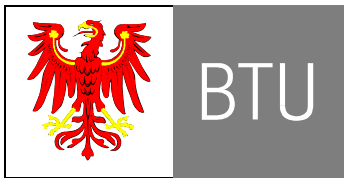

Brandenburgische Technische Universität Cottbus

TECHNISCHE UNIVERSITÄT DRESDEN

Diplomarbeit

Realisierung geschützter Speicherbereiche für Funktionen des LINUX-Systemkerns

Konzeptionierung und Implementierung von Modul-Servern mit privaten Adressräumen für das Betriebssystem LINUX auf der Basis des Fiasco-Mikrokerns

Oliver Stecklina

ost@stecklina-net.de

14. Juni 2003

Gutachter:

Prof. Dr.-Ing. Heinrich Theodor Vierhaus
Prof. Dr. Hermann Härtig

Betreuer:

Dipl.-Inf. Christian Helmuth
Dipl.-Inf. Birk Richter

Eidesstattliche Erklärung

Die vorliegende Diplomarbeit wurde von mir selbstständig angefertigt. Die verwendeten Hilfsmittel und Quellen sind vollständig im Literaturverzeichnis aufgeführt. Eingetragene Warenzeichen und Copyrights werden anerkannt, auch wenn sie nicht explizit gekennzeichnet sind.

Cottbus, den 14. Juni 2003

.....

Oliver Stecklina

Inhaltsverzeichnis

Abbildungsverzeichnis

Listing-Verzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACL ..... Access Control List
ACE ..... Access Control Entry
CPL ..... Current Privilege Level
DMphys ..... Physical Memory Dataspace Manager
DPL ..... Descriptor Privilege Level
DROPS ..... Dresden Real-Time Operating System Project
EIP ..... Instruction Pointer
ESP ..... Stack Pointer
FID ..... Function Identifier
GB ..... Gigabyte
GDT ..... Global Descriptor Table
GPL ..... GNU Public License
ID ..... Identifier
IDL ..... Interface Definition Language
IDS ..... Intrusion Detection System
IPC ..... Inter-Prozess Kommunikation
IPSec ..... Internet Protocol Security
LDT ..... Local Descriptor Table
LKM ..... Ladbare Kernel-Module
MID ..... Module Identifier
RPC ..... Remote Procedure Call
TCP ..... Transmission Control Protocol
TSS ..... Task State Segment

Kapitel 1

Einleitung

Das Ziel der hier vorgestellten Diplomarbeit ist die Unterteilung des LINUX-Systemkerns in separate Komponenten auf der Basis des Konzeptes der ladbaren Kernel-Module. Als Basisarchitektur wird hierzu die L^{4}LINUX-Portierung der Technischen Universität Dresden verwendet. Das Design gewährleistet eine separate und damit sichere Speicherverwaltung für ladbare Kernel-Module (LKM), so dass Module in getrennten Adressräumen ablaufen und voreinander sowie vor Funktionen des LINUX-Kerns geschützt sind. Eine Interaktion zwischen dem Systemkern und den einzelnen Modulen kann nur über eine zuvor definierte Schnittstelle erfolgen. Es wird gezeigt, dass damit die angestrebten Schutzziele Vertraulichkeit, Integrität und Verbindlichkeit der Systemkomponenten gesichert werden können.

1.1 Motivation

LINUX ist ein frei verfügbares UNIX-artiges Betriebssystem. Ursprünglich nur für die x86-Architektur von Intel entwickelt, läuft es heute auch auf Digital Alpha, Sparc Workstation, PowerPC und vielen anderen Architekturen. Die ersten öffentlichen Quellen des LINUX-Kerns erschienen 1991 im Internet. Es bildete sich schnell eine Gruppe von Programmierern, welche die Entwicklung des Betriebssystems vorantrieben. Sowohl der Kern als auch die Software von LINUX werden unter offenen und verteilten Bedingungen entwickelt. Dies bedeutet, dass jeder, der dazu in der Lage ist, sich an der Entwicklung beteiligen kann. Die Kommunikation zwischen den einzelnen Entwicklern erfolgt zu großen Teilen über das Internet, so dass geographische Grenzen seit jeher keine Rolle spielten.

LINUX ist frei. Dies ist wohl das bekannteste Merkmal von LINUX. Frei hat in diesem Zusammenhang jedoch zwei Bedeutungen. Zum einen ist LINUX frei, weil man es kostenlos aus dem Internet herunterladen und auf einer Workstation installieren kann. Der LINUX-Kern und ein Grossteil der verfügbaren Programme unterliegen der *GNU Public License* (GPL) [24]. Damit hat jedermann das Recht, die Programme kostenlos zu benutzen, zu kopieren und zu modifizieren. LINUX ist auch auf andere, noch wichtigere Weise frei. Da man nicht nur die Programme sondern auch den Quellcode erhält, kann man LINUX, wenn einem das Standardverhalten nicht gefällt, verändern und den eigenen Bedürfnissen anpassen. Eben diese Freiheit hat dazu geführt, dass LINUX zunächst in vielen Bereichen der Entwicklung und Forschung weit verbreitet war. Der zunehmende Einsatz von LINUX an Forschungseinrichtungen hat wohl mit dazu beigetragen, dass LINUX heutzutage auch in der Wirtschaft mehr und mehr eingesetzt wird. Im Jahr 2001 lag der Marktanteil im Bereich der Server-Installationen schon bei über 20 Prozent mit einer bis heute steigenden Tendenz [25]. Hinzu kommt eine weite Verbreitung im Bereich der eingebetteten Systeme: PDAs, Routern oder Thin-Clients, wo meist nur der Kern selbst und eine sehr geringe Anzahl an Anwendungsprogrammen zum Einsatz kommen.

Multitaskingsysteme, wie LINUX, stellen besondere Anforderungen an die Speicherverwaltung. Es muss sichergestellt werden, dass der Speicher des Systemkerns und eines Nutzerprozesses vor Zugriffen anderer Prozesse geschützt wird. Doch insbesondere bei eingebetteten Systemen ist dies meist nicht ausreichend. Hier wird nicht selten auf die Verwendung von verschiedenen Nutzern verzichtet, da die verwendete Software nicht ohne Administratorrechte lauffähig ist oder gar kein physischer Zugang zum System existiert. Eine Unterteilung des Speichers in einen Nutzer- und einen Systemkernbereich ist hier oftmals nicht ausreichend. Die Ausnutzung einer Vielzahl der bisher

bekannten Sicherheitslücken ist auch dann möglich, wenn nur ein bestimmter Dienst auf dem System ausgeführt wird. Meistens ist ein physikalischer Zugang zu dem System hierzu nicht notwendig. Mit sogenannten *Rootkits* verstecken Angreifer verräterische Meldungen des Betriebssystems, mittlerweile sogar durch direkte Manipulation des RAM-Speichers [17]. Moderne Varianten dieser Rootkits manipulieren direkt den Systemkern, indem sie Systemrufe ersetzen oder verändern. Die notwendigen Administratorrechte werden durch die Ausnutzung einer Sicherheitslücke erlangt. Anschließend werden kompromittierte LKMs eingesetzt, um den Programmcode innerhalb des Systemskerns zu verändern. Dies ist möglich, da die monolithische Struktur des Systemkerns von LINUX Module als gleichberechtigte Komponenten integriert.

Während Nutzerprogramme lediglich Daten ihres Adressraumes sehen und auf Systemkomponenten nur über wohldefinierte Schnittstellen zugreifen können, unterliegt der Systemkern keinerlei Beschränkungen. Das Sicherheitskonzept von LINUX beruht auf dem vollständigen Vertrauen gegenüber Komponenten des Systemkerns. Sie haben uneingeschränkten Zugriff auf den gesamten Arbeitsspeicher und auf alle angeschlossenen Geräte. Die globalen Rechte des Systemkerns und das Konzept der LKMs haben dazu geführt, dass in sicherheitskritischen Bereichen dem LINUX-Kern nur bedingt vertraut werden kann. Ist es einem Angreifer mithilfe eines LKMs gelungen, zusätzlichen Programmcode in den Kern einzubringen, kann er das gesamte System kompromittieren. Von diesen LKM-Rootkits sind auch viele eingebettete Systeme und Server betroffen, obwohl Nutzer meist keinen Zugang zu diesen Systemen haben.

Alle Mechanismen, die zum Schutz von Speicherbereichen in den Systemkern eingefügt werden, können durch Einbringen von zusätzlichen Programmcodes im Nachhinein deaktiviert oder umgangen werden. Da der Kern alle Rechte innerhalb des Systems besitzt, kann keine Komponente hinzugefügt werden, die dessen Rechte wirksam einschränkt. Somit können die angestrebten Schutzziele Vertraulichkeit, Integrität und Verbindlichkeit der Systemkomponenten für einen monolithischen LINUX-Kern nicht gesichert werden.

1.2 Sicherheit in LINUX-basierten Informationssystemen

Sicherheit besteht bei informationstechnischen (IT-)Systemen darin, dass Schutzziele wie Vertraulichkeit, Integrität, Verfügbarkeit und Verbindlichkeit trotz intelligenter Angreifer durchgesetzt werden [18]. Die Schutzziele werden dabei wie folgt abgegrenzt:

Vertraulichkeit: Sowohl Unbefugten als auch Betreibern des Systems muss der Zugang zu den gespeicherten Informationen verwehrt werden.

Integrität: Ein befugtes und unbemerktes Ändern von Informationen darf nicht möglich sein.

Verfügbarkeit: Die Funktionalität des Systems muss ständig gewährleistet sein, damit befugten Nutzern der Zugang zu den Informationen zu jeder Zeit möglich ist.

Verbindlichkeit: Alle Vorgänge auf dem System müssen eindeutigen einem Verursacher zugerechnet werden können, damit bei einem verantwortungslosen Umgang mit den gespeicherten Informationen rechtliche Ansprüche geltend gemacht werden können.

Insbesondere bei offenen Kommunikationssystemen kann nicht davon ausgegangen werden, dass man allen Teilnehmern vollständig vertrauen kann. So hat die Verbreitung von offenen IT-Systemen dazu beigetragen, dass nicht nur Systembetreiber und –hersteller, sondern auch Nutzer mehr Sicherheit für ihre Daten und Systeme fordern. Als besonders schützenswert galt schon frühzeitig der Verbindungspunkt zwischen dem privaten Intra- und dem allgemeinen Internet. Der Einsatz von sogenannten *Firewalls* wurde schnell zum Muss für alle größeren und kleineren Firmennetzwerke. Jedoch werden nach [19] 81 Prozent aller Angriffe auf IT-Systeme von innen verursacht, so dass Firewalls wirkungslos sind. Da mögliche Angriffe nicht mit ausreichender Sicherheit ausgeschlossen

werden können, wurde der Einsatz von Intrusion Detection Systemen (IDS)[1] und die Sicherung wichtiger Daten mittels kryptographischer Verfahren immer verbreiteter.

Problematisch wird der Einsatz von IDS und kryptographischen Verfahren auf IT-Systemen, denen nur bedingt vertraut werden kann. Beispielsweise setzen einige verteilte IDS Agenten auf den zu überwachenden Rechnern ein und übermitteln die protokollierten Daten auf einen zentralen Server, wo sie anschließend ausgewertet werden können. Hierbei muss sowohl den Agenten, der Datenübertragung als auch dem IDS vertraut werden können. Wurde eine der aufgeführten Komponenten durch einen Angreifer erfolgreich kompromittiert, ist die gesamte Auswertung nicht verlässlich. IDS sind nicht die einzigen verletzbaren Systeme. Auch die Zurechenbarkeit von mittels kryptographischen Verfahren gesicherten Informationen ist gefährdet, wenn den Algorithmen oder dem verwendeten Schlüsselmaterial nicht mehr vertraut werden kann. Ein Großteil der Funktionen zur Sicherung von IT-Systemen ist Bestandteil des Systemkerns. Die Komponenten zur Verschlüsselung von Datenverbindungen (FreeSWAN [9]) oder von Dateisystemen (CryptoFS [10]) sind für LINUX als LKMs implementiert, so dass nur ein kleiner Teil des Funktionsumfanges außerhalb des Systemkerns abläuft. Insbesondere LKM-Rootkits stellen das Vertrauen, welches dem Systemkern entgegengebracht wird, in Frage. Jedoch setzen eben diese Systeme ein Mindestmaß an Vertrauenswürdigkeit gegenüber dem Systemkern voraus. Andernfalls kann nicht sichergestellt werden, dass augenscheinlich sichere Daten durch Dritte gelesen, manipuliert oder vernichtet wurden.

Prinzipiell besteht bei allen Betriebssystemen die Gefahr, dass sie durch ein Rootkit kompromittiert werden. Allerdings ist LINUX als offenes und weit verbreitetes System besonders gefährdet. So gibt es heutzutage de facto für LINUX die meisten bekannten Rootkits, was insbesondere auch noch dadurch gefördert wird, dass nahezu jeder PC-Besitzer eine LINUX-Distribution installieren und die neusten *Exploits*[2] ausprobieren kann. Sicherlich sind auch andere Unix-Systeme wie Solaris oder die verschiedensten BSD-Derivate durch Rootkits gefährdet, jedoch ist deren Verbreitung als weitaus geringer einzuschätzen. Für Unix-fremde Systeme wie die Windows-Familie von Microsoft oder Apples MacOS existieren andere Gefahren, so dass Rootkits für diese Betriebssysteme von der Öffentlichkeit meist unbeachtet bleiben.

Moderne Rootkits wie *t0rnkit*, *adore* und *Knark* installieren Hintertüren auf dem System, mit dessen Hilfe sich ein Angreifer jederzeit Zugriff zu diesem verschaffen kann. Bei der Installation einer solchen Hintertür verdecken die Rootkits ihre Spuren, so dass auch ein aufmerksamer Administrator oftmals erst nach einer längeren Zeit entdeckt, dass sein System angegriffen wurde. Nach dem erfolgreichen Einbringen kompromittierten Programmcodes in den Systemkern von LINUX kann keines der oben genannten Schutzziele mehr sichergestellt werden. Eine beliebige Funktion des Kerns kann sämtliche Daten des Systemspeichers lesen oder verändern, die Verfügbarkeit des Systems beeinträchtigen und sämtliche Spuren in Log-Dateien entfernen. Obwohl die oben genannten Rootkits bisher nur einen Teil der Schutzziele verletzen, sind sie doch die Vorlage für weitere, weitaus gefährlichere Exploits, die über die bisher bekannten Sicherheitsverletzungen hinausgehen können. Beispielsweise diente das Exploit *Herion*, welches eigentlich keine Sicherheitsverletzung darstellte, sondern lediglich sich selbst versteckt, als Vorlage für Knark, dass zusätzlich eine Hintertür installiert und Protokolldaten verändert.

---

[1] IDS nutzen die von IT-Systemen protokollierten Aktivitäten zur automatischen Auswertung, um mögliche Sicherheitsverletzungen erkennen und später nachweisen zu können.

[2] Als ein Exploit (exploit = ausbeuten) wird ein Programm bezeichnet, welches eine Schwachstelle des Systems ausnutzt, um dem Anwender unautorisiert zusätzliche Rechte innerhalb des Systems zu gewähren.

1.3 Ziel der Arbeit

Obwohl LINUX ursprünglich als monolithisches System konzipiert wurde, existieren bereits verschiedene Ansätze zur Unterteilung des Kerns. Hierbei wird versucht die Privilegien für die einzelnen Bestandteile innerhalb des Systemkerns feingranularer zu verteilen, so dass jede Komponente nur die Rechte erhält, die sie unbedingt benötigt.

Die hier vorgestellte Arbeit beschreibt ein Konzept zur Unterteilung von LINUX, so dass für Funktionen des Systemkerns sichere Speicherbereiche bereitgestellt werden können. Hierzu ist es zunächst notwendig, den Systemkern in seinen Rechten zu beschränken. Aus diesem Grund basiert die Implementierung auf der L^{4}LINUX-Portierung, wo der Systemkern als Nutzerprozess eines Mikrokerns im *User-Mode* des Prozessors abläuft. Im *SuperVisor*-Modus wird lediglich der Mikrokern ausgeführt. Damit besteht für den LINUX-Kern nicht mehr die Möglichkeit das gesamte System zu kontrollieren. Die Unterteilung des Systemkerns wird an der Modulschnittstelle vorgenommen. Sie begrenzt die Kommunikation zwischen Systemkomponenten des Kerns, so dass eine Portierung auf eine Interprozess Kommunikation (IPC) leichter möglich wird.

Die beschriebene Architektur ermöglicht die Auslagerung von Modulen in separate L4-Tasks, so dass sie in einem eigenen Adressraum ablaufen, wo private Daten vor unbefugten Zugriffen geschützt sind. Die Kommunikation zwischen den einzelnen Tasks kann mittels einer *Access Control List* (ACL) eingegrenzt werden. Die Bereitstellung eines eigenen Adressraumes für jedes ausgelagerte Modul stellt sicher, dass private Daten nur für Funktionen des Moduls zugänglich sind. Dazu gehört auch der Programmcode des Moduls, welcher im Nachhinein nicht mehr verändert werden darf. Damit öffentliche Funktionen eines Moduls nicht durch unautorisierte Funktionen missbraucht werden können, wird eine Zugriffskontrolle eingeführt. Die Granularität der Zugriffskontrolle wird mittels einer ACL gesteuert und ist auf die Task-Ebene beschränkt, so dass eine breitere Aufteilung des Kerns in einzelne Tasks einen zusätzlichen Schutz bieten kann.

1.4 Vorgehensweise

Im zweiten Kapitel erfolgt eine Beschreibung der bereits vorhandenen Konzepte. Dabei wird zunächst auf die Implementierung des Speichermanagements von LINUX eingegangen. Die Erläuterung dient als Grundlage für das Verständnis der folgenden Abschnitte. Eine genauere Beschreibung der Architektur von LINUX ist in [4] und [7] zu finden. Anschließend wird das Palladium-Konzept [1] kurz vorgestellt. Das Konzept verfolgt ein ähnliches Ziel wie das der hier vorgestellten Arbeit, nämlich die Unterteilung von LINUX an der Modul-Schnittstelle. Es wird gezeigt, wo die Probleme des Konzeptes liegen und das es die einführend gestellten Anforderungen nur bedingt erfüllen kann. Im letzten Abschnitt des Kapitels erfolgt eine Beschreibung der L^{4}LINUX-Architektur [20]. Sie diente als Grundlage für die hier vorgestellte Implementierung.

Das Kapitel 3 befasst sich mit der Architektur zur Auslagerung von LKMs. Zunächst werden die Basiskomponenten kurz vorgestellt, bevor im Abschnitt 3.2 auf die Schnittstellen zur Verwaltung der ausgelagerten Module eingegangen wird. Die Abschnitte 3.3 und 3.4 beschreiben das Speichermanagement und die RPC-Architektur im Einzelnen. Beide Komponenten sind für die Interaktion zwischen den einzelnen L4-Tasks zwingend notwendig und bilden zusammen mit dem im Abschnitt 3.5 beschriebenen Konzept zur Verwaltung der öffentlichen Symbole eines Moduls beziehungsweise des L^{4}LINUX-Servers den Kern der hier vorgestellten Arbeit. Der letzte Abschnitt des Kapitels befasst sich mit der Umsetzung der Zugriffskontrolle mittels einer ACL.

Nach der Vorstellung des Gesamtkonzeptes im Kapitel 3 erfolgt im Kapitel 4 eine Beschreibung der Funktionen der RPC-Schnittstelle und der Integration der Architektur in die L^{4}LINUX-Umgebung. Abschnitt 4.1 erläutert zunächst eine Möglichkeit zur automatischen Generierung des Programmcodes der Schnittstellen. Eine Beschreibung der RPC-Funktionen und deren Komponenten erfolgt im Abschnitt 4.2, bevor anschließend auf die Integration in die L^{4}LINUX-Architektur eingegangen wird.

Hierbei wird zunächst im Abschnitt 4.3.1 die Erweiterung des L^{4}LINUX-Servers erläutert und später in den Abschnitten 4.3.2 und 4.3.3 die Richtlinien zur Programmierung und zum Übersetzen eines ausgelagerten Moduls sowie des Gesamtsystems kurz beschrieben.

Um Aussagen über die Nutzbarkeit des Konzeptes machen zu können, erfolgt im Kapitel 5 eine Beschreibung der ermittelten Messwerte. Da bisher keine Tests an komplexen Modulen durchgeführt werden konnten, werden zunächst im Abschnitt 5.2 typische Anwendungsfälle betrachtet. Im Abschnitt 5.3 werden die ermittelten Ergebnisse noch einmal zusammengefasst und hinsichtlich ihrer möglichen Auswirkungen auf die Gesamtleistung des Systems bewertet.

Kapitel 6 fasst die mit der hier vorliegenden Arbeit gewonnenen Erkenntnisse noch einmal zusammen und geht abschließend in einem kurzen Ausblick auf mögliche Weiterentwicklungen und Verbesserungen ein.

Kapitel 2

Speicherschutzmechanismen für LINUX

In diesem Abschnitt erfolgt zunächst eine Beschreibung der Speicherverwaltung von LINUX und des Konzeptes der ladbaren Kernel-Module. Wie in der Einleitung bereits erwähnt wurde, führte gerade die Einführung von ladbaren Kernel-Modulen dazu, dass die Integrität des Systemkerns zur Laufzeit nicht mehr sichergestellt werden kann. Der zweite Abschnitt des Kapitels beschreibt eine mögliche Lösung des Problems und dessen Grenzen. Das in dieser Arbeit vorgestellte Konzept basiert auf der Mikrokern-Architektur L^{4}LINUX, welche abschließend kurz vorgestellt wird.

2.1 Speicherverwaltung von LINUX

Die Verwaltung des Hauptspeichers ist eine der Kernaufgaben des Betriebssystems. Dieser Abschnitt enthält einen kurzen Überblick über die Speicherverwaltung von LINUX. Für ein besseres Verständnis der Arbeit sind grundlegende Kenntnisse der hier vorgestellten Mechanismen notwendig. Einen vollständigen Überblick findet man in [4] und [7].

2.1.1 Das architekturunabhängige Speichermodell von LINUX

Die ursprüngliche Implementierung von LINUX sollte auf der damals schon weit verbreiteten und preiswerten x86-Architektur von Intel laufen. Die Intel-Architektur unterstützt seit dem 80286-Modul die Segmentierung des Speichers und die Aufteilung in Seiten. Wie in Abbildung 1 dargestellt, wird eine logische Adresse durch die Segmentierungseinheit zunächst in eine lineare Adresse übersetzt. Die lineare Adresse wird anschließend der Paging-Einheit übergeben und dort in die physikalische Adresse umgewandelt.

Abbildung 1: Adressübersetzung der x86-Architektur

Heutzutage existieren LINUX-Portierungen für viele verschiedene Plattformen und nur die wenigsten davon unterstützen eine derartige Adressübersetzung. Bei der Implementierung von LINUX wird die Segmentierung des Speichers so weit wie möglich unterbunden. Dadurch wird es möglich, dass man mit einem möglichst architekturunabhängigen Programmcode arbeiten kann. Dies hat jedoch den Nachteil, dass man beim Paging nur einen Teil der zur Verfügung stehenden Speicherschutzmechanismen verwenden kann.

Segmentierung

Logische Adressen bestehen aus zwei Teilen: Einem Segment-Identifier und einem Offset, das die relative Adresse innerhalb des Segments festlegt. Der Segment-Identifier ist ein 16-Bit-Feld und wird auch *Segment-Selector* genannt, während für das Offset 32 Bit bereitgestellt werden.

Jedes Segment wird durch einen 8-byte Segment-Descriptor charakterisiert. Die einzelnen Descriptoren werden in der *Global Descriptor Table* (GDT) oder in der *Local Descriptor Table* (LDT) gespeichert. Normalerweise wird immer nur eine GDT definiert und im `gdtr` Register des Prozessors gespeichert. Jeder Prozess kann zusätzlich eine eigene LDT verwenden, die im `ldtr` Register abgelegt wird. Dies bedeutet, dass bei einem Prozesswechsel die LDT des alten Prozesses gespeichert und die neue Tabelle geladen werden muss. Um den notwendigen Aufwand beim Prozesswechsel möglichst gering zu halten, verwendet LINUX nur wenige verschiedene Segmente, die alle in der GDT abgelegt werden können. Zusätzlich wird ein LDT-Segment angelegt, welches zunächst von jedem Prozess verwendet wird.

In der GDT wird für den Kernel ein Code- und ein Datensegment angelegt. Im Datensegment des Kernels wird das *Descriptor Privilege Level* (DPL) auf null gesetzt. Damit kann auf den Inhalt des Segmentes nur dann zugegriffen werden, wenn sich der Prozessor im Kernel-Modus befindet. Der aktuelle Modus des Prozessors – *Current Privilege Level* (CPL) wird im Code Segment in einem 2-Bit-Feld abgelegt. In den Nutzersegmenten, wiederum ein Daten- und ein Code-Segment, wird das DPL und das CPL auf drei gesetzt. Damit existieren keine Einschränkungen bezüglich des Zugriffes auf diese Segmente. Alle Prozesse nutzen das gleiche Code- und das gleiche Datensegment, besitzen jedoch ihr eigenes Task State Segment (TSS). Da maximal 512 Prozesse[3] zur gleichen Zeit existieren dürfen, können alle TSS in der GDT abgelegt werden.

Jedes Segment hat die Basisadresse Null und eine Größe von 4 Gigabyte. Damit kann der ganze Adressraum über die einzelnen Segmente adressiert werden und das Segment-Offset entspricht der linearen Adresse des Paging-Mechanismus. Beim Wechsel vom Nutzer- in den Kernel-Modus und umgekehrt, werden die Segment-Selectoren in die zugehörigen Register des Prozessors geladen. Für den Nutzer- beziehungsweise den Kernel-Stack existiert kein separates Segment, so dass das Stack Segment Register immer auf das momentan aktuelle Datensegment verweist.

Paging

Die Paging-Einheit wandelt lineare in physikalische Adressen um. Zur Steigerung der Leistung werden lineare Adressen in Gruppen mit einer festen Größe eingeteilt. Diese Gruppen werden *Pages* (Seiten) genannt. Jede Page verweist auf einen ebenso großen zusammenhängenden physikalischen Speicherbereich. Der Kernel charakterisiert immer Pages, ein kleinerer Speicherbereich kann nicht separat verwaltet werden.

Im Gegensatz zur Segmentierung unterstützt der Paging-Mechanismus nur zwei Privilegienstufen: *User* und *Supervisor*. Supervisor-Seiten können nur verwendet werden, wenn das CPL geringer als drei ist, für LINUX bedeutet dies, dass sich der Prozessor im Kernel-Modus befinden muss. Auf Nutzer-Seiten kann jederzeit zugegriffen werden. Darüber hinaus unterstützt die Paging-Einheit der x86-Architektur nur zwei verschiedene Zugriffsrechte (ein Bit). Dies steht jedoch im Widerspruch zu den sonst üblichen Zugriffsrechten von Unix: *read*, *write* und *execute*. Die Beschränkung auf ein Bit bedeutet, dass lediglich zwei Zugriffsrechte (lesen, schreiben) mit einer Page verbunden werden können. Ist das Bit auf Null gesetzt, kann die Seite nur lesend verwendet werden, anderenfalls kann der Inhalt der Seite gelesen und geschrieben werden.

2.1.2 Unterteilung des Adressraumes

Die x86-Architektur von Intel unterstützt Adressräume bis zu einer Größe von vier Gigabyte. Dies bedeutet, dass der Prozessor nicht mehr als vier Gigabyte Speicher in den Adressraum eines Prozesses

[3] Die Vorgabe wird als Define `NR_TASKS` im Headern `include/linux/tasks.h` definiert. Sollte die Anzahl nicht ausreichen, kann sie vom Nutzer verändert werden. Sie ist jedoch durch die Größe der GDT auf den Maximalwert 4090 beschränkt.

einblenden kann. Verfügt ein System über mehr Speicher, so muss dieser auf mehrere Adressräume verteilt werden. Die Organisation des Adressraumes und das Einblenden des physikalischen Speichers ist Aufgabe des Betriebssystems. Die folgenden Beschreibungen beziehen sich auf das Management für Systeme mit weniger als einem Gigabyte Hauptspeicher.

Abbildung 2 zeigt die Aufteilung des Adressraumes für Systeme mit weniger als einem Gigabyte Hauptspeicher. Der Adressraum eines Prozesses unterteilt sich in den Nutzer- und den Kernel-Speicher[4]. Auf den Nutzerspeicher kann im Nutzer- als auch im Systemmodus zugegriffen werden, wohingegen der Kernel-Speicher nur im Systemmodus verwendet werden kann. Der Kernel-Bereich existiert nur ein Mal und wird in den Adressraum jedes Nutzerprozesses oberhalb der Adresse `PAGE_OFFSET` (0xc0000000 = 3 GB) eingeblendet. Dies hat den Vorteil, dass bei einem Wechsel in den Systemmodus nicht der ganze Adressraum neu initialisiert werden muss, um auf den Kernel-Speicher zugreifen zu können. Der Nutzerspeicher ist für jeden Prozess verschieden, so dass ein Prozess den Adressraum beziehungsweise die enthaltenen Daten eines anderen Prozesses nicht manipulieren kann.

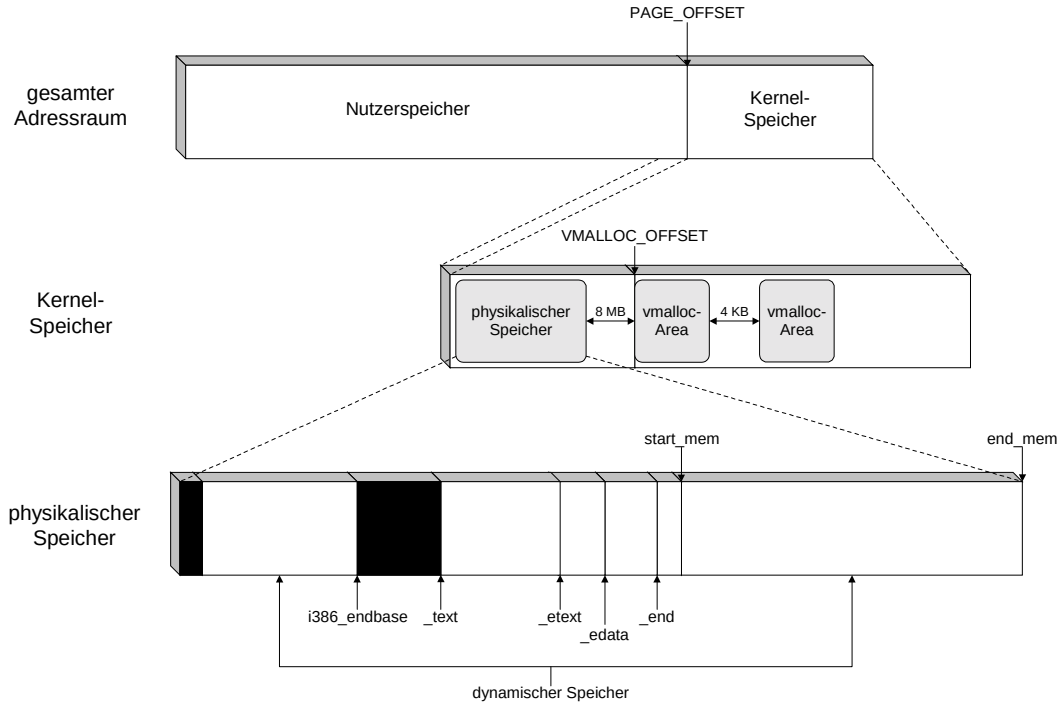

Abbildung 2: Aufteilung des Adressraumes für Systeme mit weniger als einem Gigabyte Speicher

Die Organisation des Nutzerspeichers ist für die hier vorliegende Arbeit nicht weiter interessant, so dass sich die folgenden Beschreibungen im wesentlichen auf den Kernel-Bereich beziehen. Dieser befindet sich, wie bereits beschrieben, oberhalb von drei Gigabyte im Adressraum jedes Nutzerprozesses. Der physikalische Speicher des Systems wird direkt in den Kernel-Bereich ab der Adresse `0xc0000000` eingeblendet und belegt normalerweise nur einen Teil des Bereiches. Der verbleibende Teil wird für die dynamische Verwaltung von großen Speicherbereichen innerhalb des Kernels benötigt. Eine genauere Erläuterung hierzu erfolgt im Abschnitt 2.1.3. Durch das vollständige Einblenden des physikalischen Speichers entsprechen die linearen Adressen des Kernel-Bereichs den physikalischen Adressen des Systemspeichers abzüglich des Offsets `PAGE_OFFSET`.

[4] In der Literatur werden diese beiden Bereiche auch als Nutzer- und Kernel-Segment bezeichnet. Dies ist zum Teil historisch bedingt, da in Versionen vor 2.0 noch getrennte Segmente für den System- und den Nutzermodus existierten.

Der LINUX-Kern wird immer an die physikalische Adresse `text` (`0x00100000 = 1 MB`) geladen, so dass das erste Megabyte des Hauptspeichers zunächst frei bleibt. Dies ist notwendig, da dieser Bereich auf einigen Systemen vom Bios oder Gerätetreibern genutzt wird. Nachdem das System vollständig initialisiert wurde, kann der ungenutzte Speicher dieses Bereiches frei verwendet werden. Der durch den LINUX-Kern statisch genutzte Speicherbereich erstreckt sich bis zur Adresse `_end`, darauf folgt noch ein kleiner Pufferbereich zum Schutz des Kerns vor Speicherüberläufen. Der verbleibende Bereich von der Adresse `start_mem` bis `end_mem` wird für die dynamische Speicherverwaltung verwendet.

Der statische Teil des LINUX-Kerns lässt sich noch einmal in drei Teile gliedern: Dies ist zunächst der Bereich zwischen den Symbolen `_text` und `_etext`, welcher den Programmcode des Systemkerns enthält. Im darauffolgenden Bereich bis zum Symbol `_edata` befinden sich die initialisierten Variablen des Kerns. Der letzte Teil enthält die uninitialisierten Daten des Kerns.

2.1.3 Dynamische Speicherreservierung

Der Speicherbedarf eines Programms beziehungsweise des Kerns ist abhängig von den verschiedensten Bedingungen und zur Laufzeit ständig anders. Der statische Kern belegt normalerweise nur einen sehr kleinen Teil des verfügbaren physikalischen Speichers. Der verbleibende Teil wird zur Laufzeit dynamisch den Anforderungen der einzelnen Programme beziehungsweise des Kerns angepasst. Hierzu existieren verschiedene Funktionen zum Anfordern (*Allokieren*) von Speicherbereichen.

Nutzerspeicher

Nutzerprozesse belegen auf den meisten Systemen den Großteil des physikalischen Speichers, wobei sie die Verwaltung der Bereiche nicht selbst übernehmen. Es existieren Ausnahmen, zum Beispiel große Datenbanksysteme, die eine eigene Speicherverwaltung besitzen. Aber auch sie fordern über die normalen Nutzerschnittstellen einen großen zusammenhängenden Bereich an, den sie dann anschließend selbst verwalten.

Der Nutzerspeicher wird beim Starten eines Programms initialisiert. Hierzu ruft ein Prozess die Systemfunktion `execve(...)` auf. Sie blendet den Inhalt des neuen Programms in den Adressraum des Prozesses ein. Im Gegensatz zum Kern erfolgt dies jedoch on-Demand, dass heißt der Inhalt der Programmdatei wird erst dann in den Hauptspeicher des Systems geladen, wenn der Prozess erstmals auf eine der Seiten zugreift. Der gleiche Mechanismus wird auch bei allen anderen Speicherseiten eines Prozesses verwendet. Der Inhalt einer Seite wird erst dann in den physikalischen Speicher geladen, wenn er wirklich benötigt wird. Damit wird es möglich, dass alle Prozesse zusammen wesentlich mehr Speicher verwenden können, als im System überhaupt verfügbar ist. Hinzu kommt, dass einzelne Seiten, wenn sie eine Zeit lang nicht mehr benutzt wurden, auf die Festplatte ausgelagert werden. Dieser Mechanismus wird als *Swapping* bezeichnet und ist heutzutage Bestandteil jedes modernen Betriebssystems.

Die Prozesse des Systems werden vom Kern in einer Liste verwaltet. Für jeden Prozess wird ein neues Objekt generiert und in die Liste eingehangen. Die Prozessliste befindet sich im Adressraum des Kerns und ist für einen Nutzerprozess nur zugänglich, wenn der Prozessor in den Kernel-Modus gewechselt hat. Ein Wechsel in den Systemmodus erfolgt beispielsweise immer bei der Behandlung eines Interrupts. Für Systemrufe wird dieses Konzept genutzt, indem sie einen Soft-Interrupt generieren und somit den Wechsel in den Systemmodus erzwingen. Nach dem Wechsel kann ein Prozess die Einträge seiner eigenen Prozessstruktur auszulesen oder direkten auf Hardware-Komponenten zugreifen. Die Prozessstruktur enthält alle Eigenschaften des Prozesses, dazu zählt unter anderem auch der von ihm verwendete Speicherbereich. Mittels des Systemrufs `brk(...)` kann ein Prozess seinen eigenen Speicherbereich verändern, also Speicher anfordern beziehungsweise freigeben. Der Systemkern führt beim Aufruf der Funktion lediglich eine Konsistenzprüfung durch. Speicher wird zu diesem Zeitpunkt noch nicht bereitgestellt. Die meiste Zeit sind die Speicherbereiche

eines Prozesses virtuelle. Physikalische Speicherbereiche werden einem Prozess nur bei Bedarf zugeteilt und in den meisten Fällen wird dabei eine andere Seite des gleichen oder eines anderen Prozesses aus dessen Adressraum entfernt.

Greift ein Nutzerprogramm auf eine Speicherseite zu, überprüft der Prozessor zunächst, ob diese Seite im Adressraum des Prozesses verfügbar ist. Ist dies nicht der Fall, wird eine Seitenfehler-Ausnahmeunterbrechung (*Page-Fault*) generiert. Die Behandlung einer solchen Unterbrechung obliegt dem Systemkern. Er ermittelt zunächst den Adressraum des Prozesses und überprüft, ob sich die angeforderte Seite in dessen virtuellen Speicherbereich befindet. Besitzt der Prozess die geforderte Zugriffsberechtigung für diese Seite nicht, wird das Signal `SIGSEV`, welches eine Speicherzugriffsverletzung anzeigt, an den verursachenden Prozess geschickt. Im Normalfall wird daraufhin der Prozess beendet und ein Abbild des virtuellen Speichers (*coredump*) generiert. Gehört die Seite dem verursachenden Prozess, gibt es mehrere Möglichkeiten, warum auf die Seite nicht zugegriffen werden konnte. In den meisten Fällen wurde die Seite ausgelagert oder noch gar nicht geladen. Ebenso kann es sich um eine Seite eines dynamisch angeforderten Speicherbereiches des Prozesses handeln, die nun zum ersten Mal verwendet wird. Erst zu diesem Zeitpunkt wird der Kern den entsprechenden Bereich vom Speichermedium laden oder eine Seite des physikalischen Speichers in den Adressraum des Prozesses einblenden.

Kernel-Speicher

Für das Allokieren von Speicher stellt der Systemkern drei verschiedene Möglichkeiten zur Verfügung. Jede dieser Möglichkeiten unterscheidet sich hinsichtlich der Größe der bereitgestellten Speicherbereiche und des notwendigen Aufwandes. Grundsätzlich basiert die Speicherverwaltung im Kern immer auf Seiten. Nur durch die Einführung zusätzlicher Mechanismen ist es überhaupt möglich, Speicherbereiche mit einer anderen Größe als einem Vielfachen einer Speicherseite bereitzustellen.

Speicherseitenreservierung

Teilt man das Speichermodell von LINUX in Schichten ein, bildet die Speicherseitenreservierung (*Page-Frame-Management*) die unterste Schicht. Sie arbeitet direkt auf den Speicherseiten ohne Caching oder zusätzliche Strukturierungen. Aus diesem Grund entspricht die Größe des angeforderten Speicherbereiches immer einem Vielfachen einer Seite.

Alle dem System zur Verfügung stehenden Speicherseiten werden in einer Tabelle verwaltet. Ein Tabelleneintrag ist eine doppelt verkettete Ringliste von Speicherseiten einer Größe. Insgesamt sind sechs verschiedene Größen definiert von vier bis 128 Kilobyte. Wird ein neuer Speicherbereich angefordert, wird zunächst der Tabelleneintrag mit der entsprechenden Größe durchsucht. Enthält er keinen freien Bereich, wird der darauffolgende Tabelleneintrag durchsucht. Dieser enthält Speicherbereiche der doppelten Größe. Konnte hier ein freier Bereich lokalisiert werden, wird dieser geteilt und aus dem Tabelleneintrag entfernt. Die beiden neuen Bereiche werden nun in dem darunter liegenden Eintrag eingeordnet. Der rekursive Algorithmus durchläuft die Tabelle solange, bis ein freier Bereich gefunden oder der größte Bereich durchsucht wurde. Beim Freigeben eines Bereiches wird versucht, möglichst große zusammenhängende Bereiche wiederherzustellen. Die Implementierung stellt sicher, dass niemals zwei aufeinanderfolgende Speicherblöcke frei sind, die zu einem größeren Block zusammengefügt werden könnten. Dieses Verfahren wird von Donald E. Knuth in [5] als *Buddy System* bezeichnet.

Der Mechanismus ist die Grundlage für die verbleibenden zwei Möglichkeiten zum Anfordern von Speicherbereichen. Er kann aber auch direkt verwendet werden und hat den Vorteil, dass er nur die notwendigsten Operationen durchführt, um einen Speicherbereich bereitzustellen. Die Hauptanwendung liegt beim Anfordern von nur kurzzeitig benötigten Puffern, beispielsweise um einen Dateinamen für weitere Operationen aus dem Nutzerspeicher in den Kernel-Bereich zu kopieren. Allerdings ist der kleinste Speicherbereich vier Kilobyte groß, was bei vielen Anwendungsfällen, bei

denen Speicher für eine Struktur oder einen kleinen Puffer angefordert wird, eine unnötige Verschwendung bedeuten würde.

Slab-Allokation

Der Slab-Allokation-Algorithmus wurde ursprünglich von *Sun Microsystems* für das Betriebssystem *Solaris 2.4* entwickelt und ist seit der Version 2.2 Bestandteil von LINUX. Er bietet folgende Vorteile:

- der angeforderte Speicher kann mittels eines Konstruktors initialisiert werden,
- Speicherseiten aus den Slab-Cache werden nur freigegeben, wenn sie wirklich benötigt werden,
- die Größe der Objekte ist nicht auf ein Vielfaches von 4096 Byte beschränkt, sondern kann zwischen 0 und 131072 Bytes frei gewählt werden,
- es werden spezielle Strukturen und Algorithmen zur Unterstützung des Hardware-Caching eingesetzt.

Für das Einrichten eines Slab-Caches sowie das Anfordern und das Freigeben von Objekten existieren spezielle Funktionen, die erst im letzten Schritt auf das Page-Frame-Management zurückgreifen. Der Slab-Cache implementiert ein eigenes Speichermanagement, welches fast vollständig unabhängig vom Page-Frame-Management ist.

Die Implementierung von LINUX stellt bereits vordefinierte Slab-Caches zur Verfügung, die auch in Modulen verwendet werden können. Mittels der Funktion `kmalloc(...)` kann ein Objekt aus einem der Caches angefordert werden, dabei wird dieses automatisch aus dem Cache mit dem kleinstmöglichen Speicherbereich genommen. Bereits vordefiniert sind Bereiche von 32 Byte bis 128 Kilobyte. Ein Großteil der Implementierung von LINUX verwendet diese Caches, auch wenn unter Umständen nicht der ganze Bereich belegt wird. Nur in Ausnahmefällen, wenn absehbar ist, dass sehr viele Objekte eines bestimmten Typs benötigt werden, wird ein zusätzlicher Cache definiert. So existieren für die Strukturen `sk_buff`, `inode`, `file` oder `vm_area` eigene Slab-Caches.

Nicht-zusammenhängende Speicherbereiche

Die bisher vorgestellten Möglichkeiten zur Bereitstellung von Speicherbereichen suchen immer nach einem zusammenhängenden Bereich der entsprechenden Größe innerhalb des eingeblendeten Systemspeichers. Auf Systemen mit einer längeren Laufzeit führt dies zwangsläufig zu einer Fragmentierung des Speichers. Für große Speicherbereiche, auf die nur selten ein Zugriff erfolgt, wird mittels der Funktion `vmalloc(...)` ein weiterer Mechanismus angeboten, der eine Fragmentierung des Speichers verhindert.

Hierzu wird, wie in Abbildung 2 dargestellt, ab der Adresse `VMALLOC_OFFSET` nach einem freien zusammenhängenden Speicherbereich innerhalb des Kernel-Segments gesucht. Zwischen jedem dieser Bereiche wird, um einen grundlegenden Schutz vor Speicherüberläufen zu gewährleisten, eine Lücke von vier Kilobyte gelassen. Wurde ein freier Bereich der angeforderten Größe gefunden, wird der notwendige physikalische Speicher mithilfe des Page-Frame-Managements bereitgestellt. Hierzu werden ausschließlich vier Kilobyte große Blöcke verwendet, so dass eine Fragmentierung des Speichers ausgeschlossen werden kann. Diese virtuellen Speicherbereiche werden als *vmalloc-Areas* bezeichnet.

Mithilfe dieses Mechanismus ist es möglich, größere Speicherbereiche mit mehr als 128 Kilobyte anzufordern. Allerdings ist der Zugriff auf diese Bereiche bei weitem nicht so performant wie der Zugriff auf den übrigen Kernel-Speicher. Diese Art Speicher anzufordern, wird im Kernel nur verwendet, wenn auf den Bereich nur selten zugegriffen, beziehungsweise wenn er für längere Zeit

nicht freigegeben wird. Beispielsweise wird der Objektcode von Modulen (siehe Abschnitt 2.2) in mittels `vmalloc(...)` allokierte Speicherbereiche abgelegt.

2.2 Ladbare Kernel-Module

Mit jeder neuen Version von LINUX nimmt der Umfang des Programmcodes zu. Dies geschieht sowohl durch die kontinuierliche Weiterentwicklung der integralen Bestandteile als auch durch die ständige Erweiterung des Funktionsumfangs. Durch die monolithische Struktur von LINUX müssten all diese neuen Komponenten fest in den Kern eingebunden werden. Dies würde dazu führen, dass bei jeder Änderung der Konfiguration der Kern neu übersetzt und installiert werden müsste und selten benötigte Gerätetreiber permanent im Speicher gehalten werden müssten. Diese und andere Probleme haben zur Einführung von Kernel-Modulen geführt.

2.2.1 Was sind Module?

Module sind im wesentlichen nichts anderes als Objektdateien mit einer Menge von Funktionen, die zur Laufzeit dynamisch zum Kern hinzugefügt oder aus diesem entfernt werden können. Der Objektcode wird durch ein Nutzerprogramm, den *Modul-Loader*, gleichberechtigt in den bereits laufenden Kern integriert. Dies bedeutet, dass der Programmcode des Moduls wie der Kernel-Code im Systemmodus ausgeführt wird und damit alle Rechte innerhalb des Systems besitzt.

Damit besteht nicht mehr die Notwendigkeit, alle Komponenten bereits beim Übersetzen des Kerns zu integrieren. Selten benötigte Gerätetreiber, Dateisysteme oder Netzwerkprotokolle können je nach Bedarf in den Kern geladen und wieder entfernt werden. Seit der Version 2.2 enthält der Kern einen Mechanismus zum automatischen Laden von Modulen, so dass ein Nutzer nicht vor jedem Zugriff auf ein bestimmtes Gerät das notwendige Modul laden muss. Dies ist besonders nützlich bei vorgefertigten LINUX-Distributionen wie SuSE, RedHat oder Debian, wo ein Großteil der Nutzer nicht unbedingt umfangreiche Unix- beziehungsweise Hardware-Kenntnisse mitbringt.

2.2.2 Integration in den Kern

Das Laden von Kernel-Modulen erfolgt immer durch ein Nutzerprogramm. Unter LINUX heißt dieses Programm normalerweise `insmod` und ist integraler Bestandteil jeder Distribution. Registriert der Kern, dass ein Modul nachgeladen werden muss, initiiert er die Ausführung des Nutzerprogramms. Er selbst ist nicht in der Lage, das Modul zu laden.

Der Modul-Loader holt sich zunächst den Inhalt der Objektdatei in seinen Speicher. Der Programmtext wird so übersetzt, dass dessen spätere Position frei wählbar ist. Dies ist notwendig, da nicht sichergestellt werden kann, dass ein Modul immer an die gleiche Stelle geladen wird. Die spätere Position des Moduls ermittelt der Modul-Loader mittels des Systemrufs `sys_create_module(...)`. Die Funktion bekommt den Namen des Moduls und dessen Größe übergeben und allokiert mittels der Funktion `vmalloc(...)` einen entsprechend großen Speicherbereich im Kernel-Segment. Die Startadresse des Bereiches wird als Ergebnis des Systemrufs zurückgegeben. Anschließend beginnt der Modul-Loader mit der *Relocation* des Objektcodes. Dabei werden alle absoluten Adressen im Objektcode des Moduls so verändert, dass sie der neuen Startadresse des Moduls entsprechen. Darüber hinaus werden alle unaufgelösten Referenzen des Moduls identifiziert und aufgelöst. Hierbei handelt es sich um Verweise auf Symbole, die nicht im Modul selbst implementiert sind. Eine genauere Beschreibung hierzu erfolgt im nächsten Abschnitt.

Konnten alle Symbole aufgelöst werden, wird mittels der Funktion `sys_init_module(...)` der Objektcode in den Kernel-Speicher kopiert und die `init`-Funktion des Moduls aufgerufen. Hierbei handelt es sich neben der `cleanup`-Funktion um eine der Funktionen, die jedes Modul enthalten muss. Die `init`-Funktion sollte zur Integration der Modulfunktionen in den Systemkern verwendet werden.

Sie ist neben der `cleanup`-Funktion die einzige Funktion des Moduls, die vom Kern direkt aufgerufen werden kann. Wurde die `init`-Funktion erfolgreich abgearbeitet, ist das Laden des neuen Moduls abgeschlossen.

2.2.3 Zugriff auf Symbole

Ein Modul ist eine Erweiterung des Systemkerns und keine eigenständige Komponente. Während der Kern ohne Module lauffähig sein muss, geht ein Modul immer davon aus, dass Funktionen, die es nicht selbst enthält, im Systemkern vorhanden sind. Damit ein Modul auf Funktionen des Systemkerns zugreifen kann, müssen diese exportiert werden. Andererseits erweitert ein Modul den Funktionsumfang des Systemskerns, demzufolge muss ein Modul eigene Funktionen oder Variablen beim Kern anmelden können.

Exportieren von Symbolen

Möchte eine Funktion eines Moduls auf eine Funktion oder eine globale Variable des Systemkerns oder eines anderen bereits geladenen Moduls zugreifen, so muss die Funktion oder die Variable zuvor exportiert worden sein. Dies muss bereits im Quellcode durch das `EXPORT_SYMBOL`-Makro[5] initiiert werden.

Das `EXPORT_SYMBOL`-Makro erwartet den Namen des Symbols und legt ein statisches `char`-Array für diesen und ein Objekt vom Typ `modul_symbol` an. Das Objekt wird mit der Adresse des zuvor generierten `char`-Arrays und der Adresse des Symbols initialisiert. Damit die einzelnen Objekte später zur Laufzeit wiedergefunden werden können, werden sie in einer separaten Sektion der Objektdatei namens `ksymtab` abgelegt. Die Sektion wird durch die beiden Variablen `__start___ksymtab` und `__stop___ksymtab` begrenzt. Dazwischen befindet sich alle acht Byte ein Symbol, so dass die Sektion zur Laufzeit leicht ausgelesen werden kann. Jedes exportierte Symbol wird in die Symboltabelle des Systemkerns einsortiert, die mittels der Funktion `sys_get_kernel_syms(...)` ausgelesen werden kann.

Immer wenn ein neues Modul geladen werden soll, lädt der Modul-Loader mittels des Systemrufs `sys_get_kernel_syms(...)` die Symboltabelle des Kerns und versucht alle unaufgelösten Referenzen des neuen Moduls mittels der in der Tabelle enthaltenen Einträge aufzulösen. Kann er ein Symbol nicht in der Tabelle finden, wird das Laden des Moduls abgebrochen, da es nicht möglich ist, Programmcode mit nicht aufgelösten Referenzen zu laden. Symbole, die ein Modul exportiert, werden der Symboltabelle des Kerns hinzugefügt. Hierzu liest der Modul-Loader die `ksymtab`-Sektion der Objektdatei ein und generiert eine Symboltabelle für das neue Modul. Die Tabelle wird dem Kern zusammen mit dem Objektcode übergeben. Der Systemkern übernimmt die Einträge der Symboltabelle in seine eigene Tabelle und ermöglicht damit anderen Modulen die Nutzung der neu hinzugefügten Symbole. Verwendet ein Modul Funktionen eines anderen Moduls, so werden diese als *Stacked-Moduls* bezeichnet.

Anmelden von Symbolen

Damit der Kern auf Symbole eines Moduls zugreifen kann, müssen diese mittels einer Registrierfunktion angemeldet werden. Eine Registrierfunktion erwartet immer die Adresse eines Objektes mit den anzumeldenden Symbolen. Die Initialisierung des Objektes und der Aufruf der Registrierfunktion muss vom Modul initiiert werden, dies geschieht meist innerhalb der `init`-Funktion.

---

[5] Makros sind Abkürzungen im Programmtext, hinter denen sich längere beziehungsweise schwer einzugebende Texte verbergen.

Nahezu jede funktionale Einheit des Systemkerns verfügt über eine eigene Registrierfunktion und eine globale Liste mit bereits registrierten Objekten. Wird ein Modul entfernt, müssen spätestens in der `cleanup`-Funktion alle registrierten Objekte abgemeldet werden, da anderenfalls unaufgelöste Referenzen im Kern verbleiben und dessen Verhalten beim Aufruf einer solchen Referenz zum Absturz des Systems führen kann.

2.3 Erweiterung der Speichersegmentierung

Wie bereits in der Einleitung erwähnt wurde, führte die Einführung der LKMs nicht nur zur Erweiterung der Funktionalität von LINUX, sondern öffnete zudem ein Sicherheitsloch, welches auf frei konfigurierbaren Endsystemen nur schwer zu schließen ist. Jedes Modul, welches durch den Administrator oder automatisch geladen wird, stellt eine potentielle Bedrohung für die Integrität des Systems dar. Der Programmcode eines Moduls wird gleichberechtigt in den Systemkern eingefügt. Dabei wird meist nur überprüft, ob das Modul dem privilegierten Unix-Account `root` gehört.

Ein wesentliches Hindernis bei der Lösung des Problems ist die Nutzung von nur zwei Sicherheitsstufen durch den Kern. LINUX unterstützt nur den *SuperVisor*- und den *User*-Modus der Paging-Architektur. Zusätzliche Sicherheitsstufen, wie sie beispielsweise bei der Segmentierung des Speichers durch die x86-Architektur zur Verfügung gestellt werden, werden aus Portabilitätsgründen von LINUX nicht verwendet. Die in [1] vorgestellte Architektur nutzt die zusätzlichen Mechanismen der Segmentierungseinheit, um Module in einem geschützten Bereich abzulegen, so dass der Systemkern durch diese nicht beeinflusst werden kann. Hierzu wird, wie in Abbildung 3 dargestellt, ein zusätzliches Segment innerhalb des Kernel-Speichers eingeführt, bei welchem das DPL auf eins gesetzt wird. LKMs werden nun nicht mehr direkt in den Kern integriert, sondern in dem neu generierten Segment abgelegt. Damit kann verhindert werden, dass Funktionen eines Moduls ungehindert auf den Systemkern zugreifen können. Eine Interaktion mit dem Kern kann nur noch über eine wohldefinierte Schnittstelle erfolgen.

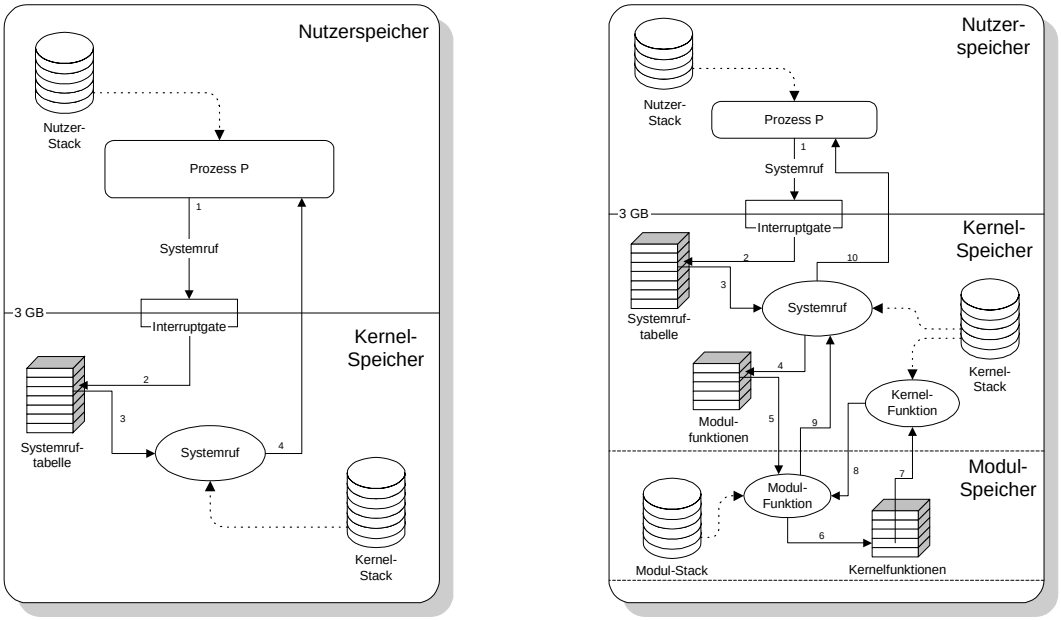

Abbildung 3: Vergleich zwischen der traditionellen Kernel-Struktur und der Erweiterung

Abbildung 3 zeigt den Unterschied zwischen dem traditionellen Ansatz von LINUX und der sicheren Kernel-Erweiterung. Der Aufruf einer Systemfunktion erfolgt bei LINUX immer über ein Interruptgate. Alle Systemrufe werden in einer globalen Tabelle verwaltet, wobei sie über ihren Index adressiert

werden. Ruft ein Nutzerprozess eine Systemfunktion auf, so wird zunächst der Index der Funktion im Register `eax` abgelegt. Die zugehörigen Parameter sind abhängig vom Systemruf und werden meist durch Bibliotheksfunktionen in die entsprechenden Register abgelegt. Nachdem der Funktionsindex und die Parameter gesichert wurden, generiert der Prozess einen Software-Interrupt `0x80`. Die Behandlung eines Interrupts erfolgt, wie bereits eingeführt, immer im Systemmodus. Beim Wechsel in diesen Modus werden die allgemeinen Register und die Segmentregister des Nutzerprozesses gesichert und der Stack auf den Kernel-Stack-Frame gesetzt. Anschließend kann die Systemfunktion aufgerufen werden. Der Index der Funktion und die notwendigen Parameter werden aus dem gesicherten Registersatz ausgelesen. Nachdem die Systemfunktion abgearbeitet wurde, kehrt der Prozessor in den Nutzermodus zurück. Hierzu wird der ursprüngliche Zustand der Register wiederhergestellt und der Rückgabewert der Systemfunktion im Register `eax` abgelegt. Die sichere Kernel-Erweiterung nutzt den gleichen Mechanismus, indem sie ein zusätzliches Segment mit einem separaten Stack-Frame generiert und dessen DPL auf eins setzt. Das Segment und der Stack-Frame werden angelegt, sobald das erste Modul geladen wird. Alle weiteren Module werden durch neue Einträge in der Modulfunktionstabelle adressiert, bekommen aber kein eigenes Segment zugeordnet, so dass die Module in der Erweiterung nicht voreinander geschützt sind. Funktionen eines Moduls können nicht direkt durch einen Nutzerprozess aufgerufen werden. Dies erfolgt immer aus einer Kernel-Funktion heraus. Der Aufruf einer Kernel-Funktion durch eine Funktion eines Moduls erfolgt analog dem Aufruf einer Systemfunktion durch einen Nutzerprozess. Hierzu wird innerhalb des Modul-Segmentes eine Tabelle mit den zugänglichen Kernel-Funktionen angelegt.

Das vorgestellte Konzept schützt den Systemkern vor unbefugten Zugriffen durch LKMs. Es kann allerdings nicht verhindert werden, dass ein anderes Modul oder ein Nutzerprozess beeinflusst wird. Deren Speicherbereiche befinden sich immer noch im Zugriffsbereich eines Moduls. Schützenswerte Daten müssen demzufolge immer im Kernel-Segment abgelegt werden. Die in der Einleitung gestellte Zielstellung - Sichere Speicherbereiche bereitzustellen, die nur von bestimmten Funktionen genutzt werden können - kann mit diesem Konzept nicht erfüllt werden. Hinzu kommt, dass das Konzept auf die x86-Architektur beschränkt ist und eine Plattform-unabhängige Implementierung nicht erstellt werden kann.

2.4 L^{4}LINUX

Es ist nicht möglich einen Systemkern in seinen Rechten zu beschränken, wenn er im *SuperVisor*-Modus betrieben wird. Wie der vorangegangene Abschnitt zeigt, ermöglicht die Verlagerung von LKMs in andere System-Level nur einen Schutz des Systemkerns vor unbekannten Modulen. Eine generelle Trennung der Komponenten wäre nur realisierbar, wenn diese in separaten Adressräumen ablaufen. Dies ist allerdings nur dann sinnvoll, wenn die Rechte und Privilegien des Systemkerns ebenfalls eingeschränkt werden. Bei der L^{4}LINUX-Architektur wird LINUX als Server auf einem minimalen Mikrokern betrieben. Alle Server des Mikrokerns laufen im *User*-Modus und besitzen damit innerhalb des Systems nur die ihnen zugedachten Rechte. Wie [2] zeigt, treten dabei keine signifikanten Leistungsverluste (unter 10 %) auf.

Die L^{4}LINUX-Architektur besteht aus zwei Komponenten: dem Mikrokern und dem L^{4}LINUX-Server. Der L^{4}LINUX-Server ist eine Portierung des Standard-LINUX-Kerns auf die L4-Architektur. Die aktuelle Implementierung verwendet den Fiasco-Mikrokern [8]. Hierbei handelt es sich um eine zum L4-Kern Schnittstellen-kompatible Implementierung eines Mikrokerns der Technischen Universität Dresden.

2.4.1 Der Fiasco-Mikrokern

In diesem Abschnitt erfolgt eine Beschreibung des L4-Mikrokerns. Dabei wird zunächst auf die Basisfunktionalität des Kerns eingegangen und anschließend die Verwaltung von Adressräumen mittels Pagern vorgestellt. Eine genaue Beschreibung hierzu ist in [3] zu finden.

Basisfunktionalität

Der Funktionsumfang des L4-Mikrokerns lässt sich auf die Unterstützung von *Threads* und Adressräumen beschränken. Ein Thread ist eine Ausführungseinheit, die immer an einen Adressraum gebunden ist. Die Kombination aus Adressraum und sein zugehöriger Thread wird als L4-Task bezeichnet. L4-Threads können mittels einer Art nachrichtenbasierten IPC, die ebenfalls vom Mikrokern bereitgestellt wird, untereinander kommunizieren. Eine L4-basierte IPC ist immer synchron. Dass heißt, entscheidet sich eine Task, eine Nachricht an eine andere Task zu senden, so muss diese eine solche Nachricht erwarten. Die sendende Task wird solange blockiert, bis ein Empfänger die Nachricht entgegennimmt oder ein zuvor definierter *Timer* abläuft.

Durch die Nutzung eines Mikrokerns ist es möglich, den Umfang des im *SuperVisor*-Modus ausgeführten Programmcodes auf die unbedingt notwendigen Komponenten zu beschränken. Nahezu alle Programmbefehle können im *User*-Modus des Prozessors ausgeführt werden. Lediglich Instruktionen, die eines privilegierten Zugriffs auf den Prozessor bedürfen, sind Bestandteil des Mikrokerns. Er allein ist jedoch kein Betriebssystem. Nur in Verbindung mit seinen Tasks kann er den kompletten Funktionsumfang eines Betriebssystems erbringen. So werden beispielsweise Interrupts als IPC-Nachrichten weitergeleitet. Die Hardware wird als eine Menge von Threads interpretiert, die spezielle Thread-Identifier besitzen und leere Nachrichten versenden. Das Übersetzen eines Interrupts in eine Nachricht ist Aufgabe des Mikrokerns. Er ist aber nicht für dessen Behandlung zuständig, da er nicht einmal die Semantik des Interrupts kennt. Hierzu wird ein Thread benötigt [14].

Adressraumverwaltung mittels Pager

Auf der Hardware-Ebene ist ein Adressraum ein Mapping, welches eine virtuelle Seite mit einem physikalischen Page-Frame verbindet oder die Seite als nicht zugreifbar markiert. Ein Adressraum definiert den virtuellen Speicher, der mit einem Thread verbunden wird. Zur Behandlung von Zugriffsverletzungen unterstützt der Mikrokern das Konzept des Pagers. Immer wenn ein Thread auf eine Seite zugreift, die momentan nicht in seinen Adressraum eingeblendet ist, sendet der Mikrokern ein Signal an den zugehörigen Pager-Thread. Ein Pager ist ein Thread, der entweder im Adressraum des verursachenden Threads oder in einem anderen Adressraum läuft. Zur Behandlung der Zugriffsverletzung wird der Pager versuchen, die angeforderte Seite in den Adressraum des Threads einzublenden. Gehört die adressierte Seite dem Thread nicht, leitet der Pager die Zugriffsverletzung an den Thread weiter.

Mikrokern-Operationen

Der L4-Mikrokern unterstützt ein hierarchisches Management von Adressräumen, das heißt der Adressraum eines Pagers kann wiederum von einem anderen Pager verwaltet werden. Der initiale Adressraum wird als σ_0 bezeichnet und beim Starten des Systems erstellt. Dieser enthält den gesamten physikalischen Speicher des Systems und wird von einem Thread namens `sigma0` verwaltet. Alle weiteren Adressräume werden mittels der Mikrokern-Operationen `grant`, `map` und `unmap` aufgebaut.

`grant`: Ein Thread kann eine oder mehrere Seiten seines Adressraumes einem anderen Thread zuweisen. Eine zugewiesene Seite wird aus dessen Adressraum entfernt. Damit ein Thread diese Operation ausführen kann, muss die Seite bereits in seinen Adressraum eingebunden sein.

`map`: Ein Thread kann jede Seite seines Adressraums in den Adressraum eines anderen Threads einblenden. Anschließend können beide Threads auf diese Seite zugreifen. Die Seite wird nicht aus dem Adressraum des einblendenden Threads entfernt. Zur Ausführung der Operation muss die Seite, wie beim Zuweisen einer Seite, für den Thread zugreifbar sein.

`unmap`: Ein Thread kann jederzeit eine seiner Seiten aus allen fremden Adressräumen ausblenden. Die ausgeblendete Seite ist daraufhin nur noch in seinem Adressraum zugreifbar.

Alle anderen Adressräume, die diese Seite direkt oder indirekt über einen anderen Adressraum empfangen haben, können anschließend nicht mehr auf diese Seite zugreifen.

Abbildung 4 zeigt ein Beispiel für die beschriebenen Operationen. Der Adressraum A_0 blendet eine seiner Seiten in den Adressraum A_1 ein, dieser wiederum blendet die gleiche Seite in den Adressraum A_2 ein. Anschließend arbeiten alle drei Adressräume auf der gleichen Seite. Wenn A_0 die Seite ausblendet, wird sie sowohl aus dem Adressraum A_1 als auch A_2 entfernt und kann daraufhin nur noch in A_0 verwendet werden.

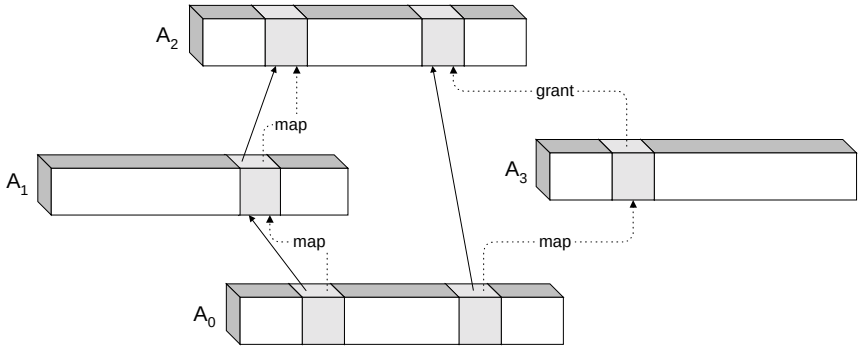

Abbildung 4: L4-Mikrokern Operationen `grant` und `map`

Die Abbildung zeigt außerdem die `grant`-Operation. Adressraum A_0 blendet eine seiner Seiten in den Adressraum A_3 ein, welcher wiederum die Seite dem Adressraum A2 zuweist. Als Resultat kann in den beiden Adressräumen A_2 und A_0 auf die Seite zugriffen werden.

Hierarchische Pager

Die Operationen `grant`, `map` und `unmap` werden innerhalb des Mikrokerns ausgeführt und bilden die Grundlage für das Konzept des Pagers. Im einfachsten Fall verwaltet ein Pager genau einen Adressraum und verwendet hierzu Seiten aus seinem eigenen Adressraum. Bei den verwendeten Seiten handelt es sich immer um virtuelle Seiten, der physikalische Speicher wird vom `sigma0`-Thread verwaltet.

Durch die Verwendung von virtuellen Seiten ist es möglich, dass der Adressraum eines Pagers wiederum durch einen anderen Pager verwaltet wird. In Abbildung 5 wird der Adressraum von Pager P_1 durch den Pager P_0 verwaltet. Der Adressraum der Anwendung App wird vom Pager P_1 verwaltet, so dass eine Zugriffsverletzung zunächst Pager P_1 geschickt wird und dieser für das Einblenden der zugehörigen Seite verantwortlich ist. Solange alle Seiten des Adressraumes von P_1 bereits einblendet sind, bleibt der Pager P_0 unberührt.

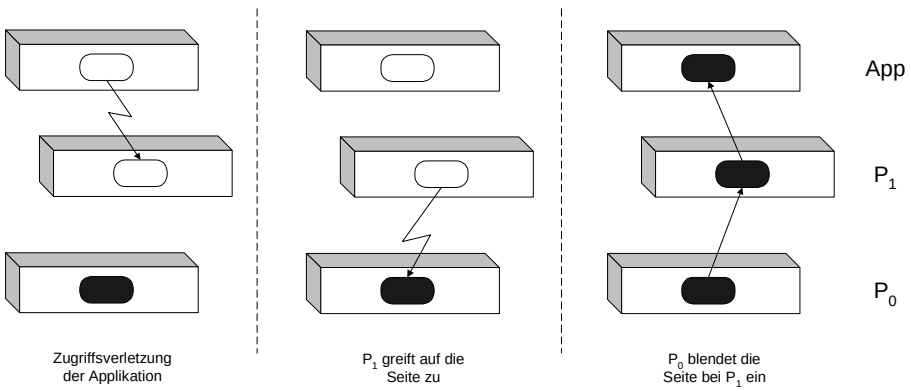

Abbildung 5: Hierarchische Pager

Schwieriger wird die Situation, wenn Seiten aus dem Adressraum von P_0 ausgelagert sind oder noch nie eingeblendet wurden. In diesem Fall würde der Pager P_1 eine Seite in den Adressraum von App einblenden, auf die er nicht zugreifen darf. Die Richtlinien des Mikrokerns erlauben zwar das Entziehen von Rechten beim Einblenden von Seiten. Jedoch ist es nicht möglich, Rechte weiterzugeben, die man selbst nicht besitzt. Somit würde in diesem Fall die Seite für die Applikation ebenfalls nicht zugreifbar sein, so dass die Zugriffsverletzung zwar behandelt, aber nicht korrekt aufgelöst wurde. Um dies zu verhindern, muss der Pager dafür sorgen, dass alle Seiten, die er an einen anderen Adressraum weitergibt, bereits bei ihm selbst eingeblendet sind. Dies erreicht er, indem er auf die Seiten vor dem Einblenden selbst zugreift. Der Zugriff verursacht eine Zugriffsverletzung, die an den Pager P_0 weitergeleitet wird. P_0 wird daraufhin die Seite in den Adressraum von P_1 einblenden. Anschließend kann P_1 die Seite an die Applikation App weitergeben.

2.4.2 Der L⁴LINUX-Server

Der L⁴LINUX-Server ist eine Portierung eines klassischen, monolithischen Unix-Kerns auf die L4-Architektur. Dabei wurde im Wesentlichen nur der architekturabhängige Teil von LINUX geändert, so dass sowohl Nutzerprogramme als auch der Unix-Kern selbst im Nutzermodus ausgeführt werden können. In diesem Abschnitt erfolgt eine kurze Vorstellung der grundlegenden Design-Merkmale des Servers, wobei der Großteil der Erläuterung aus [2] und [6] entnommen wurde.

Speicherverwaltung

Im Abschnitt 2.1.2 wurde einleitend erläutert, dass LINUX den gesamten physikalischen Speicher in das Kernel-Segment oberhalb von drei Gigabyte einblendet. Durch die Umsetzung von LINUX als Task auf den L4-Kern ist ein direkter Zugriff auf den Systemspeicher nicht mehr möglich. Ebenso laufen Nutzerprozesse und externe Treiber in einem separaten Adressraum. Um die Implementierung von LINUX nicht völlig neu gestalten zu müssen, musste ein ähnlicher Mechanismus zur Verwaltung der Adressräume implementiert werden. Hierzu bot sich das Konzept der hierarchischen Pager an. LINUX fordert bei seiner Initialisierung den physikalischen Speicher von der `sigma0`-Task an. Die Verwaltung des Kernel-Bereiches wird von einem externen root-Pager übernommen. Der Speicher, der von LINUX erzeugten Nutzerprozesse, wird jedoch vom LINUX-Server selbst organisiert. Daraus ergibt sich das in Abbildung 6 dargestellte Speicher-*Mapping*.

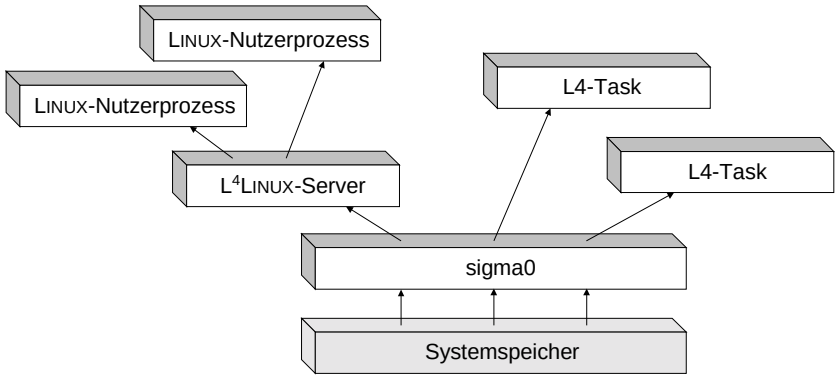

Abbildung 6: Speicher-Mapping

Obwohl der LINUX-Server nicht auf den physikalischen Speicher zugreifen kann, wurde die ursprüngliche Implementierung des Speichermanagements beibehalten. Damit ergibt sich zwar eine doppelte Speicherverwaltung, jedoch haben Benchmark-Tests gezeigt, dass sich die Leistungsverluste in einem akzeptablen Rahmen halten. Der wesentliche Vorteil bei dieser Vorgehensweise ist, dass lediglich Änderungen an der architekturabhängigen Implementierung von LINUX vorgenommen werden mussten.

Task- und Thread-Struktur

Die Basisarchitektur von L^{4}LINUX besteht aus wenigen L4-Threads, die in einem gemeinsamen Adressraum ablaufen. Alle Kernel-Dienste werden durch einen einzigen Thread ausgeführt. Für die Behandlung von Interrupts stehen jeweils dedizierte Threads zur Verfügung. Damit kann die zuvor beschriebene Architektur von L4 realisiert werden. In den meisten Fällen nimmt der jeweilige Interrupt-Thread den Interrupt nur entgegen und leitet dessen zeitunkritische Teile der Behandlung an einen anderen Thread weiter. Für jeden Nutzerprozess werden zwei Threads erzeugt. Der eine Thread ist für die Ausführung der primären Nutzeraktivitäten verantwortlich. Der zweite Thread dient der Zustellung von Signalen.

Systemrufe

Systemrufe werden mittels eines Trampolin-Mechanismus an den LINUX-Server weitergeleitet. Der Software-Interrupt `0x80` kann durch den L4-Kern nicht behandelt werden, so dass dieser eine Ausnahme generiert und an den verursachenden Thread zurücksendet. Eine Ausnahmebehandlungsroutine im Nutzer-Thread erkennt, dass es sich um einen Systemruf handelt und leitet diesen per IPC an den LINUX-Server weiter. Da die IPC immer synchron ist, wird der Nutzer-Thread solange blockiert, bis der Systemruf vollständig abgearbeitet wurde.

Die Einführung des Trampolin-Mechanismus macht es möglich, dass alle Nutzerprogramme der ursprünglichen LINUX-Implementierung in der L4-Umgebung ohne Änderungen eingesetzt werden können. Ein Leistungsgewinn könnte durch die Verwendung einer modifizierten C-Bibliothek, die den Systemruf direkt an den LINUX-Server weiterleitet, erreicht werden.

Signale

Die Zustellung von Signalen erfolgt über einen zweiten Nutzer-Thread. In der ursprünglichen LINUX-Implementierung werden Signale durch Modifikation des Nutzer-Stacks nach der Rückkehr vom Kernel- in den Nutzermodus zugestellt. Der Timer-Interrupt von LINUX bedingt einen stetigen Wechsel in den Kernel-Modus, so dass Signale mit einer geringen Latenz zugestellt werden. Um einen ähnlichen Mechanismus in der L4-Umgebung zu gewährleisten, wird der Signal-Thread über das zuzustellende Signal informiert. Dieser Thread erzwingt daraufhin den Eintritt in den Kernel-Modus des sich im Nutzermodus befindlichen Nutzer-Threads. Bei der Rückkehr von diesem Systemruf werden die fälligen Signale zugestellt.

Interrupts

Die Behandlung von Interrupts erfolgt nicht im L4-Kern, hierzu sind Nutzer-Threads notwendig. Beim Eintreffen eines Interrupts generiert der L4-Kern eine IPC-Nachricht für den zugehörigen Nutzer-Thread, welcher die Interrupt-Behandlungsfunktion implementiert. Die Behandlung von Interrupts erfolgt bei LINUX immer in zwei Schritten: Zunächst wird ein sogenannter *Top Half* ausgeführt. Dieser bestätigt meist nur das Eintreffen des Interrupts und markiert den zugehörigen *Bottom Half* zur Bearbeitung des Interrupts. Der Bottom Half wird zu einem späteren Zeitpunkt innerhalb eines separaten Threads ausgeführt und enthält den zeitunkritischen Teil der Interrupt-Behandlung.

Durch eine gestaffelte Vergabe von Prioritäten für die einzelnen Threads wird die LINUX-Semantik eingehalten. Die Top-Half-Threads bekommen die höchste Priorität und können somit nur durch einen weiteren Top Half unterbrochen werden. Bottom-Half-Threads erhalten eine höhere Priorität als der LINUX-Thread, so dass sichergestellt wird, dass sie vorrangig ausgeführt werden und damit die Behandlung des Interrupts nicht zu lange verzögert wird.

Scheduling

Das Scheduling erfolgt bei LINUX mittels der Funktion `schedule(...)`, die entweder bei der Abarbeitung eines blockierenden Systemrufs oder nach Ablauf der Zeitscheibe von 10 ms aufgerufen wird. Durch das stetige Eintreffen von Timer-Interrupts wird garantiert, dass ein Nutzer-Thread das System nicht blockieren kann.

Zur Durchsetzung von verschiedenen Scheduling-Strategien auf der Nutzerebene ist in L4 ein *Preemption Handler*-Mechanismus vorgesehen. Zum gegenwärtigen Zeitpunkt ist dieser Mechanismus jedoch noch nicht einsatzbereit, so dass das Scheduling momentan auf der Mikrokern-Ebene durchgeführt wird. Die `schedule`-Funktion des LINUX-Servers realisiert nur ein Multiplexen der LINUX-Server-Threads. Das Umschalten zwischen den einzelnen L4-Tasks erfolgt durch den L4-Kern.

2.4.3 Modularisierung des L^{4}LINUX-Servers

Obwohl LINUX als Server des Mikrokerns ausgeführt wird, bleibt dessen monolithische Struktur erhalten. LKMs werden wie bei der ursprünglichen Implementierung mittels eines Modul-Loaders in den Kernel-Speicher geladen. Somit haben Funktionen eines Moduls die gleichen Rechte wie der Systemkern. Die bedeutet, dass ein zusätzlicher Schutz weder für Datenbereiche des Systemkerns noch für Bereiche eines Moduls gegeben werden kann. Damit haben LKM-Rootkits, die in den Adressraum des LINUX-Servers geladen werden, auch die gleichen Möglichkeiten, das System zu kompromittieren.

Um dies wirksam unterbinden zu können, muss der LINUX-Server in unabhängige Komponenten unterteilt werden, die in separaten Adressräumen ausgeführt werden. Bei dem µSina-Projekt ([11], [12] und [13]) werden Teile des Netzwerk-Stacks in einen separaten Server ausgelagert, so dass unverschlüsselte Daten in einem geschützten Bereich verarbeitet werden können. Darüber hinaus existieren verschiedene Gerätetreiber, die bereits als separate Server implementiert wurden. Bei all diesen Konzepten werden Komponenten des LINUX-Kerns in separate Server ausgelagert, so dass sie nur über eine wohldefinierte Schnittstelle angesprochen werden können.

In [16] wird ein Konzept zur generischen Portierung von Gerätetreibern auf die L4-Umgebung vorgestellt. Jedoch sich das Konzept auf die Portierung von Gerätetreibern, so dass die Bereitstellung von sicheren Speicherbereichen für Funktionen des Systemkerns nur bedingt erreicht wird. In den folgenden Abschnitten wird ein Architektur zur Auslagerung von LKMs vorgestellt, welches nur geringe Änderungen an der Implementierung des Moduls erfordert und eine Trennung der Adressräume, wie sie einleitend gefordert wurde, zur Verfügung stellt.

Kapitel 3

Ladbare Kernel-Module als separate L4-Tasks

Nachdem im vorangegangenen Abschnitt das Speichermanagement von LINUX und dessen Grenzen beschrieben wurden, erfolgt in diesem Abschnitt die Vorstellung eines Konzeptes zur Separierung des LINUX-Systemkerns. Dabei werden Komponenten des L^{4}LINUX-Servers in eigenständige L4-Tasks auslagert, so dass sie in einem eigenen Adressraum ablaufen, um deren Daten vor unautorisierten Zugriffen wirksam schützen zu können. Die IPC zwischen den einzelnen Tasks wird mittels einer ACL kontrolliert, so dass zusätzlich ein Missbrauch von öffentlichen Symbolen verhindert werden kann.

3.1 Beschreibung der Architektur

Zur Realisierung von geschützten Speicherbereichen für Funktionen des Systemkerns von LINUX wurde die L^{4}LINUX-Architektur erweitert. Dieser Abschnitt beschreibt zunächst die Schnittstellen zur Aufteilung des L^{4}LINUX-Servers, die Basis-Komponenten der Architektur sowie die Unterteilung des Speichers.

3.1.1 Auslagern von LKMs

Eine Aufteilung des Systemkerns von LINUX ist aufgrund der monolithischen Architektur nur schwer möglich. Es existieren nur wenige wohldefinierte Schnittstellen, die eine solche Trennung erlauben. Kernkomponenten, wie der Netzwerk-Stack oder das virtuelle Dateisystem, sind so stark in den Kern integriert, dass eine Auslagerung in einen fremden Adressraum nicht ohne weiteres möglich ist. Viele Komponenten werden in der aktuellen Implementierung vom Kern zwingend benötigt. Das hier vorgestellte Konzept nutzt die Modul-Schnittstelle zur Unterteilung des L^{4}LINUX-Servers. Wie in Abschnitt 2.2.3 beschrieben, muss die Interaktion zwischen dem Kern und einem Modul bereits im Quellcode spezifiziert oder mittels spezieller Funktionen beim Laden des Moduls initialisiert werden. Das Ziel der Architektur ist es, ladbare Kernel-Module in separate L4-Tasks auszulagern und die Interaktion zwischen dem Kern und einem Modul mittels IPC zu realisieren. Der Zugriff auf Variablen erfolgt durch Einblenden von Speicherbereichen in den Adressraum der jeweils anderen Task.

Für Nutzerprogramme und Funktionen des L^{4}LINUX-Servers soll die Auslagerung von Modulen weitestgehend transparent gestaltet sein. Für Nutzerprogramme ist dies vollständig möglich, da sie nur über Funktionen des L^{4}LINUX-Servers auf Funktionen eines Moduls zugreifen können. Es existiert keine direkte Schnittstelle zwischen einem Nutzerprogramm und einer Funktion eines Moduls. Selbst der Modul-Loader muss nicht angepasst werden. In der hier vorgestellten Architektur ist dieser ebenfalls für das Laden der ausgelagerten Module zuständig. Welche Probleme hierbei entstehen und wie sie gelöst werden können, wird im Abschnitt6.2.2 erläutert. Eine Unterscheidung beim Laden des Moduls wird erst innerhalb des Kerns getroffen. Der Name einer Objektdatei eines auszulagernden Moduls erhält den Präfix `modld_`, so kann innerhalb der Systemfunktionen entschieden werden, ob das Modul ausgelagert oder in den L^{4}LINUX-Server integriert werden soll.

Die Auslagerung eines LKMs in eine unabhängige L4-Task ist nicht ohne eine Erweiterung des Moduls möglich. Zum einen erwartet ein Modul, dass unbekannte Symbole im Systemkern oder in einem anderen Modul implementiert sind und diese nach dem Laden direkt aufgerufen werden können. Zum andern ist der Programmcode eines Moduls nicht so implementiert, dass er als

eigenständiger Prozess lauffähig ist. Er bietet lediglich eine Menge von Funktionen an, die vom Systemkern oder von einem anderen Modul aufgerufen werden können. Damit eine Auslagerung des Programmtextes eines Moduls in eine eigenständige L4-Task möglich wird, sind zusätzliche Funktionen notwendig, die eine Steuerung des Moduls durch eine externe Instanz und die Behandlung von IPC-Anforderungen ermöglichen.

3.1.2 Basis-Komponenten

Ausgehend von der L⁴LINUX-Architektur werden alle Systemkomponenten als Server auf dem L4-Kern betrieben. Für das hier vorgestellte Konzept ist der Aufbau der L4-Umgebung nicht von Bedeutung. Im Wesentlichen können die Basis-Komponenten, wie in Abbildung 7 dargestellt, auf den Mikrokern, den L⁴LINUX-Server, dessen Nutzerprozesse und den Modul-Server sowie die ausgelagerten Module begrenzt werden.

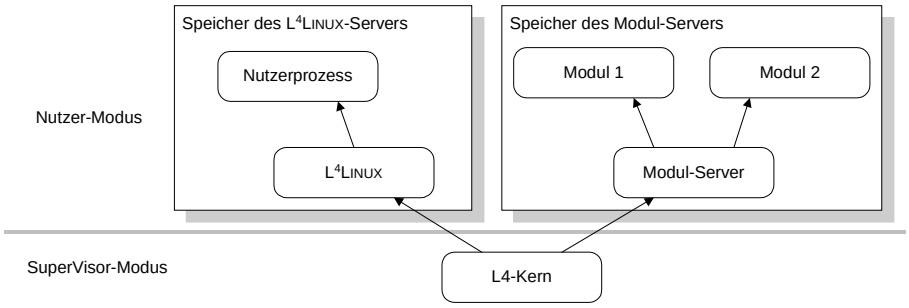

Abbildung 7: Basisarchitektur

Die einzelnen Komponenten besitzen die folgenden Eigenschaften:

L4-Kern: Als L4-Kern wird der im Abschnitt 2.4.1 beschriebene Fiasco-Kern der TU-Dresden verwendet. Änderungen am Mikrokern zur Unterstützung der hier vorgestellten Architektur sind nicht notwendig. Dies war auch eines der Designziele, um eine möglichst problemlose Integration in andere laufende Projekte gewährleisten zu können. Der L4-Kern ist die einzige Komponente des Systems, die im SuperVisor-Modus des Prozessors ausgeführt werden muss.

L⁴LINUX: Bei dem hier verwendeten L⁴LINUX-Server handelt es sich um eine Portierung des Standard-LINUX-Kernels der Version 2.2.21[6]. Zur Auslagerung von Modulen war es notwendig einige Funktionen des Kerns anzupassen. Eine ausführliche Beschreibung zu den notwendigen Änderungen erfolgt in Abschnitt 4.3.1. Der L⁴LINUX-Server wird als Task des L4-Kerns im Nutzer-Modus ausgeführt.

Modul-Server: Zur Verwaltung der einzelnen Modul-Tasks ist eine zentrale Instanz notwendig. Hierzu wurde der Modul-Server eingeführt. Er ist verantwortlich für die Bereitstellung einer Schnittstelle zum Laden von LKMs, für die Administration der einzelnen Module sowie für die Verwaltung der von den Modulen angebotenen Symbolen. Wie der L⁴LINUX-Server wird auch der Modul-Server im Nutzer-Modus ausgeführt.

Bei der Implementierung der Modul-Tasks wurde ein ähnliches Konzept wie bei den Nutzerprogrammen des L⁴LINUX-Servers verfolgt. Eine neue Modul-Task wird immer vom Modul-Server erstellt. Damit ist es möglich, beim Entfernen eines Moduls die zugehörige Task zu beenden

[6] Eine L⁴LINUX-Portierung der Version 2.4.x war zum Zeitpunkt der Erstellung der Arbeit noch nicht verfügbar. Da die Implementierung des Speichermanagements unter Umständen Auswirkungen auf die Anbindung des Modul-Servers hat, kann an dieser Stelle auch noch nicht gesagt werden, wie umfangreich eine Portierung des Konzeptes auf eine neuere Version ist.

und den angeforderten Speicher freizugeben. Die Organisation des Speichers einer Modul-Task wird ebenfalls vom Modul-Server übernommen. Eine genaue Beschreibung hierzu erfolgt im Abschnitt 3.3.

3.1.3 Unterteilung des Systemspeichers

Für die Initialisierung des Systems sind neben den in Abbildung 7 dargestellten Tasks noch zusätzliche Programme notwendig. Sie sind unter anderem verantwortlich für die Verwaltung des physikalischen Speichers sowie das Laden und Starten von L4-Prozessen. Das Laden von L4-Programmen erfolgt mittels des sogenannten *L4Loaders*. Der L4Loader wird mithilfe einer Konfigurationsdatei gesteuert, welche die zu ladenden Programme und zusätzliche Optionen enthält. Unter anderem kann an dieser Stelle festgelegt werden, wie viel Speicher einem Programm zur Verfügung gestellt werden soll.

Der verbleibende Speicher wird vom *Physical Memory Dataspace Manager* (DMphys) verwaltet und kann von beliebigen L4-Tasks über dessen öffentliches Interface angefordert werden. Die aktuelle Version des L4LINUX–Servers unterstützt noch nicht die Verwendung des DMphys. Aus diesem Grund wird der für LINUX zur Verfügung gestellte Speicher in der Konfigurationsdatei des L4Loaders vorgegeben. Der Modul-Server wurde so implementiert, dass er das vom DMphys angebotene Interface unterstützt und somit keine zusätzlichen Optionen in der Konfigurationsdatei benötigt.

Der DMphys ist verantwortlich für die dynamische Verwaltung des physikalischen Speichers. Durch die statische Vorgabe des für LINUX verwendbaren Systemspeichers wird, wie in Abbildung 7 dargestellt, der L4LINUX- und der Modul-Server in einen separaten Speicherbereich abgelegt. Die Modul-Tasks werden vom Modul-Server verwaltet und in dessen Speicherbereich integriert. Sie benötigen somit keinen eigenen physikalischen Adressbereich und kein Interface zum DMphys. Allerdings ist zu beachten, dass der dem L4LINUX-Server zur Verfügung gestellte Speicher zur Laufzeit nicht freigegeben oder verändert werden kann. Wurde für den L4LINUX-Server zuviel physikalischer Speicher bereitstellt, ist die Ausführung des Modul-Servers beziehungsweise das Laden von Modulen nicht möglich. Demzufolge muss vor dem Starten des Systems ermittelt werden, wie viel Speicher für den Modul-Server und für die ausgelagerten Module benötigt wird.

3.2 Schnittstelle zur Verwaltung von ausgelagerten Modulen

Eine der Kernaufgaben des Modul-Servers ist die Administrierung der ausgelagerten Module. Hierzu zählt das Laden und Entfernen der Module sowie die Verwaltung der Speicherbereiche. Dieser Abschnitt beschäftigt sich mit dem Laden und Entfernen der Module. Eine Beschreibung der Speicherverwaltung durch den Modul-Server erfolgt im anschließenden Abschnitt.

Bei der Entwicklung des Konzeptes wurde versucht, die Integration der ausgelagerten Module möglichst transparent zu gestalten. Aus diesem Grund bietet der Modul-Server die gleiche Schnittstelle zur Verwaltung von Modulen wie der LINUX-Kernel an. Das Laden und das Initialisieren der Module erfolgt mithilfe von zwei Funktionen, wie sie schon im Abschnitt 2.2.2 beschrieben wurden. Zum Entfernen eines Moduls wird eine weitere Funktion angeboten. Der LINUX-Kern enthält wie in der ursprünglichen Implementierung die notwendigen Funktionen zum Laden seiner eigenen LKMs. Module, die vom Modul-Server verwaltet werden und in einem eigenen Adressraum ablaufen, werden beim Laden und beim Entfernen anhand eines Namenspräfixes unterschieden. Die notwendigen Funktionen sind Bestandteil des Modul-Servers und über ein RPC-Interface erreichbar.

3.2.1 Laden eines Moduls

In der aktuellen Implementierung ist das Laden eines externen Moduls lediglich mittels des Modul-Loaders von LINUX möglich. Sie werden anhand des Präfixes `modld_` identifiziert. Damit ein Modul nicht bereits vor dem eigentlichen Laden manipuliert werden kann, müsste eine zusätzliche externe Instanz vorhanden sein. Diese ist bisher noch nicht verfügbar. Im Abschnitt 6.2.2 werden darüber

hinaus noch zusätzliche Wege aufgezeigt, wie man ein Modul sicher laden könnte. Die Umsetzung der Konzepte ist jedoch nicht Bestandteil der hier vorliegenden Arbeit, so dass im Folgenden lediglich die Vorgehensweise bei der Verwendung des Modul-Loaders von LINUX erläutert wird.

Zum Laden eines Moduls werden in der Symboltabelle des Systemkerns die beiden Einträge der Funktionen `sys_create_module(...)` und `sys_init_module(...)` durch die beiden Funktionen `modld_create_module(...)` und `modld_init_module(...)` ersetzt. Damit ist die Schnittstelle zum Modul-Server für den Modul-Loader transparent. Beide Funktionen wurden so implementiert, dass sie anhand des Namens des Moduls unterscheiden können, ob das Modul in den Adressraum des L[4]LINUX-Servers eingefügt oder dem Modul-Server übergeben werden soll. Im ersten Fall leiten sie die Operation an die ursprünglichen Funktionen des L[4]LINUX-Servers weiter. Anderenfalls werden mittels der RPC-Schnittstelle die zugehörigen Funktionen des Modul-Servers aufgerufen.

Die Umsetzung der beiden Funktionen zum Laden und Initialisieren eines Moduls im Modul-Server ist weitestgehend analog zur Implementierung der L[4]LINUX-Funktionen. Die Funktion `modld_modules_create(...)` allokiert den Speicher für den Objektcode des Moduls und sucht nach einem freien Bereich innerhalb des MODLD-Areas[7] zur Relokation des Programmcodes. Als Ergebnis gibt sie die Startadresse des gefunden Bereiches zurück, so dass der Modul-Loader die Relokation des Programmcodes durchführen kann. Zusätzlich werden die Strukturen zur Generierung der Modul-Task initialisiert. Eine neue Task wird an dieser Stelle noch nicht erzeugt, dies erfolgt erst in der Funktion `modld_modules_init(...)`. Die Funktion `modld_modules_init(...)` kopiert zunächst den Programmcode des Moduls in den zuvor allokierten Speicherbereich und überprüft den Inhalt der Modul-Struktur. Wurden keine Fehler innerhalb der Struktur gefunden, ist das Laden des neuen Moduls abgeschlossen. Anschließend kann das Modul initialisiert werden.

3.2.2 Initialisierung

Die Initialisierung eines ausgelagerten Moduls wird in zwei separate Schritte unterteilt. Dies ist notwendig, da zum einen die Infrastruktur der Modul-Task und zum anderen das Modul selbst initialisiert werden muss. Beides wird, wie in Abbildung 8 dargestellt, innerhalb der Funktion `modld_modules_init(...)` ausgeführt, so dass nach der erfolgreichen Abarbeitung der Funktion das Modul vollständig initialisiert ist und von den Funktionen des L[4]LINUX-Servers verwendet werden kann.

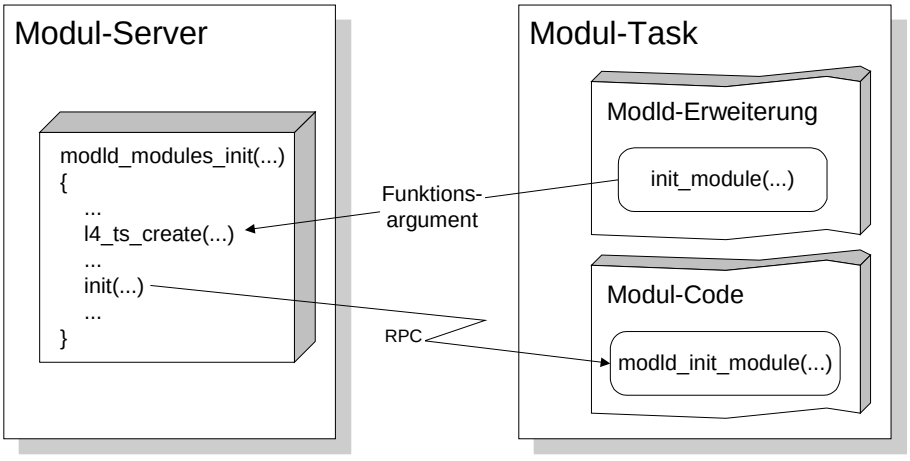

Abbildung 8: Initialisierung eines externen Moduls

[7] Das MODLD-Area ist ein gemeinsamer Speicherbereich auf den auch über Adressraumgrenzen hinaus zugegriffen werden kann.

Jedes Modul, welches als separate L4-Task verwendet werden soll, wird beim Linken durch eine `modld`-Erweiterung ergänzt. Sie enthält alle Funktionen, die zur Implementierung einer L4-Task notwendig sind. Die `init`-Funktion des Moduls muss bei ausgelagerten Modulen den Namen `modld_init_module(...)` besitzen. Die Funktion mit dem Namen `init_module(...)` wird in der `modld`-Erweiterung des Moduls implementiert. Die feste Vergabe der Namen ist notwendig, da sonst das Modul nicht korrekt initialisiert werden kann und eine Verwendung als L4-Task nicht möglich ist.

Die Funktion `modld_init_module(...)` enthält den notwendigen Programmcode zur Initialisierung des Moduls. Hierbei handelt es sich um die ursprüngliche Implementierung der `init_module(...)`-Funktion des Moduls aus der L^{4}LINUX-Implementierung. In den meisten Fällen muss sie nur unwesentlich verändert werden, um in einer separaten Task verwendbar zu sein. Aufgerufen wird die Funktion über einen RPC vom Modul-Server, nachdem die `init_module(...)`-Funktion der MODLD-Erweiterung des Moduls erfolgreich abgearbeitet werden konnte. Die notwendigen Funktionen zur Behandlung des RPCs sind Bestandteil der Erweiterung. Eine Beschreibung der RPC-Architektur erfolgt im Abschnitt 3.4, so dass hier nicht näher darauf eingegangen wird.

Der Aufruf der `init`-Funktion des Moduls ist wie in der ursprünglichen Implementierung die letzte Aktion beim Laden und Initialisieren des Moduls. Demzufolge muss sichergestellt werden, dass vor dem Aufruf der Funktion alle Komponenten der `modld`-Erweiterung vollständig initialisiert wurden. Dies ist die Aufgabe der Funktion `init_module(...)`. Die Adresse der Funktion wird vom Modul-Loader in der Modul-Struktur abgelegt, so dass sie innerhalb der Funktion `modld_modules_init(...)` bekannt ist. Beim Erstellen einer neuen L4-Task mithilfe der Funktion `l4_ts_create(...)` muss die Adresse einer Funktion angegeben werden, die als initialer *Instruction Pointer* (EIP) verwendet werden soll. Der EIP wird auf die Adresse der `init`-Funktion der MODLD-Erweiterung gesetzt, damit wird sie unmittelbar nach dem Starten der neuen Task ausgeführt. Sie sorgt für die Initialisierung der Erweiterung des Moduls. Konnte dies erfolgreich ausgeführt werden, sendet sie ein Signal an den Modul-Server, so dass dieser sicher sein kann, dass er die eigentliche `init`-Funktion des Moduls mittels eines RPCs aufrufen kann. Nach dem Senden des Signals legt sich der Thread schlafen, bis das Modul wieder entfernt wird.

3.2.3 Entfernen eines Moduls

Das Entfernen eines Moduls wird in der Regel vom Nutzer beziehungsweise durch das Programm `rmmod` initiiert. Dieses ruft die Systemfunktion `sys_delete_module(...)` des L^{4}LINUX-Servers auf. Wie für das Laden der Module, wurde der hierfür notwendige Eintrag der Funktion in der Systemruftabelle verändert, so dass zunächst die Funktion `modld_delete_module(...)` aufgerufen wird. Sie untersucht wiederum den Namen des Moduls und ruft die entsprechende Funktion des L^{4}LINUX-Servers oder des Modul-Servers auf.

Ähnlich wie das Initialisieren eines Moduls wird auch das Entfernen in zwei separate Schritte unterteilt. Zunächst muss die `cleanup`-Funktion des Moduls aufgerufen werden. Wie die `init`-Funktion muss auch sie umbenannt werden, damit sie durch die `cleanup`-Funktion der `modld`-Erweiterung ersetzt werden kann. Die `cleanup`-Funktion des Moduls erhält den Namen `modld_cleanup_module(...)` und wird mittels eines RPCs vom Modul-Server aufgerufen, sobald dieser vom L^{4}LINUX-Server dazu aufgefordert wird. Nach der Abarbeitung der `cleanup`-Funktion kann das Modul beziehungsweise die L4-Task entfernt werden. Vorher wird die `cleanup`-Funktion der `modld`-Erweiterung ausgeführt. Dies erfolgt durch Manipulieren des EIP der Modul-Task. Die Modul-Task hat sich nach der Initialisierung des Moduls schlafen gelegt, so dass der EIP des Threads manipuliert werden kann, ohne dass Seiteneffekte zu befürchten sind. Der EIP wird auf die `cleanup`-Funktion der `modld`-Erweiterung gelegt. Die Adresse der Funktion wurde beim Laden des Moduls vom Modul-Loader in die Modul-Struktur eingetragen, so dass sie dem Modul-Server bekannt ist. Das Verändern des EIP bewirkt, dass die L4-Task vom Scheduler wieder aktiviert und die `cleanup`-Funktion ausgeführt wird. Nach dem Entfernen aller Komponenten der `modld`-Erweiterung sendet sie

ein Signal an den Modul-Server. Daraufhin beendet dieser die L4-Task und gibt den zugehörigen Speicher frei.

3.3 Speicherverwaltung

Die Auslagerung der Module in separate L4-Tasks bedingt die Implementierung einer eigenen Speicherverwaltung. Um den Umfang eines Moduls nicht zu stark auszudehnen und eine zentrale Möglichkeit zur Verwaltung der einzelnen Speicherbereiche zu erhalten, wird die Speicherverwaltung für die ausgelagerten Module vom Modul-Server übernommen.

Dieser Abschnitt beschreibt zunächst die Gliederung der einzelnen Adressräume des L4LINUX-Servers, des Modul-Servers und einer Modul-Task. Anschließend wird die Allokation von Speicherbereichen erläutert. Durch die Verwendung des DMphys muss hierzu ein hierarchisches Modell implementiert werden. Der letzte Teil des Abschnittes geht noch einmal auf die Umsetzung eines der Hauptziele der Arbeit, die Bereitstellung von sicheren Speicherbereichen, ein.

3.3.1 Gliederung der Adressräume

Ausgehend von der im Abschnitt 2.1.2 erläuterten Gliederung des Adressraumes von LINUX wurde die in Abbildung 9 dargestellte Aufteilung der Adressräume des Moduls-Servers und der Modul-Task gewählt. Es sollte die Möglichkeit geschaffen werden, sowohl Speicherbereiche zwischen den einzelnen Tasks teilen zu können, als auch private Bereiche garantieren zu können.

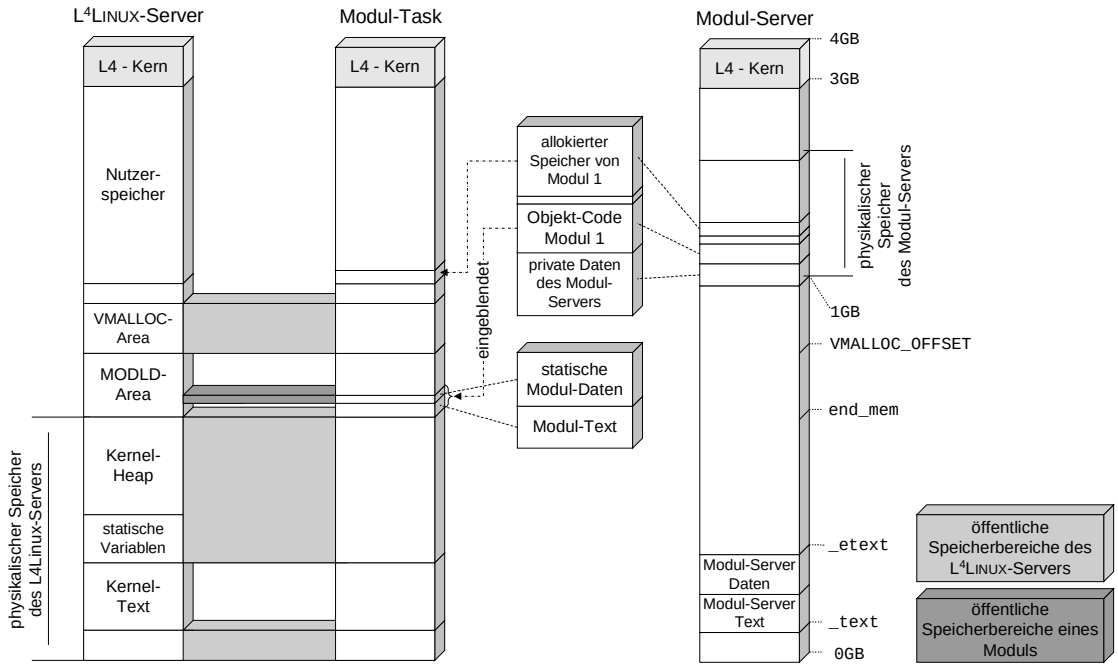

Abbildung 9: Unterteilung der Adressräume

Erweiterung des L4LINUX-Servers

Die im Abschnitt 2.1.2 erläuterte Gliederung des Adressraumes von LINUX unterscheidet sich nur unwesentlich von der Implementierung des L4LINUX-Servers. Wichtigstes Merkmal ist die Verlagerung des Kernel-Bereiches in das erste Gigabyte. Der Bereich von drei bis vier Gigabyte ist für den L4-Kern reserviert. Nutzerprozesse verwenden den verbleibenden Bereich.

Ausgelagerte Module belegen eigentlich keinen Speicher innerhalb des Adressraumes des L^{4}LINUX-Servers. Da der Zugriff auf öffentliche Variablen jedoch mittels des Einblendens von Speicherseiten erfolgt, ist es notwendig einen Bereich zu definieren, in welchen die öffentlichen Daten der Module eingeblendet werden können. Hierzu wurde das MODLD-Area eingeführt, welches sich unterhalb des VMALLOC-Areas befindet. Die Verwaltung des MODLD-Areas obliegt dem Modul-Server. Die Implementierung des L^{4}LINUX-Servers wurde dahingehend erweitert, dass dieser Bereich nicht belegt wird und Zugriffe gesondert behandelt werden. Eine genaue Erläuterung hierzu erfolgt im Abschnitt 3.5.1.

Adressraum des Modul-Servers

Wie bei allen L4-Tasks ist auch im Adressraum des Modul-Servers der L4-Kern in den Bereich oberhalb von drei Gigabyte eingeblendet. Der Programmcode des Servers wird so übersetzt, dass er sich an der gleichen Adresse wie der Programmcode des L^{4}LINUX-Servers befindet. Es muss dabei sichergestellt werden, dass das gesamte Objekt, also Daten- und Textsegment, nicht größer als der Programmcode des LINUX-Servers ist. Ansonsten könnten nicht alle Datensegmente des LINUX-Servers in den Adressraum des Modul-Servers eingeblendet werden, ohne dass es zu Überschneidung kommt. Sicherlich könnte auch ein Bereich oberhalb von einem Gigabyte für den Programmcode des Modul-Servers verwendet werden, jedoch entstehen in diesem Fall Probleme beim Laden des Moduls mittels des L^{4}Loaders.

Page-Frame-Area

Oberhalb des ersten Gigabytes werden im L^{4}LINUX-Server die Nutzerprozesse abgelegt, es ist nicht notwendig, deren Daten in den Adressraum des Modul-Servers oder einer Modul-Task einzublenden. Somit kann dieser Bereich für private Daten verwendet werden.

Der Modul-Server reserviert einen 128 Megabyte großen *Dataspace* oberhalb der Adresse `0x40000000`. Die Reservierung des Dataspace wird vom DMphys durchgeführt und später dynamisch mit physikalischen Speicherseiten belegt. Momentan ist nicht ersichtlich, dass die Module mehr Speicher benötigen. Sollte es doch einmal vorkommen, muss die Größe des Bereiches im Programmcode des Modul-Servers entsprechend verändert werden. Die Größe des Datenbereiches muss vor dem Übersetzen des Modul-Servers definiert werden, da sie für die Initialisierung des Page-Frame-Areas benötigt wird und der verwendete Algorithmus nicht vorsieht, dass die Größe des Speicherbereiches zur Laufzeit verändert werden kann. Sowohl der Modul-Server als auch alle Modul-Tasks verwenden den Dataspace für ihre privaten Daten.

Die bisherige Implementierung des Modul-Servers erlaubt lediglich die Allokation von Speicherbereichen, deren Größe ein Vielfaches von vier Kilobyte beträgt. Hierzu wurde das im Abschnitt 2.1.3 beschriebene Page-Frame-Management in den Modul-Server übernommen. Eine Slab-Allokation ist bisher nicht möglich. Wie sich im Folgenden zeigen wird, ist dies auch nicht unbedingt erforderlich, da ein Großteil der dynamischen Speicherverwaltung immer noch im Adressraum des L^{4}LINUX-Servers erfolgt.

Seiten aus dem privaten Bereich eines Moduls werden vom Modul-Server nicht in andere Adressräume eingeblendet. Sie können ausschließlich von dessen Funktionen verwendet werden. Bei der Implementierung eines ausgelagerten Moduls ist darauf zu achten, dass einem anderen Modul oder dem L^{4}LINUX-Server nicht eine Seite aus diesem Bereich als Funktionsargument oder Bestandteil einer exportierten Struktur übergeben wird. Da dieser Bereich in der anderen L4-Task unter Umständen verwendet wird, ist nicht einmal sichergestellt, dass eine Zugriffsverletzung auftritt. In den meisten Fällen wird das Verhalten der anderen Task unbestimmt sein.

Modul-Task Struktur

Die Verwaltung der L4-Tasks ist Aufgabe des Modul-Servers und ähnlich organisiert wie die Administration der Nutzerprozesse im L^{4}LINUX-Server. Für jede neue Modul-Task wird eine Modul-Struktur angelegt, welche deren Statusinformationen enthält. Hierzu zählen die von ihr belegten Speicherbereiche, die Adressen des Objektcodes, der aktuelle Status, der eigene Task-Identifier, ein Verweis auf die ACL des Moduls sowie die Thread-ID des RPC-Handlers. Die Speicherbereiche werden noch einmal in virtuelle und physikalische Speicherbereiche unterteilt. Darüber hinaus enthält die Task-Struktur des L^{4}LINUX-Servers die Thread-ID des externen Pagers[8].

Der Speicher für den Objektcode des Moduls wird ebenfalls aus dem Page-Frame-Area des Modul-Servers angefordert. Die Adresse wird in der Modul-Struktur im Eintrag addr abgelegt. Sie wird später beim Entfernen des Moduls benötigt. Da der Objektcode des Moduls allerdings auch die globalen Variablen enthält, muss es möglich sein, Teile des Speicherbereiches in einen anderen Adressraum einzublenden. Dies ist nicht möglich, wenn sich der Code im Page-Frame-Area befindet. Aus diesem Grund wird, wie in Abbildung 10 dargestellt, der Speicherbereich beim Modul innerhalb des MODLD-Areas einblendet. Die Adresse wird im Eintrag base vermerkt und später vom Modul-Pager verwendet.

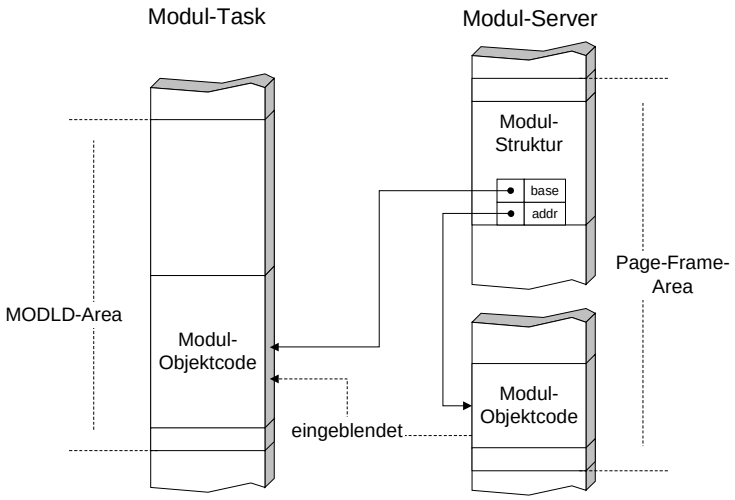

Abbildung 10: Einblenden des Modul-Objektes

Die Modul-Strukturen sind in einer doppelt-verketteten Liste organisiert. Eine Adressierung über eine Hash-Tabelle erscheint nicht als sinnvoll, da die Anzahl der Module auf den meisten Systemen sehr begrenzt ist.

Modul-Task-Stack

Jedes Modul arbeitet mit einem eigenen Stack. Dadurch kann verhindert werden, dass mithilfe dessen Manipulation durch eine andere Task unautorisiert Daten gelesen werden können. Wie bei den Nutzerprozessen des L^{4}LINUX-Servers erfolgt die Speicherung des Stacks zusammen mit der Modul-Struktur. Hierzu wird ein acht Kilobyte großer Speicherbereich allokiert. Im unteren Bereich befindet sich die Modul-Struktur, während der Stack am oberen Ende beginnt. Abbildung 11 zeigt wie die beiden Datenbereiche im Speicher abgelegt werden. Auf der Intel-Architektur wächst der Stack immer

[8] Der externe Pager ist für das Einblenden von Speicherseiten aus dem Adressraum des L^{4}LINUX-Servers verantwortlich. Der Pager behandelt lediglich Anforderungen des Modul-Servers, da alle Seiten zunächst in dessen Adressraum eingeblendet werden.

von oben nach unten. Der Inhalt des Stack-Registers esp verweist immer auf das untere Ende des Stacks.

Die Zusammenlegung des Stacks und der Modul-Struktur hat zusätzlich den Vorteil, dass innerhalb des Moduls leicht auf die eigene Modul-Struktur zugegriffen werden kann. Mittels des aktuellen Inhaltes des esp lässt sich die Adresse der Struktur leicht berechnen. Hierzu kann zusätzlich vorausgesetzt werden, dass sich die Struktur immer an einer durch 8192 Byte teilbaren Adresse befindet.

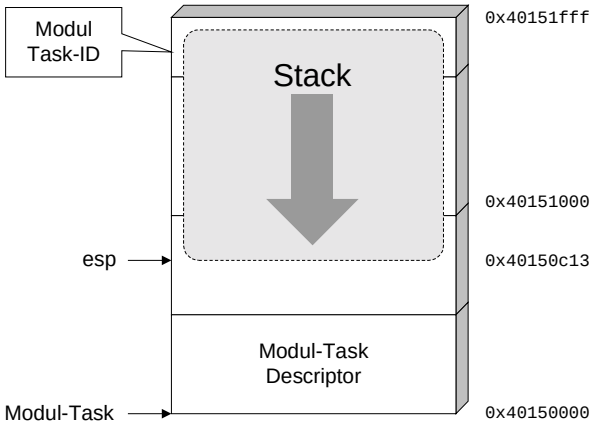

Abbildung 11: Speicherung des Modul-Task Deskriptors und des Modul-Stacks

Modul-Pager

Für die Verwaltung von Adressräumen ist ein Pager notwendig. Die initiale Task des Modul-Servers wird durch einen externen Pager verwaltet, jedoch wird für alle weiteren Threads und die Modul-Tasks ein eigener Pager angelegt. Dieser Pager wird bei der Initialisierung des Modul-Servers erzeugt und läuft in dessen Adressraum. Somit hat er Zugriff auf alle Daten des Modul-Servers. Verursacht ein Thread oder eine Modul-Task ein Zugriffsverletzung innerhalb des Objektcodes oder innerhalb des Page-Frame-Areas werden die Seiten in dessen Adressraum eingeblendet. Zugriffe auf einen anderen Bereich sollten nicht vorkommen und werden nicht gewährt.

Der Pager ist außerdem für die Organisation der Adressräume der Modul-Tasks verantwortlich. Bei einer Zugriffsverletzung erhält der Pager den verursachenden Thread und die ungültige Adresse. Mithilfe dieser Informationen kann festgestellt werden, ob es sich um einen gültigen Zugriff handelt und die Seite eingeblendet werden kann. Ein Modul hat zunächst einmal nur Zugriff auf seinen eigenen Objektcode und auf die von ihm allokierten Speicherbereiche. Darüber hinaus sind zusätzlich Zugriffe auf gemeinsame Speicherbereiche möglich. Hierzu zählen die in Abbildung 9 hellgrau markierten Bereiche des L^{4}LINUX-Servers und der Modul-Tasks. Zugriffe auf die Speicherbereiche des L^{4}LINUX-Servers werden an dessen externen Pager weitergeleitet. Eine genaue Beschreibung hierzu erfolgt im Abschnitt 3.5.1.

Adressraum einer Modul-Task

Der Adressraum einer Modul-Task wird zum Großteil durch die gemeinsamen Speicherbereiche bestimmt. Das oberste Gigabyte ist wiederum durch den L4-Kern belegt. Der Objektcode des Moduls wird, wie zuvor beschrieben, innerhalb des MODLD-Areas eingeblendet. Jedes Modul erhält einen eigenen Bereich, so dass das Datensegment auch in die Adressräume der anderen Module eingeblendet werden kann. Dies ist notwendig, damit das Konzept der Stacked-Moduls erhalten bleibt.

Private Datenbereiche, die aus dem Page-Frame-Area des Modul-Servers allokiert werden, befinden sich immer oberhalb des ersten Gigabyte und damit außerhalb des öffentlichen Bereiches. Sie werden

eins zu eins in den Adressraum des Moduls einblendet. Da alle Module ihre privaten Bereiche aus dem gleichen Page-Frame-Area erhalten, kann es nicht vorkommen, dass sich zwei Bereiche überschneiden.

Der Objektcode eines Moduls unterteilt sich in das Text- und das Datensegment. Da es nicht ohne weiteres möglich ist, beide Bereiche zu trennen, werden sie zusammen im MODLD-Area abgelegt. Es ist die Aufgabe des Pagers sicherzustellen, dass keine Seiten des Textsegmentes in den Adressraum eines anderen Moduls oder des L^{4}LINUX-Servers eingeblendet werden. Hierzu muss der Pager unterscheiden können, wo das Datensegment beginnt beziehungsweise das Textsegment endet. Eine Objektdatei wird in Sektionen unterteilt. Jede Sektion enthält einen bestimmten Typ von Daten oder Funktionen. Zum Beispiel enthält die `.text` Sektion immer den ausführbaren Programmcode und die `.data` Sektion die initialisierten Daten. Ein Modul ist immer so strukturiert, dass als erstes die `.text` Sektion in der Datei abgelegt wird, gefolgt von mehreren zusätzlichen Sektionen mit Programmcode, auf die hier nicht näher eingegangen wird. Die Datensektionen folgen erst später im Modul, so dass sie klar von der eigentlichen Textsektion abgegrenzt werden können.

Um diese Informationen zur Laufzeit innerhalb des Pagers verwenden zu können, werden beim Linken des Moduls zusätzliche globale Variablen definiert, die die Grenzen der wichtigsten Sektionen markieren. Anhand dieser Informationen kann der Pager entscheiden, ob eine Seite in den Adressraum eines anderen Moduls eingeblendet werden darf oder nicht. Allerdings beginnt das Datensegment nicht zwangsläufig mit einer neuen Seite, so dass beim Einblenden der ersten Seite unter Umständen auch ein Teil der letzten Sektion des Textsegmentes in den fremden Adressraum eingeblendet wird. Hierbei handelt es sich jedoch um die `.text.lock` Sektion[9] des Moduls, die nur einen nicht Modul-spezifischen und selten verwendeten Teil des Programmcodes enthält.

3.3.2 Hierarchie der Speicherallokation

Die Nutzung des Page-Frame-Areas durch die Threads des Modul-Servers und durch die Modul-Tasks bedingt ein mehrschichtiges Interface. Die unterste Schicht bildet der initiale Thread des Modul-Servers. Er reserviert bei der Initialisierung des Servers den Dataspace für das Page-Frame-Area.

In der darüber liegenden Schicht werden alle Threads des Modul-Servers eingeordnet. Allokieren sie einen neuen Speicherbereich aus dem Page-Frame-Area, so müssen sie dies mithilfe des initialen Threads ausführen. Dies ist notwendig, da Speicher aus einem reservierten Bereich nur von dem Thread angefordert werden kann, der die Reservierung durchgeführt hat. Die Anforderung einer Seite wird aus diesem Grund mittels IPC an den initialen Thread weitergeleitet. Zur Bedienung solcher Anforderungen wartet der Thread, nachdem er die Initialisierung des Modul-Servers abgeschlossen hat, permanent auf ein Signal. Er akzeptiert lediglich Signale von einem Thread des Modul-Servers und vom L^{4}LINUX-Server[10]. Threads des Modul-Servers können eine Nachricht zum Anfordern und zum Freigeben von Speicherbereichen senden. Da es sich um Threads handelt, ist es ausreichend, die Seite vom DMphys anzufordern und die Adresse an den anfordernden Thread zurückzuschicken. Anschließend kann dieser über den neuen Speicherbereich verfügen. Beim Freigeben einer Seite wird eine leere Nachricht zurückgeschickt.

Fordert eine Modul-Task eine neue Seite an, so muss sie dies mittels eines RPCs ausführen. Der Modul-Server stellt ein Interface zum Anfordern und zum Freigeben von Speicherseiten bereit. Es werden allerdings nur RPCs von L4-Tasks behandelt, die vom Modul-Server generiert wurden, da

---

[9] Da beim Laden der Instruktionspipe meist ein optimistischer Ansatz gewählt wird, wird die Behandlung der negativen Fälle aus dem normalen Instruktionsfluss entfernt. Sie landen in der Sektion `.text.lock`. Das gilt für Semaphore und den *„spin locks“* bei Multiprozessor-Systemen [7].

[10] Vom L^{4}LINUX-Server erhält der initiale Thread des Modul-Servers die Nachricht, dass das System heruntergefahren wird. In diesem Fall veranlasst er die Deaktivierung des Servers und aller Modul-Tasks.

sonst nicht sichergestellt werden kann, dass die Seiten freigegeben beziehungsweise Zugriffe korrekt behandelt werden können. Somit ist es für den L[4]LINUX-Server nicht möglich das Interface zu nutzen. Die Behandlung der RPCs erfolgt durch einen separaten Thread des Modul-Servers, so dass zur eigentlichen Anforderung der Seite das IPC-Interface zum initialen Thread des Modul-Servers verwendet werden muss. Konnte eine neue Seite allokiert werden, wird sie in der Modul-Task-Struktur eingetragen und die Adresse an die anfordernde Task zurückgegeben. Beim ersten Zugriff auf die Seite wird sie in den Adressraum der Task eingeblendet.

3.3.3 Sichere Speicherbereiche für Module

Ziel der hier vorgestellten Architektur war die Bereitstellung von sicheren Speicherbereichen für Funktionen des Systemkerns. Die Auslagerung von Modulen in separate L4-Tasks stellt sicher, dass sie in getrennten Adressräumen ablaufen und somit durch die Systemarchitektur voreinander geschützt sind. Da der L[4]LINUX-Server im Nutzermodus läuft, besteht für ihn auch nicht die Möglichkeit die Grenzen seines Adressraumes zu verlassen[11].

Allerdings ist es aufgrund der engen Verflechtung von Modul und Systemkern notwendig, einen Großteil der Daten in einem gemeinsamen Bereich abzulegen. Damit erhält der Kern oder eine Funktion eines anderen Moduls uneingeschränkten Zugriff auf diese Daten. Der Schutz beschränkt sich auf die privaten Bereiche des Moduls. In der ursprünglichen Implementierung werden diese Bereiche nicht verwendet, so dass der Anwender gezwungen ist, den Programmcode des Moduls entsprechend anzupassen. Die Architektur schafft somit nur die Voraussetzungen zur Bereitstellung von sicheren Speicherbereichen. Zur vollständigen Nutzung des Konzeptes sind Änderungen am Programmcode des Moduls unumgänglich.

3.4 Architektur zur IPC mittels *Remote Procedure Calls* (RPC)

Die Ausführung von RPCs basiert auf der IPC des L4-Kerns und ist somit ebenfalls synchron. Dies hat zur Folge, dass der aufrufende Prozess solange blockiert wird, bis der RPC vollständig abgearbeitet wurde. Bei verschachtelten RPCs kann dies unter Umständen zu einem *Deadlock*[12] führen, wenn, wie in Abbildung 12 dargestellt, eine Modul-Funktion einen RPC an den L[4]LINUX-Server sendet, obwohl dieser bereits verwendet wird.

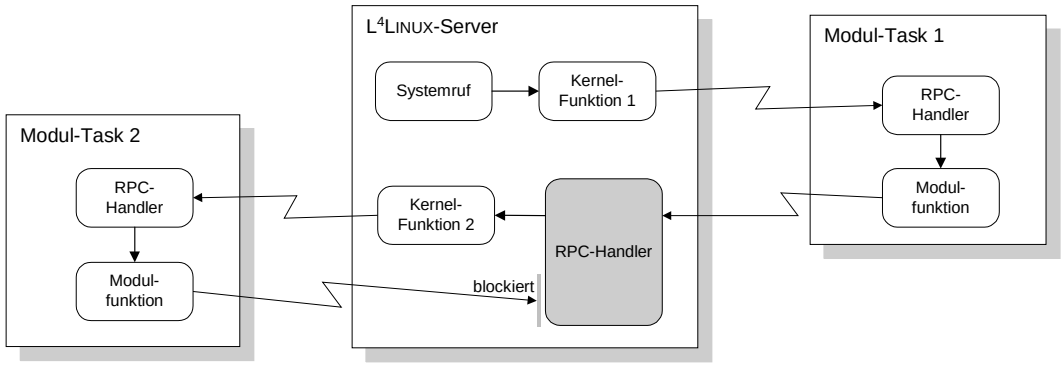

Abbildung 12: RPC-Deadlock

[11] Durch die Nutzung von DMA Transfers besteht für eine Task im Nutzermodus immer noch die Möglichkeit die Grenzen des Adressraumes zu verlassen. Dies ist ein bekanntes Problem, dessen Lösung Bestandteil aktueller Forschung ist und an dieser Stelle vernachlässigt wird [22].

[12] Ein Deadlock ist eine permanente, gegenseitige Blockierung von einer Menge von Prozessen. Diese Prozesse kämpfen um eine Systemressource oder warten gegenseitig auf ein bestimmtes Signal.

Solange der Programmcode innerhalb des RPCs bekannt ist oder hinsichtlich bestimmter Richtlinien, die genau dies verhindern, implementiert wurde, könnte ein möglicher Deadlock verhindert werden. Jedoch verwendet das hier vorgestellte Konzept Funktionen, die in den meisten Fällen für den monolithischen Kern von LINUX implementiert und nicht speziell angepasst wurden. Demzufolge ist es durchaus möglich, dass bei der Ausführung einer exportierten Modul-Funktion wiederum eine Funktion des L^{4}LINUX-Servers aufgerufen werden soll, obwohl dieser durch einen RPC-Aufruf bereits blockiert ist. Um dies zu verhindern, wurde eine Infrastruktur zur Behandlung von geschachtelten RPC-Aufrufen entwickelt, die den aufrufenden Prozess zwar blockiert, jedoch die Bearbeitung von weiteren RPCs gestattet. Es ist nicht ausreichend, den RPC asynchron zu gestalten, da sonst die Semantik des Funktionsaufrufes zerstört wird und der vorhandene Programmcode nicht ohne weiteres wiederverwendet werden könnte.

3.4.1 RPC-Threadstruktur

Die Architektur zur Behandlung von RPC-Anforderungen besteht aus den in Abbildung 13 dargestellten Threads. Sie wird in den Modul-Server, den L^{4}LINUX-Server sowie in alle Modul-Tasks integriert, so dass ein einheitliches Interface für die Kommunikation zwischen den einzelnen Tasks entsteht.

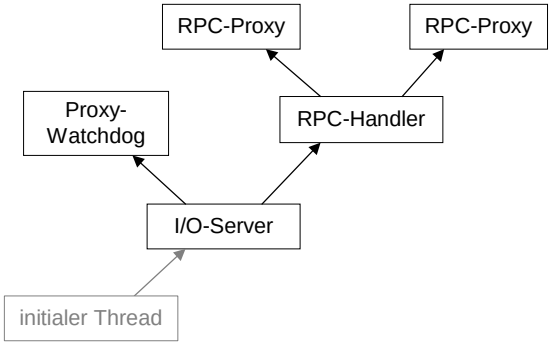

Abbildung 13: RPC-Threads

Die Initialisierung der Architektur erfolgt immer vom initialen Thread der L4-Task aus. Er selbst ist nicht für die Behandlung von RPCs verantwortlich, so dass vor der Initialisierung der Architektur keine RPCs angenommen werden können. Beim L^{4}LINUX-Server werden die Threads zusammen mit dem übrigen System initialisiert.

I/O-Server

Der I/O-Server ist ausschließlich für die Verwaltung des RPC-Handlers und des Proxy-Watchdogs verantwortlich. Er übernimmt die Initialisierung der Threads und wartet anschließend auf ein Signal von diesen oder auf eines vom initialen Thread. Die Stacks für die Threads allokiert der I/O-Server mittels des Page-Frame-Interfaces des Modul-Servers beziehungsweise des L^{4}LINUX-Servers, so dass diese nach dem Deaktivieren der Threads von ihm auch wieder freigegeben werden können.

Erhält er ein Signal vom RPC-Handler oder vom Proxy-Watchdog, so bedeutet dies, dass ein fataler Fehler aufgetreten ist und die Behandlung von weiteren RPCs nicht mehr möglich ist. In diesem Fall wird der andere Thread ebenfalls beendet und ein Signal an den initialen Thread gesendet. Dieser wird daraufhin die Beendigung der gesamten Task einleiten.

Ein Signal vom initialen Thread an den I/O-Server wird immer dann gesendet, wenn die Task beendet werden soll. Der Thread sorgt daraufhin für die Deaktivierung des RPC-Handlers und des Proxy-Watchdogs und gibt die allokierten Speicherbereiche frei.

RPC-Handler

Der RPC-Handler ist für die Delegation der RPC-Anforderungen an einen RPC-Proxy verantwortlich. Die eigentliche Behandlung eines RPCs gliedert sich in mehrere Schritte, die sich allgemein auf den Programmcode aus Listing 1 reduzieren lassen.

Mittels der Funktion `flick_server_wait(...)` wartet der Thread auf das Eintreffen einer RPC-Anforderung. Wurde eine entsprechende Anforderung erfolgreich angenommen, wird sie mittels der Funktion `if_modld_modules_server(...)` behandelt und das Ergebnis sowie die Out-Parameter mittels `flick_server_send(...)` an den aufrufenden Thread zurückgegeben. Ab dem Eintreffen der Anforderung bis zum Zurücksenden der Ergebnisse ist sowohl der aufrufende als auch der RPC-Thread blockiert. Demzufolge können in dieser Zeit keine weiteren Anforderungen entgegengenommen werden. Bei der Verwendung nur eines Threads würde dies beim Eintreffen einer weiteren Anforderung zu dem in Abbildung 12 dargestellten Deadlock führen.

```
result = flick_server_wait(&request);
if (!L4_IPC_IS_ERROR(result)) {
      int ret;

      /* dispatch request */
      ret = if_modld_modules_server(&request);
      switch (ret) {
      case DISPATCH_ACK_SEND:
            result = flick_server_send(&request);
            if (L4_IPC_IS_ERROR(result))
                  printf("IPC error (0x%08x)!\n", result.msgdope);
            break;
      default:
            printf("dispatch error %d!\n", ret);
      }
}
```

Listing 1: RPC-Behandlung

Um dies zu verhindern, delegiert der RPC-Handler die Behandlung des RPCs an einen neuen Proxy-Thread. Leider erlaubt es die Semantik des RPC-Interfaces nicht, den Programmcode von Listing 1 auf zwei verschiedene Threads zu verteilen, so dass die Annahme der Anforderung im RPC-Handler und die eigentliche Behandlung des RPCs in einem anderen Thread erfolgen könnte. Die Rückgabewerte müssen von dem Thread abgeschickt werden, welcher auch die RPC-Anforderung entgegengenommen hat. Aus diesem Grund wird die Ausführung eines RPCs noch einmal in zwei Schritte unterteilt. Der erste Schritt, die Generierung eines neuen RPC-Proxies wird vom RPC-Handler ausgeführt.

RPC-Proxy

Der RPC-Proxy besteht im Wesentlichen aus dem oben beschriebenen Programmcode. Seine einzige Aufgabe besteht aus der Annahme einer RPC-Anforderung und deren Ausführung. Zur Durchsetzung von Sicherheitsrichtlinien muss dem RPC-Proxy bekannt sein, von welchem Thread er eine Anforderung entgegennehmen darf. Um ein unnötiges Erzeugen von Threads zu verhindern, erfolgt die Authentifizierung des RPC-Clients bereits im RPC-Handler.

Damit zudem das *Hijacking*[13] einer RPC-Anforderung verhindert werden kann, überprüft der RPC-Proxy noch einmal, ob die Anforderung von dem authentifizierten Thread initiiert wurde. Ist dies nicht

[13] Als Hijacking bezeichnet man die Übernahme von bestehenden Kommunikationsverbindungen, um eine zuvor durchgeführte Authentifizierung eines anderen Kommunikationspartners auszunutzen. Besonders verbreitet ist diese Art von Angriffen beim *Transmission Control Protocol* (TCP).

der Fall, wird die Behandlung des RPCs nicht ausgeführt und ein Fehler zurückgegeben. Anschließend wartet der RPC-Proxy auf eine neue Anforderung, damit der authentifizierte RPC-Client nicht blockiert wird. Die ID des authentifizierten Threads wird vom RPC-Handler zusammen mit der Funktionsnummer in eine globale Liste eingetragen und nach der Abarbeitung vom RPC-Proxy gelöscht.

Proxy-Watchdog

Der Proxy-Watchdog ist wie der RPC-Handler für die Verwaltung der RPC-Proxies verantwortlich. Da der RPC-Handler permanent auf eine neue RPC-Anforderung wartet, muss das Aufräumen der Proxy-Reste von einem anderen Thread ausgeführt werden. Hierzu wird dem RPC-Proxy die Thread-ID des Proxy-Watchdogs vom RPC-Handler als Argument übergeben. Nachdem der Proxy die RPC-Anforderung vollständig behandelt hat, sendet er ein Signal an den Watchdog, dass er nicht mehr benötigt wird, und legt sich schlafen.

Der Watchdog wartet ständig auf ein Signal von einem der Proxies. Trifft ein solches Signal ein, gibt er den Speicher des Proxy-Stacks frei. Die Adressen der Stacks werden zusammen mit den Thread-IDs in einer Liste verwaltet, auf die der Watchdog als auch der RPC-Handler zugreifen kann. Es werden nur IPCs von Threads entgegengenommen, die in der Proxy-Liste vermerkt sind. So kann sichergestellt werden, dass nicht versehentlich ein falscher Stack freigegeben wird.

Erhält der Watchdog ein Signal vom I/O-Server, so beendet er alle momentan aktiven Proxies und gibt deren Stack-Speicher frei. Im Normalfall sollten zu diesem Zeitpunkt keine Proxies aktiv sein, da ein solches Signal nur eintrifft, wenn die Modul-Task entfernt werden soll und der RPC-Handler bereits beendet worden ist.

3.4.2 RPC-Aufruf

Nachdem im vorangegangenen Abschnitt die einzelnen Threads zur Behandlung von RPC-Anforderungen erläutert wurden, erfolgt hier noch einmal eine Beschreibung der RPC-Behandlung selbst. Da sowohl der L^{4}LINUX-Server, der Modul-Server als auch alle Modul-Tasks die gleiche RPC-Architektur verwenden, wird die Erläuterung auf einen RPC-Client und einen RPC-Server verallgemeinert. Abbildung 14 zeigt die einzelnen Komponenten und die Interaktion zwischen ihnen. Der RPC-Client wendet sich zur Ausführung einer entfernten Funktion an den RPC-Server, welcher die zuvor beschriebene Architektur enthält.

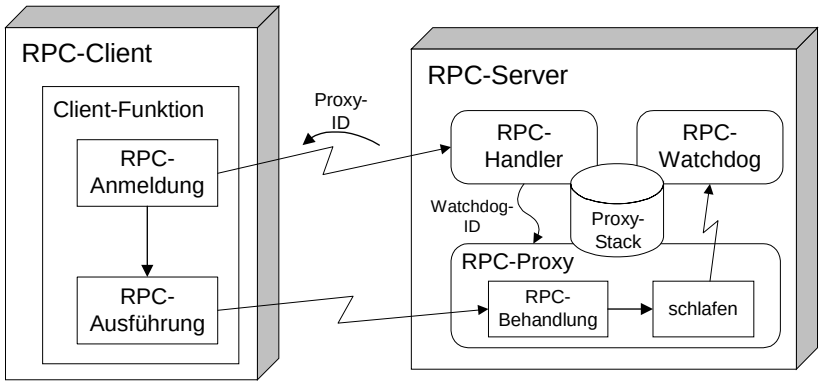

Abbildung 14: RPC-Kommunikation

Der RPC-Client kontaktiert im Rahmen der RPC-Anmeldung zunächst den RPC-Handler, um die ID des Proxy-Threads zu erfragen. Hierbei erfolgt die Authentifizierung des Clients. Der RPC-Server erhält aus den Parametern des Aufrufes die Thread-ID des Clients und die aufzurufende Funktion. Die übermittelte Task-ID wird vom L4-Kern generiert und kann vom RPC-Client nicht manipuliert

werden. Die Anmeldung einer anderen Funktion würde später vom RPC-Proxy erkannt und blockiert werden. Damit kann sichergestellt werden, dass der RPC-Client seine wahre Identität und sein wirkliches Vorhaben nicht maskieren kann.

Nach einer positiven Authentifizierung erhält der RPC-Client die Thread-ID des neu erzeugten RPC-Proxies und ist damit in der Lage die RPC-Anforderung an diesen zu senden. Der RPC-Proxy führt daraufhin nochmals die zuvor erwähnte Überprüfung des Threads und der Funktion durch. Ist beides mit der Anmeldung identisch, wird die Funktion ausgeführt und das Resultat an den RPC-Client zurückgegeben. Abschließend sendet der Proxy ein Signal an den Watchdog, dass er mit der RPC-Behandlung fertig ist. Daraufhin wird dieser den Stack des Proxies freigeben.

3.5 Verwaltung öffentlicher Symbole

Die Interaktion zwischen dem Kern und einem Modul beziehungsweise umgekehrt lässt sich auf Funktionsaufrufe und die Nutzung von Variablen beschränken. Da Zugriffe auf Variablen öfter notwendig sind als Aufrufe von Funktionen, wird für jede Zugriffsart eine separate IPC implementiert. Darüber hinaus ist es sinnvoll, bei der Initialisierung der Symbole die Zugriffe hinsichtlich ihrer Richtung zu unterscheiden. Während die unbekannten Symbole eines Moduls beim Laden aufgelöst werden können, müssen die anzumeldenden Funktionen in den Systemkern zur Laufzeit integriert werden. Beide Verfahren wurden bereits im Abschnitt 2.2.3 erläutert. Dieser Abschnitt beschreibt die Vorgehensweise zum Einbinden von Symbolen aus exportierten Modulen.

3.5.1 Zugriff auf Variablen

Der Zugriff auf exportierte Variablen erfolgt über den im Abschnitt 3.3.1 erläuterten gemeinsamen Speicherbereich. Sowohl für den LINUX-Kern als auch für eine Modul-Task soll es nicht möglich sein, eine Variable aus einem anderen Bereich des fremden Adressraumes zu lesen. Dieser Schutz wird durch den Prozessor beziehungsweise den L4-Kern umgesetzt und kann somit an dieser Stelle als gegeben vorausgesetzt werden.

Zugriffe einer Modul-Task

Die Zugriffskontrolle einer Modul-Task wird mittels des Pagers des Modul-Servers realisiert. Greift eine Modul-Task auf eine gemeinsame Seite zu, so verursacht dies eine Zugriffsverletzung, die an den Pager des Modul-Servers weitergeleitet wird. Dieser ist dann für das Einblenden der Seite verantwortlich. Dabei kann es sich um eine eigene als auch um eine fremde Seite handeln. Zugriffe auf Seiten des L^{4}LINUX-Servers können vom Pager nicht direkt aufgelöst werden, solange die Seite nicht in seinen Adressraum eingeblendet ist.

Greift eine Modul-Task auf eine Seite des MODLD-Areas zu, die bisher nicht in seinen Adressraum eingeblendet wurde, so sendet der L4-Kern einen Page-Fault an den Pager des Modul-Servers. Ist die verursachende Task Besitzer der Seite, so wird diese in dessen Adressraum eingeblendet. Anderenfalls wird zunächst überprüft, ob eine Seite des data-Segmentes angefordert wurde und ob der Zugriff autorisiert ist. Ist dies nicht der Fall wird der Zugriff verweigert. Da die Seiten des MODLD-Areas aus dem Page-Frame-Area des Modul-Servers entnommen werden, kann davon ausgegangen werden, dass die Seite bereits in dessen Adressraum einblendet ist. Die Seiten des MODLD-Areas werden an der gleichen Adresse wie bei der Besitzer-Task eingeblendet. Da jede Task einen eigenen Bereich belegt, kann es zu keiner Überschneidung von Bereichen kommen. Es ist nicht möglich Speicherbereiche oberhalb des ersten Gigabytes eins zu eins in den Adressraum einer anderen Task einzublenden. Aus diesem Grund muss bei der Implementierung des Moduls darauf geachtet werden, dass sich öffentliche Variablen immer im Bereich unterhalb des ersten Gigabytes befinden.

Zum Einblenden von Seiten des L^{4}LINUX-Servers wird ein externer Pager angesprochen. Er wartet auf die Anforderung einer Seite per IPC. Dabei werden nur Nachrichten vom Modul-Pager akzeptiert.

Allen anderen Tasks wird der Zugriff verweigert. Da die Implementierung nicht vorsieht, dass eine solche Anforderung notwendig wäre, kann es sich nur um eine Sicherheitsverletzung oder einen Programmfehler handeln. Da der Modul-Server selbst keine Daten in den geteilten Speicherbereich ablegt, kann die von ihm angeforderte Seite in dessen Adressraum an der ursprünglichen Adresse eingeblendet werden. Abschließend wird die Seite in den Adressraum der Modul-Task eingeblendet. Dies erfolgt wiederum an der Originaladresse der Seite, da eine Task nur Speicherbereiche innerhalb eines privaten Bereichs des MODLD-Areas und oberhalb des ersten Gigabytes belegen kann.

Der externe Pager des L^{4}LINUX-Servers ist nicht in der Lage zu erkennen, welche Modul-Task das Einblenden einer Seite veranlasst hat. Zugriffe von Modul-Tasks auf gemeinsame Speicherbereiche werden immer durch den Pager des Modul-Servers aufgelöst. Aus diesem Grund erfolgt die Authentifizierung des Zugriffs bereits beim Modul-Server.

Zugriffe des L^{4}LINUX-Servers

Der L^{4}LINUX-Server besitzt seinen eigenen Pager, der zur Unterstützung des Konzeptes erweitert wurde, so dass Zugriffe auf fremde Variablen erkannt und ähnlich behandelt werden können. Für den L^{4}LINUX-Server befinden sich fremde Variablen immer innerhalb des MODLD-Areas. Greift eine Funktion des Kerns auf eine Variable innerhalb dieses Bereiches zu, so führt dies zur Generierung eines Page-Faults, der an den Pager des L^{4}LINUX-Servers weitergeleitet wird. Im Normalfall löst die Funktion den Page-Fault selbst auf und unterscheidet die Behandlung lediglich hinsichtlich Zugriffen auf den Kern-Bereich oder auf das VMALLOC-Area. Die Page-Fault-Behandlung des L^{4}LINUX-Servers wurde zur Unterstützung des MODLD-Areas dahingehend erweitert, dass sie zusätzlich bei Zugriffen auf diesen Bereich eine separate Behandlung initiieren.

Alle Seiten des MODLD-Areas werden vom Modul-Server verwaltet. Zum Einblenden einer Seite aus diesem Bereich sendet der L^{4}LINUX-Server einen *Page-Request* an den Modul-Server. Der Page-Request wird vom MODLD-Area-Pager behandelt. Hierbei handelt es sich um einen separaten Thread des Modul-Servers, der für das Einblenden von Speicherseiten aus dem MODLD-Area verantwortlich ist. Die Verwendung des Pagers ist immer dann notwendig, wenn MODLD-Task mit einem eigenen Pager eine Seite des MODLD-Areas verwenden möchte. In der aktuellen Implementierung gilt dies nur für den L^{4}LINUX-Server. Vor dem Einblenden der Seite führt der Pager eine Authentifizierung des Zugriffs durch.

Integration von exportierten Symbolen

Obiges Konzept für den Zugriff auf globale Variablen innerhalb eines fremden Adressraumes ist immer möglich, wenn sich die Variable im gemeinsamen Bereich befindet. Dabei ist es nicht relevant, ob das Symbol zuvor innerhalb des Programmcodes mittels des `EXPORT_SYMBOL`-Makros exportiert wurde. Dies ist notwendig, da ein Modul zur Laufzeit nicht nur auf die exportierten Variablen zugreift, sondern zusätzlich Zeiger verfolgt oder neue Speicherbereiche anlegen kann. Eine Einschränkung des Zugriffs auf exportierte Symbole ist nicht sinnvoll, da sonst ein Großteil des Programmcodes eines ausgelagerten Moduls umgeschrieben werden müsste. Doch eben dies soll mit dem hier vorgestellten Konzept umgangen werden.

Wie verhindert werden kann, dass der Kern auf private Daten eines ausgelagerten Moduls zugreift, wurde bereits im Abschnitt 3.3.3 beschrieben. Somit ist es möglich sowohl private Speicherbereiche als auch gemeinsame Variablen wie in der ursprünglichen Implementierung bereitzustellen, so dass zur Auslagerung von Modulen nicht der gesamte Programmcode umgeschrieben werden muss. Dies kann später erfolgen, wenn ersichtlich wird, dass bestimmte Variablen geschützt werden müssen.

Das Exportieren von Variablen muss jedoch auch für ausgelagerte Module möglich sein, da sonst das Konzept der Stacked-Moduls nicht verwendet werden kann. Ein Modul benötigt die Adressen aller unbekannten Symbole bevor es geladen werden kann. Das Auflösen von Funktionssymbolen wird im anschließenden Abschnitt erläutert und ist wesentlich umfangreicher. Die Adressen von Variablen

können wie bisher mithilfe des Modul-Loaders aufgelöst werden. Hierzu wird eine Funktion in den L⁴LINUX-Servers integriert, welche die Adressen der exportierten Symbole in die Symboltabelle des Kerns einträgt und die Abhängigkeiten aktualisiert. Die exportierten Symbole werden immer innerhalb des MODLD-Areas abgelegt, so dass eine Zugriffsverletzung nicht auftreten kann.

3.5.2 Öffentliche Funktionen

Während der Zugriff auf exportierte Variablen mittels des gemeinsamen Speicherbereiches realisiert werden kann, muss für den Aufruf von öffentlichen Funktionen ein wesentlich umfangreicheres Konzept implementiert werden. Dieser Abschnitt beschreibt das Auflösen von fremden Symbolen und das Anmelden von Funktionen, so dass sie über Adressraumgrenzen hinaus verwendet werden können. Der Aufruf der öffentlichen Funktionen erfolgt mittels der im Abschnitt 3.4 beschriebenen RPC-Architektur, hier folgt zusätzlich eine Erläuterung der Parameterübergabe und der Adressierung der Zielfunktion.

Adressierung von fremden Symbolen

In der bisherigen Implementierung werden unbekannte Symbole eines Moduls vom Modul-Loader aufgelöst und die Adressen in den Programmcode eingetragen. Beim Anmelden von Modul-Funktionen wird eine Struktur mit den Adressen der öffentlichen Funktionen einer Registrierfunktion des Systemkerns übergeben. Für ausgelagerte Module können diese Konzepte nicht mehr verwendet werden, da ein direkter Zugriff auf die Funktionen nicht möglich ist. Die Adressierung einer Funktion erfolgt nun über einen *Module Identifier* (MID), bestehend aus der Task-ID der Modul-Task und der Thread-ID des RPC-Handlers, sowie einem *Function Identifier* (FID). Die FID ist innerhalb eines Moduls eindeutig, so dass zusammen mit der MID eine eindeutige Adressierung eines Symbols innerhalb des Systems möglich ist. Beide Werte können zur Übersetzungszeit nicht fest vorgegeben werden und müssen zur Laufzeit ermittelt werden.

Jede Modul-Task muss seine öffentlichen Symbole beim Modul-Server registrieren beziehungsweise anmelden. Dabei wird dem Modul-Server der Name und die FID der Funktion übergeben. Aus den Parametern des RPCs kann zudem die Task-ID ermittelt werden. Die Thread-ID des RPC-Handlers wird beim Initialisieren der RPC-Infrastruktur an den Modul-Server übergeben und von diesem in der Modul-Task-Struktur vermerkt. Die gesammelten Informationen werden vom Modul-Server in einer Symboltabelle gespeichert und auf Anfrage an die entsprechenden Tasks weitergegeben.

Symboltabelle des Modul-Servers

Für jede öffentliche Funktion wird ein Eintrag vom Typ `modld_symbol_table` in der Symboltabelle des Modul-Servers angelegt. Der Typ `modld_symbol_table` hat das folgende Format:

```
typedef struct __modld_symbol_table {
      struct __modld_symbol_table *next, *prev;

      char name[MAX_SYMBOL_NAME_LEN];
      l4_uint16_t id;
      l4_uint8_t nargs;
      l4_uint8_t retval;

      modld_struct_t *server;
} modld_symbol_table;
```

Listing 2: Elemente der Symboltabelle

wobei die einzelnen Elemente wie folgt belegt sind:

name:

Das Element name speichert den Namen des Symbols, so wie es im Quellcode und vom Modul-Loader verwendet wird. Die Quellen des L4LINUX-Servers sehen für den Namen eines Symbols ein Array mit 60 Zeichen vor.

id:

Die FID wird im Element id abgelegt. Momentan existieren ungefähr 4000 EXPORT_SYMBOL-Anweisungen im Programmcode des LINUX-Kerns, so dass eine 16-bit Zahl ausreichend erscheint.

nargs:

Beim Aufruf der RPC-Funktion wird nur die FID übermittelt. Die Argumente werden in einem gemeinsamen Speicherbereich abgelegt. Zur Realisierung dieses Konzeptes ist es notwendig, dass die RPC-Funktion die Anzahl der Funktionsargumente kennt.

retval:

Der Aufruf der öffentlichen Funktionen erfolgt typenfrei, ebenso die Rückgabe des Ergebniswertes. Um bei void-Funktionen keine fehlerhaften Rückgabewerte zu erzeugen, werden diese Funktionen gesondert behandelt. Das Element retval gibt an, ob eine Funktion einen Rückgabewert besitzt oder nicht.

server:

Die zugehörige Task wird im Eintrag server abgelegt. Dieser verweist direkt auf die Modul-Task-Struktur, welche alle notwendigen Informationen zum Aufruf des RPCs enthält.

Die Elemente next und prev dienen zur Realisierung einer doppelt verketteten Liste. Damit ist zwar keine effiziente Suche auf einer größeren Liste möglich, jedoch kann diese ohne weiteres in einer späteren Implementierung hinzugefügt werden.

Registrieren von Funktionen

Jede neue Modul-Task kennt zunächst nur den Modul-Server, dessen Task-ID der Funktion init_module(...) übergeben wird. Alle weiteren Informationen erhält die Task vom Modul-Server. Hierzu gehören auch, wie soeben beschrieben, die unbekannten Symbole. Um dies gewährleisten zu können, muss jede öffentliche Funktion beim Modul-Server registriert werden.

Das Registrieren einer Funktion wird durch die Modul-Task oder den L4LINUX-Server initiiert. Hierfür ist die Funktion modld_init(...) verantwortlich. Sie wird entweder von der Funktion init_module(...) oder bei der Initialisierung des L4LINUX-Servers aufgerufen. Die Implementierung ist allgemein gestaltet, so dass sie für alle Modul-Tasks und den L4LINUX-Server verwendet werden kann. Sie ermittelt zunächst mithilfe der Funktion modld_get_internal_syms(...) die Liste der öffentlichen Symbole. Bei dieser Funktion handelt es sich um eine Task-spezifische Implementierung, welche als Ergebnis die Anzahl der Symbole zurückgibt und den ihr übergebenen Zeiger auf ein Array mit den Symbolinformationen legt. Die Implementierung der Funktion modld_get_internal_syms(...)wird automatisch generiert, eine Erläuterung hierzu erfolgt im Abschnitt 4.1. Das eigentliche Registrieren eines Symbols erfolgt mittels der Funktion __modld_register_sym(...), welche als Argumente den Namen, die FID, die Anzahl der Argumente und den Rückgabetyp des Symbols erwartet. Damit können alle Elemente der Struktur modld_symbol_table bis auf server gesetzt werden. Die Funktion __modld_register_sym(...) ist als RPC implementiert und wird innerhalb des Modul-Servers ausgeführt. Dieser kann anhand der RPC-Parameter die ID der Task bestimmen und den server-Eintrag entsprechend belegen.

Das Element server verweist direkt auf die Modul-Task-Struktur, welche unter anderem auch den Stack sowie andere nicht öffentliche Informationen enthält. Da die Task-Struktur zudem im Page-Frame-Areas abgelegt wird, kann sie ausschließlich innerhalb des Modul-Servers verwendet werden.

Anmelden von Funktionen

Im Abschnitt 2.2.3 wurde bereits das Anmelden von Symbolen mithilfe von Registrierfunktionen erläutert. Für ausgelagerte Module kann dieses Konzept nicht übernommen werden, da die Adressen innerhalb des übergebenen Objektes auf einen fremden Adressraum verweisen würden. Die Funktionszeiger des Objektes müssen nach der Abarbeitung der Registrierfunktion auf eine Funktion im gleichen Adressraum zeigen. Da sie später an verschiedenen Stellen innerhalb des Systemkerns verwendet werden, ist eine Änderung der Funktionsaufrufe nicht möglich. Zum Anmelden von Funktionen von ausgelagerten Modulen müssen die Registrierfunktionen des Systemkerns dahingehend verändert werden, dass sie solche Symbole erkennen können und die Funktionszeiger auf einen angepassten RPC-Stub umlegen.

Markieren von ausgelagerten Symbolen

Eine Unterstützung der Registrierfunktionen des Systemkerns ist sowohl für LKMs als auch für ausgelagerte Module notwendig, so dass die ursprüngliche Funktionalität der Funktionen erhalten bleiben muss. Zudem ist es nicht möglich, die Struktur des registrierten Objektes zu verändern, da dies unter Umständen zusätzliche Änderungen am übrigen Programmcode des Systemkerns mit sich bringen würde. Aus diesem Grund muss die Registrierfunktion anhand der vorhandenen Elemente des übergebenen Objektes erkennen können, wo das Symbol implementiert ist.

Hierzu wird innerhalb des Moduls nicht die Adresse des Symbols sondern dessen FID vermerkt. Dies kann nicht automatisch erfolgen und muss mittels des Defines `MODLD_SYM` manuell im Programmcode eingebracht werden. Die FIDs eines Moduls werden von Null an aufwärts durchnummeriert. Um innerhalb der Registrierfunktion eine Funktionsadresse von einer FID unterscheiden zu können, wird diese mit dem Define `MODLD_SYM_FLAG` binär verknüpft. Das Define ist auf den Wert `0xc1000000` gesetzt, so dass eine FID einer Adresse innerhalb des L4-Kerns oberhalb von drei Gigabyte entsprechen würde. Da dies innerhalb des L^{4}LINUX-Servers einer ungültigen Adresse entsprechen würde, kann die Registrierfunktion sicher sein, dass es sich hierbei um ein fremdes Symbol handelt.

Generierung der RPC-Stubs

Der Aufruf des Symbols erfolgt mittels der vorgestellten RPC-Architektur. Die Systemfunktionen rufen an Stelle des Symbols zunächst einen RPC-Stub auf, der die Anforderungen an den RPC-Client der Task weiterleitet. Eine genauere Erläuterung hierzu erfolgt noch einmal im folgenden Abschnitt.

Für jedes anzumeldende Symbol muss im Adressraum der aufrufenden Task ein eigener RPC-Stub erzeugt werden, da die MID und die FID des Symbols fest im Programmcode des Stubs eingetragen werden muss. Das Erzeugen des RPC-Stubs erfolgt zur Laufzeit durch Kopieren und Anpassen einer Schablonenfunktion, welche den Programmcode aus Listing 3 umfasst.

```
static unsigned long stub2(unsigned long arg0, unsigned long arg1)
{
      return client2(MODLD_DUMMY_MID, MODLD_DUMMY_FID, arg0, arg1);
}

static void stub2_end(void) {return;}
```

Listing 3: RPC-Stub

Zunächst wird für den neuen RPC-Stub ein Speicherbereich im Adressraum der Task allokiert. Die Größe des Bereiches lässt sich mittels des Symbols `stub2_end(...)` berechnen. Die Differenz zwischen beiden Symbolen entspricht der Größe der Funktion. Anschließend kann der Programmcode der Schablonenfunktion in den neuen Speicherbereich kopiert werden.

Damit die Funktion später verwendet werden kann, müssen der kopierte Programmcode relokiert und die Parameter angepasst werden. Die Adresse der Funktion `client2(...)` ist relativ zur Position der

Funktion. Da diese kopiert wurde, muss die relative Adresse anpasst werden. Hierzu wird im Programmcode nach dem Byte mit dem Inhalt `e8` gesucht. Dieses bezeichnet den Maschinenbefehl `call`. Die nächsten vier Byte geben die relative Adresse der aufzurufenden Funktion an. Diese kann aus der Startadresse der Schablonenfunktion und aus der Adresse der Kopie neu berechnet und anschließend in den Programmcode eingetragen werden.

Vor dem Aufruf der Funktion werden die Parameter auf den Stack gelegt. Damit der RPC-Stub den Aufruf der korrekten Funktion initiiert, müssen die ersten beiden Parameter entsprechend angepasst werden. Die Defines `MODLD_DUMMY_MID` und `MODLD_DUMMY_FID` werden so initialisiert, dass sie im Programmcode leicht wiedergefunden werden können und immer vier Byte belegen. Wie beim `call`-Befehl werden die Parameter der beiden `push`-Befehle im Programmcode manipuliert. Die korrekte FID ist im übergebenen Objekt gespeichert. Die MID wird dem Aufruf-Stack entnommen.

Der Aufruf-Stack

Jede Modul-Task besitzt zur Speicherung der Task-ID der RPC-Clients einen Aufruf-Stack. Die Behandlung von RPCs erfolgt mittels der `execute(...)`-Funktion der MODLD-Erweiterung. Sie ist verantwortlich für den Aufruf des korrekten öffentlichen Symbols. Zuvor wird die Task-ID des Clients auf den Aufruf-Stack gelegt. Jede Modul-Task sowie der L^{4}LINUX-Server enthält eine Implementierung der Funktion. Damit ist es möglich, innerhalb der öffentlichen Symbole die ID des Clients zu ermitteln. Nach der Abarbeitung der öffentlichen Symbols wird die ID vom Stack entfernt.

Der Stack fasst bis zu 256 Task-IDs. Sollte durch einen rekursiven Aufruf oder einen Programmfehler der Stack überlaufen, wird die Abarbeitung der Funktion abgebrochen und ein Fehlercode zurückgegeben. Eine Veränderung der Stack-Größe ist zur Laufzeit nicht möglich, dies muss im Programmcode der jeweiligen Task erfolgen.

Aufruf von ausgelagerten Funktionen

Während man bei der Initialisierung zwischen exportierten und angemeldeten Symbolen unterscheiden muss, erfolgt der Aufruf der Funktionen immer auf die gleiche Art und Weise. Der Aufruf eines fremden Symbols erfolgt immer über die im Abschnitt 3.4 erläuterte RPC-Architektur. Demzufolge muss diese vor dem ersten Aufruf eines Symbols bereits vollständig initialisiert sein. Beim L^{4}LINUX-Server erfolgt dies während der Initialisierung des Kerns und somit weit vor dem ersten Laden eines Moduls. Bei einem ausgelagerten Modul wird die Initialisierung von der init-Funktion der Modul-Erweiterung durchgeführt, somit vor dem Aufruf der `init`-Funktion des Moduls. Dies stellt sicher, dass die Voraussetzungen zur Verwendung eines exportierten Symbols immer vor dem ersten Aufruf erfüllt sind.

Abbildung 15 zeigt den Aufruf eines fremden Symbols mittels der im Abschnitt 3.4 vorgestellten RPC-Infrastruktur. Die in Abbildung 14 dargestellte Interaktion zwischen dem RPC-Client und dem RPC-Server ist in der RPC-Client- und der RPC-Server-Funktion zusammengefasst und wird hier nicht weiter dargestellt.

Der Aufruf eines fremden Symbols wird immer über einen RPC-Stub geleitet. Bei exportierten Symbolen wird der Stub beim Übersetzen des Moduls hinzugefügt und ist zusätzlich für das Ermitteln der MID und der FID verantwortlich. Eine ausführlichere Beschreibung zur Generierung des Stubs erfolgt im Abschnitt 4.1. Das Ermitteln der MID sowie der FID des Symbols ist lediglich beim ersten Aufruf des Stubs notwendig und für spätere Aufrufe werden die gewonnenen Werte in einer privaten Variable vermerkt.

Nach dem Auflösen der FID und MID wird der Aufruf vom RPC-Stub an die RPC-Client-Funktion weitergeleitet. Diese erwartet die MID und die FID des Symbols als erste Parameter. Darauf folgen die Argumente der aufzurufenden Funktion in der Originalreihenfolge. Die RPC-Client-Funktion legt die

Argumente auf den Argumente-Stack und ruft anschließend mittels des RPC-Interfaces der Modul-Task die `execute(...)`-Funktion der fremden Task auf.

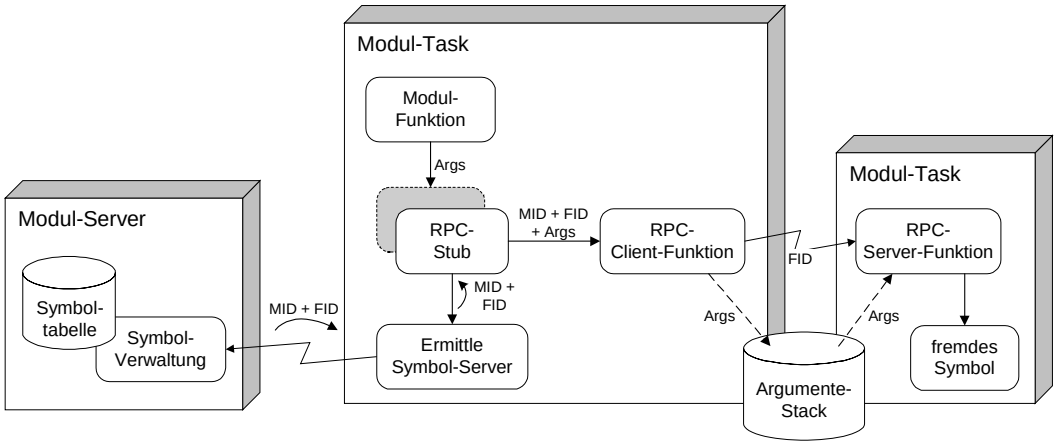

Abbildung 15: Aufruf exportierter Symbole

Da für die RPC-Client-Funktion keine variable Parameterliste implementiert wurde, enthält jede Modul-Task für jede mögliche Anzahl an Argumenten eine RPC-Client-Funktion. Die aktuelle Implementierung unterstützt bis zu 8 Argumente. Sollte der Aufruf einer Funktion mit mehr als 8 Argumenten notwendig werden, müsste die Implementierung entsprechend erweitert werden[14]. Der RPC-Stub ruft immer die entsprechende Funktion mit der passenden Anzahl an Argumenten auf. Dies ist notwendig, damit die Stubs von anzumeldenden Funktionen lediglich den Funktionsaufruf enthalten. Anderenfalls wäre das Kopieren und Anpassen der Stubs zur Laufzeit wesentlich komplizierter und unter Umständen abhängig von der Compiler-Version.

Der Argumente-Stack

Da für die Ausführung der ausgelagerten Funktionen nur die `execute(...)`-Funktion zur Verfügung steht, müssen die Argumente separat übergeben werden. Hierfür wurde der Argumente-Stack eingeführt. Er befindet sich innerhalb des MODLD-Areas und ist somit für alle Modul-Tasks sowie den L^{4}LINUX-Server verwendbar.

Die Initialisierung des Stacks erfolgt bei der Übergabe der Grenzen des MODLD-Areas durch den L^{4}LINUX-Server. Die Adresse des aktuellen Stack-Zeigers befindet sich ebenfalls im MODLD-Area und wird bei der Initialisierung der Modul-Tasks als Argument der Funktion `init_module(...)` in deren Adressraum übertragen. Da der L^{4}LINUX-Server diese Funktion nicht implementiert, erfolgt die Übergabe der Zeigeradresse im Rückgabewert der Funktion `modld_set_modld_area(...)`. Sie erwartet die Anfangsadresse sowie die Größe des MODLD-Areas und muss vom L^{4}LINUX-Server in jedem Fall aufgerufen werden, da die Grenzen des Bereiches von der Konfiguration des Servers abhängig sind.

Der Argumente-Stack besteht aus vier Seiten, wovon die mittleren beiden Seiten für den Stack verwendet werden. Die erste und die vierte Seite dienen der Erkennung von Stack-Überläufen. In der ersten Seite wird zudem der Zeiger für die aktuelle Adresse des Stacks abgelegt, so dass es möglich ist, dass alle Module sowie der L^{4}LINUX-Server den aktuellen Stand des Stacks erhalten und ihn entsprechend füllen können. Da die aktuelle Portierung des Linux-Kerns nicht Multiprozessor-fähig

[14] Das vorgestellte Konzept unterstützt momentan nicht den Aufruf von Funktionen mit einer variablen Parameterliste. Da diese Art von Funktionen nur selten vorkommt, wurde auf eine entsprechende Erweiterung der RPC-Stubs verzichtet. Stattdessen muss eine lokale Implementierung der Funktion eingesetzt werden.

ist, kann auf die Verwendung mehrerer Stacks verzichtet werden. Sollte sich dies in späteren Versionen ändern, ist an dieser Stelle eine Erweiterung des Konzeptes notwendig, so dass die Integrität des Stacks durch unabhängige Ausführungspfade nicht beeinflusst wird. Dies kann entweder mithilfe von Locks oder durch die Einführung mehrerer Stacks realisiert werden.

3.6 Zugriffskontrolle

Die Verwendung von getrennten Adressräumen erlaubt nicht nur den Schutz von bestimmten Speicherbereichen, sondern ermöglicht zusätzlich die Implementierung einer Zugriffskontrolle auf Task-Ebene. Der Aufruf von öffentlichen Symbolen ist nur über eine IPC möglich, unabhängig davon, ob eine Funktion aufgerufen oder eine Variable gelesen werden soll. In jeden Fall muss eine IPC in Form eines RPCs an die Task geschickt oder ein Page-Fault ausgelöst werden.

Die Verwendung von IPCs ermöglicht die Identifizierung der Task-ID des Senders. Zusammen mit der übermittelten FID kann für jede öffentliche Funktion eine separate Zugriffskontrolle auf der Seite des Empfängers durchgeführt werden. Bei Zugriffen auf öffentliche Variablen ist eine Unterscheidung hinsichtlich des adressierten Symbols nicht möglich. Da die L4-Architektur lediglich das Einblenden von ganzen Seiten gestattet, kann nicht verhindert werden, dass sich innerhalb der Seite keine anderen Symbole befinden. Aus diesem Grund muss beim Zugriff auf eine öffentliche Variable immer die Verwendung des gesamten öffentlichen Speicherbereiches der Modul-Task gestattet werden.

3.6.1 Nutzerschnittstelle

Der Inhalt der ACL muss bereits vor dem Übersetzen des Systemkerns oder des Moduls festgelegt werden. Eine Schnittstelle zur Konfigurierung der ACL zur Laufzeit ist bisher nicht implementiert. Dies hat zwei Gründe: Zum einen stellt die Implementierung nur eine Architektur zur Durchsetzung einer Zugriffskontrolle bereit, welche das Einbindung von zusätzlichen Schnittstellen auch zu einem späteren Zeitpunkt erlaubt. Zum anderen würde eine solche Schnittstelle die Integrität der ACL gefährden. Wenn es generell nicht möglich ist, die ACL zur Laufzeit zu verändern, kann sie von einem Angreifer auch nicht nachträglich erweitert werden.

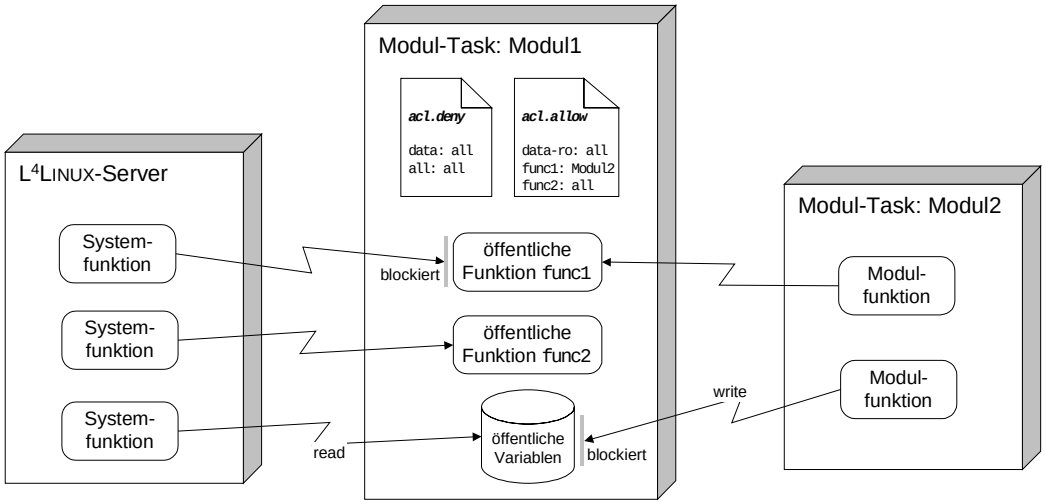

Abbildung 16: Zugriffskontrolle

Die Konfigurierung der ACL erfolgt mithilfe der beiden Dateien **acl.deny** und **acl.allow**. Die Datei **acl.deny** enthält die Negativliste und die Datei **acl.allow** die Positivliste für Zugriffe auf Symbole des Moduls. Die Definition der ACL erfolgt Modul-bezogen. Dass heißt, jedes Modul sowie der L4LINUX-Server kann mithilfe von eigenen Konfigurationsdateien die Zugriffsrichtlinien für seine

Symbole festlegen. Es existieren keine globalen Dateien. Bei der Auswertung der Konfigurationsdateien wird immer mit der Negativliste begonnen, so dass Regeln aus der Datei **acl.allow** bereits vorhandene Regeln überschreiben. In Abbildung 16 ist eine mögliche Konfiguration für ein Modul mit zwei öffentlichen Funktionen dargestellt.

Die Definition einer Regel erfolgt immer nach dem Schema:

```
[Symbolliste] : [Client-Liste]
```

wobei die Elemente der einzelnen Listen mittels eines Leerzeichens getrennt werden. Sowohl für die allow- als auch die deny-Dateien existieren vorgefertigte Schlüsselwörter:

data:
: Das Schlüsselwort data wird als Symbol verwendet und bezeichnet die öffentlichen Variablen der Task. Die Regel bezieht sowohl lesende als auch schreibende Zugriffe ein.

data-ro:
: Mittels des Schlüsselwortes data-ro können die lesenden Zugriffe gesteuert werden. Sobald eine Task schreibend auf den Datenbereich zugreifen kann, ist sie auch in der Lage den Inhalt zu lesen.

all:
: Um nicht alle Symbole und Clients einzeln aufzählen zu müssen, existiert das Schlüsselwort all. Es kann sowohl als Symbol als auch als Client verwendet werden.

l4linux:
: Der L^{4}LINUX-Server wird mittels des Schlüsselwortes l4linux bezeichnet. Modul-Tasks werden über den Namen ihrer Objektdatei adressiert.

Die Regeln der Datei **acl.deny** der Abbildung 16 verbieten den Zugriff auf alle Daten, sowohl lesend als auch schreibend und die Verwendung jeder Funktion für jede L4-Task. Die allow-Datei ermöglicht allen Tasks einen lesenden Zugriff auf die öffentlichen Variablen, sowie die Ausführung der Datei func2. Für das Modul_2 wird zusätzlich die Ausführung der Funktion func1 gestattet. Sollte das Modul_1 noch weitere Symbole enthalten, so können sie von keiner Task ausgeführt werden.

3.6.2 Aufbau der ACL

Der Aufbau der ACL ist stark an dessen Aufbewahrungsort gebunden. Beim Erstellen des Konzeptes haben sich zwei mögliche Orte ergeben. Zum einen kann jedes Modul seine individuelle ACL verwenden und innerhalb seines eigenen Adressraumes ablegen. Dies hätte den Vorteil, dass beim Aufruf von öffentlichen Funktionen der RPC-Handler direkt auf die Daten zugreifen könnte. Demgegenüber besteht jedoch der Nachteil, dass eine Konfigurierung der Liste zur Laufzeit nur über eine zusätzliche Schnittstelle möglich wäre. Diese Schnittstelle müsste für jedes Modul und den L^{4}LINUX-Server implementiert werden. Zudem kann nur schlecht verhindert werden, dass die ACL nicht nachträglich durch einen Angreifer manipuliert wird. Insbesondere beim L^{4}LINUX-Server, dessen Programmcode durch LKMs zur Laufzeit verändert werden kann, wäre eine sichere Verwahrung der ACL kaum möglich.

Aus diesem Grund wird die Verwaltung der ACL vom Modul-Server übernommen. Dessen Programmcode kann zur Laufzeit nicht verändert werden und die Integration einer zusätzlichen Schnittstelle ist leicht möglich. Da er innerhalb des Systems von jeder Task angesprochen werden kann, bleibt auch der Ort der Implementierung der Nutzerschnittstelle frei wählbar. Zusätzlich kann der notwendige Speicherbedarf für die ACL wesentlich reduziert werden, da viele Informationen, in Abbildung 17 grau dargestellt, bereits in der Modul-Task-Liste und in der Symboltabelle gespeichert sind.

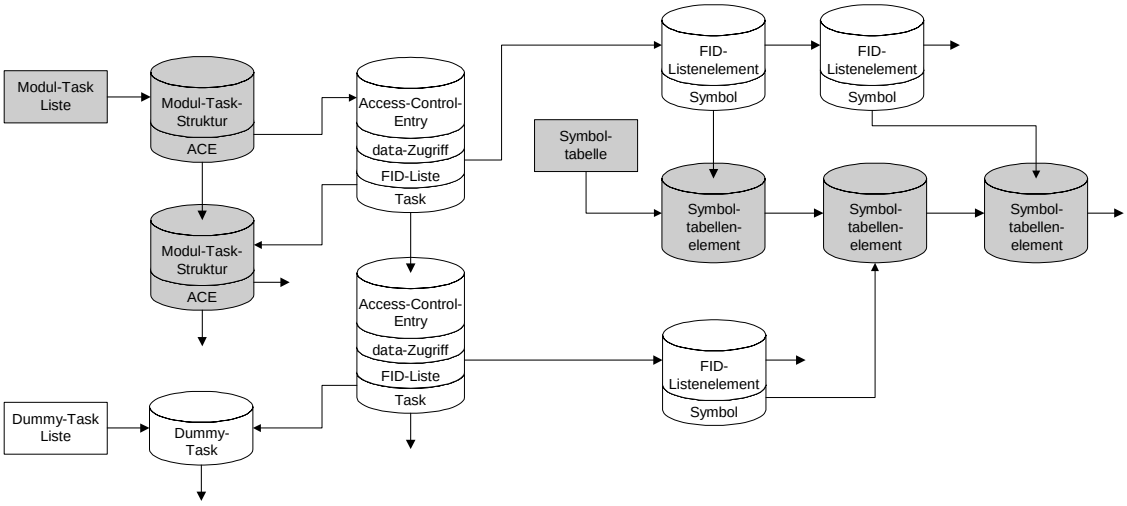

Abbildung 17: Aufbau der *Access Control List*

Die ACL einer Modul-Task besteht aus untereinander verketteten *Access Control Entries* (ACEs), welche die Zugriffsrechte für eine Task enthalten. Enthält das ACE keine Verweise auf eine Modul-Task, so gilt sie für alle Task und entspricht somit einer `all`-Regel. Die Zugriffsrechte auf den Datenbereich werden mittels des `data`-Elementes spezifiziert. Die Liste der FIDs gibt an, welche Funktionen die Task verwenden darf.

Das `data`-Element

Das `data`-Element ist immer Bestandteil des ACE. Da ein ACE unter Umständen nur für Funktionssymbole gelten soll, kann das `data`-Element als ‚ungenutzt' markiert werden. In diesem Fall wird das ACE bei der Zugriffskontrolle für Variablen vom Modul-Server nicht beachtet.

Bezieht sich das ACE auf eine `data`-Regel, so gibt das Element an, ob auf den gesamten öffentlichen Datenbereich der Task lesend, schreibend oder gar nicht zugegriffen werden darf. Insgesamt kann das `data`-Element somit vier verschiedene Werte annehmen. Sollte es vorkommen, dass für eine Task mehrere ACEs vorhanden sind, so wird der Wert mit den meisten Zugriffsrechten ausgewählt.

Die Liste der Funktionssymbole

Die Zugriffsrechte für öffentliche Funktionssymbole einer Modul-Task werden in einer separaten Liste verwaltet, die mit dem zugehörigen ACE verknüpft wird. Jedes Element der Liste enthält einen Verweis auf einen Eintrag der Symboltabelle und das Zugriffsrecht `allow` oder `deny`. Existieren zwei Elemente, die sich auf das gleiche Symbol beziehen, jedoch verschiedene Zugriffsrechte spezifizieren, wird der Zugriff gewährt[15].

Enthält ein Objekt der FID-Liste keinen Verweis auf ein Symbol, so bezieht sich die Regel auf alle öffentlichen Funktionen des Moduls. Es repräsentiert somit eine `all`-Regel für Funktionssymbole.

[15] Die aktuelle Implementierung überprüft beim Einfügen von Elementen nicht, ob bereits ein passendes Objekt vorhanden ist. Aus diesem Grund kann es vorkommen, dass für ein Funktionssymbol mehrere Objekte vorhanden sind. Dies ist jedoch immer auf einen Konfigurationsfehler zurückzuführen, so dass die Vermeidung doppelter Objekte dem Nutzer obliegt.

Dummy-Tasks

Das Konzept der *Stack-Moduls* erlaubt die Verwendung von öffentlichen Symbolen innerhalb eines anderen Moduls. Beim Laden der Module muss die Reihenfolge so gewählt werden, dass zu keinem Zeitpunkt unaufgelöste Symbole im Kern enthalten sind. Betrachtet man das Beispiel aus Abbildung 16, würde zunächst das `Modul_1` und anschließend das `Modul_2` geladen werden. Da beim Laden des Moduls dessen ACL an den Modul-Server übergeben wird, kann dieser das ACE für das `Modul_2` nicht korrekt initialisieren. Die Modul-Task-Liste enthält zu diesem Zeitpunkt noch keinen Eintrag für das `Modul_2`.

Zur Auflösung solcher Direktiven wird eine Modul-Dummy-Liste definiert. Dessen Objekte enthalten lediglich die Namen der Module, die in ACEs verwendet werden. Beim Laden eines neuen Moduls, wird die Modul-Dummy-Liste durchsucht und eventuell enthaltene Referenzen werden auf das soeben erstellte Modul-Listen-Element umgelegt.

3.6.3 Durchsuchen der ACL

Das Durchsuchen der ACL wird entweder vom Pager des Modul-Servers oder vom RPC-Handler der Modul-Task initiiert. Der Pager des Modul-Servers erhält die Task-ID des Clients aus den Parametern des Page-Faults. Mithilfe der Adresse ermittelt er die Task-Struktur des betroffenen Moduls. Dieses enthält einen Verweis auf die ACL. Ist der Zeiger nicht belegt, wird der Zugriff gewährt, anderenfalls wird die ACL nach einem passenden Element durchsucht. Existiert sowohl ein `all`- als auch ein Task-spezifisches Element, wird die `all`-Regel ignoriert.

Beim Aufruf einer Funktion sendet der RPC-Handler ein Signal mit der Task-ID des Clients und der adressierten FID an den Modul-Server. Dieser erhält aus den Parametern des Signals zusätzlich die Task-ID des Moduls. Das Durchsuchen erfolgt ähnlich wie beim Zugriff auf den Datenbereich des Moduls. Zunächst wird ein passendes ACE ermittelt und anschließend die FID-Liste durchsucht. Existieren mehrere mögliche Elemente, so wird die genauste Regel, hinsichtlich Task und FID, mit den meisten Zugriffsrechten genommen.

Enthält die ACL eine `all`-Regel für Clients mit einem passenden Eintrag für das adressierte Symbol und eine Task-spezifische Regel mit einer `all`-Regel für Symbole, so wird der Wert aus dem Task-spezifischen Element übernommen.

Kapitel 4

Aufbau und Integration der Modul-Server-Umgebung

Die Schnittstelle des Modul-Servers wurde so gestaltet, dass jede L4-Task auf sie zugreifen kann, um ausgelagerte Module zu nutzen oder selbst Module auslagern zu können. Die hier vorgestellte Implementierung befasst sich im Wesentlichen mit der Auslagerung von LKMs des L^{4}LINUX-Servers und der Interaktion zwischen den einzelnen Komponenten. Hierzu ist es notwendig, den Programmcode der Module als auch des L^{4}LINUX-Servers zu erweitern. Während im Kapitel 3 auf die einzelnen Komponenten eingegangen wurde, beschreibt dieser Abschnitt die Generierung der Schnittstellen, die Integration in die L^{4}LINUX-Architektur und die hierzu notwendigen Komponenten einer L4-Task.

4.1 Hilfsprogramme zur automatischen Generierung der Schnittstellen

Die Interaktion zwischen den ausgelagerten Modulen und dem L^{4}LINUX-Server erfolgt mittels der im Abschnitt 3.4 eingeführten RPC-Architektur. Hierzu ist es notwendig, dass jedes Symbol innerhalb der Task eine eindeutige FID bekommt und die Anzahl der Parameter bekannt sind. Dieser Abschnitt beschreibt zunächst eine Möglichkeit zum Ermitteln der Signatur der öffentlichen Symbole, bevor auf die zu generierenden Erweiterungen einer Modul-Task eingegangen wird.

4.1.1 Erweiterung der Symboltabelle

Jedes öffentliche Symbol muss innerhalb des Programmcodes mittels des `EXPORT_SYMBOL`-Makros markiert werden. Da öffentliche Symbole global definiert werden müssen, ist bereits sichergestellt, dass deren Name innerhalb des Kerns eindeutig ist. Da Zugriffe auf andere Symbole des L^{4}LINUX-Servers oder eines Moduls nicht möglich sind, müssen lediglich die Signaturen der öffentlichen Symbole untersucht werden. Abbildung 18 zeigt die Vorgehensweise zur Erweiterung des Objektcodes eines Moduls oder des L^{4}LINUX-Servers, so dass bei deren Initialisierung alle öffentlichen Symbole bekannt sind und an den Modul-Server übergeben werden können.

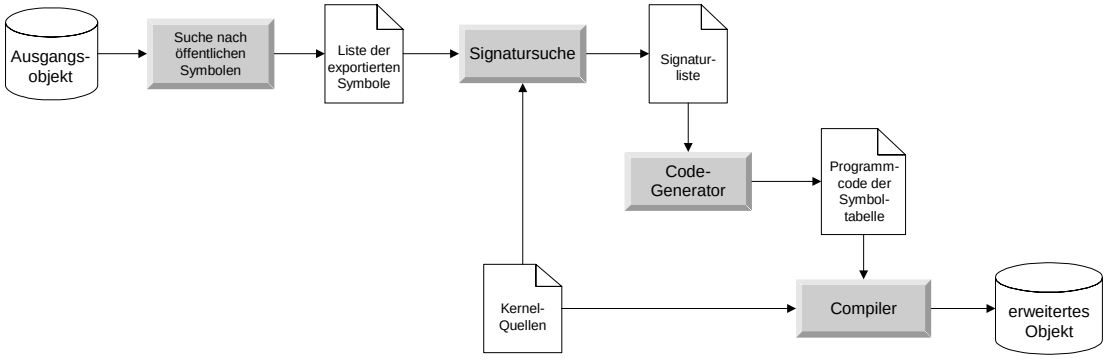

Abbildung 18: Erweiterung der Symboltabelle

Ein Modul registriert beim Modul-Server nur die Symbole, die es selbst implementiert und benötigt somit nur deren Signaturen. Da es nur schwer möglich ist, innerhalb des Programmcodes die fremden Symbole zu ermitteln, wird das Objekt zunächst ohne die Modul-Erweiterung übersetzt und nach

öffentlichen Symbolen durchsucht. Für jedes öffentliche Symbol wird beim Übersetzen des Moduls in der Objektsektion `__ksymtab` ein Eintrag angelegt. Der Name des Eintrages setzt sich aus dem Präfix `__ksymtab`, dem Symbolnamen und einem eventuell vorhandenen Identifier[16] zusammen. Mittels des Programms **`nm`** [21], welches alle Symbole einer Objektdatei auflistet, kann nun eine Liste der öffentlichen Symbole erstellt werden.

Die eigentliche Signatur des Symbols muss aus den Quellen des Kerns herausgesucht werden. Im Normalfall erfolgt eine Deklaration des Symbols bereits in einem der Header, so dass es meist ausreicht diese zu durchsuchen. In jedem Fall sollte die Funktion innerhalb der Quellen definiert sein, anderenfalls ist das Auslagern des Moduls nicht möglich und wird an dieser Stelle abgebrochen. Als Ergebnis der Signatursuche entsteht eine Textdatei mit den Signaturen der exportierten Symbole des Objektes. Diese wird anschließend dem Code-Generator übergeben.

Der Code-Generator erstellt aus der Signaturliste eine Quelle-Code-Datei, die die Funktion `modld_get_internal_syms(...)` implementiert. Wie bereits im Abschnitt 3.5.2 beschrieben, wird sie bei der Initialisierung des L^{4}LINUX-Servers und der Modul-Tasks aufgerufen. Darüber hinaus wird eine Datei mit den zugehörigen FIDs generiert. Jedes Symbol bekommt eine eindeutige FID zugeordnet, indem sie in aufsteigender Reihenfolge durchnummeriert werden.

Abschließend kann das Modul zusammen mit dem neuen Programmcode übersetzt werden. Der L^{4}LINUX-Server ermöglicht nur die Verwendung von öffentlichen Symbolen, so dass die soeben erzeugte Erweiterung vollständig ist. In die Objektdatei der Modul-Tasks müssen zusätzlich alle Funktionen hinzugefügt werden, die später mittels einer Registrierfunktion in einem fremden Adressraum angemeldet werden sollen. Eine Erläuterung hierzu erfolgt im nächsten Abschnitt.

4.1.2 Erweiterung eines exportierten Moduls

Bereits zuvor wurde erläutert, dass ein LKM, nachdem es geladen wurde, davon ausgeht, dass Funktionen beziehungsweise Variablen, die es nicht selbst definiert jedoch verwendet, im gleichen Adressraum zu finden sind. Befindet sich die Funktion in einem anderen Adressraum, muss der Aufruf über die im Abschnitt 3.4 eingeführte RPC-Architektur geleitet werden. Der Mechanismus basiert auf einer fest vorgegebenen Abfolge von Funktionsaufrufen mit bestimmten Parametern. Da die Nutzung des Mechanismus für den Anwender transparent sein soll, können Hilfsprogramme beziehungsweise Shell-Skripte eingeführt werden, die den notwendigen Programmcode automatisch generieren, übersetzen und zum Modul hinzufügen. Dieser Abschnitt gibt einen Überblick über die notwendigen Programme und deren Aufgaben.

Erzeugen der RPC-Stubs

Für jede Funktion, die nicht innerhalb des Moduls implementiert ist, muss ein RPC-Stub generiert werden. Dies muss vor dem Laden des Moduls erfolgen, da der Modul-Loader anderenfalls die Symbole bezüglich des Adressraumes des L^{4}LINUX-Servers auflösen und damit ungültige Adressen in den Objektcode des Moduls eintragen würde.

Hierzu wird, wie bei der Erzeugung der Symboltabelle, zunächst das Modul ohne jegliche Funktionalität der Modul-Erweiterung übersetzt. Die erzeugte Objektdatei kann nun mittels des Programms **`nm`** nach unaufgelösten Symbolen durchsucht werden. Anschließend wird wiederum die Signatursuche verwendet, um die Signaturen der unbekannten Funktionen zu ermitteln. Variablen können ausgelassen werden, da sie über das MODLD-Area angesprochen werden und die vom Modul-

[16] Ein Identifier wird immer dann hinzugefügt, wenn bei der Konfiguration des Kerns die Versionskontrolle aktiviert ist. Sie verhindert, dass Module, die nicht zusammen mit dem Kern übersetzt wurden, geladen werden können.

Loader ermittelte Adresse auch über Adressraumgrenzen hinaus gültig ist. Aus der Signaturliste kann nun der im Listing 4 dargestellte Stub-Code generiert werden.

```
unsigned long func(unsigned long arg1, unsigned long arg2)
{
      static modld_fid_t func__fid = {L4_INVALID_ID, -1};
      unsigned long mid;

      if (modld_invalid_fid(&func__fid) && modld_get_fid("func",
                                                         &func__fid)) {
            return -1;
      }

      mid = (unsigned long) func__fid.server.id.task;
      mid |= ((unsigned long) func__fid.server.id.lthread) << 16;

      return modld_rpc_func2(mid, (unsigned long) func__fid.fid, arg1,
                             arg2);
}
```

Listing 4: Quellcode eines RPC-Stubs

Das Beispiel zeigt die Implementierung des RPC-Stubs für die Funktion `func`, welche zwei Argumente erwartet. Zunächst werden mittels der Funktion `modld_get_fid(...)` die MID und die FID vom Modul-Server erfragt. Die Adressierung erfolgt über den Namen des Symbols. Das Ergebnis der Suche wird in dem als zweites Argument übergebenen Objekt vom Typ `modld_fid_t` abgelegt. Damit nicht bei jedem Aufruf des Stubs die MID und die FID beim Modul-Server erfragt werden muss, wird das Ergebnis permanent gespeichert und bei späteren Aufrufen wiederverwendet. Anschließend kann die RPC-Client-Funktion `modld_rpc_func2(...)` der Modul-Erweiterung aufgerufen werden.

Der Code-Generator erzeugt für jedes unbekannte Symbol eine Stub-Funktion mit deren Namen. Wenn der Code mit dem Modul zusammengebunden wird, sind alle Funktionsreferenzen aus dem Objekt-Code des Moduls aufgelöst. Variablen bleiben weiterhin unaufgelöst, so dass sie vom Modul-Loader erkannt und entsprechend behandelt werden.

Behandlung von anzumelden Funktionen

Funktionen, die mittels einer Registrierfunktion beim L^{4}LINUX-Server oder einem anderen Modul angemeldet werden, müssen in die Symboltabelle des Modul-Servers eingetragen werden. Wie bereits beschrieben, erfolgt das Eintragen der Symbole bei der Initialisierung des Moduls. Hierzu besitzt jedes Modul eine eigene Symboltabelle, die vom Code-Generator erstellt wurde. Das Auslesen der Tabelle ist mittels der Funktion `modld_get_internal_syms(...)` möglich. Das Einbinden von anzumeldenden Funktionen erfolgt, wie in Abbildung 19 dargestellt, parallel zur Suche nach öffentlichen Symbolen, so dass eine gemeinsame Liste entsteht, die der Signatursuche übergeben werden kann.

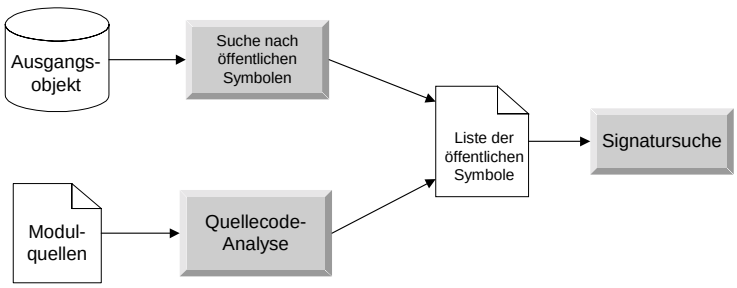

Abbildung 19: Einbinden von anzumeldenden Funktionen

Der Code-Generator erzeugt für jedes Symbol aus der Signaturliste einen separaten Eintrag. Um die Suche nach anzumeldenden Funktionen automatisieren zu können, müssen sie innerhalb des Programmcodes markiert werden. Hierzu wird das Makro `MODLD_SYMBOL(...)` verwendet.

4.1.3 Integration der ACL

ACLs sind immer Modul-bezogen, dies bedeutet, dass der Nutzer für jedes Modul eine separate Konfiguration erstellen muss. Da eine Modul-Task zur Laufzeit nicht auf die Konfigurationsdateien zugreifen kann und das Registrieren der ACL beim Modul-Server möglichst vor dem ersten Aufruf einer Funktion abgeschlossen sein sollte, wird die ACL des Moduls in dessen Objektcode integriert. Hierzu werden die Konfigurationsdateien von einem Code-Generator eingelesen, der die Funktion `modld_get_acl(...)` implementiert. Die Funktion liefert die Startadresse der ACL und die Anzahl der Elemente zurück. Das Übertragen der ACL an den Modul-Server wird bei der Initialisierung der Modul-Task beziehungsweise beim Starten des L^{4}LINUX-Servers ausgeführt.

Der erzeugte Programmcode wird zusammen mit den übrigen automatisch erstellten Funktionen übersetzt und zum Modul hinzugefügt. Demzufolge muss der Nutzer die Konfiguration vor dem Übersetzen des Moduls festlegen. Jede Änderung bedingt ein erneutes Übersetzen des Moduls.

4.2 Komponenten der RPC-Schnittstelle

Im Wesentlichen besteht die Architektur aus drei verschiedenen Servern. Hierbei handelt es sich um den Modul-Server, den L^{4}LINUX-Server und die einzelnen Modul-Tasks. Jede dieser Komponenten stellt einen Satz an RPCs zur Verfügung. Dieser Abschnitt enthält eine Beschreibung der Komponenten. Dabei wird weniger auf deren Implementierung sondern auf deren Integration in die L^{4}LINUX-Architektur eingegangen. Eine Erläuterung der Implementierung erfolgte bereits im Abschnitt 3.4.

4.2.1 IDL-Komponenten

Die Beschreibung der RPC-Schnittstelle erfolgt mittels der *Interface Definition Language* (IDL). Die IDL ist eine formale Sprache zur programmiersprachenunabhängigen Definition von Schnittstellen. Mithilfe eines RPC-Generators erfolgt die Generierung des zugehörigen Programmcodes. Der Generator ist abhängig von der verwendeten Programmiersprache als auch von der Zielplattform.

Für die L4-Architektur existiert ein speziell angepasster RPC-Generator, der aus einer `idl`-Datei und den zugehörigen Headern den Programmcode der RPCs erstellt. Um den erzeugten Programmcode nutzen zu können, ist es notwendig, eine passende Bibliothek und eine Handler-Funktion zu implementieren, wobei die Bibliothek jeweils vom RPC-Client verwendet wird.

Allgemeiner Überblick

Abbildung 20 zeigt die IDL-Komponenten der Modul-Server-Architektur. Es werden zwei verschiedene RPC-Interfaces definiert. Ein Interface wird vom Modul-Server implementiert, während das andere Interface in alle drei Server integriert wird.

Die `client-lib` enthält die RPC-Stubs des Modul-Servers und wird vom L^{4}LINUX-Server als auch von den Modul-Tasks verwendet. Die `idl`-Datei mit den Definitionen des Interfaces befindet sich im Verzeichnis `idl-modld`.. Im Folgenden wird zur leichten Unterscheidung der Schnittstellen der Verzeichnisname als Bezeichner der IDL verwendet. Zur Verwendung eines RPCs ist die Task-ID des Servers notwendig. Die ID des Modul-Servers kann beim Name-Server der L4-Architektur erfragt werden. Er registriert sich mit dem Namen `MODLD_RPC_SERVER` bei ihm.

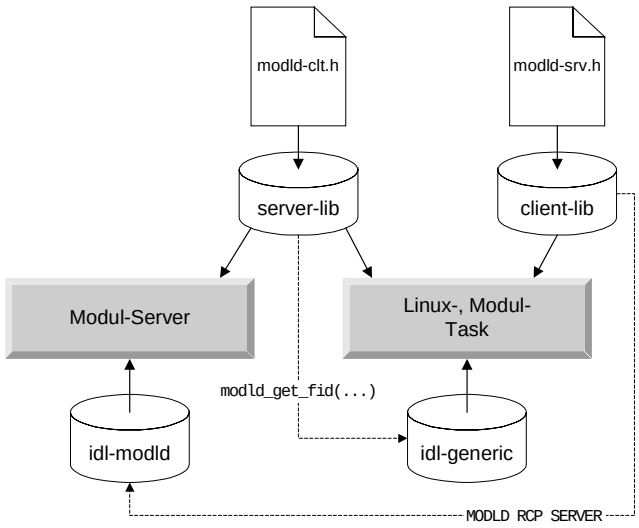

Abbildung 20: IDL-Komponenten

Jede Modul-Task und der L⁴LINUX-Server stellen ein Interface zur Verwendung ihrer öffentlichen Funktionen bereit. Die Beschreibung des Interfaces erfolgt in der `idl-generic` und wird sowohl vom Modul-Server als auch von den Tasks selbst verwendet. Demzufolge muss die Bibliothek in alle Server integriert werden. Die Adressierung des RPC-Handlers erfolgt über den im Abschnitt 3.5.2 beschriebenen Mechanismus.

4.2.2 RPC-Funktionen der `idl-modld`

Der Modul-Server stellt vier Dienste zur Verfügung. Dies ist die Modul-Verwaltung, ein rudimentäres Speichermanagement, die Symbol-Verwaltung und die Verwaltung der ACL. Jeder dieser Dienste kann mittels verschiedener RPCs angesprochen werden.

Modul-Verwaltung

Die Modul-Verwaltung umfasst das Laden sowie das Entfernen von Modulen. Funktionen zur Abfrage von Statusinformationen oder zum Auflisten der bereits geladenen Module sind bisher nicht implementiert.

`create(...):`
: Die Funktion `create(...)` wird beim Laden des Moduls verwendet. Sie bereitet das Erzeugen eines neuen Modul-Servers vor und liefert die Startadresse zur Relokation des Programmcodes.

`init(...):`
: Ebenso wie die Funktion `create(...)` ist die Funktion `init(...)` vom Modul-Server übernommen und dient dem Laden eines Moduls. Sie erzeugt und initialisiert die neue Modul-Task.

`delete(...):`
: Die Funktion `delete(...)` wird zum Entfernen einen Moduls verwendet.

Die Funktionen wurden bisher nur vom L⁴LINUX-Server verwendet, sind jedoch nicht an diesen gebunden. Damit ein Modul auch zur Laufzeit sicher geladen werden kann, wäre es sinnvoll, dies nicht über den L⁴LINUX-Server zu leiten, sondern das Laden durch eine externe Task durchführen zu lassen. Hierauf wird noch einmal genauer im Abschnitt 6.2.2 eingegangen.

Speichermanagement

Das Speichermanagement dient hauptsächlich der Allokation von privaten Bereichen. Die bisherige Implementierung enthält lediglich das Interface zum Page-Frame-Management des Modul-Servers.

`get_free_pages(...):`

Im Gegensatz zur gleichnamigen Funktion des L⁴LINUX-Servers erwartet der RPC nur die Anzahl der bereitzustellenden Seiten. Die Verwendung der zusätzlichen Flags ist nicht sinnvoll, so dass sie bereits von der Bibliotheksfunktion ignoriert werden. Die Bibliothek enthält darüber hinaus einen *Wrapper* für die Funktion `get_free_page(...)`, so dass dem Programmierer, wie gewohnt beide Funktionen zur Verfügung stehen.

`free_pages(…):`

Ähnlich wie `get_free_pages(...)` implementiert die Funktion `free_pages(...)` das vom LINUX-Kern bekannte Interface des Page-Frame-Managements. Der Aufruf der Funktion `free_page(...)` wird implementiert, in dem er auf `free_pages(...)` umgelegt wird.

Da der L⁴LINUX-Server nicht vom Modul-Server verwaltet wird, ist die Nutzung der Funktionen innerhalb des Servers nicht möglich. Sie stehen lediglich den Modul-Tasks zur Verfügung. Vor der Ausführung der Funktion überprüft der Modul-Server, ob die Anforderung von einer Task initiiert wurde, die er selbst erstellt hat. Ist dies nicht der Fall, wird der Zugriff verweigert.

Symbol-Verwaltung

Damit eine öffentliche Funktion eines Moduls oder des L⁴LINUX-Servers in einem fremden Adressraum verwendet werden kann, muss sie zunächst beim Modul-Server registriert werden. Darüber hinaus werden Funktionen benötigt, die das Auflösen eines Symbolnamens und das Ermitteln des zugehörigen Servers erlauben.

`register_sym(...):`

Die Funktion `register_sym(…)` kann von einer Task zum Registrieren eines öffentlichen Symbols verwendet werden. Die Funktion erwartet neben dem Namen der Funktion deren FID, die Anzahl der Argumente und den Typ des Rückgabewertes.

`get_fid(…):`

Der im Abschnitt 3.5.2 beschriebene Mechanismus benötigt die Funktion `get_fid(...)` zum Auflösen des Symbolnamens.

`register_rpc_handler(…):`

Da insbesondere beim L⁴LINUX-Server die ID des RPC-Threads nicht fest vorgegeben werden kann, wird eine zusätzliche Funktion zum Setzen der ID bereitgestellt. Die Funktion muss vom RPC-Handler-Thread selbst aufgerufen werden.

Für die Kontrolle der Zugriffe auf die öffentlichen Daten eines Moduls ist es notwendig, dass der Modul-Server weiß, in welchem Bereich sie zu finden sind. Um die Schnittstelle des Modul-Loaders nicht erweitern zu müssen, wird die Registrierung des Datenbereiches bei der Initialisierung des Moduls mithilfe eines RPCs durchgeführt.

`data_area(...)`

Der RPC erwartet die Grenzen des Text- und des Datenbereiches der Task. Beim L⁴LINUX-Server können diese leicht anhand der globalen Variablen `_text`, `_etext` und `_end` ermittelt werden. Für Module wurden, wie im Abschnitt 3.3.1 beschrieben, zusätzliche Variablen eingeführt.

Darüber hinaus sind zwei weitere RPCs zur Initialisierung des MODLD-Areas und zum Einblenden der Seiten notwendig.

`set_modld_area(...)`

Da die Position und die Größe des MODLD-Areas abhängig von der Konfiguration des L⁴LINUX-Servers ist und die Zugriffskontrolle vom Modul-Server durchgeführt wird, müssen die Grenzen des Bereiches zum Modul-Server übertragen werden.

`get_modld_area_pager(...)`

Zugriffe auf das MODLD-Area werden entweder direkt vom Pager des Modul-Servers oder vom Pager des L⁴LINUX-Servers behandelt. Letzterer kann die Seite, wie im Abschnitt 3.5.1 beschrieben wurde, nicht direkt verwenden. Hierzu muss ein Page-Request an den Modul-Server geschickt werden. Dieser stellt einen separaten Thread zur Behandlung der Requests bereit. Die Thread-ID kann mithilfe des RPCs `get_modld_area_pager(...)` erfragt werden.

Verwaltung der ACL

Die Verwaltung der ACL des Modul-Servers beschränkt sich bisher auf Funktionen zum Hinzufügen von ACL-Elementen. Das Löschen der ACL wird beim Entfernen des Moduls von Modul-Server selbst initiiert. Dies verhindert zwar die Möglichkeit einer dynamischen ACL, stellt aber sicher, dass die ACL eines Moduls zur Laufzeit nicht mehr manipuliert werden kann.

`add_func(...):`

Die Funktion `add_func(...)` ermöglicht das Einfügen einer Regel für eine Funktion und einen Client. Dies bedeutet, dass die Elemente einer Client-Liste, wie sie in der Konfigurationsdatei definiert werden können, separat eingefügt werden müssen.

`add_data(...):`

Für das Einfügen von ACEs, die den Zugriff auf die öffentlichen Variablen eines Moduls regeln, wird die Funktion `data_func(...)` angeboten.

Die Funktionen können sowohl von einer Modul-Task als auch vom L⁴LINUX-Server verwendet werden. Die Task-Liste des Modul-Servers enthält ein Element für den L⁴LINUX-Server, in welches dessen ACL eingehangen wird.

`check_func(...):`

Die Speicherung der ACL im Adressraum des L⁴LINUX-Servers bedingt die Bereitstellung einer Funktion zur Ermittlung der Zugriffsrechte. Hierzu wird der RPC `check_func(...)` angeboten. Er erwartet die FID des Symbols und die MID des RPC-Clients als Argumente. Aufgerufen wird die Funktion vom RPC-Handler des Moduls, welches das Symbol implementiert.

Die Zugriffskontrolle für den gemeinsamen Datenbereich erfolgt im Adressraum des Modul-Servers, so dass hierfür kein zusätzlicher RPC notwendig ist. Da der L⁴LINUX-Server keinen direkten Zugriff auf das MODLD-Area besitzt, sondern die Seiten vom MODLD-Area-Pager eingeblendet bekommt, kann die Zugriffskontrolle vollständig im Adressraum des Modul-Servers durchgeführt werden.

4.2.3 RPC-Funktionen der `idl-generic`

Die `idl-generic` wird in den L⁴LINUX-Server und in jede Modul-Task eingebunden und umfasst alle Funktionen, die aus einem fremden Adressraum heraus verwendet werden können. Die RPC-Architektur leitet alle Aufrufe von öffentlichen Symbolen über die `execute(...)`-Funktion, so dass innerhalb der `idl-generic` lediglich drei Funktionen notwendig sind.

`init(...):`

Hierbei handelt es sich um die Funktion `modld_init_module(...)` des Moduls. Sie wird nur vom Modul-Server am Ende der `init(...)`-Funktion des Modul-Interfaces aufgerufen.

`cleanup(...)`:

Analog zur `init(...)`-Funktion verweist der RPC `cleanup(...)` auf die Funktion `modld_cleanup_module(...)` des Moduls. Sie wird ebenfalls nur vom Modul-Server verwendet.

`execute(...)`:

Die `execute(...)`-Funktion leitet alle RPCs, die sich auf öffentliche Symbole des Moduls beziehen auf die entsprechende Funktion um.

Lediglich bei Modul-Tasks werden alle drei Funktionen verwendet. Der L⁴LINUX-Server implementiert das gleiche Interface, wobei die Funktionen `init(...)` und `cleanup(...)` leer bleiben. Durch die Verwendung eines einheitlichen Interfaces muss beim Modul-Server nicht zwischen einer Modul-Task und dem L⁴LINUX-Server unterschieden werden.

4.3 Integration in die L⁴LINUX-Architektur

Während der Modul-Server ein eigenständiges Programm darstellt, müssen die übrigen Komponenten in die L⁴LINUX-Architektur integriert werden. Nach Möglichkeit sollte dies für den Nutzer transparent erfolgen, so dass Anwendungsprogramme oder andere Kernel-Komponenten von der Erweiterung nicht beeinflusst werden. Dieser Abschnitt beschreibt die notwendigen Änderungen an der L⁴LINUX-Umgebung, so dass auslagerbare Module je nach Bedarf erstellt werden können.

4.3.1 Erweiterung des L⁴LINUX-Servers

Der L⁴LINUX-Server stellt eine Portierung des LINUX-Kerns auf die L4-Architektur dar. Hierzu wurden verschiedene Änderungen am LINUX-Kern vorgenommen. Die Modul-Server-Architektur ist wiederum eine Erweiterung des L⁴LINUX-Servers, so dass die Quellen an die Portierung angepasst wurden. Dieser Abschnitt beschreibt die Integration der Komponenten des Modul-Servers in den Kernel der L⁴LINUX-Umgebung.

Konfigurierbarkeit

Zum Testen der Quellen hat es sich als sinnvoll erwiesen, die neu hinzugefügten Komponenten mittels einer Option wahlweise an- und abschalten zu können. Hierzu wurde in der Kernel-Konfiguration im Menü für ladbare Kernel-Module der Eintrag ‚Enable external module loader' erstellt. Ist die Kernel-Option aktiviert, wird in der Datei `autoconf.h` das Define `CONFIG_L4_MODLD` definiert. Innerhalb des Programmcodes kann es verwendet werden, um verschiedene Konfigurationen zu implementieren.

Alle notwendigen Erweiterungen des L⁴LINUX-Servers zur Unterstützung der Modul-Server-Architektur sind mittels des Defines gekapselt. Ist die Option deaktiviert, wird der Kern in der ursprünglichen Implementierung übersetzt. Dies kann sinnvoll sein, um eventuelle Seiteneffekte der Implementierung ausschließen zu können.

Integration der Quellen

Da die Quellen des LINUX-Kerns schon einen recht großen Umfang angenommen haben, sind die einzelnen Teile in unterschiedlichen Verzeichnissen zu finden. Architekturabhängiger Programmcode befindet sich im Unterverzeichnis `arch`. Dort findet man unter anderem Unterverzeichnisse für die DEC-Alpha- (`alpha`), Sparc- (`sparc`, `sparc64`) oder Power-PC- (`ppc`) Architektur. Der Programmcode der L⁴LINUX-Portierung befindet sich ebenfalls in diesem Verzeichnis unter `l4`. Da die L4-Architektur wiederum auf verschiedene Plattformen portiert wurde, enthält das Verzeichnis ebenfalls Unterverzeichnisse für verschiedene Architekturen. Die hier vorgestellte Implementierung unterstützt bisher einzig die x86-Architektur und befindet sich somit im Unterverzeichnis `x86`.

Initialisierung des Systems

Der L⁴LINUX-Server beginnt seine Arbeit am Eintrittspunkt `startup_32`, der sich in der Datei `arch/l4/x86/boot/head.S` befindet. Hier erfolgt eine grundlegende Initialisierung des Systems. Abschließend wird die Funktion `__linux_startup(...)` aufgerufen. Sie generiert einen neuen Thread der seine Arbeit mit der Funktion `linux_startup(...)` beginnt.

Die Funktion `linux_startup(...)` löscht das `bss` Segment, initialisiert das Thread-Management und den im Abschnitt 2.4.2 beschriebenen trampoline-Mechanismus. Nachdem dies erfolgreich ausgeführt wurde, wird die Funktion `start_kernel(...)` aufgerufen. Die Funktion ist in der Datei `kernel/init.c` implementiert und befindet sich somit außerhalb des architekturabhängigen Teil des Programmcodes. Allerdings wird gleich zu Beginn die Funktion `setup_arch(...)` aufgerufen, welche wiederum architekturabhängig ist und sich in dem entsprechenden Unterverzeichnis befindet.

In der Funktion `setup_arch(...)` erfolgt die Initialisierung des Speichermanagements der L⁴LINUX-Task. Zunächst werden die Parameter vom Boot-Loader eingelesen und soweit es notwendig ist verarbeitet. Anschließend erfolgt die Initialisierung des Speichers des L⁴LINUX-Servers. Hierzu zählt die Ermittlung des verfügbaren Systemspeichers, die Initialisierung der `mm`-Struktur der init-Task und das Aufsetzen des Page-Frame-Managements. Nachdem dies erfolgreich abgeschlossen wurde, wird der idle-Thread erzeugt und die DROPS-Console initialisiert.

Ab diesem Zeitpunkt sind alle Voraussetzungen zur Initialisierung der Modul-Server-Erweiterung vorhanden, die Funktion `setup_arch(...)` wurde hier um den Aufruf der Funktion `modld_linux_init(...)` erweitert. Sie erwartet keine Parameter und meldet zunächst den L⁴LINUX-Server mit dem Namen GLINUX beim Namens-Server der L4-Architektur an. Dies ist notwendig, damit der Modul-Server den L⁴LINUX-Server innerhalb des Systems finden kann. Anschließend wird der Modul-Server mithilfe des Namens-Servers kontaktiert und ein Shake-Hands[17] durchgeführt. Als nächster Schritt wird der externe Pager aus Abschnitt 3.5.1 erstellt. An dieser Stelle ist die L⁴LINUX-spezifische Initialisierung der Modul-Server-Erweiterung abgeschlossen. Anschließend wird die Funktion `modld_init(...)` aufgerufen. Sie wird sowohl vom L⁴LINUX-Server als auch von der Modul-Task verwendet und ist verantwortlich für die Initialisierung der RPC-Architektur aus Abschnitt 3.4 und das Registrieren der öffentlichen Symbole, wie es im Abschnitt 3.5.2 beschrieben wurde. Nachdem die Funktion erfolgreich abgearbeitet wurde, werden die Grenzen des Text- und des Datensegmentes an den Modul-Server übertragen, dies erfolgt nach dem Verlassen der Funktion `modld_init(...)`, da die Grenzen beim L⁴LINUX-Server aus anderen Variablen ausgelesen werden müssen als bei einer Modul-Task. Abschließend werden die Funktionszeiger der Symboltabelle derart manipuliert, dass zunächst die Funktionen der Modul-Server-Erweiterung aufgerufen werden.

Anschließend wird die Initialisierung des Systems fortgesetzt. Durch die frühe Integration der Erweiterung soll erreicht werden, dass nur möglichst wenig Programmcode des L⁴LINUX-Servers ausgeführt wird und hinsichtlich möglicher Schwachstellen untersucht werden muss. Alle folgenden Programmfehler haben einen wesentlich geringeren Einfluss auf die Architektur, da die ID des L⁴LINUX-Servers, die öffentlichen Symbole und die Grenzen der Segmente dem Modul-Server bekannt sind und erst nach einem Neustart des Systems verändert werden können.

Erweiterung des Page-Fault-Mechanismus

Zur Integration des MODLD-Areas ist es notwendig, den Pager des L⁴LINUX-Servers zu erweitern. Der Pager ist in der Funktion `root_pager(...)` der Datei `arch/l4/x86/lib/l4_pager.c`

[17] Der Shake-Hands-Mechanismus wird innerhalb der Architektur immer beim Erzeugen eines Threads oder einer Task sowie beim Kontaktieren einer fremden Task durchgeführt. Hierbei tauschen beide Partner zwei Signale aus um sicherzustellen, dass der jeweils andere erreichbar ist.

implementiert und bisher nur verantwortlich für die Behandlung von Zugriffen auf das VMALLOC-Area. Zudem werden lediglich Zugriffsverletzungen eines L⁴LINUX-Service-Threads[18] behandelt. Für das MODLD-Area gelten die gleichen Voraussetzungen: Der Page-Fault muss von einem Service-Thread verursacht worden sein und muss auf eine Adresse des MODLD-Areas verweisen. Letzteres wird in der ursprünglichen Implementierung nicht akzeptiert.

Die Erweiterung fügt der Funktion einen weiteren zu untersuchenden Bereich hinzu. Adressiert der Zugriff ein Seite innerhalb des MODLD-Areas, wird die Funktionen `modld_page_fault(...)` aufgerufen. Die Funktion sendet einen Page-Request an den Modul-Server, damit dieser die Seite in den Adressraum des L⁴LINUX-Servers einblendet. Wird der Request nicht erfolgreich abgearbeitet, kann oder darf die Seite nicht in den Adressraum des L⁴LINUX-Servers eingeblendet werden. In diesem Fall wird ein Fehler zurückgegeben und die Behandlung dem L⁴LINUX-Server übergeben. Dieser sendet im Normalfall ein Segmentation-Fault an den verursachenden Prozess.

Erweiterung der Registrierfunktionen

Die Erweiterung der Registrierfunktionen ist die umfangreichste Änderung am L⁴LINUX-Server. Es kann leider nicht verhindert werden, dass jede in einem ausgelagerten Modul verwendete Registrierfunktion umgeschrieben werden muss. Darüber hinaus besteht nicht die Möglichkeit, eine Funktion für alle Registrierfunktionen zu implementieren, da jede Funktion für die Registrierung eines Objektes eines bestimmten Typs zuständig ist.

Die MODLD-Erweiterung enthält das Makro `IS_MODLD_FUNC(...)` welches als Argument einen Funktionszeiger erwartet. Entspricht die übergebene Adresse der im Abschnitt 3.5.2 beschrieben Markierung eines ausgelagerten Symbols, gibt das Makro einen Wert ungleich Null zurück. Mittels der Funktion `modld_replace_func(...)` kann dann der RPC-Stub generiert und die Adresse ausgetauscht werden. Wurde der Registrierfunktion ein Objekt eines ausgelagerten Moduls übergeben, wird im Folgenden davon ausgegangen, dass alle Funktionszeiger auf ausgelagerte Symbole verweisen und diese entsprechend ausgetauscht werden müssen. Eine erneute Überprüfung der Adresse ist somit nicht notwendig.

4.3.2 Erstellen eines ausgelagerten Moduls

Das Erstellen eines ausgelagerten Moduls sollte für den Nutzer so transparent wie möglich gestaltet werden. Dieser Abschnitt beschreibt die Änderungen am Programmcode und die notwendigen Schritte zum Übersetzen des Moduls.

Programmtechnische Richtlinien

Das größte Problem bei der Auslagerung von Modulen stellen die anzumeldenden Funktionen dar. Da eine automatische Erkennung innerhalb des Programmcodes nicht möglich ist, müssen sie vom Nutzer, wie im Abschnitt 3.5.2 beschrieben, markiert werden. Hierzu wird das Makro `MODLD_SYMBOL(...)` bereitgestellt, welches als Argument lediglich den Namen des Symbols erwartet. Darüber hinaus sollte sichergestellt werden, dass die Funktionen innerhalb des Programmcodes nicht als `static` definiert sind, da sonst unter Umständen das Übersetzen des Moduls nicht möglich ist. Variablen müssen hingegen nicht gesondert gekennzeichnet werden, da sie über das MODLD-Area exportiert werden.

[18] Wie bereits im Abschnitt 2.4.2 erläutert wurde, wird die Funktionalität des Systemkerns auf verschiedene Service-Threads verteilt. Nutzerprozesse wenden sich mittels eines IPCs an einen dieser Threads um Programmcode des L⁴LINUX-Servers auszuführen.

Zu Testzwecken erscheint es als sinnvoll das Modul ohne die Erweiterung zu übersetzen. Dies ist möglich, wenn innerhalb der Kernel-Konfiguration die Option für den Modul-Server deaktiviert wird. In diesem Fall werden alle Makros derart umdefiniert, dass der Programmcode der ursprünglichen Implementierung entspricht. Möchte der Nutzer innerhalb seines Moduls private Datenbereiche verwenden, muss er auf die Funktionen des Modul-Servers zurückgreifen. Da diese bei der Deaktivierung der Modul-Server-Unterstützung nicht mehr vorhanden sind, wurde zusätzlich das Makro `MODLD_FUNC(...)` eingeführt. So kann beispielsweise die Programmzeile:

```
private_data = get_free_pages(GFP_KERNEL);
```

durch die Zeile:

```
private_page = MODLD_FUNC(get_free_page)(GFP_KERNEL);
```

ersetzt werden. Dies bewirkt, dass die Funktion `get_free_page(...)` aufgerufen wird, wenn die Modul-Server-Unterstützung deaktiviert ist und anderenfalls die Funktion `modld_get_free_page(...)`, die als RPC im Modul-Server implementiert ist, verwendet wird.

Bei konsequenter Verwendung der bereitgestellten Makros kann der Nutzer ein Modul erstellen, dass sowohl im L^{4}LINUX-Server eingesetzt als auch ausgelagert werden kann. Bisher enthält der Modul-Server keine Funktionen, die für ein Modul auch dann notwendig sein könnten, wenn er nicht vorhanden ist.

Übersetzen eines Moduls

Zum Übersetzen eines ausgelagerten Moduls muss der Nutzer ein speziell angepasstes Makefile erstellen, dass zunächst den notwendigen Programmcode generiert und anschließend die MODLD-Erweiterungen einbindet.

Zunächst wird das Modul mit der Option `PRECOMPILE` übersetzt. Dies bewirkt, dass alle Erweiterungen zur Unterstützung des Modul-Servers deaktiviert werden und eine Objektdatei mit unaufgelösten Referenzen erstellt wird. Anschließend werden die Programme zur Generierung der MODLD-Erweiterung des Moduls ausgeführt. Sie erzeugen die folgenden Dateien:

`modld_[Modul-Name]_fid.h`:
: Die Datei enthält die Definition der FIDs und die extern-Deklarationen der verwendeten Funktionen der MODLD-Erweiterung.

`modld_[Modul-Name]_symbol.c`:
: Hier wird die Symboltabelle des Moduls und die Funktion `modld_get_internal_syms(...)` implementiert.

`modld_[Modul-Name]_stub.c`:
: Die RPC-Stubs für die unaufgelösten Symbole werden in der Datei `modld_stub.c` abgelegt.

`modld_[Modul-Name]_acl.c`:
: Die Datei enthält die Implementierung der Funktion `modld_get_acl(...)` und die ACL des Moduls.

Da unter Umständen mehrere Module innerhalb eines Verzeichnisses enthalten sein können, wird der Name des Moduls in die Dateinamen der MODLD-Erweiterung integriert. Jedes Modul erhält seine eigene Erweiterung. Dies ist auch bei der Datei `modld_[Modul-Name]_stub.c` notwendig, da jedes Symbol verschiedene Funktionen des Kerns verwendet und das Hinzufügen von unnötigem Programmcode so weit wie möglich vermieden werden soll.

Da erst nach dem Erzeugen des Programmcodes der MODLD-Erweiterung die Definitionen der FIDs feststehen und das Makro `MODLD_SYMBOL(...)` korrekt verwendet werden kann, muss das Modul an

dieser Stelle komplett neu übersetzt werden. Beim Linken werden nun die Objektdateien der MODLD-Erweiterung hinzugefügt, so dass das Modul anschließend keine unaufgelösten Funktionssymbole mehr enthält. Zu der im Abschnitt 3.5.2 beschriebenen Markierung der Programm- und der Datensektion kann das Skript `modld_link` aus dem Verzeichnis `scripts` verwendet werden.

4.3.3 Übersetzen des Gesamtsystems

Das gesamte System setzt sich aus den Dateien des Modul-Servers und den Erweiterungen des L⁴LINUX-Servers zusammen. Da einige Dateien erst beim Erstellen des Modul-Servers erzeugt werden, ist es erforderlich eine Reihenfolge beim Übersetzen des Gesamtsystems einzuhalten. Dieser Abschnitt beschreibt die hierfür notwendigen Voraussetzungen und die Übersetzungsreihenfolge.

Voraussetzungen

Die Schnittstellen der im Abschnitt 4.2.1 erläutert der IDL-Komponenten werden im Verzeichnisbaum des Modul-Servers beschrieben, verwendet werden die erzeugten Dateien jedoch größtenteils im L⁴LINUX-Server. Um auf sie zugreifen zu können, muss innerhalb des Makefiles der Modul-Erweiterung der Pfad zum Programmcode des Modul-Servers angegeben werden. Darüber hinaus ist es erforderlich, die Verzeichnisse mit den Headern der L4-Umgebung als Parameter dem Compiler zu übergeben.

Die L4-Umgebung gibt bereits eine Verzeichnisstruktur vor. Unter dem Hauptverzeichnis, welches in der Regel dieses den Namen `drops` trägt, befinden sich die Unterverzeichnisse `l4` und `linux22`. Das Verzeichnis `linux22` enthält die Portierung des LINUX-Kerns und die notwendigen Erweiterungen für die Integration des Modul-Servers. Im Verzeichnis `l4` befindet sich die L4-Umgebung, welche in den Unterverzeichnissen `include` alle Header der Umgebung, `pkg` die Programmdateien der L4-Server und `tools` die Programme zum Erstellen des Systems enthält. Die Implementierung des Modul-Servers ist dementsprechend im Verzeichnis `pkg/modld` zu finden.

Erfolgt die Installation der Programmdateien nach dem beschriebenen Schema sind keine weiteren Änderungen in den Makefiles des Modul-Servers und des L⁴LINUX-Servers notwendig. Die Verweise auf die Verzeichnisse der Header und der Dateien der Modul-Erweiterung sind relativ definiert, so dass der Ort des Hauptverzeichnisses `drops` variabel ist.

Reihenfolge

Bevor mit dem Übersetzen des Modul-Servers begonnen wird, sollte zunächst die gesamte L4-Umgebung einschließlich des L⁴LINUX-Servers übersetzt und getestet werden. Die Makefiles und Programmdateien des Modul-Servers sind so gestaltet, dass sie entsprechend der im Folgenden beschriebenen Reihenfolge übersetzt werden können. Da jedoch einige Header und Links erst beim Übersetzen der L4-Umgebung erstellt werden und diese unbedingt erforderlich sind, ist es vorteilhaft zunächst eine lauffähige L4-Umgebung zu erstellen.

Bedingt durch die Verwendung der IDL-Komponenten in den Quellen des L⁴LINUX-Servers ist es notwendig, zunächst diese zu erstellen. Hierzu sollten alle Komponenten des Modul-Servers übersetzt werden, sie enthalten die Objektdateien des RPC-Interfaces. Die Header des Servers werden mittels symbolischer Links im `include`-Verzeichnis der L4-Umgebung installiert, so dass ein Verweis auf die Quellen des Modul-Servers nicht notwendig ist. Anschließend kann der L⁴LINUX-Server übersetzt werden. Hierbei werden die Komponenten zur Integration des Modul-Servers erstellt und in den Kern eingebunden sowie die allgemeinen Bestandteile der Modul-Erweiterung erzeugt. Erst anschließend ist es möglich ein Modul zu übersetzen.

Kapitel 5

Leistungsbewertung des Gesamtsystems

Ein Ziel der Arbeit war die Implementierung so zu gestalten, dass sie für den Anwender möglichst transparent erscheint. Hierzu zählt unter anderem eine möglichst einfache Integration in das Gesamtsystem sowie die Vermeidung von starken Leistungseinbußen. Eine Steigerung der Sicherheit eines IT-Systems ist immer mit Leistungsverlusten verbunden. Dieses Kapitel gibt einen Überblick über die zu erwartenden Performanceverluste beim Einsatz der hier vorstellten Architektur.

Da der aktuelle Stand der Implementierung Messungen an komplexen Modulen beziehungsweise praxisnahen Anwendungsbeispielen nicht ermöglicht, beschränken sich die Ausführungen auf exemplarische Aufrufe, wie sie später in der Praxis häufig zu erwarten sind. Abschließend erfolgt eine Analyse der Messergebnisse, die eine Interpretation für umfangreichere Anwendungsbeispiele ermöglichen soll.

5.1 Testumgebung

Durchgeführt wurden die Tests auf einem Pentium(III)-System mit einer Taktfrequenz von 600MHz und 256MB Hauptspeicher. Die Prozessoren verwenden jeweils einen eigenen externen *2nd-Level-Cache* mit einer Größe von 512kB. Die Taktfrequenz für die Caches ist auf den halben Prozessortakt reduziert. Der Systemspeicher befindet sich auf einem SDRAM-Modul, welches über den Systembus mit 100MHz angesprochen wird.

Da keine Vergleichsmessungen mit anderen Systemen vorgenommen werden, sondern lediglich der Ressourcenverbrauch ermittelt und bewertet wird, sind die durchgeführten Tests im Wesentlichen Hardware-unabhängig. Die im Folgenden beschriebenen Tests messen die notwendigen Prozessortakte zur Ausführung eines möglichen Anwendungsfalls. Da moderne Prozessoren die Ausführungsreihenfolge der Mikrobefehle mitunter sehr stark verändern, um die einzelnen Prozessor-Pipelines zu füllen oder um unnötige Wartezyklen beim Zugriff auf den 2nd-Level-Cache oder gar auf den Hauptspeicher zu vermeiden, ist es durchaus möglich, dass die hier ermittelten Ergebnisse auf einem anderen System abweichen.

Die Messung der Prozessortakte erfolgt mittels der Anweisung `rdtsc`. Sie schreibt den Inhalt des Prozessor-internen 64-bit Taktzählers in die Prozessorregister `eax` und `edx`. Durch Auslesen des Zählers vor der Ausführung der Operation und danach kann die Anzahl der benötigten Takte ermittelt werden.

5.2 Taktzyklen typischer Anwendungsfälle

Für eine Bewertung der Leistung der Architektur wird zunächst nur der Ressourcenverbrauch bei der Interaktion zwischen den einzelnen Komponenten gemessen. Sie gibt darüber Auskunft wie stark die Gesamtleistung des Systems durch die zusätzliche Interprozess-Kommunikation belastet wird. Hierbei werden drei verschiedene Anwendungsfälle betrachtet: Eine IPC mit Adressraumwechsel, IPC zwischen Threads einer Task und das Auflösen eines Page-Faults.

5.2.1 RPC zum Modul-Server

Die Kommunikation zwischen den einzelnen Modul-Tasks, dem Modul-Server sowie dem L⁴LINUX-Server bedingt immer einen Wechsel des Adressraumes. Realisiert wird die Kommunikation mithilfe von RPCs, welche neben der reinen IPC auch noch die Verarbeitung der Argumente beinhalten. Um Aussagen über die Leistungseinbußen treffen zu können, wurden die Takte für die Verarbeitung des gesamten RPCs gemessen.

Zunächst wurde der RPC `register_rpc_handler(...)` vermessen. Hierzu wurde vor dem Aufruf des RPCs innerhalb der Bibliothek des Clients der aktuelle Stand des Taktzählers ermittelt. In der Handler-Funktion des Modul-Servers wurde der Zähler erneut ausgelesen und zwar noch bevor die eigentliche Registrierung des RPC-Handlers durchgeführt wurde. Der RPC erwartet keine Argumente, so dass der ermittelte Messwert den benötigten Aufwand für den reinen RPC beinhaltet. Bei mehreren Durchläufen wurden durchschnittlich 5960 Takte benötigt.

Zusätzlich wurde der RPC `create(...)` vermessen. Die Funktion erwartet zwei Argumente, den Namen des Moduls als Zeichenkette und die Größe des Objektcodes als Zahlenwert. Die Ermittlung der Taktzyklen erfolgt an den gleichen Stellen wie bei der Vermessung des `register_rpc_handler(...)`-RPCs, so dass ein ähnlicher Wert zuzüglich des Aufwandes zur Verarbeitung der Argumente zu erwarten wäre. Durchschnittlich wurden für diesen Aufruf 6900 Takte benötigt. Dies bedeutet, dass die Übergabe der beiden Argumente zusätzlich ungefähr 1000 Takte in Anspruch nimmt.

5.2.2 IPC zwischen Threads eines Modul-Servers

Neben den Aufrufen von RPCs erfolgt insbesondere innerhalb des Modul-Servers mehrfach IPC zwischen verschiedenen Threads. Beispielsweise muss für die Allokation einer Speicherseite, wie sie im Abschnitt 3.3.2 beschrieben wurde, ein Signal an den initialen Thread des Modul-Servers geschickt werden.

Um die möglichen Auswirkungen auf die Gesamtleistung des Systems ermitteln zu können, wurde das Versenden eines Signals ebenfalls vermessen. Hierzu wurde der Mechanismus zur Allokation einer Speicherseite verwendet. Der Taktzähler wurde unmittelbar vor und nach dem Senden des Signals ausgelesen. Hierbei wurde festgestellt, dass eine IPC zwischen Threads der gleichen Task kaum zusätzlichen Aufwand bedeutet. Demzufolge hat die Verwendung von Task-internen Signalen nur geringe Auswirkung auf die Gesamtleistung des Systems.

5.2.3 Auflösen von Page-Faults

Zugriffe auf Variablen eines fremden Adressraumes werden, wie im Abschnitt 3.5.1 beschrieben, über den Page-Fault-Mechanismus realisiert. Dabei muss man zwischen Zugriffen einer Modul-Task und des L⁴LINUX-Servers unterscheiden.

Page-Faults einer Modul-Task

Speicherzugriffsverletzungen einer Modul-Task werden direkt durch den Pager des Modul-Servers behandelt. Dabei ist zunächst nicht entscheidend, ob auf eine private oder auf eine öffentliche Seite zugegriffen wurde. Der Pager behandelt beide Zugriffsverletzungen. Die benötigten Taktzyklen sind abhängig vom Umfang des sigma0-Protokolls zum Einblenden der Seite als auch von der Implementierung des Speichermanagements. Da im Modul-Server bisher lediglich das Page-Frame-Area umgesetzt wurde und für spätere Anwendungen mitunter komplexere Speichermanagement-Implementierungen zum Einsatz kommen, wurde auf eine Messung des Aufwandes an dieser Stelle verzichtet. Die Anzahl der benötigten Taktzyklen zum Einblenden einer Seite ist abhängig von der Implementierung des L4-Kerns und kann demzufolge hier ebenfalls außer Acht gelassen werden.

Wesentlich aufschlussreicher ist der benötigte Aufwand zum Einblenden einer Seite des öffentlichen Speicherbereiches. Hierbei muss der Pager zunächst die Seite vom externen Pager des L⁴LINUX-Servers anfordern, bevor er sie in den Adressraum der Modul-Task einblenden kann. Beim Anfordern der ersten Seite muss der Pager zunächst die Thread-ID des externen Pagers ermitteln. Da dies jedoch bereits bei der Initialisierung des Systems erfolgte und die ID anschließend in der Modul-Task-Struktur vermerkt wurde, kann der hierfür notwendige Aufwand vernachlässigt werden. Die Messungen ergaben, dass in der derzeitigen Implementierung durchschnittlich 33.000 Taktzyklen benötigt werden, um eine Seite des L⁴LINUX-Servers in den Adressraum einer Modul-Task einzublenden. Hierzu wurde zum Test auf eine globale Variable aus dem Adressraum des LINUX-Systemkerns zugegriffen und vor sowie nach dem Zugriff der aktuelle Stand des Prozessor-internen Taktzählers ausgelesen.

Page-Fault des L⁴LINUX-Servers

Greift der L⁴LINUX-Server auf eine Seite des MODLD-Areas zu, so muss diese vom Modul-Server angefordert werden. Ähnlich dem vorgegangenen Abschnitt, muss zunächst die Thread-ID des MODLD-Area-Pagers bestimmt werden. Da die ID zur Laufzeit nicht verändert wird, kann der hierfür notwendige Aufwand im weiteren Verlauf ebenfalls vernachlässigt werden.

Die Messergebnisse aus Tabelle 1 zeigen, dass der erste Zugriff auf eine Variable des MODLD-Areas ungefähr 200.000 Taktzyklen in Anspruch nimmt. Hierbei entfielen 130.000 Taktzyklen auf das Ermitteln der Thread-ID des Pagers und ca. 65.000 Takte auf das Einblenden der Seite. Das Auslesen der Seite dauerte durchschnittlich 330 Takte.

Operation	Takte
Ermitteln der Pager-ID	*130.000*
Einblenden der Seite	*65.000*
zusätzliche Operationen	*5.000*
Gesamtaufwand	200.000
Seitenzugriff	330

Tabelle 1: Taktzyklen zum Auflösen eines Page-Faults

Dies bedeutet, dass bei allen weiteren Zugriffen auf die gleiche Seite nur ungefähr 330 Takte und für das Einblenden einer zusätzlichen Seite ca. 65.000 Takte notwendig sind. Da das gesamte MODLD-Area vom Modul-Server verwaltet wird, ist das Ermitteln der Thread-ID des MODLD-Area-Pagers während der gesamten Laufzeit des Systems nur einmal notwendig.

5.2.4 Aufruf einer Funktion des L⁴LINUX-Servers

Der Aufruf einer Funktion des L⁴LINUX-Servers ist eine der Operationen, die am häufigsten zu erwarten ist. Da die Ausführungszeit davon abhängig ist, ob die Funktion zuvor schon einmal ausgeführt wurde, erfolgt die Auswertung der Messergebnisse für beide Fälle getrennt. Der folgende Abschnitt geht zunächst auf die notwendigen Operationen beim ersten Aufruf einer Funktion ein.

Erster Aufruf einer Funktion

Abbildung 21 zeigt den typischen Verlauf eines Funktionsaufrufes. Der obere Teil veranschaulicht die Initialisierung des RPC-Proxies und das Ermitteln der Proxy-ID. Hierzu zählt auch die Kontrolle der Zugriffsrechte. Im unteren Teil der Abbildung erfolgt der eigentliche Aufruf der Funktion.

Wird die Funktion das erste Mal aufgerufen, muss zunächst die FID und die MID vom Modul-Server erfragt werden. Hierzu ist ein Adressraumwechsel vom L⁴LINUX-Server zum Modul-Server notwendig. Darüber hinaus werden bei der Suche des Eintrages in der Symboltabelle die Namen der

Funktionen verglichen. Beides ist sehr aufwendig und schlägt sich auch im Messergebnis nieder. So vergehen durchschnittlich 330.000 Taktzyklen vom Aufruf der Funktion im Adressraum der Modul-Task bis zur eigentlichen Ausführung durch die `execute(...)`-Funktion beim L⁴LINUX-Server.

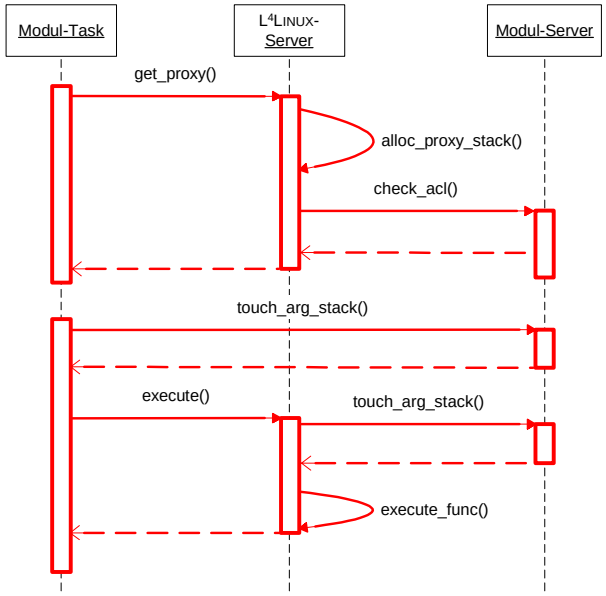

Abbildung 21: Aufruf einer Funktion des L⁴LINUX-Servers

Wird erstmals eine Funktion über die `execute(...)`-Funktion ausgeführt, muss der aktuelle Zeiger des Argumente-Stacks und der Argumente-Stack selbst in den Adressraum der Task eingeblendet werden. Hierzu sind bei der aufrufenden als auch bei der ausführenden Task mehrere Page-Faults aufzulösen. Der hierfür notwendige Aufwand wurde bereits im vorangegangenen Abschnitt erläutert. Da die Page-Faults nur beim ersten Zugriff auf den Argumente-Stack ausgelöst werden, kann der Aufwand im Folgenden vernachlässigt werden.

Messungen ergaben, dass für die erstmalige Ausführung einer Funktion im Adressraum des L⁴LINUX-Servers durchschnittlich 590.000 Taktzyklen benötigt werden. Dabei entfallen 95.000 Taktzyklen auf die Überprüfung der Zugriffsrechte, so dass ohne die ACL-Implementierung noch 440.000 Takte notwendig sind, um eine Funktion erstmals auszuführen. Da die Kontrolle der Zugriffsrechte scheinbar eine sehr teure Operation ist, sollte in Betracht gezogen werden, sie als zusätzliche Option bei der Konfiguration anzubieten oder, wie im Abschnitt 6.2.1 beschrieben wird, eine Umgestaltung der Implementierung vorzunehmen.

Zweiter Aufruf einer Funktion

Beim zweiten Aufruf einer Funktion entfällt das Auflösen des Symbolnamens und das Einblenden des Argumente-Stacks. So wurden für die gesamte Ausführung der Funktion beim zweiten Aufruf nur noch 130.000 Taktzyklen benötigt. Verzichtet man wiederum auf die ACL-Implementierung sind lediglich 58.000 Takte notwendig.

5.2.5 Aufruf einer Funktion einer Modul-Task

Der Aufruf einer Funktion einer Modul-Task beinhaltet eine aufwendige Initialisierung des RPC-Proxies, so dass die Ausführung der Funktion noch umfangreicher wird. Da die übrigen Operationen identisch zum vorangegangen Abschnitt sind, zeigt Abbildung 22 nur die Initialisierung des RPC-Proxies.

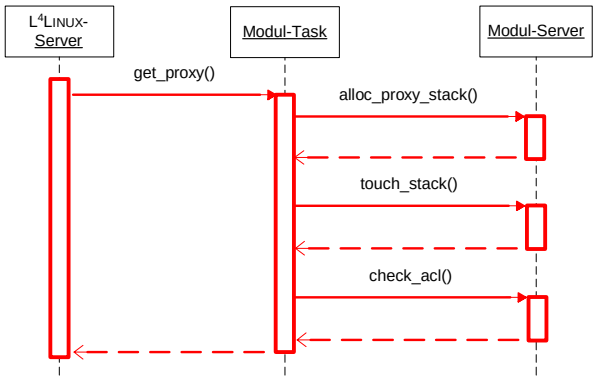

Abbildung 22: Initialisierung des RPC-Proxies

Der Speicher für den Stack des RPC-Proxies wird über den Modul-Server angefordert, so dass zunächst eine wesentlich aufwendiges Allokieren als beim L4LINUX-Server durchgeführt werden muss. Durchschnittlich sind hierfür 150.000 Taktzyklen notwendig. Da die neu angeforderte Seite nicht direkt in den Adressraum der Modul-Task eingeblendet wird, tritt beim ersten Zugriff auf den Stack ein Page-Fault ein. Für dessen Auflösung sind laut Abschnitt 5.2.3 noch einmal 30.000 Taktzyklen notwendig, so dass für die Ausführung einer Funktion eines ausgelagerten Moduls ein Mehraufwand von 180.000 Takten eingerechnet werden muss.

5.3 Analyse der Ergebnisse

Abschließend erfolgt in diesem Abschnitt noch einmal eine Zusammenfassung und Bewertung der Messergebnisse. Ein Vergleich mit Aufrufen von Funktionen von nicht ausgelagerten Modulen ist nicht sinnvoll, da hierbei nahezu keine zusätzlichen Taktzyklen notwendig sind[19]. In [2] wurden bereits Messungen der Performance der L4LINUX-Portierung durchgeführt. Allerdings weisen die Ergebnisse der Messungen eine nicht unwesentliche Differenz zu den hier ermittelten Werten auf, so dass zunächst in diesem Abschnitt die damaligen Werte auf die aktuelle Testumgebung übertragen werden. Abschließend erfolgt eine Bewertung der Messergebnisse der komplexeren Operationen hinsichtlich ihrer praktischen Relevanz.

5.3.1 Vergleich mit früheren Messergebnissen

In [2] wurde die Performance der L4LINUX-Portierung vermessen und mit einem Standard-LINUX-System sowie mit einem Mach-Kern-basierten System verglichen. Dabei zeigte sich, dass die Verwendung eines L4-Kerns lediglich zu Leistungsverlusten von unter 10 Prozent führt und insbesondere im Vergleich zu früheren Mikro-Kern-basierten Systemen, wie MkLinux, ein nicht unwesentlicher Leistungsgewinn erzielt werden konnte.

Testsystem und Messergebnisse

Die Tests wurden damals auf einem Pentium-System der ersten Generation durchgeführt. Der Hauptprozessor war ein Pentium mit 133 MHz ohne MMX-Einheit. Der Testrechner verfügte über 64 MB Speicher, der über den mit 66 MHz getakteten Systembus mit dem Hauptprozessor verbunden war. Der 2nd-Level-Cache hatte eine Größe von 256kB und wurde mit der gleichen Taktfrequenz wie der Systemspeicher angesprochen. Der eingesetzte Linux-Kern hatte die Version 2.0.21.

[19] Für die Ausführung einer lokalen Funktion werden bei einem kalten Cache durchschnittlich 48 und bei einem warmen Cache 38 Takte benötigt.

Tabelle 2 zeigt die ermittelten Takte zur Ausführung des `getpid(...)`-Systemrufs. Die auf dem Standard-LINUX-System gemessenen Taktzyklen dienten als Basis für die Bewertung der L⁴LINUX-Portierung. Unter Verwendung des im Abschnitt 2.4.2 erläuterten Trampoline-Mechanismus waren durchschnittlich 510 Taktzyklen zusätzlich notwendig. Der Mehraufwand lässt sich beim Einsatz einer eigenen Bibliothek, die den Wechsel zum L⁴LINUX-Server beim Aufruf einer Systemfunktion direkt ausführt, auf ca. 300 Taktzyklen reduzieren.

System	Takte
LINUX	223
L⁴LINUX	526
L⁴LINUX (mit Trampoline)	753

Tabelle 2: Kosten eines `getpid(...)`-Systemrufs

Die Messung der benötigten Taktzyklen erfolgte ebenfalls mit dem Maschinenbefehl `rdtsc`. Der Taktzähler wurde vor und nach dem Aufruf der Funktion ausgelesen, so dass der Messwert die komplette Abarbeitung der Systemfunktion umfasst. Um den Messwert nicht durch einen kalten Cache zu beeinflussen, wurde die Messung in einer Schleife wiederholt und anschließend der Durchschnitt gebildet. Der Pentium-Prozessor verwendet neben dem 2nd-Level-Cache des Mainboards einen internen L1-Cache mit einer geringeren Größe. Es kann davon ausgegangen werden, dass der Systemruf und die notwendigen Operationen im Systemkern nahezu vollständig in den L1-Cache abgelegt werden konnten, so dass Zugriffe auf den L2-Cache oder den Hauptspeicher nicht notwendig waren.

Übertragung der Messergebnisse

Der Aufruf der Funktion `getpid(...)` bedingt einen Wechsel in den Adressraum des L⁴LINUX-Servers, so dass die erzielten Werte mit den im Abschnitt 5.2 erläuterten Messergebnissen vergleichbar sein sollten. Jedoch weisen die Werte einen nicht unwesentlichen Unterschied auf. Da zudem die Anzahl der benötigten Taktzyklen für einen Wechsel des Adressraumes die Grundlage der hier vorgestellten Messergebnisse dargestellt, erfolgt in diesem Abschnitt eine Übertragung der im vorangegangenen Abschnitt aufgeführten Messwerte auf die im Abschnitt 5.1 beschriebene Testumgebung.

Evolution des Prozessors

Die Steigerung der Taktfrequenz der Prozessoren hat dazu geführt, dass Vergleiche über Generationsgrenzen hinweg nur schwer möglich sind. Jede neue Prozessorgeneration wird heutzutage bereits so geplant, dass im Folgenden die Taktfrequenz noch weiter gesteigert werden kann. Hierzu wird beispielsweise die Tiefe der Befehlspipeline erhöht oder die Anzahl der in Reihe geschalteten Transistoren verringert. Beides führt jedoch auch dazu, dass für einen Maschinenbefehl immer mehr Taktzyklen notwendig sind.

Die in Tabelle 3 aufgeführten Werte wurden mit Ausnahme des Pentium(III)-Systems aus [26] entnommen. Hierbei wurden wiederum die notwendigen Taktzyklen zur Ausführung des Systemrufs `getpid(...)` gemessen. Da für das hier verwendete Testsystem keine Messwerte vorhanden waren, wurde die Messung zusätzlich durchgeführt.

System	Takte
Pentium 166 MHz (ohne MMX)	125
Pentium II 450 MHz	292
Pentium III 600 MHz	*340*
Athlon 1 GHz	260

Tabelle 3: Vergleich der Prozessor-Generationen

Wie das Messergebnis zeigt, benötigt ein Pentium(II)- beziehungsweise ein Pentium(III)-System bereits ein Vielfaches an Taktzyklen zur Ausführung des Systemrufes. Ausgehend von dieser Tatsache wurden erneut Messungen auf der L⁴LINUX-Umgebung durchgeführt. Hierzu wurde wiederum das im Abschnitt 5.1 beschriebene Testsystem verwendet. Messungen ergaben, dass für die Ausführung des Systemrufes durchschnittlich 2200 Taktzyklen benötigt werden. Damit ergeben sich die in Tabelle 4 aufgeführten Skalierungsfaktoren.

System	LINUX (Takte)	L⁴LINUX (Takte)	Faktor
Pentium I – 133 MHz	223	753	3,38
Pentium III – 600 MHz	340	2.200	6,47

Tabelle 4: Systemvergleich für `getpid(...)`

Es bleibt immer noch ein Differenzfaktor zwischen dem Pentium- und dem Pentium(III)-System, der entweder auf die Kernel- oder die verschiedenen Programmversionen zurückzuführen ist. Die Tests wurden ebenfalls mehrfach in einer Programmschleife ausgeführt, so dass der Zustand des Caches keine Auswirkungen auf das Messergebnis haben sollte.

Auswirkungen eines kalten Caches

Auch unter Berücksichtigung der Architektur besteht immer noch ein nicht unwesentlicher Unterschied bei der Anzahl der benötigten Taktzyklen zum Wechsel des Adressraumes. Die im Abschnitt 5.2 durchgeführten Messungen wurden in die Implementierung des L⁴LINUX-Servers und des Modul-Servers integriert und jeweils bei der Initialisierung des Systems ausgeführt. Da der RPC `register_rpc_handler(...)` nur einmal aufgerufen wird, musste zur Ermittlung mehrerer Messwerte das System jeweils neu gestartet werden.

Da alle weiteren Messungen ähnlich durchgeführt wurden, müssen zusätzlich die Auswirkungen eines kalten Caches in die Betrachtungen einbezogen werden. Hierzu wurde die Programmschleife innerhalb des Testprogramms entfernt, so dass der Systemruf `getpid(...)` nur einmal ausgeführt wurde. Die neuen Messungen zeigten, dass mit einem kalten Cache durchschnittlich 5200 Taktzyklen zur Abarbeitung des Systemrufes notwendig sind. Dieser Wert entspricht ungefähr der Anzahl der benötigen Takte zur Ausführung des RPCs.

Messungen mit warmen Caches sind für die im Abschnitt 5.2 beschriebenen Anwendungsfälle nicht relevant. Bei der Auslagerung von Modulen ist nicht zu erwarten, dass die Adressraumwechsel in so kurzer Zeit durchgeführt werden, dass der Inhalt des Caches noch nicht überschrieben wurde. Um den Mehraufwand der hier vorgestellten Architektur so gering wie möglich zu halten, muss die Implementierung eines Moduls eher so gestaltet sein, das möglichst wenige Adressraumwechsel in kurzer Zeit notwendig sind.

5.3.2 Bewertung der Messergebnisse

Nachdem im vorangegangenen Abschnitt die Anzahl der benötigen Taktzyklen für einen Adressraumwechsel mit früheren Ergebnissen verglichen wurde, erfolgt in diesem Abschnitt eine Bewertung der komplexeren Operationen. Da bisher keine Vergleichswerte vorhanden sind, werden die gemessenen Werte hinsichtlich ihrer möglichen Auswirkungen auf die Gesamtleistung des Systems beurteilt.

Ausführen von öffentlichen Funktionen

Die benötigten Taktzyklen zum Ausführen von ausgelagerten Funktionen schwanken zwischen 50.000 im Idealfall und 800.000 im schlechtesten Fall. Es ist jedoch zu erwarten, dass während der Laufzeit des Systems wesentlich häufiger Funktionen des L⁴LINUX-Servers ausgeführt werden, so dass der

Mehraufwand für Funktionen eines Moduls die Gesamtleistung des Systems nicht übermäßig stark beeinträchtigen sollte.

Ein Modul meldet seine Funktionen beim L⁴LINUX-Server an. Bereits hierbei werden viele Operationen ausgeführt, die sehr aufwendig und damit bei allen weiteren Aufrufen nicht mehr notwendig sind. So wird beispielsweise der Argumente-Stack schon an dieser Stelle in den Adressraum der Modul-Task eingeblendet. Darüber hinaus enthält ein Modul in der Regel keine reinen Service-Funktionen[20], sondern eher aufwendige Operationen, zum Beispiel das Ver- und das Entschlüsseln von IP-Paketen oder Interrupt-gesteuerte Zugriffe auf I/O-Ressourcen, so dass der reine Aufruf der Modul-Funktionen in den meisten Fällen nur eine untergeordnete Rolle bei der Gesamtbelastung des Systems spielen sollte. Da bisher jedoch keine Tests mit vollständigen Modulen durchgeführt wurden, kann diese Aussage an dieser Stelle nicht mit Messwerten untermauert werden. Sie erscheint jedoch als naheliegend, wenn man die Programmcodes der einzelnen Module betrachtet.

Der notwendige Mehraufwand zur Ausführung von Funktionen des L⁴LINUX-Servers konnte auf 50.000 Taktzyklen reduziert werden. Das verwendete Testsystem aus Abschnitt 5.1 benötigt somit ca. 83,3µs mehr zur Ausführung einer Funktion des L⁴LINUX-Servers. Bei moderneren Systemen sollte sich der Mehraufwand entsprechend der Taktfrequenz verringern, so dass die Nutzung von ausgelagerten Modulen zunehmend transparenter werden sollte. Ein Ziel von Weiterentwicklungen sollte jedoch die Verringerung der notwendigen Adressraumwechsel sein. Auch wenn der reine Wechsel des Adressraumes nur 5.000 Taktzyklen benötigt, so zeigen doch die anderen Messwerte, dass zum Beispiel für die Auswertung der ACL, also eine komplexere Operation, bis zu 95.000 Takte notwendig sind[21].

Zugriff auf öffentliche Variablen

Der Aufwand für Zugriffe auf öffentliche Variablen wird im Wesentlichen durch das sigma0-Protokoll bestimmt. Das Ermitteln der jeweiligen Pager ist während der Laufzeit des Systems lediglich einmal notwendig, so dass Folgezugriffe entsprechend schneller bearbeitet werden können. Modul-Tasks erlangen über den Pager des Modul-Servers Zugriff auf die öffentlichen Variablen einer anderen Task, so dass die Anzahl der zu ermittelnden Pager, auf den externen Pager des L⁴LINUX-Servers und den MODLD-Area-Pager des Modul-Servers beschränkt werden kann.

Das Einblenden von ganzen Seiten hat zudem den Vorteil, dass nicht für jede Variable erneut ein Page-Fault ausgelöst wird. Befindet sich die Seite bereits im Adressraum der Task sind bei allen weiteren Zugriffen keine zusätzlichen Operationen notwendig. Da der Pager des Modul-Servers für alle Modul-Tasks zuständig ist, sind Seitenanforderungen an den externen Pager des L⁴LINUX-Servers unnötig, wenn die Seite bereits in den Adressraum einer anderen Modul-Task eingeblendet wurde. So lässt sich zusammenfassend sagen, dass der Mehraufwand bei Zugriffen auf öffentliche Variablen bezogen auf die Gesamtleistung des Systems sehr gering ausfallen sollte.

---

[20] Mit Service-Funktionen sind Funktionen gemeint, die nur einen geringen Funktionsumfang besitzen und allgemeingültige Operationen ausführen.

[21] Da keine früheren Messergebnisse zufinden waren und der vermessene Programmcode größtenteils durch den RPC-Generator erzeugt wurde, muss auf eine Auswertung der Messwerte an dieser Stelle verzichtet werden.

Kapitel 6

Zusammenfassung und Ausblick

Als Abschluss soll dieser Abschnitt die Ergebnisse der Arbeit noch einmal zusammenfassen und einen Ausblick auf mögliche Weiterentwicklungen oder Verbesserungen des Konzeptes geben.

6.1 Zusammenfassung

Ziel der Arbeit war die Bereitstellung von sicheren Speicherbereichen für Funktionen des Systemkerns von LINUX. Hierbei sollte zudem eine Zugriffskontrolle mittels einer ACL integriert werden, so dass Zugriffe auf einzelne Objekte feingranular differenziert werden können.

Im Kapitel 2 erfolgt die Beschreibung des Speichermanagements von LINUX und dessen Schwachstellen. Wie auch mit dem im Abschnitt 2.3 beschriebenen Konzept gezeigt wurde, existiert momentan keine Lösung des Problems. Die Konzepte scheitern im Wesentlichen immer an der Allmacht des Systemkerns von LINUX. Alle Mechanismen, die zum Schutz von Speicherbereichen in den Systemkern eingefügt werden, können durch Einbringen zusätzlichen Programmcodes im Nachhinein deaktiviert oder umgangen werden. Da ein monolithischer Kern alle Rechte innerhalb des Systems besitzt, kann keine Komponente hinzugefügt werden, die dessen Rechte wirksam einschränkt. Mit der im Abschnitt 2.4 beschriebenen L^{4}LINUX-Architektur ist es möglich die Privilegien des Systemkerns von LINUX wirksam einzuschränken, so dass Speicherbereiche des Systems vor Funktionen des LINUX-Kerns geschützt werden können. Allerdings werden die Möglichkeiten in der bisherigen Implementierung nur wenig genutzt, so dass das Problem lediglich in den L^{4}LINUX-Server verlagert wird. Der Server enthält den gesamten LINUX-Kern und LKMs werden, wie in der ursprünglichen Implementierung, gleichberechtigt in den Server eingefügt. Eine wirksame Unterteilung des Adressraumes ist in der aktuellen Implementierung nicht gegeben.

Ausgehend von der Aufgabe: Sichere Speicherbereiche für Funktionen des Systemkerns von LINUX bereitzustellen, wurde im Kapitel 3 ein Konzept zur Auslagerung von LKMs vorgestellt. Insbesondere der Mangel an wohldefinierten Schnittstellen innerhalb des LINUX-Kerns verhindert eine leichte Unterteilung der Funktionalität auf verschiedene weitestgehend unabhängige Komponenten. Einzig die Modul-Schnittstelle, wie sie im Abschnitt 2.2 beschrieben wurde, erlaubt die Auslagerung von Systemkomponenten, ohne dass deren Programmcode umfangreich verändert werden muss. Zur Auslagerung von LKMs war es zunächst notwendig eine Basisarchitektur zu erstellen, die den Programmcode des Moduls als eigenständige Task ausführbar macht. Hierzu wurden der Modul-Server und die Modul-Erweiterung eingeführt. Der Modul-Server ist für die Verwaltung der einzelnen Modul-Tasks verantwortlich. Hierbei wurde ein ähnliches Konzept verfolgt, wie es bereits für Nutzerprozesse in der L^{4}LINUX-Architektur verwendet wird.

Die im Abschnitt 3.3.1 beschriebene Unterteilung der Adressräume des L^{4}LINUX-Servers, des Modul-Servers sowie der einzelnen Modul-Tasks erlaubt die Bereitstellung von öffentlichen als auch von privaten Speicherbereichen. Zugriff auf private Speicherbereiche erhält nur die eigene Task, so dass für ein Modul die Möglichkeit geschaffen wurde, Bereiche im eigenen Adressraum anzulegen, die von Funktionen einer anderen Modul-Task oder des L^{4}LINUX-Servers nicht gelesen werden können. Der Programmcode eines ausgelagerten Moduls sollte ebenfalls vor unautorisierten Zugriffen geschützt werden. Um dies zu gewährleisten und um die im Abschnitt 3.6 beschriebene Zugriffskontrolle wirksam durchsetzen können, wurden die Aufrufe von öffentlichen Funktionen über RPCs umgeleitet.

Der Aufruf einer ausgelagerten Funktion erfolgt mithilfe der im Abschnitt 3.4 vorgestellten RPC-Architektur. Sie ist notwendig, um auch bei Einhaltung der Semantik eines Funktionsaufrufes eine wechselseitige RPC-Kommunikation zwischen zwei Tasks zu ermöglichen, ohne dass dies zu einem Deadlock des Systems führt. Der Zugriff auf gemeinsame Datenbereiche wird mittels des im Abschnitt 3.3.1 einführten MODLD-Areas realisiert. Beide Konzepte ermöglichen eine transparente Ausgelagerung von Modulen, so dass umfangreiche Änderungen am Programmcode unnötig sind und in der Kernel-Konfiguration entschieden werden kann, ob ein Modul in des Adressraum des L^{4}LINUX-Servers geladen oder ausgelagert werden soll.

Um das Designziel der Nutzertransparenz zu untermauern, wurden im Kapitel 4 Nutzerprogramme vorgestellt, die beim Erstellen eines Moduls die notwendigen Erweiterungen automatisch generieren und in die Objektdatei des Moduls einbinden. Unter Einhaltung der im Abschnitt 4.3 beschriebenen Richtlinien zum Erstellen eines ausgelagerten Moduls und zur Integration der Architektur in die L4-Umgebung der TU-Dresden, ist es möglich, ein LKM zu implementieren, das je nach Bedarf ausgelagert oder in den Adressraum des L^{4}LINUX-Servers geladen werden kann. Hierzu ist lediglich ein erneutes Übersetzen des Programmcodes notwendig. Die wenigen erforderlichen Änderungen am Programmcode des Moduls beschränken sich auf das Markieren von öffentlichen Funktionen und das Kapseln der Funktionen zum Allokieren von privaten Speicherbereichen. Für beides wurden dem Nutzer Makros zur Verfügung gestellt.

Im Kapitel 5 wurde die Leistung des Systems vermessen und eine Bewertung der ermittelten Werte vorgenommen. Wie sich gezeigt hat, steigt die Anzahl der benötigten Taktzyklen insbesondere bei initialen Aufrufen von ausgelagerten Funktionen stark an. Allerdings ist auch der Aufruf eines RPCs im Weiteren sehr aufwendig, so dass die Gesamtleistung des Systems entscheidend von der Anzahl der benötigen RPCs abhängt. Das aktuelle Konzept wurde nicht unbedingt hinsichtlich einer optimalen Ausnutzung der Systemleistung entwickelt, so dass es ein großes Potential an möglichen Verbesserungen enthält. Hierauf wird noch einmal genauer im folgenden Abschnitt eingegangen.

6.2 Ausblick

Die in dieser Arbeit vorgestellte Architektur bietet die Voraussetzung zur Auslagerung von LKMs und der damit verbundenen Bereitstellung von privaten, sicheren Speicherbereichen. Die Entwicklung ist jedoch bei weitem noch nicht abgeschlossen. Bereits bei der Implementierung beziehungsweise bei den Messungen haben sich Probleme gezeigt, die zur Verbesserung und zum produktiven Einsatz des Konzeptes behoben werden müssen. Für einen Großteil der Probleme sollen in diesem Abschnitt noch einmal mögliche Lösungen vorgeschlagen werden.

Abschließend werden in diesem Abschnitt mögliche Einsatzgebiete des Konzeptes vorgestellt. Hierbei wurden insbesondere die guten als auch die schlechten Leistungen des Konzeptes berücksichtigt.

6.2.1 Optimierungsmöglichkeiten der Implementierung

Ein wesentliches Problem des Konzeptes ist der Mehraufwand beim Adressraumwechsel. Wie sich gezeigt hat, ist der Zugriff auf öffentliche Variablen ausreichend performant, so dass der Fokus für Verbesserungen auf den Aufruf von Funktionen gerichtet werden sollte.

Optimierung des Funktionsaufrufes

Die im Abschnitt 3.4 vorgestellte RPC-Architektur benötigt mindestens zwei Adressraumwechsel zum Aufruf einer ausgelagerten Funktion. Jeder RPC über Adressraumgrenzen hinaus benötigt mindestens 10.000 Taktzyklen, so dass mit der Reduzierung auf nur einen Wechsel bereits eine nicht unwesentliche Steigerung der Leistung erzielt werden kann.

Die RPC-Architektur wurde so implementiert, dass zunächst der RPC-Handler nach einem RPC-Proxy befragt wird. Mit der Einführung eines zusätzlichen RPCs ist es möglich, den Aufruf des RPC-Proxies zu vermeiden. Hierzu behandelt der RPC-Handler den gesamten Aufruf. Um nicht zu blockieren, wird zur Abarbeitung der eigentlichen Anforderungen ein *Worker-Thread*, ähnlich dem RPC-Proxy, generiert. Dieser erhält jedoch direkt den Request und beginnt mit dessen Abarbeitung ohne zuvor in den Adressraum des RPC-Clients zurückzuwechseln. Um das Ergebnis zurück an den Client zu senden, wird ein zusätzlicher RPC eingeführt, der vom Worker-Thread an den RPC-Handler gesendet wird. Dieser leitet das Ergebnis an den eigentlichen RPC-Client weiter. Damit ist es möglich, den zweiten Adressraumwechsel einzusparen.

Darüber hinaus kann mit der Realisierung eines Proxy-Pools zusätzlicher Aufwand gespart werden. In der aktuellen Implementierung wird für jeden Request ein neuer Proxy erzeugt, welcher später vom RPC-Watchdog vollständig entfernt wird. Mit einer permanenten Bereitstellung einer Mindestmenge an Proxies könnte der Aufwand zum Erzeugen und Entfernen nahezu vollständig eingespart werden. Die Anzahl an verschachtelten Aufrufen kann als relativ gering angenommen werden, so dass bereits eine kleine Anzahl an Proxies für die meisten Anforderungen ausreichend sein sollte. Für Modul-Tasks bedeutet dies sogar die Einsparung eines Adressraumwechsels, da die im Abschnitt 5.2.5 erläuterte, aufwendige Allokation des Proxy-Stacks unnötig wird.

Ein weiterer Adressraumwechsel ist für die Durchsetzung der Zugriffskontrolle notwendig. Aus Sicherheitsgründen wird die ACL im Adressraum des Modul-Servers abgespeichert. Da die Kontrolle jedoch vom RPC-Handler der Modul-Task oder des L^{4}LINUX-Servers durchgeführt wird, ist ein zusätzlicher Adressraumwechsel notwendig. Eine Möglichkeit zur Lösung des Problems wäre das Einblenden der ACL in den Adressraum der jeweiligen Tasks. Eine Speicherseite könnte nur lesend eingeblendet werden, so dass sie nachträglich nicht mehr verändert werden kann. Da jedoch nur ganze Speicherseiten eingeblendet werden können, müsste die Speicherverwaltung überarbeitet werden, so dass beim Einblenden nicht zu viele Informationen offengelegt werden.

Reduzierung der RPCs

Auch wenn mit den im vorangegangenen Abschnitt erläuterten Verbesserungen die Anzahl der Adressraumwechsel auf eins reduziert werden könnte, so bedeutet dies immer noch ein Mehraufwand von ungefähr 10.000 Taktzyklen zur Ausführung einer fremden Funktion. Viele Funktionen beinhalten jedoch nur wenige Zeilen Programmcode, so dass der Aufwand mitunter nicht lohnenswert ist.

Um weitere unnötige Adressraumwechsel einsparen zu können, ist es denkbar, die Funktionen lokal in den Adressraum der Task zu integrieren. Da dies insbesondere für Modul-Tasks sinnvoll wäre, könnte eine Bibliothek solcher Funktionen in den Modul-Server integriert werden, der diese je nach Bedarf in den Adressraum einblendet. Das Auflösen der Referenzen könnten über den Modul-Loader erfolgen, hierzu müsste der Modul-Server lediglich die öffentlichen Bibliotheksfunktionen in die Symboltabelle des L^{4}LINUX-Servers einfügen.

6.2.2 Sicherheitskritische Schwachstellen

Das Hauptaugenmerk bei der Erstellung des Konzeptes war die Erhöhung der Sicherheit für Funktionen des Systemkerns. Es wurde gezeigt, dass Module in eigene Adressräume ausgelagert werden können, so dass sie vor Zugriffen einer anderen Task geschützt sind. Dieser Abschnitt beschreibt zwei sicherheitskritische Probleme, die bisher offen geblieben sind und vor einem produktiven Einsatz des Konzeptes behoben werden sollten.

Laden von Modulen

Die bisherige Implementierung sieht vor, dass ein Modul mithilfe des Modul-Loaders des L^{4}LINUX-Servers geladen wird. Hierzu ist es notwendig, dass der Loader den Programmcode des Moduls lädt und in den Adressraum des L^{4}LINUX-Servers überträgt. Anschließend wird er dem Modul-Server

übergeben. Dies bedeutet, dass das Modul während des gesamten Ladevorgangs vollkommen ungeschützt ist.

Laden bei der Initialisierung des Systems

Reduzieren könnte man das Problem, indem man das Modul vor dem Starten der Nutzerprozesse lädt. Je zeitiger das Laden erfolgt, desto geringer wäre die Gefahr einer möglichen Manipulation des Moduls. Da jedoch ein Gerätetreiber für den Zugriff auf das Medium notwendig ist, müsste das System weitestgehend initialisiert oder eine externe Quelle herangezogen werden.

Nutzerprozesse werden immer durch den init-Prozess des Systems gestartet. Dieser ist zusätzlich für deren Initialisierung verantwortlich. Verlegt man das Laden der Module in die frühe Initialisierungsphase, kann zumindest ausgeschlossen werden, dass ein Nutzer die Integrität eines Moduls verändert. Ein weitere Möglichkeit zur Lösung des Problems wäre die Nutzung des Boot-Loaders. Dieser ist für das Laden der initialen Systemkomponenten notwendig. Die L^{4}LINUX-Umgebung verwendet hierzu eine spezielle Version des **grub**. Es wäre durchaus möglich, die Module vom **grub** laden zu lassen und zur Laufzeit des L^{4}LINUX-Servers zum Modul-Server zu übertragen. In diesem Fall wären die Module zu keinem Zeitpunkt im Adressraum des L^{4}Linux-Servers.

Laden mittels externer L4-Tasks

Das Laden der Module bei der Initialisierung des Systems beinhaltet das Problem, dass die Konfiguration zur Laufzeit nicht mehr verändert werden kann. Die Schnittstellen des Modul-Servers sind nicht an den L^{4}LINUX-Server gebunden, dies bedeutet, dass die Funktionen zum Laden eines Moduls von jeder beliebigen Task aufgerufen werden kann.

Dies ermöglicht die Verwendung von externen Modul-Loadern, die als eigene Task des L4-Kerns laufen und damit vor Zugriffen durch den L^{4}LINUX-Server geschützt sind. Eine solche Task müsste die Funktionalität des Modul-Loaders des LINUX-Systems enthalten und zusätzlich das Laden des Moduls von einem Medium unterstützen. Hierzu kann es wiederum andere vertrauenswürdige L4-Tasks hinzuziehen.

Sicherung der ACL

Das bisherige Konzept der ACL sieht vor, dass die einzelnen Tasks anhand des Modulnamens identifiziert werden. Es kann jedoch nicht sichergestellt werden, dass das geladene Module wirklich dem Objekt aus der ACL entspricht. Ein Angreifer könnte dies ausnutzen, um Funktionen eines geschützten Moduls aufzurufen, in dem er ein ausgelagertes Modul erstellt und es mit einem autorisierten Namen versieht. Anschließend kann er die durch die ACL geschützten Funktionen oder Datenbereiche verwenden.

Zur Lösung des Problems würden sich kryptographische Prüfsummen anbieten. Sogenannte *One-Way-Hash*-Funktionen berechnen für einen Originaltext einen eindeutigen Hashwert, mit einer festen Länge, zum Beispiel 128-bit für MD5 [23]. Verwendet man zusätzlich zum Namen des Moduls einen solchen Hash-Wert, den man aus dem Objektcode des Moduls berechnet, könnte eine eindeutige Identifikation sichergestellt werden. One-Way-Hash-Funktionen sind so gestaltet, dass es nahezu unmöglich ist, einen Originaltext zu erstellen, der einen bestimmten Hash-Wert aufweist. Zur Realisierung des Konzeptes müssten beim Erstellen der Module die notwendigen Hashwerte berechnet und in die ACLs eingefügt werden.

6.2.3 Mögliche Anwendungsgebiete

Abschließend erfolgt eine kurze Beschreibung von möglichen Einsatzgebieten für das hier vorgestellte Konzept.

Module mit einem hohen Sicherheitsbedarf

Das hier vorgestellte Konzept zielt insbesondere auf Module mit einem hohen Bedarf an Sicherheit, deren Interaktion mit dem übrigen System möglichst gering ausfällt. Ein mögliches Anwendungsbeispiel wäre die Auslagerung des IPSec-Moduls oder des CryptFS, so dass lediglich die Funktionen zum Ver- und Entschlüsseln sowie des Schlüsselmanagements öffentlich zugänglich sind. Darüber hinaus könnte bei Einführung eines reinen Verschlüsselungsmodul, die Kommunikation noch weiter eingeschränkt werden. So wäre es nicht notwendig, dass Funktionen des L^{4}LINUX-Servers direkt die Ver- beziehungsweise Entschlüsselungsfunktion aufrufen können, da diese nur vom IPSec-Modul verwendet werden.

Insbesondere bei Modulen zur Verschlüsselung von Daten wäre der Einsatz des hier vorgestellten Konzeptes sinnvoll, da der Aufruf für die Kommunikation nur einen geringen Anteil der Rechenzeit in Anspruch nehmen würde. Zum einen lässt sich der Umfang der Kommunikation leicht reduzieren beziehungsweise ist teilweise schon sehr gering und zum anderen verbrauchen die Verschlüsselungsalgorithmen selbst sehr viel Rechenzeit.

Auslagerung von unbekannten Modulen

Die Einführung des LKM-Konzeptes wurde unter anderem notwendig, weil in immer kürzerer Zeit immer mehr Gerätetreiber notwendig beziehungsweise erstellt wurden. Einige Treiber wurden nie in den offiziellen Verzeichnisbaum aufgenommen oder es dauert sehr lange, bis sie letztendlich integriert wurden. Oftmals ist es notwendig, neue Treiber oder Module zu verwenden, die nicht ausreichend getestet wurden oder deren Ursprung teilweise unbekannt ist. Selbst wenn man den Programmcode erhält, kann nicht mit Sicherheit ausgeschlossen werden, dass das Modul genau das tut, was man erwarten würde.

Lagert man derartige Module in einen fremden Adressraum aus, kann zumindest sichergestellt werden, dass sie nur über das öffentliche Interface kommunizieren. Zugriffe auf den Datenbereich können im Pager des Modul-Servers überprüft werden. Da nur wenige Änderungen am Programmcode des Moduls notwendig sind, um es in einen fremden Adressraum auszulagern, kann dies auch zunächst nur zum Testen des Moduls durchgeführt werden. Erweist sich das Modul als sicher oder ist die Gesamtperformance akzeptabel, kann über eine weitere Verwendung entsprechend entschieden werden.

Test der Modul-Implementierung

Insbesondere bei der Implementierung von LKMs ist eine unsaubere Speicherverwaltung ein massives Problem. Zugriffe auf uninitialisierte Zeiger oder das Überschreiten von Puffergrenzen bleiben unter Umständen lange Zeit verborgen, wenn sie auf einen gültigen Bereich des Systemkerns verweisen.

Um Zugriffsverletzungen leicht erkennen zu können, wäre es denkbar, ein neues Modul zunächst in einen fremden Adressraum ablaufen zu lassen. Treten hierbei Zugriffsverletzungen auf, können diese schnell erkannt werden. Dabei wird das Entwicklungs- oder Testsystem nicht beeinträchtigt. Das Modul kann erneut getestet werden, ohne dass das System neu gestartet werden muss.

Anhang A

Literaturverzeichnis

[1] Tzi-cker Chiueh, G. Venkitachalam und P. Pradham. *Intra-Address Space Protection using Segmentation Hardware*. http://www.ecsl.cs.sunysb.edu/palladium.html.

[2] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg und J. Wolter. *The performance of µ–kernel-based systems*. In 16th ACM Symposium on Operating System Principles (SOPS), Seite 66-77, Okt. 1997.

[3] M. Aron, Y. Park, T. Jaeger, J. Liedtke, K. Elphinstone und L. Deller. *The SawMill Framework for Virtual Memory Diversity*. Sixth Australasian Computer Systems Architecture Conference (ACSAC2001), Bond University, Gold Coast, Queensland, Jan.-Feb. 2001

[4] D. P. Bovet; M. Cesati: *„Understanding the LINUX-Kernel“*, O'Reilly, Sebastopol 2001.

[5] D. E. Knuth: *„The Art of Computer Programming“*, Band 1, Addison-Wesley Longman, Amsterdam 2002

[6] M. Borriss, M. Hohmuth, J. Wolter und H. Härtig. *Portierung von LINUX auf den µ–Kern L4*. Int. wiss. Kolloquium Illmenau, Sep. 1997

[7] M. Beck; H. Böhme; M. Dziadzka; U. Kunitz; R. Magnus; C. Schröter; D. Verworrner*: „LINUX Kernel-Programmierung, Algorithmen und Strukturen der Version 2.2“*, 5. Auflage, Addison-Wesley, Bonn 1999.

[8] Fiasco-Hompage: http://os.inf.tu-dresden.de/fiasco/

[9] FreeS/WAN-Homepage: http://www.freeswan.org

[10] E. Zadok, I. Badulescu und Alex Shender. *„Cryptfs: A Stackable Vnode Level Encryption File System”***,** Computer Science Department, Columbia University http://www.cs.columbia.edu/~ezk/research/cryptfs/index.html, Columbia 1998.

[11] TU-Dresden, Institut für Systemarchitektur, Professur Betriebssysteme: *„µSina - Mikrokernbasierte Sichere Inter-Netzwerk-Architektur“*, 2002 URL: http://os.inf.tu-dresden.de/mikrosina/

[12] Bundesamt für Sicherheit in der Informationstechnik: *Sichere Inter-Netzwerk-Architektur*, 2002. URL: http://www.bsi.bund.de/fachthem/sina.

[13] secunet Security Networks AG: SINA – Hochsichere Inter-Netzwerk-Architektur für behördliche und private Anwender, 2002. Fact Sheet, URL: http://www.secunet.com.

[14] J. Liedtke: *„Lava Nucleus (LN) Reference Manual, Version 2.2”*, IBM T. J. Watson Research Center, 1998

[15] M. Borriss, H. Härtig: **„***Design and Implementation of a Real-Time ATM-Based Protocol Server”*, Proc. of the 19th IEEE Real-Time Systems Symposium (RTSS), Spanien 1998

[16] Ch. Helmuth: *„Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur“*, Diplomarbeit an der TU-Dresden, 2001.

[17] Mixter: *„Tarnkappen für Einbrecher“*, c't 4/2002, Heise Verlag, Seite 196.

[18] K. Rannenberg, A. Pfitzmann, G. Müller: *„Sicherheit, insbesondere mehrseitige IT-Sicherheit“*, in G. Müller, K.-H. Stapf: *„Mehrseitige Sicherheit in der Kommunikationstechnik“*, Band 2, Seite 19-27, Addison-Wesley, Bonn 1998.

[19] M. Sobirey: *„Datenschutzorientiertes Intrusion Detection – Grundlagen, Realisierung, Normung“*, Vieweg-Verlag, Braunschweig 1999.

[20] L^{4}LINUX-Homepage: http://os.inf.tu-dresden.de/L4/LinuxOnL4

[21] Herold Helmut: *„Linux – Unix Systemprogrammierung“*, 2. Auflage, Seite 1086, Addison-Wesley, Bonn 1999.

[22] H. Härtig: *„Security Architectures Revisited“*, 10th ACM SIGOPS European Workshop, September 2002, URL: http://os.inf.tu-dresden.de/papers_ps/secarch.pdf.

[23] MD5-Beschreibung: http://www.faqs.org/rfcs/rfc1321.html.

[24] GNU General Public License: http://www.gnu.org/copyleft/gpl.html.

[25] J. Wilcox: *„Microsoft server share jumps in 2001”* URL: http://news.com.com/2100-1001-959049.html.

[26] Andrea Arcangeli: *„[discuss] Kernel calling convention [proposal]”*, www.x86-64.org - Diskussionsforum, September 2000, URL: http://www.x86-64.org/lists/discuss/msg01800.html.