Großer Beleg

zum Thema

Ein Konsolensystem für DROPS

Christian Helmuth
Technische Universität Dresden
Fakultät Informatik

28. August 2000

Inhaltsverzeichnis

1 Einleitung **5**

1.1 Aufbau der Arbeit . . . 6

2 Stand der Technik **7**

2.1 Das DROPS-Projekt . . . 7

2.2 EZDK . . . 7

2.3 Linux . . . 8

2.3.1 Framebuffer Device Treiber . . . 9

2.3.2 Input Device Treiber . . . 10

2.3.3 Das Linux Console Project . . . 10

2.4 Das SLIM-Protokoll . . . 10

2.5 Das DROPS Streaming Interface . . . 12

2.5.1 Allgemeines . . . 12

2.5.2 Struktur des DSI . . . 13

3 Entwurf **15**

3.1 Ansätze . . . 15

3.1.1 Konsole als L^{4}Linux-Client . . . 16

3.1.2 Unabhängiger Konsolenserver . . . 17

3.1.3 Auswahl . . . 19

3.2 Grundlagen . . . 19

3.3 Struktur der Konsole . . . 20

3.3.1 Management-Komponente . . . 21

3.3.2 Eingabetreiber-Komponente . . . 21

3.3.3 Ausgabetreiber-Komponente . . . 22

3.4 Konsolenschnittstelle . . . 22

3.4.1 Daten für die Konsole . . . 22

3.5 L^{4}Linux und die Konsole . . . 24

4 Implementierung **27**

4.1 Treiberkomponenten . . . 27

4.2 Virtuelle Konsolen 27
4.3 Konsolenprotokoll 28
4.4 Umsetzung der SLIM-Funktionen 29

5 Leistungsbewertung **31**
5.1 Zeitverhalten der einzelnen pSLIM-Befehle 31
5.2 Ansatz: Maximale Nachrichtenlänge 32

6 Zusammenfassung und Ausblick **35**

A Glossar **37**

Kapitel 1

Einleitung

Die normale computergestützte Arbeitsumgebung mit mehreren nebenläufigen Prozessen — Applikationen, Systemkomponenten — und einer Standard-Benutzerschnittstelle — Tastatur, Maus, Monitor — verlangt vom Betriebssystem neben anderen Eigenschaften flexible *Konsolenfunktionalität*[1]. Dem Nutzer sollen Möglichkeiten der Interaktion geboten werden, die einerseits alle Informationen zur Verfügung stellen, aber andererseits auch eine übersichtliche bzw. selektive Präsentation zulassen.

Als Realisierung dieser Funktionalität haben sich zwei Ansätze in der Praxis durchgesetzt: ein textbasierter und ein grafisch orientierter. Die textbasierte Lösung entstand schon sehr früh in der Entwicklung von Computersystemen und wird im UNIX-Chargon *Terminal* genannt. Hier werden Informationen in Textform verarbeitet und dargestellt. Grafische Inhalte werden nicht oder nur rudimentär unterstützt.

Später entstanden mit wachsender Leistung der Rechner und Peripheriegeräte grafische Lösungen, sogenannte *grafische Nutzerschnittstellen* (engl. Graphical User Interface - GUI). Bei diesen Ansätzen wird das grafische Ausgabemedium — Monitor — in meist rechteckige Bereiche unterteilt, die verschiedenen Applikationen exklusiv zugeteilt werden. Diese Bereiche nennt man *Fenster*.

Viele Betriebssysteme unterstützen mindestens eine dieser beiden Lösungen. Aus diesem Grunde beschäftigt sich dieser Beleg mit dem Entwurf und der Implementierung eines Konsolensystems für das *Dresden Real-Time Operating System*.

Aufgrund seiner mikrokernbasierten Multi-Server-Architektur hat dieses Betriebssystem besondere Anforderungen an eine Konsolenkomponente. Das Hauptanliegen ist hierbei, neue Forschungsergebnisse und bekanntes *Look and Feel* miteinander zu verbinden und dabei eine Integration mit bestehenden sowie zukünftigen Implementierungen zuzulassen. Das Konsolensystem muß für Nutzer ausreichend transparent sein, damit ein Umstieg von einer Timesharing- auf die QoS-Umgebung ohne Probleme möglich ist.

Die Motivation für eine neue Komponente als Nutzerschnittstelle gründet also zum einen darauf die Benutzerfreundlichkeit deutlich zu erhöhen und zum anderen die Entwicklung von

[1] Konsole [<frz.] *Datenverarbeitung* Steuerpult

Applikationen mit Nutzerinteraktion zu vereinfachen.

Eine Videokonferenz parallel zu einem eMail-Client auszuführen sollte also ebenso möglich sein wie gleichzeitiges Websurfen und Abspielen von Audio-CDs. Dabei sollte die Synchronisation nicht den Applikationen überlassen sondern vom System verborgen werden. Wichtig ist auch eine Unterstützung der Konsole für vielfältigen Hardwareumgebungen.

1.1 Aufbau der Arbeit

Der Beleg ist in sechs Kapitel gegliedert und soll den Entwicklungsprozeß von Bestandsaufnahme über Entwurf und Implementierung bis zu einer kurzen Leistungsbewertung und einem abschließenden Ausblick auf zukünftige Erweiterungen vollständig darstellen.

Kapitel 2 stellt aktuelle Entwicklungen vor, die die Grundlage für diese Arbeit bilden. Schwerpunkte sind hierbei aktuelle Linuxtreiber und das SLIM-Protokoll. Außerdem wird ein früherer Anatz einer Darstellungskomponente für DROPS diskutiert.

Im nächsten Kapitel werden verschiedene Modelle zur Realisierung der Konsole vorgestellt. Nach Auswahl eines Modells werden erforderliche Grundlagen identifiziert sowie Struktur und Eigenschaften dargestellt. Besondere Beachtung gilt hierbei der Umgebung der Konsole — DROPS — und ihrer spezifischen Anforderungen bzw. Möglichkeiten. Weiterhin werden Aspekte in Hinblick auf die Integration mit L^{4}Linuxbetrachtet und eine geeignetes Design vorgestellt.

Das Kapitel 4 beschreibt die Implementierung des Modells und das Zusammenspiel der einzelnen Softwarekomponenten.

Die Leistung der Konsole wird in Kapitel 5 untersucht. Dazu werden unterschiedliche Messungen unter bestimmten Bedingungen vorgenommen und ausgewertet.

Im Kapitel 6 werden schließlich die gewonnenen Erkenntnisse zusammengefaßt und ein Überblick über den Stand der Implementierung gegeben. Es wird weiterhin ein Ausblick auf mögliche Erweiterungen und weiterführende Entwicklungen geboten.

Kapitel 2

Stand der Technik

2.1 Das DROPS-Projekt

An der TU Dresden, Lehrstuhl Betriebssysteme, konzentriert sich die Forschungsarbeit auf eine Betriebssystem mit Multi-Server-Architektur, das Anwendungen mit *QoS*-Anforderungen unterstützt. Das ***D**resden **R**eal-Time **Op**erating **S**ystem* (DROPS) nutzt den DROPS-Mikrokern, einen Mikrokern der 2. Generation, für die Realisierung dieses Ziels.

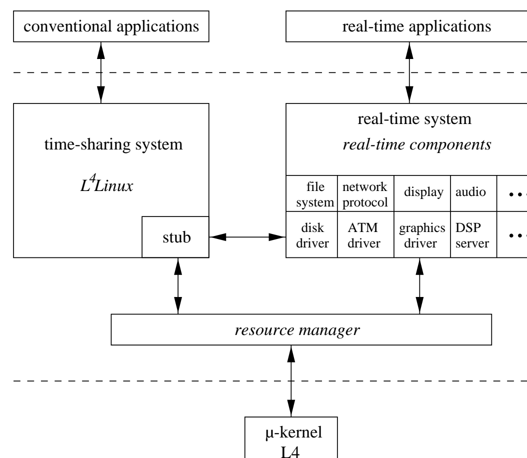

Abbildung 2.1: DROPS-Architektur (aus [1])

2.2 EZDK

Im Rahmen seiner Diplomarbeit entwarf Torsten Paul eine *Echtzeit-Darstellungskomponente (EZDK)* für DROPS [2], die es erlaubt, Grafik in Echtzeit gleichzeitig mit der normalen Arbeitsumgebung darzustellen.

Paul hat sich dabei für ein Modell entschieden, bei dem die EZDK einen eigenen Hardware-Treiber beinhaltet und somit uneingeschränkte Kontrolle über die Grafikhardware besitzt, was

eine kontrollierte Nutzung von Beschleuniger-Funktionen ermöglicht, ohne die Echtzeitdarstellung zu beeinflussen. Weiterhin kann die Synchronisation der zugreifenden Anwendungen direkt durch die EZDK durchgeführt werden. Den entscheidenden Vorteil sieht Paul jedoch in der Möglichkeit, Grafik-Applikationen zu implementieren, die direkt im DROPS-System mit EZDK ablaufen können, ohne daß ein L⁴Linux-Server o. ä. erforderlich ist.

Neben der Grafikfunktionalität bietet das Entwurfsmodell der EZDK auch eine Funktionalität für die Abfrage von Eingabeereignissen an.

Da die Schnittstelle der EZDK sehr stark an die `/dev/graphic`-Schnittstelle (siehe [2] Abschnitt 4.3.2) des GGI-Projektes angelehnt ist, wird neben den anderen Funktionen hauptsächlich von der `mmap`-Funktion Gebrauch gemacht, um den Grafikkartenspeicher in den eigenen Adreßraum einzublenden. Applikationen haben so die Möglichkeit mit normalen Speicherbefehlen direkt auf die Hardware zuzugreifen.

Verfügbar sind neben der EZDK mit Hardwaretreibern für Tastaturen und einige Grafikkarten Bibliotheken für L4-Applikationen und Linuxkern-Patches. Leider ist die Funktionalität der Bibliotheken sowie die Implementierung der Timesharing-Schnittstelle der Konsole nicht ausreichend bzw. nicht vollständig.

Ein weiteres Problem sind die zugrunde gelegten Hardwaretreiber, welche in der verwendeten Form nicht mehr weiterentwickelt werden, da der Kern des GGI-Projektes einem kompletten Redesign unterworfen wurde. Somit sind Treiber für neue Grafikkarten und Weiterentwicklungen alter Treiber nicht verfügbar.

2.3 Linux

Als frei verfügbare (siehe [3]) UNIX-Implementierung unterstützt Linux verschiedene Plattformen und bietet die wohlbekannte UNIX-Umgebung mit Netzwerkapplikationen, Editoren, Programmiertools usw.

Der Betriebssystemkern ist stets in zwei aktuellen Versionsreihen verfügbar:

stable kernel Diese Reihe wurde von Linus Torvalds als für die freie Nutzung geeignet angesehen. Derzeit ist die Version 2.2.16 aktuell.

development kernel Die Entwicklungskern-Reihe ist oft weitreichenden Änderungen von Version zu Version unterworfen. Auch wird eine Nutzung für andere Zwecke als die Entwicklung aufgrund der Instabilität der Implementierungen nicht empfohlen. Momentan Version 2.4.0-test6.

Es ist aber durchaus sinnvoll, für eigene Implementierungen einen *development kernel* zu verwenden, um die darin enthaltenen neuen Funktionen nutzen zu können, z. B. experimentelle Gerätetreiber.

Nachfolgend werde ich auf die Bestandteile des Linux-Kerns eingehen, die für den Beleg von Interesse waren. Dies sind insbesondere Treiber für Ein/Ausgabegeräte, aber auch das *Linux Console Subsystem* in Hinblick auf eine Integration der DROPS-Konsole mit L⁴Linux.

2.3.1 Framebuffer Device Treiber

Seit der Version 2.1.107 ist das *Framebuffer Device Subsystem* [4] fester Bestandteil des Linux-Kerns und bietet eine standardisierte Abstraktion für framebuffer-basierte Ausgabegeräte. Für Applikationen besteht dadurch die Möglichkeit, über eine wohl definierte Schnittstelle ohne spezifisches Wissen über die Hardware auf Videogeräte zuzugreifen.

Ein *Frame* beinhaltet dabei die vollständige Information eines Bildes, die im Grafikkartenspeicher gehalten wird. Die Grafikhardware sendet aus diesem *Puffer* abhängig von der Bildwiederholungsrate des Monitors die Bildinformationen an das Ausgabemedium. Ein Framebuffer stellt sich programmiertechnisch als linearer (flacher) Speicherbereich mit standardisierter Kodierung der Pixelwerte in Abhängigkeit vom Darstellungsmodus dar.

Auf ein Framebuffer-Gerät kann von Programmen über einen speziellen Geräteknoten *(device node)* im Linux-Dateisystem zugegriffen werden. Wie bei anderen Geräten befindet sich dieser im `/dev`-Verzeichnis:

```
crw-r--r--   1 root     root      29,   0 Feb 11 20:07 /dev/fb0
crw-r--r--   1 root     root      29,   1 Feb 11 20:07 /dev/fb1
crw-r--r--   1 root     root      29,   2 Feb 11 20:07 /dev/fb2
crw-r--r--   1 root     root      29,   3 Feb 11 20:07 /dev/fb3
```

Jeder Knoten repräsentiert ein Framebuffer-Gerät — meist Grafikkarten. Somit ist eine Grafikkarte ein Gerät mit zeichenorientierter Ein/Ausgabe *(char device)*, welches Standardoperationen wie Öffnen, Schließen, Lesen und Schreiben unterstützt. Außerdem gibt es die sogenannte `ioctl()`-Funktion, die das Setzen und Abfragen von gerätespezifischen Parametern bereitstellt, in unserem Fall z. B. Grafikmodus, Farbpalette und Bildschirmorganisation.

Zusätzlich bietet die Schnittstelle auch die Möglichkeit, den Framebuffer in den Adreßraum der Applikation einzublenden *(mmap)*. Dies ermöglicht den Zugriff über normale Operationen innerhalb des Nutzeradreßraumes. Weiterhin ist es möglich, optional Geräteregister zu *mappen* und diese zu modifizieren — *Memory Mapped I/O (MMIO)*.

Von besonderem Interesse ist natürlich der Umfang der Hardware-Unterstützung und die Stabilität der Gerätetreiber im Betrieb. Hier ist zu erwähnen, daß die Anzahl der unterstützten Grafikkarten im Vergleich zu etablierteren Umgebungen, wie z. B. dem XFree86 [5], eher klein ist. Aufgrund der wachsenden Zahl der Entwickler steigt sie jedoch stetig an und ist im Vergleich zu verwandten Projekten recht umfangreich.

Es gibt neben diesem Ansatz noch weitere Versuche, den Linuxkern um eine grafische Schnittstelle zu erweitern (GGI [6], KGI [7]). Da sich diese Projekte aber noch im *proof-of-concept* bzw. frühen Entwicklungsstadium befinden und damit oft gravierenden Designänderungen sowie Instabilitäten unterworfen sind, waren sie für diesen Beleg nicht oder nur am Rande relevant.

2.3.2 Input Device Treiber

Linux unterstützt ein Vielzahl von Eingabegeräten wie Maus, Tastatur, USB-Geräte sowie die serielle und parallele Schnittstelle. Die Treiber für diese Geräte aber haben keine einheitliche Struktur bzw. Schnittstelle.

Ein Ansatz zur Beseitigung dieses Zustands ist das *Linux Input Drivers Project* [8] von Vojtech Pavlik. Dieses Projekt beinhaltet eine Sammlung von Gerätetreibern, die zur Unterstützung aller Eingabegeräte unter Linux entworfen wurde, welche zwar noch in keiner offiziellen Linuxkern-Version enthalten ist[1], aber als Quellcode-Patch zur Verfügung gestellt wird.

Im Mittelpunkt steht das Input-Modul, das zwei Gruppen von Modulen miteinander verbindet — *device driver* und *event handler*. Die zweite Gruppe versendet, wo es nötig ist, Ereignisse über verschiedene Schnittstellen, z. B. Mausereignisse über ein simuliertes PS/2-Interface und Tastaturereignisse zum Betriebsystemkern.

In Zukunft aber sollte eine uniforme Behandlung aller Ereignisse angewendet werden, was aber eine Anpassung der Applikationen (X-Server, GPM usw.) erfordert. Hierfür steht die `evdev`-Schnittstelle zur Verfügung (`/dev/input`):

```
crw-r--r--   1 root     root      13,  64 May 23 15:05 event0
crw-r--r--   1 root     root      13,  65 May 23 15:05 event1
crw-r--r--   1 root     root      13,  66 May 23 15:05 event2
crw-r--r--   1 root     root      13,  67 May 23 15:05 event3
```

Somit existiert eine standardisierte Schnittstelle zu Eingabegeräten über ein *char device.*

2.3.3 Das Linux Console Project

Das Ziel des *Linux Console Project* ist eine komplette Überholung des Konsolensystems im Linuxkern in Bezug auf „echte“ Terminalemulation und Multiheadunterstützung. Wichtigste Bestandteile sind die besprochenen Framebuffer und Input Device Subsysteme (Abschnitte 2.3.1 und 2.3.2).

Die Ideen und Implementierungen sollen in den Linuxkern-Versionen der 2.5.x Reihe verfügbar werden.

2.4 Das SLIM-Protokoll

Das jüngste Thin-Client Produkt von Sun Microsystems heißt *Ray Hot Desk Architecture* und wird auch als Ultra Thin Client bezeichnet. Dabei ist die einzige Aufgabe des Terminals die Kommunikation mit einem Server über das SLIM-Protokoll — *Stateless Low-level Interface Machine* [9]. Diese simple Geräteschnittstelle ist hard- und softwareunabhängig. Somit könnte ein SLIM-Client sowohl X-, Win32- als auch Java-AWT-Anwendungen steuern.

[1] Das ist nicht ganz richtig, denn die Unterstützung für USB-Geräte ist ein Teil dieser Driver Suite und schon jetzt Bestandteil des Linuxkerns.

Durch diese vielseitige Verwendbarkeit ist das Protokoll als Schnittstelle für eine grafische DROPS-Konsole besonders geeignet.

Der interessante Kern des Protokolls ist die Lowlevel-API für den Austausch grafisch orientierter Informationen zwischen einem Server und einem Terminal. Diese API besteht aus nur fünf Kommandos:

SET setzt den exakten Pixelwert für einen rechteckigen Bereich

FILL füllt ein Rechteck mit dem eingestellten Pixelwert

BITMAP füllt ein Rechteck mit einem zweifarbigen Muster

COPY kopiert einen (rechteckigen) Bereich aus dem Framebuffer an eine andere Position

CSCS (color-space convert and scale) wandelt YUV-kodierte Rechtecke nach RGB mit optionaler bilinearer Skalierung

Diese Kommandos definieren die Schnittstelle zur Grafikhardware und erlauben *insbesondere* kein Einblenden des Framebufferspeichers der Grafikkarte in Adreßräume anderer Prozesse. Vielmehr wird ein *virtueller Framebuffer* zur Verfügung gestellt, der durch seine optimierte Lowlevel-API auch für Kanäle mit geringem Datendurchsatz, z. B. Netzwerke, als Schnittstelle eine geeignete Abstraktion der Grafikhardware bietet.

Hierbei sind die Befehle BITMAP und CSCS für spezielle Einsatzbereiche besonders hervorzuheben. Das letztere Kommando soll unter anderem den Transport von Videodaten auf die SLIM-Konsole beschleunigen, da dadurch die serverseitige Wandlung nach RGB entfallen kann. Das Kommando BITMAP wiederum erlaubt zweifarbige grafische Informationen, wie sie bei Textkonsolen o. ä. auftreten, effizient zu übertragen, da pro Pixel nur ein Bit und zusätzlich für den ganzen rechteckigen Bereich zwei Farbwerte — Hinter- und Vordergrundfarbe — benötigt werden.

Die in [9] (Abschnitt 4.3) besprochenen Protokollberechnungskosten zeigen, daß die speziellen Befehle trotz hoher Anlaufkosten akzeptable Ergebnisse liefern können. Entscheidend sind hierbei die Anzahl der Pixel bzw. die benötigte Bandbreite. Für genaue Werte sei auf [9] (Tabelle 5) verwiesen.

Das SLIM-Protokoll verlangt, daß Desktop-Rechner nur simple, „statuslose“ Ein/Ausgabe-Geräte sein sollen. Somit sind die Anforderungen an eine SLIM-Konsole in Hinblick Rechen- und Speicherkapazität gering. In [9] wird dazu erwähnt: Die SLIM-Konsole ist einfach ein „dummer“ Framebuffer.

Die Software — *firmware* — der SLIM-Konsole koordiniert die Aktivitäten zwischen den Komponenten, z. B. Datenströme zwischen dem Kanal und den zugehörigen Geräten. Die SLIM-Server wiederum erhalten mehr Funktionalität und steuern alle Geräte der Konsole entfernt — remote device management — die Programmabarbeitung wird vollständig von der Konsole getrennt. Das bedingt hohe Kapazitäten auf der Serverseite, was durch Serverpools und Lastverteilung ökonomisch durchführbar ist.

2.5 Das DROPS Streaming Interface

Zur Datenübergabe, egal ob lokal — innerhalb eines Adreßraumes — oder über Adreßraumgrenzen hinaus, gibt es mehrere Möglichkeiten und noch mehr Bezeichnungen. Die Grundformen sind aber nur Referenz- und Wertübergabe, wobei allein die Semantik der zweiten Form eine Kopieroperation voraussetzt[2]. Bei Überschreitung von Adreßraumgrenzen während der Datenübergabe, vor allem bei Netzwerkkommunikation - RPC, wird meist die Übergabe per Referenz durch doppeltes Kopieren emuliert. Das ist aber nicht nur teuer, sondern ergibt auch Inkonsistenzen.

Aus diesen Gründen und in Hinblick auf Echtzeit-Applikationen wurde das *DROPS Streaming Interface (DSI)* [10] entwickelt. Ich werde nun einen kurzen Vergleich zu den traditionellen Formen der IPC durchführen.

2.5.1 Allgemeines

Der Nachrichtenmechanismus des Mikrokerns bietet die bekannte Datenübergabe per IPC in zwei Ausprägungen. So ist es möglich bis zu 2 MB Daten *in-line*, d. h. direkt im Nachrichtenpuffer enthalten, zu versenden. Zu beachten ist hierbei, daß alle Daten fortlaufend (linear) im Adreßraum des Senders mit direkt vorangestelltem Nachrichtenkopf verfügbar sein müssen.

Eine andere Möglichkeit ist, verschiedene Datenpuffer zu einer Nachricht zusammenzufassen, indem man *String Dopes* für *out-line* Daten definiert. Auf diese Weise ist es möglich, bis zu 32 Strings á 4 MB im Maximalfall als eine Nachricht zu versenden. Dafür ist es lediglich nötig die Anfangsadresse und Länge der Zeichenketten in den Dopes zu kodieren. Beide Ausprägungen der IPC stellen *by-value* Parametersemantik zu Verfügung, da der Mikrokern die Daten in den Adreßraum des Empfängers kopiert.

Möchte man die Kopieroperation einsparen oder „echte“ *by-reference*-Parametersemantik durchsetzen, kommt man nicht umhin, mit Adreßregionen zu arbeiten, die in beiden Adreßräumen sichtbar sind. Diese *Shared Memory*-Technik wird vom Mikrokern durch einen Mechanismus zum Einblenden — *Mapping* — von Seiten des virtuellen Adreßraumes eines Senders in den Adreßraum eines Empfängers unterstützt.

Die sogenannten *FlexPages (Fpages)* werden genau wie andere Daten in einer IPC-Nachricht kodiert und anschließend mit einem Systemruf abgeschickt. Die Größe der Seite ist dabei mit wenigen Einschränkungen „flexibel“ festzulegen. So ist die Mindestgröße abhängig von der Prozessorarchitektur und entspricht einer Hardwareseite, z. B. 4 KB bei x86. Die mögliche Maximalgröße umfaßt den gesamten virtuellen Adreßraum.

Auf diese Art ist es relativ einfach, strukturierte Daten ohne Kopieroperation als Parameter eines Funktionsaufrufes per IPC zu übergeben. Negativ wirken sich hier nur die Kosten des Mappings aus, so daß wiederholtes Ein- und Ausblenden von Fpages keine gute Lösung

[2] Manche Systeme versuchen auch diese einzusparen und führen eine sogenannte *copy-on-write*-Semantik ein.

darstellt. An dieser Stelle setzt das DSI an und bietet unter Nutzung der besprochenen Mechanismen des Mikrokerns ein komfortable Schnittstelle mit statischen Fpages.

2.5.2 Struktur des DSI

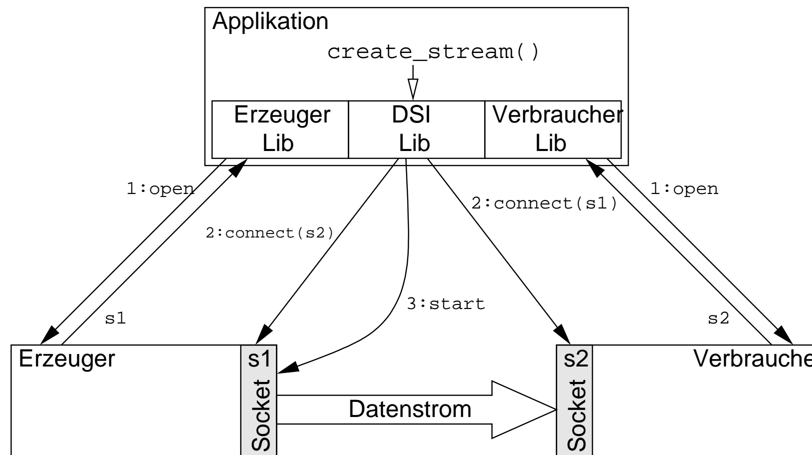

Abbildung 2.2: DROPS Streaming Interface

Es wird als Motivation die Lösung des klassischen Erzeuger-Verbraucher-Problems bei Verbergung der Komplexität für Systemkomponenten in verschiedenen Adreßräumen für DROPS zugrunde gelegt. Wegen seiner unidirektionalen Eigenschaft ist das DSI am besten mit einer *Pipe* zu vergleichen.

Wie in Abb. 2.2 zu erkennen ist, besteht das einfachste Szenario aus einer Applikation, welche zwei kooperierende Systemkomponenten nutzt. Die Applikation erstellt einen Datenstrom (`create_stream`) zwischen den Komponenten, welche anschließend selbständig den Datenaustausch durchführen.

Der interessante Aspekt ist hierbei das Design des *Stroms* bzw. die Datenschnittstelle der Komponenten. Das DSI besteht aus zwei Shared-Memory-Bereichen und einer Bibliothek mit Servicefunktionen (Abb. 2.3). Hierbei wird davon ausgegangen, daß eine signifikante Performancesteigerung durch Reduzierung von Mapping-Operationen erzielt werden kann. Deshalb präsentiert sich das Interface statisch in Bezug auf beteiligte Fpages und minimiert so die Kosten für das Ein/Ausblenden von Regionen des virtuellen Adreßraumes.

Die Aufgabe des Kontrollbereiches (siehe Abb. 2.3) ist es, DSI-interne Deskriptoren für Pakete und Daten zu halten. Der Bereich wird von der Bibliothek organisiert und bleibt vor den Komponenten verborgen. Komponenten, die das DSI nutzen, kennen nur Datentypen für Pakete und Daten sowie Bibliotheksfunktionen. Ein Paket kann dabei mehrere Datendeskriptoren beinhalten und somit verstreute „Datenschnipsel“ logisch zusammenfügen — *scatter-gather*.

Die eigentlichen Daten liegen im Datenbereich, der aus Sicht des DSI eine unstrukturierte

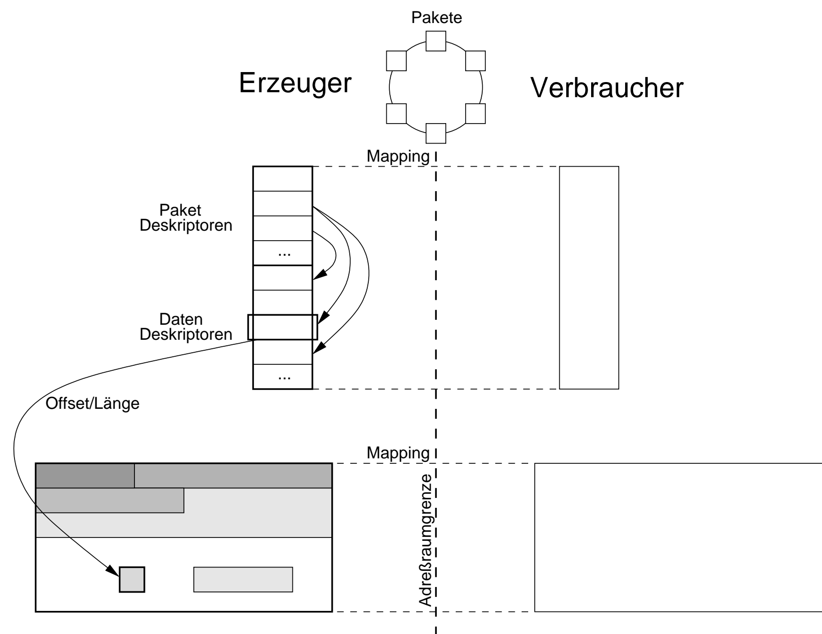

Abbildung 2.3: DSI intern

Region des Adreßraumes ist. Die Verwaltung dieses Bereiches obliegt allein den beteiligten Komponenten. Es wird insbesondere kein Schutzmechanismus eingeführt für den Fall, daß durch Deskriptoren definierte Datenbereiche sich überschneiden.

Das DSI stellt den Komponenten eine endliche Anzahl Pakete zur Verfügung, die in einer Ringliste organisiert sind, so daß zum einen die Reihenfolge erhalten bleibt und andererseits nur korrekte Pakete weiterverarbeitet werden. Das Erzeuger-Verbraucher-Problem wird mit zwei Semaphoren pro Paket gelöst.

Die Bibliotheksschnittstelle ist hier auf beiden Seiten bewußt gleich gehalten. So ist es egal ob Erzeuger- oder Verbraucherrolle, `get_packet` liefert einen Paketdeskriptor und `commit_packet` gibt einen Deskriptor frei. Der Programmabschnitt zwischen den beiden Funktionen unterscheidet sich natürlich bei den beiden Komponenten. Der Erzeuger wird hier Datenbereiche in das Paket einfügen (`add_data`), während der Verbraucher solche ausliest (`get_data`).

Die Vorteile der Schnittstelle sind leicht zu erkennen:

- Reduzierung der Kopieroperationen
- Struktur der Daten unabhängig vom DSI
- Verkettung mehrerer Komponenten unter Nutzung des gleichen Datenbereiches

Für die Zukunft ist geplant, die Funktionalität des DSI um Echzeit-Eigenschaften, z. B. Gültigkeitsdauer von Paketen, und metastrukturierte Datenbereiche mit Verwaltungsmöglichkeiten innerhalb der DSI-Bibliotheken zu erweitern.

Kapitel 3

Entwurf

Nachdem ich im vorangegangenen Kapitel die aktuellen Entwicklungen, die die Grundlage dieser Arbeit bilden, erläutert habe, möchte ich nun auf verschiedene, mögliche Modelle für das Design der Konsole und deren Schnittstelle eingehen.

Die Zielstellung der Arbeit ist ein Konsolensystems für das Dresden Real-Time Operating System (DROPS).

Das Konsolensystem soll DROPS-Komponenten eine standardisierte Schnittstelle zur Ein- und Ausgabe von Daten über Tastatur, Maus bzw. Bildschirm sowie andere Konsolengeräte anbieten. Es soll mit den Komponenten über den Nachrichtenmechanismus des DROPS-Mikrokerns (L4-IPC) kommunizieren und mit L^{4}Linux integriert sein, so daß alle beteiligten Komponenten konfliktfrei auf die Konsole zugreifen können.

3.1 Ansätze

Ein Konsolensystem, wie es hier angestrebt wird, soll eine Verbindung von Nutzerschnittstelle mit Ein/Ausgabe und Programmen bzw. Komponenten des Systems herstellen. Dabei soll diese Verbindung eine eindeutige Abbildung von Applikationen auf Konsolengräte zu einem bestimmten Zeitpunkt darstellen, d. h. Konflikte von Eingabe- und Ausgabeströmen dürfen nicht auftreten.

Weiterhin soll der Nutzer (oder möglicherweise auch eine Systemkomponente) in der Lage sein, diese Abbildung zu verändern. Hier werden sogenannte *virtuelle Konsolen*, welche verschiedene eindeutige Konsolenzustände darstellen, eingeführt und eine Umschaltung ohne inkonsistente Zwischenstufen zugelassen.

In Abb. 3.1 ist die *Abbildung* von Applikationen auf Geräte bewußt einfach gehalten. Es ist z. B. denkbar, wie es zum Beispiel Fenstermanager bei grafischen Nutzerschnittstellen tun, nur rechteckige Bereiche eines Ausgabegerätes — Grafikkarte — einer bestimmten Applikation zuzuordnen und möglicherweise sogar Überlappungen zuzulassen. Das kann aber einfach durch einen schichtenartigen Aufbau der Konsole bzw. Ersetzung der Management-Komponente erreicht werden.

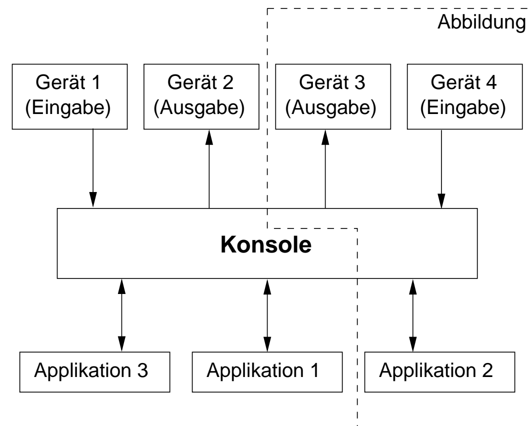

Abbildung 3.1: Schematische Konsole - eine Abbildung

Systeme mit virtuellen Konsolen sind den Nutzern vertraut, z. B. Linux oder X-Windows mit mehren X-Terminals, und so sollte eine transparente Integration in die Arbeitsumgebung ohne Probleme möglich sein.

Es sollen nun vier verschiedene Modelle für die Implementierung der Konsole untersucht werden:

1. L⁴Linux-Client unter Ausnutzung der Pseudoterminalfunktionalität
2. Textkonsolen-Server als L4-Task
3. Text/Grafikkonsolen-Server als L4-Task (mit erweiterter Schnittstelle)
4. Separate Server für Text und Grafik als L4-Tasks

3.1.1 Konsole als L⁴Linux-Client

Modell 1 Die Konsole stellt sich hier als L⁴Linux-Programm dar, das gleichzeitig Linux-funktionalitäten und den Nachrichtenmechanismus des Mikrokerns verwendet. Deshalb ist eine Nutzung der Konsole ohne L⁴Linux-Server nicht möglich.

Serverseitig öffnet die Konsole ein Linux-Pseudoterminal als Master und leitet Ausgabeströme von anderen DROPS-Komponenten auf dieses um. Das Master-Terminal „füttert" einen Slave mit den ankommenden Informationen, der nun wie ein normales Terminal angesprochen werden kann.

Die Nutzerschnittstelle der Konsole ist somit durch ein Terminalprogramm zur Verfügung zu stellen, z. B. `minicom`. Solch ein Programm kann sich dann mit einem Pseudoterminal (slave) verbinden und über die standardisierten Terminalbefehle kommunizieren.

Dieses Modell hat neben der Anforderung, daß ein L⁴Linux-Server verfügbar sein muß, noch einen weiteren großen Nachteil: Die Kommunikation mit der Konsole ist nur textbezogen,

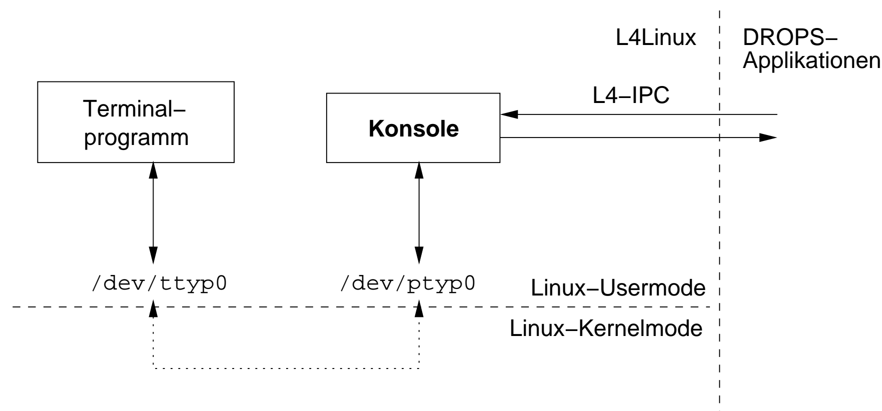

Abbildung 3.2: L^{4}Linux-Client und Pseudoterminals

d. h. die Konsole unterstützt selbst keine Grafikanwendungen und kann somit das Auftreten von Konflikten zwischen verschiedenen DROPS-Komponenten mit grafischer Ausgabe nicht verhindern.

3.1.2 Unabhängiger Konsolenserver

Modell 2 Die Konsole läuft als eigenständiges Programm (L4-Task) und bietet eine textorientierte Schnittstelle über den Nachrichtenmechanismus des Mikrokerns an. Die benötigten Gerätetreiber sind Bestandteil der Konsole. (Abb. 3.3)

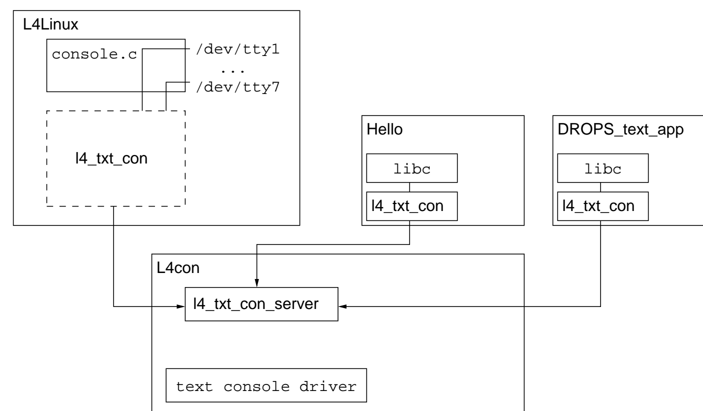

Abbildung 3.3: Eigenständiger Textkonsolen-Server

In Hinblick auf ein DROPS ohne zwingend notwendigen L^{4}Linux-Server bietet dieses Modell die Grundlage für eine „schlanke" Konfiguration ohne große Einbußen an Funktionalität. DROPS-Komponenten sind somit in der Lage (wie auch schon in Modell 1) zeichenorientierte *Terminalfunktionen* für Ein- und Ausgabe zu nutzen.

Nötig ist auch eine Anpassung des L^{4}Linux-Servers, so daß dieser für Anwendungen transpa-

rent nicht seine eigenen Hardwaretreiber benutzt, sondern über IPC auf die Konsole zugreift, da ansonsten Konflikte zwischen Linux- und Konsolenausgaben nicht auszuschließen sind.

Obwohl dieses Modell gegenüber dem ersten den Vorteil der Unabhängigkeit vom L4Linux besitzt, bleibt die nachteilige Einschränkung auf eine rein textorientierte Konsole erhalten. Eine Erweiterung in dieser Hinsicht bringt das nächste Modell.

Modell 3 Unter den gleichen Bedingungen wie bei Modell 2 soll in diesem Entwurf neben der Textkonsolenfunktionalität eine um Grafikbefehle erweiterte Schnittstelle angeboten werden. Wieder sollen alle benötigten Hardwaretreiber Bestandteil der Konsole sein. (Abb. 3.4)

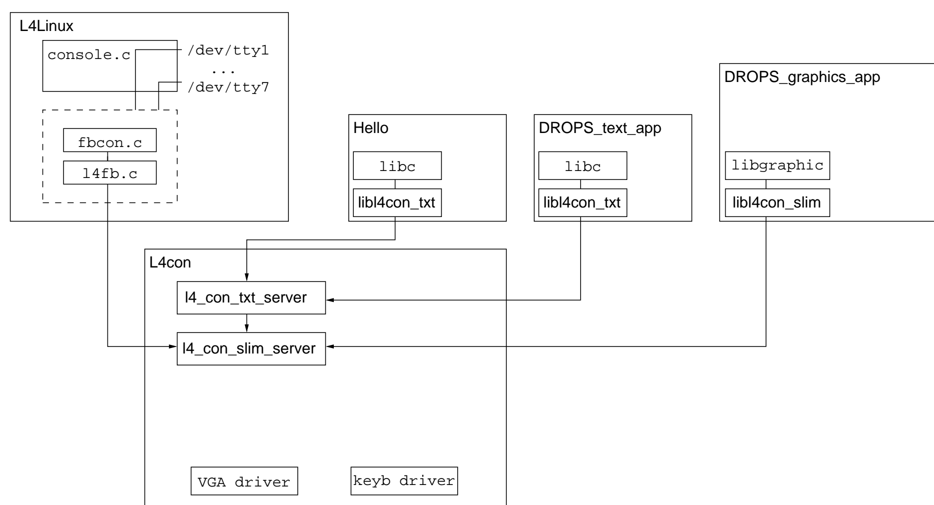

Abbildung 3.4: Kombinierter Text- und Grafikkonsolen-Server

Die Konsole bietet allen Komponenten des Systems, text- wie grafikorientierten, eine zentrale Schnittstelle. Somit lassen sich alle Konflikte zwischen verschiedenen Programmen in Hinblick auf die Nutzerschnittstelle — Ausgabe und Eingabe — unterbinden.

Dieses Modell ist der EZDK von Paul [2] sehr ähnlich. Hier wurde nur eine Ausrichtung auf Echtzeitanforderungen nicht beachtet.

Modell 4 Im Vergleich zu Modell 3 wird die Konsolenfunktionalität auf zwei Server verteilt: rein grafischer Konsolenserver und *Textwrapper*. (Abb. 3.5)

Der Textwrapper hat die Funktion eines Textrendering-Gerätes, das zeichenorientierte Informationen in grafische umwandelt und möglicherweise auch *Scrolling* o. ä. anbietet.

Da die textorientierte Schnittstelle aus dem Konsolenserver entfernt wurde, ist die Schnittstelle nun einfacher aufgebaut und bezieht sich nur noch auf grafische Operationen (plus einige zusätzliche Anfragen; siehe Abschnitt 3.4). Es lassen sich nun auch leicht Konfigurationen für rein grafische Umgebungen erstellen.

Diese Trennung von Text- und Grafikserver erlaubt auch einige andere Möglichkeiten:

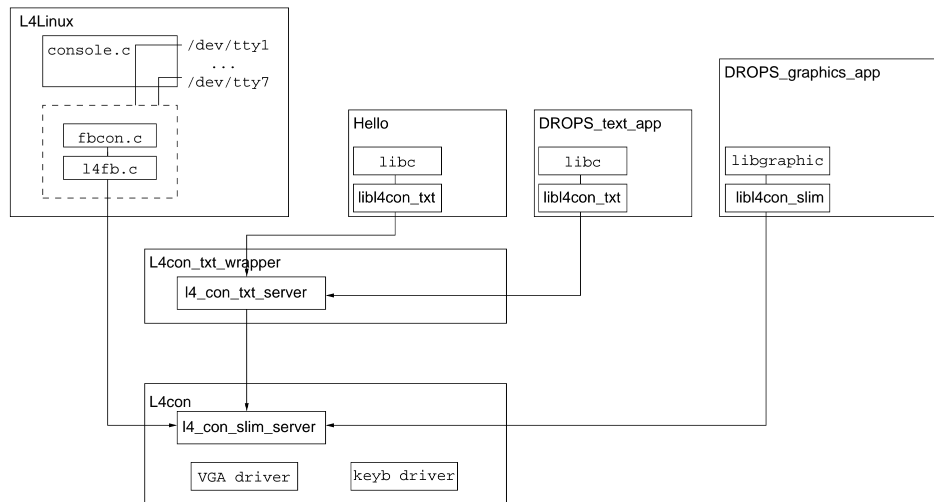

Abbildung 3.5: Grafikkonsolen-Server und Textwrapper

- Der Textwrapper könnte, z. B. zum Loggen von Meldungen, verschiedene Clients auf *eine* virtuelle Konsole multiplexen. Eine ähnliche Funktionalität bietet der DROPS-Logserver, wobei dieser aber direkt mit der Hardware arbeitet.
- Die transparente Umleitung von Textkonsolen über die serielle Schnittstelle läßt sich sehr einfach ohne Einbeziehung der „echten“ Konsole durchführen.
- Denkbar wäre auch eine Höherpriorisierung von grafischen Anwendungen durch diese Architektur.

3.1.3 Auswahl

Wie auch schon bei Paul in [2] zeigt die Betrachtung der Vor- und Nachteile der Modelle, daß eine unabhängige Implementierung die flexibelste Lösung darstellt. Außerdem ist zu beachten, daß eine zunkunftsorientierte DROPS-Konsole möglichst neben zeichenorientierten Ausgaben, z. B. für Statusausgaben während des Bootvorgangs, Minishells usw. , auch grafische Ausgaben unterstützen sollte.

Aus diesen Gründen und zur Vermeidung einer übermäßigen Ausdehnung des Umfangs des Beleges soll die Konsole nach Modell 4 entwickelt werden. Alle weiteren Betrachtungen beschäftigen sich nur mit der grafischen Komponente der Konsole. Für den Textwrapper sollten später Lösungen wie z. B. die Framebuffer Konsole [4] analysiert werden.

3.2 Grundlagen

Die wichtigsten Vorraussetzungen für die Realisierung des Modells sind Grafik- und Eingabegerätetreiber für eine möglichst große Menge von Hardware. Weiterhin sollten diese Treiber

einfach zu portieren sein und auch in Zukunft weiterentwickelt sowie gut gewartet werden. Wichtig ist an dieser Stelle auch die Möglichkeit der Integration mit der normalen Arbeitsumgebung, d. h. besonders ein mit der neuen Konsole harmonierender X-Server.

Im Gegensatz zu den Vorraussetzungen bei Paul [2] zeigte sich, daß zum momentanen Zeitpunkt mehrere Projekte an einer standardisierten Schnittstelle zur Grafikhardware arbeiten. Aus diesen Projekten soll das *Linux Framebuffer Device Project* ausgewählt werden (siehe 2.3.1), da es zum einen schon Bestandteil des Linuxkerns ist und zum zweiten die größte Anzahl an Grafikkarten unterstützt.

Als Grundlage für die Eingabe sollen die Treiber des *Input Drivers Project* dienen, da es eine Vielzahl von Geräten unterstützt, auf welche durch eine wohl definierte Schnittstelle zugegriffen wird. (siehe Abschnitt 2.3.2)

Da für beide Projekte Linux die Plattform bildet, können erprobte Methoden bei der Portierung der Treiber aus [11] genutzt werden. Außerdem müssen in L^{4}Linux Schnittstellen identifiziert werden, die die Nutzung von externen Grafik- und Eingabetreibern erlauben.

3.3 Struktur der Konsole

Die Konsole läßt sich recht einfach in drei bzw. vier Komponenten unterteilen:

- Management für virtuelle Konsolen mit Service-Threads
- Eingabetreiber
- Ausgabetreiber
- nicht eindeutig als Komponente identifizierbar - die IPC-Schnittstelle

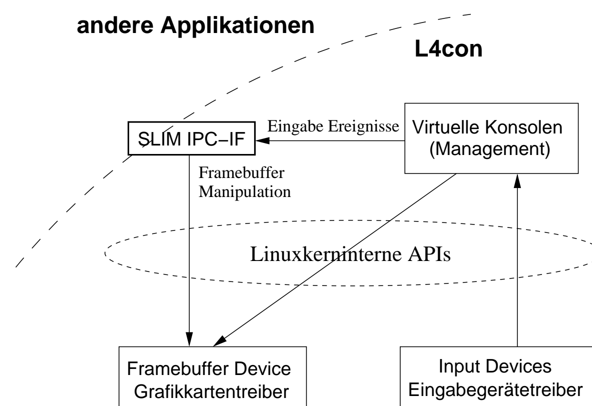

Abbildung 3.6: Konsolenstruktur

Wichtig ist an dieser Stelle, daß eindeutige Schnittstellen jeder Komponente bekannt sind, um diese wartbar und möglicherweise sogar austauschbar zu gestalten. Neuere Versionen der Treiberkomponenten sollen ohne großen Aufwand eingespielt werden können.

3.3.1 Management-Komponente

Diese Komponente ist vollständig neu, vielleicht in Anlehnung an vorhandene Implementierungen, zu entwickeln. Das Ziel ist, daß zu jedem Zeitpunkt alle Konflikte zwischen verschiedenen Applikationen — Clients der Konsole — unterbunden werden, d. h. vor allem, daß Hardwarezugriffe jeweils nur einer Applikation erlaubt werden. Außerdem ist zu beachten, daß Eingabeereignisse korrekt an angemeldete Empfänger versandt werden.

Die günstigste Lösung ist eine Management-Komponente mit mehreren Threads — Service-Threads, die jeweils 1:1 den Clients zugeordnet werden. Die Synchronisation soll hier ein Thread — Master-Thread — übernehmen. Dieser soll die Konsole auch im System repräsentieren, d. h. beim Nameserver angemeldet sein und Anfragen zum Öffnen einer virtuellen Konsole entgegennehmen.

Zusammengefaßt bedeutet das, daß bei n Kommunikationspartnern n Service-Threads, ein Master-Thread und ein Thread für die Eingabekomponente im Adreßraum der Konsole ablaufen[1].

Während des Umschaltens von einem Konsolenzustand in einen anderen ist der Master-Thread aufgrund seiner Synchronisationseigenschaft nicht im Wartezustand für IPC-Anfragen. Das sollte aber nicht kritisch sein, da der hier auszuführende Code nicht umfangreich ist und Anfragen an diesen Thread, im Regelfall nur das Öffnen einer virtuellen Konsole, nicht zeitkritisch sind.

3.3.2 Eingabetreiber-Komponente

Für diese Komponente wird der Quellcode vom *Input Drivers Project* unverändert übernommen. Nach der Initialisierung während des Starts der Konsole arbeitet ein separater Ereignis-Thread alle Eingabeereignisse ab und versendet diese an Partnerthreads. Sind solche nicht angemeldet oder während des aktuellen Konsolenzustands nicht berechtigt Ereignisse zu empfangen, werden diese verworfen.

Für die Kodierung der Ereignisse wird die des Originals übernommen, da sie neben der Information über die Art des Ereignisses und die Quelle auch einen Zeitstempel enthält. Ungünstig würde sich eine andere Kodierung auch bei der Integration mit L^{4}Linux auswirken, da hier in jedem Fall wieder die Originalkodierung hergestellt werden muß.

Die zweite wichtige Aufgabe der Eingabekomponente ist das Erkennen und Weiterleiten von Umschaltereignissen, d. h. i. d. R. gleichzeitiges Drücken von bestimmten Tastenkombinationen, an die Managementkomponente, so daß diese die nötigen Operationen durchführen kann.

[1] Es könnte hier auch sein, daß mehr Threads ablaufen, wenn z. B. die Implementierung von Timern im Linux-Treiberframe dies verlangt.

3.3.3 Ausgabetreiber-Komponente

Auch bei der Ausgabe-Komponente bleibt der Original-Quellcode unverändert erhalten. Der Unterschied zur Eingabe besteht nur darin, daß es nicht ein fester Thread ist, der auf der Grafikhardware arbeitet. Vielmehr ist es günstig den Service-Thread der aktuellen Vordergrundkonsole direkt auf der Hardware arbeiten zu lassen und alle anderen Service-Threads auf Bereichen des Arbeitsspeichers.

Es ist somit für die Service-Threads nicht nötig zu wissen, ob sie der aktuellen Vordergrundkonsole angehören. Sie greifen transparent über einen Eintrag in einer Verwaltungsstruktur auf Speicherbereiche zu — Arbeitsspeicher oder gemappter Framebuffer der Grafikkarte.

Auf die IPC-Schnittstelle und ihre Eigenschaften werde ich im nächsten Abschnitt genau eingehen.

3.4 Konsolenschnittstelle

Die Grundlage der Konsolenschnittstelle sollen der Nachrichtenmechanismus des Mikrokerns und das SLIM-Protokoll bilden (siehe Abschnitt 2.4). Da detailierte Informationen zu diesem Protokoll nicht verfügbar waren, wird der Schnittstellenentwurf nur einen ähnlichen Aufbau haben und einige Erweiterungen enthalten. Die Vorteile von SLIM sollen dabei erhalten bleiben.

Die erwähnten SLIM-Befehle zu Operationen auf einem virtuellen Framebuffer sollen in vollem Funktionsumfang verfügbar sein, da sie einen entscheidenden Vorteil in Bezug auf transparenten Austausch der Vollbild-Konsole durch einen Fenstermanager mit sich bringen. Das sogenannte *Clipping* läßt sich nämlich bei gemappten Speicherbereichen nur schwer implementieren.

Zusätzlich und im Gegensatz zum Original-SLIM soll der Grafikmodus nicht fest voreingestellt werden. Kommandos zum Abfragen und Setzen der Bildschirmauflösung und Farbtiefe sind Bestandteil der Schnittstelle.

Weiterhin sollen Kommandos zur Anforderung und Filterung eines Eingabestroms vorgesehen sein. Damit ist der Client in der Lage selbst zu bestimmen, welche Eingabeereignisse er zugesendet bekommt, z. B. nur Tastaturereignisse.

Ein wichtiger Punkt beim Entwurf der Schnittstelle ist das Design des Datenpfades, besonders wenn es sich um große Datenmengen handelt. Deshalb gehe ich auf diesen Punkt etwas genauer ein.

3.4.1 Daten für die Konsole

Wird von einer Applikation in ihrem Adreßraum ein vollständiger Frame berechnet, z. B. Fraktale, und als „großer Brocken“ (SET Befehl) an die Konsole übergeben, entstehen gleich mehrere Probleme, da das Datenvolumen recht groß ist:

- sparsame Beispielkonfiguration: 640x480 Bildpunkte bei 16 Bit Farbtiefe (2^{16} Farben)

$$640 \cdot 480 \cdot 2 = 614.400\,\text{byte}$$

- übliche Konfiguration aktueller Desktopsysteme: 1024x768 bei 24 Bit (2^{24} Farben)

$$1024 \cdot 768 \cdot 3 = 2.359.296\,\text{byte}$$

Somit muß von der Konsolenkomponente ein sehr großer Pufferspeicher für IPC-Nachrichten zur Verfügung gestellt werden. Das läßt sich aber bei mehreren Konsolen, z.B. $6 \cdot 2.359.296 \approx 13,5\,\text{MB}$, nicht mehr rechtfertigen, zumal je Konsole ein weiterer gleichgroßer Speicherbereich als virtueller Framebuffer benötigt wird. So muß nach einer anderen, wirtschaftlichen Lösung gesucht werden.

Idee 1 Der benötigte IPC-Pufferspeicher läßt sich sehr einfach begrenzen, indem man die Clients zwingt, beim Öffnen einer virtuellen Konsole selbst anzugeben, wie groß ihre Nachrichten maximal werden, und gleichzeitig eine Maximalgröße für den Puffer je Konsole festlegen. So kann man eine Obergrenze für die Speicherbenutzung definieren und im optimalen Fall (bei kooperativen Clients) noch weit unter dieser bleiben.

Wichtig sind für die Kalibrierung einer solchen maximalen Nachrichtengröße Messungen am laufenden System (siehe Kapitel 5), um Aussagen über die Leistung treffen zu können.

Idee 2 Neben der Möglichkeit große Nachrichten in mehrere kleine zu zerlegen, um die Speicheranforderungen der Konsole in einem vernünftigen Rahmen zu halten, gibt es auch eine andere unter Nutzung des DROPS Streaming Interface (siehe Abschnitt 2.5). In diesem Fall wird der Speicher nicht auf Client- und Serverseite benötigt, sondern gemeinsam genutzt.

Die zu einem Konsolenkommando gehörigen Daten werden hierbei in ein DSI-Paket kodiert und mit Hilfe der Bibliothek abgesendet. Im Normalfall sollte ein Paket neben anderen Parametern, z. B. Zielkoordinaten, also *einen* Datendeskriptor für einen Quellframe beinhalten (pSLIM SET und CSCS).

Es läßt sich aufgrund der Scatter-Gather-Eigenschaft des DSI auch leicht realisieren, nur einen Teilframe aus einem kompletten, im Speicher vorhandenen Frame als Parameter zu übergeben. Dazu muß man nur pro Zeile einen Datendeskriptor verwenden und die korrekten Zielkoordinaten angeben.

Diese Lösung würde nicht nur besser in die angestrebte DROPS-Architektur mit Datenströmen passen, sondern auch die unnötige Kopieroperation während der IPC einsparen. Es ist also lohnend eine Schnittstelle zu entwerfen, die mit dem DSI harmoniert und den Hauptendpunkt der meisten Ausgaben — Grafikkarte, Konsole — sauber an das Ende eines DSI-Stromes anhängt.

Im Rahmen dieses Beleges wird nur *Idee 1* realisiert werden. Eine Anpassung an das DSI sollte aber einfach möglich sein, wenn dessen Entwicklungsstatus ausreichend fortgeschritten ist.

3.5 L4Linux und die Konsole

Zur Integration der Konsole mit L4Linux war eine genauere Betrachtung des Linuxkerns unumgänglich. Dabei war zu erkennen, daß dieser streng modular aufgebaut ist und sich somit das Ersetzen einzelner Bestandteile recht einfach darstellt.

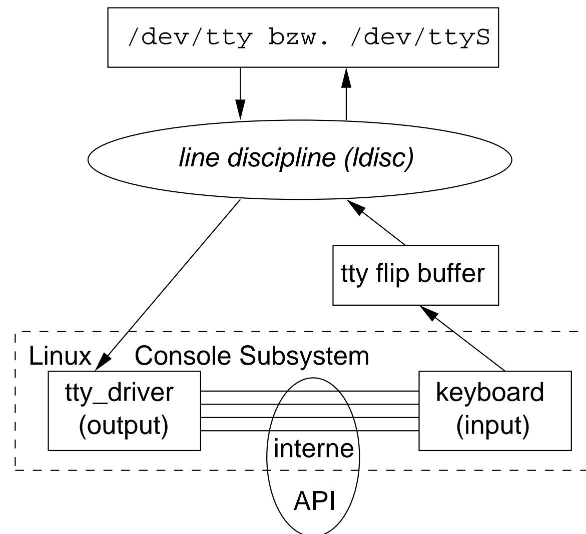

Abbildung 3.7: Struktur der Linux-Konsole

Die Linuxkonsole hat die in Abb. 3.7 dargestellte Struktur. Es ist zu erkennen, daß die Konsole eng mit dem TTY-System verbunden ist. Die Tastaturtreiber-Komponente ist durch einen Stub zu ersetzen, welcher aus einem Tastaturthread besteht, der per IPC verschickte Tastaturereignisse entgegennimmt und in den *flip buffer* schreibt.

Bei der Ausgabekomponente (`tty_driver`) mußte das Framebuffer-Subsystem näher analysiert werden (Abb. 3.8), da vorausgesetzt wird, daß L4Linux einen grafischen Ausgabetreiber verwendet. Bei einem Framebuffer-Linux übernimmt das *Framebuffer Subsystem* die Kontrolle über die Ausgabefunktionen der Linux-Konsole. Scrollen, Textrendering und Konsolenumschaltung werden vollständig vom `fbcon`-Modul durchgeführt.

In Abb. 3.8 ist zu erkennen, daß die Treiberkomponente des Framebuffer Subsystems eindeutig vom Rest des Systems getrennt ist. Dieses `fbdev`-Modul ist durch einen Stub zu ersetzen, der die Framebuffertreiber-Schnittstelle auf L4-IPC und unser Protokoll abbildet und so die DROPS-Konsole nutzt.

Daraus ergibt sich eine Architektur, die alle virtuellen L4Linux-Konsolen auf eine DROPS-Konsole „multiplext“. Das System stellt sich dem Nutzer geschichtet dar, so daß auf der ersten Ebene die virtuellen DROPS-Konsolen und auf der zweiten Ebene — also in einer virtuellen DROPS-Konsole — die L4Linux-Konsolen benutzt werden können.

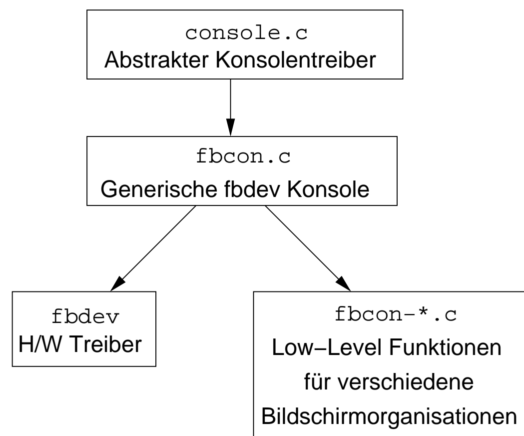

Abbildung 3.8: Ausgabekomponente (`tty_driver`) der Linux-Konsole

Kapitel 4

Implementierung

4.1 Treiberkomponenten

Aufgrund von Problemen bei der Portierung der geplanten Treibersoftware, die aus der aktuellen Linuxversion resultieren, wurde eine Übergangslösung geschaffen. Es wird nun als Grundlage die in [2] entwickelte Software verwendet, die eingeschränktes Testen erlaubt.

Von der DROPS-Konsole werden Funktionen zum Initialsieren der Grafikkarte und Setzen des Grafikmodus der Timesharing-Schnittstelle der EZDK verwendet. Während des Initialisierungsvorganges stellt die DROPS-Konsole eine Anfrage zum Einblenden des Grafikkartenspeichers an die EZDK und es ist so möglich diesen zu modifizieren. Weiterhin wird von einer rudimentären Funktionalität zum Abfragen von Eingabeereignissen im Grafikmodus Gebrauch gemacht.

Wenn die Probleme der Treiberportierung behoben sind, sollte ein Austausch der Aurufe der EZDK-Bibliotheksfunktionen durch Aufrufe der API-Funktionen des Treibers einfach durchzuführen sein. Größere Modifikationen sind nur an der Implementierung des Eingabemoduls (`ev.c`) vorzunehmen.

4.2 Virtuelle Konsolen

Die Managementfunktionalität der Konsole teilt sich in zwei Bereiche — single- und multithreaded. Der erste Teil (`main.c`) ist der Wirkungsbereich des *Master-Threads*, welcher Anfragen zum Öffnen entgegennimmt und die Umschaltung zwischen den Konsolen durchführt. Dieser Thread startet bei Bedarf *Service-Threads*, die als Kommunikationspartner für Konsolen-Clients dienen.

Wichtig ist insbesondere eine Variable (`want_vc`), die ggf. die Nummer einer gewünschten virtuellen Konsole beinhaltet (Umschaltung), und eine Sperre — *lock* — für diese. Zugriffe auf `want_vc` müssen unter gegenseitigem Ausschluß stattfinden, da verschiedene Threads auf diese Variable zugreifen. An dieser Stelle wird von einem einfachen Lock-Mechanismus

(`l4_simple_lock`) Gebrauch gemacht. Außerdem verwaltet der Master-Thread auch eine Liste der virtuellen Konsolen.

Der zweite Managementteil (`vc.v`) wird nebenläufig von verschiedenen Service-Threads bearbeitet. Eine virtuelle Konsole (`l4con_vc`) repräsentiert somit ein dedizierter Service-Thread und eine zugehörige Verwaltungsstruktur (siehe `con.h`). Neben anderen Daten enthält diese Struktur auch Referenzen auf allokierte Speicherbereiche für den Stack des Threads (`stack`) und den virtuellen Framebuffer (`vfb`).

Ein sehr wichtiger Bestandteil ist der aktuelle Framebuffer-Zeiger (`fb`) und eine zugehörige Lock-Variable. Dieser Zeiger beinhaltet entweder eine Referenz auf den virtuellen oder den eingeblendeten Framebuffer. Die Lock-Variable wiederum schützt kritische Bereiche im Zusammenhang mit Framebuffern. So ist es nicht möglich, eine Umschaltung zwischen verschiedenen Konsolen durchzuführen, wenn der Service-Thread eine Grafikfunktion ausführt und umgekehrt.

4.3 Konsolenprotokoll

Die Schnittstelle zur DROPS-Konsole kann nicht durch eine einfache API realisiert werden, da Konsole und Clients in unterschiedlichen Adreßräumen ablaufen. Es werden Mechanismen zur Interprozeßkommunikation (IPC) benötigt, die auch über die Adreßraumgrenzen hinaus funktionieren. Der DROPS-Mikrokern unterstützt diese Mechanismen zur Kommunikation zwischen Threads. Hierbei ist es nicht relevant, ob diese in einem oder in unterschiedlichen Adreßräumen ablaufen.

Hier wird ein Konsolenprotokoll definiert, das:

1. die gesamte Funktionalität der Konsole bereit stellt,
2. den Kommunikationsmechanismus des Mikrokerns sehr gut ausnutzt, und
3. eine transparente Verbergung der Adreßraumgrenzen durch geeignete Bibliotheken ermöglicht.

In Anlehnung an das Protokoll des *RMGR* werden geeignete Datenstrukturen verwendet, die alle für die Kommunikation nötigen Informationen kapseln. Jede Nachricht des Konsolenprotokolls beinhaltet ein Protokolltypfeld, um sie von Nachrichten anderer DROPS-Komponenten unterscheiden zu können. Die Protokolltyp-Nummer ist `0x41` (das ASCII-Zeichen `C`).

Die spezielle Gruppe ECODE (Tab. 4.1) definiert alle möglichen Ausgabewerte für den Fall, daß Fehlerzustände eintreten. So sind die Clients selbst in der Lage, Ausnahmesituationen zu erkennen und zu behandeln.

Neben den Filter-Funktionen gehört in die Gruppe EV — Event — natürlich auch die Kodierung der Eingabe-Ereignisse. Hier sei auf die zum Konsolen-Paket gehörige Datei `linux/input.h` verwiesen, die alle Ereignisdefinitionen des *Input Device Project* beinhaltet.

Gruppe	Kommando	Parameter		Bedeutung
		Eingabe	Ausgabe	
STD	OPENQRY	-	thread	Anfrage zum Öffnen einer virtuellen Konsole
	SMODE	mode	-	Setzen des Modus der Konsole bzgl. Ein/Ausgabe
	GMODE	-	mode	Erfragen des aktuellen Konsolenmodus
	CLOSE	-	-	Schließen einer virtuellen Konsole
pSLIM	SET	rect, values[]	-	SLIM-konform
	BMAP	rect, bits[], fgc/bgc, mode	-	SLIM-konform
	FILL	rect, value	-	SLIM-konform
	COPY	rect, dest	-	SLIM-konform
	CSCS	rect, type, scale, yuv[][]	-	SLIM-konform
GRAPH	SMODE	res, bpp	-	Setzen des Grafikmodus (Auflösung und Farbtiefe)
	GMODE	-	res, bpp	Erfragen des aktuellen Grafikmodus
EV	SFLT	filter	-	Setzen des Ereignisfilters der Eingabe
	GFLT	-	filter	Erfragen des Ereignisfilters der Eingabe
ECODE			EOK	kein Fehler
			...	...

Tabelle 4.1: Konsolen-Protokoll

4.4 Umsetzung der SLIM-Funktionen

Die in der IPC-Schnittstelle angebotenen Funktionen werden vom pSLIM-Modul (`pslim.c`)[1] der Konsole implementiert. In diesem Modul müssen Implementierungen der Funktionen für alle Organisationsmodi des Framebuffers in Bezug auf die Farbtiefe enthalten sein. Für die anderen Module ist das aber transparent.

Ein weiterer wichtiger Punkt ist, daß das pSLIM-Modul multithread-fähig sein muß, da es möglich ist, das zu einem Zeitpunkt mehrere virtuelle Konsolen grafische Funktionen aufrufen, um entweder den Framebuffer der Grafikkarte — Vordergrundkonsole — oder einen Speicherbereich zu modifizieren. Insbesondere globale Variablen wurden an dieser Stelle vermieden.

Bei Aufruf einer pSLIM-Funktion wird stets eine Referenz auf die Verwaltungsstruktur der virtuellen Konsole als Parameter übergeben, so daß innerhalb der Funktion alle aktuellen Informationen zu Framebuffergeometrie und -organisation zur Verfügung stehen. Die weiteren Parameter sind kommandospezifisch.

Die primäre Aufgabe der generischen Funktion ist die Anpassung der Parameter an die

[1] Das grafische Protokoll der DROPS-Konsole wird nicht mit SLIM bezeichnet, da wir keine vollständige Kompatibilität aus den vorn genannten Gründen erzielen konnten. Es ist nur Pseudo-SLIM (pSLIM).

Geometrie des Framebuffers. Das erlaubt einerseits eine einfache Implementierung der eigentlichen, modusspezifischen Funktionen ohne Zugriffe auf nicht erlaubte Speicherbereiche, z. B. bei negativen Koordinaten, und andererseits eine Optimierung, da immer nur sichtbare Bereiche bearbeitet werden.

Die spezifischen Grafikfunktionen wurden anhand der SVGAlib-Implementierung[2] ausgeführt und sind allgemein ausgedrückt Iterationen über dem interessanten rechteckigen Bereich des Framebuffers. Es wäre an dieser Stelle denkbar, Beschleunigerfunktionen der Grafikkarte für die Vordergrundkonsole zu verwenden, da die pSLIM-Funktionen meist 1:1 auf diese abgebildet werden können. Das ist aber abhängig von der Hardware sowie der verwendeten Treibersoftware und in der aktuellen Implementierung nicht enthalten.

[2] Die SVGAlib ist eine Bibliothek, die Funktionen zum direkten Zugriff auf Grafikkarten enthält.

Kapitel 5

Leistungsbewertung

Für die Leistungsbewertung der Konsolenkomponente sind vor allem die Ausführungszeiten der einzelnen Befehle interessant. Neben diesem Aspekt werde ich im folgenden auch auf die in Abschnitt 3.4 angesprochene Optimierung der Speicherausnutzung eingehen.

5.1 Zeitverhalten der einzelnen pSLIM-Befehle

Die Kosten für eine Darstellung auf einer virtuellen Konsole wurden mit einem Testprogramm unter Nutzung des *Time-Stamp Counters (TSC)* des Pentiumprozessors vorgenommen. Die Hardwareplattform bildeten ein Pentium 200 MMX mit 64 MB RAM und miroVideo Grafikkarte (S3 968 Chip). Die Bildschirmauflösung war 640x480 Bildpunkte bei 16 bit Farbtiefe.

Gemessen wurde die Verzögerungszeit für die Ausführung eines pSLIM-Befehls vom Beginn des Marshalling bis zur Rückkehr aus dem IPC-Call. Je Befehl wurden verschiedene Pixelanzahlen nacheinander getestet und es wurden jeweils mehrere Messungen pro Anzahl für einen Endwert gemittelt.

In Abb. 5.1 ist zu erkennen, daß die Verzögerungszeit mit linear wachsender Pixelanzahl wie erwartet bei allen pSLIM-Kommandos linear ansteigt. Dabei sind die Befehle CSCS (Videodaten) und COPY (Kopieren *innerhalb* des virtuellen Framebuffers) wesentlich teurer als die verbleibenden.

Im Falle von CSCS ist dies auf die Berechnungskosten für die Farbkonvertierung zurückzuführen, die es verlangt die Intensität für jeden RGB-Farbkanal zu berechnen und diese Werte anschließend in einen Pixelwert zu packen.

Vergleicht man die Kosten für das Kopieren mit COPY für Vorder- und Hintergrundkonsole (sichtbar bzw. nicht sichtbar), ist die Leistung im nicht sichtbaren Fall bedeutend besser (siehe auch Tab. 5.1). Dieses Verhalten gründet mit hoher Wahrscheinlichtkeit auf der Tatsache, daß Leseoperationen auf dem gemappten Framebuffer der Grafikkarte *bedeutend* langsamer sind als auf Hauptspeicherbereichen.

Zusammenfassend ist also zu sagen, daß die Leistung der Konsole zum derzeitigen Zeitpunkt für die Zukunft als nicht zufriedenstellend zu bewerten ist. So benötigt eine komplettes Bild-

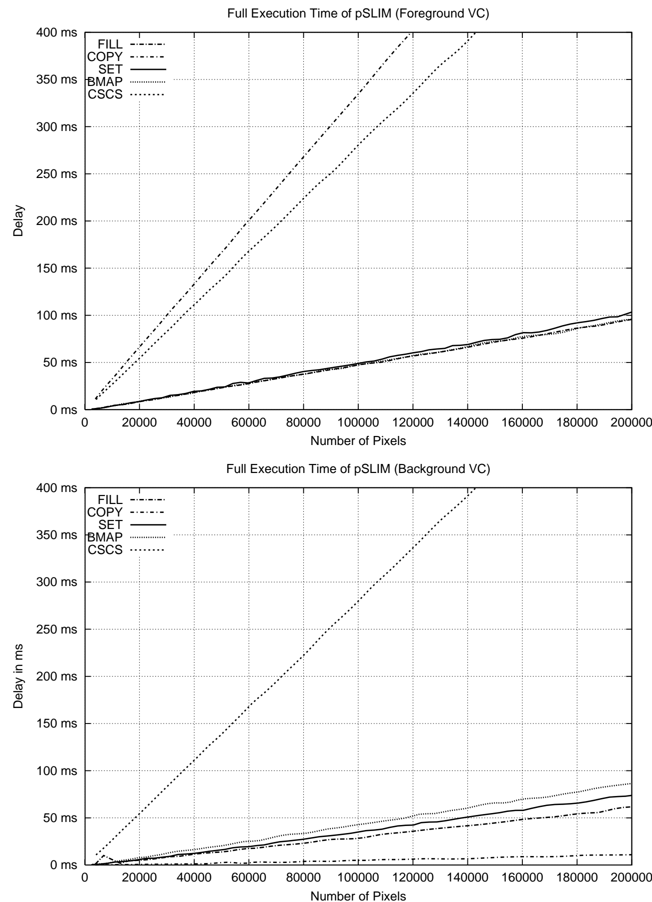

Abbildung 5.1: Bearbeitungszeiten für pSLIM-Befehle (Vorder- und Hintergrundkonsole)

schirmupdate (pSLIM SET für den gesamten Framebuffer) bei den genannten Testbedingungen im Mittel 150 ms.

Hierbei ist es nötig genaue Messungen innerhalb der Konsole vorzunehmen, um die Anteile der Kosten für IPC, Protokollverarbeitung und Zeichenoperationen unabhängig voneinander betrachten zu können.

5.2 Ansatz: Maximale Nachrichtenlänge

Um das Zeitverhalten bei Aufteilung einer großen IPC-Nachricht in mehrere kleine zu testen, wurde ein weiteres Testprogramm entwickelt. Die Testreihe wurde unter den gleichen Bedingungen wie oben für den pSLIM-Befehl SET durchgeführt. In jedem Fall wurden 640x480 16-Bit-Pixel bei unterschiedlicher Nachrichtenanzahl als *indirect strings* übertragen.

Abb. 5.2 zeigt, daß eine verteilte Übertragung keine Performanceeinbußen mit sich bringt, solange bestimmte Randbedingungen beachtet werden. Die wichtigste dieser Bedingungen ist, daß der IPC-Overhead aufgrund großer Anzahl Nachrichten (>700) nicht dominanter

pSLIM-Befehl	Kosten pro Pixel	
	Vordergrund	Hintergrund
FILL	480 ns	367 ns
COPY	3.356 ns	55 ns
SET	518 ns	367 ns
BMAP	482 ns	446 ns
CSCS	2.803 ns	2.813 ns

Tabelle 5.1: Kosten der pSLIM-Befehle

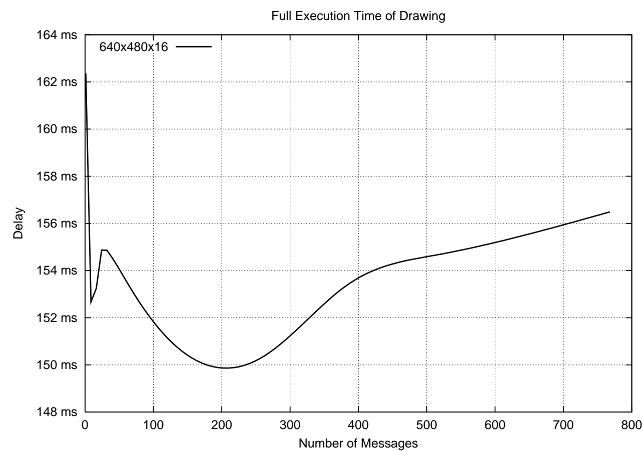

Abbildung 5.2: foo2

Bestandteil der Verzögerungszeit wird.

Es ist außerdem anzumerken, daß die Schwankungen der Zeiten bei weniger als 700 Nachrichten in einem sehr schmalen Bereich stattfinden (6 bis 8 ms). Das bedeutet, daß eine Lösung mit begrenztem Empfangspuffer für die Konsole durchaus interessant und durchführbar ist.

Kapitel 6

Zusammenfassung und Ausblick

Im Rahmen dieses Beleges wurde eine Konsolenkomponente für DROPS entworfen und implementiert. Die Konsole bietet den Systemkomponenten eine wohl definierte Schnittstelle auf grafischer Basis und erlaubt das vom Nutzer gesteuerte Umschalten zwischen mehreren virtuellen Konsolen.

Erweiterungen der Konsole sind in einigen Bereichen noch möglich. So sollten die anfangs vorgestellten Grafik- und Eingabetreiber vollständig portiert werden. Auch eine Anpassung der Komponente an das DROPSStreaming Interface mit anschließenden Leistungsmessungen würde eine interessante Aufgabe für die Zukunft darstellen.

Anhang A

Glossar

Clipping Abschneiden von nicht sichtbaren Bereichen.

Framebuffer ein Bereich des Speichers der Daten beinhaltet. Meist der Speicher einer Grafikkarte auf dem Adapter von dem das Monitorbild aufgefrischt wird.

IPC (InterProcess Communication; engl. Interprozeßkommunikation) Mechanismus des Datenaustausches zwischen verschiedenen Programmen (in unterschiedlichen Adreßräumen)

Multihead terminalähnlicher Zugriff auf einen Computer mit beliebiger Anzahl von E/A-Geräten und logischen Verknüpfungen dieser zu *Heads*.

QoS (Quality of Service) Die Möglichkeit Leistungsangaben in Datenkommunikationssystemen zu treffen.

RGB (Red-Green-Blue; engl. Rot-Grün-Blau) Farbmodell bei dem die Farbe eines Punktes aus drei Werten zusammengestzt wird, der Rot-, Grün- und Blauintensität.

Terminal Ein/Ausgabegerät für Computer mit Tastatur und Monitor

YUV Farbmodell, bei dem die Farbe eines Punktes aus drei Werten zusammengesetzt wird, Luminanz (Y) und Chrominanz (U,V).

Literaturverzeichnis

[1] Robert Baumgartl, Martin Borriss, Hermann Härtig, Claude-Joachim Hamann, Michael Hohmuth, Lars Reuther, Sebastian Schönberg, and Jean Wolter. Dresden Realtime Operating System. In *Proceedings of the First Workshop on System Design Automation (SDA'98*, pages 205–212, Dresden, March 1998.

[2] Torsten Paul. Entkopplung von Echtzeit- und Timesharing-Aktivitäten am Beispiel einer echtzeitfähigen Darstellungskomponente in Drops. Diplomarbeit, TU Dresden, September 1998.

[3] *GNU General Public License.* Free Software Foundation, Inc., http://www.fsf.org, June 1991. Version 2.

[4] Geert Uytterhoeven. *The Linux Frame Buffer Device Subsystem.* http://www.linux-fbdev.org, 1999. Linux Expo.

[5] *XFree86.* The XFree86 Project, Inc., http://www.xfree86.org, 1994-2000.

[6] Andreas Beck and Steffen Seeger. *The General Graphics Interface.* http://www.ggi-project.org.

[7] Steffen Seeger. *The Kernel Graphics Interface.* http://kgi.sourceforge.net.

[8] Vojtech Pavlik. *The Linux Input Drivers Project.* http://www.suse.cz/development/input/.

[9] Brian K. Schmidt, Monica S. Lam, and J. Duane Northcutt. The interactive performance of SLIM: a stateless, thin-client architecture. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, volume 34, pages 32–47, December 1999.

[10] Lars Reuther, Jork Löser, and Lukas Grützmacher. DSI — Drops Streaming Interface. TU Dresden, 2000.

[11] René Stange. Systematische Übertragung von Gerätetreibern von einem monolithischen Betriebssystem auf eine mikrokernbasierte Architektur. Master's thesis, TU Dresden, May 1996. In German. Available from URL: `http://os.inf.tu-dresden.de/L4/`.