

Linux - Ein Einblick in den Kernel

Markus Quaritsch, Bakk. techn. *quam@qwws.net*
Thomas Winkler, Bakk. techn. *tom@qwws.net*

Betreuer: Univ.-Ass. Dipl.-Ing. Dr. techn. Harald Krottmaier *hkrott@iicm.edu*

18. März 2004

Zusammenfassung

GNU/Linux hat sich in den letzten Jahren ohne Zweifel zu einer ernst zu nehmenden Alternative im Bereich der Unix Betriebssysteme entwickelt. Die vorliegende Arbeit soll einen kurzen Einblick in wichtige Interna des Linux Kernel geben. Nach einem einleitenden, kurzen Überblick über die historische Entwicklung des Kernels werden die Themenkreise „Synchronisation“, „Prozessverwaltung“, „Scheduling“, „Memory Management“, „System Calls“ sowie der Netzwerk Stack behandelt. Dabei werden die grundlegenden Konzepte und Datenstrukturen analysiert und beschrieben. Der Fokus liegt dabei auf der aktuellen Version 2.6, wobei für relevante Stellen Vergleiche zur vorhergehenden Kernelversion 2.4 vorgenommen werden und Neuerungen aufgezeigt werden. Als abschließendes Thema befasst sich die Arbeit mit den in Version 2.6 neu eingeführten „Linux Security Modules“.

Inhaltsverzeichnis

1	Einleitung	4
2	Linux - ein Blick zurück	4
2.1	Die Anfänge	4
2.2	Timeline	6
3	Organisation des Sourcecodes	7
4	Synchronisation	8
4.1	Atomic Operations	9
4.2	Spin Locks	9
4.3	Semaphoren	10
4.4	Abschalten von Interrupts	11
5	Prozess Management	12
5.1	Datenstrukturen zur Prozessverwaltung	12
5.2	Prozesszustände	12
5.3	Erzeugen von Prozessen	13
5.4	Threads unter Linux	14
5.4.1	Userlevel Threads	14
5.4.2	Kernel Threads	14
6	Scheduling	14
6.1	Prozessprioritäten	15
6.2	Timeslices	15
6.3	Datenstrukturen für das Scheduling	15
6.4	Der Scheduler	17
6.4.1	Auswahl des nächsten laufenden Prozesses	18
6.4.2	Neuberechnung von Priorität und Timeslice von Prozessen	19
6.5	Context Switch	20
6.6	Kernel Preemption	20
6.7	Performance-Vergleich des Schedulers	21
7	Memory Management	21
7.1	Das (N)UMA Modell	22
7.2	Das Buddy-System	25
7.3	Allokation von physikalischem Speicher	27
7.4	Der Slab-Allokator	28
7.5	Seitentabellen	30
7.6	Verwaltung des virtuellen Prozessspeichers	32

8	System Calls	35
8.1	Vorhandene Systemaufrufe	36
8.2	Realisierung von Systemaufrufen	36
9	Netzwerk Stack	38
9.1	Datenstrukturen: Socketbuffer	38
9.2	Datenübertragungsschicht	41
9.3	Vermittlungsschicht	42
9.3.1	Eingehende Pakete	42
9.3.2	Lokale Zustellung von Paketen	43
9.3.3	Routing	44
9.3.4	Forwarding von Paketen	45
9.3.5	Versenden von Paketen	45
9.3.6	Netfilter Framework	45
9.3.7	IPv6	46
9.4	Transportschicht	46
9.4.1	Übergang von Vermittlungs- zu Transportschicht	47
9.4.2	Verbindungsaufbau - Drei Wege Handshake	47
9.4.3	Empfangen von Daten	48
9.4.4	Versenden von Daten	49
9.4.5	Abbau von Verbindungen	50
9.5	Anwendungsschicht - Sockets	50
9.5.1	Datenstrukturen	51
9.5.2	Verwendung von Sockets	52
9.6	Der Weg durch den TCP-Stack im Überblick	53
10	Sicherheitsmechanismen im Kernel	54
10.1	LSM – Linux Security Modules	54
10.1.1	Architektur	54
10.1.2	Security Fields	55
10.1.3	Hooks	55
10.1.4	Registrierung von eigenen <i>Security Modules</i>	57
10.1.5	Kombination von Modulen – Stacking	57
10.1.6	Performance und Einsatzbereiche	57
10.2	SELinux – Security Enhanced Linux	58
10.2.1	Ziele	58
10.2.2	Architektur	58
10.2.3	Implementierung	60
10.2.4	Anwendung	60

A Spezielle Code-Konstrukte	62
A.1 <code>asmlinkage</code>	62
A.2 <code>likely</code> und <code>unlikely</code>	62

1 Einleitung

GNU/Linux hat sich in den letzten Jahren ohne Zweifel zu einer ernst zu nehmenden Alternative im Bereich der Unix Betriebssysteme entwickelt. Mit der vorliegenden Arbeit soll der Versuch unternommen werden, einen Einblick in die Interna des Kernel zu vermitteln. Schon allein auf Grund des Umfangs der Kernelquellen wird klar, dass es unmöglich ist, sich in diesem Rahmen mit jedem einzelnen Bereich detailliert auseinander zu setzen. Viel mehr werden bestimmte zentrale Teile herausgenommen und in ihren Grundzügen beschrieben. Diese Betrachtungen konzentrieren sich auf das Verständnis der Konzepte und Datenstrukturen, die zum Einsatz kommen. Die Auswahl der Themenbereiche ist dabei ein Kompromiss aus verschiedensten Faktoren und deshalb keineswegs vollständig. Viele wichtige Themen wie Dateisysteme, Gerätetreiber oder Module – um nur einige zu nennen – mussten ausgeklammert werden. Wieder andere Themen werden nur am Rande erwähnt, ohne jedoch in einem eigenen Abschnitt behandelt zu werden.

Sofern nicht anders erwähnt beziehen sich alle Aussagen auf die aktuelle Kernelversion 2.6.

Stellen, die sich speziell mit Eigenschaften des Kernels 2.4 beschäftigen sind, wie dieser Absatz, mit einem eigenen Symbol am rechten Rand gekennzeichnet.

2.4

2 Linux - ein Blick zurück

Linux, oder genauer gesagt GNU/Linux, ist ein Betriebssystem, das in den letzten Jahren stark an Popularität gewonnen hat. Bevor näher auf technische Aspekte der aktuellen Kernel-Serien 2.6 und 2.4 eingegangen wird, soll ein kurzer Überblick über die Geschichte von GNU/Linux gegeben werden.

2.1 Die Anfänge

Um die Gründe, die Linus Torvalds dazu bewogen haben ein eigenes Betriebssystem zu schreiben, besser zu verstehen, ist ein kurzer Ausflug in die Geschichte notwendig.

Bereits 1983 fasste Richard Stallman den Entschluss, ein eigenes Betriebssystem zu schreiben. Eine wichtige Design-Entscheidung für sein Betriebssystem war, dass es ähnlich wie das damals weit verbreitete Unix sein soll. Mehr noch, sein System sollte kompatibel mit den damals üblichen Unices sein. Er wusste zum damaligen Zeitpunkt zwar relativ wenig über Unix, allerdings hatte er den Eindruck, dass dieses ein gutes, sauberes Design aufwies. Eine weitere Design-Entscheidung war, dass sein System portabel sein soll, also nicht auf eine spezielle Hardware oder Hardwarefamilie angewiesen ist.

Im Jänner 1984 begann Stallman dann sein Projekt, dem er den Namen GNU gegeben hatte, in die Tat umzusetzen. Die Abkürzung GNU steht für „GNU's not Unix“. Diese rekursive Definition zeigt den unter Hackern¹ damals üblichen Humor. Stallman wollte zunächst alle für ein Betriebssystem notwendigen Tools wie zum Beispiel Compiler, C-Bibliothek oder Shell schreiben und den Betriebssystem-Kern dann als letztes. Die einzelnen Komponenten seines GNU-Projekts sollten frei verfügbar sein und dies auch bleiben. Um dies sicherzustellen wurde eine eigene Lizenz entworfen, die GNU General Public Licence (GPL).

Neben Stallman schrieb auch Andrew Tanenbaum an einem eigenen Betriebssystem, allerdings aus einer anderen Motivation heraus. Tanenbaum war seit 1980 Professor an der Freien Universität von Amsterdam und hielt Kurse über Betriebssysteme. Um seinen Studenten die Grundlagen von Betriebssystemen auch an praktischen Beispielen näher zu bringen, entschied sich Tanenbaum 1984 ebenfalls dazu, ein portables und Unix-kompatibles Betriebssystem namens Minix zu schreiben. Tanenbaums vordringlich-

¹It. RFC 1392 (<http://www.faqs.org/rfcs/rfc1392.html>): A person who delights in having an intimate understanding of the internal workings of a system, computers and computer networks in particular.

ste Anforderungen an sein Betriebssystem war einerseits ein sauberes, klares Design und, gleich wie bei Stallman, Portabilität. Von der Architektur her entschied sich Tanenbaum für einen Microkernel, da diese einerseits leichter zu entwickeln sind und Tanenbaum andererseits der Meinung war, dass Microkernel zukunftsweisender seien.

Als Tanenbaum Minix 1987 zusammen mit seinem Buch, das er geschrieben hatte um Minix den Studenten näher zu bringen, veröffentlichte, fand dieses großen Anklang. Minix war ein großer Erfolg, weil es ein Unix mit Sourcecode für den PC war und nur geringe Ansprüche an die Hardware stellte. Die Anwender von Minix hatten jedoch recht bald neue Anforderungen an das Betriebssystem und schickten Tanenbaum auch den entsprechenden Sourcecode. Allerdings fanden diese nur sehr spärlich Einzug, weil Minix für den Unterricht konzipiert war und dies auch so bleiben sollte. Als die ersten Personal Computer mit Intel 80386 Prozessoren verfügbar waren, wurde Minix auch auf diese Plattform portiert.

Linus Torvalds lernte bereits relativ früh mit Computern umzugehen. An dem Commodore Vic-20 Microcomputer seines Großvaters, der Statistiker an der Universität von Helsinki war, sammelte Linus seine ersten Programmiererfahrungen. Zunächst in BASIC und später dann in Assembler. Es folgten weitere Erfahrungen mit einer Sinclair QL, welche er ebenfalls in Assembler programmierte. Er entschied sich für diese Architektur einerseits, weil diese bereits multitaskingfähig war und andererseits, weil er die Architektur der damaligen Personal Computer mit dem Intel Z80 Prozessor ablehnte.

Als Linus im Herbst 1990 sein Studium an der Universität von Helsinki begann, war Stallmans GNU-Projekt fast abgeschlossen. Es fehlte nur noch ein wichtiger Teil – der Kernel selbst. Dieser befand sich unter dem Namen GNU Hurd noch in Entwicklung².

An der Universität hatte Linus den ersten Kontakt mit Unix und er begann sich für Betriebssysteme zu interessieren. Anhand von Tanenbaums Buch 'Operating Systems: Design and Implementation' und Minix als Anschauungsbeispiel eignete sich Linus die Grundlagen von Betriebssystemen an.

Kurze Zeit später, im Jänner 1991, kaufte sich Linus seinen ersten Intel basierten Computer. Diesmal entschied er sich für die Intel-Plattform, da diese mit dem 80386 den zu dieser Zeit leistungsfähigsten Prozessor bereitstellte. Es handelte sich um einen PC mit 80386 DX33 CPU (ohne Koprozessor), 4 MB RAM und 40 MB Festplatte. Linus beschäftigte sich sehr intensiv mit den neuen Features des 80386 wie zum Beispiel die Möglichkeit zwischen mehreren Tasks umzuschalten.

Im März begann Linus sich dann auch mit Tanenbaums Minix zu beschäftigen. Er schrieb einen Newsreader, eine Terminal Emulation und auch ein eigenes Filesystem für Minix. Aus diesen ersten Gehversuchen entstand schließlich die Idee eines eigenen Betriebssystems. In den darauf folgenden Sommerferien investierte Linus seine gesamte Freizeit in dieses Projekt, dem er den Arbeitstitel *Linux* gab. Am 25. August 1991 erwähnte Linus sein Projekt in einem Posting in der Newsgroup comp.os.minix :

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki
```

```
Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't be big and
professional like gnu) for 386(486) AT clones. This has been brewing
since april, and is starting to get ready. I'd like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things). I've currently ported bash(1.08) and gcc(1.40),and
things seem to work.This implies that I'll get something practical within
a few months, and I'd like to know what features most people would want.
Any suggestions are welcome, but I won't promise I'll implement them :-)
```

```
Linus (torvalds@kruuna.helsinki.fi)
```

²bis heute, Anfang 2004, ist GNU/Hurd noch nicht in Version 1.0 erschienen

```
PS. Yes - it's free of any minix code, and it has a multi-threaded fs.
It is NOT protable (uses 386 task switching etc), and it probably never
will support anything other than AT-harddisks, as that's
all I have :-(.
[comp.os.minix Linus 25.8.1991]
```

Auf dieses Posting hin kamen eine Reihe von Vorschlägen für weitere Features aber auch Angebote, Linus bei der Entwicklung seines Betriebssystems zu unterstützen.

Die erste für die Öffentlichkeit zugängliche Version von Linux wurde am 17. September 1991 freigegeben. Von Anfang an erhielt Torvalds Feedback von Testern, das in die kurz darauf folgenden Versionen 0.02 und 0.03 Eingang fand. Mit Version 0.10, die Ende November 1991 veröffentlicht wurde, war der Linux Kernel so weit gediehen, dass Torvalds nun in der Lage war, seine Entwicklung ganz unter Linux selbst durchzuführen und Minix nicht mehr als Host Betriebssystem benötigte.

Am 5. Jänner 1992 wurde Linux 0.12 freigegeben, das als wichtige Neuerung, wie in [Torvalds 2001] beschrieben, unter anderem den so genannten „Page-to-Disk“-Mechanismus beinhaltete. Damit wurde es möglich, Teile des Hauptspeichers auf die Festplatte auszulagern. Diese Funktionalität war zum damaligen Zeitpunkt in anderen freien Unix Clones wie Minix nicht verfügbar und veranlasste viele Benutzer zum Umstieg auf Linux. Zu dieser Zeit wuchs die Benutzergemeinde von ein paar Dutzend Leuten auf mehrere hundert Personen an. Linux 0.12 war auch die erste Version die unter der GPL verbreitet wurde.

In einem Posting in der Newsgroup comp.os.minix am 29. Jänner 1992 äußerte Andrew Tanenbaum seine Kritik am aufstrebenden Betriebssystem von Linus Torvalds [comp.os.minix Tanenbaum 29.1.1992]. Die Hauptkritikpunkte dieses Postings waren einerseits die fehlende Portabilität von Linux und andererseits die monolithische Architektur des Kernels. Tanenbaums Meinung nach waren Microkernel, wie er zum Beispiel in Minix verwendet wurde, zukunftsweisender. Weiters war es seiner Meinung nach ein grober Fehler, ein Betriebssystem nur für eine Architektur auszulegen. Linus erste Reaktion auf Tanenbaums Kritik war sehr emotional:

```
...
your job is being a professor and researcher: That's one hell of a
good excuse for some of the brain-damages of minix. I can only hope
(and assume) that Amoeba doesn't suck like minix does.
...
[comp.os.minix Linus 29.1.1992]
```

(Amoeba war Tanenbaums Forschungsprojekt zu dieser Zeit)

Allerdings entschuldigte sich Linus am darauf folgenden Tag für sein emotionales Posting [comp.os.minix Linus 30.1.1992] und es kam zu einer sachlichen Diskussion.

2.2 Timeline

Damit waren die ersten Schritte in der Entstehung und Entwicklung von Linux getan. Viele Programmierer schlossen sich der Entwicklung von Linux an und die Nutzergemeinde wuchs rasant.

Die weiteren Meilensteine in der Entwicklung von Linux werden im Folgenden chronologisch aufgelistet ([Torvalds 2001], [Moody 2001] und [Barger Nov. 2002]).

März 1992 :	Veröffentlichung von Linux 0.95
31.3.1992 :	Gründung der Newgroup comp.os.linux
April 1992 :	X Window läuft auf Linux 0.96
Herbst 1993 :	Erste Integration des TCP/IP Netzwerk Support
1992 :	Linus spaltet Entwicklung von TCP/IP. Schließlich bevorzugt er die Implementierung von Alan Cox gegenüber jener von Fred van Kempen
August 1993 :	Gründung der Debian GNU/Linux Distribution durch Ian Murdock
29.1.1994 :	Debian version 0.91
14.3.1994 :	Linux Kernel Version 1.0 (Archivgröße: 1MB)
April 1994 :	Erste Beta-Version von SuSE Linux
März 1995 :	Linux Kernel Version 1.2 (Archivgröße 2MB)
November 1995 :	Alpha-Port von Linux veröffentlicht
9.5.1996 :	Linus schlägt Pinguin (Tux) als Mascottchen für Linux vor
Juni 1996 :	Namensdiskussion: 'Lignus' vs. 'Gnu/Linux'
9.6.1996 :	Linux 2.0 mit deutlichen Verbesserungen veröffentlicht (Archivgröße 5MB)
17.6.1996 :	Debian 1.1 für i386 verfügbar
Juli 1998 :	Veröffentlichung von Debian 2.0
Ende 1998 :	Linus konzentriert sich auf Integration von SMP (Symmetric Multi-Processing) Unterstützung in den Kernel
25.1.1999 :	Linux Kernel 2.2.0 veröffentlicht (buggy) (Archivgröße 10MB)
Mai 1999 :	Entwicklung am 2.3er Kernel beginnt
4.1.2001 :	Linux Kernel 2.4 veröffentlicht
25.11.2001 :	Entwicklung am 2.5er Kernel beginnt
18.12.2003 :	Linux Kernel 2.6.0 veröffentlicht

3 Organisation des Sourcecodes

Nicht nur die Verbreitung von Linux nahm mit jeder neuen Version zu, sondern auch der Umfang der Kernelquellen. Die Archivgröße der aktuellen Kernelversion 2.6.0 beträgt annähernd 32 MiB – entpackt benötigen die Kernelquellen 208 MiB. Im Vergleich dazu benötigte das Archiv der ersten Version nicht einmal 1 MiB.

Zur Verdeutlichung des Umfangs der Kernelquellen hier einige statistische Daten:

Insgesamt bestehen die Quellen aus 15007 Dateien in 917 Verzeichnissen. Diese teilen sich auf in

```

6229  Header Dateien (.h)
6194  C Quelldateien (.c)
674   Assembler-Dateien (.S)
663   Makefiles (Makefile)

```

Ohne Kommentare umfassen die Header- und C-Dateien fast 4 Millionen Zeilen an Code.

Damit die Entwickler hier noch den Überblick behalten, sind die Quelldateien entsprechend der darin behandelten Aufgaben in eine Verzeichnishierarchie aufgeteilt. Folgende Verzeichnisse enthalten die zentralen Komponenten des Kernels:

kernel: Dieses Verzeichnis enthält die zentralen Bestandteile des Kernels. Es besteht aus lediglich 47 Dateien und etwa 20000 Zeilen Code. An diesem Teil des Kernel werden nur sehr selten Änderungen vorgenommen.

mm: Hier befindet sich die High-Level Speicherverwaltung. Ein Auszug der hier enthaltenen Funktionalität wird in Abschnitt 7 beschrieben. Dieses Verzeichnis ist mit 37 Dateien und etwa 15000 Zeilen Code annähernd gleich groß wie der zentrale Kern.

init: Enthält den zur Initialisierung des Kernels notwendigen Code.

ipc: Enthält die Implementierung der *System V* Interprozesskommunikations-Mechanismen

sound: Hier sind Treiber für verschiedene Soundkarten zu finden. Diese unterscheiden sich zwar von den anderen Treibern nicht wesentlich, wurden aber trotzdem in ein eigenes Verzeichnis ausgelagert.

fs: In diesem Teil der Kernelquellen sind die Implementierungen der von Linux unterstützten Dateisysteme abgelegt. Da neben den unter Linux üblichen Dateisystemen wie *ext2* und *ext3* auch noch viele weitere Dateisysteme unterstützt werden, gehört dieser Teil der Quellen ebenfalls zu den umfangreicheren.

net: Enthält die Implementierung der unterstützten Netzwerkprotokolle. Dazu zählen neben den bekannteren *IPv4* und *IPv6* Protokollen auch eine Reihe von weniger bekannten Protokollen – wie zum Beispiel *Appletalk*. Dieser Teil umfasst beinahe 10 MiB in 615 Dateien.

drivers: Dieser Teil der Verzeichnishierarchie macht mit 87 MiB den größten Teil der Kernelquellen aus. Hier befinden sich – wie der Name des Verzeichnisses bereits vermuten lässt – Treiber für die vom Kernel unterstützte Hardware. In diesem Verzeichnis erfolgt eine weitere Unterteilung nach Funktionskategorie bzw. Bustyp.

include: Dateien dieses Unterverzeichnisses enthalten alle Headerdateien für öffentlich exportierte Funktionen und Datenstrukturen. Auch dieses Verzeichnis ist in weitere Verzeichnisse unterteilt. unter *include/linux* befinden sich alle architekturunabhängigen Header während in *include/arch-<arch>* Dateien spezifisch für eine Architektur enthalten sind.

crypto: Mit Version 2.6 fanden auch Algorithmen für cryptographische Funktionen Einzug in den Kernel. Die Implementierungen der unterstützten Ciphers sind in diesem Verzeichnis abgelegt.

Documentation: Dieses Verzeichnis enthält Beschreibungen zu verschiedenen Aspekten des Kernels. Dieser Teil der Quellen enthält annähernd 3000 Dateien und umfasst 7 MiB. Allerdings sind nicht alle Dokumente davon auf dem neuesten Stand.

arch: In diesem Verzeichnis ist für jede unterstützte Architektur ein weiteres Unterverzeichnis enthalten, in dem architekturspezifische Quelldateien enthalten sind. Highlevel C-Funktionen verwenden die hier definierten Funktionen um die notwendigen Operationen auf die Zielhardware abzubilden.

scripts: Enthält Hilfsprogramme, die zur Übersetzung des Kernels benötigt werden.

4 Synchronisation

Ein inhärentes Problem beim gleichzeitigen (bzw. quasi gleichzeitigen) Zugriff auf gemeinsame Ressourcen ist die Integrität der gemeinsamen Daten. Dieses Problem ist einerseits im Userspace bei parallel laufenden Prozessen anzutreffen und andererseits auch im Kernel, wenn es sich um ein SMP³ System handelt. In diesem Fall kann es nämlich passieren, dass sich mehr als nur eine CPU zu einem bestimmten Zeitpunkt im Kernelmodus befindet, womit die Möglichkeit besteht, dass mehrere CPUs zur selben Zeit globale Datenstrukturen des Kernels manipulieren. Ein weiterer Aspekt der mit Kernel 2.6 hinzugekommen ist, ist das durch den *Kernel Preemption* Mechanismus die Möglichkeit geschaffen wurde auch Kernel Code vorzeitig zu unterbrechen, womit auch auf Uniprozessor Maschinen die Notwendigkeit besteht, kritische Teile des Kernels entsprechend durch Synchronisations-Mechanismen zu schützen.

Die Gründe, warum Concurrency⁴ Situationen im Kernel auftreten können, sind vielfältig:

³SMP: Symmetric Multi Processing - mehrere CPUs in einem Rechner

⁴concurrent - auf deutsch: gleichzeitig, neben-läufig

schlafende Tasks Ein Task kann im Kernel Kontext schlafen gehen (z.B. wegen Synchronisation mit dem Userspace) und ein anderer Task kann zur Ausführung kommen.

Interrupts Sie können jederzeit auftreten und unterbrechen den gerade laufenden Code. Dabei muss beachtet werden, ob es zwischen Interrupt-Kontext und dem Kontext eines Prozesses gemeinsam benutzte Daten gibt. Weiters besteht grundsätzlich die Möglichkeit, dass die selben Datenstrukturen von zwei verschiedenen Interrupt-Handlern verwendet werden.

Kernel Preemption Durch die Einführung des *Kernel Preemption* Mechanismus besteht seit Kernel 2.6 die Möglichkeit, dass Kernel Code unterbrochen wird und ein anderer Task ausgeführt wird.

Symmetric Multi Processing Das System besitzt 2 oder mehr CPUs, die zum selben Zeitpunkt den selben Code ausführen können.

explizites Rescheduling Kernel Code kann jederzeit über Aufruf der *schedule()* Funktion die Kontrolle abgeben, was einen anderen Task zur Ausführung bringt.

Die nachfolgend beschriebenen Mechanismen bieten die Möglichkeit, diese Problematik in den Griff zu bekommen. Es sei angemerkt, dass nur einige der Möglichkeiten, die der Kernel zur Synchronisation zur Verfügung stellt, herausgegriffen wurden.

4.1 Atomic Operations

Eine Operation wird genau dann als atomar bezeichnet, wenn sie garantiert immer ohne Unterbrechung ausgeführt wird. So zerfällt beispielsweise das einfache Inkrementieren eines Integer Wertes in das Auslesen, das Inkrementieren und das Zurückschreiben des Resultats. Wird der Thread dabei von einem anderen Thread unterbrochen, der auf der selben Variablen operiert, so ist das Endergebnis potentiell falsch.

Um solche Fälle zu vermeiden, stellt der Kernel den Datentyp `atomic_t` zur Verfügung der gleichzeitige Operationen auf einer Variablen verhindert. Technisch gesehen handelt es sich bei `atomic_t` um einen in eine `struct` verpackten `int`. Zum Zugriff dürfen jedoch ausschließlich die in `include/asm/atomic.h` definierten Inline Funktionen verwendet werden. Diese plattformabhängigen Funktionen stellen sicher, dass die Zugriffe auch tatsächlich atomar ausgeführt werden.

Verwendet wird `atomic_t` in erster Linie dann, wenn andere Locking Mechanismen unnötig aufwändig wären, wie zum Beispiel im Fall eines einfachen Zählers. Weiters steht auch eine Familie von atomaren Bitoperationen zur Verfügung, die in `include/asm/bitops.h` definiert sind.

4.2 Spin Locks

In vielen Fällen reichen die oben beschriebenen atomaren Operationen nicht aus. Um ganze Codeabschnitte vor gleichzeitigem Zugriff abzusichern, werden unter anderem *Spin Locks* eingesetzt. Ein solcher *Spin Lock* kann höchstens von einem Thread gehalten werden, während alle anderen Threads, die in den selben kritischen Bereich eintreten wollen, mittels *busy waiting* warten müssen. Auf Grund dieser Eigenschaft des *Spin Locks* wird klar, dass er nur dann zum Einsatz kommen sollte, wenn er voraussichtlich nur für kurze Zeit gehalten werden soll (kürzer als die 2 Context Switches die man benötigen würde, wenn man den blockierenden Thread durch einen lauffähigen ersetzt und später wieder zurückschaltet).

Der Einsatzzweck von *Spin Locks* ist es, kritische Code Bereiche auf SMP Maschinen davor zu schützen, dass zwei Prozessoren zur selben Zeit diesen Code ausführen. Auf Uniprozessor Maschinen kann der kritische Bereich des Kernel Codes nicht von mehr als einem Prozessor betreten werden, womit *Spin Locks* hier nicht mehr notwendig sind.

Mit der in Kernel 2.6 eingeführten *Kernel Preemption* gilt dies allerdings nicht mehr. Hier ist es möglich, dass der Kernel unterbrochen wird. Geschieht das innerhalb einer kritischen Region, so kann es passieren, dass dieselbe kritische Region vom neuen laufenden Task wieder betreten wird, womit man vor dem

selben Problem wie auf SMP Maschinen steht: kritischer Kernelcode wird (quasi) parallel abgearbeitet. Um dies zu verhindern, wird die *Kernel Preemption* innerhalb von Bereichen die mit *Spin Locks* gesichert sind deaktiviert.

Darüber hinaus gibt es Datenstrukturen, von denen jeweils eine eigene Instanz pro CPU existiert. Da normalerweise nur diese eine CPU auf diese Daten zugreift, ist hier auch kein besonderer Schutz in Form von *Spin Locks* notwendig. Durch die Einführung der *Kernel Preemption* ist es jedoch möglich, dass diese eine CPU pseudo-parallel auf solche Datenstrukturen zugreift und es so zu Inkonsistenzen kommen kann. Um das zu verhindern muss beim Zugriff auf solche Daten die *Kernel Preemption* ausgeschaltet und danach wieder aktiviert werden.

Das Interface zur Verwendung von Spinlocks ist in `include/linux/spinlock.h` zu finden. Die Verwendung von Spin Locks ist, wie Listing 1 zeigt, recht einfach.

```
1  spinlock_t lock = SPINLOCK_UNLOCKED;
2
3  spin_lock(&lock);
4
5  /* kritische Code Region */
6
7  spin_unlock(&lock);
```

Listing 1: Verwendung von Spin Locks

Im Gegensatz zu Semaphoren können Spinlocks auch in Interrupt-Handlern verwendet werden. In einem solchen Fall müssen zusätzlich lokale Interrupts (für die CPU auf der der Code läuft) deaktiviert werden. Andernfalls wäre es möglich, dass ein weiterer Interrupt auftritt, der zugehörige Interrupt Handler den Kernel unterbricht und versucht denselben Lock zu erwerben. In diesem Fall würde dieser Interrupt-Handler busy waiting machen während der erste Interrupt Handler keine Möglichkeit hat den Spin Lock freizugeben, was in einem Deadlock resultiert. Um das durch Abschalten lokaler Interrupts zu vermeiden, kann der Code aus Listing 2 verwendet werden.

```
1  spinlock_t lock = SPINLOCK_UNLOCKED;
2  unsigned long flags;
3
4  spin_lock_irqsave(&lock, flags); /* aktuellen Interrupt zustand sichern,
5                                     Interrupts lokal deaktivieren */
6
7  /* kritische Code Region */
8
9  spin_unlock_irqrestore(&lock, flags); /* gesicherten Interrupt Zustand
10                                     wiederherstellen */
```

Listing 2: Verwendung von Spin Locks in Interrupt Handlern

Neben der gerade besprochenen Form von *Spin Locks* gib es noch eine weitere Ausprägung, nämlich die so genannten *Reader Writer Spin Locks*. Oft tritt der Fall auf, dass viele Prozesse Daten lesen wollen aber nur wenige Prozesse Daten schreiben wollen. In diesem Fall macht es keinen Sinn, alle Leser durch die Verwendung eines einzigen Locks zu blockieren. Mehrere Leser können gleichzeitig, rein lesend, auf die Datenstruktur zugreifen. Schreiber dürfen jedoch nur jeweils einzeln zugreifen und weiters dürfen zu diesem Zeitpunkt keine Leser auf die Datenstruktur zugreifen.

4.3 Semaphoren

Im Gegensatz zu *Spin Locks* legen *Semaphoren* den Thread, der nicht in eine kritische Region eintreten kann, schlafen. Ein busy waiting wie im Fall der *Spin Locks* entfällt hier. Entsprechend werden *Semaphore* für längere Wartezeiten verwendet. Darüber hinaus besitzen sie die Eigenschaft, *Kernel Preemption* nicht

zu unterbinden. Wie bei *Semaphoren* üblich, bietet auch der Linux Kernel die beiden Ausprägungen *Binary Semaphore* und *Counting Semaphore*.

Analog zu den *Spin Locks* gibt es auch bei den *Semaphoren* wieder die Variante der *Reader Writer Semaphore*.

Listing 3 zeigt exemplarisch die Verwendung einer Reader/Writer Semaphore. Bei dem Code handelt es sich um den Systemcall `sys_gethostname`.

```
1  DECLARERWSEM(uts_sem); /* Semaphore anlegen */
2
3  /* ... */
4
5  asmlinkage long sys_gethostname(char __user *name, int len)
6  {
7      int i, errno;
8
9      if (len < 0)
10         return -EINVAL;
11     down_read(&uts_sem); /* Eintreten in Critical Section */
12     i = 1 + strlen(system_utsname.nodename);
13     if (i > len)
14         i = len;
15     errno = 0;
16     if (copy_to_user(name, system_utsname.nodename, i))
17         errno = -EFAULT;
18     up_read(&uts_sem); /* Verlassen der Critical Section */
19     return errno;
20 }
```

Listing 3: Verwendung von Reader/Writer Semaphoren (*kernel/sys.c*), deutsche Kommentare nachträglich im Rahmen der Seminararbeit eingefügt

4.4 Abschalten von Interrupts

Mit dem Abschalten von Interrupts will man verhindern, dass ein auftretender Interrupt und der in Folge ausgeführte Interrupt Handler dazu führen, dass der gerade laufende Code vorzeitig unterbrochen wird. Mit dem Ausschalten der Interrupts geht immer auch ein Ausschalten der *Kernel Preemption* einher. Wenn es sich um ein SMP System handelt, ist allerdings immer noch nicht sichergestellt, dass gemeinsame Daten nicht von einem anderen Prozessor des Systems gleichzeitig manipuliert werden. Erst durch die Verwendung der weiter oben beschriebenen *Spin Locks* kann auch das ausgeschlossen werden. Die Möglichkeit Interrupts global für alle CPUs abzuschalten wurde mit Kernel 2.6 fallen gelassen.

```
1  unsigned long flags;
2
3  local_irq_save(flags);
4
5  /* Interrupts sind deaktiviert */
6
7  local_irq_restore(flags);
8
9  /* alter Interrupt Zustand ist wiederhergestellt */
```

Listing 4: lokales Ausschalten der Interrupts

Der Code in Listing 4 zeigt das Aus- und Einschalten der Interrupts. Genauer gesagt werden Interrupts für die lokale CPU ausgeschaltet und danach der Originalzustand wiederhergestellt. Neben dem Ausschalten aller lokalen Interrupts besteht natürlich auch die Möglichkeit gezielt einzelne Interrupts auszumaskieren.

5 Prozess Management

Die Verwaltung von Prozessen⁵ stellt eine der zentralen Aufgaben eines jeden Betriebssystems dar. Ein Prozess besteht aus einem ausführbaren Teil, der so genannten *text section*, und einer Reihe von zum Prozess gehörender Ressourcen, die in der *data section* verwaltet werden. Zu diesen Ressourcen zählen beispielsweise globale Variablen, offene Dateien, ausstehende Signale, ein logischer (virtueller) Adressraum sowie ein oder mehrere *threads of execution*.

5.1 Datenstrukturen zur Prozessverwaltung

Die zentrale Struktur zur Repräsentation und Verwaltung von Prozessen unter Linux stellt die Struktur `struct task_struct` dar. Jeder im System vorhandene Prozess besitzt eine solche `task_struct`⁶, auch *Process Descriptor* genannt, welche alle für den Prozess relevanten Informationen enthält.

Weiters gibt es für jeden Prozess eine Instanz der Struktur `struct thread_info`, die Daten des Prozesses enthält, die aus architekturenspezifischem Assemblercode zugreifbar sein müssen. Diese Datenstruktur ist am Ende des Kernel Stack untergebracht und hält einen Pointer auf die zuvor angesprochene `task_struct` des Prozesses (welche ihrerseits mit Hilfe des Slab-Allocator im Kernel-Speicher angelegt worden ist).

5.2 Prozesszustände

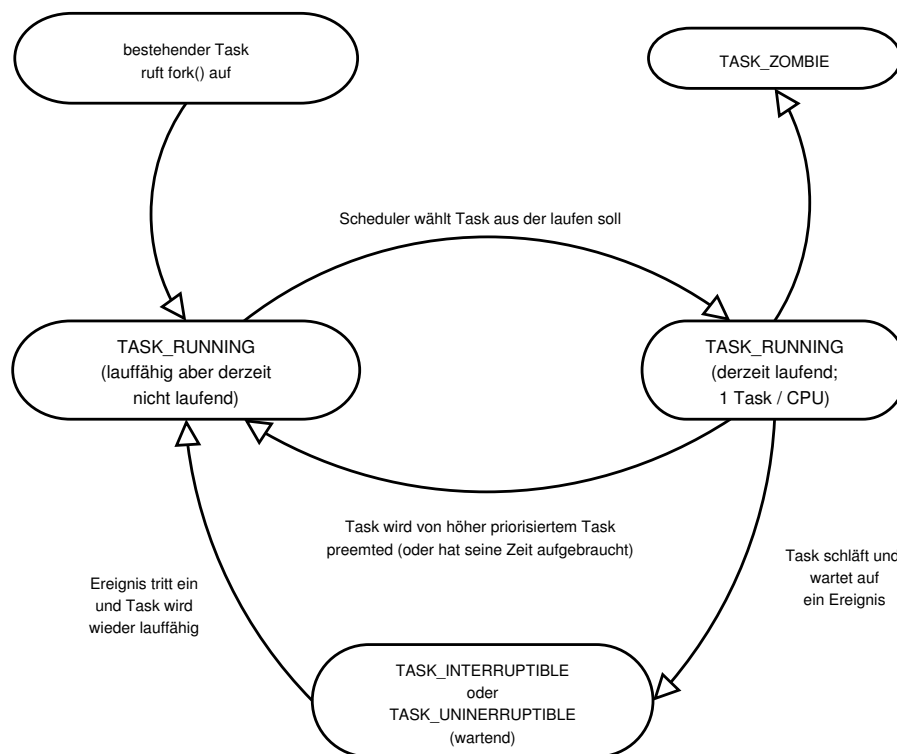


Abbildung 1: Die Task-Zustände im Überblick

Die Prozesse eines Systems können sich in einer Reihe unterschiedlicher Zustände befinden. Der Zustand eines konkreten Prozesses ist in dessen `task_struct` im Feld `state` vermerkt. Folgende Zustände sind dabei möglich:

⁵im Linux Kernel häufig auch als Tasks bezeichnet

⁶Datei: `include/linux/sched.c`

TASK_RUNNING Prozesse in diesem Zustand sind ausführbar, d.h. sie warten nicht auf I/O Operationen oder ähnliches. Auf einer SMP Maschine können maximal n Prozesse ($n =$ Anzahl der tatsächlich vorhandenen CPUs) gleichzeitig abgearbeitet werden. Alle anderen Prozesse befinden sich in einer *runqueue* und warten darauf, vom Scheduler für die Ausführung ausgewählt zu werden.

TASK_INTERRUPTIBLE Diese Prozesse sind derzeit nicht ausführbar weil sie beispielsweise auf das Eintreten eines bestimmten Ereignisses warten. Beim Eintreten eines Signals können diese Prozesse auch vorzeitig in den **TASK_RUNNING** Zustand übergehen.

TASK_UNINTERRUPTIBLE Auch diese Prozesse warten auf ein Ereignis, können während dieser Wartezeit jedoch nicht durch ein anderes Signal vorzeitig aufgeweckt werden. Folglich reagieren Prozesse dieser Art auch nicht auf ein **SIGTERM** und scheinen in der Prozessliste (*ps*) mit dem Statusflag **D** auf.

TASK_ZOMBIE Prozesse dieser Art sind beendet. Der *Process Descriptor* verbleibt aber noch im System, um dem Elternprozess die Möglichkeit zu geben, den Status des Prozesses abzufragen.

TASK_STOPPED Prozesse in diesem Zustand wurden – beispielsweise von einem Debugger – absichtlich angehalten.

5.3 Erzeugen von Prozessen

Bei der Erzeugung von neuen Prozessen verfolgt Linux, so wie für Unix Systeme üblich, einen etwas ungewöhnlichen Weg. Der Vorgang ist in zwei Teile aufgeteilt:

- **fork** – erzeugt einen Kindprozess der, bis auf wenige Ausnahmen wie beispielsweise die **PID**, eine identische Kopie des Elternprozesses ist
- **exec** – lädt ein neues Executable in der Adressraum und führt es aus

Auf den ersten Blick scheint es nicht besonders effizient zu sein eine vollständige Kopie des Prozesses anzulegen. Diese Vorgehensweise wäre einerseits zeit und ressourcenaufwändig und andererseits ist es, sofern der **exec** Aufruf direkt auf **fork** folgt, völlig unsinnig eine Kopie anzulegen wenn diese ohnehin gleich wieder überschrieben wird. Aus diesem Grund kommt der sogenannte *Copy on Write* (COW) Mechanismus zum Einsatz. Dabei wird nicht der gesamte Adressraum kopiert sondern lediglich die *Pageable*. Eltern- und Kindprozess verwenden somit die selben physikalischen Speicherseiten. Daraus ergibt sich unmittelbar, dass beide fortan nur mehr lesend auf den Speicher zugreifen dürfen – schreibender Zugriff würde unweigerlich zu Problemen führen. In den *Pageables* der beiden Prozesse werden entsprechende Vermerke vorgenommen.

Sobald einer der Prozesse schreibend auf seinen Speicher zugreift, wird vom Prozessor eine *Page Fault Exception* ausgelöst. Der *Page Fault Handler* des Kernel hat nun die Möglichkeit zu prüfen, ob die Seite eigentlich beschreibbar sein sollte. Ist das der Fall, so wird eine Kopie der Speicherseite angelegt, die vom Prozess beschrieben werden darf. Durch diese Vorgehensweise kann das Kopieren von Speicherseiten hinausgezögert bzw. sogar gänzlich unnötig gemacht werden – in vielen Fällen kommt es vor, dass Prozesse auf Speicherseiten nie schreibend zugreifen.

Neben dem weiter oben angesprochenen **fork** Aufruf bietet Linux eine weitere Möglichkeit um Prozesse zu kopieren, nämlich die **vfork** Funktion. Sie verhält sich gleich wie **fork** mit dem Unterschied, dass keine Kopie der *Pageable* des Elternprozesses angelegt wird. Der Kindprozess darf nicht schreibend auf den gemeinsamen Adressraum zugreifen und der Elternprozess blockiert so lange, bis der Kindprozess fertig abgearbeitet ist oder ein **exec** Systemaufruf erfolgt. Historisch gesehen ist **vfork** zu einer Zeit entstanden, als **fork** noch nicht über den oben beschriebenen COW Mechanismus verfügt hat. Mittels **vfork** konnte so ein unnötiges Kopieren des Adressraums umgangen werden. Heute hat **vfork** nur mehr den Vorteil, dass hier, im Gegensatz zu **fork**, auch die *Pageable* des Prozesses nicht kopiert wird. Die Kombination aus **vfork** und **exec** ist somit geringfügig schneller wenn es darum geht einen neuen Prozess zu erzeugen.

Es ist bereits daran gedacht, den COW Mechanismus von `fork` im kommenden Entwickler-Kernel 2.7 auch auf die Pagetable auszudehnen womit der `vfork` Aufruf obsolet werden würde.

5.4 Threads unter Linux

5.4.1 Userlevel Threads

Das Konzept der Threads ermöglicht es modernen Betriebssystemen, mehrere Ausführungsfäden in ein und dem selben Programm zuzulassen. Dieser Mechanismus bietet einige offensichtliche Vorteile: Die Threads können Ressourcen wie etwa Speicher oder offene Dateien miteinander teilen. Weiters ermöglichen Threads parallele Abläufe in ein und demselben Programm.

Der Linux Kernel kennt keine Threads im eigentlichen Sinne. Viel mehr werden Threads als jeweils eigene Prozesse realisiert, die mit anderen Prozessen bestimmte Daten teilen. Aus Sicht des Kernels handelt es sich aber ansonsten um gewöhnliche Prozesse, die folglich auch als solche behandelt werden.

Das Erzeugen eines Threads mit Hilfe des `clone` System Calls unterscheidet sich von der Erstellung eines neuen Prozesses mittels `fork` lediglich durch die Angabe von Parametern, die die gemeinsamen Ressourcen festlegen. Ein entsprechender Aufruf könnte wie folgt aussehen:

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

Der aufrufende Prozess (Thread) und der neu erzeugte Prozess (Thread) teilen sich in diesem Fall den Adressraum, Dateisystem Ressourcen, Datei Descriptoren und Signal-Handler. Weitere mögliche CLONE Flags sind in der Datei `include/linux/sched.h` spezifiziert.

5.4.2 Kernel Threads

Für bestimmte Aufgaben ist es nützlich, sie in einem eigenen Thread abzuarbeiten. Dazu gehört beispielsweise das Schreiben von „dirty pages“ auf die Platte wie es der `pdflush` Daemon tut. Kernel Threads werden nur im Kernelmode ausgeführt und besitzen keinen eigenen virtuellen Adressraum. Ansonsten unterliegen sie, wie normale Threads auch, dem Scheduler und sind „preemptable“.

Der Aufruf zum Erzeugen eines Kernel Threads sieht wie folgt aus:

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
```

`fn` – Pointer auf die Funktion die der Thread ausführen soll

`arg` – Argumente für Funktion `fn`

`flags` – Clone Flags wie z.B. `CLONE_KERNEL`

Typischerweise werden Kernel Threads als Endlosschleife implementiert. Stehen keine Aufgaben an, so legt sich der Thread schlafen.

6 Scheduling

Die Kernaufgabe des Schedulers ist es, die zur Verfügung stehende Rechenzeit möglichst fair zwischen den einzelnen Prozessen – unter Beachtung deren Priorität – aufzuteilen. Die zentrale Funktionen für das Scheduling im Linux Kernel befindet sich in der Datei `kernel/sched.c`. Seit Kernel 2.6 besitzt Linux einen so genannten O(1) Scheduler, d.h. die Laufzeit des Schedulers ist nicht mehr abhängig von der Anzahl der Prozesse im System, sondern konstant.

6.1 Prozessprioritäten

Unter Linux besitzt jeder Prozess eine so genannte statische Priorität. Mit dem *nice* Kommando kann diese vom Userspace aus beeinflusst werden. Die möglichen Werte reichen von -20 bis +19 wobei niedrigere Werte eine höhere Priorität des Prozesses bedeuten.

Darüber hinaus verfügt Linux über eigene Prioritäten für Echtzeitprozesse. Diese und die Prioritäten für normale Prozesse werden, wie in Abbildung 2 gezeigt, auf einen Wertebereich von 0 bis 139 abgebildet. Die Werte von 0 bis 99 sind für Echtzeitprozesse reserviert während die Werte von 100 bis 139 für normale Prozesse verwendet werden und den *nice* Werte von -20 bis +19 entsprechen.

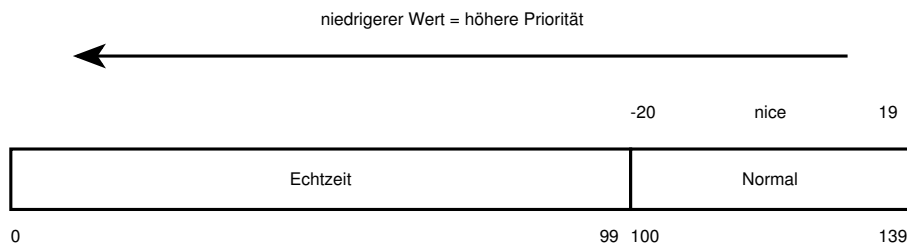


Abbildung 2: Aufteilung der statischen Prioritäten

In der Version 2.4 wird ebenso zwischen Echtzeitprozessen und normalen Prozessen unterschieden. Auch die Prioritäten von -20 bis +19 für normale Prozesse sind unverändert. Für Echtzeitprozesse stehen jedoch nur Prioritäten zwischen 1 und 99 zur Verfügung. Es erfolgt allerdings keine Abbildung der Prioritäten dieser beiden Prozessklassen auf einen gemeinsamen Wertebereich. Vielmehr werden diese in zwei Feldern abgespeichert, wobei eine Echtzeitpriorität von 0 bedeutet, dass es sich um einen normalen Prozess handelt.

2.4

6.2 Timeslices

Bei der Timeslice handelt es sich um einen Zahlenwert (`time_slice` in `task_struct`), der angibt, wie lange ein Task laufen kann, bevor ihm der Prozessor entzogen wird (der Task „preempted“ wird). Die mögliche Timeslice unter Linux reicht von 10ms bis 200ms. Der Scheduler berechnet die Timeslice eines Prozesses dynamisch in Abhängigkeit von dessen Priorität. Ein Task muss seine Timeslice nicht auf einmal aufbrauchen sondern kann, wenn er z.B. immer wieder auf I/O warten muss, seine Timeslice auch in mehreren Etappen verbrauchen.

6.3 Datenstrukturen für das Scheduling

Die `task_struct` eines jeden Prozesses beinhaltet einige Informationen, die für das Scheduling wesentlich sind.

```

1  struct task_struct {
2      /* ... */
3
4      int prio, static_prio;
5      struct list_head run_list;
6      prio_array_t *array;
7
8      unsigned long sleep_avg;
9      long interactive_credit;
10     unsigned long long timestamp;

```

```

11     int activated;
12
13     unsigned long policy;
14     cpumask_t cpus_allowed;
15     unsigned int time_slice, first_time_slice;
16
17     /* ... */
18 };

```

Listing 5: Scheduling Informationen in `task_struct` (*include/linux/sched.h*)

- `prio` enthält die dynamische und `static_prio` die statische Priorität des Prozesses
- `run_list` und `array` dienen zur Verwaltung des Prozesses im `prio_array` bzw. dessen `queue` (siehe weiter unten)
- `sleep_avg` sagt aus, wie oft und lange ein Prozess geschlafen hat
- `policy` – Scheduling Policy für den Prozess (SCHEM_NORMAL = „normales Priority Scheduling“, SCHEM_RR und SCHEM_FIFO werden für Echtzeitprozesse verwendet)
- `time_slice` verbleibende Zeit, die der Prozess noch auf der CPU laufen darf
- `first_time_slice` – wird unmittelbar nach dem `fork` auf 1 gesetzt und ist ansonsten 0. Der Sinn ist, dass das verbleibende Zeitquantum an den Eltern-Prozess zurückgegeben wird sofern der neu „geforkte“ innerhalb der ersten Timeslice mit seiner Arbeit fertig ist (kurze Threads).

Für jede CPU des Systems verwaltet der Kernel eine sogenannte `runqueue`. Jede solche Liste besitzt unter anderem 2 Arrays, genannt `active` und `expired` mit jeweils `MAX_PRIO` Elementen.

```

1     struct runqueue {
2         spinlock_t lock;
3         unsigned long nr_running, nr_switches, expired_timestamp, nr_uninterruptible;
4         task_t *curr, *idle;
5         struct mm_struct *prev_mm;
6         prio_array_t *active, *expired, arrays[2];
7         int prev_cpu_load[NR_CPUS];
8         #ifdef CONFIG_NUMA
9             atomic_t *node_nr_running;
10            int prev_node_load[MAX_NUMNODES];
11        #endif
12        task_t *migration_thread;
13        struct list_head migration_queue;
14        atomic_t nr_iowait;
15    };

```

Listing 6: `runqueue` Datenstruktur (*kernel/sched.c*)

Im `active` Array befinden sich all jene Prozesse, die noch eine Timeslice größer Null besitzen, während sich im `expired` Array jene Prozesse befinden die ihre Rechenzeit bereits aufgebraucht haben. In den beiden Arrays werden die Prozesse nach ihrer Priorität sortiert gespeichert.

- `nr_running` enthält die Anzahl der lauffähigen Prozesse in der `runqueue`.
- `expired_timestamp` gibt an, wann zum ersten Mal ein Prozess von `active` in `expired` gewechselt ist
- `curr` zeigt auf den aktuell laufenden Prozess
- `idle` zeigt auf den Leerlaufprozess, der immer dann zur Ausführung kommt, wenn keine anderen lauffähigen Prozesse vorhanden sind

Die beiden Arrays `active` und `expired` sind vom Typ `struct prio_array`. Diese Struktur enthält einerseits pro möglicher Prozess-Priorität eine Liste und andererseits eine Bitmap, über die einfach festgestellt werden kann, welche der Listen Elemente enthält und welche nicht. In den einzelnen Listen werden dabei jeweils all jene Prozesse gespeichert, die dieselbe Priorität aufweisen. Abbildung 3 stellt diese Zusammenhänge für das `active` Array der `runqueue` dar.

```

1  struct prio_array {
2      int nr_active;
3      unsigned long bitmap[BITMAP_SIZE];
4      struct list_head queue[MAX_PRIO];
5  };

```

Listing 7: `prio_array` Datenstruktur (*kernel/sched.c*)

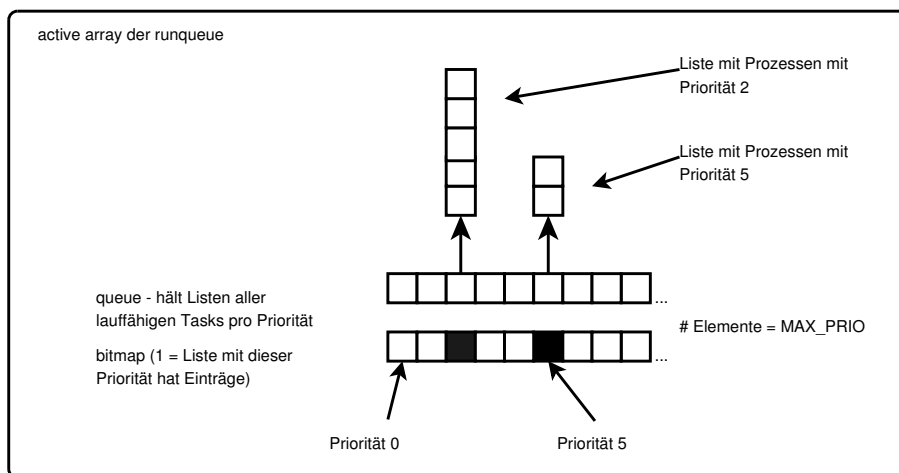


Abbildung 3: `active` Array der `runqueue` im Detail

Die hier beschriebenen Datenstrukturen wurden in Version 2.6 eingeführt. In Version 2.4 werden die lauffähigen Prozesse nicht abhängig von ihrer Priorität in verschiedenen Listen gehalten, sondern alle lauffähigen Prozesse werden auf einer gemeinsamen Liste, die unter der globalen Variablen `runqueue_head` zugänglich ist, gespeichert. 2.4

Die in der Prozessbeschreibung enthaltenen Felder sind ähnlich zu den zuvor beschriebenen. Lediglich für die Speicherung der Prozesspriorität ist ein zusätzliches Feld notwendig, da diese für Echtzeitprozesse und normale Prozesse getrennt gespeichert wird.

6.4 Der Scheduler

Aus Sicht des Systems können grob zwei Arten von Prozessen unterschieden werden: Jene Prozesse, die stark I/O-lastig sind (z.B. ein Texteditor, der häufig auf Benutzereingaben warten muss) und jene die eher CPU-lastig sind (z.B. ein Compiler). Diese beiden Prozessstypen müssen durch den Scheduler unterschiedlich behandelt werden, damit das System aus Sicht des Benutzers ein akzeptables Verhalten zeigt. So ist es bei vielen rechenintensiven Anwendungen in der Regel egal, ob sie ihre Aufgabe geringfügig schneller oder langsamer erledigen. Bei interaktiven Prozessen hingegen steht ein schnelle Reaktion auf Ereignisse im Vordergrund. Diesen Anforderungen muss der Scheduler Rechnung tragen und die *timeslices* bzw. die dynamische Priorität der Prozesse entsprechend anpassen.

6.4.1 Auswahl des nächsten laufenden Prozesses

Um den lauffähigen Task mit der höchsten Priorität zu finden genügt es, das `bitmap` des `active` Arrays der `runqueue` nach dem ersten gesetzten Bit zu durchsuchen. Da die Anzahl der Prioritäten konstant ist, ist dieser Aufwand stets derselbe. Aus der zur gerade ermittelten Priorität gehörenden `queue` in `prio_array` wird nun der erste Task ausgewählt. Innerhalb einer Priorität geht der Scheduler nach dem Round Robin Verfahren vor.

Diese Auswahl des neuen Tasks erfolgt in der Funktion `schedule`⁷. In Listing 8 ist von Zeile 27 bis 29 die Auswahl jenes Prozesses zu sehen, der als nächster laufen soll.

```

1  asmlinkage void schedule(void)
2  {
3      task_t *prev, *next;
4      runqueue_t *rq;
5      prio_array_t *array;
6      int idx;
7
8      /* ... */
9
10     preempt_disable();
11     prev = current;
12     rq = this_rq();
13
14     /* ... */
15
16     array = rq->active;
17     if (unlikely(!array->nr_active)) {
18         /*
19          * Switch the active and expired arrays.
20          */
21         rq->active = rq->expired;
22         rq->expired = array;
23         array = rq->active;
24         rq->expired_timestamp = 0;
25     }
26
27     idx = sched_find_first_bit(array->bitmap);
28     queue = array->queue + idx;
29     next = list_entry(queue->next, task_t, run_list);
30
31     /* ... */
32 }

```

Listing 8: Auszug aus `schedule` Funktion (*kernel/sched.c*)

In Version 2.4 des Kernels werden die lauffähigen Prozesse nicht wie zuvor beschrieben abhängig ihrer Priorität in einer Liste gehalten. Vielmehr werden alle lauffähigen Prozesse in einer gemeinsamen Liste – der `runqueue` – gehalten. Auch Prozesse, die ihre Zeitscheibe bereits aufgebraucht haben werden in dieser Liste gehalten, sofern diese lauffähig sind. Um den nächsten auszuführenden Prozess zu finden, wird über diese Liste iteriert und für jeden Prozess ein Gewicht berechnet. Dieses Gewicht gibt an wie gut es ist, diesen Prozess als nächstes laufen zu lassen. Dabei wird zwischen Echtzeitprozessen und normalen Prozessen unterschieden. Erstere erhalten ein Gewicht von $1000 + \text{Echtzeit-Priorität}$, zweitere ein Gewicht zwischen 2 und 77 sofern dieser seine Ausführungszeit *nicht* aufgebraucht hat und 0 anderenfalls.

Die Funktion zum Auffinden des nächsten lauffähigen Prozesses hat daher ein Laufzeitverhalten der Größenordnung $O(n)$.

⁷Datei: `kernel/sched.c`

6.4.2 Neuberechnung von Priorität und Timeslice von Prozessen

In früheren Linux Versionen wurde die `time_slice` der Tasks immer dann neu berechnet, wenn alle Tasks keine Ausführungszeit mehr zur Verfügung hatten. Dies wurde typischerweise in einer Schleife über alle Tasks erledigt und benötigt somit $O(n)$ Zeit. Ein weiteres Problem, das damit einher geht ist, dass die Taskliste bzw. die einzelnen Tasks für diese Neuberechnung gelockt werden müssen und diese Locks somit massiv zu bestimmten Zeitpunkten auftreten.

Durch die Verwendung des `active` und `expired` Arrays kann der neue $O(1)$ Scheduler diese Problematik entschärfen. Wenn die `time_slice` eines Tasks den Wert 0 erreicht wird er vom `active` in das `expired` Array verschoben. Die Neuberechnung der `time_slice` erfolgt noch bevor er in das `expired` Array eingefügt wird. Sind schlussendlich die Timeslices aller Tasks aus dem `active` Array verbraucht (d.h. es sind keine Prozesse mehr im `active` Array), so werden `active` und `expired` Array (beides Pointer) einfach ausgetauscht und so die oben beschriebenen Probleme vermieden.

Dieser Tausch der beiden Arrays, wenn keine Prozesse mehr im `active` Array vorhanden sind, ist in Listing 8 in den Zeilen 16 bis 25 zu sehen.

Neben der schon angesprochenen statischen Priorität (Variable `static_prio` in `task_struct`) kennt der Linux Kernel auch eine dynamische (Variable `prio` in `task_struct`) Priorität. Diese errechnet sich als Funktion aus der statischen Priorität und der Interaktivität eines Prozesses.

Die Funktion `effective_prio` berechnet einen Bonus (im Bereich von -5 bis +5) basierend auf `sleep_avg` des Tasks. Dieser Bonus wird zum `nice` Wert aus `static_prio` (statische Priorität des Prozesses) addiert und das Resultat in `prio` (dynamische Priorität) gespeichert. Für das Scheduling des Prozesses wird in Folge also `prio` herangezogen.

Der Wert in `sleep_avg` ist ein Maß für die Interaktivität eines Tasks und errechnet sich wie folgt: Wenn ein Task aufwacht und ausführbar wird, wird die Dauer die er geschlafen hat zur `sleep_avg` addiert (bis zur Obergrenze `MAX_SLEEP_AVG`). Mit jedem Timer-Tick während der nun folgenden Ausführung des Prozesses wird `sleep_avg` dekrementiert (bis 0). Stark I/O-lastige (interaktive) Prozesse besitzen also eine höhere `sleep_avg`, während sie für CPU-lastige Prozesse niedriger ist.

Die Berechnung der `timeslice` baut in weiterer Folge unmittelbar auf der dynamischen Priorität auf. Die Berechnung erfolgt in der Funktion `task_timeslice()`, wobei sich eine höhere Priorität in einer größeren `time_slice` niederschlägt. Die berechnete `time_slice` bewegt sich zwischen `MIN_TIMESLICE` (10ms) und `MAX_TIMESLICE` (200ms). Beim `fork` eines bestehenden Tasks wird die `timeslice` des Elternprozesses zur Hälfte an den neuen Prozess abgetreten. Damit wird verhindert, dass ein Prozess durch unzählige Forks übermäßig viel CPU Zeit erlangen kann.

	Zeitquantum	Interaktivität	nice Wert
Beginn	Hälfte des Elternprozesses	-	gleich wie Elternprozess
Minimum	10ms	niedrig	hoch
Default	100ms	mittel	null
Maximum	200ms	hoch	niedrig

Interaktive Prozesse können nach dem Aufbrauchen ihrer Timeslice statt in das `expired`-Array wieder in das `active`-Array eingefügt werden. Das findet aber nur dann statt, wenn der fragliche Prozess hoch interaktiv ist und im `expired` Array keine Tasks sind, die bereits hungern. Ein Wiedereinfügen des interaktiven Tasks in das `active` Array würde nämlich zur Folge haben, dass diese Tasks noch länger warten müssen bis sie wieder laufen können. Listing 9 zeigt den Ablauf wenn die `timeslice` des aktuellen Prozesses auf 0 dekrementiert wird.

```

1  if (!--p->time_slice) {
2      dequeue_task(p, rq->active);
3      set_tsk_need_resched(p);
4      p->prio = effective_prio(p);          /* neue Prioritaet aus statischer und
```

```

5                                     dynamischer Prioritaet berechnen */
6     p->time_slice = task_timeslice(p); /* neue timeslice berechnen */
7     p->first_time_slice = 0;
8
9     if (!rq->expired_timestamp)
10        rq->expired_timestamp = jiffies;
11    if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
12        enqueue_task(p, rq->expired);
13    } else
14        enqueue_task(p, rq->active); /* hoch interakt. Tasks wieder
15                                     ins active Array */
16 }

```

Listing 9: Ausschnitt aus `scheduler_tick` (*(kernel/sched.c)*), deutsche Kommentare nachträglich im Rahmen der Seminararbeit eingefügt

In Version 2.4 des Kernels wird die Ausführungszeit der Prozesse immer dann neu berechnet, wenn alle lauffähigen Prozesse ihre Zeitscheibe aufgebraucht haben. In diesem Fall haben alle Prozesse der Runqueue ein Gewicht von 0. 2.4

Die Neuberechnung erfolgt für *alle* (sowohl lauffähige als auch blockierte) Prozesse im System. Die neue Ausführungszeit ergibt sich aus der Hälfte der noch übrigen Ausführungszeit plus einem Wert abhängig von der Priorität des Prozesses. Dadurch erhalten I/O-lastige Prozesse eine höhere dynamische Priorität, da diese öfter blockieren und somit ihre Zeitscheibe nicht vollständig aufbrauchen.

6.5 Context Switch

Das Umschalten zwischen zwei Prozessen wird auch *Context Switch* bezeichnet. Dabei muss der aktuelle Zustand der Maschine gesichert und der für den neuen Prozess benötigte Zustand wiederhergestellt werden. Zu diesem Zustand gehören der Stack des Prozesses sowie die Inhalte der Prozessor-Register. Weiters kommt es zu einem Umschalten des virtuellen Memory Mappings vom vorigen auf den aktuellen Prozess durch die Funktion `switch_mm`.

Pro Prozess gibt es in der Struktur `thread_info` ein Flag namens `need_resched`. Dieses zeigt an, dass ein neuer Prozess laufen soll. Das Flag wird einerseits von der Funktion `scheduler_tick` gesetzt, wenn die Timeslice des aktuellen Prozesses aufgebraucht ist und andererseits von der Funktion `try_to_wake_up`, wenn ein Task aufgeweckt wird, dessen Priorität höher ist als die des gerade laufenden Tasks.

Bei der Rückkehr in den Userspace aus dem Kernspace (Systemcall) oder aus einem Interrupt Handler wird das `need_resched` Flag geprüft. Ist es gesetzt, so wird die Funktion `schedule()` aufgerufen und ein neuer Task ausgewählt. Dieser Mechanismus, der einem Prozess vorzeitig die CPU entzieht, ist auch unter dem Namen **User Preemption** bekannt.

6.6 Kernel Preemption

Mit Kernel 2.6 hat ein neues Features, Kernel Preemption, Einzug gehalten. Bisher war es nicht möglich, einem Task der gerade Kernel Code ausgeführt hat vorzeitig die CPU zu entziehen. Kernel Code lief so lange bis er vollständig abgearbeitet war – also bis zur Rückkehr in den Userspace oder bis er explizit blockiert hat. Das Scheduling im Kernel war also kooperativ und nicht preemptiv.

Seit Kernel 2.6 ist es nun möglich, einen Task an beliebiger Stelle (auch im Kernel) zu unterbrechen – vorausgesetzt er befindet sich in einem „sicheren“ Zustand. Darunter versteht man, dass vom Prozess zu diesem Zeitpunkt kein Lock gehalten werden darf.

Bei der Rückkehr aus einer Interrupt Handler in den Kernspace wird geprüft, ob `need_resched` gesetzt ist. Wenn das der Fall ist und `preempt_count` den Wert null hat, wird der Scheduler aufgerufen.

Dasselbe passiert wenn `preempt_count` des aktuellen Prozesses auf null dekrementiert wird, d.h. alle Locks freigegeben wurden. Schon vor Kernel 2.6 gab es die Möglichkeit, durch explizites Aufrufen von `schedule()` die Kontrolle über den Prozessor explizit abzugeben. In diesem Fall ist es die Aufgabe des Entwicklers sicherzustellen, dass sich der Code in einem „sicheren“ Zustand befindet. Da auch beim Blockieren des Kernels `schedule()` aufgerufen wird, kommt es auch hier zu einer Preemption des Kernels.

6.7 Performance-Vergleich des Schedulers

Die Open Source Developer Labs (OSDL) haben einen Vergleich des Schedulers in Version 2.4 mit jener in Version 2.6 durchgeführt ([developer.osdl.org]). Als Testprogramm wurde das von Paul (Rusty) Russel erstellte Programm *hackbench*⁸ verwendet. Dieses Benchmark-Programm startet eine Reihe von Client- und Server-Prozessen, die Nachrichten untereinander austauschen. Jeder Client-Prozess sendet 100 Nachrichten über einen Socket an den zugehörigen Server-Prozess. Gemessen wurde die Ausführungszeit des Benchmarks abhängig von der Anzahl der Prozess-Paare. Abbildung 4 zeigt eine graphische Auswertung der Ergebnisse.

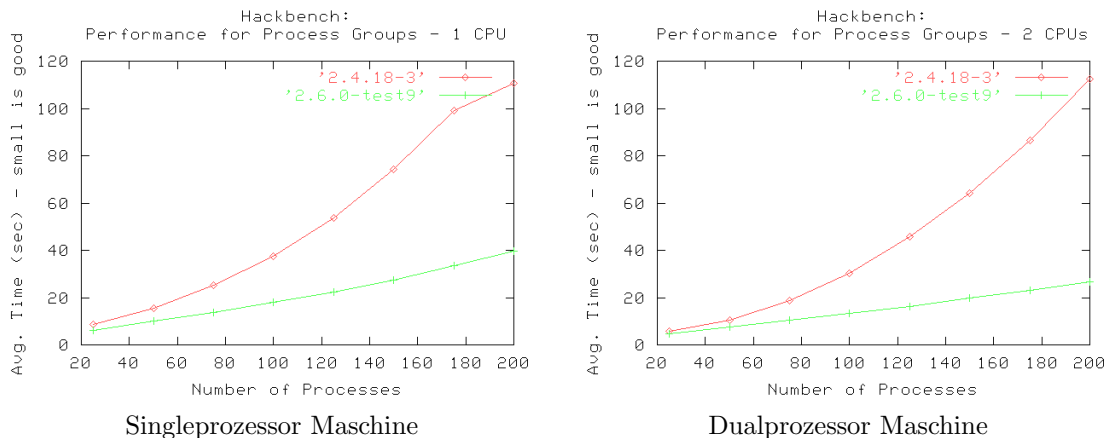


Abbildung 4: Performance-Vergleich des Schedulers (Quelle: [developer.osdl.org])

Die Gegenüberstellung der Ergebnisse (Abbildung 4) zeigt, dass die Ausführungszeit dieses Benchmarks unter Kernel 2.4 exponentiell mit der Anzahl der Prozesse im System ansteigt. Dieses Verhalten resultiert vor allem daher, dass in Version 2.4 für die Auswahl des nächsten auszuführenden Prozesses über alle lauffähigen Prozesse im System iteriert wird.

Die unter Kernel 2.6 erzielten Ergebnisse zeigen ein wesentlich lineareres Verhalten in Bezug auf die Ausführungszeit, da die Auswahl des nächsten Prozesses deutlich effizienter implementiert wurde.

Weiters ist zu erkennen, dass die Ausführungszeit unter Kernel 2.6 deutlich unter jener mit Kernel 2.4 liegt, was darauf schließen lässt, dass der Scheduler in Version 2.6 die ausführbaren Prozesse wesentlich effektiver verwaltet.

7 Memory Management

Die Verwaltung des Speichers ist eine der wichtigsten und auch komplexesten Teile des Kernels. Viele der anfallenden Aufgaben sind nur in enger Zusammenarbeit mit dem Prozessor zu bewältigen und daher stark architekturabhängig. Durch Einführung einer zusätzlichen Abstraktionsebene, des *virtuellen Speichermodells*, ist es möglich, die Speicherverwaltung architekturunabhängig zu programmieren. Dieser allgemeingültige Code bildet die notwendigen Operationen über architekturabhängige Funktionen auf die jeweilige Zielplattform ab.

⁸<http://developer.osdl.org/craiger/hackbench/src/hackbench.c> (Jänner 2004)

Die Hauptaufgaben des Kernels bei der Speicherverwaltung sind einerseits die Verwaltung des physikalischen Speichers und andererseits die Verwaltung des virtuellen Speichers für die einzelnen Prozesse. In den folgenden Abschnitten werden die im Linux Kernel verwendeten Modelle und Datenstrukturen näher vorgestellt.

7.1 Das (N)UMA Modell

Ein wesentliches Unterscheidungsmerkmal von Architekturen ist die Klassifizierung in UMA (Uniform Memory Access) und NUMA (Non Uniform Memory Access).

Uniform Memory Access: Bei diesem Speichermodell ist der verfügbare Speicher zusammenhängend organisiert. Jeder Prozessor im System (im Fall von SMP⁹-Systemen) kann auf alle Speicherbereiche gleich schnell zugreifen. Einzelprozessor Maschinen fallen in diese Kategorie.

Non Uniform Memory Access: Hier handelt es sich immer um Mehrprozessormaschinen. Jedem Prozessor steht dabei lokaler RAM-Speicher zur Verfügung, auf den dieser besonders schnell zugreifen kann. Auf den lokalen Speicher einer anderen CPU kann nur über einen Bus zwischen den Prozessoren zugegriffen werden, wodurch nicht-lokale Zugriffe im Vergleich zu lokalen Zugriffen deutlich langsamer sind

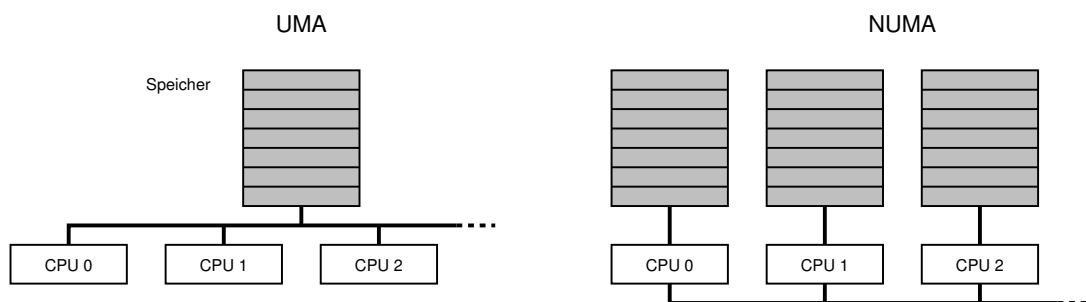


Abbildung 5: Gegenüberstellung UMA- und NUMA Systeme

Abbildung 5 veranschaulicht die Unterschiede der beiden Speichermodelle graphisch.

Der Linux Kernel kann sowohl auf UMA als auch auf NUMA Systemen eingesetzt werden. Um bei der Speicherverwaltung nicht zwischen diesen beiden Systemen unterscheiden zu müssen, werden für Maschinen mit uniformen und nicht uniformen Speicherzugriff die gleichen Datenstrukturen und Algorithmen verwendet.

Ausgehend vom allgemeineren Speichermodell, einem NUMA System, werden die notwendigen Datenstrukturen definiert. Der physikalische Speicher wird in einzelne Nodes aufgeteilt und jedem Prozessor wird eine Node zugeordnet. Jede dieser Nodes wird durch eine Instanz der Struktur `pg_data_t` repräsentiert. Alle Nodes des Systems werden vom Kernel in einer einfach verketteten Liste gehalten.

Zusätzlich wird jede Node in verschiedene Speicherbereiche aufgeteilt. Diese Aufteilung entspricht ebenfalls den verschiedenen Bereichen des physikalischen Speichers. Auf bestimmten Architekturen¹⁰ können nur die ersten 16 MiB des physikalischen Speichers für DMA¹¹-Operationen verwendet werden. Weiters kennt der Kernel noch einen Highmem-Bereich. In diese Kategorie fallen Speicherbereiche, die nicht direkt adressiert werden können (auf IA-32 Systemen können zum Beispiel nur die ersten 896 MiB direkt angesprochen werden).

Derzeit sind drei verschiedene Zonen im Kernel definiert. Diese sind der DMA-Bereich, der Highmem-Bereich und der normale Speicherbereich. Eine Node kann in höchstens drei Zonen aufgeteilt werden,

⁹SMP: symmetric multi processing

¹⁰zum Beispi IA-32 Systeme mit ISA-Geräten

¹¹DMA: Direct Memory Access

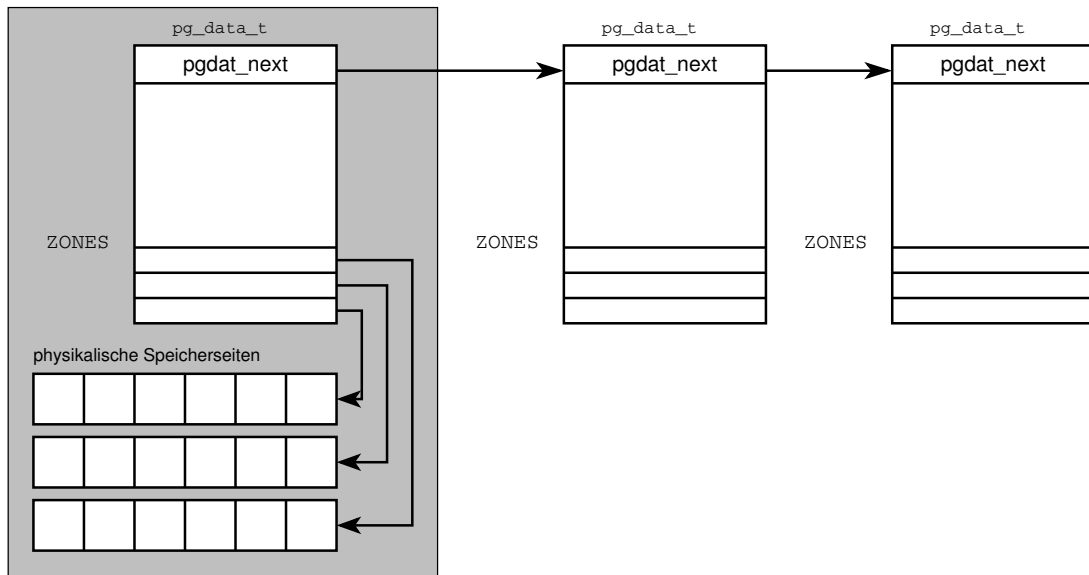


Abbildung 6: Speicherabbildung in NUMA-Systemen

allerdings ist es auch möglich, dass weniger Zonen vorhanden sind. Abbildung 6 stellt dieses Modell graphisch dar.

Diese Abbildung des physikalischen Speichers auf die internen Datenstrukturen kann für UMA-Systeme genauso verwendet werden. Allerdings gibt es dabei nur eine einzige Node (in der Abbildung grau hinterlegt).

Die für die Repräsentation verwendete Datenstruktur `pg_data_t` ist wie folgt definiert:

```

1 typedef struct pglst_data {
2     struct zone node_zones[MAX_NR_ZONES];
3     struct zonelist node_zonelists[MAX_NR_ZONES];
4     int nr_zones;
5     struct page *node_mem_map;
6     unsigned long *valid_addr_bitmap;
7     struct bootmem_data *bdata;
8     unsigned long node_start_pfn;
9     unsigned long node_present_pages;
10    unsigned long node_spanned_pages;
11    int node_id;
12    struct pglst_data *pgdat_next;
13    wait_queue_head_t kswapd_wait;
14 } pg_data_t;

```

Listing 10: Repräsentation einer Node mittels `pg_data_t` (*include/linux/mmzone.h*)

- `node_zones` speichert die Datenstrukturen für die in der Node enthaltenen Zonen (`MAX_NR_ZONES = 3`)
- `node_zonelists` stellt eine Fallback-Liste für die Speicherallokation dar. Falls in der angeforderten Zone kein Platz mehr vorhanden ist, wird in dieser Liste nach alternativen Zonen gesucht. Die Einträge sind entsprechend ihrer 'Kosten' aus Sicht der Node gereiht.
- `nr_zones` speichert die Anzahl der tatsächlich vorhandenen Zonen.
- `node_mem_map` ist ein Zeiger auf ein Array aus `page`-Instanzen¹². Das Array beinhaltet die Seiten aller in der Node enthaltenen Zonen.

¹²diese dienen zur Beschreibung der physikalischen Speicherseiten.

- `valid_addr_bitmap` ist ein Array aus `long`-Elementen. Die einzelnen Bits von `valid_addr_bitmap` zeigen an, ob die physikalische Speicherseite tatsächlich benutzt werden kann. Falls im Speicher „kleine“ Bereiche vorhanden sind, die nicht benutzt werden dürfen (bzw. können) ist dieses Vorgehen einfacher und effektiver als eine Aufteilung in weitere NUMA-Nodes.
- `bdata` enthält einen Speicherbereich für die Initialisierung der Speicherverwaltung während des Bootvorganges
- `node_start_pfn` speichert die logische Startkennzahl der ersten physikalischen Speicherseite dieser Node.
- `node_present_pages` speichert die Anzahl der verfügbaren Speicherseiten.
- `node_spanned_pages` speichert die Anzahl der physikalischen Speicherseiten in dieser Node einschließlich nicht benutzbarer Speicherbereiche.
- `node_id` globale Kennzahl der Node
- `pg_data_next` ist ein Zeiger auf die nächste NUMA-Node. Alle Nodes des Systems werden in einer einfach verketteten Liste gehalten. Das Ende wird wie üblich über einen `NULL`-Zeiger kenntlich gemacht.
- `kswapd_wait` wird für den Swap-Mechanismus verwendet.

Für die Beschreibung der Zonen wird die Datenstruktur `zone` verwendet. Die in dieser Struktur gespeicherten Daten lassen sich in folgende logische Gruppen zusammenfassen:

Allgemeine Daten: Dazu zählen die für diese Datenstruktur verwendeten Locks sowie Grenzwerte für die Anzahl der freien Seiten in dieser Zone. Diese Grenzwerte werden für den Swapping-Mechanismus verwendet.

Zugriffsstatistik: Der Kernel katalogisiert die Speicherseiten gemäß ihrer Aktivität. Dies wird dazu verwendet, bei Speichermangel jene Speicherseiten auszulagern, die nur selten benötigt werden.

Buddy-System: Die für das Buddy-System benötigten Datenstrukturen werden für jede Zone getrennt verwaltet. Sie werden in Abschnitt 7.2 vorgestellt.

Warteschlangen: Für Prozesse, die auf eine freie Speicherseite warten, sind mehrere Warteschlangen vorhanden.

Hot'n-Cold-Page Listen: Der Kernel behält auch den Überblick, welche Speicherseiten sich gerade in einem CPU-Cache befinden (*hot page*).

Selten benutzte Felder: Hier befindet sich ein Feld um der Zone eine textuelle Bezeichnung zuzuweisen. Weiters wird die Anzahl der in dieser Zone vorhandenen Speicherseiten gespeichert.

Wie bereits erwähnt, gibt es für jede Zone Fallback-Listen für den Fall, das in der gewünschten Zone nicht genügend Speicher frei ist. Für jede Zone in einer Node steht eine Fallback-Liste zur Verfügung. Da diese Liste jede Zone aus allen vorhandenen Nodes berücksichtigen muss, sind dazu `MAX_NUMNODES * MAX_NR_ZONES` Einträge (plus ein `NULL`-Zeiger zur Kennzeichnung des Listenendes) notwendig. Die Definition dieser Liste lautet wie folgt:

```
1 struct zonelist {
2     struct zone *zones[MAX_NUMNODES * MAX_NR_ZONES + 1]; // NULL delimited
3 };
```

Listing 11: Fallback-Liste (*include/linux/mmzone.h*)

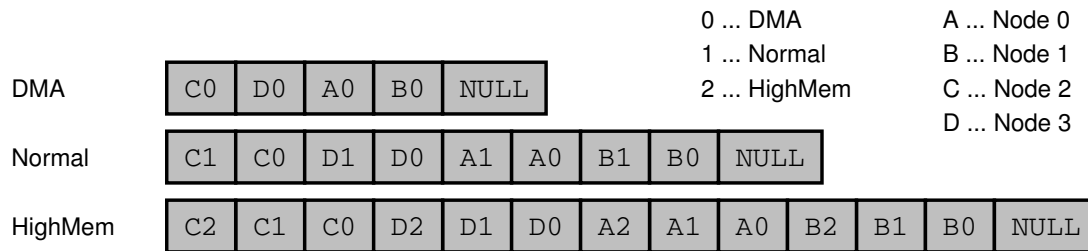


Abbildung 7: Fallback-Liste für eine Node

In Abbildung 7 ist der Inhalt der Fallback-Listen für alle drei Zonen der Node 2 aus einem System mit 4 Nodes exemplarisch dargestellt.

Wird zum Beispiel HighMem-Speicher angefordert, so wird zuerst versucht diesen zu allozieren (C2). Falls kein HighMem-Speicher mehr zur Verfügung steht, wird versucht, Speicher aus dem normalen Bereich auf der aktuellen Node zu allozieren. Schlägt auch dies aus Platzmangel fehl, so wird versucht, den Speicher im DMA-Bereich zu reservieren.

Steht auf der aktuellen Node nicht genügend Speicherplatz zur Verfügung, so wird als nächstes versucht, diesen im HighMem-Bereich der nächsten Node zu reservieren (D2) usw.

Beim DMA-Bereich ist es ähnlich. Allerdings muss hier DMA-Speicher alloziert werden. Daher wird, wenn auf der aktuellen Node nicht ausreichend DMA-Speicher verfügbar ist, sofort auf die nächste Node ausgewichen.

7.2 Das Buddy-System

Der Kernel verwendet für die Verwaltung der freien Speicherseiten in einer Zone einen effizienten und robusten Algorithmus, den *Buddy-System* Algorithmus. Dieser ermöglicht neben der schnellen Speicher-allokation auch ein effiziente Ausnutzung des physikalischen Speichers.

Dabei werden zusammenhängende Speicherseiten in zwölf Gruppen von je 1, 2, 4, 8, 16, ... bis 2048 Seiten zusammengefasst. Dies entspricht zusammenhängenden Speicherbereichen von 4 kiB bis 8 MiB. Zusätzlich muss die physikalische Adresse der ersten Seite ein Vielfaches der Gruppengröße sein. Also z.B. für einen 16 Seiten großen Block muss die Startadresse ein Vielfaches von 16×2^{12} ($2^{12} = 4$ kiB, entspricht der Größe einer Speicherseite) sein.

In der Version 2.4 sind nur zehn Gruppen vorhanden, womit zusammenhängende Speicherbereiche von 4 kiB bis 2 MiB möglich sind.

2.4

Angenommen der Kernel fordert vom Buddy-System 64 zusammenhängende Speicherseiten an, so wird zunächst versucht, diese aus der entsprechenden Gruppe zu liefern. Stehen allerdings keine zusammenhängende Blöcke von 64 Seiten zur Verfügung, so werden der Reihe nach die größeren Gruppen durchsucht. Wenn in den größeren Gruppen ein freier Speicherbereich gefunden wurde, so wird dieser in kleinere Blöcke aufgeteilt. Nehmen wir an, es wurde ein freier Speicherbereich mit 256 Blöcken gefunden, so wird dieser aus dem Buddy-System entfernt und in einen Block mit 128 Seiten sowie zwei Blöcke mit je 64 Seiten aufgeteilt. Ein Block mit 64 Seiten wird zurückgeliefert während, die beiden anderen Blöcke wieder ins Buddy-System eingefügt werden.

Wird ein Speicherbereich an das Buddy-System zurückgegeben, so wird rekursiv versucht, zusammenhängende Blöcke (Buddies) wieder zu verbinden und in die nächstgrößere Gruppe zu schieben.

Vom Buddy-System können natürlich nur ganzzahlige Zweierpotenzen von Seitenzahlen angefordert werden. Dieser Umstand zeigt sich auch in den zum Allokieren von Speicher vorgesehenen Funktionen. Als Parameter für die Größe des zu reservierenden Speichers muss die Ordnung (also die Gruppengröße) und nicht die Anzahl der Bytes oder Speicherseiten angegeben werden.

Das Buddy-System ist nur auf die Verwaltung der Seiten einer Zone konzentriert. Deshalb sind die Datenstrukturen in der zuvor vorgestellten Struktur zur Verwaltung der Zonen enthalten. Die Verbindung zwischen den einzelnen Zonen wird ausschließlich über die Fallback-Liste hergestellt.

```
1 struct zone {
2     /* ... */
3
4     struct free_area free_area[MAX_ORDER];
5
6     /* ... */
7 };
8
9 struct free_area {
10    struct list_head free_list;
11    unsigned long *map;
12 };
```

Listing 12: Datenstrukturen für das Buddy-System (*include/linux/mm.h*)

Für jede Gruppe von Speicherblöcken wird in `zone` eine Instanz von `free_area` angelegt (`MAX_ORDER = 11`). Die Indizes in diesem Array stehen für die Größe der in diesem Arrayelement verwalteten Blöcke. Im nullten Array-Element werden einzelne Speicherseiten verwaltet ($2^0 = 1$), im ersten Blöcke von 2 Speicherseiten ($2^1 = 2$) usw.

Die Hilfsstruktur `free_area` enthält eine Liste der freien Speicherblöcke und ein als Bitmap verwendetes Array von `longs`. In der Liste wird jeweils nur die erste Seite eines Speicherblocks gehalten. Die restlichen Seiten sind ohnehin physikalisch zusammenhängend und müssen daher nicht in einer eigenen Datenstruktur verkettet werden. Beim Aufspalten eines Blocks kann die Position der ersten Speicherseite des neuen Blocks berechnet werden.

Das Bitmap `map` speichert den Zustand der Buddy-Paare. Dabei wird für je ein Buddy-Paar ein Bit verwendet. Es ist ausreichend, für jeweils 2 Speicherseiten nur ein Bit zu verwenden, da nur jene Blöcke verschmolzen werden können, deren Startadresse ein Vielfaches der nächstgrößeren Gruppengröße ist. Für die Interpretation dieses Bits wird folgende Semantik verwendet:

Bit gesetzt: Beide Teile des Buddys sind vergeben oder frei

Bit ungesetzt: Einer der beiden Buddys ist frei

Wird ein Speicherbereich vom Buddy-System vergeben, so wird zuerst die Speicherseite aus der entsprechenden Liste entfernt. Dazu sind keine teuren Suchoperationen notwendig, sondern es kann der erste freie Block aus der Liste entnommen werden. Anschließend muss noch das zugehörige Bit in der Map aktualisiert werden. Dazu ist lediglich eine Negation notwendig. Die Bitposition in der Map kann aus der Speicherseite berechnet werden.

Wird ein Speicherblock wieder an das Buddy-System zurückgegeben, so wird dieser wieder in die Liste der freien Speicherblöcke eingefügt und die Bitmap gleich wie zuvor beschrieben aktualisiert. Zusätzlich muss noch geprüft werden, ob das Bit null ist, denn dann können die beiden Buddys verbunden und in die nächstgrößere Gruppe eingefügt werden¹³.

Die Verwaltung der im Buddy-System vorhandenen Speicherblöcke erfolgt über folgende Funktionen:

¹³Beide Buddys sind frei

buffered_rmqueue: Durchsucht die `free_list` der Zone nach einem ausreichend großen Speicherblock, entfernt den Speicherblock mittels `_rmqueue` aus der Buddy-Datenstruktur und aktualisiert die Seitenstatistik des Kernels.

_rmqueue: Entfernt den Speicherblock aus der Buddy-Datenstruktur und spaltet zu große Blöcke entsprechend auf.

_free_pages: Führt Sicherheitsüberprüfungen durch, fügt den freigegebenen Speicher mittels `_free_pages_bulk` wieder in die Buddy-Datenstruktur ein und aktualisiert die Seitenstatistik des Kernels.

_free_pages_bulk: Fügt den Speicherblock in die Buddy-Datenstruktur ein und verbindet vollständige Buddys.

7.3 Allokation von physikalischem Speicher

Um einen freien Speicherblock anzufordern, stehen mehrere Funktionen zur Verfügung. Diese unterscheiden sich zwar in ihrer Semantik und den notwendigen Parametern, delegieren die Arbeit allerdings immer an die Funktion `_alloc_pages`. Abbildung 8 zeigt die Zusammenhänge zwischen den einzelnen Allokationsfunktionen.

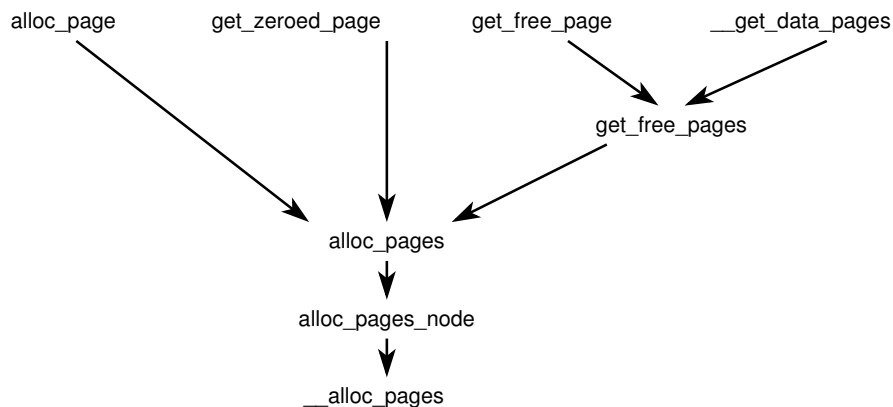


Abbildung 8: Zusammenhänge zwischen den Allokationsfunktionen

Die Hauptaufgabe der `_alloc_pages` Funktion ist das Auffinden eines freien Speicherblocks, was sich vor allem bei Speichermangel als recht schwierig herausstellen kann. Nachdem eine Auflistung des Quellcodes dieser Funktion zu detailliert wäre, werden im Folgenden nur die Grundzüge erläutert.

Der Funktion wird die Größe des zu reservierenden Speicherblocks und eine Instanz der bereits vorgestellten Fallback-Liste übergeben. An erster Stelle in dieser Fallback-Liste steht die Wunschzone für den angeforderten Speicher.

Zuerst geht `_alloc_pages` die Zonen der Fallback-Liste der Reihe nach durch und prüft, ob genügend freie Seiten in der Zone enthalten sind¹⁴. Ist dies der Fall, so wird versucht, mittels `buffered_rmqueue` einen zusammenhängenden Speicherbereich daraus zu entfernen. Dabei kann allerdings das Problem auftreten, dass wenn in der gewünschten Zone nicht mehr ausreichend Speicher zur Verfügung steht, die anderen Zonen der Fallback-Liste stärker belastet werden. Um dies zu verhindern, wurde das Konzept des *inkrementellen Minimum* eingeführt. Für jede Zone ist ein unterer Grenzwert von Speicherseiten, die zumindest frei sein *müssen*, festgelegt. Beim Iterieren über die Zonen wird allerdings nicht dieser Grenzwert verwendet, sondern die Summe der Grenzwerte aller zuvor betrachteten Zonen. Dadurch wird es wesentlich schwieriger, Speicher aus anderen Zonen zu allozieren, da deutlich mehr Speicherseiten frei bleiben müssen.

¹⁴In dieser Phase kann noch nicht gesagt werden, ob der freie Speicher auch zusammenhängend ist.

Sollte der erste Versuch Speicher zu allozieren scheitern, so bedeutet dies, dass nicht mehr allzuviel freier Speicher vorhanden ist. Im nächsten Schritt versucht der Kernel daher etwas aggressiver den angeforderten Speicher zu allozieren. Vorher wird allerdings noch der *kswapd*-Daemon aktiviert, um nicht benötigte Speicherseiten auszulagern.

Danach iteriert die Funktion ähnlich wie zuvor beschrieben nochmals über die Zonen der Fallback-Liste. Dieses mal wird allerdings auch das Flag `__GFP_HIGH` in Betracht gezogen. Dieses Flag eines Prozesses signalisiert, dass der Allokation große Wichtigkeit eingeräumt werden soll. Bei einem gesetzten `__GFP_HIGH`-Flag wird der zum inkrementellen Minimum addierte untere Grenzwert durch 4 dividiert. Dadurch wird eine Allokation in einer anderen Zone erleichtert, führt aber auch zu einer stärkeren Belastung der anderen Zonen.

Schlägt dieser Versuch freien Speicher zu allozieren abermals fehl, so werden drastischere Methoden angewendet. Dieser in den Kernelquellen als *slow path* bezeichnete Weg darf nicht in einem Interrupt-Kontext oder in besonders wichtigen Prozessen eingeschlagen werden. Es wird nochmals über alle Zonen iteriert um nach ausreichend freien Speicherseiten zu suchen, allerdings ohne den unteren Grenzwert an freien Speicherseiten zu beachten.

Abhängig von der angeforderten Anzahl an Speicherseiten und Eigenschaften des anfordernden Prozesses stehen dem Kernel noch weitere Möglichkeiten Speicher zu allozieren zur Verfügung. Diese sollen hier allerdings nicht weiter erläutert werden.

7.4 Der Slab-Allokator

Bisher wurden nur Methoden vorgestellt, um Speicherbereiche in der Größenordnung von einer Speicherseite (4 kiB) und mehr zu reservieren. Sehr häufig benötigt der Kernel jedoch deutlich kleinere Speicherbereiche für verschiedenste Datenstrukturen im Ausmaß von einigen Bytes. Um auch Speicherbereiche in dieser Größenordnung effizient zu verwalten wird der Slab-Allokator eingesetzt. Dieser bietet den zusätzlichen Vorteil, dass Speicherbereiche für häufig benutzte Datenstrukturen in einem eingebauten Cache zwischengespeichert werden können. Dadurch muss der Kernel nicht für jede Instanz einer Datenstruktur freien Speicher anfordern und diesen noch initialisieren, sondern erhält aus dem Cache einen fertig initialisierten Speicherbereich, der sofort verwendet werden kann. Dadurch lässt sich ein nicht zu vernachlässigender Performancegewinn bei der Speicherallokation erzielen. Der Slab-Allokator setzt auf das Buddy-System auf.

Grundsätzlich besteht der Slab-Allokator aus dem Cache und den einzelnen Slabs. Dabei ist jeder Cache für genau einen Objekttyp zuständig, beispielsweise für Instanzen von `struct unix_sock` oder einen allgemeinen Puffer. Im Cache sind alle für die Verwaltung der zugehörigen Slabs notwendigen Daten gespeichert. Auf den Slabs sind die verwalteten Objekte enthalten. Zusätzlich sind auch hier Verwaltungsdaten für die einzelnen Objekte notwendig. Die Struktur des Slab-Allokators ist in Abbildung 9 veranschaulicht.

Der Cache besteht im Wesentlichen aus drei Listen, auf denen die vom Cache verwalteten Slabs aufgereiht werden sowie einem Array in dem für jede CPU die zuletzt freigegebenen Objekte gehalten werden. Die drei Listen teilen die Slabs in vollständig belegt, teilweise belegt und vollständig leer ein. Zusätzlich sind noch eine Reihe von Verwaltungsdaten wie zum Beispiel die Anzahl der freien und belegten Objekte oder Informationen die beim Vergrößern und Verkleinern des Caches benötigt werden enthalten. Abbildung 10 veranschaulicht die Struktur eines Slab-Caches graphisch.

Die prozessorspezifischen, zuletzt freigegebenen Objekte sind wichtig, um die CPU-Caches optimal auszunützen. Sie werden nach dem LIFO¹⁵-Verfahren verwendet. Dabei geht der Kern davon aus, dass sich ein eben zurückgegebenes Objekt noch im Cache befindet und vergibt diese so schnell wie möglich neu. Erst wenn dieser CPU-spezifische Cache leer ist, werden unbenutzte Objekte aus den Slabs vergeben.

Auf den Slabs sind die Objekte nicht kontinuierlich aufgereiht, sondern nach einem etwas komplizierteren Schema verteilt. Für jedes Objekt wird nicht die Größe reserviert die es tatsächlich besitzt, sondern

¹⁵last in first out

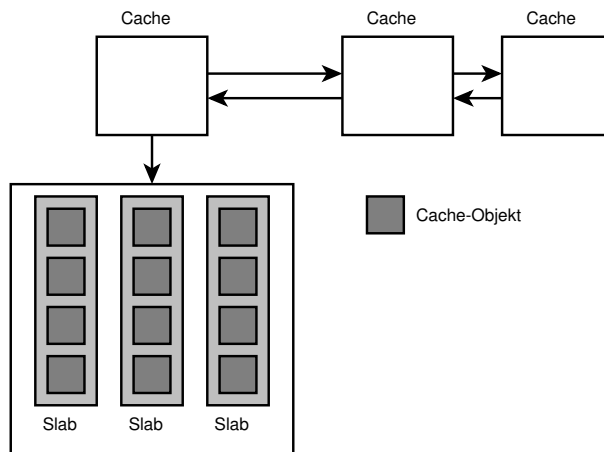


Abbildung 9: Komponenten des Slab-Allokators

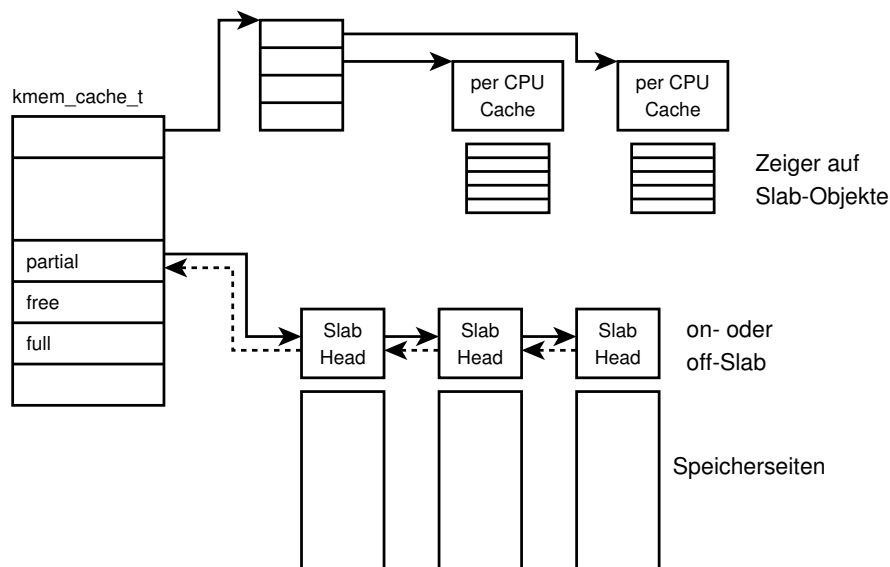


Abbildung 10: Struktur des Slab-Caches

es wird auf ein Vielfaches von `BYTES_PER_WORD` aufgerundet. Dadurch wird der Zugriff auf Objekte im Slab-Speicher beschleunigt.

Zusätzlich erhält der Slab noch eine *Färbung*. In den meisten Fällen ist die Größe des Slab-Bereichs (abzüglich der Verwaltungsdaten) nicht ganzzahlig durch den für ein Objekt verwendeten Speicherplatz teilbar. Dieser überschüssige Speicherplatz wird dazu verwendet, einen Offset am Anfang des Slabs zu vergeben, wodurch der Slab die angesprochene Farbe erhält. Alle Slabs innerhalb eines Caches erhalten unterschiedliche Offsets. Dadurch versucht der Kern ebenfalls die Hardware-Caches im System besser auszunutzen.

Die Verwaltungsdaten eines Slabs enthalten im Wesentlichen Informationen über die freien und belegten Objekte sowie Zeiger auf das nächste freie Objekt. Diese Daten können entweder auf dem Slab selbst oder in einem externen Speicherbereich angelegt werden. Die Entscheidung, ob der Verwaltungskopf on- oder off-Slab gelagert wird, ist abhängig von der Größe der Objekte und des überschüssigen Speichers.

Die Anzahl der Slabs eines Caches hängt von der Objektgröße und der Anzahl der verwalteten Objekte ab.

Zum Erstellen eines solchen Caches ist die Funktion `kmem_cache_create` vorgesehen. Als Parameter muss ein benutzerlesbarer Name¹⁶, die Größe der zu verwaltenden Objekte, ein gewünschter Offset für die Ausrichtung der Daten (meist null), einen Satz an Flags sowie Konstruktor- und Destruktorfunktionen übergeben werden.

Um Speicher vom Slab-Allokator anzufordern stehen zwei Funktionen zur Verfügung:

- `kmem_cache_alloc`
- `kmalloc`

Mit der Funktion `kmem_cache_alloc` kann Speicher für ein bestimmtes Objekt angefordert werden. Als Parameter muss der für diese Objekte zuständige Cache, der zuvor mittels `kmem_cache_create` erstellt wurde, übergeben werden. Der Kernel verwendet eine Reihe von spezialisierten Caches:

- `task_struct_cache`
- `sk_buff_head_cache`
- `inode_cache`
- `pte_chain`
- `signal_cache`
- `sighand_cache`
- ...

`kmalloc` ist das Kernel-Analogon zu `malloc` aus der C-Standardbibliothek im Userspace. Damit ist es möglich, Speicher im klassischen Sinn zu allozieren. Die Basis von `kmalloc` ist ein Array, in dem Slab-Caches für Speicherbereiche in unterschiedlichen Größen gehalten werden. Im Wesentlichen sind dies Zweierpotenzen zwischen $2^5 = 32$ und $2^{17} = 131072$ Bytes. Beim Anfordern von Speicher mittels `kmalloc` wird zunächst der passende Slab-Cache ermittelt und von diesem ein Objekt zurückgeliefert.

Zum Freigeben von Speicher stehen ebenfalls zwei Funktionen zur Verfügung:

`kmem_cache_free` um mittels `kmem_cache_alloc` allozierten Speicher wieder freizugeben

`kfree` als Analogon zu `free` um mittels `kmalloc` allozierten Speicher wieder freizugeben.

7.5 Seitentabellen

Jedem Prozess steht ein kontinuierlicher virtueller Adressraum zur Verfügung. Dieser Adressraum umfasst 4 GiB, was auf 32-Bit Architekturen dem maximal adressierbaren Speicher entspricht ($2^{32} = 4$ GiB). Die Speicherseiten des virtuellen Adressraums werden vom Kernel in Zusammenarbeit mit dem Prozessor auf physikalische Speicherseiten umgelegt. Die für diese Abbildung verwendeten Mechanismen müssen diese Aufgabe schnell und effizient lösen. Ein Adressraum von 4 GiB enthält $2^{20} = 1048576$ Speicherseiten zu je 4kiB, was bei einer naiven Implementation sehr speicheraufwändig und ineffizient werden kann.

Der Kernel verwendet daher für die Verwaltung des virtuellen Adressraums eine dreistufige Seitentabelle. Damit ist es möglich, die Abbildung von virtuellen auf physikalische Adressen nur für jene Seiten zu speichern, die der Prozess verwendet¹⁷ und die auch tatsächlich vorhanden sind¹⁸. Die drei Seitentabellen haben im Kernel folgende Bezeichnungen:

¹⁶dieser erweist bei der Ausgabe von `/proc/slabinfo`

¹⁷Die meisten Prozesse nutzen nur einen geringen Teil ihres virtuellen Adressraumes aus.

¹⁸Wenn nicht ausreichend physikalischer Speicher zur Verfügung steht können Speicherseiten eines Prozesses auch ausgelagert werden./

- Page Global Directory (PGD)
- Page Middle Directory (PMD)
- Page Table (PTE)

Die virtuellen Speicheradressen werden, wie in Abbildung 11 veranschaulicht, in vier Teile aufgespalten. Die ersten drei Teile der Adresse dienen zur Auswahl der Speicherseite in den Seitentabellen und der letzte Teil gibt die Position innerhalb der Speicherseite an.

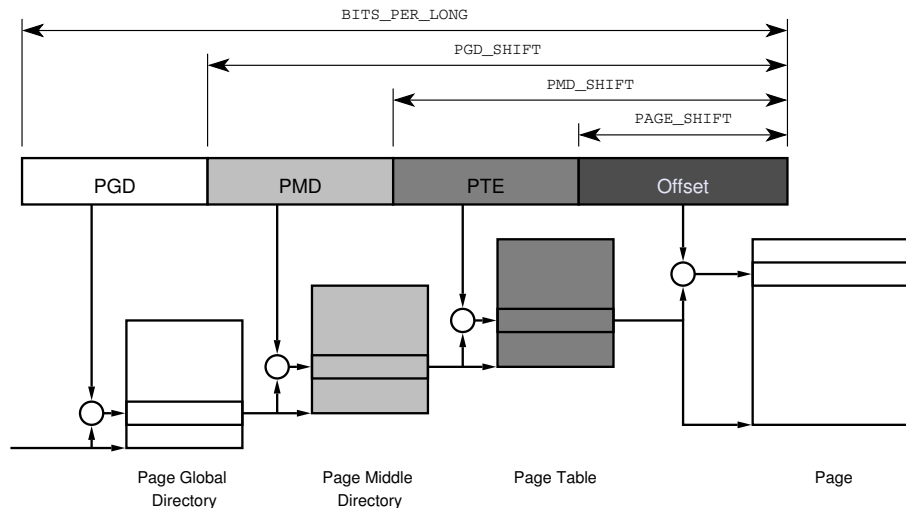


Abbildung 11: Aufteilung einer virtuellen Adresse

Abbildung 11 veranschaulicht auch die Verwendung der dreistufigen Seitentabelle. In der `task_struct` ist unter anderem ein Zeiger auf das globale Seitenverzeichnis¹⁹ enthalten. Jener Teil der virtuellen Adresse der als PGD bezeichnet wird, dient als Offset in der globalen Seitentabelle und liefert einen Zeiger auf eine mittlere Seitentabelle. Mit dem zweiten Teil der virtuellen Adresse als Index für die mittlere Seitentabelle erhält man einen Zeiger für die dritte Seitentabelle die schließlich einen Zeiger auf die Speicherseite liefert. Der letzte Teil der virtuellen Adresse ist dann die Adresse der gewünschten Speicherzelle innerhalb der Seite.

Die Positionen der Adresselemente sind durch folgende Bitshifts festgelegt:

`BITS_PER_LONG` gibt an, wie viele Bits für eine Adresse verwendet werden.

`PAGE_SHIFT` definiert, wie viele Bits für die Position innerhalb einer Speicherseite notwendig sind.

`PMD_SHIFT` gibt an, wie viele Bits von einer Seite *und* einem Eintrag aus der letzten Stufe der Seitentabelle zusammen verwendet werden. Mit Hilfe dieses Wertes kann leicht die Größe des Teiladressraumes von einem Eintrag in der mittleren Seitentabelle ermittelt werden, nämlich $2^{\text{PMD_SHIFT}}$ Bytes.

`PGD_SHIFT` analog wie `PMD_SHIFT`, jedoch inklusive der Bits für die mittlere Seitentabelle.

Zusätzlich sind auch die Anzahl der Zeiger in den verschiedenen Seitentabellen über Makros festgehalten: `PTRS_PER_PGD`, `PTRS_PER_PMD` und `PTRS_PER_PTE`.

Als Datenstruktur für die Einträge in den Seitentabellen werden folgende C-Strukturen verwendet:

```
1 typedef struct { unsigned long pte_low; } pte_t;
2 typedef struct { unsigned long pmd; } pmd_t;
```

¹⁹global bezieht sich hier auf den Adressraum eines Prozesses.

```
3 typedef struct { unsigned long pgd; } pgd_t;
```

Listing 13: Einträge der Seitentabellen (*include/asm-i386/page.h*)

Auffallend hierbei ist, dass die elementaren Datentypen in `structs` gekapselt sind. Dies wurde deshalb gemacht, damit der Inhalt nur mit den zugehörigen Hilfsfunktionen manipuliert wird. Ausserdem kann es für andere Architekturen notwendig sein, aufwändigere Tabelleneinträge zu benutzen.

Die vom Kern verwendete dreistufige Seitentabelle entspricht meist nicht den von Prozessoren verwendeten Seitentabellen. Auf IA-32 Systemen wird zum Beispiel nur eine zweistufige Seitentabelle unterstützt, während AMD-64 Systeme eine vierstufige Seitentabelle verwenden. In solch einem Fall muss die vom Kernel verwendete dreistufige Tabelle auf die Zielarchitektur entsprechend abgebildet werden. Im Fall von IA-32 wird dazu die Anzahl der Bits für das mittlere Seitenverzeichnis auf null gesetzt, die dreistufige Struktur bleibt allerdings erhalten.

7.6 Verwaltung des virtuellen Prozessspeichers

Wie bereits im vorigen Abschnitt erwähnt, umfasst der virtuelle Adressraum für jeden Prozess 4 GiB. Von diesen 4 GiB stehen dem Benutzerprozess allerdings nur die ersten drei GiB zur Verfügung, im vierten GiB befindet sich der Kernel. Dieser befindet sich physikalisch nur einmal im Speicher und wird über die Seitentabellen bei jedem Benutzerprozess im letzten GiB eingeblendet, während *jedem* Benutzerprozess drei GiB zur Verfügung stehen. Auf den Kernel-Adressraum dürfen Benutzerprozesse natürlich nicht zugreifen, dazu muss zuvor in den Kernelmodus gewechselt werden (siehe Abschnitt 8).

Im virtuellen Adressraum eines Prozesses sind unter anderem folgende Dinge enthalten:

- Binärcode des Programms, auch als Textsegment bezeichnet
- Programmcode von dynamisch geladenen Bibliotheken
- Stack zur Aufnahme lokaler Variablen
- Heap für dynamisch allozierten Speicher
- Speicherbereich mit Kommandozeilenargumenten und Umgebungsvariablen
- Memory Mappings, die den Inhalt von Daten- oder Gerätedateien in den virtuellen Adressraum einblenden²⁰

Alle diese Abschnitte dienen unterschiedlichen Zwecken und müssen auch unterschiedlich behandelt werden. So soll es zum Beispiel nicht möglich sein, das Textsegment zu modifizieren, allerdings muss der Inhalt ausführbar sein. Oder im Fall des Stacks auf den sowohl lesend als auch schreibend zugegriffen werden kann, der allerdings nicht ausführbar sein soll.

Nachdem der Adressraum aller Prozesse in den meisten Fällen deutlich größer als der physikalisch vorhandene Speicher ist, wird für jeden Prozess nur ein Teil seines virtuellen Adressraumes im physikalischen Speicher gehalten. Dieses Vorgehen ist durchaus legitim, da in den meisten Fällen nur ein geringer Teil immer gebraucht wird. Zum Beispiel muss der Programmquelltext für Initialisierungsfunktionen oder nur selten gebrauchte Funktionen wie die Hilfe oder Konfigurationsdialoge nicht immer im physikalischen Speicher vorhanden sein. Diese werden vom Kernel erst bei Bedarf in den physikalischen Speicher geladen und die Seitentabelle des Prozesses entsprechend angepasst.

Um ein solches *Demand Loading* bzw. *Demand Paging* zu ermöglichen muss der Kernel für jeden Prozess die einzelnen Regionen des virtuellen Adressraumes verwalten. Zu diesem Zweck enthält jeder Prozess in seiner `task_struct` einen Zeiger auf eine Instanz von `mm_struct` die unter anderem folgende Felder enthält:

²⁰Auf Memory Mappings soll hier allerdings nicht näher eingegangen werden.

```

1 struct mm_struct {
2     struct vm_area_struct * mmap;    /* list of VMAs */
3     struct rb_root mm_rb;
4     struct vm_area_struct * mmap_cache; /* last find_vma result */
5     /* ... */
6 }

```

Listing 14: Datenstruktur zur Verwaltung des Prozessspeichers (*include/linux/sched.h*)

Die einzelnen Regionen des virtuellen Adressraums werden durch eine Instanz von `vm_area_struct` abgebildet. Alle Regionen des Prozesses werden sortiert nach ihrer Adresse auf einer einfach verketteten Liste in `mmap` gehalten.

Zusätzlich ist noch ein Feld `mm_rb` des Typs `rb_root` vorhanden. Hierbei handelt es sich um den Wurzelknoten eines *Rot-Schwarz-Baumes* dessen Knoten Zeiger auf die `vm_area_structs` der verketteten Liste enthalten. Der Rot-Schwarz-Baum wird dazu verwendet, um Speicherregionen effizient zu finden. Beim Einfügen von Speicherregionen wird zunächst die Einfügeposition über den Rot-Schwarz-Baum ermittelt und dann die Region in die verkettete Liste eingefügt. Natürlich muss bei Einfüge- und Löschoptionen auch der Rot-Schwarz-Baum aktuell gehalten werden.

Bis zur Version 2.4.9 wurde anstelle des Rot-Schwarz-Baumes ein AVL-Baum verwendet, welcher ähnliche Eigenschaften besitzt.

2.4

`mmap_cache` dient als Cache für die zuletzt bearbeitete Region. Dieses Feld wird von den Funktionen, die auf den Speicherregionen operieren verwendet, um bei hintereinanderfolgenden Operationen auf die gleiche Speicherregion nicht jedes Mal den Rot-Schwarz-Baum durchsuchen zu müssen.

Für die Darstellung einer Speicherregion wird folgende Datenstruktur verwendet:

```

1 struct vm_area_struct {
2     struct mm_struct * vm_mm; /* The address space we belong to. */
3     unsigned long vm_start; /* Our start address within vm_mm. */
4     unsigned long vm_end; /* The first byte after our end address within vm_mm. */
5
6     /* linked list of VM areas per task, sorted by address */
7     struct vm_area_struct * vm_next;
8
9     pgprot_t vm_page_prot; /* Access permissions of this VMA. */
10    unsigned long vm_flags; /* Flags, listed below. */
11
12    struct rb_node vm_rb;
13
14    /*
15     * For areas with an address space and backing store,
16     * one of the address_space->i_mmap{,shared} lists,
17     * for shm areas, the list of attaches, otherwise unused.
18     */
19    struct list_head shared;
20
21    /* Function pointers to deal with this struct. */
22    struct vm_operations_struct * vm_ops;
23
24    /* Information about our backing store: */
25    unsigned long vm_pgoff; /* Offset (within vm_file) in PAGE_SIZE
26                            units, *not* PAGE_CACHE_SIZE */
27    struct file * vm_file; /* File we map to (can be NULL). */
28    void * vm_private_data; /* was vm_pte (shared mem) */
29 };

```

Listing 15: Datenstruktur für eine Speicherregion (*include/linux/mm.h*)

`vm.start`, `vm.end` geben die Start- und Endadresse der Speicherregion an. In den Kernelquellen wird für virtuelle Adressen gerne der Datentyp `unsigned long` anstelle von `void *`, da diese etwas einfacher zu handhaben und zu manipulieren sind. Dies ist gefahrlos möglich, da die notwendige Bedingung `sizeof(unsigned long) == sizeof(void *)` auf allen von Linux unterstützten Architekturen eingehalten wird.

`vm.next` zeigt auf die nächste Speicherregion

`vm.mm` zeigen auf die `mm_struct` des Prozesses zu dem diese Region gehört

`vm.rb` ist der Knoten zur Repräsentation der Region im Rot-Schwarz Baum.

`vm.page_prot`, `vm.flags` speichern die Zugriffsflags sowie allgemeine Flags wie z.B. ob eine Region geshared ist (Code einer dynamischen Bibliothek) oder in welche Richtung die Region wachsen darf (der Stack z.B. wächst nach unten).

`shared` Wenn diese Region geshared ist, wird in diesem Feld eine Liste aller beteiligten Regionen gespeichert.

`vm.ops` ist ein Zeiger auf einen Satz von Operationen auf diese Region. Zu diesen Operationen gehört zum Beispiel eine Funktion zum Einlesen ausgelagerter Speicherseiten. Nachdem es unterschiedliche Arten von Regionen gibt, müssen diese auch unterschiedlich behandelt werden.

Die restlichen Felder werden benötigt, um die notwendigen Daten von einem Sekundärspeicher für diese Region einzulesen.

Im Kernel ist ein Satz von Funktionen vorhanden, um die Regionen eines Prozesses zu manipulieren:

- Auffinden eines geeigneten Speicherbereiches für eine neue Region
- Erstellen einer neuen Region
- Löschen einer Region

Beim Erstellen und Löschen einer neuen Region nimmt der Kernel einige Optimierungen vor. Wird eine neue Region unmittelbar vor oder hinter eine bereits bestehende Region eingefügt, so verschmilzt der Kernel diese zu einer einzigen Region. Dies ist natürlich nur möglich, wenn die Zugriffsrechte der beteiligten Regionen identisch sind und es sich um Daten vom gleichen Backing Store handelt. Wird ein Abschnitt am Anfang oder am Ende einer Region entfernt, so muss die bestehende Datenstruktur nur entsprechend verkürzt werden.

Um das Zusammenspiel zwischen der Seitentabelle und dem virtuellen Adressraum zu verdeutlichen, soll im Folgenden der Ablauf bei Behandlung eines Seitenfehlers beschrieben werden. Ein Seitenfehler tritt immer dann auf, wenn ein Prozess auf eine virtuelle Speicheradresse zuzugreifen versucht, für die im Prozessor keine Abbildung auf eine physikalische Adresse vorhanden ist. In diesem Fall löst der Prozessor einen Seitenfehler (*Page Fault*) aus, der vom Kernel behandelt wird. Dabei gilt es einige Dinge zu beachten. Eine vereinfachte Darstellung der vom Kernel durchgeführten Überprüfungen ist in [Abbildung 12](#) dargestellt.

Zunächst muss geprüft werden, ob es sich um eine Kernel- oder Userspace-Adresse handelt. Im Fall einer Kerneladresse wird geprüft, ob sich der Prozessor zum Zeitpunkt des Fehlers im Kernelmodus befunden hat. War dies der Fall, so muss lediglich die Seitentabelle des Prozessors auf den aktuellen Stand gebracht werden; anderenfalls handelt es sich um einen Speicherzugriffsfehler und dem Benutzerprozess wird das entsprechende Signal geschickt (*SigSegv*). Handelt es sich um eine Userspace-Adresse und es existiert ein Mapping für die fehlerhafte Adresse (es existiert eine `vm_area_struct` die die fehlerhafte Adresse einschließt) und die Privilegien für diese Region sind entsprechend gesetzt, so muss der Kernel die gewünschten Daten evtl. von einem Backing Store in das RAM laden und die Seitentabelle entsprechend anpassen. Trifft eines der Kriterien nicht zu, so handelt es sich ebenfalls um einen Speicherzugriffsfehler und dem Benutzerprozess wird das entsprechende Signal gesendet.

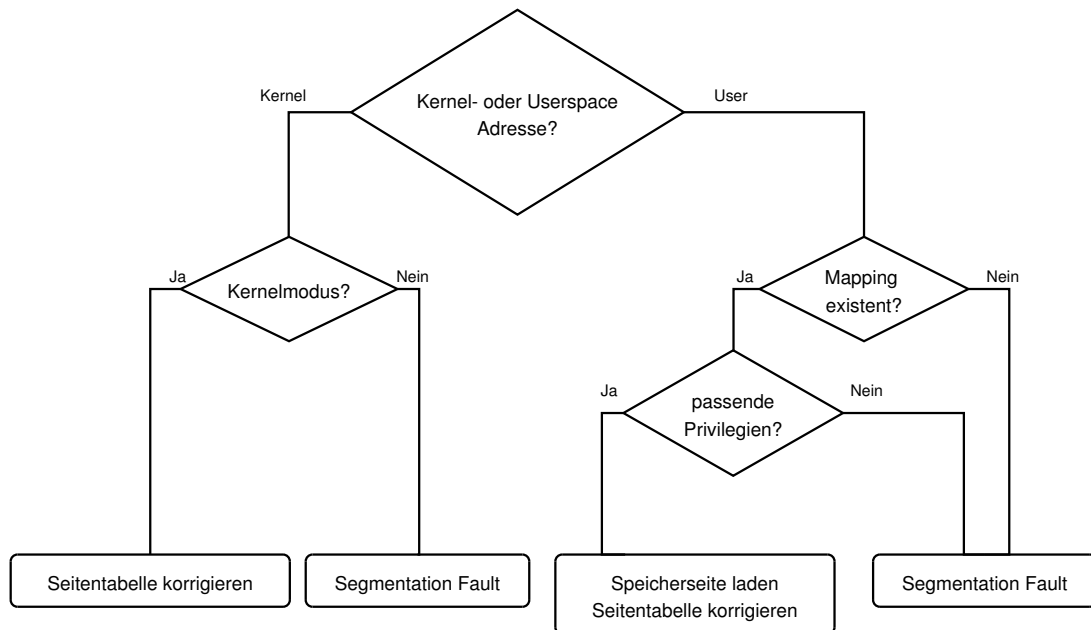


Abbildung 12: Behandlungsroutine eines Seitenfehlers

Die tatsächliche Implementierung dieser Funktion umfasst noch eine Reihe weiterer Überprüfungen, welche auch berücksichtigen, ob es sich bei der betroffenen Region um den Stack oder den Heap handelt und diese entsprechend vergrößert.

8 System Calls

Alle Benutzerprozesse werden im sogenannten *Usermode* ausgeführt. In diesem unprivilegierten Modus, der auch vom Prozessor unterstützt werden muss, ist es nicht möglich, direkt auf die Ressourcen des Systems zuzugreifen. Dazu muss der Prozess vorgegebene Servicerroutinen des Kernels verwenden, die auch als *System Calls* bezeichnet werden. Die Systemaufrufe werden im privilegierten Kernelmode ausgeführt, in dem direkt auf die Ressourcen zugegriffen werden kann. Zuvor führt der Kern allerdings gewisse Überprüfungen durch, zum Beispiel, ob der Prozess die notwendigen Rechte besitzt oder ob die gewünschte Ressource auch frei ist.

Nachdem System Calls die einzige Möglichkeit für Benutzerprozesse darstellen mit dem Kernel zu interagieren, sind diese standardisiert. Dadurch ist es leicht möglich, Benutzerprogramme auf andere Betriebssysteme zu portieren. Leider gibt es allerdings nicht nur einen Standard für Systemaufrufe. Der wohl bekannteste und auch am weitesten verbreitete Standard ist der *POSIX*-Standard (Portable Operating System Interface). Linux bemüht sich – in Zusammenarbeit mit der C-Bibliothek – diesen Standard zu implementieren. Seit der ersten Version Ende der 80er Jahre hat dieser Standard deutlich an Umfang gewonnen – die aktuelle Version verteilt sich auf vier Bände²¹.

Neben POSIX gibt es auch noch weitere Standards, die allerdings im Gegensatz zu POSIX nicht einem Komitee entstammen, sondern ihre Wurzeln in der Entwicklung von Unix und seinen Derivaten haben. Diese sind der *System V*-Standard sowie der *4.3BSD*-Standard. Linux stellt auch Systemaufrufe aus diesen beiden Standards zur Verfügung.

²¹In elektronischer Form unter <http://www.opengroup.org/onlinepubs/007904975/> zu finden

8.1 Vorhandene Systemaufrufe

Die exakte Zahl der Systemaufrufe im Kernel ist plattformabhängig, da nicht alle Aufrufe auf allen Architekturen Sinn machen. Man kann aber sagen, dass es über 200 Stück sind. Um dabei den Überblick zu behalten werden diese in folgende Kategorien eingeteilt:

Prozessverwaltung: Systemaufrufe aus dieser Gruppe stellen Funktionalität zur Verwaltung von Prozessen bereit. Dazu zählen u.a. das Abfragen von Informationen über einen Prozess, Starten und Beenden eines Prozesses und das Ändern der Priorität.

Zeitoperationen: Dabei handelt es sich nicht nur um Funktionen zur Abfrage oder Änderung der Systemzeit, sondern sie geben vor allem Prozessen die Möglichkeit, zeitgesteuerte Operationen durchführen zu können.

Signalverarbeitung: Signale dienen zum Austausch einfacher Nachrichten zwischen Prozessen. Systemaufrufe aus dieser Kategorie stellen den Prozessen die notwendigen Funktionen dazu bereit.

Scheduling: Diese Systemaufrufe gehörten ursprünglich zur Kategorie der Prozessverwaltung. Nachdem Linux allerdings eine Vielfalt an Parametrisierungen für das Prozessverhalten bereitstellt, wurden diese in eine eigene Kategorie verlagert. Zu dieser Kategorie gehören u.a. `sched_setparam` zum ändern der Scheduling-Klasse.

Dateisystem: In dieser Gruppe sind alle Funktionen die das Dateisystem betreffen zusammengefasst, z.B. Erstellen und Löschen von Verzeichnissen, Bearbeiten von Dateien, Ändern der Berechtigungen usw.

Speicherverwaltung: Mit Systemaufrufen aus dieser Kategorie gelangen Benutzerprogramme nur sehr selten in Berührung, da dieser Teil von der C-Standardbibliothek abgedeckt und in Form von abstrakteren Funktionen zur Verfügung gestellt wird. Die wichtigsten Aufrufe in dieser Kategorie sind jene zur Verwaltung des dynamischen Speichers.

Interprozesskommunikation und Netzwerkfunktionen Hier stehen nur zwei Systemaufrufe zur Verfügung. Einer, der alle Aufgaben das Netzwerk betreffend abdeckt und ein weiterer für Interprozesskommunikation. Diese Systemaufrufe sind aus historischen Gründen sogenannte Multiplex-Systemaufrufe. Der erste Parameter gibt die gewünschte Funktion an. Die Standardbibliothek stellt allerdings für Anwendungen getrennte Funktionen zur Verfügung.

Systeminformationen und -einstellungen Stellt Systemaufrufe zum Auslesen von Informationen über den laufenden Kernel und seine Konfiguration sowie Ausstattung und Auslastung des Systems zur Verfügung.

Systemsicherheit und Capabilities: Systemaufrufe dieser Kategorie dienen dazu, normalen Benutzerprozessen mehr Rechte einzuräumen. Weiters stehen Aufrufe zur Verfügung um sicherheitsrelevante Überprüfungen durchzuführen.

Die symbolischen Bezeichnungen der Systemaufrufe erhalten vom Kernel eine eindeutige Kennzahl zugewiesen. Dies ist notwendig, damit sie von Benutzerprozessen eindeutig identifiziert werden können, wie in Abschnitt 8.2 noch erläutert wird. Ein direkter Aufruf der Kernel-Funktionen ist nicht möglich.

Die zugewiesenen Kennzahlen sind ebenso wie die vorhandenen Systemaufrufe architekturabhängig. Diese sind in `arch/<arch>/kernel/entry.S` aufgelistet.

8.2 Realisierung von Systemaufrufen

Bei einem Systemaufruf wird, wie bereits erwähnt, vom Usermode in den Kernelmode geschaltet. Dieses Umschalten des Berechtigungslevels erfordert auch spezielle Mechanismen, die vom Prozessor bereitgestellt werden müssen, denn der Benutzerprozess darf nicht direkt auf den Kernelspeicher zugreifen. Die dazu notwendigen Schritte sind daher stark systemabhängig und durchgehend in Assembler geschrieben.

Um nicht die gesamte Funktionalität für alle unterstützten Architekturen neu zu implementieren, gibt es allgemeingültig implementierte Handler-Funktionen. An diese wird die gesamte Arbeit letztendlich delegiert. Diese Handler-Funktionen sind im Quellcode durch den Präfix `sys_` gekennzeichnet und unterliegen folgenden Einschränkungen²²:

- es dürfen maximal fünf Parameter übergeben werden
- es sind nur primitive Datentypen möglich
- es ist kein direkter Speicherzugriff auf den Prozessspeicher möglich

Der Ablauf des Umschaltens vom Usermode in den Kernelmode soll im Folgenden für die IA-32 Architektur näher betrachtet werden. Auf den übrigen von Linux unterstützten Architekturen wird nach demselben Prinzip vorgegangen, jedoch muss auf die speziellen Eigenschaften des Prozessors eingegangen werden.

Da ein direkter Zugriff auf den Kernelspeicher nicht erlaubt ist, muss ein anderer Weg für den Wechsel in den Kernelmode gewählt werden. Im Fall der IA-32 Architekturen erfolgt dies über Auslösen eines Software-Interrupts, konkret ist dies der Interrupt `0x80`. In der Interrupt-Behandlungsroutine, die vom Kernel bereitgestellt wird, erfolgt dann der tatsächliche Wechsel in den privilegierten Modus. Dies geschieht auf allen Architekturen über spezielle Assembler-Anweisungen.

Im nächsten Schritt muss der gewünschte Systemaufruf identifiziert werden. Dazu wird im Register `eax` die Nummer des Systemcalls übergeben. Dieser Wert wird als Index für ein Array aus Pointern auf die Highlevel C-Funktionen verwendet. Dabei entspricht die einem Systemcall zugewiesene Kennzahl dem Index im Array.

Die für den Aufruf notwendigen Parameter werden in den Registern `ebx`, `ecx`, `edx`, `esi` und `edi` übergeben. Daraus erkennt man die zuvor bereits erwähnten Einschränkungen hinsichtlich Anzahl und des Typen der Parameter. Dieses Vorgehen ist deshalb notwendig, weil im Kernelmode ein anderer Stack als im Usermode verwendet wird, wodurch eine Übergabe über den Stack nicht möglich ist.

In der Interrupt-Routine werden die Parameter aus den Registern auf den Kernel-Stack gelegt und anschließend die entsprechende highlevel C-Funktion aufgerufen.

Müssen dem Systemaufruf mehr als fünf Parameter bzw. komplexere Datenstrukturen übergeben werden, so kann dies über Pointer realisiert werden. Der Kernel muss dabei allerdings unterscheiden, ob es sich um einen aus Kernel-Sicht gewöhnlichen Pointer oder um einen Pointer in den Userspace handelt. Ein Pointer in den Userspace darf vom Kernel nicht direkt dereferenziert werden, da es sich um eine virtuelle Adresse handelt und der Kernel nicht sicher sein kann, ob für diese Adresse ein mapping auf eine physikalische Adresse existiert. Daher werden spezielle Funktionen verwendet um die von einem Zeiger referenzierten Datenstrukturen in den Kernelspeicher zu kopieren. Um eine korrekte Verwendung von Zeigern in den Userspace sicherzustellen werden diese mit dem Attribut `__user` gekennzeichnet. Dadurch ist eine automatische Überprüfung mit Hilfe spezieller Tools möglich.

Der Rückgabewert eines Systemcalls ist in der Regel ebenfalls ein primitiver Datentyp. Dieser wird von der architekturabhängigen Wrapper-Funktion ausgelesen und über das `eax`-Register dem Benutzerprozess zurückgegeben. In manchen Fällen ist es allerdings notwendig, mehrere Daten an den Benutzerprozess zurückzugeben. In diesem Fall werden die restlichen Rückgabewerte mittels *Call-by-Reference* zurückgegeben. Allerdings muss hier wieder die Grenze zwischen Kernel- und Userspace überwunden werden. Daher steht auch zum Kopieren von Daten aus dem Kernelspace in den Userspace eine spezielle Funktion zur Verfügung.

Der aus Sicht des Userprogramms aufwändig und kompliziert scheinende Aufruf eines Systemcalls – Ablegen der Parameter und Systemcall-Nummer in den Registern, Auslösen des Softwareinterrupts, Auslesen des Return-Wertes aus dem Register – wird von Wrapper-Funktionen in der C-Bibliothek übernommen.

²²Diese Einschränkungen ergeben sich aus den verwendeten Mechanismen zum Umschalten in den Kernelmode.

Für ein Benutzerprogramm können Systemaufrufe somit wie gewöhnliche Funktionsaufrufe verwendet werden.

Abbildung 13 veranschaulicht den Ablauf eines Systemaufrufs grafisch.

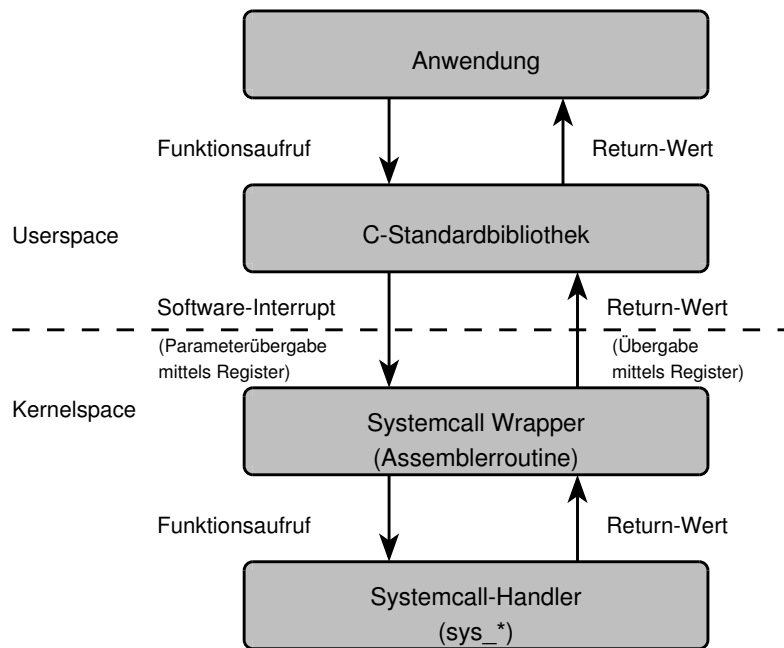


Abbildung 13: Aufruf eines Systemcalls

9 Netzwerk Stack

Gängige Netzwerk-Modelle wie beispielsweise das ISO/OSI oder das TCP/IP Modell sind in verschiedene Schichten unterteilt, die jeweils eine klar definierte Aufgabe haben und über ein sauber definiertes Interface mit den unmittelbar darüber bzw. darunter liegenden Schichten kommunizieren.

Auch der Linux Kernel macht sich diese Systematik zu eigen, weswegen der Netzwerk Code ebenfalls als Schichtmodell realisiert ist. Auf Grund von praktischen Überlegungen, wie beispielsweise der Wichtigkeit und großen Verbreitung, orientiert sich der Kernel stark am TCP/IP Modell.

Da TCP/IP derzeit eines der wichtigsten Protokolle ist, werden sich auch die nachfolgenden Betrachtungen hauptsächlich auf diese Protokollfamilie konzentrieren. Selbstverständlich werden vom Kernel noch eine Vielzahl anderer Protokolle unterstützt. Eine genauere Betrachtung würde den Rahmen dieser Arbeit jedoch bei weitem sprengen. Außerdem bleiben die Mechanismen, wie die einzelnen Schichten miteinander kommunizieren, weitgehend gleich.

9.1 Datenstrukturen: Socketbuffer

Netzwerkpakete müssen auf möglichst effiziente Weise zwischen den einzelnen Schichten im Kernel weitergereicht werden können. Zu diesem Zweck wird eine eigene Datenstruktur namens `struct sk_buff` verwendet. Listing 16 gibt diese Datenstruktur etwas verkürzt und mit einigen zusätzlichen Kommentaren wieder.

```

1
2  struct sk_buff {
3      /* These two members must be first. */

```

```
4     struct sk_buff    *next;
5     struct sk_buff    *prev;
6
7     struct sk_buff_head *list;
8     struct sock        *sk;
9     struct timeval     stamp;
10    struct net_device  *dev;
11    struct net_device  *real_dev;
12
13    /* Transport layer header */
14    union {
15        struct tcphdr *th;
16        struct udphdr *uh;
17        struct icmphdr *icmph;
18        struct igmpchr *igmpch;
19        struct iphdr  *iph;
20        unsigned char *raw;
21    } h;
22
23    /* Network layer header */
24    union {
25        struct iphdr  *iph;
26        struct ipv6hdr *ipv6h;
27        struct arphdr *arph;
28        unsigned char *raw;
29    } nh;
30
31    /* Link layer header */
32    union {
33        struct ethhdr *ethernet;
34        unsigned char *raw;
35    } mac;
36
37    struct dst_entry *dst;
38    struct sec_path  *sp;
39
40    /*
41     * This is the control buffer. It is free to use for every
42     * layer. Please put your private variables there. If you
43     * want to keep them across layers you have to do a skb_clone()
44     * first. This is owned by whoever has the skb queued ATM.
45     */
46    char    cb[48];
47
48    unsigned int    len,
49                  data_len,
50                  csum;
51    unsigned char   local_df,
52                  cloned,
53                  pkt_type,
54                  ip_summed;
55    __u32    priority;
56    unsigned short protocol,
57               security;
58
59    void      (*destructor)(struct sk_buff *skb);
60
61
62    /* These elements must be at the end, see alloc_skb() for details. */
63    unsigned int    truesize;
64    atomic_t        users;
65    unsigned char   *head,
66                  *data,
67                  *tail,
68                  *end;
69 };
```

Listing 16: Datenstruktur `sk_buff` (*include/linux/skbuff.h*)

Für jede Schicht, die von einem Paket durchlaufen wird, kommt in der Regel ein eigener Header hinzu. So beinhaltet ein Ethernet Frame zu Beginn typischer Weise einen MAC Header während im Nutzdatenanteil der IP Header mit seinen Nutzdaten untergebracht wird. Dieser beherbergt dann beispielsweise einen TCP Header mit zugehörigem Nutzdatenteil welcher schlussendlich Header und Daten des Applikationsprotokolls beinhaltet. Die `struct sk_buff` Datenstruktur ermöglicht einen effizienten Umgang mit Paketen dieser Art. Das Hinzufügen bzw. Entfernen von Headern reduziert sich mit ihrer Hilfe auf eine einfache Manipulation von Zeigern. Die wichtigsten Elemente der Struktur sollen nun kurz beschrieben werden.

next und prev dienen zur Verwaltung der `sk_buff` Instanzen in einer doppelt verketteten Liste

stamp Ankunftszeit des Pakets

dev Netzwerkgerät auf dem das Paket eingegangen ist

dst weiterer Weg des Paketes durch den Netzwerk Stack

head und end zeigen auf Anfang bzw. Ende des Speicherbereichs, der für das Paket reserviert ist (Achtung: dieser Bereich ist in der Regel größer als eigentlich notwendig weil nicht immer von vorne herein klar ist wie groß das Paket schlussendlich sein wird)

data und tail zeigen auf Anfang bzw. Ende des Protokoll Datenbereiches

mac zeigt auf den Beginn des MAC Headers

nh zeigt auf den Beginn des Headers der Vermittlungsschicht (z.B. IP)

h zeigt auf den Beginn des Headers der Transportschicht (z.B. TCP)

Wie aus Listing 16 ersichtlich handelt es sich bei `h`, `nh` und `mac` jeweils um Unions, wodurch sie je nach dem tatsächlichen Protokoll (z.B. TCP, UDP, ICMP usw. in der Transportschicht) behandelt werden können ohne die Allgemeinheit des Codes zu beschneiden.

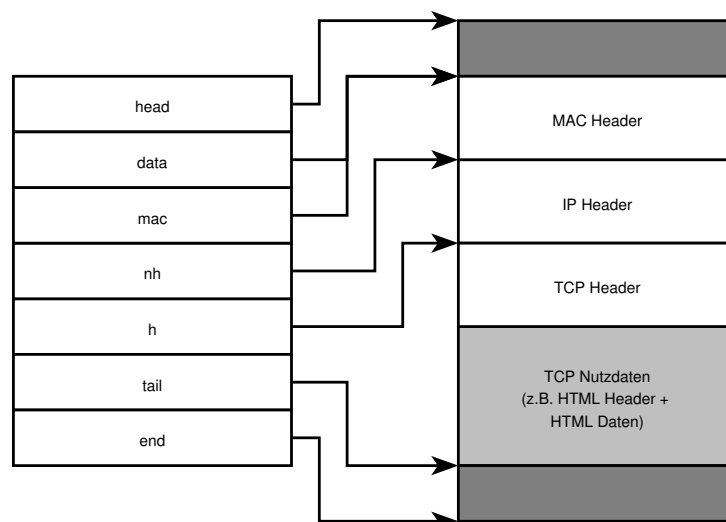
Abbildung 14: Zusammenhang `sk_buff` Datenstruktur mit einem Netzwerkpaket

Abbildung 14 verdeutlicht nochmals die Zusammenhänge zwischen der gerade beschriebenen `sk_buff` Datenstruktur und einem Netzwerkpaket. Beim Versenden eines TCP Pakets wird zuerst Speicher im

Kern reserviert. `head` und `end` zeigen auf Anfang bzw. Ende dieses Speicherbereiches. Danach werden TCP Header und Daten in diesen Speicherbereich kopiert. `data` und `tail` zeigen auf den Anfang des TCP Headers bzw. das Ende der TCP Nutzdaten. Im nächsten Schritt wird das Paket an die IP Schicht weitergereicht. Da der Speicherbereich zu Beginn extra größer angelegt wurde, kann der IP Header nun einfach über dem TCP Header einkopiert werden. Lediglich der `data` Zeiger der `sk_buff` Datenstruktur muss nun umgesetzt werden, sodass er auf den Beginn des IP Headers zeigt. TCP Header und TCP Daten bilden nun die zum IP Header gehörenden Nutzdaten. Auf diese Weise kann ein und dieselbe `sk_buff` Datenstruktur verwendet werden, um Daten zwischen den Netzwerkschichten auszutauschen. Eingehende Pakete durchlaufen dieselbe Prozedur, nur in umgekehrter Reihenfolge.

Abbildung 15 verdeutlicht die gerade beschriebenen Zusammenhänge.

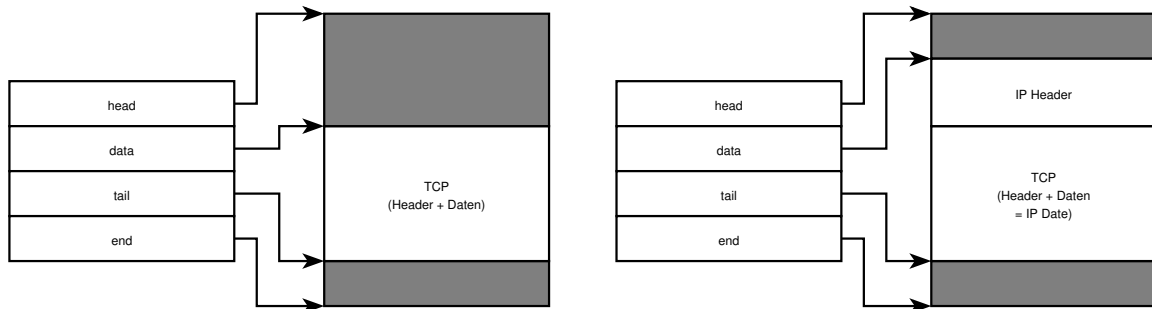


Abbildung 15: Aufbauen eines ausgehenden TCP Pakets und Zusammenhang mit `sk_buff` Datenstruktur: Übergang TCP zu IP Schicht

9.2 Datenübertragungsschicht

Netzwerkarten werden mit der Datenstruktur `struct net_device`²³ repräsentiert. Hier werden neben allgemeinen Informationen, wie dem symbolischen Namen der Karte (z.B. `eth0`), die folgenden Eigenschaften festgehalten:

mtu (maximum transfer unit) bestimmt maximale übertragbare Framelänge - macht unter Umständen eine Fragmentierung der Pakete in der Vermittlungsschicht notwendig

type Hardwaretyp des Geräts (802.2 Ethernet, AppleTalk, Loopback etc.)

dev_addr Hardware Adresse des Geräts (z.B. MAC Adresse)

weilers: protokollspezifische Daten (IPv4, IPv6, AppleTalk)

Die Mehrzahl der Elemente von `struct net_device` sind jedoch Funktionszeiger, die typische Funktionen von Netzwerkkarten zur Verfügung stellen. Durch diesen Mechanismus ist gewährleistet, dass das Interface mit dem der Kernel auf Netzwerkhardware zugreift stets dasselbe ist, auch wenn die einzelnen Treiber je nach den Gegebenheiten der Hardware unterschiedlich ausgeführt sind. Zu diesen Interfacefunktionen zählen unter anderem:

open, stop (De-)Initialisierung der Hardware; Belegung und Freigabe von Systemressourcen

hard_stat_xmit dient zum Verschicken von fertig zusammengestellten Paketen aus der Warteschlange

get_stats dient zum Abfragen von Statistikdaten (z.B. über `ifconfig`)

tx_timeout dient der Problemlösung wenn der Versand eines Paketes fehlgeschlagen ist

²³ Datei: `include/linux/netdevice.h`

Beim Eintreffen von Paketen lösen moderne Netzwerkkarten einen Interrupt aus, der in Folge von einem Interrupt Handler verarbeitet wird. Dieser ist Teil des Gerätetreibers der Netzwerkkarte. Da solche Interrupts zu beliebigen, nicht vorhersehbaren Zeitpunkten auftreten gilt, dass die Interrupt Handler stets so kurz wie möglich sein müssen und nur die notwendigsten Arbeiten ausführen sollen, um das System nur so kurz wie möglich von seinen aktuellen Tätigkeiten abzuhalten.

Der Empfang eines Paketes unterteilt sich in folgende Abschnitte:

net_interrupt eigentlicher Interrupt Handler; prüft ob der Interrupt durch ein neu eingetroffenes Paket ausgelöst wurde; wenn ja wird **net_rx** aufgerufen

net_rx legt eine neue Instanz von **sk_buff** an und kopiert Daten von der Netzwerkkarte in die neue **sk_buff** Instanz

netif_rx letzte Funktion die im Interrupt Kontext ausgeführt wird; sie reiht das empfangene Paket in eine prozessorspezifische Warteschlange ein; allgemeine Kernelfunktion (nicht mehr treiberspezifisch)

In diesen auf der Datenstruktur **struct softnet_data** basierenden Warteschlangen werden alle Netzwerkpakete gespeichert, die mit Hilfe eines Interrupt Handlers empfangen, aber noch nicht abgearbeitet worden sind.

Über den *SoftIRQ* Mechanismus des Kernels wird die Funktion **net_rx_action** ausgeführt, die die nachfolgenden Abläufe anstößt:

- eine Instanz von **sk_buff** aus der Warteschlange entnehmen
- Typ (Protokoll) des Pakets analysieren
- Anhand des Typs wird die passende Handlerfunktion der Vermittlungsschicht ausgewählt und aufgerufen. Dieser Aufruf stellt den Übergang zwischen Datenübertragungsschicht und Vermittlungsschicht dar. Die möglichen Handlerfunktionen der Vermittlungsschicht sind in einer Hashtable abgelegt, womit eine Erweiterung um zusätzliche Protokolle einfach möglich ist.

Diese Tätigkeiten könnten, vom logischen Zusammenhang her, auch direkt im Interrupt Handler erledigt werden. Allerdings sollte die Zeit die für die Interruptbehandlung aufgewendet wird, wie bereits angesprochen, immer so gering wie möglich gehalten werden. Aus diesem Grund werden im Interrupt Handler nur die allernotwendigsten Tätigkeiten durchgeführt und der Rest der Arbeit „auf später“ verschoben. Der *SoftIRQ Mechanismus* ist eine Möglichkeit der sich der Kernel bedient, um Tätigkeiten verzögert auszuführen. Neben *SoftIRQs* gibt es noch eine Reihe weiterer Techniken, die allesamt unter dem Oberbegriff *Bottom Halves* zusammengefasst werden.

9.3 Vermittlungsschicht

Neben dem Versenden und Empfangen von Daten ist diese Schicht auch für das Routing von Paketen verantwortlich. Eine weitere wichtige Aufgabe besteht in der (De-)Fragmentierung von Paketen.

Abbildung 16 gibt einen Überblick über den Aufbau der IPv4 Vermittlungsschicht des Kernels. In den folgenden Abschnitten sollen die Pfade, die die Pakete innerhalb dieser Schicht durchlaufen, näher beschrieben werden. Wie bereits weiter oben festgestellt, wollen wir uns bei der Betrachtung der Vermittlungsschicht auf das IP Protokoll beschränken.

9.3.1 Eingehende Pakete

Empfangene Pakete werden von der Datenübertragungsschicht durch Aufruf von **ip_rcv** an die Vermittlungsschicht weitergereicht. Im Zuge dieses Aufrufs werden die Zeiger der **sk_buff** Datenstruktur, wie in Abschnitt 9.1 beschrieben, aktualisiert.

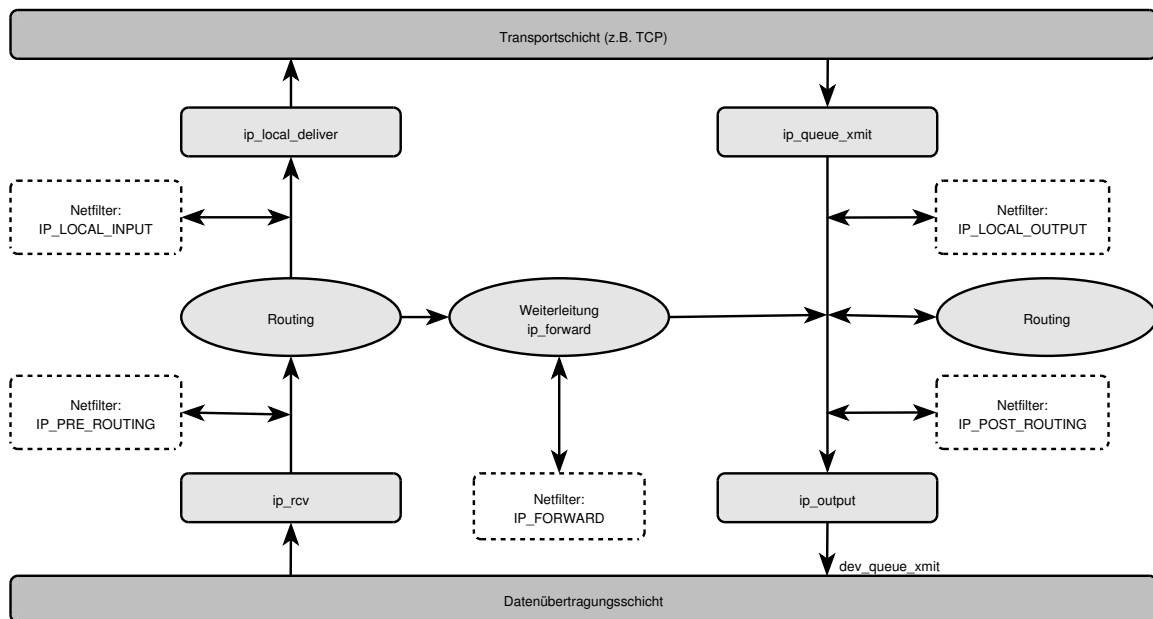


Abbildung 16: Struktur der IPv4 Vermittlungsschicht

Nach einer Reihe von Tests (Checksummen, minimale Paketgröße usw.) prüft der Kernel, ob für den Netfilter Hook `IP_PRE_ROUTING` zusätzliche Routinen registriert worden sind. Über diese Hooks besteht die Möglichkeit, aus dem Userspace in die Verarbeitung von Netzwerkpaketen einzugreifen. Sind für den Hook entsprechenden Routinen registriert worden, so werden diese nacheinander abgearbeitet. Im Anschluss läuft die Abarbeitung des Pakets im Kernel weiter. Hier wird nun entschieden, ob das Paket für das lokale System bestimmt ist oder aber an einen anderen Rechner weitergeleitet werden muss. Diese beiden Fälle werden in den Abschnitten 9.3.2 und 9.3.4 genauer betrachtet.

9.3.2 Lokale Zustellung von Paketen

Im vorangegangenen Schritt wurde festgestellt, dass das Paket für das lokale System bestimmt ist und dementsprechend die Funktion `ip_local_deliver` aufgerufen. Hauptaufgabe dieser Funktion ist es, das Paket an die darüber liegende Transportschicht (z.B. TCP oder UDP) weiterzuleiten.

Bevor das allerdings passieren kann, muss geprüft werden, ob das Paket vollständig vorhanden ist, d.h. nicht fragmentiert vorliegt. Für das Defragmentieren von Paketen dient die Funktion `ip_defrag`²⁴. Für die Verwaltung von IP Fragmenten verwendet der Kernel einen eigenen Cache, in dem die einzelnen Fragmente in Warteschlangen gesammelt werden. Die Funktion `ip_find` wird verwendet um festzustellen, ob es schon eine Warteschlange mit verwandten Fragmenten gibt (die Identifikation von zusammengehörigen Fragmenten erfolgt über ein Hashingverfahren bei dem Ziel und Quelladresse sowie Protokolltyp und Fragment ID berücksichtigt werden). Wird keine passende Warteschlange gefunden, so wird eine neue erzeugt. Das eingegangene Fragment wird dann in die neue bzw. schon bestehenden Warteschlangen eingefügt. Nur wenn alle Fragmente verfügbar sind, wird die Verarbeitung fortgesetzt. Mit der Funktion `ip_frag_reasm` wird aus den Fragmenten eine neue `sk_buff` Instanz erzeugt.

Bevor die Übergabe des fertig zusammengesetzten Pakets erfolgt, kommt noch einmal das Netfilter Framework zum Zug (Hook `IP_LOCAL_INPUT`). Vor der endgültigen Übergabe der Daten an die Transportschicht muss noch ermittelt werden, um welches Transportprotokoll es sich handelt. Zu diesem Zweck ist für jedes auf IP aufsetzende Protokoll eine Instanz von `struct inet_protocol` vorhanden, die einen Funktionszeiger auf die Verarbeitungsroutine in der Transportschicht enthält (z.B. `tcp_v4_rcv` im Fall von TCP).

²⁴ Datei: `net/ipv4/ip_fragment.c`

Diese Funktion wird, nach vorheriger Aktualisierung der Zeiger der `sk_buff` Instanz (Entfernung des IP Headers), aufgerufen.

9.3.3 Routing

Bevor das Forwarding und das Versenden von Paketen besprochen wird, sollen die Routingmechanismen kurz vorgestellt werden

Dabei müssen zwei grundlegende Fälle unterschieden werden:

- Das Paket ist für einen Rechner bestimmt zu dem das lokale System eine direkte Verbindung hat. In diesem Fall reduziert sich das Routing auf das Auffinden jener lokalen Netzwerkkarte über die das Paket versandt werden soll.
- Das Paket ist für einen Rechner bestimmt zu dem es nur über Gateways eine Verbindung gibt.

Die für das Routing notwendigen Einträge werden in mehreren Tabellen verwaltet. Für die grundlegende Funktionalität sind lediglich zwei Tabellen notwendig:

local: Diese Tabelle wird vom Kernel verwaltet und enthält einen Eintrag für jede lokale IP-Adresse.

main: Hier sind alle Routing-Einträge vorhanden, die mit dem Befehl `route` vorgenommen werden. Der Kernel fügt zusätzlich automatisch neue Einträge hinzu wenn neue Netzwerkgeräte aktiviert werden.

Falls beim Übersetzen des Kernels die Option `CONFIG_IP_MULTIPLE_TABLES` aktiviert wurde, so stehen weitere 253 Routing-Tabellen zur Verfügung. Diese bieten zusätzliche Flexibilität und ermöglichen weitere Dienste wie zum Beispiel *Policy Routing*²⁵.

Bei den folgenden Beschreibungen wird davon ausgegangen, dass die zuvor erwähnte Option *nicht* aktiviert ist, um den Blick auf das Wesentliche zu beschränken.

Eine Routing-Tabelle wird durch eine Instanz der Struktur `fib_table`²⁶ (*fib* ist die Abkürzung für Forward Information Base) repräsentiert. Hier sind neben einem Feld für eine eindeutige Kennung eine Reihe von Funktionspointern zur Manipulation der Routing-Tabelle und ein Pointer auf die eigentlichen Einträge der Tabelle enthalten.

Die einzelnen Routing-Einträge werden in Zonen zusammengefasst, wobei eine Zone jene Zieladressen enthält, die den gleichen Längenpräfix²⁷ aufweisen. Demnach werden 32 Zonen unterschieden. Die zugehörige Datenstruktur lautet `fn_zone` und ist in der Datei `net/ipv4/fib_hash.c` definiert. Für die einzelnen Einträge werden Instanzen der Struktur `fib_node` verwendet, welche unter anderem einen Pointer auf eine `fib_info`-Instanz hält. Letztere enthält Informationen die von vielen Routern verteilt werden.

Bereits aus dieser Beschreibung der verwendeten Datenstrukturen ist erkennbar, dass es mit sehr viel Aufwand verbunden sein kann, aus dieser Routing-Tabelle den zugehörigen Eintrag zu finden. Deshalb verwendet der Kernel einen Cache um auf die zuletzt benutzten Routing-Einträge schnell zugreifen zu können. Dieser Cache wurde als Hash-Tabelle mit Überlaufliste realisiert. Als Key für diese Tabelle wird ein Hash über Quelladresse, Zieladresse und Netzwerkdevice verwendet.

Zur Ermittlung der Route von eingehenden Paketen ist die Funktion `ip_route_input`²⁸ zuständig. Diese versucht zuerst eine passende Route in dem zuvor beschriebenen Cache zu finden. Kann hier kein passender Eintrag gefunden werden, so muss die gesamte Routing-Tabelle nach der zu verwendenden Zieladresse durchsucht werden. Diese Aufgabe wird an `ip_route_input_slow` delegiert. Nach einer Reihe von Gültigkeitsprüfungen betreffend die Quell- und Zieladresse wird mit Hilfe der Funktion `fib_lookup`

²⁵Nähere Details dazu: <http://linux-ip.net/>

²⁶Datei: `include/net/ip_fib.h`

²⁷Die IP-Adresse eines Hosts kann in einen Netzwerk- und einen Host-Teil aufgespaltet werden. So beschreibt die Adresse `192.168.0.15/24` einen Host im Subnetz `192.166.0.*`. Der Längenpräfix wäre in diesem Fall `24`.

²⁸Datei: `net/ipv4/route.c`

eine passende Route gesucht. Dazu werden zunächst die Zonen der *local*-Tabelle und anschließend jene der *main*-Tabelle durchsucht. Über die Funktion `fib_semantic_match`²⁹ wird geprüft, ob eine Route für die gewünschte Zieladresse gültig ist.

Für ausgehende Pakete wird die Route mit Hilfe der Funktion `ip_route_output_flow` ermittelt. Auch diese durchsucht zuerst die Hash-Tabelle nach einer passenden Route. Ist im Cache kein Eintrag vorhanden, so wird die Arbeit an `ip_route_output_slow` delegiert, welche in der Routing-Tabelle nach einem passenden Eintrag sucht.

Konnte eine passende Route gefunden werden (sowohl für eingehende als auch ausgehende Pakete), so wird der weitere Weg des Paketes im Feld `dst` der `sk_buff`-Struktur hinterlegt.

9.3.4 Forwarding von Paketen

Wenn das Paket nicht für das lokale System bestimmt ist wird die Funktion `ip_forward` aufgerufen.

Zuerst wird das *TTL* (Time to Live) Feld des Pakets geprüft. Ist die TTL kleiner oder gleich 1 so wird das Paket verworfen und eine *ICMP_TIME_EXCEEDED* Nachricht an den Absender geschickt. Im Anschluss wird mit der Funktion `ip_decrease_ttl` die TTL dekrementiert und die Prüfsumme neu berechnet. An dieser Stelle kommt wieder das Netfilter Framework mit dem *IP_FORWARD* Hook an die Reihe. Im Anschluss wird das Paket dann über die ermittelte Route versandt, was im folgenden Abschnitt näher erläutert wird.

9.3.5 Versenden von Paketen

Aus Sicht der Transportschicht stellt die Funktion `ip_queue_xmit` die wichtigste Funktion dar, um Pakete an die Vermittlungsschicht weiterzureichen.

Ist in der `sk_buff`-Struktur noch keine Route eingetragen, so wird zunächst die Funktion `ip_route_output_flow` aufgerufen, um den passenden Weg für das ausgehende Paket zu senden. Diese Information wird wie in Abschnitt 9.3.3 in der `sk_buff`-Struktur abgelegt. Damit muss für einen Socket nur einmal eine Route ermittelt werden. Alle weiteren Daten verlassen das System über die selbe Route.

Im nächsten Schritt erfolgt mit der Funktion `ip_send_check` die Berechnung der Prüfsumme. Nachdem etwaige Netfilter-Module über den Hook *IP_LOCAL_OUT* ihre Arbeit erledigt haben wird das Paket über die Funktion `skb->dst->output` (`ip_output` im Fall von IPv4), die als Ergebnis des Routing gesetzt worden ist, verschickt. Über diese Funktion verlassen auch die weiter oben angesprochenen weitergeleiteten Pakete das System.

In Abhängigkeit von der MTU der verwendeten Netzwerkhardware muss nun das Paket möglicherweise fragmentiert werden. Diese Aufgabe wird von der Funktion `ip_fragment`³⁰ übernommen. Dabei wird das ausgehende Paket in mehrere Pakete aufgeteilt, die zur MTU der Netzwerkhardware passen. Für jedes dieser Pakete wird eine eigene Instanz von `struct sk_buff` erzeugt, die den selben IP Header bekommt wie das ursprüngliche Paket. Dieser Header wird in Folge so modifiziert, dass Fragment-ID, Fragment Offset und das „More Fragments“ Flag (mit Ausnahme des letzten Fragments) entsprechend gesetzt werden. Zu guter Letzt können die nun einzelnen Pakete – nach der Berechnung der IP Prüfsumme – über die Funktion `ip_output` verschickt werden.

Bevor die Pakete endgültig an die Datenübertragungsschicht weitergegeben werden, bekommt das Netfilter Framework ein letztes Mal die Gelegenheit, die Daten über den Hook *IP_POST_ROUTING* zu manipulieren.

9.3.6 Netfilter Framework

In den vorangegangenen Abschnitten ist das Netfilter Framework schon mehrfach erwähnt worden. Es bietet eine einfache Möglichkeit zum Filtern von Paketen anhand von vorgegebenen Kriterien. Neben dieser

²⁹Datei: `net/ipv4/fib_semantics.c`

³⁰Datei: `net/ipv4/ip_output.c`

Paketfilterung besitzt es weiters die Möglichkeit, NAT (Network Adress Translation) durchzuführen. Damit ist es möglich Quelle bzw. Zieladressen von Paketen umzuschreiben und so das Teilen von Internetverbindungen zu erlauben.

Neben den schon bekannten Hooks bilden die Module die zweite wesentliche Komponente des Netfilter Frameworks. Dabei handelt es sich um Kernelmodule die zur Laufzeit nachgeladen werden können. Sie stellen jenen Code zur Verfügung der für die einzelnen Hooks registriert und ausgeführt werden kann. In der globalen Datenstruktur `nf_hooks` werden alle registrierten Hook-Funktionen, geordnet nach Protokoll-Kennung und Hook-Kennzahl, gespeichert. Bei den, in den vorangegangenen Abschnitten, angesprochenen Hook-Aufrufspunkten (z.B. `IP_POST_ROUTING`) wird mit dem `NF_HOOK` Makro geprüft, ob einer oder mehrere Hook-Funktionen registriert worden sind. Ist dies der Fall so werden diese, nacheinander und nach ihrer Priorität sortiert, aufgerufen.

9.3.7 IPv6

Strukturell gibt es zwischen der Implementierung von IPv6 und IPv4 im Linux Kernel, wie Abbildung 17 zeigt, keinen großen Unterschied.

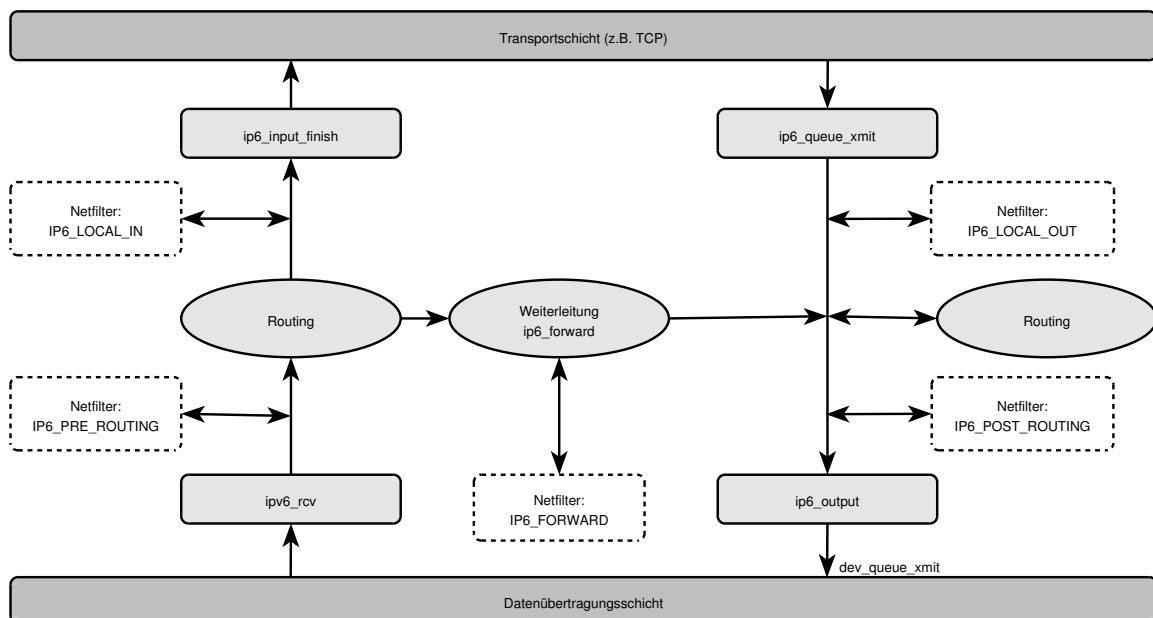


Abbildung 17: Struktur der IPv6 Implementierung

9.4 Transportschicht

In der Transportschicht werden, aufbauend auf der gerade beschriebenen IP-Vermittlungsschicht, üblicherweise zwei verschiedene Protokolle eingesetzt:

TCP *Transmission Control Protocol*, verbindungsorientiert, Verbindungsaufbau über Drei-Wege-Handshake, Reihenfolge der Pakete ist garantiert, wiederholte Übertragung von verloren gegangenen Pakete

UDP *User Datagram Protocol*, paketorientiert, kein expliziter Verbindungsaufbau, Pakete können verloren gehen, Empfangsreihenfolge entspricht nicht zwangsläufig Sendereihenfolge

9.4.1 Übergang von Vermittlungs- zu Transportschicht

Eingehende Daten aus der oben beschriebenen Vermittlungsschicht werden, im Fall von IPv4 und TCP, über die Funktion `tcp_v4_rcv` an die Transportschicht weitergegeben (die Aktualisierung der Zeiger in `sk_buff` erfolgt auch hier wie schon weiter oben beschrieben). Für die eingehenden Daten wird nun geprüft ob es bereits einen zugehörigen TCP-Socket gibt. Dabei kann es sich um bereits verbundene Sockets handeln, um solche die auf eingehende Verbindungen warten und um Sockets deren Verbindung sich gerade im Aufbau befindet. Für die Zuordnung zu einem dieser Sockets werden die IP Adressen von Client und Server, die beiden Port Nummern sowie die Kennung des Netzwerk Interfaces verwendet.

9.4.2 Verbindungsaufbau - Drei Wege Handshake

Für den Aufbau von TCP Verbindungen wird ein Verfahren verwendet, dass als *Drei-Wege-Handshake* bekannt ist. Wie dieser im Linux Kernel realisiert ist sollen die folgenden Unterabschnitte beschreiben. Dabei wird entsprechend des Ablaufs zwischen der Sicht des Servers und des Clients gewechselt.

Serverseite: lauschender Socket Auf Seiten des Servers gibt es Sockets die sich im Zustand `TCP_LISTEN` befinden und auf eingehende Verbindungen warten. Solche Sockets sind immer an bestimmte IP Adressen und Ports gebunden.

Clientseite: initialer Verbindungsaufbau Der Aufbau einer Netzwerkverbindung zu einem Server erfolgt in der Regel über den Aufruf der `open` Funktion der Standardbibliothek. Dieser wird über den `socketcall` Systemaufruf an den Kernel weitergeleitet wo, wie in Abbildung 18 gezeigt, die Funktion `tcp_v4_connect`³¹ ausgeführt wird.

Zunächst muss die Route für das zu sendende TCP-Paket gesucht werden. Dies erfolgt mit Hilfe der Funktion `ip_route_connect`³² welche als Inline-Funktion definiert ist und in weiterer Folge die Aufgabe an `ip_route_output_flow` delegiert.

Anschließend wird TCP Paket mit gesetztem SYN Flag an die Gegenstelle verschickt und der Zustand des Sockets auf `TCP_SYN_SENT` gesetzt. Mit der Funktion `tcp_reset_xmit_timer`³³ wird ein Timer angelegt der den Versand des Pakets wiederholt wenn für eine bestimmte Zeitspanne keine Bestätigung eingetroffen ist.

Serverseite: eingehende Verbindung (SYN) Die Funktion `tcp_v4_rcv` ermittelt den zugehörigen Socket für die eingehende Daten und ruft mit diesem als Parameter die Funktion `tcp_v4_do_rcv` auf. Die im TCP Modell spezifizieren Zustandsübergänge werden in der Funktion `tcp_rcv_state_process`³⁴ abgebildet die als nächstes ausgeführt wird. Hier wird anhand der im TCP Header gesetzten Flags und des aktuellen Socket Zustandes, geprüft welcher Zustandsübergang ausgeführt werden muss. Im aktuellen Fall hat der Client durch das Setzen des *SYN* Flag seinen Wunsch zum Verbindungsaufbau signalisiert. Befindet sich außerdem der lokale Socket im Zustand `TCP_LISTEN`, so wird die Funktion `tcp_v4_conn_request`³⁵ ausgeführt. Als Resultat wird an die Gegenstelle ein Paket geschickt, dass sowohl das *SYN* als auch das *ACK* Flag gesetzt hat.

Clientseite: Verbindung fertig aufbauen (SYN ACK) Der Client empfängt vom Server ein Paket in dem sowohl *SYN* als auch *ACK* Flag gesetzt sind. Der zur Verbindung gehörende Socket auf Client Seite befindet sich im Zustand `TCP_SYN_SENT`. Die Verarbeitung der eingehenden Daten erfolgt gleich wie zuvor auf Seiten der Servers. In der Funktion `tcp_rcv_state_process` wird, auf Grund des aktuellen Socket

³¹Datei: `net/ipv4/tcp_ipv4.c`

³²Datei: `include/net/route.h`

³³Datei: `net/ipv4/tcp_output.c`

³⁴Datei: `net/ipv4/tcp_input.c`

³⁵Datei: `net/ipv4/tcp_ipv4.c`

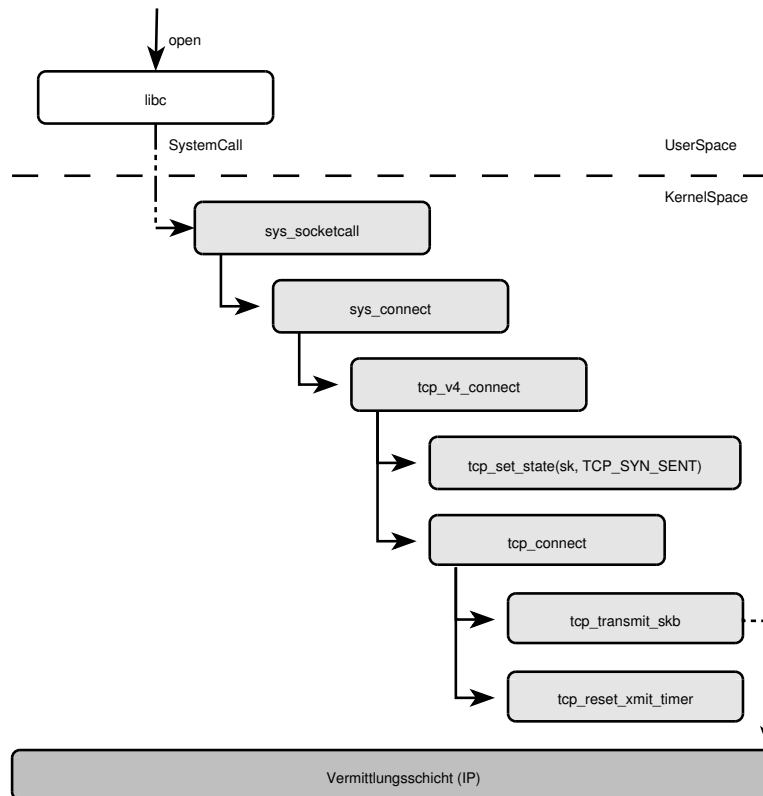


Abbildung 18: Aktives Öffnen einer Netzwerkverbindung

Zustandes, die Funktion `tcp_rcv_synsent_state_process` aufgerufen. Nachdem `SYN` und `ACK` Flag geprüft worden sind, kann der Zustand des Sockets auf `TCP_ESTABLISHED` gesetzt werden. Als Bestätigung wird an die Gegenstelle mittels der Funktion `tcp_send_ack` ein Paket mit gesetztem `ACK` Flag geschickt. Abbildung 19 fasst den Ablauf zusammen.

Serverseite: Verbindung fertig aufbauen (ACK) Der Client empfängt vom Server ein Paket mit gesetztem `ACK` Flag. Der zur Verbindung gehörende Socket auf Server Seite befindet sich im Zustand `TCP_SYN_RECV`. Die eingehenden Daten nehmen den schon bekannten Weg durch die Funktionen `tcp_rcv`, `tcp_v4_rcv` und `tcp_rcv_state_process` wo diesmal keine weitere Funktion aufgerufen wird sondern, wie auch Abbildung 20 zeigt, der Zustand des Sockets direkt auf `TCP_ESTABLISHED` gesetzt wird.

9.4.3 Empfangen von Daten

Eingehende Pakete nehmen den üblichen Weg über `tcp_v4_rcv` und `tcp_v4_do_rcv`. An dieser Stelle wird geprüft, ob es einen zugehörigen Kernelsocket gibt der sich im Zustand `TCP_ESTABLISHED` befindet. Wenn das der Fall ist, wird die Funktion `tcp_rcv_established`³⁶ aufgerufen. Hier werden die Pakete in 2 Klassen unterteilt: Solche die schnell bearbeitet werden können und solche die größeren Aufwand erfordern.

Schnell verarbeitbare Pakete sind solche die:

- nur eine Empfangsbestätigung für gesendete Daten beinhalten.
- als nächstes erwartete Daten enthalten (folgen unmittelbar auf das zuletzt empfangene Segment).
- keines der Flags `SYN`, `URG`, `RST` oder `FIN` gesetzt haben.

³⁶Datei: `net/ipv4/tcp_input.c`

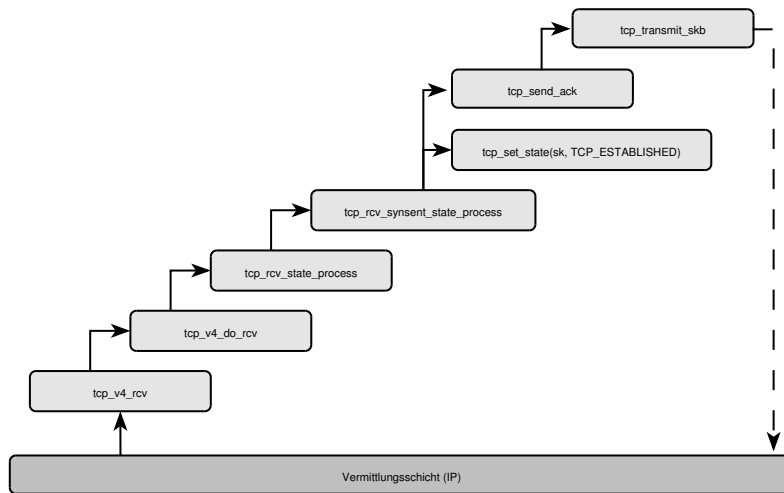
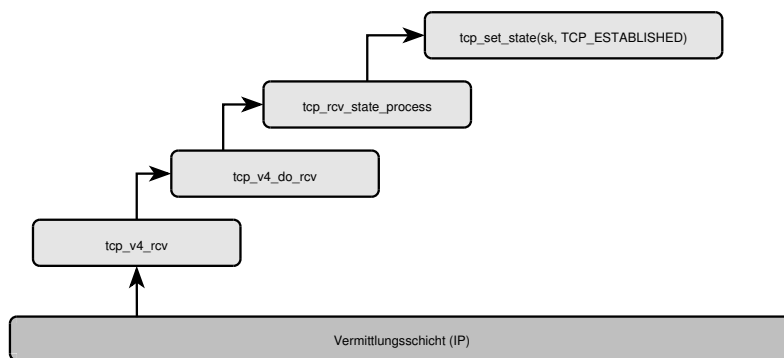
Abbildung 19: Ablauf bei eingehendem *SYN ACK* am ClientAbbildung 20: Ablauf bei eingehendem *ACK* am Server

Abbildung 21 zeigt den Pfad den eingehende Pakete nehmen.

Diese Pakete, die den so genannten *Fast Path* durchlaufen, werden anhand ihrer Länge in Bestätigungspakete (enthalten keine Nutzdaten) und Datenpakete unterteilt. Erstere werden mit der Funktion `tcp_ack` behandelt. Neben der Bestimmung des Empfangsfensters der Gegenstelle hat diese Funktion auch die Aufgabe bestätigte Daten aus der so genannten *Retransmission Queue* zu entfernen. In dieser Datenstruktur werden alle noch nicht bestätigten Pakete gespeichert. Trifft nach einer gewissen Zeit keine Bestätigung der Gegenstelle ein, werden die Pakete automatisch erneut versandt. Der Erhalt von eingehenden Datenpakete wird der Gegenstelle bestätigt, das Empfangene Paket wird in die Warteschlange der `sock`-Struktur eingefügt (siehe Abschnitt 9.5.1) und schließlich der Funktionszeiger `sk->data_ready` aufgerufen. Damit wird dem Benutzerprozess mitgeteilt, dass neue Daten eingetroffen sind.

Im so genannten *Slow Path* werden unter anderem jene Pakete behandelt die in nicht geordneter Reihenfolge eintreffen. Diese Pakete müssen gesammelt werden und können erst wenn der Bereich komplett vorhanden ist nach oben weitergegeben werden.

9.4.4 Versenden von Daten

Das Versenden von Daten beginnt in der TCP Transportschicht mit der Funktion `tcp_sendmsg`. Sollte sich der zugehörige Socket noch nicht im Zustand `TCP_ESTABLISHED` befinden, so wartet der Kernel solange bis das der Fall ist. Im Anschluss werden die Daten aus dem Userspace kopiert und die Zusammenstellung des TCP Pakets wird begonnen. Darüber hinaus müssen eine Reihe von TCP Mechanismen beim Versenden

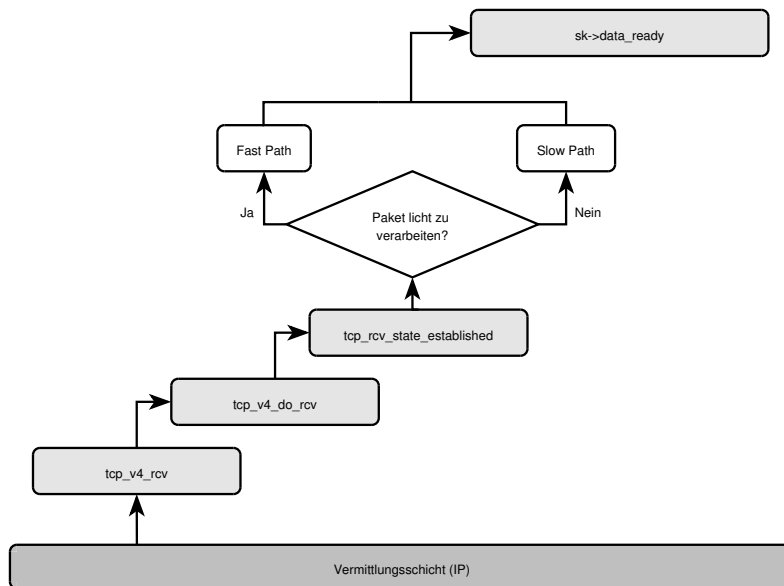


Abbildung 21: Ablauf bei eingehenden Paketen

beachtet werden. Dazu gehören unter anderem das *Sliding Window* Protokoll, das *Slow Start* Verhalten von TCP und das erneute Übertragen von unbestätigten Paketen.

9.4.5 Abbau von Verbindungen

Nicht nur für den Aufbau einer Verbindung ist in TCP ein klares Schema vorgegeben sondern natürlich auch für den Abbau. Bei einer bestehenden Verbindung müssen beide Teilnehmer über einen 3-Wege-Handshake die Verbindung wieder schließen.

Wenn eine Applikation eine Verbindung beenden will, so geschieht das mit dem Aufruf der `close` Funktion der C-Standardbibliothek. Über den Systemcall `socketcall` wird im Kernel die Funktion `tcp_close` aufgerufen. Befindet sich der fragliche Socket im Zustand `TCP_LISTEN`, so kann der Socket direkt geschlossen und der Status auf `TCP_CLOSED` gesetzt werden.

Ansonsten wird mit der Funktion `tcp_send_fin` ein Paket mit gesetztem *FIN* Flag an die Gegenstelle geschickt. Zuvor bereits wird – nicht ganz konform mit dem TCP Standard – der Zustand des Sockets mit der Funktion `tcp_close_state` auf `TCP_FIN_WAIT_1` gesetzt.

Die Gegenstelle befindet sich ebenfalls im Zustand `TCP_ESTABLISHED` und empfängt das *FIN* Paket welches mit einem *ACK* beantwortet wird. Der Zustand des Sockets wird auf `TCP_CLOSING` gesetzt.

Auf der anderen Seite wird auf den Empfang des *ACK* Pakets in der Funktion `tcp_rcv_state_process` der Übergang zu `TCP_FIN_WAIT_2` vollzogen. Das empfangene *ACK* wird bestätigt und anschließend in den Zustand `TIME_WAIT` gewechselt. In diesem Zustand verbleibt die Verbindung noch eine Zeit lang, falls noch 'verirrte' Pakete zu diesem Socket eintreffen bis endgültig in den Zustand `CLOSED` gewechselt wird.

9.5 Anwendungsschicht - Sockets

Sockets sind ein gängiges Konzept um im Userspace mit Netzwerkverbindungen zu arbeiten. Unix-typisch gilt auch hier, dass man bestrebt ist Netzwerkverbindungen so zu behandeln als würde es sich dabei um eine Datei handeln. Für den Übergang zwischen Userspace und Kernelspace dient ein einziger Systemcall, nämlich `socketcall`. Zu jedem von einer Applikation verwendeten Socket im Userspace (bereitgestellt von der C-Standardbibliothek) gibt es im Kernel einen so genannten Kernelsocket die aus den beiden Datenstrukturen `struct socket` und `struct sock` besteht. Erstere beinhaltet dabei jene Informationen

die aus Sicht des Userspace von Interesse sind während Letztere jene Informationen bereithält, die der Kernel benötigt.

9.5.1 Datenstrukturen

Die `socket`-Struktur enthält, wie bereits erwähnt, Daten die für die Anwendung interessant sind. Sie ist wie folgt definiert:

```
1  struct socket {
2      socket_state  state;
3      unsigned long  flags;
4      struct proto_ops  *ops;
5      struct file      *file;
6      struct sock      *sk;
7      short           type;
8      /* ... */
9  };
```

Listing 17: Datenstruktur `socket` (*include/linux/net.h*)

Die einzelnen Felder haben dabei folgende Bedeutung:

state: Gibt den Verbindungszustand des Sockets an. Diese Zustände haben nichts mit den Zuständen des verwendeten Transportprotokolls (TCP) zu tun. Sie repräsentieren vielmehr generelle, für den Anwendungsprogrammierer relevante Zustände. Diese sind zum Beispiel `SS_UNCONNECTED`, `SS_CONNECTED` oder `SS_DISCONNECTING`.

flags:

ops: Ein Socket kann verschiedene Transportprotokolle verwenden. Dieser Pointer zeigt auf eine Datenstruktur welche protokollspezifische Funktionen enthält mit denen der Socket manipuliert werden kann. Diese Funktionen stellen die Schnittstelle zwischen der Anwendung und dem Socket-Layer dar.

file: Enthält einen Pointer auf eine Instanz einer Pseudo-Datei die zur Kommunikation über diesen Socket verwendet wird.

sk: Hält einen Pointer auf den kernelspezifischen Teil der Socket-Struktur

type: Speichert den Typ des Sockets. Diese sind unter anderem `SOCK_STREAM` für verbindungsorientierte Protokolle (TCP), `SOCK_DGRAM` für verbindungslose Protokolle (UDP) oder `SOCK_RAW` für einen Raw-Socket.

Die Definition der `sock`-Struktur welche die Repräsentation eines Sockets für den Netzwerk-Layer darstellt ist sehr umfangreich weshalb hier nur die wesentlichen Felder aufgelistet werden.

```
1  struct sock {
2      /* ... */
3      struct sk_buff_head  sk_receive_queue;
4      struct sk_buff_head  sk_write_queue;
5
6      struct proto      *sk_prot;
7      unsigned short    sk_type;
8
9      void              (*sk_data_ready)(struct sock *sk, int bytes);
10     /* ... */
11 }
```

Listing 18: Datenstruktur `sock` (*include/net/sock.h*)

sk_receive_queue: In dieser Liste werden die empfangenen TCP-Pakete welche durch Instanzen von `sk_buff` repräsentiert werden gespeichert.

sk_write_queue: Ausgehende Pakete – ebenfalls `sk_buff`-Instanzen – werden in dieser Liste gehalten

sk_prot: Dieses Feld hält einen Zeiger auf eine Instanz der Struktur `proto` welches Funktionspointer zur Kommunikation zwischen dem Socket-Layer und dem Transport-Layer enthält.

sk_type: Speichert den Typ der Verbindung analog zu `type` in `struct socket`

sk_data_ready: Stellvertretend für eine Reihe von weiteren Callbacks wurde dieser aufgelistet. Diese Funktionen werden vom Kernel verwendet um Statusänderungen durchzuführen oder auf besondere Ereignisse aufmerksam zu machen.

9.5.2 Verwendung von Sockets

In diesem Abschnitt wird noch ein kurzer Überblick über die Verwendung von Sockets aus Sicht der Anwendung gegeben. Für die Kommunikation zwischen Userspace und Kernel steht wie bereits erwähnt nur ein einziger Systemaufruf namens `socketcall` zur Verfügung. Dieser dient eigentlich nur als Multiplexer für eine Reihe von weiteren Kernelfunktionen. Als erster Parameter wird daher eine Konstante übergeben, welche die gewünschte Funktion identifiziert. Die Definition dieser Konstanten befindet sich in `include/linux/net.h`. Bei den weiteren Ausführungen wird direkt auf die konkrete Funktion verwiesen. Die Implementation dieser Funktionen befindet sich in `net/socket.c`.

Sockets erzeugen: Zuerst muss ein Socket erstellt werden. Dies erfolgt über den Systemaufruf `sys_socket`. Dieser alloziert zunächst den für die Strukturen benötigten Speicher. Danach werden die Datenstrukturen abhängig vom Protokolltyp initialisiert. Abschließend wird noch eine Pseudo-Datei für diesen Socket erzeugt und ein Dateideskriptor alloziert, welcher der Anwendung als Resultat des Systemaufrufs übergeben wird.

Daten empfangen: Zum Empfangen von Daten über einen Socket stehen zwei netzwerkspezifische Funktionsaufrufe zur Verfügung. Diese sind `sys_recv` und `sys_recvfrom`. Der Unterschied zwischen den beiden Aufrufen liegt darin, dass letzterer zusätzlich zu den Daten auch die Adresse der Quelle liefert. Es soll hier nur die Aufgabe von `sys_recvfrom` kurz beschrieben werden, da `sys_recv` die Arbeit mit leicht geänderten Parametern ebenfalls an diese delegiert.

Nachdem es sich aus Sicht der Anwendung um einen Dateideskriptor handelt, können auch die Systemaufrufe `sys_readv` und `sys_read` verwendet werden. Aber auch diese delegieren die Arbeit an `sys_recvfrom`.

Die erste Aufgabe des Systemaufrufs ist es, die zum übergebenen Dateideskriptor passende `socket`-Instanz zu finden. Danach wird über `sock_recvmsg` die protokollspezifische Empfangsroutine `socket->ops->recvmsg` aufgerufen. Dieser Funktionspointer zeigt im Fall von IPv4 (sowohl bei verbindungsorientierten als auch verbindungslosen Protokollen) auf die Funktion `inet_sendmsg`³⁷ welche in weiterer Folge `socket->sk->sk_prot->sendmsg` aufruft. Die Verarbeitung landet schließlich bei `tcp_recvmsg` oder `udp_recvmsg` landet. Diese kopieren dann die empfangenen Daten aus der entsprechenden Warteschlange der `sock`-Struktur in einen Buffer der dann in den Speicher der Anwendung kopiert wird.

Daten versenden: Der Vorgang des Versendens von Paketen erfolgt analog zum zuvor beschriebenen Empfang von Daten. Auch hier stehen zwei netzwerkbezogene Systemaufrufe (`sys_sendto` und `sys_send`) sowie zwei dateisystembezogene Systemaufrufe (`sys_write` und `sys_writew`) zur Verfügung welche letztendlich in der Funktion `sys_sendto` landen. Danach wird ebenfalls die zum übergebenen Dateideskriptor

³⁷Datei: `net/ipv4/af_inet.c`

gehörende `sock`-Instanz ermittelt. Bevor die Daten versendet werden können müssen noch die benötigten Daten aus dem Userspace in den Kernelspace kopiert werden.

Das Versenden der Daten erfolgt über die Funktion `inet_rcvmsg`³⁸ welche als Pointer in `socket->ops->sendmsg` hinterlegt ist. Analog zum Versenden von Daten ruft diese Funktion ebenso `socket->sk->sk_proto->sendmsg` auf. Dieser Pointer zeigt auf die Funktion `tcp_sendmsg` bzw. `udp_sendmsg` welche die Schnittstelle zum TCP-Stack darstellen

Dieser – auf den ersten Blick kompliziert scheinende – Weg über mehrere Funktionspointer ist deshalb notwendig, da Sockets nicht nur für Netzwerk-Kommunikation verwendet werden. Mit diesem Konzept ist es leicht möglich, weitere Kommunikationsstrukturen im Hintergrund zu nutzen.

9.6 Der Weg durch den TCP-Stack im Überblick

Abschließend sollen die Funktionen die bei der Verarbeitung von über das Netzwerk übertragenen Daten verwendet werden in Zusammenhang gebracht werden. In Abbildung 22 sind die einzelnen Schichten, welche durchlaufen werden müssen, sehr vereinfacht dargestellt. Die linke Aufrufhierarchie zeigt dabei die Aufrufhierarchie beim Versenden von Daten, während rechts der Empfang skizziert ist. Neben den Pfeilen sind die Namen der zuständigen Funktionen angeführt. Die Richtung der Pfeile gibt dabei die Aufrufhierarchie an. Der zugehörige Datenfluss ist im Fall des Versendens von Paketen von der Anwendung den gesamten Netzwerk-Stack hinunter zur Datenübertragungsschicht. Jene beim Empfang von Daten ist genau entgegengesetzt – also zur Anwendung.

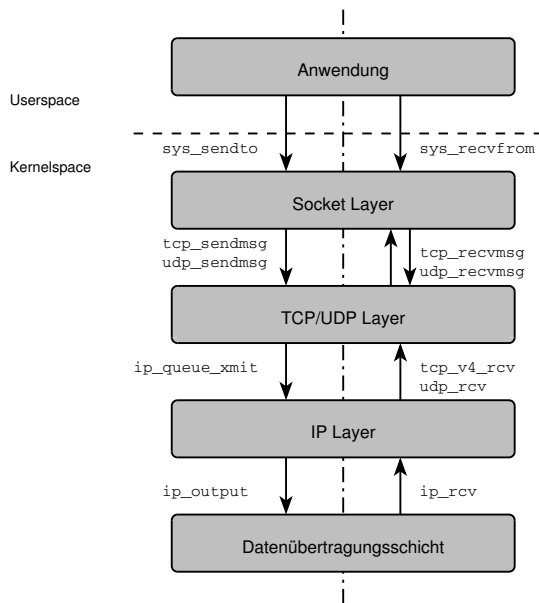


Abbildung 22: Der Weg durch den TCP-Stack im Überblick

Als Datenstruktur wird für die Datenübertragungsschicht, die Vermittlungsschicht und die Transportschicht die im Abschnitt 9.1 vorgestellte `sk_buff`-Struktur verwendet. Dadurch wird vermieden, dass die Daten zwischen den einzelnen Layern kopiert werden. Der Socket-Layer dient nur als Mediator zwischen Anwendung und TCP-Stack.

³⁸Datei: `net/ipv4/af_inet.c`

10 Sicherheitsmechanismen im Kernel

Dieser Abschnitt soll einen Einblick in zwei Sicherheitstechniken bieten, die mit Version 2.6 Einzug in den Linux Kernel gehalten haben. Dabei handelt es sich einerseits um das *Linux Security Modules* Framework, das als Basis für verschiedene Sicherheitskonzepte dienen kann und andererseits um *Security Enhanced Linux*, das auf diesem Framework aufbaut.

10.1 LSM – Linux Security Modules

Bei den *Linux Security Modules* (LSM) handelt es sich um ein Framework um Zugriffskontrollmechanismen in Form von nachladbaren Modulen in den Kernel zu integrieren.

Die Entstehungsgeschichte der LSM ist eng verknüpft mit der Entwicklung von *Security-Enhanced Linux* (SELinux) (siehe auch Abschnitt 10.2) durch den US-Nachrichtendienst NSA³⁹. SELinux wurde zunächst als Kernel Patch realisiert und sollte Eingang in den Entwicklerkernel 2.5 finden. Dieses Vorhaben wurde grundsätzlich von Linus Torvalds begrüßt, jedoch bevorzugte er die Schaffung eines allgemeinen Security-Frameworks, mit dem man nach Belieben Sicherheitsmodule in den Kernel laden kann ohne sich von vorn herein auf eine bestimmte Implementierung wie SELinux festzulegen. Dieses Konzept wurde folglich in Form der LSM in die Tat umgesetzt. Damit wurde eine Möglichkeit geschaffen um verschiedene Sicherheitskonzepte in den Kernel integrieren zu können, ohne die zentralen Bereiche des Kernels an das jeweilige Modell anpassen zu müssen.

10.1.1 Architektur

Die zentrale Aufgabe von LSM ist es, den Zugriff auf interne Kernel-Objekte zu regeln. Die Frage die sich dabei stellt ist, ob ein bestimmtes *Subjekt* (typischer Weise eine Applikation) eine bestimmte Operation auf einem bestimmten Kernel-Objekt ausführen darf. Aus dieser Problemstellung wird klar, dass es notwendig ist, Berechtigungsprüfungen im Kernelcode vor Zugriffen auf Kernel-Objekte einzuführen. Zu diesem Zweck wurden an kritischen Stellen so genannte LSM Hooks eingeführt, die vor dem Zugriff auf ein Objekt eine, für diesen Hook registrierte, Routine des geladenen Security Moduls aufrufen. Diese Routine prüft, wie in Abbildung 23 gezeigt, anhand des aktuellen Kontexts, ob der Zugriff zulässig ist und kann ihn entsprechend unterbinden oder erlauben.

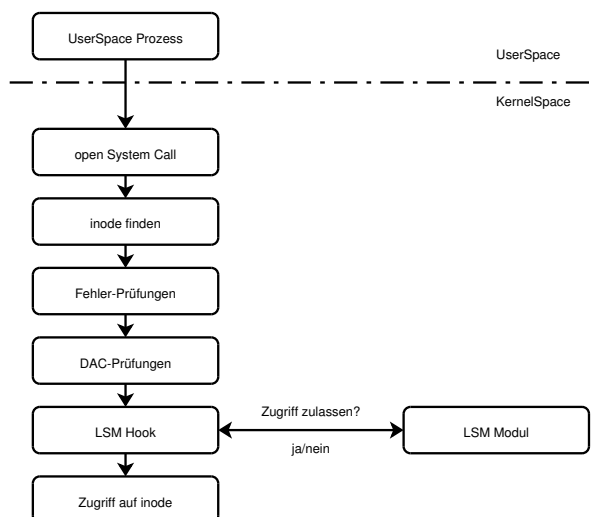


Abbildung 23: Einsatz des LSM Hooks im Rahmen des `open` System Call

³⁹NSA – National Security Agency, Web: <http://www.nsa.gov>

Die Entscheidungen des Kernel selbst wiegen dabei schwerer als die Entscheidungen der LSM Hook Routinen. Das bedeutet, dass mit den Hooks zwar Zugriffe die der Kernel selbst erlauben würde unterbunden werden können nicht aber umgekehrt Zugriffsbeschränkungen des Kernels aufgehoben werden können. Neben diesen so genannten *restrictive Hooks* gibt es nur einige wenige Ausnahmen, die so genannten *permissive Hooks*, die auch negative Entscheidungen des Kernels abändern können.

Benötigt werden solche *permissive Hooks* für die Untermenge des POSIX.1e Standards der vom Kernel unterstützt wird. Dafür ist es notwendig, dass Zugriffe gewährt werden, die der Kernel auf Grund von *Discretionary Access Control* (DAC) Prüfungen (basierend auf Benutzer-IDs usw.) eigentlich unterbinden würde. Der zugehörige Ablauf wird in Abbildung 24 dargestellt.

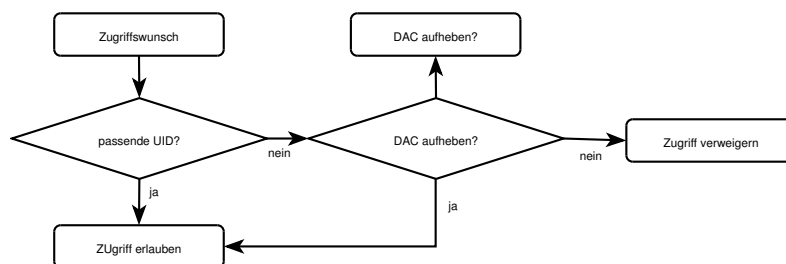


Abbildung 24: permissive Hooks können DAC Entscheidungen aufheben

Die Unterstützung für einen Teil des POSIX.1e Standards war schon vor Version 2.6 im Kernel vorhanden. Mit der Einführung von LSM wurde der Code in ein eigenes *Security Module* ausgelagert.

10.1.2 Security Fields

Zusätzliche zu den bereits angesprochenen Hooks wurde mit den so genannten *Security Fields* eine Möglichkeit geschaffen, sicherheitsrelevante Daten zu bestimmten Kernel-Datenstrukturen hinzuzufügen.

Es handelt sich dabei um `void*` Pointer mit deren Hilfe die Module von ihnen benötigte Informationen direkt in den betroffenen Kernel-Objekten abspeichern können. Die Verwaltung dieser Daten sowie ein möglicherweise notwendiges Locking bleibt dabei den Modulen überlassen. Für die Manipulation der Daten stehen eigene Hooks zur Verfügung, die beim Anlegen und Löschen der Kernel-Objekte sowie bei relevanten Ereignisse aufgerufen werden.

Auch bei den *Security Fields* ist das Verhalten im Zusammenhang mit *Module Stacking* nicht klar definiert. Es ist derzeit kein Mechanismus vorgesehen, wie mehrere Module Daten in den Feldern ablegen können. Es bleibt auch hier den einzelnen Modulen überlassen, geeignete Mechanismen (beispielsweise die Verwendung von verketteten Listen, Hashtables usw.) zu verwenden.

10.1.3 Hooks

Die Hook Funktionen, die ein *Security Module* bereitstellt, werden in einer globalen Instanz der Datenstruktur `struct security_operations`⁴⁰ namens `security_ops` registriert.

`struct security_operations` beinhaltet für jeden Security Hook einen Funktions-Pointer. Die hier hinterlegte Funktion wird vom Hook aufgerufen und entscheidet darüber, ob der Zugriff auf das Kernel Objekt ausgeführt werden darf. Listing 19 zeigt einen Ausschnitt aus der Datei `include/linux/security.h`. Der in Zeile 16 exemplarisch gezeigte `task_setnice` Funktions-Pointer wird aufgerufen wenn der `nice` Wert eines Prozesses geändert werden soll. Das LSM Framework ermöglicht so eine Prüfung, ob eine Änderung der Prozesspriorität zugelassen werden soll.

1

⁴⁰Datei: `include/linux/security.h`

```
2  /* ... */
3
4  * @task_setnice:
5  * Check permission before setting the nice value of @p to @nice.
6  * @p contains the task_struct of process.
7  * @nice contains the new nice value.
8  * Return 0 if permission is granted.
9
10 /* ... */
11
12 struct security_operations {
13
14     /* ... */
15
16     int (*task_setnice) (struct task_struct * p, int nice);
17
18     /* ... */
19 };
20
21 /* global variables */
22 extern struct security_operations *security_ops;
23
24 /* ... */
25
26 static inline int security_task_setnice (struct task_struct *p, int nice)
27 {
28     return security_ops->task_setnice (p, nice);
29 }
```

Listing 19: Datenstruktur `security_operations` mit `task_setnice` Funktions-Pointer und zugehöriger `security_task_setnice` Funktion *include/linux/security.h*

Listing 20 zeigt in Zeile 15 einen Hook im System Call `sys_nice` der die aus Listing 19 bekannte Funktion `security_task_setnice` aufruft um zu prüfen, ob die Priorität des aktuellen Prozesses geändert werden darf.

```
1
2  /*
3  * sys_nice - change the priority of the current process.
4  * @increment: priority increment
5  *
6  * sys_setpriority is a more generic, but much slower function that
7  * does similar things.
8  */
9  asmlinkage long sys_nice(int increment)
10 {
11     int retval;
12
13     /* ... */
14
15     retval = security_task_setnice(current, nice);
16     if (retval)
17         return retval;
18
19     set_user_nice(current, nice);
20     return 0;
21 }
```

Listing 20: System Call `sys_nice` mit LSM Hook

Nach dem gerade gezeigten Schema wurden Security Hooks in die verschiedensten Subsysteme des Kernel eingebaut. Die Datenstruktur `struct security_operations` umfasst etwa 130 Funktions-Pointer für Hook-Funktionen. Die wichtigsten Bereiche, die mit LSM Hooks versehen worden sind, sind die Folgenden:

Prozess Hooks: bieten Kontrolle über Prozess-relevante Operationen (z.B. kill, setuid, nice)

Hooks beim Laden eines Programms: bieten Kontrolle im Rahmen der verschiedenen exec Aufrufe

IPC Hooks: bieten Kontrollmöglichkeiten im Rahmen der Interprocess Communication

Dateisystem Hooks: bieten Kontrolle bei Dateisystem Operationen wie read oder write

Netzwerk Hooks: bieten, unter anderem, die Möglichkeit eingehende Pakete bzgl. ihrer Zielapplikation zu prüfen noch bevor diese über den User Space Socket an die Anwendung weitergereicht werden

10.1.4 Registrierung von eigenen *Security Modules*

Ein *Security Module*, das in den Kernel geladen wird, muss sich selbst beim LSM Framework registrieren. Zu diesem Zweck kommt die Funktion `register_security` aus `security/security.c` zum Einsatz. Damit wird die globale `security_ops` Tabelle mit den vom Modul angegebenen Funktions-Pointern befüllt. Die Funktion kann allerdings nur dann verwendet werden, wenn zuvor noch kein anderes *Security Module* in den Kernel geladen worden ist.

Beim Entladen des Moduls muss die Funktion `unregister_security` aufgerufen werden. Damit wird die `security_ops` Tabelle wieder auf Standardwerte zurück gesetzt.

10.1.5 Kombination von Modulen – Stacking

Das LSM Framework bietet grundsätzlich die Möglichkeit mehr als ein Security Modul in den Kernel zu laden. Dieser Mechanismus wird als *Module Stacking* bezeichnet. Allerdings ist es Aufgabe der Module dieses Konzept umzusetzen. Zu diesem Zweck gibt es zwei spezielle LSM Hooks namens `register_security` und `unregister_security` (nicht zu verwechseln mit den beiden Funktionen aus Abschnitt 10.1.4). Ein *Security Module* kann eigene Funktionen für diese beiden Hooks registrieren. Die erste davon wird aufgerufen, wenn sich ein Modul mit der Funktion `mod_reg_security` beim LSM Framework registrieren will, die zweite wenn sich ein Modul mit `mod_unreg_security` beim LSM abmeldet.

Somit ist es die Aufgabe der Module, die sich weiter vorne in der Kette befinden, entsprechende Hook Funktionen zur Verfügung zu stellen. Wird eine Hook Funktion des ersten Moduls aufgerufen, so ist es seine Aufgabe, die gleiche Hookfunktion des Moduls das als sein Nachfolger registriert ist, aufzurufen und deren Rückgabewert in passender Art mit seinem eigenen zu kombinieren. Dieser Ansatz vertraut also auf die Kooperation zwischen den Modulen.

Ein Modul, das sich weiter vorne in der Kette befindet, kann also nachfolgende Module und deren Entscheidungen ignorieren bzw. nachfolgende Module gar nicht erst zulassen. Dieser *Module Stacking* Mechanismus wird möglicher Weise in zukünftigen LSM Versionen noch weiter ausgebaut werden. Aus Sicht des LSM Frameworks ist stets nur ein *Security Modul* bekannt. Eine Kombination von verschiedenen Modulen und die logischen Abläufe in diesem Zusammenhang bleiben den einzelnen Modulen selbst überlassen.

10.1.6 Performance und Einsatzbereiche

Ein wichtiger Aspekt für die Akzeptanz der *Linux Security Modules* im Standardkernel war, dass das Framework für sich alleine – ohne geladenen Module – einen möglichst geringen Einfluss auf die Performance des Systems hat. Messungen der Entwickler [LSM, Usenix 2002] weisen einen typischen Performance Overhead von 1 bis 2 Prozent aus. Lediglich im Bereich der Netfilter LSM Hooks wurde ein Overhead von 5 bis 7 Prozent gemessen.

Auch wenn das LSM Framework noch recht jung ist und erst mit Version 2.6 erstmals offiziell Teil des Linux Kernels ist, gibt es schon einige Projekte die von dieser Infrastruktur Gebrauch machen. Das

prominenteste ist mit Sicherheit *SELinux* (Abschnitt 10.2). Weitere Module sind beispielsweise *DTE Linux*⁴¹ oder *Openwall*⁴².

10.2 SELinux – Security Enhanced Linux

Security Enhanced Linux wurde ursprünglich vom US-Nachrichtendienst NSA als Patch für den Linux Kernel implementiert. Nach der Einführung des LSM Frameworks wurde *SELinux* darauf angepasst und hat so Einzug in Linux 2.6 gehalten.

10.2.1 Ziele

Das bekannte Sicherheitskonzept von Linux baut auf dem *Discretionary Access Control* (DAC) Konzept auf. Dabei kann man grob zwei Gruppen von Benutzern unterscheiden: Jene, die volles Vertrauen genießen und alle Freiheiten besitzen (üblicherweise Benutzer mit *root* Rechten). Auf der anderen Seite stehen jene Benutzer, die nur vergleichsweise eingeschränkte Rechte besitzen. Besonders augenscheinlich wird dieses Ungleichgewicht wenn es zum Beispiel darum geht, einen System-Dienst zu betreiben. Dieser muss typischerweise auch auf Ressourcen zugreifen, die für normale Benutzer nicht zugänglich sind. Aus diesem Grund kommt es häufig vor, dass solche Dienste mit weit mehr Rechten ausgeführt werden (*root*) als tatsächlich benötigt werden. Und das passiert deshalb, weil es, im Rahmen des angesprochenen „alles oder nichts“ Prinzips keine geeigneten Möglichkeiten gibt, einem Prozess nur genau jene Rechte zu geben, die er auch tatsächlich benötigt. Weist ein solcher Dienst nun Sicherheitsmängel auf (z.B. Buffer-Overflows) die von einem Angreifer ausgenutzt werden können, so erhält dieser die Möglichkeit vollen Zugriff auf das System zu erhalten.

Ein weiteres Problem von DAC ist, dass es einem Benutzer vollkommen freigestellt ist, wie er die Zugriffsrechte für seine Dateien setzt. Das System kann eine bestimmte Vergabe von Rechten nicht erzwingen. Hier setzt *Mandatory Access Control* MAC im Unterschied zu DAC an und erlaubt, dass das System selbst die Kontrolle über das Recht hat und auch durchsetzt, ohne dass der Benutzer die Möglichkeit hat, dieses Verhalten zu umgehen. Das System stützt sich bei seinen Entscheidungen auf eine Reihe von vorab festgelegten Regeln, auch als *Policy* bezeichnet. Hinzu kommt noch die Möglichkeit, die Rechte die ein Prozess (auch als Subjekt bezeichnet) im Umgang mit einem Objekt (z.B. einer Datei) bekommt viel feiner abzustufen als das mit den bekannten DAC Mechanismen möglich ist. Dadurch kann man jedem Prozess genau jene Rechte einräumen, die er auch tatsächlich benötigt. Durch die Verankerung der Zugriffskontrollmechanismen im Kernel ist außerdem sichergestellt, dass diese auf Ebenen des User Space nicht umgangen werden können.

10.2.2 Architektur

Das Konzept von SELinux baut auf der so genannten *Flask Security Architectur* [Flask 1999] auf die für das experimentelle Fluke Betriebssystem entworfen worden ist. Die Kernkomponenten der Architektur sind:

Security Server: Es existiert genau eine Instanz des Security Servers. Hier werden zentral alle Entscheidungen über Zugriffsberechtigungen auf Objekte anhand einer konfigurierbaren Policy getroffen. Die Bezeichnung *Server* für diese Komponente mutet auf den ersten Blick vielleicht etwas eigenartig an. Es wird allerdings klarer, wenn man weiß, dass es sich bei Fluke um ein System handelt, das auf der Microkernel Architektur beruht. Der Security Server läuft hier tatsächlich als eigener Prozess im User Space.

Objekt Manager: Die Objekt Manager sind für die Umsetzung der Vorgaben des Security Servers zuständig. Es gibt somit eine klare Trennung zwischen Regelwerk und Anwendung bzw. Umset-

⁴¹DTE Linux: <http://www.cs.wm.edu/~hallyn/dte/>

⁴²Openwall: <http://www.openwall.com/>

zung der Regeln. Die Objekt Manager werden unter Linux durch die einzelnen Kernel-Subsysteme (Prozessverwaltung, Dateisystem, ...) repräsentiert.

Access Vector Cache (AVC): Der AVC wird von den Objekt Managern verwendet, um Resultate von Anfragen an den Security Server zwischenspeichern.

Die Teilnehmer des Systems werden in 2 Gruppen geteilt:

Subjekte: dabei handelt es sich um die Prozesse, die im System laufen und auf Objekte zugreifen wollen

Objekte: dabei handelt es sich um Dateien bzw. Kernel-Objekte, die Prozesse, Netzwerkgeräte oder Sockets repräsentieren

Alle Subjekte und Objekte werden mit einem *Security Label* versehen. Diese Labels bilden die Grundlage für Entscheidungen des Security Servers anhand der zur Zeit aktiven Policy. Die Umsetzung dieser Entscheidung bleibt den einzelnen Objekt Managern überlassen womit die klare Trennung zwischen Entscheidungsfindung und Umsetzung gewährleistet ist.

Zwischen Objekt Manager und dem Security Server befindet sich der AVC. Seine Aufgabe ist es, Resultate zu Anfragen der Objekt Manager an den Security Server zwischenspeichern. Damit wird die potentiell zeitaufwändige Berechnung der Zugriffsberechtigung durch den Server für wiederholte Anfragen abgemildert. Abbildung 25 stellt die gerade beschriebenen Zusammenhänge grafisch dar.

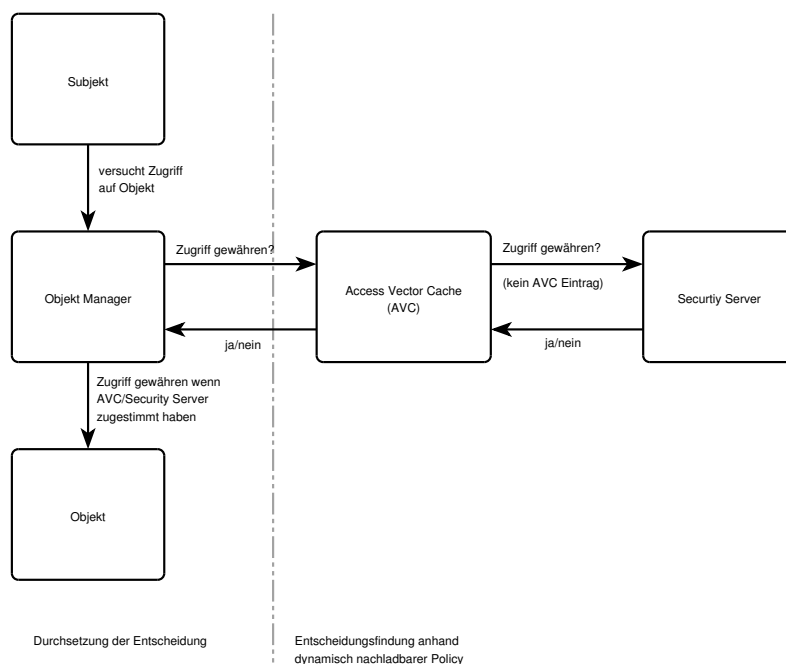


Abbildung 25: Architektur von SELinux

Um die verschiedenen Objekte und Subjekte mit *Security Labels* zu versehen kommen die von den *Linux Security Modules* eingeführten *Security Fields* (zusätzliche void* Pointer für zentrale Kernel Datenstrukturen) zum Einsatz. Beim Labeling unterscheidet man zwischen *Security Context* und *Security Identifier* (SID).

Security Context: String variabler Länge, der sich aus Benutzererkennung, Rolle und Typ/Domäne zusammensetzt (jeweils getrennt durch einen Doppelpunkt). Diese Benutzererkennung wird beim Login zugewiesen und hat nichts mit der ansonsten unter Linux bekannten UID zu tun.

Security Identifier: 32Bit Integer – Objekte werden bei ihrer Erzeugung mit einer zufälligen SID versehen, die vom Security Manager generiert wird. Für Dateien sind solche zufälligen SIDs nicht brauchbar. Aus diesem Grund gibt es hier persistente SIDs (PSID) die den Dateien zugeordnet werden und somit auch bei einem Neustart des Systems erhalten bleiben.

Der Security Server verwaltet ein Mapping zwischen Security Context und SID. Greift ein Subjekt auf ein Objekt zu, so fragt das betroffene Subsystem über den AVC beim Security Server nach, ob der Zugriff gestattet werden soll. Für die Berechnung wird die SID von Objekt und Subjekt sowie die Objekt-Klasse übergeben. Anhand dieser Daten prüft der Server, ob der Zugriff erlaubt ist. Als Resultat gibt er einen so genannten Zugriffsvektor zurück. Dieser besteht aus den beiden SIDs, der Klasse sowie den vom Server berechneten Zugriffsrechten. Dabei ist es möglich, dass der Server für mehr Szenarien die Zugriffsrechte berechnet und zurück gibt als eigentlich angefordert wurden. Diese werden dann vom AVC zusätzlich zwischengespeichert und stehen bei möglichen zukünftigen Anfragen sofort zur Verfügung.

SELinux bietet natürlich auch die Möglichkeit die zu Grunde liegende Policy zur Laufzeit auszutauschen. Dabei werden zuerst die aktuellen AVC-Einträge als ungültig markiert. Danach werden die AVC Einträge entsprechend der neuen Policy aktualisiert. Der AVC verständigt dann, sofern notwendig, die Objekt Manager über Callback Funktionen. Diese aktualisieren die Zugriffsrechte die z.B. Kernel-Objekten zwischengespeichert wurden. Nachdem das erfolgt ist, schickt der AVC eine Benachrichtigung an den Security Server um ihm mitzuteilen, dass die neue Policy nun aktiv ist. Die meisten Rechte werden jedoch bei jedem Zugriff neu gegen den AVC geprüft, weshalb nur wenige Kernel-Objekte mittels Callback aktualisiert werden müssen.

Wie weiter oben schon angesprochen hat die Benutzererkennung nichts mit der Linux UID zu tun. Sie wird beim Login⁴³ zugewiesen und ist in Folge nicht mehr änderbar. Auch wenn der Benutzer seine UID beispielsweise mittels `su` ändert, bleibt die *SELinux* Benutzererkennung dieselbe.

10.2.3 Implementierung

Für die Implementierung wurde das schon aus Abschnitt 10.1 bekannte *Linux Security Modules Framework* verwendet. *SELinux* ist somit ein *Security Module* das die von LSM angebotenen Hooks nützt und eigene Hook-Funktionen registriert. Der Sourcecode von *SELinux* befindet sich, klar abgegrenzt vom Rest des Kernels, unter `security/selinux/`.

10.2.4 Anwendung

Für die Definition der Policy kommt bei *SELinux* eine Mischung aus verschiedenen Modellen zum Einsatz:

Benutzeridentität: die schon angesprochene *SELinux* Benutzererkennung

Type Enforcement (TE): Subjekte werden Domänen zugeordnet während Objekte zu Typen zugeordnet werden. Die Beziehungen zwischen Domänen und Typen werden in einer Matrix im Security Server definiert. Sowohl Domänen als auch Typen können – je nach Bedarf – in beliebiger Anzahl definiert werden, womit ein sehr feinmaschiges Beziehungs-Netzwerk ermöglicht wird. Intern unterscheidet *SELinux* nicht zwischen Domänen und Typen – es handelt sich dabei in erster Linie um eine sprachliche Unterscheidung. Die Implementierung kann dadurch einfacher gestaltet werden – am Konzept ändert sich deswegen aber nichts.

Role-Based Access Control (RBAC): Jeder Benutzer kann einer oder auch mehreren Rollen zugeordnet werden. Jeder Rolle wiederum ist eine Reihe von Domänen zugeordnet. Eine Rolle hat für die ihr untergeordneten Domänen jeweils bestimmte Rechte. Ein Benutzer hat die Möglichkeit mit dem `newrole` Kommando zwischen den für ihn zulässigen Rollen zu wechseln.

⁴³SELinux benötigt folglich ein modifiziertes Login Programm

Abbildung 26 veranschaulicht die logischen Zusammenhänge zwischen *SELinux* Benutzeridentitäten, Rollen und Domänen bzw. Typen.

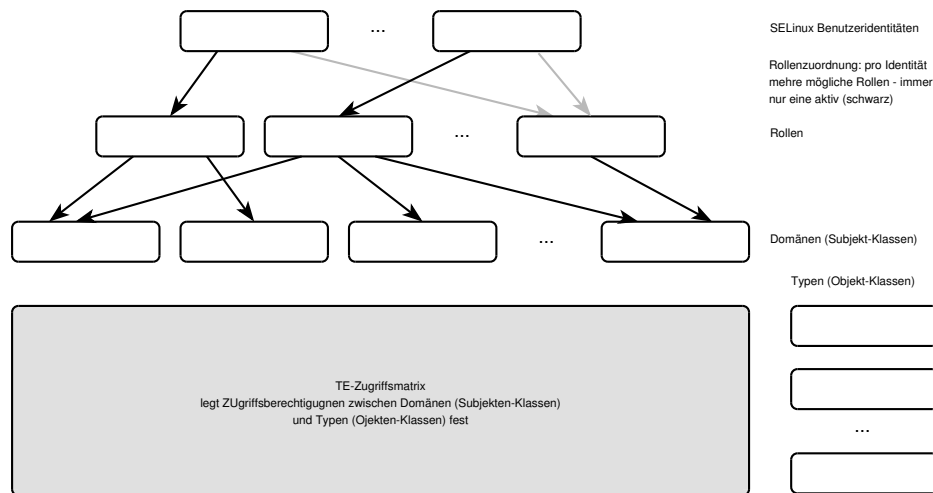


Abbildung 26: Benutzer können in verschiedene Rollen schlüpfen. Je nach Rolle sind verschiedene Domänen verfügbar. Die Domänen legen, in Kombination mit den Typen, die Zugriffsrechte der Subjekte auf die Objekte fest.

Die im System vorhandenen Domänen und Typen werden in Konfigurationsdateien beschrieben. Dabei wird genau festgelegt, welche Domänen in welcher Art und Weise auf bestimmte Typen zugreifen dürfen. Alle Zugriffe, die nicht ausdrücklich erlaubt werden, werden von *SELinux* als verboten angesehen. Die Zuordnung zwischen Typen und Dateien geschieht dann wiederum in eigenen Konfigurationsdateien, den so genannten *Files Contexts*.

Im Gegensatz zur Rolle kann die Domäne im Normalfall nicht explizit geändert werden⁴⁴, sondern ein Domänen-Wechsel passiert automatisch. Solche Wechsel sind als so genannte *Type Transitions* von der Policy festzulegen. Notwendig werden solche Wechsel immer dann wenn ein Programm durch ein anderes Programm gestartet wird aber nicht dessen Domäne erben soll. Das ist zum Beispiel bei einem Webserver, der beim Start des Rechners mittels Init Skript gestartet wird der Fall. Init Skripte laufen in einer eigenen Domäne, die mit mehr Rechten ausgestattet ist als für den Betrieb des Webserver notwendig sind. Damit sich der Schaden beim Angriff auf den Webserver in Grenzen hält sollte dieser mit minimalen Rechten in einer eigenen Domäne laufen, was mit dem Domänen-Wechsel erreicht werden kann.

⁴⁴Die Ausnahme bildet die Verwendung eines speziellen Systemcalls was speziell angepasste Programme erforderlich macht. Auch ein solcher expliziter Domänen-Wechsel muss durch die Policy ausdrücklich gestattet werden.

A Spezielle Code-Konstrukte

In diesem Abschnitt sollen einige spezielle Konstrukte, die beim Lesen der Kernelquellen immer wieder auffallen, kurz näher beschrieben werden.

A.1 `asmlinkage`

Bei `asmlinkage` handelt es sich um einen zusätzlichen Qualifier, der als Makro in der Datei `include/asm/linkage.h` definiert ist und zu dem in Listing 22 gezeigten Code expandiert. Eine typische Verwendung des Qualifiers sieht wie in Listing 21 aus.

```
1  asmlinkage long sys_socketcall(int call, unsigned long __user *args)
2  {
3      /* ... */
4  }
```

Listing 21: Verwendung von `asmlinkage` (*net/socket.c*)

Das Makro wird auf den Architekturen IA32 und IA64 benutzt um über die GCC Erweiterung `__attribute__` den Compiler auf die speziellen Aufrufkonventionen der Funktion aufmerksam zu machen.

```
1  #define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))
```

Listing 22: Code des `asmlinkage` Makros

Die `regparm` Direktive bedeutet, dass die Parameter der Funktion nicht konventionell über den Stack sondern mittels Registern übergeben werden. Damit ist auch geklärt, warum System Calls mit `asmlinkage` gekennzeichnet sind. Der Parameter von `regparm` gibt die Anzahl der maximal übergebenen Parameter an. Weiters werden mit `asmlinkage` gekennzeichnete Funktionen stets aus Assemblercode aufgerufen – daher auch der Name. Aus diesem Grund wird `regparm` auch mit einer 0 als Parameter angegeben – im aufrufenden Assemblercode werden die fraglichen Register von Hand befüllt.

A.2 `likely` und `unlikely`

Diese Anweisung ist häufig bei bedingten Anweisungen zu finden. Listing 23 zeigt die Verwendung im Scheduler:

```
1
2  if (unlikely(!array->nr_active)) {
3      /* ... */
4  }
```

Listing 23: Verwendung von `likely` (*kernel/sched.c*)

Auch hier handelt es sich um Präprozessor-Makros welche in `include/linux/compiler.h` definiert sind:

```
1  #define likely(x) __builtin_expect(!!(x), 1)
2  #define unlikely(x) __builtin_expect(!!(x), 0)
```

Listing 24: Code der `likely` und `unlikely` Makros

Die Intention dieser Makros ist es, dem Compiler einen Hinweis für die Branch-Prediction zu geben, damit dieser die möglichen Ausführungszweige optimal im Speicher anordnen kann. Dadurch wird die

Ausführung beschleunigt, da weniger bedingte Sprünge ausgeführt werden und somit die Hardware-Caches besser ausgenutzt werden.

`__builtin_expect` ist ab GCC 2.96 verfügbar. Für alle anderen Compiler ist diese Funktion über weitere Makros als leere Funktion definiert.

Abbildungsverzeichnis

1	Die Task-Zustände im Überblick	12
2	Aufteilung der statischen Prioritäten	15
3	<code>active</code> Array der <code>runqueue</code> im Detail	17
4	Performance-Vergleich des Schedulers (Quelle: [developer.osdl.org])	21
5	Gegenüberstellung UMA- und NUMA Systeme	22
6	Speicherabbildung in NUMA-Systemen	23
7	Fallback-Liste für eine Node	25
8	Zusammenhänge zwischen den Allokationsfunktionen	27
9	Komponenten des Slab-Allokators	29
10	Struktur des Slab-Caches	29
11	Aufteilung einer virtuellen Adresse	31
12	Behandlungsroutine eines Seitenfehlers	35
13	Aufruf eines Systemcalls	38
14	Zusammenhang <code>sk_buff</code> Datenstruktur mit einem Netzwerkpaket	40
15	Ausgehendes TCP-Paket in Zusammenhang mit <code>sk_buff</code> Datenstruktur	41
16	Struktur der IPv4 Vermittlungsschicht	43
17	Struktur der IPv6 Implementierung	46
18	Aktives Öffnen einer Netzwerkverbindung	48
19	Ablauf bei eingehendem <code>SYN ACK</code> am Client	49
20	Ablauf bei eingehendem <code>ACK</code> am Server	49
21	Ablauf bei eingehenden Paketen	50
22	Der Weg durch den TCP-Stack im Überblick	53
23	Einsatz des LSM Hooks im Rahmen des <code>open</code> System Call	54
24	permissive Hooks können DAC Entscheidungen aufheben	55
25	Architektur von SELinux	59
26	Zusammenhang zwischen Benutzeridentitäten, Rollen und Domänen	61

Listings

1	Verwendung von Spin Locks	10
2	Verwendung von Spin Locks in Interrupt Handlern	10
3	Verwendung von Reader/Writer Semaphoren (<code>kernel/sys.c</code>)	11
4	lokales Ausschalten der Interrupts	11
5	Scheduling Informationen in <code>task_struct</code> (<code>include/linux/sched.h</code>)	15
6	<code>runqueue</code> Datenstruktur (<code>kernel/sched.c</code>)	16
7	<code>prio_array</code> Datenstruktur (<code>kernel/sched.c</code>)	17
8	Auszug aus <code>schedule</code> Funktion (<code>kernel/sched.c</code>)	18
9	Ausschnitt aus <code>scheduler_tick</code> (<code>kernel/sched.c</code>)	19

10	Repräsentation einer Node mittels <code>pg_data_t</code> (<code>include/linux/mmzone.h</code>)	23
11	Fallback-Liste (<code>include/linux/mmzone.h</code>)	24
12	Datenstrukturen für das Buddy-System (<code>include/linux/mm.h</code>)	26
13	Einträge der Seitentabellen (<code>include/asm-i386/page.h</code>)	31
14	Datenstruktur zur Verwaltung des Prozessspeichers (<code>include/linux/sched.h</code>)	33
15	Datenstruktur für eine Speicherregion (<code>include/linux/mm.h</code>)	33
16	Datenstruktur <code>sk_buff</code> (<code>include/linux/skbuff.h</code>)	38
17	Datenstruktur <code>socket</code> (<code>include/linux/net.h</code>)	51
18	Datenstruktur <code>sock</code> (<code>include/net/sock.h</code>)	51
19	Ausschnitt aus <code>include/linux/security.h</code>	55
20	System Call <code>sys_nice</code> mit LSM Hook	56
21	Verwendung von <code>asmlinkage</code> (<code>net/socket.c</code>)	62
22	Code des <code>asmlinkage</code> Makros	62
23	Verwendung von <code>likely</code> (<code>kernel/sched.c</code>)	62
24	Code der <code>likely</code> und <code>unlikely</code> Makros	62

Literatur

- [Torvalds 2001] Linus Torvalds, David Diamond: „Just For Fun“, Carl Hanser Verlag, München (2001)
- [Moody 2001] Glyn Moody: „Rebel Code“, The Penguin Press (2001)
- [Love 2003] Robert Love: „Linux Kernel Development“, SAMS Publishing, Indianapolis (2003)
- [Mauerer 2003] Wolfgang Mauerer: „Linux Kernelarchitektur - Konzepte, Strukturen und Algorithmen von Kernel 2.6“, Carl Hanser Verlag, München (2003)
- [Bovet, Cesati 2002] Daniel P. Bovet, Marco Cesati: „Understanding the Linux Kernel, Second Edition“, O'Reilly & Associates, Sebastopol (2002)
- [Stallings 2001] William Stallings: „Operating Systems, Fourth Edition“, Prentice-Hall, Inc., New Jersey (2001)
- [Peterson Davie 2003] Larry L. Peterson, Bruce S. Davie: „Computer Networks - A Systems Approach, 3rd Edition“, Morgan Kaufmann Publishers, San Francisco (2003), imprint of Elsevier Science
- [LSM, Usenix 2002] Wright, Cowan, Smalley, Morris, Kroah-Hartman: „Linux Security Modules: General Security Support for the Linux Kernel“, USENIX Association, 11th USENIX Security Symposium, San Francisco (2002)
- [LSM, Ottawa 2002] Wright, Cowan, Smalley, Morris, Kroah-Hartman: „Linux Security Module Framework“, 11th Ottawa Linux Symposium, Ottawa Canada (2002)
- [Flask 1999] Spencer, Smalley, Loscocco, Hibler, Andersen, Lepreau: „The Flask Security Architecture: System Support for Diverse Security Policies“, USENIX Association, 8th USENIX Security Symposium, San Francisco (1999)
- [SELinux, NSA 2001] Loscocco, Smalley: „Integrating Flexible Support for Security Policies into the Linux Operating System“, National Security Agency, NAI Labs (2001)
- [SELinux, Module 2002] Stephen Smalley, Chris Vance, Wayne Salamon: „Implementing SELinux as a Linux Security Module“, NCSC, NAI Labs (2002)

- [SELinux, iX 12/2003] Oliver Tennert: „Hinter Schloss und Riegel“, iX 12/2003, Heise Zeitschriften Verlag GmbH, Hannover, Deutschland (2003)
- [SELinux, Linux Magazine 01/2003] Carsten Grohmann, Konstantin Agouros: „Regel-recht“, Linux Magazin 01/2003, Linux Media AG, München, Deutschland (2003)
- [SELinux, Linux Magazine 02/2003] Carsten Grohmann: „Regel-Praxis“, Linux Magazin 02/2003, Linux Media AG, München, Deutschland (2003)
- [developer.osdl.org] Performance-Vergleich des Schedulers in Version 2.4.18 und 2.6.0-test9 <http://developer.osdl.org/craiger/hackbench/index.html> (Jänner 2004)
- [comp.os.minix Linus 25.8.1991] Artikel <1991Aug25.205708.9541@klaava.Helsinki.FI> in der Newsgroup: comp.os.minix <http://groups.google.com/groups?selm=1991Aug25.205708.9541%40klaava.Helsinki.FI> (Jänner 2004)
- [comp.os.minix Tanenbaum 29.1.1992] Artikel <12595@star.cs.vu.nl> in der Newsgroup comp.os.minix <http://groups.google.com/groups?selm=12595%40star.cs.vu.nl> (Jänner 2004)
- [comp.os.minix Linus 29.1.1992] Artikel <1992Jan29.231426.20469@klaava.Helsinki.FI> in der Newsgroup comp.os.minix <http://groups.google.com/groups?selm=1992Jan29.231426.20469%40klaava.Helsinki.FI> (Jänner 2004)
- [comp.os.minix Linus 30.1.1992] Artikel <1992Jan30.153816.1901@klaava.Helsinki.FI> in der Newsgroup comp.os.minix <http://groups.google.com/groups?selm=1992Jan30.153816.1901%40klaava.Helsinki.FI> (Jänner 2004)
- [Barger Nov. 2002] Jorn Barger: „Timeline of GNU/Linux and Unix“ <http://www.robotwisdom.com/linux/timeline.html> (Jänner 2004)