Großer Beleg
Dynamisches Nachladen von Komponenten in DROPS

Jens Nerche
Technische Universität Dresden

7. Mai 1999

Inhaltsverzeichnis

Kapitel 1

Einleitung

Ein zentrales Problem im Dresden Realtime Operating System (DROPS) ist der Umgang mit Komponenten. Diese werden derzeit beim Booten geladen und gestartet. Ihr Adressraum wird vom Ressourcenmanager mittels einer einfachen Abbildung physischer auf virtuelle Adressen implementiert. Ein dynamisches Nachladen und Starten von Komponenten und das Anfordern von Speicher mit speziellen Eigenschaften ist nicht möglich. Mit diesen Unzulänglichkeiten befasst sich diese Arbeit. Das Ziel besteht darin, das Nachladen von Programmen während der Laufzeit des System zu ermöglichen und Speicher mit speziellen Semantiken bereitzustellen.

Das dynamische Nachstarten und der flexiblere Umgang mit Speicher setzen eine Low-Level-Speicherverwaltung voraus, deren Fähigkeiten über die der schon existierenden Ressourcenmanager und Puffermanager hinausgehen. Diese Speicherverwaltung muss in der Lage sein, Speicher mit und ohne besondere Eigenschaften bereitzustellen und freigegebenen Speicher von der Anwendung zurückzunehmen. Auf höherer Ebene wird ein Regionenverwalter den virtuellen Adressraum der Applikation organisieren. Vorgesehen ist weiterhin, dass ein Pager mit Hilfe der Speicherverwaltung Seitenfehler auflöst und die betreffende Speicherseite in den Zieladressraum einblendet.

Ein Loader, der die Applikationen laden wird, muss ELF-Binärdateien parsen können. Somit können Entwickler die Standardentwicklungswerkzeuge des Linux zum Erzeugen der Komponenten verwenden.

Im folgenden Kapitel werden der Mikrokern L4, auf den DROPS aufbaut, bereits vorhandene DROPS-Komponenten und bekannte Speicherverwaltungsmodelle vorgestellt. Im Kapitel 3 wird eine Speicherverwaltung entworfen. Darauf aufbauend werden der Regionenverwalter, der Pager und der Loader entwickelt. Das Kapitel 4 beschreibt die Implementierung, zu der der Entwurf führte.

An dieser Stelle möchte ich all jenen danken, die mich bei dieser Arbeit unterstützt haben, insbesondere gilt mein Dank der Gruppe Betriebssysteme der TU Dresden für deren Rat und Tat.

Kapitel 2

Stand der Technik

DROPS basiert auf dem Mikrokern L4. Die für diese Arbeit interessanten Eigenschaften dieses Kerns werden kurz umrissen. Hernach wird auf das DROPS an sich eingegangen, gefolgt von wichtigen Protokollen und für dieses Projekt bedeutenden Komponenten. Da die Speicherverwaltung die Grundlage dynamischen Nachladens ist, werden ferner bekannte Speicherverwaltungsmodelle betrachtet. Die Schnittstellen der etablierten C-Bibltiothek geben Anhaltspunkte für die Schnittstelle, die die Speicherverwaltung nach außen anbietet.

Zuerst jedoch einige Worte über den Betriebssystemkern, auf den DROPS aufbaut.

2.1 Der Mikrokern L4

L4 wurde von Jochen Liedtke an der GMD für Intel-CPUs entwickelt. Er ist ein Mikrokern der zweiten Generation und wurde nach dem Minimalitätsprinzip konstruiert. Der Kern selbst implementiert nur virtuelle Adressräume, Threads und Interprozesskommunikation (IPC). Ein Adressraum mit mindestens einem Thread ist eine Task. Die IPC ist die einzige Möglichkeit, mit der Tasks miteinandern kommunizieren können.

L4 stellt Mechanismen bereit, mit denen sämtliche andere benötigte Komponenten eines Betriebssystems flexibel auf Nutzerebene realisiert werden können. Für eine Speicherverwaltung ist vor allem der Umgang mit Seitenfehlern (Page Faults, PF) interessant. Für jeden Thread ist ein *Pager* festgelegt. Ein Pager ist ein Thread, der Seitenfehler für andere Threads auflöst. Greift ein Thread auf eine Adresse zu, die nicht zu seinem Adressraum gehört, wird ein Seitenfehler ausgelöst und der Thread wird unterbrochen. Der L4-Kern wandelt den Seitenfehler in eine Nachricht um und sendet sie an den zuständigen Pager. Jener löst den Seitenfehler auf, der unterbrochene Thread wird fortgesetzt. Die Kommunikation zwischen Kern und Pager erfolgt über das *Page-Fault-Protokoll*.

Für die Handhabung von Speicherseiten sind Flexpages vorgesehen. Flexpages sind ein durch den L4-Kern definierter Datentyp. Sie sind flexibel große Regionen in virtuellen Adressräumen. Die minimale Größe einer Flexpage ist die Größe einer Hardware-Speicherseite[1].

L4 unterstützt das Konstruieren von Adressräumen durch drei Operationen:

- *Grant:* Der Eigner eines Adressraumes kann jede seiner eigenen Speicherseiten in einen anderen Adressraum granten, vorausgesetzt der Besitzer des anderen

[1] Bei x86-Prozessoren sind dies 4096 Byte.

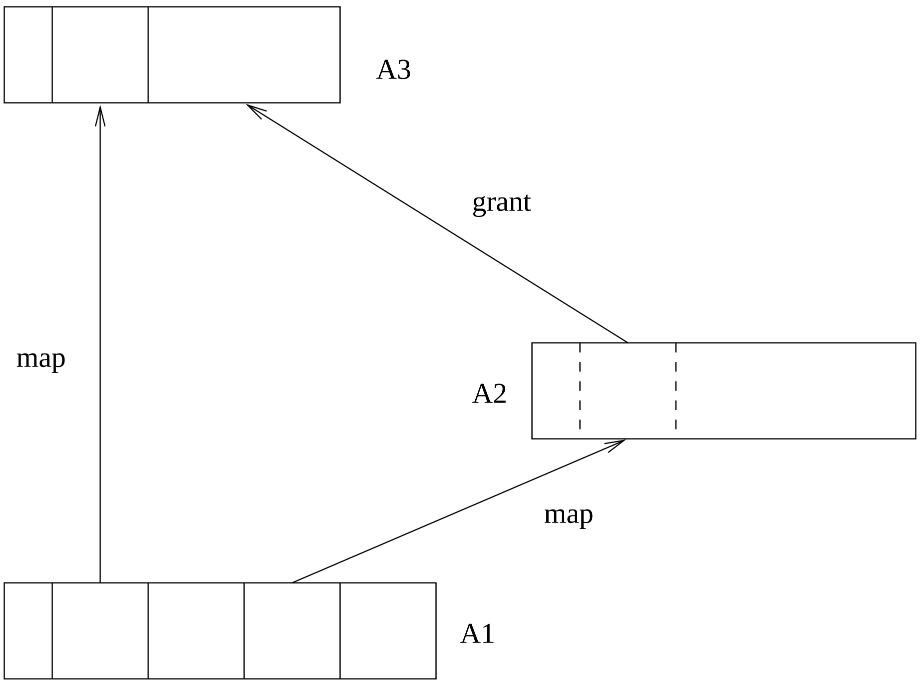

Abbildung 2.1: grant- und map-Operationen des L4-Kerns

Adressraumes stimmt zu. Granten bedeutet, dass die entsprechende Speicherseite aus dem Adressraum des aktuellen Besitzers entfernt und zum anderen Adressraum hinzugefügt wird.

- *Map:* Der Eigner eines Adressraumes kann jede seiner eigenen Speicherseiten in einen anderen Adressraum mappen, vorausgesetzt der Besitzer des anderen Adressraumes stimmt zu. Mappen bedeutet, dass diese Seite zum anderen Adressraum hinzugefügt wird, aber im aktuellen Adressraum verbleibt. Nun haben beide Tasks Zugriff darauf.

- *Flush:* Der Eigner eines Adressraumes kann jede seiner eigenen Speicherseiten flushen. Flushen bedeutet, dass die Seite in diesem Adressraum verbleibt, jedoch aus allen anderen Adressräumen entfernt wird, in die diese Speicherseite direkt oder indirekt gegrantet oder gemappt wurde.

Die Abbildung 2.1 verdeutlicht die grant- und map-Operationen. Eine Flexpage wurde von A1 nach A3 gemappt. Eine andere wurde von A1 nach A2 gemappt und nach A3 gegrantet. Da sie nicht mehr zu A2 gehört, ist sie nur gestrichelt dargestellt.

Eine ausführliche Beschreibung des Kerns ist in [4] zu finden.

2.2 Das Dresden Realtime Operating System Project

Verteilte Multimediaanwendungen stellen neue Anforderungen an Computer und Netzwerke. Eine bedeutende dieser Anforderungen ist, dass Echtzeit- und Nichtechtzeit - Komponenten gemeinsam auf einem Rechner laufen.

Das Dresden Realtime OPerating System Project befasst sich mit der Kooperation von Echtzeit- mit Nichtechtzeit - Komponenten. Der verwendete Betriebssystemkern ist der oben erwähnte L4-Mikrokern. Die Nichtechtzeit - Komponente,

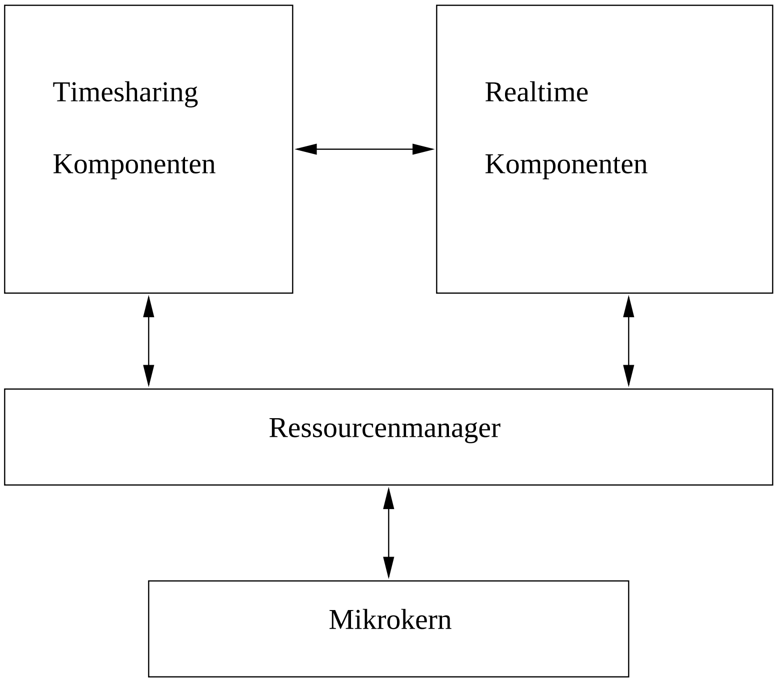

Abbildung 2.2: Architektur von DROPS

auch Timesharing - Komponente genannt, ist ein Linux, welches auf L4 portiert wurde ([9]). Damit steht eine komfortable Entwicklungsumgebung bereit, welche binärkompatibel zum verbreiteten Linux ist. Weiterhin wurden mehrere Echtzeit-Komponenten realisiert, so zum Beispiel ein SCSI-Treiber, ein Echzeitdateisystem, ein ATM-Treiber und ein ATM-basierter Echtzeit-Protokollserver. DROPS - Applikationen können mit (Timesharing - Applikationen) oder ohne (z.B. Echtzeit - Applikationen) das L4-Linux laufen.

Die Ressourcen, die die Hardware bietet, müssen verwaltet und an die anfragenden Komponenten und Applikationen verteilt werden. Hierfür existiert ein Ressourcenmanager.

Die Grafik 2.2 veranschaulicht die Struktur von DROPS.

Einige der für diese Arbeit wichtigen Komponenten werden in den folgenden Unterkapiteln vorgestellt.

2.2.1 Der Ressourcenmanager

Auf magische Art und Weise entsteht beim Booten der initiale Adressraum σ_0. Er ist zu Beginn der einzige Adressraum im System. Beim Booten des Systems fordert der Ressourcenmanager (rmgr) sämtliche für Anwendungen verfügbare Seiten von σ_0 an. Der rmgr kann jetzt als Speicherverwalter für den noch freien Speicher operieren, der fast der gesamte Hauptspeicher ist. Er erweitert die Fähigkeiten von σ_0, z.B. um Speicherquoten für einzelne Adressräume.

Mit den map-, grant- und flush-Operationen des L4-Kerns können Speicherseiten zu neuen Adressräume hinzugefügt werden.

Über ein spezielles Protokoll, das σ_0-Protokoll, kann eine Task Speicher beim rmgr anfordern. Es werden immer ganze Speicherseiten gehandhabt. Der Ressourcenmanager nutzt die map- und grant-Operationen des Mikrokerns, um die Speicherseiten zum Adressraum der anfordernden Task hinzuzufügen. Damit ist es einfach Adressräume zu schaffen, die von anderen Speicherverwaltern gepflegt werden. Zur

Steuerung des Speicherverbrauches können Quoten angegeben werden. Ein Adressraum erhält dann nur soviel Speicher, wie es seine Quota erlaubt.

Mit Hilfe des *Supervisor-Protokolls* können Applikationen den Ressourcenmanager veranlassen, eine Task zu starten, zu beenden oder Prioritäten zu setzen.

2.2.2 Die Pufferverwaltung

Die Pufferverwaltung hat die Verantwortung über ein eigenes Stück Speicher, der als Puffer bezeichnet wird. Sie kann Pufferbereiche vergeben, zurücknehmen, teilen, suchen und den Inhalt ausgeben.

Ein Puffer ist ein Zwischenspeicher für Datenströme. Da diese in DROPS durch das Modell schwankungsbeschränkter Ströme[2] beschrieben werden, sind Puffer sehr wichtig, um die Schwankungen auszugleichen. Durch die begrenzten Möglichkeiten der Speicherbeschaffung sind auch der Flexibilität der Pufferverwaltung Grenzen gesetzt.

Durch die Anwendung von DMA-Operationen wird an Puffer die Anforderung gestellt, dass deren physischer Speicher kontinuierlich sein muss. Der rmgr bietet aber keine Möglichkeit, diese Forderung durch einen Dienst zu erfüllen. So muss die zu implementierende Speicherverwaltung entsprechende Funktionalität bereitstellen.

2.3 Speicherverwaltung auf niedriger Ebene

Auf der untersten Ebene sind die Speicherseiten die kleinsten Einheiten, die sinnvoll zu verwalten sind. Die map- und grant-Operationen, die Adressräume beeinflussen, bauen auf Flexpages auf (siehe 2.1.), und die kleinste Größe einer Flexpage ist eine Speicherseite. Eine Seite kann entweder frei oder benutzt sein.

In der Literatur (z.B. [1]) werden reichlich Ansätze zur Freispeicherverwaltung beschrieben. Die oft beschriebenen Methoden benutzen Bitmaps, verkettete Listen oder das Buddy-System. Diese betrachten wir diese hier genauer:

- *Bitmaps:* Bei der Verwendung von Bitmaps wird der zu verwaltende Speicher in Allokationseinheiten unterteilt. Diese Allokationseinheit ist hier aus oben genannten Gründen eine Speicherseite. Jede Allokationseinheit korresponiert mit einem Bit der Bitmap. Ist das Bit 0, ist die Einheit frei, ist das Bit 1, ist sie belegt.

- *Verkettete Listen:* Benutzte Bereiche und Löcher zwischen diesen Bereichen werden in verketteten Listen verwaltet. Bei einer Speicheranforderung wird die Liste der Löcher durchsucht, bis hinreichend viel Speicher gefunden wurde. Wird Speicher freigegeben, so wird dieser aus der Liste der benutzten Bereiche ausgekettet und in die Liste der Löcher eingefügt. Falls möglich, werden benachbarte Löcher verschmolzen.

- *Das Buddy-System:* Dieses System basiert auf verschieden großen Speichersegmenten. Eine Speicheranforderung wird auf die Größe der nächsten Zweierpotenz aufgerundet, und es wird nach einem Segment dieser Größe gesucht. Wird kein passendes Segment gefunden, so wird nach einem doppelt so großen Speichersegment gesucht, um dieses in zwei Teile zu zerbrechen, usw. Auf diese Weise entstehen mit der Zeit eine Reihe verschieden großer Segmente, die aber alle die Größe einer Zweierpotenz haben. Diese besondere Eigenschaft bezüglich der Größe ermöglicht Performancevorteile beim Suchen und

[2] Schwankungsbeschränkte Ströme werden in [10] ausführlich behandelt.

Verschmelzen freier Bereiche, da solche Algorithmen sehr effizient arbeiten können.

2.4 Ein Framework für verschiedenartigen virtuellen Speicher

Neben der Low-Level-Speicherverwaltung ist auch auf höherer Ebene eine Organisation des Speichers vonnöten.

Am IBM T.J. Watson Research Center wurde ein Framework entwickelt, welches die Entwicklung flexibler Virtual Memory - Services ermöglicht. Hier seien die Fakten genannt, die für diese Arbeit interessant sind. Ausführlichere Informationen sind in [5] zu finden.

Das Framework basiert auf dem Lava-Mikrokern, der kompatibel zu L4 ist, folglich also auch die map-, grant- und flush-Operationen anbietet. Aus diesem Grund empfiehlt es sich, dieses Framework etwas näher zu betrachten.

Ein Kerngedanke ist die Einführung von “Dataspaces”. Ein Dataspace[3], zu deutsch Datenraum, ist eine zusammengehörige Menge von Speicherseiten, die zu einem kontinuierlichen Stück virtuellen Speicher zusammengefasst werden. Datenräume werden von sogenannten “Dataspace-Managern”, also Speicherverwaltungen, gepflegt. Die Eigenschaften des Speichers, der durch den Datenraum beschrieben wird, werden durch den Speicherverwalter festgelegt. Speicher wird also dadurch verschiedenartig, dass er von verschiedenen Speicherverwaltern angeboten wird. Ein Speicherverwalter könnte z.B. nicht auslagerbaren Speicher anbieten, ein anderer Nur-Lese-Speicher, ein dritter Speicher, der mit bestimmten Daten einer Datei gefüllt ist usw.

Um Datenräume benutzen zu können, muss es Operationen über diese geben. Im Framework sind neun vorgesehen: Identify, Attach/Open, Detach/Close, Interrogate, Share, Copy, Transfer, Create und Delete. Während die ersten vier unterstützt werden müssen, sind die restlichen fünf optional.

Damit Applikationen auf die Daten in den Datenräumen zugreifen können, werden die Datenräume in Regionen im virtuellen Adressraum der Applikation eingeblendet. Pager erfüllen diese Aufgabe mit Hilfe der map- und grant-Operationen.

Flexibilität und Erweiterbarkeit wird dadurch erreicht, dass die Speicherverwaltungen stapelbar sind und die Datenräume von einer Speicherverwaltung zu einer anderen transferiert werden können. So ist es leicht möglich, die Semantik zu variieren oder neue Sematiken in das System einzuführen.

2.5 Das OSKit

Das Flux Operating System Toolkit (OSKit) wurde und wird von der Flux Research Group der Universität Utah entwickelt. Es soll die Entwicklung von Betriebssystemkernen und -komponenten erleichtern. Dafür bietet es modularisierte Komponente und Bibliotheken an.

Genauer betrachten wollen wir hier

- *Die Speicherverwaltung:* Sie stützt sich auf einen “List-based Memory Manager” (LMM). Die Funktion des OSKit zur Speicherallokation (malloc) greift direkt auf den LMM (lmm_alloc) zurück, der wiederum die Listen zur Speicherverwaltung direkt ändert. Zur Unterstützung des OSKit-malloc bietet sich z.B. an, lmm_alloc so abzuändern, dass nicht auf die Listen, sondern auf einen von der Speicherverwaltung zur Verfügung gestellten Pager zugegriffen wird.

[3] Der Begriff “Dataspace” wurde von L3-Entwicklern geprägt.

- *Der Executable Program Interpreter (EPI)* Er erleichtert das Laden ausführbarer Programme, ohne Einschränkungen aufzuerlegen. Das eigentliche Laden und Speichern auf physische Seiten erledigt der EPI nicht selbst, sondern ruft dazu entsprechende Callback-Funktionen auf, die der "Client" bereitstellen muss. Zur Zeit wird das a.out- und das Executable and Linkable Format (ELF) unterstützt.

2.6 Verbreitete Speicherverwaltungsfunktionen

Etablierte (C-)Bibliotheken und das OSKit bieten verbreitete und bekannte Schnittstellen zum Speichermanagement für Applikationen. Diese sind:

- *malloc* erwartet als Parameter die Grösse in Byte und gibt einen Zeiger vom Typ void auf den Beginn des freien Speichers zurück. Eine entsprechende Funktion ist in der Speicherverwaltung vorzusehen.
- *free* erwartet einen Zeiger, der von einem malloc zurückgegeben wurde, und gibt diesen Speicherbereich frei. Auch diese Funktion ist unentbehrlich.
- *mmap* erwartet als Parameter den Beginn, die Länge, Schutzbits, Typflags, einen Filedeskriptor und einen Offset als Parameter und gibt einen Zeiger auf den gemappten Bereich zurück. Das mmap eignet sich vor allem zum Einblenden von Dateien in den Adressraum, jedoch kann damit auch anonymer Speicher mit besonderen Eigenschaften gemappt werden.

  Mit mmap können schon mehr Anforderungen an den Speicher definiert werden, jedoch ist diese Funktion zu schwach für das von anderen DROPS-Komponenten Geforderte. Daher wird ein Dienst das Ziel sein, der zwar ähnlich dem mmap, jedoch allgemeiner und besser auf die Gegebenheiten von DROPS abgestimmt ist.

2.7 Resultierende Vorgaben

Aus der beschriebenen Umgebung ergeben sich einige Vorgaben. Die Verwendung des Mikrokerns L4 oder einer kompatiblen Version legt die Verwendung virtueller Adressräumen nahe[4] und impliziert die Verwendung von Flexpages.

Auf den Ressourcenmanager aufsetzend wird es einen Speicherverwalter geben, der vom rmgr eine Menge von Speicherseiten anfordert und verwaltet. Verstreut liegende, aber zusammengehörende Speicherseiten werden mit Hilfe von Datenräumen zusammengefasst.

Die Auflösung von Seitenfehlern erfolgt auf Nutzerebene durch Pager, die Speicherverwaltung sollte also auch Pagerfunktionalität erfüllen.

Die Schnittstelle, die den Anwendungen angeboten wird, sollte ähnlich oder gleich der etablierter (C-)Bibliotheken sein, um den Einarbeitungsaufwand für die Anwender dieser Arbeit zu minimieren.

[4] Möglich wäre auch, einen einzigen Adressraum zu verwenden, wie man in [11] nachlesen kann.

Kapitel 3

Entwurf

In diesem Kapitel werden drei Szenarien vorgestellt, aus denen Anforderungen abgeleitet werden. Von der hohen Ebene der Applikation beginnend, werden einzelne Bestandteile entworfen, immer weiter absteigend, bis die tiefen Schichten erreicht wurden. Am Ende werden die Entwurfsentscheidungen zusammengefasst.

Um ein dynamisches Nachladen von Komponenten zu ermöglichen, müssen drei wichtige Funktionseinheiten implementiert werden:

- Eine Speicherverwaltung, die das Allokieren und Freigeben von Speicher ermöglicht. Da die Speicherverwaltung in einem Betriebssystem eine zentrale Rolle einnimmt, werden wir beim Entwurf ausführlich darauf eingehen.

  Den Anwendungen soll der gesamte adressierbare Speicher als virtueller Adressraum zur Verfügung stehen.

  Da im Laufe der Entwicklung neue, noch nicht absehbare, Anforderungen auftreten könnten, ist genügend Raum für Erweiterungen vorzusehen.

- Ein Pager, der die Seitenfehler behandelt.

- Ein Loader, der in der Lage ist, Standard-ELF-Binaries zu laden und zu starten.

Weiterhin soll eine Minishell entworfen werden, die Interaktionen mit dem System erlauben soll. Dabei werden die Features auf das nötigste Minimum reduziert, da das Hauptaugenmerk auf den drei oben genannten Komponenten liegt.

3.1 Einige Szenarien

Zur Einführung seien einige Szenarien betrachtet, um ein Gefühl für die Problematik zu vermitteln.

1. *Ein einfaches Programm:* Es wird durch einen Nutzer mittels Eingabe des Programmnamens in eine Shell gestartet, beschäftigt sich mit irgendetwas einfachem und endet. Typisch dafür ist z.B. ein "Hallo Welt!"-Programm. Nur für den Programmcode, die Daten und den Stack muss Speicher bereitgestellt werden. Deren Größen sind zur Compilierzeit bekannt und werden in der Binärdatei codiert. Auftretende Seitenfehler müssen von einem Pager behandelt werden. Nachdem das Programm beendet wurde, muss der verwendete Speicher wieder zur Menge des freien Speichers hinzugefügt werden.

Um die Standardentwicklungswerkzeuge zur Programmerstellung nutzen zu können, sollten normale ELF-Binärdateien die zu startenden Programme enthalten. Der Loader, der das Programm lädt, muss also in der Lage sein, ELF-Dateien zu parsen und die zum Programmstart notwendigen Informationen zu extrahieren.

2. *Ein Programm mit Allokation von anonymem Speicher:* Wenn es nicht nur triviale Aufgaben wie im ersten Beispiel erfüllen soll, benötigt ein Programm zusätzlich oft Speicher, von dem zur Compilierzeit noch nicht feststeht, wie groß er sein muss oder ob er überhaupt benötigt wird. Beispiele sind das Laden von Dateien oder der Aufbau von verketteten Listen. Hierzu muss das Programm dynamisch anonymen Speicher beschaffen. In Standard-C-Bibliotheken gibt es dazu die Funktion `malloc()`.

   Wird der anonyme Speicher nicht mehr benötigt, muss ihn die Applikation wieder freigeben können.

3. *Allokation von anonymem Speicher mit besonderen Eigenschaften:* Hin und wieder stellt ein Programm bestimmte Anforderungen an den anonymen Speicher. Soll dieser etwa für den Einsatz als DMA-Puffer geeignet sein, muss er physisch zusammenhängen, Treiber müssen die physische Adresse in Erfahrung bringen können. Die unter L4 verwendeten Flexpages stellen die besondere Anforderung, dass ihre Ausrichtung im Speicher (Alignment) von ihrer Größe abhängt. Will eine Anwendung Speicher per IPC einer anderen Anwendung zugänglich machen und sendet ihr dazu eine Flexpage, die 2^n Byte groß ist, muss die Flexpage bei einer auf 2^n ausgerichteten Adresse beginnen.

4. *Echtzeitanwendungen:* Eine Besonderheit in DROPS ist, dass neben den Nichtechtzeitanwendungen auch Echtzeitanwendungen laufen. Da letztere an ein bestimmtes Zeitverhalten gebunden sind, dürfen sie nicht durch Seitenfehler unterbrochen werden. Dies gilt sowohl für den zum Programmstart (für Code, Stack und Daten) zugewiesenen als auch für den dynamisch allokierten Speicher. Wird einer Echtzeitanwendung Speicher zugewiesen, sollte er möglichst vor dem ersten Zugriff in den Adressraum eingeblendet werden. Einmal zugewiesener Speicher darf nicht entzogen werden (was u.a. ein Auslagern von echtzeitfähigem Speicher auf Festplatte ausschließt).

Bei den Beispielen wird auch deutlich, dass es im virtuellen Adressraum der Applikation Regionen mit unterschiedlichen Semantiken gibt - im einfachsten Fall Code, Daten und Stack, bei komplexeren Fällen Speicherbereiche, die dynamisch allokiert und von verschiedenen Speicherverwaltungen zur Verfügung gestellt werden. Somit ergibt sich die Notwendigkeit, die Regionen im virtuellen Adressraum der Anwendung zu verwalten.

Zur Schonung von Systemressourcen ist es sinnvoll, zwischen Echtzeit- und Nichtechtzeitanwendungen zu unterscheiden. Bei Nichtechtzeitanwendungen kann jede Seite erst bei einem Seitenfehler zum Adressraum hinzugefügt werden (mapping on demand). Seiten, auf die nie zugegriffen wird - sei es, dass bestimmte Teile des Programmcodes nie in einer Sitzung ausgeführt werden, sei es, dass hinreichend viel Speicher für den schlechtesten Fall reserviert wird, der aber in dieser Sitzung nicht eintritt - brauchen der Anwendung nicht zur Verfügung gestellt und können somit für andere Zwecke eingesetzt werden.

Zusammenfassend seien hier noch einmal die Anforderungen genannt, die an die Komponente gestellt werden, welche Programme dynamisch lädt:

- Speicher für den Programmcode und die Daten bereitstellen

- Dynamisch allokierbaren und freigebbaren Speicher zur Verfügung stellen; dieser Speicher soll gewissen Anforderungen genügen (siehe unten)
- Den virtuellen Adressraum verwalten, in dem die Programme laufen
- Seitenfehler sind aufzulösen (in DROPS durch einen User-Level-Pager)
- Eine Minishell für Ein- und Ausgaben
- ELF-Binärdateien laden, parsen und die Programme starten

Die besagten speziellen Anforderungen, die an Speicher gestellt werden können, sind:

- Physisch kontinuierlich
- Bekannte physische Adresse
- Nicht auslagerbar; allgemeiner: Speicherseiten der Anwendung nicht entziehbar
- Ausrichtung im Speicher (Alignment)
- Kein mapping-on-demand, also sind bei der Speicheranforderung die Seiten sofort dem Adressraum der Anwendung hinzuzufügen

Es können im Laufe der Zeit weitere, jetzt noch nicht absehbare Forderungen auftreten. Bei den Definitionen bezüglich der besonderen Eigenschaftes des Speichers ist genügend Raum für Erweiterungen vorzusehen.

3.2 Organisation des virtuellen Speichers

Ohne Nutzung der Segmentierung lässt sich mit den 32-Bit-Adressen des x86 ein virtueller Raum von 4GB adressieren. In diesem Adressraum müssen der Code, die Daten und der Stack untergebracht werden, außerdem muss genug Platz für anonymen Speicher bleiben. Es entstehen also verschiedene Bereiche. Diese Bereiche seien hier *Regionen* genannt. Eine Verwaltung wird über Regionen im virtuellen Adressraum wachen. Der *Regionenverwalter* führt eine Tabelle, in der er seine Verwaltungsinformationen einträgt.

Beim Laden eines Programmes entstehen die Regionen, die durch die Binärdatei festgelegt sind, zumindest also Code, Daten und Stack. In der Binärdatei können schon Vorgaben zu den virtuellen Adressen sein, müssen aber nicht. Insbesondere ist beim Stack oftmals nur wichtig, dass er vorhanden ist, seine Lage ist für den Programmierer und den Anwender uninteressant.

Anders die Situation bei anonymem Speicher. Er wird dynamisch während der Laufzeit des Programmes angefordert. Der Regionenverwalter entscheidet deshalb, in welcher Region der Speicher eingeordnet wird und übergibt dem Aufrufer diese virtuelle Adresse.

3.2.1 Datenräume

Damit eine Applikation auf Daten zugreifen kann, müssen die Regionen mit den Speicherseiten hinterlegt werden, in denen sich die Daten befinden. Die Speicherseiten, in denen zusammengehörige Daten liegen (z.B. ein Bild), werden zusammengefasst und *Datenräume* genannt. Wird ein Datenraum in eine Region eingeblendet, hat die Anwendung Zugriff auf die Daten. Die Idee der Datenräume wurde aus [5] aufgegriffen.

Ein Datenraum kann zugleich in verschiedene Adressräume eingeblendet werden. Dadurch haben mehrere Tasks die Möglichkeit, die Daten zu nutzen - sie teilen sich quasi den Speicher. Auf diese Weise kann Shared Memory implementiert werden.

3.2.2 Anordnung des Regionenverwalters

Ein Pager hat die Aufgabe, Seitenfehler aufzulösen und eine entsprechende Speicherseite in den betroffenen Adressraum zu granten oder mappen. Da der verantwortliche Pager für eine virtuelle Speicheradresse (bzw. den diese Adresse betreffenden Seitenfehler) von der Region abhängt, in der diese Adresse liegt, muss der Regionenverwalter aus der Seitenfehleradresse erst den Pager bestimmen. Für jeden L4-Thread muss ein Pager angegeben werden. Es wird der Regionenverwalter als offizieller Pager eingesetzt. Der L4-Kern sendet ihm die Seitenfehlernachricht. Er kann anhand seiner Regionentabelle den passenden Speicherverwalter bestimmen, der der eigentliche Pager ist und den Seitenfehler auflöst.

Der Regionenverwalter kann im Adressraum der Applikation, einer Speicherverwaltung oder in einem extra Adressraum angeordnet sein. Drei mögliche Fälle werden im folgenden diskutiert:

- **Im extra Adressraum:** Nach dem Auftreten des Seitenfehlers bestimmt der Regionenverwalter den verantwortlichen Speicherverwalter und lässt sich von ihm eine Seite mappen, die er in den Adressraum des unterbrochenen Threads grantet. Schlecht ist, dass immer eine map- und eine grant-Operation erfolgen muss.

- **Im Adressraum einer Speicherverwaltung:** Das Vorgehen ist ähnlich wie oben, jedoch kann eine Optimierung für den Fall vorgenommen werden, wenn die Speicherverwaltung schon im Besitz der betreffenden Speicherseite ist. Hier ist nur eine map-Operation erforderlich.

  Da vermutlich Applikationen eine Speicherverwaltung haben werden, von der sie hauptsächlich ihren Speicher beziehen und nur in seltneren Fällen Speicher von anderen Speicherverwaltern Speicher anfordern, kann davon ausgegangen werden, dass die Speicherverwaltung häufig schon im Besitz der entsprechenden Speicherseite ist und die map/grant-Paar der ersten Variante eher selten sind.

- **Im Adressraum der Applikation:** Ein Thread im Adressraum der Applikation - der Regionenverwalter - ist offiziell Pager für die anderen Threads der Anwendung, die dann als Nutzerthreads bezeichnet werden. Der Regionenverwalter bestimmt bei einem Seitenfehler die für den Datenraum der betreffenden Region verantwortliche Speicherverwaltung und leitet die Nachricht an sie weiter. Die Speicherverwaltung bestimmt die passende Seite und mappt sie in den Adressraum des Regionenverwalters. Dieser ist gleichzeitig der Adressraum der Nutzerthreads. Der unterbrochene Nutzerthread kann sodann fortfahren. Hier ist nur eine map-Operation nötig, aber gegenüber dem optimierten Fall der zweiten Variante zwei weitere Short-IPCs.

Weil Short-IPC unter L4 extrem schnell ist, fiel die Wahl auf die dritte Variante. Diese impliziert folgenden Ablauf bei einem Seitenfehler (Bild 3.1):

1. Ein Thread einer Anwendung verursacht einen Seitenfehler.

2. Der L4-Kern wandelt den Seitenfehler in eine Nachricht um und sendet sie dem Pager, der für diesen Thread angegeben wurde. Im Falle eines Nutzerthreads ist dies der Pagerthread im Adressraum der Applikation.

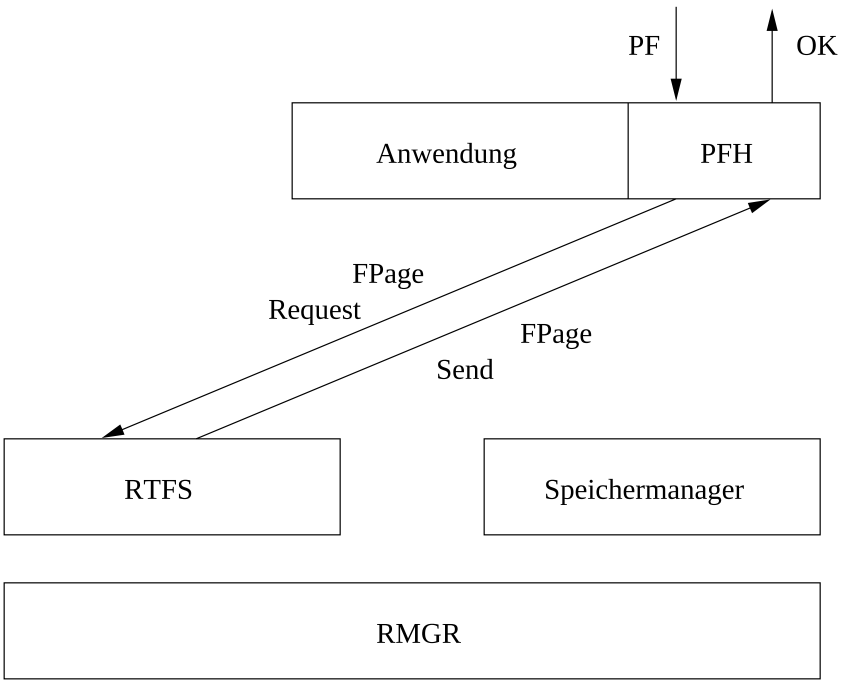

Abbildung 3.1: Die Behandlung eines Page Faults ist dargestellt.

3. Der Regionenverwalter sucht in einer Tabelle die zur Page-Fault-Adresse passende Region und ermittelt die Speicherverwaltung, die für den Datenraum in dieser Region verantwortlich ist.

4. Wird keine Region gefunden → "segmentation fault" (Abbruch der Task mit einer entsprechenden Fehlermeldung)

5. Der Pager sendet eine Nachricht an die Speicherverwaltung mit einer eindeutigen Kennung des Datenraumes sowie dem Offset im Datenraum, der sich aus der Seitenfehleradresse ergibt.

6. Die Speicherverwaltung prüft ggf. Attribute; Fehler → "segmentation fault"

7. Die Speicherverwaltung ermittelt die passende Seite und blendet sie in den Adressraum des unterbrochenen Threads ein. Dies geschieht mit einer IPC an den Pager.

8. Der Pager beantwortet den L4 - Page Fault mit einer kurzen Nachricht (Short-IPC des L4)

3.3 Das Tasklayout

Die letzten Betrachtungen auf der hohen Ebene der Sicht des Anwenders gelten dem Tasklayout. Es wird besprochen, was alles letztendlich zu einer Anwendertask gehört.

Einmal ist die Threadstruktur zu betrachten:

- Wir haben uns im vorigen Abschnitt für die Unterbringung des Pagers im Adressraum der Applikation entschieden. Daher agiert ein Thread als Pager.

- Die restlichen Threads stehen dem Nutzer zur freien Verfügung.

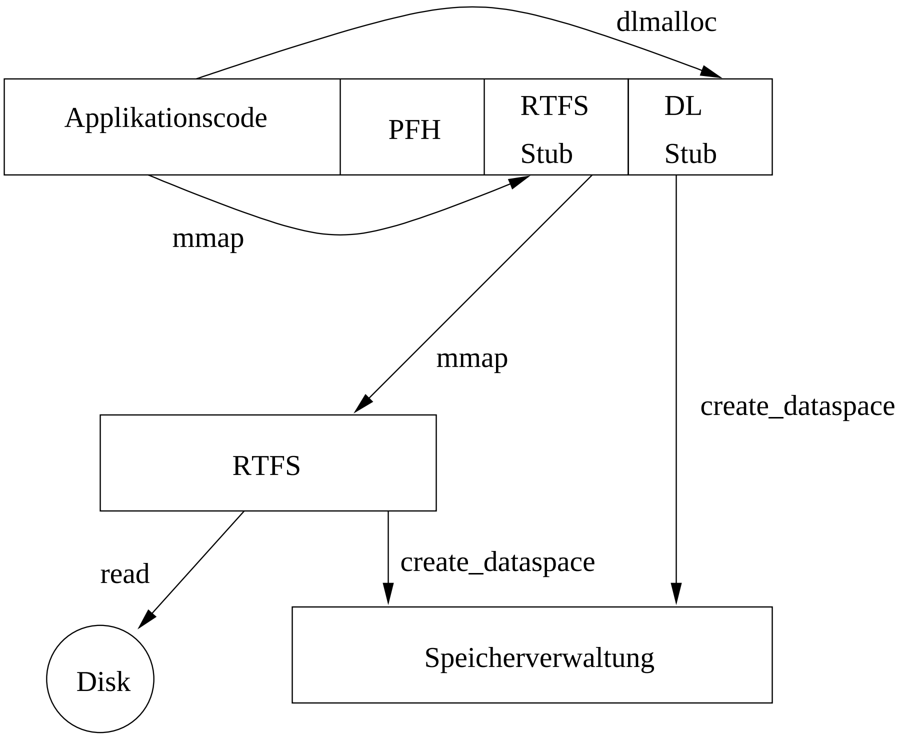

Abbildung 3.2: Beispiel einer Applikation mit RTFS und Speicherverwaltung

Zur Vereinfachung der Programmierung ist es erwünscht, dass die Kommunikation mit den tieferen Schichten der Speicherverwaltung in einem Stub gekapselt wird. Der Stub stellt dem Anwendungsprogrammierer bekannte, verbreitete Schnittstellen zur Verfügung. Pro genutzter Speicherverwaltung gibt es also einen Stub, der in die Applikation eingebunden wird.

Somit können wir die Codekomponenten zusammentragen, die zu einer Task gehören:

- Der Nutzercode: Er erfüllt die eigentliche Funktionalität der Applikation
- Die Speicherverwaltungsstubs: Sie bilden die Schnittstelle zu den Speicherverwaltungen
- Der Regionenverwalter: Er ordnet den virtuellen Adressraum der Task
- Der Pager: Er löst Seitenfehler der Nutzerthreads auf

In der Abbildung 3.2 ist dies grob dargestellt. Der Übersicht wegen wurde der Regionenverwalter weggelassen. Eine *DL Stub* genannte Codekomponente gehört zur Speicherverwaltung und bietet in seiner Schnittstelle die Funktion `dlmalloc()` an, mit der anonymer Speicher allokiert werden kann. Der *RTFS Stub* bietet ein `mmap()` an. Beide Stubs kommunizieren mit den tieferen Schichten, um den Speicher zu reservieren.

3.4 Speicherobjekte

3.4.1 Einführung von Speicherobjekten

In Kapitel 2 wurden Datenräume eingeführt, und zu Beginn dieses Kapitels wurde gezeigt, wie diese Datenräume in Regionen des virtuellen Adressraumes von Appli-

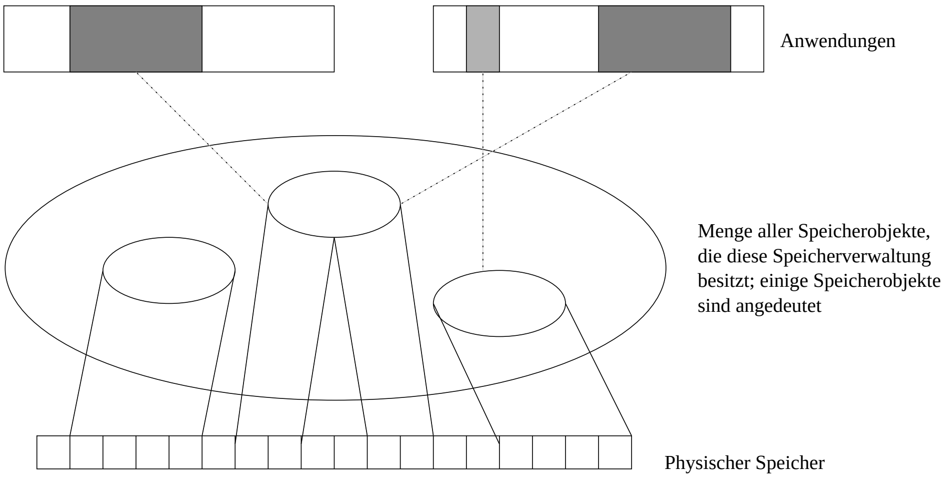

Abbildung 3.3: Physischer Speicher, Speicherobjekte, Speicherverwaltung und virtuelle Adressräume

kationen eingeblendet werden kann, um den Zugriff auf die Daten zu ermöglichen. Das 3. der einführenden Szenarien hat außerdem verdeutlicht, dass nicht nur die Datenräume an sich, sondern auch einige Zusatzinformationen verwaltet werden müssen. In diesem Entwurf bietet ein und derselbe Speicherverwalter Speicher mit mehreren Semantiken an. So müssen zu diesem Speicher auch die Semantik betreffende Informationen gespeichert werden. Zu einem Datenraum muss z.B. vermerkt werden, ob dessen Speicherseiten auslagerbar sind oder nicht.

Zum geeigneten Zusammenfassen und Pflegen dieser Informationen führen wir *Speicherobjekte* ein. Einem Speicherobjekt ist ein Datenraum zugeordnet, desweiteren Attribute, die (schon mehrfach genannte) Eigenschaften dieses Speicherobjektes beschreiben. Jedes Speicherobjekt besitzt eine eindeutige Identifikationsnummer (ID).

Im Bild 3.3 ist auf unterster Ebene der physische Speicher dargestellt, unterteilt in Speicherseiten. Darüber befindet sich die Menge der Speicherobjekte, die eine Speicherverwaltung besitzt. Die einzelnen Speicherobjekte sind im Inneren oval dargestellt. Den Speicherobjekten können physisch kontinuierliche, aber auch physisch nicht-kontinuierliche Speicherbereiche zugeordnet sein. Das Übersetzen der Ansammlung von verstreut liegenden physischen Seiten in einen kontinuierlichen virtuellen Bereich erfolgt durch eine Seitentabelle (im Bild nicht dargestellt). Das mittlere Speicherobjekt wurde, durch die gepunkteten Linien dargestellt, zwei Adressräumen zugeordnet. Sie haben jetzt beide Zugriff auf den damit beschriebenen Speicher. Das rechte Speicherobjekt wurde nur mit einem Adressraum assoziiert.

3.4.2 Besitzer und Besitzerwechsel von Speicherobjekten

Ein Speicherobjekt wird durch einen bestimmten Speicherverwalter angelegt und von ihm gepflegt. Es gehört also dem Speicherverwalter, dieser ist "Besitzer", "Eigner" oder "Eigentümer" des Speicherobjektes. Es kann der Fall eintreten, dass das Speicherobjekt den Eigentümer wechseln muss, weil die Applikation vom Speicher über kurze oder lange Zeit eine andere Semantik erwartet, die durch einen anderen Speichermanager realisiert wird[1]. Das Vorgehen, die Semantik eines Datenraumes

[1] Über diesen Mechanismus ist auch ein Migrieren auf andere Rechnerknoten sowie die Durchführung von Wartungsarbeiten im laufenden Betrieb möglich. Eine fehlerbereinigte Version

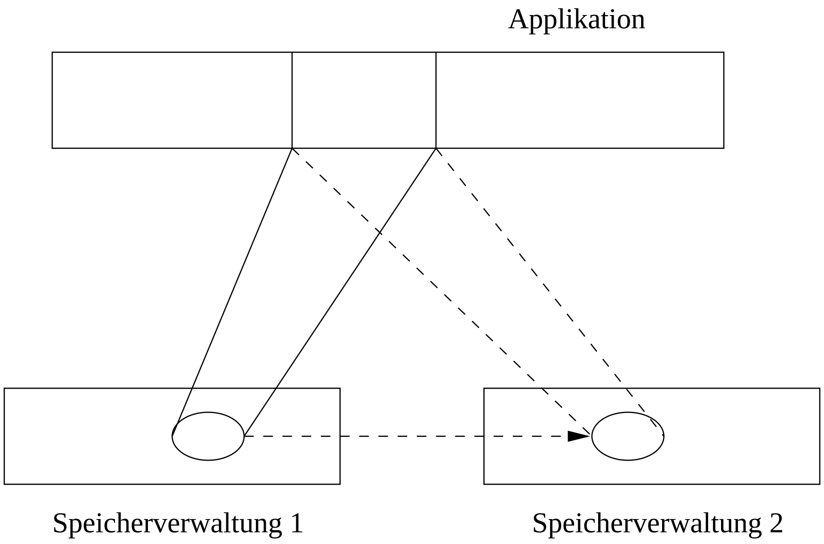

Abbildung 3.4: Besitzerwechsel

von dem Speicherverwalter abhängig zu machen, dem der Datenraum gehört, wurde in [5] diskutiert und in 2.4 als eine der Grundlagen dieser Arbeit vorgestellt.

Im Bild 3.4 besitzt die Speicherverwaltung 1 ein Speicherobjekt, dessen Speicher in den Adressraum der Applikation eingeblendet ist (durch die durchgezogenen Linien angedeutet). Nach dem Transfer (gestrichelter Pfeil) gehört das Speicherobjekt der Speicherverwaltung 2, die neue Zuordnung zwischen Speicherobjekt und Region im virtuellen Adressraum ist durch die gestrichelten Linien dargestellt.

Für den Wechsel existiert der Dienst "Transfer" des Protokolls für Grundoperationen über Speicherobjekte. Ein Nachteil dieses Vorgehens ist, dass der physische Speicher fragmentiert wird, da der Speicherverwaltung Seiten entzogen werden.

Ein anderer Fall ist das Stapeln von Speicherverwaltern. In Bild 3.2 ist eine Applikation dargestellt, die das RTFS benutzt. Das RTFS als zusätzliche Schicht erweitert die Semantik des anonymen Speichers, indem es ihn mit Teilen von Dateien füllt. Vom Standpunkt der Applikation aus betrachtet ist das Einfügen einer Schicht sehr ähnlich dem Wechsel des Eigners. Die Applikation wendet sich nach dem Einfügen an einen anderen Speicherverwalter (das RTFS). Das Speicherobjekt verbleibt jedoch - für die Anwendung transparent - bei der ersten Speicherverwaltung. Die zweite Schicht braucht keinen vollen Verwaltungsapparat für Speicherobjekte[2]. Die Applikation wird ihre Seitenfehler an die Schicht unmittelbar unter ihr weiterleiten, im Beispiel das RTFS. Diese entscheidet, ob sie die Behandlung selbst vornehmen kann oder sich ihrerseits an die nächsttiefere Schicht wenden muss.

3.5 Protokolle

Zur Kommunikation mit der Speicherverwaltung wurden Protokolle definiert. Da davon ausgegangen wird, dass im Laufe der Zeit weitere Komponenten entstehen, die Speicherverwaltungsfunktionalität erfüllen können, wurden die Funktionen auf mehrere Protokolle verteilt. Einem Speicherverwalter steht es frei zu entscheiden, welche Protokolle er implementiert und welche Funktionen aus den Protokollen er

eines Speicherverwalters übernimmt beispielsweise alle Speicherobjekte seines alten, fehlerhaften Pendants.

[2] Es steht ihr natürlich frei, trotzdem einen zu implementieren.

unterstützt.

Bei der Definition der Protokolle wurde entschieden, ein Doppelwort (=32 Bit) zur Auswahl eines Protokolles aus der Protokollfamilie und einer oder mehrerer Funktionen aus diesem Protokoll zu verwenden. Die 32 Bit wurden in 8 und 24 Bit unterteilt. Die 8 Bit codieren 256 Protokolle, während die 24 Bit mit dem 1-aus-n-Verfahren 24 Funktionen codieren. Somit können mit einem Aufruf mehrere Funktionen parallel angefordert werden. Die sich aus diesen Festlegungen ergebenden Beschränkungen, dass ein Thread eines Servers nur zwischen 256 Protokollen unterscheiden kann und jedes dieser Protokolle nur 24 Funktionen haben kann, erscheinen tragbar.

3.5.1 Protokoll für Grundoperationen über Speicherobjekte

In diesem Protokoll sind die Grundoperationen festgelegt, die auch andere Speicherverwaltungen - in dieser Form oder Teilmengen davon - implementieren sollten. Dabei sind Identify, Attach, Detach und Interrogate Funktionen, deren Unterstützung durch die Speicherverwaltung sehr angeraten wird.

Diese Speicherverwaltung stellt folgende Grundoperationen über Speicherobjekte bereit, die an die in 2.4 vorgeschlagenen angelehnt sind:

- Identify: Frage nach der ID eines bestimmten Speicherobjektes
- Attach: Assoziiert ein schon vorhandenes Speicherobjekt mit einem Adressraum; Parameter ist die ID des Speicherobjektes; Verwendung vor allem beim Umgang mit Shared Memory
- Detach: Löscht die Assoziation zwischen dem Adressraum und dem Speicherobjekt; das Speicherobjekt bleibt erhalten; Parameter ist die ID des Speicherobjektes;
- Interrogate: Die Speicherverwaltung gibt bekannt, welche Operationen sie unterstützt
- Share: Weist die Speicherverwaltung an, das gegebene Speicherobjekt als "shareable" zu kennzeichnen; ab diesem Zeitpunkt darf es auch mit anderen Adressräumen assoziiert werden; Parameter ist die ID des Speicherobjektes
- Create: Erzeugt ein neues Speicherobjekt; Parameter sind Größe und Flags, die den gewünschten Speicher beschreiben; kein Adressraum wird beeinflusst; Verwendung vor allem beim Allokieren mit Sonderwünschen;
- Delete: Ein Speicherobjekt wird gelöscht; es wird aus allen Adressräumen entfernt; Parameter ist die ID des Speicherobjektes
- Transfer: Das Speicherobjekt wechselt den Besitzer; Parameter sind die ID des Speicherobjektes und der neue Speicherverwalter; die Pflege des Speicherobjekts geht vollständig auf die neue Speicherverwaltung über, das Speicherobjekt wird aus dem alten Verwalter entfernt

Das Allokieren eines Speicherbereiches besteht also intern aus Create und Attach. Sehr häufig ist es jedoch der Fall, dass die Größe der einzige Parameter ist - wenn ein Stück anonymer Speicher ohne besondere Anforderungen allokiert werden soll. Daher wurde eine Optimierung vorgenommen:

- Open: Assoziiert ein Speicherobjekt mit einem Adressraum; das Speicherobjekt wird erzeugt; Parameter ist die gewünschte Größe in Byte; Verwendung vor allem beim Allokieren

3.5.2 Protokollaufsätze für Erweiterungen

Neben den Grundoperationen über Speicherobjekte gibt es eine Reihe weiterer Dienste, die die Speicherverwaltung anbietet. Diese Dienste wurden in drei Kategorien gegliedert, und es wurden drei Erweiterungen für das obige Protokoll definiert:

- Information: Informationen über ein Speicherobjekt abfragen, z.B. die Adresse im physischen Speicher oder Attribute; da der Freispeicher aus logischer Sicht auch ein Speicherobjekt ist, sind in dieser Protokollerweiterung auch Funktionen zur Abfrage des freien und benutzten sowie des gesamten verfügbaren Speichers enthalten

- Task New: Durch die Trennung in Speicherverwaltung, Loader und Applikationsstub ist eine Kommunikation untereinander nötig. Sie wird durch diese Protokollerweiterung definiert.

- Memory Object: Die hier entworfene Speicherverwaltung stellt Speicher mit unterschiedlichen Semantiken bereit. Aus diesem Grund müssen die besonderen Eigenschaften als Attribute im Speicherobjekt vermerkt werden. Dieser Protokollaufsatz definiert die Art der Attribute und deren mögliche Werte.

3.6 Freispeicherverwaltung

Im Kapitel 2 wurden verschiedene Varianten von Freispeicherverwaltungen vorgestellt. Die Wahl fiel auf eine Abwandlung einer Bitmap, der Bytemap. Einer Speicherseite wird darin nicht nur ein Bit, sondern ein ganzes Byte zugeordnet. Neben der einfachereren, schnelleren Suche nach einer freien Seite bietet diese Variante die Möglichkeit, in den 7 übrigen Bit statistische Informationen zu Speichern, die beim Auslagern von Speicherseiten auf die Festplatte bedeutend sind. Diese Speicherverwaltung ist zwar noch nicht in der Lage, Seiten auf einen Hintergrundspeicher auszulagern (zu swappen). Das Swapping ist jedoch ein wichtiger Mechanismus moderner Betriebssysteme, so dass anzunehmen ist, dass es auch in DROPS integriert werden wird. Algorithmen zur Auswahl von auszulagernden Seiten bedienen sich oft Methoden, die auf Zugriffsstatistiken von Speicherseiten aufbauen ([1]). In der Bytemap werden die statistischen und die Verfügbarkeitsinformationen zusammengefasst.

3.7 Weitere Aspekte der Speicherverwaltung

3.7.1 Shared Memory

Für verschiedene Aufgaben ist Speicher sinnvoll, der in mehrere Adressräume eingeblendet ist, so dass verschiedene Tasks darauf Zugriff haben. Bei der Realisierung sind verschiedene Politiken denkbar:

- Am einfachsten dürfte öffentlicher Speicher sein, auf den jeder beliebigen Zugriff hat. Hier muss im Speicherobjekt nur vermerkt werden, ob dieser Speicher privat oder öffentlich ist. Shared Libraries sind ein Anwendungsfall hierfür.

- Etwas aufwendiger ist, einer Anzahl von ausgewählten Teilnehmern den Zugriff zu gewähren. Hier müsste eine Liste der Teilnehmer geführt werden.

- Sehr leistungsfähig, dafür aber auch aufwendig zu implementieren ist Speicher, der mit Rechten versehen ist. Basierend auf dem Teilnehmermodell kommen noch Rechte hinzu, wie etwa lesen, schreiben, weitere Teilnehmer hinzunehmen oder Teilnehmer entfernen.

- Sicherlich sind weitere sinnvolle Politiken denkbar.

Aufgrund der vielfältigen Varianten wird von einer Implementierung von Shared Memory in dieser Speicherverwaltung abgesehen. Vielmehr wird empfohlen, die Stapelbarkeit zu nutzen und einen auf diese Speicherverwaltung aufsetzenden Speicherverwalter zu implementieren, der die jeweils gewünschte Politik bereitstellt. Im Protokoll der Operationen über Speicherobjekte wurde aber eine Share - Operation schon vorgesehen, um den Mechanismus vorzugeben und Einheitlichkeit zu gewährleisten.

3.7.2 Der Stack

Jede Task braucht einen User-Level-Stack. In einem System ohne Swapping muss die Größe des Stacks begrenzt werden, doch auch in anderen Fällen sind Begrenzungen sinnvoll und erwünscht. Die Entwickler von Applikationen sollten also die Größe des Stacks festlegen können. Zusätzlich bietet die Speicherverwaltung die Option, auch die Lage des Stacks frei zu wählen. Eine praktikable Lösung ist, beides in einer Section in der ELF-Datei festzulegen.

3.7.3 Ein Heap

Eine auf Speicherseiten beruhende Speicherverwaltung, wie sie bisher beschrieben wurde, hat gravierende Nachteile bei vielen "kleinen" Speicheranforderungen[3]. Auch wenn nur ein Byte reserviert wird, verbraucht das eine komplette Speicherseite, die üblicherweise einige Kilobyte groß ist.

Eine Lösung ist, aus einem "Speicherhaufen" (Heap) kleine Portionen zuzuteilen. Der Heap stellt einen Puffer dar, der kleine Speicheranforderungen erfüllen kann. Aus Gründen der Sicherheit sollte es pro Adressraum einen Heap geben, der - in gewissen Grenzen - dynamisch wächst.

Die Verwaltung des Heaps könnte im Speichermanager oder im Regionenverwalter erfolgen. Die Speichermanager-Variante hat den Vorteil, dass die Verwaltung zentral vorgenommen wird. Für die Regionenverwalter-Variante hingegen spricht der Geschwindigkeitsvorteil und die leichte Einbettung in den virtuellen Adressraum. Also fiel die Wahl auf die zweite Variante.

3.7.4 Konsistenz

Mit der Einführung von User-Level-Speicherverwaltungen entstehen Konsistenzprobleme, denn neben den Verwaltungsstrukturen des Kerns gibt es jetzt auch speicherverwaltungseigene Strukturen. Beide Strukturen müssen konsistent gehalten werden. Dabei sind zwei Fälle zu unterscheiden: Speicher allokieren und Speicher freigeben. Das Allokieren ist unkritisch, denn beim Zugriff auf noch nicht in den Nutzeradressraum eingeblendete Seiten werden Seitenfehler generiert, die von der Speicherverwaltung abgearbeitet werden.

Beim Freigeben des Speicher muss dem Kern jedoch explizit mitgeteilt werden, dass die Seiten nicht mehr zum Nutzeradressraum gehören. Mit Hilfe eines L4-Systemrufes (flush) lassen sich diese Seiten auch in den L4-Kontrollstrukturen aus dem betreffenden Nutzeradressraum entfernen.

[3] "Klein" hier im Sinne von "viel kleiner als eine Speicherseite", etwa in der Relation einige hundert Byte zu einer 4kB-Seite.

3.8 Die Minishell

Eine Minishell bildet die Schnittstelle zum Benutzer. Sie soll nur die minimalen Anforderungen erfüllen, um das System steuern zu können. Neben den Ein- und Ausgaben (die auch Editiermöglichkeiten und eine History umfassen) gibt es einige eingebaute Kommandos, die die Shell unmittelbar betreffen. Ein Beispiel für ein solches Kommando ist "path", welches die Festlegung des Verzeichnisses erlaubt, aus welchem die Programme geladen werden. Die Shell übergibt den Namen zusammen mit dem Pfad an den Loader.

Zur besonderen Beobachtung und Steuerung von Projekten während der Entwicklung ist ein Monitor oft von Nutzen. Deshalb bietet die Minishell die Möglichkeit, einen beliebigen Monitor einzubinden. Die eingegebenen Kommandos durchlaufen erst die Shell, dann den Monitor, und wenn es von keinem von beiden abgearbeitet wurde, wird es an das Ziel (Programm bzw. Loader) weitergeleitet.

Da zu erwarten ist, dass im Laufe der Entwicklung einmal die Möglichkeit gegeben sein wird, mehrere Shells gleichzeitig zu bedienen, soll die Minishell nur lose an den Loader gekoppelt werden. Einem Loader stehen dann viele Shells gegenüber.

3.9 Der Loader

Ein Programm muss von einer Quelle (meist eine Festplatte lokal oder im Netz) in den RAM geladen werden. Verschiedene Möglichkeiten bieten sich an: TFTP, ein Dateisystem, auch NFS, oder L4-Linux. Im folgenden werden die einzelnen Möglichkeiten diskutiert.

- TFTP: TFTP (Trivial File Transfer Protocol, [8]) erlaubt den Transport von Dateien von und zu entfernten Rechnern. Aufgrund seiner Einfachheit ist es als Übergangslösung prädestiniert, bis geeignetere Software zur Verfügung steht. Die Einschränkungen, die TFTP auferlegt, machen es als Standardlösung untauglich.

- Dateisystem: Ideal wäre, die benötigten Dateien direkt von einem persistenten Medium, z.B. Festplatte oder Diskette, zu lesen. Ein entsprechendes Dateisystem steht allerdings unter L4 zur Zeit nicht zur Verfügung. Somit scheidet diese Variante aus.

- NFS: Für NFS sind IP- und TCP- oder UDP-Komponenten Voraussetzung. Da aber im Moment keines dieser vier für L4 implementiert ist, scheidet auch diese Variante aus.

- L4-Linux: Linux und dessen Portierung auf L4 bieten eine weitreichende Unterstützung von Dateisystemen und Hardware. Die Daten könnten unter L4-Linux geladen und per L4-IPC an die Loader-Komponente übermittelt werden. Ungünstig ist jedoch, dass immer ein L4-Linux laufen muss.

Aufgrund der Einfachheit und der Tatsache, dass das L4-Linux (insbesondere für kleinere Anwendungen des DROPS) verzichtbar sein sollte, fiel die Wahl auf TFTP.

3.9.1 Anforderungen an die Binaries

Ein Wunsch war, normale Binaries im EL-Format (kompatibel zu x86) benutzen zu können. Bis auf kleine Vorgaben kann dieser Wunsch problemlos erfüllt werden. Konkret sind dies:

- Alle Sections müssen so ausgerichtet sein, dass sie bei einer Adresse beginnen, die ein Vielfaches von 4096 ist

- Die Größe des Stack muss angegeben werden.

Mit einem Linkerscript, das dann standardmäßig verwendet werden kann, sind diese Vorgaben jedoch sehr leicht zu erfüllen.

Benutzt ein Programmierer eine objektorientierte Sprache, kann er globale Objekte vereinbaren. Die Konstruktoren dieser globalen Objekte müssen vor Beginn des eigentlichen Programmlaufes ausgeführt werden. Der Startcode wurde dafür modifiziert, er arbeitet mit dem Loader zusammen.

3.10 Zusammenfassung der Entwurfsentscheidungen

In diesem Abschnitt werden die wichtigsten Entwurfsentscheidungen noch einmal zusammengefasst.

- Eine Applikation besteht aus einem Regionsverwalter, Speicherverwaltungsstubs und dem Anwendercode. Ein spezieller Thread einer Anwendertask agiert als offizieller Pager für die Nutzerthreads.

- Es wird ein auf Datenräume und Speicherobjekte aufbauendes Modell verwendet. Um die Datenräume und Speicherobjekte zu manipulieren, sind Protokolle festgelegt worden. Jeder Thread kann über diese Protokolle mit der Speicherverwaltung kommunizieren.

- Ein einziger Server stellt Speicher mit mehreren Semantiken zur Verfügung. Gegenüber einem Multiserveransatz wurde diesem der Vorzug gegeben, weil es einige Vorteile bei der Implementierung hat. Es wird nicht erwartet, dass sich daraus Probleme ergeben. Außerdem besteht trotzdem weiterhin die Möglichkeit, unter Verwendung der schon vorhandenen Komponenten weitere Speicherverwaltungen zu implementieren.

- Die Verwaltung der Speicherseiten erfolgt mit einer Bytemap.

- Der Loader kann Binaries im EL-Format laden und eine entsprechende Task starten (lassen).

- Eine Minishell bildet die Schnittstelle zum Benutzer.

Kapitel 4

Implementierung

Bei der Wahl der Programmiersprache waren vier Gesichtspunkte zu beachten: Nutzung der Vorteile von OOP (z.B. ermöglicht die Datenkapselung eine bessere Strukturierung des Programmes), Einbindung schon vorhandener Projekte, Verfügbarkeit und die Möglichkeit, notfalls auf tiefstem Level zu arbeiten. Alles in allem betrachtet kam nur C++ infrage.

Es wurden die vorgestellten Komponenten größtenteils implementiert. Zur Zeit sind Teile des σ_0 - Protokolls und die Shared Libraries nicht benutzbar. Der Heap ist nicht implementiert. Außerdem ist das Management von Vorder- und Hintergrundprozessen noch unklar, zur Zeit werden die gestarteten Programme per Monitor gesteuert. Der Transfer ist - aus Mangel an sinnvollen Einsatzmöglichkeiten - noch nicht realisiert.

4.1 Klassen

Die Module der Speicherverwaltung wurden in C++-Klassen implementiert. Es folgt eine kurze Beschreibung der Klassen und ihrer Aufgaben.

- *Memory Object*: Objekte dieser Klassen werden zur Verwaltung der Datenräume sowie der speziellen Eigenschaften des durch dieses Objekt beschriebenen Speichers eingesetzt.

- *Page Table*: In Page Tables werden die Zuordnungen von Speicherseiten zu Datenräumen festgehalten.

- *Dataspace Manager*: Er ist der zentrale Bestandteil der Speicherverwaltung. Der Freispeicher sowie die Speicherobjekte mit ihren Eigenschaften werden von ihm verwaltet.

- *Page Fault Handler*: Die den Applikationen zugeordneten Stubs (Regionenverwalter) leiten Page Faults an ein Objekt dieser Klasse weiter. Es ermittelt die zugehörige Speicherseite und mapt sie in den Adressraum des unterbrochenen Threads.

- *L4 Page Fault Handler*: Der Thread des Regionenverwalters kann nicht sein eigener Pager sein. Ein Objekt dieser Klasse agiert als Pager für die Regionenverwalter des Systems.

- *Task*: Task-Objekte enthalten die Speicherobjekte, die Applikationen zugeordnet sind und vom Page Fault Handler benötigt werden.

- *Seizer*: Ein Objekt dieser Hilfsklasse allokiert mit Hilfe des σ_0-Protokolls Speicher für die Speicherverwaltung, der dann in die Freispeicherliste eingetragen wird.
- *Heap*: Interne Speicheranforderungen werden durch den Heap bearbeitet. Solche treten z.B. beim Anlegen neuer Objekte - etwa Speicherobjekte - auf.
- *Monitor*: Ein Objekt der Monitor-Klasse kann mit der Shell assoziiert werden. Monitore sind zur Vereinfachung der Programmentwicklung gedacht.
- *Minishell*: Die Minishell bildet die Schnittstelle zum Benutzer. Sie veranlasst den Loader, Programme zu laden und zu starten.
- *Loader*: Der Loader lädt eine Binärdatei in den Speicher, parst sie, allokiert den benötigten Speicher und veranlasst den RMGR, die Applikation zu starten.

Loader, Minishell und Monitor gehören nicht direkt zur Speicherverwaltung. In einer Weiterentwicklung des Projektes sollten sie ausgegliedert werden.

4.2 Threads in der Speicherverwaltung

Der Einsatz mehrerer Threads kann - besonders auf SMP-Maschinen - die Performance erhöhen. Ein anderer Grund mehrere Threads einzusetzen war, dadurch die Kommunikation auf Short-IPC reduzieren zu können.

Der Server besteht aus fünf Threads:

- Ein Thread kommuniziert mit anderen Komponenten. Diese Kommunikation erfolgt über die definierten Protokolle (siehe 3.5).
- Ein extra Thread wurde für die Create-Funktion des Grundprotokolles eingerichtet. Das hat rein praktische Gründe, denn dadurch wurde es möglich, durchweg die schnellere Short-IPC zu verwenden.
- Nachdem der Pager im Adressraum der Applikation die entsprechende Speicherverwaltung ausgewählt hat, sendet sie dieser eine Nachricht. Ein Thread der Speicherverwaltung wartet auf solche Nachrichten, wählt die passende Seite aus und beantwortet die Nachricht durch das Einfügen dieser Seite in den Adressraum der Applikation.
- Der L4-Page-Fault-Handler-Thread behandelt die Seitenfehler, die durch die Pager in den Nutzertasks selbst verursacht werden. (Da sie nicht ihre eigenen Seitenfehler auflösen können.)
- In dieser Implementation wurde die Minishell mit in den Adressraum des Servers mit einem eigenen Thread eingebunden. Wenn sich DROPS soweit entwickelt hat, dass mehrere Shells parallel laufen können, sollte dieser Thread durch einen Loader-Thread ersetzt werden.

4.3 Die Bibliothek

Es wurde eine Bibliothek implementiert (librm), die zu jeder Applikation gelinkt werden sollte, die diese Speicherverwaltung benutzt. Die Bibliothek erfüllt folgende Aufgaben:

- Bereitstellung der API

- Kapselung der Protokolle
- Regionenverwaltung
- Starten des ersten Nutzerthreads

Diese Bibliothek bildet den bei der Definition des Tasklayouts eingeführten Stub für die Speicherverwaltung. Auch sie ist in Form einer C++-Klasse implementiert. Die Methoden, die die API bilden, sind als "static" deklariert. So ist es leicht einen Wrapper zu schreiben, der die Methoden als Funktionen für C bereitstellt.

4.3.1 Die API

Der als Bibliothek implementierte Speicherverwaltungsstub stellt folgende Funktionen bereit:

- `void *dlmalloc(size_t size)`: Speicher der Größe size wird der Applikation zur Verfügung gestellt. Es wird garantiert, dass die Anwendung den Speicher erhält, jedoch wird er "lazy" eingeblendet, d.h. nur Speicherseiten werden zugeordnet, auf die auch zugegriffen wird.
- `void dlfree(void *ptr)`: Der Speicher, auf den ptr zeigt, wird freigegeben.
- `void *dlmmem(size_t size, dword_t flags)`: Speicher der Größe size wird der Applikation zur Verfügung gestellt. Im Parameter flags können bestimmte gewünschte Eigenschaften dieses Speichers angegeben werden. Wird Speicher gefunden, der aber nicht den Anforderunge genügt, schlägt die Anforderung fehl.
- `dword_t physical_position(void *ptr)`: Ermittelt die physische Position der virtuellen Adresse ptr.
- `void dlexit()`: Die Task mit allen Threads wird beendet. Sämtlicher allokierter Speicher wird freigegeben.

"Speicher freigeben" bedeutet in diesem Zusammenhang, dass die Assoziation zwischen Speicherobjekt und Adressraum entfernt wird. Das Speicherobjekt wird gelöscht, der Speicher wird zum Freispeicher hinzugefügt.

Darüberhinaus steht jeder Anwendung frei, diese API zu umgehend und mittels der definierten Protokolle direkt mit der Speicherverwaltung zu kommunizieren.

4.4 Registrierte Namen

Die Speicherverwaltung bietet Dienste mit bestimmten L4-Thread-IDs an. Diese IDs sind nicht fest, aber die Dienste registrieren sich bei dem Nameserver von DROPS. So hat jede Anwendung auch direkten Zugriff auf die Speicherverwaltung, falls sie es wünscht. Es steht ihr frei, auf Vorhandenes aufzubauen und z.B. ein Memory-Management mit besonderen Eigenschaften anzubieten, welches als neue Schicht über dieser Speicherverwaltung angesiedelt ist. So lassen sich zum Beispiel in den Speicher eingeblendete Dateien realisieren, wenn das Dateisystem der Applikation gegenüber Speicherverwaltungsfunktionalität anbietet.

Konkret sind folgende Dienste registriert:

- *mmem*: Die allgemeine Schnittstelle zur Speicherbeschaffung und -freigabe
- *page fault handler*: Behandelt die Page Faults, die der Regionenverwalter weiterleitet

- *L4 page fault handler und* σ_0: Behandelt die vom L4-Kern gesendeten Page Faults und außerdem das σ_0-Protokoll
- *create dataspace*: Ein extra Dienst zum Anlegen von Speicherobjekten; damit wird es möglich, durchweg L4-Short-IPC zu verwenden, und die potentielle Parallelität wird größer
- *Loader*: Er lädt Programme und veranlasst das Starten. Die Shell kann damit den Loader finden, dem sie die vom Nutzer eingegebenen Kommandos übergibt.

Ausserdem wurde der Startup-Code (crt0) so modifiziert, dass Konstruktoren globaler Objekte vor Beginn der Programmausführung aufgerufen werden.

4.5 Speicherbeschaffung

Bevor eine Speicherverwaltung Speicher bereitstellen kann, muss sie sich ihn beschaffen. Der *Seizer* allokiert mit Hilfe des σ_0-Protokolls (siehe [4]) eine Menge von Speicherseiten, die zur Freispeicherliste zusammengefasst werden.

Denkbare Weiterentwicklungen der Speicherbeschaffung wären:

- Auslagern auf einen Hintergrundspeicher
- Entzug von Speicherseiten aus L4-Linux

Kapitel 5

Zusammenfassung und Ausblick

Das Ziel war, die Voraussetungen für das dynamische Nachladen von Komponenten unter DROPS zu schaffen und eine Lösung zum Starten dieser Programme zu implementieren.

Mit der Minishell, dem Loader, der Speicherverwaltung, den Pagern und der Library wurde eine solche Lösung geschaffen. Sämtliche Komponenten laufen im Nutzeradressraum, eine Änderung des Mikrokerns erfolgte nicht.

In der Arbeit wurde schon zuweilen auf Verbesserungsmöglichkeiten aufmerksam gemacht. Einige Ansätze sind:

- Entwicklung bzw. Portierung einer mächtigeren Shell
- Vorder- und Hintergrundprozesse (mit Weiterleitung der Shelleingaben)
- Praktische Untersuchungen und vernünftige Implementierung eines anwendungsspezifischen Heaps; ein Ansatz hierfür wäre, dass "dlmalloc" grundsätzlich auf den Heap zurückgreift und dlmmem den Speicher immer direkt von der Speicherverwaltung bezieht
- Implementierung von Speicherverwaltern, die Shared Memory zur Verfügung stellen
- Implementierung des Auslagerns von Speicherseiten auf Hintergrundspeicher (Swapping)

Anhang A

Glossar

Adressraum: Menge gültiger Speicheradressen, auf die ein Thread zugreifen kann

API: "Application Programming Interface"; beschreibt die Programmierschnittstelle, die ein Dienstanbieter zur Verfügung stellt

Applikation: Anwendung, Anwendungsprogramm

DROPS: Dresden Realtime Operating System; Forschungsprojekt der Betriebssystemgruppe der Technischen Universität Dresden

Flexpage: L4-Einheit zur Speichermanipulation; mehrere *Seiten* lassen sich unter L4 zu einer Flexpage zusammenfassen

IPC: "Inter Process Communication"; in Betriebssystemen verbreiteter Mechanismus, mit dem Nachrichten innerhalb und zwischen verschiedenen *Adressräumen* ausgetauscht werden können

Library: Programmmodul, das eine Menge von Funktionen bereitstellt

Linker: Programm, welches einzeln programmierte Module zusammenbindet

Linkerskript: Kurzes Programm, welches beschreibt, wie der *Linker* die Programmmodule zusammenbinden soll

Loader: Programm oder Programmteil, welches ein Anwendungsprogramm in den Speicher lädt und startet bzw. das Starten veranlasst

Pager: Unter L4 ein Thread, der bei Seitenfehlern benachrichtigt wird

Page Fault: siehe *Seitenfehler*

Page Table: siehe *Seitentabelle*

RTFS: "Real Time File System"; ein spezielles Dateisystem unter *DROPS*, welches echtzeitfähig ist

Seite: Prozessorarchitekturdefinierte Speichereinheit

Seitenfehler: Unerlaubter Zugriff auf eine Speicheradresse; entweder gehört diese Adresse nicht zum Adressraum oder der Thread greift schreibend auf Nur-Lese-Speicher zu

Seitentabelle: Datenstruktur, die *Adressräume* aufeinander abbildet

Shared Libraries: *Library*, dessen Exemplar im System von mehreren Programmen gleichzeitig genutzt werden kann

Shared Memory: Speicher, der gleichzeitig in mehrere *Adressräume* eingeblendet ist

SMP: "Symmetric Multi Processing"; in einem Rechner sind mehrere gleiche CPUs eingebaut, die dieselben Aufgaben erledigen können; das Betriebssystem verwaltet die Prozessoren

Task: Im *DROPS*-Kontext ein Adressraum, in dem Threads laufen kännen

Thread: Kontrollfluss einer Task

Literaturverzeichnis

[1] Andrew S. Tanenbaum, *Moderne Betriebssysteme*, Prentice Hall und Hanser Verlag, 1995

[2] Bjarne Stroustrup, *Die C++ Programmiersprache*, Addison-Weslay, 1998

[3] Hans-Peter Messmer, *PC-Hardwarebuch*, Addison-Wesley, 1995

[4] Jochen Liedtke, *L4 Reference Manual (486, Pentium, PPro)* Arbeitspapier der GMD 1021, GMD - German National Research Center for Information Technology, Sankt Augustin, 1996. Available from ftp://borneo.gmd.de/pub/rs/L4/l4refx86.ps

[5] Mohit Aron, Jochen Liedtke, Yoonho Park, Kevin Elphinstone, Trent Jaeger *A Framework for VM Diversity*, Submitted to HotOS-VII

[6] M. Condict, D. Mitchell, F. Raynolds, *Optimizing Performance of Mach-based Systems By Server Co-Location: A Detailed Design*, OSF Research Institute, 1993

[7] B. N. Vershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, S. Eggers *Extensibility, Safety and Performance in the SPIN OPerating System*, SOSP, 1995

[8] K. Sollins, *RFC 1350: The TFTP Protocol (Revision 2)*, July 1992

[9] Michael Hohmuth, *Linux-Emulation auf einem Mikrokern*, Diplomarbeit, Fakultät Informatik, TU Dresden, 1996

[10] Cl.-J. Hamann, *On the quantitative specification of jitter constrained periodic streams*, In MASCOTS, Haifa, Israel, 1997

[11] G. Heiser, K. Elphinstone, S. Russell, J. Vochteloo, *Mungi: A Distributed Single Address-Space Operating System*, University of New South Wales, Technical report UNSW-CSE-TR-9704