Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Thomas Setzer

PromethOS v2.0 - Memory Resource Control in the Linux 2.4 Kernel

Diploma Thesis DA-2002.31
September 2002 to March 2003

Supervisor: Lukas Ruf
Co-Supervisor: Matthias Bossardt
Professor: Bernhard Plattner

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Zusammenfassung

Aktive Netzwerkknoten ermöglichen es, Netzwerke individuell zu programmieren und diese somit auf flexible Weise um beliebige Funktionalitäten erweitern zu können. Am Institut für Technische Informatik und Kommunikationsnetze (TIK) der ETH Zürich (ETHZ) wurde ein Node Operating System (NodeOS), PromethOS, unter Linux entwickelt. PromethOS erweitert die Netfilter-Paketbearbeitungsfunktionalität, indem es die Installation von Code und dessen Ausführung im Linux Kernel zur Laufzeit des NodeOS gestattet. PromethOS stellt ein PromethOS Execution Environment zur Verfügung, in dessen Instanzen spezielle, den Service Code enthaltende Kernel Module - sogenannte Plugins - installiert und zur Ausführung gebracht werden können.

Unter PromethOS war es jedoch nicht möglich, den Speicherzugriff von PromethOS Plugins zu kontrollieren, da das gegenwärtige Memory-Management-Subsystem von Linux, welches unter PromethOS Verwendung fand, eine Speicherzugriffskontrolle im Kernel nicht unterstützt. Ein Plugin konnte somit beliebig in den Speicher anderer Plugins oder des Kernels hineinschreiben oder Daten auslesen, was negative Konsequenzen für die Systemeigenschaften Sicherheit und Stabilität mit sich brachte.

Im Rahmen dieser Diplomarbeit wurde nun das PromethOS Framework dahingehend erweitert, dass es die Installation und Ausführung von Plugins erlaubt, ohne die Integrität des NodeOS durch nicht autorisierte Speicherzugriffe der Plugins zu gefährden.

Hierfür wurde zunächst das Linux Memory Management um eine Speicherzugriffs- und -verbrauchskontrolle im Kernel Space erweitert. Auf dieser Erweiterung aufbauend wurde eine Memory Resource Control Architektur entwickelt, welche diese erweiterte Funktionalität nutzt und damit den Plugins Execution Environments zur Verfüging stellen kann, welche in exakt eingrenzbaren Speicherbereichen, sogenannten Protection Domains, instanziert werden. Plugins können nun ausgeführt werden ohne das die Funktion des NodeOS beeinträchtigt werden kann.

Interfaces wurden definiert und implementiert, die einen funktionierenden Austausch von Daten- und Kontrollinformationen sowohl zwischen dem Netfilter- und dem Memory Resource Control Framework, also auch zwischen dem Memory Resource Control Framework und den Plugins ermöglichen. Hierfür wurden Teile der bereits entwickelte Funktionalitäten und Schnittstellen des bestehenden PromethOS genutzt. Das erstellte System kann somit als NodeOS genutzt werden.

Abstract

Active Network Nodes (ANN) provide the opportunity to individually program networks, thus allowing to extend their functionality in a flexible manner. Therewith, a Service Provider is able to process single or aggregated flows according to his needs. For example, he can install encryption and decryption modules in the network to overcome plain text transfer of his undisclosed data.

The Computer Engineering and Networks Laboratory (TIK) at the Swiss Federal Institute of Technology Zurich (ETH) developed a Linux based Node Operating System (NodeOS) named PromethOS, which provides the ability to install and execute code in the Linux Kernel. Code gets installed and processed in PromethOS Execution Environments (PromethOS-EE) by installing special Kernel Modules, so called PromethOS Plugins.

The NodeOS has to take care such that installed Plugin Service Code cannot affect or damage the NodeOS' functionality. For this purpose, a control mechanism of a plugin's resource access and usage has to be implemented to detect and prevent such unauthorized behavior. PromethOS was unable to control memory acces and memory usage of plugins, thus the underlying Linux Memory Management does not support Memory Resource Control in the Kernel Space. A plugin could address the whole Kernel Memory Space including both privat data of other plugins and system critical memory areas. This excessively downgraded system security and stability.

During this diploma thesis the Linux Memory Management System has been extended. It now provides Memory Resource Control in Kernel Space by executing PromethOS Plugins as the image of a specially created and adapted Kernel Thread. Such a thread operates solely on a restricted memory range. Furthermore a Thread Management System has been developed which creates, schedules and controls these threads.

Based on these new features of the Linux Memory and Thread Management a new PromethOS Framework has been created which uses these new features to provide Execution Environments and corresponding Protection Domains in the Linux Kernel Space. Plugin code can now get executed without the risk of harming the system. Interfaces have been defined to enable communication between Plugins and the PromethOS Framework.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Aktive Netzwerke

Heutige Datennetzwerke, zu denen auch und vor allem das im Folgenden betrachtete Internet zu zählen ist, transportieren Bits in der Regel auf passive Weise von einem Endsystem zum anderen. Die Eigenschaft “passiv” ist hier in dem Sinne zu vestehen, dass die zu übertragenden Daten im Netz selbst, also von den passierten Netzwerkknoten, nicht modifiziert werden. Idealerweise werden Daten in solchen Netzen ganz und gar transparent transferiert; das Netzwerk ist somit indifferent gegenüber den zu übertragenden Inhalten, d.h. den IP-Paketinhalten. Paketbearbeitung findet hier nur sehr eingeschränkt, meist in Form einfachen Header-Processings, statt.[1]

Aktive Netzwerke (Active Networks (AN)) unterscheiden sich nun dahingehend von oben genannten passiven Netzen, dass Netzwerkknoten, über den Transport der Pakete hinaus, auf die zu transferierenden Benutzerdaten zugreifen und diese manipulieren. Der Namensgebung entsprechend wird also “aktiv” auf die Paketinhalte zugegriffen.[TeWe96]

Aktive Netze ermöglichen es somit, Netzwerke individuell zu programmieren und diese somit auf flexible Weise um beliebige Funktionalitäten erweitern zu können. Ein Service-Provider wird dadurch in die Lage versetzt, einzelne Flows oder aggregierte Flows speziell nach seinen Bedürfnissen zu verarbeiten. Beispielsweise kann ein benutzerdefiniertes Ver- und Entschlüsselungs-Plugin im Netz installiert werden, um zu erreichen, dass bestimmte Datenströme verschlüsselt transferiert werden.

1.2 PromethOS - ETH NodeOS

Am Institut für Technische Informatik und Kommunikationsnetze (TIK)[2] der ETH Zürich (ETHZ)[3] wurde ein Betriebssystem für aktive Netzwerkknoten[4], PromethOS [iwan02], unter Linux entwickelt. Abbildung 1.1 stellt das PromethOS-Framework schematisch dar.

PromethOS gestattet die Installation von Code und dessen Ausführung im Linux Kernel zur Laufzeit des NodeOS. Weiter stellt PromethOS ein Execution Environment (PromethOS-EE) zur Verfügung, in dessen Instanzen spezielle, den Service Code enthaltende Kernel Module - sogenannte Plugins - installiert und zur Ausführung gebracht werden können.

Es folgt nun eine Beschreibung des PromethOS-Frameworks in einem für das Verständnis der Motivation dieser Arbeit erforderlichen Maße.

[1] Beispielsweise zur Ermittlung des Destination-Hosts
[2] http://www.tik.ee.ethz.ch
[3] http://www.ethz.ch
[4] Im Folgenden (Node Operating System (NodeOS) genannt

Eine ausführliche Betrachtung desselben erfolgt in Kapitel 2.1.

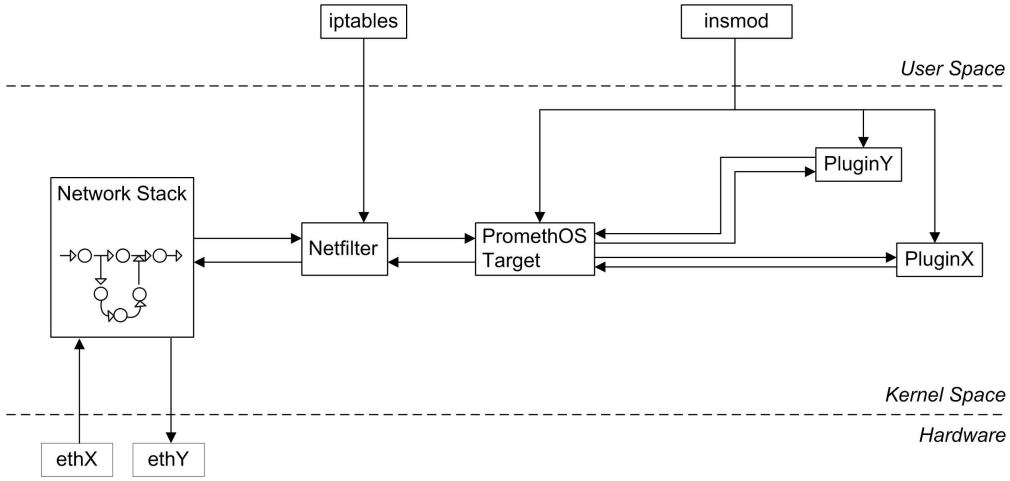

Abbildung 1.1: PromethOS v1.0 Architektur

PromethOS basiert auf den durch das Netfilter-Framework [Rus03] angebotenen Paketfilterfunktionalitäten. [Guin01]

Netfilter, entwickelt von Rusty Russel und seinem Netfilter Core Team, ist ein Framework zur Paketbehandlung, welches in zwei Blöcke aufgeteilt werden kann [WPRMB01]:

- Die sogenannten *Netfilter Hooks*[5] bieten eine komfortable Möglichkeit, um die zu verarbeitenden (oder bearbeiteten) IP-Pakete an verschiedenen Stellen auf ihrem Weg durch den Linux-Kern abzufangen und zu manipulieren (oder wieder zu reinjizieren). An jedem dieser Punkte wird nun Netfilter mit den Parametern *Paket* und *Hook-Nummer* aufgerufen. Teile des Kernels können sich an den verschiedenen Hooks für jedes Protokoll registrieren. Wenn ein Paket also an Netfilter weitergereicht wird, wird überprüft, ob irgend jemand für diesen Hook registriert ist; falls dies der Fall ist, bekommt jeder von ihnen[6] der Reihe nach die Chance, das Paket zu untersuchen und es möglicherweise zu verändern, das Paket zu verwerfen oder es durchzulassen.
- Hierauf aufbauend implementiert das *iptables*-Modul Regellisten für die Filterung hereinkommender, weitergeleiteter und ausgehender IP-Pakete.

PromethOS erweitert nun dieses Netfilter-Framework um die Möglichkeit, Plugins dynamisch, also zur Laufzeit, zu installieren und auszuführen. Dies ist notwendig, da das Netfilter Framework selbst keine dynamische Installation von Plugins unterstützt; Plugins müssen unter Netfilter zur Compilierzeit schon vorhanden sein, um statisch in das Framework eingebunden zu werden. Ermöglicht wird diese erweiterte Funktionalität dadurch, dass das *PromethOS Target*[7] die Verwaltung von - und Kommunikation mit - zur Laufzeit geladenen Plugins übernimmt. Das PromethOS Target leitet ankommende Pakete, die einem Filter entsprechen, an die entsprechende Plugininstanz weiter.

1.3 Problembeschreibung und Motivation

Unter PromethOS werden Plugins aus Performancegründen[8] nicht im User- sondern im Kernelspace instanziert. PromethOS nutzt hierfür die Möglichkeit der Kernelmodularisierung

[5] Hooks sind wohldefi nierte Punkte auf dem Weg eines Pakets durch den Protokoll-Stack

[6] Sogenannte Netfi lter-Targets

[7] Aus der Sicht des Netfi lter-Framework ein “normales”, statisch eingebundenes Target

[8] Teure Kontext-Switches beim Umschalten zwischen Kernel- und Usermode werden so vermieden. Auch entfällt das zweimaliges Kopieren der Pakete in - und aus dem Userspace sowie die Notwendigkeit der Verwendung von Systemcalls.

(c.f. [BoCe00], Appendix B), welche von Linux ab der Kernelversion[9] 2 unterstützt wird. PromethOS-Plugins sind demnach spezielle Kernelmodule, die zur Laufzeit in den Kern des Betriebssystems hinzugelinkt werden.

Aktuell ist es in PromethOS jedoch nicht möglich, den Speicherzugriff von PromethOS Plugins zu kontrollieren, da das gegenwärtige Memory-Management-Subsystem von Linux, welches ja unter PromethOS Verwendung findet, eine Speicherzugriffskontrole im Kernel nicht unterstützt.[10] Ein Plugin kann somit beliebig in den Speicher anderer Plugins oder des Kernels hineinschreiben oder Daten auslesen, was gravierende Auswirken auf nachfolgend genannten Systemeigenschaften hat:

- *Sicherheit*
  Sensible Daten eines Plugins - beispielsweise unverschlüsselte Passwörter oder Keys - können von anderen Plugins zum einen ausgelesen (passiver Angriff), zum anderen sogar manipuliert werden (aktiver Angriff); Systemsicherheit ist somit nicht, oder nur sehr eingeschränkt vorhanden.

- *Stabilität*
  Es besteht die Möglichkeit, dass Plugins - sei es aufgrund fehlerhaften Programmcodes oder aber auch mit böswilliger Absicht - in Speicherbereiche anderer Plugin hineinschreiben und diese zum Absturz bringen. Im schlimmsten Fall kann eine Manipulation kritischer Kernel-Speicherbereiche sogar zu einem “Kernel Panic”[11] führen, wodurch das komplette Betriebssystem unbrauchbar wird und neu gebootet werden muss.

Plugins sollten jedoch auf einem ANN installiert werden können, ohne dass die Funktion des ANNs durch nicht-autorisierte Speicherzugriffe in einer wie vorangehend beschriebenen Weise beeinträchtigt werden kann. Damit aber PromethOS eine solche Kontrolle der Zugriffe auf die Ressource Speicher zur Verfügung stellen kann, muss das Linux-Memory-Management um die Möglichkeit einer Speicherzugriffskontrolle im Kernelspace erweitert werden.

1.4 Ziel dieser Diplomarbeit

Im Rahmen dieser Diplomarbeit soll das Memory Management von Linux so erweitert werden, dass PromethOS im Kernelspace EEs zur Verfügung stellen kann, welche jeweils in einem eigenen, exakt eingrenzbaren Speicherbereich (Protection Domain) instanziert werden. Dadurch soll gezeigt werden, dass der Speicherzugriff und -verbrauch von Kernel-Modulen kontrolliert werden kann. Das PromethOS Framework ist so zu erweitern, dass die entwickelte Linux Memory Management Funktionalität auf dem ANN genutzt werden kann; um also die Ausführung von fremdem Code zu ermöglichen, ohne die Integrität des NodeOS zu gefährden.

Das erweiterte Memory-Management soll möglichst nahtlos in den Linux Kernel eingebunden werden können. Standard-Linux-Programme, wie beispielsweise `insmod` für das Laden der Plugin-Kernel-Module, sollen gegebenenfalls erweitert werden. Auch soll der implementierte Code so weit als möglich portabel gehalten werden.

Zusätzlich müssen für den Einsatz von PromethOS als NodeOS Interfaces definiert werden, welche die Kommunikation zwischen den Plugins und dem PromethOS-Framework für Kontrollinformationen gleich wie auch für Datenaustausch ermöglichen.

Stabilität und Effizienz des Systems sollen durch Messungen des Erreichten ermittelt und dokumentiert werden.

[9] Kernelversionen sowie deren Entwicklungsgeschichte fi nden Sie unter http://www.kernel.org
[10] Die Ursache hierfür werden in den Sektionen 2.3, 3.1 und 3.2 hergeleitet
[11] System-Crash

1.5 Gliederung dieses Berichts

Der vorliegende Bericht ist in 4 weitere Kapitel unterteilt.

Zunächst wird in *Kapitel 2* das bestehende PromethOS v1.0 Framework vorgestellt. Im Anschluss hieran wird auf diejenigen, im Kontext der Aufgabenstellung relevanten Aspekte der Linux 2.4 Architektur[12] genauer eingegangen, deren Verständnis für die Erarbeitung eines Lösungsansatzes, sowie für die Generierung der Zielarchitektur unerlässlich sind. Insbesondere wird hierbei die Linux-Speicherverwaltung detaillierter betrachtet.

In *Kapitel 3* wird zu Beginn mit den aus Kapitel 2 gewonnenen Erkenntnissen das in der Linux-Memory-Management-Architektur begründete Problem genau lokalisiert und spezifiziert. Hieraus wird zunächst ein Lösungsansatz abgeleitet, welcher dann in einer Memory-Resource-Control-Architektur seine Umsetzung findet. Die Darstellung dieser Architektur, ihrer Komponenten sowie die Interaktion mit dem Netfilter-Framework bildet den Hauptteil dieser Ausarbeitung.

Kapitel 4 befasst sich mit Evaluation und Test der Implementation.

In *Kapitel 5* werden zum einen Schlussfolgerungen aus den während der Bearbeitung des Themas - bzw. bei Entwicklung und Umsetzung der Architektur - gewonnenen Erkenntnissen gezogen, zum anderen zeigt dieses Kapitel Ausblicke in zukünftige Weiterentwicklungen.

1.6 Dankesworte

An dieser Stelle bedanke ich mich bei allen, die mich bei meiner Diplomarbeit am Institut für Technische Informatik und Kommunikationsnetze der ETH Zürich und bei der Erstellung dieses Abschlussberichts unterstützt haben. Die sechs Monate meiner Diplomarbeit zähle ich zu den interessantesten und lehrreichsten Monaten meines bisherigen Daseins. Die Komplexität der Aufgabenstellung, die vielen Nüsse die es hierbei zu knacken galt, machten diese Arbeit faszinierend und nervenaufreibend zugleich.

Diesbezüglich möchte ich in mehrerer Hinsicht ein großes Dankeschön an Lukas Ruf richten. Dass ich während der Dauer der Arbeit nie den Mut, die Freude und die Orientierung verloren habe, lag sicherlich zu einem großen Teil auch an ihm und der Art und Weise seiner Betreuung. Es war immer sehr hilfreich und hat Spaß gemacht, zusammen Ideen, Konzepte oder das weitere Vorgehen zu besprechen. Auch konnte ich ihn bei den vielen Fragen, die im Laufe der Diplomarbeit auftraten, zu jeder Tages - und fast zu jeder Nachtzeit - kontaktieren und musste meist nicht lange auf eine weiterhelfende Antwort warten.

Herrn Prof. Dr. Berhard Plattner vom Institut für Technische Informatik und Kommunikationsnetze der ETH Zürich sowie Herrn Prof. Dr. H. Schmeck vom Institut für Angewandte Informatik und Formale Beschreibungsverfahren der Universität Karlsruhe möchte ich, neben Lukas Ruf, für das in mich gesetzte Vertrauen danken.

Meiner Freundin Sylvia Prenzel danke ich dafür, dass Sie sich die Zeit genommen hat, dieses technische Schriftstück Probe- und Korrekturzulesen.

Vor allem aber bedanke ich mich bei meinen Eltern, die all die Jahre immer an mich geglaubt und mich unterstützt haben. Sie haben mir mein Studium, welches in dieser Arbeit nun seinen Abschluss findet, ermöglicht.

Vielen Dank!

[12] Auf welchem das PromethOS-System basiert

Kapitel 2

Analyse der bestehenden PromethOS-Architektur

PromethOS wurde unter Linux entwickelt. Ein Standard-Linux-Kernel der Version 2.4 wurde hierfür entsprechend den Anforderungen an die gewünschte Funktionalität des System modifiziert und erweitert. Dementsprechend “erbt” PromethOS sämtliche architektonischen Grundkonzepte, die nicht explizit abgeändert wurden, von dem ihm zugrunde liegenden Linux-Kernel.

Aus diesem Grunde werden - neben der Analyse des eigentlichen PromethOS-Frameworks in *Sektion 2.1* - ebenfalls die im Kontext der Aufgabenstellung relevanten Subsysteme des Linuxkerns in *Sektion 2.2* näher beleuchtet. Die Integration eines separaten Unterkapitels diesbezüglich ist sinnvoll, da aus den in *Sektion 2.2* genannten Gründen auf keine Literatur verwiesen werden kann, die sämtliche Subsysteme in einer notwenigen Tiefe abhandelt.

2.1 PromethOS Framework

Das PromethOS Framework wurde in Kapitel 1.2 der Einleitung bereits vorgestellt.

In diesem Abschnitt wird nun auf diejenigen funktionalen Aspekte und deren Implementierungen eingegangen, die für die Kooperation der zu erstellenden *Memory Resource Control Architektur* mit dem *Netfilter-Framework* von besonderem Interesse sind. Eine detaillierte Beschreibung das kompletten Frameworks, dessen Komponenten sowie deren Implementierung ist in [Guin01] zu finden.

Das derzeitige PromethOS Framework erweitert das Netfilter-Framework um die Möglichkeit, Plugins zur Laufzeit installieren und auszuführen zu können. Diese Funktionalität kann PromethOS dadurch bereitstellen, dass das *PromethOS Target* die Verwaltung von - und Kommunikation mit - den Plugins übernimmt. Aus der Sicht von Netfilter ist das PromethOS-Target ein normales, statisch in das Framework eingebundenes Target. Die Existenz von Plugins ist dem Netfilter Framework nicht bekannt. Dementsprechend wird ein ankommendes Paket, welches einer von PromethOS angegebenen Matching-Regel entspricht, an die von PromethOS bei Netfilter registrierte Target-Funktion, target(), weitergeleitet.[1] [Rus03]

Die Funktion `target()` verteilt nun die ankommenden Pakete an die einzelnen Instanzen der geladenen Plugins. Hierzu wird zunächst die Plugininstanz ermittelt, und im Anschluss hieran das Paket an dieses Plugin[2] weitergeleitet.

Damit ein Plugin Pakete bearbeiten kann, muss dieses als Modul in das *Kernel Memory Space* geladen werden und dem *PromethOS-Framwork* bekannt gemacht werden. Es muss sich regi-

[1] Funktionen des PromethOS-Targets wurde in `net/ipv4/netfilter/ipt_PROMETHOS.c` implementiert
[2] Die Instanznummer wird dem Plugin ebenfalls übergeben

strieren. Hierzu ruft das Plugin eine vom PromethOS-Target exportierte Registrierungsfunktion auf, und teilt diesem auf diesem Wege alle für dessen Integration relevanten Informationen mit.

```
unsigned int promethos_register(char *name,
                                promethos_target_func_t tfunc,
                                promethos_config_func_t cfunc,
                                promethos_config_func_t recfunc,
                                promethos_print_func_t pfunc)
```

Beispielsweise gibt der erste übergebene Parameter den Namen des Plugins an, während der zweite Parameter die Funktion bestimmt, welche bei ankommenden, für das Plugin bestimmten Paketen aufgerufen werden soll.[3] Die Verwendung der restlichen Parameter kann in [Guin01] nachgelesen werden.

2.2 Linux 2.4

Die Entwicklung des Betriebssystems *Linux* - und damit einhergehend die Entwicklung des *Linux-Memory-Management-Subsystems* - ist in sofern ungewöhnlich, da sie auf keinem bewährten, aus dem Software Engineering bekannten Vorgehensmodell basiert. Anstatt Entwicklungszyklen zu folgen, bei denen Anforderungen ermittelt, ein Design erstellt und dieses implementiert und getestet wird, werden Änderungen am *Linux-Source-Code* in Reaktion auf das tatsächliche Verhalten von *Linux* in der wirklichen Welt und aufgrund intuitiver Entscheidungen von Entwicklern vorgenommen. Auch findet eine neue Version des Linux-Kernels keinen Abschluss in einer Dokumentation oder eines Handbuchs über dieses.

Dieses führte zu der Situation, dass aktuelle Versionen des Betriebssystemkerns kaum oder gar nicht dokumentiert sind. Mit Ausnahme einiger, eher allgemein gehaltener Abschnitte in einer kleinen Anzahl von Büchern oder Webseiten - die sich meist allerdings auf etwas ältere Kernelversionen beziehen - ist man bei der Linux-Kernelentwicklung meist auf des Lesen und Verstehen des Quellcodes oder auf einige wenige Mailinglisten und Foren angewiesen (c.f. [Gor03], Kapitel 1).

Ein gewisses Verständnis des Linux-Betriebssystems als solchem, als auch ein zum Teil recht detailliertes Wissen und Verstehen einzelner, im Kontext der Aufgabenstellung relevanter Aspekte, Funktionalitäten und Zusammenhänge - besonders im Bereich Memory-Management - ist für ein Verständnis des in der Linux-Architektur begründeten, zu lösenden Problems unerlässlich. Aus diesem Grunde werden nun in den folgenden Abschnitten dieses Kapitels eben diese Komponenten genauer betrachtet.

Es sei hier ausdrücklich erwähnt, dass ausschließlich auf diejenigen Komponenten und architektonischen Konzepte des Linux-Kernels eingegangen wird, die für ein Verständnis des Problems, des gewählten Lösungsansatzes sowie der Lösungsarchitektur unabdingbar sind.

Es würde den Rahmen dieser Diplomarbeit bei weitem sprengen, an dieser Stelle einen Anspruch auf Vollständigkeit erheben zu wollen. Einen guten Gesamtüberblick über den Kernel, die einzelnen Kernelkomponenten sowie deren Zusammenhänge vermitteln, neben anderen, [BoCe00], [Beck01] sowie [Aiva01]. Auch werden komplexe Zusammenhänge, die im Kontext dieser Diplomarbeit kein tieferes Verständis vermitteln, teilweise abstrahiert dargestellt. Es werden jedoch - wann immer möglich und sinnvoll - Referenzen angegeben, unter denen die einzelnen Sachverhalte in ihrer vollen Komplexität nachgelesen werden können.

[3] An dessen `target()` weitergeleitet werden soll

2.3 Linux Memory-Management

Aus Sicht der Speicherverwaltung bestehen ausführbare Einheiten eines Computersystems (Execution Context (EC))[4] - unter Linux sowohl Prozesse als auch der Kernel selbst - etwas vereinfacht ausgedrückt, aus Code und Daten, welche sich zur Ausführungszeit im Hauptspeicher (Random Access Memory (RAM)) befinden müssen.[5][6] Der Adressbereich einer ausführbaren Einheit besteht somit aus dem Speicherbereich seines Algorithmus (seinem Code) und den Speicherbereichen der Daten, die für dessen Abarbeitung Verwendung finden. Im folgenden Beschränken wir uns auf Prozesse; obgleich Wesentliches auch auf den Kernel zutrifft, bestehen doch einige, gerade im Rahmen der Aufgabenstellung dieser Arbeit relevante Unterschiede zwischen Aufbau und Management des Adressraums des Kernels und dem eines Prozesses, auf die in Abschnitt 2.3.4 eingegangen wird.

Oftmals ist jedoch nicht im Vorneherein bekannt, wieviel Speicher ein Prozess während seiner Ausführung zur Speicherung seiner Daten benötigen wird. Dieses hat zur Folge, dass Speicher dynamisch[7] von einem Prozess alloziert und adressiert können werden muss. Diese Adressbereiche sind dem Adressraum eines Prozesses ebenfalls zuzuordnen. Letzendlich ist der Adressraum eines Prozess somit die Gesamtheit der Speicherbereiche, welche von diesem referenziert werden können.

Als Speicherverwaltung (Memory-Management) bezeichnet man nun die Art und Weise, also die Verfahren und Techniken, wie die Verwendung des Hauptspeichers eines Computers koordiniert und kontrolliert wird.[8]

Ein Multitaskingbetriebssystem[9] wie Linux stellt bestimmte Anforderungen an seine Speicherverwaltung. So muss der Speicher eines Prozesses sowie der vom Kernel verwendete Speicher vor Zugriffen anderer Prozesse geschützt sein [Beck01]. Wäre dieser Schutz nicht vorhanden, könnte ein Prozess beliebig in den Speicher anderer Prozesse oder des Kernels hineinschreiben oder Daten auslesen, was die in *Kapitel 1.3* dargestellten negativen Auswirken auf System-Sicherheit und -Stabilität hätte.

Linux, oder genauer das Linux Memory-Management-Subsystem, bedient sich bei der Realisierung seines Speicherschutzkonzepts der eleganten Methode der Einführung virtueller Adressräume für jeden Prozess sowie für den Kernel selbst.

Das Prinzip virtueller Adressräume sowie dessen Anwendung und Ausprägung unter Linux 2.4 wird in der Sektion 2.3.1 (Seite 17) in einem im Kontext dieser Diplomarbeit relevantem Maß aufgezeigt. Weitere wichtige Aufgaben der Speicherverwaltung eines Betriebssystems, wie beispielsweise Segmentation, das Vermeiden von Speicherfragmentierung, die Verwendung von Caches oder die Auslagerung momentan nicht benötigter Informationen auf Sekundärmedien (Swapping) sind im Rahmen dieser Diplomarbeit nicht relevant und werden aus diesem Grunde hier nicht behandelt. Es sei hier auf [BoCe00]: Kapitel 2, 6 und 16, sowie auf [Gor03]: Kapitel 8, 11, 12 und 13 verwiesen, welche einen sehr guten Überblick über die genannte Memory-Management-Aspekte bieten.

2.3.1 Virtuelle Adressräume und Protection Domains

Die Notwendigkeit der Bereitstellung geschützter Speicherbereiche (sogenannter Protection Domains) wurde im vorhergehenden Abschnitt dieser Arbeit erläutert. Die Realisierung von Protection Domains unter Linux basiert auf dem Konzept virtueller Speicherverwaltung.

[4] EC, Task und Kernel Thread Thread und Prozess werden in *Sektion 2.4* genauer erläutert

[5] Hinzu kommen noch Verwaltungs- und Statusinformationen

[6] Auf die Möglichkeit der Auslagerung von Hauptspeicherinhalten sei hingewiesen.

[7] D.h. während der Laufzeit des Prozesses

[8] Defi nitionen und Beschreibungen des im Rahmen der Darstellung des Memory-Managements verwendeten Vokabulars fi nden Sie unter http://www.memorymanagement.org/glossary/m.html

[9] Der Begriff Multitasking bezeichnet das scheinbar parallele Ausführen mehrerer Programme durch schnelles Umschalten zwischen den Programmen.

Unter virtuellem Speicher versteht man, seiner Namensgebung entspechend, Speicher der nicht wirklich existent ist, ein programmiertechnisches Konstrukt also, welches zunächst einmal nichts mit dem physikalisch vorhandenen RAM eines Computers zu tun haben muss.

Virtuelle Speicherverwaltungssysteme funktionieren wie nachfolgend skizziert:

Einzelne Prozesse adressieren niemals direkt physikalischen Hauptspeicher des Computers; sie wissen nicht einmal etwas von dessen Strukturierung oder Größe. Wird nun auf eine Speicheradresse zugegriffen, so wird diese von dem Memory-Management-System als eine virtuelle, und nicht als eine physikalische Adresse interpretiert. Diese virtuelle Adresse wird nun von der CPU oder einer separaten Memory-Management-Unit (MMU) - auch Paging Unit genannt - mit Hilfe spezieller Tabellen, den Page-Tables[10], auf eine physikalische Adresse abgebildet.[11] Diese wird dann, sofern der Zugriff als autorisiert verifiziert wurde, zurückgegeben.[12] Für das Programm, welches den Speicherzugriff initiierte, ist dieser Übersetzungsprozess transparent.

Jedem Prozess sind nun individuelle Page-Tables zugeordnet, welche die Regeln für das *Mapping* seines virtuellen Adressraumes auf physikalisches RAM angeben. Die Speicherverwaltung des Betriebssystems trägt nun dafür Sorge, dass virtuelle Adressen verschiedener Prozesse niemals auf den gleichen physikalischen Speicher abgebildet werden. Die tatsächlich verwendeten Speicherbereiche verschiedener Prozesse (i.e. die Abbildungsmengen ihrer virtuellen Speicherbereiche) sind, wie in Abbildung 2.1 dargestellt, disjunkt.[13]

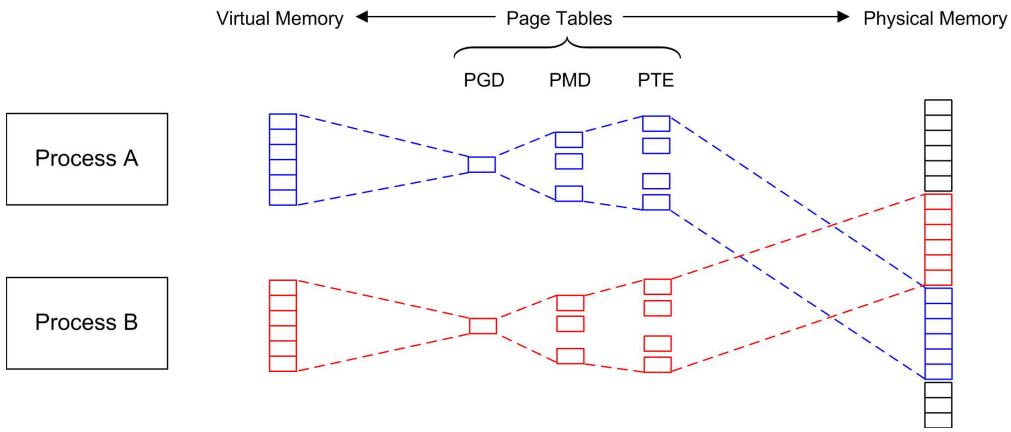

Abbildung 2.1: User Space Protection Domains

2.3.2 Speicherseiten

Physikalischer Speicher wird aus Effizienzgründen in *Page Frames*[14] (Speicherseiten) fester Länge strukturiert. Die Größe einer Speicherseite ist durch das Makro `PAGE_SIZE` in der Datei `asm/page.h` definiert. Für 32-Bit-Architekturen beträgt diese Größe meist 4 Kbyte, für 64-Bit-Architekturen meist 8 KByte (c.f. [Beck01], Seite 61).

Der Kernel muss permanent über die aktuellen Zustände aller *Page Frames* des Hauptspeicher informiert sein. Wenn ein Prozess dynamisch Speicher anfordert, so muss bekannt sein, welche

[10] Aufbau und Funktionsweise von Page-Tables werden in der Sektion 2.3.2 detailliert betrachtet

[11] Da aktuelle Betriebssysteme wie Linux ein Flat-Memory-Model werwenden, kann hier auf eine Darstellung der Transformation von logischen- in lineare Adressen durch die Segmentation Unit der CPU verzichtet werden, da diese für das weitere Verständnis nicht relevant ist [Shan01]. Falls nicht explizit deklariert, wird hier auf eine Unterscheidung von logischen und linearen Adressen verzichtet und ausschliesslich der Begriff der virtuelle Adresse verwendet.

[12] Ansonsten generiert die Paging Unit, wie in Sektion 2.5 beschrieben, eine Page-Fault-Exception

[13] Memory-Sharing bildet hier ein Ausnahme

[14] Die Bedeutung des Begriffs Page ist in der Literatur nicht eindeutig festgelegt. Während in einigen Werken unter Page die Struktur verstanden wird, die Verwaltungsinformationen über die physikalische Speicherseite - dort als Page Frame bezeichnet - enthält, bezeichnet Page in anderen Werken die physikalische Speicherseite als solche. Im Folgenden wird der Begriff Page in dem erstgenannten Sinne verstanden.

Speicherseiten noch frei sind - somit in den virtuellen Adressraum des Prozesses gemappt werden können - und welche bereits von anderen Prozessen oder dem Kernel alloziert wurden. Weiter muss bei jedem Zugriff eines Prozesses auf eine Speicherseite überprüft werden, ob der Zugriff legitim ist, d.h. ob die Seite tatsächlich im Adressraum des Prozesses liegt, oder ob sie beispielsweise zum Kernel-Address-Space gehört. Dieser Check ist notwendig, da das Linux-Virtual-Memory-Management zwar eine Inter-Prozess-Speicherverletzung durch die Einführung von Protection Domains ausschließt[15], unautorisierte Zugriffe eines Prozesses auf Speicherseiten des Kernels mit diesem Mechanismus jedoch nicht verhindern kann. Die Ursache dieses Sachverhalts wird im Abschnitt 2.3.4 dargestellt.

Die benötigten Statusinformationen der *Page Frames* werden in *Page-Deskriptoren*, das sind spezielle Strukturen vom Typ `page` gespeichert,[16] welche alle in einem einzigen Array gespeichert und verwaltet werden.

```
typedef struct page {
        struct list_head list;
        struct address_space *mapping;
        unsigned long index;
        struct page *next_hash;
        atomic_t count;
        unsigned long flags;
        struct list_head lru;
        struct page **pprev_hash;
        struct buffer_head * buffers;
#if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
        void *virtual;
#endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */
}
```

An dieser Stelle ist es wichtig zu erwähnen, dass eine Speicherseite die kleinste Einheit darstellt, welche einem virtuellen Speicherbereich zugeordnet werden kann. Benachbarte Speicheradressen innerhalb einer Page werden somit ausnahmslos auf benachbarte virtuelle Speicheradressen abgebildet; die *Offsets* der Speicheradressen bzgl. der Basisadresse ihrer Speicherseite bleiben somit erhalten. Abbildung 2.2 veranschaulicht diesen Sachverhalt.

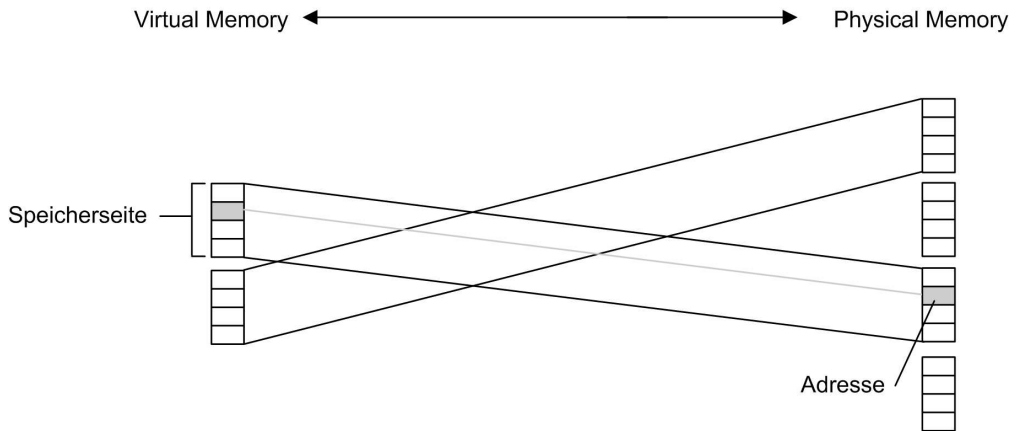

Abbildung 2.2: Atomares Mapping von Speicherseiten

Eine solche Zuordnung von virtuellen zu physikalsicher Adressen, *Mapping*, basiert auf dem im nachfolgenden Abschnitt beschriebenen *Paging-Verfahren*.

[15] Wie in Sektion 2.3.1 beschrieben
[16] Defi niert in `include/linux/mm.h`

2.3.3 Speicheradressierung und Paging

In dem vorhergehenden Abschnitt wurde auf die Strukturierung des Hauptspeichers in *Pages* fester Länge sowie auf deren atomare Behandlung bezüglich der Zuordnung von virtuellem und physikalischem Speicher eingegangen.

In diesem Abschnitt wird nun beschrieben, wie die *MMU* das *Mapping* zwischen virtuellen und physikalischen Adressen realisiert, d.h., wie eine physikalische Adresse anhand einer virtuellen Adresse ermittelt wird.

Jedem Prozess wird bei dessen Generierung ein sogenanntes *Page Global Directory (PGD)* eindeutig zugewiesen. Die Adresse dieses *PGD* wird, sobald der Prozess “gescheduled” wird, in ein hierfür vorgesehenes Register geladen und während der Ausführung des Prozesses von der *MMU* , wie später in dieser Sektion beschrieben, für dessen *Mapping* verwendet.

Ein *PGD* ist eine Speicherseite, welche ein Array von `pgd_t`[17] Struktures enthält. Die Adresse des *PGD* enspricht der des untersten *PGD*-Eintrags und wird fortan *Page Global Directory Base Address (PGDBA)* genannt. Jeder Eintrag im *PGD* zeigt auf ein *Page Middle Directory (PMD)*. Ein *PMD* ist eine Speicherseite, welche ein Array des Typs `pmd_t` enthält. Einträge des *PMD* zeigen wiederum auf *Page Table Entry (PTE)* Speicherseiten, die Einträge des Typs `pte_t` enthalten. Diese Einträge zeigen nun auf physikalische Speicherseiten, die die eigentlichen Daten enthalten.

Wird nun ein Speicherzugriff von einem Prozess initiiert, zerlegt die MMU die angeforderte Adresse nach dem in Abbildung 2.3 dargestelltem Schema in 4 Teile. Wie die Bits jeweils auf die verschiedenen Abschnitte verteilt werden ist architekturabhängig.

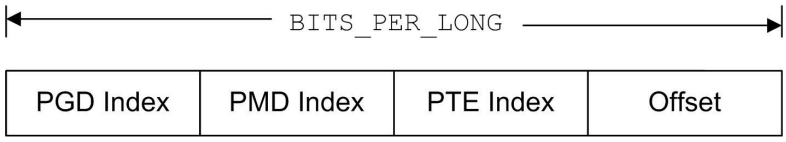

Abbildung 2.3: Format einer virtuellen Adresse

Mit Hilfe dieser vier Abschnitte werden nun die Page Tables - um die physikalische Adresse zu ermitteln - wie in Abbildung 2.4 dargestellt traversiert:

Der erste Abschnitt wird als Index in das *PGD* verwendet.[18] Der ermittelte Eintrag zeigt auf eine *PMD*. Der zweite Teil der virtuellen Adresse stellt nun ein Index in dieses *PMD* dar. Der so referenzierte Eintrag verweist auf eine Pagetabelle. Der dritte Teil dient als Index in dieser Pagetabelle. Der nun referenzierte Eintrag zeigt auf einen Speicherseite im physikalischen Speicher. Der vierte Teil gibt das *Offset* innerhalb der selektierten Speicherseite an, welches der angeforderten Adresse entspricht. Dieses Verfahren wird *Three-Level-Paging-Verfahren* genannt, weil die physikalische Speicheradresse durch die Traversierung von 3 *Page Tables* erfolgt. *Linux* verwendet ausschliesslich das *Three-Level-Paging-Verfahren*, auch wenn die unterliegende MMU ein *Two-Level-Paging-Verfahren* benutzt. Die dritte Stufe wird dann von dem *Linux Memory Management* emuliert.

2.3.4 Kernel Space Versus User Space

Prozessen sind jeweils individuelle *Page-Tables* zugeordnet, die vom Kernel so konstruiert werden, dass virtuelle Speicheradressen verschiedener Prozesse niemals auf dieselbe physikalische Adresse abgebildet werden.[19]Unter Linux wird zur Adressierung von Speicher, wie in Abbildung 2.3 dargestellt, ein `unsigned-long`-Typ verwendet. In einer 32-Bit-Architektur[20]

[17]Die Typen `pgt_t`, `pmd_t` und `pte_t` werden in `include/asm/page.h` defi niert
[18]*PGDBA* ist der *MMU* wie oben beschrieben als Registerinhalt zugänglich
[19]Auf Memory-Sharing sei hingewiesen
[20]Wie der für die Implementierung gewählten IA32

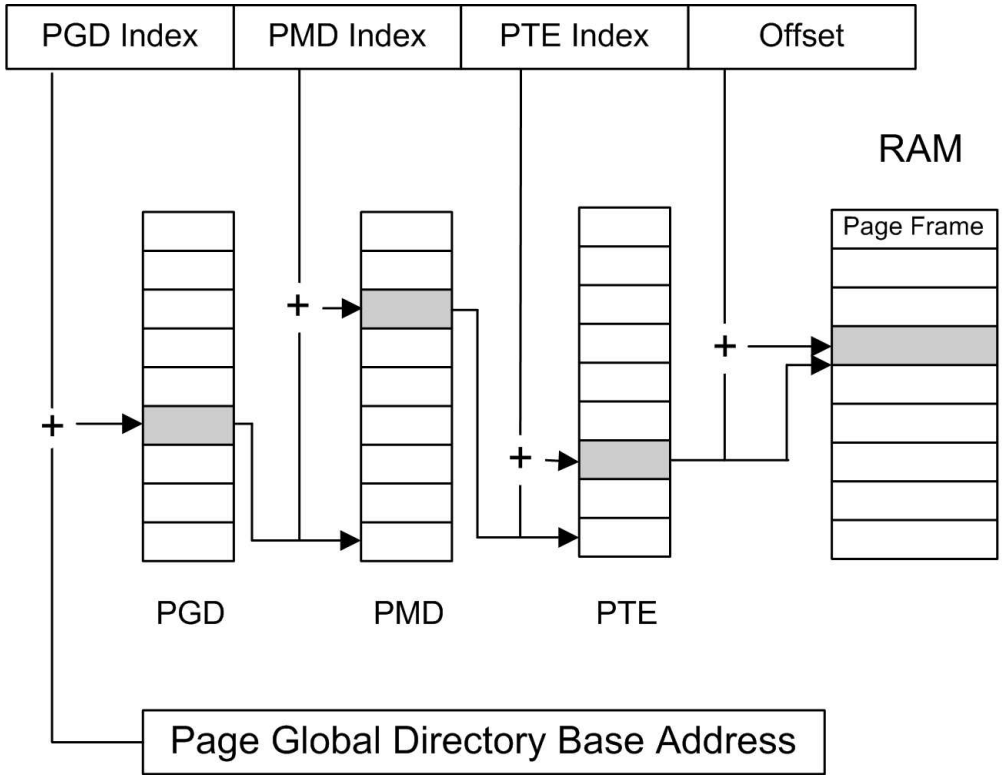

Abbildung 2.4: Paging: Übersetzung der virtuellen Adresse

können somit maximal 4GB adressiert werden.

Linux stellt Prozessen diesen maximal adressierbaren virtuellen Speicherraum von 4GB zur Verfügung, wobei folgende Einschränkung gilt:

- Virtuelle Adressen von `0x00000000` bis `PAGE_OFFSET -1` können von Prozessen in ihren virtuellen Adressraum eingebunden und adressiert werden.
- Virtuelle Adressen von `PAGE_OFFSET - 0xffffffff` dürfen von Prozessen nur dann adressiert werden, wenn diese sich im *Kernel Mode* befinden, beispielsweise gerade ein System Call ausgeführt wird. Diese Adressen werden niemals in den Adressraum eines Prozesses integriert.

`PAGE_OFFSET` ist ein Makro, definiert in `include/asm/page.h` als `0xC0000000` (3GB). Virtuelle Adressen oberhalb `PAGE_OFFSET` sind somit dem Kernel vorbehalten; diesen Raum nennt man dementsprechend *Kernel Memory Space*.

Während nun durch die Speicherverwaltung von Linux dafür gesorgt wird, dass virtuelle Adressen verschiedener Prozesse unterhalb `PAGE_OFFSET` nicht auf die gleichen physikalischen Adressen abgebildet werden, entspricht das *Mapping* von Adressen oberhalb `PAGE_OFFSET` bei allen Prozessen dem des Kernels. Dies wird dadurch erreicht, dass alle Einträge sämtlicher *PGDs* oberhalb `PAGE_OFFSET` auf die gleichen *PMDs* zeigen. Dieser Aspekt wird in Abschnitt 2.3.3 vertieft betrachtet.

2.3.5 Memory Address Space und Memory Descriptor eines Prozesses

Der Adressraum eines Prozesses besteht, wie bereits erwähnt, aus allen virtuellen Adressen die der Prozess benutzen darf. Prozesse nutzen jedoch nicht die kompletten 3 GB [21], sondern nur Teile hiervon. Informationen über die tatsächlich von dem Prozess verwendeten virtuellen

[21] Sektion 2.3.4

Adressbereiche, die Memory Regions, werden in Strukturen vom Typ `vm_area_struct`[22] gespeichert.

```
struct vm_area_struct {
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;

        struct vm_area_struct *vm_next;

        pgprot_t vm_page_prot;
        unsigned long vm_flags;

        rb_node_t vm_rb;

        struct vm_area_struct *vm_next_share;
        struct vm_area_struct **vm_pprev_share;

        struct vm_operations_struct * vm_ops;

        unsigned long vm_pgoff;
        struct file * vm_file;
        unsigned long vm_raend;
        void * vm_private_data;
};
```

Jede dieser *Memory Region Descriptors* identifiziert ein virtuelles Adressintervall. Durch eine einfache Verkettung dieser Deskriptoren über das Feld `vm_next` kann der komplette Adressraum des Prozesses ermittelt werden. Wie diese Adressen dann letztendlich auf physikalische Adressen gemappt werden, ist in diesem Zusammenhang nicht relevant. Es kann sogar vorkommen, dass Seiten innerhalb einer Memory Region eines Prozesses noch überhaupt nicht gemappt wurden, was dann zu einem sogenannten Page Fault führt. Hierüber ist, wie in [BoCe00] beschrieben, eine ressourcenfreundlichere Speicherverwaltung möglich.

Sämtliche Informationen bezüglich des *Process Address Space* sind dem *Memory Descriptor* des Prozesses zugänglich. Der *Memory Descriptor* ist vom Typ `mm_struct` (definiert in `include/linux/sched.h`) und wie folgt aufgebaut:

```
struct mm_struct {
      struct vm_area_struct * mmap;
      rb_root_t mm_rb;
      struct vm_area_struct * mmap_cache;
      pgd_t * pgd;
      atomic_t mm_users;
      atomic_t mm_count;
      int map_count;
      struct rw_semaphore mmap_sem;
      spinlock_t page_table_lock;

      struct list_head mmlist;
      unsigned long start_code, end_code, start_data, end_data;
      unsigned long start_brk, brk, start_stack;
      unsigned long arg_start, arg_end, env_start, env_end;
      unsigned long rss, total_vm, locked_vm;
      unsigned long def_flags;
      unsigned long cpu_vm_mask;
      unsigned long swap_address;
```

[22] Defi niert in `include/linux/mm.h`

```
     unsigned dumpable:1;
    /* Architecture-specific MM context */
    mm_context_t context;
};
```

Zum weiteren Verständnis sind zwei Felder von besonderer Relevanz:

- `vm_area_struct` zeigt auf die erste Memory Region des Prozesses. Durch Traversierung über die `vm_next` Felder der Memory Regions kann der komplette gültige Speicherbereich des Prozesses ermittelt werden.
- `pgd` zeigt auf die *Page Global Directory Base Adresse des Prozesses*. Dieser Wert wird in das entsprechende Register geladen und sorgt für ein korrektes *Mapping* virtueller auf physikalische Adressen sobald der Prozess durch den Scheduler zur Ausführung gebracht wird.

2.4 Linux Execution-Context-Management

Eines der fundamentalsten Konzepte eines Multitaskingbetriebssystems wie Linux ist das *Prozessmodell*[23].

Ein Prozess (Process) ist definiert als eine Instanz eines sich gerade in Ausführung befindenden Programms. Um diese Definition zu veranschaulichen stelle man sich vor, dass 16 User eines Computersystems zur gleichen Zeit den Editor `vi` ausführen. Obgleich es sich immer um das selbe Programm handelt (und sogar jeweils auf denselben ausführbare Progammcode zugegriffen wird)[24] sind hier 16 eigenständige *Prozesse* aktiv (c.f. [BoCe00], S. 64).

Prozesse werden im Linux-Source-Code und in der Literatur oft auch als *Tasks* bezeichnet. Im Kontext dieser Diplomarbeit werden die beiden Begriffe jedoch nicht synonym verwendet. Diese Unterscheidung ist für die in Kapitel 3 folgende Problemlokalisierung und das Verständnis des Lösungsansatzes, sowie der diese umsetzenden Zielarchitektur, erforderlich.

In den folgenden Sektionen werden nun die Begriffe Execution Context, Task, Prozess und Kernel Thread charakterisiert und gegeneinander abgegrenzt.

2.4.1 Execution Context - Charakterisierung

Die Definition eines *Prozesses* aus Sektion 2.4 entspricht prinzipiell der des *Tasks*. Jedoch laufen bestimmte Teile eines *Tasks* in einem weniger priviligiertem *User Mode*, innerhalb welchem nicht direkt auf das *Kernel Memory Space* oberhalb `PAGE_OFFSET` zugegriffen werden darf. Zugriffe hierauf werden über *System Calls* realisiert, wodurch die Task-Priviligierungsstufe auf Kernel-Niveau angehoben, und somit dieser Bereich adressierbar wird. Die Teile der Task im *User Mode* werden als Prozess bezeichnet [Beck01].

Da nun aber die Begriffe Task und Prozess oft synonym verwendet werden, wird fortan, um Konfusionen zu vermeiden, der Begriff Task keine Verwendung mehr finden und ansatt dessen der Begriff *Execution Context (EC)* benutzt.

Nach dieser eher anwendungsorientierten Sicht auf einen *Execution Context* folgt nun eine technische Charakterisierung desgleichen.

Um *ECs* verwalten zu können, benötigt der Kernel exakte Informationen über diese. So muss beispielsweise die Priorität eines *ECs* bekannt sein, ob dieser gerade von der CPU ausgeführt wird oder auf ein bestimmtes Ereignis wartet, welcher Adressraum mit diesem assoziiert ist, an welcher Stelle der Codeausführung der *EC* unterbrochen wurde und wie sein Stack aufgebaut

[23] Abstraktion, welche sämtliche Benutzeraktivitäten in Prozesse einteilt und als solche verwaltet
[24] Ermöglicht durch Memory Sharing

ist.

Hierzu ist jedem *EC* ein eigener *process descriptor* zugeordnet, der Informationen über eben diese Eigenschaften bereitstellt; der Aufbau eines *process descriptors* wird in Sektion 2.4.3 betrachtet.

Da jedem *EC* ein individueller *process descriptor* zugeordnet ist, wird der Begriff *EC* in dieser Dokumentation folgendermassen verwendet: Zusammengehörende Teile, welche einem bestimmten *process descriptors* zugeordnet sind, und so beispielsweise “gescheduled” werden können, werden als *ECs* bezeichnet.

2.4.2 User Process versus Kernel Thread

Der Begriff Prozess wurde in Abschnitt 2.4 erläutert. Ein Prozess ist ein *Execution Context* im *User Mode*, für eine Operation auf virtuelle Speicheradressen oberhalb `PAGE_OFFSET` dementsprechend nicht privilegiert. Jeder Prozess hat - wie in Sektion 2.3.5 beschrieben - einen eigenen *Memory Descriptor*, welcher den ihm zugänglichen virtuellen Speicherraum unterhalb `PAGE_OFFSET` beschreibt.

Ein *Kernel Thread* hingegen ist ein *EC*, welcher ausschliesslich im *Kernel Space* operiert. Es gilt:

- Ein Kernel Thread besitzt auch alle sonstigen Befugnisse von Kernelkomponenten. Er läuft auf *Kernelprivilegierungsstufe*.
- Da ein Kernel Thread sich den *Kernel Memory Space* mit allen weiteren Kernel Komponenten teilt und nicht im *User Memory Space* operiert, ist einem *Kernel Thread* auch kein virtueller Adressraum zugewiesen. Ein Kernel Thread besitzt somit keinen *Memory Descriptor*. Das `mm`-Feld seines in Sektion 2.4.3 beschriebenen *Descriptors* enthält einen Pointer auf `NULL`.
- Ein Kernel Thread ist *nonpreemptiv*. Wird ein Kernel Thread einmal von dem Scheduler ausgeführt, wird er solange ausgeführt (nach jedem Interrupt wieder gescheduled) bis er “freiwillig” die *CPU* an andere *ECs* abgibt.

2.4.3 Execution Context Descriptor

Ein *Execution Context Descriptor* ist eine Struktur vom Typ `task_struct`, definiert in `include/linux/sched.c`. Er enthält Informationen über den *EC*, welche der Kernel zur Verwaltung desselben benötigt.

Abbildung 2.5 zeigt den Aufbau des *Execution Context Descriptors*. Felder dieser Struktur, die im Zusammenhang dieser Diplomarbeit von besonderem Interesse sind, werden im Folgenden erläutert.

- `state`
  Aktueller Status des *ECs*. `state` kann fünf Werte annehmen. Hiervon sind drei von besonderem Interesse:
  - `TASK_RUNNING` Der *EC* wird gerade von der CPU ausgeführt oder ist bereit hierfür, d.h. der *EC* soll sobald als möglich “gescheduled” werden
  - `TASK_INTERRUPTABLE` Der *EC* “schläft” bis ein Ereignis eintritt, auf welches dieser wartet
  - `TASK_STOPPED` Der *EC* wurde gestoppt. Er wird, solange er diesen Status hat, nicht ausgeführt, d.h. er wird nicht gescheduled
- `pid`
  EC-Identifikationsnummer. Pro CPU besteht eine 1:1-Relation zwischen *EC* und `pid`. Somit ist jeder *EC* auf Uniprozessorsystemen eindeutig anhand seiner `pid` identifizierbar.

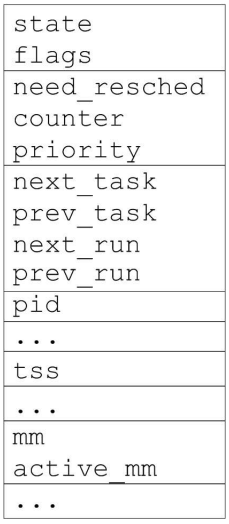

Abbildung 2.5: Execution Context Descriptor

- `mm`
  Dieses Feld zeigt auf den *Memory Descriptor* des *ECs*. Falls diesem kein *Memory Descriptor* zugewiesen ist (*Kernel Thread*), zeigt `mm` auf `NULL`

- `active_mm`
  Dieses Feld zeigt ebenfalls auf den *Memory Descriptor* des *ECs*. Bei Prozessen sind die Werte von `mm` und `active_mm` per Definition identisch. Einem Kernel Thread wird, wenn dieser gescheduled wird, der Wert des `active_mm` Feldes des vor diesem aktiven *Execution Contexts* zugewiesen. Ansonsten zeigt der `active_mm`-Pointer bei Kernel Threads auf `NULL`

- `tss`
  Referenziert das *Task State Segment des ECs*. Die Betrachtung dieses Segments ist Gegenstand von Sektion 2.4.4

2.4.4 Execution Context Switching and Scheduling

Linux ist ein Multitaskingbetriebssystem. Um nun ECs *quasiparallel* ausführen zu können, muss der Kernel in der Lage sein, sich gerade in Ausführung befindende ECs anzuhalten und die Ausführung eines anderen EC dort fortzusetzen, wo dieser zuvor unterbrochen wurde. Diese Aktivät bezeichnet man als *process switching, task switching oder context switching*. Das hierfür zuständige Linux-Subsystem ist der *Scheduler*.

ECs bemerken nichts von diesem *Scheduling*, es muss transparent geschehen. Dies wird dadurch ermöglicht, dass der Zustand des ECs, in welchem er von dem *Scheduler* unterbrochen wurde, bei einem *Context Switch* wieder exakt rekonstruiert werden muss. Somit werden auch sämtlich Registerinhalte (*Hardware Context*) auf die vorhergehenden Werte zurückgesetzt. Der aktuelle *Hardware Context* muss also bei jedem *Context Switch* gespeichert, und der des zu schedulenden ECs muss wieder in die Register zurückgeladen werden. Diese Informationen hierfür werden in speziellen Segmenten, den *Task State Segments* abgespeichert, welche von dem Feld `tss` des Process Descriptors (beschrieben in Sektion 2.4.3) referenziert werden (c.f. [BoCe00], S. 78-86).

Under anderem muss bei einem *Context Switch* der korrekte *Page Global Directory Base Address* Wert in das hierfür vorgesehene Register geladen werden, damit der EC die richtigen Page Tables verwendet (also auf den richtigen Adressraum zugreift). Der Scheduler behandelt Prozesse und Kernel Threads in zweierlei Hinsicht unterschiedlich:

- User Space Prozesse sind preemptiv. Kernel Threads sind es nicht.

- Falls ECs mit eigenen *Memory Descriptor* (Prozesse) gescheduled werden, wird die über das `pgd`-Feld des *Memory Descriptors* referenzierte *Page Global Directory Base Address* für das *Mapping* der virtuellen Adressen verwendet. Falls einem EC kein *Memory Descriptor* zugewiesen ist (Kernel Thread) “erbt” dieser den Deskriptor des unmittelbar vor ihm ausgeführten ECs; auch die PGDBA wird geerbt (das entsprechende Register wird nicht verändert). Dies ist bei Kernel Thread aus dem Grunde keine Problem, da ja das *Mapping* von Adresssen oberhalb `PAGE_OFFSET` sowieso bei allen Page Table Sets identisch ist, und ein Thread ausschliesslich im *Kernel Memory Space* operiert. Folgendes Codefragment aus `kernel/sched.c` ist hierfür verantwortlich:

```
prepare_to_switch();
{
      struct mm_struct *mm = next->mm;
      struct mm_struct *oldmm = prev->active_mm;
      if (!mm) {
            if (next->active_mm) BUG();
            next->active_mm = oldmm;
            atomic_inc(&oldmm->mm_count);
            enter_lazy_tlb(oldmm, next, this_cpu);
      } else {
            if (next->active_mm != mm) BUG();
            switch_mm(oldmm, mm, next, this_cpu);
      }

      if (!prev->mm) {
            prev->active_mm = NULL;
            mmdrop(oldmm);
      }
}
```

2.4.5 Inter Process Communication

Speicherbereiche verschiedener Prozesse sind, wie in Sektion 2.3 beschrieben, disjunkt. Somit können Prozesse nicht einfach Informationen dadurch austauschen, dass von *Prozess A* in den Speicher an der Adresse X geschrieben, und von einem *Prozess B* von dort ausgelesen wird.

Auch eine Kommunikaton zwischen Prozessen und dem Kernel funktioniert nicht in der beschriebenen Weise, da zum einen ein Prozess nicht auf das *Kernel Memory* zugreifen kann, der Kernel selbst auch nicht direkt auf den *User Space Memory* zugreift.

Oft ist eine solche *Inter Process Communication* aber notwendig und es existieren Verfahren, mit denen diese unter *Linux* realisiert werden kann. Bei der Entwicklung der Zielarchitektur werden die beiden nachfolgend genannten Verfahren eingesetzt:

- *Memory Sharing*
  Virtuelle Speicherseiten verschiedener ECs werden auf identische physikalische Speicherseiten abgebildet.

- *Das proc filesystem*
  Wird hier eine Datei registriert, so werden Schreib- und Leseaktionen auf diese Datei als Aufrufe explizit hierfür entwickelter, registrierter Funktionen interpretiert. Auf diesem Wege können beliebige Daten zwischen ECs ausgetauscht werden.

2.5 Linux Interrupt and Exception Handling

2.5.1 Interrupts und Exceptions

Ein *Interrupt* ist ein Ereignis, welches eine Unterbrechung der Instruktionsfolge, die gerade von der CPU ausgeführt wird, zur Folge hat.

Solche Ereignisse korrespondieren mit elektronischen Signalen, die entweder in oder außerhalb des CPU-Chips generiert werden. Beispielsweise generiert der Systemtimer periodisch solche Interrupts, aber auch ein Tastaturdruck hat ein solches elektronisches Signal zur Folge, auf welches dann entsprechend von der CPU reagiert wird. Eine sogenannte *Interrupt Service Routine (ISR)* wird ausgeführt, welche notwendigen Schritte als Reaktion auf diesen *Interrupt* ausführt.

Exceptions sind ebenfalls Interrupts, die von der CPU direkt behandelt werden. Eine Exception entspricht einer Ausnahmesituation, beziehungsweise einem Fehler. So wird beispielsweise bei dem Versuch, auf eine virtuelle Adresse zuzugreifen, welche nicht auf eine physikalische Adresse gemappt ist, oder falls der Zugriff nicht autorisiert ist (beispielsweise wenn ein Prozess versucht auf eine Speicheradresse im Kernel Memory Space zuzugreifen) eine solche Exception generiert. Auf eine Exception wird ebenfalls durch die Ausführung einer Routine - einem *Exception Handler* - reagiert.

Wie nun das Interrupt- und Exceptionhandling funktioniert, beziehungsweise welche Schritte hierbei ausgeführt werden, wird in 2.5.2 genauer dargestellt.

2.5.2 Interrupt Handling und die Interrupt Descriptor Table

Nach der Ausführung eines Befehls durch die CPU enthält das Registerpaar `cs` und `eip` die Adresse der nächsten auszuführenden Instruktion [BoCe00].

Registriert die *Interrupt Control Unit* der *CPU* ein elektronisches Signal, so wird dieses als Interrupt interpretiert. Dieses Signal ist so kodiert, dass es einem Wert zwischen 0 und 255 entsprechend decodiert werden kann. Die Control Unit initiiert nun folgende Schritte:

1. Der Interrupt wird in einen Wert `i` decodiert.
2. Die Adresse der *Interrupt Descriptor Table* wird aus dem hierfür bestimmten Register gelesen.[25] Die *Interrupt Descriptor Table (IDT)* ist eine Systemtabelle, welche allen registrierten *Interrupts* bzw. *Exceptions* jeweils die Adresse ihrer korrespondierenden Behandlungsroutine zuordnet. Die Adresse der *IDT* wird in ein Register geladen und somit der CPU zugänglich gemacht.
3. Der Wert `i` wird als Index in diese *IDT* verwendet; der so ermittelte Eintrag in der Tabelle wird gelesen.
4. Die Basisadresse des Segments, welches den *Interrupt- oder Exception-Handler* beinhaltet wird nun ermittelt. Weiter werden einige Sicherheitsüberprüfungen vorgenommen.
5. Falls es sich um einen Fehler handelte, wird die logische Adresse des Befehls über das Registerpaar `cs` und `eip` erneut geladen, sodass dieser Befehl erneut ausgeführt werden kann.
6. Die Inhalte der Register `eflags`, `cs` und `eip` werden auf den Stack kopiert; falls es sich um eine Exception mit Error-Code handelte, wird dieser Code ebenfalls auf den Stack kopiert.
7. Die Register `cs` und `eip` werden entsprechend den aus dem `i`-ten Eintrag der IDT ermittelten Werten geladen. Diese Werte definieren die logische Adresse der ersten Instruktion des Interrupt- oder Exception-Handlers; der Handler wird nun ausgeführt.

[25] Auf einer IA32 ist dies das idtr Register

8. Nachdem der Handler die Bearbeitung abgeschlossen hat, gibt dieser die Kontrolle wieder an das unterbrochene EC ab. Dies geschieht durch den Aufruf der `iret`-Instruktion, welche unter anderem die auf dem Stack gespeicherten `cs`, `eip` und `eflags` Werte wieder in die entsprechenden Register lädt, sowie etwaige Bereinigungen auf dem Stack vornimmt.

2.5.3 Interrupt- und Exception-Handling versus Context-Switch

Registriert der für das Interrupt-Handling zuständige Controller der CPU ein elektronisches Signal, welches einem Interrupt bzw. einer Exception entspricht, wird - wie in Abschnitt 2.5.2 beschrieben - über die IDT die Adresse der mit dem Interrupt assoziierten ISR ermittelt, und nach einigen notwendigen Operation auf dem aktuellen *Stack* ausgeführt. Anschliessend wird die Kontrolle wieder an den zuvor unterbrochenen EC übertragen.

Hierbei ist Folgendes zu beachten: Während des *Interrupt- oder Exception Handlings* wird kein *Context Switch* durchgeführt. Die Interruptbehandlung erfolgt vollständig in dem aktuellen Kontext und verwendet somit für das Mapping von virtuellen Speicheradressen die *Page Tables* des gerade aktiven *ECs*. Da sämtliche *ECs* des Systems den *Kernel Memory Space* identisch mappen spielt es für den *Interrupt Handler* keine Rolle, welche PGDBA gerade in dem Register der *MMU* geladen ist.[26]

2.6 Linux Kernel Modules

2.6.1 Module - Erweiterung der Kernel-Funktionalität zu dessen Laufzeit

Aus der Sicht des Kerns ist ein Modul zur Laufzeit link- und entfernbarer Objektcode. Dieser Objektcode wird gleichberechtigt in den Kern integriert, läuft also im *Kernel Mode*. [Beck01]

2.6.2 Implementierung von Modulen

Module werden im *ELF Object File* Format im Dateisystem gespeichert (c.f. [BoCe00], Kapitel 19.2). Mit Hilfe des User-Space-Tools `sbin/insmod` können sie in das RAM des Computers geladen werden. Sie bestehen aus den folgenden Komponenten:

- Module Object, das ist eine Struktur die das Modul beschreibt
- Ein String der den Modulnamen enthält
- Code, welcher die eigentliche Modulfunktionalität bereitstellt.

Module nutzen die Funktionalität des Kernels, indem Sie die Liste der von diesem exportierten Symbole auflösen. Weiter können Module auch selbst Funktionalitäten als Symbole exportieren, und somit anderen Komponenten des Kerns zugänglich machen.

Wichtig ist hier noch zu erwähnen, dass Module bei deren Installation in den Kernel initialisiert werden. Der im Modul enthaltende Initialisierungscode wird noch wärend des Linking-Vorgangs ausgeführt.

Eine detaillierte Beschreibung der einzelnen Felder des *Module Objects*, des Linking- und Unlinking-Vorgangs eines Moduls sowie des Symbol-Handlings findet man in *[BoCe00], Appendix B*.

2.6.3 Kernel Module versus Kernel Thread

Die Gemeinsamkeit von *Kernel Modulen* und *Kernel Threads* ist, dass deren beider Code auf Kernelprivilegierungsniveau ausgeführt wird und somit im *Kernel Memory Space* operiert. Ansonsten ist diesen beiden Konstrukten wenig gemein:

[26] Da dieser lediglich im Kernel Memory Space operiert

- Ein *Kernel Modul* ist zur Laufzeit link- und entfernbarer Objektcode, der, sobald er in den Kernel gelinkt wurde, als “normaler” Code des Kernels betrachtet werden kann.
- Ein *Kernel Thread* ist ein *EC*. Er hat einen eigenen *process descriptor*, und wird vom Scheduler behandelt wie andere *ECs* (z.B. *Prozesse*) auch.[27]

[27] Mit der in Sektion 2.4.2 bechriebenen Einschränkung

Kapitel 3

PromethOS v2.0

3.1 Implikationen aus Sektion 1.4 und Sektion 2

Die Zielsetzung dieser Arbeit kann wie folgt in Teilziele untergliedert werden:

1. Erweiterung des *Linux 2.4 Memory Management Subsystems* um das Potential, *Protection Domains* im Kernel Space zur Verfügung stellen zu können. Folgende Kriterien hinsichtlich des Designs sollen hierbei soweit wie möglich Berücksichtigung finden:
   - Nahtlose Integration in den Kernel
   - Verwendung und gegebenenfalls Erweiterung von Linux-Standardprogrammen
   - Portabilität des erstellten Codes
2. Das PromethOS Framework ist zu erweitern, so dass die in *Punkt 1* entwickelte *Linux MM* Funktionalität auf dem ANN genutzt werden kann. Ein funktionierender Kontroll- und Datenaustausch zwischen Plugins und dem PromethOS-Core, sowie zwischen dem PromethOS-Core und Netfilter ist hierfür unabdingbar.

Schlussendlich sind Effizienz und Stabilität des entwickelten Systems zu ermitteln.

Aus der in Sektion 2.3 - 2.6 durchgefürten Analyse können folgende, für die Problemlösung relevanten Erkenntnisse[1] extrahiert werden:

1. Das Linux MM basiert auf dem Prinzip *virtueller Speicherverwaltung*.
2. Das *Matching* virtueller auf physikalische Adressen - *Paging* - wird durch *Page Tables* bestimmt.
3. Welche Page Tables verwendet werden, wird bestimmt durch die sich in dem entsprechenden Register befindende Basisadresse des *PGD*, die *PGDBA*. Verschiede Prozesse haben verschiedene *PGDs*
4. RAM wird Seitenweise verwaltet. Eine Speicherseite ist die kleinste *Paging-Einheit*
5. *Projektionsmengen* virtuellen Speichers verschiedener ECs sind im User Memory Space (unterhalb `PAGE_OFFSET`) *disjunkt*.[2]
6. Sämtliche Page Table Sets mappen den gesamten Kernel Memory Space auf identischen physikalischen Speicher.
7. Komponenten mit Kernelprivilegien haben uneingeschränkten Zugriff auf das gesamte *Kernel Memory Space*. User Space Applikationen dürfen diesen Speicher nicht direkt adressieren.

[1] Nachfolgend durch E1, E2, ... referenziert
[2] Auf Memory Sharing sei hingewiesen

8. Ein *Prozess* ist ein *EC* mit eigenem *Memory Descriptor* und eigenem *PGD*, der ausschliesslich im User Memory Space operiert.

9. Ein *Kernel Thread* ist ein *EC* ohne *Memory Descriptor* und *PGD*. Er verwendet immer die gerade aktiven *Page Tables*. Dies ist möglich, da er nur im *Kernel Memory Space* operiert (dementsprechend Kernel-Privilegien besitzt), welches von allen *Page Table Sets* identisch gemappt wird.

10. *Kernel Threads* sind *nonpreemptiv*. Mit Hilfe eines sogenannten *Preemption Patchs* kann diese Eigenschaft der Kernel Threads allerdings geändert werden, sodass diese wie Prozesse “gescheduled” werden.

11. *Interrupts und Exceptions* unterbrechen die aktuelle Befehlssequenz der CPU um eine Routine - *ISR* - auszuführen, durch welche auf diesen *Interrupt* reagiert wird.

12. *Interruptbehandlung* findet innerhalb des gerade aktiven Kontexts statt. Die aktuellen Page Tables werden weiterverwendet.

13. Die *IDT* befindet sich im RAM des Computers. Sie assoziiert *Interrupts und Exceptions* mit Adressen der für diese zuständigen *ISRs* (bzw. Exception Handler).

14. Ein *Modul* ist zur Laufzeit link- und entfernbarer Objektcode, welcher gleichberechtigt in den Kern integriert wird. Er läuft somit im *Kernel Mode*.

Implikationen hieraus bezüglich Ziel 1:

- E5, E8: Das Linux MM unterstützt im *User Memory Space* bereits *Protection Domains*.
- E2, E3: Protection Domains im User Space basieren auf zwei Konzepten:
  - Jeder Prozess hat eigene Page Tables.
  - Diese Page Tables mappen die virtuellen Speicherbereiche verschiedener Prozesse auf verschiedene physikalische Speicherbereiche.
- E6, E7: Im *Kernel Memory Space* können keine Protection Domains bereitgestellt werden. Alle Kernelkomponenten, auch Kernel Threads, habe die gleiche Sicht und vollen Zugriff auf Adressen oberhalb `PAGE_OFFSET` .
- E9, E10: *Kernel Threads* werden im *Kernel Space* instanziert und bilden dort eigene Kontexte, die gescheduled werden und auch sonst Prozessen von ihrer Struktur her sehr ähnlich sind. Sie haben keinen eigenen virtuellen Speicherbereich.

Die aufgeführten Implikationen lokalisieren die eigentliche Problemursache. Sie führen vor Augen, warum unter Linux in Kernel Space keine *Memory Resource Control* implementiert ist, und diese somit hinzugefügt werden muss. Abbildung 3.1 illustriert diesen Sachverhalt.

Da die unter *Punkt 1* der Zielsetzung genannte Erweiterung des *Linux MM* Grundlage für die unter Punkt 2 genannte Nutzung dieser Erweiterung ist, wird in 3.2.1 zunächst ein Ansatz aufgezeigt, wie eine solche Erweiterung realisert werden kann. Im Anschluss daran folgt die Beschreibung der Architektur, welche diesen Ansatz umsetzt. In dem letzten Abschnitt des Kapitels wird dargestellt, wie die entwickelten *Memory Resource Contol Architektur* in das *Netfilter-Framework* integriert, und somit als *NodeOS* eingesetzt werden kann.

3.2 Memory-Resource-Control-Architektur

3.2.1 Lösungsansatz

Die Lösungsidee, wie das *MM* um einen Speicherschutz im *Kernel Space* erweitert werden kann, besteht in der *Transformation* des in Abbildung 3.1 aus Sektion 3.1 dargestellten Speicherschutzmodells aus dem User Space in den Kernel Space.

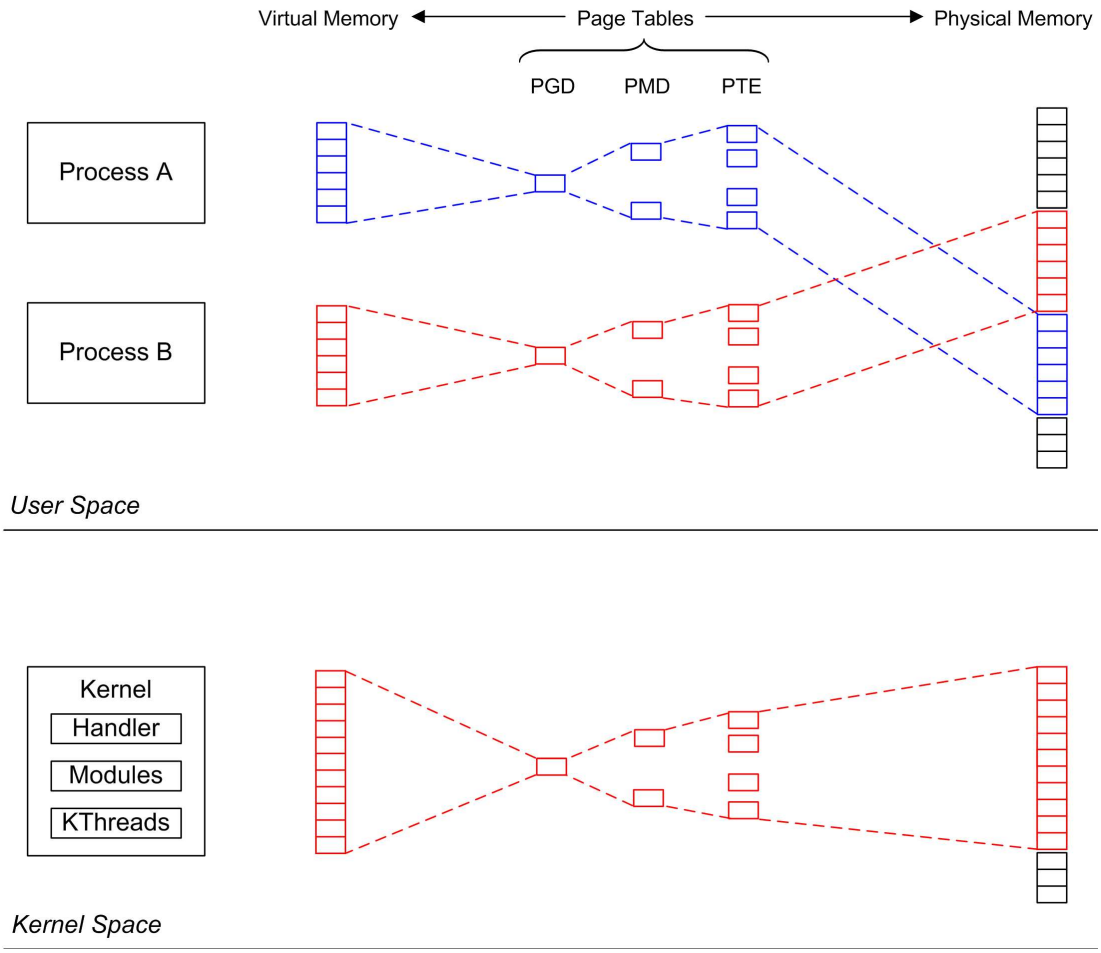

Abbildung 3.1: Problemidentifikation

Abbildung 3.2 veranschaulicht das im Folgenden beschriebene Transformationsverfahren:

Protection Domains im User Space basieren darauf, dass unterschiedlichen Prozessen unterschiedliche Page Tables zugeordnet sind, welche die virtuellen Speicherbereiche der Prozesse auf disjunkte physikalische Speicherbereiche abbilden.

Um Protection Domains im Kernel zu ermöglichen, kreiert man zunächst ein neues Set von Page Tables, da alle im System vorhandenen Page Tables den kompletten Kernel Memory Space mappen. Diese Page Tables werden nun insofern eingeschränkt, als dass sie lediglich bestimmte Ausschnitte des *Kernel Memory Space*, nämlich die für ein Plugin erlaubten Speicherbereiche, mappen. Alle anderen virtuellen Adressen werden nicht gemappt. Sie bekommen den Status `NOT_PRESENT` zugewiesen, was zur Folge hat, dass eine Adressierung dergleichen nicht den Inhalt einer physikalischen Adresse zurückgibt, sondern einen *Page Fault* auslöst (mit Hilfe dessen auf den unerlaubten Zugriff reagiert werden kann).

Plugins werden als Kernelmodule implementiert. Der Kernel betrachtet den *Module-Code* als “normalen” Kernel-Code. Damit für das Paging der Plugins aber die generierten, eingeschränkten Page Tables (*Restricted Page Tables*) Verwendung finden, muss die Basisadresse des zugehörigen PGDs in das von der MMU hierfür vorgesehene Register geladen werden.

Nach *E8, E9* und den Erkenntnissen aus Abschnitt 2.4.4 übernimmt der Memory Manager bzw. der Scheduler dieses Umschalten auf die mit einem Kontext assoziierten *Page Tables*, *bei Prozessen*, bei jedem Context Switch. Um diesen Mechanismus für die Plugins nutzen zu können, müssen diese in einem EC aufgeführt werden, dessen `pgd`-Feld seines *Memory Descriptors* auf die Basisadresse der *Restricted Page Tables* zeigt.

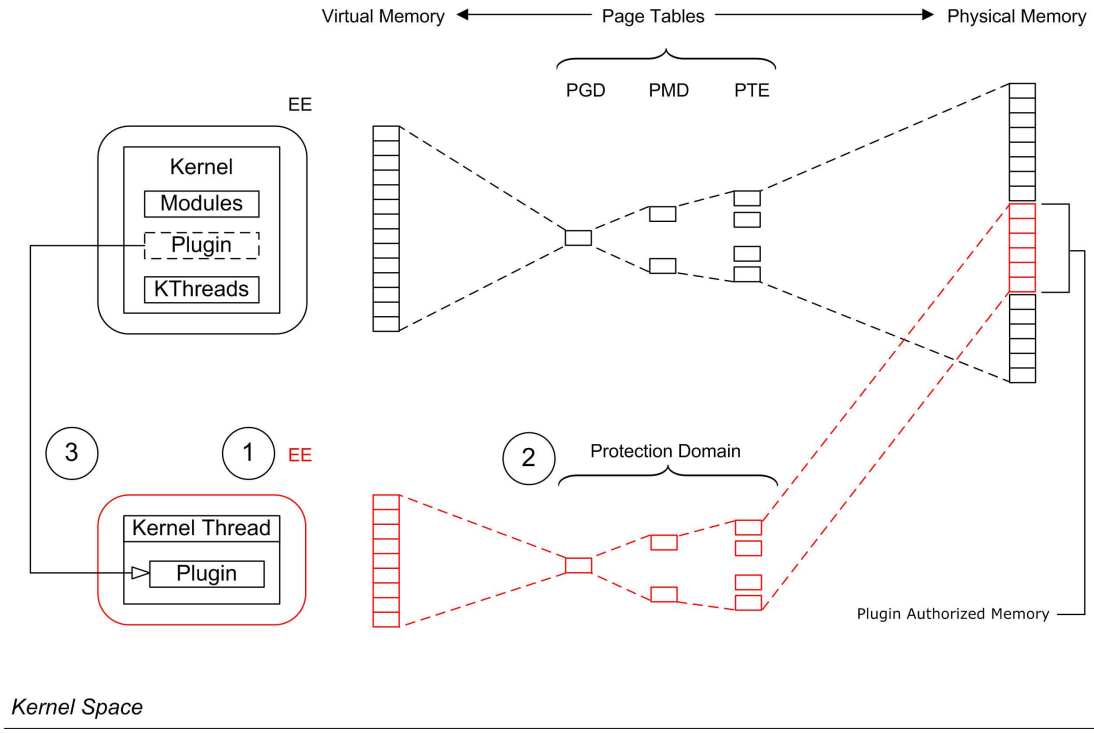

Abbildung 3.2: Generierung einer EE mit Protection Domain im Kernel Space

Ein solcher EC wird unter Linux nur im *User Space* angeboten. Es existieren zwar eigene Kontexte auch im *Kernel Space*, Kernel Threads, diese besitzen jedoch keinen eigenen *Memory Descriptor* und “erben” somit die Page Tables des vor diesen geschedulten ECs.

Alloziert man nun eine Struktur vom Typ `mm_struct`, und weisst die Adresse der Struktur dem `mm`-Feld des *Process Descriptors* des Kernel Threads zu, so hat dieser Kernel Thread fortan einen eigenen *Memory Descriptor*. Dem `pgd`-Feld des *Memory Descriptors* wird nun die Basisadresse der *PGD* der *Restricted Page Tables* zugewiesen. Auf diese Weise kümmert sich von da an der *Context Switch Mechanismus* von Linux um die korrekte Umschaltung auf die *Restricted Page Tables* sobald das Plugin gescheduled wird. Abbildung 3.3 stellt die Referenzierungen der einzelnen Strukturen im Zusammenhang dar.

Nachdem nun ein Kernel Thread kreiert wurde, diesem ein *Memory Descriptor* zugewiesen wurde - welchem wiederum das PGD der Restricted Page Tables zugewiesen wurde - muss noch der ausführbare Code des Plugins als *Image*[3] dem Kernel Thread übergeben werden. Abschnitt 3.3.2 beschreibt das Handling der auf diese Weise generierten *EEs*. Abschnitt 3.3.5 geht auf damit einhergehende, notwendige Änderungen des Linux Interrupt Handlings ein. In Sektion 3.2.2 wird die Architektur vorgestellt, welche den Ansatz im Sinne der Aufgabenstellung der Diplomarbeit umsetzt.

3.2.2 Gesamtarchitektur im Überblick

Die um eine Speicherzugriffskontrolle der Plugins durch deren Ausführung innerhalb von Protection Domains erweiterte PromethOS-Architektur zeigt Abbildung 3.4.

Verglichen mit dem PromethOS v1.0 Framework, dargestellt in Abbildung 1.1 (Kapitel 1.2), zeigt die Abbildung ein um mehrere Komponenten und Kommunikationpfade erweitertes Framework. Die einzelnen, neu in das Framework integrierten Komponenten und deren Schnittstellen werden in Abschnitt 3.2.3 vorgestellt. Ein Blick auf deren Implementierung und

[3] Die Funktion, welche durch den Thread ausgeführt wird

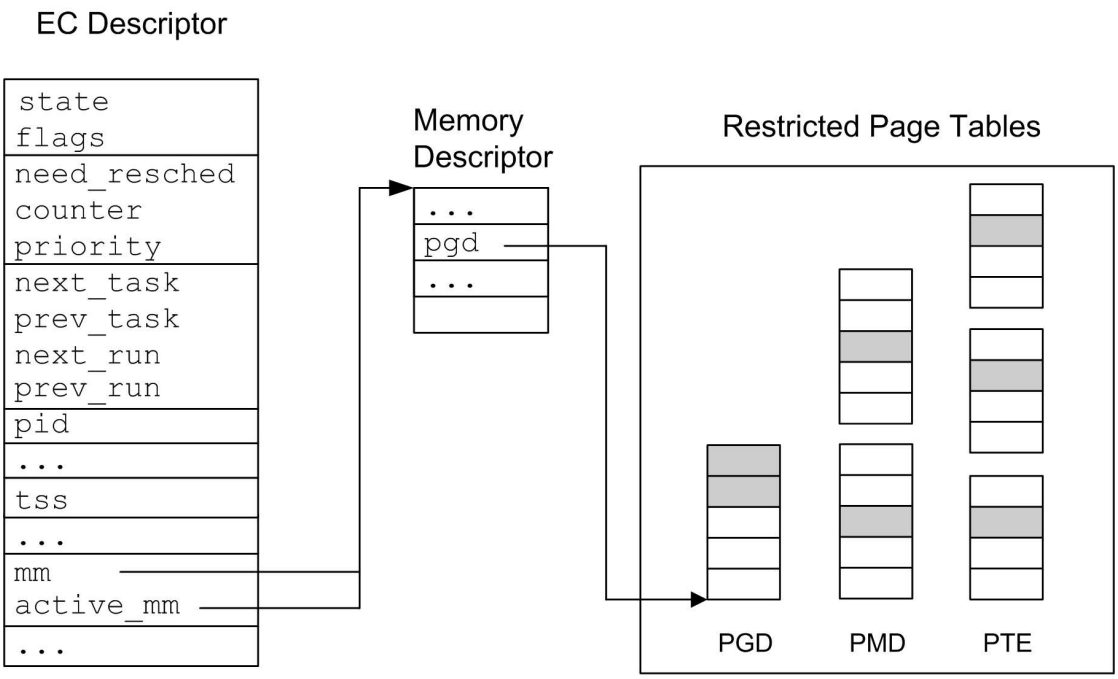

Abbildung 3.3: Process Descriptor, Memory Descriptor und PGD eines Plugins

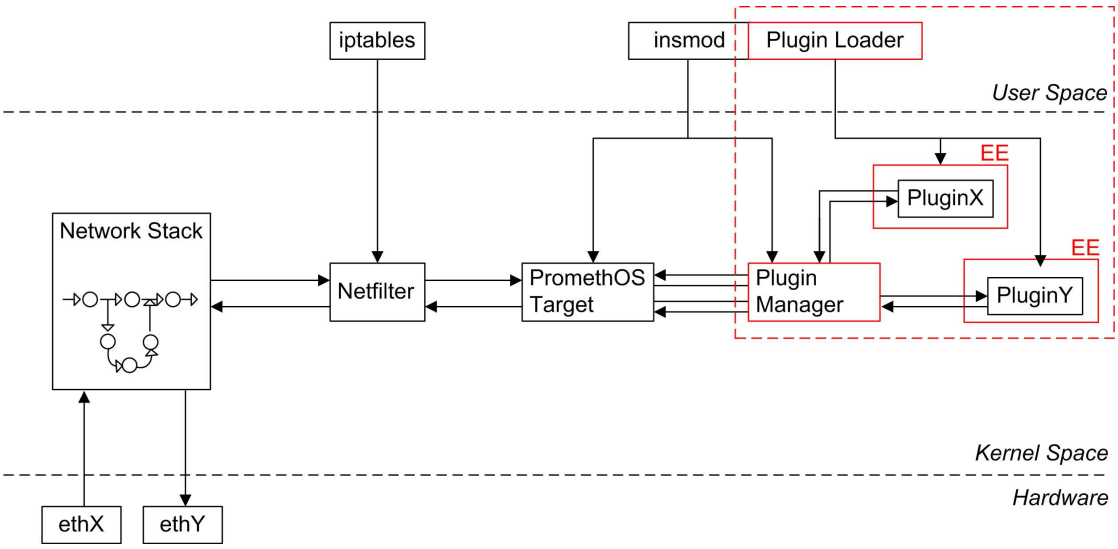

Abbildung 3.4: PromethOS v2.0 Architektur

Interaktion erfolgt in Abschnitt 3.3.

3.2.3 Vorstellung der Komponenten

- *Plugin Manager*
  Der *Plugin Manager* ist die zentrale Komponente der *Memory Resource Control Architektur*. Seine Funktionalität kann in zwei Teilfunktionalitäten unterteilt werden:

  – *Steuerung*
  Der Plugin Manager übernimmt die Steuerung der Plugins. Plugin Code wird - um dessen Speicherzugriffe kontrollieren zu können - innerhalb einer Protection Domain ausgeführt. Das EE, welches diese Protection Domain bereitstellt, ist ein spezieller, in einer dem Abschnitt 3.2.1 entsprechenden Weise manipulierter, Kernel Thread. Solche Kernel Threads werden im Folgenden *Plugin Threads* genannt. Wird nun

ein Plugin installiert, generiert der Plugin Manager zunächst einmal eine solche, auf autorisierte Speicherzugriffe beschränkte EE für dieses Plugin. Der Status der Plugin Threads wird auf STOPPED gesetzt. Dadurch kann ein unnötiger Verbrauch von CPU Zyklen durch gerade inaktive Plugins - Plugins die keine Pakete verarbeiten aber dennoch periodisch vom Scheduler ausgeführt würden - vermieden werden.

Sobald nun ein für das Plugin bestimmtes Paket eintrifft muss das Plugin gestartet werden. Signalisiert ein Plugin das Ende der Paketbearbeitung, sollte das Plugin gestoppt und ein eventuell von diesem zu sendendes Paket in das Netfilter-Framework reinjiziert werden. Wird das Plugin wieder deinstalliert, entfernt der Plugin Manager schlussendlich die Strukturen des Plugin-EEs aus dem Hauptspeicher.

- *Kommunikation*
  Der Plugin Manager dient als Kommunikationsschnittstelle zwischen dem Netfilter-Framework (bzw. dem in Abschnitt 2.1 vorgestellten PromethOS Target) und den Plugins. Diese Funktion bezieht sich sowohl auf den Austausch von Kontroll-Informationen als auch auf den Austausch der eigentlichen Daten, also der IP-Pakete, die für ein Plugin bestimmt sind (oder von diesem gesendet werden).

Der Plugin Manager wurde als Kernel Thread implementiert. Das hat den Vorteil, dass dieser periodisch vom Scheduler aufgerufen wird und somit gewährleistet ist, dass er auf Situationen, beispielsweise auf ein Fehlverhalten eines Plugins, reagieren kann.[4] Auch ist es dadurch möglich, zyklisch ein Polling von gestoppten Plugins zu initiieren, um diesen die Möglichkeit einer Signalisierung zu geben.[5]

- *Execution Environment*
  Das EE ist die Umgebung, in welcher der Plugin Code ausgeführt wird. Dieses EE erlaubt die Ausführung von Code in einer Protection Domain.

- *Plugin Loader*
  Plugins werden als Kernel Module installiert. Jedoch wird der durch das Modul bereitgestellte Service- oder Initialisierungs-Code nicht[6] während des Installationsvorgangs ausgeführt. Vielmehr muss der Plugin Loader den Plugin-Code wie ein normales Kernel-Modul in den Kernel Memory Space laden[7], darf diesen aber nicht ausführen. Anstatt dessen stellt er dem Plugin Manager Informationen über das Modul bereit, welche es diesem wiederum ermöglichen, eine individuell auf das Plugin abgestimmte sichere EE zu generieren, in welcher das Plugin fortan ausgeführt und kontrolliert werden kann.

- *Plugin*
  Plugins enhalten den eigentlichen Service Code, durch welchen die Paketdaten bearbeitet werden. Sie werden zwar fortan als Plugin Threads ausgeführt, allerdings ist dies für deren Implementierung - bis auf die Notwendigkeit der Einhaltung einiger Regeln bezüglich Signalisierung und Paket-Handling - transparent, muss also bei deren Entwicklung nicht berücksichtigt werden.

- *Interrupt- und Exception Handler*
  Diese Komponente ist in Abbildung 3.4 nicht dargestellt, da sie für ein Grundverständnis der Architektur nicht relevant ist. Dennoch hat diese Komponente sozusagen “Aufgaben im Hintergrund” zu erfüllen, ohne die die dargestellte Architektur nicht implementierbar ist.

  Nach E12, E13[8] sind aus folgendem Grunde Anpassungen am Linux Interrupt- und Exception-Handling notwendig: Das Handling von Interrupts und Exceptions findet nach E12 innerhalb des aktiven Kontexts statt; die presenten Page Tables werden für das Mapping verwendet. Tritt nun ein Interrupt oder eine Exception auf, während gerade ein PromethOS Plugin Thread aktiv ist, wird auf einen Speicherbereich zugegriffen

[4] Dieses beispielsweise beendet
[5] Das Polling besteht hier aus dem kurzzeitigen Ausführen der gestoppten Plugins
[6] Wie dies bei Standard-Kernel-Modulen der Fall ist
[7] In einer Form, in der dieser Code ausgeführt werden kann
[8] Erkenntnisse der Sektion 3.1

- nämlich auf den der *IDT* und der ensprechenden *ISR* - der von einem Plugin aus Sicherheits- und Stabilitätsgründen nicht gemappt werden darf. Die Routine kann nicht ausgeführt werden; das System ist nicht mehr lauffähig. Ein funktionierendes Interrupt- und Exception-Handling ist Vorraussetzung für die Lauffähigkeit des kompletten Systems.[9] Einen Ausweg aus diesem Dilemma schafft ein erweiterer Interrupt-Handler, der dieses Problem - wie in Kapitel 3.3.5 beschrieben - durch spezielles Handling der Plugin-Kontexte löst.

Aus folgendem Grunde ist desweiteren der Linux Page Fault Handler anzupassen: Adressiert ein Plugin eine nicht-autorisierte (und daher nicht “gemappte”) virtuelle Speicheradresse, wird dies von der MMU als ein Fehler interpretiert, für dessen Behandlung der Page Fault Handler aufgerufen wird.

Der Page Fault Handler reagiert hierauf, wie bei der detaillierten Betrachtung des Exception-Handlers in Sektion 3.3.5 beschrieben, mit einem *Kernel “Oops”*[10]. Das Verhalten des Page Fault Handlers muss nun in so fern angepasst werden, dass die Systemstablilität bei einer Deadressierung eines Plugins nicht gefährdet wird.

3.3 Implementierung der Komponenten

In dieser Sektion wird die Implementierung der Komponenten der in Abbildung 3.4 veranschaulichten Zielarchitektur dargestellt. Hierbei wurde ein Augenmerk darauf gelegt, die Komponenten möglichst stark zu kapseln, um das Framework so Modular wie möglich aufzubauen.

3.3.1 Plugin-Manager

Im vorhergehenden Kapitel wurde die Funktionalität des Plugin Managers in zwei Teilfunktionalitäten untergliedert. Dies war zum einen die Steuerung und Kontrolle der Plugins, zum anderen war das die Schnittstellenfunktion des Plugin Managers, durch welche die Plugins in das Netfilter-Framework integriert werden können.

Es wird nun zunächst die Steuerungs- und Kontrollfunktionalität des *Plugin Managers* betrachtet. Hierzu gehört zum einen das “sichere” Laden eines zu installierenden *Plugins* in den *Kernel Space* sowie die Generierung einer EE, in welcher dessen Code ausgeführt werden kann. Zum anderen muss ein Plugin Thread, in dessen Kontext der Plugin Code ausgeführt wird, bei einem ankommenden Paket gestartet, und nach dessen Verarbeitung wieder gestoppt werden.

Da der Ladevorgang des Pluginmoduls in das Kernel Memory Space in Kooperation mit dem Plugin Loader stattfindet, wird, um Redundanz zu vermeiden, der gesamte Installationsvorgang eines Plugins in der Sektion *3.3.3* beschrieben. Dies beinhaltet auch die Darstellung des hierin bezüglich der Koordination involvierten Codes des Plugin Managers. Die Generierung der EE, welche die Plugin Protection Domain bereitstellt, der Plugin Thread, wird in *Sektion 3.3.2* im Detail betrachtet.

Die zweite Funktion des Plugin Managers ist diejenige der Kommunikationsschnittstelle zwischen dem Netfilter-Framework (bzw. dem in Abschnitt 2.1 vorgestellten PromethOS Target) und den Plugins.

Während die Kommunikation zwischen Plugin-Manager und den Plugins in der *Sektion 3.3.4* beschrieben wird, zeigt der *Abschnitt 3.4*, wie der Plugin Manager[11] mit dem *Netfilter-Framework* interagiert.

[9] Ohne die Behandlung von Timer-Interrupts wäre beispielsweise kein Scheduling möglich
[10] Schwerwiegender Fehler im Kernel. Entspricht einem Crash des Betriebssystems
[11] Und über diesen letztendlich die Plugins

Der Plugin Manager wurde als Kernel Thread in der Datei

```
net/ipv4/netfilter/ipt_PROMETHOS.c
```

implementiert. Diese Datei ist das Kernstück sowohl des PromethOS v1.0 als auch des PromethOS v2.0 Frameworks. Hier sind beispielsweise das PromethOS v1.0 Target sowie sämtliche Registrier-, Konfigurations- und sonstige Verwaltungsfunktionen implementiert.

Der Code dieser Datei wird dynamisch als Modul in den Kernel gelinkt. In der Initialisierungsmethode des Moduls wird zum einen das *proc file* registriert, über welches die *IPC* mit dem *Plugin Loader* realisiert wird, zum anderen wird dort der Plugin Manager Thread gestartet:

```
start_kthread(promethos_plugin_manager_thread, \
              &kthread_t_container[0])
```

Die Funktion, die als Kernel Thread ausgeführt wird, stellt die Implementierung des Plugin Managers dar:

```
promethos_plugin_manager_thread()
```

Sobald der Plugin Manager aktiv ist, führt dieser eine zyklische Überprüfung verschiedener Flags durch, auf die er entsprechend den Zuständen der Flags reagiert. Dieses wird im Folgenden betrachtet:

1. `is_plugin_loader_waiting(promethos_procfile_data.m_flag)`

   Falls der Wahrheitswert des Rückgabewertes dieser Funktion dem Wert `true` entspricht, wird dies als ein Signal des Plugin Loaders interpretiert, dass dieser ein neues Plugin installiert (der Plugin Manager kann eine Installation neuer Plugins dadurch unterbinden, dass er das Flag `promethos_procfile_data.m_flag` selbst auf einen Wert größer `0` setzt). Der Plugin Manager generiert nun einen Execution Context inkl. Protection Domain, in welcher der gerade installierte Plugin Code ausgeführt werden kann. Dieses wird von diesem durch den Aufruf folgender Funktion initiiert:

   ```
   plugin_thread_install()
   ```

   Sobald das EE des Plugins erstellt wurde, aktiviert der Plugin Manager den Plugin Thread durch den Aufruf der Funktion

   ```
   plugin_thread_wake_up(plugin_pid)
   ```

   Sobald das Plugin das erste mal von dem Scheduler zur Ausführung gebracht wird, registriert sich dieses, wie in Abschnitt 3.3.2 beschrieben, bei dem PromethOS Framework. Durch den Aufruf der Funktion:

   ```
   plugin_thread_lull_to_sleep(plugin_pid)
   ```

   wird nun das Plugin aus der `RUNQUEUE`[12] wieder enfernt; es ruht nun solange, bis ein Paket für dieses bereitsteht und es für dessen Bearbeitung von Plugin Manager wieder aktiviert wird. Der letzte Schritt, den der Plugin Manager im Zuge der Plugin-Installation nun durchführt, ist die Registrierung des Plugins bei dem Netfilter-Framework (bzw. bei dem PromethOS Target). Auf diesem Wege kann, wie unter 3.4 beschrieben, das *PromethOS v1.0 Target* als Schnittstelle zum *Netfilter Framework* genutzt werden:

   ```
   promethos_register(...)
   ```

[12] Liste der vom Scheduler berücksichtigten, ausführungsbereiten ECs

2. `status_flag_netfilter`

   Falls der Wert dieser Variable `1` ist, wurde dem Plugin Manager von Netfilter ein Paket übergeben, welches von dessen `standard-target` in die Protection Domain des zuständigen Plugins kopiert wurde. Damit das entsprechende Plugin nun das Paket bearbeiten kann, führt der Plugin Manager folgende Befehlssequenz aus:

   ```
   ptr_promethos_ipc_data->status_flag_plugin_active=1;
   ptr_promethos_ipc_data->status_flag_plugin_new_pkt=1;
   if (!plugin_thread_runnig){
          plugin_thread_wake_up(plugin_pid);
          plugin_thread_runnig=1;
   }
   status_flag_netfilter=0;
   ```

   In Zeile 1 wird dem Plugin Thread signalisiert, dass dieser von dem Plugin Manager (re)aktiviert wurde. Zeile 2 zeigt dem Plugin an, dass ihm ein neues Paket übergeben wurde. Die Zeilen 3-5 starten den Plugin Thread, falls dieser zuvor von dem Plugin Manager gestoppt wurde. Die letzte Zeile signalisiert dem Netfilter Framework (bzw. dem Plugin Manager Standard Target) die Bereitschaft des Plugin Managers, weitere Pakete entgegenzunehmen.

3. `ptr_promethos_ipc_data->status_flag_plugin_nr_ip_pkts_to_send`

   Falls dieser Wert ungleich `0` ist, signalisiert das Plugin hierdurch ein zu sendendes Paket. Der Plugin Manager reagiert hierauf mit dem Aufruf der Funktion:

   ```
   promethos_v2_build_skb(temp_skb, ptr_promethos_ipc_data)
   ```

   Dieser Funktion wird als Parameter ein Pointer auf einen Socket Buffer sowie der Pointer auf die IPC-Strukur des Plugins übergeben, welche das Paket bereithält. In einer späteren Version wird hier vor dem Aufruf dieser Funktion eine `sk_buff-Struktur` alloziert, welche dann für das reinjizieren des Pakets in den Netzwerk-Stack Verwendung findet. Im Rahmen dieser Diplomarbeit wird an dieser Stelle die Adresse des Socket Buffers des letzten angenommenen Pakets verwendet. Dies hat den Vorteil, dass Felder der Struktur bereits initialisiert sind und somit eine diesbezügliche mögliche Fehlerquelle entfällt. Diese Zuweisung des Pointers `temp_sk_buff` erfolgte zuvor in dem PromethOS-v2-Strandard-Target.

   Die Funktion kopiert nun das Paket des Plugins, welches sich in dem Feld `ip_pkt_from_plugin` der IPC-Datenstruktur befindet in denjenigen Hauptspeicherbereich, welcher von `temp_skb` adressiert wird. Die Bytes dieses Hauptspeicherbereichs werden zuvor auf den Wert `0` initialisiert:

   ```
   memset(skb->nh.iph, 0, \
          (unsigned long)iph_pkt_out->tot_len);
   memcpy(skb->nh.iph, &ptr_promethos_ipc_data-> \
          ip_pkt_from_plugin, \
          (unsigned long)iph_pkt_out->tot_len);
   ```

   Es sollte nun eine Anpassung einiger Felder in der Socketbufferstruktur erfolgen. Beispielsweise ist eventuell der Pointer auf den Layer-3-Header anzupassen oder die länge des Datenpakets zu korrigieren. In dieser Diplomarbeit hatten Pakete, die von den Plugins zum senden bereitsgestellt wurden, jeweils die gleichen Grösen wie das angekommen Paket, sodass hier keine Anpassungen notwendig waren. Dies ist dann in einem zweiten Schritt zu implementieren.

   Als nächstes wird die Route des ausgehenden Pakets sowie seine neue Checksumme berechnet und das Paket gesendet:

```
route_mirror(skb);
update_checksum(skb);
ip_direct_send(skb);
```

Diese Funktionssequenz funktioniert in dem Falle, dass das Paket wieder an den Sender des Pakets, welches ursprünglich von dem Socket Buffer referenziert wurde, zurückgesendet wird. Für einen späteren Einsatz des Systems muss diese Beschränkung durch eine Subsitution obiger Funktionen aufgehoben werden. Nach dem die Funkton ausgeführt - und das Paket gesendet - wurde, wird nun noch der Wert der Variable `status_flag_plugin_nr_ip_pkts_to_send` der Plugin-IPC-Stuktur zurück auf `0` gesetzt. Hiermit wird dem Plugin signalisiert, dass das Paket abgeholt und gesendet wurde.

4. `(ptr_promethos_ipc_data->status_flag_plugin_active==0) && (plugin_thread_runnig)`

   Diese Kombination trifft auf ein Plugin zu, welches zwar noch “gescheduled” wird, allerdings keine Aufgaben mehr zu erledigen hat. Um CPU-Zyklen zu sparen wird dieser Plugin Thread nun gestoppt.

   ```
   plugin_thread_lull_to_sleep(plugin_pid); plugin_thread_runnig=0;
   ```

Am Ende jeder Iteration ruft der Plugin Manager jeweils den Scheduler auf und gibt somit die CPU an einen anderen EC, i.d.R. an einen Plugin Thread, ab. Dieses Vorgehen verkürzt die Latenz innerhalb des Systems, da beispielsweise ein Plugin nach einer Paketübergabe an dieses ansonsten frühestens nach dem nächsten Timerinterrupt gescheduled werden könnte, selbst wenn der Plugin Manager bis dahin keine Aufgaben mehr durchführen würde.

3.3.2 Execution Environment und Protection Domain

Die Generierung einer EE, welche die Plugin Protection Domain bereitstellt, wird mittels eines speziellen Kernel Threads (Plugin Thread) realisiert, dessen Mapping anhand individueller *Restricted Page Tables* vorgenommen wird. Das prinzipielle Vorgehen wurde bereits bei der Schilderung des Lösungsansatzes in Sektion 3.2.1 vorgestellt. *Abbildung 3.2* illustriert diese Methode. Da die Generierung eines solchen Execution Environments im Kernel Space den komplexesten Teil der Architektur darstellt, wird die Implementierung des diesbezüglichen Vorgehens nachfolgend ausführlich dargestellt.

Execution Environment

Der Code der Funktionen, mit deren Hilfe dieses Vorgehen implementiert wird, befindet sich in der Datei `net/ipv4/netfilter/ipt_PROMETHOS.c`, in welcher auch der Plugin Manager implementiert wird. Im Folgenden wird nun ein Blick auf diese Implementierung geworfen:

Nachdem ein Plugin Modul - wie in Sektion 3.3.3 dargestellt - in den Kernel Memory Space geladen wurde, initiiert der Plugin Manager die Generierung eines EE für den Plugin Code. Hierzu ruft er folgende Funktion auf, welche die im weiteren Verlauf beschriebenen Schritte ausführt:

```
plugin_thread_install( \
      &kthread_t_container[kthread_t_plugin_container_index])
```

Der Funktion wird ein Pointer auf eine Struktur des in der Header-Datei des Plugin Managers, `linux/promethos_plugin_manager.h`, definierten Typs `kthread_struct` übergeben.[13]

```
typedef struct kthread_struct
{
```

[13] Es wurde zuvor ein Array solcher Strukturen, `kthread_t_container`, defi niert, um mehrere Threads verwalten zu können. `kthread_t_plugin_container_index dient als Index innerhalb dieses Arrays.`

```
        struct task_struct *thread;
        struct tq_struct tq;
        void (*function) (struct kthread_struct *kthread);
        struct semaphore startstop_sem;

        wait_queue_head_t queue;
        int terminate;
        void *arg;
} kthread_t;
```

Diese Struktur stellt Informationen über den zu generierenden Kernel Thread bereit, welche zu dessen Erstellung und Verwaltung benötigt werden. So zeigt das Feld `thread` beispielsweise auf dessen *EC Descriptor*; `function` ist ein Pointer auf diejenige Funktion, die als Kernel Thread ausgeführt werden soll.

Die Funktion `plugin_thread_install()` hat im wesentlichen zwei Aufgaben. Zum einen wird hierin die Funktion `start_plugin_kthread()` aufgerufen, zum anderen wird, nachdem der Plugin Thread durch diese generiert wurde, dem Plugin Loader die (Wieder-)Bereitschaft des Plugin Managers bezüglich der Installation eines weiteren Plugins signalisiert (dieser Sachverhalt ist in 3.3.3 im Detail beschrieben).

```
promethos_procfile_data.m_flag=0;
```

Der Funktion `start_plugin_kthread` werden zwei Parameter übergeben. Zum einen ist dies der Pointer auf die Funktion, welche von dem Thread ausgeführt werden soll, zum anderen ist das der Wert des Pointers auf den Typ `kthread_struct`, welcher bereits der Funktion `plugin_thread_install` übergeben wurde und nun an diese Funktion “durchgereicht” wird.

In der Funktion `start_plugin_kthread()` werden die Felder der übergebenen `kthread_struct` initialisert und die Ausführung der Funktion `kthread_plugin_launcher` veranlasst. Unter anderem wird hier die Initialisierungsfunktion des in den Kernel geladenen Plugin Moduls als die von dem Thread auszuführende Funktion festgelegt:

```
kthread->function=func;
```

`kthread` zeigt hierbei auf die übergebene Adresse der `kthread_struct`-Struktur;
`func` ist ein Pointer auf eine Funktion, welchem per Parameterübergabe der Wert `promethos_procfile_data.m_entry_point` zugewiesen wurde.[14]

Nach der Initialisierung der Felder wird der Plugin Launcher zur Ausführung gebracht:

```
kthread->tq.routine = kthread_plugin_launcher;
kthread->tq.data = kthread;
schedule_task(&kthread->tq);
```

Dieser generiert nun letztendlich den Kernel Thread durch die Ausführung der Assembler-Funktion:

```
promethos_v2_plugin_kernel_thread()
```

Der Code dieser Assembler-Funktion befindet sich in der Datei `arch/i386/kernel/process.c`.[15] Sie liefert die *Process ID* des generierten Kernel Threads zurück, durch welche der Thread fortan eindeutig identifiziert werden kann.[16]

Nachdem nun der Kernel Thread generiert wurde, wird diesem ein eigener *Memory Descriptor* inkl. Restricted Page Tables zugewiesen.

[14] Entspricht der Adresse der `init_module() Funktion des Plugin Threads`
[15] Die Implementierung der Assemblerfunktion ist hardwareabhängig
[16] Gilt nur auf Uniprozessorsystemen, da jeder Prozessor seine eigenen PIDs vergibt.

```
make_promethos_plugin_page_tables(find_task_by_pid(plugin_pid))
```

Die Erstellung der Protection Domain, die durch diese Funktion implementiert wird, wird im Folgenden betrachtet. Im Anschluss hieran wird der gerade erzeugte Plugin Thread einmalig gescheduled, damit sich dieser bei dem PromethOS Plugin Manager registrieren kann (Die Abläufe nach der Registrierung werden in Sektion 3.3.4 dargestellt).

Protection Domain

Die Bildung einer Protection Domain besteht im Kern aus der Generierung eines Page Tables Sets, welches nur die für das Plugin erlaubten und notwendigen virtuellen Speicheradressen auf physikalischen Adressen abbildet. Die Basisadresse des PGD der generierten Page Tables wird nun in das Feld eines neu allozierten Memory Descriptors eingetragen, dessen Adresse wiederum in das mm Feld des Process Descriptors des Kernel Threads eingetragen wird, welcher den Plugin Code ausführen wird. Im vorhergenden Abschnitt wurde beschrieben, wie der Erstellungvorgang einer Plugin-EE implementiert ist. Durch den Aufruf der Funktion

```
make_promethos_plugin_page_tables(find_task_by_pid(plugin_pid))
```

wird diesem eine Protection Domain zugewiesen. Die Funktion nimmt als Parameter die Adresse der Process Descriptors des Kernel Threads entgegen, dessen mm-Feld die Adresse des erstellte Memory Descriptor zugewiesen wird.

Die Implementierung dieser Funktion sowie deren Hilfsfunktionen werden nun im Folgenden vorgestellt. Die Bildung der Protection Domains wird in zwei Schritten vollzogen:

1. Allokation und Initialisierung der benötigen Strukturen
   In diesem ersten Schritt werden alle Strukturen angelegt, die benötigt werden, um eine Protection Domain aufzubauen. Dies sind sowohl der Memory Descriptor als auch die Page Tables. Implementiert ist dies wie folgt in der Funktion:

   ```
   make_promethos_plugin_page_tables(...)
   ```

   Zu Beginn wird ein neuer Memory Descriptor alloziert. `new_mm` zeigt hierauf

   ```
   if (!(new_mm =mm_alloc())) {return -ENOMEM;}
   ```

   Als nächstes wird eine Speicherseite für das PGD des zu generierenden Page Table Sets alloziert; `new_pgd` zeigt auf die PGDBA dieses PGD

   ```
   if (!(new_pgd = (pgd_t *)pgd_alloc())) {return -ENOMEM;}
   ```

   Die Adresse des PGD wird nun dem `pgd`-Feld des *Memory Descriptors* zugewiesen; Einträge des PGD werden mit `0` (`NOT_PRESENT`) initialisiert.

   ```
   new_mm->pgd=new_pgd;
   memset(new_pgd, 0, PTRS_PER_PGD * sizeof(pgd_t));
   ```

   Nachdem das PGD angelegt und initialisiert wurde, werden nun die PTE-Speicherseiten alloziert, die das Mapping der Kernel Virtual Memory Space Adressen bestimmen.[17] Die Adressen dieser Seiten werden in das zuvor allozierte PGD eingetragen. Es ist zu beachten, dass bei einen 3-Level-Paging an dieser Stelle vor diesem Schritt noch PMD-Speicherseiten alloziert werden müssen. Dieser Schritt entfällt bei einer 2-Level-Paging-Architektur wie der für die Implementierung gewählten IA32.

   ```
   for (i=USER_PTRS_PER_PGD; i<PTRS_PER_PGD; i++){
          new_pte = (pte_t *)__get_free_page(GFP_KERNEL);
          memset(new_pte, 0, PTRS_PER_PTE * sizeof(pte_t));
          memset(new_pgd+i,new_pte, sizeof(pgd_t));
   }
   ```

[17] USER_PTRS_PER_PGD liefert ein Index auf den ersten Eintrag des Kernelspeicherbereichs

In die soeben allozierten und auf `NOT_PRESENT` initialisierten PTE-Speicherseiten werden nun die Speicherbereiche aus den Page Tables des Kernels kopiert, die dem Plugin Thread zugänglich gemacht werden sollen. So ist das Adressintervall des Plugin Moduls freizuschalten sowie die Funktion `promethos_v2_register`, damit sich das Plugin bei dem Plugin Manager registrieren kann. Weiter ist die IDT sowie der 8KB große Speicherbereich zu mappen, welcher den *Process Descriptor* sowie den Stack des *Plugin Threads* beinhaltet.

Adressintervalle, welche durch *Restricted Page Tables* gemappt werden sollen, werden der Funktion `extend_promethos_plugin_page_tables(...)` in Form von Start- und Endadresspaaren als Parameter übergeben. Weiter erwartet die Funktion die Adresse des Process Descriptors des Plugin Threads, dessen Page Tables um dieses Intervall erweitert werden sollen. Das Mapping des Modulspeicherbereichs wird beispielsweise durch folgenden Aufruf veranlasst:

```
extend_promethos_plugin_page_tables(
         tsk, \
         promethos_procfile_data.m_address, \
         promethos_procfile_data.m_address + \
         promethos_procfile_data.m_size)
```

Wobei `tsk` ein Pointer auf den Process Descriptor des Kernel Threads darstellt und die beiden letzten Parameter die Anfangs- und die Endadresse des Plugin Module Codes angeben.

Eine Darstellung der Implementation der Funktion

```
extend_promethos_plugin_page_tables()
```

erfolgt weiter unten in diesem Abschnitt. Um dem Plugin beispielsweise bestimmte Kernelsymbole zur Verfügung zu stellen, muss lediglich die Funktion ein weiteres mal aufgerufen werden und dieser das entsprechenden Adressintervall als Start- und Endadresspaar übergeben werden.

Die Restricted Page Tables wurden kreiert und werden von dem neuen Memory Descriptor verwendet. Als letzter Schritt wird nun dieser Memory Descriptor dem mm Feld des Process Descriptor des Kernel Threads zugewiesen.

```
tsk->mm=new_mm; tsk->active_mm=new_mm;
```

2. Mapping autorisierter Speicherbereiche

```
extend_promethos_plugin_page_tables(struct task_struct *tsk,
                                    unsigned long mem_start,
                                    unsigned long mem_end)
```

Diese Funktion implementiert das Mapping des ihr als Parameterpaar übergebenen Adressinterveralls, indem die dieses Intervall abbildenden PTE-Einträge der PTE-Page-Tables des Kernels in die PTE-Page-Tables des Plugin Threads kopiert werden.

Die Parameter `mem_start` und `mem_end` sind Pointer auf die Start- und Zieladresse des Intervalls. `tsk` zeigt auf den Process Descriptor des Plugin Threads, dessen *Restricted Pages Tables* erweitert werden sollen.

Das Vorgehen zur Definition des Quell- und Zielspeicherbereichs bezüglich des durchzuführenden Kopiervorgang wird nun beschrieben. Zuerst wird die PGDBA des Kernel Threads ermittelt, sodass diese über den Pointer `plugin_pgd` referenziert werden kann. Die PGDBA des Kernels selbst wird ebenfalls ermittelt und kann fortan über den Pointer

`page_dir` referenziert werden.

```
plugin_pgd=tsk->mm->pgd;
page_dir=pgd_offset(&init_mm, 0);
```

Nun werden über die in der Datei `include/asm/pgtable.h` definierten Standard APIs zur Manipulation und Traversierung von Page Tables die Offsets und Indizes der Start- und Endadresse in den Page Tables des Kernels und des Plugins[18] ermittelt. Während als Offsets die Adressen der so ermittelten Einträge bezeichnet werden, bestimmt ein Index die Nummer des Eintrags in der entsprechenden Table. [Gat02]

Da im Rahmen diese Studienarbeit ein 2-Level-Paging Verwendung fand, können nach der Ermittlung der Indizes und Offsets der Start- und Endadresse in der PGD und der PTE(s)[19] 3 Fälle unterschieden werden:

(a) Start- und Endadresse werden über die gleiche PTE-Page-Table gemappt:

`(pmd_offset_mem_start->pmd==pmd_offset_mem_end->pmd)`

`pmd_offset_mem_start->pmd` ist hierbei ein Pointer auf die über die Standard-Kernel-APIs ermittelte Adresse der PTE-Speicherseite, welche die Startadresse mappt. Entprechend enthält `pmd_offset_mem_end->pmd` die PTE-Speicherseite, über welche das Mapping der Endadresse vorgenommen wird. Das Erweitern der *Restricted Page Tables* kann hier relativ unkompliziert vollzogen werden

```
memcpy(pte_offset_mem_start_plugin,
       pte_offset_mem_start,
       ((pte_offset_mem_end-pte_offset_mem_start)+1) * \
         sizeof(pte_t));
```

`pte_offset_mem_start` und `pte_offset_mem_end` sind zuvor ermittelte Adressen der PTE Einträge in den Kernel-PTE-Tables. `pte_offset_mem_start_plugin` ist die Adresse des PTE Eintrags in einer Plugin PTE, die die Startadresse mappt. Nun wird der Speicherbereich in der Kernel PTE von der Start- bis zur Endadresse in die korrespondierenden Plugin PTE kopiert.

(b) Start-und Endadresse werden über PTE-Page-Tables gemappt, welche von aufeinanderfolgenden PMD-Einträgen referenziert werden:

`(pmd_offset_mem_start->pmd)==((pmd_offset_mem_end->pmd)-1)`

Da von aufeinanderfolgenden PMD-Einträgen referenzierte PTE-Seiten[20] nicht notwendigerweise aufeinanderfolgend im Speicher angeordnet sein müssen, kann hier im Gegensatz zu Fall a. nicht einfach der Speicherbereich zwischen `pmd_offset_mem_start->pmd` und `pmd_offset_end_start->pmd` kopiert werden. Es müssen hier die gültigen Einträge beider PTE-Seiten separat kopiert werden.[21]

```
memcpy(pte_offset_mem_start_plugin,
       pte_offset_mem_start,
       (((pte_mem_start_plugin+PTRS_PER_PTE)-  \
 pte_offset_mem_start_plugin)+1) * sizeof(pte_t));

memcpy(pte_mem_end_plugin,
       pte_mem_end ,
 (((pte_offset_mem_end_plugin)- \
 (pte_mem_end_plugin))+1) * sizeof(pte_t));
```

[18] Diejenigen Offsets, die die Adressen später mappen sollen
[19] Welches diese Adressen mappen
[20] Mit PMD==PGD aufgrund des 2-Level-Pagings
[21] Die PTE-Seite, welche die Startseite mappt und die PTE-Seite, welche die Endadresse mappt.

Mit dem ersten Befehl werden die PTE-Einträge der Speicherseite ab dem Offset des Eintrags, welcher die Startadresse mappt, kopiert. Der zweite Befehl kopiert die Einträge ab der Basisadresse der PTE-Seite welche die Endadresse abbildet bis zu dem Offset des Eintrags welcher diese Adresse mappt.

(c) Start-und Endadresse werden über PTE-Page-Tables gemappt, welche nicht von aufeinanderfolgenden PMD-Einträgen referenziert werden:

```
(pmd_offset_mem_start->pmd<((pmd_offset_mem_end->pmd)-1))
```

In diesem Fall wird das Mapping zunächst auf identische Weise wie in Fall b. vorgenommen. Zudem müssen nun aber noch die *Restricted Page Tables* um die intermediären PTE-Pages erweitert werden. Dies wird dadurch implementiert, dass nicht die kompletten PTE-Seiten kopiert werden, sondern die PMD-Einträge der Restricted Page Table, die zwischen denen Offsets der beiden Adressen in der Plugin-PMD liegen, die Adressen der PTE-Seiten zugewiesen bekommen, auf die auch die Kernel PMD-Einträge zeigen. Folgendes Codefragment nimmt diese Zuweisung vor:

```
i=pmd_index_mem_end-pmd_index_mem_start; /* i>=2  */
for (;i>=2;i--){
       memcpy(++pmd_offset_mem_start_plugin, \
       ++pmd_offset_mem_start , sizeof(pmd_t));
}
```

3.3.3 Plugin-Loader

Aufbau und Aufgaben des Plugin-Loaders

Die Rolle des *Plugin Loaders* unterscheidet sich nur in nachfolgend genannten Punkten von der Rolle des Linux-Standardprogramms zum Laden von Kernel Modulen, `insmod`. Daher lag es nahe, für dessen Implementierung die Sourcen von `insmod` und referenzierter Dateien zu verwenden und um die benötigte Funktionalität zu erweitern. Dies entspricht auch den Designkriterien, da ja soweit möglich auf vorhandenem Code und Standardprogrammen, falls nötig durch deren Anpassung, aufgebaut werden sollte.

Das Verhalten des Plugin Loaders unterscheidet sich wie folgt von dem des *Linux Standard Module Loaders* `insmod`:

1. Identifikation eines Plugins, um differenzierte Verfahren bzgl. dem Laden eines Plugins und dem eines Standard-Modules zu ermöglichen.
2. Identifizierte Plugins werden lediglich in den Speicherbereich des Kernels geladen; deren Initialisierungsfunktion wird jedoch nicht ausgeführti
3. Für den Plugin Manager relevante Informationen über das Plugin werden diesem über ein eigens hierfür kreiertes *proc-file* kommuniziert.
4. Plugins werden nur dann installiert, wenn der Plugin Manager eine solche Installation akzeptiert. Hierfür wird ein entsprechendes Status-Flag abgefragt, welches ebenfalls über das vorangehend genannte *proc-file* kommuniziert wird.

Der Plugin Loader wurde nun dadurch implementiert, dass der Source-Code des *User Space Tools* `insmod`[22] sowie die von diesem referenzierte Funktion `sys_init_module()` in der Datei `kernel/module.c` erweitert wurden. Im weiteren Text wird diese Implementation im Detail erläutert:

Mit der Einbindung des Header-Files `include/linux/promethos_v2_plugin_loader.h` sowohl in `insmod.c` als auch in `kernel/module.c`, steht dem *Plugin Loader* die Struktur zur Verfügung, welche zur *IPC* zwischen ihm und dem *Plugin Manager* verwendet wird. Der *Plugin Manager* bindet das genannte Header-File ebenfalls ein. Somit ist gewährleistet, dass

[22] Bestandteil des *Modutils-Packages*, `http://www.kernel.org/pub/linux/utils/kernel/modutils/`

alle involvierten Komponenten für die Kommunikation Datenstrukturen des gleichen Typs und mit gleichen Namen verwenden. Die Inhalte dieser Struktur werden zwischen beiden Komponenten per Lese- und Schreibaktionen auf ein *proc-file* kommuniziert.[23] Der Name dieses Files wird ebenfalls als Inhalt der Variable `promethos_proc_filename` von obigem Header-File bekanntgegeben.[24] Die genannte Struktur ist vom Typ `promethos_procfile_data_t` und beinhaltet sämtliche Informationen, die zwischen den beiden Komponenten ausgetauscht werden.

```
struct promethos_procfile_data_t {
      char m_name[64];
      unsigned long m_address;
      unsigned long m_size;
      unsigned long m_entry_point;
      int m_flag;
}promethos_procfile_data;
```

Dies sind zum einen Informationen über das zu installierende Plugin selbst, wie dessen Name `m_name`, Startadresse `m_address` und Größe `m_size` sowie die Adresse `long m_entry_point` der `init_module()`-Funktion des Moduls. Diese Informationen werden von dem Plugin Manager, wie unter Sektion 3.3.1 beschrieben, für die Generierung des Pluging-EEs und die Verwaltung des Plugins-Threads[25] benötigt. Sie werden bei einem Module-Linking-Vorgang von `insmod` ermittelt und werden hier noch den richtigen Feldern in `promethos_procfile_data` zugewiesen (c.f. [BoCe00], Appendix B.3).

Bevor der *Plugin Loader* das Modul in den Kernel lädt und dem *Plugin Manager* die genannten Informationen übermittelt, muss zunächst geprüft werden, ob der Plugin Manager ein neues Plugin akzeptiert. Der Plugin Manager akzeptiert kein neues Plugin, falls er gerade mit der Installation oder Initialisierung eines anderes Plugin beschäftigt ist; ein Verweigern kann aber auch bei allgemeinen Überlastsituationen sinnvoll sein. Zu einer diesbezüglichen Koordination dient das letzte Feld der Struktur: `m_flag`.

Der Ablauf der Plugin Installation

Nun wird im Folgenden der genaue Ablauf der Plugin-Installation beschrieben. Abbildung 3.5 veranschaulicht den Installationsvorgang und die Abstimmung zwischen *Plugin Manager* und *Plugin Loader*.

Soll ein neues Plugin in das PromethOS-Framework installiert werden, so wird hierfür - wie bei dem Laden eines “normalen” Kernel Modules auch - das Kommando ’`/sbin/insmod MODULNAME.o`’ ausgeführt.[26]

Der Linking-Vorgang läuft nun bis zu dem Punkt auf einer dem Ladevorgang eines “normalen” Kernel Moduls entsprechenden Weise ab, bis die vom *Plugin Manager* benötigeten Werte ermittelt und der Struktur `promethos_procfile_data` zugewiesen werden konnten.[27]. So werden Modulname, benötiger Speicherplatz und die Adresse der `init_module()`-Funktion bestimmt. Auch wird der System-Call `create_module()` aufgerufen, welcher die endgültige Startadresse des Moduls im Kernel Memory-Space zurückgibt.

Nun wird entschieden, ob es sich um ein *PromethOS v2.0 Plugin* handelt. Hierzu wird der Rückgabewert der Funktion `promethos_is_v2_plugin()`[28] überprüft. Diese Funktion gibt den Wert `0` zurück, falls es sich um ein *PromethOS v2 Plugin* handelt, ansonsten wird der Wert `1` zurückgegeben. Folgende Zeile ist hierfür verantwortlich:

[23] Dies an entsprechender Stelle genauer erläutert
[24] Das `proc-file` wird vom *Plugin Manager* zuvor angelegt
[25] Plugin-Thread, da der Plugin Code als Funktion eines spezieller Kernel-Thread ausgeführt wird
[26] Der Parameter MODULENAME.o ist der Dateiname des zu ladenden Plugins
[27] Der genaue Ablauf eines Linking-Vorgangs kann in [BoCe00], Appendix B.3 nachgelesen werden
[28] Defi niert in `promethos_v2_plugin_loader.h`

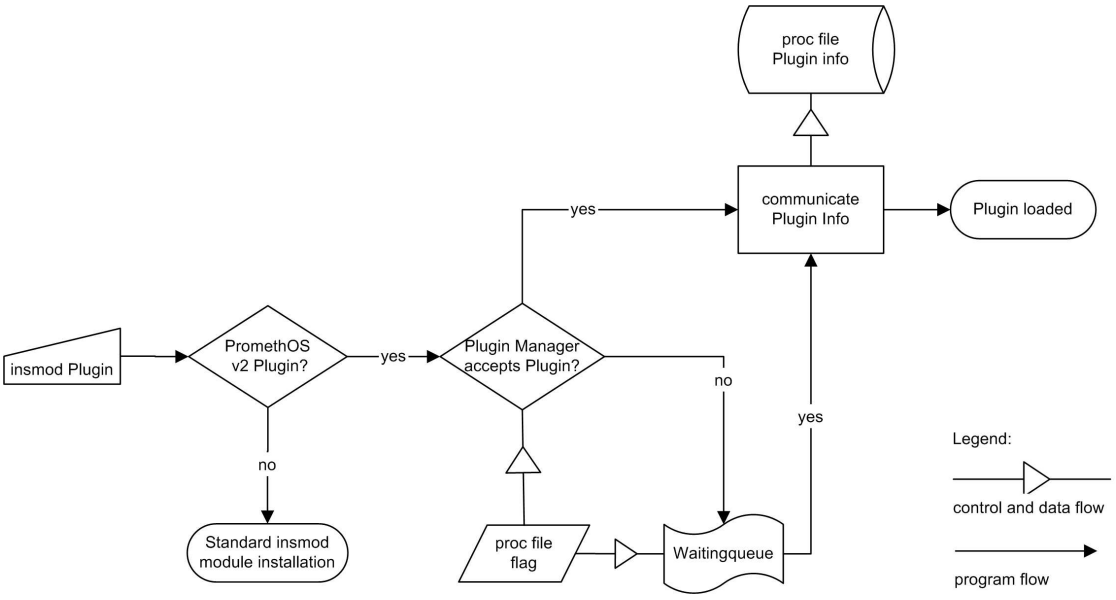

Abbildung 3.5: Der Vorgang des Plugin-Loadings

```
if(!promethos_is_v2_plugin(promethos_procfile_data.m_name))
```

Der Algorithmus zur Identifikation eine Plugins in der Funktion `promethos_is_v2_plugin()` besteht darin, dass der der Funktion übergebene Modulname mit dem String `plugin_name`[29] auf Gleichheit überprüft wird. Da dieser Identifikationsalgorithmus in einer separaten Funktion gekapselt wurde, kann dieser später durch einen komplexeren Algorithmus substituiert werden (hierbei sind eventuell notwendige Anpassungen an der zu übergebenden Parameterliste zu beachten).

Falls kein Plugin detektiert wurde, wird nun der originale `insmod`-Code abgearbeitet. Falls jedoch ein Plugin detektiert wurde, erfolgt die nachfolgend beschriebene Kommunikation mit dem Plugin Manager:

Zuächst wird überprüft, ob ein neues Plugin vom Plugin Manager akzeptiert wird. Falls der Plugin Manager gerade beschäftig ist[30], wird die *Plugin Loader Instanz* in eine *Waitingqueue* eingereiht, und erst dann wieder fortgesetzt, wenn der Plugin Manager das entsprechende Signal hierfür gibt.[31] Realisiert wird diese Koordination duch den Aufruf der Funktion

```
promethos_v2_proc_read(promethos_proc_filename)
```

Diese Funktion ist direkt in `insmod.c` implementiert, da diese nur von hier aufgerufen wird. Sie führt eine Leseoperation auf der als Parameter übergebenen Datei `promethos_proc_filename` aus.[32]
Nun ist jedem `proc file` eine Funktion zugeordnet, welche bei Leseoperationen auf das File ausgeführt wird, und eine Funktion, die bei Schreiboperationenen auf dieses File ausgeführt wird. Die für Leseoperationen registrierte Funktion ist im *Plugin Manager* Code[33] implementiert:

```
promethos_proc_read()
```

Folgendes Code-Fragment aus dieser Funktion stellt nun die aufrufende Plugin Loader Instanz

[29] Defi niert in `promethos_v2_plugin_loader.h`
[30] Das Flag `promethos_procfile_data.m_flag` hat den Wert `1`
[31] Das Flag `promethos_procfile_data.m_flag wird auf den Wert 0 zurückgesetzt`
[32] IPC-Datei, über welche die Kommunikation zwischen *Manager* und *Loader* abgewickelt wird.
[33] File `linux/net/ipv4/netfilter/ipt_PROMETHOS.c`

in einer Warteschlange ein, falls der Plugin Loader gerade keine neuen Plugins akzeptiert:

```
if (wait_event_interruptible( \
       wait_until_plugin_manager_is_ready,\
       !promethos_procfile_data.m_flag)){ \
       printk("PM: Plugin Manager not ready. \
       PL enqueued\n");
}
```

Die Warteschlange `wait_until_plugin_manager_is_ready` wird im zuvor durch die Zeile

```
static DECLARE_WAIT_QUEUE_HEAD(wait_until_plugin_manager_is_ready);
```

deklariert. Signalisiert der Plugin Manager die Bereitschaft zur Installation des Plugins[34], setzt der *Plugin Loader* die Plugin-Installation fort. Dem Plugin Manager werden nun die Informationen bezüglich des Plugins kommuniziert. Dies geschieht durch den Aufruf der Funktion:

```
promethos_v2_proc_write(promethos_proc_filename)
```

welche die Plugin-Daten in das *proc-file* schreibt. Für Schreiboperationen auf dieses File ist wiederum eine im Plugin Manager Code implementierte Funktion registriert, welche diese Daten entgegennimmt und in gleichnamiger Struktur abspeichert:

```
promethos_proc_write(...)
```

Der im Rahmen dieser Diplomarbeit nicht manipulierte `insmod`-Code führt nun alle notwendigen Schritte durch, um den Code des Plugins ausführungsbereit - und in das *Standard Module Handling* von Linux integriert - im *Kernel Memory Space* abzulegen. Da `insmod` nun aber über den Aufruf der Funktion `sys_init_module()`[35] die Initialisierungsfunktion des Plugins ausführt, muss hier erneut eine Fallunterscheidung implementiert werden, um eine Ausführung des Plugin-Codes in dem “ungeschützten” Kernel Kontext zu verhindern. Diese bedingte Ausführung wird[36] durch den Rückgabewert der Funktion `promethos_is_v2_plugin()` bestimmt. Die durch diese Abfrage mögliche Ignoranz der folgenden Codesequenz hat - wie gewünscht - das Nichtausführen der *PromethOS v2 Plugin* Initialisierungsfunktion zur Folge.

```
if(promethos_is_v2_plugin(name))
{
      if (mod->init && (error = mod->init()) != 0) {
      atomic_set(&mod->uc.usecount,0);
      mod->flags &= ~MOD_INITIALIZING;
      if (error > 0)      /* Buggy module */
      error = -EBUSY;
      goto err0;
      }
}
```

Das Image dieses Plugins kann nun von dem Plugin Manager, wie in Sektion 3.3.1 dargestellt, innerhalb einer sicheren EE ausgeführt werden.

3.3.4 Plugin

In den PromethOS Plugins findet die Bearbeitung der Paketinhalte statt. Um diese Bearbeitung durchführen zu können, muss ein Plugin zum einen die gewünschten Algorithmen der Paketbearbeitung implementieren, zum anderen muss das Plugin über definierte Schnittstellen mit dem PromethOS Framework kommunizieren können. Dies gilt sowohl für den Austausch der eigentlichen Datenpakte als auch für den Austausch von Kontrollinformationen. So muss ein Plugin beispielsweise signalisieren können, dass ein zu sendendes Paket bereitsteht, welches

[34] `promethos_procfile_data.m_flag == 0`
[35] Implementiert in `kernel/module.c`
[36] Wie schon im Code der Datei *insmod.c*

dann von dem Plugin Manager “abgeholt”, geprüft und gesendet werden kann. Für Plugins bildet der Plugin Manager die exklusive Schnittstelle zum PromethOS Framework. Damit ist sichergestellt, dass dieser jederzeit über die Stati der Plugins informiert ist und, falls notwendig, reagieren kann.

Für die Kommmunikation mit dem Plugin Manager bindet ein Plugin das Header-File `linux/netfilter_ipv4/promethos_v2.h` ein, welches ihm die benötigte Struktur sowohl für die Datenkommunikation als auch für etwaige Signalisierungen bereitstellt:

```
struct promethos_ipc_data_t {
        unsigned int    hooknum;
        const struct    net_device *in;
        const struct    net_device *out;
        unsigned long   instance;
        unsigned char   ip_pkt_to_plugin[PROMETHOS_MTU \
                        - sizeof(struct ethhdr)];
        unsigned long   status_flag_plugin_new_pkt;
        unsigned char   ethhdr_pkt_from_plugin \
                        [sizeof(struct ethhdr)];
        unsigned char   ip_pkt_from_plugin \
                        [PROMETHOS_MTU-sizeof(struct ethhdr)];
        unsigned char   datarefp_pkt_from_plugin[16];
        unsigned long   status_flag_plugin_active;
        unsigned int    status_flag_plugin_nr_ip_pkts_to_send;
        char message[128] ;
}promethos_ipc_data;
```

Diese Struktur befindet sich innerhalb des Modul-Adressbereichs und liegt somit in der Protection Domain des Plugins.[37] Da sich eine Kommunikation über ein *proc file*[38] aus Performancegründen hier nicht eignet, wurde eine direkte Kommunikation über *Shared Memory* implementiert. Dies ist möglich, da der Plugin Manager das gleiche Mapping für die autorisierten virtuellen Adressen des Plugins verwendet wie der Plugin Thread selbst. Abbildung 3.6 veranschaulicht dieses Kommunikationsprinzip.

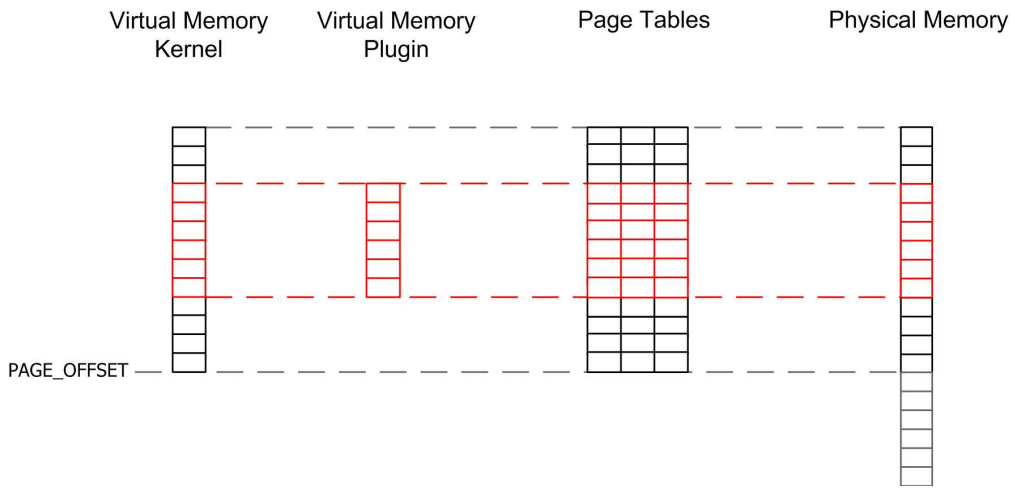

Abbildung 3.6: Shared Memory zwischen Plugin und Plugin Manager

Der Pluginmanager bindet ebenfalls das Header-File `linux/netfilter_ipv4/promethos_v2.h` ein und definiert einen Pointer auf den Typ `promethos_ipc_data_t`

```
static struct promethos_ipc_data_t * ptr_promethos_ipc_data;
```

[37] Dieser Bereich wird durch die Restricted Page Tables des Plugins gemappt und kann von diesem adressiert werden
[38] IPC zwischen Plugin Loader und Plugin Manager wurde auf diesem Wege realisert

Damit nun Plugin Manager und Plugin über die sich in der Protection Domain des Plugins befindende Struktur `promethos_ipc_data` kommunizieren können, ruft das Plugin in seiner Initialisierungsmethode eine Registrierungsfunktion des Plugin Managers auf. Dieser Funktion werden nun als Parameter neben dem Namen des Plugins[39] auch die Adresse der IPC-Struktur übergeben. Im Anschluss hieran erfolgt der Aufruf der Funktion, die der Plugin-Entwickler für die Bearbeitung der Paketdaten vorgesehen hat (in diesem Fall wurde die Funktion beispielhaft `function_that_does_packet_processing()` genannt).

```
int init_module(void)
{
      promethos_v2_register("v2TEST",&promethos_ipc_data);
      function_that_does_packet_processing();
      return 0;
}
```

Der Plugin Manager prüft nun, ob der Speicherbereich der als Parameter übergebenen Struktur `promethos_ipc_data`[40] in der Protection Domain des Plugins liegt.[41] Für die IPC steht somit die Struktur `promethos_ipc_data` zur Verfügung, dessen Felder im Folgenden erläutert werden:

- `hooknum, in, out`
  Informationen, die standardmäßig von Netfilter an ein Netfilter Target bei dessen Aufruf als Parameter mitübergeben werden [Rus03]. Damit diese Informationen den PromethOS Plugins ebenfalls zur Verfügung stehen, werden diese Parameter in die dem Plugin zugänglichen Felder kopiert, sobald ein Paket dem Plugin übergeben wird.
- `ip_pkt_to_plugin`
  Der Plugin Manager übergibt dem Plugin ein Paket dadurch, dass er es in den Speicherbereich dieses Arrays kopiert.
- `instance`
  Falls mehrere Instanzen eines Plugins existieren (beispielsweise um verschiedene Flows differenziert zu bearbeiten) gibt diese Nummer die ID der Instanz an, welche für die Bearbeitung des Paketes vorgesehen ist [Guin01].
- `status_flag_plugin_new_pkt`
  Dieses Feld dient der Koordination zwischen Plugin und Plugin Manager. Übergibt der Plugin Manager dem Plugin ein Paket, signalisiert er dies dem Plugin indem er den Wert dieses Feldes auf `1` setzt. Das Plugin kann nun durch eine Abfrage auf diesen Wert feststellen, ob ein Paket zur Bearbeitung vorliegt oder nicht. Wenn das Plugin das Paket bearbeitet hat[42], setzt er dieses Flag wieder auf den Wert `0` zurück. Dadurch wird dem Plugin Manager signalisiert, dass er dem Plugin ein neues Paket übergeben kann, ohne eventuell das zuletzt übergebene Paket zu überschreiben.
- `ip_pkt_from_plugin`
  Der Speicherbereich, in welchem das *Plugin* dem *Plugin Manager* ein zu sendendes IP-Paket bereitstellt.
- `ethhdr_pkt_from_plugin`
  Dieses Array wurde eingeführt, um später genügend Speicherplatz zur Verfügung zu haben, um einen Layer-2-Header vor den IP-Header platzieren zu können. Dieses Feld ist für das Plugin nicht von Bedeutung.
- `datarefp_pkt_from_plugin`
  Dieses Array dient dazu, um bei einer späteren Verwaltung des Pakets durch eine Socket-Buffer-Struktur genügend Platz für einen Reference-Counter hinter dem eigentlichen Paket zur Verfügung zu haben. Dieses Feld ist für das Plugin ebenfalls ohne Bedeutung.

[39] Hier wurde als Beispiel v2TEST gewält

[40] Bzw. deren Startadresse

[41] Dieser Check ist notwendig, um Sicherzustellen, dass der Kernel nicht durch eine ungültige Adresse außerhalb der Protection Domain kompromittiert werden kann

[42] Oder beispielsweise an einen anderen Speicherplatz kopiert hat

- `status_flag_plugin_active`
  Dieses Flag dient als Indikator für die Aktivität des Plugins. Hat das Plugin ein Paket bearbeitet (und dies dem Plugin Manager entsprechend signalisiert) und wurde dem Plugin kein weiteres Paket übergeben, setzt das Plugin diesen Wert auf `0`. Für den Plugin Manager ist dies das Signal, dass der Plugin Thread gestoppt werden kann bis das nächste, für dieses Plugin bestimmte Paket eintrifft.

- `status_flag_plugin_nr_ip_pkts_to_send`
  Falls das Plugin ein Paket senden möchte (bzw. in den Protokoll-Stack (re)injizieren möchte), signalisiert er dies dem Plugin Manager dadurch, dass er diesen Wert auf `1` setzt. Sobald das Paket von dem Plugin Manager entgegengenommen und gesendet wurde, wird dieser Wert wieder dekrementiert.

Der minimale Aufbau eines PromethOS-Plugins ist - neben den generellen Anforderungen an Kernel Module, welche die Linux Modulverwaltung impliziert - der Folgende:

1. Es muss die Header-Datei `linux/netfilter_ipv4/promethos_v2.h` eingebunden werden

2. Die Initialisierungsfunktion `init_module` muss implementiert werden.
   Hierin ist zum einen das Plugin durch den Aufruf der Funktion `promethos_v2_register()` zu registrieren, zum anderen ist die Funktion aufzurufen die die eigentliche Paketbearbeitung durchführt.

3. Die paketverarbeitende Funktion ist entsprechend den Vereinbarungen bezüglich der IPC zwischen dieser und dem Plugin Manager zu programmieren. Abbildung 3.8 veranschaulicht die IPC zwischen Plugin Manager und Plugin und ordnet diese in das PromethOS v2.0 Framework ein.

   Da der Plugin-Code als Thread ausgeführt wird, muss dieser so aufgebaut sein, dass er nicht terminiert. Dies ist notwendig, da der Plugin Manager den Plugin Thread bei einem ankommenden Paket lediglich in die RUNQUEUE einträgt damit dieser gescheduled wird, nicht aber von neuem startet. Es ist also eine Endlosschleife zu implementieren, in welcher der eigentliche Bearbeitungscode ausgeführt wird.

   ```
   while (1){ ... }
   ```

   Das Plugin prüft nun zyklisch ob eine neues Paket für dieses angekommen ist:

   ```
   if (ptr_promethos_ipc_data->status_flag_plugin_new_pkt)
   ```

   Falls diese Überprüfung den Wahrheitswert `true` zurückgibt, dann wurde dem Plugin ein IP-Paket übergeben. Dieses wurde in das Feld `ip_pkt_to_plugin` seiner IPC-Struktur kopiert und kann nun bearbeitet werden. Zunächst einmal sollte das Plugin das Paket in einen anderen Speicherbereich innerhalb seiner Protection Domain kopieren (beispielsweise in das Array `ip_pkt_from_plugin` seiner IPC-Struktur, welches für ein zu sendendes Paket des Plugins reserviert ist). Dies ist sinnvoll, da er nach dem Kopieren der Daten dem Plugin-Manager signalisieren kann, dass der Empfangsspeicher `ip_pkt_to_plugin` für das nächste Paket bereitsteht:

   ```
   memcpy(&ptr_promethos_ipc_data-> \
          ip_pkt_from_plugin, \
          &ptr_promethos_ipc_data->ip_pkt_to_plugin,
          PROMETHOS_MTU-sizeof(struct ethhdr));
   ```

   `PROMETHOS_MTU-sizeof(struct ethhdr)` entspricht der maximalen Größe des IP-Pakets. Nun wird dem Plugin-Manager signalisiert, dass der Empfangsspeicherbereich des Plugins wieder frei ist:

```
ptr_promethos_ipc_data->status_flag_plugin_new_pkt=0
```

Nun kann das Plugin mit seiner eigentlichen Paketbearbeitung beginnen. Je nachdem, welche Aufgabe das Plugin zu erfüllen hat, ist natürlch der Bearbeitungscode von Plugin zu Plugin unterschiedlich. Nachdem das Plugin die Bearbeitung abgeschlossen hat, signalisiert es dem Plugin Manager, falls es ein Paket senden möchte, die Sendebereitschaft durch das Umsetzen des entsprechenden Flags:

```
ptr_promethos_ipc_data->status_flag_plugin_nr_ip_pkts_to_send=1
```

Anschliessend signalisiert es dem Plugin Manager das Ende seiner Bearbeitung, sodass es von diesem gesoppt werden kann:

```
ptr_promethos_ipc_data->status_flag_plugin_active=0
```

3.3.5 Interrupt- und Exception-Handler

- Interrupt und Exception Handling

  Die Erkenntnisse E11-E13 aus Abschnitt 2.5 führen folgendes Problem bei der Umsetzung der Memory Resource Control Architektur vor Augen:

  Ereignet sich ein Interrupt oder eine Exception während der Ausführung eines PromethOS Plugin Threads, so kann die korrespondierende ISR, die von der *Control Unit* als Reaktion auf diesen Interrupt ausgeführt werden muss, nicht aufgerufen werden. Der Grund dafür sind die gerade aktiven - und somit während des Interrupt-Handlings Verwendung findenden - *Restricted Page Tables* des *Plugin Threads*. Diese bilden die ensprechenden Bereiche aus Sicherheits- und Stabilitätsgründen nicht ab, was impliziert, dass auf diese nicht zugegriffen werden kann.

  Der PromethOS Interrupt Handler manipuliert nun das Linux Interrupt-Handling wie folgt:

  Nachdem dem Interrupt Controller der CPU ein Interrupt signalisiert wurde, führt dieser die in 2.5.2 beschriebenen Schritte zur Behandlung dieses Interrupts durch. Das PromethOS Interrupt-Handling greift nun in diese Sequenz wie folgt ein:

  Bevor die ISR aufgerufen wird, wird geprüft, ob es sich um ein PromethOS Plugin Thread handelt. Falls dies der Fall ist, wird die gerade aktive Plugin-PGDBA gesichert und die Basisadresse der PGD des Kernels[43] wird in das entsprechende Register [44] geladen.

  Auf der für die Implementierung der Architektur verwendeten IA32 wird diese Funktionalität durch eine Erweiterung des Makros `BUILD_IRQ(nr)` realisiert. `BUILD_IRQ(nr)` wird jeweils vor dem Aufruf einer ISR ausgeführt und stellt somit eine geeignete Posititon für die Implementierung dar.

  Nach der Ausführung einer ISR wird das Makro `return_from_interrupt` ausgeführt. In diesem Makro wird nun wieder geprüft, ob der aktuelle Kontext der eines Plugin Threads ist, und - falls dies so ist - wird die vorher gespeicherte PGDBA des Plugins in das entsprechende Register zurückgeladen.

  Das Makro `return_from_interrupt` ist, wie auch das Makro `BUILD_IRQ(nr)`, hardwarespezifisch und wurde unter *IA32* in der Datei `arch/i386/kernel/entry.S` implementiert. Da die Implementierung in der nur schwer les- und interpretierbaren

[43] Unter `swapper_pg_dir` referenzierbar
[44] Bei einer *IA32* ist dies das `cr3` Register

Sprache `Assembler` vorgenommen wurde, wird hier auf die Darstellung des Codes, welches die Funktionaliät implementiert, verzichtet.

- Page Fault Handler

Die Notwendigkeit der Anpassung des Linux Page Fault Handlings im Rahmen der Implementierung der *Memory Resource Control Architecture* liegt in der Art und Weise begründet, wie dieser auf einen Speicherzugriffsfehler einer Kernelkomponente reagiert.

Die ISR, welche bei Speicheradressierungsfehlern aufgerufen wird, ist die Funktion

```
do_page_fault( )
```

Die Implementierung dieser Funktion ist architekturabhängig. Auf einer IA32 befinden sich die Sourcen dieses Handlers in der Datei `arch/i386/mm/fault.c`

Adressiert ein Plugin nun eine nicht-autorisierte virtuelle Speicheradresse, wird diese Adressierung von der MMU als Fehler interpretiert, da die Adresse durch die Restricted Page Tables nicht gemappt ist und dementsprechend keiner physikalischen Adresse entspricht. Diese generiert nun ein Page Fault Signal, wodurch der Page Fault Handler aufgerufen wird.

Der Page Fault Handler reagiert nun wie in Abbildung 3.7 gezeigt, mit einem *Kernel "Oops"*[45].

Folgendes Code-Fragment aus dieser Funktion ist hierfür verantwortlich und muss

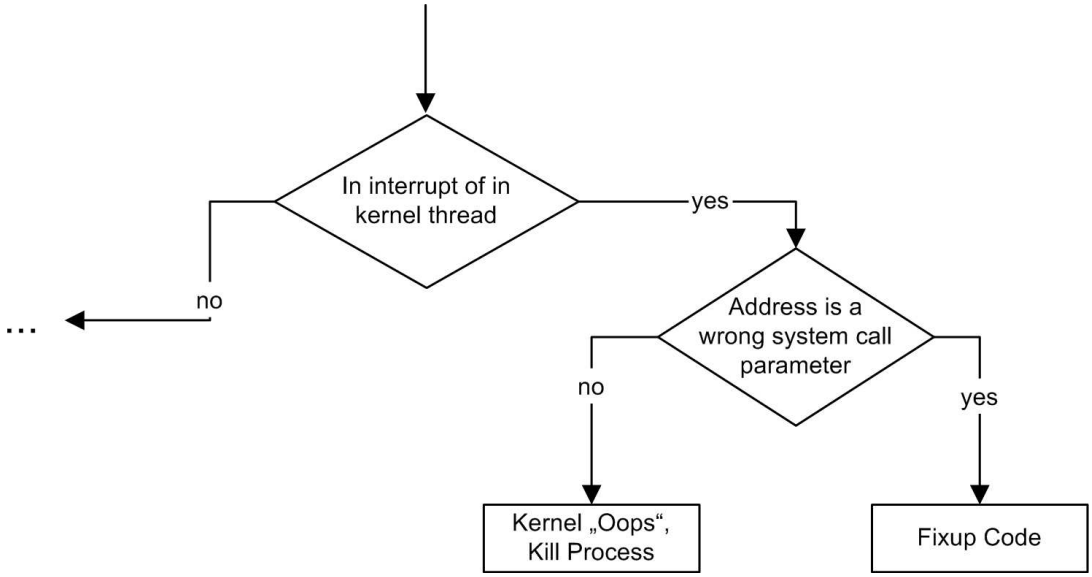

Abbildung 3.7: Page Fault Handler

entsprechend manipuliert werden, damit das System angemessen auf eine Deadressierung eines Plugins reagieren kann:[46]

```
/*
```

[45] Schwerwiegender Fehler im Kernel. Entspricht einem Crash des Betriebssystems, da sich das NodeOS im Kernel Mode, und nicht im User Mode befi ndet

[46] Bei der Memory Resource Control Architektur wurde der Befehl `die("Oops", regs, error_code)` enfernt, damit das NodeOS als ganzes nicht zum Stillstand kommt.

```
 * Oops. The kernel tried to access some bad page. We'll have to
 * terminate things with extreme prejudice.
 */
bust_spinlocks(1);
if (address < PAGE_SIZE)
       printk(KERN_ALERT "Unable to handle kernel \
              NULL pointer dereference");
else
       printk(KERN_ALERT "Unable to handle kernel \
              paging request");
 printk(" at virtual address %08lx\n",address);
 printk(" printing eip:\n");
 printk("%08lx\n", regs->eip);
 asm("movl %%cr3,%0":"=r" (page));
 page = ((unsigned long *) __va(page))[address >> 22];
 printk(KERN_ALERT "*pde = %08lx\n", page);
 if (page & 1) {
        page &= PAGE_MASK;
  address &= 0x003ff000;
  page = ((unsigned long *) __va(page))[address >> PAGE_SHIFT];
  printk(KERN_ALERT "*pte = %08lx\n", page);
       }
 die("Oops", regs, error_code);
 bust_spinlocks(0);
 do_exit(SIGKILL);
```

3.4 PromethOS v2.0 mit Netfilter

In Kapitel 2.1 wurde dargestellt, wie das *PromethOS Target* die Verwaltung von - und Kommunikation mit - den Plugins übernimmt, und hiermit eine dynamische Installation von Plugins ermöglicht.

Dieses Prinzip der Registrierung einer zentralen Komponente, welche dann von dem *Netfilter Framework* kommende Pakete an die dynamisch installierten Plugins weiterleitet (und vice versa) wurde auch für die Integration des entwickelten *Memory Resource Control Frameworks* angewendet. Die zentrale Komponente ist hierbei der *Plugin Manager*.

Um die bereits entwickelten Funktionalitäten des bestehenden *PromethOS Frameworks* nutzen zu können, und diese um die Möglichkeit der Ausührung von *Plugins* innerhalb *Protection Domains* zu erweitern, registriert sich der Plugin Manager nicht direkt bei *Netfilter*, sondern bei dem *PromethOS-Target*. Die Implementierung wird im Folgenden beschrieben.

Sobald der *Plugin Loader* dem *Plugin Manager* die Installation eines neues *Plugins* signalisiert, führt der *Plugin Manager* die in Abschnitt 3.3.1 unter *Punkt 1* genannten Schritte durch; ein “sicheres” *EE* wird generiert und das Plugin wird im Kontext dieses *EEs* zur Ausführung gebracht, damit dieses sich bei dem *Plugin Manager* registrieren kann.

Die Registrierung des Plugins bei dem Plugin Manager erfolgt durch den Aufruf der von dem Plugin Manager hierfür exportierten Registrierungsfunktion in der Initialisierungsmethode des Plugin-Moduls:

```
promethos_v2_register(char *plugins_name, \
                      struct promethos_ipc_data_t *ptr)
```

Hierdurch wird dem Plugin Manager der Name des Plugins sowie die Adresse der Struktur mitgeteilt, über welche die spätere *IPC* zwischen *Plugin Manager* und *Plugin* stattfindet (sonstige Informationen, die der Plugin Manager zur Verwaltung des Plugins benötigt, wurden diesem bereits von dem *Plugin Loader* kommuniziert).

Mit diesen Informationen registriert der *Plugin Manager* das *Plugin* nun wiederum bei dem *PromethOS Target*[47]

```
promethos_register(char *name,
                   promethos_target_func_t tfunc,
                   promethos_config_func_t cfunc,
                   promethos_config_func_t recfunc,
                   promethos_print_func_t pfunc)
```

Als den zu registrierendenden Plugin-Namen wird der `String` übergeben, unter welchem sich das Plugin bei dem Plugin Manager registrierte. Als Funktionspointer werden für jedes *PromethOS v2.0 Plugin* die gleichen, von dem Plugin Manager implementierten, Standard-Funktionen übergeben.[48] Als 2. Parameter beispielsweise, welcher der Funktion entspricht, die bei ankommenden - für das Plugin bestimmten - Paketen aufzurufen ist (das Target) wird der Name der Funktion

```
unsigned int promethos_standard_target(struct sk_buff **pskb,
                                       unsigned int hooknum,
                                       const struct net_device *in,
                                       const struct net_device *out,
                                       unsigned long instance)
```

übergeben. Auf die Semantik der einzelnen Parameter wird in 3.3.4 eingegangen.[49] Nach der Installation und Registrierung des Plugins kann dieses nun Pakete annehmen, vearbeiten und versenden. Abbildung 3.8 zeigt die nun im Folgenden beschriebenen Zusammenhänge und Abläufe des PromethOS v2.0 Frameworks.

[47] Bezüglich Registrierung siehe 2.1

[48] Implementiert wurde der Plugin Manager als auch die Standard-Funktionen in der Datei `net/ipv4/netfilter/ipt_PROMETHOS.c`

[49] Die übrigen Funktionen werden derzeit nicht verwendet und wurden als 'Dummy'-Funktionen ohne Wirkung implementiert

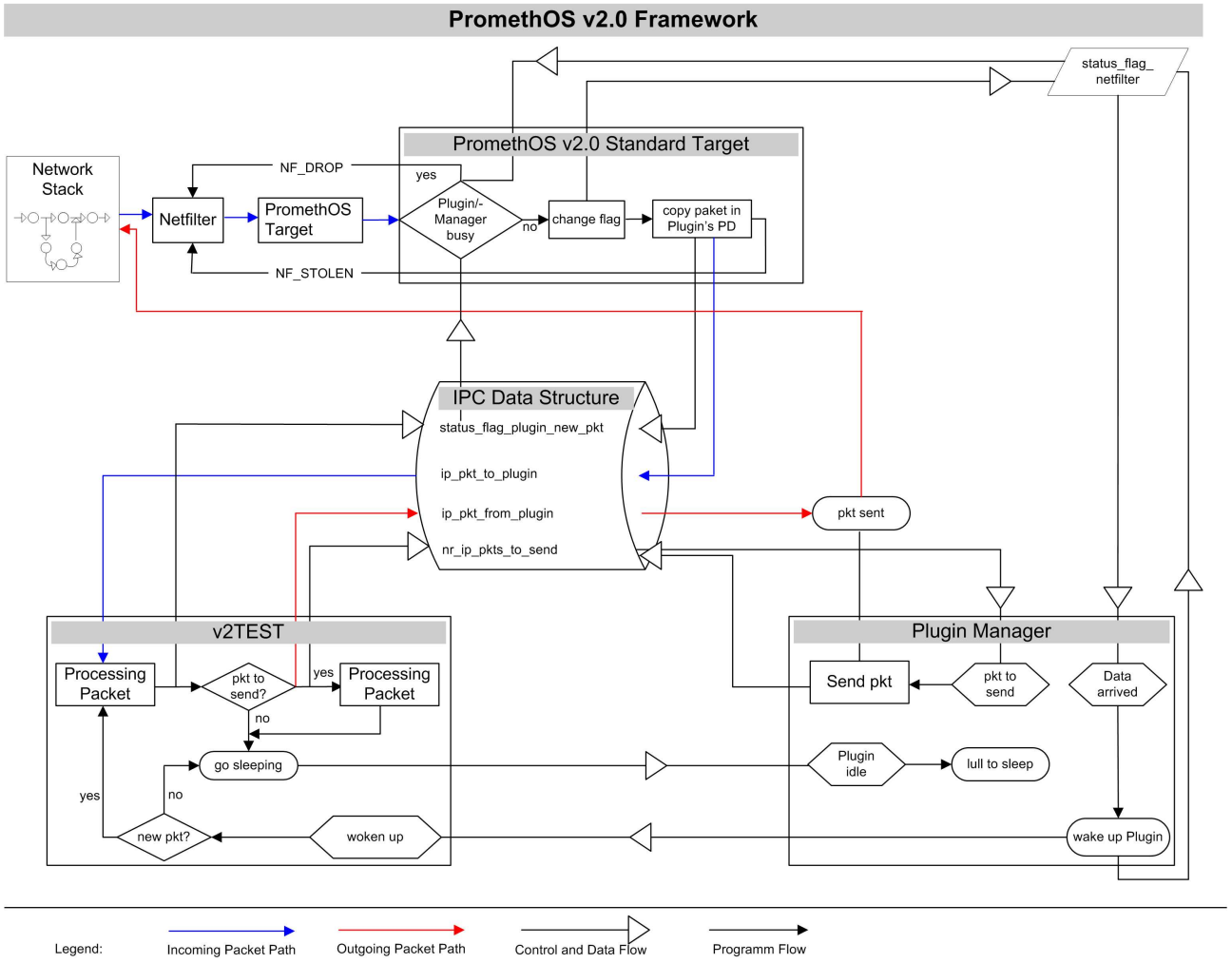

Abbildung 3.8: Das PromethOS 2.0 Framework im Detail

Sobald ein für das Plugin bestimmtes Paket ankommt, ruft Netfilter die Funktion `target()` des PromethOS-Targets auf, welche wiederum die Funktion `promethos_standard_target()` aufruft, die folgende Aktionen ausführt:

Zunächst wird die Variable `status_flag_netfilter` überprüft. Der Wert `1` zeigt an, dass der *Plugin Manager Thread* zuvor während der Paketbearbeitung unterbrochen wurde.

Der Plugin Manager ist ein Kernel Thread, wird gescheduled, und kann somit unterbrochen worden sein. Er muss bei ankommenden Paketen beispielsweise einen “schlafenden ”Plugin Thread reaktivieren. Er setzt die genannte Variable auf den Wert `1` sobald er mit dem notwendigen Management der Plugin Threads nach einem Paketeingang beginnt. Sobald er alle diesbezüglichen Aktivitäten abgeschlossen hat, setzt er die Variable wieder zurück auf den Wert `0`. Damit signalisiert er, wie in 3.3.1 beschrieben, seine (Wieder)bereitschaft.

Falls die Variable `status_flag_netfilter` den Wert `1` hat, so wird das Paket nicht weiter verarbeitet; die Funktion gibt den Wert `NF_DROP` zurück[50]. Andernfalls wird wie folgt Verfahren:

Anhand des Wertes `instance` kann die *IPC*-Struktur des Plugins ermittelt werden. Über die Abfrage des Wertes des Feldes `status_flag_plugin_new_pkt` dieser Struktur wird ermittelt, ob das zuletzt an dieses Plugin adressierte Paket von diesem “abgeholt” und verarbeitet wurde. Falls das Paket nicht abgeholt wurde, wird - um ein Überschreiben des von dem Plugin gerade bearbeiteten IP-Pakets zu vermeiden - eine Warnmeldung aus - und der Wert `IPT_DROP` zurückgegeben.[51]

Falls der Wert des Flags `status_flag_plugin_new_pkt` `0` entspricht (Paket wurde abgeholt), werden die Felder der *IPC*-Struktur entsprechend den Werten der übergebenen Parameter gesetzt und das Paket in das hierfür vorgesehene Array der Struktur kopiert.

```
ptr_promethos_ipc_data->hooknum=hooknum;
ptr_promethos_ipc_data->in=in;
ptr_promethos_ipc_data->out=out;
ptr_promethos_ipc_data->instance=instance;
memcpy(&ptr_promethos_ipc_data->ip_pkt_to_plugin,
       (*pskb)->nh.iph,
       (unsigned long)((*pskb)->tail) - \
       (unsigned long)((*pskb)->nh.iph)  );
```

Aufrund der Tatsache, das der nicht als Kernel Thread ausgeführte Netfilter-Code nonpreemptiv ist, muss diesem ein Wert zurückgegeben werden, damit dieser seine Aktivität abschliessen - und damit ein Scheduling des Plugin Managers und später des Plugins erfolgen kann. Es wird hier der Wert `NF_STOLEN` zurückgegeben, da das Paket aus dem Protokoll-Stack zur späteren Verarbeitung “entwendet” wurde.

Der Plugin Manager sorgt nun dafür, dass der Plugin-Thread gestartet wird und somit das Paket verarbeiten kann. Die plugin-internen Abläufe, die Kommunikation zwischen Plugin und Plugin Manager sowie die Kontrolle des Plugin Threads durch den Plugin Manager funktionieren wie in Sektion 3.3.4 beschrieben.
Aufgrund der angesprochenen Asynchronität zwischen Paketeingang, Paketbearbeitung und einer Paketrückgabe muss nun eine separate Funktion implementiert werden, welche von einem Plugin als `zu senden` markierte Pakete in den Protokoll-Stack (re)injiziert. Die Implementierung dieser Funktion wurde in Sektion 3.3.1 vorgestellt.

Bezüglich des Feldes `instance` ist noch Folgendes zu erwähnen: Als Instanznummer des Plugins wird die *PID* des *ECs* gewählt. Somit werden zwar derzeit keine unterschiedlichen Instanzen eines Plugins unterstützt, allerdings gewährleistet dieses Vorgehen

[50] An dieser Stelle sollte in einer späteren Version ein Paketpuffer implementiert werden, damit der Plugin Manager, sobald dieser die Paketbearbeitung abgschlossen hat, mit der Bearbeitung des aktuellen Pakets fortfahren kann

[51] In einer späteren Version ist dieses Paket in einen Puffer zu stellen, damit diesem dem Plugin übergeben werden kann sobald dieser das Ende der Bearbeitung des “alten” Pakets signalisiert

eindeutige Identifikationsnummern der Plugins.[52] Die Instanznummer des zu registrierenden Plugins wird dem PromethOS Target dadurch mitgeteilt, dass diese in ein proc-file, `/proc/promethos/net/instance`, geschrieben wird [Guin01].

[52] Dies gilt nicht für Mulitprozessorarchitekturen

Kapitel 4

Tests und Evaluation

Im Folgenden wird nun die *Memory Resource Control Architektur (MRC)* sowie deren Integration in das *Netfilter-Framework* getestet und evaluiert. Hierbei wird die Effizienz der Schnittstellen innerhalb des MRCs (zwischen Plugin Manager und Plugins), als auch die der Schnittstellen zwischen dem Plugin Manager und dem Netfilter-Framework betrachtet.

4.1 Aufbau der Testumgebung

Um Systemtests durchführen zu können, wurde zum einen ein Paket Generator entwickelt, welcher periodisch von einem Plugin zu verarbeitende Pakete senden und auch wieder empfangen kann. Zum anderen wurde ein Test-Plugin entwickelt, welches Pakete entgegennimmt, vearbeitet und zurückgibt.

Abbildung 4.1 zeigt das Testszenario im Überblick. Das Verfahren sowie die beiden oben genannten Komponenten werde nun im Detail dargestellt.

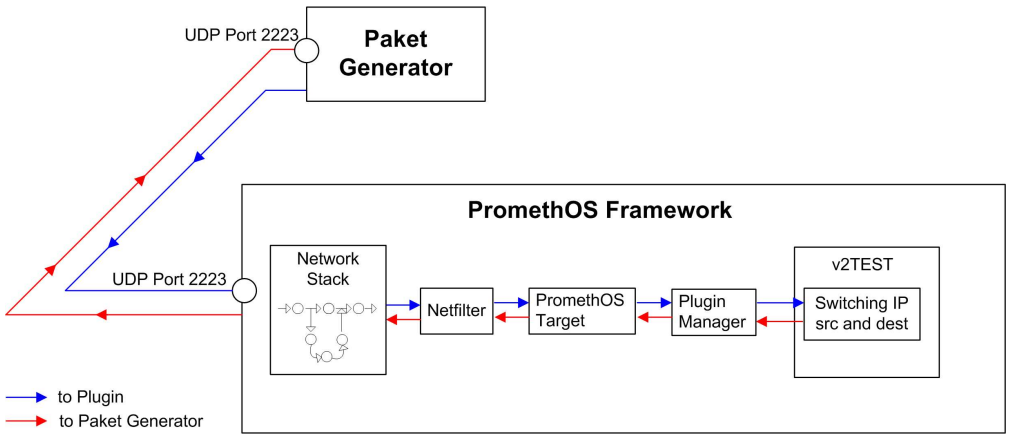

Abbildung 4.1: Aufbau der Testumgebung

- Paket Generator

  Ein Paket Generator wurde bereits im Rahmen einer anderen Diplomarbeit am *TIK* als User-Space-Programm implementiert und musste somit nur noch geringfügig angepasst werden.[1] [Ern03] Der *Paket Generator* sendet periodisch eine bestimmbare Anzahl von UDP-Test-Paketen an den *PromethOS Node*. Diese Pakete sind an einen Port adressiert, für welchen das *v2Test-Plugin* registriert ist. Nach dem Senden der *UDP-Test-Pakete* sendet der Paket Generator ein *UDP-Control-Paket*. Da die Pakete als

[1] Implementiert wurde dieser in der Datei echo_client.c

Mirror-Pakete von dem Plugin an den gleichen Port zurückgesendet werden an welchen der Paket Generator diese sendete, kann der Paket Generator an diesem Port “lauschen ” und somit die Mirror-Pakete empfangen und auswerten.

Erweitert wurde der Paket Generator um die Möglichkeit, eine festlegbare Verzögerung zwischen zwei Sendevorgängen einzufügen. Somit kann die Korrelation zwischen Sendefrequenz und Fehlerrate ermittelt werden.

Die Parameter des Paket Generators im Überblick:

- `IP_DEST`
  Die IP-Adresse des PromethOS-Nodes
- `UDP_PORT`
  Der Empfangsport von Plugin und Paket Generator
- `NO_OF_TEST_PKT`
  Anzahl der zu sendenden UDP-Testpakete
- `TEST_PKT_SIZE`
  Größe der Datenpakete
- `SLEEP_MICROSECONDS`
  Die Verzögerung zwischen zwei Sendevorgängen in Mikrosekunden

- PromethOS v2.0 Test Plugin

Als TEST-Plugin wurde das Plugin *v2TEST* entwickelt.[2] Die Funktionalität dieses *Test-Plugins* entpricht der eines sogenannten *UDP-Mirrors*.[3] Die Implementierung des Test-Plugins wird im Folgenden beschrieben:

Das Plugin prüft zyklisch, ob ihm ein neues Paket vom Plugin Manager übergeben wurde:

```
if (ptr_promethos_ipc_data->status_flag_plugin_new_pkt)
```

Ist dies der Fall, wird das Paket an die Lokation für ausgehende Pakete kopiert und dessen Source- und Destination-Address getauscht.[4]

```
u32 odaddr = iph_pkt_out->saddr;
u32 osaddr = iph_pkt_out->daddr;
...
memcpy(&ptr_promethos_ipc_data->ip_pkt_from_plugin, \
       &ptr_promethos_ipc_data->ip_pkt_to_plugin,\
       PROMETHOS_MTU-sizeof(struct ethhdr));
...
odaddr = iph_pkt_out->saddr;
osaddr = iph_pkt_out->daddr;
iph_pkt_out->daddr = odaddr;
iph_pkt_out->saddr = osaddr;
```

Anschliessend wird dem Plugin Manager ein zu sendendes Paket sowie das Ende der Paketbearbeitung signalisiert:

```
ptr_promethos_ipc_data->status_flag_plugin_nr_ip_pkts_to_send=1;
ptr_promethos_ipc_data->status_flag_plugin_active=0;
```

Das Paket wird nun von dem Plugin Manager registriert und gesendet. Der *Plugin Thread* wird gestoppt.

[2] Implementiert wurde das Plugin in der Datei `/v2TEST/promethos_v2TEST.c`

[3] Source- und Destination-Address eines eingehenden UDP-Pakets werden vertauscht und das Paket wird wieder gesendet. Damit wird ein simples Echo auf ein UDP-Paket erzeugt

[4] `iph_pkt_out` zeigt auf den IP-Header des ausgehenden Pakets. Die Semantik der restlichen Variablen und Pointer wurde in Sektion 3.3.4 beschrieben

4.2 Testmethode

Als Testumgebung wurde bei allen nachfolgend beschriebenenen Tests die im vorhergehenen Abschnitt beschriebene Testumgebung verwendet.

Der Paket Generator erfasst die Zeitpunkte der zu sendenden Pakete unmittelbar vor dem Sendevorgang. Weiter werden die Zeitpunkte der eintreffenden Mirror-Test-Pakete sowie des Mirror-Kontroll-Pakets erfasst. Die Auswertung startet mit dem Eingang dieses Kontrollpakets. Da die Paketgröße als Testparameter vorgegeben wird, können nun folgende Daten ermittelt werden:

- *Durchsatz*
  Der Durchsatz wird ermittelt, indem die Anzahl der angekommenen Pakete mit der Paketgröße multipliziert wird. Der errechnete Wert wird nun durch die Zeitdifferenz zwischen dem Sendezeitpunkt des zuerst gesendeten Pakets und dem Eingangszeitstempel des Kontrollpakets bestimmt.

- *Fehlerrate*
  Der Prozentsatz der transferierten Pakete wird durch Division der Anzahl empfangener Testpakete und der Anzahl gesendeter Testpakete ermittelt. Subtrahiert man diesen Wert von 1, erhält man die Fehlerrate.

- *Latenz*
  Um die Latenz des Frameworks zu bestimmen, werden die Zeitpunkte des Eintritts ankommender Pakete in das Framework sowie die des Austritts der entsprechenden Mirror-Pakete aus dem Framework erfasst. Die Differenz zwischen dem Sende- und dem Eingangzeitpunkt entpricht der Latenz des Frameworks.

- *Round Trip Time (RTT)*
  Dies ist die Zeitdifferenz zwischen dem Senden eines Pakets durch den Paket Generator und des Eintreffens dessen Mirrorpakets bei dem Paket Generator. Auf das Erfassen der beiden Zeitpunkte wurde bei der Vorstellung des Paket Generators bereits eingegangen.

Es wird nun jeweils das Verhalten des Frameworks bei unterschiedlichen Senderaten/Paketgrößen-Kombinationen hinsichtlich Durchsatz, Paketverlustrate, Latenz und Round Trip Time betrachtet.

Betrachtet wird das Verhalten des Frameworks bei Paketgrößen von 300, 600, 900 und 1200 Byte, sowie einer sukzessiven Erhöhung der Senderate. Für jede Senderate/Paketgröße-Kombinationen werden hierfür einmal 10 und einmal 100 Testpakete von dem Paket Generator an den PromethOS v2.0 Knoten gesendet.

Ursprünglich sollte das Framework mit den in nachfolgender Tabelle angegebenen Verzögerungsdauern bezüglich der Sende-Iterationen gestest werden. Aufgrund von Stabilitätsproblemen des Framworks bei einer Unterschreitung von dessen gemessener Latenzzeit, konnten die Tests lediglich für Verzögerungsdauern von 1000000 Mikrosekunden durchgeführt werden. Eine aussagekräftige Analyse des Durchsatzes und der Fehlerrate ist bei dieser langsamen Sendefrequenz nicht sinnvoll. Da während der Tests keine Paketverluste auftraten war der Durchsatz einzig durch Senderate und Paketgröße bestimmt.

Neben dem Test des in Sektion 3.2 dargestellen PromethOS v2.0 Frameworks wird dieses Framework ebenfalls in modifizierter Form getestet. Durch einen Vergleich der Testergebnisse bei verschiedenen Varianten können Rückschlüsse auf die Einflüsse verschiedener Komponenten und Schnittstellen auf Performance und Latenz gezogen werden. Nachfolgende Tabelle zeigt die getesteten Framework-Varianten:

SLEEP_MICROSECONDS
1000000
250000
100000
50000
10000
5000
1000
500
100
0

Tabelle 4.1: Variation der zeitlichen Abstände zwischen zwei Sendevorgängen

Variante	Beschreibung
1	Das unmodifizierte Framework. Für die Bearbeitung des Pakets sind mehrere *Kontext Switches* notwendig: *Kernel - Plugin Manager - Plugin - Plugin Mangager*. Darüber hinaus wird das Paket drei mal kopiert: In die PD, innerhalb der PD von dem Empfangs-Array in das Ausgangs-Array, von dem Ausgangsarray an die von dem Socket Buffer erwartete, ursprüngliche Stelle.
2	Der Plugin-Manager übernimmt selbst die Mirror-Funktion des Plugins. Ein Plugin wird hier nicht gescheduled; Kontextswitches zwischen Plugin Manager und Plugin entfallen. Ausserdem wird das Paket nicht kopiert.
3	Das registrierte Target führt selbst die Mirror-Funktion aus. Netfilter-Pakete werden nicht mehr per NF_STOLEN aus dem Netzerk-Stack entnommen und später wieder eingefügt, sondern sie werden direkt bearbeitet. Es finden keine Context-Switches und keine Kopieraktivitäten statt.

Tabelle 4.2: Die Framework-Testvarianten

4.3 Resultate

In diesem Abschnitt werden die Resultate der Variationen der Paketgrößen für unterschiedlichen Framework-Varianten bei einer Sendefrequenz von einem Paket pro Sekunde tabellarisch dargestellt. Die Werte in der Tabelle stellen die Durchschnittswerte von 100 Testpaketen dar. Diese entsprechen tendenziell den Durchschnittwerten beim Senden von 10 Paketen, sodass hier nicht beide Testergebnisse separat aufgeführt werden. Eine Beurteilung dieser Ergebnisse folgt im Anschluss. Die vollständige Liste der Testergebnisse befindet sich im Anhang dieser Dokumentation.

Paketgröße (Byte)	RTT (Microsec.)	Latenz (Microsec.)	Latenzanteil
300	289701	289332	0.9987
600	296775	295999	0.9979
900	300072	299234	0.9972
1200	300185	299147	0.9965

Tabelle 4.3: Testresultate Framework 1

Paketgröße (Byte)	RTT (Microsec.)	Latenz (Microsec.)	Latenzanteil
300	590	353	0.5998
600	893	272	0.3040
900	1299	448	0.3446
1200	1411	321	0.2278

Tabelle 4.4: Testresultate Framework 2

Paketgröße (Byte)	RTT (Microsec.)	Latenz (Microsec.)	Latenzanteil
300	360	1	0.0024
600	588	1	0.0014
900	811	1	0.0009
1200	1034	1	0.0008

Tabelle 4.5: Testresultate Framework 3

4.4 Beurteilung der Resultate

Für eine Analyse der Testergebnisse wurden diese in Abbildung 4.2 grafisch dargestellt. Die Abbildung zeigt die Latenz- und Round-Trip-Zeiten der drei getesteten Framework-Varianten in Abhängigkeit von den gesendeten Paketgrößen. Aufrund der Tatsache, dass Latenzen und RTTs verschiedener Frameworks teilweise um mehrere Größenordnungen voneinander abweichen, wurde hierbei ein logarithmischer Maßstab verwendet.

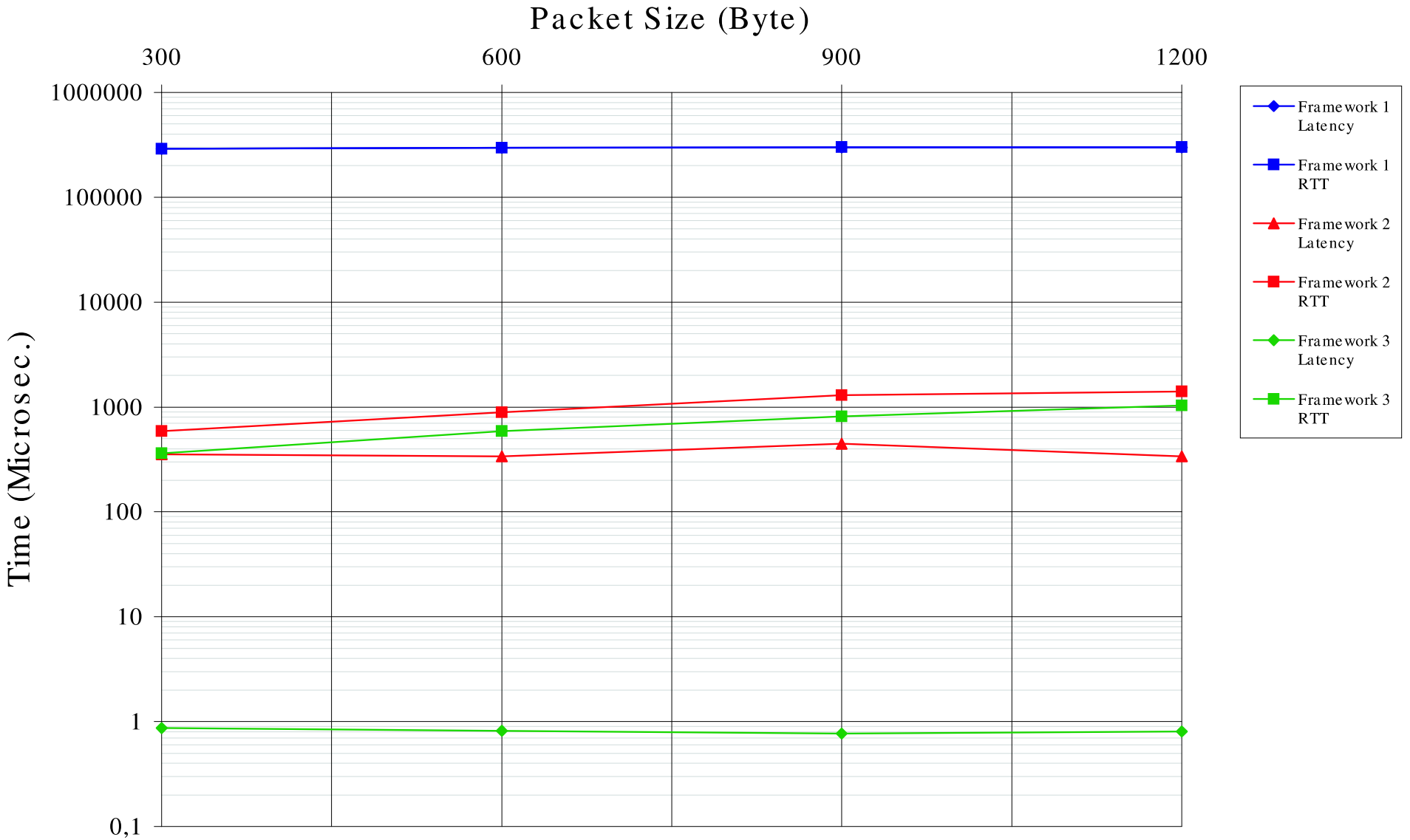

Abbildung 4.2: Grafische Darstellung der Testergebnisse

Die RTT zeigt eine positive Korrelation mit der Paketgröße. Je mehr Daten pro Paket gesendet und empfangen werden, desto länger dauert eine RTT. Die Differenzen zwischen RTT und Latenz ist bei den drei betrachteten Frameworks annähernd gleich. Sie entspricht in etwa der Framework 3 RTT.

Die eigentliche Aufgabe der Paketbearbeitung, nämlich das Vertauschen von Quell- und Zieladresse sowie das Neuberechnen der Paketchecksumme, kostet auf dem PromethOS Testknoten (wie die Latenzzeit von Framework 3 zeigt) weniger als eine Mikrosekunde. Es ist auch zu sehen, dass die Paketbearbeitungsdauer indifferent bezüglich der Größe der Testpakete ist. Dies ist dadurch zu erklären, das unabängig von der Paketgröße immer die exakt gleichen Schritte ausgeführt werden. Der Anteil der Latenz dieses 'Direct-Return-Frameworks' im Vergleich zur Round Trip Time liegt, je nach Paketgröße, zwischen 0,24% und 0,08%.

Framework 2 unterscheidet sich von Framework 3 dadurch, dass das Paket per `NF_STOLEN` aus dem Netfilter-Framework genommen wird. Bei dem nächsten Scheduling des Plugin-Managers führt dieser die Paketbearbeitung durch und reinjiziert das Paket wieder in den Netzwerk-Stack. Die Latenzzeit dieses Frameworks liegt bei mehr als 300% über der von Framework 3. Auch die RTT des Frameworks liegt dementsprechend um den Wert der höheren Latenz über der von Framework 3. Der Anteil der Latenz an der RTT liegt hier schon bei 23% - 60%.

In Framework 1, dem eigentlichen PromethOS v2.0 Memory Resource Control Framework, beträgt die Latenzzeit des Systems annähernd 300000 Mikrosekunden. Im Vergleich zu Framework 3 ist dieser Wert mehr als 300000 mal , im Vergleich zu Framework 2 in etwa 1000 mal so hoch. Der Anteil der Latenz an der RTT liegt hier bei appoximativ 99.8%.

Diese hohe Latenz kann auf folgende Ursachen zurückgeführt werden:

Während des Weges eines Pakets durch das Frameworks wird dieses insgesamt drei mal kopiert. Einmal in die PD des Plugins, einmal vom Eingangs- in den Ausgangsspeicher des Plugins, und einmal aus der PD des Plugins an seine ursprüngliche Stelle. Weiter finden mehrere Kontextswitches statt. Vor dem Empfang und nach dem Sendewunsch eines Plugins wird jeweils immer der Plugin Manager gescheduled. Der Scheduling-Algorithmus ist der Linux-Standard-Round-Robbin-Algorithmus mit den Standard-Zeitschlitzen pro EC. Da kein Realtime-Scheduling betrieben wird, sind die entsprechenden Zeitschlitze sehr groß. Hierdurch werden, nachdem der Plugin Manager oder ein Plugin ihre Schritte erledigt haben, wertvolle CPU-Zyklen nicht genutzt. Bis zum Ablaufen des Zeitschlitzes befindet sich das Framework in einem quasi-inaktiven Zustand; der Ablauf innerhalb des Framework verzögert sich hierdurch.

Kapitel 5

Schlussfolgerungen und Ausblick

5.1 Was wurde erreicht

In dieser Arbeit wurde gezeigt, wie das *Linux Memory Management* um eine Speicherzugriffs- und -verbrauchskontrolle im *Kernel Space* erweitert werden kann. Das entwickelte PromethOS v2.0 Framework stellt den Plugins nun EEs zur Verfügung, die in Protection Domains instanziert werden. Auch wurden Interfaces definiert, die einen funktionierenden Austausch von Daten- und Kontrollinformationen sowohl zwischen Plugin und Plugin Manager, als auch zwischen Plugin Manager und dem Netfilter-Framework ermöglichen. PromethOS v2.0 kann somit als NodeOS genutzt werden. Bezüglich den Design-Anforderungen kann Folgendes festgestellt werden:

- *Integration in den Kernel und Verwendung (gegebenfalls Erweiterung) von Linux-Standard-Komponenten*
  Es wurde, wo immer möglich und sinnvoll, auf Linux-Standard-Komponenten aufgebaut und die Konzepte der Linux-Subsysteme bei der Realisierung berücksichtig bzw. benutzt. So wurde beispielsweise für das Laden des Modulcodes in den Kernelspeicher sowie die Koordination und Kommunikation mit dem Plugin Manager der Linux-Standard-Module-Loader `insmod` verwendet und entsprechend angepasst. Plugin Module können damit weiterhin von dem Linux-Module-Handling-Subsystem verwaltet werden. Für das Umschalten auf die einem Plugin zugewiesene Protection Domain bei dessen Ausführung wurde der Context-Switch-Mechanismus des Schedulers verwendet. Dies ist möglich, da als EE der Kontext eines modifizierten Kernel Threads gewählt wurde.

- *Portabilität*
  Der Code ist bis auf hardwareabhängige Bereiche des PromethOS Interrupt-Handlers sowie bei der Generierung der Protection Domains plattformunabhängig. Der hardwareabhängige Code ist jedoch im Falle der Protection Domains in zwei Funktionen gekapselt, sodass diese lediglich bei einem Hardwarewechsel substituiert werden müssen. Das gleiche gilt für den Interrupt-Handler; hier müssen die entprechenden Assembler-Makros an die neue Architektur angepasst werden. Auch wurden wo immer möglich Linux-APIs verwendet, was eine positive Auswirkung auf die Kompatibilität zu zukünftigen Linux-Kernel-Versionen hat.

Der modulare Aufbau des Frameworks und dessen Kooperation mit dem vorhandenen PromethOS v1.0 Framework erlaubt es sogar, “unsichere” PromethOS v1.0 Plugins parallel zu “sicheren” PromethOS v2.0 Plugins auszuführen. Es könnten somit beispielsweise Plugins zuerst innnerhalb von Protection Domains ausgeführt werden, und falls diese über einen bestimmten Zeitraum kein nicht-autorisiertes Verhalten aufweisen, könnten diese dann als v1.0 Plugins ausgeführt werden.[1]

Die Evaluation des Frameworks hat gezeigt, dass die Performance des Systems, sowie dessen Verhalten bei Überlast, für eine sinnvolle Nutzung dieses Frameworks als NodeOS nicht geeignet sind. Im nächsten Abschnitt werden mögliche Ansätze angesprochen, wie Stabilität

[1] Die Plugins sind hiefür entsprechend zu modifi zieren

und Performance des Systems gesteigert werden können.

5.2 Ausblick

Diese Arbeit konnte zeigen, dass PromethOS den Plugins Protection Domains im Kernel Space zur Verfügung stellen kann. Das entwickelte Framework hat, wie die Betrachtung der Latenzzeit das Systems zeigt, jedoch eher konzeptionellen Charakter; ist für eine sinnvolle Nutzung als NodeOS in der entwickelten Basisversion nicht geeignet. Neben der Identifikation und Behebung des Stabilitätsproblems bezüglich zu hoher Paketeingangsfrequenzen sollte folgenden Punkten in zukünftigen Folgearbeiten besondere Beachtung geschenkt werden:

- *Real-Time-Scheduling*
  Der Scheduling-Algorithmus von Linux sollte so geändert werden, dass dieser den Echtzeit-Anforderungen des Frameworks gerechter wird. Ein schnelles Scheduling, also mit kurzen Zeitschlitzen pro EC, würde sich günstig auf die Performance des Frameworks auswirken. Die Zeitschlitze der Plugin Threads sollten so justiert werden, dass sie in etwa deren durchschnittlichen Bearbeitungszeiten entsprechen. Längere Zeitschlitze sind gleichbedeutend mit einer “Verschwendung” von CPU-Zyklen. Der Plugin Manager sollte nach einem Bearbeitungsvorgang direkt die Kontrolle an einen bestimmen Plugin Thread abgeben können, sodass dieser direkt mit der Paketbearbeitung beginnen kann.

- *Paket-Pufferung*
  Derzeit reagiert das Framework mit dem Verwerfen neu von Netfilter übergebener Pakete, falls ein Plugin noch nicht das Ende der Bearbeitung signalisiert hat. Auch werden Pakete verworfen, falls der Plugin Manager gerade Pakete an ein Plugin übergibt oder von diesem entgegennimmt. Aufgrund der Asynchronität zwischen Paketein- und -ausgang sollten hier angemessen große Paketwarteschlangen implementiert werden. Auch sollte das Framework die Paketannahme nicht verweigern, falls das Paket nicht an das sich gerade in Bearbeitung befindende Plugin adressiert ist. Hier ist eine differenzierteres Verhalten des Plugin Managers zu implementieren.

- *Kopieren des Pakets*
  Ein von einem Plugin zu verarbeitendes Paket wird in dem derzeitigen Framework mehrmals kopiert. Zum einen wird ein ankommendes Paket in die Protection Domain des zuständigen Plugins kopiert, zum anderen wird ein von einem Plugin verarbeitetes und als “zu senden“ markiertes Paket wieder aus der Protection Domain an seine ursprüngliche Speicherlokation zurückkopiert. Damit kann der Plugin Manager den “alten” Empfangs-Socket-Buffer eines Pakets auch zum senden des Pakets verwenden. Eine Performance-Steigerung könnte dadurch erreicht werden, dass - anstatt das Paket zurückzukopieren - die Pointer des das Paket im Kernel repräsentierenden Socketbuffers auf die Speicherlokation des Pakets in der Protection Domain gesetzt werden.

Als prototypische Implementation der PromethOS Memory Resource Control Architektur wurde diese, um die Komplexität des Frameworks überschaubarer zu halten und einzelne Konzepte effizienter überprüfen zu können, zunächst auf die einmalige Registrierung und Ausführung eines einzelnen Plugins beschränkt.

Der Plugin Manager kann aber in einem zweiten Schritt - sobald alle Konzepte optimiert und erfolgreich geprüft werden konnten - um die Verwaltungsfähigkeit mehrerer gleichzeitig registrierter Plugins erweitert werden. Bei der Implementation des Plugin Managers wurde an vielen Stellen eine zukünftige Multipluginfähigkeit des Framework miteingeplant. So wird beispielsweise ein Array von Thread-Verwaltungsinformationen bereitgestellt, welches für eine Verwaltung mehrerer Plugin Thread notwendig ist.[2]

Der Interrupt-Handler konnte aus zeitlichen Gründen nicht vollständig implementiert werden. Aus diesem Grunde konnte die Funktionsfähigkeit des Protection-Domain-Konzepts noch nicht

[2] anstatt lediglich eine einzelne Struktur hierfür bereitzustellen

belegt werden. Es wurden allerdings Tests durchgeführt, die zeigten, dass Speicherzugriffe von nicht durch die Restricted Page Tables gemappten Adressen zu Page Faults führen, was die konzeptionelle Korrektheit des Ansatzes zeigt. Um das System testen zu können, verwendet ein Plugin derzeit die Kernel Page Tables.

Auch müssen Deinstallationsroutinen geschrieben bzw. optimiert werden. Derzeit können Plugins nicht “sauber” deinstalliert werden. Dies gilt sowohl für den Plugin Loader, welcher derzeit noch keine Unload-Funktion bereitstellt, als auch für den Plugin Manager, welcher eine Routine zum endgültigen Beenden des Plugin Threads bereitstellen muss. Darüber hinaus sind bei der Deinstallation eines Plugins die Speicherbereiche der Restricted Page Tables sowie der des Memory Descriptors freizugeben. Derzeit werden lediglich die entsprechenden Pointer auf `NULL` gesetzt.

Die Funktion `kernel_thread()`, welche in derzeit unabgewandelter Form unter dem Namen `promethos_v2_plugin_kernel_thread()`[3] zur Erstellung eines neuen Plugin Threads verwendet wird, sorgt während der Generierung des Plugin Threads für die einmalige Ausführung des Kernel Threads. Dieses muss unterbunden werden, da die initiale Ausführung des Plugin-Codes erst nach der Generierung und Zuweisung der Protection Domain erfolgen darf.

[3] Der Code dieser Assembler-Funktion befi ndet sich in der Datei `arch/i386/kernel/process.c`

Anhang A

Offizielle Aufgabenstellung

Sommersemester 2002

Diplomarbeit

für

Thomas Setzer

Betreuer: Lukas Ruf
Stellvertreter: Matthias Bossardt

Ausgabe: 10. September 2002
Abgabe: 10. März 2003

Memory Resource Control in the Linux 2.4 Kernel

1 Einführung

Aktive Netzwerkknoten (Active Network Nodes (ANN)), die dem Plugin Modell [3] folgen, bieten die Möglichkeit an, ihre Funktionalität zur Laufzeit durch Installation von Service Code zu erweitern. Das Ziel sollte es sein, Code von Service Providern auf einem ANN installieren zu können, ohne dass die Funktion des ANNs beeinträchtigt werden kann. Auf einem ANN führt ein Node Operating System (NodeOS) die Funktion der Code-Verwaltung durch.
Am Institut für Technische Informatik und Kommunikationsnetze (TIK) der ETH Zürich (ETHZ) wurde ein Node Operating System, PromethOS [7], unter Linux [6] entwickelt, welches die Installation von Code und deren Ausführung im Linux Kernel (sog. Kernel-Module[1]) gestattet. PromethOS stellt ein PromethOS Execution Environment (PromethOS-EE) zur Verfügung, in dessen Instanzen PromethOS Plugins installiert und zur Ausführung gebracht werden können.
Knoten-Kompromittierung erfolgt durch die nicht-autorisierte Verwendung von Ressourcen durch ausgeführten Code. Das NodeOS sollte mit Hilfe des Betriebssystems in der Lage sein, den Zugriff auf Ressourcen festzulegen und zu kontrollieren. Die Ressourcen, welche duch PromethOS kontrolliert werden sollen, sind CPU Zyklen und Speicherzugriff.
In einer aktuellen Semesterarbeit [2] wird PromethOS erweitert, so dass es den Verbrauch von CPU Zyklen (Scheduling) kontrollieren kann. Aktuell ist es in PromethOS jedoch nicht möglich, den Speicherzugriff von Kernel-Modulen zu kontrollieren.
Die Kontrolle des Speicherzugriffs wurde in [4] durch einen Ansatz verfolgt, welcher auf Cyclone [1] basiert. Cylcone, ein "Safe C Dialect", wurde um Krediteinheiten erweitert, welche beim Laden und bei der Ausführung der Kernel Module kontrolliert werden. In [4] wird somit eine Restriktion hinsichtlich des verwendeten Compilers auferlegt, welcher über Node Management Wissen verfügen muss, um entsprechenden Code generieren zu können.
Am TIK wird ein Ansatz verfolgt, welcher durch die Verwendung von virtuellen Adressräumen Kernel Address Space, die dem Mach-Task Modell [8] folgen, die Installation von ungeprüftem Code ermöglichen soll.
In [5, 8] wurde gezeigt, dass virtuelle Adressräume mit Kernel-Prioritäten (Protection Domains) implementierbar sind. Diese virtuellen Adressräume gestatten das Ausführen von fremdem Code, ohne die Integrität des Betriebssystems zu gefährden.

[1] Kernel-Module, die durch PromethOS verwaltet werden, werden PromethOS Plugins genannt.

Für den Einsatz von PromethOS unter Linux ist es notwendig, dass der Speicherzugriff von PromethOS Plugins kontrolliert werden kann.

2 Aufgaben: Memory Resource Control in Linux 2.4

Im Rahmen dieser Diplomarbeit soll das Memory Management von Linux so erweitert werden, dass PromethOS EEs zur Verfügung stellen kann, welche jeweils in einem eigenen Protection Domain instanziert werden. Dadurch soll gezeigt werden, dass der Speicherzugriff und -verbrauch von Kernel-Modulen kontrolliert werden kann.
Das erweiterte Memory Management soll möglichst nahtlos in den Linux Kernel eingebunden werden können. Standard Linux Programme wie z.B. insmod fürs Laden der Kernel Module, sollen erweitert werden, sofern dies als sinnvoll erscheint. Der implementierte Code soll so weit als möglich portabel gehalten werden.
Zusätzlich müssen für den Einsatz von PromethOS als NodeOS Interfaces definiert werden, welche die Kommunikation sowohl zwischen den Plugins und dem PromethOS Framework als auch zwischen den PromethOS Plugins (Intra-EE) und zwischen PromethOS-EE Instanzen (Inter-EE) für Control-Informationen gleich wie auch für Datenaustausch ermöglichen.
Die Effizienz der Inter- und der Intra-EE-Kommunikation soll durch Messungen des Erreichten dokumentiert und mit der aktuell vorhandenen PromethOS v1.0 verglichen werden.

3 Vorgehen

- Richten Sie sich eine Entwicklungsumgebung (GNU Tools) unter Linux ein.
- Machen Sie sich vertraut mit den Unterlagen (Dokumentation und Source Code) zu Linux 2.4.17 und PromethOS sowie den Eigenschaften des Linux Kernel Module Loaders.
- Orientieren Sie sich an den Resultaten der Semesterarbeit [2] und der Diplomarbeit [5].
- Erstellen Sie einen Zeitplan, in welchem Sie die von Ihnen zu erreichenden Meilensteine Ihrer Arbeit identifizieren.
- Entwickeln Sie ein Modell, mit dem Sie die Verwaltung der PromethOS-EEs und PromethOS Plugins vornehmen können.
- Definitieren Sie die Interfaces.
- Identifizieren Sie die Orte im Linux Source Code, die verändert werden müssen, um die benötigte Funktionalität zu erreichen.
- Implementieren Sie die benötigte, in Ihrem Modell beschriebene Funktionalität für den Intel IXP1200.
- Implementieren Sie die benötigten Hilfsfunktionen für die Installation, Konfiguration, Laufzeit-Kontrolle und Entfernung der PromethOS-EEs und PromethOS-Plugins.
- Dokumentieren Sie das Erreichte ausführlich.
- Verifizieren, evaluieren und demonstrieren Sie das Erreichte durch eine Beispielapplikation. Vergleichen Sie das Erreichte mit den Resultaten zu PromethOS v1.0.
- Dokumentieren Sie die Resultate der Evaluation.

Auf eine klare und ausführliche Dokumentation wird besonders Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte und untersuchten Varianten vor dem definitiven Variantenentscheid ausführlich schriftlich festzuhalten.

4 Organisation der Arbeit

- Am Ende der ersten Woche ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Mit dem Betreuer sind regelmässige, zumindest wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen sollen die Studenten mündlich über den Fortgang der Arbeit und die Einhaltung des Zeitplanes berichten und anstehende Probleme diskutieren.

- Am Ende des ersten Monates muss eine Vorabversion des Inhaltsverzeichnis zur Dokumentation dem Betreuer abgegeben und mit diesem besprochen werden.
- Nach der Hälfte der Arbeitsdauer soll ein kurzer mündlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt. Dieser Zwischenbericht besteht aus einer viertelstündigen, mündlichen Darlegung der bisherigen Schritte und des weiteren Vorgehens gegenüber Professor Plattner.
- Am Schluss der Arbeit muss eine Präsentation von 15 Minuten im Fachgruppen- oder Institutsrahmen gegeben werden. Anschliessend an die Schlusspräsentation soll die Arbeit Interessierten praktisch vorgeführt werden.
- Die Arbeit muss regelmässig auf dem CVS-Server cvs.promethos.org gesichert werden.
- Der Topsy [9] oder der Linux Coding Style muss eingehalten werden.
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden.
- Die Dokumentation ist mit dem Satzsystem LaTeX zu erstellen.
- Es ist ein Schlussbericht über die geleisteten Arbeit abzuliefern (4 Exemplare). Dieser Bericht besteht aus einem Abstract, einer Einleitung, einer Analyse von verwandten und verwendeten Arbeiten, sowie einer vollständigen Beschreibung der Konfiguration von den eingesetzten Programmen. Der Bericht ist in Deutsch oder Englisch zu halten und beinhaltet sowohl eine deutsche wie auch eine englische Zusammenfassung (sog. Executive Summary), die Aufgabenstellung und den Zeitplan.
- Die Arbeit muss auf CDROM archiviert abgegeben werden. Stellen Sie sicher, dass alle Programme sowie die Dokumentation sowohl in der lauffähigen, resp. druckbaren Version als auch im Quellformat vorhanden, lesbar und verwendbar sind.
- Diese Arbeit steht unter der GNU General Public License (GNU GPL).
- Diese Arbeit wird als Diplomarbeit an der ETH Zürich durchgeführt. Es gelten die Bestimmungen hinsichtlich Kopier- und Verwendungsrechte der ETH Zürich.

Literatur

[1] AT&T Research Labs and Cornell University. Cyclone. http://www.research.att.com/projects/cyclone, 2001.
[2] M. Daenzer and J. Gyger. *Scheduling of PromethOS Plugins.* TIK, ETH Zurich, 2002.
[3] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next-generation routers, 2000.
[4] Herbert Bos and Bart Samwel. The Open Kernel Environment. OpenSIG 2001, 2001.
[5] C. Jeker and B. Lutz. *Memory, IO and Process/Thread Management for Topsy v3.* TIK, ETH Zurich, 2002.
[6] Linux Homepage. http://www.linux.org, 2002.
[7] Lukas Ruf. The PromethOS Homepage. http://www.promethos.org, 2001.
[8] Michael Young Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian. Mach: A new kernel foundation for UNIX development. pages 93–113.
[9] The Topsy Core Team. Topsy coding style. http://www.topsy.net/Standards, 2002.

Zürich, den 28.08.2002

Anhang B

Konfiguration

B.1 Linux Kernel Konfigurationsdatei für PromethOS v2.0

Die Datei .config wurde durch make config erzeugt. Im folgenden sind die aktivierten Optionen gelistet:

```
#
# Automatically generated by make menuconfig: don't edit
#
CONFIG_X86=y
CONFIG_ISA=y
CONFIG_UID16=y

#
# Code maturity level options
#
CONFIG_EXPERIMENTAL=y

#
# Loadable module support
#
CONFIG_MODULES=y
CONFIG_MODVERSIONS=y
CONFIG_KMOD=y

#
# Processor type and features
#
CONFIG_MPENTIUMIII=y
CONFIG_X86_WP_WORKS_OK=y
CONFIG_X86_INVLPG=y
CONFIG_X86_CMPXCHG=y
CONFIG_X86_XADD=y
CONFIG_X86_BSWAP=y
CONFIG_X86_POPAD_OK=y
CONFIG_RWSEM_XCHGADD_ALGORITHM=y
CONFIG_X86_L1_CACHE_SHIFT=5
CONFIG_X86_TSC=y
CONFIG_X86_GOOD_APIC=y
CONFIG_X86_PGE=y
CONFIG_X86_USE_PPRO_CHECKSUM=y
```

```
CONFIG_NOHIGHMEM=y
CONFIG_PREEMPT=y
CONFIG_X86_UP_APIC=y
CONFIG_X86_UP_IOAPIC=y
CONFIG_X86_LOCAL_APIC=y
CONFIG_X86_IO_APIC=y
CONFIG_HAVE_DEC_LOCK=y

#
# General setup
#
CONFIG_NET=y
CONFIG_PCI=y
CONFIG_PCI_GOANY=y
CONFIG_PCI_BIOS=y
CONFIG_PCI_DIRECT=y
CONFIG_PCI_NAMES=y
CONFIG_SYSVIPC=y
CONFIG_SYSCTL=y
CONFIG_KCORE_ELF=y
CONFIG_BINFMT_AOUT=y
CONFIG_BINFMT_ELF=y
CONFIG_BINFMT_MISC=y

#
# Block devices
#
CONFIG_BLK_DEV_LOOP=m
CONFIG_BLK_DEV_NBD=m

#
# Networking options
#
CONFIG_PACKET=y
CONFIG_PACKET_MMAP=y
CONFIG_NETLINK_DEV=m
CONFIG_NETFILTER=y
CONFIG_FILTER=y
CONFIG_UNIX=y
CONFIG_INET=y

#
#   IP: Netfilter Configuration
#
CONFIG_IP_NF_PROMETHOS=m
CONFIG_IP_NF_TARGET_PROMETHOS=m
CONFIG_IP_NF_PROMETHOS_LOG=m
CONFIG_IP_NF_PROMETHOS_TEST=m
CONFIG_IP_NF_PROMETHOS_LOG=m
CONFIG_IP_NF_CONNTRACK=m
CONFIG_IP_NF_QUEUE=m
CONFIG_IP_NF_IPTABLES=m
CONFIG_IP_NF_FILTER=m
CONFIG_IP_NF_MANGLE=m

#
# SCSI support
#
```

```
CONFIG_SCSI=y
CONFIG_BLK_DEV_SD=y
CONFIG_SD_EXTRA_DEVS=40
CONFIG_CHR_DEV_SG=m
CONFIG_SCSI_DEBUG_QUEUES=y
CONFIG_SCSI_MULTI_LUN=y
CONFIG_SCSI_BUSLOGIC=y

#
# Network device support
#
CONFIG_NETDEVICES=y

#
# ARCnet devices
#
CONFIG_DUMMY=m
CONFIG_TUN=m

#
# Ethernet (10 or 100Mbit)
#
CONFIG_NET_ETHERNET=y
CONFIG_LANCE=m
CONFIG_NET_PCI=y
CONFIG_PCNET32=m

#
# Character devices
#
CONFIG_VT=y
CONFIG_VT_CONSOLE=y
CONFIG_SERIAL=y
CONFIG_UNIX98_PTYS=y
CONFIG_UNIX98_PTY_COUNT=256

#
# File systems
#
CONFIG_AUTOFS4_FS=y
CONFIG_EXT3_FS=y
CONFIG_JBD=y
CONFIG_TMPFS=y
CONFIG_ISO9660_FS=y
CONFIG_PROC_FS=y
CONFIG_DEVPTS_FS=y
CONFIG_EXT2_FS=y

#
# Network File Systems
#
CONFIG_NFS_FS=m
CONFIG_NFS_V3=y
CONFIG_NFSD=m
CONFIG_NFSD_V3=y
CONFIG_SUNRPC=m
CONFIG_LOCKD=m
CONFIG_LOCKD_V4=y
```

```
#
# Partition Types
#
CONFIG_MSDOS_PARTITION=y

#
# Console drivers
#
CONFIG_VGA_CONSOLE=y

#
# Kernel hacking
#
CONFIG_DEBUG_KERNEL=y
CONFIG_MAGIC_SYSRQ=y
```

Anhang C

Liste geänderter und hinzugefügter Dateien

/cdrom/sources/linux/include/linux/	
promethos_irq.h	Header File zu promethos_irq.c
promethos_plugin_manager.h	Header File des Plugin Managers
promethos_v2_plugin_loader.h	Enthält IPC-Vereinbarung zwischen Plugin-Manager und Plugin-Loader (Ein Link auf diese Datei existiert in /cdrom/linux/modutils/include)

/cdrom/sources/net/ipv4/netfilter/	
ipt_PROMETHOS.c	Implementierung der PromethOS Target Funktionalität, des Plugin Managers sowie aller von dem Plugin Manager referenzierter Funktionen
promethos_TEST.c	Ergänzung des PromethOS v1.0 TEST-Plugins um eine printk()-Anweisung

/cdrom/sources/linux/modutils/insmod/	
insmod.c	Implementierung des PromethOS v2.0 Plugin Loaders

/cdrom/sources/linux/v2TEST/	
promethOS_v2TEST.c	Implementierung des PromethOS UDP-Mirror v2TEST Plugins
promethOS_v2TEST.h	Header File des PromethOS UDP-Mirror v2TEST Plugins
Makefile_promethos_v2TEST	Makefile zum Erstellen des Plugin Moduls

/cdrom/sources/linux/kernel/	
module.c	Die Identifikation eines Plugins hat das Nichtausführen seines Initialisierungscodes zur folge

/cdrom/sources/paket_generator/	
echo_client.c	Paket Generator für das PromethOS v1.0 TEST Plugin
echo_client.h	Header File des gleichnamigen Paket Generator
echo_client_v2.c	Paket Generator für das PromethOS v2TEST Plugin
echo_client_v2.h	Header File des gleichnamigen Paket Generator
Makefile	Makefile zum Erstellen der Paket Generatoren

/cdrom/sources/linux/include/linux/netfilter_ipv4/	
promethos_v2.h	Definiert die IPC-Strukturen zwischen Plugins und dem Plugin Manager

/cdrom/sources/linux/i386/kernel/	
entry.S	Interrupt Handling bezüglich iret
promethos_irq.c	Identifikation eines Plugins und gegebenenfalls Rückgabe der Plugin-PGDBA zu Beginn des Interrupt-Handlings
i386_ksyms.c	Erweiterung der Symbolexportliste des Kernels
process.c	Implementiert die Funktion promethos_v2_plugin_kernel_thread() - Generiert Plugin EE
Makefile	Änderung des Makefiles zur Integration der Datei promethos_irq.c

/cdrom/sources/linux/include/asm-i386/	
processor.h	Definiert promethos_v2_plugin_kernel_thread() aus /cdrom/linux/i386/kernel/process.c
hw_irq.h	Interrupt Handling zur Umschaltung auf die Kernel Page Tables bei aktiven Plugin Threads

Tabelle C.1: Hinzugefügte und geänderte Dateien

Anhang D

Demonstration

In diesem Kapitel wird erklärt, wie PromethOS v2.0 installiert und zur Ausführung gebracht werden kann. Weiterhin wird die parallele Ausführung eines Plugins der Version 1.0 und eines Plugins der Version 2.0 demonstriert. Hierfür wird die für die Systemevaluation erstellte Testumgebung verwendet.

Zunächst muss aus den Quelldateien unter `/cdrom/sources/linux`[1] ein neuer Kernel kompiliert werden. Folgende Optionen sind bei der Kernelkonfiguration zu beachten:

```
Code maturity level options --->
  [*] Prompt for development and/or incomplete code/drivers

Loadable module support  --->
  [*] Enable loadable module support
  [*]   Set version information on all module symbols
  [*]   Kernel module loader

Networking options  --->
  <*> Unix domain sockets
  [*] TCP/IP networking
  ...
  IP: Netfilter Configuration  --->
    <M>   PromethOS plugins (EXPERIMENTAL)
      <M>     TEST PromethOS-plugin support (EXPERIMENTAL)
    <M> IP tables support (required for filtering/masq/NAT)
      <M>   Packet filtering
      <M>   Packet mangling
```

Ansonsten sind meist die Standardoptionen zu wählen, wobei je nach Konfiguration des Rechners weitere Konfigurationsparameter zu ändern sind. Nachdem der Kernel erfolgreich gebootet werden konnte, wird als nächstes der Plugin Loader installiert. Hierfür ist wie folgt vorzugehen:

```
root:~# cd /cdrom/sources; mkdir modutils_install
root:~# cd modutils_install
root:~# /cdrom/sources/linux/modutils/configure
root:~# make; make install
```

Nun muss noch das Test-Plugin v2TEST als Kernel-Module kompiliert werden. Die Quelldatei trägt den Namen `promethos_v2TEST.c` und befindet sich im Verzeichnis `/cdrom/linux/v2TEST/`. Zur Kompilierung genügt der Aufruf `make` in dem genannten Verzeichnis. PromethOS kann nun durch folgende Befehlszeile gestartet werden, wobei das PromethOS v1.0 Plugin TEST für UDP-Pakete registriert wird:

[1] Das Root-Verzeichnis der auf die Festplatte kopierten CD-ROM wird im Folgenden mit /cdrom referenziert.

```
root:~# iptables -t promethos -A INPUT -s HOSTNAME -p udp   \
        --destination-port 2222 --j PROMETHOS --plugin TEST \
        --autoinstance'
```

Als HOSTNAME ist die IP-Adresse des Paket-Generator-Rechners einzusetzen, von welchem später die Testpakete gesendet werden sollen. Als *DESTINATION_PORT* ist 2222 eingetragen, da dieser Port im Paket Generator Source Code `echo_client.c` als `UDP_PORT` definiert wird.[2]

Nun kann das v2TEST-Plugin geladen und an einen anderen Flow gebunden werden. Man wähle die `Destinations-Port-Nummer` 2223, da diese im Paket Generator v2 Source File `echo_client_2.c` bestimmt ist (Diese Ports müssen auf beiden Computern frei und verfügbar sein).

```
root:~# cd /cdrom/linux/v2TEST
root:~# make -f Makefile_promethos_v2TEST
root:~# insmod promethos_v2TEST.o
root:~# iptables -t promethos -A INPUT -s HOSTNAME -p udp   \
        --destination-port 2223 --j PROMETHOS --plugin TEST \
        --autoinstance'
```

Nun "lauscht" das PromethOS v2 Plugin `v2TEST` an Port 2223, während das PromethOS v1 Plugin an Port 2222 "lauscht". Auf dem Paket Generator Rechner sind nun die beiden Dateien `/cdrom/sources/paket_generator/echo_client.c` und `/cdrom/sources/paket_generator/echo_client_v2.c` anzupassen. Hierzu muss zumindest `IP_DEST` auf die IP-Adresse des PromethOS-Knotens gesetzt werden. Anschliessend generiert man die Binaries durch ein Ausführen von `make` in diesem Verzeichnis.

`/cdrom/sources/paket_generator/echo_client` ruft das Promethos v1.0 Test-Plugin auf. Dieses gibt keine Pakete zurück und meldet sich lediglich bei jedem Aufruf mit einer Textzeile, welche in `/var/log/messages` eingesehen werden kann.

`/cdrom/sources/paket_generator/echo_client_v2`-Pakete sind an Port 2223 addressiert, für welchen das `v2TEST`- Plugin registriert ist. Dieses Plugin dient nun als UDP-Mirror und sendet die Pakete mit vertauschten Source- und Destination-Adressen wieder zurück zum Paket Generator.

[2] Paket Generator Benutzung wird nachfolgend beschrieben.

Anhang E

Testresultate

Resultate Framework 1

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=300 NO_OF_TEST_PKT=10
Latency (Microsec.);RTT (Microsec.)
269438;269835 309431;309799 289429;289801 269427;269800 309433;309807
289446;289815 269430;269804 309430;309801 289428;289799 269429;269802

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=600 NO_OF_TEST_PKT=10
Latency (Microsec.);RTT (Microsec.)
277924;278651 279451;280028 269445;270025 319456;320075 309445;310034
299478;300072 289445;290026 279445;280028 269445;270030 269445;270025

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=900 NO_OF_TEST_PKT=10
Latency (Microsec.);RTT (Microsec.)
289475;290495 269430;270244 309475;310282 289441;290247 289462;290276
309468;310277 289432;290241 269430;270238 309438;310249 289447;290249

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=1200 NO_OF_TEST_PKT=10
Latency (Microsec.);RTT (Microsec.)
279450;280531 279431;280482 309169;310202 289434;290483 269432;270464
309441;310470 289434;290480 279432;280475 309429;310473 269431;270479

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=300 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
279464;279861 269436;269806 309427;309791 289429;289799 269430;269804
309430;309801 289428;289799 269427;269797 309451;309807 289429;289795
269429;269795 309428;309792 289428;289796 269429;269798 309429;309801
289430;289802 269429;269801 309428;309796 289443;289816 269429;269803
309428;309799 289430;289797 269427;269788 309437;309806 289431;289805
269430;269800 309430;309793 289429;289803 269442;269813 309427;309791
289428;289801 269431;269800 309428;309790 289428;289799 269429;269798
309428;309799 289431;289801 269427;269797 309449;309817 289430;289801
269428;269798 309430;309801 289428;289798 269428;269797 309430;309801
289432;289801 269429;269799 309427;309796 289441;289811 269430;269801
309428;309796 289428;289797 269427;269798 309436;309804 289430;289786
269465;269812 309428;309799 289429;289798 269439;269809 309431;309800
289428;289804 269431;269788 309427;309798 289429;289804 269429;269794
309464;309836 289429;289798 269427;269801 309451;309820 289427;289795
269429;269800 309430;309801 289430;289800 269429;269798 309427;309799
289430;289802 269430;269800 309429;309800 289442;289811 269427;269795
309429;309805 289429;289799 269429;269798 309436;309803 289431;289804
269429;269788 309430;309801 289427;289799 269442;269813 309444;309811
289430;289779 269431;269799 309428;309800 289429;289801 269431;269804

309429;309800 289429;289800 269426;269798 309453;309824 289433;289805

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=600 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
309451;320075 269445;310034 299456;300072 309445;290026 299478;280028
294458;270030 279445;310034 269445;300072 319445;290026 309467;280028
289445;270025 279451;320075 269445;310034 319456;300072 299478;290026
289445;280028 279445;270030 269445;270030 309467;310034 299445;300072
269445;290026 309456;280028 309445;270025 299478;320075 289445;310034
279451;270030 269445;310034 319456;300072 309445;290026 299478;280028
294458;320075 279445;270025 269445;320075 319445;310034 289467;270033
299445;310034 279451;320074 269445;290026 319456;300075 299478;310024
289445;300022 279445;290026 269445;280028 309467;270034 299445;310034
269445;270030 319456;310034 309445;300068 299478;290028 289445;280028
279451;270025 269445;320005 319456;270025 309445;320075 299478;310034
294458;270033 279445;300014 269445;320074 300445;270025 309467;320075
299445;310034 279451;270030 269445;310034 319456;300072 299478;300026
289445;280028 279445;270025 269445;300075 309467;280025 299445;300075
269445;300022 319456;280028 309445;270034 299478;310034 289445;270030
279451;310034 269445;300068 319456;290028 309448;260028 299478;270025
294458;320005 279445;270025 269445;320075 319444;310034 300467;270033
289445;310014 279450;310074 269445;270025 319456;320075 299478;310034
289445;270030 279445;310034 269445;300072 309467;300026 299445;280028
269445;270025 309456;300075 279445;270032 299478;310034 269445;270075

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=900 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
279457;280460 319429;320238 299430;300237 279444;280248 319431;320238
299433;300238 279432;280238 319431;320238 299430;300237 279435;280244
319430;320236 299429;300237 279479;280281 319456;320263 299435;300239
279434;280238 319430;320238 299430;300235 279430;280235 319430;320237
299433;300234 279431;280239 319432;320237 299444;300249 279433;280238
319432;320239 299430;300237 279430;280234 319433;320238 299430;300236
279434;280239 319429;320251 299449;300269 279445;280250 319430;320235
299434;300260 279429;280251 319430;320234 299430;300252 279435;280259
319432;320241 299431;300241 279430;280253 319443;320250 299434;300233
279454;280257 319430;320233 299429;300235 279428;280235 319465;320273
299433;300246 279431;280245 319430;320249 299443;300251 279431;280237
319432;320246 299430;300238 279429;280250 319430;320251 299432;300246
279433;280989 319431;320244 299430;300250 279444;280248 319429;322112
299431;300253 279430;280245 319430;320236 299430;300237 279434;280250
319432;320242 299430;300238 279476;280282 319444;320250 299434;300244
279434;280238 319430;320236 299443;300251 279443;280249 319430;320224
299430;300238 279431;280238 319441;320257 299445;300260 279431;280252
319431;320235 299428;300235 279430;280234 319430;320237 299430;300236
279432;280239 319432;320247 299430;300254 279445;280271 319430;320252
299434;300258 279430;280246 319430;320220 299431;300252 279435;280240

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=1200 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
269440;270518 319432;320475 299431;300472 279430;280473 319432;320478
299431;300585 279433;280477 319431;320477 299430;300455 279430;280469
319431;320457 299432;300469 279431;280470 319432;320467 299430;300470
279453;280491 319464;320504 299430;300462 279431;280452 319431;320469
299431;300471 279433;280455 319434;320472 299430;300475 279432;280455
319433;320472 299432;300475 279431;280473 319430;320471 299430;300466
279429;280465 319432;320471 299431;300469 279432;280465 319432;320470
299430;300471 279433;280463 319429;320472 299430;300470 279992;281020

319434;320465 299432;300451 279431;280471 319434;320477 299431;300469
279792;280832 319432;320469 299431;300473 279430;280460 319431;320470
299431;300475 279433;280476 319432;320473 299431;300473 279429;280462
319431;320455 299432;300465 279430;280462 319433;320463 299431;300465
279432;280472 319433;320476 299432;300464 279430;280455 319430;320469
299432;300469 279432;280471 319432;320464 299429;300447 279431;280466
319431;320469 299432;300472 279432;280472 319431;320453 299431;300468
279430;280463 319433;320457 299431;300471 279431;280473 319431;320454
299468;300503 279434;280467 319431;320456 299446;300481 279446;280483
319432;320473 299432;300473 279434;280476 319444;320484 299431;300466
279431;280471 319435;320474 299431;300471 279430;280471 319433;320474
299430;300475 279433;280477 319431;320473 299430;300471 279991;281032

Resultate Framework 2

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=300 NO_OF_TEST_PKT=10
Latency (Microsec.);RTT (Microsec.)
5;476 1;365 2;365 1;365 1;363 2;366 1;365 2;362 2;644 1;365

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=600 NO_OF_TEST_PKT=10
Latency (Microsec.);RTT (Microsec.)
4;612 1;586 1;587 2;577 1;588 2;586 2;583 2;588 2;585 1;516

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=900 NO_OF_TEST_PKT=10
Latency (Microsec.);RTT (Microsec.)
4;859 5;844 4;834 5;844 5;846 5;842 4;815 6;846 5;841 5;842

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=1200 NO_OF_TEST_PKT=10
Latency (Microsec.);RTT (Microsec.)
6;1100 5;1073 4;1066 4;1065 5;1072 5;1071 4;1067 4;1067 6;1071
5;1080

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=300 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
4;411 5;396 4;401 5;401 4528;397 4;383 4;400 5;383 4;384 5;384
5;400 5;390 4;391 4;388 5;390 5;382 5;393 5;390 5;394 4;384
5;384 4;386 5;391 4;1333 5;11330 22948;386 5;400 4;397 5;399
4;388 5;400 5;392 5;395 4;392 5;397 5;398 5;397 4;394 4;394
45;393 2702;388 5;382 5;376 5;385 5;392 5;387 5;396 5;392 5;390
1729;371 5;381 4;381 4;398 4;382 5;431 5;373 5;385 5;381 5;394
5;385 4;389 5;391 5;386 5;386 5;401 4;396 5;408 5;398 5;391
5;334 4;396 2250;393 5;457 4;388 4;396 4;399 5;400 5;388 4;450
5;377 4;401 4;383 5;387 5;395 5;378 5;373 5;395 5;406 4;399
4;1180 4;403 5;394 5;5419 4;364 4;396 47;386 5;471 5;1586 4;376

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=600 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
5;742 5;625 4;628 5;621 4;606 4;615 4;608 4;610 5;605 4;619
4;614 5;613 4;617 4;621 5;612 5;626 4;607 4;606 5;608 5;621
4;618 57;675 5;617 5;621 5;621 4;607 5;614 5;607 4;602 5;612
5;611 50;663 5;607 4;612 4;620 5;620 5;606 5;633 5;610 4;613
4;616 5;624 4;614 6;623 5;618 5;615 4;609 5;623 5;601 53;669
5;619 4;621 4;615 5;634 5;621 5;621 5;611 11593;12227 4;612
4;625 5;615 5;634 1511;2125 5;631 1512;2127 5;648 4;599 5;635
5;615 5;637 5;626 11407;12021 5;625 6;629 4;621 5;636 4;607
5;623 5;619 6;641 5;621 5;622 5;614 5;626 5;631 5;620 5;614
5;624 5;613 5;638 4;614 4;624 5;614 5;629 5;623 5;638 4;600 6;630 5;610

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=900 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
7;887 5966;6829 6;866 7;861 6;863 7;850 6;859 8991;9862 6;857
6;851 6;835 7;859 6;838 6;861 8;866 6;854 7;870 7;862
6;860 6;858 6;840 5;854 5;828 6;858 6;862 7;866 6;873
6412;7277 6;860 6;848 5;836 6;867 7;848 6;852 5;854
5;850 6;859 5;846 6;863 2550;3408 6;841 6;857 6;838 6;859
6;860 7;870 6;851 6;860 5;863 5;850 7;862 6;861 5;844 6;860
6;863 5;857 6;863 6;858 6;870 5;857 7;866 6;867 6;838 6;859
6;856 6;853 6;873 6;859 8;871 5;855 5;842 6;851 6;847 6;861
5;831 7;863 7;863 5;837 5;866 6638;7522 5;841 6;858 7;859
6;862 6;857 6;853 6;857 6;7478 6603;2803 1944;5078 4223;855
6;836 5;849 6;859 5;849 6;845 6;855 6;868

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=120 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
6;1261 7317;839 5;1085 6;1089 6;1089 6;1075 7;1080 5;1082 6;1069
6;1076 6;1087 6;1078 6;1082 6;1084 6;1081 6;1082 6;1086 7;1087
6;1087 6;1082 6;1083 7;1093 7;1089 6;1087 6;1080 7;1088 5;1073
8;1100 6;1087 6;1091 5;1077 5;1087 6;1086 15521;16609 6;1090
6;1076 7;1089 6;1084 5;1081 6;1087 6;1082 6;1087 7;1087 6;1095
6;1082 6;1086 5;1083 6;1088 6;1073 6;1083 6;1076 6;1090 6;1086
8085;1088 5;1071 7;9539 6;1082 5;1096 6;1071 7;1076 8;1084
6;1082 7;1093 21;1081 7;1096 6;1116 6;1107 6;1103 5;1091 4;1097
4;1073 4;1070 4;1071 4;1069 3;1070 3;1073 3;1060 4;1058 3;1045
60;1060 3;1046 3;1117 3;1043 3;1052 4;1036 4;1039 4;1055 5;1057
4;1051 4;1071 4;1058 5;1074 5;1072 4;1070 4;1068 3;1077 5;1073
5;1071 4;1054

Resultate Framework 3

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=300 NO_OF_TEST_PKT=10
Latency (Microsec.);RTT (Microsec.)
3;502 1;366 1;361 1;364 3;391 1;363 1;357 1;355 1;363

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=600 NO_OF_TEST_PKT=10
Latency (Microsec.);RTT (Microsec.)
3;723 1;586 0;585 0;586 2;587 1;583 0;587 0;586 1;586

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=900 NO_OF_TEST_PKT=10
Latency (Microsec.);RTT (Microsec.)
3;847 1;809 1;802 1;811 0;808 1;801 1;809 1;799 1;812

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=1200 NO_OF_TEST_PKT=10 Latency (Microsec.);RTT (Microsec.)
3;1237 1;1032 1;1034 1;1033 1;1036 0;1022 1;1721 1;1034 1;1048

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=300 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
3;386 1;365 1;362 0;360 1;361 0;354 1;362 2;368 1;360 0;362
1;363 1;364 1;351 0;362 1;360 1;351 1;357 0;359 1;362 1;363
1;358 0;355 1;350 0;372 1;365 1;361 1;358 1;360 0;364 1;355
0;358 1;355 0;357 1;372 3;373 2;371 1;360 1;362 1;358 1;363
2;371 1;366 1;352 1;361 1;364 1;362 1;364 1;374 1;363 1;378
1;363 1;363 0;363 1;365 1;365 0;363 0;358 1;348 1;356 0;354
1;358 0;357 1;354 1;361 1;363 1;361 0;348 2;359 1;354 1;356
1;357 1;363 0;353 1;361 1;352 1;374 1;358 1;360 1;360 1;361
1;356 1;360 1;352 0;360 1;366 1;359 0;363 1;367 1;363 1;362

1;377 1;363 1;374 1;363 1;363 1;375 1;364 0;365 0;355 0;355

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=600 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
4;730 1;585 1;582 1;583 1;584 1;584 1;584 0;588 1;585 0;583
1;588 1;585 1;583 0;584 1;584 0;585 1;585 1;586 1;584 1;587
0;586 1;583 1;583 1;584 1;585 1;585 1;596 0;582 1;585 0;593
1;595 1;587 0;593 0;585 1;590 0;589 1;584 1;590 1;587 1;587
0;588 1;594 1;583 1;585 1;585 1;597 0;584 1;600 0;583 1;588
1;586 1;587 1;584 1;588 1;586 0;589 1;587 1;583 1;586 1;584
1;583 0;584 1;585 1;586 1;605 1;589 1;589 1;588 1;585 1;601
1;584 1;588 1;585 1;590 1;583 1;597 1;584 1;585 1;594 1;589
1;586 0;587 1;584 1;587 0;584 1;585 1;585 1;584 1;585 0;585
0;584 0;584 1;583 0;594 1;583 1;586 1;585 0;585 1;581 1;586

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=900 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
4;961 1;813 1;807 1;811 0;810 0;817 1;808 1;811 1;813 0;812
0;804 0;815 1;812 0;809 0;806 1;810 1;810 1;809 1;811 0;809
1;811 1;810 1;811 1;805 0;810 0;802 1;803 1;806 1;806 1;803
1;816 1;804 1;806 1;803 0;808 0;809 1;812 1;812 1;823 0;810
1;813 0;809 0;812 1;810 1;809 1;815 1;808 1;810 0;808 1;810
1;812 1;809 1;812 1;812 1;811 0;812 1;813 0;812 3;832 0;812
0;814 1;811 1;811 0;811 1;811 1;811 0;817 0;814 1;815 1;812
1;804 1;812 1;807 0;809 1;808 1;810 1;813 1;811 1;811 1;810
1;812 1;809 1;813 1;802 1;803 1;805 1;821 1;805 0;805 0;805
1;805 0;808 0;813 1;806 1;809 1;809 1;810 1;813 1;812 1;813

SLEEP_MICROSECONDS=1000000 TEST_PKT_SIZE=1200 NO_OF_TEST_PKT=100
Latency (Microsec.);RTT (Microsec.)
3;1059 1;1030 1;1034 1;1029 1;1024 1;1029 1;1031 1;1028 0;1029
1;1035 1;1036 2;1026 1;1025 0;1034 1;1023 0;1030 1;1030 1;1028
1;1028 1;1036 1;1027 1;1036 0;1034 0;1042 1;1037 0;1033 1;1035
0;1025 1;1036 1;1039 1;1036 1;1030 0;1036 1;1036 1;1025 1;1038
1;1039 1;1027 1;1039 1;1039 0;1036 1;1032 0;1032 1;1024 1;1033
1;1038 1;1035 1;1027 1;1038 0;1021 0;1038 1;1038 1;1034 1;1037
1;1035 1;1029 1;1031 1;1028 1;1029 1;1031 1;1019 0;1032 1;1028
1;1030 1;1031 1;1037 0;1034 0;1038 1;1030 1;1037 1;1039 1;1038
1;1055 1;1021 1;1037 1;1030 1;1037 1;1032 1;1026 0;1037 1;1042
0;1029 1;1039 1;1032 1;1035 0;1035 3;1065 0;1036 1;1028 1;1034
0;1039 0;1035 0;1038 1;1036 0;1037 1;1035 1;1044 1;1037 1;1030

Anhang F

Zeitplan

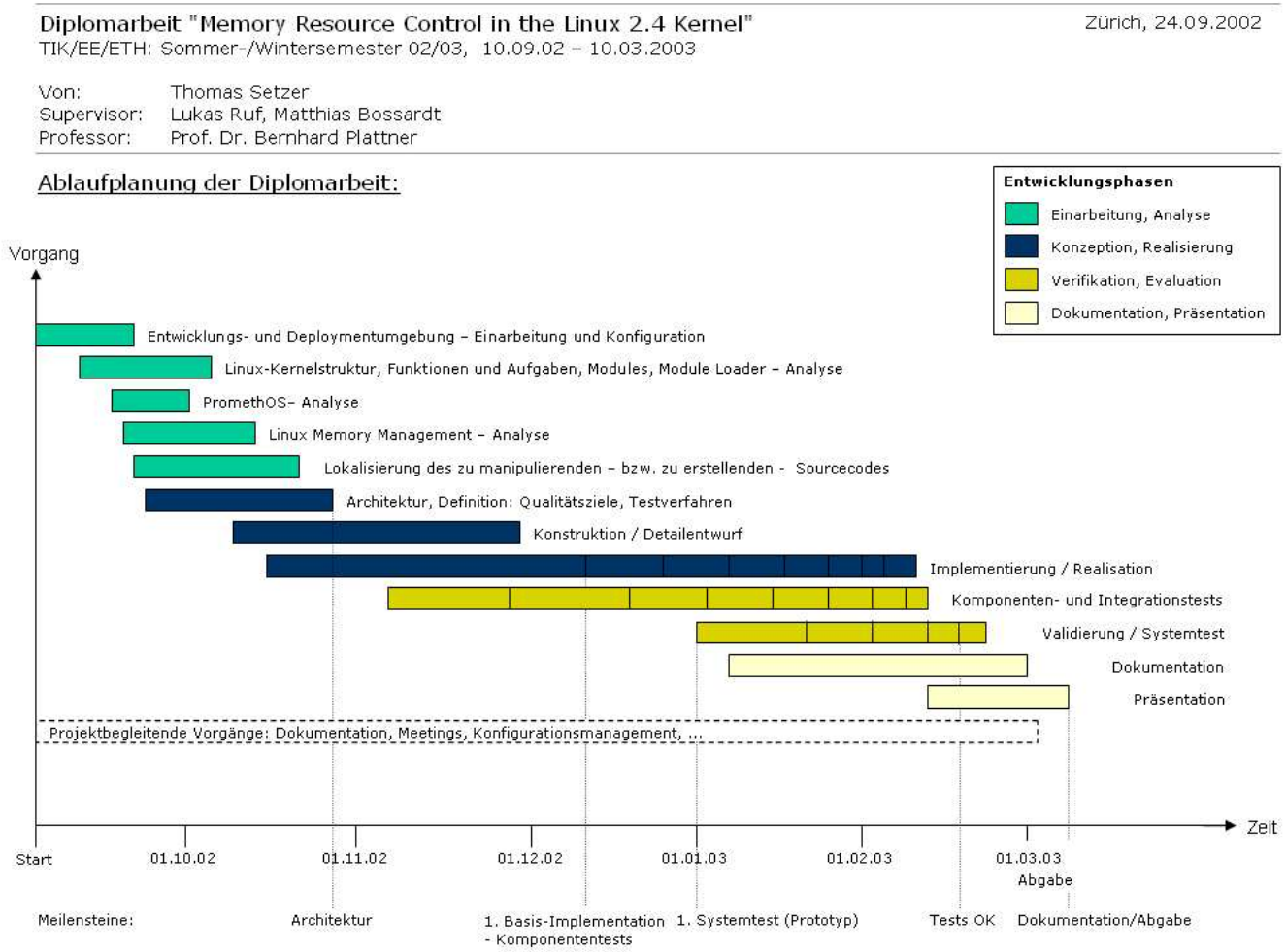

Abbildung F.1: Zeitlicher Ablauf der Diplomarbeit

Anhang G

Akronymverzeichnis

AN	Active Network
ANN	Active Network Node
EC	Execution Context
EE	Execution Environment
IA32	Intel 32 Bit Architecture
IDT	Interrupt Descriptor Table
ISR	Interrupt Service Routine
MM	Memory Management
NodeOS	Node Operating System
PID	Process Identification Number
PGD	Page Global Directory
PGDBA	Page Global Directory Base Address
PMD	Page Middle Directory
PTE	Page Table Entry
RC	Resource Control
VM	Virtual Memory

Tabelle G.1: Akronymverzeichnis

Anhang H

Literaturverzeichnis

Literaturverzeichnis

[BoCe00] D. Bovet and Marco Cesati, *Understanding the Linux Kernel*;
O'Reilly Press, 2000.

[WPRMB01] Wehrle, Pählke, Ritter, Müller, Bechler, *Linux-Netzwerkarchitektur*;
Addison-Wesley, 2002.

[Beck01] Michael Beck, *Linux-Kernelprogrammierung, Algorithmen und Strukturen der Version 2.4*;
Addison-Wesley, 2001.

[Shan01] Tom Shanley, *Protected Mode Software Architecture*;
Mindshare, Inc., 2002

[Kna02] Joe Knapka , *Outline of the Linux Memory Management System*;
Notes on the Linux memory-management system - Download 2002-12-6
http://home.earthlink.net/ jknapka/linux-mm/vmoutline.html

[Gat02] William Gatliff, *The Linux Kernel's Memory Management Unit API*;
Bill Gatliff's Home Site - Download 2002-09-10
http://www.billgatliff.com/articles/emb-linux/mmu.pfd

[Ott02] James Otto, *Linux Notes*;
Homepage of James Otto at the University of New Mexico - Download 2002-12-6
http://www.cs.unm.edu/ jotto/linux/linux.html

[Rus03] Rusty Russell and the Netfilter core team, *Netfilter/Iptables project*;
Netfilter/Iptables project homepage
http://www.netfilter.org/

[PromethOS] Lukas Ruf, *The PromethOS Homepage*;
http://www.promethos.org

[iwan02] Ralph Keller, Lukas Ruf, Amir Guindehi, Berhard Plattner, *A Dynamically Extensible Router Architecture Supporting Explicit Routing*;
Proceedings of the Fourth Annual International Working Conference on Active Networks (IWAN)
Springer Verlag, 2002

[Gor03] Mel Gorman, *Understanding the Linux Virtual Memory Manager*;
Mel Gorman's Home Site - Download 2003-01-28
http://www.csn.ul.ie/ mel/projects/vm/guide/pdf/understand.pdf

[Aiva01] Tigran Aivazian, *Linux 2.4 Kernel Internals*;
Linux Kernel Documentation Project - Download 2002-09-15
http://www.moses.uklinux.net/patches/lki.sgml

[TeWe96] D.L. Tennenhouse, D.J. Wetherall, *Towards an Active Network Architecture*;
Telemedia, Networks and Systems Group, MIT - Download 2002-12-10
ftp://ftp.tns.lcs.mit.edu/pub/papers/ccr96.ps

[Beek00] Gerard Beekmans, *Linux From Scratch*;
Linux Kernel Documentation Project - Download 2002-09-15
http://www.linuxfromscratch.org

[NaGoCa02] A. Nayani, M. Gorman and R. de Casto, *Memory Management in Linux - Desktop Companion to the Linux Source Code*;
Linux Kernel Documentation Project(Savannah) - Download 2002-09-03
http://freesoftware.fsf.org/lkdp

[Ruf02] Lukas Ruf, *Latex Essentials*;
TIK, ETH Zuerich, 2002.

[Oet02] Tobias Oetiker, *The Not So Short Introduction to LateXe*;
http://people.ee.ethz.ch/ oetiker/lshort/lshort.pdf - Download 2003-01-10

[Guin01] Amir Guindehi, *Semester Thesis: COBRA Component Based Routing Architecture*;
TIK, ETH Zuerich, 2002.

[Ern03] Pascal Erni, *Diploma Thesis: Einsatz und Programmierung des IBM NP4GS3 für aktive Netzwerkknoten unter Linux*;
TIK, ETH Zuerich, 2003.