Großer Beleg
Kapselung Mobiler Programme

Lars Reuther
Technische Universität Dresden

17. Februar 1997

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Einführung der Programmiersprache Java Ende 1995 eröffnete eine Fülle neuer Möglichkeiten für die Verbreitung von Programmen. Ursprünglich als kleine, portable Sprache für sogenannte *embedded systems* gedacht, entwickelte sich Java zu *der* Programmiersprache für die Erzeugung dynamischer Elemente im World Wide Web. Recht schnell wurden jedoch auch die Risiken dieser neuen Technik bekannt. Konnten bis dahin die Daten eines lokalen Rechnersystems durch sorgfältige Administration und Zugangskontrolle wirkungsvoll vor unbefugten Zugriffen geschützt werden, boten diese Mechanismen beim Einsatz von Java-Applets keinen wirkungsvollen Schutz mehr. Kritiker bezeichneten Java aus diesem Grund bereits als Einladung für die Programmierung Trojanischer Pferde.

Das eigentliche Thema dieser Arbeit lautete „Portierung der Java Virtual Machine auf das Linux/L4 System unter Ausnutzung spezieller Eigenschaften des L4-Mikrokerns“. Ziel der Arbeit sollte sein, einen Interpreter für Java-Applets auf das Linux/L4 System zu portieren und dabei einen Schutzmechanismus des L4 Mikrokerns für die Kontrolle dieser Applets zu benutzen. Es zeigte sich jedoch recht schnell, daß die hier behandelten Probleme bei weitem nicht auf Java beschränkt sind. Es wird derzeit an verschiedenen Systemen gearbeitet, die zu dem Oberbegriff „Mobile Programmen“ zusammengefaßt werden können [Mil96]. Diese Systeme haben eines gemein; es werden nicht mehr nur Daten zwischen Rechnersystemen ausgetauscht, sondern auch Programmfragmente bzw. vollständige Programme zur Bearbeitung dieser Daten. Diese Arbeit behandelt ein Konzept, mit dem derartige Programme sicher auf einem Rechnersystem ausgeführt werden können.

Spricht man von Sicherheit in Rechnersystemen (im Sinne von *security*), kann man drei allgemeine Schutzziele definieren [Pfi95]:

1. Vertraulichkeit
2. Integrität
3. Verfügbarkeit

Vertraulichkeit behandelt die Frage, wem Daten zugänglich gemacht werden, Integrität ob Daten richtig und vollständig sind und Verfügbarkeit ob der Zugriff auf Daten gewährleistet ist. Im Zusammenhang mit Mobilen Programmen sollen in dieser Arbeit vorwiegend die ersten beiden Problemstellungen behandelt werden. Das Problem der Verfügbarkeit von Daten ist keineswegs uninteressant (als Stichwort sollen hier die *Denial of Service Attacks* dienen), eine Behandlung dieses Themas würde aber den Rahmen dieser Arbeit sprengen. Wird im folgenden von Sicherheit gesprochen, so ist dies unter dem Gesichtspunkt der ersten beiden Schutzziele zu sehen.

Die derzeit bekannten Konzepte zum Schutz von Daten sind aus verschiedenen Gründen für den Umgang mit Mobilen Programmen nicht geeignet:

- sie verbieten generell den Einsatz dieser Programme. Ein Beispiel dafür sind *Firewalls*, diese verhindern bereits das Betreten eines Rechnersystems durch ein solches Programm.
- die Einschränkungen für die Programme sind so groß, daß der Anwendungbereich auf kleine, eher als Spielereien zu bezeichnende Programme beschränkt bleibt. Das von den derzeit verfügbaren Java Interpretern angewendete *Sandboxing*-Prinzip ist dazu zu zählen.
- die Systeme sind allgemein nicht für den Einsatz in großen, heterogenen Systemen geeignet, wie das WWW eines darstellt. Dazu sind viele der derzeit bekannten Forschungssysteme zu zählen, z.B. die *BirliX Security Architecture* (siehe dazu Kapitel 2).

Ausgangspunkt für diese Arbeit sind bereits bekannte Sicherheitskonzepte, die auf *Capability-Listen* basieren (z.B. die Subjektrestriktionen der BirliX Security Architecture). Diese sollen zusammen mit Mechanismen eines modernen Mikrokerns benutzt werden, um eine einfaches, aber effektives System zur sicheren Abarbeitung Mobiler Programme zu entwickeln.

Das folgende Kapitel beschreibt derzeit bekannte Techniken zum Schutz von Rechnersystemen vor unbefugten Zugriffen auf Daten durch Programme. Anschließend daran werden im Kapitel 3 einige Grundlagen vermittelt, die das Verständnis der weiteren Arbeit erleichtern.

Im Kapitel 4 wird ein System zur sicheren Abarbeitung Mobiler Programme entworfen. Das darauf folgende Kapitel behandelt die Erfahrungen bei der Implementierung dieses Systems. Abschließend wird untersucht, welchen Einfluß die Benutzung dieses Systems auf die Ausführungsgeschwindigkeit einer Anwendung hat.

Kapitel 2

Stand der Technik

2.1 Java

Obwohl Java nun nicht mehr direkt Thema dieser Arbeit ist, soll an dieser Stelle etwas näher auf dieses System eingegangen werden, um anzudeuten, wodurch die Sicherheitsprobleme ausgelöst werden.

Java wurde bei Sun Microsystems unter Leitung von James Gosling entwickelt. Die Entwicklung begann 1990 (damals noch unter dem Namen Oak) mit dem Ziel, eine einfache, robuste und portable Programmiersprache für den Einsatz in *embedded systems* zu entwerfen. Mit der Entwicklung des World Wide Web (WWW) ab 1993, eröffnete sich für Java ein komplett neues Anwendungsgebiet. Durch die Eigenschaft der Portabilität eignet sich Java sehr gut für die Erstellung dynamischer Elemente auf WWW-Seiten, den sog. Applets.

Java besteht im wesentlichen aus zwei Komponenten, der Programmiersprache Java und der *Java Virtual Machine*. Die Programmiersprache Java ist an C++ angelehnt (wenn auch stark vereinfacht) und ist der hier eher uninteressante Teil. Die Java Virtual Machine (Java VM) definiert eine hardwareunabhängige Ausführungsumgebung für Java-Anwendungen. Ein Java-Programm wird nicht in eine systemspezifische Binärdatei übersetzt, sondern in einen sog. *Bytecode*. Dieser Bytecode wird von der Java VM bei der Abarbeitung interpretiert. Ein Java-Programm kann auf jeder Hardwareplattform ausgeführt werden, für die eine Implementierung der Java VM existiert. Für einen Großteil der gegenwärtig benutzten Plattformen ist eine derartige Implementierung verfügbar, durch den Einsatz von Java können somit sehr effektiv platformübergreifende Anwendungen erstellt werden.

Applets sind kleine Programme, die in WWW-Seiten eingebettet werden können. Beim Ansehen einer solchen WWW-Seite mit einem geeigneten Browser werden diese Applets auf den lokalen Rechner geladen und ausgeführt. Dies ist der kritische Punkt in Bezug auf die Sicherheit des Rechnersystems. Da Java auch Möglichkeiten zum Zugriff auf Dateisysteme bietet, könnte ein Applet ohne Wissen des Benutzers auf die Daten des Rechners zugreifen und diese über das Netz übertragen. Derartige Applets könnten dann als klassische Trojanische Pferde bezeichnet werden. Um dieses zu verhindern, besitzen alle Implementierungen der Java VM einen „Sicherheitsmanager“ (*Security Manager*). Diese Kontrollieren den Zugriff der Applets auf das System, indem sie Zugriffe auf Daten fast vollständig verhindern. Dieses Vorgehen stellt natürlich eine sehr große Einschränkung der Funktionalität für die Applets dar, die eine sinnvolle Nutzung von Java stark erschwert.

2.2 Heutige Sicherheitsmechanismen

Natürlich ist die Sicherheit in Rechnersystemen nicht erst seit Java Gegenstand der Forschung. Bedrohungen z.B. durch Trojanische Pferde sind seit langem bekannt und es gibt eine Reihe von Ansätzen, diesen entgegenzuwirken.

Der vielleicht bekannteste und dennoch kaum eingesetzte Ansatz zum Umgang mit potentiellen trojanischen Pferden sind regelbasierte Sicherheitssysteme (*mandatory access control, MAC*), wie z.B. das durch D. Elliot Bell und Leonard LaPadula formalisierte Modell in seiner Multics-Interpretation [Bel76]. Es beschränkt den Fluß von Informationen auf der Basis einiger Regeln, deren Einhaltung das zugrundeliegende System durchsetzen muß. In der Praxis zeigte sich jedoch, daß solche Systeme sehr schwer zu administrieren sind. Hinzu kommt, daß bislang relativ wenig solcher regelbasierten Systeme bekannt sind, deren praktische Einsetzbarkeit auch deutlich limitiert ist. Dies gilt insbesondere, wenn sie in Umgebungen angewendet werden sollen, in denen sich Programme in globalen Netzwerken frei bewegen.

Firewalls werden vor allem eingesetzt, um den Zugang zu einem lokalen Netzwerk (z.B. einem Firmennetz) zu kontrollieren. Dazu wird die Kommunikation auf dem Netzwerk an einem zentralen Punkt, z.B. dem Gateway des Netzwerks zum Internet gefiltert. Durch diese Filterung kann die Benutzung einzelner Netzprotokolle, z.B. Telnet oder HTTP, unterbunden werden. Soll die Benutzung Mobiler Programme auf einem Rechner eines lokalen Netzwerkes ermöglicht werden, muß diesen der Zugang zu dem Rechner erlaubt werden, eine Firewall ist damit wirkungslos. Die Verwendung Mobiler Programme erfordert die Isolation eines einzelnen Programms in einem System, gewissermaßen Firewalls innerhalb eines Rechners.

Discretionary Access Control, z.B. durch Zugriffssteuerlisten in UNIX-Systemen, sind ein weitverbreiteter Ansatz zum Schutz von Daten auf lokalen Rechnersystemen. Der Eigentümer einer Datei kann bestimmen, welche Operationen andere Benutzer des Rechnersystems auf dieser Datei ausführen dürfen. Zum Umgang mit Mobilen Programmen könnte etwa für jedes (bzw. eine Gruppe von Mobilen Programmen) ein neuer Nutzer bzw. eine neue Nutzergruppe eingeführt und für diese die Zugriffsrechte entsprechend gesetzt werden. Eine große Anzahl Mobiler Programme in einem System führt jedoch schnell zu einer unübersichtlichen Verteilung der Zugriffsrechte, so daß eine sichere Verwaltung erschwert wird bzw. unmöglich ist, in anderen Worten: Zugriffssteuerlisten skalieren nicht. Allgemein reichen Zugriffssteuerlisten für die Durchsetzung des Prinzips der geringstmöglichen Privilegisierung (*least privilege*) bekanntlich nicht aus [Här93].

Einige Betriebssysteme — die Pioniersysteme sind Hydra [Wul75], CAP [Wil79] und Amoeba [Tan86] — nutzen *Capability-Listen* für die Kapselung von Prozessen. Diese enthalten alle Objekte, auf die ein Prozeß im Moment zugreifen darf. In der BirliX-Sicherheitsarchitektur [Här93, Kow90] erhalten suspekte Programme sogenannte *Subjektrestriktionen*. Diese Mechanismen lassen zwar eine sehr feingranulare Rechtevergabe nach dem Prinzip der geringstmöglichen Rechteausstattung (*least privilege*) zu, besitzen aber einen elementaren Nachteil. Sie sind nur in homogenen Systemen einsetzbar und somit für die Anwendung in oben skizzierten Umgebungen ungeeignet.

Die eben beschriebenen Konzepte stellen allgemeine Sicherheitskonzepte in Betriebssystemen dar. Speziell für den Umgang mit Mobilen Programmen wurden in der letzten Zeit verschiedene Systeme vorgeschlagen:

- Die gegenwärtig verwendeten Systeme zur Abarbeitung Mobile Programme verwenden ein sog. *Sandbox*-Prinzip. Die Programme werden in einem engen, abgegrenzten Rahmen abgearbeitet, der *Sandbox*. Zugriffe über die Grenzen dieses Rahmens hinaus, z.B. auf das lokale Dateisystem, werden durch das System verboten. Dieses generelle Verbot stellt eine sehr starke Einschränkung für die Funktionalität Mobiler Programme dar. Eine Anwendung zum Durchsuchen einer Informationsdatenbank ist auf einem solchen System im Prinzip nicht realisierbar.
- Ein anderer Ansatz schlägt die Einführung von *Trusted Applets* vor [Joh97]. Diese sind signiert, wodurch ein Applet als „vertrauenswürdiges, hochwertiges Programm" gekennzeichnet werden soll, dem Zugriff auf das lokale Rechnersystem gestattet werden kann. Es ist bis jetzt jedoch noch nicht klar, wer diese Signatur vergeben soll.
- Autoren oder Verteilern eines Mobilen Programms signieren dies, so daß sie im Schadensfall zur Rechenschaft gezogen werden können. Viele Sicherheitsverletzungen werden jedoch erst weit nach

der Abarbeitung eines Programms, in einigen Fällen, z.B. bei Vertraulichkeitsverletzungen gar nicht erkannt und können demzufolge keinem konkreten Verursacher zugeordnet werden.

- Das an der University of California in Berkeley entwickelte *Janus*-System führt suspekte Programme unter der Kontrolle eines anderen, *framework* bezeichneten Programms aus [Gol96]. Dazu wird eine eigentlich für Debug-Zwecke gedachte Möglichkeit des *„tracen“* eines Programms im Solaris-Betriebssystem verwendet. Das Framework-Programm wird bei jedem Systemruf des überwachten Programms durch das Betriebssystem davon informiert und kann diesen vor der Abarbeitung kontrollieren und gegebenenfalls unterbinden. Die Kontrolle erfolgt durch sog. *Module*, die der Benutzer vor dem Start des Systems in einer Konfigurationsdatei definiert. Dies ermöglicht jedoch nur eine statische Konfiguration des Systems, zur Laufzeit können keine Änderungen an der Kontrollstrategie vorgenommen werden.

- Bei IBM wird gegenwärtig an dem *FlexxGuard*-System gearbeitet. Es erweitert eine Implementierung der Java VM um die Möglichkeit, Capability-Listen für die Kontrolle der Zugriffe auf Ressourcen des Systems zu benutzen.

  Ein Applet definiert in einer Liste alle Ressourcen, die es für seine Abarbeitung benötigt. Vor der Ausführung wird für jeden Eintrag dieser Liste durch das System geprüft, ob der Zugriff auf diese Ressource gestattet werden soll oder nicht. Dazu kann der Benutzer eine Strategie definieren, die abhängig von dem Autor und der Herkunft des Applets sind. Wurden alle Einträge der Liste erfolgreich geprüft, wird diese als Capability-Liste während der Abarbeitung des Applets verwendet.

  Dieses Konzept erlaubt eine sehr flexible Kontrolle von Applets, für jedes Applet kann bei Bedarf eine eigene Überwachungsstrategie definiert werden. Der Nachteil ist, daß die Kontrolle auf Java-Applets beschränkt ist. Will ein Applet ein lokal installiertes Programm (z.B. einen Editor) starten, muß dieses verboten werden, da das System keine Kontrolle über den Editor hat.

Die beiden zuletzt vorgestellten Systeme besitzen interessante Eigenschaften. Das Janus-System verwendet eine allgemeine Schnittstelle für die Kontrolle, es ist also unabhängig von speziellen Ausführungsumgebungen für Mobile Programme. FlexxGuard bietet über die Verwendung von Capability-Listen eine sehr flexible Möglichkeit, spezielle Sicherheitsstrategien für einzelne Applets zu definieren. Im folgenden soll ein System beschrieben werden, daß diese beiden Eigenschaften verwendet, um eine flexible und allgemein einsetzbare Sicherheitsstrategie zu implementieren.

Kapitel 3

Grundlagen

In diesem Kapitel sollen die Grundlagen der *-Listen in Linux/L4 vermittelt werden.

3.1 *-Listen

Das Konzept der *-Listen bietet die Möglichkeit, für einzelne Mobile Programme spezielle Sicherheitsstrategien zu entwerfen. Für die Überwachung der Ressourcen werden Capability-Listen verwendet, die dynamisch zur Laufzeit erzeugt werden und weitergegeben werden können.

Die prinzipielle Idee soll anhand der allgemeinen, d.h. in verschiedensten Umgebungen einsetzbaren Vorgehensweise der teilnehmenden Parteien erläutert werden.

Zunächst fügt der Hersteller oder Anbieter eines Programms diesem eine Liste symbolischer Namen bei. Diese Liste enthält die Namen aller Betriebsmittel (Dateien, Netzwerkverbindungen, Prozesse, ...), welche das Programm für die Erbringung der ausgewiesenen Funktionalität benötigt. Zusätzlich gibt der Hersteller seinem Programm einen Namen. Das Programm wird zusammen mit dieser Liste und dem Namen durch den Hersteller signiert (es wird dabei davon ausgegangen, daß es einen sicheren und effektiven Weg zur Überprüfung dieser Signatur gibt, z.B. mittels zertifizierter öffentliche Schlüssel). Das Ergebnis dieses Schritts ist ein Programm, erweitert um eine Liste der benötigten Betriebsmittel. Diese Liste wird im folgenden die *Wunsch-Liste* eines Programms genannt.

Der Administrator (bzw. Eigentümer) eines Rechnersystems bestimmt, in welchem Umfang er einem Programm bzw. dessen Hersteller vertraut. Dies geschieht durch die Angabe aller Betriebsmittel, für die Programme eines bestimmten Herstellers Zugriffsrechte besitzen sollen. So kann z.B. Programmen des Herstellers Foo-soft Zugriff auf alle Dateien des Verzeichnisses `~/windows` erlaubt werden, während Shareware Programmen nur Zugriffe auf Dateien im Verzeichnis `~/tmp/otto` gestattet sein sollen. Darüberhinaus können bestimmten Programmen mehr Rechte eingeräumt werden als es alleine durch die seinem Hersteller eingeräumten Rechte besitzen würde. Ergebnis dieses Schritts ist also eine Datenbank, die Herstellern und Programmen Rechte zum Zugriff auf Betriebsmittel zuordnet. Im folgenden wird diese Datenbank *Vertrauens-Liste* genannt.

Soll nun ein Mobiles Programm ausgeführt werden, wird als erstes der Hersteller des Programms ermittelt und die Signatur des Programms und der dazugehörenden Wunsch-Liste überprüft. Dann wird überprüft, ob die zu dem Programm gehörende Wunsch-Liste eine Teilmenge der zu dem Hersteller bzw. Programm gehörenden Vertrauens-Liste ist. Ist dies der Fall, kann das Programm sofort ausgeführt werden, anderenfalls wird dazu eine interaktive Komponente, im weiteren als *Supervisor* bezeichnet, zur Zulässigkeit des Zugriffs befragt. Das Ergebnis dieses Schritts, also die Menge von Ressourcen auf die der das Mobile Programm ausführende Prozeß zugreifen darf, wird im folgenden *Kann-Liste* genannt und stellt eine Variante der eingangs genannten Capability-Listen dar.

Während der Abarbeitung des Programms wird die Kann-Liste (wie klassische Capability-Listen) zur Überprüfung von Zugriffen auf Ressourcen verwendet. Es werden nur Zugriffe erlaubt, für die ein entsprechender Eintrag in der Kann-Liste existiert. Bei Zugriffen auf Ressourcen, für die kein solcher Eintrag existiert, wird eine entsprechende Anfrage an den Supervisor gesendet. Dieser kann den Zugriff erlauben, ablehnen oder die Ressource transparent für das Programm durch eine andere ersetzen.

Startet ein Programm ein weiteres, wird dieses auf die gleiche Weise überprüft. Die neue Kann-Liste wird gebildet, indem die Schnittmenge der Kann-Liste des aktiven Programms, der Vertrauens-Liste und der Wunsch-Liste des neuen Programms gebildet wird. Dadurch wird erreicht, daß ein Programm durch Ausführen eines anderen Programms keine zusätzlichen Rechte erlangen kann; die Zugriffsrechte können so nur geringer werden.

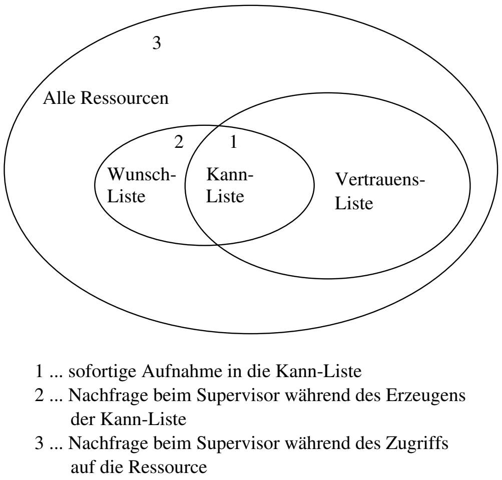

Abbildung 3.1: *-Listen

Einträge in der *-Listen sind symbolische Namen, welche auch Wildcards enthalten können.

Abb. 3.2 zeigt ein Beispiel für die Anwendung der *-Listen. Die Vertrauens-Liste für den Hersteller Foo-soft erlaubt den Zugriff auf alle Dateien ab dem Verzeichnis `/pub/docs`. Die Wunsch-Liste des Programms verlangt den Zugriff auf alle Dateien ab `/pub/docs/techreports`, demzufolge wird `/pub/docs/techreports+` in die Kann-Liste aufgenommen.

Vertrauens-Listen können auch Alias-Einträge enthalten. Beispielsweise enthält die Wunsch-Liste aus Abb. 3.2 den Alias-Namen `DISPLAY`. Dieser Alias-Name wird durch den Eintrag in der Vertrauens-Liste in den Namen `georg:0.0` aufgelöst. Ein wichtiger Einsatzbereich von Alias-Einträgen in Vertrauens-Listen sind Negativeinträge. Mit diesen kann der Zugriff auf einzelne Ressourcen gezielt verboten werden, obwohl dieser z.B. durch einen Wildcard-Eintrag zuvor erlaubt wurde.

3.2 L4

Der Mikrokern L4 wurde von Jochen Liedtke an der Gesellschaft für Mathematik und Datenverarbeitung (GMD) entwickelt [Lie96]. Es ist ein Mikrokern der zweiten Generation, der neben seiner Kompaktheit und hohen Effizienz einen interessanten Mechanismus für die Implementation von Sicherheitskonzepten zur Verfügung stellt – Clans & Chiefs. L4 Tasks können zu Gruppen zusammengefaßt werden, den *Clans*. Jeder Clan besitzt eine *Chief*-Task. Tasks kommunizieren miteinander in L4 über IPC. Versucht ein Mitglied eines Clans eine Kommunikation über die Grenzen seines Clans hinaus, wird diese Kommunikation über dessen Chief-Task umgeleitet (siehe Abb. 3.3).

Der Chief kann diese IPC-Nachricht weiterleiten, verändern oder anderweitig verarbeiten. Dieser Mechanismus arbeitet hierarchisch, d.h eine Nachricht kann auf dem Weg vom Sender zum Empfänger über

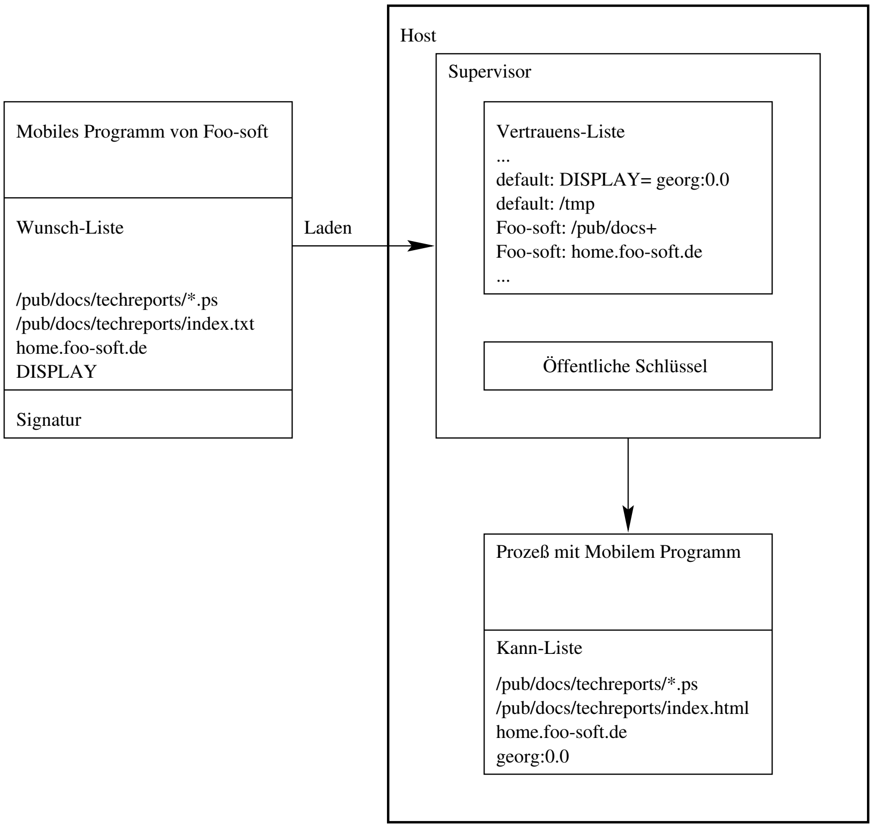

Abbildung 3.2: Ein Beispiel für *-Listen

mehrere Chiefs umgeleitet werden.

Da der Chief eine normale L4-Task ist, erlaubt dieses Konzept die Implementierung von Sicherheitsstrategien ohne eine weitere Unterstützung durch den L4-Kern.

3.3 Linux

Linux ist eine frei verfügbare UNIX-Implementation, die für verschiedene Hardwareplattformen verfügbar ist. Der Linux-Kern ist ein traditioneller monolithischer Kern, der eine Vielzahl an Funktionen zur Benutzung und Verwaltung von Ressourcen wie z.B. Dateisysteme, Netzwerkprotokolle oder auch Rechenzeit und Speicher zur Verfügung stellt. Da das Thema dieser Arbeit die Kontrolle der Zugriffe auf Ressourcen ist, sollen die Ressourcen und Funktionen des Linux-Kerns an dieser Stelle etwas genauer betrachtet werden.

3.3.1 Ressourcen

Die Ressourcen des Linux-Kerns können vereinfacht in zwei Kategorien eingeteilt werden:

- physische Ressourcen
- logische Ressourcen

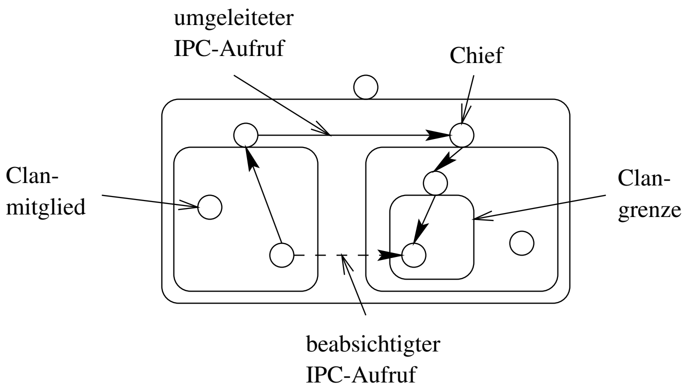

Abbildung 3.3: Clans & Chiefs

Zu den physischen Ressourcen zählen u.a. CPU-Rechenzeit, Hauptspeicher und Übertragungskapazitäten von Bus-Systemen. Die Kontrolle der Benutzung derartiger Ressourcen soll hier nicht weiter behandelt werden, einige Ideen dazu sind im Kapitel 7 aufgeführt.

Bei den logischen Ressourcen lassen sich drei große Gruppen unterscheiden:

1. Prozesse
2. Dateien
3. Netzwerkverbindungen

Prozesse werden in Linux durch Prozessidentifikationsnummern (*Process ID, PID*) bezeichnet. Diese werden bei der Erzeugung eines Prozesses durch den Linux-Kern vergeben.

Zu der Gruppe der Dateien gehören neben den eigentlichen Dateien auch Verzeichnisse, symbolische Links und diverse Spezialdateien für Hardwarekomponenten (z.B. für Terminals oder Festplatten). Die Bezeichnung erfolgt über symbolische Namen. Durch die Verwendung von Verzeichnissen kann eine hierarchische Namensstruktur aufgebaut werden. Symbolische Links enthalten Verweise auf Namen innerhalb dieser Hierarchie.

Die Bezeichnung von Netzwerkverbindungen hängt vom verwendeten Netzwerkprotokoll ab. Eine Verbindung über ein IP-Netzwerk wird durch eine IP-Adresse und eine Port-Nummer identifiziert. Lokale Verbindungen, z.B. die *UNIX Domain sockets*, werden über symbolische Namen analog zu Dateinamen bezeichnet.

3.3.2 Systemrufe

In Linux kann nur der Kern direkt auf die Ressourcen des Systems zugreifen. Der Kern stellt Anwendungsprogrammen eine Schnittstelle zur Verfügung, über diese auf Ressourcen benutzen können, die *Systemrufe*. Die Ausführung eines Systemrufs bewirkt den Übergang aus der Nutzeraktivität in die Kernaktivität des Prozesses[1]. Dieser Übergang wird durch das Auslösen eine speziellen Ausnahme (`int 0x80`) bewirkt. Die Argumente eines Systemrufes werden In den Prozessorregistern übergeben. Die Linuxversion für Intel x86 Prozessoren erlaubt maximal 5 Argumente für einen Systemruf, die in den Registern EBX, ECX, EDX, ESI und EDI übergeben werden. Nach der Abarbeitung des Systemrufes kehrt der Prozeß in die Nutzeraktivität zurück.

[1]Die Abstraktion eines Linuxprozeß in Nutzer- und Kernaktivität wurde zum besseren Verständnis aus der Arbeit von Michael Hohmuth [Hoh96] übernommen. Eine genauere Beschreibung der Linux-Prozeßkonzepts ist in [Bec95] zu finden

3.4 Linux/L4

Linux/L4 ist eine Portierung des Linux-Kerns auf den L4 Mikrokern. Sie wurde von der Betriebssystemegruppe der TU Dresden durchgeführt [Bau96, Hoh96]. Die Abbildung 3.4 zeigt die prinzipielle Struktur der derzeitigen Implementierung.

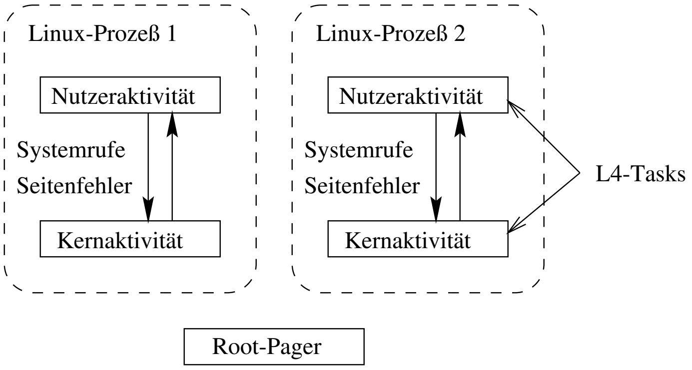

Abbildung 3.4: Struktur der Linux/L4 Portierung

Die Nutzer- und Kernaktivität eines Linux-Prozesses werden in separaten L4-Tasks ausgeführt. Der Übergang zwischen den beiden Aktivitäten erfolgt durch Benutzung von L4-IPC. Seitenfehler der Nutzeraktivität werden von der Kernaktivität behandelt, dazu werden durch den L4-Kern Seitenfehler in IPC-Nachrichten an die Kernaktivität übersetzt. Die Unterscheidung dieser verschiedenen Nachrichten erfolgt durch ein 32-Bit Wort, das Bestandteil beider IPC-Nachricht ist. Bei einem Seitenfehler enthält dieses Argument den Befehlszähler des Befehls, der den Fehler ausgelöst hat. Eine Systemruf-Nachricht ist durch den Wert -2 dieses Arguments gekennzeichnet, dieser Wert stellt keinen gültigen Befehlszähler dar. Die Übergabe der Argumente eines Systemrufes kann nicht mehr direkt in den Registern erfolgen. Linux/L4 benutzt dafür einen von der Nutzer- und Kern-Aktivität gemeinsam benutzten Speicherbereich, die *exklusive Seite*. Für die Taskverwaltung wird eine separate Task, der *Root-Pager*, verwendet.

Kapitel 4

Entwurf

Nachdem im vorherigen Kapitel einige Grundlagen dieser Arbeit erläutert wurden, soll jetzt ein Konzept für die Implementierung der *-Listen in Linux/L4 entworfen werden.

4.1 Ausgangspunkt

Ziel dieser Arbeit ist es, ein Sicherheitskonzept auf Anwendungsebene ohne direkte Unterstützung durch den Betriebssystem-Kern zu implementieren. Dabei kann von folgenden Voraussetzungen ausgegangen werden:

- als Basis dient das Betriebssystem Linux/L4
- die Kapselung eines Programms erfolgt durch die Benutzung des Clans&Chiefs-Konzept des L4-Mikrokerns
- die Kontrolle der Zugriffe auf Ressourcen erfolgt an der Systemrufschnittstelle
- die die Kontrolle ausführende Komponente soll als Anwendungsprogramm implementiert werden, soll also normalen Zugriff auf Ressourcen des System haben
- als Beispiel für eine Sicherheitsstrategie sollen die *-Listen verwendet werden

Der Entwurf soll auch Aspekte der Effizienz und Anwendungsfreundlichleit berücksichtigen, da diese entscheidende Faktoren für den praktischen Einsatz eines solchen Systems darstellen.

4.2 Implementierung einer Sicherheitsstrategie

Obwohl die Grobstruktur des Entwurfs durch die Aufgabenstellung bereits definiert ist, sollen an dieser Stelle einige allgemeine Probleme bei der Implementierung eines Sicherheitskonzepts diskutiert werden.

Konzepte zum Schutz von Ressourcen vor unbefugten Zugriffen können auf verschiedene Weise umgesetzt werden. Denkbar sind u.a.:

- Ein Programm erhält nur Zugriff auf „virtuelle“ Ressourcen. Diese werden beim Zugriff in die tatsächlichen Ressourcen durch das System umgesetzt. Dieses hat somit die vollständige Kontrolle über alle Zugriffe eines Programms. Zum Beispiel könnte einem Programm anstelle eines vollständigen Dateisystems ein virtuelles, unter Benutzung von symbolischen Links aufgebautes Dateisystem vorgetäuscht werden, das nur die Dateien enthält, auf die das Programm zugreifen darf. Die Umsetzung der virtuellen in die tatsächlichen Dateien würde dann durch die Auflösung der symbolischen Links geschehen.

Derartige Systeme lassen sich jedoch meistens nur schwer implementieren. Das skizzierte Beispiel scheitert z.B. an speziellen Eigenschaften symbolischer Links.

- Die Rechte zum Zugriff auf Ressourcen werden zur Laufzeit nach Bedarf vergeben. Ein Programm besitzt beim Start nur die Rechte, die unmittelbar für die Ausführung benötigt werden, z.B. Rechte zum Benutzen der Systembibliotheken. Der erste Zugriff auf eine andere Ressource führt zu einem Fehler. Als Reaktion auf diesen Fehler kann das System über die Zulässigkeit dieses Zugriffes entscheiden und die Rechte des Programms entsprechend erweitern.

  Der Nachteil dieser Vorgehensweise liegt vor allem in der schwierigen Administration, insbesondere wenn damit größere Systeme geschützt werden sollen. Weiterhin stützen sich diese Systeme auf Mechanismen, die das Betriebssystem zur Verfügung stellt. Diese sind jedoch meistens nicht ausreichend um eine derartige Vorgehensweise vernünftig umzusetzen.

- Zugriffe auf Ressourcen werden aktiv kontrolliert. Aktive Kontrolle bedeutet, daß Zugriffe auf Ressourcen zur Laufzeit durch ein geeignetes Werkzeug gefiltert werden. Eine derartige Kontrolle kann auf verschiedenen Wegen realisiert werden:

  – sie kann Bestandteil der Ausführungsumgebung für bestimmte Programme sein, zum Beispiel eines Interpreter des Java-Bytecodes. Ein derartiger Ansatz wird z.B. von dem FlexxGuard-System verfolgt. Diese Systeme bieten die Möglichkeit, die Überprüfung direkt an die Gegebenheiten der jeweiligen Umgebung anzupassen, wodurch im allgemeinen eine effiziente Implementierung möglich wird. Der Nachteil ist allerdings, daß nur Programme der jeweiligen Ausführungsumgebung überwacht werden können, andere Programme erfordern eine neue Implementierung des Kontrollwerkzeuges.

  – die Kontrolle erfolgt an einer allgemeinen Schnittstelle des Systems, z.B. der Systemrufschnittstelle. Dieser Variante erfordert meistens eine aufwendigere Implementierung, ist aber nicht an eine bestimmte Gruppe von Programmen gebunden. Eine Umsetzung kann an verschiedenen Stellen einer Betriebssystemearchitektur erfolgen. Während eine Integration in den Betriebssystemkern eine effiziente Implementierung ermöglicht, bietet eine Realisierung auf Anwendungsebene mehr Flexibilität bzgl. Änderungen der Überwachungsstrategie.

Die in dieser Arbeit behandelte Technik ist ein Beispiel für die letztgenannte Vorgehensweise.

4.3 *-Listen in Linux/L4

Nachdem im vorhergehenden Abschnitt einige allgemeine Aspekte der Implementierung einer Sicherheitsstrategie diskutiert wurden, soll jetzt die Struktur der Implementierung von *-Listen in Linux/L4 entworfen werden. Ziel dieses Entwurfs ist es, ein beliebiges Programm auf einem Rechner so zu isolieren, daß alle Zugriffe des Programms auf Systemressourcen kontrolliert und gegebenenfalls unterbunden werden können. Der Entwurf muß zwei Problemkreise berücksichtigen:

1. Isolation des Programms
2. Umsetzung der *-Listen zur Kontrolle der Zugriffe auf Ressourcen

4.3.1 Kapselung in Linux/L4

Wie in Abschnitt 3.4 beschrieben, werden für die Nutzer- und und Kernaktivität eines Linux-Prozeß getrennte L4-Tasks verwendet. Die prinzipielle Idee zur Kapselung eines Programms besteht darin, diese beiden Tasks durch einen Clan voneinander zu trennen. Die Task der Nutzeraktivität soll innerhalb des Clans liegen, die der Kernaktivität außerhalb. Dadurch wird erreicht, daß jegliche Kommunikation zwischen den beiden Aktivitäten über den Chief des Clans umgeleitet wird. Der Chief hat so die Möglichkeit, alle Zugriffe auf das System zu überwachen und gegebenfalls zu unterbinden, indem es einen Systemruf des Nutzers nicht an die Kernaktivität weiterleitet.

Struktur der Kapselung

Als erstes soll die Frage behandelt werden, wieviele derartige Clans es in dem System geben soll.

Eine erste Möglichkeit ist, alle überwachten Programme in genau einem Clan zusammenzufassen. Der Vorteil dieser Variante ist der geringe Verbrauch an Ressourcen, es wird nur eine Chief-Task benötigt. Diese muß allerdings für den Umgang mit mehreren Programmen ausgelegt sein, ein gegenseitiges Blockieren von Programmen muß ausgeschlossen werden. Die Chief-Task kann die Kommunikation der Nutzeraktivitäten mit den zugehörigen Kernaktivitäten überwachen, IPC-Nachrichten zwischen Nutzeraktivitäten können jedoch nicht kontrolliert werden. Dadurch können zwei kooperierende Programme ihre Rechte austauschen, ohne daß dies durch die Chief-Task unterbunden werden kann, die Durchsetzung unterschiedlicher Rechte für einzelne Programm ist daher nicht möglich.

Aus diesem Problem resultiert der nächste Ansatz. Ein Clan enthält nur noch Programme, die mit denselben Rechten ausgeführt werden. Das System besteht demzufolge in der Regel aus mehreren Clans. Mit dieser Struktur ist eine sichere Abarbeitung der Programme möglich, sie stellt jedoch sehr hohe Anforderungen an die Implementierung. Diese resultieren daraus, daß sich die Rechte eines Programms zur Ausführungszeit ändern können (vgl. Abschnitt 3.1), wodurch Tasks zur Laufzeit aus einem Clan entfernt und in einen neuen einefügt werden müssen. Dies ist jedoch nicht direkt möglich, da die Clan-Hierarchie nicht dynamisch erzeugt und verändert werden kann. Sie wird durch den L4-Kern festgelegt, wird eine neue Task durch eine andere erzeugt, wird die erzeugende Task automatisch Chief der neuen Task. Es sind Lösungen dieses Problems denkbar, diese erfordern jedoch eine umfangreiche Unterstützung durch das Betriebssystem und sollen daher hier nicht weiter behandelt werden.

Um die dynamische Änderung der Clan-Hierarchie zu vermeiden, kann für jedes Programm eine separater Clan verwendet werden. Eine Änderung der Rechte dieses Programms hat dann keine Auswirkungen auf andere Programme. Der offensichtliche Nachteil dieser Variante ist der recht hohe Verbrauch an Ressourcen, pro überwachtem Programm wird mindestens eine weitere L4 Task, der Chief, benötigt. Andererseits ermöglicht diese Struktur eine flexible und sichere Kapselung der Programme und wird daher als Grundlage für den weiteren Entwurf verwendet.

Umsetzung der Struktur in Linux/L4

In diesem Abschnitt soll die Umsetzung der eben beschriebenen Struktur diskutiert werden. Für die Bewertung möglicher Alternativen sind zwei Kriterien von Bedeutung:

1. Systemrufe in Linux übergeben ihre Argumente (z.B. Dateinamen) zum Teil als Zeiger in den Adreßraum der Nutzeraktivität. Damit derartige Argumente ausgewertet werden können, muß ein Chief Zugriff auf diesen Adreßraum haben[1].

2. Wie wird die Taskstruktur der Variante beim Erzeugen eines neuen Linux-Prozeß aufgebaut.

Bevor die einzelnen Varianten besprochen werden, soll an dieser Stelle auf die Besonderheiten der Taskverwaltung in Linux/L4 eingegangen werden, da diese beim Entwurf berücksichtigt werden müssen.

[1] Prinzipiell ist eine Lösung dieses Problems auch durch eine Veränderung der Systemruf-IPC des Linux/L4 Systems denkbar, indem die Argumente direkt in der IPC-Nachricht übergeben werden. Der Aufwand dafür erscheint sehr groß, da sowohl Änderungen an der Systemruf-IPC als auch Änderungen an der Systemrufschnittstelle des Linux-Kerns notwendig sind. Aus diesem Grund soll dieser Ansatz hier nicht weiter betrachtet werden.

Damit eine Task eine neue anlegen kann, muß sie das Recht dazu besitzen. Diese Rechte werden von einem *Root-Server* verwaltet, der durch den L4 Kern beim Starten des System erzeugt wird. Dieser Root-Server kann einer Task das Recht übertragen, indem er eine *inaktive* Task erzeugt. Eine inaktive Task belegt keine Ressourcen, ihr wird jedoch die Task, der das Recht zum Anlegen gewährt werden soll, als Chief zugewiesen. Eine Task kann dann eine neue anlegen, indem sie diese inaktive Task zum Anlegen der aktiven benutzt. Ein Chief kann das Recht zur Erzeugung einer Task an ein Mitglied seines Clans weitergeben, indem er wiederum die Task als Chief der inaktiven Task angibt. Dieser Mechanismus soll eine hierarchische Struktur der Clans garantieren[Lie96].

In Linux/L4 wird der Root-Server durch den *Root-Pager* (vgl. Abb. 3.4 auf Seite 11) implementiert. Dieser legt die Tasks der Kern- und Nutzeraktivität eine Linux-Prozeß an. Demzufolge sind alle Tasks des Linux/L4 Systems direkte Mitglieder des Clans des Root-Pagers.

Um die oben beschriebene Struktur in diese System zu integrieren, sind zwei Ansätze denkbar:

1. **Benutzung einer einzelnen Chief-Task**

   Bei diesem Ansatz wird eine einzelne Chief-Task zwischen die Tasks der Kern- und Nutzeraktivität eine Linuxprozeß geschoben. In der Abbildung 4.1 ist die Taskstruktur dieser Variante zu sehen.

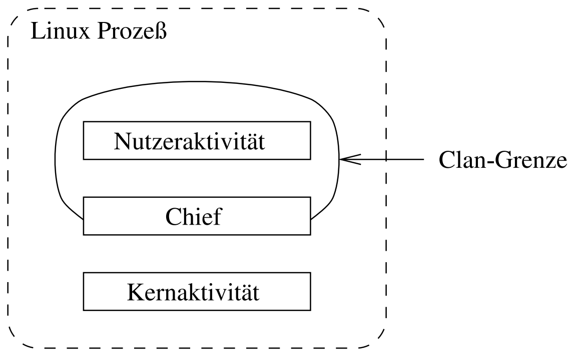

Abbildung 4.1: Eine einzelne Chief-Task

   Diese Struktur erfordert, daß die Kernaktivität zwei verschiedene Nutzeraktivitäten (das Anwendungsprogramm und den Chief) gleichzeitig bedienen kann. Die Linux/L4 Implementierung läßt eine solche Struktur jedoch nicht zu. Dieses Problem kann umgangen werden, indem der Chief alle Aktionen der Anwendung unter seiner Identität durchführt.

   Ein überwachter Linux-Prozeß kann folgendermaßen erzeugt werden:

   - Die Kernaktivität erzeugt eine neues Task-Paar
   - Die für die Nutzeraktivität vorgesehene Task wird als Chief-Task verwendet. Sie läßt sich von dem Root-Pager das Recht zur Erzeugung einer neuen Task geben und legt mit diesem die Task an, in der die Nutzeraktivität ausgeführt wird.

   Für den Zugriff des Chiefs auf den Adreßraum des Anwendungsprogramm ist eine Struktur denkbar, in der dieser in die Task des Chiefs eingeblendet wird. Abbildung 4.2 stellt den Aufbau der Adreßräume der einzelnen Tasks bei dieser Vorgehensweise dar.

   Der Adreßbereich zwischen 3 und 4 Giga-Byte wird von L4 verwendet. Unterhalb davon liegt in der Task der Nutzeraktivität ein Bereich, der durch das Linux/L4 System benutzt wird. Dieser ist ein halbes GB groß. Der Rest des Adreßraums (2.5 GB) steht einem Anwendungsprogramm zur Verfügung. Die Kernaktivität realisiert den Zugriff auf den Adreßbereich des Anwendungsprogramms, indem dieser in den Adreßraum der Kernaktivität eingeblendet wird.

   Die oben beschriebene Idee zum Zugriff des Chiefs auf den Adreßraum des Anwendungsprogramms läßt sich umsetzen, indem der Chief in einem genau definierten Bereich seines Adreßraumes aus-

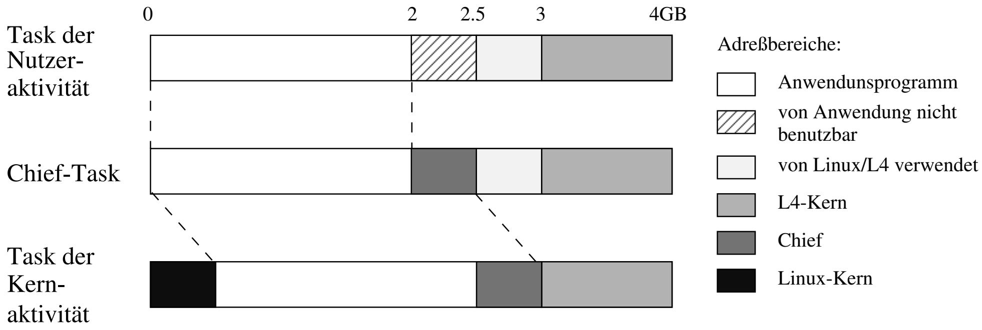

Abbildung 4.2: Aufbau der Adreßräume

geführt wird (in der Abbildung zwischen 2 und 2.5 GB). Der restliche Bereich wird dazu verwendet, den Adreßraum des Anwendungsprogramms, der auf 2 GB begrenzt wird, einzublenden.

Bei der Umsetzung dieser Alternative sind einige Probleme zu berücksichtigen:

- um diese Struktur aufzubauen, muß die Programmdatei des Chiefs so erzeugt werden, daß der Chief beim Start an die entsprechende Position im Adreßraum geladen wird.
- der Start eines Anwendungsprogramms kann nicht mehr durch den Linux-Kern erfolgen. Der Kern entfernt vor der Ausführung eines neuen Programms sämtliche Objekte des alten Programms aus dem Adreßraum. Dadurch wird die oben beschriebene Aufbau zerstört. Stattdessen muß zur Ausführung eines neuen Programms der Chief neu gestartet werden, der dann seinerseits für den korrekten Start des Anwendungsprogramms verantwortlich ist. Dies hat zur Folge, das der Chief für alle auf dem System verwendeten Binärformate entsprechende Laderoutinen zur Verfügung stellen muß.
- für die Ausführung einiger Funktionen benutzt die Kernaktivtät Informationen über die Lage des Programms im Adreßraum, die sie beim Start des Programms erhält. Da für die Kernaktivität der Chief das ausgeführte Programm ist, muß dieser sicherstellen, das alle derartigen Operationen den Anwendungsprogramms abgefangen und durch entsprechend modifizierte Operationen ersetzt werden.

Diese Probleme resultieren daraus, daß die Kernaktivität zwei Anwendungen behandeln muß, ohne daß sie das explizit unterstützt. Aus diesem Umstand heraus ergibt sich die zweite Variante für die Implementierung der Kapselungsstruktur.

2. **Schachtelung zweier Linux-Prozesse**

In dieser Variante besitzt der Chief eine eigene Kernaktivität, kann von diesem Standpunkt aus also als eine normaler Linux-Prozeß angesehen werden. Abbildung 4.3 zeigt die daraus resultierende Taskstruktur.

Dabei wird ein in einem Linux-Prozeß 1 laufendes Programm durch einen in einem Linux-Prozeß 2 laufenden Chief gekapselt. Die Verwendung eines derartige Prozeßpaares löst die oben beschriebenen Probleme, wird allerdings zwei neue Frage auf:

- Wie wird diese Struktur aufgebaut?
  Wie bereits beschrieben, werden in Linux/L4 alle Tasks durch den Root-Pager angelegt. Will ein Prozeß einen neuen erzeugen, sendet die Kernaktivität eine entsprechende Anforderung an diesen, der nacheinander die Tasks der neuen Kern- und Nutzeraktivität anlegt. Um die oben beschriebene Struktur anzulegen, muß diese Vorgehensweise geändert werden:
  - bei der Erzeugung eines neuen Anwendungs-Prozeß wird zuerst eine neuer Chief-Prozeß angelegt. Dazu kann die gerade beschriebene Mechanismus benutzt werden.
  - der neue Chief läßt sich das Recht zur Erzeugung eine neuen Task geben[2].

[2]Linux/L4 bietet dazu eine Schnittstelle über das `/proc`-Dateisystem an.

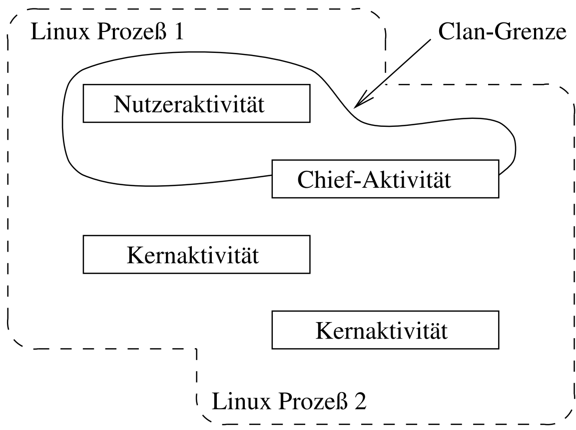

Abbildung 4.3: Zwei geschachtelte Linux-Prozesse

  – Die ID dieser inaktiven Task wird der Kernaktivität des alten Prozeß als Parameter für die Erzeugung des neuen Prozeß übergeben. Durch den Root-Pager wird dann nur noch die Task der Kernaktivität angelegt.
  – Nachdem die Kernaktivität angelegt wurde, kann der neue Chief-Prozeß die Task der Nutzeraktivität anlegen.

  Dieses Vorgehen erfordert die Implementierung eines neuen Systemrufes, der als Argument die ID einer Task erwartet und mit dieser ID einen neuen Linux-Prozeß erzeugt, wobei nur die Kernaktivität angelegt wird.

- Wie kann der Chief-Prozeß auf den Adreßbereich des Anwendungsprozeß zugreifen?
  Die Vorgehensweise der ersten Variante läßt sich hier nicht mehr anwenden. Der Hauptgrund dafür ist, daß die Adreßräume von verschieden Kernaktivitäten, und damit Pagern, verwaltet werden. Würde beim Zugriff des Chiefs auf eine Seite des Adreßraumes der Anwendung ein Seitenfehler auftreten, würde dieser Fehler an den Pager des Chief-Prozesses geleitet, der diesen jedoch nicht behandeln kann. Stattdessen müßte der Fehler durch den Pager des Anwendungsprozeß behandelt werden.

  Der Chief kann auf den Adreßraum des Programms zugreifen, indem er sich bei Bedarf einzelne Seiten von der Kernaktivität des Programms geben läßt. Dazu täuscht er dieser durch das Senden einer entsprechenden IPC-Nachricht einen Seitenfehler auf die entsprechende Seite vor. Die Nachricht muß der Nachricht entsprechen, die der L4-Kern beim Auftreten eines realen Seitenfehlers erzeugen würde.

Beide Varianten stellen einen gangbaren Weg für die Kapselung eines Programms dar. Gegenüber dem Original-Linux/L4 System wird der Aufwand für eine Nachricht der Nutzeraktivität an die Kernaktivität verdoppelt. Anstelle der direkten Kommunikation Nutzer $\rightarrow$ Kern $\rightarrow$ Nutzer wird eine Kommunikation Nutzer $\rightarrow$ Chief $\rightarrow$ Kern $\rightarrow$ Chief $\rightarrow$ Nutzer durchgeführt.

Die erste Variante verzichtet weitestgehend auf Änderungen am Linux/L4 System. Dafür erfordert sie die Implementierung von Routinen zum Laden und Starten von Binärdateien durch den Chief. Der Chief muß ebenfalls den Teil des Linux-Kerns nachbilden, der Informationen über die Lage eines Programms im Adreßraum benötigt.

Die zweite Variante erfordert die Implementierung eines neuen Systemrufes für die spezielle Vorgehensweise bei der Erzeugung eines neuen Prozeß. Ein Großteil dieser Implementierung (der Aufbau des Eintrags in der Prozßtabelle, Erzeugung der Kernaktivität) kann jedoch aus dem Original-Linux/L4 System übernommen werden. Der Aufwand für den Zugriff auf den Adreßraum des Programms ist höher als in Variante 1, da eine Seite erst explizit angefordert werden muß.

Für die Implementierung wurde die zweite Variante gewählt. Zwar sind die Kosten für den Zugriff auf den Nutzeradreßraum höher als in Variante 1, jedoch fallen die Kosten nur dann an, wenn eine Überprüfung eines Systemrufes durchgeführt wird. In Variante 1 fallen die Kosten für die Nachbildung einiger Teile der Funktionalität des Linux-Kerns generell an. Desweiteren erschien zum Zeitpunkt dieser Entscheidung die Implementierung der zweiten Variante mit weniger Problemen verbunden zu sein als die der ersten.

4.3.2 Kommunikation zwischen Kern- und Nutzeraktivität

Im letzten Abschnitt wurde die Struktur für die Kapselung eines Programms in Linux/L4 entworfen. Daran anschließend sollen jetzt die verschiedenen IPC-Nachrichten zwischen der Nutzer- und der Kernaktivität behandelt werden, die über den Chief umgeleitet werden.

Von der Task der Nutzeraktivität gehen zwei Arten von IPC-Nachrichten aus:

- Seitenfehler-Nachrichten
  Eine Nutzeraktivität löst einen Seitenfehler aus, indem sie auf einen Speicherbereich zugreift, für die keine gültige Seite in ihren Adreßraum eingeblendet ist. Ein Seitenfehler wird durch den L4-Kern behandelt, indem er mit der Identität dieser Task eine Nachricht an den Pager sendet, der dieser Task zugeordnet ist. Im Linux/L4 System wird dieser Pager durch die Kernaktivität bereitgestellt. Diese Nachricht besteht aus zwei Argumenten, der Adresse an der der Seitenfehler aufgetreten ist und den Befehlszähler des Befehls, der auf diese Adresse zugegriffen hat. Durch den Pager wird dann die entsprechende Seite in den Adreßraum der Task eingeblendet.

- Ausnahme-Nachrichten
  Durch eine Ausnahme wird die Abarbeitung des Programms unterbrochen, welches diese ausgelöst hat. Derartige Ausnahmesituationen können gewollt sein, z.B. wird durch die Operation `int 0x80` bei der Ausführung eines Systemrufes eine solche Situation hervorgerufen. Ausnahmen werden aber auch durch Fehler im Programm, z.B. einer Division durch Null, ausgelöst[3]. Zur Behandlung einer Ausnahme ruft der L4-Kern eine Funktion auf, die durch das Programm in einer Tabelle (der *Interrupt Descriptor Table, IDT*) spezifiziert wird. Ein Programm kann durch dieses Verfahren geeignete Routinen zur Behandlung von Ausnahmen definieren.

  In Linux/L4 werden Ausnahmen einer Anwendung also der Task der Nutzeraktivität zugestellt. Um diese zu behandeln, enthält diese Task eine Emulationsbibliothek, die beim Auftreten einer Ausnahme aufgerufen wird[4]. Damit die Kernaktivität die Ausnahme bearbeiten kann, wird an diese durch die Emulationsbibliothek eine IPC-Nachricht mit entsprechendem Inhalt gesendet. Um diese Nachricht von einer Seitenfehler-Nachricht zu unterschieden, enthält das Argument des Befehlszählers einen ungültigen Wert (-2). Der Typ der Ausnahme ist in einem Fehlerkode gespeichert, der nicht in der IPC-Nachricht, sondern über einen von der Emulationsbibliothek und dem Kern gemeinsam genutzten Speicherbereich übergeben wird.

Im Linux/L4 System werden Signale behandelt, indem an die Nutzeraktivität des Empfängerprozesses eine IPC-Nachricht gesendet wird. In der Nutzeraktivität wird parallel zu der Anwendung ein Thread ausgeführt, der für die Behandlung dieser Nachrichten verantwortlich ist. Dieser Thread unterbricht die Abarbeitung des Programms und simuliert eine Ausnahmesituation. Die Behandlung dieser Ausnahme erfolgt wie oben beschrieben, durch die Aktivierung des Linux-Kerns kann das Signal behandelt werden.

Damit die korrekte Abarbeitung eines Programms gewährleistet bleibt, muß der Chief diese Schnittstelle nachbilden:

- Signal-Nachrichten können direkt an den Signal-Thread der Nutzeraktivität weitergereicht werden.

[3]Genau genommen stellt ein Seitenfehler ebenfalls eine Ausnahme dar, diese wird jedoch bereits durch den L4-Kern gesondert behandelt.

[4]Dazu werden in der IDT der Nutzeraktivität die entsprechenden Behandlungsroutinen eingetragen

- Eine Seitenfehler-Nachricht wird an die Kernaktivität weitergeleitet. Durch diese wird daraufhin eine Seite in den Adreßraum des Chiefs eingeblendet[5]. Der Chief muß für diese Seite einen geeigneten Zielbereich festlegen, so daß es keine Konflikte beim Einblenden der Seite gibt. Wurde eine Seite erfolgreich in den Adreßraum des Chiefs eingeblendet, muß dieser diese an die Nutzeraktivität weiterreichen.

- Die Behandlung von Ausnahme-Nachrichten ist umfangreicher. Um zu ermitteln, ob ein Systemruf Ursache für eine Ausnahme ist, muß der Chief den Fehlerkode der Ausnahme auswerten. Dazu benötigt er den Zugriff auf den von der Emulationsbibliothek zur Übergabe benutzten Speicherbereich (der *exklusiven Seite*). Dies ist recht einfach möglich, da der Bereich an einer festen Adresse im Adreßraum der Nutzeraktivität liegt und somit Seitenfehler auf diese Adresse überprüft werden können. Bei einem Zugriff auf diese Seite durch die Nutzeraktivität kann diese dann auch in den Adreßraum des Chiefs eingeblendet werden. Durch die Vorgehensweise bei der Erzeugung eines neuen Prozeß durch Linux/L4 ist sichergestellt, daß auf diesen Bereich vor dem eigentlichen Start des neuen Prozeß zugegriffen wird, die Seite also rechtzeitig für den Chief verfügbar ist. Die exklusive Seite wird durch den Linux-Kern als Kern-Speicher angefordert. Dadurch wird verhindert, daß diese aus dem Speicher verdrängt wird, es können also auch keine Seitenfehler beim Zugriff auf diese auftreten.

  War ein Systemruf Ursache für die Ausnahme, kann der Chief diesen Systemruf überprüfen. Die dazu notwendigen Informationen (Nummer des Systemrufs und dessen Argumente) werden durch die Emulationsbibliothek ebenfalls auf der exklusiven Seite gespeichert. Wird der Systemruf durch den Chief gestattet, kann die Nachricht an die Kernaktivität gesendet werden. Soll der Systemruf unterbunden werden, kann der Chief die Nachricht unbearbeitet an die Nutzeraktivität zurücksenden.

  Der Chief hat auch die Möglichkeit, die Argumente eines Systemrufes zu verändern. Dazu benötigt er allerdings einen Speicherbereich im Adreßraum der Nutzeraktivität. Diesen kann er sich durch die Kernaktivität des Anwendungsprozeß reservieren lassen, indem er dieser einen entsprechenden Systemruf vortäuscht.

  Alle anderen Ausnahmen können direkt an die Kernaktivität weitergereicht werden.

4.3.3 *-Listen

In den letzten beiden Abschnitten wurden die Grundlagen für die Implementation einer Sicherheitsstrategie geschaffen. Diese sollen jetzt angewendet werden, um die *-Listen in Linux/L4 zu integrieren.

Nachdem die Frage nach der Isolation eines Programms geklärt ist, verbleiben für den Entwurf der Integration 3 Fragestellungen die zu beantworten sind:

1. Welche Systemrufe des Programms müssen Abgefangen werden?

2. Wie werden die verschiedenen Listen verwaltet?

3. Wie wird die interaktive Komponente in das System integriert?

Linux-Systemrufe

Linux (Version 2.0.21) kennt 164 verschiedene Systemrufe. Diese können hier nicht alle einzeln aufgeführt werden, eine genaue Beschreibung der Systemrufe ist im Abschnitt 2 der Linux-Manpages und in [Bec95] zu finden. Grob kann eine Aufteilung der Systemrufe in 6 Gruppen vorgenommen werden:

[5] Das Einblenden von Speicherseiten in den Adreßraum einer Task stellt in L4 einen Spezialfall von IPC dar. Demzufolge wird diese Operation entsprechend der Clan-Struktur umgeleitet.

- **Datei-Verwaltung**
  Zu dieser Gruppe gehören eine Reihe Systemrufe. Neben den offensichtlich dazu gehörenden Systemrufe wie `open`, `read`, `write` oder `close` sind auch die Systemrufe zu Arbeit mit Verzeichnissen (`chdir`), Links (`symlink`) oder die Systemrufe `fcntl` und `ioctl` zu zählen, die der Verwaltung spezieller Eigenschaften einer Datei bzw. eines Gerätes dienen. Für die Bezeichnung der Datei, auf die die entsprechende Operation angewendet werden soll, gibt es zwei Möglichkeiten:

  – Angabe eines Dateinamens
    Der Name einer Datei bezieht sich auf eine symbolische Namenshierarchie. Ein Verzeichnis repräsentiert einen Knoten innerhalb dieser Hierarchie, eine Datei ein Blatt. Dateinamen können absolut zur Wurzel oder relativ zu einem Knoten dieser Hierarchie angegeben werden. Der Linux-Kern wertet einen solchen Dateinamen schrittweise aus, bis er die durch den Namen bezeichnete Datei (genau genommen die *Inode*) gefunden hat. Jeder Zugriff auf eine Datei über einen Dateinamen erfordert es, daß der Chief diesen Dateinamen in der Kann-Liste des Programms sucht. Um eine einheitliche Basis für die Suche zu haben, müssen relativ angegebene Dateinamen vorher in absolute Dateinamen überführt werden.

  – Verwendung eines Dateideskriptors
    Die Datei wird über einen Deskriptor bezeichnet. Dieser verweist direkt auf die Verwaltungsstruktur der Datei im Linux-Kern, so daß die aufwendige Auswertung eines Dateinamens entfallen kann. Diese Struktur muß jedoch aufgebaut werden, z.B. durch einen `open`-Systemruf. Wird die Datei durch die entsprechende Operation nur gelesen, kann daher eine erneute Überprüfung der Datei erfolgen. Bei Schreiboperationen muß lediglich die Größe der zu schreibenden Daten kontrolliert werden, um die maximale Größe einer Datei einzuhalten.

Linux bietet die Möglichkeit zur Verwendung von symbolischen Links. Diese sind spezielle Dateien, die einen Verweis auf die eigentliche Datei enthalten. Durch symbolische Links können Querverweise innerhalb der Namenshierarchie angelegt werden.

Symbolische Links bereiten bei der Auswertung eines Dateinamens beträchtliche Probleme. Für die Umwandlung relativer Dateinamen in absolute muß der Chief Kenntnis des aktuellen Arbeitsverzeichnisses besitzen. Dieses erhält er als symbolischen Namen durch einen `chdir`-Systemruf. Im Linux-Kern verwendet jedoch nicht den symbolischen Namen des Verzeichnisses, sondern dessen Inode. Diese erhält er durch eine schrittweise Auflösung des Namens. Ein Wechsel in das Eltern-Verzeichnis durch einen `chdir("..")` Aufruf bewirkt daher einen Wechsel in das physische Elternverzeichnis, nicht in das Verzeichnis, das den symbolischen Link enthält. Eine derartige Situation ist in Abbildung 4.4 dargestellt[6].

```
lars@Obelix:/home/lars/test$ ls -l
lrwxrwxrwx lars  users   14 Feb 4 16:17 doc -> /usr/local/doc
lars@Obelix:/home/lars/test$ cd doc
lars@Obelix:/home/lars/test/doc$ cd ..
lars@Obelix:/usr/local$
```

Abbildung 4.4: Ein Beispiel für symbolische Links

Diese Verhalten macht es erforderlich, daß der Chief Dateinamen auf symbolische Links überprüft, um relative Namen korrekt auswerten zu können. Dieser Test ist recht aufwendig, da er eine schrittweise Auswertung des Namens von der Wurzel her erfordert. Um die Kosten dieses Tests zur Laufzeit des Programms so gering wie möglich zu halten, ist folgende Vorgehensweise denkbar:

– Die in der Kann-Liste eines Programms gespeicherten Dateinamen enthalten keine Links. Links können bei der Erzeugung der Kann-Liste, also vor dem Start des Programms, aufgelöst werden.

[6] Die Abbildung zeigt das Verhalten der `tcsh`-Shell. Einige Shells (z.B. die `bash`) lösen Links auf, um diese Verhalten zu vermeiden.

- Ein als Argument eines Systemrufs übergebener Dateiname wird erst auf Links überprüft, wenn er nicht in der Kann-Liste gefunden wurde. Dadurch entstehen für normale Dateinamen keine zusätzlichen Kosten.
- Der Chief merkt sich den tatsächlichen Namen des aktuellen Arbeitsverzeichnis. Links müssen gegebenenfalls aufgelöst werden. Dadurch wird gewährleistet, das relativ angegebene Dateinamen korrekt ausgewertet werden können.

Durch diese Vorgehensweise wird erreicht, daß die Kosten zur Auswertung symbolischer Links nur entstehen, wenn diese tatsächlich angewendet werden. Die Kosten können vermieden werden, indem der Nutzer und der Administrator auf die Verwendung symbolischer Links verzichten.

- `execve`**-Systemruf**
  Dieser Systemruf wird meistens in die vorhergehende Gruppe eingeordnet. Da er jedoch für die Verwaltung der *-Listen eine besondere Relevanz besitzt, soll er hier getrennt behandelt werden. Durch den `execve`-Systemruf wird eine neue Programmdatei für einen Prozeß bestimmt. Bevor die neue Programmdatei gestartet werden kann, müssen die Rechte des Prozeß entsprechend des in Abschnitt 3.1 beschriebenen Verfahrens neu bestimmt werden. Um die Einhaltung der neu bestimmten Rechte zu garantieren, ist es notwendig, Dateien oder Sockets, auf die der Prozeß nach Start des neuen Programms keine Zugriffsrechte mehr besitzt, zu schließen.

- **Socket-Verwaltung**
  Sockets werden zur Verwendung von Netzwerkverbindungen benötigt, können aber auch zur lokalen Kommunikation zwischen Prozessen verwendet werden (z.B. benutzt diese das XWindow System). In Linux können Sockets eine Reihe von Netzwerkprotokollen bearbeiten (z.B. Novell IPX oder Macintosh Appletalk), in dieser Arbeit werden jedoch nur *UNIX Domain Name*-Sockets und IP-Sockets betrachtet. Domain-Name-Sockets werden für lokale Kommunikation verwendet, sie erhalten Dateinamen als Argumente. IP-Sockets erhalten zwei Argumente, die IP-Adresse des Kommunikationspartners und eine Portnummer, über die die verschiedenen Kommunikationsdienste (Telnet, FTP, ...) ausgewählt werden können.

  Domain-Name-Sockets werden u.a. durch das XWindow System für die Kommunikation zwischen dem X-Server und einer Anwendung benutzt. Wird einer Anwendung die Benutzung eines solchen Sockets gestattet, hat diese uneingeschränkten Zugriff auf den X-Server. Wird dieser auch von anderen Anwendungen benutzt, können sich diese gegenseitig beeinflussen, eine Anwendung kann zum Beispiel den Inhalt eines Fensters einer anderen auslesen. Dieses kann durch die Verwendung eines *X-Gateways* [Kah95] oder eines *nested X-Servers* verhindert werden.

- **Prozeß-Verwaltung**
  Zu dieser Gruppe gehören die Systemrufe zum Anlegen (`fork`, `clone`) und Löschen (`exit`) eines Linux-Prozeß. Desweiteren gehören die Systemrufe zur Verwaltung der Prozeß, Nutzer und Gruppen-ID eines Prozeß und zur Behandlung von Signalen in diese Gruppe.

- **Speicher-Verwaltung**
  Diese Gruppe enthält die Systemrufe zur dynamischen Verwaltung Adreßraums eines Programms. Dazu gehört die Vergrößerung des BSS-Segments eines Prozeß durch den `brk`-Systemruf und das Einblenden einer Datei durch den `mmap`-Systemruf. Dieser verlangt die Angabe des Dateideskriptors einer Datei, muß also nicht weiter behandelt werden (siehe dazu den Abschnitt über die Dateiverwaltung).

- **Der ganze Rest**
  Zu dieser Gruppe gehören z.B. die Systemrufe für die Beeinflussung der Schedulingstrategie des Linux-Kerns oder zur Initialisierung des Systems.

Für die Überwachung eines Programms sind vor allem die ersten 4 Gruppen von Bedeutung.

Verwaltung der *-Listen

Wie im Abschnitt 3.1 beschrieben, muß das System 3 verschiedene Listen verwalten:

- **Wunsch-Liste**
  Die Wunsch-Liste eines Programms wird nur zum Aufbau der Kann-Liste des Programms benötigt und muß daher nicht zur Laufzeit des Programms verfügbar sein.

- **Kann-Liste**
  Die Kann-Liste eines Programms dient als Grundlage für die Überprüfung eines Zugriffs auf eine Ressource durch den Chief. Da jedes Programm eine eigene Kann-Liste besitzt, kann diese direkt im Chief, der das Programm überwacht, gespeichert werden.

  Um die Abarbeitung eines Systemrufes durch die Suche in dieser Liste so wenig wie möglich zu verzögern, wird diese im Chief als Hash-Tabelle implementiert.

- **Vertrauens-Liste**

  Die Vertrauens-Liste wird benötigt, um vor dem Start eines neuen Programms die Kann-Liste dieses Programms zu bestimmen. Daher ist es notwendig, daß alle Chiefs des Systems Zugriff auf diese Liste haben. Dieser Zugriff kann prinzipiell auf zwei verschiedenen Wegen erfolgen:

  – Verwendung von gemeinsam benutzten Speicher (*shared memory*)
    Jeder Chief hat die Vertrauens-Liste in seinen Speicherbereich eingeblendet, kann somit direkt auf diese zugreifen. Für das Einblenden dieses Speicherbereich müßte allerdings der durch den Linux-Kern zur Verfügung gestellte System V IPC-Mechanismus verwendet werden. Der L4-Mechanismus zum Einblenden eines Speicherbereiches kann nicht benutzt werden, da dann Seitenfehler auf diesen Bereich nicht korrekt behandelt werden könnten. Desweiteren erfordert die Verwendung von gemeinsam benutztem Speicher einen Synchronisationsmechanismus zur Verhinderung von gleichzeitigen Zugriffen.

  – Verwaltung durch einen separaten Prozeß, Zugriff über IPC
    Die Vertrauens-Liste wird durch einen separaten Prozeß, einen *Supervisor*, verwaltet. Will ein Chief auf die Liste zugreifen, sendet er eine entsprechende Anforderung an den Supervisor. Für diese Anforderung kann der L4 IPC-Mechanismus verwendet werden, der wesentlich effizienter als die durch Linux bereitgestellten Mechanismen (lokale Sockets, System V IPC) ist.

  Für die Implementierung wurde die zweite Variante gewählt, da diese eine einfache Verwaltung der Vertrauens-Liste ermöglicht.

Integration des Supervisors

Der im letzten Abschnitt eingeführte Supervisor kann neben der Verwaltung der Vertrauens-Liste noch für eine Reihe weiterer Aufgaben benutzt werden:

- Nachfragen bei unerlaubtem Zugriff auf Ressourcen
  Wird durch ein Programm auf eine Ressource zugegriffen, die nicht in der Kann-Liste dieses Programms enthalten ist, soll der Administrator (bzw. die Person, die das Programm überwacht ausführt) davon informiert und über die weitere Behandlung des Zugriffs befragt werden. Aus Gründen der Benutzerfreundlichkeit empfiehlt es sich, diese Anfragen von einem zentralen Punkt des Systems aus für alle überwachten Programme durchzuführen. Als Ergebnis einer solchen Anfrage kann eine Ressource auch in die Vertrauens-Liste eingetragen werden. Durch die Integration in den Supervisor ist dies auf direktem Weg möglich.

- Aufbau der Kann-Listen
  Eine neue Kann-Liste muß aufgebaut werden, wenn ein überwachtes Programm ein anderes Programm ausführen will. Dazu ist der Zugriff auf die Kann-Liste des alten Programms, die Wunsch-Liste des neuen Programms und die Vertrauens-Liste notwendig (vgl. Abschnitt 3.1). Eine Integration dieser Funktion in den Supervisor bedeutet, daß der Chief-Prozeß seine Kann-Liste an den Supervisor überträgt. Dazu kann L4-IPC verwendet werden, diese erlaubt die Übertragung von maximal 6 MB Daten. Die neue Kann-Liste wird auf dieselbe Art an den Chief-Prozeß gesendet.

  Prinzipiell kann die neue Kann-Liste auch durch den Chief-Prozeß aufgebaut werden. Dieser müßte dazu alle Einträge für ein Programm in der Vertrauens-Liste vom Supervisor anfordern, bzw. jede Ressource einzeln durch den Supervisor überprüfen lassen. Diese Vorgehensweise bringt in der Praxis jedoch keinerlei Vorteile.

- Überprüfung von Signaturen
  Vor der Ausführung eines neuen Programms muß ebenfalls dessen Signatur überprüft werden, z.B. durch ein externes Programm wie PGP. Während der Abarbeitung dieses Programms ist der Supervisor blockiert[7], dies bedeutet, daß während der Überprüfung einer Signatur keine weiteren Anfragen von Check-Prozessen bearbeitet werden können. In der Praxis dürfte dies jedoch keinen großen Nachteil darstellen, da die Überprüfung einer Signatur mit geeigneten Werkzeugen sehr effizient und damit schnell ist.

4.4 Zusammenfassung der Entwurfsentscheidungen

In diesem Kapitel wurden verschiedene Ansätze zur Implementierung der *-Listen in Linux/L4 diskutiert. Abbildung 4.5 stellt die Struktur dar, die anhand der Ergebnisse dieser Diskussion für die Implementierung gewählt wurde.

Für die Überwachung werden 2 Linux-Prozesse ineinander geschachtelt. Die Task der Nutzeraktivität des Programms liegt in einem Clan, der Chief dieses Clans ist die Nutzeraktivität eines zweiten Prozesses, dieser wird im weiteren als *Check-Prozeß* bezeichnet. Die Vertrauens-Liste wird in einem gesonderten Supervisor-Prozeß verwaltet. Anfragen an diesen Supervisor werden über L4-IPC Aufrufe realisiert.

[7] Es ist auch eine nicht-blockierende Lösung denkbar, diese würde den Aufwand für die Implementierung des Supervisors jedoch unverhältnismäßig erhöhen.

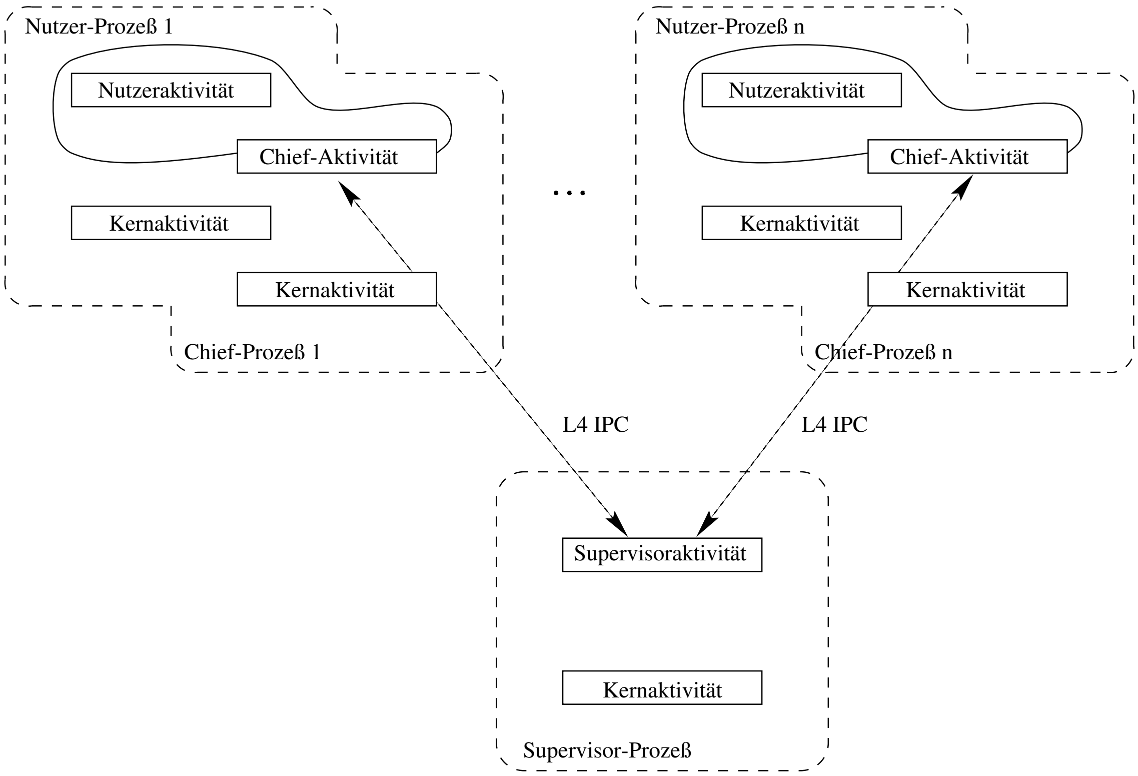

Abbildung 4.5: Gesamtentwurf

Kapitel 5

Implementierung

Der in Kapitel 4 beschriebene Aufbau zur Überwachung eines Programms wurde fast vollständig implementiert. Es werden derzeit noch nicht alle Linux-Systemrufe überwacht. Die Vervollständigung der Kontrolle ist jedoch ohne Probleme möglich, die Schnittstellen dafür sind bereits umgesetzt.

Für die Implementierung wurde eine C++ Klassenhierarchie gewählt. Diese besteht aus den Klassen:

- `IPCInterception`
  Diese Klasse realisiert die Funktionen zum Empfangen und Weiterleiten der umgeleiteten IPC-Nachrichten durch einen Chief. Dabei wird von einem Client-Server-Modell ausgegangen, eine Kommunikation besteht aus zwei IPC-Nachrichten, einer Anforderung eines Clients an einen Server und dessen Antwort. Eine empfangene Anforderung wird einer Behandlungsroutine (`handle_client_message`) übergeben, die den Inhalt der Nachricht beurteilen und entscheiden kann, ob die Nachricht weitergegeben werden soll oder nicht. Eine Antwort durch den Server wird ebenfalls einer Behandlungsroutine (`handle_server_message`) übergeben, die diese allerdings nur auswerten kann, die Nachricht wird generell an den Client weitergeleitet. Die beiden Behandlungsroutinen besitzen in dieser Klasse keinerlei Funktionalität. Soll eine konkrete Funktionalität implementiert werden, müssen diese Routinen durch eine abgeleitete Klasse überschrieben werden.

- `SyscallInterception`
  Um die IPC-Nachrichten zwischen der Kern- und Nutzeraktivität auszuwerten, überschreibt diese Klasse die Behandlungsroutinen der Superklasse `IPCInterception`. In der Methode `handle_client_message` werden die Nachrichten in Seitenfehler und Systemrufe unterschieden und entsprechende Methoden zu ihrer Auswertung aufgerufen. Seitenfehler werden direkt an die Kernaktivität weitergegeben, Systemrufe werden genauer ausgewertet. Dazu wird für jeden Systemruf eine eigene Methode aufgerufen, die die Argumente des Systemrufs entsprechend der Linux-Systemrufschnittstelle übergeben bekommt. Die Namen dieser Methoden werden durch die Vorschrift `pre_`*`systemrufname`* gebildet, für einen `open`-Systemruf wird zum Beispiel die Methode `pre_open(const char *filename, int flags, int mode)` aufgerufen. Analog dazu werden die Antworten der Kernaktivität behandelt. Eine Seite, die als Antwort auf einen Seitenfehler an die Nutzeraktivität gesendet werden soll, wird sofort weitergegeben. Die Antwort auf einen Systemruf wird zuerst an eine entsprechende Methode, z.B. `post_open(int fd)` übergeben und danach an die Nutzeraktivität gesendet.

  In den einzelnen `pre_`-Methoden können die Argumente eines Systemrufes ausgewertet und über die Zulässigkeit des Systemrufs entschieden werden. Die `post_`-Methoden können die Ergebnisse der Systemrufe auswerten. In der Klasse `SyscallInterception` besitzen diese Methoden keine Funktionalität, Systemrufe werden generell an die Kernaktivität weitergeleitet[1]. Soll ein Systemruf überwacht werden, kann eine abgeleitete Klasse die Methoden dieses Systemrufs überschreiben.

[1] Ausnahmen sind die Methoden für die Behandlung der `fork` und `exit` Systemrufe. Diese implementieren die Funktionalität zum Auf- bzw. Abbau der Taskstruktur.

- `SLists`
  Diese Klasse wird von der Klasse `SyscallInterception` abgeleitet, um eine konkrete Überwachungsstrategie zu implementieren. Dazu werden für alle überwachten Systemrufe die entsprechenden `pre_`- und `post_`-Methoden überschrieben.

Die Verwendung eine Klassenhierarchie ermöglicht eine einfache Implementierung anderer Ideen. So kann zum Beispiel eine andere Sicherheitsstrategie mit dieser Hierarchie implementiert werden, indem analog zur der `SLists`-Klasse eine Klasse von `SyscallInterception` abgeleitet wird und die entsprechenden Methoden überschrieben werden.

Im folgenden sollen einzelne Teile der Implementierung näher betrachtet werden.

5.1 Aufbau der Taskstruktur

Für den Aufbau der in Abbildung 4.3 dargestellten Taskstruktur wurde der Systemruf `fork_by_chief` implementiert. Die Funktionalität konnte größtenteils von dem `fork`-Systemruf übernommen werden, anstelle eine Task für die Nutzeraktivität anzulegen wird lediglich die als Argument übergebene ID dieser in die Taskstruktur eingetragen. Um einem Check-Prozeß das Anlegen der Task der Nutzeraktivität zu ermöglichen, werden die dafür benötigten Daten (Startadresse, Adresse des Stacks, ID des Pagers) in eine Struktur der exklusiven Seite eingetragen[2].

Bei der Benutzung diese Systemrufes sind zwei Situationen zu unterscheiden:

- Starten des ersten überwachten Programms
  Für die Etablierung der Überwachung wird ein einzelner Check-Prozeß gestartet. Dieser benutzt den `fork_by_chief`-Systemruf, um eine überwachte Kopie von sich selbst zu erzeugen. In diesem Prozeß wird dann das zu überwachende Programm durch einen `exec`-Systemruf gestartet.

- `fork`-Systemruf eines überwachten Programms
  Wird von einem Check-Prozeß ein `fork`-Systemruf eines Programms erkannt, wird zunächst ein neuer Check-Prozeß durch einen normalen `fork`-Aufruf erzeugt. Der neue Prozeß läßt sich das Recht zur Erzeugung einer neuen L4-Task geben und sendet die ID dieser Task an den alten Check-Prozeß. Dieser ersetzt den `fork`-Aufruf des Programms durch einen `fork_by_chief`-Aufruf mit dieser ID. Die auf der exklusiven Seite gespeicherten Ergebnisse dieses Systemrufs werden dem neuen Prozeß gesendet, damit dieser die Task der Nutzeraktivität anlegen kann.

5.2 Zugriff auf den Nutzeradreßraum

Der Zugriff auf den Adreßraum wird an 2 Stellen benötigt:

1. zur Auswertung von Systemruf-Argumenten
   Die Argumente eines Systemrufes werden teilweise als Zeiger in den Adreßraum der Nutzeraktivität übergeben (z.B. ist der Dateiname bei einem `open`-Systemruf ein Zeiger auf eine Zeichenkette im Nutzeradreßraum). Für die Auswertung dieser Argumente wird der Kernaktivität ein Seitenfehler der Nutzeraktivität vorgetäuscht, indem der Check-Prozeß eine Seitenfehlernachricht an diese sendet. Die Kernaktivität beantwortet diese Nachricht mit dem Senden der Seite, auf die sich der Zeiger bezieht. Erstreckt sich das Argument über mehrere Seiten, müssen entsprechend viele Seitenfehlernachrichten gesendet werden.

[2] Diese Vorgehensweise ergab sich aus dem Umstand, daß zum Zeitpunkt der Implementierung des `fork_by_chief`-Systemrufs der Check-Prozeß keine andere Möglichkeit zum Zugriff auf den Adreßraum der Anwendung hatte als über die exklusive Seite. In einer überarbeiteten Version kann für die Übergabe ein Puffer verwendet werden, der dem `fork_by_chief`-Systemruf als zusätzliches Argument übergeben wird.

2. zum Ändern von Systemrufargumenten
   Soll ein Argument eines Systemrufes vor der Abarbeitung durch die Kernaktivität geändert werden, benötigt der Check-Prozeß einen Puffer im Adreßraum der Nutzeraktivität, der für die Übergabe des neuen Arguments benutzt wird. Dieser Puffer wird angelegt, indem der Check-Prozeß einen `mmap`-Systemruf an die Kernaktivität des Anwendungsprogramms sendet. Durch diesen wird ein Speicherobjekt im Adreßraum der Anwendung angelegt. Dieser Speicherbereich wird durch den Check-Prozeß in seinen eigenen Adreßraum eingeblendet. Seitenfehler bei Zugriffen auf diesen Bereich werden durch Sperren des Bereichs mittels des `mlock`-Systemrufs verhindert.

5.3 *-Listen

5.3.1 Interne Darstellung

Wie erwähnt, werden die Kann- und Vertrauens-Listen als Hashtabellen verwaltet. Die Kann-Liste ist als einfache Hashtabelle implementiert. Die Vertrauens-Liste wird in zwei Stufen realisiert, die erste Stufe enthält für jeden Hersteller bzw. für jedes Programm genau einen Eintrag. In diesem Eintrag wird auf eine zweite Liste verwiesen, in der die Namen der Ressourcen gespeichert sind, auf die ein Programm des Herstellers bzw. das konkrete Programm zugreifen darf.

In diesen Listen werden zwei Arten von Einträgen gespeichert:

1. Dateinamen
   Neben dem Dateinamen enthalten diese Einträge die Operationen die auf die Datei ausgeführt werden können (Lesen, Schreiben, Ausführen, Erzeugen) und die maximale Größe, die die Datei bei Schreibzugriffen erreichen darf. Um die Größe der Datei bei einem Schreibzugriff bestimmen zu können, wird in der Kann-Liste zusätzlich die aktuelle Position des Schreibzeigers der Datei gespeichert.

2. IP-Adressen
   IP-Adressen werden als 32-Bit Werte gespeichert, die Port-Nummer als 16-Bit Wert. Eine Beschränkung der Operationen gibt es für IP-Adressen nicht.

Die Hash-Funktion wird nach der Vorschrift

$$H(c) = h(c_n) \bmod M, \quad h(c_i) = h(c_{i-1})^2 + c_i, \quad h(c_0) = c_0, \quad c = c_0 c_1 \ldots c_n$$

berechnet. Eine Zeichenkette c wird dadurch auf einen Tabellenindex im Bereich *0 ... M-1* abgebildet. Kollisionen beim Zugriff auf die Hashtabelle werden durch *lineares Austesten* beseitigt.

Für die Berechnung der Hash-Funktion einer IP-Adresse werden die 6 Byte des entsprechenden Eintrags (4 Byte IP-Adresse + 2 Byte Port-Nummer) als Zeichenkette interpretiert.

5.3.2 Externe Repräsentation

Der Supervisor liest die Vertrauensliste aus einer Textdatei. Für jeden Eintrag in der Vertrauensliste existiert eine Zeile in dieser Textdatei. Abbildung 5.1 zeigt zwei Beispieleinträge einer solchen Datei.

```
V:foo-soft:I:www.foo-soft.de:80
P:bash:F:/etc/profile:1:0
```

Abbildung 5.1: Ein Beispiel für die Vertrauens-Liste

Die einzelnen Elemente eines Eintrags werden durch einen Doppelpunkt voneinander getrennt. Durch das erste Element wird festgelegt, ob sich der Eintrag auf einen Hersteller oder eine spezielles Programm bezieht, das zweite Element enthält den Namen dieses Herstellers bzw. dieses Programms. Der Typ der Ressource wird durch das dritte Element bestimmt. Für eine Datei (Typ `F`) werden neben dem Namen die zulässigen Operationen in Form einer Bitmaske (fünftes Element) und die maximale Größe der Datei bei Schreibzugriffen (sechstes Element) angegeben. Als mögliche Operationen können Lesen, Schreiben, Ausführen und Erzeugen angegeben werden. Ein Eintrag für eine IP-Adresse (Typ `I`) enthält die Adresse des Hosts in Form eines Domain-Namens. Dieser wird durch den Supervisor durch eine Anfrage an einen DNS-Server in eine IP-Adresse übersetzt. Das fünfte Element bezeichnet die Port-Nummer.

Die Wunsch-Liste eines Programms wird in einem ähnlichen Format wie die Vertrauens-Liste dargestellt, es fehlen jedoch die Elemente für den Namen des Herstellers bzw. des Programms. Die Wunsch-Liste wird an die Programmdatei angehängt, Abbildung 5.2 stellt den Aufbau einer solchen Datei dar.

Abbildung 5.2: Aufbau einer Programmdatei

Die Signatur wird über die Wunsch-Liste und das Programm berechnet, dazu wird des Programm *PGP* benutzt. Die Kennung wird verwendet, um ein derartiges Programm bei einem `exec`-Systemruf zu erkennen. Der Header enthält Angaben über die Lage und Größe der Wunsch-Liste und der Signatur.

5.4 Supervisor

Durch den Supervisor werden zwei Funktionen realisiert:

- Nachfragen bei unerlaubten Zugriffen
  Wird durch einen Check-Prozeß ein Zugriff auf eine Ressource entdeckt, für die keine Eintrag in der Kann-Liste des Programms existiert, sendet dieser eine L4-IPC Nachricht an den Supervisor. Diese Nachricht enthält neben dem Namen der Ressource den Namen des Programms und des Herstellers dieses Programms. Diese Daten werden dem Benutzter des Supervisors angezeigt. Dieser hat die Möglichkeit, den Zugriff zu verbieten, einmalig oder generell zu erlauben. Wird der Zugriff generell erlaubt, wird ein neuer Eintrag in die Vertrauens-Liste eingefügt.

  Für die Anzeige wird derzeit eine einfach Terminal-Ein-/Ausgabe verwendet. In einer nächsten Version kann dazu eine XWindow basierte Oberfläche verwendet werden.

- Bestimmen neuer Kann-Listen, Überprüfen von Signaturen
  Eine neue Kann-Liste muß bei einem `exec`-Systemruf eines überwachten Programms bestimmt werden. Dazu sendet der Check-Prozeß dieses Programms eine L4-IPC Nachricht an den Supervisor, die die Kann-Liste des alten Programms und den Namen des neuen Programms enthält. Für die Übertragung der Kann-Liste wird diese in eine Darstellung analog der externen Repräsentation der Wunsch-Liste überführt.

  Die neue Kann-Liste wird in folgenden Schritten erzeugt:

  - die Signatur des neuen Programms wird überprüft
  - die Wunsch-Liste des Programms eingelesen
  - für jedes Element der Wunsch-Liste wird überprüft, ob für dieses ein Eintrag in der Vertrauens-Liste und in der Kann-Liste des alten Programms existiert. Trifft eine dieser Bedingungen nicht zu, wird der Benutzer des Supervisors wie bereits oben beschrieben befragt.

  Konnte die neue Kann-Liste erfolgreich aufgebaut werden, wird diese an den Check-Prozeß zurück gesendet und die Ausführung des neuen Programms erlaubt.

5.5 Anmerkungen

Die derzeitige Implementierung ist funktionstüchtig, sie enthält aber einige Ungereimtheiten:

- Umgehung von Fehlern des L4-Kerns
  Während der Programmierung wurden einige Fehler des L4-Kerns entdeckt. Den größten Einfluß auf die Implementierung hat dabei eine nicht korrekt funktionierende Umleitung der IPC-Nachrichten über den Chief. Aus diesem Grund werden derzeit alle Nachrichten der Nutzeraktivität direkt an den Check-Prozeß gesendet, dazu werden beim Anlegen dieser Task die IDs der Kernaktivität und des Pagers entsprechend geändert.

- Signalbehandlung
  Während ein Systemruf durch den Check-Prozeß überprüft wird, kann dieser keine weiteren Nachrichten empfangen. Wird während dieser Zeit durch eine Kernaktivität ein Signal an das Anwendungsprogramm gesendet, wird dieses theoretisch über den Chief umgeleitet. Somit würde das Signal verloren gehen, da Signale mit einem Timeout von 0 gesendet werden. Durch den oben beschriebenen Fehler des L4-Kerns wird die Nachricht derzeit direkt an das Anwendungsprogramm gesendet. Funktioniert die Umleitung über den Check-Prozeß korrekt, muß dieser sicherstellen, das kein Signal verlorengeht. Gegebenfalls muß er dazu Signale zwischenspeichern.

- Eingriff in die Synchronisation zwischen Nutzer- und Kernaktivität
  Die gegenwärtige Implementierung des Linux/L4 Systems verwendet einen recht komplizierten Mechanismus zur Synchronisierung der Kern- und Nutzeraktivität. Um z.B. einen Seitenfehler zur Auswertung der Argumente eines Systemrufs vorzutäuschen, muß der Check-Prozeß derzeit in diesen Mechanismus eingreifen. Inwieweit durch diesen Eingriff die Stabilität des Systems beeinflußt wird, wurde noch nicht weiter untersucht. Da sich der Synchronisationsmechanismus in einer nächsten Version der Linux/L4 Portierung jedoch wesentlich vereinfacht, soll dieses Problem im Moment nicht weiter behandelt werden.

Kapitel 6

Leistungsbewertung

Für die Bewertung der Implementierung wurden die Abarbeitungszeiten einiger ausgewählter Systemrufe jeweils mit und ohne der Verwendung der *-Listen gemessen und gegenübergestellt. Um einen genaueren Überblick über die benötigte Zeit für die Überprüfung eines Systemrufes zu erhalten, wurden im Anschluß daran Teilfunktionen der Implementierung separat ausgemessen.

6.1 Testdurchführung

Die Test wurden auf einem PC mit einem Pentium-Prozessor (133 MHz), 64 MB Arbeitsspeicher, NE2000-Netzwerkkarte und einer SCSI-Festplatte durchgeführt. Für die Zeitmessung wurde der *Time Stamp Counter* des Pentium-Prozessors verwendet. Dieser 64-Bit Zähler wird bei jedem Takt um 1 erhöht und kann mit dem `rdtsc`-Assemblerbefehl ausgelesen werden.

Im einzelnen wurden folgende Tests durchgeführt:

- Seitenfehler
  Für die Bestimmung der Dauer einer Seitenfehlerbehandlung wurde eine Seite durch den L4-Systemruf `l4_fpage_unmap` aus dem Adreßraum des Testprogramms explizit ausgeblendet und danach auf diese Seite zugegriffen und die Dauer dieses Zugriffs gemessen.

- `getpid`-Systemruf
  Der `getpid`-Systemruf wird nicht überwacht. Durch die Messung sollten die Kosten die generell durch die Umleitung entstehen ermittelt werden.

- `open`-Systemruf
  Der `open`-Systemruf wurde beispielhaft für die Verwendung von Dateinamen in *-Listen verwendet. Durch die Messung sollte die Verzögerung bestimmt werden, die für einen erlaubten Aufruf durch das Auswerten des Dateinamens und das Suchen in der Kann-Liste entstehen.

- `connect`-Systemruf
  Analog zu den Dateinamen wurde der `connect`-Systemruf für die Auswertung von IP-Adressen benutzt. Für den Test wurde eine lokale Verbindung (IP-Adresse 127.0.0.1) zum Port 7 aufgebaut. Dieser sendet die Anforderung zurück und schließt die Verbindung wieder (Echo).

6.2 Meßergebnisse

Die Tabelle 6.1 enthält die Meßergebnisse für die oben beschriebenen Tests. Die Werte für die benötigten Takte wurden durch 10000 Einzelmessungen ermittelt. Die angegebenen Zeiten wurden aus den benötigten

Takten bei einer Taktfrequenz von 133 MHz berechnet. Zum Vergleich wurden die selben Tests ohne die Verwendung von *-Listen durchgeführt.

	Linux/L4 mit *-Listen		Linux/L4 ohne *-Listen		Vergleich	
	Takte	μs	Takte	μs	Δ	relativ
Seitenfehler	5500	41	3400	27	2100	162%
`getpid`-Systemruf	3100	23	1700	13	1400	182%
`open`-Systemruf	36400	274	18800	141	17600	194%
`connect`-Systemruf	45300	341	29600	223	15700	153%

Tabelle 6.1: Ausgewählte Meßergebnisse

Im Anschluß an diese Tests wurden die einzelnen Schritte bei der Auswertung eines Systemrufs separat gemessen. Tabelle 6.2 enthält die Ergebnisse für einen `open`-Systemruf, Tabelle 6.3 für einen `connect`-Systemruf.

	Takte	μs	Anteil an der Gesamtzeit
Umleitung über Check-Prozeß	1400	11	4%
Beschaffen des Dateinamens	5500	41	15%
Umwandlung relativer in absolute Dateinamen	4000	30	11%
Suche in Kann-Liste	1100	8	3%
Bearbeitung durch Linux-Kern	18800	141	52%
Seitenfehler auf exklusive Seite	5500	41	15%

Tabelle 6.2: Behandlung eines `open`-Systemrufs

	Takte	μs	Anteil an der Gesamtzeit
Umleitung über Check-Prozeß	1400	11	3%
Beschaffen der Argumente	8300	62	18%
Suche in Kann-Liste	500	4	1%
Bearbeitung durch Linux-Kern	29600	223	65%
Seitenfehler auf exklusive Seite	5500	41	12%

Tabelle 6.3: Behandlung eines `connect`-Systemrufs

6.3 Interpretation der Meßergebnisse

Die gemessenen Werte stellen keine Überraschung dar, weder im negativen noch im positiven Sinn. Die bei der Implementierung gesammelten Erfahrungen liesen Werte in diesen Bereichen erwarten.

Im folgenden sollen die einzelnen Werte näher betrachtet werden, um mögliche Ansätze für Verbesserungen zu identifizieren.

- `getpid`-Systemruf:
  Die Differenz von 1400 Takten bei der Abarbeitung eines `getpid`-Systemrufs wird durch den Mehr-

aufwand bei der Umleitung der IPC-Nachricht über den Check-Prozeß verursacht. Die Umleitung erfordert zwei zusätzliche L4-IPC Aufrufe und die Überprüfung, ob der Systemruf ausgewertet werden soll oder nicht. Diese Verzögerung betrifft alle Systemrufe, unabhängig davon, ob diese durch den Check-Prozeß ausgewertet werden oder nicht.

- Seitenfehler:
  Da auch Seitenfehler-IPC-Nachrichten umgeleitet werden, entstehen auch hier die gerade beschriebenen Kosten. Zusätzlich dazu muß vom Check-Prozeß ein Bereich für das temporäre Einblenden der Seiten in seinem Adreßraum verwaltet werden. In der derzeitigen Implementierung ist diese Verwaltung recht aufwendig, da der *Grant*-Mechanismus des L4-Kerns nicht korrekt funktioniert. Die Verwendung dieses Mechanismus vereinfacht die Behandlung eines Seitenfehlers deutlich, so daß an dieser Stelle eine Verbesserung zu erwarten ist.

- `open`-Systemruf:
  Tabelle 6.2 enthält die Zeiten, die die einzelnen Schritte der Behandlung eines `open`-Systemrufs benötigen. Am auffälligsten ist die Behandlung des Seitenfehlers auf die exklusive Seite. Diese wird vor der Überprüfung auf Nur-Lesen gesetzt, um eine Veränderung der Argumente durch die Anwendung zu verhindern. Nach der Abarbeitung des Systemrufs schreibt die Emulationsbibliothek jedoch auf diese Seite, so daß dieser Seitenfehler ausgelöst wird.

  Ein weitere Ansatzpunkt für eine Verbesserung ist die Umwandlung relativer in absolute Dateinamen. Gegenwärtig wird dazu an einigen Stellen das dynamische Anlegen von Puffern benötigt.

- `connect`-Systemruf:
  Bei der Behandlung von IP-Adressen entfällt der Aufwand für die Umwandlung absoluter in relative Dateinamen. Ebenso ist die Suche in der Kann-Liste schneller, da die Berechnung der Hash-Funktion schneller erfolgen kann als bei Dateinamen. Der Aufwand für die Beschaffung der Argumente ist jedoch höher, da zweimal auf den Adreßraum des Anwendungsprogramms zugegriffen werden muß. Dies ist durch den Aufbau der Argumente dieses Systemrufs bedingt.

Es sind also eine Reihe von Verbesserungen denkbar, so daß eine effizientere Implementierung möglich erscheint.

Kapitel 7

Weiterführende Arbeiten

7.1 Anwendung der *-Listen auf physische Ressourcen

Gegenstand dieser Arbeit war die Überwachung von Zugriffen auf logische Ressourcen (Dateien, Netzwerkverbindungen) eines Systems. Die Überwachung physischer Ressourcen (Hauptspeicher, CPU-Zeit) ist aber nicht weniger interessant. Zum einen sind auch Angriffe auf diese Ressourcen bekannt (*Denial of Service Attacks*), zum anderen besitzt die Beschränkung der Benutzung physischer Ressourcen in den so genannten QoS-Architekturen eine große Bedeutung. Gegenstand einer weiteren Arbeit kann sein, wie die *-Listen in Verbindung mit dem *Preempter*-Mechanismus des L4-Mikrokerns für diese Zwecke eingesetzt werden können.

7.2 Arbeiten an der Implementierung

Die derzeitige Implementierung kann an verschiedenen Stellen überarbeitet werden:

- Einarbeitung der Fehlerkorrekturen am L4-Kern
  Wie bereits im Abschnitt 5.5 vermerkt, enthält die Implementierung derzeit an verschiedenen Stellen Ungereimtheiten, die aus Fehlern des L4-Kerns resultieren. Sind diese Fehler beseitigt, kann an den entsprechenden Stellen die eigentliche Funktionalität korrekt implementiert werden.

- Integration in eine neue Linux/L4-Struktur
  Gegenwärtig wird die Struktur de Linux/L4 Systems überarbeitet. Die neue Struktur sieht nur noch eine Task für die Kernaktivität im gesamten System vor, welche von allen Nutzeraktivitäten gemeinsam benutzt wird. Um die in dieser Arbeit beschriebene Technik in dieses System zu integrieren, sind einige wenige Änderungen bei der Erzeugung der Tasks (`fork_by_chief`-Systemruf) notwendig. Das neue Linux/L4-System besitzt einen wesentlich einfacheren Synchronisationsmechanismus zwischen der Nutzer- und Kernaktivität. Dieser Mechanismus ermöglicht eine weitaus sauberere Implementierung von vorgetäuschten Systemrufen und Seitenfehlern des Chiefs als im aktuellen System.

- Verbesserung der Leistung
  Im vorangegangenen Kapitel wurden einige Ansatzpunkte für die Verbesserung der Leistung der Implementierung erläutert.

Anhang A

Glossar

Aktivität Abstraktion eines Kontrollflusses in einem Kontext, siehe [Hoh96].

Capability Lists Capability-Listen sind eine Sicherheitskonzept, bei dem einem Subjekt, z.B. einem Prozeß, Rechte zum Zugriff auf Objekte (z.B. Dateien) gegeben werden (die Capability-Liste). Zugriffe werden nur gestattet, wenn eine Subjekt ein entsprechendes Recht für das Objekt besitzt.

Clans&Chiefs Sicherheitskonzept des L4-Mikrokerns, siehe Abschnitt 3.2.

Embedded System Rechnersysteme, die als Steuereinheiten z.B. Haushaltsgeräten eingesetzt werden.

Hash-Funktion Funktion zur Berechnung eines Hash-Index. Wandelt eines Suchschlüssel in eine Integer-Zahl (den Hash-Index).

Hash-Index siehe *Hash-Funktion*

Hash-Tabelle Datenstruktur für die effiziente Verwaltung von Listen (siehe [Sed92]).

Inode Interne Verwaltungsstruktur für Dateien des Linux-Kerns.

IPC *Inter Process Communication* Kommunikationsform zwischen Prozessen.

Java VM *Java Virtual Machine* Ausführungsumgebung für Java Programme.

Kollision Mehrdeutigkeit beim Zugriff auf eine *Hash-Tabelle*. Dabei besitzen verschiedene Suchschlüssel denselben *Hash-Index*.

Lineares Austesten Algorithmus zur Beseitigung einer *Kollision*.

Pager Mechanismus zur Verwaltung von Adreßräumen.

PGP *Pretty Good Privacy* Programm zur Verschlüsselung von Daten.

Signale Einfache Möglichkeit zur Prozeßkommunikation in UNIX-Systemen.

Signatur Digitale Unterschrift.

Task Im L4-Kontext ein Adreßraum, in dem mehrere *Threads* ablaufen können.

Thread Kontrollfluß innerhalb eine *Task*.

WWW *World Wide Web* Informationsdienst im Internet.

Literaturverzeichnis

[Bar93] John Barkley et.al. *Security in Open Systems*. NIST Special Publication 800-7, U.S. Department of Commerce, National Institute of Standards and Technology 1993

[Bau96] Robert Baumgartl, Martin Borriss, Michael Hohmuth und Jean Wolter. *Linux port documentation*. 1996

[Bec95] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus und Dirk Verworner. *Linux-Kernel-Programmierung – Algorithmen und Strukturen der Version 1.2*. Addison Wesley 1995

[Bel76] D. Elliot Bell and Leonard LaPadula. *Secure Operating Systems: Unified exposition and Multics interpretation*. Technical Report 2997, MITRE, March 1976

[Gol96] Ian Goldberg, David Wagner, Randi Thomas and Eric Brewer. *A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)*. Proceedings of the Sixth USENIX UNIX Security Symposium, 1996

[Här93] Hermann Härtig, Oliver C. Kowalski and Winfried E. Kühnhauser. *The BirliX Security Architecture*. Journal of Computer Security,Vol. 2, 1993

[Här97] Hermann Härtig und Lars Reuther. *Encapsulating Mobile Objects*. Eingereicht zur 17^{th} International Conference on Distributed Computing Systems, Baltimore, 1997

[Hoh96] Michael Hohmuth. *Linux-Emulation auf einem Mikrokern*. Diplomarbeit TU Dresden, 1996

[Jav96] *JavaSoft FORUM 1.1 – Java Security* 1996

[Joh97] Michael Johann. *Neu aufgelegt – Beta 2 des Java Development Kitt 1.1*. iX Februar 1997, S. 52-55

[Kah95] Brian L. Kahn. *Safe use X window system protocol across a firewall*. Proceedings of the Fifth USENIX UNIX Security Symposium, 1995

[Kop94] Helmut Kopka. *LaTeX Einführung, Band 1* Addison Wesley 1994

[Kow90] Oliver Kowalski and Hermann Härtig. *Protection in the BirliX Operating System*. Proceedings of the 10th International Conference on Distributed Computing Systems, IEEE Computer Society Press 1990

[Lie96] Jochen Liedtke. *L4 Reference Manual – 486, Pentium, Pentium Pro*. 1996, verfügbar unter ftp://borneo.gmd.de/pub/rs/L4/doc/

[Mid96] Stefan Middendorf, Reiner Singer und Stefan Strobel. *JAVA – Programmierhandbuch und Referenz*. dpunkt Verlag Heidelberg 1996

[Mil96] D.S. Milojicic, M. Condict, F. Reynolds, D. Bolinger and P. Dale. *Mobile Objects and Agents*. Advanced Topics Workshop at 2nd USENIX COOTS, 1996

[Pfi95] Andreas Pfitzmann. *Datensicherheit und Kryptographie*. Vorlesungsskript TU Dresden WS 95/96

[Sed92] Robert Sedgewick. *Algorithmen*. Addison-Wesley 1992

[Tan94] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Carl Hanser Verlag München Wien; Prentice Hall International Inc. London, 1994

[Tan86] A.S. Tanenbaum, S. Mullender and R. van Renesse. *Using sparse capabilities in a Distributed Operating System*. Proceedings of the 6th International Conference on Distributed Computing Systems, IEEE Computer Society Press 1986

[The92] Marvin M. Theimer, David A. Nichols and Douglas B. Terry. *Delegation through Access Control Programs*. Proceedings of the 12^{th} International Conference on Distributed Systems, IEEE Computer Society Press, 1992

[Wac94] John P. Wack, Lisa J. Carnahan. *Keeping Your Site Comfortably Secure: An Introduction to Internet Firewalls*. NIST Special Publication 800-10, U.S. Department of Commerce, National Institute of Standards and Technology 1994

[Wil79] M.V. Wilkes and R.M. Needham. *The Cambridge CAP computer and its Operating System*. Computer Science Library, 1979

[Wul75] W.A. Wulf, R. Levin and C. Pierson. *Overview of the Hydra operating system development*. Proceedings of the 5th ACM Symposium on Operating System Principles, Operating Systems Review 1975