesses zurück.

PR_SET_DUMPABLE — arg2 wird als dumpable-Flag des Prozesses eingetragen.

Fehler

EINVAL — wenn ein ungültiger Wert übergeben wird.

EBADF — wenn ein ungültiger Deskriptor verwendet wird.

EMFILE — wenn bei sys_prctl() kein Deskriptor mehr frei ist.

Systemruf ptrace

Datei: arch/i396/kernel/ptrace.c

```
#include <linux/ptrace.h>

long sys_ptrace(long request, long pid, long addr, long data);
```

Mit Hilfe des Systemrufs *ptrace* kann ein Prozess die Abarbeitung eines anderen Prozesses kontrollieren. Der Systemruf wird zum Beispiel bei der Implementierung von Debug-Algorithmen verwendet. Ein Prozess, in dessen Taskstruktur das Flag PT_PTRACED gesetzt ist, wird bei einem Signal angehalten. Er stoppt, und sein Elternprozess wird über den Systemruf *wait* benachrichtigt. Der Speicher des gestoppten Prozesses kann dann gelesen und geschrieben werden. Der Elternprozess kann die Weiterarbeit des Kindprozesses veranlassen.

Als Erstes wird getestet, ob der Request P_TRACEME ist. Falls dieses zutrifft und falls das Bit PT_PTRACED in den Flags der Taskstruktur schon gesetzt ist, wird ein Fehler zurückgegeben. Andernfalls wird das Bit gesetzt und return 0 aufgerufen.

In pid wird die PID des gewünschten Prozesses angegeben. Selbstverständlich kann nicht jeder beliebige Prozess überwacht werden. Von init hat man generell die Finger zu lassen, weitere Bedingungen sind vom gewünschten request abhängig. Der Wert in request bestimmt die genaue Bedeutung des Aufrufs:

PTRACE_TRACEME — Wie bereits oben beschrieben wurde, setzt der Prozess das Flag PT_PTRACED. Der Elternprozess soll den Prozess überwachen. Sollte dieses Flag bereits gesetzt sein, kommt es zu einer Fehlermeldung.

PTRACE_ATTACH — setzt das Flag PT_PTRACED bei dem mit pid angegebenen Prozess. Dazu muss eine der folgenden Bedingungen erfüllt sein: pid ist nicht die PID des aktuellen Prozesses, die UID (GID) des aktuellen Prozesses muss mit der UID, EUID oder SUID (GID, EGID oder SGID) des gewünschten Prozesses übereinstimmen, oder

das Kind ist „willig“ (`dumpable`). Zusätzlich muss das Recht `CAP_SYS_PTRACE` gesetzt und darf das Flag `PF_TRACED` noch nicht gesetzt sein. Sind alle diese Hürden überstanden, wird das Flag gesetzt, der aktuelle Prozess wird Vater des Kindes und sendet ihm das Signal `SIGSTOP`.

`PTRACE_PEEKTEXT, PTRACE_PEEKDATA` — liest ein Wort (`long`) von der Adresse `addr`. Der Wert wird in `data` abgelegt und zurückgegeben. Bis jetzt wird noch nicht zwischen Text- und Datensegment unterschieden.

`PTRACE_PEEKUSR` — liest ein Wort von der Adresse `addr` aus der Struktur `user` des Prozesses. Es wird getestet, ob die Adresse auch wirklich innerhalb der Struktur liegt. Der Wert wird in `data` abgelegt und zurückgegeben.

`PTRACE_POKETEXT, PTRACE_POKEDATA` — schreibt den in `data` stehenden Wert an die Adresse `addr`.

`PTRACE_POKEUSR` — schreibt den in `data` stehenden Wert an die Adresse `addr` der `user`-Struktur. Hierbei wird sorgfältig darüber gewacht, dass keine Register oder Informationen aus der Taskstruktur überschrieben werden. Nur einige Debug-Register sind zugelassen.

`PTRACE_SYSCALL, PTRACE_CONT` — setzen den Ablauf des Kindprozesses fort. Bei `PTRACE_SYSCALL` wird das Flag `PT_TRACESYS` gesetzt. Dadurch stoppt der Ablauf nach dem `return` des nächsten Systemrufs (bei `PTRACE_CONT` wird dieses Flag gelöscht). Dann wird der Inhalt aus `data` in den Exit-Code des Kindes eingetragen, und das Kind wird aufgeweckt. Zuletzt wird das *Trap*-Flag[7] gelöscht.

`PTRACE_KILL` — weckt den Kindprozess auf, trägt ein `SIGKILL`-Signal als Exit-Code ein und löscht das Trap-Flag.

`PTRACE_SINGLESTEP` — Das Flag `PT_TRACESYS` wird gelöscht. Dafür wird das Trap-Flag gesetzt und `data` als Exit-Code eingetragen.

`PTRACE_DETACH` — Gibt den mit `PTRACE_ATTACH` gestoppten Prozess frei. Dabei werden die Flags `PF_TRACED` und `PT_TRACESYS` gelöscht, der Prozess wird wieder aufgeweckt, `data` als Exit-Code und der Originalvater wieder als Elternprozess eingetragen, und das Trap-Bit wird gelöscht.

`PTRACE_GETREGS` — Der Inhalt der Register (EBX, ECX, EDX, ESI, EDI, EBP, EAX, DS, ES, FS, GS, ORIG_EAX, EIP, CS, EFLAGS, ESP und SS) wird nacheinander an die Adresse `data` geschrieben.

`PTRACE_SETREGS` — Ab der Adresse `data` werden 17 `long`-Werte gelesen und in die Register des Prozesses geschrieben. Um größere Manipulationen zu verhindern, werden die Werte einer Kontrolle unterzogen.

7 Das Flag (auch Single-Step genannt) steht im EFlags-Register des Prozessors. Wenn es gesetzt ist, und ein SIGTRAP an den kontrollierten Prozess gesendet wird, führt dieser genau eine Anweisung aus.

ORIG_EAX — Dieses Register darf nicht gesetzt werden.

FS,GS — Der Wert darf nicht 0 sein, und die beiden unteren Bits dürfen nicht gesetzt werden.

DS,ES — Der Wert darf nicht 0 sein, und die beiden unteren Bits dürfen nicht gesetzt werden. Bits über 0xffff werden ausgeblendet.

SS,CS — Die unteren beiden Bits dürfen nicht gesetzt werden. Bits über 0xffff werden ausgeblendet.

EFLAGS — Vor dem Schreiben werden die Bits über `FLAG_MASK` ausgeblendet, und der Wert wird mit den bereits gesetzten Flags überlagert (ODER-Verknüpfung).

PTRACE_GETFPREGS — Der Inhalt der Register der FPU (`user_i387_struct`) wird an die Adresse `data` geschrieben. Sollte der zu überwachende Prozess keine FPU benutzen, wird eine leere FPU simuliert[8]. Die Werte werden aus der Union `i387_union` der Thread-Struktur gelesen.

PTRACE_SETFPREGS — Die Struktur `user_i387_struct` wird von der Adresse `data` gelesen und in die Union `i387_union` der Thread-Struktur geschrieben.

PTRACE_GETFPXREGS — Hierbei wird die Struktur `user_fxsr_struct` verwendet, ansonsten entspricht diese Angabe dem Request `PTRACE_GETFPREGS`.

PTRACE_SETFPXREGS — Die Struktur `user_fxsr_struct` wird von der Adresse `data` gelesen und in die Union `i387_union` der Thread-Struktur geschrieben.

Implementierung

Da bei den Peek-Rufen der `data`-Wert nicht belegt werden muss, aber der Parameter mit auf den Stack (für den Interrupt) gelegt wird, sorgt die C-Bibliothek für einen sicheren Pointer, indem sie einen Dummy-Wert auf den Stack legt.

```
int ptrace(int request, int pid, int addr, int data)
{
   long ret; long res;
   if (request > 0 && request < 4) (long *)data = &ret;
      __asm__ volatile ("int $0x80"
                :"=a" (res)
                :"0" (SYS_ptrace),"b" (request), "c" (pid),
                 "d" (addr), "S" (data));
   if (res >= 0) {
      if (request > 0 && request < 4) {
         errno = 0; return (ret);
      } return (int) res;
   }
   errno = -res; return -1;
}
```

8 Für die Register CWD, SWD und TWD werden die Werte 0xffff037f, 0xffff0000 und 0xffffffff eingetragen.

Fehler

EPERM — wenn für den mit `pid` angegebenen Prozess kein `sys_ptrace()` ausgeführt werden kann oder er schon überwacht wird.

ESRCH — wenn der Prozess `pid` nicht existiert bzw. `PT_PTRACED` nicht gesetzt ist.

EIO — wenn ein ungültiger Wert für `request` angegeben wird.

Systemruf reboot LINUX

Datei: kernel/sys.c

```
long sys_reboot(int magic1, int magic2, int cmd, void * arg);
```

`sys_reboot()` bootet das System oder schaltet die Möglichkeit um, mit der Tastenkombination CTRL+ALT+DEL zu booten. Für die Parameter `magic1` und `magic2` werden nur zwei Werte akzeptiert, und zwar muss `magic1` mit `0xfee1dead` und `magic2` mit `672274793`, `85072278` oder `369367448`[9] belegt sein. Die genaue Funktion hängt von `cmd` ab. Ist `cmd` gleich

`LINUX_REBOOT_CMD_RESTART` dann wird für alle Prozesse, die sich in die Liste `reboot_notifier_list` eingetragen haben, `notify_call()` mit `SYS_RESTART` als Parameter aufgerufen. Anschließend wird die architekturabhängige Funktion `machine_restart()` aufgerufen, die den Rechner rebootet.

`LINUX_REBOOT_CMD_HALT` dann wird für alle Prozesse, die sich in die Liste `reboot_notifier_list` eingetragen haben, die Funktion `notify_call()` mit `SYS_HALT` als Parameter aufgerufen. Anschließend wird die architekturabhängige Funktion `machine_halt()` aufgerufen.

`LINUX_REBOOT_CAD_ON` wird das Booten mit CTRL+ALT+DEL ermöglicht.

`LINUX_REBOOT_CAD_OFF` wird das Booten mit CTRL+ALT+DEL abgeschaltet.

`LINUX_REBOOT_POWER_OFF` dann wird für alle Prozesse, die sich in die Liste `reboot_notifier_list` eingetragen haben, die Funktion `notify_call()` mit `SYS_POWER_OFF` als Parameter aufgerufen. Anschließend wird die architekturabhängige Funktion `machine_power_off()` aufgerufen. Ist das *Advanced Power Management* konfiguriert, wird es hier benutzt.

`LINUX_REBOOT_RESTART2` dann wird für alle Prozesse, die sich in die Liste `reboot_notifier_list` eingetragen haben, die Funktion `notify_call()` mit `SYS_RESTART` als Parameter aufgerufen. Anschließend wird die architekturabhängige Funktion `machine_restart()` aufgerufen, die den Rechner rebootet. Vorgesehen

9 Wenn ihnen diese Zahlen etwas seltsam vorkommen, sollten sie sie hexadezimal betrachten.

ist, dass an der Adresse `arg` ein Kommando steht, das beim Neustart des Systemes ausgeführt wird. Bei Intel-Maschinen ist diese Funktionalität jedoch nicht implementiert.

Beachten Sie, dass `sys_reboot()` nicht `sys_sync()` aufruft!

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EINVAL — wenn ein ungültiger Wert für einen der Parameter angegeben wurde.

EPERM — wenn ein nicht privilegierter Prozess die Funktion aufruft.

Systemruf	**rt_sigreturn**	**rt_sigaction**	LINUX
	rt_sigprocmask	**rt_sigpending**	
	rt_sigtimedwait	**rt_sigqueueinfo**	
	rt_sigsuspend		

Datei: arch/i386/kernel/signal.c

kernel/signal.c

```
#include <linux/signal.h>
long sys_rt_sigreturn(unsigned long __unused);
long sys_rt_sigaction(int sig, const struct sigaction *act,
                struct sigaction *oact, size_t sigsetsize);
long sys_rt_sigprocmask(int how, sigset_t *set,
                sigset_t *oset, size_t sigsetsize);
long sys_rt_sigpending(sigset_t *set, size_t sigsetsize);
long sys_rt_sigtimedwait(const sigset_t *uthese, siginfo_t *uinfo,
                const struct timespec *uts, size_t sigsetsize);
long sys_rt_sigqueueinfo(int pid, int sig, siginfo_t *uinfo);
long sys_rt_sigsuspend(sigset_t *unewset, size_t sigsetsize);
```

Diese Systemrufe sind für die Behandlung von Realtime-Signalen vorgesehen. Je nach Art des Systemrufs gibt es mehr oder weniger Unterschiede zu den „normalen“ Versionen. Der wichtigste Unterschied ist, dass die neuen Funktionen mit einem Signalumfang von 64 Signalen arbeiten. Laut POSIX können bei diesen Signalen zusätzliche Informationen übermittelt werden, so z.B. die PID des sendenden Prozesses. Diese Daten werden in der Strucktur `siginfo` gehalten:

```
typedef struct siginfo {
  int si_signo; int si_errno; int si_code;
```

```
  union {
    int _pad[SI_PAD_SIZE];
    struct {          /* für SIGKILL                    */
      pid_t _pid;     /* PID des Senders                */
      uid_t _uid;     /* UID des Senders                */
    } _kill;
    struct {                /* für POSIX-Timer          */
      unsigned int _timer1;
      unsigned int _timer2;
    } _timer;
    struct {            /* für POSIX-Signale            */
      pid_t _pid;       /* PID des Senders              */
      uid_t _uid;       /* UID des Senders              */
      sigval_t _sigval; /* Signalinformationen          */
    } _rt;
    struct {            /* für SIGCHLD                  */
      pid_t _pid;       /* PID des Kindes               */
      int _status;      /* Exitcode                     */
      clock_t _utime;   /* Zeit im Usermodus            */
      clock_t _stime;   /* Zeit im Systemmodus          */
    } _sigchld;
    struct {            /* für SIGFPE, SIGSEGV, SIGBUS */
      void *_addr;      /* Adresse                      */
    } _sigfault;
    struct {            /* für SIGPOLL                  */
      int _band;        /* POLL_IN, POLL_OUT, POLL_MSG */
      int _fd;          /* Dateideskriptor              */
    } _sigpoll;
  } _sifields;
} siginfo_t;
```

Bei `rt_sigaction()`, `rt_sigpending()` und `rt_sigsuspend()` gibt es keine Unterschiede zu den alten Funktionen.

Die Funktion `rt_sigreturn()` arbeit wie das „normale“ `sigreturn()`, nur dass die Werte `sas_ss_sp` und `sas_ss_size` des angelegten Frames in die Taskstruktur übernommen werden.

In `rt_sigprocmask()` sind nur `SIG_BLOCK` und `SIG_UNBLOCK` als Funktionen implementiert, ein `SIG_SETMASK` hat keinen Effekt.

Für `rt_sigtimedwait()` gibt es keine Entsprechung. Hiermit kann ein Prozess auf das Eintreffen eines Signals aus einem Set warten. Das Set wird in `uthese` angegeben und die Wartedauer in `uts`. Beim Eintreffen eines Signals wird seine Nummer zurückgegeben, und in `uinfo` wird die Information aus der Signalqueue für dieses Signal gespeichert. Läuft der Timer ohne Unterbrechung ab, gibt die Funktion `-EAGAIN`, ansonsten `-EINTR` zurück.

Neu ist ebenfalls die Funktion `rt_sigqueueinfo()`, die ein Realtime-Signal an einen Prozess sendet. Dabei ist `pid` der Zielprozess, `sig` das Signal, und `uinfo` sind die zu-

sätzlichen Daten, die an `pid` weitergegeben werden. Die Funktion testet, ob in `uinfo` der Wert `si_code` kleiner 0 ist, denn niemand darf Signale „als Kern“ versenden.

Fehler

EFAULT — wenn das Kopieren einer Struktur in oder aus dem User-Adreßraum fehlschlug.

EINTR — wenn der Prozess bei `sys_rt_sigtimedwait()` durch ein nicht blockiertes Signal unterbrochen wurde.

EPERM — wenn bei `rt_sigqueueinfo()` ein falscher Signalcode verwendet wurde.

Systemruf	**sched_getparam**	**sched_getscheduler**	LINUX
	sched_setparam	**sched_setscheduler**	

Datei: kernel/sched.c

```
long sys_sched_getparam(pid_t pid, struct sched_param *param);
long sys_sched_setparam(pid_t pid, struct sched_param *param);
long sys_sched_getscheduler(pid_t pid);
long sys_sched_setscheduler(pid_t pid, int policy,
        struct sched_param *param);
```

Ein Prozess kann seine Behandlung durch den Scheduler steuern. Dazu gibt es Parameter (bis jetzt nur einer), die in einer Struktur zusammengefasst werden:

```
struct sched_param { int sched_priority; };
```

Die Funktion `sys_sched_getparam()` gibt die Basispriorität für Realtime-Prozesse (`rt_priority`) des Prozesses `pid` in der Struktur `param` zurück.

`sys_sched_setparam()` trägt den übergebenen Wert als `rt_priority` für den Prozess `pid` ein und ruft den Scheduler auf. Der Wert muss zwischen 0 und 99 liegen. Ein nicht berechtigter Prozess (`CAP_SYS_NICE`) darf nur seine eigene Priorität verändern.

Die Funktion `sys_sched_getscheduler()` gibt die Scheduler-Taktik des Prozesses zurück. Der Scheduler kennt drei Taktiken:

`SCHED_OTHER` — Die `rt_priority` dieser Prozesse ist 0. Damit erhalten sie bei der Neuberechnung der Priorität einen normalen Wert.

`SCHED_FIFO` — Die kleinen zeitkritischen Prozesse. Sie erhalten einen Prioritätszuschlag von 1.000.

`SCHED_RR` — Die großen, zeitkritischen Prozesse. Sie werden, wenn ihr `counter` abgelaufen ist, im Scheduler in der Prozessliste ganz hinten einsortiert.

Mit `sys_sched_setscheduler()` kann ein Prozess seine Taktik und seinen Wert `rt_priority` ändern. Ist `policy` negativ, wird der alte Wert behalten. Die Priorität muss außerdem der Taktik entsprechen, und unberechtigte Prozesse dürfen nur ihre eigenen Daten ändern.

Fehler

EPERM — wenn ein normaler Prozess einen anderen Prozess ändern will.

ESRCH — wenn der Prozess `pid` nicht gefunden wurde.

EINVAL — wenn ein übergebener Parameter ungültig ist.

Systemruf	**sched_rr_get_interval**	**sched_yield**	LINUX
	sched_get_priority_max		
	sched_get_priority_min		

Datei: kernel/sched.c

```
long sys_sched_get_priority_min(int policy);
long sys_sched_get_priority_max(int policy);
long sys_sched_yield(void);
long sys_sched_rr_get_interval(pid_t pid,
        struct timespec *interval)
```

Die `get_priority`-Funktionen geben die Untergrenze bzw. die Obergrenze für die `rt_priority`-Werte der einzelnen Scheduler-Taktiken zurück.

Mit `sys_sched_yield()` kann ein Prozess sich in sein Schicksal ergeben. Er trägt in seinen Policy-Wert das Flag `SCHED_YIELD` ein und wird an das Ende der Liste der laufenden Prozesse einsortiert und entsprechend vom Scheduler behandelt.

Die Funktion `sys_sched_rr_get_interval()` liefert im Moment noch den festen Wert von 150 Millisekunden.

Fehler

EINVAL — wenn eine falsche Tatik übergeben wurde.

Systemruf	**setdomainname**

Datei: kernel/sys.c

```
long sys_setdomainname(const char *name, int len);
```

Die Funktion `sys_setdomainname()` überschreibt den Domainnamen mit dem in `name` angegebenen Namen. Der Name muss kein Nullbyte am Ende haben, dieses trägt die Funktion selbst ein.

Implementierung

Der Systemruf `setdomainname` wird über das Syscall-Makro umgesetzt. Der Ruf *getdomainname* ist in der C-Bibliothek implementiert. `__uname()` wird aufgerufen, und der Domainname wird ausgelesen. Der Rückkehrwert ist (bei Erfolg) 0.

```
int getdomainname(char *name, size_t len)
{
   struct utsname uts;

   if (name == NULL) {
      errno = EINVAL; return -1;
   }
   if (__uname(&uts) == -1) return -1;
   if (strlen(uts.domainname)+1 > len) {
      errno = EINVAL; return -1;
   }
   strcpy(name, uts.domainname);
   return 0;
}
```

Fehler

EINVAL — wenn bei `getdomainname()` der von `sys_uname()` gelieferte String auf NULL zeigt oder größer als `len` ist oder wenn bei `sys_setdomainname()` `len` zu groß ist.

EPERM — wenn ein nicht privilegierter Nutzer `sys_setdomainname()` aufruft.

Systemruf	**getgroups**	POSIX
	setgroups	

Datei: kernel/sys.c

```
#include <linux/types.h>

long sys_getgroups(int len, gid_t *groups);
long sys_setgroups(int len, gid_t *groups);
```

Die Funktionen `sys_getgroups()` und `sys_setgroups()` erlauben das Lesen und Setzen von mehreren Gruppenrechten für einen Prozess. Die Gruppen sind ein Teil der Taskstruktur (siehe Abschnitt 3.1.1).

`sys_getgroups()` liefert die Gruppen, wobei die Bedeutung des Parameters etwas unlogisch ist. Wenn `len` 0 ist, wird nur die Anzahl der Gruppen zurückgegeben. Ist `len` kleiner 0 oder kleiner als die Anzahl der Gruppen, liefert der Aufruf ein `EINVAL`. Ansonsten werden die Gruppen des Prozesses an die Adresse `groups` geschrieben.

`sys_setgroups()` setzt Gruppenrechte. Es können nur alle Gruppen auf einmal gesetzt werden, da die alten Gruppen dabei überschrieben werden. `len` gibt die Anzahl der Gruppen an, die an der Adresse `groups` stehen. Diesen Aufruf darf nur ein Prozess mit dem Recht `CAP_SETGID` ausführen.

Implementierung

Beide Systemrufe werden über das Syscall-Makro umgesetzt.

Fehler

EINVAL — wenn bei `sys_setgroups()` der Wert `len` größer als NGROUPS ist.

EPERM — wenn ein nicht privilegierter Prozess `sys_setgroups()` aufruft.

Systemruf	**sethostname** **gethostname**	4.3+BSD

Datei: kernel/sys.c

```
long sys_sethostname(char *name, int len);
long sys_gethostname(char *name, int len);
```

Mit diesen Funktionen hat man Zugriff auf den Rechnernamen. Die erste Funktion, `sys_sethostname()`, trägt `name` als Hostnamen (`nodename`) des Rechners ein und kann nur von einem Prozess mit dem Recht `CAP_SYS_ADMIN` ausführt werden. Bei Erfolg wird 0 zurückgegeben. `sys_gethostname()` liest den `nodename` des Rechners aus und gibt ihn in `name` zurück.

Implementierung

Der Systemruf *sethostname* wird über das Syscall-Makro umgesetzt. Der Systemruf *gethostname* ist für Intel-Systeme in der C-Bibliothek unter Verwendung von `__uname()` implementiert.

```
int gethostname(char *name, size_t len)
{
   struct utsname uts;

   if (name == NULL)
   {
      errno = EINVAL; return -1;
```

```
    }

    if (__uname(&uts) == -1) return -1;

    if (strlen(uts.nodename)+1 > len)
    {
       errno = EINVAL; return -1;
    }
    strcpy(name, uts.nodename);
    return 0;
}
```

Fehler

EINVAL — wenn bei `sys_sethostname()` der String `name` auf NULL zeigt oder die in `len` angegebene Größe `__NEW_UTS_LEN` übersteigt. Der Wert ist in `<linux/utsname.h>` mit 64 definiert.

EPERM — wenn ein nicht berechtigter Prozess `sys_sethostname()` aufgerufen hat.

Systemruf	**getitimer**
	setitimer

Datei: kernel/itimer.c

```
#include <linux/time.h>

long sys_getitimer(int which, struct itimerval *value);
long sys_setitimer(int which, const  struct  itimerval  *value,
           struct itimerval *ovalue);
```

Diese Funktionen ermöglichen eine bessere Zeitüberwachung eines Prozesses als `sys_alarm()`. Für den aktuellen Prozess können drei spezielle Timer programmiert werden, die durch `which` festgelegt werden:

`ITIMER_REAL` — bezieht sich auf die reale Zeit. Der Alarm wird jedes Mal aktualisiert, wenn der Prozess im Scheduler angestoßen wird, und liefert das Signal `SIGALRM`, wenn er abgelaufen ist.

`ITIMER_VIRTUAL` — ist die Zeit, in der der Prozess aktiv ist, sich aber nicht in einem Systemruf (Systemmodus) befindet. Der Alarm wird in der Routine `do_timer()` aktualisiert und liefert bei Ablauf das Signal `SIGVTALRM`.

`ITIMER_PROF` — bezeichnet die gesamte Zeit, die der Prozess läuft. Nach Ablauf des Alarms wird das Signal `SIGPROF` gesendet. Zusammen mit dem vorherigen Timer ermöglicht das eine Unterscheidung zwischen der im Systemmodus und der im Nutzermodus verbrauchten Zeit.

Die Zeiten werden mit folgender Struktur angegeben:

```
struct itimerval {
        struct timeval it_interval; /* Intervall  */
        struct timeval it_value;    /* Startwert  */
};

struct timeval {
        long tv_sec; long tv_usec;
};
```

`sys_getitimer()` gibt den aktuellen Wert für den in `which` gesetzten Alarm zurück. `sys_setitimer()` setzt den mit `which` angegebenen Alarm auf den Wert `value`. Der alte Wert wird in `ovalue` zurückgegeben. Beim ersten Starten wird der Timer auf den Wert `it_value` gesetzt. Ist der Timer abgelaufen, wird ein Signal erzeugt und der Alarm wieder neu gesetzt — ab dann auf den Wert `it_intervall`, wie in Abschnitt 3.2.5 beschrieben. Der Alarm wird möglicherweise etwas später als die angegebene Zeit ausgelöst, das hängt vom Systemtakt ab. Die übliche Verzögerung beträgt zehn Millisekunden.

Unter Linux erfolgen die Erzeugung und das Versenden von Signalen getrennt. So ist es möglich, das bei *pathologisch* schwerster Auslastung ein `SIGALRM` gesendet wird, bevor der Prozess das Signal des vorherigen Ablaufs erhalten hat. Das zweite Signal wird dann ignoriert.

Implementierung

Beide Systemrufe werden über das Syscall-Makro umgesetzt.

Fehler

EFAULT — wenn `value` oder `ovalue` ein ungültiger Zeiger ist.

EINVAL — wenn `which` ungültig ist.

Systemruf	**getrlimit**	4.3+BSD
	setrlimit	
	getrusage	

Datei: kernel/sys.c

```
#include <linux/resource.h>

long sys_getrlimit(unsigned int resource, struct rlimit *rlim);
long sys_setrlimit(unsigned int resource, struct rlimit *rlim);
long sys_getrusage(int who, struct rusage *usage);
```

`sys_getrlimit()` liest die Größe einer Ressource des aktuellen Prozesses und speichert sie in `rlim` ab. Das Setzen ist mit der Funktion `setrlimit()` möglich. Die Struktur `rlimit` ist in `linux/ressource.h` definiert:

```
struct rlimit {
        int  rlim_cur; /* Softlimit  */
        int  rlim_max; /* Hardlimit  */
};
```

Es gibt zwei Grenzen für einen Prozess, das *Softlimit* (die aktuelle Grenze) und das *Hardlimit* (die Obergrenze). Ein nicht privilegierter Prozess kann sein Softlimit auf einen Wert zwischen null und dem Hardlimit einstellen und das Hardlimit bis auf das Softlimit senken. Eine Verkleinerung des Hardlimits ist endgültig. Ein Prozess mit dem Recht `CAP_SYS_RESOURCE` kann die Limits (bis auf `RLIMIT_NOFILE`) beliebig setzen, denn für den Wert `RLIMIT_NOFILE` ist `NR_OPEN` die maximale Obergrenze. Als `resource` sind in `<asm/resource.h>` folgende Werte definiert (in Klammern stehen die Initialisierungen für das Soft- und Hardlimit) :

`RLIMIT_CPU` — die maximale CPU-Zeit (Summe von `utime` und `stime` des Prozesses) in Millisekunden (`LONG_MAX, LONG_MAX`)

`RLIMIT_FSIZE` — die maximale Dateigröße (`LONG_MAX, LONG_MAX`)

`RLIMIT_DATA` — die maximale Größe des benutzten Datensegments, initialisiert mit (`LONG_MAX, LONG_MAX`)

`RLIMIT_STACK` — die maximale Stackgröße (`_STK_LIM, _STK_LIM`)

`RLIMIT_CORE` — die maximale Größe einer `core`-Datei (`0, LONG_MAX`)

`RLIMIT_RSS` — der maximale Speicherplatz für Argumente und Umgebung (RSS) (`LONG_MAX, LONG_MAX`)

`RLIMIT_NPROC` — die maximale Anzahl der Kindprozesse, die Anfangsgrenzen sind (`MAX_TASKS_PER_USER, MAX_TASKS_PER_USER`)

`RLIMIT_NOFILE` — die max. Anzahl der offenen Dateien (`NR_OPEN, NR_OPEN`)

`RLIMIT_MEMLOCK` — der maximale Speicherplatz, den ein Prozess blockieren kann (`LONG_MAX, LONG_MAX`)

`RLIMIT_AS` — der maximale Adressraum (`LONG_MAX, LONG_MAX`)

Hat eine Ressource den Wert `RLIM_INFINITY`, so gibt es keine Beschränkung. Ein Prozess, der sein aktuelles Softlimit überschreitet, wird abgebrochen. Beide Rufe geben bei erfolgreicher Ausführung 0 zurück.

Während die obigen Funktionen die Beschränkungen verwalten, in deren Grenzen ein Prozess die Ressourcen des Systems nutzen kann, liefert die Funktion `sys_getrusage()` Informationen über deren aktuelle Nutzung. Die einzelnen Werte sind in der Struktur `rusage` definiert:

```
struct    rusage {
        struct timeval ru_utime; /* Nutzerzeit              */
        struct timeval ru_stime; /* Systemzeit              */
        long ru_maxrss;          /* maximaler RSS           */
        long ru_ixrss;      /* Größe des geteilten RSS      */
        long ru_idrss;      /* Größe des ungeteilten RSS    */
        long ru_isrss;      /* Größe des Stacks             */
        long ru_minflt;     /* Anzahl der Minor-Faults      */
        long ru_majflt;     /* Anzahl der Major-Faults      */
        long ru_nswap;      /* Swap-Operationen             */
        long ru_inblock;    /* Block-Eingabe-Operationen    */
        long ru_oublock;    /* Block-Ausgabe-Operationen    */
        long ru_msgsnd;     /* gesendete Nachrichten        */
        long ru_msgrcv;     /* empfangene Nachrichten       */
        long ru_nsignals;   /* empfangene Signale           */
        long ru_nvcsw;      /* freiwillige Kontextwechsel   */
        long ru_nivcsw;     /* unfreiwillige Kontextwechsel */
};
```

Die Funktion füllt aber nicht die komplette Struktur. Nur die Werte für `ru_utime` und `ru_stime` und die Angaben zu den Speicherseiten (Minor-Faults, Major-Faults und Swap-Operationen) werden belegt. Wenn der Wert `RUSAGE_SELF` in `who` steht, beziehen sich die Informationen auf den Prozess selbst. Die Daten der Kindprozesse erhält man durch `RUSAGE_CHILDREN`. Alle anderen Werte für `who` liefern die Summe aus beiden Werten.

Implementierung

Beide Systemrufe werden über das Syscall-Makro umgesetzt.

Fehler

EINVAL — wenn die `limit`-Funktionen mit einem für `resource` ungültigen Wert aufgerufen werden oder `who` bei `sys_getrusage()` ungültig ist.

EPERM — wenn ein nicht privilegierter Benutzer `sys_setrlimit()` aufruft.

Systemruf sigaltstack — POSIX

Datei: arch/i386/kernel/signal.c
kernel/signal.c

```
#include <linux/signal.h>

int sys_sigaltstack(const stack_t *uss, stack_t *uoss);
```

Diese Funktion ermöglicht es den Signalstack eines Prozesses zu manipulieren. Ist `uoss` ungleich NULL, werden darin die alten Daten zurückgegeben. Die Strucktur `stack_t` hat folgenden Aufbau:

```
typedef struct sigaltstack {
        void *ss_sp;    /* Zeiger auf den Stack */
        int ss_flags;   /* Flags                */
        size_t ss_size; /* Größe des Stacks     */
} stack_t;
```

Als Flags sind zwei Werte möglich:

SS_ONSTACK — Der Stackpointer und die Stackgröße werden in die Taskstruktur eingetragen. Wichtig ist dabei, dass der neue Zeiger nicht in den alten Stack zeigen darf und der neue Stack größer als `MINSIGSTKSZ` sein muss.

SS_DISABLE — Als Zeiger wird NULL und als Größe wird 0 eingetragen. `uss` darf aber nicht NULL sein.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EFAULT — wenn ein Kopieren der Werte fehlschlug.

EINVAL — wenn `SS_ONSTACK` gesetzt ist.

ENOMEM — wenn die neue Stackgröße zu klein ist.

EPERM — wenn der neue Stackzeiger innerhalb des alten Stacks liegt.

Systemruf	**signal**	**sigaction**	POSIX
	sigpending	**sigsuspend**	
	sgetmask	**ssetmask**	
	sigprocmask	**sigreturn**	

Datei: kernel/signal.c
arch/i386/kernel/signal.c

```
#include <signal.h>

unsigned long sys_signal(int signum,  __sighandler_t handler);
int sys_sigaction(int signum, const struct old_sigaction *new,
        struct old_sigaction *old);
long sys_sgetmask(void);
```

```
long sys_ssetmask(int newmask);
long sys_sigpending(sigset_t *buf);
int sys_sigsuspend(int restart, unsigned long oldmask,
        unsigned long set);
long sys_sigprocmask (int how, const sigset_t *set,
        sigset_t *old_set);
int sys_sigreturn(unsigned long __unused);
```

Die Funktion `sys_signal()` setzt die Behandlungsroutine des Signals `signum`. Die Routine `handler` kann eine selbst definierte Funktion oder ein Makro aus `<signal.h>` sein. Möglich sind dabei:

`SIG_DFL` — Es erfolgt die Standardbehandlung des Signals.

`SIG_IGN` — Das Signal wird ignoriert.

Die Behandlungsroutine wird in die `sigaction`-Struktur des aktuellen Prozesses eingetragen. Dabei werden die Flags `SA_ONESHOT` und `SA_NOMASK` gesetzt und alle anderen Werte mit 0 initialisiert. Bei Erfolg wird die Adresse der alten Routine zurückgegeben, sonst ein negativer Wert (−1). Für die Signale `SIGKILL` und `SIGSTOP` können keine neuen Handler implementiert werden, und die Signalnummer muss kleiner als 32 sein (im Quelltext festgelegt).

Nach POSIX 3.3.1.3 gilt dabei: Wird als Routine `SIG_IGN` angegeben, wird ein eventuell noch vorliegendes Signal gelöscht (außer `SIGCHLD`). Bei der Routine `SIG_DFL` wird das Signal gelöscht, wenn es sich nicht um `SIGCONT`, `SIGCHLD` oder `SIGWINCH` handelt. In beiden Fällen ist es ohne Bedeutung, ob das Signal blockiert wird oder nicht. Das übernimmt die Funktion `recalc_sigpending()`, die nach dem Eintragen der neuen Routine aufgerufen wird.

Die Funktion `sys_sigaction()` ist die moderne und erweiterte Version von `sys_signal()` (beide benutzen `do_sigaction()`). Sie wird verwendet, um über eine Struktur das Verhalten des Prozesses beim Eintreffen des Signals genauer festzulegen. In `new` steht die neue Struktur. Ist `old` ungleich `NULL`, wird darin die alte Struktur zurückgegeben. Die Struktur `sigaction` ist wie folgt definiert:

```
struct sigaction {
  __sighandler_t sa_handler; /* Signalbehandlungsroutine      */
  old_sigset_t sa_mask;      /* Maske der blockierten Signale */
  unsigned long sa_flags;    /* Flags                         */
  void (*sa_restorer)(void); /* Routine, die nach sa_handler  *
                              * aufgerufen werden soll        */
};
```

Die Flags haben folgende Bedeutung:

SA_NOCLDSTOP — Bei einem Signal `SIGSTOP` wird der Vater nicht durch ein Signal `SIGCHLD` benachrichtigt.

SA_SIGINFO — Für die Rückkehr aus der Routine wird ein Stack mit der Struktur `rt_sigframe` statt `sig_frame` benutzt.

SA_ONSTACK — Als Rückkehrstack wird der Signalstack der Taskstruktur verwendet.

SA_RESTART — Wurde durch das Eintreffen des Signals ein Systemruf unterbrochen (`regs->eax == ERESTARTSYS`), wird `EINTR` zurückgeliefert.

SA_NODEFER — Nach dem ersten Signal `signum` wird dieses Bit in der Maske `sa_mask` gesetzt.

SA_RESETHAND — Nach dem Eintreffen des Signals wird der Handler auf die Routine `SIG_DFL` zurückgesetzt.

SA_NOMASK — wie `SA_NODEFER`

SA_ONESHOT — wie `SA_RESETHAND`

SA_RESTORER — Als Rückkehrfunktion wird `sa_restorer()` verwendet.

Für das einfache Setzen und Abfragen der Signalmaske der blockierten Signale gibt es die Funktionen `sys_sgetmask()` und `sys_ssetmask()`. Die erste Funktion gibt die Maske der ersten 32 Signale[10] (`current->blocked.sig[0]`) zurück, und die zweite löscht die Signale `SIGKILL` und `SIGSTOP` aus der übergebenden Maske und trägt sie in die Taskstruktur (`blocked.sig[0]`) ein.

`sys_sigpending()` fragt ab, ob blockierte Signale für den Prozess vorliegen. Die Signale werden in `buf` abgelegt, der Rückgabewert ist 0.

Mit den Funktionen `sys_sgetmask()` und `sys_ssetmask()` kann das Blockieren von Signalen ein- und ausgeschaltet werden. `sys_sigsuspend()` erlaubt nun das Setzen einer Signalmaske und das Stoppen des Prozesses in einer einzelnen Aktion. Der Prozess wird eingeschläfert, bis ein nicht blockiertes Signal eintrifft.

Um die Maske der zu blockierenden Signale gezielt zu verändern, kann die Funktion `sys_sigprocmask` verwendet werden. Der Parameter `how` gibt an, wie die neue Signalmaske verwendet werden soll:

SIG_BLOCK — Die in der Signalmaske gesetzten Signale werden blockiert. Die neue Maske ergibt sich durch die Verknüpfung der alten und übergebenen Maske mittels `|=`.

SIG_UNBLOCK — Die in der Signalmaske gesetzten Signale werden gelöscht. Die neue Maske ergibt sich durch die Verknüpfung der alten und übergebenen Maske mittels `& = ~`.

SIG_SETMASK — Die Signalmaske wird als Signalmaske des aktuellen Prozesses übernommen.

Die Signalbits für `SIGKILL` und `SIGSTOP` löscht die Funktion vorher aus der übergebenen Maske. Falls `old_set` ungleich `NULL` ist, wird darin die alte Maske zurückgegeben.

10 Genauer gesagt werden die ersten `_NSIG_BPW` Signale berücksichtigt.

Die Funktion `sys_sigreturn()` organisiert die Rückkehr aus einer Signalunterbrechung. Sie wird intern aufgerufen, um nach einer Signalbehandlungsroutine wieder in den Systemmodus zu gelangen. Dazu wird für jedes zu behandelnde Signal ein Frame auf dem Stack des Prozesses angelegt, der dafür sorgt, dass der Systemruf *sigreturn* ausgelöst wird[11]. Die Funktion wird nicht aufgerufen, wenn bei *sigaction* ein Restore-Handler gesetzt ist.

Implementierung

Die Funktionen `sys_signal()`, `sys_sigprocmask()`, `sys_sgetmask()` und `sys_ssetmask` stehen auf Alpha-Maschinen nicht zur Verfügung. Die Bibliotheksfunktion `signal()` arbeitet seltsamerweise nicht mit dem Systemruf *signal*, sondern baut auf *sigaction* auf.

```
__sighandler_t signal (int sig, __sighandler_t handler)
{
   int ret;
   struct sigaction action, oaction;
   action.sa_handler = handler;
   __sigemptyset (&action.sa_mask);
   action.sa_flags = SA_ONESHOT | SA_NOMASK | SA_INTERRUPT;
   action.sa_flags &= ~SA_RESTART;
   ret = __sigaction (sig, &action, &oaction);
   return (ret == -1) ? SIG_ERR : oaction.sa_handler;
}
```

Die restlichen Rufe arbeiten mit dem Syscall-Makro, außer `sys_sigreturn()`: Für diese Funktion gibt es keine Schnittstelle nach außen.

Fehler

EINVAL — wenn eine ungültige Signalnummer verwendet wird.

EFAULT — wenn die Behandlungsroutine größer als die erlaubte Größe eines Prozesses (`TASK_SIZE`) ist.

EINTR — wenn der Prozess von `sys_sigsuspend()` zurückkehrt.

Systemruf sysctl

Datei: kernel/sysctl.c

```
#include <linux/sysctl.h>

int sys_sysctl(struct __sysctl_args *args);
```

11 Zu diesem Zweck werden alle Register und der Maschinencode (!), der *sigreturn* auslöst, als Frame auf den Stack gelegt.

In einigen Fällen ist es praktisch, ein standardisiertes Interface für Kernel- oder Systemvariablen zu haben, die zur Laufzeit eingestellt oder beobachtet werden müssen. Die Funktion `sys_sysctl()` ermöglicht eine umfangreiche Verwaltung von systemrelevanten Informationen. Diese Informationen werden in internen Tabellen gehalten und auch nach `/proc/sys` abgebildet. Für die Nutzung dieser Funktion muss jedoch während der Konfiguration des Kernels die Frage *Sysctl support?* mit Ja beantwortet werden.

Vor der eigentlichen Funktionsbeschreibung kommen noch einige Ausführungen zu den internen Datenstrukturen. Für jede Datei (und jedes Verzeichnis) in `/proc/sys` existiert eine Tabelle mit folgendem Aufbau:

```
struct ctl_table {
   int ctl_name;               /* binäre ID                */
   const char *procname;       /* Text ID für /proc/sys    */
   void *data;                 /* Datenbereich             */
   int maxlen;                 /* Größe des Datenbereichs  */
   mode_t mode;                /* Schreib-/Leserechte      */
   ctl_table *child;           /* nächste Tabelle          */
   proc_handler *proc_handler; /* Funktion für Textausgabe */
   ctl_handler *strategy;      /* Funktion für I/O         */
   struct proc_dir_entry *de;  /* /proc-Kontrollblock      */
   void *extra1;
   void *extra2;
};
```

Die Unterscheidung, ob ein `ctl_table`-Eintrag als Verzeichnis oder als Datei dargestellt wird, ist durch die Komponente `child` gegeben. Ist `child` gleich `NULL`, handelt es sich um einen Endknoten (Leaf-Node). Wird ein Pointer auf ein Feld von `ctl_table` übergeben, handelt es sich um ein Verzeichnis. So können Tabellenfelder hierarchisch verschachtelt werden. Ein NULL-Eintrag schließt ein Tabellenfeld ab.

Beim Zugriff auf eine in das *Proc*-Dateisystem gespiegelte Variable übernimmt eine Handler-Routine (`proc_handler`) die Formatierung bzw. die Auswertung der Daten. Die Strategie-Routine (`strategy`) wird bei allen Lese- und Schreibzugriffen aufgerufen und soll die korrekte Behandlung von Werteänderungen sichern. Wird zum Beispiel eine Cache-Größe verändert, kann diese Routine die Speicherplatzverwaltung übernehmen. Für die Strategie-Funktion gilt folgende Semantik: War deren Abarbeitung erfolgreich, wird null zurückgegeben und mit der normalen Abarbeitung fortgefahren. Bei einem anderen Wert wird dieser sofort zurückgegeben. Dabei bedeutet ein Wert kleiner als null einen Fehler, und ein Wert größer als null heißt: Die eigentliche Arbeit (das Lesen bzw. das Schreiben) wurde schon von der Strategie-Funktion durchgeführt.

Mit der `mode`-Komponente kann wie bei ganz normalen Dateien die Zugriffsberechtigung in der `rwxrwxrwx`-Notation eingestellt werden[12]. Im Netzwerkinterface (`net/core/sysctl_net_core.c`) werden so beispielsweise die Endknoten für einige Systemvariablen definiert:

12 So bedeutet `mode=0644` Schreibberechtigung für root; alle anderen Bits sind auf nur Lesen eingestellt.

```
#include <linux/sysctl.h>

extern __u32 sysctl_wmem_max;
extern __u32 sysctl_rmem_max;
…
ctl_table core_table[] = {
        {NET_CORE_WMEM_MAX, "wmem_max", &sysctl_wmem_max,
         sizeof(int), 0644, NULL, &proc_dointvec},
        {NET_CORE_RMEM_MAX, "rmem_max", &sysctl_rmem_max,
         sizeof(int), 0644, NULL, &proc_dointvec},
        …
        { 0 }
};
```

`proc_dointvec` ist die Standard-Behandlungsroutine für Integer-Variablen oder Integer-Felder. Hier könnten jedoch eigene Behandlungsroutinen eingebunden werden, die die Anzeige bzw. Abbildung bestimmter Modi oder Werte, wie beispielsweise „on“ oder „off“, übernehmen oder spezielle Datentypen abbilden.

Im übergeordneten Netzwerksystem werden so die verschiedenen Netzwerkparameter in Verzeichnissen zusammengefasst (`net/sysctl_net.c`):

```
#include <linux/sysctl.h>

extern ctl_table core_table[];

ctl_table net_table[] = {
        {NET_CORE,   "core",      NULL, 0, 0555, core_table},
        …
        {0}
};
```

Um diese Tabellen beim Kern an- bzw. abzumelden, stehen die folgenden Funktionen zur Verfügung:

```
struct ctl_table_header * register_sysctl_table(ctl_table * table,
                                  int insert_at_head);
void unregister_sysctl_table(struct ctl_table_header * table);
```

Die Tabellen bilden dynamische Listen. Beim Start des Systems sind acht[13] dieser Listen im Kern angelegt, und sie werden in `root_table_header` verwaltet.

`CTL_KERN` — Kern- und Kontrollstrukturen

`CTL_VM` — VM-Managment

`CTL_NET` — Netzwerk

`CTL_PROC` — Prozessinformationen

`CTL_FS` — Dateisysteme

13 Wurde kein Netzwerk konfiguriert, dann werden natürlich nur sieben Listen angelegt.

CTL_DEBUG — Debugging

CTL_DEV — Geräte

CTL_BUS — Informationen über den Bus (ISA etc.)

Nach der Erklärung der Datenstrukturen kommen wir nun zur genauen Wirkung der Funktion. Sie wird über die Struktur `__sysctl_args` gesteuert.

```
struct __sysctl_args {
          int *name;       /* Name der Information      */
          int nlen;        /* Bereich der Information   */
          void *oldval;    /* Zeiger auf den alten Wert */
          size_t *oldlenp; /* Länge des alten Wertes    */
          void *newval;    /* Zeiger auf den neuen Wert */
          size_t newlen;   /* Länge des neuen Wertes    */
          unsigned long __unused[4];
};
```

Der Parameter `name` gibt an, auf welche Information zugegriffen werden soll. Die dazu möglichen Werte stehen in `<linux/sysctl.h>`, hier folgt ein kleiner Auszug (mit dem Typ des Wertes):

KERN_OSTYPE — ein String: das Betriebssystem

KERN_OSRELEASE — ein String: die Version

KERN_VERSION — ein String: die Compile-Information

KERN_NODENAME — ein String: der Hostname

KERN_DOMAINNAME — ein String: der Domainname

VM_SWAPCTL — eine Struktur: die Parameter des Swap-Prozesses

VM_FREEPG — drei Zahlen: die Werte der Freie-Seiten-Stufen

VM_BDFLUSH — eine Struktur: die Parameter des `bd_flush()`-Prozesses

Die Funktion durchsucht alle Tabellen, bis sie die Tabelle mit der gesuchten ID (`name`) gefunden hat. Hat diese Tabelle einen Nachfolger und eine Strategie-Funktion, wird diese ausgeführt, ansonsten kommt die Defaultstrategie zur Anwendung. Diese überprüft zunächst die Zugriffsrechte (ist `oldval` ungleich null, ist das Leserecht für die Tabelle erforderlich; ist `newval` ungleich null ist das Schreibrecht erforderlich). Dann wird die tabelleneigene Strategie-Funktion ausgeführt (wenn eine definiert ist). Zum Schluss speichert die Funktion den alten Wert und dessen Größe in `oldval` und `oldlenp`, wenn beide ungleich 0 sind. Sind `newval` und `newlen` ungleich 0, wird der von `newval` adressierte Speicherbereich der Größe `newlen` als neuer Wert in die Tabelle eingetragen.

Implementierung

Die C-Bibliothek bietet keine Schnittstelle. Die Kernelfunktion testet auch noch keine Superuser-Rechte!

Fehler

EFAULT — wenn ein Kopieren der Werte fehlschlug.

ENOPERM — wenn Zugriffsrechte die Operation verbieten.

ENOTDIR — wenn keine Tabelle für `name` gefunden wurde.

Systemruf sysinfo LINUX

Datei: kernel/info.c

```
#include <linux/sys.h>
#include <linux/kernel.h>

int sys_sysinfo(struct sysinfo *info);
```

`sys_sysinfo()` liefert Informationen über die Auslastung des Systems. Die Daten werden in folgender Struktur zurückgegeben:

```
struct sysinfo {
   long uptime;              /* Sekunden seit dem Start         */
   unsigned long loads[3];   /* Auslastung vor 1, 5 und 15 min  */
   unsigned long totalram;   /* Größe des RAM-Speichers         */
   unsigned long freeram;    /* freier RAM-Speicher             */
   unsigned long sharedram;  /* Größe des Shared Memory         */
   unsigned long bufferram;  /* Größe des Pufferspeichers       */
   unsigned long totalswap;  /* Größe des Swap-Speichers        */
   unsigned long freeswap;   /* freier Swap-Speicher            */
   unsigned short procs;     /* Anzahl der laufenden Threads    */
   unsigned long totalhigh;  /* Größe des User-Speichers        */
   unsigned long freehigh;   /* Größe des freien User-Speichers */
   unsigned int mem_unit;    /* Seitengröße                     */
   char _f[20-2*sizeof(long)-sizeof(int)];   /* Füllbytes       */
};
```

`sys_sysinfo` liefert eine allgemein zugängliche Methode, um die Systeminformationen abzufragen. Das ist einfacher und risikoloser, als `/dev/kmem` zu lesen (wobei die Werte fünf, zehn und elf noch mit 0 gefüllt werden).

Implementierung

Dieser Systemruf wird nicht von der C-Bibliothek unterstützt. Wenn Sie ihn benutzen wollen, müssen Sie eine Datei `sysinfo.c` mit folgendem Inhalt anlegen:

```
#include <unistd.h>

_syscall1( int , sysinfo , struct sysinfo *, s)
```

Das entspricht außerdem der Vorgehensweise bei der Implementierung eines Systemrufs.

Fehler

EFAULT — wenn der Zeiger auf `info` ungültig ist.

Systemruf syslog

Datei: kernel/printk.c

```
long sys_syslog(int type, char *buf, int len);
```

`sys_syslog()` verwaltet das *Logbuch* des Systems und legt fest, ab welcher Priorität (*Loglevel*) Nachrichten darin erscheinen. Das Logbuch ist ein acht KByte großer Speicher im Kern und wird von der `printk()`-Funktion gefüllt (siehe Anhang E). Das Loglevel ist eine Prioritätsstufe für das Verhalten der `printk()`-Funktion. Nur Nachrichten, deren Priorität über dem Loglevel liegt, werden von `printk()` auf die Konsole ausgeschrieben.

```
#define LOG_BUF_LEN     8192
static char log_buf[LOG_BUF_LEN];
```

Für den Zugriff auf das Logbuch gibt es drei Variablen:

```
unsigned long log_size = 0;
static unsigned long log_start = 0;
static unsigned long logged_chars = 0;
```

Die erste Variable beschreibt die Größe des Logbuches, sie kann zwischen 0 und `LOG_BUF_LEN` schwanken, die zweite kennzeichnet den Beginn der aktuellen Nachricht. Mit dem Zugriff

```
(log_start+log_size) & (LOG_BUF_LEN-1)
```

kommt man also an die letzte Stelle der aktuellen Eintragung. `logged_chars` enthält die Gesamtzahl der Zeichen im Logbuch. So viel zum internen Aufbau der Funktion.

Die genaue Arbeitsweise der Funktion kann mit `type` festgelegt werden. Es gibt folgende Werte:

0 schließt das Logbuch, die Funktion ist nicht implementiert. Es wird 0 zurückgegeben.

1 öffnet das Logbuch, die Funktion ist nicht implementiert. Es wird 0 zurückgegeben.

2 liest `len` Zeichen ab der Stelle `log_start` aus dem Logbuch. Dazu wird die Variable `log_size` ausgewertet. Wenn das Buch leer ist (`log_size` gleich 0), blockiert dieser Aufruf, bis ein Prozess einen Eintrag hinterlassen hat, und liest diesen dann aus. `log_size` wird um die (tatsächlich gelesene) Zeichenzahl vermindert.

3 liest Einträge aus dem Logbuch in den Speicher `buf` der Größe `len`. Diese Funktion blockiert nicht. `len` wird vorher mit die Größen `LOG_BUF_LEN` und `logged_chars` verglichen und (falls größer) auf diesen Wert gesetzt.

4 wie bei 3, zusätzlich löscht der Aufruf das Logbuch, indem er den Zähler `logged_chars` auf 0 setzt.

5 löscht das Logbuch.

6 setzt den Loglevel für die Funktion `printk()` auf 1. Damit werden nur Nachrichten mit der höchsten Prioritätsstufe auf der Konsole ausgeschrieben.

7 setzt den Loglevel für die Funktion `printk()` auf den Standardwert (7).

8 setzt den Loglevel für die Funktion `printk()` auf den in `len` stehenden Wert. `len` muss in diesem Fall zwischen 0 und 9 liegen.

Zurückgegeben wird die Anzahl der tatsächlich gelesenen Zeichen (im Fall 2, 3 und 4) oder 0.

Implementierung

Es existiert keine Umsetzung in der C-Bibliothek. Eine Einbindung ist mit folgender Datei möglich:

```
#include <unistd.h>

_syscall1( int , syslog , int, type, char *, buf, int, len)
```

Fehler

EPERM — wenn ein nicht privilegierter Benutzer `sys_syslog()` mit einem anderen `type` als 3 aufruft.

EINVAL — wenn `buf` NULL oder `len` negativ ist.

Systemruf	time	stime	POSIX
	gettimeofday	settimeofday	SVR4 4.3BSD

Datei: kernel/time.c

```
#include <time.h>
long sys_time(int *t);
long sys_stime(const time_t *t);
long sys_gettimeofday(struct timeval *tv,
        struct timezone *tz);
long sys_settimeofday(struct timeval *tv,
        struct timezone *tz);
```

`sys_time()` speichert in `t` die seit dem 1. Januar 1970, 0 Uhr, vergangene Zeit in Sekunden und gibt sie außerdem zurück. Dazu wird das Makro `CURRENT_TIME` genutzt.

`sys_stime()` setzt die Systemzeit, genauer gesagt `xtime.tv_sec`, auf den mit `t` angegebenen Wert. `xtime.tv_usec` wird auf 0 gesetzt. Diese Funktion darf nur ein Prozess mit den entsprechenden Rechten (`CAP_SYS_TIME`) ausführen. Er gibt 0 bei Erfolg und eine negative Zahl bei einem Fehler zurück.

`sys_gettimeofday()` und `sys_settimeofday()` ermöglichen eine präzisere Zeitverwaltung. `tv` ist dieselbe Struktur wie die in `sys_setitimer()` angegebene:

```
struct timeval {
   long tv_sec;   /* Sekunden       */
   long tv_usec;  /* Mikrosekunden  */
};
```

`tz` ist eine Zeitzone:

```
struct timezone {
   int  tz_minuteswest;
   /* Minuten westlich von Greenwich */
   int  tz_dsttime;
   /* Verwendung von Sommerzeit */
};
```

Die Funktion `sys_settimeofday()` ist wie `sys_stime()` nur für berechtigte Prozesse (`CAP_SYS_TIME`) erlaubt. Wird für `tv` oder `tz` NULL angegeben, ändert sich dieser Systemwert nicht. Beim ersten Aufruf[14] mit gesetztem `tz` wird außerdem die CMOS-Uhr auf UTC umgestellt. Für das Setzen der `tv`-Werte werden die Interrupts ausgeschaltet, und `time_status` wird auf `TIME_BAD` gesetzt. Bei der Ausführung werden die Daten in den Adressraum des Kerns kopiert und die Systemwerte aktualisiert. Der Systemruf gibt 0 bei Erfolg zurück. Die Arbeitsweise des zugrunde liegenden Timers ist in Abschnitt 3.1.6 beschrieben.

Implementierung

Alle vier Systemrufe werden über das Syscall-Makro umgesetzt.

Fehler

EFAULT — wenn ein Kopieren der Daten vom User- in den Kernel-Adreßraum (bzw. umgekehrt) fehlschlug.

EPERM — wenn der Prozess, der `sys_stime()` oder `settimeofday` aufruft, nicht `CAP_SYS_TIME`-Rechte hat.

EINVAL — wenn ein ungültiger Wert (Zeitzone o. Ä.) angegeben ist.

14 Das sollte so früh wie möglich geschehen, um andere evtuell laufende Programme nicht durcheinander zu bringen. Üblicherweise geschieht das in einem Skript in /etc/rc.

Systemruf times POSIX

Datei: kernel/sys.c

```
#include <linux/times.h>

long sys_times(struct tms *buf);
```

`sys_times()` schreibt die vom aktuellen Prozess und seinen Kindern verbrauchte Zeit in die Struktur `buf`. Die Struktur `tms` wird in `<linux/times.h>` wie folgt definiert:

```
struct tms {
        time_t tms_utime;  /* Nutzerzeit           */
        time_t tms_stime;  /* Systemzeit           */
        time_t tms_cutime; /* Nutzerzeit der Kinder     */
        time_t tms_cstime; /* Systemzeit der Kinder     */
};
```

`sys_times()` gibt die `jiffies` des Systems zurück.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EFAULT — wenn das Kopieren der Werte fehlschlug.

Systemruf uname POSIX

Datei: kernel/sys.c

```
#include <linux/utsname.h>

long sys_newuname(struct new_utsname *buf);
```

`sys_uname()` gibt Informationen über das System zurück. Die Informationen befinden sich dann in `buf`. Die Struktur `utsname` hat folgendes Aussehen:

```
struct utsname {
        char sysname[65];     /* Name des Betriebssystems    */
        char nodename[65];    /* Name des Rechners           */
        char release[65];     /* Release des Betriebssystems */
        char version[65];     /* Version des Betriebssystems */
        char machine[65];     /* Prozessortyp                */
        char domainname[65];  /* Rechnerdomain               */
};
```

Die Release ist der aktuelle Entwicklungsstand des Systems (`2.4`). Die Version ist die Anzahl der bisherigen Konfigurationen des Kerns sowie die Zeit der letzten Übersetzung (`#95 Sat Dez 4 05:08:15 MET DST 1999`).

Aus Kompatibilitätsgründen gibt es noch zwei abgerüstete Versionen: der ersten (`old_utsname`) fehlt die Domain, bei der zweiten (`oldold_utsname`) ist die Länge der Einträge zusätzlich auf neun Byte beschränkt (POSIX definiert nur acht Byte lange Einträge der Struktur (sowie Platz für das Nullbyte)).

Fehler

EFAULT — wenn `buf` NULL ist.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Systemruf vm86 LINUX

Datei: arch/i386/kernel/vm86.c

```
#include <linux/vm86.h>

int sys_vm86(unsigned long subfunction,
                struct vm86plus_struct * v86);
```

Die Funktion `sys_vm86()` setzt den Prozess in den virtuellen 8086-Modus. Zur Steuerung kann folgende Struktur belegt werden:

```
struct vm86plus_struct {
        struct vm86_regs regs;  /* Registersatz         */
        unsigned long flags;
        unsigned long screen_bitmap;
        unsigned long cpu_type;
        struct revectored_struct int_revectored;
        struct revectored_struct int21_revectored;
        struct vm86plus_info_struct vm86plus;
};
```

Die genaue Arbeitsweise der Funktion steuert der Parameter `subfunction`. Ist `subroutine` gleich einer der folgenden fünf Werte, wird `vm86` als eine Integerzahl `irqnumber` interpretiert.

`VM86_GET_AND_RESET_IRQ` — Falls `irqnumber` zwischen 3 und 15 liegt und der Eintrag `vm86_irqs[irqnumber]` der aktuelle Prozess ist, wird in das zu dieser Nummer gehörende Bit in `irqbits` invertiert und der alte Wert zurückgegeben.

`VM86_GET_IRQ_BITS` — Die `irqbits` werden zurückgegeben.

VM86_REQUEST_IRQ — `irqnumber` wird als `((sig<<8)|(irq & 255))` interpretiert, und die beiden Werte werden extrahiert. Der Prozess muss das Recht `CAP_SYS_ADMIN` haben, `sig` ein gültiges Signal sein und `irq` muss zwischen 3 und 15 liegen. Dann wird die Funktion `irq_handler()` als Interrupt-Behandlungsroutine eingetragen und in `vm86_irqs[irq]` wird `sig` als Signal und der aktuelle Prozess als Task eingetragen. Zurückgegeben wird `irq`.

VM86_FREE_IRQ — Falls der Interrupt `irqnumber` zwischen 3 und 15 liegt und Eintrag `vm86_irqs[irqnumber]` der aktuelle Prozess ist, wird die zugehörige Interrupt-Behandlungsroutine gelöscht, `vm86_irqs[irqnumber].tsk` auf 0 gesetzt und das Bit in `irqbits` gelöscht.

VM86_PLUS_INSTALL_CHECK — Der Aufruf gibt null zurück. Da alte Versionen diesen Parameter nicht kennen, ermöglicht dieser Parameter eine Versions-Abfrage.

Ansonsten wird von dieser Adresse eine Struktur `pt_regs` gelesen und als Register für den Prozess eingetragen. Vorher wird das EFlags-Register kontrolliert, so dass der *Protected Mode* erhalten bleibt.

Die Funktion speichert den aktuellen Stack des Kerns und springt dann in den virtuellen Modus. Verwendet wird der Aufruf vom DOS-Emulator.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EPERM — wenn der Stack bereits gespeichert ist.

Systemruf	**wait4** **waitpid**	4.3+BSD POSIX

Datei: kernel/exit.c

```
long sys_waitpid(pid_t pid, unsigned int *stat_addr,
        int options);

long sys_wait4(pid_t pid, unsigned int *stat_addr,
        int options, struct rusage *ru);
```

`sys_wait4()` wartet auf die Beendigung eines durch `pid` angegebenen Prozesses. Zusätzlich schreibt die Funktion den Exit-Code an die Adresse `stat_addr` und in der Struktur `ru` Informationen über benutzte Ressourcen des Prozesses. Als Optionen sind folgende Werte möglich:

__WCLONE — Es wird nur auf mit `clone()` erzeugte Prozesse gewartet.

WUNTRACED — Es werden auch die gestoppten Prozesse berücksichtigt, bei denen PF_PTRACE nicht gesetzt wurde.

WNOHANG — sys_wait4() blockiert nicht.

Die Funktion fragt in einer Schleife alle Kindprozesse des aktuellen Prozesses ab, ob sie zu PID passen. Ist pid

> 0, wartet wait4() auf den Kindprozess mit PID gleich pid.

0, wartet wait4() auf jeden Kindprozess, dessen PGRP gleich der PGRP des rufenden Prozesses ist.

-1, wartet wait4() auf jeden Kindprozess.

< -1, wartet wait4() auf jeden Kindprozess, dessen Prozessgruppe gleich -pid ist.

Wenn kein Prozess gefunden wurde und das Flag WNOHANG gesetzt ist, kehrt sys_wait4() zurück. Ansonsten wird der Scheduler aufgerufen und die Schleife erneut durchlaufen.

sys_wait4() kehrt zurück, wenn der Prozess (auf den gewartet wird) terminiert oder ein Zombie ist, WNOHANG gesetzt ist oder ein nichtblockiertes Signal eintraf. Zurückgegeben wird eine negative Zahl im Fehlerfall, die PID des beendeten Prozesses oder 0 (bei WNOHANG).

sys_waitpid() wartet auf den Prozess pid mit den angegebenen Optionen options. Die Funktion sys_waitpid() delegiert den Aufruf an die Funktion sys_wait4() weiter. Sie wird nur noch aus Kompatibilitätsgründen bereitgestellt und könnte in zukünftigen Versionen auch in der C-Bibliothek implementiert werden.

```
asmlinkage long sys_waitpid(pid_t pid,
        unsigned int * stat_addr, int options)
{
        return sys_wait4(pid, stat_addr, options, NULL);
}
```

Das genaue Zusammenspiel von sys_wait4(), sys_exit() und dem Scheduler ist in Abschnitt 3.3.3 beschrieben.

Implementierung

Die Systemrufe *wait4* und *waitpid* werden über das Syscall-Makro umgesetzt. Die Funktion waitpid() steht auf Alpha-Maschinen nicht zur Verfügung. wait() wird bereits nicht mehr als Systemruf, sondern nur noch als Inline-Funktion (in unistd.h) angeboten.

```
static inline pid_t wait(int * wait_stat)
{
        return waitpid(-1,wait_stat,0);
}
```

Fehler

ERESTARTSYS — wenn `WNOHANG` nicht gesetzt wurde und der Prozess ein nicht-blockiertes Signal oder ein `SIGCHLD` erhielt.

ECHILD — wenn der Kindprozess `pid` nicht existiert.

A.2 Das Dateisystem

Die folgenden Systemrufe stellen den Kontakt mit dem Dateisystem her. Aufgrund des virtuellen Dateisystems in Linux ist der Übergang vom Nutzer zum Kern nur eine Zwischenstufe der eigentlichen Arbeit.

Fast alle Systemrufe führen erst einen Parameter-Check durch und rufen dann die entsprechende Inode- oder File-Operation des zugehörigen Dateisystems auf. Alle Systemrufe, die einen Parameter `path` besitzen, verwenden die Funktion `open_namei()`. Diese Funktion ermittelt die zum Namen gehörige Inode und ist in Abschnitt 6.2.10 genauer erläutert.

Systemruf access POSIX

Datei: fs/open.c

```
#include <unistd.h>

long sys_access(const char *filename, int mode);
```

Die Funktion `sys_access()` überprüft, ob ein Nutzer die Zugriffsrechte `mode` auf die Datei `filename` hat. In `mode` dürfen nur die letzten drei Bits gesetzt sein; mögliche Werte sind:

`S_IROTH` — falls die Datei lesbar ist.

`S_IWOTH` — falls die Datei schreibbar ist.

`S_IXOTH` — falls die Datei ausführbar ist.

`sys_access()` verwendet für den Zugriff auf das Dateisystem nicht die effektive UID und GID, sondern nur die normalen, die für den Zugriffstest in die FSUID und FSGID kopiert werden. Ist die aktuelle UID zudem ungleich 0, werden für den Test alle Prozessrechte gelöscht (`cap_clear()`).

Wenn die Inode-Operationen eine Komponente `permission` bereitstellen, wird diese Funktion zum Ermitteln der Zugriffsrechte genutzt. Ansonsten wird mit `inode->i_mode` anhand der Unix-Rechte entschieden.

Implementierung

Die Umsetzung geschieht einfach über das Syscall-Makro.

Fehler

EINVAL — wenn die in `mode` angegebenen Rechte ungültig sind oder die Datei `filename` nicht existiert.

EROFS — wenn bei einem *Read-Only*-Dateisystem das Schreibrecht getestet wurde.

EACCESS — wenn der Zugriff mit den angegebenen Rechten nicht erlaubt ist.

Systemruf bdflush

Datei: fs/buffer.c

```
long sys_bdflush(int func, long data);
```

Die Kernfunktion `sys_bdflush()` organisiert das Ausschreiben der als „dirty" gekennzeichneten Blöcke im Puffercache. Der Linux-Kern verwaltet den Puffercache (unter anderem) mit zwei Tabellen:

```
#define NR_LIST 3

static struct buffer_head * lru_list[NR_LIST] = {NULL, };
int nr_buffers_type[NR_LIST] = {0,};
```

Die erste enthält Zeiger auf doppelt verkettete Listen, die jeweils eine Klasse von Blöcken enthalten. Eine Klasse kann zum Beispiel `BUF_SHARED` oder `BUF_LOCKED` sein. Die zweite Tabelle enthält die Anzahl der Blöcke in der jeweiligen Liste.

Der Pointer `lru_list[BUF_DIRTY]` zeigt auf die von `bdflush()` verwaltete Liste, sie enthält die noch nicht auf das Speichermedium geschriebenen Blöcke. Die Anzahl der Blöcke in dieser Liste steht in `nr_buffers_type[BUF_DIRTY]`. Ein Block wird mit Hilfe der Struktur `buffer_head`, im Folgenden Pufferkopf genannt, verwaltet und hat folgendes Aussehen:

```
struct buffer_head {
   struct buffer_head * b_next; /* Zeiger auf nächsten Eintrag */
   unsigned long b_size;        /* Blockgröße                  */
   unsigned long b_blocknr;     /* Blocknummer                 */
   kdev_t b_dev;                /* Gerät                       */
   kdev_t b_dev;                /* Gerät                       */
   unsigned long b_rsector;     /* Position der Puffer         */
   struct buffer_head * b_this_page; /* Puffer der akt. Page   */
   unsigned long b_state;            /* Status-Bitmap          */
   struct buffer_head * b_next_free;
```

```
   unsigned short b_count;      /* Nutzeranzahl                */
   char * b_data;               /* Zeiger auf Datenblock       */
   unsigned long b_flushtime;   /* letzte Schreibzeit          */
   unsigned long b_lru_time;    /* letzte Nutzungszeit         */
   struct wait_queue * b_wait;
   struct buffer_head ** b_pprev;
   struct buffer_head * b_prev_free; /* Liste der Puffer       */
   struct buffer_head * b_reqnext;   /* Request Liste          */
   void (*b_end_io)(struct buffer_head *bh, int uptodate);
   void *b_dev_id;
};
```

Um das Ausschreiben zu kontrollieren, gibt es eine Struktur, die die dazu notwendigen Parameter enthält.

```
static union bdflush_param {
   struct {
      int nfract;  /* Aktivierungsschwelle in Prozent     */
      int ndirty;  /* maximale Anzahl der in einem Zyklus */
                   /* auszuschreibenden Blöcke            */
      int nrefill; /* Anzahl der freien Blöcke, die       */
                   /* durch refill_freelist geladen       */
                   /* werden                              */
      int nref_dirt;  /* Aktivierungsschwelle             */
      int dummy1;
      int age_buffer; /* Alterungsdauer für Daten-        */
                      /* blöcke                           */
      int age_super;  /* Alterungsdauer für Meta-         */
                      /* blöcke (Verzeichnisse usw.)      */
      int dummy2;
      int dummy3;
   } b_un;
   unsigned int data[N_PARAM];
} bdf_prm = {{40, 500, 64, 256, 15, 30*HZ, 5*HZ, 1884, 2}};
```

Der Kern legt nicht nur die initialen Werte für die einzelnen Einträge der Struktur fest, sondern auch die Ober- und Untergrenze.

```
static int bdflush_min[N_PARAM] =
            {  0,  10,    5,   25,  0,   100, 100, 1, 1};
static int bdflush_max[N_PARAM] =
            {100,5000, 2000, 2000, 100, 60000, 60000, 2047, 5};
```

Der Ruf kann nur von einem Prozess mit dem Recht `CAP_SYS_ADMIN` ausgeführt werden. Ist das Argument `func` gleich 1, werden alle markierten Blöcke ausgeschrieben. Das Verwalten der Parameter geschieht mit `func` größer 1. Die Nummer des Parameters ist dabei `(func-2)>>1`. So bedeutet ein gerader Wert[15] von `func`, dass der Parameter mit dem in `data` stehenden Wert gefüllt wird, während ein ungerader Wert das Auslesen

15 Zum Beispiel heißt 2: Schreibe Parameter 0, 3 heißt: Lies Parameter 0, 4 heißt: Schreibe Parameter 1 usw.

des Parameters zur Folge hat. Beim Schreiben wird überprüft, ob `data` innerhalb der festgelegten Parameterminima und -maxima liegt.

Implementierung

Die C-Bibliothek bietet keine Schnittstelle zur Umsetzung des Systemrufs in die Kernelfunktion.

Fehler

EPERM — Nur Prozesse mit dem Recht `CAP_SYS_ADMIN` dürfen diese Funktion ausführen.

EINVAL — Der Wert `func` oder `data` ist ungültig.

Systemruf	**chdir** **fchdir**	POSIX

Datei: fs/open.c

```
long sys_chdir(const char *path);
long sys_fchdir(unsigned int fd);
```

`sys_chdir()` setzt das aktuelle Arbeitsverzeichnis auf das in `path` angegebene Verzeichnis. Dazu ermittelt die Funktion den zu `path` gehörigen DEntry und trägt ihn in die Komponente `fs->pwd` der Taskstruktur ein. `sys_fchdir()` funktioniert genauso, mit dem Unterschied, dass die Funktion anhand des übergebenen Dateideskriptors die Inode einfacher ermitteln kann.

Implementierung

Beide Systemrufe verwenden das Syscall-Makro.

Fehler

ENOTDIR — wenn `path` kein Verzeichnis ist.

EBADF — wenn `fd` ungültig ist.

ENOENT — wenn es für `path` keine Inode gibt.

EACCES — wenn keine Ausführungsrechte für das Verzeichnis gesetzt sind.

Systemruf	chmod	fchmod	POSIX
	chown	fchown	
	lchown		

Datei: fs/open.c

```
#include <linux/types.h>
#include <linux/stat.h>

long sys_chmod(const char *filename, mode_t mode);
long sys_fchmod(unsigned int fildes, mode_t mode);
long sys_chown(const char *filename, uid_t owner, gid_t group);
long sys_fchown(unsigned int fd, uid_t owner, gid_t group);
long sys_lchown(unsigned int fd, uid_t owner, gid_t group);
```

`sys_chmod()` setzt die Rechte der Datei `filename` auf die in `mode` angegebenen Rechte. Bits, die höher liegen als `S_ISUID` (z. B. `S_IFIFO`), werden dabei ausgeblendet, um Manipulationen zu verhindern. Gibt man für `mode` `-1` an, bleiben die aktuellen Zugriffsrechte (bis auf das Ausblenden) unverändert. Nur `ctime` wird neu gesetzt. Bei `sys_fchmod()` wird statt des Namens der Datei ein Deskriptor angegeben.

`sys_chown()` ändert den Besitzer und die Gruppe einer Datei in `owner` und `group`. Der Aufruf `sys_fchown()` hat dieselbe Funktion, nur dass ein Deskriptor angegeben wird. `sys_lchown()` funktioniert wie `sys_chown()`, mit einem Unterschied: Wenn `filename` ein symbolischer Link ist, wird er aufgelöst.

Wenn sich beim Ändern der Rechte die UID bzw. GID der Datei ändert, wird das zugehörige gesetzte S-Bit gelöscht (`S_ISGID` nur bei nicht gesetztem `S_IXGRP`). Bei allen drei Funktionen werden die Quotas den neuen Nutzern entsprechend aktualisiert (transferiert).

Alle fünf Aufrufe aktualisieren ihre Inode-Informationen. Dafür gibt es die Funktion `notify_change()`.

Implementierung

Alle fünf Systemrufe arbeiten mit dem Syscall-Makro.

Fehler

ENOENT — wenn die Datei nicht existiert.

EROFS — wenn das Dateisystem nur lesbar ist.

EPERM — wenn die Datei als `IS_IMMUTABLE` oder als `IS_APPEND` gekennzeichnet ist.

EDQUOT — wenn die Quotas des neuen Besitzers die Operation nicht erlauben.

Systemruf chroot

Datei: fs/open.c

```
long sys_chroot(const char * filename);
```

`sys_chroot()` setzt das Verzeichnis `filename` als `root`-Verzeichnis für den aufrufenden Prozess. Der Aufruf ermittelt die zu `filename` gehörige Inode, testet, ob sie sich auf ein Verzeichnis bezieht und ob das EXECUTE-Bit gesetzt ist. Wenn dann der Prozess noch `CAP_SYS_CHROOT` Rechte hat, wird das Verzeichnis als `fs->root` in die Taskstruktur eingetragen.

Implementierung

Die Umsetzung geschieht über das Syscall-Makro.

Fehler

EPERM — wenn ein nicht privilegierter Prozess den Aufruf ausführt.

ENAMETOOLONG — wenn der angegebene Name zu lang ist.

ENOENT — wenn das Verzeichnis nicht existiert.

ENOTDIR — wenn ein Teil des Pfades kein Verzeichnis, sondern eine Datei ist.

Systemruf dup dup2 POSIX

Datei: fs/fcntl.c

```
long sys_dup(unsigned int filedes);
long sys_dup2(unsigned int oldfd, unsigned int newfd);
```

`sys_dup()` und `sys_dup2()` erzeugen eine Kopie des Dateideskriptors. Die beiden Deskriptoren verweisen danach auf die gleiche File-Struktur. Ein gesetztes `close_on_exec`-Flag wird gelöscht. `sys_dup()` gibt den ersten freien Deskriptor für die Kopie zurück. `sys_dup2()` benutzt `newfd` als Kopie. Falls `newfd` noch nicht frei ist, wird die entsprechende Datei vorher geschlossen. Die beiden Systemrufe werden auf die Funktion `dupfd()` abgebildet.

Implementierung

Die Umsetzung beider Rufe erfolgt über das Syscall-Makro.

Fehler

EBADF — wenn ein ungültiger Deskriptor verwendet wird.

EMFILE — wenn bei `sys_dup()` kein Deskriptor mehr frei ist.

Systemruf execve POSIX

Datei: fs/exec.c
arch/i386/kernel/process.c

```
int sys_execve(struct pt_regs regs);
```

`sys_execve()` führt ein neues Programm aus. Die notwendigen Parameter stehen in der Registerstruktur. So enthält `regs.ebx` einen Zeiger auf den Dateinamen `filename` des Programms, `regs.ecx` einen Zeiger auf die Argumente, die dem angegebenen Programm übergeben werden sollen, und `regs.edx` die Adresse der Umgebung, in der der Prozess laufen soll.

Die Funktion bekommt im Kern als Argumente den Registerstack `pt_regs`. Da diese für die einzelnen Architekturen natürlich unterschiedlich ausfallen, befindet sich die Funktion (die eine Schnittstelle zu `do_execve()`, der eigentlichen Funktion, ist) im Architektur-Verzeichnis.

Die Datei `filename` muss ein Binärprogramm, dessen Format Linux bekannt ist, oder ein Skript sein. `sys_execve()` analysiert die Datei `filename`, geht die Liste der konfigurierten Binärformate (plus `kmod`, falls konfiguriert) durch und versucht, mit der Funktion `load_binary` die Datei aufzurufen.

Das mit `sys_execve()` aufgerufene Programm überlagert den rufenden Prozess vollständig. Das heißt, sowohl das Text- und Datensegment als auch der Stack und das BSS werden mit denen des geladenen Programms überschrieben. Das Programm übernimmt die PID des rufenden Prozesses und seine geöffneten Dateideskriptoren. Noch anstehende Signale werden gelöscht. Bei einem Fehler wird eine negative Zahl zurückgegeben, bei Erfolg gibt es keinen Rückgabewert. Die Implementierung des Aufrufs ist in Abschnitt 3.3.3 beschrieben.

Wenn der aktuelle Prozess mit `ptrace()` ausgeführt wird, liefert der Systemruf *execve* nach erfolgreicher Beendigung ein Signal `SIGTRAP`.

Implementierung

Die Struktur `pt_regs` aus `<asm/ptrace.h>` enthält genau ein Abbild der Register auf dem Stack, die bei einem Systemruf vor dem Aufruf der Kernelfunktion dort abgelegt werden. Linux unterstützt mehrere Binärformate, und jedes Format bringt eine eigene Funktion für das Laden der Binaries mit. Ihr werden der Name des Programms (eventuell ein Interpreter mit der Skript als Argument) und die Register übergeben (siehe Abschnitt 3.3.3).

Die üblichen Systemrufe, wie zum Beispiel *execl* oder *execv*, sind als Bibliotheksfunktionen implementiert. Bei `execv()` wird die aktuelle Umgebung dem eigentlichen Systemruf mitgegeben, und bei `execvp()` wird der Kommandoname im aktuellen Pfad gesucht und ein neues Argument zusammengebaut. Bei den Funktionen mit Argumentlisten, wie `execl()`, wird zusätzlich die übergebene Liste in einen Vektor kopiert.

Fehler

EACCESS — wenn `filename` keine normale Datei ist.

EPERM — wenn das Dateisystem mit `MS_NOEXEC` gemountet wurde.

ENOEXEC — wenn keine Dateiidentifikation (die *Magic Number*) oder keine Shell nach `#!` gefunden wurde.

E2BIG — wenn kein Speicher im Kern frei ist, um Argumente und Environment zu übergeben.

Systemruf fcntl POSIX

Datei: fs/fcntl.c
net/inet/sock.c

```
#include <fcntl.h>

long sys_fcntl(unsigned int fd, unsigned int cmd,
               unsigned long arg);
```

Der Systemruf `sys_fcntl()` ändert die Eigenschaften einer geöffneten Datei `fd`. Die entsprechende Operation wird mit `cmd` festgelegt:

`F_DUPFD` — Der Dateideskriptor `fd` wird in `arg` dupliziert. Das entspricht der Funktionalität von `sys_dup2()`. Bei Erfolg wird der neue Deskriptor zurückgegeben.

`F_GETFD` — Das `close-on-exec`-Flag des angegebenen Dateideskriptors wird gelesen.

`F_SETFD` — Wenn in `arg` das unterste Bit gesetzt ist, wird das `close-on-exec`-Flag des angegebenen Dateideskriptors gesetzt, ansonsten wird es gelöscht.

`F_GETFL` — Die Flags des Deskriptors werden zurückgegeben. Die Flags sind dieselben, wie in `sys_open()` beschrieben.

`F_SETFL` — Setzt die Flags auf den in `arg` angegebenen Wert. Die Flags und ihre Semantik entsprechen denen bei `sys_open()`. Intern werden nur `O_APPEND`, `O_NDELAY`, `FASYNC` und `O_NONBLOCK` gesetzt. Wenn die Datei als Append-Only-Datei eingerichtet und `O_APPEND` in den Flags nicht angegeben (soll gelöscht werden) ist, reagiert die Funktion mit einer Fehlermeldung. Bei einer Änderung des Flags `FASYNC` wird die File-Operation `fasync()` aufgerufen.

F_GETLK, F_SETLK und **F_SETLKW** — Setzt bzw. liest die Sperren einer Datei. Die Funktionsweise und Nutzung von Dateisperren ist ausführlich in Abschnitt 5.2.2 beschrieben.

F_GETOWN — Gibt die PID (PGRP) des Prozesses zurück, der den Deskriptor `fd` benutzt. Prozessgruppen werden als negative Werte zurückgegeben! Der Wert steht in `f_owner` der File-Struktur.

F_SETOWN — Trägt `arg` als PID in die `f_own`-Struktur für den angegebenen Dateideskriptor ein. Außerdem werden `uid` und `euid` der Struktur auf die Werte des aktuellen Prozesses gesetzt.

F_GETSIG — Liefert das Signal, das gemäß POSIX bei einem IO-Fehler ausgelöst wird (steht in der `f_owner`-Struktur).

F_SETSIG — Trägt `arg` als Signal in die `f_owner`-Struktur ein.

Wenn der Dateideskriptor `fd` mit einem Socket verbunden ist, wird der Aufruf auf die entsprechende Funktion für Sockets abgebildet.

Implementierung

Die Umsetzung geschieht über das Syscall-Makro.

Fehler

EBADF — wenn `fd` kein Deskriptor einer geöffneten Datei ist.

EINVAL — wenn bei `F_DUPFD` ein negativer oder zu großer Wert für `arg` angegeben wurde oder der Prozess sein Maximum an offenen Dateien schon erreicht hat oder wenn für `cmd` ein ungültiger Wert angegeben wurde.

EPERM — wenn bei `F_SETOWN` keine Berechtigung vorliegt.

Systemruf flock POSIX

Datei: fs/locks.c

```
long sys_flock(unsigned int fd, unsigned int cmd);
```

Mit Hilfe dieser Funktion können Sperren auf Dateien verwaltet werden. Existieren mehrere Sperren auf eine Datei, werden sie in einer Liste verwaltet. Dazu enthält jede Inode einen Verweis auf eine `file_lock`-Struktur:

```
struct file_lock {
   struct file_lock *fl_next;  /* Liste aller Locks für diese   */
                               /* Inode                         */
   struct list_head fl_link;   /* Liste aller Locks             */
```

```
   struct list_head fl_block;  /* Liste aller blockierten       */
                               /* Prozesse                      */
   fl_owner_t fl_owner;        /* Dateien des sperrenden Pro-   */
                               /* zesses                        */
   unsigned int fl_pid;        /* sperrender Prozess            */
   wait_queue_head_t fl_wait;  /* Warteschlange aller blockier- */
                               /* ten Prozesse                  */
   struct file *fl_file;
   unsigned char fl_flags;
   unsigned char fl_type;
   loff_t fl_start;            /* Anfang der Sperre             */
   loff_t fl_end;              /* Ende der Sperre               */

   void (*fl_notify)(struct file_lock *); /* Freigabe-Callback  */
   void (*fl_insert)(struct file_lock *); /* Insert-Callback    */
   void (*fl_remove)(struct file_lock *); /* Lösch-Callback     */

   struct fasync_struct * fl_fasync;

   union {
      struct nfs_lock_info   nfs_fl;
   } fl_u;
};
```

Die Funktion wirkt sich immer auf die ganze Datei aus. Untergeordente Funktionen lassen durch die Angabe von Dateibereichen ein gezielteres Sperren zu. Als Werte für `cmd` sind möglich:

`LOCK_SH` — Die Datei wird für Lesezugriffe gesperrt.

`LOCK_EX` — Die Datei wird für Schreibzugriffe gesperrt.

`LOCK_UN` — Alle Locks werden aufgehoben. Alle Prozesse, die durch das Zugreifen auf einen Lock in eine Warteschleife gerieten, werden aufgeweckt.

`LOCK_MAN` — Eine obligatorische (*mandatory*) Sperre. Damit werden SMB-Zugriffe simuliert.

Implementierung

Die Umsetzung geschieht zweistufig. Zuerst wird die Kernelfunktion über das Syscall-Makro aufgerufen. Sollte das fehlschlagen, wird `sys_fcntl()` bemüht.

Fehler

EBADF — Der Deskriptor ist ungültig.

EINVAL — Der Wert `cmd` ist ungültig.

ENOLCK — Es kann kein neuer Eintrag in der Liste angelegt werden.

EBUSY — Das Flag `F_POSIX` ist gesetzt.

Systemruf getcwd LINUX

Datei: fs/dcache.c

```
long sys_getcwd(char *buf, unsigned long size);
```

Dieser Ruf speichert das aktuelle Arbeitsverzeichnis in `buf` ab. Sollte dieses gelöscht sein, wird `" (deleted)"` angefügt. `size` gibt die Größe des übergebenen Puffers an. Ist der Pfad-String länger, wird ein Fehler zurückgegeben.

Implementierung

Fehler

ENOENT — wenn das Ermitteln des Verzeichnisses fehlschlug.

ERANGE — wenn der Pfad länger als `size` ist.

EFAULT — wenn das Kopieren in den Puffer `buf` fehlschlug.

Systemruf ioctl 4.3+BSD

Datei: fs/ioctl.c

```
#include <fs/ioctl.h>

long sys_ioctl(unsigned int fd, unsigned int cmd,
               unsigned long arg);
```

Die Funktion `sys_ioctl()` manipuliert die Parameter eines Geräts. Benutzt wird diese Funktion hauptsächlich bei der Steuerung von Gerätetreibern. Der erste Parameter ist ein geöffneter Deskriptor der entsprechenden Datei.

Im Argument `cmd` wird die gewünschte Funktion angegeben. Makros und Definitionen für die Verwendung dieses Aufrufes finden Sie in `<linux/ioctl.h>`. Einige Funktionen sind für alle Dateideskriptoren erlaubt:

`FIOCLEX` — Das Flag `close_on_exec` wird gesetzt.

`FIONCLEX` — Das Flag `close_on_exec` wird gelöscht.

`FIONBIO` — Ist der durch die Adresse `arg` angegebene Wert gleich 0, wird das Flag `O_NONBLOCK` gelöscht, ansonsten wird es gesetzt.

`FIOASYNC` — Wie bei `FIONBIO` wird hier das Flag `O_SYNC` gesetzt oder gelöscht. Das Synchronisationsflag ist noch nicht implementiert, wird aber der Vollständigkeit halber mit bearbeitet.

Diese vier Kommandos bearbeitet die Funktion selbst. Alle anderen werden weitergegeben: entweder an die Funktion `file_ioctl()`, wenn `fd` sich auf eine reguläre Datei bezieht, oder an die File-Operation `ioctl()` (siehe Abschnitt 6.2.9). Die Funktion `file_ioctl()` behandelt folgende Kommandos:

`FIBMAP` — liest den Block mit der Nummer `arg` aus der Datei und speichert ihn an der Adresse `arg` ab.

`FIGETBSZ` — speichert an der Adresse `arg` die Größe des Superblocks ab.

`FIONREAD` — trägt die an der Adresse `arg` stehende (Integer-) Zahl als Position des Dateizeigers ein.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EBADF — wenn `fd` ungültig ist.

ENOTTY — wenn `fd` sich nicht auf ein zeichenorientiertes Gerät bezieht oder der benutzte `cmd` vom Gerät `fd` nicht unterstützt wird.

EINVAL — wenn `cmd` oder `arg` ungültig sind.

Systemruf	link	unlink	POSIX
	rename	rmdir	
	symlink		

Datei: fs/namei.c

```
long sys_link(const char *oldname, const char *newname);
long sys_rename(const char *oldname, const char *newname);
long sys_rmdir(const char *name);
long sys_symlink(const char *oldname, const char *newname);
long sys_unlink(const char *name);
```

`sys_link()` und `sys_symlink()` erzeugen Verweise (Hardlinks) bzw. symbolische Verweise (Softlinks) mit dem Namen `newname`, die auf `oldname` verweisen.

Zuerst werden die beiden Parameter (aus Laufzeitgründen) in das Kernel-Adreßraum kopiert, und danach wird die Funktion `do_link()` mit den beiden Namen aufgerufen. Symbolische Links als Namen werden nicht aufgelöst! Die Funktion `do_link()` legt den Verweis an und initialisiert die Quotastruktur für das Verzeichnis, in dem der Verweis angelegt wird. Für Dateien, die als `S_APPEND` oder `S_IMMUTABLE` gekennzeichnet sind, können keine Verweise angelegt werden.

Die Funktion `sys_symlink()` kopiert ebenfalls `oldname` und `newname` und ruft dann `do_symlink()` auf. Sie arbeitet ähnlich wie `do_link()`, führt aber keine Überprüfungen für `oldname` durch.

`sys_rename()` legt die Datei unter dem Namen `newname` neu an und löscht die alte Datei. Eine Datei, die als `S_APPEND` gekennzeichnet ist, kann nicht umbenannt werden. Zuletzt werden die Quotastrukturen für `oldname` und `newname` initialisiert.

Die Funktion `sys_unlink()` dekrementiert den Link-Zähler für die Datei `name` und löscht die Datei, falls der Zähler 0 ist. `sys_rmdir()` arbeitet ähnlich wie `sys_unlink()`, entfernt aber das Verzeichnis.

Diese Kernelfunktionen werden intern immer auf die entsprechende Inode-Operation umgesetzt, nachdem die nötigen Rechte getestet wurden.

Implementierung

Die Systemrufe werden über das Syscall-Makro umgesetzt.

Fehler

EACCESS — wenn das Verzeichnis keine Ausführungsrechte hat.

ENOENT — wenn `oldname` nicht existiert oder der Pfadname ungültig ist.

ENOTDIR — beim Umwandeln des Dateinamens in eine `dentry`-Struktur trat ein Fehler auf (keine Inode oder die Inode-Operation `lookup()` wurde nicht gefunden).

EPERM — wenn die Inode der Datei den Link nicht erlaubt, `newname` ungültig ist oder das Dateisystem die Operation nicht unterstützt.

EROFS — Es handelt sich um ein *Read-only*-Dateisystem.

EBUSY — Es können keine Mount-Points gelöscht werden.

EXDEV — wenn `oldname` und `newname` bei `sys_link()` nicht im selben Dateisystem liegen.

Systemruf	**lseek**	POSIX
	llseek	

Datei: fs/read_write.c

```
#include <linux/types.h>

long sys_lseek(unsigned int fd, off_t offset, unsigned int origin)
long sys_llseek(unsigned int fd, unsigned long offset_high,
                unsigned long offset_low, loff_t * result,
                unsigned int origin);
```

`sys_lseek()` setzt eine neue aktuelle Position der Datei (`file->f_pos`) auf `offset` relativ zu `origin` (2 — vom Dateiende, 1 — von der aktuellen Position). LINUX versucht erst die `llseek()`-Funktion des Dateisystems zu verwenden, zu dem die Datei gehört. Wenn dieses keine `llseek`-Funktion besitzt, wird die neue Position selbst berechnet.

Die neue absolute Adresse wird zurückgegeben. Die aktuelle Position und die Dateigröße lassen sich dann einfach ermitteln:

```
pos  = lseek(fd, 0, 1); size = lseek(fd, 0, 2);
```

Die Funktion `sys_llseek()` positioniert in „großen" Dateien. Dazu wird aus `offset_high` und `offset_low` ein neuer Offset vom Typ `long long` zusammengebaut.

```
offset =  ((loff_t) offset_high << 32) | offset_low;
```

Unterstützt das Dateisystem eine Inode-Operation `llseek()`, dann wird diese aufgerufen. Ansonsten versucht die Funktion die neue Position selbst zu berechnen. Diese Funktion gibt bei Erfolg 0 zurück. Die neue Position steht in `result`.

Implementierung

Während der Systemruf *lseek* wie üblich über das Syscall-Makro umgesetzt wird, ist `sys_llseek()` auf Intel-Rechnern in der C-Bibliothek nur eingeschränkt implementiert.

```
loff_t __llseek (int fd, loff_t offset, int origin);
```

Fehler

EBADF — wenn `fd` ungültig ist.

EINVAL — wenn `origin` größer als 2 ist oder sich beim Berechnen der neuen Position ein Wert < 0 ergab.

Systemruf	**mount** **umount**	SVR4

Datei: fs/super.c

```
int sys_mount(char * dev_name, char * dir_name, char * type,
        unsigned long new_flags, void * data);
int sys_umount(const char *name, int flags);
```

`sys_mount()` richtet das Dateisystem, das sich auf dem Blockgerät `devname` befindet, im Verzeichnis `dirname` ein. Wichtig ist, dass der Prozess dazu das Recht `CAP_SYS_ADMIN` hat. In `type` steht der Typ des Dateisystems, z.B. `ext2`. Die `new_flags` steuern den Ablauf des Mountens und die Eigenschaften des gemounteten Dateisystems.

MS_RDONLY — Dateisystem ist nur lesbar.

MS_NOSUID — SUID- und SGID-Bit werden ignoriert.

MS_NODEV — Der Zugriff auf Gerätedateien ist untersagt.

MS_NOEXEC — Das Ausführen von Dateien ist untersagt.

MS_SYNCHRONOUS — Schreibzugriffe werden sofort auf der Festplatte aktualisiert.

MS_REMOUNT — Die Flags werden bei einem schon gemounteten Dateisystem geändert (*Remount*).

S_QUOTA — Beim Anlegen oder Löschen einer Inode wird ihre Quota-Struktur aktualisiert.

S_APPEND — Beim Öffen von Dateien zum Schreiben muss das Flag `O_APPEND` gesetzt sein.

S_IMMUTABLE — Die Dateien und ihre Inodes dürfen nicht verändert werden.

MS_NOATIME — Beim Zugriff auf eine Datei wird ihre *access time* nicht aktualisiert.

MS_NODIRATIME — Beim Zugriff auf ein Verzeichnis wird seine *access time* nicht aktualisiert.

MS_MGC_VAL — zeigt die neuere Version des Systemrufs *mount* an. Ohne diese Signatur in den Bits 16 bis 31 werden nur die ersten vier Optionen ausgewertet.

`data` ist ein Zeiger auf eine beliebige, maximal `PAGE_SIZE-1` große Struktur, die dateisystemspezifische Informationen enthalten kann[16].

Bei `MS_REMOUNT` muss kein Typ und kein Gerät angegeben werden. Dann aktualisiert der Ruf nur die in `new_flags` und `data` stehenden Informationen.

`sys_umount()` entfernt das Dateisystem wieder. Als Name kann dabei sowohl das Verzeichnis als auch das Gerät angegeben werden. Die Funktion schreibt den Superblock zurück und gibt das Gerät frei. Als `flag` kann `MNT_FORCE` angegeben werden; dann wird als Erstes die Superblock-Funktion `umount_begin()` aufgerufen (sofern sie implementiert ist).

Implementierung

Beide Systemrufe werden über das Syscall-Makro umgesetzt.

Fehler

EPERM — wenn keine Superuserrechte vorhanden sind.

ENODEV — wenn für `type` ist kein Dateisystem bekannt ist.

EACCES — wenn `dev_name` ist kein Gerät ist.

16 Diese Daten werden in der `u`-Union des Superblocks abgelegt, siehe Abschnitt 6.2.1.

ENOTBLK — wenn `dev_name` kein Blockgerät ist oder es keine File-Operationen zur Verfügung stellt.

ENXIO — wenn die Major-Nummer des Gerätes ungültig ist.

EBUSY — wenn ein Prozess im Verzeichnis arbeitet oder das Verzeichnis schon gemountet ist.

ENOTDIR — wenn `dir_name` kein Verzeichnis ist.

EINVAL — wenn `read_super()` fehlgeschlagen oder `dev_name` nicht gemountet ist.

Systemruf nfsservctl LINUX

Datei: fs/filesystems.c
fs/nfsd/nfsctl.c

```
int sys_nfsservctl(int cmd, void *argp, void *resp);
```

Dieser Ruf steuert die Arbeit des NFS-Dämons. Er kann als Modul konfiguriert werden. Dazu muss während der Übersetzung des Kerns `CONFIG_NFSD_MODULE` aktivert werden, ansonsten liefert der Aufruf den Fehler `ENOSYS`.

Die eigentliche Funktion `handle_sys_nfsservctl()` ist in `nfsctl.c` zu finden. Die Übergabe erfolgt entweder als Delegierung oder als Präprozessor-Makro.

Der Pointer `argp` zeigt auf eine `nfsctl_arg`-Struktur, der Pointer `resp` auf eine `nfsctl_res`-Struktur.

```
struct nfsctl_arg {
        int     ca_version;  /* Sicherheitsparameter */
        union {
                struct nfsctl_svc       u_svc;
                struct nfsctl_client    u_client;
                struct nfsctl_export    u_export;
                struct nfsctl_uidmap    u_umap;
                struct nfsctl_fhparm    u_getfh;
                struct nfsctl_fdparm    u_getfd;
                unsigned int            u_debug;
        } u;

union nfsctl_res {
        struct knfs_fh          cr_getfh;
        unsigned int            cr_debug;
};
```

Der Parameter `cmd` gibt, wie immer, die genaue Funktion an:

`NFSCTL_SVC` — Erzeugt einen neuen Server; angegeben werden die Portnummer und die Anzahl der gewünschten Threads. Gibt bei Erfolg 0 zurück.

NFSCTL_ADDCLIENT — Erzeugt einen neuen Klient. Gibt bei Erfolg 0 zurück.

NFSCTL_DELCLIENT — Löscht den angegebenen Klient. Gibt bei Erfolg 0 zurück.

NFSCTL_EXPORT — Exportiert das angegebene Dateisystem. Gibt bei Erfolg 0 zurück.

NFSCTL_UNEXPORT — Beendet den Export des angegebenen Dateisystems. Gibt bei Erfolg 0 zurück.

NFSCTL_GETFH — Füllt `resp` mit den Werten der Datei, die mit Geräte- und Inode-Nummer in `addr.u.u_export` beschrieben ist.

NFSCTL_GETFD — Füllt `resp` mit den Werten der Datei, die mit der Pfadangabe in `addr.u.u_export` gekennzeichnet ist.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EINTR — wenn ein nicht blockiertes Signal eintraf.

EFAULT — wenn die Werte nicht kopiert werden konnten.

Systemruf	creat	open	POSIX
	mkdir	mknod	4.3+BSD
	close		SVR4

Datei: fs/open.c
fs/namei.c

```
#include <linux/types.h>

long sys_close(unsigned int fd);
long sys_creat(const char *file_name, int mode);
long sys_mkdir(const char *file_name, int mode);
long sys_mknod(const char *file_name, int mode, dev_t dev);
long sys_open(const char *file_name, int flag, int mode);
```

`sys_open()` öffnet eine mit `file_name` angegebene Datei für die mit `flag` angegebenen Operationen. Die möglichen Werte für `flag` sind:

O_RDONLY — Die Datei wird nur zum Lesen geöffnet.

O_WRONLY — Die Datei wird nur zum Schreiben geöffnet.

O_RDWR — Lesen und Schreiben ist möglich.

O_CREAT — Die Datei wird angelegt, wenn sie nicht existiert. Dabei muss der dritte Parameter `mode` angegeben werden. `mode` wird dann mit `umask` verknüpft (`~umask & mode`).

O_EXCL — Es wird ein Fehler zurückgegeben, wenn `O_CREAT` angegeben ist und die Datei schon existiert (ermöglicht einen einfachen Lock-Mechanismus).

O_NOCTTY — Das in `file_name` stehende Terminal wird das *Controlling Terminal.*

O_TRUNC — Wenn die Datei existiert und schreibbar ist, wird sie auf die Größe 0 gesetzt.

O_APPEND — Die Daten bei nachfolgenden Schreiboperationen werden an die Datei angehängt.

O_NONBLOCK — Operationen auf der Datei blockieren nicht.

O_NDELAY — Ist auf `O_NONBLOCK` geerdet.

O_SYNC — Änderungen der Datei im (Puffer-)Speicher werden sofort auf das Gerät übertragen. Diese Option ist nur bei Blockgeräten und Dateien des *Ext2*-Dateisystems implementiert.

`sys_creat()` ist zwar als Systemruf vorhanden, der Kern ruft aber `sys_open()` mit den entsprechend gesetzten Flags auf.

```
asmlinkage int sys_creat(const char * pathname, int mode)
{
        return sys_open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
}
```

Die Funktion `sys_close()` schließt den Dateideskriptor `fd`. Eventuell vorhandene Dateisperren (`i_flock`) werden dabei gelöscht.

`sys_mkdir()` legt nach dem Test der Rechte mit Hilfe der Inode-Operation `mkdir()` das Verzeichnis `file_name` an. `sys_mknod()` legt eine Pseudodatei an. Dabei gibt `mode` den Typ sowie die Zugriffsrechte der zu erzeugenden Pseudodatei an. Verzeichnisse können mit dieser Funktion nicht, FIFOs nur von berechtigten Nutzern (`CAP_SYS_ADMIN`) angelegt werden. Für Gerätedateien enthält `dev` die Gerätenummer.

Implementierung

Alle fünf Systemrufe arbeiten mit dem Syscall-Makro. Der Systemruf *mkfifo* ist in der C-Bibliothek mittels `mknod()` implementiert:

```
int mkfifo(const char path, mode_t mode)
{
   return mknod (path, mode | S_IFIFO, 0);
}
```

Fehler

EMFILE — wenn zu viele Dateien offen sind.

EACCESS — wenn das Verzeichnis keine Ausführungsrechte hat.

ENFILE — wenn keine freien Dateideskriptoren für das System oder den Prozess zur Verfügung stehen. Beide Werte sind in `<linux/fs.h>` festgelegt.

EEXIST — wenn eine Datei angelegt werden soll, die schon als Verzeichnis existiert.

EISDIR — wenn ein Verzeichnis geöffnet werden soll, das nicht lesbar ist, oder wenn bei `sys_open()` die Flags `O_CREATE` bzw. `O_TRUNC` gesetzt sind.

ENOENT — wenn der Pfadname ungültig ist.

EPERM — wenn die Inode der Datei die ausgeführte Operation nicht erlaubt.

Systemruf pipe POSIX

Datei: arch/i386/kernel/sys_i386.c

```
int sys_pipe(unsigned long * fildes);
```

`sys_pipe()` erzeugt zwei Deskriptoren und schreibt sie in ein durch `filedes` adressiertes Feld. `filedes[0]` wird für Leseoperationen und `filedes[1]` für Schreiboperationen geöffnet (d.h. die Flags, die Operationen und der Modus werden entsprechend gesetzt). Voraussetzung ist, dass der Prozess noch zwei freie Deskriptoren zur Verfügung hat.

Implementierung

Der Systemruf wird über das Syscal-Makro umgesetzt. Da andere Architekturen die Stackregister als Argument für die Kernelfunktion verwenden, wurde die Funktion in das architekturabhängige Verzeichnis verschoben.

Fehler

EMFILE — wenn keine Dateideskriptoren mehr frei sind.

ENFILE — wenn keine File-Strukturen mehr frei sind.

EINVAL — wenn `filedes` ungültig ist.

Systemruf pivot_root Linux

Datei: fs/super.c

```
long sys_pivot_root(const char *new_root, const char *put_old)
```

Die Funktion `sys_pivot_root()` trägt `new_root` als neues Root-Verzeichnis des aktuellen Prozesses ein. Das alte Root-Verzeichnis wird nach `put_old` verschoben. Um diese Funktion aufrufen zu können, ist die Berechtigung `CAP_SYSADMIN` notwendig.

Für die Parameter `new_root` und `put_old` gelten folgende Bedingungen: Es müssen Verzeichnisse sein, `new_root` und `put_old` dürfen nicht in demselben Dateisystem liegen wie das aktuelle Root-Verzeichnis, und `put_old` muss im Dateisystem unterhalb von `new_root` liegen.

Ist das aktuelle Root-Verzeichnis kein Mount-Point (z. B. `/nfs/my_root`), dann wird nicht das Verzeichnis, sondern der Mount-Point nach `put_old` verschoben.

Zum Schluss werden bei allen Prozessen, deren Root oder deren aktuelles Verzeichnis auf `old_root` zeigen, dieses auf `new_root` gesetzt.

Fehler

EBUSY — wenn `new_root` oder `put_old` das Root-Verzeichnis des eigenen Prozesses ist.

EINVAL — wenn `put_old` nicht unterhalb von `new_root` liegt.

ENOENT — wenn `new_root` ein gelöschtes, aber noch offenes (ein totes) Verzeichnis ist oder wenn weder `new_root` noch `put_old` ein Verzeichnis ist.

EPERM — wenn ein nicht privilegierter Prozess die Funktion aufruft.

Systemruf poll SVR4

Datei: fs/select.c

```
#include <linux/time.h>
#include <linux/types.h>

long sys_poll(struct pollfd * ufds,
              unsigned int nfds, int timeout);
```

Dieser Systemruf ist das Äquivalent zu dem Systemruf `select()`. Anstatt jedoch drei Felder (für jede Bedingung eins) zu übergeben, wird ein Feld verwendet, das den Deskriptor und die Bedingungen aufnimmt.

```
struct pollfd {
        int fd;         /* Deskriptor                */
        short events;   /* zu überwachende Ereignisse */
        short revents; /* aufgetretene Ereignisse    */
};
```

Im Gegensatz zu `select()` bleiben die `events` unverändert. Das Pollen geschieht in einer Endlosschleife. In einer inneren Schleife wird für jede Datei die Dateioperation `f_op->poll` aufgerufen, das Ergebnis wird mit `events` verknüpft (ODER-Verknüpfung) und in `revents` eingetragen. Nach dem Durchlaufen der inneren Schleife wird geprüft, ob sich ein `revents` geändert hat (mittels Flag), ob der in `timeout` angegebene Timeout abgelaufen ist oder ob ein Signal vorliegt. Ist das der Fall, wird die Schleife unterbrochen und die Anzal der geänderten `revents` zurückgegeben. Ansonsten wird der Scheduler aufgerufen und die äußere Schleife erneut durchlaufen.

Als Timeout sind drei Einstellungen möglich:

`timeout < 0` — Der Timeout wird auf den Wert unendlich gesetzt.

`timeout == 0` — Der Ruf überprüft die Deskriptoren und kehrt sofort zurück.

`timeout > 0` — Der Wert wird von Millisekunden in `jiffies` umgerechnet und als `timeout` des Prozesses eingetragen.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EINTR — wenn ein nicht blockiertes Signal eintraf.

EFAULT — wenn die Werte nicht kopiert werden konnten.

Systemruf quotactl LINUX

Datei: kernel/dquot.c

```
#include <linux/sys.h>
#include <linux/quota.h>

long sys_quotactl(int cmd, const char *special,
           int id, caddr_t addr);
```

Diese Kernelfunktion stellt den Einsprungspunkt für das Quota-Programm dar. Im Moment werden nur Diskquotas berücksichtigt; für Prozessbeschränkungen ist eine Nutzung der `rlimits` wahrscheinlich der geeignete Weg.

Die Quotas haben folgendes Aussehen:

```
struct dquot {
  unsigned int dq_id;          /* für welche ID (uid, gid) */
  short dq_type;               /* Typ                      */
  kdev_t dq_dev;               /* Gerät                    */
```

```
  short dq_flags;              /* Flags                      */
  short dq_count;              /* Verweis-Zähler             */
  unsigned long dq_referenced; /* Anzahl der Zugriffe        */
  struct vfsmount *dq_mnt;     /* VFS-Mount-Point            */
  struct dqblk dq_dqb;         /* Benutzung                  */
  struct wait_queue *dq_wait;  /* Prozesse, die auf eine     */
                               /* Quota-Änderung warten      */
  struct dquot *dq_prev, *dq_next;
  struct dquot *dq_hash_prev, *dq_hash_next;
};

struct dqblk {
    __u32 dqb_bhardlimit; /* Hardlimit der belegbaren Blöcke */
    __u32 dqb_bsoftlimit; /* Softlimit der belegbaren Blöcke */
    __u32 dqb_curblocks;  /* aktuelle Blockanzahl            */
    __u32 dqb_ihardlimit; /* Hardlimit der belegbaren Inodes */
    __u32 dqb_isoftlimit; /* Softlimit der belegbaren Inodes */
    __u32 dqb_curinodes;  /* aktuelle Inode-Anzahl           */
    time_t dqb_btime;     /* Zeitbeschränkung für eine Soft- */
                          /* überschreitung (Blöcke)         */
    time_t dqb_itime;     /* Zeitbeschränkung für eine Soft- */
                          /* überschreitung (Inodes)         */
};
```

Die Zeitbeschränkungen haben Bedeutungen. Normalerweise und bei Abfragen erhält man den Zeitpunkt des Ablaufens der Beschränkung in Sekunden. Beim Setzen wird der Wert in die `vfsmount`-Struktur des Gerätes eingetragen und dient als Intervallwert beim Aktualisieren der Beschränkung (neuer Ablaufpunkt gleich aktuelle Zeit plus Intervall). Hinzu kommt noch eine Struktur zur Verwaltung:

```
struct dqstats {
   __u32 lookups;
   __u32 drops;
   __u32 reads;            /* Anzahl der gelesenen Quotas     */
   __u32 writes;           /* Anzahl der geschriebenen Quotas */
   __u32 cache_hits;
   __u32 pages_allocated;  /* Anzahl der belegten Seiten      */
   __u32 allocated_dquots; /* verwendete Quotas               */
   __u32 free_dquots;      /* freie Quotas                    */
   __u32 syncs;            /* Anzahl der sync-Operationen     */
};
```

Der Parameter `cmd` enthält das Kommando und den Typ des Aufrufs. Er kann mit dem Makro[17] `QCMD(cmd, type)` aufgebaut werden. Beim Abarbeiten wird `cmd` in `cmds` und `type` zerlegt. Ist kein `cmd` gegeben, wird `QUOTA_SYSCALL` eingetragen. Die Parameter haben (falls nicht anders beschrieben) folgende Bedeutung:

`cmds` — genaue Funktion des Rufes

`type` — Angabe, ob `id` eine UID oder GID ist bzw. der Index für das `dq_mnt`-Feld

17 Das Makro ist definiert als `(((cmd)<<SUBCMDSHIFT) | ((type) & SUBCMDMASK))`.

special — das gewünschte Gerät

id — gibt an, auf welche ID sich die Quotas beziehen sollen

Die meisten Funktionen können nur ausgeführt werden, wenn der Prozess das Recht CAP_SYS_RESOURCE besitzt. Ausgenommen davon sind nur Q_SYNC und Q_GETSTATS (diese Funktion kann jeder aufrufen) sowie Q_GETQUOTA (dazu muss die UID oder GID (je nach type) des aktuellen Prozesses gleich id sein).

Q_GETQUOTA — Der Ruf liefert die Quotas und ihre augenblickliche Benutzung. Ein nicht berechtigter Prozess darf nur seine eigenen Quotas abrufen.

Q_GETSTATS — addr ist ein Zeiger auf eine dqstats-Struktur. Eingetragen wird allerdings nur die Anzahl der verwendeten und freien Quotas.

Q_RSQUASH — Der Wert addr wird an der Stelle type in das Feld rsquash des Gerätes dev eingetragen.

Q_SYNC — Ein Sync auf die Liste der Quotas wird ausgeführt.

Q_QUOTAON — Die Quotas werden aktiviert. In addr steht der Name der Datei, in der die Quota-Werte (als dqblk-Struktur) gespeichert sind; type dient als Index für das mnt_quotas-Feld.

Q_QUOTAOFF — Schaltet die Quotas ab. Bei type=-1 werden alle mnt_quotas-Quotas für das Gerät abgeschaltet.

Die nächsten drei Funktionen werden in einer extra Unterfunktion behandelt. Zuerst wird versucht, die durch dev, id und type definierte dquot-Struktur zu finden (Hashfunktion). Sollte noch keine existieren, wird eine erzeugt und initialisiert, ansonsten wird cache_hits erhöht. addr wird als ein Zeiger auf eine dqblk-Struktur verwendet.

Q_SETQLIM — Ist id>0, werden alle Limits auf die übergebenen Werte gesetzt.

Q_SETUSE — Es werden die Werte curinodes und curblocks gesetzt. Bei einer Überschreitung des Softlimits werden auch die Zeitbeschränkungen aktualisiert.

Q_SETQUOTA — kombiniert die Wirkung der letzten beiden Flags.

Ist in der Unterfunktion id==0, werden die Expire-Zeiten für Blöcke und Inodes für die neue Struktur gesetzt (auf die übergebenen Werte).

Implementierung

Die C-Bibliothek enthält kein Syscall-Makro, dazu ist das Quota-Package zu verwenden.

Fehler

EINVAL — wenn type größer als MAXQUOTAS ist.

EPERM — wenn ein nicht berechtigter Prozess versucht, andere Quotas zu ändern oder ein privilegiertes Kommando aufzurufen.

ENOTBLK — wenn `special` kein Blockgerät ist.

ESRCH — wenn keine passende Quotastruktur gefunden wurde.

Systemruf	**write** **read**	POSIX

Datei: fs/read_write.c

```
#include <linux/types.h>

long sys_read(unsigned int fd, char * buf,unsigned int count);
long sys_write(unsigned int fd, char * buf,unsigned int count);
```

`sys_read()` versucht, `count` Bytes aus der Datei `fd` zu lesen. Die Bytes werden im Speicher `buf` abgelegt. Der Systemruf `sys_write()` arbeitet mit denselben Parametern, mit dem Unterschied, dass die Bytes in den Buffer geschrieben werden. Vorher wird versucht, den entsprechenden Bereich zu sperren (zu *locken*). Für ein obligatorisches Locking ist es wichtig, dass die Datei das Bit `IS_ISGID`, aber nicht `S_IXGRP` gesetzt hat.

Zurückgegeben wird die Anzahl der tatsächlich gelesenen bzw. geschriebenen Bytes, bei EOF oder nicht gelungenem Sperren eine 0 und eine negative Zahl im Fehlerfall. Aufgerufen werden letztendlich die entsprechenden File-Operationen.

Implementierung

Beide Systemrufe arbeiten über das Syscall-Makro.

Fehler

EBADF — wenn `fd` ungültig ist oder die Datei falsch geöffnet wurde.

EINVAL — wenn für die Datei keine Schreib- bzw. Leserechte gesetzt sind.

Systemruf	**readv** **writev**	POSIX

Datei: fs/read_write.c

```
#include <linux/types.h>
#include <linux/uio.h>
```

```
long sys_readv(unsigned long fd, const struct iovec * vector,
        long count)
long sys_writev(unsigned long fd, const struct iovec * vector,
        long count)
```

Beide Funktionen lesen (beziehungsweise schreiben) Daten über einen Dateideskriptor. Der Unterschied zu den bekannten Schreib-Lese-Funktionen besteht in der Art der übergebenen Puffer. Die Struktur `struct iovec` hat folgendes Aussehen:

```
struct iovec {
  void *iov_base;          /* Zeiger auf einen Speicherbereich */
  __kernel_size_t iov_len; /* Größe des Speicherbereichs       */
};
```

Die Anzahl der Puffer steht in `count`, wobei `UIO_MAXIOV` eine Obergrenze ist. Die Bereiche werden nacheinander auf Les- bzw. Schreibbarkeit geprüft, ihre Länge addiert und kontrolliert, ob es eine Sperre (`FLOCK`) für diesen Bereich gibt.

Bezieht sich `fd` auf einen Socket, werden die Socket-Operationen benutzt, ansonsten werden die File-Operationen verwendet. Bei `sys_readv()` werden die Puffer nacheinander durch Lesen aus dem Deskriptor gefüllt, bei `sys_write()` werden sie nacheinander in den Deskriptor geschrieben.

Wenn die Datei-Operationen (siehe Abschnitt 6.2.9) keine `readv()`- oder `writev()`-Operation anbieten, werden die Puffer nacheinander mit normalen `read()`- oder `write()`-Aufrufen abgearbeitet.

Zurückgegeben wird die Anzahl der insgesamt gelesenen bzw. geschriebenen Bytes, 0 bei EOF und eine negative Zahl im Fehlerfall.

Implementierung

Beide Systemrufe arbeiten über das Syscall-Makro.

Fehler

EBADF — wenn `fd` ungültig ist oder die Datei falsch geöffnet wurde.

EINVAL — wenn ein Parameter ungültig übergeben wurde.

Systemruf	**pread** **pwrite**	POSIX

Datei: fs/read_write.c

```
#include <linux/types.h>

long sys_pread(unsigned int fd, char * buf,
        size_t count, loff_t pos);
long sys_pwrite(unsigned int fd, const char * buf,
        size_t count, loff_t pos);
```

Beide Funktionen lesen (beziehungsweise schreiben) Daten über einen Dateideskriptor. Der Unterschied zu den bekannten Schreib-Lese-Funktionen besteht darin, dass nicht die aktuelle Position benutzt wird, sondern `pos`. Die aktuelle Position bleibt unverändert. Anders ausgedrückt: Der Ruf führt ein `lseek()` auf die neue Position aus, liest (oder schreibt) `count` Zeichen und führt zum Schluss ein `lseek()` auf die alte Position aus. Zurückgegeben wird die Anzahl der gelesenen bzw. geschriebenen Bytes, 0 bei EOF und eine negative Zahl im Fehlerfall.

Fehler

EBADF — wenn `fd` ungültig ist oder die Datei falsch geöffnet wurde.

EINVAL — wenn ein Parameter ungültig übergeben wurde.

Systemruf	**readdir**	POSIX
	getdents	Linux

Datei: fs/readdir.c

```
int old_readdir(unsigned int fd, void * dirent,
                unsigned int count)
long sys_getdents(unsigned int fd, void * dirent,
                  unsigned int count)
```

Die Funktion `old_readdir()` füllt die Struktur, auf die `dirent` zeigt, mit den Daten des Verzeichnisses `fd`. Der Parameter `count` wird ignoriert. Die Struktur hat folgenden Aufbau:

```
struct linux_dirent {
        unsigned long   d_ino;
        unsigned long   d_off;
        unsigned short  d_reclen;
        char            d_name[1];
};
```

Linux gibt den Aufruf an die Operationen des Virtuellen Dateisystems weiter, indem dieser Aufruf die entsprechende File-Operation aufruft (siehe Abschnitt 6.2.9).

Inzwischen gibt es die neue, verbesserte Funktion `sys_getdents()`. Dazu gehört auch eine neue Struktur:

```
struct getdents_callback {
        struct linux_dirent * current_dir;
        struct linux_dirent * previous;
        int count;
        int error;
};
```

Die Funktion liest mehrere Einträge, solange die Größe der Summe (diese berechnet sich aus dem Offset des aktuellen Namens und dessen Länge) den Wert `count` nicht übersteigt. Zurückgegeben wird die Differenz aus `count` und der Größe der gelesenen Einträge.

Implementierung

Der Systemruf `sys_readdir()` wird in `entry.S` auf `old_readdir()` abgebildet. Beide Systemrufe werden von der Bibliotheksfunktion `readdir()` verwendet. Der Ruf `old_readdir()` wird aufgerufen, wenn kein `sys_getdents()` verfügbar ist. Beide Kernelfunktionen werden direkt über den Interrupt 0x80 angesprungen.

Fehler

EBADF — wenn `fd` ungültig ist.

ENOTDIR — wenn keine File-Operation `sys_readdir()` existiert.

Systemruf readlink POSIX

Datei: fs/stat.c

```
long sys_readlink(const char *path, char *buf, int bufsize);
```

Die Funktion `sys_readlink()` liest den Pfad aus, auf den ein symbolischer Link verweist. Es werden maximal `bufsize` Zeichen in dem Puffer `buf` durch die entsprechende Inode-Operation abgelegt. An das Ende von `buf` wird kein Null-Bit (`'\0'`) angehängt. Die Funktion liefert dafür aber die Länge des Pfades zurück.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EINVAL — wenn `bufsize` negativ ist, der Aufruf vom Dateisystem nicht unterstützt wird oder `path` kein Verweis ist.

ENOENT — wenn `path` nicht existiert.

Systemruf select 4.3+BSD

Datei: fs/select.c

```
#include <linux/time.h>
#include <linux/types.h>

long sys_select(int n, fd_set *inp, fd_set *outp,
        fd_set *exp, struct timeval *tvp)
```

Die Funktion `sys_select()` erlaubt das Multiplexen von Ein- und Ausgabeoperationen. Das Zeitintervall wird in `jiffies` umgerechnet und als Timeout des Prozesses eingetragen. Der Prozess schläft nach dem Aufruf, bis einer der Deskriptoren in `inp`, `outp` oder `exp` verfügbar oder das Zeitintervall `tvp` abgelaufen ist. Die Deskriptoren sind dabei als Bitfelder angelegt. Der Parameter `n` enthält die Länge der Bitfelder.

Die Funktion gibt die Anzahl der verfügbaren Deskriptoren zurück. Wenn in `personality` das Flag `STICKY_TIMEOUTS` gesetzt ist, wird außerdem `tvp` aktualisiert. Für die Verwendung sind mehrere Makros definiert:

`FD_ZERO(`*`fdset`*`)` — löscht alle Bits in *fdset*.

`FD_CLR(`*`fd`*`,`*`fdset`*`)` — löscht den Deskriptor *fd* in *fdset*.

`FD_SET(`*`fd`*`,`*`fdset`*`)` — setzt den Deskriptor *fd* in *fdset*.

`FD_ISSET(`*`fd`*`,`*`fdset`*`)` — testet den Deskriptor *fd* in *fdset*. Gibt einen Wert ungleich 0 zurück, wenn *fd* gesetzt ist.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EBADF — wenn sich in einem der Felder ein ungültiger Deskriptor befindet.

EINTR — wenn ein nicht blockiertes Signal eintraf.

EINVAL — wenn `n` negativ ist.

ENOMEM — wenn im Kern nicht genug Speicher für interne Tabellen frei ist.

Systemruf	stat	newstat
	fstat	newlstat
	lstat	newfstat

Datei: fs/stat.c

```
#include <linux/stat.h>

long sys_stat(const char *file_name, struct old_stat *buf);
long sys_fstat(unsigned int fd, struct old_stat *buf);
long sys_lstat(const char *file_name, struct old_stat *buf);
```

`sys_stat()`, `sys_fstat()` und `sys_lstat()` geben eine gefüllte Datenstruktur zurück, die Informationen über die angegebene Datei enthält. Inzwischen gibt es eine neue Struktur (`stat`), die in `<asm/stat.h>` definiert ist. Die alte Struktur enthält nicht die Füllbits und die Werte `st_blocksize` und `st_blocks`. Mit dieser neuen Struktur arbeiten die Funktionen mit „`new_`“.

```
struct stat {
   unsigned short st_dev;     /* Gerät                */
   unsigned short __pad1;     /* Füllgröße            */
   unsigned long  st_ino;     /* Inode                */
   unsigned short st_mode;    /* Zugriffsrechte       */
   unsigned short st_nlink;   /* Anzahl der Hardlinks */
   unsigned short st_uid;     /* UID des Besitzers    */
   unsigned short st_gid;     /* GID des Besitzers    */
   unsigned short st_rdev;    /* Gerätetyp            */
   unsigned short __pad2;     /* Füllgröße            */
   unsigned long  st_size;    /* Größe in Bytes       */
   unsigned long  st_blksize; /* Blockgröße           */
   unsigned long  st_blocks;  /* belegte Blöcke       */
   unsigned long  st_atime;   /* letzte Zugriffszeit  */
   unsigned long  __unused1;  /* Füllgröße            */
   unsigned long  st_mtime;   /* letzte Änderungszeit (Datei) */
   unsigned long  __unused2;  /* Füllgröße            */
   unsigned long  st_ctime;   /* letzte Änderungszeit (Inode) */
   unsigned long  __unused3;  /* Füllgröße            */
   unsigned long  __unused4;  /* Füllgröße            */
   unsigned long  __unused5;  /* Füllgröße            */
   unsigned int   st_gen;     /* Füllgröße            */
};
```

`sys_stat()` gibt die Daten für die Datei `file_name` zurück. Für Verweise gibt es `sys_lstat()`; diese Funktion gibt die Daten für den symbolischen Link selbst zurück. `sys_fstat()` ist identisch mit `sys_stat()`, es verwendet jedoch statt des Namens einen Deskriptor `fd`.

Alle drei Aufrufe ermitteln die Inode des übergebenen Objekts und rufen die Kernelfunktion `cp_old_stat()` auf. Diese liest die meisten Daten einfach aus dem Inode. Die neueren Funktionen verwenden dazu `cp_new_stat()`. Wenn das Dateisystem `st_blocks` und `st_blksize` nicht unterstützt, werden sie mit Hilfe eines einfachen Algorithmus ermittelt.

Implementierung

Die Systemrufe werden über das Syscall-Makro umgesetzt.

Fehler

EBADF — wenn `fd` ungültig ist.

ENOENT — wenn `file_name` nicht existiert.

Systemruf	**statfs**	SVR4
	fstatfs	

Datei: fs/open.c

```
#include <linux/vfs.h>

long sys_statfs(const char *path, struct statfs *buf);
long sys_fstatfs(unsigned int fd, struct statfs *buf);
```

Die Funktion `sys_statfs()` gibt die Informationen des Dateisystems zurück, auf dem sich die Datei `path` befindet. Bei `sys_fstatfs()` wird statt des Namens ein Deskriptor angegeben. Die Struktur `buf` ist in der architekturabhängigen Datei `statfs.h` definiert:

```
struct statfs {
        long    f_type;     /* Typ des Dateisystems         */
        long    f_bsize;    /* optimale Blockgröße          */
        long    f_blocks;   /* Anzahl der Blöcke            */
        long    f_bfree;    /* Gesamtzahl der freien Blöcke */
        long    f_bavail;   /* freie Blöcke für Nutzer      */
        long    f_files;    /* Anzahl der Inodes            */
        long    f_ffree;    /* Anzahl der freien Inodes     */
        fsid_t  f_fsid;     /* ID des Dateisystems          */
        long    f_namelen;  /* maximale Dateinamenlänge     */
        long    f_spare[6]; /* nicht benutzt                */
};
```

Felder, die im Dateisystem nicht definiert sind, werden mit −1 gefüllt. Die Daten werden mit den Superblock-Operationen ausgelesen (siehe Abschnitt 6.2.3).

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EBADF — wenn `fd` kein gültiger Deskriptor ist.

EFAULT — wenn `buf` auf eine ungültige Adresse zeigt.

ENOSYS — wenn keine Superblock-Operation vorhanden ist.

Systemruf	**sync**		SVR4
	fsync	**fdatasync**	4.3BSD

Datei: fs/filemap.c

```
long sys_sync(void);
long sys_fsync(unsigned int fd);
long sys_fdatasync(unsigned int fd);
long sys_msync(unsigned long start, size_t len, int flags);
```

`sys_sync()` schreibt alle im Speicher gehaltenen Informationen wie Puffer, Superblöcke und Inodes auf die Platte. Diese Funktion gibt immer 0 zurück. Sie arbeitet über `fsync_dev()` mit einer 0 als Parameter. Das bedeutet, dass alle Blockgeräte synchronisiert werden sollen.

Die Funktion `sys_fsync()` schreibt die im Speicher gehaltenen Daten der Datei `fd` zurück. Dazu ruft sie die File-Operation `fsync()` auf und sperrt währenddessen den Zugriff auf die Datei über den Inode-Semaphor. Die Funktion `sys_fdatasync()` hat die gleiche Funktionalität wie `sys_fsync()`, nur der Semaphor wird nicht gesetzt.

Die Funktion `sys_msync()` synchronisiert die Dateien, die in den angegebenen Speicherbereich eingeblendet sind. Zuerst werden die eingeblendeten Bereiche synchronisiert (Aufruf von `vm_ops->sync`). Dabei wird, wenn das Flag `MS_INVALIDATE` nicht gesetzt ist, die Datei Speicherseite für Speicherseite zurückgeschrieben. Ist dann das Flag `MS_SYNC` gesetzt, wird zuletzt noch der DEntry der Datei synchronisiert.

Implementierung

Beide Systemrufe werden über das Syscall-Makro umgesetzt.

Fehler

EBADF — wenn sich für `fd` keine Datei ermitteln lässt.

EINVAL — wenn die Datei keine Datei-Operationen oder keine Sync-Operation besitzt.

EFAULT — wenn das Intervall [`start`,`start`+`len`] Bereiche enthält, in die keine Datei eingeblendet ist (für den Rest wird trotzdem ein Sync ausgeführt).

Systemruf sysfs SVR4

Datei: fs/super.c

```
long sys_sysfs(int option, unsigned long arg1,
        unsigned long arg2);
```

Die Funktion `sys_sysfs()` gibt Informationen über die dem Kern bekannten Dateisysteme zurück. Dazu wird die Liste `file_systems` ausgelesen: Im Argument `option` wird die gewünschte Funktion angegeben:

1 — Der Ruf liefert den Index des angegebenen Dateisystems. `arg1` enthält dessen Namen.

2 — Es wird der Name des angegebenen (des `index`-ten) Dateisystems zurückgegeben. `arg1` ist der Index, in `arg2` wird der Name abgelegt.

3 — Der Ruf gibt die Anzahl der bekannten Dateisysteme zurück.

Implementierung

Es gibt noch keine Schnittstelle in der C-Bibliothek.

Fehler

EINVAL — wenn `option`, `index` oder `name` ungültig sind.

Systemruf truncate, ftruncate 4.3+BSD

Datei: fs/open.c

```
long sys_truncate(const char *path, unsigned long len);
long sys_ftruncate(unsigned int fd, unsigned long len);
```

`sys_truncate()` kürzt oder verlängert die Datei `path` auf die Größe `len` Byte. `sys_ftruncate()` führt die gleiche Operation für die hinter `fd` stehende Datei durch.

Dafür ist Voraussetzung, dass es sich nicht um ein Verzeichnis handelt und dass die Inode-Flags eine Änderung (Verkürzung) zulassen. Auf dem zu ändernden Bereich darf auch kein Lock liegen.

Die Quotas der Inode werden aktualisiert, die entsprechende Inode-Operation wird ausgeführt, und `ctime` wird aktualisiert. Ist die Datei in den Speicher gemappt, wird der entsprechende Speicherbereich auch geändert.

Implementierung

Beide Systemrufe werden über das Syscall-Makro umgesetzt.

Fehler

EACCES — wenn die Datei `path` keine Schreibrechte hat oder ein Verzeichnis ist.

EROFS — wenn die Datei sich in einem `IS_RDONLY`-Dateisystem befindet.

EPERM — wenn die Datei sich in einem `IS_IMMUTABLE`- bzw. `IS_APPEND`-Dateisystem befindet.

EBADF — wenn bei `sys_ftruncate()` der Deskriptor ungültig ist.

ENOTDIR — wenn ein Bestandteil von `path` kein Verzeichnis ist.

ENOENT — wenn die Datei nicht existiert.

Systemruf uselib — Linux

Datei: fs/exec.c

```
long sys_uselib(const char *library);
```

`sys_uselib()` wählt eine *Shared Library* für den aktuellen Prozess aus. Dazu wird die Datei mit `sys_open()` geöffnet. Dann wird versucht, für alle registrierten Binärformate die Operation `load_shlib()` auszuführen. Der erste erfolgreiche Versuch beendet den Ruf. Wichtig ist dabei, dass für die Datei `library` Lese- und Ausführungsrechte gesetzt sind.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

ENOEXEC — wenn kein passendes Binärformat oder keine passende Ladefunktion für `library` gefunden wurde.

EACCES — wenn `library` nicht lesbar ist.

Systemruf umask POSIX

Datei: kernel/sys.c

```
int sys_umask(int mask);
```

`sys_umask()` setzt die Maske für die Zugriffsrechte einer Datei. Als neue Maske wird der Wert `mask & S_IRWXUGO` (0777) verwendet. Der alte Wert wird zurückgegeben. Die Maske wird in `open_namei()` bei der Erzeugung einer Datei verwendet. Der dort angegebene `mode` wird mit `umask` überlagert:

```
mode &= S_IALLUGO & ~current->fs->umask;
```

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Systemruf ustat Linux

Datei: fs/super.c

```
long sys_ustat(dev_t dev, struct ustat * ubuf);
```

Der Ruf füllt die Struktur `ustat` mit den Informationen des Superblocks, der als Gerät `dev` eingetragen hat. Diese Struktur ist architekturabhängig, ihr i386-Pendat hat folgenden Aufbau:

```
struct ustat {
        long f_type;     /* Magic Number des Dateisystems */
        long f_bsize;    /* Blockgröße                    */
        long f_blocks;   /* Anzahl der Blöcke             */
        long f_bfree;    /* Anzahl der freien Blöcke      */
        long f_bavail;   /* Anzahl der freien und nicht   */
                         /* reservierten Blöcke           */
        long f_files;    /* Anzahl der Dateien            */
        long f_ffree;    /* Anzahl der freien Inodes      */
        __kernel_fsid_t f_fsid; /* ID (bisher ungenutzt)  */
        long f_namelen;  /* maximale Länge eines Namens   */
        long f_spare[6]; /* ungenutzt                     */
};
```

Implementierung

Beide Systemrufe verwenden das Syscall-Makro.

Fehler

EINVAL — wenn `dev` kein gültiges Gerät ist.

ENOSYS — wenn es für den Superblock keine Operation `statfs()` gibt.

Systemruf utime — POSIX

Datei: fs/open.c
fs/attr.c

```
#include <utime.h>

long sys_utime(char *filename, struct utimbuf *buf);
```

`sys_utime()` setzt die Zeitstempel der Datei `filename` auf die in `buf` angegebenen Werte. Die Struktur `buf` ist wie folgt definiert:

```
struct utimbuf {
        time_t actime;
        time_t modtime;
};
```

Beide Zeitangaben sind UNIX-Sekunden, entsprechend `sys_time()`. Bei der Ausführung wird `sys_utime()` auf `CURRENT_TIME` gesetzt. Ist `buf` NULL, werden alle Werte auf (`CURRENT_TIME`) gesetzt. Die Werte werden nur gesetzt, wenn die UID der Inode gleich der FSUID des Prozesses ist oder das Recht `CAP_FOWNER` gesetzt ist.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

ENOENT — wenn die Datei `filename` nicht existiert.

EROFS — wenn das Dateisystem der Datei als *read only* gekennzeichnet ist.

EACCES — wenn die Inode der Datei nicht geändert werden darf.

Systemruf vhangup — Linux

Datei: fs/open.c

```
long sys_vhangup(void);
```

`sys_vhangup()` führt ein *Hangup* für das aktuelle Terminal durch. Der Aufruf wird zum Beispiel von `init` benutzt, um den Anwendern ein sauberes Login-Terminal zur Verfügung zu stellen, auf dem keine Prozesse mehr arbeiten. Bei Erfolg gibt der Ruf immer 0 zurück. Zum Aufruf muss der Prozess das Recht `CAP_SYS_TTY_CONFIG` besitzen.

Die aufgerufene Funktion `tty_vhangup()` ist in `drivers/tty_io.c` implementiert. Darin werden alle Prozesse, die mit dem Terminal arbeiten, aufgeweckt, die laufende Sitzung wird abgebrochen und der entsprechende `tty`-Wert aller Prozesse auf -1 gesetzt. Die Funktion wird nur ausgeführt, wenn der aktuelle Prozess überhaupt ein Terminal besitzt (`current->tty`).

Das V im Namen des Systemrufs steht für *virtuell*. Das heißt aber nicht, dass ein Hangup nur simuliert wird, sondern dass dieser Aufruf für die virtuellen Terminals verwendet wird.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EPERM — wenn ein nicht berechtigter Prozess `sys_vhangup()` aufruft.

A.3 Die Kommunikation

Für die Kommunikation gibt es nur zwei Systemrufe. Das mag zwar auf den ersten Blick verwirren, aber die volle Schönheit der gebräuchlichen Systemaufrufe steht als Bibliotheksfunktionen zur Verfügung.

Ursprünglich war es vermutlich die Absicht, durch diese Zusammenfassung die Implementierung einfacher zu machen. Wenn es allerdings darum geht, die richtigen Parameter vom Stack zu lesen, wird es schnell undurchsichtig. Natürlich gibt es die gewohnten Systemrufe wie *semget* als Bibliotheksfunktionen.

Systemruf ipc LINUX

Datei: arch/i386/kernel/sys_i386.c

```
#include <linux/ipc.h>

int sys_ipc (uint call, int first, int second,
             int third, void *ptr, long fifth);
```

`sys_ipc()` erlaubt die vollständige Verwendung der SVR4-Interprozesskommunikation mit Hilfe eines Aufrufs. Aufgrund einiger Unterschiede zwischen den einzelnen Architekturen ist dieser Ruf in das `arch`-Verzeichnis gewandert.

Alle Systemrufe, die mit Messagequeues, Shared Memory oder Semaphoren arbeiten, werden auf diesen Systemruf abgebildet. Da die Implementierung des Rufes allerdings nicht gerade optimal ist, finden sich mittlerweile die einzelen Rufe als schon fertig implementierte Kernelfunktionen im Verzeichnis `ipc/`; allerdings sind sie noch nicht nach oben freigeschaltet.

Der Parameter `call` legt die genaue Funktion fest. Dabei kann mit dem Makro `IPCCALL(version,call)` noch eine Versionsnummer in `call` versteckt werden. Die `call`-Werte sind in `<asm/ipc.h>` definiert:

`SEMOP` — Funktion entspricht `semop()`.

`SEMGET` — Funktion entspricht `semget()`.

`SEMCTL` — Funktion entspricht `semctl()`.

`MSGSND` — Funktion entspricht `msgsnd()`.

`MSGRCV` — Funktion entspricht `msgrcv()`. Ist `version` gleich 0 wird von einer alten `msgbuf`-Struktur ausgegangen, und die Parameter werden konvertiert.

`MSGGET` — Funktion entspricht `msgget()`.

`MSGCTL` — Funktion entspricht `msgctl()`.

`SHMAT` — Funktion entspricht `shmat()`. Bei `version` 1 wird vom iBCS2 ausgegangen und vorher getestet, ob FS und DS auf dasselbe Segment zeigen.

`SHMDT` — Funktion entspricht `shmdt()`.

`SHMGET` — Funktion entspricht `shmget()`.

`SHMCTL` — Funktion entspricht `shmctl()`.

Durch die Angabe von `call` werden die restlichen Parameter festgelegt. Wenn bei der Konfiguration des Kerns kein `CONFIG_SYSVIPC` eingestellt wurde, gibt dieser Ruf `-ENOSYS` zurück.

Implementierung

Die Bibliothek stellt natürlich die gewohnten Funktionen der Interprozesskommunikation zur Verfügung. Sie werden auf die Bibliotheksfunktion `ipc()` abgebildet, die über das Syscall-Makro definiert ist. Als Beispiel sei hier die Implementierung des Systemrufs *semget()* angegeben:

```
int semget (key_t key, int nsems, int semflg)
{
        return __ipc (SEMGET, key, nsems, semflg, NULL);
}
```

Fehler

EINVAL — wenn für `call` ein ungültiger Wert angegeben wird.

Systemruf socketcall LINUX

Datei: net/socket.c

```
#include <linux/socketcall.h>

long sys_socketcall(int call, unsigned long *args);
```

So wie es genau einen Systemruf für die SVR4-IPC gibt, gibt es einen Ruf, der die komplette Programmierung der Sockets-Schnittstelle ermöglicht. Bei der Funktion `sys_socketcall()` können Sie durch die Angabe eines Parameters alle gewohnten Rufe als Bibliotheksfunktionen implementieren. Allerdings zeigt die Programmierung der Umsetzung des Parameters `args` auch sehr schön die Gegensätze zwischen optimiertem und lesbaren Code.

Der Parameter `call` legt die genaue Funktionalität fest. Definiert sind folgende Makros in `<linux/net.h>`:

`SYS_SOCKET` — Funktion entspricht `socket()`.

`SYS_BIND` — Funktion entspricht `bind()`.

`SYS_CONNECT` — Funktion entspricht `connect()`.

`SYS_LISTEN` — Funktion entspricht `listen()`.

`SYS_ACCEPT` — Funktion entspricht `accept()`.

`SYS_GETSOCKNAME` — Funktion entspricht `getsockname()`.

`SYS_GETPEERNAME` — Funktion entspricht `getpeername()`.

`SYS_SOCKETPAIR` — Funktion entspricht `socketpair()`.

`SYS_SEND` — Funktion entspricht `send()`.

`SYS_RECV` — Funktion entspricht `recv()`.

`SYS_SENDTO` — Funktion entspricht `sendto()`.

`SYS_RECVFROM` — Funktion entspricht `recvfrom()`.

`SYS_SHUTDOWN` — Funktion entspricht `shutdown()`.

`SYS_SETSOCKOPT` — Funktion entspricht `setsockopt()`.

`SYS_GETSOCKOPT` — Funktion entspricht `getsockopt()`.

SYS_SENDMSG — Funktion entspricht `sendmsg()`.

SYS_RECVMSG — Funktion entspricht `recvmsg()`.

Entsprechend der Angabe von `call` sind in `args` die notwendigen Parameter zu setzen.

Implementierung

Ebenso wie die IPC-Aufrufe werden auch die bekannten Socketfunktionen über die C-Bibliothek aufgerufen. Als Beispiel soll `socket()` vorgestellt werden; die anderen Funktionen sind im Prinzip genauso implementiert.

```
static inline
_syscall2(long,socketcall,int,call,unsigned long *,args);

int sys_socket(int family, int type, int protocol)
{
        unsigned long args[3];
        args[0] = family; args[1] = type; args[2] = protocol;
        return socketcall(SYS_SOCKET, args);
}
```

Fehler

EINVAL — wenn für `call` ein ungültiger Wert angegeben wird.

A.4 Die Speicherverwaltung

Als nächste Gruppe werden die Systemrufe zur Speicherverwaltung beschrieben. Hier gibt es nur wenige Aufrufe, obwohl dieses Gebiet eines der wichtigsten bei Multitaskingsystemen ist. Allerdings ist die mit der Verwaltung verbundene Arbeit nicht ganz unkompliziert, und je weniger (störende) Einflüsse vorliegen, desto sicherer läuft das System.

Systemruf madvise LINUX

Datei: mm/filemap.c

```
#include <linux/mman.h>

long sys_madvise(unsigned long start, size_t len,
                 int behavior)
```

Mit diesen Funktionen kann eine Anwendung das Verwalten ihrer Virtuellen Speicherbereiche beeinflussen. So kann z.B. eingestellt werden, dass bei einem Zugriff der gesamte Bereich im Voraus gelesen werden soll. Diese Einstellungen betrachtet der Kern jedoch

nur als Hinweise — allerdings wird der korrekte Ablauf nicht beeinträchtigt, falls eine andere Strategie zum Einsatz kommen sollte. Der Speicherbereich ist durch die Startadresse `start` und die Größe `len` adressiert und in `behavior` steht die gewünschte Strategie. Mögliche Werte dafür sind:

`MADV_NORMAL` — Bei einem Zugriff soll der gesamte Cluster gelesen werden, was zu einem gewissen Read-Ahead und Read-Behind führt. Das ist das Normalverhalten.

`MADV_RANDOM` — Bei einem Zugriff soll das System nur die angeforderten Daten lesen. Dieses Verhalten ist günstig, wenn es unwahrscheinlich ist, dass die Anwendung noch mehr Daten liest.

`MADV_SEQUENTIAL` — Bei einem Zugriff werden alle Seiten des Bereichs auf einmal eingelesen und danach sobald wie möglich freigegeben.

`MADV_WILLNEED` — Die Anwendung weist den Kern an, einige Seiten im Voraus zu lesen. Das ist nur möglich, wenn sich der Bereich auf eine eingeblendete Datei bezieht. Die Lese-Anforderungen werden nur generiert, so dass der Ruf nicht blockiert.

`MADV_DONTNEED` — Die Anwendung greift nicht mehr auf den Bereich zu; der Kern kann also die zugehörigen Ressourcen freigeben. Der Kern „wirft" die Ressourcen weg. Wer ein gesichertes Wegschreiben benötigt (*dirty pages*), sollte `sys_msync()` verwenden.

Als Umsetzung der zweiten und dritten Strategie wird ein Flag in die `vm_flags` der Virtuellen Speicherbereiche eingetragen.

Fehler

EACCES — wenn `start` plus `len` kleiner als 0 oder `behavior` ungültig ist oder wenn die Anwendung versucht, gesperrte bzw. geteilte Seiten freizugeben.

ENOMEM — wenn eine Adresse im angegebenen Bereich ist nicht gemappt ist oder überhaupt nicht zum Adressraum des Prozesses gehört.

EIO — wenn beim Ein- oder Auslagern der Seiten ein Fehler auftrat.

EBADF — wenn der Virtuelle Speicherbereich existiert, sich aber nicht auf eine Datei bezieht.

EAGAIN — wenn eine Kern-Ressource gerade nicht verfügbar ist.

Systemruf mincore LINUX

Datei: mm/filemap.c

```
asmlinkage long sys_mincore(unsigned long start, size_t len,
        unsigned char * vec);
```

`sys_mincore()` ermittelt den Status der Speicherseiten des aktuellen Prozesses. Die zu untersuchenden Seiten sind durch das Adress-Intervall `[start, start + len]` gegeben. Ihr Status wird in dem Vector `vec` abgelegt. Ist das unterste Bit eines Bytes 1, so befindet sich die Seite bereits im Speicher, genauer gesagt: Sie befindet sich im Page-Cache und ist gültig. Die Adresse `start` muss eine Seitengrenze sein.

Da sich der Zustand der Seiten durch parallele Speicheranforderungen ändern kann, sind die Informationen nicht langfristig gültig. Eine Garantie kann nur für gesperrte Seiten gegeben werden.

Fehler

EFAULT — wenn `vec` auf eine ungültige Adresse zeigt.

EINVAL — wenn `addr` nicht auf einer Seitengrenze liegt oder `len` kleiner als 0 ist.

ENOMEM — wenn der Bereich `[addr, addr + len]` für den Prozess ungültige Adressen oder nicht eingeblendete Bereiche enthält.

EAGAIN — wenn das Abspeichern des Resultats in `vec` aufgrund der Speicheranforderung fehlschlug.

Systemruf	**mmap** **munmap**	4.3+BSD

Datei: arch/i386/kernel/sys_i386.c
mm/mmap.c

```
#include <linux/types.h>
#include <linux/mman.h>

long old_mmap(struct mmap_arg_struct *arg);
long sys_mmap2(unsigned long addr, unsigned long len,
               unsigned long prot, unsigned long flags,
               unsigned long fd, unsigned long pgoff);
long sys_munmap(unsigned long addr, size_t len);
```

`old_mmap()` blendet eine Datei in den Speicher ein. Das `old`-Präfix liegt in der Art der Parameterübergabe begründet. Dabei wird der Funktion die Adresse `buffer` eines Feldes von Werten als Parameter übergeben. Dabei entspricht:

```
struct mmap_arg_struct {
  unsigned long addr;   /* Adresse im Hauptspeicher     */
  unsigned long len;    /* Größe des Bereichs           */
  unsigned long prot;   /* einzutragende Zugriffsrechte */
  unsigned long flags;  /* einzutragende Flags          */
  unsigned long fd;     /* Dateideskriptor              */
```

```
  unsigned long offset; /* Offset in der Datei          */
};
```

Als Zugriffsrechte sind folgende Werte möglich:

PROT_EXEC — Seiten können ausgeführt werden.

PROT_READ — Seiten können gelesen werden.

PROT_WRITE — Seiten können beschrieben werden.

Der Parameter `flags` gibt den Typ und die Behandlung der Speicherseiten an; die letzten vier Flags sind linux-spezifisch.

MAP_FIXED — Es muss genau die angegebene Adresse benutzt werden. Dabei muss `addr` ein Vielfaches der Seitengröße sein.

MAP_PRIVATE — Änderungen wirken sich nur im Speicher aus.

MAP_SHARED — Änderungen im Speicher wirken sich auch auf die Datei aus.

MAP_ANONYMOUS — Es wird keine Datei eingeblendet.

MAP_GROWSDOWN — Der Speicherbereich wird nach unten orientiert (Stack).

MAP_DENYWRITE — Schreibzugriffe auf die Datei liefern den Fehler `-ETXTBSY`.

MAP_EXECUTABLE — Der geblendete Speicherbereich wird als Bibliothek gekennzeichnet.

MAP_LOCKED — Der Speicherbereich ist blockiert.

MAP_NORESERVE — Es wird vor dem Mappen nicht getestet, ob genügend Speicher frei ist.

Vor der eigentlichen Arbeit wird (außer bei `MAP_ANONYMOUS`) der Dateideskriptor überprüft, und die Flags `MAP_EXECUTABLE` und `MAP_DENYWRITE` werden ausgeblendet, bevor die eigentliche Funktion `do_mmap2()` aufgerufen wird. Der Aufruf `sys_mmap()` ist dazu gleichwertig, nur dass hier die Parameter einzeln übergeben werden.

Mit der Funktion `sys_unmap()` werden die eingeblendeten Seiten wieder ausgeblendet und der verwendete Speicher wird freigegeben.

Implementierung

Der Systemruf `sys_mmap()` wird bei Intel-Systemen auf `old_mmap()` abgebildet. Der Systemruf `munmap()` verwendet einfach das Syscall-Makro.

Fehler

EACCES — wenn die in den Flags angegebenen Werte nicht mit den Rechten der angegebenen Datei übereinstimmen.

EBADF — wenn keine Datei geöffnet ist.

EINVAL — wenn ein ungültiger Wert für die Flags angegeben wurde, die Summe aus Adresse und Länge größer ist als der erlaubte Prozessspeicher oder wenn `MAP_FIXED` angegeben wurde und `addr` keine Seitengrenze ist.

ENOMEM — wenn die Adresse (bei `MAP_FIXED`) nicht verfügbar ist.

Systemruf mprotect LINUX

Datei: mm/mprotect.c

```
#include <linux/types.h>
#include <linux/mman.h>

long sys_mprotect(unsigned long addr, size_t len,
        unsigned long prot);
```

Mit dieser Funktion gibt es die Möglichkeit, den Zugriffsschutz für eingeblendete Bereiche auch nachträglich zu ändern. Der Bereich wird mit der Adresse `addr`, die auf einer Seitengrenze liegen muss und der Größe `len` angesprochen. Für `prot` sind die Werte `PROT_READ` (für das Lesen), `PROT_WRITE` (für das Schreiben) und `PROT_EXEC` (zum Ausführen) möglich.

Da der gesuchte Bereich innerhalb der (zum Prozess gehörenden) `vma`-Bereiche liegen muss, wird zuerst der Bereich ermittelt, der die Startadresse enthält. Dort werden die Rechte neu eingetragen, wobei die alten überschrieben werden. Anschließend werden über die verkettete Liste alle weiteren Bereiche geändert, bis die gesamte Länge `len` erreicht ist.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EFAULT — wenn kein passender `vma`-Bereich existiert.

EBADF — wenn ein Parameter ungültig ist.

EACCES — wenn `prot` einen ungültigen Wert enthält.

EINVAL — wenn die Summe aus `addr` und `len` größer als der erlaubte Prozessspeicher ist.

Systemruf mremap

Datei: mm/mremap.c

```
#include <linux/mman.h>

unsigned long sys_mremap(unsigned long addr,
        unsigned long old_len, unsigned long new_len,
        unsigned long flags)
```

Die Größe eines eingeblendeten Speicherbereichs kann mit dieser Funktion verändert werden. Die Startadresse `addr` muss dabei auf einer Seitengrenze liegen. `old_len` gibt die alte, `new_len` die neue Größe an. Der alte Bereich darf nicht die Grenze eines Virtuellen Speicherbereichs überschreiten.

Bei einer Verkleinerung wird der frei gewordene Bereich mittels `do_nmap()` ausgeblendet. Bei der Vergrößerung eines gesperrten Speicherbereichs dürfen die Prozessressourcen nicht überschritten werden. Ist noch genügend Platz bis zum Beginn des nächsten Virtuellen Speicherbereichs frei, wird der alte Bereich erweitert. Ist als Flag nicht `MREMAP_MAYMOVE` gesetzt und würde das Erweitern den nächsten VMA-Bereich überschneiden, wird ein Fehler zurückgegeben, ansonsten wird ein neuer Bereich in der gewünschten Größe angelegt, der alte kopiert und freigegeben. Der Rückgabewert ist die (eventuell neue) Adresse des Speicherbereiches.

Implementierung

Die C-Bibliothek stellt keine Schnittstelle zur Verfügung.

Fehler

EINVAL — wenn `addr` nicht auf einer Seitengrenze liegt.

EFAULT — wenn sich für die angegebene Adresse kein Virtueller Speicherbereich findet lässt, oder wenn sich die Größe über mehr als einen Bereich erstreckt.

EAGAIN — wenn die Prozessressource `RLIMIT_MEMLOCK` überschritten wird.

ENOMEM — wenn für eine Erweiterung nicht genügend Speicher zur Verfügung steht.

Systemruf msync POSIX

Datei: mm/filemap.c

```
long sys_msync(unsigned long start, size_t len,
        int flags)
```

Die Funktion ändert die Flags für einen gemappten Speicherbereich. `start` ist sein Anfang, `len` seine Größe. Sollte sich darin ein nicht gemappter Bereich befinden, wird das mit einer Fehlermeldung quittiert. Da dieser Ruf die lokalen Eigenheiten des Speichermanagements berücksichtigt, ist er *sehr* architekturabhängig. Als Flag können die folgenden Werte angegeben werden:

MS_SYNC — Es wird ein Rückschreiben des Cachepuffers ausgeführt. Dabei werden Änderungen im Kernelsegment auf das Nutzersegment übertragen.

MS_ASYNC — Es wird ein Rückschreiben des Cachepuffers ausgeführt und zusätzlich ein `sync()`-Aufruf an die Inode der eingemappten Inode abgesetzt.

MS_INVALIDATE — Es wird ein Rückschreiben des Cachepuffers ausgeführt, und die belegten Bereiche werden danach freigegeben.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

Fehler

EINVAL — wenn einer der übergebenen Parameter ungültig oder ein Teil des angegebenen Bereiches nicht gemappt ist.

EFAULT — wenn sich zu `start` kein VMA finden lässt.

Systemruf sendfile POSIX

Datei: mm/filemap.c

```
#include <linux/capability.h>

long sys_sendfile(int out_fd, int in_fd,
        off_t *offset, size_t count);
```

Wenn man Daten von einer Datei in eine andere Datei[18] schreiben will, ist diese Funktion einem kombinierten read/write-Aufruf überlegen, da der Aufruf vollständig im Kern abläuft und deswegen keine Daten zwischen den verschiedenen Adressräumen kopiert werden.

Die Funktion liest `count` Bytes aus der Datei `in_fd` von der Position `offset` und schreibt sie in `out_fd` an die aktuelle Position. Ansonsten treffen die normalen Schreib- bzw. Lesebeschränkungen zu (keine Sperren etc.). Wegen der gleichen Adressräume wird (im Unterschied zu `sys_read()`) nicht die Datei-Operation `read()`, sondern die Inode-Operation `readpage()` verwendet.

Implementierung

Der Systemruf wird über das Syscall-Makro umgesetzt.

18 Eine Datei kann auch ein Gerät sein!

Fehler

EBADF — wenn ein Dateideskriptor ungültig ist.

EINVAL — wenn für die Datei keine `readpage`-Operation existiert.

EIO — wenn ein Lesefehler auftrat.

Systemruf	**swapon** **swapoff**	LINUX

Datei: mm/swapfile.c

```
long sys_swapon(const char * specialfile, int swap_flags);
long sys_swapoff(const char *file);
```

`sys_swapon()` schaltet den Auslagerungsspeicher (*Swap*) ein. Als Speicherplatz wird die Datei oder das Blockgerät `specialfile` benutzt. In den `swap_flags` kann das Flag `SWAP_FLAG_PREFER` gesetzt werden. Dann werden die letzen drei Bytes als eine Priorität interpretiert, die die Reihenfolge bei der Nutzung mehrerer Swaps festlegt. Ist keine Priorität angegeben, wird eine statische Variable ausgelesen, die bei jedem Zugriff automatisch dekrementiert wird.

Der Aufruf darf nur von einem Prozess mit dem Recht `CAP_SYS_ADMIN` und für jedes `specialfile` nur einmal durchgeführt werden. Insgesamt können maximal `MAX_SWAPFILES` gleichzeitig genutzt werden.

Bei Erfolg wird 0 zurückgegeben und eine Nachricht auf die Konsole geschrieben. Interessanterweise kann man beim Aufruf dieser Funktion die Meldung `Unable to start swapping: out of memory :-)` erhalten. Die Ursache liegt darin, dass der Kern vor der Initialisierung noch eine Seite alloziert.

`sys_swapoff()` schaltet den Auslagerungsspeicher wieder aus. Auch dieser Aufruf darf nur von einem Prozess mit dem Recht `CAP_SYS_ADMIN` ausgeführt werden.

Implementierung

Beide Systemrufe werden über das Syscall-Makro umgesetzt.

Fehler

EPERM — wenn ein nicht privilegierter Benutzer die Funktion aufruft.

EINVAL — wenn `file` existiert, aber keine Swap-Datei oder kein Blockgerät ist.

EBUSY — wenn `file` schon als Swap benutzt wird.

ENOMEM — wenn kein Speicher frei ist. Bei `sys_swapon()` werden im Kern zwei Speicherseiten für die Initialisierung benötigt.

A.5 Und der ganze Rest

Hier finden sich die Systemrufe, die nicht implementiert sind. Um jedoch mit Programmen und Konfigurationsskripten (wie dem aus der GNU-Welt bekannten `configure`) arbeiten zu können, wurde eine einheitliche Schnittstelle implementiert.

Systemruf	**acct**	**afs_syscall**	
	break	**ftime**	**idle**
	lock	**mpx**	**phys**
	prof	**profil**	**stty**
	gtty	**ulimit**	

Datei: kernel/sys.c

Diese Systemrufe sind nicht implementiert und werden intern auf `sys_ni_syscall()` umgelegt.

```
asmlinkage long sys_ni_syscall(void)
{
        return -ENOSYS;
}
```

Implementierung

Die Umsetzung geschieht über das Syscall-Makro.

Pearson Education

ADDISON-WESLEY

B Kernnahe Kommandos

Viele sind berufen,
aber nur wenige sind auserwählt.
Matthäus 20,16

Dieses Kapitel beschäftigt sich mit *kernnahen* Kommandos. Das sind Kommandos, die besondere Eigenschaften des Linux-Kerns nutzen bzw. direkt mit dem Kern operieren. Da dies eine sehr dehnbare Definition ist, haben wir eine Auswahl getroffen und beschreiben nur die Kommandos, die in den vorherigen Kapiteln erwähnt wurden oder thematisch zu diesen gehören.

Von vielen Programmen gibt es aber verschiedene Versionen. Der Grund dafür besteht in der großen Anzahl unterschiedlicher Distributionen und in der weiten Verbreitung des frei verfügbaren Linux. Wir haben im Folgenden solche Programme ausgesucht, die besondere Eigenschaften von Linux nutzen.

Bei einigen Programmen gibt es Versionen, die über das *Proc*-Dateisystem arbeiten. Der Vorteil dieser Programme besteht in einer erhöhten Sicherheit sowie in der Unabhängigkeit vom Kern. So ist es beispielsweise bei dem Kommando `ps` (eigentlich `procps`) kein Zugriff auf den Kernspeicher (`/dev/kmem`) erforderlich.

B.1 free — Übersicht über den Systemspeicher

Das Programm `free` zeigt die Belegung des vorhandenen Speichers an. Dabei wird zwischen RAM- und Swap-Speicher unterschieden. Angezeigt werden die Gesamtgröße, die belegte Größe und der freie Teil. Zusätzlich gibt `free` für den RAM-Speicher den als Shared Memory und den als Puffer genutzten Teil an.

Als Optionen existieren die Schalter:

`-b` Die Ausgabe erfolgt in Bytes.

`-k` Die Ausgabe erfolgt in Kilobytes.

`-m` Die Ausgabe erfolgt in Megabytes.

`-o` Es wird zusätzlich in einer Zeile angezeigt, wie groß der benutzte (freie) RAM-Speicher ohne (mit) Pufferspeicher ist.

`-t` Eine Zeile mit den Gesamtgrößen wird angezeigt.

`-s`*sec* `free` wiederholt seine Ausgabe alle `sec`. Die Angabe kann als *Float* mit Mikrosekunden-Auflösung erfolgen.

`-V` Die Version wird ausgegeben.

```
$ free
             total       used   free  shared  buffers  cached
Mem:         30956      30512    444   15792     9372    9136
-/+ buffers:            12004  18952
Swap:        34236       4176  30060
```

Dieses Programm arbeitet mit dem *Proc*-Dateisystem. Es liest die Datei `meminfo` aus und formatiert das Ergebnis:

```
$ cat /proc/meminfo
        total:    used:    free:  shared: buffers:  cached:
Mem:  31698944 31227904   471040 16183296  9584640  9351168
Swap: 35057664  4276224 30781440
```

B.2 ps — Ausgabe der Prozessstatistik

Der Befehl `ps` gibt einen Überblick über die im System laufenden Prozesse. Diese Übersicht ist nur eine Momentaufnahme; für eine dauernde Überwachung sollte `top` benutzt werden.

In Linux liest das `ps`-Kommando seine Daten aus dem *Proc*-Dateisystem. Dadurch läuft es unabhängig von der Kernelversion. Es braucht auch keine speziellen Rechte oder gesetzten s-Bits. Die Optionen können mit einem Minuszeichen (`-`) beginnen, müssen es aber nicht. Es gibt folgende Optionen:

`-O` Die Ausgabe wird sortiert. Dabei können mehrere Schlüssel angegeben werden:

- **`G`** Der Sortierschlüssel ist die TPGID.
- **`J`** Der Sortierschlüssel ist die im Nutzermodus verbrachte (mit Summe der Kinder) Zeit.
- **`K`** Der Sortierschlüssel ist die im Nutzermodus verbrachte Zeit.
- **`M`** Der Sortierschlüssel ist die Anzahl der Major Faults des Prozesses.
- **`N`** Der Sortierschlüssel ist die Anzahl der Major Faults des Prozesses und seiner Kinder.
- **`P`** Der Sortierschlüssel ist die PPID.
- **`R`** Der Sortierschlüssel ist die Größe des nicht ausgelagerten Speichers.
- **`S`** Der Sortierschlüssel ist die Größe des mehrfach benutzten Speichers.
- **`T`** Der Sortierschlüssel ist die Startzeit.
- **`U`** Der Sortierschlüssel ist die UID.
- **`c`** Der Sortierschlüssel ist die Kommandozeile.
- **`f`** Der Sortierschlüssel ist das Feld `F`.
- **`g`** Der Sortierschlüssel ist die PGRP.
- **`j`** Der Sortierschlüssel ist die im Systemmodus verbrachte (mit Summe der Kinder) Zeit.

k Der Sortierschlüssel ist die im Systemmodus verbrachte Zeit.

m Der Sortierschlüssel ist die Anzahl der Minor Faults des Prozesses.

n Der Sortierschlüssel ist die Anzahl der Minor Faults des Prozesses und seiner Kinder.

o Der Sortierschlüssel ist die SID.

p Der Sortierschlüssel ist die PID.

r Der Sortierschlüssel ist das Feld `RSS`.

s Der Sortierschlüssel ist die Größe des physisch belegten Speichers.

t Der Sortierschlüssel ist der TTY-Name.

u Der Sortierschlüssel ist der Nutzername.

v Der Sortierschlüssel ist die Summe der belegten Virtuellen Speicherbereiche.

y Der Sortierschlüssel ist die Priorität des Prozesses (`nice`-Wert).

-S Bei den Angaben *CPU-Zeit* und *Page Faults* werden die Werte der Kindprozesse addiert.

-X Es wird die Stack- und Registerbelegung ausgegeben.

-a Die Prozesse der anderen Nutzer werden auch angezeigt.

-c Es wird der Kommandoname ausgegeben, der in der Taskstruktur steht.

-e Das Enviroment der Prozesse wird mit ausgegeben.

-f Die Ausgabe wird baumartig formatiert, dabei bilden Kindprozesse die Äste (siehe `pstree` am Ende des Abschnitts).

-h Es wird keine Titelzeile ausgegeben.

-j Es erscheinen zusätzlich: PPID, PGID, TPGID und UID.

-l Es wird eine ausführliche Ausgabe erzeugt.

-m Die Speicherdaten werden ausgegeben.

-n Für die Felder `USER` und `WCHAN` werden die numerischen Werte ausgegeben (UID und Adresse).

-o Die Ausgabe wird nicht sortiert.

-p Speicherbezogene Angaben erfolgen nicht in Byte, sondern in Seiten.

-r Es werden nur laufende Prozesse ausgegeben.

-s Zusätzlich werden die gesetzten Signalmasken ausgegeben.

-u Der Nutzername sowie die prozentuale CPU- und Speicherbenutzung werden ausgegeben.

-v Zusätzlich werden die virtuellen Speicherdaten ausgegeben.

-w Es erfolgt eine erweiterte Ausgabe der Kommandozeile des Prozesses. Fehlt diese Option, schneidet `ps` die Ausgabe ab, damit sie auf eine Zeile passt; sie kann mehrmals angegeben werden!

-x Nur Prozesse ohne *Controlling Terminal* werden ausgegeben.

-t*xx* Nur Prozesse, die mit dem Terminal *xx* verbunden sind, werden ausgegeben.

Ausgewertet werden alle laufenden Prozesse oder (falls angegeben) eine durch Kommata getrennte Liste von PIDs. Je nach verwendeter Option liefert `ps` ein unterschiedliches Ausgabeformat. Tabelle B.1 gibt eine Übersicht über die Ausgabe der einzelnen Optionen. Dabei stellen die Spalten die angegebene Option und die Zeilen die ausgegebenen Daten dar. Der einfache Aufruf von `ps` ist die erste (leere) Spalte. Von den Optionen `j`, `l`, `s`, `u`, `v`, `m` und `X` kann nur eine angegeben werden, sie wirken exklusiv. `a`, `x`, `S`, `r` und `n` arbeiten in Kombination mit den übrigen Optionen und ändern nicht das Format der Ausgabe. Nicht aufgeführt sind die Zeilen TTY, PID und COMMAND. Sie werden immer ausgegeben. Die einzelnen Felder haben folgende Bedeutung:

ALARM der Alarm-Timer des Prozesses

BLOCKED die Signalmaske der vom Prozess blockierten Signale

CATCHED die Signalmaske der vom Prozess behandelten Signale

COMMAND die Kommandozeile des Prozesses

%CPU das Verhältnis von Systemzeit (`stime`) und Nutzerzeit (`utime`)

DRS der RSS-Anteil des Datensegments

DSIZ die Größe des Datensegments

DT die Anzahl der Bibliotheksseiten, auf die zugegriffen wurde

EIP das Register EIP

ESP das Register ESP

FLAGS die Flags des Prozesses, mittlerweile linksbündig

IGNORED die Signalmaske der vom Prozess ignorierten Signale

LIB die Speichergröße für verwendete Shared Libraries

LIM das Speicherlimit des Prozesses; wenn kein Limit gesetzt ist, wird `xx` ausgegeben

MAJFLT die Anzahl der Seitenzugriffsfehler, die dazu führen, dass die entsprechenden Seiten von der Festplatte geladen werden

%MEM der vom Prozess belegte Speicher (RSS) im Verhältnis zum vorhandenen Speicher (nur RAM-Speicher)

MINFLT die Anzahl der Seitenzugriffsfehler, bei denen sich die angeforderte Seite schon im Speicher befindet

	u	j	s	v	m	l	X		u	j	s	v	m	l	X
ALARM							x	PPID			x			x	
BLOCKED			x					PRI						x	
CATCHED			x					RSS		x			x	x	
%CPU	x							SHRD					x		
DRS					x			SIGNAL			x				
DSIZ				x				SID			x				
DT					x			SIZE		x			x	x	
EIP							x	STACK							x
ESP							x	START		x					
FLAGS						x		STAT	x	x	x	x	x		x	x
IGNORED			x					SWAP					x		
LIB					x			TIME	x	x	x	x	x		x	x
LIM				x				TMOUT							x
MAJFLT					x			TPGID			x				
%MEM	x			x				TRS					x		
MINFLT					x			TSIZ				x			
NI						x		UID			x	x		x	
NR							x	USER		x					
PAGEIN				x				WCHAN						x	
PGID		x													

Tabelle B.1: Optionen des `ps`-Programms

NI der (umgerechnete) `priority`-Wert des Prozesses

NR siehe FLAGS

PAGEIN die Anzahl der Seitenzugriffsfehler, die dazu führen, dass die entsprechenden Seiten von der Festplatte geladen werden (wie `MAJFLT`)

PGID die Prozessgruppe des Prozesses

PID die PID des Prozesses

PPID die PPID des Prozesses

PRI der (umgerechnete) `counter`-Wert des Prozesses

RSS die Größe des Programms im Speicher in KByte

SHRD die Größe des Shared Memory

SIGNAL das Signal, das der Prozess erhält (`task->signal`)

SID die SID des Prozesses

SIZE virtuelle Speichergröße; die Summe der belegten Virtuellen Speicherbereiche

STACK die Startadresse des Stacks

START die Startzeit des Prozesses

STAT der Status des Prozesses; dabei bedeutet:

- **D** Der Prozess schläft und kann nicht durch ein Signal geweckt werden.
- **R** Der Prozess ist aktiv.
- **S** Der Prozess schläft, kann aber durch ein Signal geweckt werden.
- **T** Der Prozess ist gestoppt oder läuft mit `ptrace`.
- **Z** Der Prozess befindet im Zombie-Status.

Es können Zusatzinformationen folgen:

- **W** Der Prozess hat keine Seiten im Speicher (RSS=0), ist aber kein Zombie.
- **<** `task->nice` ist kleiner 0.
- **N** `task->nice` ist größer 0.

SWAP der benutzte Swap-Speicher in KByte; mit der Option `-p` erhält man die Größe in Seiten

TIME die Laufzeit des Prozesses

TMOUT das gesetzte Timeout des Prozesses

TPGID die Prozessgruppe des Prozesses, der das Terminal besitzt

TRS der RSS-Anteil des Textsegments

TSIZ die Größe des Textsegments

TTY das Terminal, mit dem der Prozess verbunden ist

UID die EUID des Prozesses

USER der zur UID des Prozesses gehörende Name

WCHAN die Kernroutine, in der sich der Prozess gerade befindet

Normalerweise wird nur die Adresse ausgegeben. Um eine vernünftige Ausgabe für das Feld zu erhalten, ist es notwendig, das Programm `psupdate` aufzurufen. Dieses legt im Verzeichnis `etc` eine Datei `psdatabase` an. Sie enthält die Kernelfunktionen und ihre Adressen im Kern, dadurch kann `ps` den Namen der Funktion ausgeben und nicht nur die Adresse.

Das Programm `pstree` erzeugt eine grafische Ansicht des Prozessbaums. Es gibt, ausgehend von `init`, alle Prozesse in einer Baumstruktur aus, z. B.:

```
init-+-5*[agetty]
     |-amd
     |-crond
     |-gpm
     |-inetd-+-2*[in.rshd—tcsh—xterm—tcsh—vim]
     |       |-in.rshd—tcsh—xterm—tcsh
     |       |-nmbd
     |       `-smbd
     |-kflushd
     |-klogd
     |-kswapd
     |-lpd
     |-4*[nfsiod]
     |-rpc.mountd
     |-rpc.nfsd
     |-rpc.portmap
     |-4*[rsh]
     |-sendmail
     |-syslogd
     |-tcsh—startx—xinit-+-X
     |                   `-fvwm-+-GoodStuff
     |                          |-xbiff
     |                          |-xload
     |                          `-xterm—tcsh
     |-timed
     |-update
     `-ypbind
```

B.3 top — Die CPU-Charts

Das Programm `top` gibt, ähnlich wie `ps`, einen Überblick über den aktuellen Status des laufenden Systems. `top` läuft jedoch in einer Schleife und gibt alle fünf Sekunden eine neue Übersicht aus.

`top` kennt folgende Optionen:

d *xxx* `top` wartet `xxx` Sekunden zwischen seinen Ausgaben. Die Zeit kann Dezimalzahl mit Mikrosekundenauflösung angegeben werden.

q Die Wartezeit beträgt 0 Sekunden. Bei Root-Rechten wird die Priorität auf -10 gesetzt, um Kollisionen mit `kswapd` zu vermeiden.

c Statt des Kommandonamens wird die ganze Kommandozeile ausgegeben.

S Die angezeigte Zeit ist die Gesamtzeit des Prozesses und seiner Kinder. Bei dem Kommandozeilen-Parameter wird (leider) nicht `CTIME` angezeigt.

i Prozesse, deren Status `S` oder `Z` ist, werden nicht angezeigt.

s `top` läuft im *secure*-Modus. Damit sind folgende interaktive Kommandos nicht mehr verfügbar: `k` (*kill*), `r` (*renice*) und `s` (*sleeping time*).

Der Aufruf mit der Option `-q` veranlasst `top`, die Ausgabe ohne Wartezeit zu wiederholen. Wenn der Benutzer Superuser ist, läuft das Programm mit der höchstmöglichen Priorität.

Das Programm kann während der Arbeit durch die Eingabe von Kommandos gesteuert werden. Möglich sind folgende Eingaben:

- `^L` Der Bildschirminhalt wird neu gezeichnet.
- `M` Die Prozesse werden nach `%MEM` sortiert.
- `P` Die Prozesse werden nach `%CPU` sortiert.
- `S` Die Anzeige der Gesamtzeit (einschließlich Kindprozesse, *cumulative mode*) wird geschaltet.
- `T` Die Prozesse werden nach `[C]TIME` sortiert.
- `W` Die aktuellen Einstellungen werden in der Datei `~/.toprc` gespeichert.
- `c` Es wird zwischen der Anzeige des Kommandonamens und der gesamten Kommandozeile umgeschaltet.
- `f` **oder** `F` Die angezeigten Felder können verändert werden.
- `h` **oder** `?` Eine kurze Übersicht über die unterstützten Kommandos wird ausgegeben.
- `i` Das Anzeigen der Prozesse im Status *idle* wird ein- bzw. ausgeschaltet.
- `k` Der Benutzer kann ein Signal an einen Prozess versenden. Die PID des Prozesses und das Signal werden abgefragt.
- `l` Die Anzeige der *uptime*-Informationen (erste Zeile) wird ein- bzw. ausgeschaltet.
- `m` Die Anzeige der *free*-Informationen (vierte und fünfte Zeile) wird ein- bzw. ausgeschaltet.
- `n` **oder** `#` Die Anzahl der ausgegebenen Prozesse kann geändert werden.
- `o` **oder** `O` Die Reihenfolge der angezeigten Felder kann geändert werden.
- `q` Das Programm wird beendet.
- `r` Die Priorität eines Prozesses kann neu festgelegt werden.
- `s` Der Benutzer kann die Zeitspanne des Update-Intervalls festlegen.
- `t` Die Anzeige der Prozessstatistik (zweite und dritte Zeile) wird ein- bzw. ausgeschaltet.

Ausgegeben wird eine Mischung aus `uptime`, `free` und `ps`. Zusätzlich wird noch eine Übersicht über alle Prozesse und die CPU-Auslastung angezeigt. Die Ausgabe ist nach abnehmender Priorität sortiert.

```
 10:44am  up  1:01,  5 users,  load average: 0.02, 0.06, 0.09
55 processes: 52 sleeping, 1 running, 0 zombie, 2 stopped
```

```
CPU states:  4.1% user,  3.6% system,  7.7% nice, 92.3% idle
Mem:  30956K av, 30424K used,   532K free, 24636K shrd, 4164K buff
Swap: 34236K av,     0K used, 34236K free              10840K cach

  PID USER     PRI  NI  SIZE  RSS STAT %CPU %MEM   TIME COMMAND
  352 magnus    18   0   688  688 R     5.5  2.2   0:01 top
  107 root       5   0  5952 5952 S     1.8 19.2   1:39 X
  111 root       2   0  1592 1592 S     0.4  5.1   0:04 xterm
    1 root       0   0   520  520 S     0.0  1.6   0:01 init
    2 root       0   0     0    0 SW    0.0  0.0   0:00 kflushd
    3 root     -12 -12     0    0 SW<   0.0  0.0   0:00 kswapd
```

Vorher werden noch einige Angaben über den aktuellen Status des Systems gemacht. Die erste Zeile enthält die Zeiten und Systemauslastung analog zu `uptime`. Die zweite Zeile gibt die Anzahl der Prozesse an; dabei werden die einzelnen Zustände unterschieden. Die dritte Zeile gibt über die verbrauchte CPU-Zeit (in Prozent) für Prozesse im Nutzermodus, für Prozesse im Systemmodus für Prozesse und mit einem negativem `nice`-Wert sowie für den `idle`-Prozess Aufschluss.

Die restlichen Felder entsprechen den gleichnamigen Feldern der Kommandos `free` (siehe Anhang B.1) und `ps` (siehe Anhang B.2). Sie werden deshalb nicht extra erläutert.

B.4 Init — Primus inter pares

Der *Init*-Prozess mit der Prozessnummer 1 wird meist als Vater aller Prozesse bezeichnet. Dies ist in Linux zwar *nicht* der Fall, da diese Funktion vom *Idle*-Prozess wahrgenommen wird, der *Init*-Prozess wird aber aus Gewohnheit, auch in den Quellen, so bezeichnet.

Die hier beschriebenen Eigenschaften beziehen sich auf das Programm in der Version 2.75, das zu System V kompatibel ist. Seine Hauptaufgabe ist die kontrollierte Initialisierung des Systems und die Verwaltung der Start-Prozesse für die jeweiligen Systemzustände (*Runlevel*). Die Konfiguration geschieht mit Hilfe der Datei `/etc/inittab`. Dort können mehrere Konfigurationen durch die Angabe verschiedener *Runlevel* festgelegt werden. Ein Runlevel ist eine festgelegte Prozesskonfiguration des Systems.

Die Startprozesse werden in einer speziellen Struktur verwaltet, die im Folgenden Child genannt wird:

```
typedef struct _child_ {
   int flags;            /* Status des Eintrags          */
   int exstat;           /* Exit-Status                  */
   int pid;              /* PID des Prozesses            */
   time_t tm;            /* letzter Start-Zeitpunkt      */
   int count;            /* Starts in den letzten 2 Min. */
   char id[8];           /* Inittab-Id (eindeutig)       */
   char rlevel[12];      /* Runlevel                     */
   int action;           /* Aktion (siehe weiter unten)  */
   char process[128];    /* Kommandozeile                */
```

```
   struct _child_ *new;  /* Neuer Eintrag (neue inittab) */
   struct _child_ *next; /* Verweis auf nächsten Eintrag */
} CHILD;
```

Gesteuert wird `init` durch die Datei `/etc/inittab`. Darin stehen einzelne Zeilen, von denen jede eine Aktion für ein bestimmtes Ereignis auslöst. Eine Kommentarzeile beginnt mit einem #.

name:*level*:*action*:*kommando*

Die einzelnen Elemente haben folgende Bedeutung:

name Ein Bezeichner, der die Zeile eindeutig identifiziert.

Alte *Init*-Versionen (`a.out` und solche, die mit Bibliotheken vor der Version 5.2.18 übersetzt wurden) erlauben nur zwei Zeichen.

Achtung: Für *getty*-oder *login*-Prozesse muss der Name mit der Endung des zugehörigen `tty` übereinstimmen. Der Grund besteht darin, dass der Name als `utmp`-Eintrag benutzt wird — das *Accounting* würde sonst nicht funktionieren (deshalb auch die Größenbeschränkung: Der Eintrag darf nicht länger sein als die Größe von `utmp.ut_id`).

level Ein oder mehrere (maximal elf) Runlevel des Systems. Mögliche Werte sind die Zahlen 0 bis 9 und die Buchstaben S, A, B und C. `init` unterscheidet nicht zwischen Groß- und Kleinschreibung. Der Wert S steht für den Single-User-Modus. Fehlt die Angabe des Levels, wird die `action` des Prozesses bei jedem Wechsel des Runlevels aktiviert. Für die *action*-Felder `sysinit`, `boot` und `bootwait` wird der Level ignoriert.

action Diese Angabe sagt dem Prozess, wie *Init* das mit *kommando* angegebene Programm ausführen soll. Folgende Aktionen werden erkannt:

`boot` Das Kommando wird einmal beim Starten des Systems ausgeführt. Der Prozess wartet jedoch nicht auf die Beendigung, sondern wertet die Datei `inittab` weiter aus.

`bootwait` Das Kommando wird einmal beim Start des Systems ausgeführt, und Init wartet auf seine Beendigung; meist wird die Datei `/etc/rc` auf diese Art ausgeführt.

`ctrlaltdel` Das Kommando wird bei der Betätigung von Ctrl + Alt + Del[1] ausgeführt.

`initdefault` Der Level, der beim Starten des Systems verwendet wird. Wenn diese Zeile fehlt, fragt `init` den Level beim Starten des Systems an der Konsole ab. Der *kommando*-Eintrag wird ignoriert.

`kbrequest` Das Kommado wird beim Betätigen einer speziellen Tastaturkombination ausgeführt. Dazu muss in den Quellen `KDSIGACCEPT` konfiguriert werden.

`off` Es wird nichts ausgeführt.

1 Auch bekannt als Informatikerkralle oder *Three Finger Salute*.

once Das Kommando wird einmal zu Beginn des Levels ausgeführt.

ondemand Das Kommando wird immer dann ausgeführt, wenn `init` in den entsprechenden Runlevel übergeht. Übliche Level für `onedemand` sind `A+`, `B+` und `C+`.

powerfail Wie `powerwait`, allerdings wartet *Init* nicht auf die Beendigung.

powerfailnow Das Kommando wird ausgeführt, falls das Signal `SIGPWR` auftritt und der Eintrag `L` in `/etc/powerstatus` steht (z. B. bei fast leeren Akkus in Laptops).

powerokwait Das Kommando wird ausgeführt, sobald die Stromversorgung wiederhergestellt ist (das Signal `SIGPWR` und der Eintrag `'O'` in `/etc/powerstatus`). Üblich ist der Aufruf von `shutdown -c`.

powerwait Das Kommando wird ausgeführt, wenn *Init* das Signal `SIGPWR` erhält. Mit diesem Signal zeigen unterbrechungsfreie Stromversorgungen (UPS) einen Stromausfall an. Üblicherweise wird ein `shutdown` ausgeführt.

respawn Das Kommando wird nach Beendigung (des dadurch erzeugten Prozesses) neu ausgeführt. Ein klassischer Kandidat dafür ist `getty`.

sysinit Das Kommando wird beim Starten des Systems ausgeführt. Diese Einträge werden noch vor `boot` und `bootwait` ausgeführt.

wait Das Kommando wird einmal zu Beginn des Levels gestartet, und `init` wartet auf Beendigung des Prozesses.

kommando Hier steht ein Unix-Kommando. Eine Parameterangabe ist möglich. Wenn das Kommando mit einem Pluszeichen beginnt, wird es nicht in die Dateien `wtmp` und `utmp` eingetragen.

Die Verarbeitung der Datei erfolgt in der Routine `read_inittab()`. Sie nimmt folgende Aufgaben wahr: Lesen der Datei `/etc/inittab`, Aufbauen einer neuen Child-Liste sowie Beenden und Starten der notwendigen Prozesse. Das Lesen ist der einfachste Teil, die Datei wird Zeile für Zeile gelesen und dekodiert. Als Runlevel für `SYSVINT` wird dabei **`"#"`** und für `BOOT` und `BOOTWAIT` der Wert **`"*"`** verwendet. Als letzter Eintrag wird ein Shell-Aufruf eingetragen. Alle Children werden in der Liste `newFamily` verwaltet, die alten Children in `family`. Gibt es Einträge mit der gleichen ID in beiden Listen, wird das neue Child als `new` in der alten eingetragen. Der Lesen und der Aufbau der Liste geschieht parallel. Anschließend wird eine zweimalige Schleife durchlaufen, und alle unnötigen Prozesse aus `family` werden beendet. Dazu senden wir ihnen ein `SIGINT` beim ersten und ein `SIGTERM` beim zweiten Mal. Bleibt die Frage: Was sind die unnötigen Prozesse? Das sind alle Children, die keinen `new`-Eintrag haben oder deren `action`-Feld sich geändert hat, OneDemand-Einträge im Single-User-Modus und Children, die in ihren Flags `RUNNING` gesetzt haben. Bei den übrig gebliebenen Children kopieren wir Flags, die PID und den Exit-Status in den `new`-Eintrag und retten damit die alten Werte. Nachdem wir die Schleife zweimal durchlaufen haben, aktualisieren wir die `wtmp`- und `utmp`-Einträge für die beendeten Prozesse. Damit ist im Prinzip schon alles getan. Jetzt geben wir noch die alte Liste `family` frei, setzen den Zeiger auf `newFamily`

und löschen[2] die Datei `initrunlvl`. Diese liegt je nach Konfiguration unter `/etc/` oder `/var/log`.

Wenn das Programm `init` aufgerufen wird, überprüft es zuert, ob es als `init` oder als `telinit` aufgerufen wurde. Auf den zweiten Fall gehen wir später ein.

```
if (getpid() == INITPID || !strcmp(p, "init.new") ||
    !strcmp(p, "sh"))
{
```

Die Kommandozeile wird auf vorhandene Parameter untersucht und `init boot` wird als Kommandoname eingetragen. Wenn der `init`-Prozess gestartet wurde, öffnet er eine Pipe mit dem festen Deskriptor `STATE_PIPE`. Wenn diese Pipe schon existiert, handelt es sich um einen Neustart, und wir können die alten Werte aus der Pipe lesen und dann zu `init_main()` übergehen.

```
if (check_pipe(STATE_PIPE)) {
   […]
   reload = 1;
   init_main();
}
```

Existiert die Pipe nicht, müssen wir die Argumente der Kommandozeile auswerten:

```
     […]
maxproclen = strlen(argv[0]) + 1;
  for(f = 1; f < argc; f++) {
   if (!strcmp(argv[f], "single") ||
       !strcmp(argv[f], "-s"))
      dfl_level = 'S';
   else if (!strcmp(argv[f], "-a") ||
            !strcmp(argv[f], "auto"))
      putenv("AUTOBOOT=YES");
   else if (!strcmp(argv[f], "-b") ||
            !strcmp(argv[f], "emergency"))
      emerg_shell = 1;
   else if (strchr("0123456789sS", argv[f][0]) &&
            strlen(argv[f]) == 1)
      dfl_level = argv[f][0];
   /* "init u" in the very beginning makes no sence */
   if (dfl_level == 's') dfl_level = 'S';
   maxproclen += strlen(argv[f]) + 1;
}
maxproclen--;

argv0 = argv[0];
argv[1] = NULL;
setproctitle("init boot");
```

2 Wenn die Datei ein Link ist, wird dieser nicht gelöscht, sondern die Datei, auf die der Link zeigt, wird auf die Größe 0 gesetzt.

```
    init_main(dflLevel);
  }
```

Die Funktion `init_main()` übernimmt die Hauptarbeit. Dabei wird zwischen dem ersten Start von `init` beim Booten des Systems und einem Neustart unterschieden. Der folgende Code wird nur ausgeführt, wenn `reload` nicht gesetzt ist.

Da ein Debuggen des `init`-Prozesses aufgrund der (durchaus vernünftigen) Eigenschaften des Systemrufs *ptrace* nicht möglich ist, kann bei der Übersetzung das Flag `INITDEBUG` gesetzt werden. Dadurch wird Programmcode aktiviert, der an dieser Stelle ein `fork()` ausführt und damit ein Überwachen ermöglicht.

```
#if INITDEBUG
   if ((f = fork()) > 0) {
      while(wait(&st) != f) ;
      […]
   }
#endif
```

Danach wird der Kern darüber benachrichtigt, dass wir beim Betätigen der Tastaturkombination CTRL+ALT+DEL gern ein `SIGINT`-Signal und bei einer speziellen Tastaturkombination[3] ein `SIGWINCH`-Signal hätten. Außerdem ignorieren wir ersteinmal alle Signale.

```
   init_reboot(BMAGIC_SOFT);
   if ((f = open(VT_MASTER, O_RDWR | O_NOCTTY)) >= 0) {
      (void) ioctl(f, KDSIGACCEPT, SIGWINCH);
      close(f);
   } else
      (void) ioctl(0, KDSIGACCEPT, SIGWINCH);

   /*
    *   Ignore all signals.
    */
   for(f = 1; f <= NSIG; f++) signal(f, SIG_IGN);
```

Damit ist der Teil beendet, der nur beim Booten ausgeführt wird. Nun tragen wir für die Signale ALARM, HUP, INT, CHLD, PWR, WINCH, USR1, STOP, TSTP; CONT und SEGV eigene Behandlungsroutinen ein:

```
  SETSIG(sa, SIGALRM,  signal_handler);
  SETSIG(sa, SIGHUP,   signal_handler);
  SETSIG(sa, SIGINT,   signal_handler);
  SETSIG(sa, SIGCHLD,  chld_handler);
  SETSIG(sa, SIGPWR,   signal_handler);
  SETSIG(sa, SIGWINCH, signal_handler);
  SETSIG(sa, SIGUSR1,  signal_handler);
  SETSIG(sa, SIGSTOP,  stop_handler);
```

3 Welche Tastaturkombination das ist, hängt von der Konfiguration der Tastatur ab, weitere Informationen dazu finden Sie in den kbd-Packages.

```
SETSIG(sa, SIGTSTP,  stop_handler);
SETSIG(sa, SIGCONT,  cont_handler);
SETSIG(sa, SIGSEGV,  segv_handler);
```

Die Routine `signal_handler()` setzt das Signal in einer globalen Bitmaske `got_signals()`; die Routinen `cont_` und `stop_handler()` realisieren einen Semaphor, und `segv_` liefert noch eine Ausgabe über `syslog()` und landet dann in einer Endlosschleife. Der `chld`-Handler ist etwas komplizierter. Er ermittelt anhand der PID, welches Child gestorben ist. Dann trägt er das Signal in `exstat` ein und setzt `ZOMBIE` in den `flags`. Hat das Child einen `new`-Eintrag, werden die beiden Werte dort ebenfalls gesetzt.

Jetzt unterscheiden wir wieder zwischen Erst- und Neustart. Im ersten Fall werden die Standardein- und -ausgabe sowie die Standardfehlerausgabe geschlossen, und die Konsole wird mit `SetTerm()` initialisiert. Die Werte (Geschwindigkeit und Flags) werden dafür aus der Datei `/etc/ioctl.save` ausgelesen, wenn sie existiert. Das Programm trägt sich als Führer einer neuen Prozessgruppe ein und initialisiert eine `PATH`-Variable. Außerdem wird die *utmp*-Datei neu angelegt — und man sagt[4] „Hallo“.

```
close(0); close(1); close(2);
if ((s = getenv("CONSOLE")) != NULL)
   console_dev = s;

SetTerm(0); setsid();

if (getenv("PATH") == NULL) putenv(PATH_DFL);
(void) close(open(UTMP_FILE, O_WRONLY|O_CREAT|O_TRUNC, 0644));

Log(L_CO, bootmsg);
```

Für Systeme mit speziellen Sicherheitsanforderungen ist es durch die Angabe von `-b` möglich, an dieser Stelle über das Programm `sulogin` eine Shell zu starten, bevor *irgendetwas* anderes gemacht wird. `sulogin` fragt nach dem Root-Passwort und startet eine (über `SUSHELL` festlegbare) Shell — standardmäßig die in `/etc/passwd` eingetragene bzw. `/bin/sh`. Nach dem Beenden der Shell wird der normale Boot-Prozess fortgesetzt.

```
if (emerg_shell) {
   SETSIG(sa, SIGCHLD, SIG_DFL);
   if (spawn(&ch_emerg, &f) > 0) {
      while(wait(&st) != f)
         ;
   }
   SETSIG(sa, SIGCHLD,  chld_handler);
}
```

Der Boot-Prozess sieht so aus: `init` liest ersteinmal seine Konfigurationsdatei `inittab` ein.

4 Die Funktion `Log()` gibt den String über den Syslog-Dämon bzw. auf der Konsole aus.

```
    runlevel = '#';
    read_inittab();
```

Damit ist der Teil beendet, der nur beim Booten des Systems ausgeführt wird. Bei einem Neustart geben wir lediglich alle Signale frei. Dann werden alle erforderlichen Prozesse gestartet.

```
  } else {
    /* Neustart: Signale zulassen und dann weiter */
    log(L_CO, bootmsg);
    sigfillset(&sgt);
    sigprocmask(SIG_UNBLOCK, &sgt, NULL);
  }
  start_if_needed();
```

Nachdem alle erforderlichen Prozesse gestartet sind (im Englischen spricht man von „aufspannen“, *spawn*), tritt `init` in eine Schleife ein. Hier werden die Übergänge vom Start (`runlevel='#'`) über das Booten (`runlevel='*'`) zum normalen Betrieb realisiert. Laufen keine Prozesse mehr und sind keine Signale aufgetreten, wird die Kontroll-Fifo überprüft.

```
  while(1) {
     boot_transitions();
     for(ch = family; ch; ch = ch->next)
        if ((ch->flags & RUNNING ) && ch->action != BOOT ) break;
     if (ch != NULL  && got_signals == 0) check_init_fifo();
```

Über diese Pipe (`/dev/initctl`) kann *Init* zusätzlich beeinflusst werden. Programme (`telnetd`) können Requests in die Datei schreiben und so verschiedene Aktionen auslösen. Ein Request hat folgende Struktur:

```
struct init_request {
  int magic;                    /* magische Nummer             */
  int cmd;                      /* Art des Request             */
  int runlevel;                 /* neuer Runlevel              */
  int sleeptime;                /* Zeit zwischen TERM und KILL */
  char gen_id[8];               /* nicht benutzt               */
  char tty_id[16];              /* TTY-Name ohne /dev/tty      */
  char host[MAXHOSTNAMELEN];    /* Hostname                    */
  char term_type[16];           /* Terminal-Typ                */
  int signal;                   /* zu versendendes Signal      */
  int pid;                      /* Empfänger des Signals       */
  char exec_name[128];          /* auszuführendes Programm     */
  char reserved[128];           /* zukünftige Erweiterungen    */
};
```

Bis jetzt werden allerdings erst vier Request-Arten erkannt:

`INIT_CMD_RUNLVL` — Die neue `sleeptime` wird eingetragen, und dann wird in den gewünschten Level gewechselt.

INIT_CMD_POWERFAIL — Die neue `sleeptime` wird eingetragen, und die zur Aktion `POWERFAIL` gehörenden Kommandos werden ausgelöst.

INIT_CMD_POWERFAILNOW — Die neue `sleeptime` wird eingetragen, und die zur Aktion `powerfailnow` gehörenden Kommandos werden ausgelöst.

INIT_CMD_POWEROK — Die neue `sleeptime` wird eingetragen, und die zur Aktion `powerok` gehörenden Kommandos werden ausgelöst.

Die nächste Funktion hat folgenden Hintergrund: Stellt `init` fest, dass *respawn*-Prozesse sehr oft innerhalb kurzer Zeit (zehnmal innerhalb von zwei Minuten) aufzurufen sind, wird ein Fehler vermutet und der Eintrag der `inittab` für fünf Minuten deaktiviert. Dies verhindert eine unnötige Systembelastung aufgrund eines fehlerhaften `inittab`-Eintrags. `fail_check()` aktiviert die Einträge nach fünf Minuten wieder.

```
    fail_check();
```

Wenn `init` ein Signal erhält, werden je nach Art des Signals die in der `inittab` angegebenen Prozesse aktiviert. Die Verarbeitung der meisten Signale erfolgt nicht beim Eintreffen des Signals, sondern die Signalnummer wird in einer Bitmaske gesetzt und dann an dieser Stelle zentral ausgewertet.

SIGALRM Es passiert nichts.

SIGCHLD Ein Kindprozess ist beendet. Die Flags RUNNING, ZOMBIE und WAITING werden gelöscht und die Dateien `utmp` und `wtmp` aktualisiert.

SIGHUP `init` versucht, den neuen Runlevel aus `/etc/initrunlvl` zu lesen. Wenn die Datei nicht existiert, wird der alte Level weiterverwendet. Zusätzlich wird die Datei `inittab` neu gelesen und alle „Schläfer“ (`fail_cancel()`) werden geweckt.

SIGINT Das zur Aktion `ctrlaltdel` gehörende Programm wird aktiviert.

SIGPWR Die Datei `/etc/powerstatus` wird ausgelesen und gelöscht. Die Funktion `do_power_fail()` aktiviert die entsprechenden Kommandos.

SIGUSR1 Die Kontroll-Pipe `/dev/initctl` wird geschlossen und neu geöffnet.

SIGWINCH Das zur Aktion `kbrequest` gehörende Programm wird aktiviert.

Nachdem die eingetroffenen Signale bearbeitet sind, werden die notwendigen Prozesse gestartet, und die Schleife ist zu Ende:

```
    /* Signale verarbeiten */
    process_signals();

    /* Starten der Prozesse */
    start_if_needed();
  } /* while */
} /* init_main() */
```

Damit ist die Funktion `init_main()` beendet, und wir gehen zu dem Fall zurück, dass `init` bzw. `telinit` aufgerufen werden, um den Runlevel zu ändern. Das darf nur der

Superuser. Nach der obligatorischen Syntaxprüfung werden zwei Strategien verwendet: Zuerst wird ein Request zusammengebaut und in die Kontroll-Fifo geschrieben. Gekapselt wird dieser Versuch durch einen `alarm()`, um ein Blockieren zu vermeiden:

```
signal(SIGALRM, signal_handler);
alarm(3);
if ((fd = open(INIT_FIFO, O_WRONLY)) >= 0 &&
          write(fd, &request, sizeof(request))
              == sizeof(request)) {
      close(fd); alarm(0); return 0;
}
```

Schlägt dieser Versuch fehl, wird der „bekannte" Weg eingeschlagen: neuen Runlevel in der Datei `/etc/initrunlvl` abspeichern und sich selbst ein `SIGHUP` senden.

Das Programm `telinit` ist ein Link auf `init`. Es wird dazu benutzt, den Runlevel von der Kommandozeile aus zu ändern. Folgende Argumente können angegeben werden:

`0-6` Es wird in den angegebenen Runlevel gewechselt.

`a-c` Es werden nur die Prozesse ausgeführt, die in der `inittab` den angegebenen Runlevel haben.

`Qq` Die Datei `inittab` wird neu gelesen.

`Ss` Das System geht in den Single-User-Modus über.

`Uu` `init` wird erneut (mittels `execl()`) ausgeführt.

`-t` *`sec`* Mit Hilfe der Option `-t` *sec* kann die Wartezeit zwischen den Signalen SIGTERM und SIGKILL geändert werden. Der Standard ist 20 Sekunden.

B.5 shutdown — das Herunterfahren des Systems

Das Programm `shutdown` fährt das System sicher herunter. Alle Benutzer werden benachrichtigt, und `login` wird blockiert.

`shutdown [-t` *sec*`] [-rkhncfF] [-i` *level*`] [-g]` *time* `[`*message*`]`

Folgende Optionen können verwendet werden:

`-a` Die Zugriffskontrolle wird aktiviert.

`-c` Ein laufendes `shutdown` wird abgebrochen. Dabei darf natürlich keine Zeit angegeben werden, eine Nachricht ist allerdings möglich.

`-f` Beim Hochfahren des Systems wird kein `fsck` durchgeführt (*fast reboot*).

`-F` Beim Hochfahren des Systems wird auf jeden Fall ein `fsck` durchgeführt.

`-g` Angabe der Zeit nach dem alten Syntax.

`-h` Nach dem Herunterfahren hält das System an.

-i Das System wechselt in den angegebenen Runlevel.

-k Das Herunterfahren wird simuliert, es werden nur die entsprechenden Meldungen ausgegeben.

-n Es wird nicht über `init` gearbeitet, sondern `shutdown` fährt das System selbst herunter. Die Option ist nur in Kombination mit `-h` oder `-r` möglich.

-r Nach dem Herunterfahren soll neu gestartet werden.

-t*sec* Das Programm wartet *sec* Sekunden zwischen dem Senden der Signale SIGTERM und SIGKILL an alle Prozesse.

time Die Zeit, zu der `shutdown` das System herunterfährt.

message Diese Nachricht wird beim Herunterfahren an alle Benutzer ausgegeben.

Das Argument *time* hat zwei unterschiedliche Formate. Es kann in der Form *hh*`:`*mm* (`%d:%2d`) angegeben werden. Dann steht *hh* für Stunden und *mm* für Minuten. Oder das Format ist `+`*m*, wobei *m* die Anzahl in Minuten ist. Die Angabe `now` ist ein Alias für `+0`.

Beim Aufruf setzt `shutdown` zuerst seine UID auf seine effektive UID. Wenn die UID danach nicht 0 ist, bricht das Programm ab. Es arbeitet also nur mit Root-Rechten. Dann erfolgt die Auswertung der Kommandozeilenparameter.

Wenn die Datei `/etc/shutdown.allow` existiert und lesbar ist und die Nutzung der Zugriffskontrolle aktiviert wurde, wird die Datei ausgelesen. Darin stehen (zeilenweise) die Nutzer, die das System herunterfahren dürfen. Fehlt diese Datei, ist nur der Superuser berechtigt, das System herunterzufahren. Die Länge der Datei ist auf 32 Zeilen begrenzt. Es werden alle aktuellen Benutzer ermittelt und mit den erlaubten Benutzern verglichen. Dabei werden Remote-Nutzer ausgeschlossen. Erkannt werden die Benutzer an ihrem Loginnamen. Root ist implizit berechtigt (aber nicht remote).

Außerdem wird versucht, aus der Datei `/etc/shutdownpid` die Prozessnummer eines schon laufenden `shutdown` festzustellen. Läuft schon ein `shutdown` und ist die Option `-c` gesetzt, wird das laufende `shutdown` beendet, indem ihm ein SIGINT gesendet wird. Wenn die Option nicht gesetzt ist, und es existiert ein Prozess mit dieser PID, bricht das Programm ab.

Nun wird eine Datei angelegt, in der die eigene PID gespeichert ist, und es werden alle Signale außer SIGINT blockiert. Wenn `shutdown` ein SIGINT erhält, löscht es alle angelegten Dateien und ruft `exit(0)` auf. Das Programm wechselt in das Root-Verzeichnis. Wurde die Option `-f` angegeben, wird zusätzlich noch die Datei `/fastboot` angelegt, bei der Option `-F` wird die Datei `/forcefsck` angelegt.

Als letzter Schritt wird die Zeit ausgewertet. Steht dort `now` oder `+0`, wird sofort die Funktion `shutdown()` aufgerufen. Ansonsten wird gewartet, wobei ab einer Viertelstunde vor dem Herunterfahren alle fünf Minuten Warnungen ausgegeben werden. Außerdem wird durch das Anlegen einer Datei `/etc/nologin` das Einloggen blockiert.

Die Funktion `shutdown()` geht dann in den gewünschten Runlevel. Sie gibt noch eine letzte Warnung aus. Ist `-k` gesetzt (nur Simulation), werden alle angelegten Dateien gelöscht, und das Programm wird beendet. Bei `-n` (ohne Umweg über `init`) werden alle Prozesse in der üblichen Weise (`SIGTERM` plus `SIGKILL`) beendet, das Skript `/etc/rc.d/rc.halt` (wenn vorhanden) wird ausgeführt, und letztendlich wird das System angehalten (oder rebootet). Ansonsten baut `shutdown()` eine Kommandozeile zusammen und ruft damit das Programm `init` auf. Dies führt bei der Option `-h` zum Übergang in den Runlevel 0 und bei der Option `-r` zum Übergang in den Runlevel 6. Ist keine der beiden Optionen angegeben, geht das System in den Runlevel 1 über. Dabei gibt es möglicherweise eine böse Überraschung: Alte `inittab`-Dateien enthalten manchmal die Zeile „`x1:6:wait:/etc/rc.d/rc.6`“. Damit landet man nach `shutdown -r` vor einem X-Login, anstatt dass der Rechner bootet.[5]

B.6 strace — Observierung eines Prozesses

Die Fehlersuche in Programmen ist ein mühseliges Geschäft. Oft taucht dabei der Wunsch auf, eine Übersicht über alle ausgeführten Systemrufe mit ihren Parametern zu bekommen. Genau das bietet `strace`.

```
strace [ -dffhiqrtttTvVxx ] [ -a column ] [ -e expr ]
                [ -o filename ] [ -p pid ]
                [ -s strsize ] [ -u username ]
                command

strace -c [ -e expr ] [ -O overhead ]
                [ -S sortby ] command
```

`strace` kontrolliert die Abarbeitung des Kommandos *command*. Es registriert die Systemrufe und Signale. Die Ausgabe erfolgt auf die Standardfehlerausgabe oder, bei der Option `-o`, in eine Datei.

Jede Zeile der Ausgabe enthält einen Systemruf, seine Argumente (in Klammern) und den Rückgabewert. Bei einem Fehler (Rückgabewert `== -1`) wird die Fehlernummer (als symbolischer Name) und ihre Beschreibung mit angegeben (z. B. „`EINVAL (Invalid Argument)`“). Signale werden mit ihrem Namen ausgegeben.

Die Argumente werden, wenn möglich, in lesbarer Form ausgegeben. Zeiger auf Strukturen werden dereferenziert, und deren Komponenten und ihre Belegung werden in geschweiften Klammern angezeigt. Zeiger auf Zeichenketten werden ebenfalls derefenziert, und ihr Inhalt wird in Anführungsstrichen ausgegeben. Nichtdruckbare Zeichen sind als Escape-Sequenz, wie in C üblich, angegeben. Während für Strukturen geschweifte Klammern verwendet werden, zeigen eckige Klammern Felder an.

Um die Kontrolle genauer zu steuern, gibt es eine Vielzahl von Optionen. So ist es zum Beispiel möglich, sich auf bestimmte Systemrufe zu beschränken oder mittels `fork()` erzeugte Kindprozesse weiterzuverfolgen. Folgende Optionen sind möglich:

5 Anmerkung: Zur Nachahmung nur bedingt empfohlen.

-c `strace` erstellt eine Zeitstatistik für jeden Systemruf und gibt sie zum Schluss aus.

-d `strace` arbeitet im Debugging-Modus und gibt selbst Informationen aus.

-f erzeugt ein überwachter Prozess mittels `fork()` Kinder, werden diese ebenfalls überwacht.

-ff arbeitet mit `-o` *filename* zusammen. Jeder Kindprozess schreibt seine Ausgabe in die Datei *filename.pid*, wobei *pid* die PID des Kindes ist. Die Option `-f` wird zusätzlich eingeschaltet.

-h Eine Benutzungshilfe wird ausgegeben.

-i Zu Beginn jeder Zeile wird der Instruction-Pointer (EIP) ausgegeben.

-q Ausgaben über das Anhalten und Freigeben von Prozessen werden unterdrückt. Dies geschieht automatisch, wenn die Ausgabe nicht in eine Datei umgeleitet wird. Die Option ist nur in Verbindung mit `-f` oder `-p` *pid* wirksam, da nur dann Prozesse mittels `ptrace(PTRACE_ATTACH,pid,data)`gesteuert werden (siehe auch Seite 321).

-r Für jeden Systemruf wird der Zeitabstand zum vorherigen Ruf in Sekunden und Mikrosekunden ausgegeben.

-t Jede Zeile beginnt mit der aktuellen Uhrzeit im Format `HH:MM:SS`.

-tt Zusätzlich werden die Mikrosekunden ausgegeben.

-ttt Jede Zeile beginnt mit der aktuellen Uhrzeit in Sekunden und Mikrosekunden.

-T Die vom Systemruf verbrauchte Zeit wird angezeigt. Gemessen wird zwischen Aufruf und Rückkehr.

-v Alle komplexen Daten werden ausführlich ausgegeben. Dazu gehören zum Beispiel Argumentenvektoren und Strukturen. Ansonsten werden nur die ersten Komponenten oder Zeichen ausgeschrieben.

-V `strace` gibt seine Versionsnummer aus.

-x Nichtdruckbare Zeichen in Strings werden im Hexadezimalformat ausgegeben.

-xx Alle Zeichen in Strings werden im Hexadezimalformat ausgegeben.

-a *column* Der Rückkehrwert des Systemrufs wird in die Spalte *column* geschrieben, der Standard ist 40.

-e *expr* Hier kann ein Ausdruck *expr* angegeben werden, der den Trace-Vorgang genauer steuert. Der Ausdruck hat folgendes Format:

`[`*Typ*`=][!]`*Wert1*`[,`*Wert2*`]...`

Der Typ kann `trace`, `abbrev`, `verbose`, `raw`, `signal`, `fault`, `read` oder `write` sein. Der Wert ist abhängig davon entweder ein Name oder eine Zahl. Der Standardtyp ist `trace`. Das Ausrufezeichen negiert den Wert. Ein Beispiel: `-e open`, was `-e trace=open` entspricht, verfolgt nur die `open()`-Rufe — im Gegensatz zu

`-e trace=!open`, bei dem alle Rufe außer `open()` verfolgt werden. Als Spezialfälle für Werte gibt es zusätzlich `all` und `none`.

`abbrev=`*set* Beeinflusst die Ausgabe einzelner Komponenten großer Strukturen. Die Option `-v` schaltet `abbrev=none`. Standard ist `all`.

`faults` Falsche Speicherzugriffe werden mit ausgegeben. Diese Option ist nur bei SysV wirksam!

`raw=`*set* Schreibt die Argumente der Rufe *set* unkodiert (Hexadezimalformat) aus.

`read=`*set* Für alle Leseoperationen auf dem Dateideskriptor *set* wird ein Dump im ASCII- und Hexadezimalformat ausgeschrieben.

`signal=`*set* Auftretende Signale, die in *set* enthalten sind, werden ausgegeben. Standard ist `all`.

`trace=`*set* Nur die in *set* angegebenen Rufe werden verfolgt. Standard ist `all`. Als *set* können hier folgende Klassen angegeben werden:

`file` Es werden alle Systemrufe, die zum Dateisystem gehören, verfolgt. Im Einzelnen sind das :

```
access(), acct(), chdir(), chmod(), chown(),
chroot(), creat(), execve(), link(), lstat(),
mkdir(), mknod(), mount(), open(), readlink(),
rename(), rmdir(), stat(), statfs(), swapon(),
symlink(), truncate(), umount(), unlink(),
uselib(), utime()
```

`ipc` Es werden alle Systemrufe, die zur IPC (SysV) gehören, verfolgt. Im Einzelnen sind das:

```
msgctl(), msgget(), msgrcv(), msgsnd(), semctl(),
semget(), semop(), shmat(), shmctl(), shmdt(),
shmget()
```

`network` Es werden alle Systemrufe, die zur Netzwerkkommunikation gehören, verfolgt. Im Einzelnen sind das:

```
accept(), bind(), connect(), getpeername(),
getsockname(), getsockopt(), listen(), recv(),
recvfrom(), recvmsg(), send(), sendmsg() sendto(),
setsockopt(), shutdown(), socket(), socketpair()
```

`process` Es werden alle Systemrufe, die zur Prozessverwaltung gehören, verfolgt. Im Einzelnen sind das:

```
_exit(), fork(), waitpid(),
execve(), wait4(), clone()
```

`signal` Es werden alle Systemrufe verfolgt, die zur Signalverwaltung gehören. Im Einzelnen sind das:

```
pause(), kill(), signal(), sigaction(),
siggetmask(), sigsetmask(), sigsuspend(),
sigpending(), sigreturn() und sigprocmask()
```

`verbose=`*set* Steuert die Ausgabe der Argumente für die Systemrufe als Zeiger oder als dereferenzierte Strukturen. Für *set* werden die Strukturen ausführlich angegeben, für den Rest nur die Zeiger als Hexadezimaladressen. Standard ist `all`.

write=*set* Für alle Schreiboperationen auf dem Dateideskriptor *set* wird ein Dump im ASCII- und Hexadezimalformat ausgeschrieben.

-o *filename* Die Ausgabe wird in die Datei *filename*, bei gesetztem `-ff` in die Datei *filename.pid*, umgeleitet.

-p *pid* Der Prozess *pid* soll überwacht werden.

-O *Overhead* Durch das Verfolgen der Systemrufe entsteht ein Overhead, der die mit `-c` erstellte Statistik verfälscht. Der heuristisch vom Programm selbst ermittelte Wert kann hier korrigiert werden. Die Genauigkeit kann ein Nutzer selbst überprüfen. Dazu muss er nur die Systemzeit, die das zu überwachende Programm verbraucht, mit dem Programm `time` und mit `-c` ermitteln und beide Werte vergleichen. Der Wert für den Overhead ist in Mikrosekunden anzugeben.

-s *strsize* Bei Strings werden standardmäßig nur die ersten 32 Zeichen ausgegeben. Mit Hilfe dieser Option kann das geändert werden.

-S *sortby* Die mit `-c` erhaltene Tabelle wird nach der Spalte *sortby* sortiert. Mögliche Werte sind `time`, `calls`, `name` und `nothing`. Der Standard ist `time`.

Zum Abschluss ein Beispiel:

```
# strace sync
execve("/bin/sync", ["sync"], [/* 32 vars */]) = 0
sync()                                  = 0
_exit(0)                                = ?
```

B.7 Konfiguration des Netzwerk-Interfaces

`ifconfig` konfiguriert die Schnittstelle des Netzwerks. Normalerweise wird es beim Hochfahren des Systems ausgeführt und stellt die Parameter der Netzwerkgeräte ein.

`ifconfig [`*interface* `[[`*options* `…] `*address*`]]`

Wird `ifconfig` ohne Parameter aufgerufen, wird die aktuelle Konfiguration des Netzwerkinterface ausgegeben. Ansonsten wird die Schnittstelle mit den angegebenen Parametern *options* auf die IP-Adresse *address* konfiguriert. Folgende Parameter sind zulässig:

interface Der Name der Schnittstelle (z. B. `lo` oder `eth0`).

up Das Interface wird aktiviert. Wenn eine neue Adresse angegeben wird (siehe unten), wird diese Option implizit gesetzt.

down Das Interface wird abgeschaltet.

metric *N* Der Parameter setzt die Interface-Metrik auf den Wert *N*. Hier sollte eine 0 eingetragen werden.

mtu ***N*** Der Parameter legt die maximale Paketgröße[6] fest. Für Ethernetkarten ist ein Wert zwischen 1.000 und 2.000 günstig, für SLIP ein Wert zwischen 200 und 4.096.

[-]arp Die Verwendung des ARP-Protokolls wird an- oder ausgeschaltet. Steht ein Minuszeichen (-) davor, wird das Protokoll ausgeschaltet.

[-]trailer Diese Option wird von Linux ignoriert.

broadcast ***aa.bb.cc.dd*** Die Adresse wird als Broadcast-Adresse für das Interface eingetragen.

dstaddr ***aa.bb.cc.dd*** Die angegebene Adresse wird bei einer Punkt-zu-Punkt-Verbindung (PPP) als „anderes Ende" eingesetzt. Diese Option wird zur Zeit nicht unterstützt.

netmask ***aa.bb.cc.dd*** Der Wert wird als Netzmaske für das Interface eingetragen.

B.8 traceroute — der Ariadnefaden im Internet

Das Internet ist ein weiträumiges Konglomerat aus unterschiedlichsten Netzen. Dadurch ist das Auftreten von Verbindungsproblemen unausweichlich. Um nun Problemen wie „`network unreachable`" auf den Grund zu gehen, gibt es das Programm `traceroute`.

```
traceroute [-dnrv] [-m max˙ttl] [-p port#] [-q nqueries]
         [-s src˙addr] [-t tos] [-w wait] host [data size]
```

`traceroute` verfolgt den Weg von UDP-Paketen von der Quelle zum Ziel. Die Zieladresse kann als Rechnername oder als IP-Adresse angegeben werden. Das Programm verwendet zwei Techniken, einen Zeitstempel (einen kleinen *TTL-Wert*[7]) und eine ungültige Portadresse, um den Weg der Pakete zu verfolgen.

Die erste Technik findet den Weg zum Zielrechner. Jedes Gateway im Internet, das ein IP-Paket erhält, dekrementiert den TTL-Wert und schickt, wenn 0 erreicht ist, ein ICMP-Paket des Typs `ICMP_TIMXCEED` zurück. Beim Start sendet nun `traceroute` seine Pakete mit einem TTL-Wert 1 und inkrementiert ihn mit jeder erhaltenen `ICMP_TIMXCEED`-Nachricht. Zusätzlich gibt `traceroute` bei Erhalt des Pakets eine Statuszeile für das Gateway aus:

```
8  bnl-pppl.es.net (134.55.9.33)  516 ms  687 ms  771 ms
```

Die Zeile enthält den TTL-Wert, den Namen des Rechners, von dem die Nachricht stammt, seine IP-Adresse und die Übertragungszeiten (`traceroute` schickt jeweils drei Pakete).

Wenn das UDP-Paket das angegebene Ziel erreicht hat, sendet der Zielrechner ein ICMP-Paket mit dem Typ `ICMP_UNREACH` zurück. Der Grund liegt darin, dass `traceroute`

6 MTU — Maximal Transfer Unit.

7 TTL — Time To Life.

Pakete benutzt, die eine ungültige Portnummer (33.434) enthalten, um genau diesen Fehler zu erzeugen. Das Programm verwendet folgende Optionen:

-d In der `sock`-Struktur wird das Flag `debug` gesetzt. Dadurch werden mehr Informationen ausgegeben.

-m ***ttl*** `ttl` wird als der maximale Zeitstempel verwendet. Dadurch kann die Reichweite voreingestellt werden. Der Standardwert ist 30.

-n Das Programm gibt nur die IP-Adresse und nicht den Domainnamen des Zielrechners aus.

-p ***port*** `port` wird als Portnummer verwendet. Der Standardwert ist 33434.

-q ***nqueries*** Die Anzahl der pro Rechner zu sendenden Pakete. Normalerweise sind das drei Pakete.

-r `traceroute` versucht unter Umgehung der Routingtabellen das Paket direkt zuzustellen. Erreicht wird das durch das Setzen des `localroute`-Flags in der `sock`-Struktur. Ist das Ziel nicht direkt erreichbar, wird ein Fehler zurückgegeben.

-s ***addr*** Als Adresse der Quelle wird *addr* verwendet. Es ist eine Internet-Adresse anzugeben, kein Domainname. Wenn ein Rechner mehr als eine Adresse hat, kann dadurch die Sendeadresse geändert werden. Falls der angegebene Wert für die Maschine ungültig ist, wird ein Fehler zurückgegeben.

-t ***tos*** In das IP-Paket wird *tos* als *type of service* eingetragen. Möglich sind Werte zwischen 0 und 255. Weitere Informationen sind in den IP-Spezifikationen zu finden.

-v Alle empfangenen Pakete werden ausgeschrieben, nicht nur diejenigen, die mit `TIME_EXCEEDED` und `UNREACHABLE` zurückkommen.

-w ***n*** Die Wartezeit für die Antwort wird auf *n* Sekunden gesetzt. Der Standard liegt bei 5. Wenn innerhalb der festgelegten Wartezeit kein ICMP-Paket eintrifft, wird ein Sternchen (`*`) für diese Tests ausgegeben.

Ein Beispiel[8] für den Ablauf eines `traceroute`-Aufrufs:

```
# ./traceroute ice3.ori.u-tokyo.ac.jp
traceroute to ice3.ori.u-tokyo.ac.jp (157.82.132.65),
           30 hops max, 40 byte packets
 1 delta.informatik.hu-berlin.de (141.20.20.19)  6 ms  4 ms  4 ms
 2 141.20.20.9 (141.20.20.9)  4 ms  4 ms  4 ms
 3 192.2.6.2 (192.2.6.2)  98 ms  41 ms  33 ms
 4 Berlin1.WiN-IP.DFN.DE (188.1.132.250)  653 ms  549 ms  1037 ms
 5 ipgate2.WiN-IP.DFN.DE (188.1.133.62)  856 ms  669 ms  559 ms
 6 usgate.win-ip.dfn.de (193.174.74.65)  466 ms  392 ms  506 ms
 7 pppl-frg.es.net (192.188.33.9)  2288 ms  2964 ms  1057 ms
 8 umd2-pppl2.es.net (134.55.12.162)  361 ms  1375 ms  1441 ms
 9 umd1-e-umd2.es.net (134.55.13.33)  1669 ms  2334 ms *
```

8 Für Interessierte: 132.160.252.2 ist ein Rechner in Hawaii.

```
10 pppl-umd.es.net (134.55.6.34)  1421 ms  1346 ms  1477 ms
11 llnl-pppl.es.net (134.55.5.97)  1970 ms  1440 ms *
12 * ames-llnl.es.net (134.55.4.161)  1255 ms  2781 ms
13 ARC5.NSN.NASA.GOV (192.203.230.12)  2499 ms  2353 ms  2223 ms
14 132.160.252.2 (132.160.252.2)  2417 ms  1798 ms *
15 tko3gw.tisn.ad.jp (133.11.208.3)  1738 ms  1141 ms  1053 ms
16 uts4gw.tisn.ad.jp (133.11.210.2)  1575 ms  1505 ms  1498 ms
17 ncgw.nc.u-tokyo.ac.jp (133.11.127.127) 1740 ms 1677 ms 1170 ms
18 hongogw.nc.u-tokyo.ac.jp (130.69.254.3)  1222 ms * *
19 nakanogw.nc.u-tokyo.ac.jp (157.82.128.2) 1380 ms 594 ms 680 ms
20 origw1.nc.u-tokyo.ac.jp (157.82.128.65) 1770 ms 2006 ms 1650 ms
21 origw3.nc.u-tokyo.ac.jp (157.82.129.3) 2669 ms 854 ms 1254 ms
22 * ice3.ori.u-tokyo.ac.jp (157.82.132.65)  1262 ms *
```

`traceroute` schreibt ein Ausrufezeichen (`!`) hinter die Ausgabe der Zeitdauer, um auf Probleme hinzuweisen.

`!` Der Port ist nicht erreichbar (`ICMP_UNREACH_PORT`), und der zurückkommende `ttl`-Wert ist kleiner 2.

`!F` Eine Fragmentierung ist nötig (`ICMP_UNREACH_NEEDFRAG`).

`!P` Ein Protokollfehler trat auf (`ICMP_UNREACH_PROTOCOL`).

`!H` Der Rechner ist nicht erreichbar (`ICMP_UNREACH_HOST`).

`!N` Das Netzwerk ist nicht erreichbar (`ICMP_UNREACH_NET`).

`!S` Die Verwendung der Source-Adresse schlug fehl (`ICMP_UNREACH_SRCFAIL`).

B.9 Konfiguration einer seriellen Schnittstelle

Das Programm `setserial` (Version 2.12) setzt oder liest die Parameter und Flags einer seriellen Schnittstelle. Dabei können Portnummer und IRQ geändert werden. Wird ein schon belegter IRQ angegeben, gibt `setserial` die Fehlermeldung `Device busy` zurück.

```
setserial [ -abqvVW ] device [ opt [ arg ] ] …
setserial  -g [ -abv ] device1 …
```

Die folgenden Optionen können angegeben werden:

`-a` Alle Parameter des Geräts werden ausgegeben.

`-b` Die Standardeinstellungen (Port, IRQ und UART) werden ausgegeben.

`-g` Eine Liste von Geräten kann angegeben werden.

`-q` Bei `-W` werden keine Ausgaben erzeugt.

`-v` Nach dem Setzen des Geräts werden seine neuen Daten ausgegeben.

`-V` Das Programm gibt nur seine Version aus.

-W Versucht, alle freien Interrupts zu finden (und gibt sie mit `-va` aus).

Die folgenden Optionen können bei Bedarf angegeben werden. Nummern können in mehreren Formaten (Dezimal, Oktal- oder Hexadezimal) angegeben werden. Ein ^ vor einer Option negiert diese.

[^]auto_irq Während der Selbstkonfiguration wird versucht, den eigenen IRQ selbst festzustellen.

[^]callout_nohup Wurde das Gerät als *callout* geöffnet, wird kein `hangup()` beim Schließen durchgeführt (nur wenn `ASYNC_CALLOUT_NOHUP` konfiguriert wurde).

[^]fourport Der Port wird als AST- Fourport konfiguriert.

[^]pgrp_lockout Nur eine Prozessgruppe kann auf den `cua`-Port zugreifen, andere werden blockiert.

[^]sak Der *Secure Attention Key* wird verwendet.

[^]session_lockout Nur eine Sitzungsgruppe kann auf den `cua`-Port zugreifen, andere werden blockiert.

[^]skip_test Während der Selbstkonfiguration wird der UART-Typ getestet.

[^]split_termios *dialin* und *callout*-Geräte benutzen unterschiedliche termios-Einträge. Diese Option kann nur bei konfiguriertem `ASYNC_SPLIT_TERMIOS` verwendet werden.

autoconfigure Der Kernel versucht, das Gerät automatisch zu konfigurieren.

base ***base*** Siehe `baud_base`.

baud_base ***baud_base*** Diese Option setzt die Baud-Rate.

close_delay ***number*** Die Wartezeit des Prozesses, wenn er das Gerät schließt, wird gesetzt. Der Wert wird in Hundertstelsekunden angegeben.

closing_wait2 ***number*** Diese Option wird vom Kern nicht mehr unterstützt! Ursprünglich war die Wartezeit des Prozesses nach dem Schließen des Geräts gemeint.

closing_wait ***number*** Die Zeit beim Schließen des Geräts, in der der Prozess noch Daten annehmen darf (nur, wenn `ASYNC_CLOSING_WAIT_NONE` konfiguriert wurde).

divisor ***divisor*** Der Teiler für die Baudrate wird festgelegt. Der Teiler wird verwendet, wenn die Option `spd_cust` gesetzt ist und der Port auf 38,4 kBaud eingestellt ist.

get_multiport Die Konfiguration von Multiport-Geräten wird ausgegeben.

hup_notify Das Programm `getty` wird bei einem `hangup` und `close` des Ports benachrichtigt.

irq ***number*** Das Programm setzt den IRQ auf die angegebene Nummer. *number* ist dabei eine Zahl von 0 bis 15.

port ***number*** Das Programm setzt den Port auf die angegebene Nummer.

set_multiport Multiport-Geräte können konfiguriert werden. Dazu sind die Parameter mit `port[n]`, `mask[n]` und `match[n]` zu übergeben.

spd_cust Die Baudrate ergibt sich aus *baud_base/divisor*.

spd_hi Die Baudrate wird auf 57,6 kBaud gesetzt, wenn die Anwendung, die das Gerät benutzt, 38,4 kBaud verlangt.

spd_normal Es wird die Standardbaudrate von 38,4 kBaud verwendet.

spd_vhi Die Baudrate wird auf 115,2 kBaud gesetzt, wenn die Anwendung, die das Gerät benutzt, 38,4 kBaud verlangt.

termios_restore Die Terminaleinstellungen werden nach dem Freigeben der Blockierung restauriert.

uart ***type*** Der Parameter legt den verwendeten UART-Typ fest. Unterstützte Typen sind `8250`, `16450`, `16550`, `16550A` und `none`. Wenn FIFOs verwendet werden sollen, ist der Typ `16550A` anzugeben, da die anderen Typen dies nicht bzw. nur fehlerhaft unterstützen. Die Angabe `none` schaltet den Port ab.

In Linux ist es auf x86-Maschinen nicht möglich, dass sich mehrere serielle Schnittstellen einen IRQ teilen (nur spezielle Hardware, wie der AST-FourPort, unterstützt dies). Standardmäßig werden 38 Ports, `ttyS0` bis `ttyS38`, in der Datei `drivers/char/serial.c` initialisiert.

Wenn andere Konfigurationen gewünscht werden, kann das durch den Aufruf von `setserial` in `/etc/rc.local` realisiert werden. Die Angabe eines neuen IRQ ist nicht ganz einfach, da die meisten IRQs schon belegt sind. Als günstig hat sich die Verwendung von IRQ 5 erwiesen, der normalerweise für LPT2 zuständig ist. Andere Möglichkeiten sind 3, 4 und 7, oder, wenn die Karte 16 Bit unterstützt, die IRQs 10, 11, 12 und 15. Der IRQ 9 wird standardmäßig auf IRQ 2 abgebildet.

Um an die Daten zu kommen, wird das Gerät mit Hilfe eines `ioctl`-Rufs ausgelesen.

```
void getserial(char *device, int fd)
{
   struct serial_struct serinfo;

   if (ioctl(fd, TIOCGSERIAL, &serinfo) < 0) {
      perror("Cannot get serial info");
      exit(1);
   }
   printf("%s, Type: %s, Line: %d, Port: 0x%.4x, IRQ: %d\n",
      device, serial_type(serinfo.type),
      serinfo.line, serinfo.port, serinfo.irq);
}
```

Das Setzen der Portnummer oder des Interrupts geschieht ebenso mit einem `ioctl`-Ruf. Dieser Ruf testet außerdem, ob der angegebene Port oder Interrupt schon belegt ist, und gibt eine Fehlermeldung zurück. Bei der Angabe der Portnummer ist besondere Vorsicht geboten; die Angabe einer falschen Nummer kann den Computer stilllegen.

B.10 Konfiguration einer parallelen Schnittstelle

So wie für die Einstellung der seriellen Schnittstellen ein Programm existiert, gibt es auch eins für die parallelen Ports: `tunelp` (Version 1.5). Verwendet wird es hauptsächlich für die Druckerkonfiguration. Als Parameter gibt es folgende Optionen:

```
tunelp device [-i irq | -t time | -c chars | -w wait | -a [on|off]]
        | -o [on|off]] | -C [on|off]] | -r | -s | -q [on|off]]
```

`-a [on|off]` Hier kann angegeben werden, ob der Drucker beim Auftreten eines Fehlers abbrechen soll. Wenn man selbst am Drucker sitzt, ist das möglicherweise günstiger, da ein auftretender Fehler gleich behoben werden kann. Wenn das nicht der Fall ist, sollte sich der Druckerspooler darum kümmern, den Job selbst beenden und dem Benutzer eine Mail senden. Die (Qual der) Wahl bleibt dem Nutzer überlassen, der Standard ist `off`.

`-C [on|off]` Wenn diese Option gesetzt ist, definiert der Treiber den Drucker als *online* und ignoriert alle Fehlermeldungen. Das ist für Drucker nützlich, die auch im *offline*-Status Daten annehmen können.

`-c` ***chars*** Der Wert *chars* ist die Anzahl der Versuche, ein Zeichen auf dem Druckerport auszugeben. 120 ist ein guter Wert für die meisten Drucker. Als Standard ist `LP_INIT_CHAR` eingestellt, da einige Drucker etwas länger brauchen. Für sehr schnelle Drucker (HP-Laserjet) ist ein Wert von 10 sinnvoller.

`-i` ***irq*** Der verwendete Interrupt wird festgelegt. Wenn der Port keinen Interrupt benutzt (Polling), bricht diese Angabe den Druckvorgang ab. `tunelp -i 0` stellt das Polling wieder her, und der Drucker sollte wieder arbeiten.

`-o [on|off]` Vor dem Drucken wird das Gerät geöffnet und eine Statusabfrage durchgeführt.

`-q [on|off]` Diese Option kann mit allen anderen kombiniert werden. Ist sie gesetzt, wird zum Schluss ausgegeben, ob die Steuerung über IRQ (plus Nummer) oder Polling arbeitet.

`-r` Es wird ein *Reset* an das Gerät gesendet.

`-s` Der Status des Geräts wird ausgegeben. Diese Option schaltet `-q` auf *off*.

`-t` ***time*** Hier wird angegeben, wie viel Zeit der Gerätetreiber nach `-c`*char* Versuchen warten soll. Die Schnittstelle wird mit dem Wert `LP_INIT_TIME` initialisiert. Wenn so schnell wie möglich gedruckt werden soll und die Systembelastung keine Rolle spielt, kann dieser Wert auf 0 gesetzt werden. Wenn das Drucktempo keine Rolle spielt oder der Drucker langsam ist, ist 50 ein guter Wert. Außerdem wird das System kaum belastet. Die Angabe erfolgt in Hundertstelsekunden.

`-w` ***wait*** Das ist die Wartezeit für das *Strobe*-Signal. Es handelt sich dabei um ein *Busy Waiting*. Die meisten Drucker kommen mit einem extrem kurzen Signal zurecht, deswegen ist dieser Wert mit 0 (`LP_INIT_WAIT`) initialisiert. Eine Erhöhung macht,

außer der Verwendung entsprechender Drucker, auch ein längeres Druckerkabel möglich.

Linux verwaltet Drucker in einer Tabelle. Das BIOS unterstützt vier Drucker, in der Realität wird das selten zu finden sein. Deswegen werden nur drei Einträge initialisiert.

```
struct lp_struct lp_table[LP_NO];
```

B.11 Wir basteln uns einen Verzeichnisbaum

Die Verwaltung der Linux-Dateisysteme geschieht mit den Kommandos `mount` und `umount`. Sie dienen zum Mounten und Unmounten von Dateisystemen. Die Programme in der Version 2.5m haben dabei folgende Parameter:

```
mount [-hV]
mount -a [-nfrvw] [-t vfstypes]
mount [-nfrvw] [-o options] special | node
mount [-nfrvw] [-t vfstype] [-o options] special node

umount -a [ -t type ]
umount special | node
```

Ohne Parameter gibt `mount` die Liste der aktuell gemounteten Dateisysteme aus. Sonst wird mit Hilfe des Systemrufs *mount* versucht, das auf dem Gerät *special* befindliche Dateisystem auf den Mount-Point *node* zu mounten. Fehlt die Angabe von *special* oder *node*, so wird die fehlende Information aus der Datei `/etc/fstab` ermittelt. Die Optionen von `mount` haben folgende Bedeutung:

`-a` Es wird versucht, alle in der Datei `/etc/fstab` angegebenen Dateisysteme zu mounten. Wird zusätzlich die Option `-t` verwendet, gilt dies nur für die Dateisysteme des angegebenen Typs.

`-f` Mit dieser Option wird das Mounten des Dateisystems simuliert. Zusammen mit der Option `-v` kann überprüft werden, welche Aktionen das Programm `mount` durchführen würde.

`-h` Das Programm gibt eine Benutzungshilfe aus.

`-n` Die Datei `/etc/mtab` wird nicht geändert. Das ist notwendig, wenn sich das `etc/`-Verzeichnis auf einer CD befindet.

`-o` ***options*** Die Option `-o` gibt die durch Komma (`,`) getrennten Mount-Optionen *options* an. Diese sind oft sehr spezifisch; die folgenden Optionen kennen alle Dateisysteme:

`defaults` Dies ist die Standardoption. Sie entspricht den Optionen:

```
rw,exec,suid,dev,async,auto,nouser
```

`[no]auto` Das Gerät wird beim Aufruf von `mount -a` (nicht) mit erfasst.

`[no]dev` Die Nutzung von Gerätedateien auf diesem Dateisystem ist erlaubt (bzw. untersagt).

[no]exec Das Ausführen von Binaries des gemounteten Dateisystems ist erlaubt bzw. untersagt.

remount Ein Dateisystem wird neu gemountet. Im Allgemeinen nutzt man diese Option, um neue Flags einzutragen (z. B. um von `ro` nach `rw` zu wechseln).

ro Das Dateisystem wird nur lesbar gemountet.

rw Das Dateisystem wird les- und schreibbar gemountet.

[no]suid Beim Ausführen von Binaries des gemounteten Dateisystems werden eventuell gesetzte S-Bits beachtet (bzw. ignoriert).

[a]sync Alle Ein- und Ausgaben auf diesem Dateisystem werden (a-)synchron durchgeführt.

[no]users Normale Benutzer können dieses Gerät (nicht) mounten. Diese Option impliziert `noexec`, `nosuid` und `nodev`, wenn diese nicht explizit anders gesetzt sind.

Die weiteren Mount-Optionen der einzelnen Dateisystemtypen werden später beschrieben.

-r Das Dateisystem wird nur lesbar gemountet.

-w Das Dateisystem wird les- und schreibbar gemountet.

-t *type* Das zu mountende Dateisystem ist vom angegebenen Typ *type*. Ist kein Typ angeben oder `-a` gesetzt, versucht `mount()` über das Auslesen des Superblocks den Typ zu erkennen. Falls das angegebene „Gerät“ *special* einen Doppelpunkt (`:`) enthält, wird automatisch angenommen, dass es sich bei dem Dateisystemtyp um NFS handelt. Bei zusätzliche angegebener Option `-a` können mehrere Dateisystemtypen durch Komma (`,`) getrennt angegeben werden. In diesem Fall kann dem Typ auch die Zeichenkette „`no`“ vorangestellt werden. So mountet zum Beispiel das Kommando

```
mount -a -t nomsdos,nonfs
```

alle in der `fstab`-Datei angegebenen Dateisysteme, die weder vom Typ `msdos` noch vom Typ `nfs` sind.

-v Das Programm macht ausführliche Ausgaben, die den Nutzer über Aktionen informieren.

-V Die Version des Programms wird ausgegeben.

Das Programm `umount` entfernt gemountete Dateisysteme aus dem Dateisystembaum. Es kennt die Optionen `-a` und `-t` mit analoger Semantik wie oben.

Die Datei `/etc/fstab` hat einen einfachen Aufbau. Kommentarzeilen werden dabei durch ein Doppelkreuz (#) eingeleitet. Die anderen Zeilen enthalten vier durch *Whitespace* (Leerzeichen und Tabulatoren) getrennte Felder der Form:

special *node* *type* *option*

Hierbei sind alle Felder erforderlich, so dass bei einem nicht angegebenem Gerät oder Mount-Point `none` und bei Standardoptionen `defaults` anzugeben ist. Zusätzlich zu

Dateisystemen gibt die Datei auch Swap-Geräte und -Dateien an. In diesem Fall ist der Typ `swap`, und der Eintrag wird von `mount` und `umount` ignoriert.

Beide Programme nehmen Änderungen an der Datei `/etc/mtab` vor, so dass diese Datei normalerweise den aktuellen Zustand des Dateisystembaums beschreibt. Wenn diese Datei schreibgeschützt ist oder sich auf einem schreibgeschützten Dateisystem (CD) befindet, ist die Option `-n` anzugeben. Dann muss die Datei `/proc/mounts` benutzt werden!

Eine mit der Option `-o` oder in der Datei `/etc/fstab` angegebene Zeichenkette mit Mount-Optionen wird von `mount` auf die Mount-Flags (siehe Tabelle 6.1) hin untersucht, und die verbleibenden Mount-Optionen werden dem Systemruf *mount* als Parameter übergeben. Eine Ausnahme bildet hierbei das NFS, bei dem erst eine Verbindung zum Remote-Rechner aufgebaut wird und danach in einer Struktur `nfs_mount_data` die bereits interpretierten Mount-Optionen zusammen mit dem Socketdateideskriptor dem Systemruf übergeben werden.

Mount-Optionen des *Ext2*-Dateisystems

`bsddf` Wird diese Option angegeben, entfernt der Systemruf *statfs* die Anzahl der Blöcke, die für die Strukturen des *Ext2*-Dateisystems benötigt werden, aus der Gesamtanzahl der freien Blöcke. Dies ist das Standardverhalten des *Ext2*-Dateisystems.

`check=`*wert* Die Option stellt ein, wie viele Sicherheitstests das *Ext2*-Dateisystem während seiner normalen Arbeit durchführen soll. `none` stellt alle Tests ab. Beim Standard `normal` werden bei jedem Mounten bzw. Remounten die Inode- und die Blockbitmaps daraufhin überprüft, ob die Anzahl der dort eingetragenen freien Inodes bzw. Blöcke mit den Werten im Superblock übereinstimmt. In der Einstellung `strict` wird zusätzlich bei jeder Allokation sowie bei der Freigabe eines Datenblocks überprüft, ob die entsprechende Blocknummer einen Datenblock beschreibt.

`debug` Die Option schaltet den Debug-Modus des *Ext2*-Dateisystems ein. Bei jedem Mounten bzw. Remounten gibt das Dateisystem eine Meldung über seine Version sowie die Parameter des gerade gemounteten Dateisystems, wie Blockgröße und Anzahl der Blockgruppen, aus.

`errors=`*aktion* Diese Option definiert das Verhalten des Dateisystems, wenn ein Fehler entdeckt wird. In der standardmäßigen Einstellung `continue` wird lediglich der Fehler gemeldet. Die Einstellung `remount-ro` mountet nach einem Fehler das Dateisystem im Nur-Lese-Modus, so dass sich ein möglicher Fehler nicht vergrößern und das Programm `e2fsck` benutzt werden kann. Die Einstellung `panic` hält den Linux-Kern beim Auftreten eines Fehlers mit einer Panik-Meldung an.

`grpid, bsdgroups` Wenn diese Option angegeben wird, erhalten alle erzeugten Dateien die Gruppenidentifikation des Verzeichnisses, in dem sie angelegt werden. Im Standard `nogrpid` erhalten die neuen Dateien nur bei gesetztem Gruppen-S-Bit des Verzeichnisses die Gruppe des Verzeichnisses.

minixdf Im Gegensatz zur Option `bsddf` gibt der Systemruf *statfs* die Anzahl aller Blöcke auf dem zugrunde liegenden Gerät zurück.

nocheck entspricht der Option `check=none`.

nogrpid, sysvgroups Neu angelegte Dateien und Verzeichnisse erhalten bei gesetztem Gruppen-S-Bit des Verzeichnisses, in dem sie angelegt werden sollen, die Gruppenidentifikation des Verzeichnisses und nicht die des Prozesses. Zusätzlich erhalten in diesem Fall neu angelegte Verzeichnisse ebenfalls das Gruppen-S-Bit. Diese Option ist der Standard.

resgid=*gid* Die GID der Gruppe, deren Nutzer die reservierten Blöcke ebenfalls belegen darf. Ist keine der Optionen gesetzt, darf nur der Superuser auf die reservierten Blöcke zugreifen.

resuid=*uid* Die UID des Nutzers, der neben dem Superuser die reservierten Blöcke belegen darf.

sb=*blocknummer* Die Nummer des Blocks auf dem Gerät, der als Superblock eingelesen werden soll. Standardmäßig wird der Block 1 gelesen. Der Superblock wird alle 8912 Blöcke abgespeichert, die Nummerierung geht von 1-KByte-Blöcken aus.

quota, usrquota, noquota, grpquota Diese Optionen werden erkannt, aber ignoriert.

Mount-Optionen des MS-DOS-Dateisystems (FAT)

sys_immutable Dateien oder Inodes des Dateisystems dürfen nicht verändert werden.

[no]dots Dateien, die das Attribut *hidden* tragen, bekommen einen (bzw. keinen) Punkt (`.`) vor ihren Namen. Der Standard ist `nodots`.

dotsOK=[yes|no] Siehe `[no]dots`.

blocksize=[512|1024] Setzt die Sektorgröße des MS-DOS-Dateisystems. Diese beträgt normalerweise 512 Byte. Da MS-DOS 3.0 nur Partitionen bis maximal 65.536 Sektoren verwalten konnte, setzen einige Partitionsmanager die Sektorgröße auf 1024 Byte und erlauben so den Zugriff auf Partitionen bis zu 64 MByte. Um auf solche Partitionen zugreifen zu können, muss der Parameter `blocksize=1024` angegeben werden.

debug Schaltet den Debug-Modus ein.

fat=[12|16] Stellt ein, ob das zu mountende Dateisystem eine 12- oder eine 16-Bit-FAT besitzt. Da sich die Größe der FAT im Normalfall aus der Anzahl der Cluster auf dem Dateisystem ergibt, sollte es nicht notwendig sein, diesen Parameter zu benutzen.

quiet Da das MS-DOS-Dateisystem normalerweise keine Änderung des Besitzers der Inode und auch nicht alle UNIX-Zugriffsrechte unterstützt, liefern diese Operationen den Fehler `EPERM` zurück. Durch das Setzen der Option `quiet` wird diese Fehlermeldung unterdrückt, d.h. die Operationen lassen sich scheinbar durchführen.

check=*typ* Schaltet verschiedene Stufen der Dateinamenüberprüfung ein.

r[elaxed] Groß- und Kleinbuchstaben werden gleichwertig behandelt, lange Dateinamen werden gekürzt, und Leerzeichen sind erlaubt.

n[ormal] Wie `relaxed`, nur dürfen die Dateinamen keine Zeichen aus dem Feld `bad_chars[]` (d.h. `*?<>|"`) enthalten. Das ist die Standardeinstellung.

s[trict] Wie `strict`, nur dürfen die Dateinamen keine Zeichen aus dem Feld `bad_if_strict[]` (d.h. `+=,;`) enthalten.

conv=*typ* Schaltet die Umwandlung von MS-DOS-Textdateien ein und aus.

auto Konvertierungen werden bei „unbekannten" Dateitypen durchgeführt. Das sind diejenigen, deren Endung in in `ascii_extensions[]` zu finden ist. Da praktisch immer irgendeine Endung fehlt, können Programme, die auf solchen Dateien mit `lseek()` arbeiten, sehr ungnädig reagieren. Es ist also Vorsicht[9] angebracht!

binary Es findet keine Umwandlung statt. Das ist der Standard.

text Alle Dateien werden konvertiert. Beim Lesen werden alle *CR*-Zeichen entfernt, und beim Auftreten des Dateiendezeichens `^Z` (ASCII 0x1a) wird zum Ende der Datei gesprungen. Beim Schreiben der Datei werden alle *Newline*-Zeichen wieder in die MS-DOS-übliche Kombination umgewandelt.

gid=*gid* Die Gruppe des Eigentümers aller Dateien. Standardmäßig ist dies 0.

showexec Nur Dateien, deren Endung `.exe`, `.com` oder `.bat` ist, bekommen ein *executable*-Flag.

uid=*uid* Da MS-DOS keine Dateieigentümer kennt, zeigt bei dieser Option das MS-DOS-Dateisystem für alle Dateien den Eigentümer *uid* an. Standardmäßig wird 0 eingetragen.

umask=*umask* Benutzt die (oktal!) angegebene *Umask*, um die Zugriffsrechte aller Dateien zu berechnen. Wurde diese Option nicht angegeben, wird die Umask des Prozesses verwendet, der den Systemruf *mount* aufgerufen hat.

Mount-Optionen des ISO-Dateisystems

norock Schaltet die Verwendung der *Rock Ridge Extensions* aus.

check=*typ* Schaltet verschiedene Stufen der Dateinamenüberprüfung ein:

r[elaxed] Vor dem Zugriff werden Groß- in Kleinbuchstaben umgewandelt. Diese Option ist nur zusammen mit `norock` und `map=normal` sinnvoll.

s[trict] Es werden keine Umwandlungen vorgenommen.

9 In Gegensatz zu älteren Linux-Versionen, die im Feld `bin_extensions[]` die binären Endungen verzeichnet hatten, sind in der Version 2.4 diejenigen Endungen zu finden, die konvertiert werden müssen. Damit ist die Gefahr bedeutend geringer geworden, eine binäre Datei fälschlicherweise zu konvertieren.

cruft Diese Option maskiert die obersten acht Bit der Dateilänge aus, da einige Hersteller von CD-ROMs diese benutzen, um dort zusätzliche Informationen unterzubringen. Es können dann jedoch keine Dateien gelesen werden, die länger als 16 MByte sind. Bei einer (vermutlich) defekten CD wird diese Option automatisch gesetzt. Dazu zählen CDs mit mehr als 800 MByte und einr *Volume Number* ungleich 0 oder 1.

map=[normal|off] Schaltet die Namenskonventionen für CD-ROMs, die keine *Rock Ridge Extensions* besitzen, um. Standardmäßig ist `normal` eingestellt, so dass alle Dateinamen in Kleinbuchstaben und Semikolons in Punkte umgewandelt werden. Die Endung „`;1`“ wird entfernt. In der Stellung `off` werden die Namen nicht konvertiert.

conv=[auto|binary|text|mtext] Seit LINUX 1.3.54 wird diese Option ignoriert.

block=[512|1024|2048] Setzt die verwendete Blockgröße beim Zugriff auf die CD-ROM. Das ISO-Dateisystem setzt standardmäßig 1.024 an.

mode=*wert* Setzt die Dateizugriffsrechte für alle Dateien, falls das Medium keine *Rock Ridge Extension* enthält. Standardmäßig wird der Wert `S_IRUGO`, also das Leserecht für alle Nutzer, eingetragen.

uid=*uid* Die Nutzeridentifikationsnummer des Eigentümers aller Dateien. Standardmäßig wird 0 eingetragen.

gid=*gid* Die Gruppe des Eigentümers aller Dateien. Standardmäßig ist dies 0.

unhide Verborgene Dateien werden mit angezeigt.

Mount-Optionen des HPFS

uid=*uid* Die Nutzeridentifikationsnummer des Eigentümers aller Dateien. Standardmäßig wird 0 eingetragen.

gid=*gid* Die Gruppe des Eigentümers aller Dateien. Standardmäßig ist dies 0.

umask=*umask* Benutzt die angegebene *Umask*, um die Zugriffsrechte aller Dateien zu berechnen. Wurde diese Option nicht angegeben, wird die Umask des Prozesses verwendet, der Mount aufgerufen hat.

case=[lower|asis] Schaltet die Namenskonvertierung ein und aus. In der Standardeinstellung `lower` werden Großbuchstaben in Kleinbuchstaben konvertiert, in der Einstellung `asis` wird nichts geändert.

conv=*typ* Schaltet die Umwandlung von Dateien ein und aus. In der Standardeinstellung `binary` findet keine Umsetzung statt. Die Einstellung `text` setzt alle Dateien um. Dabei wird die *Carriage-Return-Linefeed*-Folge in das UNIX-übliche einzelne *Newline* überführt. In der Einstellung `auto` wird der erste zu lesende Block der Datei mit Hilfe einer Heuristik dahingehend überprüft, ob es sich um eine Text- oder eine binäre Datei handelt, und danach wird die Konvertierung für die restliche Datei durchgeführt.

nocheck Das Mounten wird nicht abgebrochen, falls Fehler bei der Konsistenzprüfung auftreten sollten.

Mount-Optionen des NFS

acdirmax=*n* Die maximale Zeitspanne in Sekunden, in der Attribute von Verzeichnissen zwischengespeichert werden, bevor aktuelle Informationen vom NFS-Server geholt werden. Der Standard ist 60.

acdirmin=*n* Die minimale Zeitspanne in Sekunden, in der Attribute von Verzeichnissen zwischengespeichert werden, bevor aktuelle Informationen vom NFS-Server geholt werden. Der Standard ist 30.

acregmax=*n* Die maximale Zeitspanne in Sekunden, in der Attribute von regulären Dateien zwischengespeichert werden, bevor aktuelle Informationen vom NFS-Server geholt werden. Der Standard ist 60.

acregmin=*n* Die minimale Zeitspanne in Sekunden, in der Attribute von regulären Dateien zwischengespeichert werden, bevor aktuelle Informationen vom NFS-Server geholt werden. Der Standard ist 3.

actimeo=*n* Die Optionen `acregmin`, `acregmax`, `acdirmin` und `acdirmax` werden auf den angegebenen Wert gesetzt.

addr=*n* Diese Option wird ignoriert.

mounthost=*n* Der Name des Rechners, auf dem der Mount-Dämon läuft.

mountport=*n* Die Portnummer des Mount-Dämons.

mountprog=*n* Ein anderes RPC-Programm soll verwendet werden. Standard ist 100.005, die normale RPC-Mount-Nummer.

mountvers=*n* Dem Mount-Dämon des Remote-Rechners wird diese Zahl als RPC-Version vorgesetzt.

namlen=*n* Die Abfrage der Dateinamenlänge des Remote-Dateisystems wird erst ab dem RPC-Mount-Protokoll Version 2 unterstützt. Mit dieser Angabe kann das Verhalten von älteren Servern festgelegt werden.

nfsprog=*n* Ein anderes RPC-Programm soll verwendet werden, um sich an den Remote-NFS-Dämon zu binden.

nfsvers=*n* Dem NFS-Dämon des Remote-Rechners wird diese Zahl als RPC-Version vorgesetzt.

port=*n* Die Nummer des UDP-Ports zur Verbindung mit dem NFS-Server. Die Voreinstellung ist der Standard-NFS-Port 2049.

retrans=*n* Die Anzahl der Minor-Timeouts und Neuübertragungen, bevor ein Major-Timeout ausgelöst wird. Der Standard ist 3. Danach wird die Operation abgebrochen oder die Meldung „server not responding“ ausgeschrieben.

retry=*n* Die Anzahl der Versuche eines HUHU. Der Standardwert ist 10 000.

rsize=*n* Die Anzahl der Bytes, die das NFS beim Lesen der Dateien von einem NFS-Server nutzt. Der Standard ist 1.024, besser wäre 8.192.

timeo=*n* Die Zeit in Zehntelsekunden, bevor Daten nach dem ersten RPC-Timeout nochmals übertragen werden. Der Standard sind sieben Zehntelsekunden. Bei nochmaligem RPC-Timeout wird die Zeit sukzessive verdoppelt bis zu einer maximalen Zeit von 60 Sekunden oder dem Erreichen eines Major-Timeouts.

[no]ac Die Option `noac` verbietet das Zwischenspeichern der Attribute von Dateien und Verzeichnissen. Dies führt zur erhöhten Belastung des Servers, ist aber wichtig, wenn mehrere NFS-Clients gleichzeitig aktiv auf einem Dateisystem des Servers schreiben. Der Standard ist `ac` mit den durch `acregmin`, `acregmax`, `acdirmin` und `acdirmax` gesetzten Werten.

[no]bg Wenn das Mounten zu einem Timeout führt, wird bei `bg` versucht, das Mounten im Hintergrund fortzuführen. Der Standard ist `fg` bzw. `nobg`.

[no]cto Die Option `nocto` verhindert das Nachfragen der Dateiattribute nach dem Erzeugen einer Datei. Der Standard ist `cto`.

[no]fg Wenn das Mounten zu einem Timeout führt, wird bei `fg` ein Fehler geliefert. Der Standard ist `fg` bzw. `nobg`.

[no]hard Wenn eine NFS-Operation einen Major-Timeout hat, wird bei `hard` die Meldung „server not responding“ auf die Konsole ausgegeben und versucht, die Operation fortzusetzen. Dies ist der Standard.

[no]intr Wenn eine NFS-Operation einen Major-Timeout hat und das Dateisystem `hard` gemountet wurde, ist es bei `intr` erlaubt, die Operation durch Signale zu unterbrechen. In diesem Fall wird der Fehler `EINTR` dem rufenden Programm zurückgegeben. Der Standard ist `nointr`, und somit ist das Unterbrechen von NFS-Operationen nicht erlaubt.

[no]posix Ein mit der Option `posix` gemountetes Dateisystem verhält sich POSIX-konform. Der Standard ist `noposix`.

[no]soft Wenn eine NFS-Operation einen Major-Timeout hat, wird bei `soft` ein Fehler an das aufrufende Programm geliefert. Der Standard `nosoft` oder `hard` setzt die NFS-Operation nötigenfalls unendlich fort.

[no]tcp Zur Kommunikation wird das Protokoll TCP verwendet. Standard ist UDP.

[no]udp Zur Kommunikation wird UDP verwendet. Viele NFS-Server unterstützen auch nur dieses Protokoll.

wsize=*n* Die Anzahl der Bytes, die das NFS beim Schreiben in Dateien eines NFS-Servers nutzt. Der Standard ist 1.024, besser wäre 8.192.

C Das *Proc*-Dateisystem

Der Computer redete und tickerte weiter, dreist und munter, als verkaufe er Waschpulver. »Nehmen Sie bitte zur Kenntnis, dass ich dazu da bin, Ihnen bei der Lösung Ihrer Probleme zu helfen, egal worum's dabei geht.«

Douglas Adams

Dieser Anhang beschreibt die Komponenten des *Proc*-Dateisystems, das in Linux unter `/proc` gemountet wird. Einige der System-Utilities, wie etwa `ps`, verlassen sich darauf. In ihren Quellen ist der Pfad fest vorgegeben. Die Aufgabe des *Proc*-Dateisystems ist es, Informationen über den Kern und die Prozesse auf einfache Weise bereitzustellen. Die teilweise kryptischen Aufrufe von *ioctl* sollen umgangen und die Informationen auf möglichst lesbare Weise angeboten werden.

Die einzelnen Dateien und Verzeichnisse werden meist erst beim Auslesen (Systemrufe *open* und *readdir*) generiert und enthalten somit die aktuellen Zustände des Linux-Systems.

Das Wurzelverzeichnis des *Proc*-Dateisystems hat die Inode-Nummer 1. Jeder existierende Prozess besitzt darin ein Unterverzeichnis. Um eine eindeutige Zuordnung gewährleisten zu können, ist der Name dieses Unterverzeichnisses die Prozess-ID. In diesem Unterverzeichnis befinden sich die Dateien mit den auf den Prozess bezogenen Informationen. Das Wurzelverzeichnis enthält zusätzlich die folgenden Unterverzeichnisse und Dateien. Die Inode-Nummern der einzelnen Dateien sind in der Datei `<linux/proc_fs.h>` als Aufzählungskonstanten definiert und entsprechen der angegebenen Reihenfolge.

C.1 Das Verzeichnis `/proc/`

`bus/` Dieses Verzeichnis enthält die Dateien des Bussystems. Im Allgemeinen ist es die Beschreibung des PCI-Busses.

`cmdline` Die Kommandozeile, die dem Kern beim Starten übergeben wurde.

```
root=/dev/hdc2 delay=700 BOOT_IMAGE=vmlinuz
```

`cpuinfo` Diese Datei enthält die beim Booten des Systems ermittelten Parameter des Prozessors (Datei `arch/i386/kernel/setup.c`).

```
processor       : 0                 Nummer des Prozessors
vendor_id       : GenuineIntel      Hersteller
cpu family      : 5                 CPU-Typ
```

```
model           : 2                 Modell
model name      : Pentium 75 -200   Modellname
stepping        : 5                 Stepping
cpu MHz         : 100.230096        interner CPU-Takt
fdiv_bug        : no                Rechnet er richtig?
hlt_bug         : no                Stoppt er korrekt?
f00f_bug        : yes               f00f-Bug?
coma_bug        : yes               Cyrix coma-Bug?
fpu             : yes               FPU-Einheit vorhanden?
fpu_exception   : yes               Funktioniert Exception 16?
cpuid level     : 2                 Maximaler cpuid-Level
wp              : yes               WP-Bit im Supervisor-Modus?
flags           : fpu vme de pse tsc msr mce cx8
                                    Prozessoreigenschaften
bogomips        : 39.94             Bogomips-Wert
```

devices Diese Datei enthält die Informationen über registrierte Gerätetreiber. Sie kann vom Skript `MAKEDEV` genutzt werden, um die Konsistenz des Verzeichnisses `/dev/` herzustellen. Zuerst steht die Major-Nummer des Gerätetreibers, und dann folgt der Name. Dabei handelt es sich um den Namen, der bei der Registrierung des Geräts angegeben wurde.

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
128 ptm
136 pts
162 raw

Block devices:
  2 fd
  3 ide0
  8 sd
22 ide1
```

dma Die belegten DMA-Kanäle können aus dieser Datei gelesen werden:

```
1: SB16 (8bit)
4: cascade
5: SB16 (16bit)
```

filesystems Dieser Datei können die vorhandenen (bzw. aktuell geladenen) Dateisystem-Implementierungen des Linux-Kerns entnommen werden. Der Name zu jedem mit der Funktion `register_filesystem()` angemeldeten Dateisystem wird ausgegeben. Benötigt das Dateisystem kein Gerät, ist dem Namen die Zeichenkette „`nodev`“ vorangestellt.

```
        ext2
nodev   proc
```

```
nodev   nfs
        iso9660
nodev   autofs
nodev   devpts
        vfat
```

ide Dieses Verzeichnis enthält die Beschreibungen der IDE-Controller im System und der daran angeschlossenen Geräte.

interrupts Die belegten Hardwareinterrupts können dieser Datei entnommen werden. Sie gibt für jede CPU die Anzahl der erhaltenen Interrupts, den behandelnden Interrupt-Controller sowie den Namen des Gerätetreibers, der den Interrupt behandelt, an. Zuletzt werden die Anzahl der aufgetretenen nicht maskierbaren Interrupts (NMI) sowie die Anzahl der aufgetretenen Wartezyklen bei Inter-Prozessor-Interrupts (IPI) angezeigt.

```
           CPU0       CPU1
  0:       6598       6763    IO-APIC-edge  timer
  1:          0          2    IO-APIC-edge  keyboard
  2:          0          0          XT-PIC  cascade
  8:          1          0    IO-APIC-edge  rtc
  9:        155        158   IO-APIC-level  eth0
 12:          7          2    IO-APIC-edge  PS/2 Mouse
 13:          1          0          XT-PIC  fpu
 14:        847        759    IO-APIC-edge  ide0
 15:          4          0    IO-APIC-edge  ide1
NMI:          0
ERR:          0
```

iomem Diese Datei enthält eine Auflistung, welcher Speicher von Hardware-Treibern bzw. vom Kern belegt ist. Angegeben sind die Start- und Endadresse des belegten Bereiches und der Name der Hardware.

```
00000000-0009ffff : System RAM
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000c8000-000cbfff : Extension ROM
000f0000-000fffff : System ROM
00100000-03ffffff : System RAM
  00100000-001e963f : Kernel code
  001e9640-001faacb : Kernel data
e0000000-e1ffffff : S3 Inc. Vision 968
e4000000-e40000ff : NCR 53c860
```

ioports Diese Datei enthält die mit `request_region()` belegten I/O-Ports.

```
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0080-009f : dma page reg
00a0-00bf : pic2
```

```
00c0-00df : dma2
…
```

kcore Die Datei `kcore` enthält den Speicherabzug (Core) des Kerns. Somit ist ein Debuggen des Kerns während der Laufzeit des Systems möglich. Dies kann mit Hilfe von

```
# gdb /usr/src/linux/vmlinux /proc/kcore
```

geschehen. Des Weiteren entspricht die Größe der Datei der Hauptspeichergröße plus der Page-Größe.

kmsg Diese Datei liefert beim Auslesen die noch nicht mit dem Systemruf *syslog* (siehe Seite 343) gelesenen Kernmeldungen. Ein Nebeneffekt des Auslesens ist, dass die gelesenen Meldungen aus dem Log-Ringpuffer entfernt werden. So kann eine Kernmeldung nur einmal gelesen werden. Deshalb sollte das Auslesen der Datei bei laufendem `syslogd`-Dämon unterbleiben. Die Datei ist außerdem nur für `root` lesbar.

ksyms Diese Datei enthält alle vom Kern exportierten Symbole mit Angabe der Adresse, des Namens und (evtl.) des Moduls, das sie enthält. Diese Kernsymbole können von Modulen verwendet werden. Die Datei existiert nur, wenn `CONFIG_MODULES` konfiguriert wurde.

```
c485299c packet_command_texts   [ide-cd]
c48528c4 sense_key_texts        [ide-cd]
c4852728 ide_cdrom_init [ide-cd]
…
```

loadavg Beim Auslesen dieser Datei erhält man die durchschnittliche Systemauslastung für die letzten 1, 5 und 15 Minuten. Diese Werte werden während jedes Timerinterrupts aktualisiert. Zusätzlich werden die Anzahl der laufenden Prozesse, deren Gesamtzahl und die PID des letzten aktiven Prozesses angegeben. Dazu wird die (eigene) Funktion `loadavg_read_proc()` benutzt.

```
0.14 0.12 0.05 2/57 242
```

locks Die vorhandenen Dateisperren liefert das Auslesen dieser Datei:

```
1: BROKEN ADVISORY WRITE 72 03:42:8072 0 2147483647
01f88d98 00000000 00000000 00000000 00000000
```

Jede Zeile enthält die Inode-Nummer der Datei, gefolgt von einem Doppelpunkt, die Flags (zwei Angaben) der Sperre, ihren Typ (READ oder WRITE), die PID ihres Besitzers, dann die Major- und Minornummer des Gerätes, auf dem die Datei liegt, sowie die Inode-Nummer (getrennt durch Doppelpunkte) und die Start- und Endposition der Sperre. Dann folgen die Adresse der eigenen Sperre, der vorhergehenden und nachfolgenden Liste und des vorhergehenden und nachfolgenden Blocks. Warten auf diese Sperre Prozesse, werden sie in den folgenden Zeilen ausgegeben, dabei beginnt die Zeile mit `<nr>: ->`, der Rest sieht aus wie in der vorhergehenden Zeile.

malloc Das Überwachen der Operationen `kmalloc()` und `kfree()` ist hier möglich. Wurde `CONFIG_DEBUG_MALLOC` nicht konfiguriert, fehlt diese Datei.

md Wurde *Multiple devices driver support* (CONFIG_BLK_DEV_MD) konfiguriert, so enthält diese Datei eine Nutzungsstatistik.

meminfo Diese Datei enthält die Anzahl sämtlicher benutzter und freier Bytes des Hauptspeichers und des Swapbereichs. Außerdem beinhaltet sie die Größe des von mehreren Prozessen geteilten Speichers sowie des Cachespeichers analog zum Kommando free. Im Unterschied zu free ist die jeweilige Größe in Byte und nicht in Kilobyte angegeben.

```
        total:    used:    free:  shared: buffers:  cached:
Mem:  64958464 56840192  8118272 31260672  5582848 35291136
Swap: 33021952    77824 32944128
MemTotal:     63436 kB
MemFree:       7928 kB
MemShared:    30528 kB
Buffers:       5452 kB
Cached:       34464 kB
SwapTotal:    32248 kB
SwapFree:     32172 kB
```

modules Diese Datei enthält Informationen über einzelne geladene Module, deren Größe und Zustand. Die Datei existiert nur, wenn CONFIG_MODULES konfiguriert wurde.

```
serial                  17300   1 (autoclean)
nfs                     32132   1 (autoclean)
lockd                   28104   1 (autoclean) [nfs]
sunrpc                  49884   1 (autoclean) [nfs lockd]
nls_iso8859-1            2012   6 (autoclean)
nls_cp437                3536   6 (autoclean)
vfat                    13484   6 (autoclean)
fat                     23988   6 (autoclean) [vfat]
```

mounts Die Datei (entspricht mtab in früheren Versionen) enthält die Liste der aktuell gemounteten Dateisysteme.

```
/dev/root / ext2 rw 0 0
/dev/hdc3 /usr ext2 rw 0 0
none /proc proc rw 0 0
/dev/hda1 /dos/c vfat rw 0 0
/dev/hda5 /dos/d vfat rw 0 0
/dev/hdb5 /dos/e vfat rw 0 0
/dev/sdb5 /dos/f vfat rw 0 0
/dev/sdb6 /dos/g vfat rw 0 0
murdock:(pid79) /nfs nfs rw,rsize=1024,wsize=1024,acregmin=1,
     acregmax=1,acdirmin=0,acdirmax=0,intr,
     addr=pid79@murdock:/nfs 0 0
/dev/sda4 /dos/z vfat rw 0 0
/dev/cdrom /mnt iso9660 ro 0 0
```

net/ Dieses Verzeichnis enthält einige Dateien, die den Zustand der Netzwerkschicht beschreiben. Genauere Informationen finden sich in Abschnitt C.2 und Kapitel 8.

partitions Diese Datei enthält die Angaben zu den Partitionen aller Blockgeräte. Das sind die Major- und Minor-Nummer des Geräts, seine Größe und sein Name.

```
major minor  #blocks  name
   8     0      98304 sda
   8     4      98288 sda4
   8    16    4233600 sdb
   8    17          1 sdb1
   8    21    2096451 sdb5
   8    22    2096451 sdb6
   3     0    1251936 hda
   3     1      52384 hda1
   3     2          1 hda2
   3     5    1197472 hda5
   3    64    1251936 hdb
   3    65          1 hdb1
   3    69    1247872 hdb5
  22     0     833616 hdc
  22     1      32255 hdc1
  22     2      32256 hdc2
  22     3     769104 hdc3
```

pci Hier befinden sich (die alten) Informationen über die Belegung der PCI-Slots. Wenn der Kern für PCI konfiguriert wurde, befinden sich die Informationen normalerweise unter `/proc/bus/pci`. Nur wenn zusätzlich *Backward-compatible /proc/pci* eingestellt wurde, existiert diese Datei.

```
PCI devices found:
  Bus  0, device   0, function  0:
    Host bridge: Acer Labs M1531 Aladdin IV (rev 178).
    Slow devsel.  Master Capable.  Latency=32.
  Bus  0, device   2, function  0:
    ISA bridge: Acer Labs M1533 Aladdin IV (rev 180).
    Medium devsel.  Master Capable.  No bursts.
  Bus  0, device   4, function  0:
    VGA compatible controller: S3 Inc. Vision 968 (rev 0).
    Medium devsel.  IRQ b.
    Non-prefetchable 32 bit memory at 0xe0000000 [0xe0000000].
  Bus  0, device   5, function  0:
    SCSI storage controller: NCR 53c875 (rev 3).
    Medium devsel.  IRQ 9.  Master Capable.  Latency=64.
    Min Gnt=17.Max Lat=64.
    I/O at 0x6400 [0x6401].
    Non-prefetchable 32 bit memory at 0xe4000000 [0xe4000000].
    Non-prefetchable 32 bit memory at 0xe4001000 [0xe4001000].
  Bus  0, device  11, function  0:
    IDE interface: Acer Labs M5229 TXpro (rev 32).
    Medium devsel.  Fast back-to-back capable.
    IRQ f.  Master Capable.  Latency=64.
    Min Gnt=2.Max Lat=4.
    I/O at 0xf000 [0xf001].
```

rtc Wurde *Enhanced Real Time Clock Support* (CONFIG_RTC) konfiguriert, enthält diese Datei die RTC-Werte.

scsi In diesem Verzeichnis befinden sich die Dateien (beginnend mit Inode 256), die die Informationen zu den einzelnen Geräten enthalten. Wenn der Kern nicht für SCSI konfiguriert wurde, ist dieses Verzeichnis leer. Ansonsten gibt es darin eine Datei scsi, die die Angaben des Controllers enthält, und für jedes (konfigurierte) Gerät gibt es ein Unterverzeichnis.

```
Host: scsi0 Channel: 00 Id: 05 Lun: 00
Vendor: IOMEGA   Model: ZIP 100          Rev: E.08
Type:   Direct-Access                    ANSI SCSI revision: 02
Host: scsi0 Channel: 00 Id: 06 Lun: 00
Vendor: IBM      Model: DCAS-34330W      Rev: S65A
Type:   Direct-Access                    ANSI SCSI revision: 02
```

self/ Dieses Verzeichnis enthält Informationen über den Prozess, der auf das *Proc*-Dateisystem zugreift. Es stimmt mit dem Verzeichnis überein, das die PID des Prozesses trägt. Genauere Informationen finden sich in Abschnitt C.3.

slabinfo Diese Datei enthält eine Übersicht über alle verwendeten Cache-Objekte. Es wird der Name des Caches ausgegeben, die Anzahl der aktuell benutzten Einträge und deren Gesamtzahl. Wenn in mm/slab.c SLAB_STATS gesetzt wurde, wird außerdem eine Zugriffsstatistik ausgegeben.

```
slabinfo - version: 1.0
kmem_cache            27     42
tcp_tw_bucket          0     42
tcp_bind_bucket       28    127
tcp_open_request       0     63
skbuff_head_cache      5     75
sock                  86     99
filp                 465    504
signal_queue           0      0
buffer_head         8348   8358
mm_struct             48     62
…
```

smp Die Datei enthält Informationen zu den einzelnen CPUs bei SMP-Systemen. Für sie muss SMP_PROF konfiguriert werden.

stat In dieser Datei ist eine allgemeine Linux-Kern-Statistik[1] enthalten.

```
cpu  7191 0 1542 341934  jiffies im Nutzer-, Nice-, Systemmodus und Idle-Prozess
cpu0 7191 0 1542 341934  für die jeweilige CPU
disk 102 6118 4 0        Anzahl der Plattenanforderungen
disk_rio 102 3586 4 0    Lesezugriffe pro Platte
disk_wio 0 2532 0 0      Schreibzugriffe pro Platte
disk_rblk 108 7146 8 0   gelesene Sektoren pro Platte
disk_wblk 0 5064 0 0     geschriebene Sektoren pro Platte
```

1 Die Angaben *pro Platte* beziehen sich bei IDE-Geräten auf die vier Controller, bei SCSI auf die ersten vier Geräte.

```
page 19588 5865          Speicherseiten ein- und ausgeblendet
swap 1 0                 Swapseiten ein- und ausgeblendet
intr 421905 350667 23464 0 0 41434 0 2 0 0 92 0 0 0 1 6225 20
…
                         Summe und Anzahl der Interrupts
ctxt 244058              Anzahl der Context-Switches
btime 841823772          Unix-Zeit des Bootens
processes 473            PID des aktuellen Prozesses
```

swaps Enthält die Daten zu den einzelnen Auslagerungsbereichen.

```
Filename        Type            Size    Used    Priority
/dev/hdc1       partition       32248   148     -1
```

sys/ Dieses Verzeichnis enthält Informationen, die die wichtigsten Algorithmen des Kerns steuern. Eine genauere Beschreibung findet sich in Abschnitt C.4.

sysipc/ Dieses Verzeichnis enthält Informationen, die die im System benutzen Messages, Semaphoren und Shared-Memory-Bereiche beschreiben.

tty/ Dieses Verzeichnis enthält Informationen, die die Terminals und ihre Treiber beschreiben.

uptime Diese Datei gibt die seit dem Systemstart verstrichene Zeit sowie die Zeit, die davon der Idle-Prozess verbrauchte, in Sekunden an.

```
501.05 344.11
```

version Die Datei `version` repräsentiert die Variable `linux_banner`. Dies ergibt zum Beispiel eine Ausgabe der Form:

```
Linux version 2.4.2 (root@murdock) (gcc version 2.96)
#1 Sat Feb 3 01:24:03 CET 2001
```

C.2 Das Verzeichnis `net/`

Dieses Verzeichnis enthält einige Dateien, die den Zustand der Linux-Netzwerkschicht beschreiben. Die Dateien haben die Inode-Nummern ab 128. Genauere Informationen finden sich in Kapitel 8. Die einzelnen Dateien sind:

arp gibt den Inhalt der ARP-Tabelle in lesbarer Form wieder.

```
IP address      HW type Flags HW address          Mask Device
141.20.22.210 0x1      0x2   08:00:5A:C7:10:24   *    eth0
141.20.22.203 0x1      0x2   00:00:C0:1B:E2:1B   *    eth0
141.20.22.204 0x1      0x2   00:00:C0:34:DE:24   *    eth0
```

dev Diese Datei enthält die vorhandenen Netzwerkgeräte sowie deren Statistik. Aufgrund der Ausgabelänge ist die Tabelle zweigeteilt.

```
Inter-|   Receive
 face |bytes    packets errs drop fifo frame compressed multicast
    lo:  252822    1710    0    0    0     0          0         0
  eth0:  196448    1734    0    0    0    47          0       778
```

```
Inter-| Transmit
 face |bytes    packets errs drop fifo colls carrier compressed
    lo: 252822  1710    0    0    0     0       0          0
  eth0:  27846   116    0    0    0     0       0          0
```

dev_mcast liefert eine Ausgabe der Multicast-Liste aller Netzwerkgeräte. Ausgeschrieben werden der Index und Name des Geräts, die UID und GID seines Benutzers sowie die Hardware-Adresse.

```
2    eth0            1     0     01005e000001
3    dummy0          1     0     01005e000001
```

dev_stat liefert Daten zur Netzwerknutzung. Die erste Zahl ist die Anzahl der verloren gegangenen Pakete. Für die weiteren Ausgaben sind bei den Netzwerkoptionen *Forwarding between high speed interfaces* und *Fast switching* zu konfigurieren.

```
00000000 00000000 00000000 00000000 00000000
```

netlink liefert Angaben über die Netlink-Sockets. Ausgegeben werden die Adresse, das Protokoll, die PID und die Gruppen, die Größe des verbrauchten Speichers, die Adresse seines Callbacks und die Anzahl der Verweise.

```
sk       Eth Pid    Groups   Rmem     Wmem     Dump     Locks
c1135740 0   0      00000000 0        0        00000000 2
```

netstat liefert die Netzstatistik für SNMP-Zwecke.

```
TcpExt: SyncookiesSent SyncookiesRecv SyncookiesFailed EmbryonicRsts
        PruneCalled RcvPruned OfoPruned
TcpExt: 0 0 0 0 26 96 616
```

packet liefert die Angaben zu den einzelnen Sockets der Packet-Treiber, und zwar die Adresse des Sockets, die Anzahl der Verweise, den Typ, die Protokollnummer, den Index des Netzwerkgeräts, das Running-Flag, den bisher allozierten Lesespeicher, sowie die UID und die Inode-Nummer der Socket-Inode.

```
sk       RefCnt Type Proto  Iface R Rmem   User   Inode
```

raw liefert Informationen über geöffnete Sockets des Typs RAW.

```
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt
         uid  timeout inode
   0: 00000000:0001 00000000:0000 07 00000000:00000000 00:00000000 00000000
            0        0 0
   1: 00000000:0006 00000000:0000 07 00000000:00000000 00:00000000 00000000
            0        0 0
```

route Diese Datei enthält die Routingtabelle in ungewohnter Form. Das Programm `route` bezieht seine Informationen aus dieser Datei.

```
Iface Destination Gateway  Flags RefCnt Use Metric Mask     MTU  Win IRTT
eth0  C01D148D    00000000 01    0      0        0 C0FFFFFF 1500 0   0
lo    0000007F    00000000 01    0      2        0 000000FF 3584 0   0
eth0  00000000    C11D148D 03    0      0        1 00000000 1500 0   0
```

rt_cache enthält die Angaben zum Routing-Cache.

```
Iface Destination Gateway  Flags    RefCnt Use  Metric Source   MTU
        Window IRTT TOS HHRef HHUptod SpecDst
```

```
lo    C21D148D    C21D148D 80000000 6      29   0      C21D148D 3924
        0300   00    2   1     C21D148D
lo    C21D148D    C21D148D 80000000 0      1633 0      E31D148D 0
        00     00    -1  0     C21D148D
lo    C21D148D    C21D148D 80000000 0      65   0      C31D148D 0
        00     00    -1  0     C21D148D
```

snmp Diese Datei enthält die MIBs (*Management Information Bases*) für das SNMP-Protokoll.

sockstat Diese Datei gibt die Anzahl der `proto`-Strukturen für die einzelnen Socket-Typen aus.

```
sockets: used 79
     TCP: inuse 25 highest 33
     UDP: inuse 15 highest 16
     RAW: inuse 2 highest 3
```

tcp liefert Informationen über geöffnete TCP-Sockets. Die Ausgabe erfolgt im gleichen Format wie bei `raw`.

udp liefert Informationen über UDP-Sockets, die Ausgabe erfolgt im gleichen Format wie bei `raw`.

unix liefert Informationen zu jedem geöffneten Unix-Domain-Socket, wie Pfad, Status, Typ, Flags, Protokoll und Referenzzähler.

```
  Num       RefCount Protocol Flags    Type St Inode Path
  00ecddfc: 00000002 00000000 00000000 0001 01  1015 /dev/log
  00ecda04: 00000002 00000000 00000000 0001 03  1014
```

C.3 Das Verzeichnis `self/`

Die Prozessverzeichnisse und das Verzeichnis `self/` haben folgenden Aufbau. Dabei gilt für die entsprechende Inode: *PID* << 16 + *angegebener Wert*.

status [+3] Die (Kurz-)Charakteristik des Prozesses lässt sich dieser Datei entnehmen.

```
Name:   xman                                    Name des Kommandos
State:  S (sleeping)                            Zustand des Prozesses
Pid:    120                                     seine PID
PPid:   108                                     seine PPID
Uid:    15216   15216   15216   15216           seine [ESF]UID
Gid:    15200   15200   15200   15200           seine [ESF]GID
FDSize: 256                                     max. Anzahl Dateideskriptoren
Groups: 15200                                   seine Gruppen
VmSize:     2296 kB                             Größe der VM-Bereiche
VmLck:         0 kB                             gesperrte VM-Bereiche
VmRSS:       308 kB                             RSS-Größe
VmData:      500 kB                             Datensegment (ohne Stack)
VmStk:        24 kB                             Stacksegment
VmExe:        40 kB                             geladenes Programm
VmLib:      1620 kB                             geladene Bibliotheken
```

```
SigPnd: 00000000                    anliegende Signale
SigBlk: 00000000                    Maske der blockierten Signale
SigIgn: 80000000                    Maske der ignorierten Signale
SigCgt: 00000000                    Maske der erhaltenen Signale
CapInh: 00000000fffffeff            geerbte Rechte
CapPrm: 0000000000000000            erlaubte Rechte
CapEff: 0000000000000000            effektive Rechte
```

mem [+4] Während `/dev/mem` den physischen Speicher vor einer Adressumsetzung repräsentiert, spiegelt diese Datei `mem` den linearen Adressraum des entsprechenden Prozesses wider.

Standardmäßig ist es nicht möglich, in diese Datei zu schreiben, da durch fehlende Tests auch im Speicher des Linux-Kerns geschrieben werden kann. In der Datei `fs/proc/mem.c` kann durch Entfernen der Makrodefinition von `mem_write` das Schreiben ermöglicht werden.

cwd [+5] Dies ist ein Verweis auf das PID-Verzeichnis des Prozesses im *Proc*-Dateisystem.

root [+6] Über den Verweis kann das Wurzelverzeichnis des Prozesses erreicht werden (siehe auch *chroot* auf Seite 355).

exe [+7] Dies ist ein Verweis auf die ausführbare Datei.

fd/ [+8] In diesem Verzeichnis befindet sich für jede vom Prozess geöffnete Datei ein Verweis mit dem Namen des Dateideskriptors. Der Verweis zeigt auf die geöffnete Datei.

Da sich hier auch die Standardeingabe (0) und die Standardausgabe (1) befinden, lassen sich Programme, die zum Beispiel nicht von der Standardeingabe lesen wollen, dazu überreden, diese über `/proc/self/fd/0` doch zu lesen.

Doch dabei können auch Probleme auftreten, da in den Dateien dieses Verzeichnisses nicht positioniert werden kann.

environ [+9] Diese Datei enthält die aktuelle Umgebung (Environment) des Prozesses. Die einzelnen Werte werden ohne Trennzeichen ausgegeben. Ist der gesamte Prozess ausgelagert oder ein Zombie, so ist die Datei leer.

cmdline [+10] Analog zur Datei `environ` enthält diese Datei die Kommandozeile des Prozesses.

stat [+11] Das Auslesen dieser Datei liefert genauere Informationen über den Prozess, z. B.:

```
185 (bash)    S  1 185 185 1028 330 256 699
615    542 2458 28  10  34   20  18   0   0
        0     16669    1990656        320 4294967295
134512640 134956915 3221225072 3221224492 1074512521
        0     65536    3686404  260128507 3222375209
        0         0         17          0
```

Die Datei gibt der Reihe nach[2] einzelne Werte der Taskstruktur des Prozesses wieder und stellt dem Anwender somit eine vollständige Zustandsbeschreibung des Prozesses zur Verfügung.

Da die Auswertung der einzelnen Einträge meist in Programmen erfolgt (siehe `ps` in Anhang B.2), sind sie hier im `scanf()`-Format angegeben. Handelt es sich bei dem Eintrag um den Wert einer Komponente der Taskstruktur, so ist der Name der Komponente in Klammern angegeben.

Die einzelnen Einträge der Datei sind:

- **`%d`** die Prozess-Id (`pid`)
- **`(%s)`** der Name der ausführbaren Datei in Klammern (auch sichtbar, wenn der Prozess ausgelagert ist)
- **`%c`** der Prozessstatus (`state`; „R“ für *Running*, „S“ für *Interruptable Sleeping*, „D“ für *Uninterruptable Sleeping or Swapping*, „Z“ für *Zombie*, „T“ für *Traced or Stopped* und „W“ für *Swapped*)
- **`%d`** die PID des Elternprozesses (`p_pptr->pid`)
- **`%d`** die Prozessgruppe (`pgrp`)
- **`%d`** die SID des Prozesses (`session`)
- **`%d`** das vom Prozess genutzte Terminal (`kdev_t_to_nr()`)
- **`%d`** die Prozessgruppe, die das vom Prozess genutzte Terminal besitzt
- **`%lu`** die Flags des Prozesses (`flags`)
- **`%lu`** die Anzahl der Minor Faults[3] (`min_flt`), die der Prozess hatte
- **`%lu`** die Anzahl der Minor Faults (`cmin_flt`), die der Prozess und seine Kinder hatten
- **`%lu`** die Anzahl der Major Faults[3] (`maj_flt`), die der Prozess hatte
- **`%lu`** die Anzahl der Major Faults (`cmaj_flt`), die der Prozess und seine Kinder hatten
- **`%ld`** die Anzahl der Jiffies (`times.tms_utime`), die der Prozess im Nutzermodus verbrachte
- **`%ld`** die Anzahl der Jiffies (`times.tms_stime`), die im Kernmodus verbracht wurden
- **`%ld`** die Anzahl der Jiffies (`times.tms_cutime`), die vom Prozess und seinen Kindern im Nutzermodus verbracht wurden
- **`%ld`** die Anzahl der Jiffies (`times.tms_cstime`), die vom Prozess und seinen Kindern im Kernmodus verbracht wurden
- **`%ld`** die maximale Anzahl der Jiffies (`counter` skaliert), die der Prozess in einer Zeitscheibe laufen kann

2 Zur besseren Lesbarkeit ist die Ausgabe leicht formatiert.

3 Ein *Minor Fault* ist ein Fehler beim Zugriff auf Speicherseiten, der ohne Zugriff auf ein externes Medium behandelt wird. Ein *Major Fault* dagegen muss durch einen Zugriff auf ein externes Medium behandelt werden.

%ld der UNIX-nice-Wert, (priority skaliert), der zur Berechnung eines neuen Wertes für counter genutzt wird

0 früher der Wert in Jiffies bis zum Auslösen eines Timeouts (timeout)

%lu der Wert des Interval-Timers (it_real_value)

%ld die Zeit des Prozessstarts (start_time), in Jiffies seit dem Systemstart

%u die Größe des Speichers in Bytes, auf die der Prozess zugreifen darf

%u die Anzahl (mm->rss) der im physischen Speicher befindlichen Seiten des Prozesses

%u die maximale Anzahl (rlim[RLIMIT_RSS].rlim_cur) der Seiten, die sich gleichzeitig für den Prozess im Speicher befinden dürfen

%lu die Adresse (mm->start_code) des Textsegmentanfangs

%lu die Adresse (mm->end_code) des Textsegmentendes

%lu die Adresse (mm->start_stack) des Stackbeginns

%lu der aktuelle Stackpointer[4] des Prozesses

%lu der aktuelle Befehlszeiger[4] des Prozesses

%lu der Signalvektor[5] (signal) der erhaltenen Signale

%lu der Signalvektor (blocked) der blockierten Signale

%lu der Signalvektor der ignorierten Signale

%lu der Signalvektor der mit Behandlungsroutinen versehenen Signale

%lu die Adresse der Kernfunktion, in der sich der Prozess befindet

%lu die Anzahl der Swap-Operationen (nswap)

%lu die Anzahl der Swap-Operationen der Kinder (cnswap)

%d das Exit-Signal (exit_signal)

%d die CPU, auf der der Task gerade läuft (processor)

statm [+12] Hier sind Speicherinformationen des Prozesses abgelegt. Die Ermittlung dieser Werte ist mit einem gewissen Zeitaufwand verbunden, so dass sie deshalb nicht in der Datei stat zu finden sind.

```
220 143 60 4 0 139 12
```

%d die Gesamtanzahl der genutzten Speicherseiten (*size*)

%d die Anzahl der Speicherseiten (*resident*), die sich gerade im physischen Speicher befinden

%d die Anzahl der Speicherseiten (*share*), die der Prozess mit anderen Prozessen teilt

4 Während der Ermittlung der Parameter befindet sich der Prozess gerade im Kernmodus, so dass die aktuellen Werte ESP und EIP zusätzlich umgerechnet werden und in das Nutzersegment zeigen. Als EIP erhält man dann meist eine Adresse in der C-Bibliothek.

5 Der Signalvektor ist eine 32-Bit-Zahl, bei der jedes Signal durch je ein Bit repräsentiert wird. Durch die Beschränkung auf 32 ist die Angabe hier veraltet, es sollten die Informationen aus status verwendet werden.

%d die Anzahl der Textseiten (*trs*), die sich im physischen Speicher befinden

%d die Anzahl der Bibliotheksseiten (*lrs*), die sich im physischen Speicher befinden

%d die Anzahl der Datenseiten (*drs*), einschließlich der beschriebenen Bibliotheksseiten und des Stacks, die sich gerade im physischen Speicher befinden

%d die Anzahl der Bibliotheksseiten (*dt*), auf die zugegriffen wurde

maps [+15] Hier kann man Informationen über die virtuellen Adressbereiche (siehe auch `vm_area`-Strukturen in Abschnitt 4.2.2) des Prozesses finden. Dabei werden für jeden virtuellen Adressbereich die Anfangs- und Endadresse, die Zugriffsrechte und der Offset in der eingeblendeten Datei, die Major- und die Minor-Nummer des Geräts, die Nummer der Inode sowie der Name der Datei angegeben. Die Zugriffsrechte werden in der UNIX-üblichen Schreibweise (`rwxsp`) angegeben, wobei zusätzliche Flags anzeigen, ob der Bereich geteilt (`s`) beziehungsweise privat (`p`) ist. Ist ein virtueller Speicherbereich anonym eingeblendet, ist die Nummer der Inode 0.

```
08048000-0807d000 r-xp 00000000 16:02 4077   /bin/tcsh
0807d000-08081000 rw-p 00034000 16:02 4077   /bin/tcsh
08081000-080b5000 rwxp 00000000 00:00 0
40007000-40008000 rw-p 00000000 00:00 0
…
```

C.4 Das Verzeichnis `sys/`

Die Unterverzeichnisse des `/proc/sys` Verzeichnisses ermöglichen die Abfrage von systemrelevanten Informationen. Diese Informationen werden in internen Tabellen gehalten und nach `/proc/sys` abgebildet. Die Inode-Nummern beginnen mit 4096. Einige Dateien sind schreibbar, dadurch können die entsprechenden Parameter im Kern zur Laufzeit geändert werden. In den folgenden Aufzählungen sind sie durch den Anhang `[w]` gekennzeichnet.

fs/ Dieses Verzeichnis enthält Daten und Einstellungen, die das Dateisystem betreffen.

dentry-state die Werte der `dentry_stat`-Struktur: Die Gesamtzahl der DEntrys, die Anzahl der freien DEntrys, das Maximalalter in Sekunden, die Anzahl der vom System angeforderten Seiten und zwei Fülleinträge.

```
0   2834   45   0   0   0
```

dquot-max[w] die maximale Quota-Anzahl (`NR_DQUOTS`)

```
0
```

dquot-nr die aktuelle Anzahl der Quotas und freien Quotas

```
0       0
```

file-max[w] die maximale Anzahl an Dateideskriptoren (`NR_FILES`)

```
4096
```

file-nr die Anzahl der benutzen Dateideskriptoren, der freien Deskriptoren und die Maximalzahl offener Dateideskriptoren.

```
216   3   4096
```

inode-nr[w] die aktuelle Anzahl der Inodes und freien Inodes

```
3072    2802
```

inode-state die Werte der inodes_stat-Struktur (inode-nr und fünf Füllwerte)

```
3072    2802    0    0    0    0    0
```

super-max[w] die maximale Anzahl der Superblöcke (NR_SUPER),

```
256
```

super-nr die aktuelle Anzahl der Superblöcke

```
5
```

kernel/ In diesem Verzeichnis befinden sich Informationen zum Kern und seinen Kontrollstrukturen.

random/ Dieses Unterverzeichnis enthält Dateien, die einen Zugriff auf den Zufallszahlengenerator ermöglichen.

boot_id eine 128 Bit (16*8 Byte) große Zufallszahl, die beim Starten des Systems berechnet wurde

```
e7b2d9c0-aa0d-4a5b-bc74-f6954cc8024b
```

entropy_avail das Maß der Zufälligkeit der Nummern

```
0
```

poolsize[w] die Größe des Entropie-Pools

```
512
```

read_wakeup_threshold[w] die Untergrenze der Entropie. Wird diese überschritten, werden alle Prozesse aufgeweckt, die random_read() (z. B. beim Lesen von einem Zufallsgerät) aufgerufen haben.

```
8
```

write_wakeup_threshold[w] die Obergrenze der Entropie. Wird diese unterschritten, werden alle Prozesse aufgeweckt, die mittels random_poll() warten.

```
128
```

uuid Eine 128 Bit (16*8 Byte) große Zufallszahl, die bei jedem Auslesen neu berechnet wird

```
2f532b1d-4df7-4296-89fe-c4009f64c885
```

acct die Steuerung der Prozessverwaltung. Wenn der freie Speicherplatz (für die Log-Dateien) unter den zweiten Wert sinkt, wird die Verwaltung deaktiviert, und wenn der frei Speicherplatz über den ersten Wert steigt, wird wieder sie aktiviert. Der dritte Wert ist die Überwachungsfrequenz in Sekunden. Diese Datei existiert nur, wenn bei der Übersetzung des Kerns das *BSD Process Accounting* aktiviert wurde.

```
4    2    50
```

ctrl-alt-del Eine 1 gibt an, dass CTRL+ALT+DEL die Machine neu startet, eine 0, dass das Signal SIGINT an den Prozess mit der PID 1 geschickt wird.

domainname der Domainname des Systems

hostname der Rechnername

modprobe der Pfad des Programmes, das Module lädt

osrelease die Version des Kerns

ostype der Name des Betriebssystems

panic der Timeout nach einer `panic`-Meldung

printk der aktuelle Loglevel, der Standardlevel, der minimale und der maximale Loglevel

version Compiler-Informationen bei der Übersetzung des Kerns

net/ Je nach Netzkonfiguration können hier die unterschiedlichsten Verzeichnisse und Dateien auftauchen. Für jedes Netzwerk-Subsystem (entspricht einem Gerät) gibt es ein Verzeichnis.

vm/ Dieses Verzeichnis enthält keine Daten über die aktuelle Speicherbelegung, sondern die Steuerungsparameter der Prozesse, die für die Speicherverwaltung verantwortlich sind.

bdflush[w] die Steuerungsparameter des `bd_flush()`-Prozesses

```
40 500 64 256 500 3000 500 1884 2
```

buffermem[w] die Steuerungsparameter für das Zurückschreiben verwendeter Speicherseiten (minimale, normale und maximale Größe)

```
2   10   60
```

freepages[w] die drei Stufen für die Freispeicherverwaltung. Die erste Zahl ist die absolute Untergrenze. Sinkt der Freispeicher unter die dritte (zweite) Zahl, startet der (intensive) Swap-Prozess.

```
128   256   384
```

kswapd[w] die Steuerungsparameter des Kswap-Dämons (Berechnungsbasis für die Anzahl der Swap-Versuche, minimale Anzahl der Versuche, Swap-Cluster-Größe)

```
512   32   32
```

overcommit_memory[w] steuert `vm_enough_memory()`. Ist dieser Wert ungleich 0, wird immer eine 1 zurückgegeben.

```
0
```

page-cluster[w] die Anzahl der gemeinsam ausgelagerten Seiten.

```
4
```

pagecache[w] die Steuerungsparameter des Page-Prozesses (wie `buffermem`)

```
2   15   75
```

pagetable_cache[w] die minimale und maximale Größe für den Page-Table-Cache

```
25   50
```

Pearson Education

ADDISON-WESLEY

D Der Boot-Prozess

Den Computer neu booten geht schneller als erst einen genialen Trick zu versuchen, um anschließend aus- und wieder einschalten zu müssen.

Murphys Computergesetze

Das ordentliche „Hochfahren" des Linux-Kerns wurde in Kapitel 2 und Kapitel 3 schon beschrieben. Nun gibt es aber mehrere Möglichkeiten, den Kern zum Starten zu veranlassen. Die einfachste ist, den kompletten Kern mit Hilfe von

```
# dd if=zImage of=/dev/fd0
```

ab dem Sektor 0 auf Diskette zu schreiben und später von der Diskette zu booten. Eine wesentlich elegantere Art ist das Booten von Linux durch den Linux-Lader (*linux loader* — LILO).

D.1 Der Ablauf des Bootens

Das Booten wird im PC vom BIOS übernommen. Nach dem Abschluss des *Power-On SelfTests* (POST) versucht das BIOS, den ersten Sektor der ersten Diskette, den Boot-Sektor, zu lesen. Schlägt dies fehl, versucht das BIOS, den Boot-Sektor von der ersten Festplatte zu lesen. Neuere BIOS-Versionen können diese Reihenfolge auch umdrehen und gleich von der Festplatte booten. Da die meisten BIOS keinen SCSI-Support besitzen, müssen deshalb SCSI-Adapter ein eigenes BIOS mitbringen, wenn von SCSI-Platten gebootet werden soll. Kann kein gültiger Boot-Sektor gefunden werden, startete der Ur-PC sein eingebautes ROM-BASIC, bzw. wurde der Nutzer mit der Meldung „NO ROM-BASIC" konfrontiert.

Das Booten eines Betriebssystems verläuft dann meist in mehreren Schritten. Da im Boot-Sektor sehr wenig Platz für Code ist, lädt dieser meist einen zweiten Lader nach usw., bis dann endgültig der eigentliche Betriebssystemkern geladen ist. Wie Abbildung D.1 zeigt, ist der Aufbau eines Boot-Sektors relativ einfach; seine Länge beträgt stets 512 Byte (so dass er sich sowohl auf einer Diskette als auch auf einer Festplatte befinden kann). Dabei spielen die Diskparameter nur für MS-DOS eine Rolle. Wichtig ist, dass der Code bei Offset 0 beginnt und der Boot-Sektor mit der *Magic Number* beendet wird.

Das Booten von Diskette ist jetzt relativ einfach, da jede Diskette genau einen Boot-Sektor hat. Dies ist der erste Sektor. Danach folgen beliebige Daten. Das Booten von der Festplatte ist etwas schwieriger, da diese in Partitionen unterteilt ist. Davon weiß das BIOS aber nichts, folglich lädt es ebenso wie von der Diskette den ersten Sektor, der *Master Boot Record* (MBR) genannt wird.

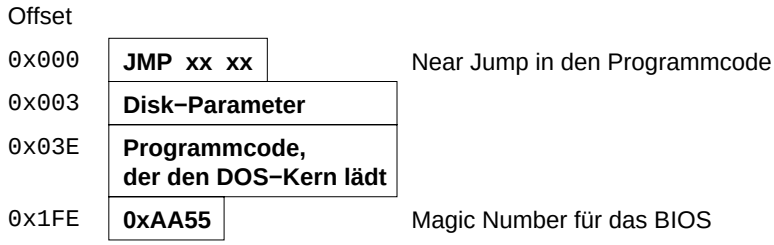


Abbildung D.1: Der MS-DOS-Boot-Sektor

Der MBR muss also auch denselben Aufbau besitzen. Das heißt, ab Offset 0 beginnt Code und auf Offset 0x1FE steht die Magic Number 0xAA55. Am Ende des MBR ist die Partitionstabelle untergebracht. Diese hat stets vier Einträge, wie in Abbildung D.2 zu sehen ist. Ein Eintrag in der Partitionstabelle besteht dabei aus 16 Byte. Sein Aufbau wird in Abbildung D.3 gezeigt.

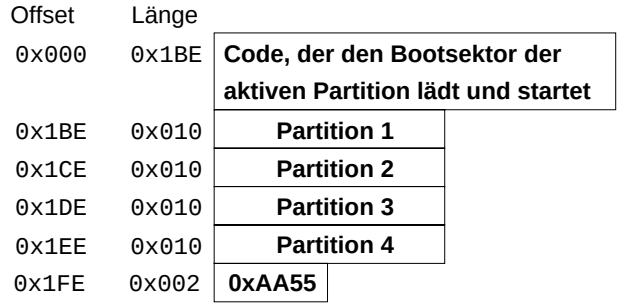


Abbildung D.2: Aufbau des Master-Boot-Records und der erweiterten Partitionstabelle

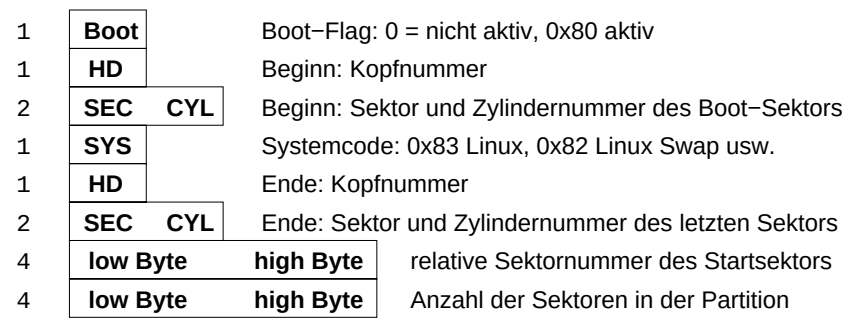


Abbildung D.3: Struktur eines Partitionseintrags

Eine Festplatte kann also in vier Partitionen unterteilt sein, die *primäre Partitionen* genannt werden. Sollte das nicht ausreichen, kann eine so genannte *erweiterte Partition* angelegt werden. Diese enthält wiederum mindestens ein *logisches Laufwerk*. Da man hier aber offensichtlich keine weitere Struktur einführen wollte, entspricht der Aufbau

des ersten Sektors einer erweiterten Partition einfach dem des MBR. Der erste Partitionseintrag dieser erweiterten Partitionstabelle enthält das erste logische Laufwerk der Partition. Der zweite Eintrag wird als Zeiger benutzt, falls weitere logische Laufwerke existieren. Er zeigt dann hinter das erste logische Laufwerk, wo sich wiederum eine Partitionstabelle mit dem Eintrag für das nächste logische Laufwerk befindet. Die einzelnen Einträge der logischen Laufwerke sind also untereinander in einer einfach verketteten Liste verbunden. Eine erweiterte Partition könnte also theoretisch beliebig viele logische Laufwerke enthalten.

Der erste Sektor einer jeden primären oder erweiterten Partition enthält einen Boot-Sektor mit dem bereits beschriebenen Aufbau. Da nur von einer dieser Partitionen gebootet werden kann, bestimmt das Bootflag die *aktive Partition*.

Ursprünglich gab es nur primäre Partitionen, darum können `fdisk` unter MS-DOS und auch die meisten ähnlichen Programme nur diese Partitionen aktivieren. Der Code im MBR braucht demnach nur folgende Operationen durchzuführen:

- die aktive Partition bestimmen
- den Boot-Sektor der aktiven Partition mit Hilfe des BIOS laden
- in den Boot-Sektor ab Offset 0 springen

Dazu genügen die im MBR vorhandenen Bytes völlig. Da sich, wie oben beschrieben, im Prinzip in jeder Partition ein Boot-Sektor befindet, und außerdem der Aufbau einer möglicherweise vorhandenen zweiten Festplatte dem der ersten gleicht, gibt es mittlerweile eine Vielzahl von Ersetzungen für den Standard-MS-DOS-MBR, so genannte *Bootmanager*. Allen gemeinsam ist, dass sie entweder den MBR mit eigenem Code ersetzen oder den Boot-Sektor einer aktiven Partition belegen. Zum Booten von Linux werden wohl die meisten den Linux-Lader LILO verwenden.

D.2 LILO — der Linux-Lader

Der LILO-Boot-Sektor enthält Platz für eine Partitionstabelle. Deshalb kann LILO sowohl in einer Partition als auch in den MBR installiert werden. LILO besitzt die volle Funktionalität des Standard-MS-DOS-Boot-Sektors. Zusätzlich kann er auch logische Laufwerke oder Partitionen auf der zweiten Festplatte booten. LILO kann auch in Kombination mit einem anderen Bootmanager benutzt werden, so dass viele Varianten der Installation denkbar sind.

D.2.1 MS-DOS-MBR startet LILO

Befindet sich wenigstens eine primäre Linux-Partition[1] auf der ersten Festplatte, so kann LILO dort installiert werden. Nach Aktivierung dieser Partition verläuft der Bootvorgang wie folgt:

1 Keine Swap-Partition, da in dieser auch der erste Sektor benutzt wird!

- Das BIOS lädt den MBR.
- Der MBR lädt den Boot-Sektor der aktiven Partition, den LILO-Boot-Sektor.
- Der Lader bootet Linux oder ein anderes Betriebssystem.

Auch eine Deinstallation verläuft denkbar einfach: Eine andere Partition wird aktiviert. Da außerhalb der Linux-Partition keine Daten (bis auf das Boot-Flag) verändert werden, ist dies die „sicherste" Variante.

D.2.2 LILO wird von einem Bootmanager gestartet

Ein solcher Ansatz empfiehlt sich, wenn man auf seinen alten Bootmanager nicht verzichten will oder LILO nicht in der Lage ist, ein fremdes Betriebssystem zu booten. Je nach Fähigkeit des anderen Bootmanagers bieten sich aber noch andere „Plätze" für die LILO-Installation.

- Kann der Bootmanager erweiterte Partitionen booten, bieten sich diese als idealer Platz für LILO an.
- Kann der Bootmanager Partitionen der zweiten Festplatte booten, so lässt sich LILO auch dort installieren.
- Manche Bootmanager können sogar logische Laufwerke booten, dann kann man LILO dort installieren.

Bei dieser Vorgehensweise sollte man jedoch Folgendes beachten:

- Die Installationsprogramme einiger Betriebssysteme[2] schreiben ihren eigenen MBR ohne Nachfrage auf die Platte. Dabei könnte der andere Bootmanager zerstört werden.
- Eine Repartitionierung könnte auch den Boot-Sektor der erweiterten Partition zerstören; in diesem Fall müsste LILO erneut installiert werden.

Die Deinstallation hängt stark vom verwendeten Bootmanager ab, entweder muss man die verwendete LILO-Bootpartition abmelden, oder der Bootmanager bietet selbst an, jede vorhandene Partition zu booten. Dann entfernt eine Repartitionierung oder ein Formatieren der Partitionen sowohl Linux als auch LILO.

D.2.3 LILO im Master-Boot-Record

Befindet sich Linux komplett auf der zweiten Festplatte und gibt es auf der ersten keine erweiterte Partition, so muss LILO in den MBR installiert werden. Dabei wird der ehemalige MBR überschrieben. Man sollte also vor einer solchen Installation vom alten MBR (in dem sich auch die Partitionstabelle befindet) ein Backup anlegen. Dazu bieten sich diverse DOS-Utilities an. Unter Linux kann ein Backup einfach durchgeführt werden:

2 Als unrühmliches Beispiel seien hier die Installationen einiger MS-DOS-Versionen genannt.

```
# dd if=/dev/hda of=/backup/MBR bs=512 count=1
```

Mit Hilfe von

```
# dd if=/backup/MBR of=/dev/hda bs=446 count=1
```

wird der MBR ohne die Partitionstabelle wieder zurückgeschrieben. Soll die alte Partitionstabelle ebenfalls restauriert werden, ist der Parameter `bs=512` zu substituieren. **Vorsicht!** Dabei kann man leicht seine Partitionstabelle zerstören!

D.2.4 LILO-Dateien

Die LILO-Dateien befinden sich normalerweise im Verzeichnis `/boot/`[3], die Konfigurationsdatei `lilo.conf` in `/etc/`. Die Map-Datei enthält die eigentlichen Informationen, die zum Booten des Kerns benötigt werden, und wird vom Map-Installer `/sbin/lilo` angelegt. Zur Installation des LILO muss die Konfigurationsdatei an die persönlichen Bedürfnisse angepasst werden.

Die Konfigurationsdatei

Die Konfigurationsdatei besteht im Prinzip aus Variablenzuweisungen. In jeder Zeile befindet sich entweder eine Flag-Variable oder eine Variablenzuweisung. Flag-Variablen sind einfache Bezeichner, Variablenzuweisungen bestehen aus dem Namen der Variablen, gefolgt von einem Gleichheitszeichen und dem Variablenwert. Zusätzlich ist die Konfigurationsdatei durch spezielle Variablenzuweisungen in Bootkonfigurationen geteilt, jede Bootkonfiguration bootet entweder einen Kern oder ein anderes Betriebssystem. Die folgenden Variablen gelten global für alle LILO-Konfigurationen.

`boot=`*Gerät* gibt an, welches Gerät (bzw. welche Diskpartition) den Boot-Sektor enthalten soll. Fehlt `boot`, wird der Boot-Sektor auf das aktuelle Root-Gerät gelegt.

`compact` schaltet einen Modus ein, in dem LILO Lese-Anforderungen von benachbarten Sektoren mit Hilfe einer einzelnen Anfrage an das BIOS durchzuführen versucht. Dies reduziert die Ladezeit drastisch, insbesondere beim Booten von Diskette.

`delay=`*Zehntel* gibt die Zeit in Zehntelsekunden an, die LILO auf einen Tastendruck warten soll, bevor die erste Bootkonfiguration gebootet wird. Ohne Angabe von `delay` bootet LILO sofort.

`linear` lässt LILO lineare statt der üblichen Sektor/Kopf/Zylinder-Adressen erzeugen. Lineare Adressen hängen nicht von der Geometrie des Geräts ab.

`install=`*Boot-Sektor* installiert statt des Standard-Boot-Sektors `/boot/boot.b` den angegebenen Boot-Sektor.

`disktab=`*Disktab* gibt den Pfad der *Disktab*-Datei (enthält die Geometriedaten besonderer Platten) an, falls sich diese nicht in `/boot/disktab` befindet.

3 In älteren LILO-Versionen auch in `/etc/lilo/`.

map=*Map-Datei* gibt den Pfad der Map-Datei an.

message=*Datei* gibt den Pfad einer Datei an, deren Inhalt als Startmeldung beim Booten angegeben werden soll. Wird kein `message` angegeben, erscheint die Meldung „LILO". Da diese Startmeldung in die Map-Datei eingefügt wird, muss nach jeder Änderung der Map-Installer `/sbin/lilo` gestartet werden.

verbose=*Stufe* stellt die *Debug*-Stufe für LILO ein. Dabei sind die Stufen 0 (keine Meldungen) bis 5 (alle Statusmeldungen) möglich.

backup=*Backup-Datei* gibt den Namen der Datei an, in der der ehemalige Boot-Sektor gespeichert wird. Sonst wird `/boot/boot.`*Gerätenummer* gewählt.

force-backup=*Backup-Datei* wie `backup`, die Datei wird jedoch überschrieben, falls sie schon existiert.

prompt erzwingt die Eingabe einer Boot-Konfiguration per Tastatur. Das heißt, LILO bootet nicht mehr automatisch die erste angegebene Konfiguration.

timeout=*Zehntel* setzt einen Timeout-Wert, nachdem eine Eingabe über die Tastatur erfolgt sein muss. Ansonsten wird die erste Konfiguration gebootet. Analog dazu wird die Eingabe eines Passwortes ungültig, wenn zwischen zwei Eingaben zu viel Zeit verstreicht. Standardmäßig ist dieser Wert unendlich.

serial=*Port, Bps Parität Bits* stellt die Parameter für die serielle Schnittstelle ein, falls LILO auch Eingaben von dieser akzeptieren soll. Falls eine der Komponenten *Bps*, *Parität* oder *Bits* weggelassen wird, müssen auch die folgenden entfallen. *Port* wählt eine der vier seriellen (Standard-) Schnittstellen, 0 entspricht `COM1` bzw. `/dev/ttyS0`. Es werden Baudraten von 100 bis 9.600 bps unterstützt; 2.400 bps ist die Standardeinstellung. Alle Paritätseinstellungen (`n` keine, `e` gerade und `o` ungerade) werden unterstützt sowie 7 oder 8 Bit Daten. Die Standardeinstellung ist also `serial=0,2400n8`.

ignore-table weist LILO an, beschädigte Partitionstabellen zu ignorieren.

fix-table erlaubt LILO die Anpassung der Sektor/Kopf/Zylinder-Adressen an die linearen Adressen in jeder Partition. Normalerweise beginnt jede Partition auf einer Zylindergrenze, manche anderen Betriebssysteme könnten dies jedoch ändern. Da LILO seinen Boot-Sektor nur auf Partitionen schreiben kann, bei denen beide Adressen gleich sind, lassen sich falsche 3D-Adressen mit Hilfe von `fix-table` korrigieren. Dieses garantiert jedoch nicht, dass diese Korrekturen erhalten bleiben, deshalb ist eine Neupartitionierung, die sich an Zylindergrenzen hält, dieser Option vorzuziehen.

password=*Passwort* setzt ein Passwort für alle Boot-Konfigurationen.

restricted lockert die Passwort-Beschränkung. Passwörter müssen nur dann angegeben werden, wenn man dem Kern zusätzlich Boot-Parameter übergeben will.

optional erlaubt das Fehlen eines der in einer Bootkonfiguration angegebenen Kerne. Ohne Angabe von `optional` bricht der Map-Installer mit einer Fehlermeldung ab.

Jede Bootkonfiguration für einen Linux-Kern wird mit den Zuweisungen

```
image=Kern
   label=Name
```

eingeleitet. `image` muss den Pfad des zu bootenden Kerns enthalten und `label` den Namen, unter dem der Kern vom LILO-Prompt aus ausgewählt werden kann. Ist als `image` ein Gerät wie z. B. `/dev/fd0` angegeben, muss zudem noch der Bereich, in dem sich der Kern befindet, mittels

```
range=Bereich
```

angegeben werden. Der Bereich ist entweder als `Startsektor-Endsektor` oder als `Startsektor + Länge` anzugeben, z. B. so:

```
image=/dev/fd0
   label=floppy
   range=1+512
```

Variablenzuweisungen innerhalb einer Boot-Konfiguration wirken gewissermaßen lokal. Folgende Zuweisungen sind möglich:

`append=`*String* übergibt die Zeichenkette *String* dem Kern als Boot-Parameter. Auf diese Weise lassen sich z. B. Hardwareparameter an die Linux-Gerätetreiber übergeben (siehe Abschnitt 7.4.3).

`literal=`*String* wie `append`, die Zeichenkette wird aber ausschließlich als Boot-Parameter übergeben! Weil dabei auch lebenswichtige Einstellungen verloren gehen könnten, kann `literal` nicht global angegeben werden.

`ramdisk=`*Größe* überschreibt die Standardeinstellung des Kerns für die Größe der RAM-Disk.

`read-only` gibt an, dass das Root-Dateisystem read-only gemountet werden soll.

`read-write` analog

`root=`*Gerät* gibt den Namen des Geräts an, auf dem sich das Root-Dateisystem befindet.

`vga=`*Modus* überschreibt den Standardvideomodus des Kerns. Als Einstellungen sind `normal`, `extended` und `ask` möglich. Zusätzlich kann auch die Nummer des Videomodus angegeben werden.

Die **Boot-Konfigurationen anderer Betriebssysteme** werden mit

```
other=Gerät
   label=Name
```

eingeleitet. `other` beschreibt das Gerät (bzw. die Partition), auf der sich der Boot-Sektor des fremden Betriebssystems befindet. Für fremde Betriebssysteme können folgende Variablen eingestellt werden:

`loader=`*Lader* gibt den Pfad des Laders an, der zum Booten des Betriebssystems benutzt werden soll. Standardmäßig wird `/boot/chain.b` gewählt. Zusätzlich enthält die LILO-Distribution folgende Lader:

os2_d.b — kann OS/2 von der zweiten Festplatte booten.

any_d.b — versucht, vor dem Booten des Betriebssystems die erste und die zweite Festplatte zu vertauschen, um so Betriebssysteme von der zweiten Festplatte zu booten.

`table=`*Gerät* gibt das Gerät an, auf dem sich die Partitionstabelle für das zu bootende Betriebssystem befindet. Fehlt die Angabe von `table`, reicht LILO keine Informationen über die Partitionstabelle an den Boot-Sektor des fremden Betriebssystems weiter.

`unsafe` schaltet die Überprüfung des zu bootenden Betriebssystems ab. Dieser Schalter sollte nur verwendet werden, wenn eine Konfiguration von Diskette booten soll. Ohne diesen Schalter müsste sonst bei jedem Start des Map-Installers die Boot-Diskette in das Laufwerk eingelegt werden.

Die Disktab-Datei

Die Disktab-Datei enthält Informationen über die Geometrie des Geräts, von dem LILO booten soll. Normalerweise können diese Informationen vom Gerätetreiber angefordert werden; eine Disktab-Datei ist also nur nötig, falls dies nicht funktioniert. LILO gibt dann die Fehlermeldung

```
geo_query_dev HDIO_GETGEO (dev …)
```

oder

```
HDIO_REQ not supported for your SCSI controller.
Please use /boot/disktab
```

aus. In diesem Fall müssen die Geometriedaten per Hand eingegeben werden:

```
# /boot/disktab - LILO Parametertabelle
#
# Diese Tabelle enthält die Geometrie-Parameter für SCSI und
# IDE-Disks, die nicht automatisch erkannt werden können.
# Einträge in dieser Datei überschreiben erkannte Parameter!
#
# Dev.  BIOS    Secs/   Heads/  Cylin-  Part.
# num.  code    track   cylin.  ders    offset
#                                      (optional)

#0x800  0x80    32      64      202     0       # /dev/sda
```

Dabei bedeuten die einzelnen Felder:

0x800 Die Gerätenummer als Kombination der Major- und Minor-Nummer.

0x80 Der BIOS-Code für dieses Laufwerk. 0x80 ist die erste Festplatte im System, 0x81 die zweite usw. Dabei wird das gesamte physische Gerät als Einheit und nicht einzelne Partitionen betrachtet!

32, 64, 202 Die Geometriedaten: Anzahl der Sektoren pro Spur, Anzahl der Köpfe und Zylinder.

0 Der Beginn der Partition in relativen Sektoren von Sektor 0 der Festplatte an. Da diese Information auch aus der Partitionstabelle gelesen werden kann, ist ihre Angabe optional.

D.2.5 LILO-Boot-Parameter

Wird beim Booten des LILO eine der Tasten Ctrl, Shift oder Alt gedrückt, waren CapsLock oder ScrollLock gesetzt oder die Direktive `prompt` angegeben, so geht LILO in den interaktiven Modus über. Um eine Boot-Konfiguration auszuwählen, muss der als *label* definierte Name eingegeben werden. Durch Drücken der Taste Tab werden alle verfügbaren Bootkonfigurationen angezeigt. Zusätzlich können, ähnlich wie beim Start eines Programms aus der Shell, Parameter übergeben werden. Diese Parameter ergeben zusammengesetzt die Kommandozeile, die LILO dem Kern beim Start übergibt. Manche der Parameter werden vom Kern und den Gerätetreibern ausgewertet. Später werden die Parameter, die ein Gleichheitszeichen = enthalten, in das Environment des Init-Programms aufgenommen, die anderen werden als Parameter übergeben.

Folgende Boot-Parameter werden unter anderem vom Kern bzw. vom Init-Programm erkannt:

`root=`*Gerät*

`ro` und `rw` mountet das Root-Dateisystem explizit read-only oder read/write.

`debug` Alle Meldungen des Kerns werden auf der Systemkonsole ausgegeben.

`vga=`*Videomodus* wählt den Standardvideomodus des Kerns.

`S` teilt dem Init-Programm mit, dass Linux im Single-User-Modus zu starten ist.

`reserve=`*Portadresse, Bereich, ...* verbietet die Erkennung der Hardware auf den I/O-Adressen von *Portadresse* bis *Portadresse+Bereich*. Meist wird Hardware von den Treibern durch das Schreiben und Lesen von magischen Werten auf Portadressen erkannt. Dies kann bei Hardware, die zufällig dieselben Ports belegt, zu undefiniertem Verhalten bis hin zum Absturz des Systems führen.[4] `reserve=0x300,8` verbietet also dem Kern, auf diesen Adressen nach Hardware zu suchen (siehe Abschnitt 7.2.3).

4 Insbesondere bei der ISA-Architektur, bei der nur zehn Bit der Portadresse auf dem Bus liegen, kann es so zu „ungewollten" Überschneidungen kommen. Dies ist auch der Grund dafür, warum S3-Karten scheinbar die Portadresse der vierten seriellen Schnittstelle belegen.

Eine weitergehende Beschreibung der Boot-Parameter der Linux Version 2.4 befindet sich in den Kernelquellen in der Datei `Documentation/kernel-parameters.txt`

Zusätzlich fügt LILO der Kommandozeile stets den Parameter `BOOT_IMAGE=`*label* hinzu sowie das Wort `auto`, falls automatisch die erste Boot-Konfiguration gebootet wurde. Die Übergabe der Kommandozeile an den Kern geschieht auf sehr einfache Weise: LILO schreibt die magische Zahl `0xA33F` auf die physische Adresse `0x9000:20` und den Offset der Adresse der Kommandozeile relativ zu `0x9000:0` nach `0x9000:22`.

D.2.6 LILO-Startmeldungen

Während des Boot-Prozesses gibt der Lader die Meldung „LILO“ aus. Falls der Ladeprozess abgebrochen wurde, können die bis dahin ausgegebenen Zeichen zur Fehlerdiagnose dienen. Einige dieser Fehlermeldungen sollten jedoch nicht auftreten, da sie nur durch eine Zerstörung des LILO oder ein fehlerhaftes BIOS hervorgerufen werden können.

keine Ausgabe Kein Teil von LILO wurde geladen. LILO wurde nicht installiert, oder die Partition, die LILO enthält, ist nicht aktiv.

L*Nummer* Die erste Stufe des Laders wurde geladen und gestartet, er kann aber die zweite Stufe nicht laden. Die zweistellige Fehlernummer charakterisiert das Problem (siehe Abschnitt D.2.7). Dies kann durch einen physischen Fehler auf der Festplatte oder Diskette oder durch eine falsche Geometrie (falsche Parameter in `disktab`) hervorgerufen werden.

LI Die erste Stufe des Laders konnte zwar die zweite Stufe laden, aber deren Abarbeitung schlug fehl. Dies kann durch eine falsche Geometrie oder durch das Verschieben der Datei `boot.b` ohne Neuinstallation des Laders hervorgerufen werden.

LIL Die zweite Stufe des Laders wurde gestartet, sie kann jedoch die Deskriptortabelle nicht aus der Map-Datei lesen. Dieser Fehler deutet auf einen physischen Defekt oder auf eine falsche Geometrie hin.

LIL? Die zweite Stufe des Laders wurde an eine falsche Adresse geladen. Dieses Verhalten resultiert aus denselben Gründen wie **LI**.

LIL– Die Deskriptortabelle ist fehlerhaft. Dieser Fehler deutet auf eine falsche Geometrie oder auf ein Verschieben der Datei `map` ohne eine Neuinstallation des Laders hin.

LILO Alle Teile des Laders wurden geladen.

D.2.7 Fehlermeldungen

Meldet das BIOS einen Fehler, während LILO einen Kern lädt, so wird die Fehlernummer angezeigt.

0x00 *Interner Fehler*
Dieser Fehler wird von der Sektorleseroutine erzeugt, wenn eine interne Inkonsistenz festgestellt wird. Die wahrscheinlichste Fehlerursache ist eine falsche Map-Datei.

0x01 *Illegaler Befehl*
Diese Fehlermeldung beschreibt einen internen LILO Fehler und sollte nicht auftreten.

0x02 *Adressmarke nicht gefunden*
Beim Lesen des Mediums trat ein Fehler auf.

0x03 *Diskette ist schreibgeschützt*
Diese Fehlermeldung sollte nicht auftreten.

0x04 *Sektor nicht gefunden*
Dieser Fehler wird von falschen Geometriedaten erzeugt. Falls von einer SCSI-Platte gebootet wird, erkennt der Kern die Geometriedaten nicht bzw. ist die Disktab-Datei falsch. Das Flag `compact` erzeugt in seltenen Fällen ebenfalls diesen Fehler.

0x06 *Change line aktiv*
Dieser Fehler wird normalerweise durch ein Öffnen und Schließen der Laufwerksklappe während des Bootens verursacht.

0x08 *DMA-Überlauf*
Diese Fehlermeldung tritt bei einem Fehler bei der Programmierung des DMA-Controllers durch LILO ein. Sie sollte nicht auftreten.

0x09 *DMA-Transfer über 64-KByte-Grenze*
Diese Fehlermeldung tritt bei einem Fehler bei der Programmierung des DMA-Controllers durch LILO ein. Sie sollte nicht auftreten.

0x0C *Ungültiges Medium*
Diese Fehlermeldung wird durch ein defektes Medium erzeugt. Sie sollte aber nicht auftreten.

0x10 *CRC-Fehler*
Die Daten auf dem Medium sind fehlerhaft. Eine Neuinstallation von LILO könnte helfen (um den Sektor neu zu schreiben). Falls dieser Fehler beim Booten von Festplatten auftritt, sollte mittels `fsck` die Liste der fehlerhaften Sektoren ergänzt werden.

0x20 *Controller-Fehler*
Diese Fehlermeldung tritt bei einem Fehler bei der Programmierung des Floppy-Controllers durch LILO ein. Sie sollte nicht auftreten.

0x40 *Seek-Fehler*
Diese Fehlermeldung zeugt von einem Problem auf dem Boot-Medium.

0x80 *Timeout*
Das Laufwerk ist nicht bereit. Die Laufwerksklappe könnte offen sein.

Generell ist, besonders beim Booten von einem Diskettenlaufwerk, eine Wiederholung des Boot-Versuchs eine gute Idee, falls nicht explizit eine andere mögliche Fehlerursache angegeben ist.

Pearson Education

ADDISON-WESLEY

E Nützliche Kernfunktionen

Erfahrungen sind wie Schrot im Hintern;
wer noch nichts abgekriegt hat,
bezweifelt, dass es so etwas überhaupt gibt.

Hans Lippmann

Beim Programmieren am Kern gibt es einige Aufgaben, die immer wieder erledigt werden müssen. Nun kann man im Linux-Kern nicht auf die komfortable C-Bibliothek zurückgreifen, die Lösungen für diese Aufgaben anbietet. Nichtsdestotrotz gibt es im Linux-Kern eine Vielzahl von Funktionen, die Entwicklern die Arbeit erleichtern.

Diese Funktionen sollen hier Gegenstand einer näheren Betrachtung sein, um bei zukünftigen Entwicklungen am Linux-Kern Neuimplementierungen bereits vorhandener Funktionalität zu ersparen. Die Kenntnis dieser Funktionen hilft außerdem beim Lesen und Verstehen der Kernquellen. Viele dieser Funktionen sind schon in den vorherigen Kapiteln vorgestellt worden und werden deshalb hier nur erwähnt.

Kernfunktion	`close()`	`dup()`	`execve()`
	`exit()`	`open()`	`setsid()`
	`wait()`	`write()`	

Der Kern stellt auch eine ganze Reihe von Funktionen bereit, die als Systemrufe bekannt sind. Sie können (mit der bekannten Funktionalität) im Kern benutzt werden und arbeiten über das Syscall-Makro.

Kernfunktion	`set_bit()`	`clear_bit()`
	`change_bit()`	`test_and_set_bit()`
	`test_and_clear_bit()`	`test_and_change_bit()`
	`__constant_test_bit()`	`__test_bit()`
	`find_first_zero_bit()`	`find_next_zero_bit()`
	`ffz()`	

Diese Funktionen sind als Inline-Funktionen in der Datei `<asm/bitops.h>` programmiert. Je nach Architektur handelt es sich um Assembler- oder C-Funktionen. Sie garan-

tieren atomare Bitoperationen. Bit 0 ist das unterste Bit von `addr`, Bit 32[1]das unterste von `addr+1`.

```
inline int set_bit(int nr, void * addr);
inline int clear_bit(int nr, void * addr);
inline int change_bit(int nr, void * addr);
inline int test_and_set_bit(int nr, volatile void * addr);
inline int test_and_clear_bit(int nr, volatile void * addr);
inline int test_and_change_bit(int nr, volatile void * addr);
inline int __constant_test_bit(int nr,
        const volatile void * addr);
inline int __test_bit(int nr, volatile void * addr);
inline int find_first_zero_bit(void * addr, unsigned size);
inline int find_next_zero_bit (void * addr, int size, int offset);
inline unsigned long ffz(unsigned long word);
```

Die Funktion `set_bit()` setzt das Bit `nr` an der Adresse `addr`. Die Funktion `clear_bit()` löscht das Bit `nr` an der Adresse `addr`, und `change_bit()` invertiert es. `test_and_set_bit()` testet einfach, ob das Bit gesetzt ist. War das Bit `nr` vorher 0, geben diese Funktionen 0 zurück, ansonsten einen Wert ungleich 0.

Die Funktion `find_first_zero_bit()` durchsucht einen Bereich dahingehend, ob sich darin ein Nullbit befindet. Der Bereich beginnt mit der Adresse `addr` und hat die Größe `size`. Demgegenüber durchsucht `find_next_zero_bit()` den Bereich ab einem Offset `offset`. Die Funktion `ffz()` sucht das erste freie Bit im Wert `word` und gibt seine Position zurück. Das Verhalten der Funktion ist nicht definiert, wenn der Wert kein Nullbit enthält[2]. Das muss vorher getestet werden. Alle drei Funktionen liefern die Position des gefundenen Bits zurück.

Kernfunktion `iget()` `iput()`

Diese zwei Funktionen, implementiert in `fs/inode.c`, ermöglichen den Zugriff auf eine bestimmte Inode-Struktur.

```
struct inode * iget(struct super_block * sb,unsigned long ino);
void iput(struct inode * inode);
```

Die Funktion `iget()` liefert die durch den Superblock `sb` und die Inode-Nummer `ino` beschriebene Inode und aktualisiert gleichzeitig die Hashtabelle (erhöht den Referenzzähler bzw. trägt ihn neu ein). Die mit `iget()` erhaltenen Inodes müssen mit `iput()` freigegeben werden. Diese Funktion dekrementiert den Referenzzähler um 1 und gibt die Inode frei, falls der Zähler 0 sein sollte.

1 Das gilt natürlich nur für Architekturen mit 32-Bit-Adressen.

2 Die i386-Version liefert 0.

Kernfunktion	`sprintf()` `vsprintf()`

Um verschiedene Daten in einem String unterzubringen, wird in C die Funktion `sprintf()` verwendet. Da die Kernprogrammierung unabhängig von der C-Bibliothek erfolgen muss, ist es recht praktisch, dass die Kernquellen eine solche Funktion zur Verfügung stellen.

```
int sprintf(char * buf, const char *fmt, …);
```

Diese baut aber nur ihre Argumente in variable Argumentlisten um und ruft `vsprintf()` auf. Der Rückgabewert ist die Anzahl der nach `buf` geschriebenen Zeichen. Die Funktion `vsprintf()` führt die eigentliche Umwandlung durch.

```
int vsprintf(char *buf, const char *fmt, va_list args);
```

Der String `fmt` enthält zwei Arten von Zeichen: normale Zeichen, die einfach kopiert werden, und Konvertierungsangaben, die jeweils die Umwandlung und Ausgabe des nächsten Arguments veranlassen. Jede Konvertierungsangabe beginnt mit einem Prozentzeichen und endet mit einem Umwandlungszeichen. Dazwischen können, in der angegebenen Reihenfolge, die nächsten Angaben stehen:

Steuerzeichen — Ein oder mehrere Zeichen in beliebiger Reihenfolge, die die Umwandlung modifizieren. Möglich sind folgende Zeichen:

Ein Leerzeichen. Wenn das erste Zeichen kein Vorzeichen ist, wird ein Leerzeichen eingefügt.

- Das umgewandelte Argument wird linksbündig ausgegeben.

+ Wenn die Zahl größer 0 ist, wird ein + ausgegeben.

Zahlen werden alternativ ausgegeben. Bei gesetztem Oktalformat wird der Zahl eine `0` vorangestellt, beim Hexadezimalformat ein `0x` bzw. `0X`.

0 Zahlenangaben werden bis zur Feldbreite mit führenden Nullen aufgefüllt.

Zahl — Die minimale Feldbreite wird festgelegt.

Punkt — Er trennt die Feldbreite von der Genauigkeit.

Zahl — Die Genauigkeit: Sie legt die maximale Anzahl von Zeichen fest, die von einer Zeichenkette ausgegeben werden soll, oder die Anzahl der Ziffern, die nach einem Komma ausgegeben werden sollen.

Buchstabe — Ein `h` steht für *short*-Argumente und ein `l` oder `L` für *long*-Argumente.

Die folgende Aufzählung erklärt die Umwandlungszeichen:

c — Typ `int`. Das Zeichen `arg` wird ausgegeben.

s — Typ `char *`. Die Zeichenkette `arg` wird bis zum Nullbyte oder der geforderten Genauigkeit ausgegeben.

p — Typ `void *`. Die Adresse `arg` wird in hexadezimalem Format ausgegeben.

n — Typ `int`. Das Argument `arg` ist ein Zeiger auf einen Integer (bei `h`) oder ein Long (bei `l`). Darin wird die Anzahl der bisher ausgegebenen Zeichen abgespeichert.

o — Typ `int`. Die Zahl `arg` wird in oktalem Format (ohne führende `0`) ausgegeben.

x — Typ `int`. Die Zahl `arg` wird in hexadezimalem Format (mit Kleinbuchstaben und ohne führende `0x`) ausgegeben.

X — Typ `int`. Die Zahl `arg` wird in hexadezimalem Format (mit Großbuchstaben und ohne führende `0X`) ausgegeben.

d, i — Typ `int`. Die Zahl `arg` wird mit Vorzeichen ausgegeben.

u — Typ `int`. Die Zahl `arg` wird ohne Vorzeichen ausgegeben.

Der String `buf` wird zum Schluss mit einem Nullbyte versehen. Die Anzahl der tatsächlich in `buf` geschriebenen Zeichen wird zurückgegeben.

Kernfunktion `printk()`

Niemand, der schon umfangreichere Stücke Software geschrieben hat, wird die Zweckmäßigkeit von Kontroll- und Debug-Ausschriften leugnen. Da es sich bei Linux inzwischen auch um ein solches Stück handelt, darf eine Funktion dafür natürlich nicht fehlen. Um die Möglichkeiten des bekannten `printf()` zu bieten und zu erweitern, stellt der Kern die Funktion `printk()` zur Verfügung:

```
int printk(const char *fmt, …);
```

Der Funktion `printk()` werden die Parameter in derselben Art und Weise wie `printf()` übergeben. Zusätzlich kann an erster Stelle ein Makro stehen, das die Wichtigkeit (im Weiteren Level genannt) der Nachricht festlegt. Möglich sind die Makros:

```
#define KERN_EMERG   "<0>" /* System nicht mehr nutzbar  */
#define KERN_ALERT   "<1>" /* Aktion sofort beenden      */
#define KERN_CRIT    "<2>" /* kritische Bedingung        */
#define KERN_ERR     "<3>" /* Fehler                     */
#define KERN_WARNING "<4>" /* Warnung                    */
#define KERN_NOTICE  "<5>" /* normal, aber bemerkenswert */
#define KERN_INFO    "<6>" /* Information                */
#define KERN_DEBUG   "<7>" /* Debug-Level                */
```

Ist kein Level angegeben, wird der Default-Level eingetragen. Der Default-Level ist normalerweise[3] `KERN_WARNING`.

3 Selbstverständlich kann auch ein anderer Level als Standard festgelegt werden. Steht „debug“ in den LILO-Boot-Parametern, wird der Default-Loglevel auf 10 gesetzt.

Die Nachrichten, die mit `printk()` ausgegeben werden, landen erstens im Logbuch, einem Speicher der Größe 16 Kbyte, und werden zweitens, je nach Level, auf die Konsole geschrieben:

```
#define LOG_BUF_LEN     (16384)
static char log_buf[LOG_BUF_LEN];

unsigned long log_size = 0;
static unsigned long log_start = 0;
static unsigned long logged_chars = 0;
```

Die erste Variable beschreibt die Größe des Logbuches. Sie kann zwischen 0 und `LOG_BUF_LEN` schwanken. Die zweite Variante gibt den Beginn der aktuellen Nachricht an. Mit dem Zugriff `(log_start+log_size)&(LOG_BUF_LEN-1)` erhält man also die letzte Stelle der aktuellen Eintragung. Die Gesamtzahl der Zeichen im Logbuch steht in `logged_chars`.

Im Folgenden wird die Abarbeitung der Funktion beschrieben. Zuerst sichert `printk()` die Flags des Prozessors und sperrt alle Interrupts. Der übergebene String (und die eventuell darin enthaltenen Parameter) werden in einen internen 1024 Byte großen Puffer kopiert. Ein Überlauf wird nicht getestet, und ein größerer String kann daher den Kern zum Absturz bringen. Beim Kopieren werden die ersten drei Zeichen für einen eventuell nachzutragenden Level freigehalten.

```
spin_lock_irqsave(&console_lock, flags);
va_start(args, fmt);
i = vsprintf(buf + 3, fmt, args);
                              /* hopefully i < sizeof(buf)-4 */
buf_end = buf + 3 + i;
va_end(args);
```

Jetzt werden die Nachrichten ausgeschrieben. Wurde beim Aufruf kein Level angegeben, wird zusätzlich nun `KERN_INFO` als Level eingetragen.

```
for (p = buf + 3; p < buf_end; p++) {
    msg = p;
    if (msg_level < 0) {
        if ( p[0] != '<' || p[1] < '0' ||
             p[1] > '7' || p[2] != '>') {
                p -= 3;
                p[0] = '<';
                p[1] = default_message_loglevel + '0';
                p[2] = '>';
        } else
                msg += 3;
        msg_level = p[1] - '0';
    }
```

Nun wird die Nachricht in den dafür im Kern vorgesehenen Speicher geschrieben — in das *Logbuch* des Systems. Da dieser Speicher als Ringpuffer angelegt ist, wird beim Überschreiten seiner Größe wieder am Anfang angefangen.

```
        for (; p < buf_end; p++) {
            log_buf[(log_start+log_size) & (LOG_BUF_LEN-1)] = *p;

            if (log_size < LOG_BUF_LEN)
                log_size++;
            else {
                log_start++;
                log_start &= LOG_BUF_LEN-1;
            }
            logged_chars++;
            if (*p == '\n') {
              linefeed = 1; break;
            }
        }
```

Die Anzahl der geschriebenen Zeichen wird selbstverständlich mitgezählt. Sollte ein `printk()`-Aufruf mehrere, mittels `'\n'` getrennte, Nachrichten enthalten, werden diese getrennt behandelt.

Nun wird die Nachricht auf die Konsole geschrieben. Nur Mitteilungen, die wichtig genug sind (d.h. in der Priorität höher als der Loglevel liegen), werden ausgegeben. Der Loglevel kann durch den Systemruf *syslog*, siehe Seite 343, eingestellt werden. Während `p` nun auf das Ende der aktuellen Nachricht verweist, zeigt `msg` noch auf den Anfang.

```
        if (msg_level < console_loglevel && console_drivers) {
          struct console *c = console_drivers;
          while(c) {
            if ((c->flags & CON_ENABLED) && c->write)
                c->write(c, msg, p - msg + line_feed);
            c = c->next;
          }
        }
```

Ist die aktuelle Nachricht zu Ende, muss eine neue Priorität ermittelt werden.

```
        if (linefeed) msg_level = -1;
    }
```

Zum Schluss werden die Flags wieder auf den Stack gelegt und alle Prozesse aufgeweckt, die durch das Lesen in einem leeren Logbuch blockiert sind.

```
    spin_unlock_irqrestore(&console_lock, flags);
    wake_up_interruptible(&log_wait);
    return i;
}
```

Zurückgegeben wird die Anzahl der in das Logbuch geschriebenen Zeichen.

Kernfunktion `panic()`

Die Funktion `panic()` ist ein `printk()` mit der festen Priorität `KERN_EMERG`. Zusätzlich wird die Kernfunktion `sys_sync()` aufgerufen, vorausgesetzt, der betroffene Prozess ist nicht die *Swap Task*, oder die CPU befindet sich in einer Interrupt-Behandlungsroutine. Dann wird noch `panic_timeout` Sekunden gewartet und anschließend der Rechner neu gestartet.

```
NORET_TYPE void panic(const char * fmt, …);
```

Kernfunktion `memcpy()` `bcopy()` `memset()` `memcmp()` `memmove()` `memscan()`

Diese Funktionen bearbeiten Speicherbereiche. Normalerweise handelt es sich um Speicher in der Form von Zeichenketten, die nicht durch ein Nullbyte abgeschlossen sind. Keine der folgenden Funktionen testet Bereichsüberschreitungen. Die Funktionen ähneln nicht ohne Grund denen zur Stringbehandlung. Meistens sind dieselben Algorithmen implementiert, nur eine Längenangabe ist zusätzlich nötig.

```
void * memset(void * s,char c,size_t count);
```

Die Funktion `memset()` füllt den mit `s` adressierten Bereich der Größe `count` mit dem Zeichen `c`.

```
void * memcpy(void * dest,const void *src,size_t count);
char * bcopy(const char * src, char * dest, int count);
```

Beide Funktionen haben dieselbe Wirkung: Der Bereich `dest` wird mit den ersten `count` Zeichen aus `src` gefüllt.

```
void * memmove(void * dest,const void *src,size_t count);
```

Diese Funktion kopiert `count` Zeichen aus dem Bereich `src` in den Bereich `dest`. Sie ist aber etwas intelligenter als die vorherige Funktion. Sie stellt fest, ob das Kopieren die Quelle überschreiben könnte (Zieladresse kleiner als Quelladresse), dann beginnt sie mit dem Kopieren am Ende des Bereichs.

```
int memcmp(const void * cs,const void * ct,size_t count);
```

Die Funktion `memcmp()` vergleicht zwei Speicherbereiche maximal `count` Zeichen lang. Zurückgegeben wird eine Zahl größer Null, gleich Null oder kleiner Null, je nachdem ob `cs` lexikographisch größer, gleich oder kleiner als `ct` ist.

```
void * memscan(void * addr, unsigned char c, size_t size);
```

Die Funktion `memscan()` durchsucht `size` Zeichen im Bereich `addr` nach dem Zeichen `c`. Zurückgegeben wird die Adresse des ersten Vorkommens.

Kernfunktion	`register_chrdev()`	`unregister_chrdev()`
	`register_blkdev()`	`unregister_blkdev()`

Beide Registrierfunktionen melden einen Gerätetreiber im Kern an. Die erste registriert einen Treiber für Zeichengeräte, die zweite einen Treiber für Blockgeräte.

```
int register_chrdev(unsigned int major, const char * name,
                    struct file_operations *fops);
int unregister_chrdev(unsigned int major, const char * name);
int register_blkdev(unsigned int major, const char * name,
                    struct file_operations *fops);
int unregister_blkdev(unsigned int major, const char * name);
```

Die Parameter sind in beiden Fällen gleich. Sie geben an, wie und unter welchen Namen das Gerät registiert werden soll.

`major` — die gewünschte Major-Nummer des Gerätes

`name` — der symbolische Name , zum Beispiel `"tty"` oder `"lp"`

`fops` — die Adresse der `file_operations`-Struktur

Der Treiber wird, mit der Major-Nummer als Index, in die entsprechende Tabelle im Kern eingetragen. Ist die Major-Nummer 0, wird der letzte freie Eintrag verwendet, die Werte dort eingetragen und sein Index zurückgegeben. Ist kein Eintrag in der Tabelle frei oder die Major-Nummer bereits belegt, gibt die Funktion `-EBUSY` zurück.

Die `unregister`-Funktionen melden den Treiber wieder ab, indem sie `NULL` in die Tabelle eintragen.

Kernfunktion	`register_binfmt()`	`unregister_binfmt()`
	`register_exec_domain()`	`unregister_exec_domain()`
	`register_filesystem()`	`unregister_filesystem()`

Diese Funktionen melden Binärformate und Dateisysteme im Kern an und ab. Im Gegensatz zu den Gerätetreibern ist die Anzahl dieser Formate beliebig, da sie nicht in einer Tabelle, sondern in Listen verwaltet werden. Wird versucht, das gleiche Format zweimal im Kern anzumelden, liefert die Funktion ein `-EBUSY` zurück.

```
int register_binfmt(struct linux_binfmt * fmt);
int unregister_binfmt(struct linux_binfmt * fmt);
```

Die Funktion `register_binfmt()` macht dem Kern ein Binärformat bekannt, indem sie es in die Liste der bekannten Formate einhängt. Ein Binärformat hat folgende Struktur:

```
struct linux_binfmt {
   struct linux_binfmt * next;
   struct module *module;
   int (*load_binary)
       (struct linux_binprm *, struct  pt_regs * regs);
   int (*load_shlib)(int fd);
   int (*core_dump)(long signr, struct pt_regs * regs);
};
```

Ein bekanntes Binärformat kann mit `unregister_binfmt()` wieder entfernt werden.

```
int register_exec_domain(struct exec_domain *it);
int unregister_exec_domain(struct exec_domain *it);
```

Die Funktion `register_exec_domain()` meldet eine neue *Exec Domain* im Kern an. Mit `unregister_exec_domain()` wird eine Domain abgemeldet. Eine Exec-Domain hat folgenden Aufbau:

```
struct exec_domain {
   char *name;
   lcall7_func handler;
   unsigned char pers_low, pers_high;
   unsigned long * signal_map;
   unsigned long * signal_invmap;
   struct module *module;
   struct exec_domain *next;
};
```

```
int register_filesystem(struct file_system_type * fs);
int unregister_filesystem(struct file_system_type * fs);
```

Diese Funktionen melden einen Dateisystemtyp im Kern an und ab. Die übergebene Struktur hat folgendes Aussehen:

```
struct file_system_type {
   char *name;
   int fs_flags;
   struct super_block *(*read_super)
      (struct super_block *, void *, int);
   struct file_system_type * next;
};
```

Kernfunktion	**register_serial()**	**unregister_serial()**
	register_netdev()	**unregister_netdev()**

Diese Funktionen melden Geräte an, die von den „normalen“ Geräten in UNIX abweichen.

```
int register_serial(struct serial_struct *req);
int unregister_serial(int line);
```

Normalerweise ist die Anzahl der seriellen Geräte eines Computers fest und ändert sich nicht während des Betriebs. Diese Funktion ist auch nur für Besitzer einer PCMCIA-Schnittstelle (Laptop-Besitzer) interessant, da mit ihrer Hilfe ein serieller Port zur Laufzeit eingebunden werden kann. Die Struktur wird in die Tabelle `rs_table[]` eingetragen, und ihr Index wird zurückgegeben. Kann kein passender Port in der Tabelle gefunden werden, wird der erste unbelegte Port belegt, der frei ist. Das Abmelden geschieht mittels `unregister_serial()`. Der Parameter `line` bestimmt den abzumeldenden Port. Sollte dieser noch mit einem Terminal verbunden sein, wird es (mittels `tty_hangup()`) geschlossen.

```
int register_netdev(struct device *dev);
void unregister_netdev(struct device *dev);
```

Diese Funktionen melden abstrakte Netzwerkgeräte an und ab. Das Arbeiten mit diesen Funktionen ist, nicht zuletzt bedingt durch die Größe der Struktur `device`, nicht einfach. Für die genauere Funktion sei deshalb auf Abschnitt 8.3 verwiesen.

Kernfunktion	**tty_register_driver()**
	tty_unregister_driver()

```
int tty_register_driver(struct tty_driver *driver);
int tty_unregister_driver(struct tty_driver *driver);
```

Diese Funktionen registrieren einen neuen TTY-Treiber im Kern. Beim Anmelden eines Treibers wird intern `register_chrdev()` aufgerufen. In `driver` stehen die Major-Nummer und der Name dafür, als File-Operationen werden die Standard-TTY-Operationen eingetragen. Besitzt der Treiber noch keine Funktion, um ein einzelnes Zeichen auszugeben, trägt die Registrierfunktion eine ein. Die Abmeldung des Treibers erfolgt mittels `tty_unregister_driver()`. Sollte der Treiber noch benutzt werden, wird `-EBUSY` zurückgegeben.

Kernfunktion	**strcpy()**	**strncpy()**	**strchr()**
	strcat()	**strncat()**	**strspn()**
	strcmp()	**strncmp()**	**strpbrk()**
	strlen()	**strnlen()**	**strtok()**

Die hier aufgeführten Funktionen stellen die meisten aus der C-Bibliothek bekannten Routinen zur Verfügung. Sie sind in der Datei `lib/string.c` als generische C-Funk-

tionen und in `<asm/string.h>` für die jeweilige Prozessorarchitektur als optimierte Inline-Assembler-Funktionen definiert.

```
char * strcpy(char * dest,const char *src);
char * strncpy(char * dest,const char *src,size_t count);
```

Die Funktion `strcpy()` kopiert den String `src` in den durch `dest` adressierten Puffer, einschließlich dem angehängten Nullbyte. Die Strings sollten sich nicht überlagern, und der Zielpuffer sollte genügend Fassungsvermögen haben. Die Funktion `strncpy()` arbeitet äquivalent, wobei nur die ersten `count` Zeichen kopiert werden. Sollte `src` (einschließlich Nullbyte) kürzer als `count` sein, wird nur `src` kopiert. Beide Funktionen geben den Pointer `dest` zurück.

```
char * strcat(char * dest, const char * src);
char * strncat(char *dest, const char *src, size_t count);
```

Die Funktion `strcat()` hängt eine Kopie des Strings `src` an den String `dest` an. Die Funktion `strncat()` hängt maximal `count` Zeichen des Strings `src` an den String `dest` an.

```
int strcmp(const char * cs,const char * ct);
int strncmp(const char * cs,const char * ct, size_t count);
```

Die Funktion `strcmp()` vergleicht den String `cs` mit `ct` zeichenweise über die Länge von `cs`. Zurückgegeben wird eine Zahl größer als Null, gleich Null oder kleiner als Null, je nachdem, ob `cs` lexikographisch größer, gleich oder kleiner als `ct` ist. Die Funktion `strncmp()` vergleicht maximal `count` Zeichen.

```
char * strchr(const char * s,char c);
```

Die Funktion `strchr()` liefert einen Zeiger auf das erste Auftreten des Zeichens `c` im String `s`. Ist das Zeichen nicht enthalten, wird ein `NULL`-Zeiger zurückgegeben.

```
size_t strlen(const char * s);
size_t strnlen(const char * s, size_t count);
```

Die Funktion `strlen()` liefert die Anzahl der Zeichen in `s`, ohne das Nullbyte. Demgegenüber liefert `strnlen()` das Minimum aus Stringlänge und `count`. Verwendet wird diese Funktion von `vsprintf()` bei Formatanweisungen für Strings.

```
size_t strspn(const char *s, const char *accept);
char * strpbrk(const char * cs,const char * ct);
```

Die Funktion `strspn()` liefert die Größe des Anfangsstücks von `s`, das keine Zeichen aus `accept` enthält. Die Funktion `strpbrk()` liefert einen Zeiger auf die Stelle von `cs`, an der zum ersten Mal ein Zeichen aus `ct` vorkommt.

```
char * strtok(char * s,const char * ct);
```

Die Funktion `strtok()` liefert die erste Zeichenkette (*Token*) des Strings `s`, die keine Zeichen aus `ct` enthält. Wiederholte Aufrufe der Funktion mit einem `NULL`-Zeiger für `s` zerlegen den String in eine Folge von Zeichenketten.

Kernfunktion `simple_strtoul()`

```
unsigned long simple_strtoul(const char *cp,
                    char **endp,unsigned int base);
```

Das ständige Problem der Umwandlung eines Strings in eine Zahl wird durch diese Funktion bewältigt. Der String `cp` enthält die umzuwandelnde Zahl. `base` enthält die Basis des zu verwendenden Zahlensystems. Steht darin eine 0, wird versucht, die Basis automatisch zu bestimmen. Der Standard ist Basis 10; beginnt `cp` aber mit „`0`“, wird die Basis 8 und bei „`0x`“ die Basis 16 verwendet. Dann wird `cp` zeichenweise gelesen und umgewandelt, bis ein Zeichen nicht mehr zum Zahlensystem passt. Der verbleibende Reststring wird in `endp` gespeichert, und das errechnete Resultat wird zurückgegeben.

Kernfunktion `verify_area()`

```
int verify_area(int type, const void * addr,
                unsigned long size);
```

Diese Funktion überprüft, ob eine Operation `type` für einem Speicherbereich erlaubt ist, der durch die Adresse `addr` und die Größe `size` definiert ist. Zwei Operationen sind möglich: `VERIFY_WRITE` als `type` testet, ob ein Schreibzugriff erlaubt ist; `VERIFY_READ` testet den Lesezugriff.

Kernfunktion `get_user()` `put_user()` `copy_to_user()` `copy_from_user()`

Die ersten zwei Funktionen ermöglichen den Zugriff auf Daten, die im Adressraum des Nutzers liegen. Die Größe des Pointers wird selbst festgestellt. Das Makro `get_user()` liest den Wert `x` von der Adresse `ptr`, und `put_user()` schreibt ihn.

Für das Kopieren größerer Datenbereiche gibt es die `copy`-Funktionen. Sie kopieren einen Datenbereich der Größe `size` vom Kernel-Adressraum in den User-Adressraum (`to`) oder umgekehrt.

Kernfunktion `suser()` `fsuser()`

Diese zwei Funktionen prüfen, ob ein Prozess Superuserrechte bzw. Superuserrechte gegenüber dem Dateisystem hat. Beides sind einfache Funktionen in der Headerdatei `<linux/sched.h>` der Form:

```
extern inline int suser(void)
{
  if (!issecure(SECURE_NOROOT) && current->euid == 0) {
    current->flags |= PF_SUPERPRIV; return 1;
  }
  return 0;
}
```

Neue Programme sollten die Rechte des Prozesses prüfen und dafür die Funktion `capable()` benutzen!

Kernfunktion `capable()`

Mit dieser Funktion kann jetzt getestet werden, ob für den aktuellen Prozess ein bestimmtes Recht gesetzt ist.

```
#define cap_raised(c, flag)  ((c) & (1 << (flag)))

extern inline int capable(int cap)
{
  if (cap_raised(current->cap_effective, cap)) {
    current->flags |= PF_SUPERPRIV;
    return 1;
  }
  return 0;
}
```

Kernfunktion `cap_emulate_setxuid()`

Diese Funktion wird aufgerufen, wenn sich eine der UIDs[4] des aktuellen Prozesses ändert. Dann werden nämlich die Rechte verändert. Unterschieden werden drei Fälle:

1. Vorher war mindestens eine UID 0 und jetzt sind alle UIDs ungleich 0. Dann werden alle erlaubten und effektiven Rechte gelöscht.
2. Die EUID ändert sich von 0 in einen anderen Wert. Dann werden alle effektiven Rechte gelöscht.
3. Die EUID ändert sich von einem Wert ungleich 0 in 0. Dann werden die erlaubten Rechte als effektive Rechte eingetragen.

4 UIDs bedeutet hier entweder UID, EUID oder SUID.

Kernfunktion `add_wait_queue()` `remove_wait_queue()`

Das Verwalten von Warteschlangen ist nicht kompliziert. Aber es muss sichergestellt werden, dass nicht zwei Prozesse oder Interrupts gleichzeitig eine Warteschlange modifizieren. Deswegen dürfen für den Zugriff nur diese zwei Funktionen verwendet werden.

```
inline void add_wait_queue(struct wait_queue ** p,
                struct wait_queue * wait);
inline void remove_wait_queue(struct wait_queue ** p,
                struct wait_queue * wait);
```

Die erste Funktion trägt den Eintrag `wait` in die Warteschlange `p` ein (logischerweise als ersten Eintrag), die zweite entfernt ihn wieder. Beide Funktionen laufen atomar ab.

Kernfunktion `up()` `down()`

```
extern inline void up(struct semaphore * sem);
extern inline void down(struct semaphore * sem);
```

Diese zwei Funktionen ermöglichen die Synchronisation zwischen Prozessen mittels eines Semaphors:

```
struct semaphore {
  atomic_t count;
  int sleepers;
  struct wait_queue_head_t wait;
};
```

`down()` testet, ob der Semaphor noch frei (größer oder gleich 0) ist, und dekrementiert ihn bei Erfolg. Ansonsten trägt sich der Prozess in eine Warteschlange ein und blockiert, bis der Semaphor wieder frei wird.

`up()` erhöht den Semaphor um 1. War der Wert des Semaphors vorher negativ, wird ein `wake_up()` auf die zum Semaphor gehörende Warteschlange ausgeführt.

Kernfunktion `list_add()` `list_del()` `list_empty()` `list_entry()` `list_splice()`

In der Linux-Version 2.2 wurden die meisten dynamisch erzeugten Datenstrukturen auf eine generische Listenimplementation umgestellt. Diese Umstellung stellt sicher, dass die anfallenden Listen-Operationen, wie das Hinzufügen oder Entfernen von Objekten, automatisch richtig implementiert sind.

Die generische Listenimplementation unterstützt doppelt verkettete Listen und basiert auf der Struktur `list_head`. Sie enthält jeweils einen Zeiger auf das vorherige und das nächste Element in einer Liste.

```
struct list_head {
   struct list_head *next, *prev;
};
```

Diese Struktur wird nun benutzt, um sowohl Startpunkte (Anker) als auch Elemente einer Liste zu bilden. Zunächst dient das Makro `LIST_HEAD` zur Definition einer leeren Liste. Die Liste ist leer, da sowohl das erste als auch das letzte Element auf den Anker zeigt.

```
#define LIST_HEAD(name) \
   struct list_head name = { &name, &name }
```

Analog dazu initialisiert das Makro `INIT_LIST_HEAD` einen bereits definierten Anker vom Typ `struct list_head *`:

```
#define INIT_LIST_HEAD(ptr) do { \
   (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)
```

Das Anfügen von Elementen erledigt die Funktion `list_add()`. Sie fügt das Element `new` hinter dem Element `head` ein. `head` kann dabei auch der Anker der Liste sein. Die Funktion `list_del()` entfernt das Element `entry` aus der Liste. Da es sich um eine doppelt verkettete Liste handelt, muss dazu ihr Anker nicht bekannt sein.

```
void list_add(struct list_head *new, struct list_head *head);
void list_del(struct list_head *entry);
```

Zusätzlich ist es möglich, ein Element einer Liste gegen ein anderes auszutauschen. Dazu dient die Funktion `list_splice()`. Sie ersetzt das Element `list` durch das Element `head`.

```
void list_splice(struct list_head *list, struct list_head *head);
```

Um nun eine eigene Liste von Elementen des Typs `struct T` zu erzeugen, sind mehrere Schritte notwendig. Zunächst muss eine Komponente vom Typ `list_head` eingefügt werden.

```
struct T {
   struct list_head   t_list;   /* verkettet struct T */
   …                           /* Rest der Struktur */
};
```

Nachdem ein Anker definiert wurde, können Elemente verkettet werden:

```
LIST_HEAD(liste);

struct T a, b;
```

```
list_add(a, liste);
list_add(b, liste);
```

Wie greift man nun auf die einzelnen Elemente einer Liste zu, wenn doch eigentlich nur ihre Komponenten `t_list` miteinander verkettet sind? Dazu dient das Makro `list_entry`:

```
#define list_entry(ptr, type, member) \
   ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

Es sorgt dafür, dass die verkette Komponente in einen Zeiger auf die verkettete Struktur „umgerechnet" wird. Dazu benötigt es ein Argument vom Typ `list_head`, den Typ der verketteten Struktur sowie den Namen der Komponente, über die die Verkettung erfolgt.

```
struct *T elem;

/* holt erstes Element der Liste */
elem = list_entry(liste.next, struct T, t_list);

/* holt nächstes Element hinter a */
if (a.t_list.next != &liste)
   elem = list_entry(a.t_list.next, struct T, t_list);

/* holt Elemtent vor b */
if (b.t_list.prev != &liste)
   elem = list_entry(b.t_list.prev, struct T, t_list);
```

Da `list_entry` ein Makro ist, muss man jedoch dafür sorgen, dass das Argument `ptr` auf ein Element zeigt. Dies geschieht durch einen Vergleich auf den Beginn der Liste. Da `LIST_HEAD` den Anfang und das Ende der Liste stets auf den Anker initialisiert, lassen sich so leicht beide Enden einer Liste erkennen.

Schließlich existiert auch eine Funktion zur Erkennung einer leeren Liste.

```
int list_empty(struct list_head *head)
{
   return head->next == head;
}
```

Literaturverzeichnis

[Bac86] Maurice J. Bach. *The Design of the UNIX(R) Operating System*. Prentice Hall International, Inc., London, 1986.

Bach beschreibt den Aufbau und die Funktionsweise von UNIX System V. Dies ist zusammen mit [LMKQ89] die Standardliteratur zum Thema Unix-Betriebssystemimplementation.

[Bac98] Jean Bacon. *Concurrent Systems*. Addison-Wesley, second edition, 1998.

Ein gutes Grundlagenwerk über parallele Systeme

[BC01] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., 2001.

Ein hervorragendes Buch über den Linux Kernel 2.2 und die Implementation auf i386 Hardware

[Bur95] Richard A. Burgess. *Developing your own 32-Bit Operating System*. Sams Publishing, 1995.

Dieses Buch beschreibt den Aufbau und die Funktionsweise von MMURTL, einem Multitasking-Betriebssystem für 80386-Systeme. Der vollständige Sourcecode ist enthalten, so daß Experimenten nichts im Wege steht.

[CDM98] Remy Card, Eric Dumas, and Franck Mevel. *The Linux Kernel book*. John Wiley & Sons Ltd., 1998.

Das Konkurrenzwerk zu unserem Buch. Remy Card ist der Programmierer des Standarddateisystems ext2fs für Linux.

[Cla90] Ludwig Claßen. *Programmierhandbuch 80386/80486*. Verlag Technik, Berlin, 1990.

Sehr kompakte Einführung in die Programmierung des 80386. Teilweise sind Bezüge auf [CW90] vorhanden.

[CO87] Ludwig Claßen und Ulrich Oefler. *UNIX und C — Ein Anwenderhandbuch*. Verlag Technik, Berlin, 2. Auflage, 1987.

Eine Einführung in Unix und die Programmiersprache C. Da sich das Buch auf Unix Version 7 bezieht, ist es vielleicht nicht mehr ganz aktuell, aber immer noch lesbar.

[Com91] Douglas E. Comer. *Internetworking With TCP/IP*, Volume I — Principles, Protocols and Architecture. Prentice Hall International, Inc., London, second edition, 1991.

Dies ist das Standardwerk zu TCP/IP. Es werden alle Basis-Protokolle (u.a. ARP, TCP, IP, RIP) erläutert.

[CS91] Douglas E. Comer and David L. Stevens. *Internetworking With TCP/IP*, Volume II — Design, Implementation, and Internals. Prentice Hall International, Inc., London, first edition, 1991.

Hier wird die TCP/IP-Implementierung des Xinu-Systems erläutert. Das Xinu-System ist eine freie Implementierung eines UNIX-kompatiblen Systems.

[CW90] Ludwig Claßen und Ulrich Wiesner. *Wissensspeicher 80286-Programmierung*. Verlag Technik, Berlin, 1990.

[DC85] S. E. Deering and D. R. Cheriton. *RFC 966* — host groups: A multicast extension to the internet protocol, Dezember 1985.

Das IGMP-Protokoll wird hier beschrieben. Um IP-Multicast-Pakete effizient zu routen, ist ein Informationsaustausch zwischen den IP-Routern notwendig. Zu diesem Zweck wurde ein neues Protokoll geschaffen, das stark an das ICMP angelehnt ist.

[Dee89] S. Deering. *RFC 1112* — host extensions for ip multicasting, August 1989.

Dieser RFC behandelt die notwendigen Erweiterungen und Änderungen, die in den Netzwerkimplementationen vorzunehmen sind, um IP-Multicast zu unterstützen. Dabei geht dieser RFC nur auf die Notwendigkeiten der einzelnen Rechner ein.

[ELF] Executable and linkable format (elf).

Eine Beschreibung des ELF-Binärformats. Sie ist auf vielen großen FTP-Servern zu finden. So unter anderem in gepackter Form auf [FTP] in der Datei `/pub/os/linux/packages/GCC/ELF.doc.tar.gz`.

[Fei93] Sidnie Feit. * *TCP/IP — Architecture, Protocols, and Implementation*. McGraw-Hill, Inc., New York, 1993.

Eines der vielen Bücher zu TCP/IP. Allerdings ist die Darstellung in [Com91] umfassender.

[FTP] ftp.informatik.hu-berlin.de:/pub/linux.

Das ist der Heimat-FTP-Server der Autoren. Hier befindet sich neben den wichtigsten Daten anderer LINUX-FTP-Server auch der in diesem Buch beschriebene PC-Speaker-Treiber (im Verzeichnis `hu-sound/`).

[Gal95] Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Inc., 1995.

Dieses Buch beschreibt die Real-Time-Ergänzungen des POSIX-Standards 1003.1b, besser bekannt als POSIX.4. Man findet hier auch Informationen über asynchrone IO und POSIX IPC.

[GC94] Berny Goodheart and James Cox. *The Magic Garden Explained*. Prentice Hall International, Inc., 1994.

Eine aktuelle Beschreibung der Interna von UNIX System V Release 4.

[Gir90] Gintaras R. Gircys. *Understanding and Using COFF*. O'Reilly & Associates, Inc., 1990.

Eine recht ausführliche Beschreibung des COFF-Objektformats. Hier werden der Aufbau und die Bearbeitung genauer beschrieben. Da das Objektformat aus dem allseits bekannten `a.out` hervorgegangen ist und sich diese beiden nicht wesentlich unterscheiden, kann das Buch auch als Literatur dafür benutzt werden.

[GPL] Gnu public license.

Die GNU Public License bestimmt die Lizenzbedingungen, unter denen der Linux-Kern sowie die meisten bei Linux-Distributionen benutzten Programme verwendet werden dürfen. Der volle Wortlaut findet sich in der Datei COPYING auf der beiligenden CD.

[HH+96] Sebastian Hetze, Dirk Hohndel, u.a. *Linux Anwenderhandbuch*. LunetIX Softfair, 6. erweiterte und aktualisierte Auflage, 1996.

Für den angehenden Linux-Nutzer ein gutes Nachschlagewerk zum Thema Installation und Wartung des Linux-Systems.

[Joh95] Michael K. Johnson. *Linux Kernel Hacker's Guide*. Linux Document Project, draft 0.6 edition, 1995.

Das sollte eigentlich einmal das Standardwerk für den Linux-Kern werden. Die letzte erschienene Version ist vom Januar 1995 und soll in Zukunft eher zu einer Artikelsammlung werden. Wie alle anderen Dokumentationen des Linux-*Document-Projects* ist der Text dieses Buches auf jedem guten Linux-FTP-Server vorhanden.

[Knu98] Donald E. Knuth. *The Art of Computer Programming*, Volume 1. O'Reilly & Associates, Inc., third edition, 1998.

Der erste Band des genialsten Werks, das je über Programmierung geschrieben worden ist.

[LDP] Linux document project.

Bei vielen größeren Softwareprojekten findet sich meist niemand, der die notwendige Dokumentationen verfasst. Bei Linux ist dies glücklicherweise nicht der Fall. Schon früh fand sich eine Gruppe von Entwicklern, die unter dem Namen *Linux Document Project* wichtige Texte verfassen. Dazu gehören ausführliche *Manual Pages*, ein Handbuch für die Installierung von Linux, der *Network Administrator's Guide*, der *Linux Kernel Hacker's Guide* sowie der *Linux Programmer's Guide*.

[Lew91] Donald Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, Inc., 1991.

Wenn Sie schon nicht den POSIX-Standard auf Ihrem Schreibtisch haben (und wer hat das schon), dann sollten Sie sich zumindest dieses Buch ansehen.

[Lio77] John Lions. *Lion's Commentary on UNIX 6th Edition*. 1977.

Einer der Klassiker der Unix Literatur. Lange Zeit nur als Kopie erhältlich, gibt es jetzt einen ‚offiziellen' Nachdruck.

[LMKQ89] S. J. Leffler, M. K. McKusick, M. J. Karels und J. S. Quaterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison-Wesley Publishing, Reading, 1989.

Im Gegensatz zu [Bac86] geht es hier um die Implementation der BSD-Variante von UNIX. Ebenfalls ein Standardwerk zum Thema: „Wie schreibe ich mein eigenes UNIX-System?". Von diesem Buch gibt es auch einen Nachfolger für 4.4BSD [MBKQ96].

[Max99] Scott Maxwell. *Linux Core Kernel Commentary*. Coriolis Open Press, 1999.

In Anlehnung an [Lio77] besteht das Buch aus 2 Teilen. Zuerst wird auf über 400 Seiten Original Linux Sourcecode abgedruckt. Dessen Funktionsweise wird im zweiten Teil auf circa 150 Seiten beschrieben.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, und John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing, Reading, 1996.

Der Nachfolger von [LMKQ89] beschreibt alle Interna der neuesten BSD-Version.

[Mes93] Hans-Peter Messmer. *PC-Hardwarebuch*. Addison-Wesley, Bonn, 1993.

In diesem Buch wird die Standardhardware des PCs gut und verständlich beschrieben. Auf neuere Entwicklungen wird jedoch nicht eingegangen. Die prinzipielle Einführung in die Thematik DMA ist sehr zu empfehlen.

[MM01] Jim Mauro and Richard McDougall. *Solaris Internals*. Prentic Hall/Sun Microsystems Press, 2001.

Das ist das definitive Buch über den Sun Solaris Kernel, einer modernen kommerziellen UNIX-Implementation.

[Pos81] Jon Postel. *RFC 793* — transmission control protocol: Protocol specification, September 1981.

Dieses RFC ist eigentlich die Grundlage für jede Implementation des TCP-Protokolls. Wenn sich alle daran halten, gibt es keine Probleme mit der Kommunikation via TCP.

[PT+91] Rob Pike, Ken Thomson, et al. Plan 9: The early papers, July 1991.

Die sehr interessante Zusammenfassung einiger älterer Arbeiten zum experimentellen Betriebssystem Plan 9. Wenn ihnen die Namen der Autoren bekannt vorkommen, dann liegen Sie richtig. Dieselben Leute haben vor 20 Jahren auch die ersten UNIX-Systeme geschrieben. Hier können Sie nachlesen, wie sie heute zu den Konzepten von UNIX stehen. Diese Reports sind auch im Internet veröffentlicht worden und auf vielen guten FTP-Servern vorhanden.

[Rub98] Allessandro Rubini. *Linux device drivers*. O'Reilly & Associates, Inc., 1998.

Ein sehr guter Leitfaden für die Entwicklung von Gerätetreibern unter Linux.

[Sal94] Peter H. Salus. *A Quarter Century of Unix*. Addison-Wesley Publishing, Reading, 1994.

Kein Buch über die Interna von Unix, sondern über die Geschichte dieses faszinierenden Betriebssystems.

[San93] Michael Santifaller. *TCP/IP und ONC/NFS in Theorie und Praxis*. Addison-Wesley, Bonn, 2., aktualisierte und erweiterte Auflage, 1993.

Einführung in eine Thematik, deren Schwerpunkt bei der Benutzung liegt.

[Sch94] Curt Schimmel. *UNIX Systems for Modern Architectures*. Addison-Wesley Publishing, Reading, 1994.

Nach einer allgemeinen Einführung in den UNIX-Kern geht der Autor detailliert auf die Probleme und Möglichkeiten von Multiprocessing und Caching für Unix-Systeme ein.

[SG94] Avi Silberschatz und Peter Galvin. * *Operating System Concepts*. Addison-Wesley Publishing, Reading, 1994.

Eine weitere Einführung in die Materie.

[SMP] Multiprocessor specification.

Die Intel-Multiprozessor-Spezifikation beschreibt das Zusammenwirken von mehreren Intel-Prozessoren in einem System. Zu erhalten ist sie unter anderem unter `http://developer.intel.com/pro/datashts/24201605.pdf`

[Sta94] Stefan Stapelberg. *UNIX System V.4 für Einsteiger und Fortgeschrittene*. Addison-Wesley, Bonn, 1994.

[Ste92a] W. Richard Stevens. *Advanced Programming in the UNIX(R) Environment*. Addison-Wesley Publishing, Reading, 1992.

Dies ist das ultimative Buch zum Programmieren unter Unix. Stevens beschreibt hier auf über 700 Seiten die gesamte Bandbreite der Systemrufe von BSD 4.3 über System V Release 4 bis zum POSIX-Standard — inklusive der Anwendung der Systemrufe — an *sinnvollen* Beispielen.

[Ste92b] W. Richard Stevens. *Programmieren von UNIX-Netzen*. Coedition Verlage Carl Hanser und Prentice-Hall, München und London, 1992.

Wenn ihnen das Kapitel über Netzwerke im [Ste92a] zu kurz ist, finden Sie hier alles über die Programmierung von Unix-Netzen. Ebenfalls ein sehr empfehlenswertes Buch.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated: The Protocols*, Volume 1. Addison-Wesley Publishing, Reading, 1994.

Dies ist definitiv das Buch, über das man TCP/IP kennen lernen sollte. Es beschreibt die Materie so, wie sie der UNIX-Anwender erlebt. Eine Reihe von frei verfügbaren Tools hilft dabei, das Netz zu erkunden.

[Ste98] W. Richard Stevens. *UNIX network programming — Interprocess Communications*. Prentice Hall International, Inc., Upper Saddle River, NJ 07458, 1998.

Dies ist das ultimative Buch zur Interprozeßkommunikation. Von System V IPC, der neuen IPC-Spezifikation für POSIX, und der Kommunikation zwischen POSIX-Threads ist alles in der für den Autor üblichen ausgezeichneten Qualität erklärt.

[Tan86] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall International, Inc., London, 1986.

Im Gegensatz zu [Tan90] geht es hier nicht um Minix. Das Buch beschreibt grundlegende Prinzipien der Arbeitsweise von klassischen und verteilten Betriebssystemen. Diese werden anschließend an jeweils zwei konkreten Beispielen (MS-DOS, Unix sowie Amoeba und Mach) erläutert. Uns ist allerdings immer noch unklar, weshalb die Beschreibung von MS-DOS in ein Buch mit dem Titel „Moderne Betriebssysteme“ gekommen ist.

[Tan89] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall International, Inc., London, second edition, 1989.

Das Buch ist eine Übersicht über die Arbeitsweise von Netzwerken. Ausgehend vom OSI-Referenzmodell werden die theoretischen Grundlagen der einzelnen Schichten und ihre Umsetzung in der Praxis beschrieben. Ein gutes Grundlagenbuch, wenn auch nicht mehr ganz aktuell.

[Tan90] Andrew S. Tanenbaum. *Betriebssysteme — Entwurf und Realisierung — Teil 1 Lehrbuch*. Coedition Verlage Carl Hanser und Prentice-Hall, Berlin und London, 1990.

Tanenbaum beschreibt hier den Aufbau und die Funktion seines MINIX-Systems. MINIX (*Mi*ni U*nix*) wurde von Tanenbaum für Ausbildungszwecke geschrieben. Es verdeutlicht sehr gut die Konzepte der Implementation von Unix-Systemen, ist aber aufgrund seiner Beschränkungen nicht unbedingt praxistauglich. Minix war das erste Unix-System, dessen Quelltexte man relativ preiswert bekommen konnte. Deswegen war es bei Informatikstudenten ziemlich beliebt. Die Entwicklung von LINUX begann unter MINIX.

[Val96] Uresh Valhalla. *UNIX Internals — The New Frontiers*. Prentic Hall, 1996.

Ein neueres Buch über Konzepte bei der Entwicklung des UNIX-Betriebssystems.

[WE94] Kevin Washburn und Jim Evans. *TCP/IP*. Addison-Wesley, Bonn, 1994.

Dies ist eine weitere umfassende Einführung in die Thematik TCP/IP. Der Schwerpunkt liegt eindeutig bei der Beschreibung der Protokolle und ihrer Anwendung.

Index

Der Index soll eine Hilfe beim Arbeiten am Linux-Kern sein. Er wurde mit dem auf der CD befindlichen Programm `cloneidx` erstellt, und kann somit von jedem für neue Kern-Versionen angepaßt werden.

Bei Kernelfunktionen, Variablen und Strukturen ist in Klammern die Datei angegeben, in der sich die Definition befindet. Dateien sind dabei relativ zur Wurzel der Linux-Quellen (siehe Kapitel 2) angegeben. Ausnahme sind Headerdateien, die sich im Include-Pfad (also meist auch unter `/usr/include`) befinden. Sie sind dann analog der Include-Anweisung des C-Präprozessors in „spitzen Klammern" dargestellt. Das Wort *static* hinter einer Variable oder Funktion weist darauf hin, daß diese Variable bzw. Funktion in den angegebenen Dateien jeweils lokal definiert ist.

Ist bei C-Präprozessor-Makros der Wert eine einfachen Zahl, so ist sie nach einen Gleichheitszeichen (=) angegeben. Auch hier ist zusätzlich eine Datei, in der das Makro definiert wurde, mit angegeben.

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z