Diplomarbeit:

Linux-Portierung auf den P4 Mikrokernel

vorgelegt von:	Alexander Züpke
Matrikelnummer:	99200868
Abgabetermin:	30.07.2003
Gutachter:	Prof. Dr. Ulrich Kaiser (Erstgutachter)
	Prof. Dr. Gerhard Juen (Zweitgutachter)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Diese Diplomarbeit handelt von der Portierung eines aktuellen Linux 2.4.20 Betriebssystemkerns [9] auf den P4 Mikrokernel. Der P4 ist eine Eigenentwicklung der Firma SYSGO und baut auf den Prinzipien des L4 Mikrokernels [10] auf, der von Jochen Liedtke und anderen entwickelt worden ist. Linux ist als Open Source Software unter der GPL (Gnu Public License) [6] im Internet erhältlich.

1.1 Motivation

Die Firma SYSGO [15] hat 1998 für ihre OSEK-Implementierung den P4 Mikrokernel auf einer MIPS-Plattform entwickelt. Angelehnt ist er an den L4, einen Mikrokernel der zweiten Generation [4]. Der Mikrokernel ist sehr klein und bietet nur die wirklich notwendige Abstraktion der Hardware an: getrennte und voreinander geschützte Adressräume (Tasks), Operationen, um die Adressräume zu manipulieren (Speicherseiten ein- und ausblenden), ein nachrichtenbasiertes Interprozesskommunikationsprotokoll und eine hohe Geschwindigkeit. Gerade durch diese Beschränkung bleibt der eigentliche Betriebssystemkern sehr klein und schlank.

Alle weiteren Betriebssystem-Komponenten müssen dabei als Anwendungen auf den Mikrokernel gesetzt werden. Neben einer OSEK-Implementierung können so beispielsweise noch andere Betriebssystem-Server laufen. Die Idee war nun, dazu den Linux Betriebssystemkern zu nehmen. Dieser ist für viele Architekturen verfügbar und bietet eine große Anzahl an Gerätetreibern und Protokollen.

Ziel der Diplomarbeit war es nun, möglichst schnell ein aktuelles Linux 2.4.20 auf dem P4 für i386 Hardware zu portieren. Dabei sollte die Binärkompatibilität (ABI) der unterliegenden Architektur erhalten bleiben, um den Portierungsaufwand für Applikationen und Linux Gerätetreiber so gering wie möglich zu halten.

Neben dieser Arbeit wurden im Jahre 2002 bei der Firma SYSGO in verschiedenen Diplomarbeiten Portierungen des P4 auf die PowerPC- [14], Intel386 und ARM-Plattform [5] angefertigt, dazu wird aktuell neben dem Linux-Server, der hier behandelt wird, noch

an einem TCP/IP-Stack und einer POSIX-Bibliothek gearbeitet.

1.2 Gliederung der Diplomarbeit

Die Diplomarbeit ist so gegliedert, dass der Leser in den ersten Kapiteln eine kurze Einführung in die Grundlagen des P4 Mikrokernels und des Linux-Kerns bekommt. In Kapitel 4 und 5 werden die Konzepte für ein Linux unter dem P4 Kernel erläutert sowie die eigentliche Portierung beschrieben.
In den letzten Kapiteln wird die Leistungsfähigkeit und Performance des Systems analysiert und das Fazit gezogen.

Fach- und Fremdwörter sind, wenn sie nicht im Kontext erklärt werden, im Anhang noch einmal separat aufgeführt.

1.3 Danksagung

Mein Dank gilt den Mitarbeitern und Diplomanden der Firma SYSGO, insbesondere Robert Kaiser, Christian Löber, Martinus Flöck und Stephan Wagner für ihre Unterstützung und Beantwortung meiner vielen Fragen.

Kapitel 2

Der P4 Mikrokernel

Der P4 Mikrokernel basiert, wie eingangs erwähnt, auf den Konzepten des L4 Mikrokernels [10] von Jochen Liedtke und ist eine Eigenentwicklung der Firma SYSGO. Dieses Kapitel gibt nun einen Einblick in die allgemeine Design-Philosophie dieser Mikrokernelarchitektur und erläutert auch die Unterschiede der Realisierung.

2.1 Was ist ein Mikrokernel?

Monolithische Betriebssystemkerne wie zum Beispiel Linux [9] enthalten in ihrem Kern möglichst viel an Funktionalität. Das ist im wesentlichen die Prozessverwaltung, Protokolle, Dienste, Dateisysteme und Gerätetreiber. Allen diesen Komponenten wird uneingeschränkt vertraut und sie dürfen aus diesem Grund im privilegierten Modus des Prozessors laufen. Obwohl dieses Modell in der Programmierung sehr einfach und effizient ist, führt es doch auch zu einer unübersichtlichen Verschachtelung und somit schlechter Wartbarkeit.

Dagegen stellt ein Mikrokernel selbst kein vollständiges Betriebssystem dar. Er ist die minimale Basis, um darauf Betriebssysteme zu implementieren. Es werden dabei nur die unbedingt notwendigen Teile im privilegierten Modus des Prozessors ausgeführt, die aus Sicherheitsgründen nicht im normalen Modus laufen dürfen. Alle weiteren Funktionalitäten, die ein Betriebssystem benötigt, können auch im unprivilegierten Modus (Userspace) als sogenannte *Server* realisiert werden.

Daraus ergibt sich nun, dass die Server untereinander geschützt werden und nur über genau definierte Schnittstellen miteinander kommunizieren können. Dadurch wird das komplette System sicherer und zuverlässiger, da das Versagen eines Servers nicht zum Absturz des gesamten Systems führt. Andererseits erfordert dies mehr Aufwand für ein sauberes Design, da jede Art von Zugriff genau spezifiziert werden muss.

2.2 Konzepte und Operationen des P4 Mikrokernels

Um diese Minimalanforderungen für die Server zu gewährleisten, bietet der Mikrokernel sehr allgemein gehaltene Konzepte an. Diese kann man sich als eine nächst höhere logische Schicht über der Hardware vorstellen.

Virtuelle Adressräume Virtuelle Adressräume bestehen aus eine Menge von Speicherseiten. Das P4 Konzept sieht dabei sogenannte Flexpages vor, damit sind flexible Speicherseiten der Größe 2^n gemeint. Die Adressräume können sich Flexpages miteinander teilen (diese Operation wird *map* genannt) oder an andere Adressräume weitergeben (*grant*). Auch kann das Zugriffsrecht auf solche Speicherseiten wieder entzogen werden (*flush*).
Von einem Ursprungsadressraum, der σ_0 genannt wird und dem physikalisch vorhandenen Speicher entspricht, werden Flexpages an andere Adressräume weitergegeben. Diese können die erhaltenen Speicherseiten wiederum an andere Adressräume weiterreichen und es wird eine komplexe Struktur von Abhängigkeiten erzeugt, die von Liedtke mit dem Begriff *rekursives Mapping* geprägt worden ist [13].

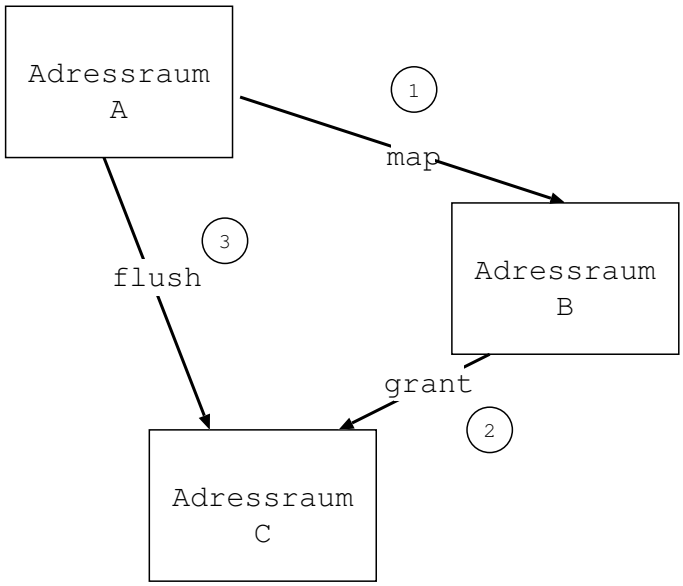

Abbildung 2.1: Rekursive Adressräume

In diesem Beispiel *mappt* Adressraum A eine Speicherseite an B. Diese Seite kann nun beiden Adressräumen als gemeinsamer Speicher dienen. Im weiteren Verlauf *grantet* dann B diese Seite an C, wobei B alle Rechte an dieser Seite verliert und A zum Transfer nicht zustimmen muss. Trotzdem ist es A mit einem *flush* möglich, C den Zugriff auf die Seite zu entziehen.

Tasks & Threads Eine Task besteht aus einem virtuellen Adressraum und ihren Threads. Die Threads sind die Aktivitäten in einer Task, sie stellen einen Ausführungskontext für den Prozessor dar und existieren nebeneinander. Über die *UID* (Unique IDentifer), einer systemweit eindeutigen Nummer, können die Threads identifiziert

werden. Sie dient als Schlüssel bei der Kommunikation der Threads untereinander. Der Adressraum einer Task kann nur von den Threads innerhalb dieser manipuliert werden, wodurch die Task gegen Veränderungen von aussen geschützt ist.

Interprozesskommunikation (IPC) über Nachrichten Die Kommunikation der Threads untereinader basiert auf Nachrichten (engl. *messages*). Die Nachrichten können sowohl Daten als auch Flexpages enthalten. Wenn es möglich ist, werden die Nachrichten im Registerkontext übergeben, um Kopieraktionen im Kern zu vermeiden.

Der Mechanismus zum Versand und Empfang der Nachrichten ist besonders schnell und effizient implementiert, da dies die einzige Technik zum Datenaustausch ist und (fast) alles mit Nachrichten übertragen werden kann.

Sicherheit mit Clans und Chiefs Ein Clan besteht aus einer *Chief*-Task[1] und einigen Untertan-Tasks. Innerhalb eines Clans können alle Mitglieder ohne Einschränkung miteinander kommunizieren.

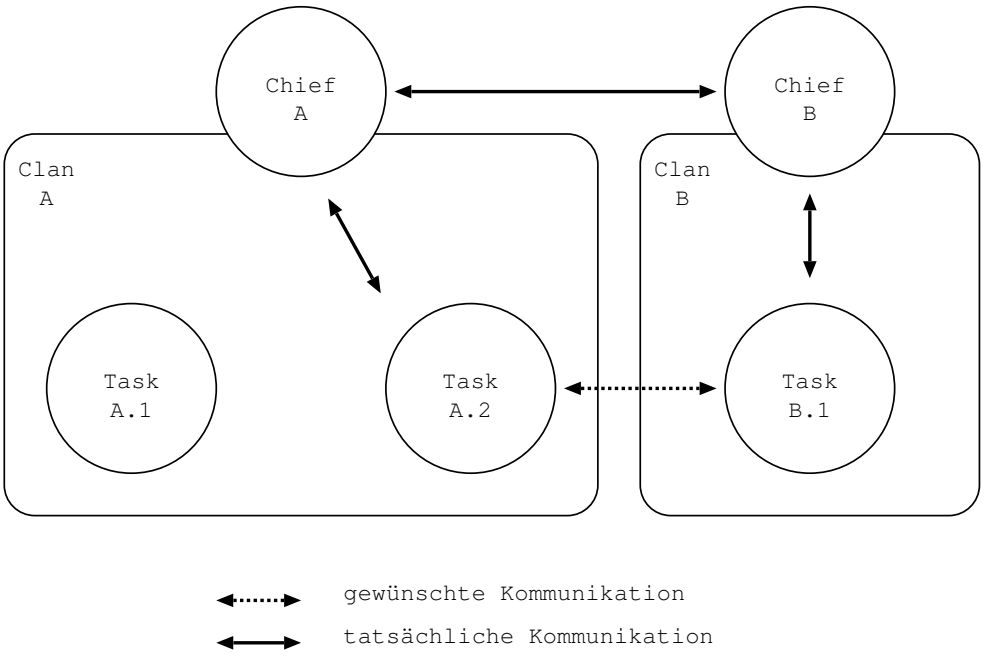

Abbildung 2.2: Clans und Chiefs

Hier versucht Task A.2 eine Nachricht an B.1 zu schicken. Der direkte Weg ist dabei verwehrt, und die Nachrichten werden über die jeweiligen Chiefs A und B geroutet.

Sobald aber Kontakt mit einer ausserhalb des Clans liegenden Task aufgenommen werden soll, läuft diese immer über die jeweiligen Chiefs ab. Der Chief entscheidet dann, ob er die Kommunikation zulässt, verweigert oder den Inhalt der Nachricht verändert. Mit diesem Mechanismus können Protokollumsetzer erzeugt, aber

[1]Der engl. Begriff *Chief* meint einen Stammeshäuptling und der *Clan* ist sein Stamm.

auch bösartige Software gegenüber dem Rest des Systems abgeschottet werden. Die gewünsche Sicherheit obliegt dabei dem Chief, nicht im Mikrokernel.

Externe Pagefault- und Exception-Handler Verursacht ein Thread einen Pagefault (Seitenfehler), indem er auf eine nicht vorhandene Speicherseite zugreift, wandelt der Kernel dies in eine Nachricht für den Pagefault-Handler des Threads um. Darin enthalten ist die Fehleradresse und im Regelfall kann der Pager dann eine Flexpage als Antwort an den Verursacher schicken.
Die Pagefault- und Exceptionhandler werden einem Thread bei seiner Erzeugung zugeordnet.

Der Mikrokernel stellt hier nur die Mechanismen bereit, um Pagefaults als Nachricht zuzustellen, die notwendige Speicherverwaltung muss im zugehörigen Handler ausserhalb des Kerns realisiert werden.

Auch wenn ein Thread eine Exception provoziert, zum Beispiel durch eine ungültige Instruktion, einen unbekannten Systemcall oder unerlaubten Zugriff, bekommt sein Exception-Handler eine Nachricht zugestellt, die den Registerkontext des Verursachers enthält. Der Exception-Handler kann dann diesen nach seinen Vorstellungen manipulieren, den Verursacher neu Starten oder auch Abbrechen.

I/O und Hardware Interrupts Beim einem Mikrokernel sind nur die wirklich notwendigen Treiber zur Interrupt- und Speicherverwaltung im Kern selbst enthalten, alle weiteren Gerätetreiber müssen im Userspace realisiert werden. Dazu werden I/O-Adressbereiche durch Flexpages abgebildet, die ein Gerätetreiber dann von σ_0 anfordern kann. Somit erhält er das Recht zum Zugriff auf einen *bestimmten* Teil der Hardware.
Auch Hardware-Interrupts lassen sich durch die schon bekannten Mechanismen abbilden. Sie werden vom Mikrokernel in Nachrichten umgewandelt und den interessierten Empfängern zugestellt.

2.3 Vergleich L4 und P4

Das Designziel [12] beim Entwurf des L4 war eine möglichst hohe Geschwindigkeit der IPC gewesen. Deshalb gibt es für jede Architektur einen eigenen speziell an die Hardware angepassten und optimierten Kernel. Der Aufwand ist somit relativ groß, den Mikrokernel auf andere Architekturen zu portieren.

Dagegen ist das Designziel des P4 auf größt mögliche Portabilität angelegt. Nur ca. 30% des Mikrokernels ist architekturabhängiger Code wie die Verwaltung der Pagetables oder die Eintrittspunkte für Exceptions. Dadurch ist der P4 zwar langsamer als der L4, der Aufwand für Wartung und Erweiterung halten sich aber in Grenzen.

Dazu kommt der Gedanke, dass der P4 später vielleicht einmal in sicherheitskritischen Umgebungen laufen soll. Dort macht die Implementierung in einer plattform-unabhängigen Hochsprache wie C Sinn, denn eine dann notwendige Zertifizierung ist bei Assemb-

lerprogrammen besonders teuer[2].

Der P4 ist von API-Seite her weitestgehend kompatibel zum L4 gehalten. Software, die für den L4 entwickelt worden ist, kann mit möglichst geringem Aufwand auf den P4 portiert werden. Die Erweiterungen des P4 machen den Mikrokernel flexibler in Richtung Echtzeit.

So ist der P4 etwa voll präemptiv, um Interrupt-Latenzen niedrig zu halten. Daneben kommen noch einige praktische Erweiterungen wie Debugger-Support im Mikrokernel oder Exception-Handling über Taskgrenzen hinweg.

2.4 Aufbau des P4

Der P4 gliedert sich in drei Komponenten, die jeweils modular aufeinander aufbauen und austauschbar sind:

- PSP (Platform Support Package) [3]

  Die unterste Schicht stellt das PSP dar. Es ist speziell auf die boardspezifischen Eigenheiten zugeschnitten und übernimmt den Start des Systems, initialisiert die Hardware und bereitet die Speicherverwaltung vor. Prinzipiell kann es die erste Software sein, die beim Start des Systems ausgeführt wird.
  Hier ist eine einfache Ausgabemöglichkeit zur Fehlersuche, die Interruptverwaltung und die Systemuhr integriert. Dazu werden Schnittstellen angeboten, um den Mikrokernel zu debuggen.

- ASP (Architecture Support Package)

  Das ASP baut auf dem PSP auf und stellt den architekturabhängigen Teil des Mikrokernels dar. Es enthält im wesentlichen die Exception-Trampoline, das Prozessor-Kontext-Management, die Pagetableverwaltung und das C-Binding für die jeweilige Architektur.

- Generischer Teil

  Der generische Teil des P4 ist komplett in C programmiert und architekturunabhängig. Hier sind Komponenten wie die Nachrichtenverteilung und der Scheduler enthalten. Die Funktionen im generischen Teil werden von den Handlern des PSP und ASP aufgerufen und bedienen sich wiederum der Speicher- und Interruptverwaltung, die von diesen beiden Stufen bereitgestellt wird.

[2] Denn es existieren für Hochsprachen diverse Tools, die eine Analyse automatisieren. Weiterhin lassen sich Testfälle zur Codeabdeckung einfacher generieren.

[3] Der Aufbau des PSP und des ASP wird in den Diplomarbeiten von Philipp Roebrock (PowerPC) [14] und Martinus Flöck (ARM) [5] sehr detailliert beschrieben.

2.5 Schnittstellen des P4

Im Vergleich zu anderen Betriebssystemkernen wie Linux mit über 240 Systemcalls scheinen *sieben* des P4 sehr wenige zu sein. Es wird hier aber auch nur ein Bruchteil der Funktionalität geboten.

Hier eine kurze Übersicht über die Schnittstellen des Mikrokernels:

2.5.1 Interprozesskommunikation

Der IPC-Systemcall ist sicherlich der komplizierteste. Mit ihm lassen sich Nachrichten an andere Threads verschicken und natürlich auch empfangen.

Versand und Empfang von Nachrichten wird über nur einen Systemcall abgewickelt. Zuerst kommt die optionale Sendephase, bei der versucht wird, die Nachricht dem Empfänger zuzustellen. Nach einem spezifizierbaren Timeout kann der Sendeversuch abgebrochen werden, um Deadlocks zu vermeiden. Nach dem Versand geht der Thread in eine optionale Empfangsphase, die ihr eigenes Timeout hat.

So lassen sich folgende Varianten realisieren:

- Senden an einen Thread (*send*),
- Empfang von einem spezifizierten Thread (*closed receive*),
- Empfang von einem beliebigen Thread (*open receive*),
- Senden an und Empfang von einem spezifizierten Thread (*call*),
- Senden an einen Thread, Empfang von einem beliebigen Thread (*reply and wait*) und
- Warten ohne Versand und Empfang auf den Timeout (*sleep*).

Um den Overhead eines Kerneintrittes zu vermindern, sollte immer versucht werden, die beiden Sende- und Empfangskombinationen zu nutzen. Ein Client wird über *call* mit einem Server kommunizieren, der in seiner Verarbeitunsschleife mit *reply and wait* auf Anfragen von unterschiedlichen Threads wartet.

Für den Versand und den Empfang wird ein Messagedescriptor benötigt. Dieser zeigt jeweils auf eine Struktur, die die Sende- bzw. Empfangspuffer beschreiben und er gibt an, ob Mappings in einer Nachricht enthalten sind.

Um eine gute Performance zu erreichen, sollten die Messagewörter möglichst in den Prozessorregistern übergeben werden, denn dies erspart unnötige Kopieraktionen im Kernel. Den kleinsten gemeinsamen Nenner definiert hierbei die i386-Architektur, die nur einen sehr begrenzten Registervorrat besitzt. So stehen dort nur noch 2 freie Register für Messagewörter zur Verfügung.

Diese kurze Art der Nachrichten wird *Direct Message* genannt und ist als spezielles C-Binding in allen Architekturen definiert, um Software möglichst portabel zu halten.

Eine weitaus detailliertere Beschreibung zu den unterschiedlichen Varianten dieses Systemcalls bietet das L4 User Manual [1].

2.5.2 id_nearest()

Innerhalb eines Clans kann eine Nachricht direkt zugestellt werden, aber sobald die Nachricht den Clan verlässt, muss der Chief des Clans involviert werden. Dazu liefert dieser Systemcall den nächsten Kommunikationspartner in der Hierarchie. Eine Abwandlung von `id_nearest()` ist `id_myself()`, hier wird die eindeutige UID des Aufrufers zurückgegeben.

2.5.3 task_new()

Der Systemcall `task_new()` erzeugt eine neue Task mit dem Aufrufer als Chief. Dabei wird ein initialer Thread 0 in der neuen Task angelegt und aktiviert. Als Aufrufparameter werden ein Exception-Handler, ein Pager, die maximale Priorität (Maximum Controlled Priority, kurz MCP), die Startadresse und der Stack benötigt. Da eine neue Task zuerst immer einen leeren Adressraum besitzt, provoziert der neue Thread somit Pagefault und der Pager muss die fehlenden Seiten in diesen Adressraum mappen oder granten.

Eine Abwandlung davon ist `task_donate()`. Hiermit werden die Threads und der Adressraum der Task zerstört und es kann ein neuer Chief für die Task festgelegt werden. Der angegebene neue Chief erhält eine leere Task-Hülle, eine sogenannte *passive Task*, und damit das Recht, diese Task mit einem Adressraum und einem Thread 0 neu zu aktivieren.

2.5.4 lthread_exregs()

Mit `lthread_exregs()` können die Threads innerhalb einer Task manipuliert werden. `lthread_exregs()` erlaubt das Verändern des Stack- und Instructionpointers. Solange dabei gültige Werte angegeben werden, wird ein neuer Thread gestartet. Auch Pager und Exception-Handler können über diesen Systemcall ausgetauscht werden.

Wie auch bei einer Exception-Nachricht gibt dieser Systemcall dem Anwender die Möglichkeit, den Registerkontext eines anderen Threads zu verändern. Es lassen sich damit User-Threads auf einen P4 Thread multiplexen, wobei der Scheduler dann mit `lthread_exregs()` die Registerkontexte selbst austauscht.

2.5.5 thread_schedule()

Über den Systemcall `thread_schedule()` kann die Priorität und die Zeitscheibe eines Threads verändert werden. Ein aufrufender Thread kann keine Threads mit einer höheren

Priorität als seiner eigenen manipulieren oder sich über die MCP der Task hinwegsetzen.

Insgesamt gibt es 256 Prioritätsstufen. Der P4 Scheduler aktiviert immer nur den rechenwilligen Thread mit der höchsten Priorität, innerhalb einer Prioritätsstufe findet Round-Robin-Scheduling mit einstellbaren Zeitscheiben ab. Dieses führt auch dazu, dass niederpriore Prozesse *verhungern* (Starvation), solange es höherpriore gibt. Dieses Verhalten ist aber erwünscht, denn es ist nicht Aufgabe des Mikrokernels, für *faires*[4] Scheduling zu sorgen. Dies kann im Userspace erledigt werden.

Mit diesem Mechanismus wird auch der gegenseitige Ausschluss aus kritischen Bereichen (Critical Sections) realisiert. Dazu setzt ein Thread kurzzeitig seine eigene Priorität über die der anderen.

2.5.6 thread_switch()

Mit `thread_switch()` kann ein gerade laufender Thread Rechenzeit an einen anderen abgeben. Dieser bekommt dann die restliche Zeitscheibe des Aufrufers zur Verfügung gestellt. So lässt sich relativ einfach ein *User-Level-Scheduling* ausserhalb des Mikrokernels realisieren, wobei ein höherpriorer Scheduler seine Rechenzeit an einen niederprioren Thread weitergibt. Wenn hierbei kein Thread angegeben wird, geht die restliche Zeitscheibe an den nächsten rechenbereiten Thread.

2.5.7 fpage_unmap()

Neben den Operationen *map* und *grant* auf Flexpages ist es mit `fpage_unmap()` möglich, allen Tasks, die eine bestimmte Seite bekommen haben, diese wieder zu entziehen. Dabei kann das Zugriffsrecht eines Mappings auf Nur-Lesen (*Read-Only*) gesetzt oder komplett entzogen werden. Dies setzt das stille Einvernehmen voraus, dass der Empfänger eines Mappings generell zustimmt, dass ihm die Seiten auch wieder entzogen werden können. Als Variante dieses Aufrufs kann zusätzlich die Seite auch aus dem Adressraum des Aufrufers ausgeblendet werden.

2.5.8 Protokolle

Neben den Systemcalls existeren noch diverse Protokolle zur Kommunikation der einzelnen Komponenten untereinander. Die drei Protokolle, die auch dem Mikrokernel bekannt sein müssen, sind festgelegt:

- das *Pagefault-Protokoll*, denn der Mikrokernel erzeugt hier eine Nachricht an den Pagefault-Handler, die die Fehleradresse und den aktuellen Program-Counter des Auslösers enthält,
- das *Exception-Protokoll*, das den Aufbau des Registerkontextes definiert,

[4] Wie die Scheduling-Strategie unter Linux: *jeder* Prozess bekommt Rechenzeit.

- und das σ_0-*Protokoll* zum Anfordern von Speicherseiten, die der Mikrokernel zur internen Verwaltung benötigt.

Andere Tasks können bei der σ_0-Task, die den Verwalter des Uradressraums darstellt, physikalischen Speicherseiten anfordern. Daneben ist σ_0 auch der erste Chief im System, denn alle weiteren Tasks müssen von ihm erzeugt werden.

Dies soll wie gesagt nur einen kleinen Überblick über die Möglichkeiten dieser Mikrokernel-Architektur geben, detailliertere Beschreibungen finden sich in den Manuals [1] und [13] wieder.

Kapitel 3

Der Linux Betriebssystemkern

3.1 Übersicht

Der Begriff *Linux* bezeichnet den Betriebssystemkern eines *GNU/Linux*-System. Fälschlicherweise wird heutzutage das gesamte System als Linux bezeichnet, aber Anwendungen und Betriebssystemkern sind voneinander getrennt und auch austauschbar. Das GNU-Projekt [6] der FSF[1] hat es sich zur Aufgabe gemacht, einen vollständigen Unix-Nachbau als freie Software zur Verfügung zu stellen. Der *Hurd* ist der eigentliche Kernel dieses Projektes.

Der Linux-Kern ist ein monolithischer Kern, d.h. alle Komponenten wie Gerätetreiber, Dateisysteme, Protokolle und die Resourcenverwaltung von Speicher und Rechenzeit laufen im Gegensatz zum Mikrokernel im privilegierten Modus des Prozessors ab. Zusätzlich besteht eine starke Abhängigkeit der einzelnen Komponenten untereinander. Es gibt zwar saubere Schnittstellen zwischen den Systemen, aber ein Subsystem kann durch einen Absturz oder ungültigen Speicherzugriff den gesamten Kern zum Stillstand bringen. Gerade dies soll durch den Einsatz des Mikrokernels verhindert werden.

Daneben gibt es noch eine andere, orthogonale Ebene: die Abhängigkeit von der Prozessorarchitektur. Anfangs unterstütze Linux nur die i386-Architektur und brachte einen geringen Anteil an Gerätetreibern mit, doch über die Jahre wurde Linux auf viele weitere Architekturen und Prozessoren portiert, das Spektrum reicht vom kleinen Embedded-System mit geringem Ressourcenverbrauch bis zum Multiprozessor-Hochleistungsserver. Daher gibt es im Linux-Kern eine gute Trennung zwischen der architekturabhängigen Schicht und der generischen Schicht.

Das Ziel der Arbeit ist nun, eine neue Architektur zu etablieren, die auf den Mechanismen des Mikrokernels anstatt direkt auf der Hardware aufbaut. Diese Architektur-Schicht muss folgendes leisten:

- Das System initialisieren, soweit der Mikrokernel dies noch nicht getan hat,

[1] Free Software Foundation, `http://www.fsfeurope.org/index.de.html`

- Anwendungen eine Schnittstelle zum Eintritt in den Linux Kern bereitstellen, um Betriebssystemfunktionen aufrufen können,
- Anwendungen Signale zustellen,
- Prozesswechsel ermöglichen,
- elementare Grundfunktionen der Speicherverwaltung ausführen (Seiten mappen oder unmappen),
- den Gerätetreibern eine Verwaltung für Hardware-Interrupts bereitstellen,
- sowie Speicherzugriffe in den Adressbereich der Anwendungen ermöglichen.

Weiterhin sollte diese neue Architektur das selbe ABI (Application Binary Interface) benutzen wie die unterliegende Architektur auf der der Mikrokernel läuft, damit vorhandene Software möglichst, ohne sie neu zu kompilieren, weiter genutzt werden kann.

Der Mikrokernel stellt für alle Architekturen eine weitgehend gleiche Schnittstelle bereit, und eine Linuxportierung auf eben diese Schnittstelle sollte auf allen vom Mikrokernel unterstützten Architekturen lauffähig sein. Gesucht ist nun eine möglichst geeignete Linux-Ausgangsarchitektur, denn diese Linux Portierung soll später einmal auf anderen Architekturen als i386 laufen.

3.2 Usermode-Linux

Usermode-Linux [3, 16], oder kurz UML, ist ein Linux-Kern, der auf einem anderen Linux als Anwendung läuft. Ein UML nutzt dabei die Schnittstellen des Hostsystems wie die Speicherverwaltung, um seine Applikationen laufen zu lassen. Es bietet damit genau die Abstraktionen, die für eine Portierung auf einen Mikrokernel notwendig sind und scheint damit ein idealer Ansatzpunkt zu sein, um möglichst schnell ein P4/Linux entwickeln zu können. UML ist für die i386- und die PowerPC-Architektur bereits implementiert worden und gestattet keinen direkten Zugriff auf die Hardware. Alle Gerätetreiber setzen dabei auf den Schnittstellen des Hostsystems auf und existeren somit nur virtuell.

Daneben bietet UML auch eine einfache Möglichkeit, die Subsysteme des Linux-Kernels bequem zu debuggen oder auch ein Linux auf anderen Betriebssystemen zum laufen zu bringen. Da das unterliegende System nicht korrumpiert werden kann, können mit UML auch *Sandboxen*, in denen Software ohne Bedenken getestet werden kann, und *Honeypots*, die im wörtlichen Sinne als *Fliegenfallen* für Hacker im Internet plaziert werden, realisiert werden.

UML stellt selber eine eigenständige Architektur im Linux-Kernel dar, benötigt aber ein vorhandenes Betriebssystem. Schauen wir uns also mal die Schnittstellen des UML zum Host-Betriebssystem genauer an:

- Gerätetreiber

  Es gibt unter UML keine Gerätetreiber, die direkt auf die Hardware zugreifen können. Die Geräte existieren also nur virtuell und stellen sich nach aussen hin einfach als Zugriffe auf Dateien des Hostsystems dar.

- Signale als Hardware Interrupts

  Auch die Hardware Interrupts werden emuliert. Über das Signal `SIGIO` wird dem UML Kernel mitgeteilt, dass sich bei einer beobachteten Datei etwas getan hat. Daraus wird dann ein Interrupt für die Gerätetreiber generiert. Die Signale `SIGALRM` und `SIGVTALRM` erzeugen das Timing für UML.

  Der Ausschluss bei kritischen Bereichen wird über die Sperrmechanismen für Signale mit `sigaction()` sichergestellt.

- Die Speicherverwaltung

  Der gemeinsame Speicher des Kernels und der Anwendungen ist eine in den virtuellen Speicher eingeblendete Datei. Dieser "physikalische Speicher" kann mit `mmap()`, `mprotect()` und `munmap()` ein- oder ausgeblendet und natürlich auch mit anderen Prozessen geteilt werden.

- Systemcalls

  Nutzerprozesse unter UML werden mit `ptrace()` kontrolliert. `ptrace()` ist eine Debugging-Schnittstelle des Linux-Kernel. Eine "getracte" Anwendung wird angehalten, wenn sie einen Systemcall macht und der Observierende kann dann den Registerkontext des Aufrufers vor und nach dem Systemcall inspizieren.

  Da der Systemcall immer zum Hostsystem durchgereicht wird, verändert UML ihn vorher zu `getpid()`, einem Null-Systemcall,[2] um und ruft danach seinen eigenen Systemcall-Handler auf.

- Kontrolle der Nutzerprozesse über Signale

  Mit dem Signal `SIGSTOP` kann der UML-Kern seine Userspace-Prozesse unterbrechen und über `SIGSEGV` wird ein Pagefault-Handler simuliert. `SIGILL` und `SIGBUS` informieren den Kern darüber, dass in einem Userspace-Prozess ein Fehler aufgetreten ist.

In der ursprünglichen Fassung von UML gab es nur den *Tracing Threads* genannten Betriebsmodus. Dabei wird für jeden Userspace-Prozess des UML ein eigener Prozess auf dem Hostsystem benötigt, indem auch noch Teile des UML-Kerns eingeblendet sind, damit die Prozesse kommunizieren können. Daneben wurde im Herbst 2002 die Betriebsart *Separate Kernel Address Space*, kurz SKAS, eingeführt, um den Resourcenverbrauch auf dem Host-System zu verringern.

[2] Dieser einfache Systemcall liefert dem Aufrufer die Prozess-ID und wird gerne zu Messungen herangezogen, um den Overhead für den Kernein- und austritt zu bestimmen.

Ein Patch auf dem Host-Linux ermöglicht es, auf einen Host-Prozess mehrere umschaltbare Adressräume abzubilden, da UML nur diese Eigenschaft vom Hostsystem benötigt. So reduziert sich der Aufwand auf dem Hostsystem auf genau zwei Prozesse: einen für den UML-Kernel und einen für den Userspace. Dabei stellt der Host über die Datei `/proc/mm` ein Interface zur Kontrolle der Adressräume bereit: Wird die Datei `/proc/mm` geöffnet, erzeugt das Hostsystem einen neuen Adressraum im Userspaceprozess. Im Gegensatz dazu führt das Schliessen der Datei zur Zerstörung des Adressraumes. Ein Schreibzugriff in die Datei wird als Kommando interpretiert, und der Host blendet dann Speicherseiten in den Userspaceprozess ein oder aus. Zusätzlich gibt es noch eine Schnittstelle zur Umschaltung der Speicherkontexte über den `ptrace()`-Systemcall.

3.3 UML als Ausgangsbasis für P4/Linux

Die L4/Linux Portierung[8, 11] der TU Dresden aus dem Jahre 1996 wurde ausgehend von der i386 Architektur entwickelt[3]. Der aktuelle Stand dieser Portierung ist bei der Kernel Version 2.2.25[4] angelangt, dieser Port soll aber ein aktuelles Linux 2.4.20 als Ausgangsbasis bekommen.

Da die Schnittstellen zur Kontrolle der Userspaceprozesse im UML sehr gut ausgearbeitet sind, schien dies eine bessere Ausgangsarchitektur für die P4/Linux-Portierung zu sein.

Vorteilhaft erscheint dabei:

- Diese Vorgehensweise beim UML ist mit dem Task-Konzept und seinen virtuellen Adressräumen unter P4 vergleichbar. Dabei laufen Kernel und Userspaceprozesse in jeweils eigenen virtuellen Adressräumen und das Ein- und Ausblenden von Speicherseiten kann auf die Flexpage-Operationen *map* und *unmap* abgebildet werden.
- Die `ptrace()`-Schnittstelle lässt sich dabei durch das Pagefault- und Exception-Management des P4 abbilden.
- Unterstützung für mehrere Plattformen (es wird später i386, PowerPC, ARM und MIPS benötigt) ist gegeben.

Es gibt aber auch Nachteile:

- Bei UML sind keine Gerätetreiber für reale Hardware vorgesehen. Die Schnittstellen dazu müssen aus vorhandenen Architekturen rückportiert werden.
- Möglicherweise ist diese Lösung langsamer als eine Anpassung einer vorhandenen Hardwarearchitektur an den P4.

Die Entscheidung fiel aufgrund des zu erwartenden geringeren Aufwands auf UML als Ausgangsbasis, hinzu wird das UML-Projekt von der Linux-Gemeinde weiter gepflegt,

[3] Damals existerte das UML-Projekt noch nicht
[4] Stand 17. März 2003

somit bestehen gute Chancen, dass die hier geleistete Arbeit auf zukünftige Versionen des Linux Kerns leicht übertragbar sein wird.

Kapitel 4

P4/Linux

In diesem Kapitel werden die Konzepte des Linux Kernels und seiner Interaktion mit den Userspace-Anwendungen erläutert. Die Reihenfolge orientiert sich dabei am zeitlichen Fortschreiten der Entwicklung und den Problemstellungen, die dabei auftraten.

4.1 Vorbereitungen am UML-Kern

Die erste Aufgabe nach einer gründlichen Analyse des UML-Kerns bestand darin, die Abhängigkeiten vom Hostsystem herauszuarbeiten. UML benötigt ca. 120 Bibliotheksfunktionen und Linux-Systemcalls, die auf dem P4 nicht bereitstehen.

Dazu wurden im ersten Schritt alle Treiber für die virtuellen Geräte auf dem Hostsystem entfernt, da sie alle über Filedescriptoren auf Dateien arbeiten, die auf dem P4 nicht vorhanden sind.

Danach wurde die starke Bindung des Codes an die Bibliotheken des Hostsystems gelöst. Zu beachten war dabei, dass im architekturabhängigen Teil einige Teile gegen die libc-Bibliothek gebunden wurden, andere Teile wiederum gegen den generischen Teil des Linux-Kernels. Die strikte Trennung wurde dabei aufgehoben, und die sehr vielen kleinen Funktionen aus beiden Kontexten zusammengefasst. Zusätzlich wurde noch die Unterstützung des Tracing Threads Modus komplett entfernt.

Am Ende wurden alle Abhängigkeiten nach aussen durch eine kleine Bibliothek abgebildet, die sich auf nur noch 30 Linux-Systemcalls beschränkte.

Neben dieser Bibliothek blieben nur noch die `ptrace()`-Aufrufe, ein reduziertes Signalhandling zur Kontrolle der Nutzerprozesse sowie ein sehr einfacher Konsolen-Gerätetreiber zur Ausgabe übrig. Dieses minimale UML-System funktionierte noch auf dem Hostsystem, benötigte aber eine einkompilierte, initiale Ramdisk. Die Anwendung auf der Ramdisk machte nur einige Ausgaben und beendete danach den UML-Kern.

Nach dieser Phase des "Entkernens" und dem Ausarbeiten der Schnittstellen wurde versucht, den Linux-Kern auf dem P4 zu starten. Dazu wurde ein Ladeprogramm entwickelt, das sich mit dem Speicherlayout und den Bedürfnissen des Linux Kerns auskennt.

4.2 Speicherlayout

Der P4 Kern beansprucht bei einem System mit 4 GB virtuellem Adressraum das obere 1 GB für sich, die unteren 3 GB stehen den Tasks zur Verfügung. Der komplette Linux Kern wird in einer einzelnen Task realisiert, die den Bereich von `0xa0000000` bis `0xbfffffff` für sich beansprucht.

Das Ladeprogramm stellt dem Linux Kern dazu ab der Basisadresse `0xa0000000` einen kontinuierlichen Adressraum von vorher festgelegter Größe bereit, in dem auch das Kernelimage und eine initiale Ramdisk eingeblendet werden. Dieser Speicher stellt dann für Linux den Ausgangspool seiner eigenen Speicherverwaltung dar.

Daneben existiert noch ein Bereich für virtuellen Speicher, das *Kernel Virtual Memory*, den der Kernel selber belegen kann.

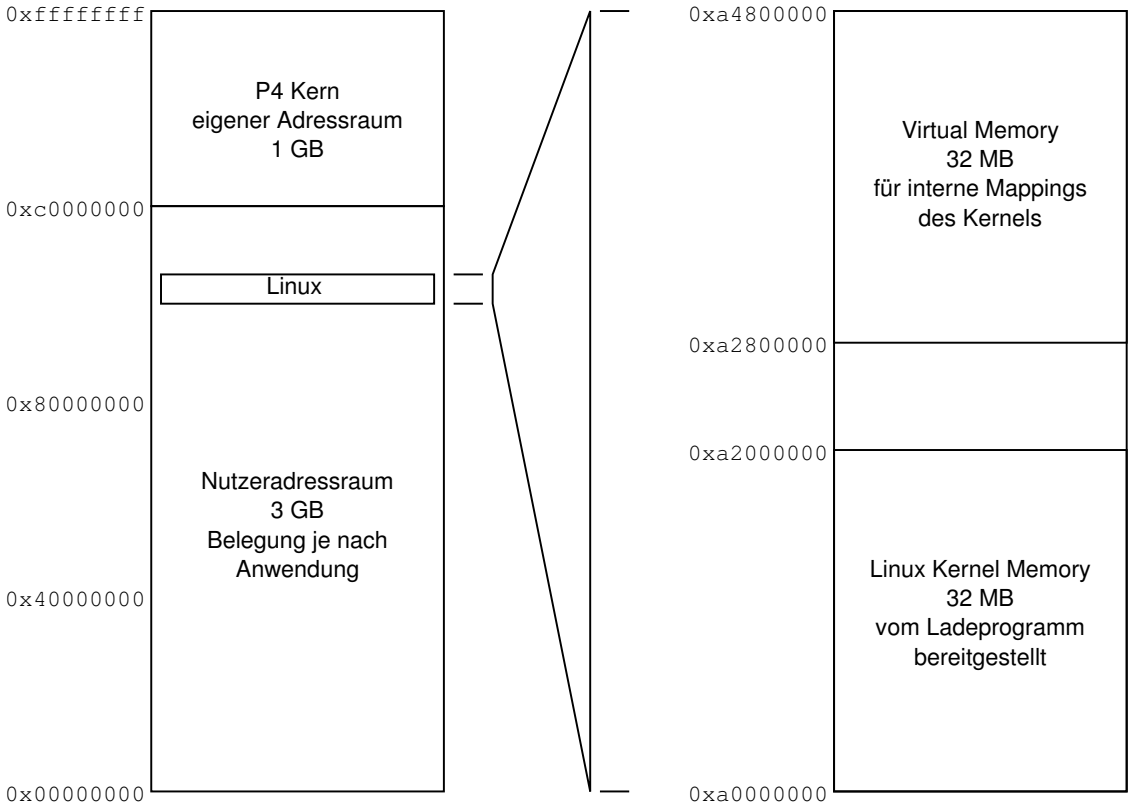

Abbildung 4.1: Belegung des Adressraums

Zusätzlich bekommt der Kern vom Ladeprogramm die Position der Ramdisk sowie die Zuordnung zwischen dem virtuellen und dem physikalisch vorhandenen Speicher mitgeteilt, um so zum Beispiel DMA zu ermöglichen.

4.3 Ladeprogramm

Das Ladeprogramm ist eine kleine Anwendung auf dem P4 und muss Linux als dessen Chief folgende Dienste zur Verfügung stellen:

- Das gewünschte Mapping erzeugen

  Zuerst besorgt sich das Ladeprogramm die gewünschte Anzahl Speicherseiten von der σ_0 Task, denn nur so bleibt die Information über die physikalische Position erhalten. Linux unterstützt explizit fragmentierten physikalischen Speicher, daher müssen die Bereiche nicht unbedingt zusammenhängen. Die Anzahl dieser *Speicherbänke* ist aber trotzdem begrenzt.

  Der Linux Kernel wiederum greift in einer kleinen Programmschleife einfach lesend auf jede Speicherseite zu, die er für sich beansprucht, und provoziert so einen Pagefault. Der Pager, den das Ladeprogramm dazu bereitstellt, füllt die Seiten mit den gewünschten Daten wie dem Kernelimage oder der Ramdisk, bevor sie danach dem Linux Kernel zugestellt werden.

- Ein- und Ausgabe

  Die Ein- und Ausgabe muss an einen Konsolenserver ausserhalb des Clans weitergeleitet werden, da Kommunikation ausserhalb der Clangrenzen immer den Chief involviert. Das Ladeprogramm wartet dabei in einer Schleife auf Anfragen des Linux-Kernels und stellt diese dann dem passenden Server zu. Neben der Verwendung als Protokollumsetzer werden hierüber noch weitere Dienste angeboten.

- Der Linux Kern muss von seinem Chief das Recht bekommen, eigene Tasks anzulegen.

  Direkt nach dem Start des Linux Kernels gibt das Ladeprogramm das Recht, Tasks zu erzeugen, an den Kern mit `task_donate()` ab. Wird die Linux Task zerstört, fällt das Recht natürlich wieder an das Ladeprogramm zurück.

- Neustart einleiten

  Auf Verlangen des Linux Kerns wird dieser neu gestartet. Dazu wird einfach über der alten Task eine neue angelegt. Dabei werden alle Kindtasks zerstört und der Anfangszustand eines leeren Adressraumes ist damit wiederhergestellt.

- Bootparameter

  Das Ladeprogramm muss einen Parameterblock vorbereiten. Hier drin wird die Zuordnung von virtuellen in physikalischen Speicher, die Größe des Kernel Speichers sowie die Kommandozeile des Kernels für weitere Bootparameter eingetragen. Linux erwartet diese Angaben in der ersten Seite seines Speicherbereiches.

Während der Initialisierungsphase wertet Linux die Bootparameter aus und fordert seinen Speicher an. Danach hat das Ladeprogramm, bis auf Neustarts und Weiterleitung der Ein- und Ausgabe natürlich, seine Aufgabe erfüllt, und Linux kann seine eigenen Threads für

Pager, Exception-Handler und Interrupts starten. Diese Phase ist im Prinzip vergleichbar mit dem Startup-Code in Assembler beim i386, wo die MMU und Trampoline-Code für den Kerneintritt bei Interrupts und Exceptions initialisiert werden.

Die Entwicklung des Ladeprogramms und die Änderungen am Linux Kernel verliefen parallel. Da der Linux Kern nach seiner ersten Phase autark arbeiten kann, wurde zunächst die Ausgabe zum Konsolentreiber in die oben genannte Bibliothek integriert. Dieser startete auch sofort bei den ersten Versuchen und generierte Debug-Ausgaben, blieb aber bei der Kalibrierung seines Zeitgebers stehen, da die Unterstützung eines Ticker-Interrupts in dieser Phase der Entwicklung noch fehlte.

4.4 Aktivitäten im Kern

Wie auch beim P4 gibt es im Linux Umfeld Threads. Um diese vom P4-Äquivalent zu unterscheiden, verwende ich hier den Begriff *Aktivitäten* wie bei [8].

Jede Aktivität hat ihren eigenen Prozessorkontext und einen eigenen Stack. Weiterhin gibt es noch die Unterscheidung zwischen Kern- und Nutzeraktivitäten. Die Kernaktivitäten laufen im Adressraum des Kernels ab, wogegen die Nutzeraktivitäten ihren eigenen Adressraum ausserhalb des Kerns haben. Jede Nutzeraktivität hat dabei auch eine zugehörige Kernaktivität, die den Kontext der Nutzeraktivität bei einem Kerneintritt sichert und bei einem -austritt wiederherstellt, die interne Prozessumschaltung gewährleistet, sowie den Stack für Aufrufe innerhalb des Kernels bereitstellt.

Nach der oben beschriebenen Urinitialisierung wird die Funktion `start_kernel()` auf ihrem Stack `init_task` angesprungen. Sie ist die erste Aktivität des Kerns und übernimmt alle weiteren Schritte bis zum lauffähigen System.

Für die Erzeugung und Umschaltung der Kern-Aktivitäten muss die architekturabhängige Schicht einige Funktionen bereitstellen. Ganz unten werden dazu einige in Assembler geschriebene Funktionen benutzt, die an die Bibliotheksfunktionen `setjmp()` und `longjmp()` angelehnt sind und die Registerkontexte umschalten. Sie werden durch Trampoline-Code gekapselt, der die korrekte Beendigung einer Aktivität und die Umschaltung in den Userspace sicherstellt.

Die internen Zustände einer jeden Aktivität werden in einem *Thread Control Block* (TCB, `struct task`) festgehalten, die am unteren Ende des Stacks der Aktivität liegt. Wenn eine neue Aktivität gestartet werden soll, muss zuerst ein Bereich für den Stack und den TCB von vier Speicherseiten bereitgestellt werden, danach erzeugt der Trampoline-Code einen gültigen Kontext auf diesem Stack und springt diesen an.

Die Zusammenlegung von Stack und TCB ist durch die Tatsache bedingt, genau wissen zu müssen, welche Aktivität gerade aktiv ist. Der Stackpointer wird durch einen atomaren Befehl umgeschaltet, und durch geschicktes Ausblenden der unteren Bits erhält man die Adresse des aktuellen TCB. Das Makro `current` macht genau dies bei einigen Architekturen[1].

[1] Architekturen, die nicht an Registerverknappung wie i386 leiden, spendieren ein eigenes Register.

Alle Kernaktivitäten laufen dabei in *einem* P4 Thread ab, denn innerhalb des Linux Kernels gibt es nur *kooperatives Multitasking*, d.h. jede Kernaktivität muss sich durch den Aufruf der Funktion `schedule()` selbst suspendieren und Rechenzeit an andere Aktivitäten abgeben.

Die andere denkbare Modellierung mit je einem P4 Thread pro Aktivität zieht eine nichttriviale Synchronisierung der Kernaktivitäten untereinander nach sich und stößt schnell an die Begrenzung auf nur 128 Threads pro Task.

4.5 Interrupts

Hardware-Interrupts stellen für den Linux-Kernel asynchrone Nachrichten dar, die zu jeder Zeit eintreffen können. Sie stehen in keinem Zusammenhang mit der gerade ablaufenden Aktivität, müssen aber eventuell auf den selben Datenstrukturen arbeiten, die eine Aktivität gerade für sich beansprucht. Eine Aktivität kann sich nun zum Schutze eines *kritischen Bereiches* gegen Interrupts schützen, indem es diese für eine kurze Zeit sperrt oder verhindert.

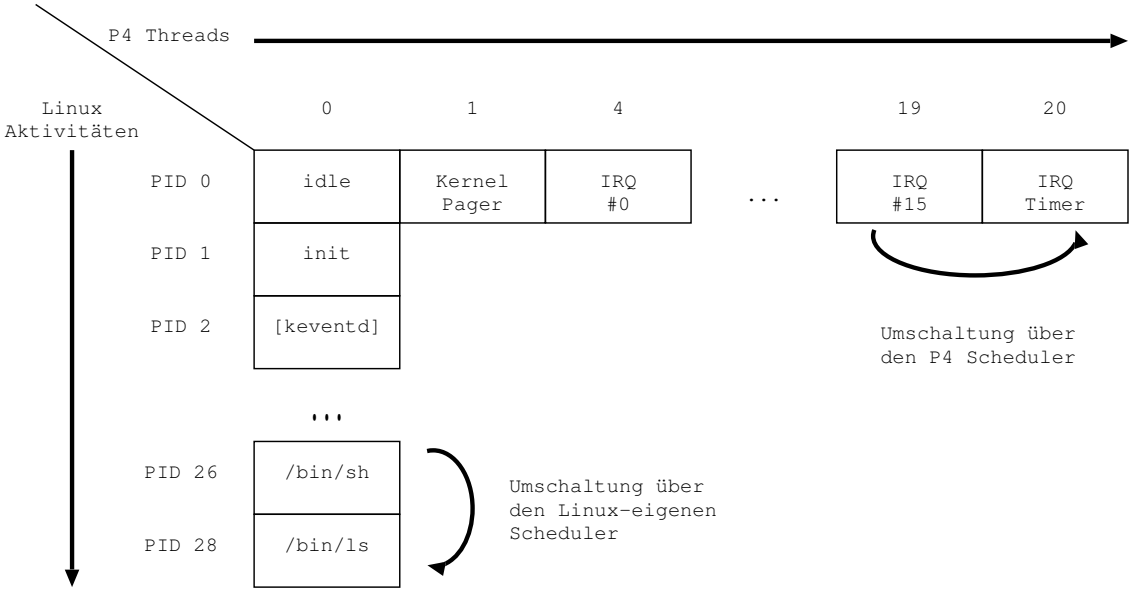

Abbildung 4.2: P4 Threads und Linux Aktivitäten

Die Priorisierung des Systems mit Interrupts sieht folgendermassen aus:

- Hardware Interrupts

  Die Hardware Interrupts laufen sehr schnell ab und sind nicht durch andere unterbrechbar. Sie bestätigen lediglich den Empfang bei der auslösenden Hardware und signalisieren der nächsten Stufe, dass Arbeit ansteht. Diese Zweiteilung führt zu einer guten Reaktionszeit des Gesamtsystems.

- Softirqs, Tasklets und Bottom-Halves

  Diese drei Typen werden durch einen Hardware Interrupt geweckt. Während ihrer Abarbeitung können aber weitere Hardware Interrupts eintreffen. Die Namen bezeichnen jeweils eine andere Strategie der Serialisierung:
  Bei den *Bottom-Halves* wird sichergestellt, dass nur immer genau *einer* aktiv ist. Dies führte aber auf Mehrprozessormaschinen zu starken Performanceengpässen und wurde im Verlauf der Linux-Entwicklung durch die *Softirqs* abgelöst[2]. Diese sind für jeden Prozessor einmal vorhanden und können parallel ablaufen, müssen sich aber gegebenenfalls synchronisieren, wenn sie auf gemeinsame Datenstrukturen zugreifen. Von den *Tasklets* wiederum kann es beliebig viele geben, die parallel ausgeführt werden. Linux verhindert hier nur die parallele Ausführung des selben Tasklets.

- Kernaktivitäten

  Die oben besprochenen Kernaktivitäten können jederzeit durch Hardware Interrupts und nachgezogene weitere Aktivitäten unterbrochen werden, solange sie sich nicht dagegen schützen.

Die Hardware Interrupts werden für gewöhnlich auf dem Stack der gerade laufenden Aktivität bearbeitet, deren Zustand natürlich gesichert wird. Dies ist aber bei der Implementierung auf P4 nicht möglich, da Interrupts dem Empfänger durch eine IPC mitgeteilt werden und dieser explizit im Empfangszustand sein muss. Die Tatsache, dass Interrupts sonst nichts mit der aktuellen Aktivität zu tun haben, ermöglichte es, die Interruptbehandlung in einzelne Threads auszulagern.

Je ein Thread pro Interrupt wartet auf eine Nachricht, um dann nach Erhalt die Funktion `do_IRQ()` aufzurufen, die die weitere Interruptbehandlung übernimmt.

Dabei kann zwischen drei unterschiedlichen Quellen für Interrupts unterschieden werden:

- Echte Hardware Interrupts

  Hierbei versucht eine Routine mit `ipc_associate_intr()` das Recht zu bekommen, Hardware Interrupt Meldungen zu erhalten. Sie wartet danach in einer Schleife mit `ipc_receive_intr()` auf das Eintreffen der Nachrichten.

  Es gibt sogar von Seiten des Mikrokernel die Unterstützung für geteilte Interrupts zwischen mehreren Threads. Dieses Feature wird aber hier nicht genutzt.

- Zeitgeber

  Der Zeitgeber besteht aus einer kleinen Schleife und wartet mit `ipc_sleep()` auf das Eintreten eines Timeouts von 10ms.

- externen Nachrichten

  Diese Threads warten auf Nachrichten von beliebigen externen Servern, die alle über das Ladeprogramm dem Linux Kern zugestellt werden können. Beim Empfang muss der Nachrichteninhalt zwischengespeichert werden.

[2]Mehr dazu ist in der Dokumentation der Linux Kernel Quellen unter `Documentation/DocBook/kernel-locking.tmpl` zu finden.

Linux bietet zum Schutze eines kritischen Bereiches und zur Synchronisierung der Zugriffe auf gemeinsame Daten der Aktivitäten zahlreiche optimierte Lockingvarianten an, die auf einem Einprozessorsystem immer darauf hinauslaufen, die Interrupts für kurze Zeit zu sperren.

Da ein echtes Sperren der Interrupts Kernelprivilegien benötigt, die der Linux Kern unter P4 nicht hat, wird dies durch kurzfristiges Erhöhen der Thread-Priorität gegenüber den Interrupt Threads emuliert. Andere Threads wie Pager und Mappinghelfer dürfen dabei nicht durch Interrupts unterbrochen werden und erhalten eine noch höhere Priorität. Der gegenseitige Ausschluss aus kritischen Bereichen ist damit sichergestellt und eine echte Sperrung ist auch nicht nötig, da keine kritische Hardware wie Interruptcontroller programmiert werden muss.

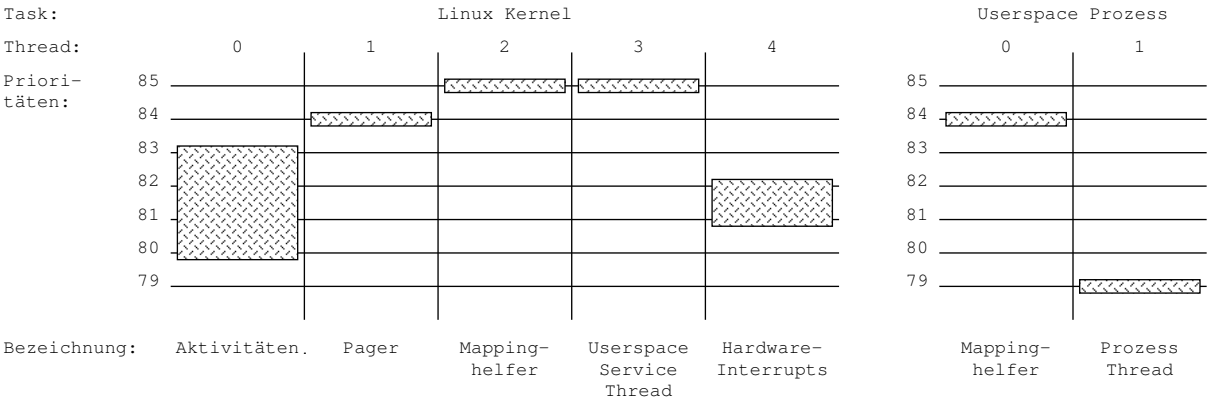

Abbildung 4.3: Prioritäten der Threads

Das Diagramm zeigt die Prioritäten der Threads mit Beispielwerten. Je wichtiger der Dienst ist, der in der Thread abläuft, desto höher ist die Priorität. Die untere Grenze des Balkens zeigt dabei die Priorität im Normalbetrieb an, die obere Grenze im geschützten Zustand.

Leider erwies sich hierbei die Annahme, dass Interrupts und Aktivitäten nichts miteinander zu tun haben, als nicht ganz zutreffend. Denn ein Interrupt setzt im TCB der aktuellen Aktivität das `need_reschedule` Flag, um diese aufzufordern, den Scheduler aufzurufen. Nur schlägt der Zugriff über `current->need_reschedule` fehl, da die Interrupt Routine in einem anderen P4 Thread und somit auf einem anderen Stack läuft.
Daher wurde `current` in eine globale Variable umgewandelt, was bei Einprozessorsystemen zu keinerlei Problemen führt.

4.6 Exceptions im Kern

Im normalen Betrieb sollten im Linux Kernel keine Exceptions auftreten. Denn dies ist nicht gewollt und weist meistens auf einen schwerwiegenden Fehler hin.

Aber hier gibt es Ausnahmen. Beim Start untersucht der Kernel den Prozessor auf even-

tuelle Fehler[3], und genau diese können Exceptions erzeugen. Für die gewollten Exceptions kann die Aktivität dynamisch zur Laufzeit einen Wiedereintrittspunkt setzen, die der Exceptionhandler berücksichtigt. Nach einer Exception kann die Aktivität ihre Arbeit dort wieder fortsetzen.

Daneben gibt es noch eine andere Methode. Bei manchen auf hohe Geschwindigkeit optimierten Routinen wird bewusst in Kauf genommen, dass sie fehlschlagen können. Zum Beispiel können so aufwendige Tests auf gültige Zeiger und Speicherseiten entfallen. Daher wird für diese Funktionen in einer zur Compile-Zeit festgelegten `ex_table` Tabelle die Adressen vom Wiederherstellungscode abgelegt, den der Exceptionhandler berücksichtigt.

Nicht behandelbare Exceptions im Linux Kernel führen zu einem Neustart.

4.7 Kernel Virtual Memory

Linux unterstützt das Konzept des Kernel Virtual Memory. In einem festgelegten Bereich wird dazu ein zweiter Speicherpool angelegt. In diesen Bereich mappt der Kernel sich Seiten aus seinem normalen Hauptspeicherpool hinein. Benötigt wird dieser virtuelle Speicher im Kernel für nachladbare Module und zur Allokierung von dynamischen Speicher über `vmalloc()`.

Der Kernel Pager ist darauf eingestellt und versucht, das gewünschte Mapping im Virtual Memory herzustellen. Alle anderen Pagefaults des Kernels werden nicht behandelt und führen zu einer Exception, die mit ziemlicher Sicherheit einem Neustart des Kernels einleitet.

Damit der Kernel dieses Mapping auch ohne den Pager ändern kann, steht ihm dazu ein P4 Thread zur Seite, der Kernel Mappinghelfer. Dieser wartet einfach hochprior in einer Endlosschleife auf neue Mappings.

4.8 Userspace

Jede Nutzerapplikation läuft unter Linux normalerweise[4] in ihrem eigenen Adressraum und hat keinerlei Zugriff auf Speicherbereiche des Kernels oder den anderer Prozesse. Um dies nachzubilden, bot sich das Task-Konzept des P4 an.

Alle Prozesse sind dabei Tasks in einem Clan (mit dem Linux Kernel als Chief) und haben nicht das Recht, eigene Tasks anzulegen. Die Kommunikation der Tasks untereinander ist zwar nicht erwünscht, lässt sich aber auch nicht verhindern. Zumindest können sie keine IPC mit Threads ausserhalb des Clans machen, da der Linux Kernel dies herausfiltert. Somit besteht für den Rest des Systems ein Schutz gegen böswillige oder fehlerhafte Tasks.

[3] Zum Beispiel der F0 0F-Bug des Pentium Prozessors, der es einem nicht-privilegierten Programm erlaubt, das System zum Stillstand zu bringen.

[4] Es gibt auch Linux-Portierungen für Prozessoren ohne MMU, dort sieht es naturgemäß anders aus.

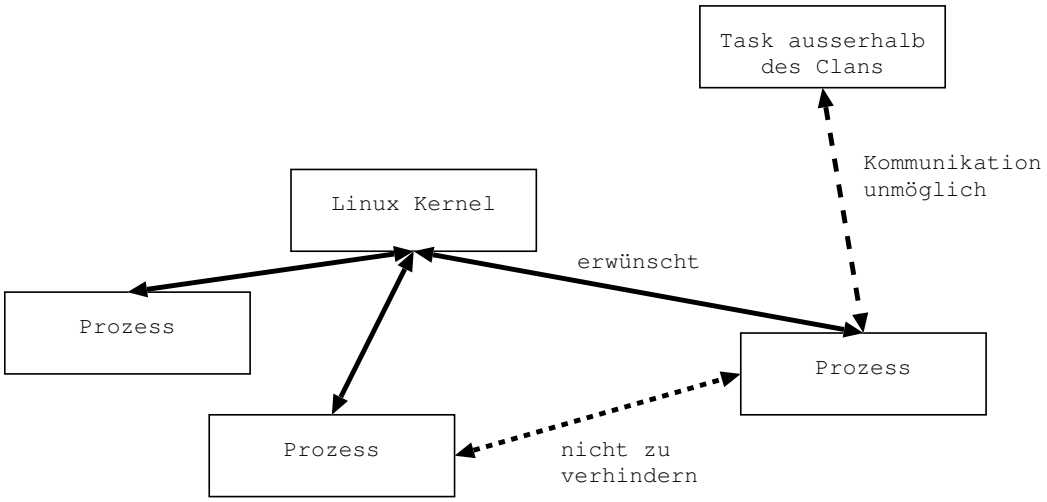

Abbildung 4.4: Kommunikation

4.8.1 Mappinghelfer

Das P4 Konzept der *Taskautonomie* verbietet es Tasks wie dem Linux Kernel, den Adressraum einer anderen Task zu beeinflussen, denn eine Task kann ein neues Mapping nur durch eine IPC empfangen. Das Mapping ist zwar nur auf Zeit, denn es kann vom Sender jederzeit mit `fpage_unmap()` wieder entzogen werden, aber das Problem ist, dass der Task die Empfangsbereitschaft für ein Mapping nicht aufgezwungen werden kann.
Da auch Pagefaults vom Mikrokernel in Nachrichten umgewandelt werden, ist es mit einem externen Pager möglich, von aussen Code einzuschleusen. Denn bei einem Pagefault setzt der Mikrokernel den Verursacher auf die nötige Empfangsbereitschaft.

Daher wird als initialer Thread beim Erzeugen der Task ein Hilfsthread angelegt. Dieser *Userspace Mappinghelfer* ist *Trusted Code* des Linux-Kernels und wird in den oberen 4 MB der Userspace-Task eingeblendet. Dieser Bereich steht dem Prozess für eigene Mappings nicht zur Verfügung.

Der Mappinghelfer wartet auf Nachrichten des Linux Kernels und kann folgende Aufgaben erfüllen:

- Speicherverwaltung

  Der Mappinghelfer nimmt `fpage_unmap()` Aufforderungen entgegen und führt diese in seinem Adressraum aus. Ein vom Kernel direkt ausgeführtes `fpage_umap()` würde dagegen die jeweilige Seite aus den Adressräumen *aller* Linux Prozesse entfernen, was bei Techniken mit gemeinsam genutzten Speicherseiten (*Copy On Write* oder *Shared Libraries*) sehr teuer wäre. Die beim `fpage_unmap()` angegebene Seite stellt der Mappinghelfer danach als Empfangspuffer ein und der Linux Kernel kann diesem dann ein neue Seite schicken.

- (remote-)lthread_exregs()

Auch `lthread_exregs()` lässt sich nur auf Threads innerhalb der selben Task anwenden. Daher wartet der Mappinghelfer auf die passende Nachricht, um für den Linux Kernel in seiner Task ein `lthread_exregs()` auszuführen. So kann der Linux Kernel von aussen einen neuen Thread für den eigentlichen Linux Prozess anlegen.

- Präemption

  Der Mappinghelfer kann auf Befehl des Linux Kernels den Nutzerthread in eine Exception zwingen. Dieser Mechanismus muss vom Mappinghelfer mit `lthread_exregs()` ausgeführt werden, da bisher keine anderen Möglichkeiten existieren, einen Thread zu präempten. Doch dazu später mehr.

Über den Mappinghelfer ist es so möglich, den Adressraum eines Linuxprozesses zu manipulieren und dort neue Threads anzulegen.

4.8.2 Speicherverwaltung

Der Linux Kernel besitzt eine dreistufige, hierarchische Speicherverwaltung und führt eigene Speichertabellen für jeden Prozess. Eine Architektur muss dies an die Gegebenheiten ihrer Hardware anpassen. Zum Beispiel unterstützt die i386 Architektur nur eine zweistufe Verwaltung mit einem *Pagedirectory* und mehreren *Pagetables*. Die mittlere Stufe wird durch Makros auf die erste Stufe abgebildet (gefaltet) und durch den Compiler wegoptimiert.

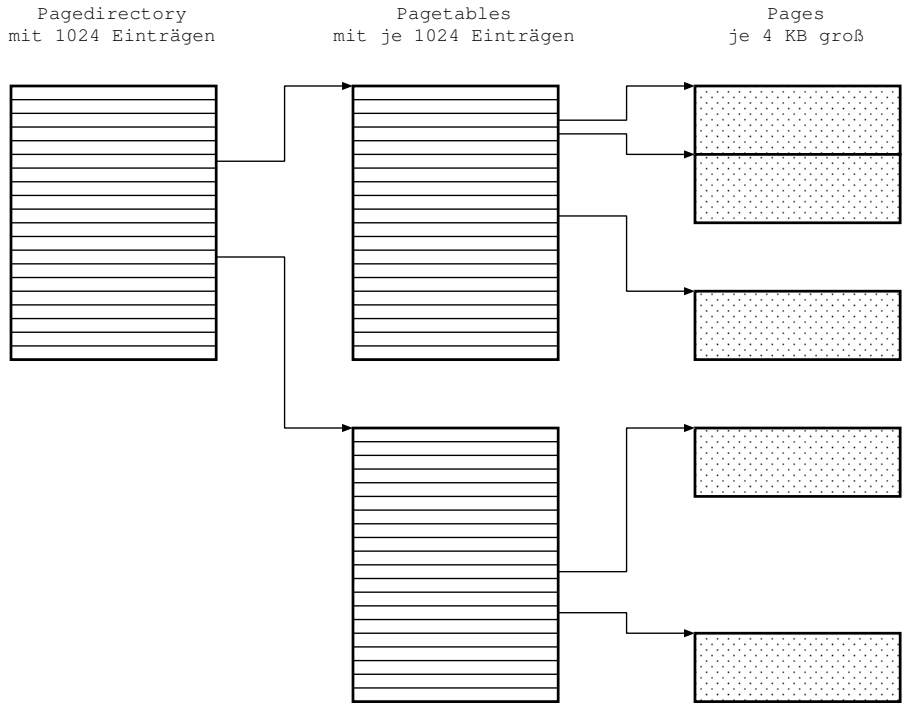

Abbildung 4.5: Speicherverwaltung mit Pagedirectory und Pagetables

Die oberen 10 Bit einer Speicheradresse von 32 Bit Breite zeigen dabei auf den Eintrag im Pagedirectory, der einen Zeiger auf die Pagetable enthält. Die nächsten 10 Bit beschreiben einen der 1024 Einträge in dieser Pagetable. Hier ist ein Zeiger auf die gewünschte physikalische Speicherseite enthalten. Da die Seiten immer auf 4K Größe ausgerichtet sind, können die unteren 12 Bit eines Pagetableeintrags Zugriffsrechte enthalten. Freie Bits, die von der Hardware nicht genutzt werden, stehen für Zusatzinformationen zur Verfügung.

Die Pagetables werden von der generischen Speicherverwaltung direkt manipuliert. Der architekturabhängige Teil stellt hierzu einige Funktionen bereit, die eine sichere Navigation in den Pagetables gewährleisten. Für einen Zugriff in die Pagedirectories (`struct pgd_t`), Page-Middledirectories (`struct pmd_t`) und Pagetables (`struct pte_t`) stehen die Funktionen `pgd_offset()`, `pmd_offset()` und `pte_offset()` zur Verfügung. Die Einträge in den Tabellen lassen sich mit Makros wie `pgd_none()` oder `pte_present()` auf Gültigkeit testen, die Zugriffsrechte einer Seite mit `pte_write()` auslesen und mit `ptr_mkwrite()` setzen.

Die Änderung in den Pagetables werden dem architekturabhängigen Teil über die `flush_cache()` und `flush_tlb()` Funktionen mitgeteilt. Mit `flush_cache()` können die Instruktions- und Datencaches des Prozessor geleert werden, wenn der Prozessor dies nicht in Hardware selber machen kann. Viel interessanter ist aber `flush_tlb()`[5]. Der TLB (*Translation Lookaside Buffer*) stellt eine Art Cache für die Umsetzung von virtuellen in physikalische Adressen im Prozessor dar. Je nach Architektur werden die Einträge im TLB vom Prozessor selber verwaltet (wie bei i386), oder es gibt eine Software-Routine, die bei einem *TLB Miss*[6] aus den Pagetables die virtuelle Adresse heraussuchtkönne und die physikalische Auflösung in den TLB einträgt (PowerPC). Über `flush_tlb()` wird nun dieser Vorgang von Hand angestoßen und alle Einträge aus dem TLB entfernt, die im Bereich des veränderten Mappings liegen.

Dies ist der Ansatzpunkt, um ein Mapping auf das Taskmodell des P4 zu übertragen. Auf dieser Schnittstelle wurden nun zwei unterschiedliche Strategien realisiert:

1. Sofortiges Mappen

   Bei der aus UML übernommenen Vorgehensweise werden die geänderten Mappings über den Mappinghelfer *sofort* in den Adressraum des Nutzerprozesses übertragen. Dabei durchsucht eine Routine die Pagetables, entfernt zunächst alte Zuordnungen mit Hilfe des Mappinghelfers und stellt dann neue ein.

   Diese Vorgehensweise reduziert die Anzahl an Pagefaults im laufenden Betrieb erheblich, da das Mapping schon vorher bekannt ist.

   Allerdings sind die Kosten auch sehr hoch, denn eine Änderung an nur einer Speicherseite bedeutet zwei Nachrichten an den Mappinghelfer (*unmap* und *map*) und damit zwei Taskwechsel.

   Nebenbei geht bei dieser Lösung auch noch die Zugriffsinformationen über die gemappte Seite verloren, da der Pager nicht aufgerufen wird und die Hardware die

[5] Weitere Informationen sind in [7] oder in der Dokumentation der Linux Kernel Sourcen unter `Documentation/cachetlb.txt` zu finden.

[6] Das Mapping ist nicht im TLB vorhanden.

Pagetables des Linux Kernels natürlich nicht anpasst.

2. Mappen bei Bedarf

   Auch hier durchsucht eine Routine zuerst die Pagetables und entfernt die geänderten Mappings. Es werden aber keine neuen Mappings eingetragen.

   Erst bei einem Pagefault wird das benötigte Mapping dem Nutzerprozess zugestellt. Im Gegensatz zur oberen Lösung steigt damit das Aufkommen der Pagefaults stark an.

   Die Anzahl an IPC ist dabei im schlechtesten Fall genauso groß wie beim sofortigen Mappen. Unter der Annahme, dass jedoch auf einen großen Teil der zu mappenen Seiten nicht zugegriffen wird, sinkt die Anzahl der IPC erheblich ab. Dies tritt zum Beispiel bei der häufig verwendeten `fork()/execve()` Kombination zum Starten eines anderen Prozesses auf, daher sollte diese Variante performanter sein.

   Zusätzlich bleibt hier ein Teil der Information über Zugriffe erhalten, denn der Pager kann sie in die Pagetables eintragen.

Eine weitergehende Vergleichsanalyse der beiden Alternativen wird in Kapitel 6 durchgeführt.

4.8.3 Zugriff in den Userspace

Für Speicherzugriffe in den Adressraum eines Prozesses stellt der architekturabhängige Teil des Linux Kernels die Funktionen `copy_from_user()` und `copy_to_user()` zur Verfügung. Diese prüfen zuerst die Gültigkeit der übergebenen Adressen, ob an der angegeben Adresse auch eine Seite liegt, und kopieren dann die gewünschten Daten einfach aus dem Adressraum des Prozesses in den Kernelbereich oder umgekehrt. Dies ist aber hier leider mit getrennten Adressräumen von Kernel und Userspace nicht möglich, und die Idee, über den Mappinghelfer die Kopieraktionen abzuwickeln, wäre sehr teuer.

Da aber ein Prozess sein Mapping grundsätzlich aus dem Pagepool des Linux Kernels erhält, reicht hier ein Blick in die Pagetables des Prozesses und es kann auf die dazugehörige Seite im Adressraum des Linux Kernels direkt zugegriffen werden.

Mit den bisher genannten Funktionen ist es nun dem generischen Teil möglich, ein Programm von einem Dateisystem zu laden und an die zugehörigen Stellen im Adressraum des Prozesses zu mappen. Es fehlt nur noch die Synchronisierung des Kerns mit den Prozessen im Userspace, die nun beschrieben wird.

4.9 Kerneintritt

Ein Nutzerprozess betritt den Linux Kern, wenn Aufgaben oder Ereignisse anstehen, die nicht im Userspace gelöst werden können. Dies kann freiwillig durch einen Systemcall (`int 0x80` auf i386) oder eher unfreiwillig durch Pagefaults oder Exceptions geschehen.

Für den P4 stellen diese Ereignisse Nachrichten dar, die an die jeweilige Kernaktivität weitergeleitet werden. Diese wartet in einer Endlosschleife auf die Nachrichten und ruft dann den gewünschten Systemcall im Kern auf oder versucht über die Speicherverwaltung des Linux Kernels, den Pagefault zu beheben.

Mit diesem Modell ist auch die Bedingung, dass sich ein Prozess entweder im Kern oder im Userspace aufhält, erfüllt, da jeweils eine Aktivität auf die Nachricht der anderen wartet.

Die Exception stellt dabei den *sicheren Zustand* dar, denn die Kernaktivität kann den Nutzerprozess beliebig lange warten lassen und den Scheduler aufrufen. Und es können Signale zugestellt werden, da der Registerkontext verändert werden kann.

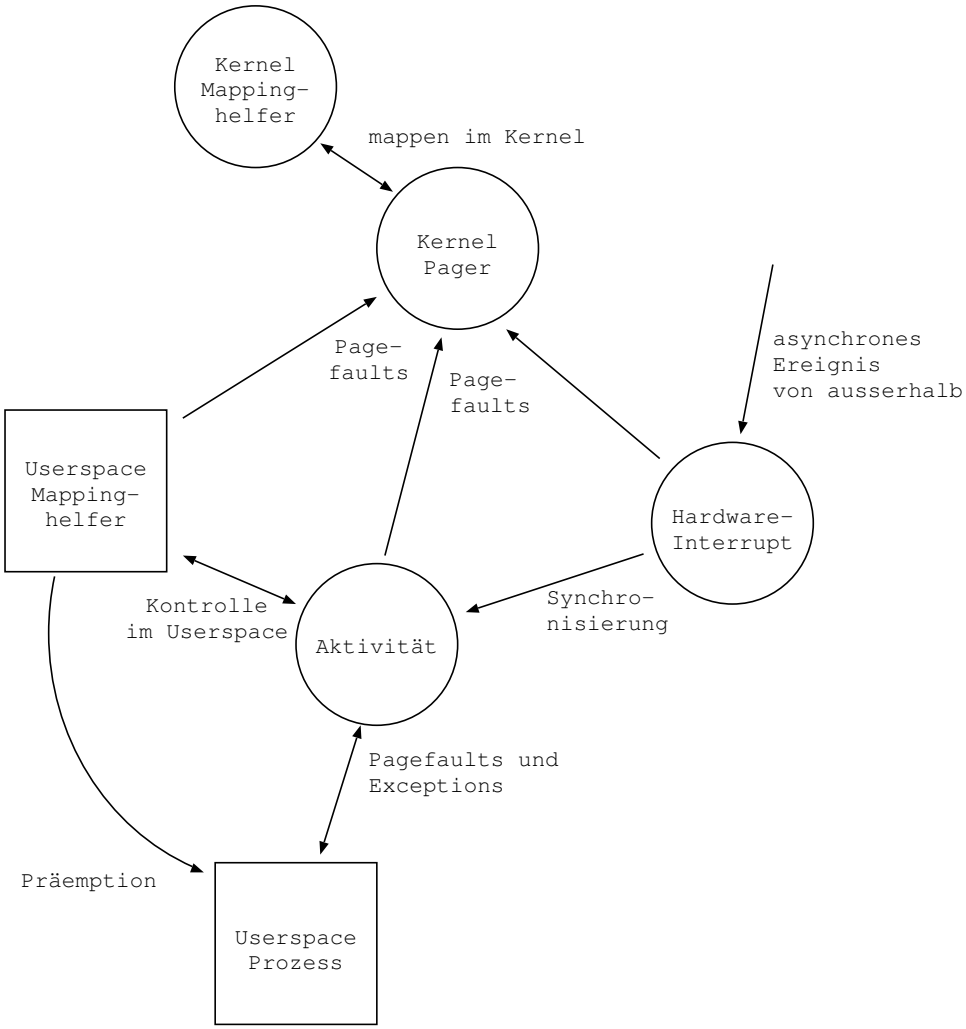

Abbildung 4.6: Abhängigkeiten der Threads untereinander

Die Position der Threads in der Vertikalen zeigt ihre Priorität an.

Problematisch sind die Hardware-Interrupts, da normalerweise dazu der Kern betreten werden muss, denn gewöhnliche Einprozessor-Architekturen haben ja nur einen Ausführungskontext. Hierbei sichert dann die Kernaktivität den Prozessorkontext auf ihrem Stack ab

und ruft den Interrupthandler auf. Nach Beendigung der Interrupts prüft die Kernaktivität, ob sie den Scheduler aufrufen muss, weil zum Beispiel die Nachbehandlung eines Interrupts ansteht oder die Rechenzeit eines Prozesses abgelaufen ist.

Dieser Mechanismus muss auf dem P4 emuliert werden und erfordert eine komplexe Synchronisierung. Dabei wird nach einem Interrupt der Kernaktivität mitgeteilt, dass sie den Scheduler aufrufen soll. Dazu sollte sich aber der Nutzerprozess im sicheren Zustand einer Exception befinden.

Die Kernaktivität prüft nun zuerst, ob der Nutzerprozess noch nicht den P4 Kern betreten hat. Es ist gut möglich, dass zum Beispiel durch einen Pagefault der P4 Kernel betreten wurde, die Abarbeitung aber durch den höherprioren Interrupt unterbrochen wurde. Wenn dies der Fall sein sollte, signalisiert die Kernaktivität Empfangsbereitschaft und nimmt die Nachricht entgegen.

Andernfalls versucht sie mit Hilfe des Mappinghelfers, den Nutzerprozess durch eine Exception zu zwingen, den Kern zu betreten. Dazu setzt dieser mit `lthread_exregs()` den Program-Counter des Nutzerprozesses auf ein Codefragment, das zu einem speziellen Systemcall (`int 0x81`) führt. Dieser Systemcall ist im Linux ABI nicht vorgesehen und wird hier für den erzwungenen Kerneintritt benutzt. Der alte Wert im Program-Counter wird dann an die Kernaktivität weitergegeben, damit diese einen sauberen Zustand des Nutzerprozesses wiederherstellen kann.

Linux vertraut auf diesen Mechanismus, denn wenn ein Prozess nicht den Kern betritt, kann dieser die gesamte Rechenzeit für sich in Anspruch nehmen, da kein Scheduling stattfindet.

4.10 Prozesserzeugung

Mit den Systemcalls `fork()`, `vfork()` und `clone()` kann ein Prozess weitere Kindprozesse anlegen. Bei einem `fork()` wird der aktuelle Prozess dupliziert, d.h. alle benutzten Ressourcen werden aus der Sicht des Prozesses verdoppelt und der Kindprozess ist unabhängig von seinem Vater. Dazu muss natürlich eine neue Task für den virtuellen Adressraum des Kindes angelegt und mit dem passenden Mapping versehen werden.

Dabei wird mit der *Copy On Write* Technik versucht, den Aufwand zu reduzieren. Anstatt direkt alle Speicherseiten vom Vater und vom Kind-Prozess doppelt anzulegen, werden erst die Seiten bei beiden auf das Zugriffsrecht *Nur-Lesen* gesetzt. Nur bei einem Schreibzugriff wird dem Verursacher eine Kopie der Seite zugewiesen.

Diese Vorgehensweise ist bei den meisten Architekturen an sich sehr schnell, da im Kernel nur neue Pagetables angelegt werden und der TLB geflusht wird. Letzteres dauert hier seine Zeit, denn eine neu erzeugte Task besitzt immer einen leeren Adressraum und bei Änderungen am Mapping ist immer der Mappinghelfer involviert. Die dadurch entstehenden Performance-Einbußen werden in Kapitel 6 besprochen.

Anders sieht es bei einem `clone()` aus. Dieser Systemcall ist eine Spezialität von Linux und ermöglicht es, *Linux Threads* zu erzeugen. Die mit `clone()` erzeugten Prozesse haben

ihre eigene Kernaktivität, laufen aber im selben Nutzeradressraum ab. Da beim Aufruf des Schedulers sichergestellt ist, dass der Nutzerprozess auf die Antwort einer Exception-Nachricht wartet, kann jede Kernaktivität den Registerkontext beliebig verändern und so mehrere Nutzeraktivitäten auf nur einen P4 Thread multiplexen. Hierbei waren keine großartigen Einbußen zu spüren.

Eine spezielle Variante von `clone()` stellt dabei `vfork()` dar. Die Spezifikation dieses Systemcalls[7] besagt, dass der Kindprozess nach einem `vfork()` sofort einen anderen Prozess mit `execve()` laden oder sich über `_exit()` beenden muss. Dabei dürfen keine Änderungen am Speicher gemacht werden, da im Gegensatz zum `fork()` Vater und Kind im selben Adressraum laufen. Erst beim `execve()` wird dann ein neuer Adressraum angefordert. Dies erspart gegenüber einer `fork()`/`execve()`-Kombination alle Änderungen im Adressraum des Vaterprozesses.

4.11 Verschiedenes

Mit den in diesem Kapitel beschriebenen Techniken können Anwendungen aus einer Ramdisk heraus gestartet werden. Der Linux Kern braucht dabei einen externen Konsolenserver, um mit dem Anwender zu kommunizieren, denn aufgrund fehlender Hardwarezugriffe laufen andere Gerätetreiber noch nicht.

Auch werden einige Systemcalls nicht unterstützt. Dies sind prozessorspezifische Aufrufe wie `modify_ldt()` und `vm86()` zur Steuerung von DOS-Emulatoren, sowie `iopl()` und `ioperm()`, die einem Nutzerprozess das Recht geben, direkt auf den IO-Adressbus des i386 zu zugreifen. Dies ist aber eine Einschränkung des P4 Mikrokernels, da diese Befehle nur im privilegierten Modus ausgeführt werden können.

Daneben fehlt noch die Unterstützung für Gleitkomma Operationen. Dazu müssen vom P4 her die Gleitkomma Register des Koprozessors bei Taskumschaltungen gesichert werden, sowie ein Protokoll zur Abfrage dieser definiert werden. Dieses Feature ist im Mikrokernel auf dem i386 bisher nicht implementiert. Daher liegt die Unterstützung für das Sichern und Wiederherstellen des Gleitkomma Kontextes von Nutzerprozessen zur Zeit noch brach.

Trotzdem können zu diesem Zeitpunkt schon viele Kompatibilität-Tests und Performance Messungen gemacht werden, da alles notwendige dazu vorhanden ist; und es lassen sich auch mehrere Linuxe nebeneinander auf dem P4 starten.

[7] Siehe dazu die Manpages: `man 2 vfork`

Kapitel 5

Gerätetreiber

Im Linux Kern sind viele, meist sogar plattformunabhängige, Gerätetreiber enthalten. Diese sollen natürlich auch ohne großartige Änderungen auf dem P4 genutzt werden.

Der Mikrokernel stellt dazu Speicherseiten auf Hardware-Register bereit, die eine Task anfordern kann. Dies sind für den Mikrokernel normale Flexpages, die per IPC weitergegeben werden können.

Aber nicht auf jeder Plattform ist die Hardware immer als *Memory Mapped IO*[1] verfügbar. Bei der i386 Architektur gibt es zum Beispiel einen separaten IO-Bus, der über die Prozessorbefehle `in` und `out` *portweise* angesprochen wird. Der Mikrokernel sieht dazu spezielle IO-Flexpages vor, die mit einer Granularität von 1 den gesamten IO-Bus abdecken, denn der i386-Prozessor erlaubt eine genaue Zuordnung der Zugriffsrechte über eine Bitmaske für alle 65536 Ports. Leider war im P4 die Unterstützung noch nicht ausprogrammiert. Als Workaround wurde daher der Linux Kernel Task das Recht gegeben, auf alle Ports direkt zuzugreifen.

5.1 Hardwarezugriff

Die architekturabhängige Schicht des Linux Kernels stellt den Gerätetreibern einige Funktionen für den Zugriff auf die Hardware zur Verfügung.

- `inb()`, `inw()` und `inl()` sind dabei zum Auslesen von und
- `outb()`, `outw()` und `outl()` zum Schreiben auf IO-Port gedacht sind.

Die Bezeichnung IO-Port hängt mit der Geschichte des Linux Kernels zusammen, der zuerst auf der i386-Architektur realisiert wurde. Unterstützt die Zielplattform keinen IO-Bus, werden stattdessen Speicherzugriffe auf die jeweilige Hardware gemacht.

[1] Dabei liegen die Kontrollregister einer Hardware-Komponente an festgelegten Stellen im Speicher.

Die angehängten Buchstaben *b* für Byte, *w* für Word und *l* für Long bezeichnen die Breite eines Ports.

Mit dem oben angegebenen Workaround war es dann möglich, direkt auf eine Netzwerkkarte zuzugreifen.

5.2 Timing

Leider stellte sich dabei heraus, dass die `delay()`-Funktionen, die Verzögerungen im Millisekundenbereich bereitstellen, nicht funktionierten. Die Zeiten waren viel zu kurz für die relativ langsame Netzwerkkarte. Nach der Korrektur der Timingfunktionen wurde die Netzwerkkarte richtig erkannt und arbeitete ohne weitere Probleme.

Grundsätzlich ist damit aber das Problem nicht gelöst. Denn in der Bootphase wird zur Zeitmessung die Anzahl Schleifendurchläufe zwischen zwei Ticker-Interrupts gezählt. Problematisch ist dabei, dass sowohl die Zeitmessung als auch der Ticker durch andere Threads unterbrochen werden können und damit die Verzögerung wieder zu gering wird.

5.3 Hardware-Interrupts

Die ersten Gerätetreiber, die aktiviert werden konnten, waren eine 3Com Netzwerkkarte Typ 3c509 und der Festplattencontroller auf der Hauptplatine. Diese Geräte haben eine Sonderstellung, da die benutzten Interrupts bekannt sind.

Problematischer werden sich wohl die Gerätetreiber verhalten, die ein *IRQ Probing* machen, um den Interrupt herauszufinden. Dieser Fall ist bisher ungetestet. Der Linux Kernel bewacht dabei alle Hardware-Interrupts, während der Treiber versucht, einen auszulösen. Viele Gerätetreiber lassen es aber zu, den jeweils verwendeten Interrupt direkt anzugeben, somit ließe sich das Problem erst einmal umgehen.

5.4 DMA

Theoretisch besteht die Möglichkeit für den Linux Kernel, einen DMA-Transfer einzuleiten. Dazu wird neben dem Zugriff auf einen DMA-Controller auch die exakte physikalische Adresse einer Speicherseite benötigt, die Linux von seinem Ladeprogramm bekommt.

Die notwendigen Architekturanpassungen wurden aber bisher nicht durchgeführt, da in einem sicherheitskritischen System eine unzertifizierte Anwendung keine DMA-Transfers machen darf. [2] In dieser Hinsicht sollte DMA nur über einen externen, separat zertifizierten, Server realisiert werden.

[2] Da die Anwendung auf *jede* physikalische Adresse nach belieben Zugriff hat.

5.5 Zusammenfassung

Mit etwas Handarbeit ist es möglich, im generischen Linux die Gerätetreiber zu aktivieren. Dabei werden bisher längst nicht alle Konzepte und Möglichkeiten des Mikrokernels ausgenutzt, da teilweise die Unterstützung fehlt oder aus Sicherheitsaspekten heraus die Unterstützung eher in externen Servern realisiert werden sollte. Dieser Umstand war aber schon am Anfang der Portierung bekannt.

Kapitel 6

Performance

6.1 Testumgebung

Als Testumgebung wurde in der ersten Phase der Portierung am Linux Kern der *Bochs*[2] benutzt. Bochs ist ein freier Emulator, der einen vollständigen i386-kompatiblen PC (eine Bo*x*, daher der Name) inklusiv BIOS, Grafik- und Netzwerkkarte zur Verfügung stellt. Dazu kommt noch eine Schnittstelle für den *gdb*-Debugger, um sämtliche Registerinhalte jederzeit kontrollieren zu können. Vorteilhaft beim Bochs ist das deterministische Verhalten, was es dem Entwickler ermöglicht, den gleichen Fehler schnell wieder zu reproduzieren.

In der späteren Phase der Entwicklung, als die Unterstützung für den Usermode eingebaut wurde, war die Performance der Bochs Emulation nicht mehr ausreichend. Von da an wurde ein reales Testsystem eingesetzt. Es ist ein normaler PC mit einem AMD K6 Prozessor mit 233 MHz, dazu 128 MB RAM-Speicher, einer sehr alten Hercules-Monochrom Grafikkarte, einer 3Com 3c509 Netzwerkkarte und den üblichen Schnittstellen auf der Hauptplatine.

Zur Fehlersuche im System wurde dabei die im P4 Mikrokernel enthaltene gdb Unterstützung genutzt. Dabei kommuniziert der Debugger auf dem Hostsystem über eine serielle Verbindung mit der gdb-Schnittstelle im Mikrokernel und kann darüber die Register auslesen oder Breakpoints setzen.

6.2 Testsuite

Die Konformität des P4/Linux zu den Linux Standards wurde mit der Testsuite des *Linux Test Project*[1] geprüft. Vor allen Dingen war eine saubere Implementierung der Systemcalls wichtig.

Anhand der Testsuite wurde nun jeder Systemcall geprüft, und bei Unstimmigkeiten ver-

[1] Im Internet unter `http://ltp.sourceforge.net/` zu finden.

sucht, die Probleme zu beheben. Die beiden wichtigsten Systemcalls, die einen Großteil des architekturabhängigen Codes testeten, waren `fork()` und `execve()`. Dabei zeigten sich dann sehr schnell die noch vorhandenen Schwächen in der Speicherverwaltung, die nun beseitigt sind.

6.3 Ausgangsbasis

Bei allen Geschwindigkeitsmessungen, die in diesem Kapitel erwähnt werden, wurde versucht, den unterschiedlichen Systemen immer eine gleiche Ausgangsbasis zu geben. Da dem Linux Kern unter P4 nur 32 MB Arbeitsspeicher zur Verfügung steht, wurde auch beim nativen Linux die Speichergröße mit dem Bootparameter `mem=32M` eingeschränkt.

Beide nutzen dieselbe Kernelkonfiguration und ähnliche Compiler-Parameter wie `-O2` zur Optimierung. Das ist sehr wichtig, da Linux sich auf eine saubere "Wegoptimierung" von Schleifen und unnötigen Codezweigen verlässt. Vor allem in den architekturabhängigen Headern befinden sich sehr viele Definitionen wie

```
#define flush_page_to_ram(page)   do { } while (0)
```

Daneben wurde die Debuggerunterstützung im Mikrokernel abgeschaltet, um auch dort die volle Geschwindigkeit zu erhalten.

Auch der Festplattentreiber wurde in seiner Grundeinstellung gelassen, da unter P4/Linux Features wie PCI-Busmaster-DMA noch nicht zur Verfügung stehen.

6.4 Anwendungsbenchmark

Der erste hier besprochene Benchmark ist ein Anwendungsbenchmark: Kompilieren eines Linux-Kernels. Dies deckt alle wichtigen Subsysteme des Kernels ab und gibt einen guten Eindruck über die Gesamtperformance des Systems.

Ein kleines Shell-Skript automatisierte die Tests. Dabei wurde für jeden Test eine Einzelzeitmessung vorgenommen, die Gesamtzeit aller Tests sowie eine Referenzzeit von einem anderen Rechner gemessen. Der Vergleich mit einer Referenzzeit ist wichtig, da P4/Linux unter Umständen einige Zeitgeber-Interrupts verpassen kann und keine Echtzeituhr zur Synchronisierung benutzt.

Es waren dabei folgende Aufgaben zu erfüllen:

- Dekomprimieren des Linux-Kernel-Archivs ohne Schreibzugriffe auf die Festplatte
  `$ bzcat linux-2.4.20.tar.bz2 >/dev/null`
- Entpacken des Archives mit Schreiben der Dateien
  `$ bzcat linux-2.4.20.tar.bz2 | tar xv`

- Vorbereiten des Kompiliervorganges
  `$ make distclean && cp ../kconfig .config`
- Erfassung der Abhängigkeiten der Dateien untereinander. (Durch einen Fehler im Mess-Skript wurde die zeitintensive Arbeit der Erfassung der Abhängigkeiten doppelt ausgeführt.)
  `$ yes n | make oldconfig && make dep`
  `$ make dep`
- Kompilierung des Linux Kernels
  `$ make bzImage`
- Aufräumarbeiten, Löschen aller Dateien
  `$ rm -rf linux-2.4.20`

Alle Tests liefen für jede Konfiguration drei mal durch. Die Ergebnisse zeigen das Arithmetische Mittel der jeweiligen Messreihen pro Typ, die Abweichungen der Gesamtzeiten der Messreihen des selben Typs betrugen maximal 1%. Aufgrund der nur sekundengenauen Messung und dem Drift der Systemzeit zur Referenzzeit setze ich die Messungenauigkeit bei 3% der Gesamtzeit an.

	Natives	Mappinghelfer		fpage_remap	
	Linux	Sofort	Bedarf	Sofort	Bedarf
bzcat allein	127	136	135	137	136
bzcat und tar	187	234	232	234	234
Vorbereitung	14	27	28	24	29
Abhängigkeiten (2x)	335	602	617	528	549
Kompilierung	474	700	716	639	667
Löschen	17	27	26	25	27
Summe	1154	1725	1755	1587	1642
Lokale Zeit	1158	1732	1762	1592	1650
Referenzzeit	1158	1761	1793	1607	1666
Abweichung	1	29	31	15	16

Einheit: Sekunden

Die erste Spalte zeigt die Werte eines nativen Linux auf dem Rechner. Diese sind als Referenz zu sehen. In der zweiten und dritten Spalte werden die Messwerte der in Kapitel 4.8.2 vorgestellten Speicherverwaltung gezeigt.

Die letzten beiden Spalten zeigen die Performance nach einer weiteren Optimierung. Dazu wurde im Mikrokernel zu Testzwecken ein neuer Systemcall integriert, `fpage_remap()`. Damit kann der Adressraum einer Task direkt manipuliert werden, anstatt den Umweg über den Mappinghelfer nehmen zu müssen. Die beiden Strategien der Speicherverwaltung bleiben aber auch hier erhalten. Im besten Falle können somit zwei Nachrichten, und damit verbunden zwei Kontextwechsel, mit genau einem P4 Systemcall abgedeckt werden.

Die Abweichungen zwischen lokaler und Referenzzeit unter den P4/Linux-Varianten lassen auf einige verpasste Zeitgeberinterrupts schliessen.

Weiterhin fällt das schlechte Abschneiden bei der Erstellung der Abhängigkeiten auf. Dabei wird rekursiv über den Verzeichnisbaum des Linux-Kernels ein Makefile gestartet, das nur ein kleines Hilfsprogramm aufruft. Dies deutet auf große Performanceverluste bei der Prozesserzeugung hin, die es lohnt, weiter zu untersuchen.

Die Unterschiede zwischen den beiden Varianten der Mappingstrategien, sofortiges Mappen oder erst bei Bedarf, sind weniger gravierend, als zunächst vermutet wurde. Die erste Strategie zeigt aber einen kleinen Vorsprung. Der Einsatz des `fpage_remap()` Systemcalls bringt dagegen einen großen Performanceschub.

6.5 Prozesserzeugung

Ein nicht sehr praxisrelevanter Benchmark kommt hier zum Einsatz, um die Zeiten für Systemcalls und Prozesserzeugung zu messen. Eine einfache Schleife ruft dabei 1000 mal das zu testende Codefragment auf, um die bestmöglichen Werte zu bekommen, da alle Instruktionen in den Caches des Prozessors vorhanden sein sollten.

Zur Zeitmessung bot sich der *Timestamp-Counter* im Prozessor an. Dieser 64 Bit breite Zähler erfasst jeden Prozessorzyklus und ermöglicht so eine taktgenaue Messung, die unabhängig von der Systemuhr ist. Hierbei wurden für jede Variante jeweils 5 Durchläufe gewertet; diese zeigten Schwankungen von bis zu 5%.
Die Tabelle zeigt das arithmetische Mittel dieser Durchläufe in 1000 Prozessorzyklen an.

	Natives	Mappinghelfer		fpage_remap	
	Linux	Sofort	Bedarf	Sofort	Bedarf
getpid()	0,3	9,5	9,6	8,8	8,9
fork() + _exit()	60,5	6454,8	6850,2	3044,9	3025,7
vfork() + _exit()	13,6	90,9	92,7	92,7	92,1
fork() + execve()	190,4	10814,6	11779,4	5145,7	5488,1
vfork() + execve()	186,7	10662,0	11622,2	5133,0	5471,6

Einheit: 1000 Prozessorzyklen

In der ersten Zeile werden die Ergebnisse eines `getpid()`[2] Aufrufs miteinander verglichen. Da beim P4/Linux zwei IPC und zwei Taskwechsel nötig sind, erklärt dies den großen Unterschied der Ergebnisse zwischen nativem Linux und P4/Linux. Die Schwankungen in den Messwerten der Mappinghelfer gegenüber der fpage_remap Variante sind durch Hintergrundprozesse zu erklären, die von der höheren Geschwindigkeit des `fpage_remap()` Systemcalls profitieren.

Die nächsten beiden Zeilen zeigen die Messwerte für eine Taskerzeugung. Bei einem `fork()` wird sofort ein neuer Adressraum angelegt, was sehr teuer unter P4/Linux ist. `vfork()`

[2] Siehe Kapitel 3.2.

als Spezialfall des `clone()` Systemcall ist hier wesentlich schneller. Dass hier für den Kindprozess kein neuer Adressraum angelegt werden muss, zeigen die Messwerte sehr deutlich.

Bei den Messwerten in den letzten beiden Zeilen ruft der Kindprozess noch eine andere Anwendung mit `execve()` auf. Hierbei wird der gesamte Adressraum des Kindes geleert und anschliessend mit neuen Mappings befüllt.

Die Messungen mit dem `fpage_remap()` Systemcall in der vierten und fünften Spalte zeigen eine Halbierung der Zeiten gegenüber der Variante mit dem Mappinghelfer. Dies entspricht den Erwartungen, da hier nur ein Systemcall gegeüber zwei IPC notwendig ist.

6.6 Bewertung

Auch wenn die Leistungsunterschiede im Prozesserzeugungsbenchmark sehr drastisch erscheinen, die Performance im Anwendungsbenchmark ist ausreichend. Denn bei den Gesamtzeiten benötigt das P4/Linux je nach Ausführung 40% bis 55% mehr Zeit als ein natives Linux. Dies liegt im erwarteten Rahmen für einen so frühen Stand der Portierung, da noch einige Optimierungen möglich sind, sowohl im P4/Linux als auch im Mikrokernel.

Weitere Messungen und Analysen an der Speicherverwaltung sind daher notwendig, um die Ursachen der Performanceeinbußen genauer herauszuarbeiten.

Kapitel 7

Fazit

Nach fünfmonatiger Entwicklungszeit wurde eine erste lauffähige Portierung des Linux Betriebssystemkerns auf den P4 Mikrokernel erreicht. Die Anwendungen einer SuSE 7.0 Distribution wie der KDE Desktop oder Netscape laufen ohne weitere Modifikationen.

Die Entscheidung für Usermode-Linux als Ausgangsbasis des Projektes hat sich als richtig erwiesen, da so die Entwicklungszeit kurz gehalten werden konnte.

Die unvermeidlichen Performanceeinbußen stellen sich als akzeptabel für einen frühen Stand des P4/Linux und auch des Mikrokernels dar. Da sich auch der P4 noch in seiner Testphase befindet, sind durch die Portierung einige Fehler gefunden worden.

Für die Zukunft stehen bei diesem Projekt noch folgende Aufgaben an:

Stabilität: Die aktuelle Implementierung hat noch Probleme mit der Stabilität bei Fehlverhalten der Anwendungen, was zu einem gewollten Absturz führt, um die Fehlerursache genauer analysieren zu können. An diesen Stellen muss im Linuxserver noch einige Arbeit geleistet werden.

Tests: Es müssen viele weitere Tests gemacht werden, um bisher unentdeckte Fehler zu beseitigen.

Optimierung: Da die Leistung im Bereich der Prozesserzeugung noch signifikante Einbußen zeigt, sind auch hier weitere Analysen und Optimierungen notwendig.

Anpassung: Der P4 Mikrokernel befindet sich noch in der Entwicklung zu einem zertifizierbaren und sicheren Betriebssystemkern für Echtzeitanwendungen in sicherheitskritischen Bereichen. Daher muss auch die Linuxportierung an veränderten Spezifikationen und Schnittstellen des Mikrokernels angepasst werden.

Weitere Architekturen: Eine Portierung auf weitere Architekturen wie ARM, PowerPC und MIPS muss durchgeführt werden. Der Aufwand dazu dürfe möglichst gering ausfallen, da das Mikrokernel API auf allen Plattformen gleich implementiert ist.

Mir persönlich hat es Spaß gemacht, an der Portierung zu arbeiten, denn ich habe einen tiefen Einblick in die Funktionsweise der beiden Kerne bekommen.

Anhang A

Glossar

ABI: Engl. Abk. für "Application Binary Interface". Genaue Definition der Binärschnittstellen eines Systems. Hier werden u. a. die bei einem Funktionsaufruf zu rettenden Register und das Stacklayout festgelegt.

Aktivität: Meint in dieser Arbeit das logische Linux-Äquivalent zu einem Thread des P4. In Anlehnung an [8] wurde diese Bezeichnung übernommen, um den Unterschied zwischen Linux Threads und P4 Threads besser zu verdeutlichen.

API: Engl. Abk. für "Application Programming Interface". Meint die logische Schnittstelle in einer Hochsprache wie *C*.

ARM: 32-Bit-Prozessorarchitektur der Firma *ARM Ltd.*, `http://www.arm.com/`.

Cache: Schneller Zwischenspeicher.

Chief: "Häuptling" eines *Clans*. Der Chief ist die übergeordnete, verwaltende Task innerhalb eines Clans.

Clan: Menge von Tasks, die alle von der selben anderen Task, dem *Chief*, erzeugt worden sind.

DMA: Engl. Abk. für *Direct Memory Access*. Diese Technik ermöglicht externer Hardware einen direkten Zugriff auf den Hauptspeicher.

Exception: Engl. für Ausnahme. Ein Programm löst einen (Fehler-)Zustand aus, der ausserhalb behoben werden muss. Der Mikrokernel generiert daraus eine Exception-Nachricht.

Exceptionhandler: Ein Exceptionhandler wartet auf die Exception-Nachricht eines Threads und versucht, den Fehler zu beheben. Der Exceptionhandler kann den kompletten Registerkontext des Threads ändern.

GNU: Rekursives Akr. für *GNU's Not Unix*. Das GNU-Projekt hat einen freien Unix-Nachbau als Ziel. Siehe auch `http://www.gnu.org/`.

Honeypot: Engl. für Honigtopf. Als *Honeypot* wird ein System bezeichnet, das zur Studie von Angriffen auf dieses System betreitgestellt wird.

i386: 32-Bit-Prozessorarchitektur, von der Firma *Intel Corp.* entwickelt, `http://www.intel.com/`.

Interrupt: Engl. für Unterbrechung. Ein externes Gerät teilt damit dem Prozessor mit, dass dieser die aktuelle Arbeit zwecks Behandlung eines Ereignisses unterbrechen soll.

Interrupt-Controller: Steuerbaustein zum Multiplexen vieler Interruptquellen auf wenige Interrupt-Leitungen zum Prozessor. Hiermit können auch einzelne Interrupts gesperrt werden.

IPC: Engl. Abk. für *Inter Process Communication*, Interprozesskommunikation.

I/O: Engl. Abk. für *Input/Output*, Ein-/Ausgabe. Bezeichnet einen Mechanismus, über den ein Prozessor mit Peripherie kommunizieren kann.

Mapping: Zuordnung von Speicher im virtuellen Adressraum auf physikalische Seiten.

MIPS: Prozessorarchitektur der Firma *MIPS Technologies, Inc.*, `http://www.mips.com/`. Der P4 Mikrokernel wurde u. a. auf die Prozessorserien R3000, R4000 und R5000 im 32-Bit Modus portiert.

MMU: Engl. Abk. für *Memory Management Unit*. Komponente des Prozessors, die sich um die Auflösung von virtuellen in physikalische Adressen kümmert.

OS: Engl. Abk. für *Operating System*, Betriebsystem.

OSEK: *Open systems and the corresponding interfaces for automotive electronics*, `http://www.osek-vdx.org/`. Ein Standard für Betriebssysteme im Automobilbereich.

Page: Engl. für *Speicherseite*.

Pagedirectory: Eine Tabelle mit Startadressen der jeweiligen Pagetables. Die Größe beträgt 4096 Byte mit jeweils 1024 Einträgen. Somit kann ein Adressraum von $2^{32} = 4294967296$ Byte abgedeckt werden.

Pagefault: Engl. für *Seitenfehler*. Ein Seitenfehler tritt auf, wenn im Pagedirectory oder in den Pagetables der zu einer Adresse gehörige Eintrag ungültig ist oder die Zugriffsrechte missachtet wurden.

Pager: Kurzform für *Pagefault-Handler*. Bezeichnet einen Thread, der auf *Pagefault-Nachrichten* eines anderen Threads wartet und diesem dann an ein Mapping gibt, um den Fehler zu beheben.

Pagetable: Eine Tabelle mit Einträgen zur Übersetzung von virtuellen Speicherseiten in physikalische. Eine Pagetable hat hier die Größe von 4096 Byte und kann 1024 Einträge verwalten. Sie deckt damit $2^{22} = 4194304$ Byte ab. Die feste Ausrichtung der Speicherseiten auf ein Vielfaches ihrer Größe ermöglicht es, in den unteren 10 Bit eines Eintrags zusätzlich Statusinformationen und Zugriffsrechte abzulegen.

POSIX: Engl. Akr. für *Portable Operating System Interface for computing environments*. Beschreibt einen portablen Standard zwischen Anwendungen und Betriebssystem.

PowerPC: Eine Prozessorarchitektur. Der P4 Mikrokernel wurde auf die 32-Bit 60x-Serie portiert.

Präemption: *Unterbrechung* eines Threads, wenn seine *Zeitscheibe* abgelaufen ist.

Prozess: Nutzerprogramm unter Linux mit Adressraum, Prozessorkontext, Zugriffsrechten, geöffneten Dateien sowie vielen weiteren Statusinformationen. Jeder Prozess ist durch seine *PID* (Process ID) eindeutig zu identifizieren.

Prozessorkontext: Inhalt aller *Prozessorregister* wie Akkumulator, Stackpointer oder Program-Counter.

Sandbox: Engl. für Sandkasten. Meint eine gesicherte Umgebung zur Ausführung von unbekannten Applikationen.

Scheduling: Die Zuteilung von *Zeitscheiben* und damit Rechenzeit an Threads.

Seitenfehler: Siehe *Pagefault*.

Speicherseite: Eine Speicherseite beschreibt eine feste Anzahl von zusammenhängenen Speicheradressen. Die Größe einer Speicherseite beträgt bei der i386-Architektur zum Beispiel 4096 Byte. Weiterhin fangen Speicherseiten immer bei Adressen an, die ein Vielfaches der Größe betragen.

Stack: Engl. Bezeichnung für den Stapel- oder Kellerspeicher. Dieser arbeitet nach dem *Last In First Out* Prinzip. Wird von einem Programm zur Speicherung von lokalen Variablen und Funktionsparametern genutzt.

Systemcall: Engl. für *Systemruf*. Aufruf einer bestimmten Funktion im Kern. Vom P4 Mikrokernel nicht behandelte Systemcalls werden in Nachrichten für einen Exceptionhandler umgewandelt.

Task: Eine Task besteht aus einem virtuellen Adressraum und ihren Threads.

TCP/IP: Abk. für *Transmission Control Protocol / Internet Protocol*. Das Übertragungsprotokoll des Internets.

Thread: Logische ausführende Einheit auf dem P4 Mikrokernel. Zu einem Thread gehört auch der Kontext der Prozessorregister und ein Stack.

TLB: Engl. Abk. für *Translation Lookaside Buffer*. Ein Cache, der die Umsetzungen von virtuellen in physikalische Adressen im Prozessor zwischenspeichert.

UID: Engl. Abk. für *Unique Identifer*. Bezeichnet einen eindeutigen Schlüssel, um einen Thread im P4 Mikrokernel zu identifizieren.

UML: Abkürzung für *Usermode Linux*. Dieses spezielle Linux läuft als vollständiges Betriebsystem als Anwendung auf einem anderen Linux ab.

Userspace: Alles, was ausserhalb des Kerns ist. Dies gilt für den P4 mit seiner Sicht auf den Speicher der Anwendungen wie auch für den Linux Kernel.

Virtueller Adressraum: Ein Adressraum besteht aus der Menge aller Speicherseiten und einer Abbildungsvorschrift, auf die ein Programm zugreifen kann.

Zeitscheibe: Die Rechenzeit eines Threads.

Literaturverzeichnis

[1] Au, Alan; Heiser, Gernot: *L4 User Manual, Version 1.14*, 1999. http://www.cse.unsw.edu.au/~disy/L4/MIPS/l4uman.ps.gz.

[2] *The Bochs IA-32 Emulator Project.* http://bochs.sourceforge.net/.

[3] Dike, Jeff: *User-mode Linux*, 2001. http://user-mode-linux.sourceforge.net/als2001/.

[4] Härtig, Hermann; Hohmuth, Michael; Liedtke, Jochen; Schönberg, Sebastian; Wolter, Jean: *The Performance of µ-Kernel-based Systems.* 16th ACM Symposium on Operating Systems Principles (SOSP), 1997. http://os.inf.tu-dresden.de/papers_ps/sosp97.ps.gz.

[5] Flöck, Martinus: *Diplomarbeit: Portierung des Mikrokernels P4 auf einen ARM-Prozessor*, 2002.

[6] *The GNU Project.* http://www.gnu.org/.

[7] Gorman, Mel: *Thesis: Unterstanding The Linux Virual Memory Manager*, 2003. http://www.csn.ul.ie/~mel/projects/vm/.

[8] Hohmuth, Michael: *Diplomarbeit: Linux-Emulation auf einem Mikrokern*, 1996. http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-l4/diplom/.

[9] *The Linux Kernel Archives.* http://www.kernel.org/.

[10] *The L4-µKernel Family.* http://os.inf.tu-dresden.de/L4/.

[11] *L4-Linux: Linux on L4.* http://os.inf.tu-dresden.de/L4/LinuxOnL4/.

[12] Liedtke, Jochen: *On µ-Kernel Construction.* 15th ACM Symposium on Operating Systems Principles (SOSP), 1995. http://os.inf.tu-dresden.de/papers_ps/jochen/Mikern.ps.

[13] Liedtke, Jochen: *L4 Reference Manual, Version 2.0*, 1996. http://os.inf.tu-dresden.de/L4/l4refx86.ps.gz.

[14] Roebrock, Philipp: *Diplomarbeit: Portierung des P4-Mikrokernels auf eine PowerPC Plattform*, 2002.

[15] *SYSGO Realtime Solutions*. http://www.sysgo.de/.

[16] *The User-mode Linux Kernel Home Page*. http://user-mode-linux.sourceforge.net/.