Diplomarbeit

zum Thema

Entkopplung von Echtzeit- und Timesharing-Aktivitäten am Beispiel einer echtzeitfähigen Darstellungskomponente in DROPS

an der

Technischen Universität Dresden

Fakultät Informatik

Institut für Betriebssysteme, Datenbanken und Rechnernetze

Lehrstuhl für Betriebssysteme

Eingereicht von: Torsten Paul

Geboren am: 23. März 1974

Geboren in: Dresden

Matrikel-Nummer: 2213116

Eingereicht am: 21. September 1998

Betreuender Hochschullehrer:

Prof. Dr. H. Härtig

Selbständigkeitserklärung

Hiermit erkläre ich, daß ich diese Arbeit selbständig und nur mit den zugelassenen und aufgeführten Hilfsmitteln erstellt habe.

Dresden, den 21. September 1998

Torsten Paul

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Die Übertragung von *Continuous-Media*-Strömen über Netzwerke und die Speicherung dieser Ströme auf Medien-Servern ist bereits in einer Anzahl von Projekten untersucht worden. Beispiele dafür sind der *Tiger Video Fileserver* [B+96, WJB97], das *UCB Distributed Continuous Media Filesystem* (CMFS) [AOG91] sowie die Forschungen an der Boston University, Massachusetts [CLV95, CL94] zur effizienten Speicherung von Video-Daten.

Die Unterstützung von *Continuous-Media*-Anwendungen auf dem Endsystem ist trotz dieser umfangreichen Untersuchungen im Bereich *Continuous-Media* nur bei wenigen Systemen vorhanden. In vielen Fällen ist das Endsystem auf Nutzerseite ein normales Timesharing-System, wodurch die Vorteile der *Quality of Service* Garantien für die Datenübertragung an dieser Stelle verloren gehen [JR98].

Da nur wenige Anwendungen auf einem Endsystem wirklich Echtzeit-Anforderungen an das System stellen, ist es sinnvoll, ein Betriebssystem zu entwickeln, welches einerseits die gewohnte Arbeitsumgebung eines Timesharing-Systems (z.B. UNIX mit *X-Window-System*) bereitstellt, andererseits aber auch die Nutzung von Multimedia-Applikationen mit *QoS*-Anforderungen (z.B. Videokonferenzsysteme) erlaubt und diese möglichst transparent für den Benutzer in die Arbeitsumgebung integriert.

Aus diesem Grund konzentriert sich die Forschung am Lehrstuhl Betriebssysteme der TU Dresden auf ein echtzeitfähiges Betriebssystem. Um eine hohe Flexibilität bei der Entwicklung zu ermöglichen, kommt der μ-Kern L4 zum Einsatz, der die Basis für dieses Echtzeit-System bildet.

Diese Arbeit beschäftigt sich mit dem Entwurf und der Implementierung einer Komponente für dieses Echtzeit-System.

1.1 Über diese Arbeit

Die Arbeit ist in sechs Kapitel gegliedert. Kapitel 2 stellt aktuelle Entwicklungen im Bereich Grafik vor, die die Grundlage für diese Arbeit bilden. Den Schwerpunkt bildet dabei das GGI-Projekt.

In Kapitel 3 werden mögliche Modelle für das Design einer Darstellungskomponente diskutiert. Nach Auswahl eines Modells wird dieses näher untersucht und seine Eigenschaften dargestellt. Weiterhin wird eine allgemeine Schnittstelle für die Kommunikation zwischen Echtzeit-Komponenten entworfen, die es ermöglicht, mehrere Komponenten zu einer Kette zusammenzufügen. Innerhalb dieser Kette übernimmt jede Komponente einen einfachen Verarbeitungsschritt für einen Datenstrom.

Im nächsten Kapitel wird die Implementierung des Modells aufgezeigt, indem dargestellt wird, wie vorhandene Software-Module kombiniert und an die Zielumgebung angepaßt wurden.

Kapitel 5 untersucht, ob die implementierte Darstellungskomponente die an sie gestellten Forderungen erfüllt. Dafür wurden umfangreiche Messungen am laufenden System vorgenommen.

Das 6. Kapitel faßt schließlich die während dieser Arbeit gewonnenen Erkenntnisse zusammen, gibt einen Überblick über den Stand der Implementierung der Darstellungskomponente und bietet einen Ausblick auf mögliche weitergehende Entwicklungen.

1.2 Danksagung

Bedanken möchte ich mich an dieser Stelle ganz herzlich bei der gesamten Gruppe Betriebssysteme der TU Dresden. Insbesondere möchte ich mich bei meinem Betreuer Prof. Härtig und bei Jean Wolter bedanken, die immer bereit waren zuzuhören, wenn Probleme bei der Entstehung dieser Arbeit auftraten und mich bei Entwurf und Implementierung unterstützten.

Kapitel 2

Stand der Technik

2.1 DROPS

Ziel der Forschung und Entwicklung am Lehrstuhl Betriebssysteme der TU Dresden ist es, ein Betriebssystem mit integrierter Unterstützung für *QoS*-Garantien für die vom System verwalteten Ressourcen zu schaffen: DROPS, das Dresden Real Time Operation System.

Die beiden Schwerpunkte bei der Entwicklung von DROPS sind Design und Implementierung eines μ-Kern-basierten Client-Server-Betriebssystems mit *QoS*-Unterstützung sowie die Anwendung eines allgemeinen mathematischen Modells zur Beschreibung des Schedulings der Echtzeit-Komponenten [Ham97].

2.1.1 Architektur

Basis des Systems ist der von Jochen Liedke an der GMD entwickelte μ-Kern L4 [Lie96]. Durch die extrem effiziente Implementierung von L4 [Lie95] ist es möglich, ein echtes Client-Server System zu entwickeln, ohne Kompromisse aufgrund langsamer Interprozeß-Kommunikation in Kauf nehmen zu müssen. Weiterhin ist durch entsprechende Mechanismen sowohl die Speicherverwaltung auf Basis von *external pagern* als auch die Implementierung von Scheduling-Strategien auf Nutzerebene möglich.

Die Bereitstellung einer UNIX-Umgebung erfolgt bei DROPS durch eine Portierung des frei verfügbaren, UNIX-ähnlichen Betriebssystemkernes *Linux* [Lin] auf den μ-Kern L4 [Hoh96]. Diese Umgebung stellt auch die Plattform für die Implementierung von Echtzeit-Komponenten dar.

Aktuelle Arbeiten dienen der Entwicklung einer Reihe von Echtzeit-Komponenten für DROPS, darunter

- eine echtzeitfähige Netzwerk-Komponente auf Basis von ATM-Technologie [Dan97, BH98b],

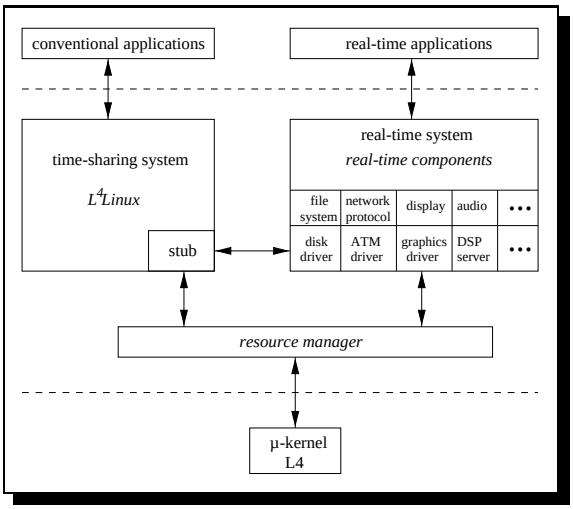


Abbildung 2.1: DROPS System-Architektur (aus [BBH+98])

- ein Dateisystem für *Continuous-Media*-Daten mit *QoS*-Anforderungen sowie Unterstützung für traditionelle Datenspeicherung [Meh98, Reu98, Rud98] und
- ein DSP-basiertes Coprozessor-System für die Auslagerung von rechenintensiven Algorithmen auf Spezialprozessoren [BH96, BH97, BH98a].

Abbildung 2.1 zeigt einen allgemeinen Überblick über die geplante und bereits teilweise realisierte Architektur von DROPS und macht die Integration von Echtzeit- und Timesharing-System deutlich. Insbesondere soll der Einsatz von Stub-Treibern in L4Linux die Nutzung von Gerätetreibern des Echtzeit-Systems ermöglichen, wenn diese einen nicht echtzeitfähigen Zugriff erlauben. Beispiele dafür sind das Dateisystem und die Darstellungskomponente, die im Rahmen dieser Arbeit entstehen soll.

2.1.2 Verwandte Projekte

Es existieren einige Projekte, die analog zu DROPS eine Unterstützung von Multimedia-Applikationen auf Betriebssystemebene bieten wollen. Die Umsetzung dieser Zielstellung unterscheidet sich jedoch in einigen Punkten. Insbesondere ist DROPS als umfangreiches und flexibles System mit verschiedenen Komponenten für spezielle Datenströme wie Audio, Video oder auch Meßdaten von Sensoren geplant.

Der Rialto-μ-Kern ist für einen relativ eingegrenzten Einsatzzweck gedacht und bietet einige interessante Ansatzpunkte, die in anderen Echtzeit-Kernen nicht vorhanden sind.

RT-Mach unterstützt allgemeine Funktionen für die Reservierung von Ressourcen wie CPU [MST93] und Speicher [TNR90] im Systemkern, die in

einer Reihe von Projekten untersucht und für Applikationen genutzt werden [NKT93, LRM96]. Es existiert aber kein Betriebssystem, welches die von RT-Mach angebotenen Möglichkeiten nutzt. Insbesondere verwendet der auf Mach portierte UNIX-Server [GDFR90] die Reservierungsfunktionen nicht.

Rialto

Der von Microsoft Research entwickelte Echtzeit-Kern Rialto wurde mit ähnlichen Zielen entwickelt wie DROPS, darunter die gleichzeitige Unterstützung von Echtzeit- und Nichtechtzeit-Prozessen, dynamische Reservierung von Ressourcen sowie gemeinsame Nutzung von Hardware durch beide Prozeßtypen [JRR97].

Zum Einsatz kommt das System allerdings nur als Basis für Set-Top Boxen für Microsoft-Interactive-TV-Projekte in Yokosuka, Japan. Für andere Projekte von Microsoft Research mit Echtzeit-Anforderungen wie den Tiger Video Fileserver [B+96] wird jedoch Windows NT eingesetzt.

Interessant ist eine dem Scheduling-Framework in DROPS ähnliche Definition von Programmabschnitten, die mit definierten Zeitschranken ausgeführt werden sollen. In Rialto werden diese Zeitschranken für ein spezielles Programmstück *Constraint* genannt und der entsprechende Teil des Programmes durch `BeginConstraint()` und `EndConstraint()` eingeschlossen. Im Unterschied zu DROPS erfolgt jedoch keine Vorausreservierung, sondern es wird bei jedem Aufruf von `BeginConstraint()` eine erneute Reservierung vorgenommen. Dadurch ist das Verhalten des Systems im Überlastfall nicht mehr vorhersagbar [JBIF+96]:

> „Of course, time constraints do not in any way guarantee a feasible schedule, and in fact, may behave very badly in overload conditions. Thus, higher level mechanisms are also needed to solve resource contention issues."

Eine Besonderheit des Schedulers in Rialto besteht darin, daß der Zeitgeber, der das Scheduling steuert, nicht periodisch arbeitet. Das ermöglicht eine extrem flexible Definition von Unterbrechungszeitpunkten für Prozesse, da diese Zeitpunkte nicht an den Grenzen einer fest vorgegebenen Periode liegen müssen.

Real-Time Mach

Das Ziel bei der Entwicklung von Real-Time Mach (RT-Mach) war es, eine Echtzeit-Version von Mach zu implementieren, die eine vorhersagbare Echtzeit-Umgebung bereitstellt. Zusätzlich sollen auch eine Reihe von Hilfsprogrammen für die Nutzung dieser Umgebung angeboten werden.

RT-Mach bietet vier Abstraktionen:

1. ein Echtzeit-Threadmodell,

2. einen Echtzeit-Scheduler, basierend auf einem modifizierten Ratenmonotonen Scheduler,

3. Mechanismen zur Synchronisation und Interprozeß-Kommunikation mit Verhinderung von Prioritätsumkehr und

4. speicherresidente Objekte, um Zugriffe auf den Speicher vorhersagbar zu machen.

Reservierungen erfolgen in RT-Mach über Attribute, die nach dem Erzeugen von Threads für diese gesetzt werden können. Unterschieden werden dabei periodische und aperiodische Prozesse. Periodische Prozesse sind durch Startzeit, maximale Ausführungszeit, Periode und Verschiebung innerhalb der Periode definiert, aperiodische Prozesse dagegen durch maximale Ausführungszeit, maximale Abstände der Ankunftszeitpunkte und ihre Zeitschranke. Diese Attribute dienen u.a. der Reservierung von Prozessorkapazität [MST93]. Da Attribute für Threads zu jeder Zeit gesetzt werden können, ist in RT-Mach eine dynamische Anpassung von Reservierungen problemlos möglich und kann vom Nutzer durch interaktive Programme durchgeführt werden [MR94] bzw. über *QoS*-Manager automatisch erfolgen [LRM96].

2.2 Linux

Linux ist eine frei verfügbare UNIX-Implementierung, von der Portierungen auf verschiedenste Systeme existieren, so z.B. für Rechner mit Intel-x86, Sun-Sparc, Motorola-68k und Digital-Alpha CPUs. Der Ausgangspunkt dafür war ein Hobby-Projekt eines finnischen Studenten, Linus Torvalds, woraus unter Mitwirkung einer ganzen Reihe von Helfern in kurzer Zeit ein kompletter Betriebssystemkern entstand. Heute arbeiten eine Anzahl von Teams weltweit in ihrer Freizeit an den verschiedenen Portierungen. Linus Torvalds koordiniert weiterhin die Arbeiten und trägt zur Weiterentwicklung des Linux-Kernes bei.

Da für diese Arbeit die in Linux implementierten Gerätetreiber und dabei speziell die *TTY*-Treiber von Interesse sind, sollen diese hier etwas näher beschrieben werden.

2.2.1 Gerätetreiber

Eine der wichtigsten Aufgaben eines Betriebssystems ist es, die speziellen Eigenschaften unterschiedlicher Hardware vor dem Nutzer des Systems so weit wie möglich zu verbergen.

In UNIX wird das im allgemeinen dadurch erreicht, daß die Behandlung von Geräten abstrahiert und eine einheitliche Schnittstelle vom System angeboten wird. Alle Geräte sehen daher vom Standpunkt des Nutzers wie normale Dateien aus. Sie können geöffnet, geschlossen, gelesen und geschrieben werden. Dabei kommen die gleichen Systemrufe zum Einsatz wie auch für reguläre Dateien.

Innerhalb des Systems werden diese Operationen in gerätespezifische Aufträge umgesetzt und an die entsprechende Hardware gesandt.

Linux unterstützt drei Typen von Geräten: *Character*-, *Block*- und *Netzwerk*-Geräte. *Character*-Geräte (z.B. die serielle Schnittstelle `/dev/ttyS0`) werden ohne Pufferung direkt gelesen und geschrieben. *Block*-Geräte werden nur mit Vielfachen ihrer Blockgröße beschrieben, die typischerweise bei 512 Byte liegt. Der Zugriff erfolgt dabei immer über den *buffer cache*. In den meisten Fällen wird auf diese Geräte jedoch nicht direkt über die Geräteschnittstelle (z.B. `/dev/sda` für die erste Festplatte am SCSI-Bus) zugegriffen, sondern indirekt über ein Dateisystem. *Netzwerk*-Geräte werden über die BSD-Socket-Schnittstelle angesprochen.

Character-Geräte sind die einfachste Art von Geräten. Der Zugriff erfolgt über spezielle Einträge im Dateisystem, wodurch im Kern erkannt werden kann, daß es sich nicht um eine normale Datei, sondern um einen Geräteeintrag handelt. Traditionell befinden sich diese Einträge im `/dev`-Verzeichnis. Der Systemkern hat keinerlei Kenntnis von Lage und Namen der Geräteeinträge. Sie sind nur für Programme wichtig, die auf die entsprechenden Geräte zugreifen wollen. Für den Kern sind lediglich die beim Erzeugen des Geräteeintrages vergebene *Major*- und *Minor*-Nummer von Interesse. Dabei definiert die *Major*-Nummer den Typ des Gerätes und damit auch den Gerätetreiber. In Linux entspricht z.B. die *Major*-Nummer 6 der parallelen Schnittstelle (`/dev/lp0`) und die *Major*-Nummer 14 der Soundkarte (`/dev/dsp`, `/dev/mixer`, ...). Die *Minor*-Nummer unterscheidet innerhalb eines Gerätetreibers zwischen mehreren Geräten des gleichen Typs oder verschiedenen Zugriffsmöglichkeiten bzw. Funktionen eines Gerätes. Die aktuellen Werte für in Linux zugewiesene Gerätenummern können [Anv96] entnommen werden. Weiterführende Informationen zum Linux-Kern enthält [Rus98].

2.2.2 TTY-Subsystem

Die Ein- und Ausgabe von Zeichen auf einem Terminal erfolgt in UNIX durch das *TTY*-Subsystem. Durch die wachsende Anzahl von unterstützten Funktionen ist das *TTY*-Subsystem in aktuellen UNIX-Systemen ein relativ komplexes System.

Das *TTY*-Subsystem in Linux (Abbildung 2.2) ist trotz der Komplexität streng modular aufgebaut, was eine große Flexibilität ermöglicht. *TTYs* sind *Character*-Geräte und werden über einen Eintrag im `/dev`-Verzeichnis angesprochen. In Linux ist den Geräten vom Typ *TTY* die *Major*-Nummer 4 zugeordnet:

```
crw--w--w-   1 tp       tty        4,   0 Jul  6 22:46 /dev/tty0
crw-------   1 root     root       4,   1 Jul  6 22:46 /dev/tty1
crw-------   1 root     root       4,   2 Jul  6 22:46 /dev/tty2
crw-rw----   1 root     uucp       4,  64 Mar  5 00:49 /dev/ttyS0
crw-rw----   1 root     uucp       4,  65 Mar  5 00:49 /dev/ttyS1
```

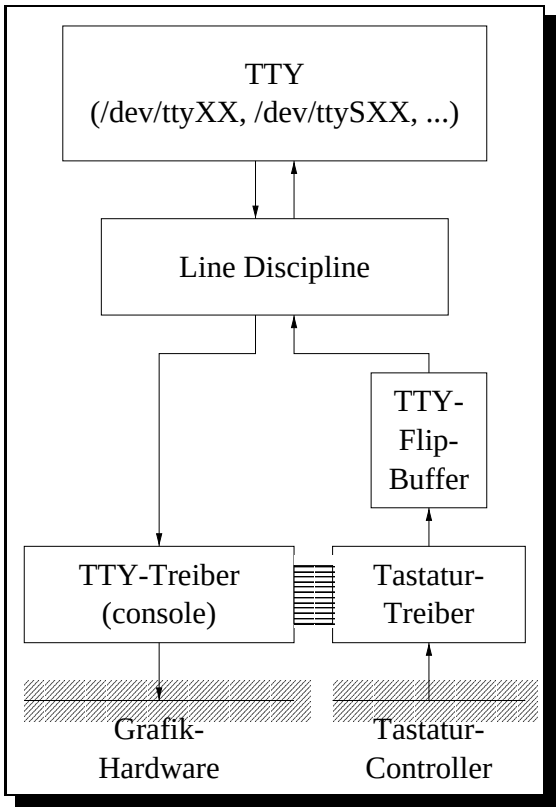


Abbildung 2.2: Architektur des TTY-Subsystems in Linux

Für diese *Major*-Nummer wurden bei Systemstart eigene Funktionen für `open`, `close`, `read`, `write` usw. registriert. Wenn ein *TTY* geöffnet wird, muß im Systemkern eine *TTY*-Struktur (Tabelle 2.1) erzeugt werden. Diese enthält alle für ein *TTY* nötigen Zustandsinformationen, Verweise auf den Low-Level-Treiber (`struct tty_driver`) und die aktuelle *Line Discipline* (`struct tty_ldisc`).

Element	Beschreibung
Pufferspeicher	getrennte Puffer für die asynchrone Ablage von Zeichen und Attributen durch eine Interrupt-Funktion
Hardware-Parameter	Gerätenummer Funktionen für Ausgabe-Start/Stop Funktionen zum Setzen des Hardware Zustandes
Warteschlangen	Warteschlange für lesende Prozesse Warteschlange für schreibende Prozesse
Zustandsinformationen	`termios`-Struktur Prozeß-Gruppe Session-Nummer Zeilen- und Spalten-Anzahl

Tabelle 2.1: Die Elemente der TTY-Struktur

Die Auswahl des Low-Level-Treibers erfolgt über die *Minor*-Nummer. So wird in Linux z.B. den *TTY*-Geräten mit den *Minor*-Nummern 1-63 der Consolen-Treiber (entspricht unter x86-Linux normalerweise der VGA-Karte des Computers) und den Geräten mit den *Minor*-Nummern 64-127 der serielle Treiber zugeordnet.

Der Low-Level-Treiber dient nur der Ausgabe von Zeichen und bietet Funktionen zur Flußkontrolle. Die Eingabe von Zeichen erfolgt nicht über diese Treiber-Schnittstelle. Die Zeicheneingabe von externen Geräten wie der Tastatur wird typischerweise durch Interrupt-Funktionen behandelt. Dabei wird bei jedem Interrupt genau ein Zeichen vom externen Gerät eingelesen. Würde jedes eingelesene Zeichen sofort über die Treiber-Schnittstelle zur sofortigen Bearbeitung weitergeleitet, könnten die Bearbeitungszeiten für ein Zeichen so groß werden, daß weitere Interrupts ignoriert werden und dadurch Zeichen verloren gehen. Aus diesem Grund existiert für die Eingabe der sogenannte Flip-Buffer (`struct tty_flip_buffer`), der als schneller Pufferspeicher zwischen Interrupt-Behandlung und *Line Discipline* dient. Der in Linux benutzte Flip-Buffer entspricht den in 4.4BSD verwendeten C-Lists (siehe [MBKQ96], Seite 344).

Die *Line Discipline* in POSIX- und 4.4BSD-Systemen nutzt als Basis für die Verarbeitung des Zeichenstromes die `termios`-Struktur (die auf der in System V benutzten `termio`-Struktur aufbaut). Diese Struktur speichert auch den Zustand der *Line Discipline*.

Zur Kommunikation eines Anwendungsprogrammes mit der *Line Discipline* dient der `ioctl`-Systemruf. Über diesen kann der Bearbeitungsmodus geän-

dert bzw. abgefragt werden, Parameter für die serielle Hardware gesetzt werden und Einstellungen für spezielle Zeichen festgelegt werden.

Der modulare Aufbau des *TTY*-Subsystems ermöglicht eine dynamische Zuordnung der *Line Discipline*. Dadurch kann z.B. ein *TTY*, das einem seriellen Low-Level-Treiber (`/dev/ttyS0`) zugeordnet ist, die verschiedensten Aufgaben übernehmen:

Serielles Terminal: Normale Nutzung des seriellen Ports zur Ansteuerung eines Terminals wie z.B. VT100.

Serial Line IP (SLIP): IP-Protokoll über die serielle Schnittstelle. Diese *Line Discipline* ermöglicht eine langsame, aber preiswerte Vernetzung von zwei Computern ohne Netzwerk-Karte.

Das Programm `slattach` öffnet ein serielles Gerät, setzt die Baud-Rate des Gerätes und schaltet auf die SLIP *Line Discipline*. Die `open`-Funktion der SLIP *Line Discipline* verbindet das serielle Gerät mit einem vorkonfigurierten Netzwerk-Interface und bereitet das Senden und Empfangen von Netzwerk-Paketen vor. Ab diesem Zeitpunkt steht ein voll funktionstüchtiges Netzwerk-Gerät zur Verfügung, welches mit den Programmen `ifconfig` und `route` konfiguriert werden kann.

Point-to-Point Protocol (PPP): PPP ist der Nachfolger von SLIP und beseitigt einige Nachteile in der Protokoll-Implementierung von SLIP (siehe [Sim92]).

Generischer Maustreiber: Diese Art der Nutzung der *Line Discipline* erfolgt unter Linux normalerweise nicht. Stattdessen greifen Programme, die die Maus benutzen wollen, direkt auf die serielle Schnittstelle zu. Das Problem dabei ist, daß jeder Maustyp die Daten in einem speziellen Format sendet. Dadurch muß jedes Programm alle Maustypen direkt unterstützen.

Für das General Graphic Interface (GGI) wird jedoch von dieser Möglichkeit Gebrauch gemacht. Es existiert ein Modul für den Linux-Kern, der sich als *Line Discipline* für die serielle Schnittstelle registriert und die empfangenen Daten in standardisierte Ereignisse umwandelt. Dadurch muß ein Programm nur noch diese Ereignisse behandeln können.

2.3 Das GGI-Projekt

Das Team, das am GGI-Projekt arbeitet, hat sich eine schwierige, aber für Linux wichtige Aufgabe gestellt [Nie98]:

> „Wir wollen ein zuverlässiges, stabiles und schnelles Grafik-System, welches auf den verschiedensten Plattformen funktioniert. Wir wollen, daß Programme, die GGI benutzen, auf andere Plattformen durch einfaches Neuübersetzen portierbar sind."

2.3.1 Motivation

Der Ausgangspunkt für die Entstehung dieses Projektes war, daß die Benutzung von Grafik unter Linux zur Zeit bedeutet, entweder das *X-Window-System* oder die SVGAlib[1] zu benutzen. Bei gleichzeitiger Nutzung beider Möglichkeiten ist dabei die Wahrscheinlichkeit hoch, daß es zu inkonsistenter Programmierung der Grafikkartenregister kommt. In einigen Fällen kann dadurch das komplette System zum Absturz gebracht werden. Bei praktisch jeder anderen Hardware versucht der Linux-Kern nur solche Zugriffe auf die Hardware zu erlauben, die die Systemintegrität nicht verletzen. Das GGI-Projekt hat es sich zur Aufgabe gemacht, genau dies auch für Zugriffe auf Grafikkarten durchzusetzen.

Das bedeutet speziell:

- Der Systemkern muß zu jeder Zeit die Fähigkeit besitzen, die Grafik-Hardware neu zu initialisieren und somit in einen definierten Zustand zu bringen.

  Nur wenn diese Bedingung erfüllt ist, kann das System wirkungsvoll vor fehlerhaften Grafik-Anwendungsprogrammen geschützt werden.

  Weiterhin ist es nur so möglich, einen echten *secure attention key (SAK)*[2] zu implementieren.

- Um Programmen, die nicht das *X-Window-System* nutzen, den Zugriff auf die Grafik-Hardware zu ermöglichen, ohne ihnen spezielle Privilegien geben zu müssen, muß der Kern kritische Operationen ausführen.

2.3.2 Designprinzipien

Wie bereits im Abschnitt 2.3.1 erläutert, ist eines der wichtigsten Designprinzipien, alle sicherheitskritischen Operationen in den Systemkern zu verlagern und alle anderen Funktionen einer Bibliothek zu überlassen, die keine speziellen Privilegien benötigt.

Ein Problem dabei stellt die Definition der Schnittstellen dar. Aufgrund der Vielzahl von Grafikkarten mit extrem unterschiedlichem internen Aufbau variiert der sicherheitskritische Funktionsumfang sehr stark.

- Bei Grafikkarten, die nur eine spezielle, fest eingestellte Auflösung bieten, kann der Systemkern den kompletten Video-Speicher (Framebuffer) an Anwendungsprogramme exportieren. Im schlimmsten Fall können fehlerhafte Programme lediglich das aktuell dargestellte Bild zerstören. Die Systemintegrität können sie nicht beeinträchtigen.

[1] Die SVGAlib ist eine Bibliothek, die Funktionen zum direkten Zugriff auf Grafikkarten enthält. Ein Programm, das die SVGAlib benutzt, muß mit dem Recht ausgestattet sein, direkt auf physischen Speicher und IO-Ports zugreifen zu können.

[2] Diese spezielle Taste ermöglicht es dem Anwender durch einfachen Tastendruck sämtliche Programme, die am aktuellen Terminal laufen, abzubrechen.

- Ungünstig aufgebaute Grafikkarten exportieren Register für die Ansteuerung von Beschleuniger-Funktionen gemischt mit Registern für die Einstellung kritischer Hardware-Eigenschaften, die bei fehlerhafter Benutzung das ganze System beeinträchtigen können. Bei solchen Grafikkarten können dadurch auch die eigentlich sicheren Register nicht vom Systemkern exportiert werden.

  Bietet die Grafikkarte die entsprechenden Register jedoch in getrennten Bereichen an, so kann der Zugriff auf die Beschleuniger-Funktionen komplett durch eine Bibliothek im Nutzeradreßraum übernommen werden.

- Einige Grafikkarten benutzen Interrupts zur Signalisierung des Zustandes des Prozessors für Beschleuniger-Funktionen oder ermöglichen DMA-Zugriffe. In beiden Fällen müssen diese Funktionen im Systemkern behandelt werden, um die Sicherheit des Systems zu gewährleisten.

Die Lösung für dieses Problems besteht darin, die Schnittstelle zwischen den Kerntreibern und der GGI-Bibliothek hardwareabhängig zu lassen und erst über die GGI-Bibliothek ein einheitliches API anzubieten (Abbildung 2.3).

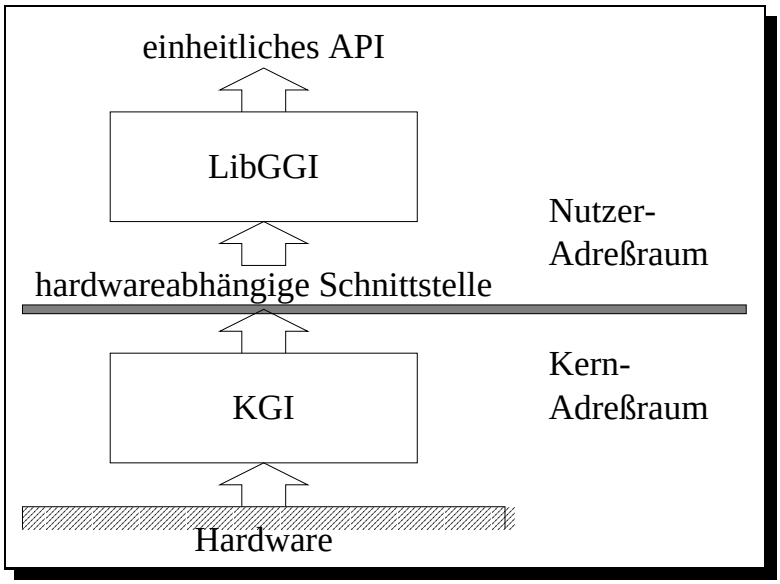


Abbildung 2.3: GGI-Design

Ein weiterer Vorteil dieser Lösung ist, daß die Kerntreiber wirklich minimal gehalten werden können, da die Emulation von nicht durch die Hardware unterstützten Funktionen durch die GGI-Bibliothek erfolgen kann und nicht im Systemkern erfolgen muß.

2.3.3 Kernel Graphics Interface

Das Kernel Graphics Interface (KGI) hat im wesentlichen drei Aufgaben:

1. Gleichzeitige Unterstützung von mehreren Grafikkarten und Monitoren.
2. Unterstützung beliebiger Eingabegeräte einschließlich mehrerer Tastaturen.
3. Bereitstellen aller Funktionen, die für die Implementierung einer sicheren Grafik-Bibliothek im Nutzeradreßraum benötigt werden.

Das KGI für Linux besteht aus vier Modulen. Einer dieser Module, der KGI-Manager, wird statisch zum Systemkern gebunden. Die anderen Module können zur Laufzeit nachgeladen werden.

Der KGI-Manager übernimmt die Verwaltung der verschiedenen Module und stellt eine einheitliche Schnittstelle zu den Treiber-Modulen bereit.

Der Display-Treiber ist abhängig von der benutzten Grafikkarte und enthält die Funktionen zur Steuerung der Grafik-Hardware. Dieser Modul ist wiederum in verschiedene Funktionsgruppen eingeteilt, die die Struktur einer normalen Grafikkarte widerspiegeln (siehe Abschnitt 2.3.4).

Der Tastaturtreiber bearbeitet Tastatureingaben und wandelt diese in standardisierte Ereignisse um. Die Unterstützung mehrerer Tastaturen ist möglich.

Der Maustreiber wandelt den von der Maus gelieferten und vom Maustyp abhängigen Zeichenstrom in ein einheitliches Format, wodurch die Unterstützung von Mäusen verschiedener Hersteller vereinfacht wird.

Die Schnittstelle zum KGI wird über ein normales *Character*-Gerät bereitgestellt (`/dev/graphic`). Das Öffnen dieses Gerätes liefert einen Datei-Identifikator, der für die weitere Kommunikation mit dem KGI benutzt wird. Mögliche Operationen sind `mmap` zum Einblenden des Grafikspeichers in den Nutzeradreßraum, `read` zum Lesen von Ereignissen von Eingabegeräten wie Tastatur und Maus, sowie `ioctl` zum Setzen des Grafikmodus bzw. Auslösen von Beschleuniger-Funktionen.

2.3.4 Display-Treiber

Der Display-Treiber des KGI hat die Aufgabe, die Hardware der Grafikkarte zu programmieren. Grafikkarten bestehen normalerweise aus verschiedenen Funktionsgruppen [Pau97]. Diese Struktur spiegelt sich auch im Aufbau der GGI Display-Treiber wider.

Chipset-Modul: Über diesen Modul erfolgt die Ansteuerung des Grafikchips. Dessen Register dienen dem Setzen des Grafikmodus und vieler anderer Eigenschaften der Grafikkarte.

RAMDAC-Modul: Dieser Modul steuert die Behandlung von Farbpalette und Hardware-Cursor.

Clockchip-Modul: In diesem Modul erfolgt die Berechnung der für den aktuellen Grafikmodus notwendigen Steuersignale und die Programmierung des Clockchips zur Erzeugung der entsprechenden Frequenzen.

Monitor-Modul: Dies ist kein echter Hardware-Treiber, sondern er begrenzt die Werte für Grafikmodi, um die Beschädigung des Monitors zu verhindern.

Acceleration-Modul: Die Ansteuerung kartenspezifischer Beschleuniger-Funktionen erfolgt über diesen Modul.

Dieser streng modulare Aufbau erleichtert die Erstellung von Display-Treibern für neue Grafikkarten, wenn diese aus bereits bekannten Hardware-Bauteilen bestehen. Wird z.B. ein Treiber für eine Grafikkarte benötigt, die lediglich einen anderen RAMDAC besitzt, ansonsten aber einer Grafikkarte entspricht, für die bereits ein Treiber existiert, muß nur für den neuen RAMDAC ein Treiber geschrieben werden.

2.3.5 LibGGI

Die GGI-Bibliothek stellt eine einheitliche Schnittstelle zu Grafikfunktionen bereit, wobei nach Möglichkeit Beschleuniger-Funktionen der verwendeten Grafikkarte benutzt werden. Dabei besteht die GGI-Bibliothek aus einer Reihe allgemeiner Funktionen, die durch sogenannte Targets ergänzt werden (Abbildung 2.4). Die eigentlichen Operationen zur Ausgabe von Grafikdaten werden durch die Targets ausgelöst. Ein Target kann ein Treiber für eine Grafikkarte sein, der mit Unterstützung des KGI direkt auf die Grafik-Hardware zugreift. Möglich ist jedoch auch, daß ein Target auf anderen Funktionen zur Grafikausgabe wie z.B. der Xlib aufbaut. In diesem Fall werden die Grafikdaten an den X-Server weitergeleitet, der dann die eigentliche Ausgabe erledigt.

Targets sind nicht direkt in der LibGGI enthalten, sondern werden durch diese beim Starten eines Programmes ausgewählt und über den dynamischen Linker geladen. Dieser Aufbau ermöglicht einen extrem flexiblen Einsatz von Programmen, die zur Ausgabe die GGI-Bibliothek benutzen. Ohne das Programm zu ändern, kann die Ausgabe auf die verschiedenen Targets geleitet werden.

Eine Auswahl der bereits implementierten Targets soll kurz vorgestellt werden:

KGI-Target: Das ist das Target, für das die LibGGI ursprünglich entworfen wurde. Jetzt ist es nur eines unter vielen, nimmt aber trotzdem eine Sonderstellung ein. Dieses Target benutzt zur Ausgabe die Schnittstelle des KGI (siehe Abschnitt 2.3.3). Es ermöglicht die Nutzung von Funktionen zur Beschleunigung der Grafikausgabe. Für den Nutzer der LibGGI erfolgt dies jedoch vollständig transparent.

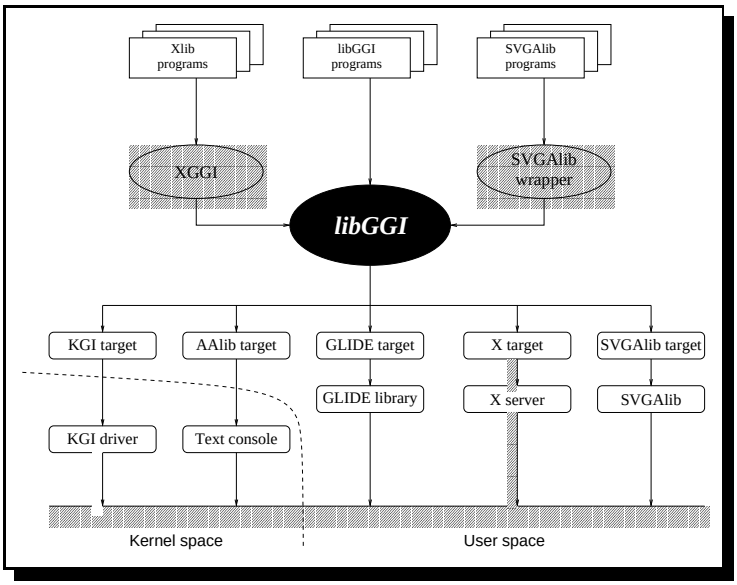


Abbildung 2.4: GGI-Überblick

X-Target: Ein Target, das die über die LibGGI erfolgenden Ausgaben über die Xlib an einen X-Server weitergibt. Die Nutzung dieses Targets erfordert lediglich das Vorhandensein eines X-Servers und ermöglicht dadurch den Einsatz der LibGGI auf nahezu allen UNIX-Systemen.

GLIDE-Target: Dieses Target benutzt zur Ausgabe die GLIDE-Bibliothek. Diese ermöglicht die Ansteuerung von 3D-Grafikkarten der Firma 3Dfx.

AA-Target: Ein etwas ungewöhnliches Target. Es erlaubt die Darstellung von Grafik durch geschickte Anordnung von Zeichen auf einem Text-Bildschirm über die Bibliothek AAlib.

2.3.6 GGI Consolen-Treiber

Zusätzlich zu den bereits vorgestellten Teilen des GGI-Projektes existiert noch ein Teilprojekt, das einen neuen Consolen-Treiber für Linux implementiert. Dieser soll einige Probleme mit der derzeitigen Implementierung des Consolen-Treibers in Linux beheben. Möglich sein soll z.B. eine flexiblere und dynamischere Zuordnung von Eingabegeräten zu Ausgabegeräten. Dieser Treiber ist fest in den Systemkern integriert, ermöglicht aber das Laden Modulen zur Erweiterung der Funktionalität. So können modifizierte *Console Parser* (Abbildung 2.5) geladen werden, die eine andere Umsetzung von Steuerzeichen vornehmen als der XTerm *Console Parser*.

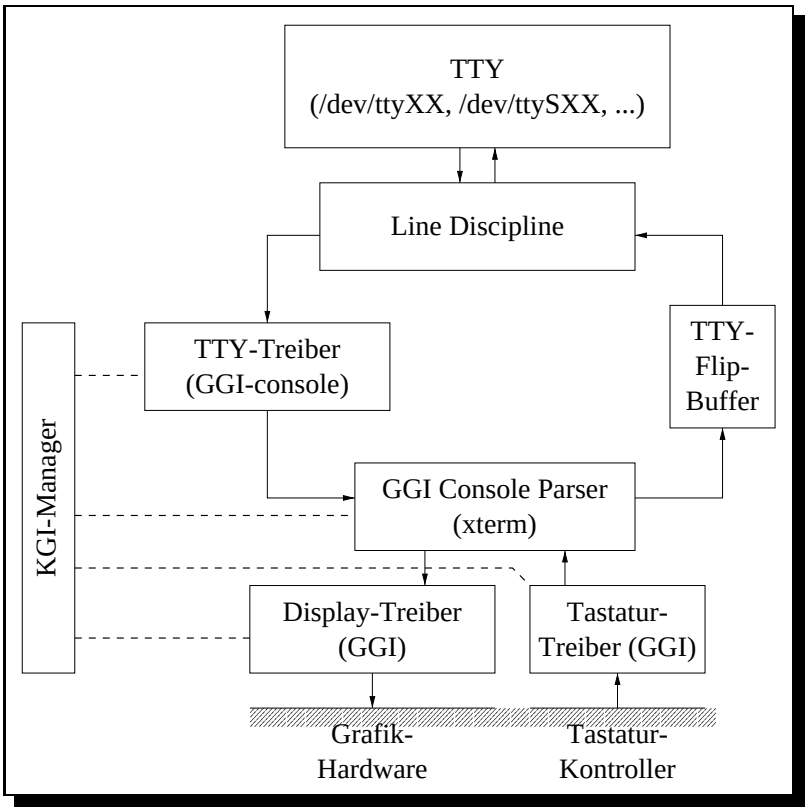


Abbildung 2.5: Architektur des TTY-Subsystems in GGI-Linux

2.4 X-Window-System

Das *X-Window-System* ist ein portables, netzwerktransparentes Grafiksystem, das auf einer großen Anzahl von Computern mit Rastergrafik-Bildschirmen lauffähig ist. Es unterstützt sich überlappende hierarchische Fenster sowie Text- und Grafik-Operationen auf Monochrom- und Farbbildschirmen.

2.4.1 Client-Server Modell

Die eigentliche Darstellung auf dem Bildschirm übernimmt ein spezielles Programm, der X-Server. Die Aufgaben des X-Servers sind:

- mehreren Clients den Zugriff auf ein Display zu ermöglichen,
- die über das Netzwerk gesendeten Nachrichten zu interpretieren,
- Nutzereingaben über das Netzwerk zum Client zu senden,
- 2-dimensionale Grafikausgaben zu erzeugen; die Berechnung erfolgt dabei im X-Server und nicht im Client, was unter anderem die Nutzung von Beschleuniger-Hardware der Grafikkarte ermöglicht und
- Ressourcen zu verwalten (Fenster, Cursor, Zeichensätze, Grafik-Kontexte usw.) und die gemeinsamen Nutzung von Ressourcen durch mehrere Clients zu überwachen; der Zugriff erfolgt über serverweit eindeutige Identifikatoren (Resource Ids).

Die Applikationen übernehmen die Rolle der X-Clients und kommunizieren mit dem X-Server über das X-Protokoll.

2.4.2 X-Protokoll

Grundlage des *X-Window-Systems* ist das X-Protokoll, welches die Kommunikation zwischen Applikation (X-Client) und dem X-Server beschreibt. Ein wichtiges Merkmal des X-Protokolls ist, daß es eine asynchrone Datenübertragung ermöglicht. Das begründet, daß Anforderungen des Clients normalerweise nicht bestätigt werden. Möglich ist das allerdings nur durch die Nutzung eines zuverlässigen Transport-Protokolls (z.B. TCP).

Eine der wichtigsten Eigenschaften des X-Protokolls ist, daß es unabhängig von Betriebssystemen und Programmiersprachen ist.

2.4.3 Xlib

Mit dem X-Protokoll kommt ein Applikations-Programmierer normalerweise nicht in Berührung. Es wird vollständig durch eine Schnittstelle in einer bestimmten Programmiersprache gekapselt. Diese Schnittstelle existiert für die

unterschiedlichsten Sprachen. Eine Auflistung kann [Lew97] entnommen werden. Zum Einsatz kommt jedoch in den meisten Fällen die Anbindung zur Sprache C, die Xlib.

Die Xlib bietet nur die grundlegenden Funktionen zur Nutzung des *X-Window-Systems*. Sie ist nicht dafür gedacht, die Basis für komplette Applikationen zu bilden. Stattdessen soll die Xlib die Voraussetzungen für die Implementierung sogenannter *Toolkits* schaffen, die die für eine komplexe Applikation nötigen Oberflächenelemente anbieten[3]. *Toolkits* haben außerdem die Aufgabe, das „Look and Feel“ von Applikationen festzulegen.

[3]Beispiele für solche *Toolkits* sind das *Athena Widget-Set (Xaw)* des MIT, das *Motif Widget-Set* der Open Software Foundation (OSF) und das frei verfügbare GIMP Toolkit (gtk).

Kapitel 3

Entwurf

Die Zielstellung dieser Arbeit ist es, die Darstellung von Grafik in Echtzeit zusammen mit der Darstellung der normalen Arbeitsumgebung zu ermöglichen. Gleichzeitig sollen das Echtzeit-System und das Timesharing-System so entkoppelt werden, daß das Timesharing-System das Echtzeit-System nicht beeinflussen kann.

Es soll eine Echtzeit-Darstellungskomponente (EZDK) entworfen werden, die diese Anforderungen erfüllt.

3.1 Ausgangspunkt

Die normale Arbeitsumgebung in UNIX bedeutet die Verwendung des *X-Window-Systems* als praktisch einzigen 2D-Grafikstandard im UNIX-Bereich. Die Unterstützung des *X-Window-Systems* durch die EZDK ist dadurch notwendig. Daraus ergibt sich die Frage, wie ein problemloses Zusammenspiel der beiden Teilsysteme, Echtzeit-Darstellungskomponente und *X-Window-System*, möglich ist.

Schwerpunkte für diese Untersuchung sind:

- die Prozeßstruktur bei gleichzeitiger Nutzung von Timesharing- und Echtzeit-Grafikanwendungen,
- Abhängigkeiten der Prozesse untereinander,
- die Synchronisation der Teilsysteme und
- die Möglichkeit der Nutzung von Funktionen zur hardwarebeschleunigten Grafikausgabe

Drei verschiedene Modelle für die Implementierung der Darstellungskomponente sollen untersucht werden:

1. Direkte Einbindung der EZDK in den X-Server.

2. EZDK als eigenständiges Programm, das den Hardware-Treiber des X-Servers nutzt.

3. EZDK als Programm mit eigenem Hardware-Treiber; der X-Server nutzt die EZDK zur Darstellung.

3.1.1 Modell 1: EZDK im X-Server

Bei diesem Modell erfolgt eine direkte Einbindung der EZDK in den X-Server (Abbildung 3.1).

Dieses Modell kann von der Erweiterbarkeit des X-Protokolles und des X-Servers profitieren. Vorteilhaft ist die direkte Integration, wenn die Echtzeit-Anwendungen auch eine komplette Benutzeroberfläche anbieten sollen. Diese kann ohne zusätzlichen Aufwand über Funktionen des *X-Window-Systems* erzeugt werden, während die Daten für die Echtzeit-Darstellung über eine spezielle Schnittstelle an den erweiterten X-Server übergeben werden. Nachteilig daran ist offensichtlich, daß in diesem Fall jede Echtzeit-Komponente eine Implementierung der Xlib benötigt, um mit dem X-Server zu kommunizieren.

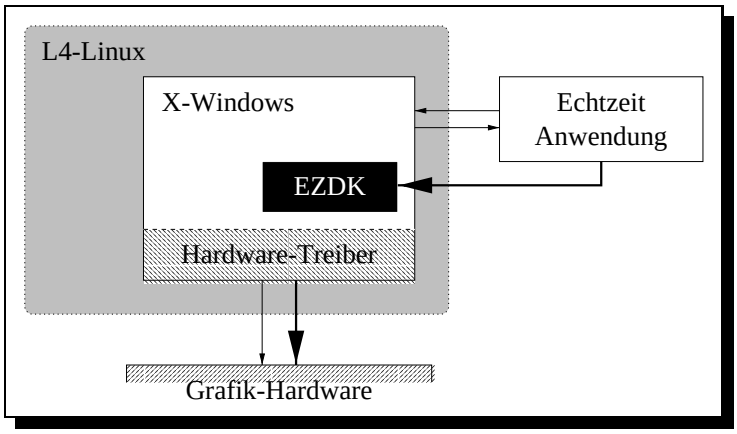


Abbildung 3.1: Modell 1 – EZDK integriert in den X-Server

Über den Timesharing-Teil des X-Servers können jedoch keine Aussagen bezüglich des zeitlichen Verhaltens gemacht werden. Die EZDK muß daher direkt auf den Grafikspeicher zugreifen, um Zusagen über die Dauer der verschiedenen Operationen machen zu können. Nur so ist eine vollständige Entkoppelung der Threads der EZDK vom Timesharing-System möglich. Diese ist notwendig, da der Grafiktreiber des X-Servers nicht echtzeitfähig ist. Die Nutzung von Funktionen zur hardwarebeschleunigten Grafikausgabe muß ebenfalls ausgeschlossen werden, da diese Funktionen normalerweise auch vom Timesharing-Teil des X-Servers verwendet werden. Der Zugriff auf die Beschleuniger-Hardware erfolgt typischerweise über einen FIFO-Speicher begrenzter Länge auf der Grafikkarte. Das hat zur Folge, daß Timesharing-Aufträge den FIFO-Speicher füllen und damit andere Aufträge blockieren können, bis die entsprechenden Operationen

ausgeführt worden sind. Eine Kontrolle der Timesharing-Aufträge, um diese Blockierung der Echtzeit-Aufträge zu verhindern, würde erhebliche Änderungen im X-Server erfordern. Das ist insbesondere aus Gründen der einfachen Nutzung neuer Versionen des X-Servers nicht sinnvoll, wenn Alternativen existieren.

Der direkte Zugriff auf den Grafikspeicher hat den Nachteil, daß er am X-Server vorbei erfolgt. Dadurch muß eine Synchronisation derart erfolgen, daß Speicherbereiche, die von der EZDK benutzt werden, dem X-Server bekannt gemacht werden, damit dieser nicht auch auf diese Bereiche zugreift.

Weiterhin ist es möglich, daß bei laufendem X-Server der Grafikmodus umgeschaltet wird. Das bedeutet, daß sich die Eigenschaften des Grafikspeichers ändern, was bei direktem Zugriff der EZDK beachtet werden muß, um Darstellungsfehler zu vermeiden.

Ein großer Nachteil dieses Modells ist, daß die Nutzung von Grafikanwendungen nur dann möglich ist, wenn L^{4}Linux läuft. Die Portierung des X-Servers direkt auf L4 ist eine Aufgabe, die nur mit extrem hohem Aufwand lösbar ist. Das bedeutet, daß sowohl L^{4}Linux als auch ein darauf laufender X-Server Voraussetzung dafür sind, von L4-Programmen aus Grafik darzustellen. Diese Einschränkung kann nur dann akzeptabel sein, wenn es keine andere Möglichkeit gibt, eine Darstellungskomponente für DROPS zu implementieren.

3.1.2 Modell 2: EZDK nutzt X-Server

Die Darstellungskomponente läuft als eigenständiges Programm, nutzt jedoch den Hardware-Treiber des X-Servers (Abbildung 3.2). Das könnte bedeuten, daß die EZDK beim Start mit dem X-Server kommuniziert und der X-Server die Hardware-Adresse des Grafikspeichers sowie dessen Eigenschaften mitteilt. Danach kann die EZDK direkt auf den Grafikspeicher zugreifen.

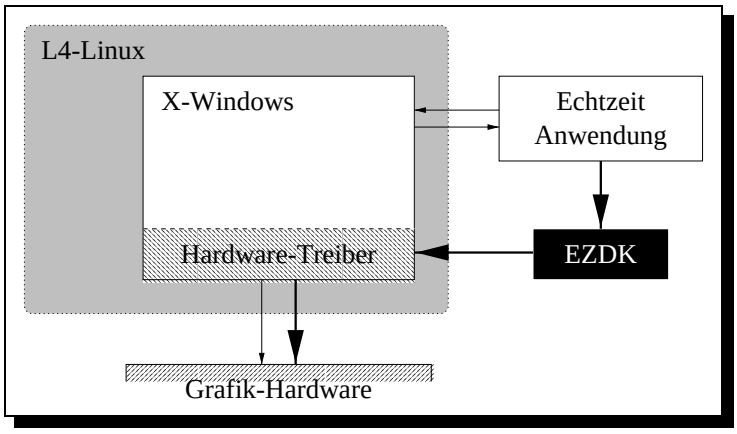


Abbildung 3.2: Modell 2 – EZDK als separater Prozeß, der den Hardware-Treiber des X-Servers nutzt

In diesem Fall kann die Entwicklung der EZDK unabhängig vom X-Server erfolgen, was einige Vorteile gegenüber der direkten Integration in den X-Server

bietet. So ist z.B. die Nutzung von neuen Versionen des *X-Window-Systems* einfacher möglich, da für dieses Modell nur geringe Änderungen am X-Server notwendig sind.

Der Zugriff auf den Grafikspeicher kann wie bei Modell 1 nur direkt über die bekannte Hardware-Adresse erfolgen. Eine Nutzung von Beschleuniger-Funktionen ist nicht möglich, da nach der Initialisierung keine Verbindung der EZDK zum X-Server mehr bestehen darf. Andernfalls könnte das Timesharing-System die EZDK unkontrollierbar beeinflussen.

Einige Probleme die auch bei der direkten Integration auftreten, werden aber auch durch diese Strukturierung nicht gelöst. So ist insbesondere auch hier jede Grafikdarstellung von L^{4}Linux und einem X-Server abhängig.

3.1.3 Modell 3: EZDK eigenständig

Wie bei Modell 2 läuft die Darstellungskomponente als eigenständiges Programm, nutzt jedoch nicht den X-Server bzw. dessen Hardware-Treiber, sondern einen eigenen Grafiktreiber (Abbildung 3.3). Dieser Grafiktreiber wird auch vom X-Server für dessen Grafikausgaben angesprochen, ist aber unabhängig von L^{4}Linux.

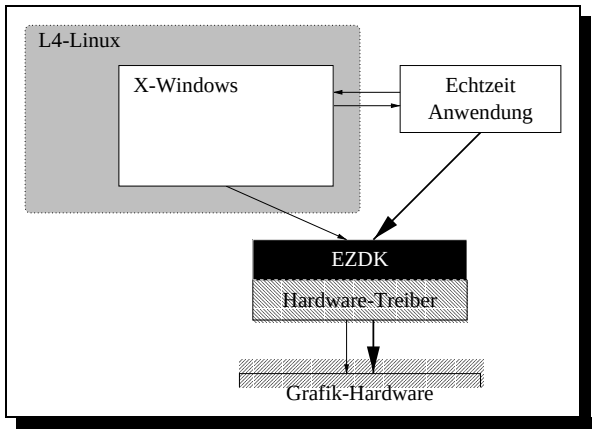


Abbildung 3.3: Modell 3 – EZDK mit eigenem Hardware-Treiber

Das Hauptproblem bei dieser Variante ist, daß es einen separaten Grafiktreiber geben muß, der mit möglichst vielen verschiedenen Grafikkarten umgehen kann. Außerdem muß ein X-Server implementiert werden, der zur Grafikausgabe keinen eigenen Grafiktreiber enthält, sondern mit einem externen Treiber per IPC kommuniziert.

Da alle Grafikausgaben über die EZDK erfolgen, hat sie in diesem Fall die uneingeschränkte Kontrolle über die Grafikhardware, was eine kontrollierte Nutzung von Beschleuniger-Funktionen ermöglicht. Bekannt sind an dieser Stelle auch die Bereiche des Bildschirmes, die für die Anzeige von Echtzeit-Datenströmen

verwendet werden. Dadurch besteht die Möglichkeit, Zugriffe des X-Servers auf diese Bereiche zu verhindern.

Die Synchronisation von Timesharing- und Echtzeit-Anwendungen, die auf die EZDK zugreifen, kann direkt durch die EZDK durchgeführt werden. Das ermöglicht eine zuverlässige Einschätzung des Einflusses von Grafikausgaben der Timesharing-Anwendungen. Notwendig ist eine Synchronisation beispielsweise, wenn Timesharing-Anwendungen der Zugriff auf Beschleuniger-Funktionen der Grafikkarte erlaubt werden soll.

Der entscheidende Vorteil dieses Modells ist jedoch die Möglichkeit, Grafik-Applikationen zu implementieren, die direkt auf L4 laufen, ohne daß diese ein laufendes L^{4}Linux erfordern.

3.1.4 Auswahl

Die Betrachtung der Vor- und Nachteile der verschiedenen Modelle macht deutlich, daß eine Implementierung nach Modell 3 die flexibelste Lösung darstellt. Insbesondere ist die Nutzung von Grafik möglich, ohne daß L^{4}Linux und ein X-Server laufen.

Daher soll nun untersucht werden, ob dieses Modell praktisch umsetzbar ist und welche Eigenschaften die Laufzeitumgebung besitzen muß.

3.2 Voraussetzungen

Eine wichtige Voraussetzung für die Realisierung von Modell 3 ist das Vorhandensein eines flexiblen und einfach zu portierenden Grafiktreibers. Ohne einen solchen Grafiktreiber ist die Implementierung einer allgemein einsetzbaren EZDK nicht möglich, da bereits der Aufwand für die Neuentwicklung eines Grafiktreibers für eine spezielle Grafikkarte relativ hoch ist. Weiterhin muß auch ein X-Server an diesen Grafiktreiber angepaßt werden, um eine normale Arbeitsumgebung zu ermöglichen.

Die Untersuchung aktueller Projekte zeigt, daß es ein Projekt gibt, welches eine standardisierte Schnittstelle zur Grafik-Hardware bereitstellen will: Das GGI-Projekt (siehe Abschnitt 2.3). Die aktuelle Implementierung ist dabei zwar auf Linux ausgerichtet, Zielstellung ist jedoch die Bereitstellung einer portablen Schnittstelle. So ist z.B. eine Portierung auf FreeBSD geplant.

Da für L4 eine Linux-Portierung existiert [Hoh96] und bereits mehrere Linux-Treiber auf L4 portiert wurden [Sta96, Meh96, Pau97], ist die derzeitige Ausrichtung des GGI-Projektes auf Linux eher von Vorteil. Dadurch können erprobte Methoden zur Bereitstellung einer Linux-Umgebung für Linux-Kerntreiber genutzt werden.

Um die Hardware-Treiber des GGI-Projektes für die EZDK nutzbar zu machen, müssen die von diesem benutzten Funktionen des Linux-Kernes bereitgestellt

werden. Außerdem müssen Schnittstellen gefunden werden, die die Nutzung eines externen Grafiktreibers von L^{4}Linux aus ermöglichen.

Im Rahmen des GGI-Projektes wurde bereits ein X-Server an die entwickelten Grafiktreiber angepaßt. Dieser nutzt die `/dev/graphic`-Schnittstelle, so daß bei Implementierung dieser Schnittstelle in L^{4}Linux keine Veränderungen am GGI-X-Server vorgenommen werden müssen.

3.3 Prozeßstruktur

Da in Abschnitt 3.1 lediglich eine grobe Betrachtung der beteiligten Prozesse erfolgte, soll nun die durch einen Entwurf nach Modell 3 entstehende Prozeß- und Threadstruktur etwas genauer dargestellt werden.

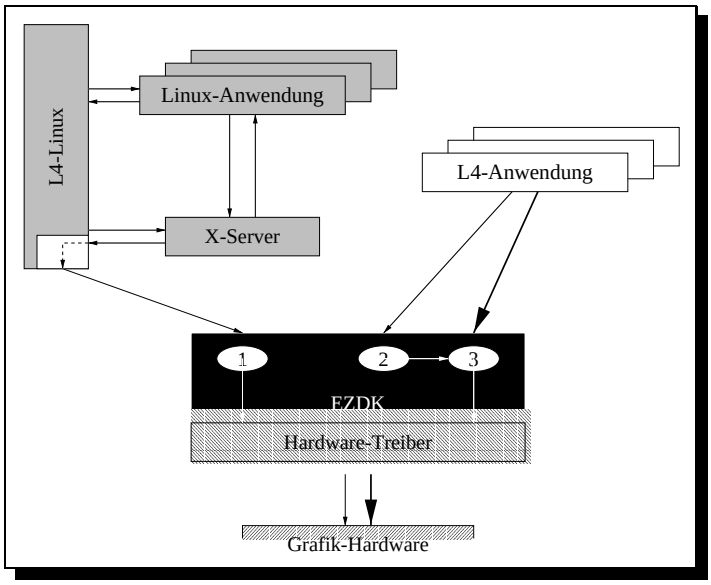


Abbildung 3.4: Prozeßstruktur der EZDK

Wie in Abbildung 3.4 zu sehen ist, erfolgt die Zusammenführung der Timesharing- und Echtzeit-Datenströme in der EZDK. Das bedeutet, die Entkoppelung der verschiedenen Datenströme muß in der EZDK erfolgen. Wichtig dabei ist, daß die Datenströme des Timesharing-Systems die des Echtzeit-Systems nicht beeinflussen.

Zur Entkoppelung der verschiedenen Datenströme existieren daher in der EZDK eine Reihe von Threads (vgl. Abbildung 3.4):

1. Ein Thread zur Kommunikation mit L^{4}Linux und L4-Tasks ohne Echtzeit-Anforderungen. Dieser Thread bearbeitet alle Funktionen, die für die Emulation der GGI-Schnittstelle `/dev/graphic` (Abschnitt 2.3.3) notwendig sind.

2. Ein Thread, der Verbindungen für Echtzeit-Datenströme verwaltet und deren Steuerung ermöglicht. An dieser Stelle erfolgt auch die Kontrolle, ob das Öffnen eines weiteren Datenstromes möglich ist, sowie die Anforderung der für diesen Strom notwendigen Ressourcen (insbesondere die Reservierung der für die Darstellung nötigen Prozessorzeit beim Scheduler).

3. Ein Thread pro Echtzeit-Datenstrom, der die eigentliche Übergabe der Daten an die Grafikkarte übernimmt.

In den meisten Fällen benötigt eine Anwendung, die im Echtzeit-System läuft und Grafik nutzt (z.B. zur Ausgabe von Videos), auch die Möglichkeit, eine Benutzerschnittstelle darzustellen und auf Benutzereingaben zu reagieren. Da für diese Aufgaben eine Angabe von Zeitschranken weder möglich noch sinnvoll ist, müssen diese Anwendungen auch Zugriff auf die Timesharing-Schnittstelle der EZDK haben. Dafür sind verschiedene Lösungen denkbar:

1. Wenn im Timesharing-System L^{4}Linux mit einem X-Server läuft, kann die Anwendung entweder direkt mit dem X-Server kommunizieren, oder indirekt über einen Stellvertreter-Prozeß im L^{4}Linux.

   Die programmtechnisch einfachere Methode ist dabei die Nutzung eines Stellvertreter-Prozesses, da für dessen Entwicklung die komplette Programmier- und Testumgebung inklusive aller Oberflächen-Bibliotheken in Linux zur Verfügung steht. Dieser Prozeß könnte über ein im Vergleich zum X-Protokoll einfachen Protokoll mit der Echtzeit-Komponente kommunizieren.

   Für die direkte Kommunikation mit dem X-Server muß die von Anwendungsprogrammen benötigte Bibliothek (Xlib) auf L4 portiert werden, sowie entweder in L^{4}Linux oder im X-Server eine Möglichkeit bereitgestellt werden, über L4-IPC eine Socket-Kommunikation durchzuführen. Denkbar wäre auch ein minimaler Server-Prozeß, der die Daten des X-Protokolls über L4-IPC empfängt und an den X-Server weiterleitet.

2. Ist kein L^{4}Linux bzw. kein X-Server im System vorhanden, muß die Anwendung die Möglichkeit haben, direkt mit der EZDK über deren Timesharing-Schnittstelle zu kommunizieren.

Da Lösung 2 relativ leicht zu implementieren und bei Fehlen eines laufenden L^{4}Linux in jedem Fall notwendig ist, wird in Abschnitt 3.6 eine minimale Schnittstelle für den (nicht echtzeitfähigen) direkten Zugriff auf den Grafikspeicher vorgestellt.

3.4 Eingabetreiber

Das Kernel Graphic Interface des GGI-Projektes enthält nicht nur Treiber für die Steuerung von Grafikkarten, sondern auch Eingabetreiber für Tastatur und

Maus. Der Tastaturtreiber ist dabei direkt im GGI-Linux enthalten, da er sofort nach dem Booten des Kernes zur Verfügung stehen muß. Der Maustreiber ist als *Line Discipline* realisiert (siehe Abschnitt 2.2.2).

Da für L4 noch kein eigener Tastaturtreiber existiert, ist es sinnvoll, den Treiber des KGI allgemein für L4-Tasks verfügbar zu machen. Das bedeutet aber auch, daß L^4Linux angepaßt werden muß, um diesen Treiber benutzen zu können.

Die Verwendung des kompletten *TTY*-Subsystems von GGI-Linux ermöglicht L4-Tasks auch die direkte Nutzung von virtuellen Consolen und damit eine echte Integration in die gewohnte Arbeitsumgebung.

3.5 Echtzeit-Schnittstelle

Die Schnittstelle zu den Echtzeit-Funktionen der EZDK dient der Integration in das Echtzeit-System von DROPS. Diese Schnittstelle basiert auf zeitgesteuertem Einblenden von Speicher vom Erzeuger in den Verbraucher. D.h. der Erzeuger von Daten organisiert einen Speicherbereich im Adreßraum des Verbrauchers (der Erzeuger arbeitet als *external pager*). Diese Technik ist ähnlich den in [DP93] vorgestellten fbufs, erweitert diese jedoch um Zeitintervalle in denen ein bestimmter Speicherbereich für den Verbraucher gültig ist [HBM+98] (Abbildung 3.5).

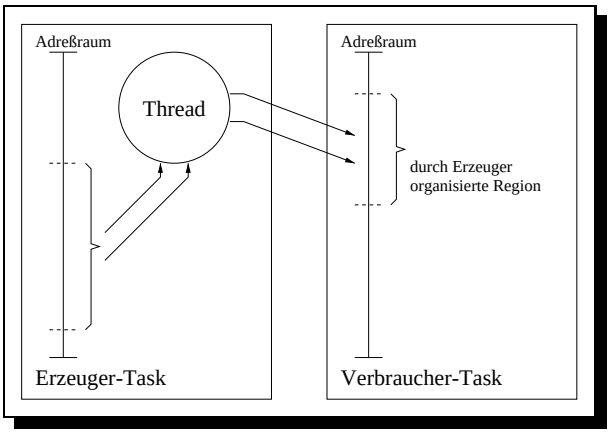


Abbildung 3.5: Prinzip der Echtzeit-Schnittstelle

Diese Schnittstelle ermöglicht es dem Erzeuger-Thread, Daten im Voraus zu bearbeiten und in einem Ausgabepuffer abzulegen. Die in diesem Puffer gespeicherten Daten werden durch einen zeitgesteuerten, zum Erzeuger-Thread asynchron arbeitenden Sende-Thread an den Verbraucher weitergeleitet. Dadurch ist es möglich, kurzzeitige Schwankungen des Ausgabestromes auszugleichen und einen stabilen Strom bereitzustellen (Abbildung 3.6).

Um diese Schnittstelle für die Programmierung von Echtzeit-Komponenten und Anwendungsprogrammen nutzbar zu machen, ist es wichtig, eine Bibliothek

mit einem klar definierten API zur Verfügung zu stellen. Nur so ist gewährleistet, daß die Komplexität der Schnittstelle vor den Anwendungsprogrammen verborgen werden kann. Diese Bibliothek muß für jede im System vorhandene Komponente um eine komponentenspezifische Bibliothek erweitert werden, die hauptsächlich die Transformation von Parametern der Anwendungs-Ebene auf Parameter der zugrundeliegenden Bibliothek übernimmt. Im einfachsten Fall sind dabei lediglich die `open` und `close` Funktion zu implementieren, wobei nur die `open`-Funktion für die Anwendung sichtbar ist, da die `close`-Funktion durch die allgemeine Bibliothek intern aufgerufen werden kann.

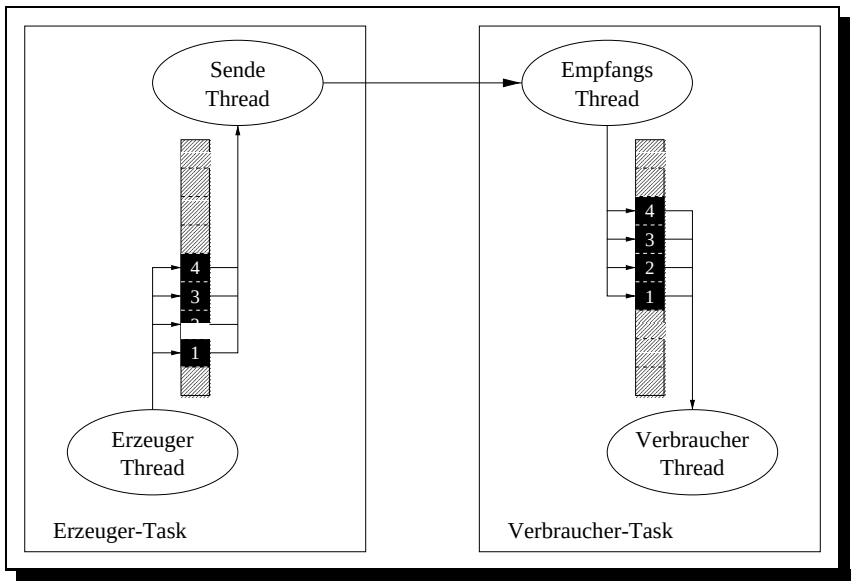


Abbildung 3.6: Threads der Echtzeit-Schnittstelle

Ziel dieses API ist, die Echtzeit-Schnittstelle allgemein zu kapseln, um eine einfache und flexible Implementierung von Komponenten zu ermöglichen. Komponenten, die diese Schnittstelle benutzen, können von einem Anwendungsprogramm zu einer Kette gekoppelt werden, um verschiedene Aufgaben zu lösen.

Die Funktionen des API teilen sich in 4 Gruppen.

3.5.1 Anwendungen

Diese Funktionen dienen dem Öffnen und Schließen von Verbindungen zu Echtzeit-Komponenten und dem Verbinden mehrerer Komponenten zu einer Kette die einen Datenstrom verarbeitet.

Die `open`-Funktion fehlt hier, da diese durch eine komponentenspezifische Bibliothek verborgen wird. Diese ermöglicht der Anwendung, die zu erzeugenden Ströme über Applikations-Parameter wie z.B. Höhe, Breite und Bildrate bei Videoströmen zu spezifizieren, anstatt in Einheiten wie Bytes pro Sekunde, die von der allgemeinen Bibliothek intern benutzt werden.

```
rt_error_t
rt_connect(rt_stream_t sender, rt_stream_t receiver);

rt_error_t
rt_close(rt_stream_t handle);
```

3.5.2 Ausgabeströme

Die Funktionen dieser Gruppe dienen dem Erzeugen und Verwalten von Ausgabeströmen und werden von Komponenten genutzt, die Daten erzeugen bzw. verarbeiten und dann an andere Komponenten zur weiteren Bearbeitung übergeben.

Über die Funktion `rt_ostream_create()` wird innerhalb der Bibliothek ein Sende-Thread erzeugt, der das eigentliche Absenden der Daten übernimmt. Weiterhin wird der Arbeits-Thread erzeugt und mit der als Argument angegebenen Funktion gestartet.

Die Funktion `rt_ostream_get_buffer()` liefert einen Ausgabepuffer mit definierten Eigenschaften:

1. Unabhängig von der internen Verwaltung und des Füllstandes des Pufferspeichers wird garantiert, daß ein Speicherbereich von mindestens der in `rt_ostream_create()` angegebenen Puffergröße zur Verfügung steht oder ein Fehler gemeldet wird. Zugriffe auf einen größeren Bereich sind nicht definiert.

2. Das Senden des Puffers erfolgt ohne zusätzliche, in der Bibliothek verborgene Kopieroperation.

3. Die zum garantierten Speicherbereich gehörenden physischen Seiten sind in keiner anderen Task eingeblendet, wodurch dem Sender der exklusive Zugriff auf diesen Speicher möglich ist. (Zugriff aus Sicht der Sende-Task, d.h. es erfolgte vor Bereitstellen des Puffers ein `l4_fpage_unmap(fpage, L4_FP_FLUSH_PAGE | L4_FP_OTHER_SPACES)`. Der für diesen Speicherbereich zuständige *external pager* hat im Normalfall noch die Möglichkeit, auf diesen Bereich zuzugreifen. Dieser muß allerdings in jedem Fall von der Sende-Task als vertrauenswürdig eingestuft werden.) Nur so ist die Autonomie der Sende-Task beim Zugriff auf den Puffer gewährleistet. Andernfalls wäre es möglich, daß nachfolgende Filter-Tasks, die die empfangenen Daten direkt modifizieren, um eine Kopieroperation zu sparen, den Sendepuffer vor dem Absenden unkontrolliert modifizieren.

Nach Füllen des Puffers mit den zu übertragenden Daten muß die Komponente der Bibliothek mitteilen, wie groß der wirklich genutzte Bereich des Puffers ist. Dazu dient `rt_ostream_commit_buffer()`. Notwendig ist dieser Wert insbesondere für Komponenten, die einen lückenlosen Bytestrom erzeugen müssen, jedoch Eingabedaten mit schwankenden Größen erhalten. Allgemein können dadurch

beliebige Strategien für die Nutzung des Pufferspeichers durch die Komponente umgesetzt werden.

```
rt_stream_t
rt_ostream_create(l4_threadid_t src_threadid,
       void (*thread_func)(rt_stream_t, void *),
       void *client_data,
       long buffer_size,
       int  buffer_count,
       long bytes_per_second,
       int  buffer_align);

rt_error_t
rt_ostream_destroy(rt_stream_t stream_handle);

void *
rt_ostream_get_buffer(rt_stream_t stream_handle);

rt_error_t
rt_ostream_commit_buffer(rt_stream_t stream_handle, int size);
```

3.5.3 Eingabeströme

Analog zu den Funktionen für Ausgabeströme dienen diese Funktionen dem Erzeugen und Verwalten von Eingabeströmen. Sie werden von Komponenten genutzt, die Daten von anderen Komponenten empfangen und verarbeiten.

Mit der Funktion `rt_istream_create()` werden in der Bibliothek der Empfangs- und der Arbeits-Thread erzeugt. Der Arbeits-Thread wird mit der als Argument angegebenen Funktion gestartet.

Über die Funktion `rt_istream_get_buffer()` erhält die Komponente einen Puffer mit den Eigenschaften:

1. Bei korrektem Zeitverhalten von Sender und Empfänger ist der Zugriff auf einen Speicherbereich mit der im Sender spezifizierten Größe ohne Seitenfehler möglich.
2. Der Pufferspeicher wird zeitgesteuert vom Sender entzogen.
3. Ein Zugriff nach dem Entzug des Speicher wird innerhalb der Bibliothek erkannt. Die Komponente wird davon benachrichtigt.

Eine explizite Freigabe des Puffers ist durch den zeitgesteuerten Speicherentzug nicht notwendig.

```
rt_stream_t
rt_istream_create(l4_threadid_t src_threadid,
       void (*thread_func)(rt_stream_t, void *),
       void *client_data,
```

```
        long bytes_per_second);

rt_error_t
rt_istream_destroy(rt_stream_t stream_handle);

void *
rt_istream_get_buffer(rt_stream_t stream_handle);
```

3.5.4 Filter

Diese Funktionsgruppe kombiniert das Erzeugen von zusammengehörigen Ein- und Ausgabe-Datenströmen. Die Nutzung erfolgt durch Komponenten, die zu einem Datenstrom gehörende Daten empfangen, verarbeiten und danach an eine weitere Komponente senden. Prinzipiell ist diese Gruppe von Funktionen nicht erforderlich, da sie durch getrennte Aufrufe von Funktionen für Ein- und Ausgabe-Datenströmen ersetzt werden können. Da jedoch der Einsatz von Filtern den Normalfall bei der Verarbeitung von Datenströmen darstellt, vereinfacht sich die Nutzung der Schnittstelle.

Spezielle Funktionen zur Verwaltung der Puffer sind nicht notwendig. Der Zugriff auf Puffer erfolgt über die gleichen Funktionen, die für Ein- bzw. Ausgabe-Datenströme zur Verfügung stehen.

```
rt_stream_t
rt_iostream_create(l4_threadid_t src_threadid,
        void (*thread_func)(rt_stream_t, rt_stream_t, void *),
        void *client_data,
        long bytes_per_second_in,
        long buffer_size,
        int  buffer_count,
        long bytes_per_second_out,
        int  buffer_align);

rt_error_t
rt_iostream_destroy(rt_stream_t stream_handle);
```

3.5.5 Synchronisation

Die Synchronisation der Threads der Echtzeit-Schnittstelle erfolgt ausschließlich mit Mitteln des DROPS-Scheduling-Framework. Dadurch arbeiten jeweils Erzeuger- und Sende-Thread sowie Empfänger- und Verarbeitungs-Thread vollständig asynchron. Diese Arbeitsweise ermöglicht es dem Erzeuger-Thread, eine nur vom vorhandenen Pufferspeicher abhängige Vorarbeit zu leisten. Bei synchronem Senden der Daten wäre das nicht einfach möglich.

3.6 Timesharing-Schnittstelle

Die Timesharing-Schnittstelle der Darstellungskomponente soll eine einfache Integration in L^{4}Linux ermöglichen. Das bedeutet, daß der Zugriff auf Ein- und Ausgabe von Textzeichen über das *TTY*-Subsystem von Linux (Abschnitt 2.2.2) erfolgt. Daher muß ein *TTY*-Treiber entwickelt werden, der den normalen Consolen-Treiber ersetzt und Ausgaben an die Darstellungskomponente weiterleitet. Für L4-Tasks muß eine Bibliothek (`libl4console`) zur Verfügung gestellt werden. Diese Bibliothek spricht die Darstellungskomponente über die gleiche Schnittstelle an wie der *TTY*-Treiber und bietet deren Funktionalität über ein eigenes API an. Abbildung 3.7 zeigt die daraus resultierende Timesharing-Schnittstelle.

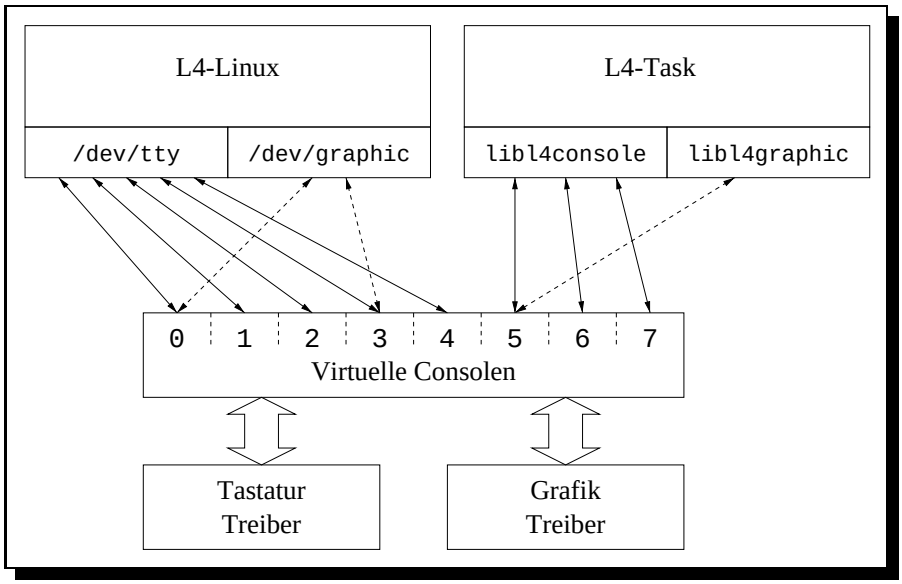


Abbildung 3.7: Timesharing-Schnittstelle zur Ein- und Ausgabe

Analog dazu erfolgt der Zugriff auf die Grafik-Funktionen über die Geräteschnittstelle `/dev/graphic` (Abschnitt 2.3.3), die durch das GGI-Projekt definiert wurde. Dafür muß die `/dev/graphic`-Schnittstelle von GGI-Linux in den L^{4}Linux-Kern übernommen werden und so angepaßt werden, daß die eigentlichen Funktionen über IPC in der Darstellungskomponente aufgerufen werden. Für L4-Tasks muß diese Schnittstelle wieder durch eine Bibliothek (`libl4graphic`) bereitgestellt werden.

Zur Ausgabe von Grafik müssen von `libl4graphic` mindestens die folgenden Funktionen bereitgestellt werden:

- Öffnen und Schließen der Verbindung zur EZDK

```
int l4_graphic_open(int vt);
int l4_graphic_close(int vt);
```

- Setzen des Grafikmodus und der Farbtiefe sowie das Einblenden des Grafikspeichers in den Adreßraum der Applikation.

```
int l4_graphic_set_mode(int vt,
                        int width, int height,
                        int vwidth, int vheight,
                        int depth);
void * l4_graphic_mmap(int vt);
```

- Ausgabe von Zeichen und Bildpunkten

```
void l4_graphic_puts(int vt, int x, int y, const char *text);
void l4_graphic_putpixel(int vt, int x, int y, dword_t col);
```

Für eine flexible und unkomplizierte Nutzung der Möglichkeiten der EZDK wäre eine Portierung der Bibliothek des GGI-Projektes (LibGGI) notwendig. Die Bibliothek benutzt jedoch in umfangreichem Maße die Möglichkeiten eines dynamischen Linkers. So werden die in Abschnitt 2.3.5 beschrieben Targets, die die eigentlichen Funktionen zur Grafikausgabe enthalten, erst zur Laufzeit eines Programmes ausgewählt und geladen. Aufgrund des hohen Arbeitsaufwandes, diese Struktur in eine statische Bibliothek umzusetzen bzw. einen dynamischen Linker unter L4 zu implementieren, kann eine Portierung im Rahmen dieser Arbeit nicht erfolgen.

Kapitel 4

Implementierung

Eine wesentliche Erleichterung für die Implementierung der Darstellungskomponente für DROPS ist die freie Verfügbarkeit der für das GGI-Projekt entwickelten Grafikkarten-Treiber. Um einen funktionsfähigen Treiber für DROPS zu erhalten, ist es dabei wichtig, die von den GGI-Grafikkarten-Treibern angebotene Schnittstelle zu nutzen. Das ermöglicht den Einsatz von unveränderten GGI-Grafikkarten-Treibern. Damit sind Weiterentwicklungen, die im Rahmen des GGI-Projektes erfolgen, auch für die Darstellungskomponente für DROPS nutzbar.

Die aktuellen Grafikkarten-Treiber des GGI-Projektes basieren auf dem Modul-Konzept des Linux-Kernes, d.h. sie liegen als Objekt-Datei vor, die die Schnittstelle für Grafikkarten-Treiber (KGI) implementieren. Der Aufruf der angebotenen Funktionen erfolgt aus dem Linux-Kern durch den KGI-Manager. Dieser fest im Linux-Kern enthaltene Manager kontrolliert und verwaltet alle Zugriffe auf die Grafikkarte.

4.1 KGI-Manager

In GGI-Linux enthält der im Kern integrierte KGI-Manager zusätzlich zu den Verwaltungs- und Zugriffsoperationen für ladbare Grafiktreiber auch minimale Treiber für VGA- und MDA-Karten. Diese Treiber existieren, da beim Booten des Kernes noch kein Grafiktreiber-Modul geladen ist, die Ausgabe von Systemmeldungen aber notwendig ist. Diese Treiber ermöglichen lediglich die Ausgabe von Text. Auf Systemen, die keine Grafikausgabe nutzen, kann dadurch auf das Laden eines speziellen Grafiktreiber-Moduls verzichtet werden, was den vom Systemkern belegten Speicher minimiert. Wird ein Grafiktreiber-Modul geladen, ersetzt dieser den im Kern enthaltenen Boot-Treiber.

Die allgemeine Struktur von GGI-Linux ist in Abbildung 4.1 dargestellt. Ein Linux-Anwendungsprogramm nutzt für den Grafikzugriff die GGI-Bibliothek (siehe Abschnitt 2.3.5). Diese greift über die UNIX-Geräteschnittstelle auf den KGI-Manager zu, der die Befehle an die verschiedenen Funktionseinheiten des Grafiktreibers weiterleitet. Als Gerät dient `/dev/graphic`.

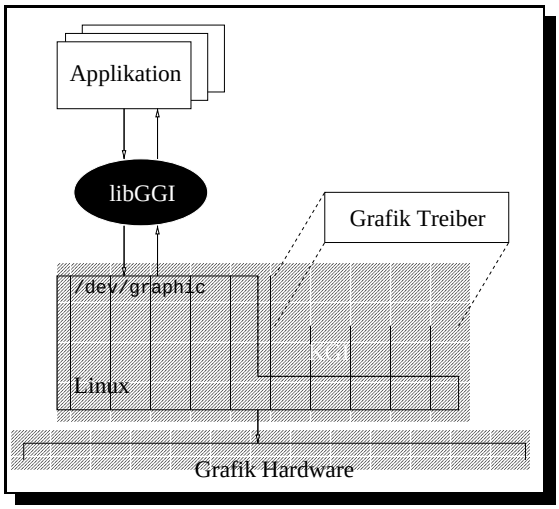


Abbildung 4.1: GGI-Linux

Daraus ist ersichtlich, daß eine Darstellungskomponente, die auf den GGI-Grafiktreibern aufbauen und diese möglichst unverändert nutzen soll, aus drei Funktionseinheiten besteht:

1. dem KGI-Manager, der aus dem Linux-Kern herausgelöst und an die L4-Umgebung angepaßt werden muß,
2. dem unveränderten GGI-Grafiktreiber (auch als Linux-Kernmodul im Binärformat) und
3. einer Linux-Umgebung, die von KGI-Manager und GGI-Grafiktreiber genutzte Kern-Funktionen emuliert und auf L4-Mechanismen abbildet.

Da das Linken des Grafiktreiber-Moduls zur Laufzeit unter L4 nicht ohne großen Aufwand implementiert werden kann, muß auf das dynamische Laden des Grafiktreibers verzichtet werden. Die Flexibilität der Implementierung wird aber dadurch nur wenig beeinträchtigt, da für verschiedene Systeme (d.h. für Systeme mit unterschiedlichen Grafikkarten) nur der jeweils notwendige Treiber-Modul statisch zur Darstellungskomponente gebunden werden muß.

Die daraus resultierende Struktur der Darstellungskomponente macht Abbildung 4.2 deutlich. Diese muß im nächsten Schritt noch um die Funktionen erweitert werden, die für die Darstellung von Datenströmen in Echtzeit notwendig sind.

Einen Überblick über die Linux-Funktionen, die von der Linux-Umgebung der Darstellungskomponente bereitgestellt werden müssen, gibt Tabelle 4.1.

Durch die Bereitstellung dieser Funktionen kann der KGI-Manager nahezu unverändert für L4 übernommen werden. Das betrifft speziell die Boot-Treiber für VGA- und MDA-Karten, den Tastaturtreiber und den Consolen-Treiber

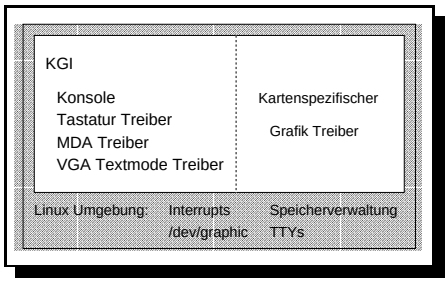


Abbildung 4.2: Struktur der EZDK

einschließlich des XTerm-kompatiblen *Console Parsers*. Die Änderungen beschränken sich auf die Funktionen der `/dev/graphic`-Schnittstelle (siehe Abschnitt 4.3.2).

Weiterhin ist es notwendig, Funktionsaufrufe vom L^{4}Linux-Kern per IPC an die Darstellungskomponente weiterzuleiten, sowie Tastatureingaben von der Darstellungskomponente an den L^{4}Linux-Kern zu senden. Dafür muß der normale Consolen-Treiber aus dem L^{4}Linux-Kern entfernt und durch einen Stub ersetzt werden (Abbildung 4.3).

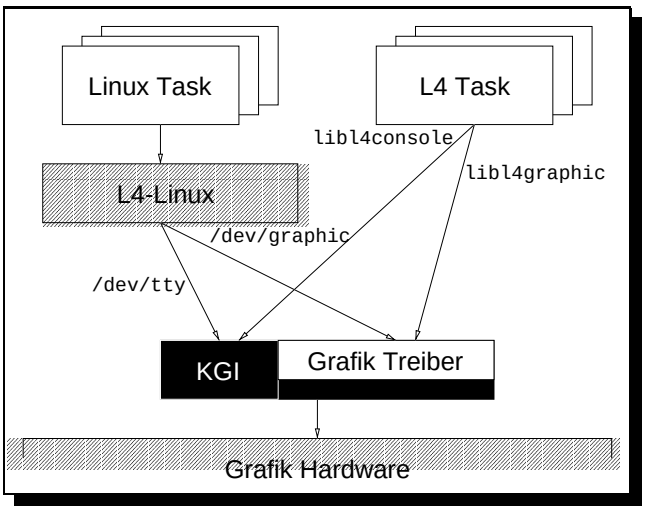


Abbildung 4.3: L^{4}Linux mit EZDK

4.2 Echtzeit-Schnittstelle

Wie bereits in Abschnitt 3.5 beschrieben, ist für eine problemlose Integration verschiedener Komponenten eine gemeinsam genutzte Bibliothek (`librt`) sinnvoll, um den Aufwand für die Implementierung neuer Komponenten möglichst gering zu halten und Fehlerquellen zu minimieren. Beschrieben werden soll hier eine Beispielimplementierung, die Abbildung 4.4 entspricht.

Linux-Funktion	L4-Mechanismus
`pci_init()` `pcibios_*()`	Nutzung des original Linux-Codes (`bios32.c`, `pci.c`) ist möglich, da dieser keine weiteren Kern-Funktionen aufruft.
`request_irq()` `do_bottom_half()`	Ein L4-Thread, der auf IPC-Nachrichten des L4-Kernes wartet, empfängt Interrupts (der Tastatur) und bearbeitet die entsprechenden Interrupt-Funktionen; ein weiterer L4-Thread behandelt die von den Interrupt-Funktionen aktivierten Bottom-Half-Funktionen.

`check_region()`
`request_region()`
`release_region()`
	Funktionen zum Reservieren und Freigeben von IO-Bereichen sind im Referenz-Handbuch definiert, werden aber in der verwendeten L4-Implementierung für Intel-x86 nicht unterstützt; Reservierungen werden daher ignoriert.

`vremap()` `vfree()`	`vremap()` dient dem Einblenden von beliebigen physischen Speicherbereichen in den Kernadreßraum; eine Abbildung auf L4 ist nur unvollständig möglich, da L4 nur 4 MB-Seiten für IO-Bereiche unterstützt.
`kmalloc()` `kfree()`	Funktionen zur normalen Anforderung von Speicher; da kein Einschränkungen wie z.B. für die Nutzung dieses Speichers als DMA-Puffer existieren, ist eine beliebige Speicherzuteilung im virtuellen Adreßraum ohne Kenntnis der physischen Lage möglich.
`tty_init()` `tty_register_driver()`	Diese Funktionen dienen der Initialisierung des *TTY*-Treibers und der Registrierung von Low-Level-Treibern.
`flush_to_ldisc()`	Diese Funktion wird periodisch aufgerufen, um von den Interrupt-Routinen im Flip-Buffer abgelegte Zeichen an die *Line Discipline* zu übertragen.

Tabelle 4.1: Benötigte Linux-Funktionen

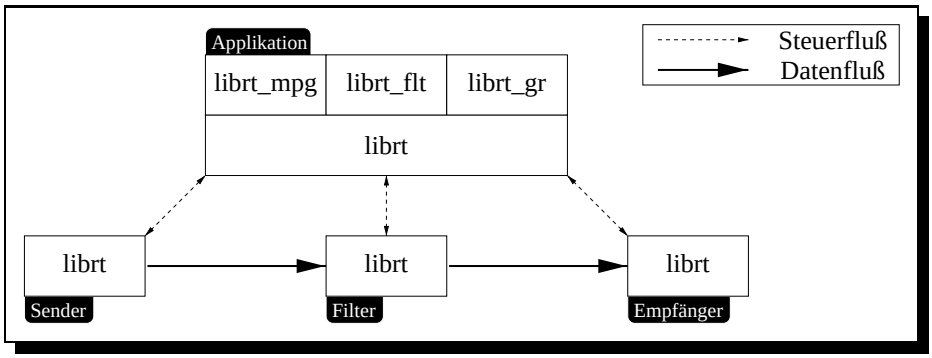


Abbildung 4.4: Beispielimplementierung für die Nutzung der Echtzeit-Schnittstelle

Das Beispiel simuliert auf einfache Art und Weise eine Videodarstellung mit einem zwischengeschalteten Filter. Da der Schwerpunkt der Implementierung bei der Untersuchung der Echtzeit-Schnittstelle liegt, kommt, um die Komplexität des Beispiels nicht unnötig hoch werden zu lassen, kein Videodekoder für ein Standardformat zum Einsatz. Stattdessen erzeugt der Sender direkt eine einfache Bildfolge. Der Filter bearbeitet für jedes Einzelbild jeden Bildpunkt nach einer einfachen Transformationsvorschrift und sendet die erzeugte Bildfolge an die Darstellungskomponente, die dann die Darstellung auf dem Bildschirm übernimmt.

Der entscheidende Unterschied zu traditionellen Systemen liegt jedoch darin, daß die Applikation in diesem Beispiel keine datenverarbeitenden Aufgaben übernimmt. Die Funktion der Applikation besteht darin, den Datenfluß zwischen den Komponenten zu etablieren und danach deren Steuerung zu übernehmen.

Für die Anzeige eines Videos muß die Applikation die folgenden Schritte ausführen:

1. Öffnen der Verbindung zum Erzeuger des Videostromes.

   ```
   fd1 = rt_mpg_open("filename");
   ```

2. Abfragen der Parameter des Videostromes.

   ```
   rt_mpg_get_param(fd1, &mpg_params);
   ```

3. Öffnen der Verbindung zum Filter.

   ```
   fd2 = rt_flt_open(mpg_params.width, mpg_params.height,
                     mpg_params.rate);
   ```

4. Abfragen der Parameter des gefilterten Videostromes.

   ```
   rt_flt_get_param(fd2, &flt_params);
   ```

5. Öffnen der Verbindung zur Darstellungskomponente.

```
fd3 = rt_gr_open(flt_params.width, flt_params.height,
                 flt_params.rate);
```

6. Verbinden der Komponenten zu einem Datenstrom.

```
rt_connect(fd1, fd2);
rt_connect(fd2, fd3);
```

Nach dem Verbinden der Komponenten funktioniert die Darstellung des simulierten Videos ohne Beteiligung der Applikation.

4.3 Timesharing-Schnittstelle

Die modulare Struktur von Linux erleichtert das Ersetzen von Treibern erheblich, da diese praktisch vollständig gekapselt sind und nur über eine genau definierte Schnittstelle angesprochen werden.

4.3.1 Consolen-Treiber

Im Fall von *TTY*-Treibern existieren sogar zwei aufeinander aufbauende Abstraktionsebenen.

Ebene 1 ist die für Anwendungsprogramme sichtbare Geräteschnittstelle, d.h. ein Eintrag im `/dev`-Verzeichnis. Im Normalfall existieren dort mindestens zwei verschiedene Gruppen von Einträgen, die jedoch beide als *TTY* betrachtet werden. Dies sind `/dev/tty0` bis `/dev/tty63` für lokale Consolen und `/dev/ttyS0` bis `/dev/ttyS3` für über eine serielle Schnittstelle angeschlossene Terminals.

Beide Gerätegruppen werden in vielen Aspekten gleich behandelt und werden deshalb in Linux durch einen gemeinsamen Gerätetreiber behandelt (`drivers/char/tty_io.c`). Die Operationen jedoch, die die eigentliche Ansteuerung der Hardware übernehmen, werden an die tieferliegende Abstraktionsebene weitergeleitet.

Ebene 2 stellt den eigentlichen Hardware-Treiber für *TTY*-Geräte dar. Für lokale Consolen erfolgt dabei der Zugriff auf die Grafikkarte (`drivers/char/console.c`) und für serielle Terminals der Zugriff auf die serielle Schnittstelle (`drivers/char/serial.c`)

Diese Hardware-Treiber registrieren sich beim Systemstart über die Funktion `tty_register_driver()` bei der höher liegenden Ebene, indem sie dieser eine Struktur mit Funktionszeigern (`struct tty_driver`) übergeben.

Das Ersetzen des Consolen-Treibers besteht dadurch aus dem Entfernen des originalen Treibers und der Implementierung der Stub-Funktionen. Diese müssen dann in eine Struktur vom Typ `struct tty_driver` eingetragen werden. Außerdem muß noch die Funktion `con_init()` angepaßt werden, die die Registrierung des Consolen-Treibers beim *TTY*-Treiber vornimmt.

4.3.2 Grafik-Schnittstelle

Für die `/dev/graphic`-Schnittstelle existiert im Standard-Linux-Kern und damit auch in L^{4}Linux kein Treiber. Im einfachsten Fall wäre es also möglich, einen Treiber zu schreiben, der lediglich alle Operationen an die Darstellungskomponente weiterleitet.

Die Geräteschnittstelle ist durch die in `linux/fs.h` definierte Struktur `struct file_operations` festgelegt:

```
struct file_operations {
  int (*lseek) (struct inode *, struct file *, off_t, int);
  int (*read) (struct inode *, struct file *, char *, int);
  int (*write) (struct inode *, struct file *, const char *, int);
  int (*readdir) (struct inode *, struct file *, void *, filldir_t);
  int (*select) (struct inode *, struct file *, int, select_table *);
  int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

  int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  void (*release) (struct inode *, struct file *);
  int (*fsync) (struct inode *, struct file *);
  int (*fasync) (struct inode *, struct file *, int);
  int (*check_media_change) (kdev_t dev);
  int (*revalidate) (kdev_t dev);
};
```

Diese Struktur definiert alle Funktionen für *Character*-Gerätetreiber. Ein Treiber muß jedoch nur die Funktionen implementieren, die für das entsprechende Gerät notwendig bzw. sinnvoll sind. Im Fall des `/dev/graphic`-Treibers sind das `open`, `release`, `mmap`, `read`, `select` und `ioctl`.

Der GGI-Treiber implementiert jedoch zusätzlich auch Operationen für den Zugriff auf den Grafikspeicher (`struct vm_operations_struct`).

`graph_mmio_nopage()` wird aufgerufen, wenn der erste Zugriff auf den Grafikspeicher erfolgt. In diesem Fall wird nicht nur die entsprechende Seite in den virtuellen Adreßraum der Task eingeblendet, sondern der komplette Bereich des Grafikspeichers, auf den diese Task zugreifen darf. Dadurch wird verhindert, daß beim Beschreiben des Grafikspeichers für jede Seite einzeln Seitenfehler auftreten.

`graph_mmio_unmap()` stellt sicher, daß der Grafikspeicher auch komplett wieder ausgeblendet wird.

Analog dazu existieren Funktionen zum Ein- und Ausblenden von sogenannten *Ping-Pong-Puffern* (`graph_accel_nopage()` und `graph_accel_unmap()`). Diese dienen einer effizienten Datenübertragung vom Nutzeradreßraum in den Kernadreßraum für die Ansteuerung von Beschleuniger-Funktionen der Grafikkarte. Die Standard-Schnittstelle definiert für die Kommunikation der Bibliothek im Nutzeradreßraum mit dem Kerntreiber `ioctl`-Systemrufe. Dadurch wären z.B. Funktionen zum Zeichnen von Dreiecken über die Hardware der Grafikkarte extrem langsam, da für jede Übergabe von Daten ein Systemruf durchgeführt werden müßte. Stattdessen existieren die *Ping-Pong-Puffer*, die durch von Kerntreiber und Bibliothek gemeinsam genutzten Speicherbereich implementiert sind. Die Bibliothek schreibt eine Reihe von Aufträgen an die Grafikkarte in einen freien Puffer und übergibt diesen mit einem `ioctl`-Systemruf an den Kerntreiber.

Diese Funktionen müssen in an den L^{4}Linux-Kern angepaßter Form implementiert werden, um eine kompatible `/dev/graphic`-Schnittstelle für L^{4}Linux anbieten zu können. Aus diesem Grund müssen auch im Stub-Treiber in L^{4}Linux Statusinformationen geführt werden, um eine korrekte Speicherverwaltung zu ermöglichen.

4.4 X-Server

Die in Abschnitt 3.2 gemachte Aussage zur direkten Nutzung des GGI-X-Servers ist zwar korrekt, es existiert aber ein Problem bei der Implementierung der `/dev/graphic`-Schnittstelle, welches nicht sofort offensichtlich ist. Die für L^{4}Linux angepaßte `/dev/graphic`-Schnittstelle ermöglicht den direkten Zugriff des X-Servers auf die von der EZDK verwaltete Grafik-Hardware. Diese Schnittstelle ermöglicht jedoch in GGI-Linux dem X-Server auch den Zugriff auf die von der Maus gesendeten Daten. Dazu existiert die in Abschnitt 2.2.2 erwähnte *Line Discipline* (mit dem Namen `N_MOUSE`) für die serielle Schnittstelle, die das herstellerabhängige Protokoll der Maus in standardisierte Ereignisse umwandelt und diese an den KGI-Manager im Kern weiterleitet. Dazu registriert sich die *Line Discipline* bei der Initialisierung beim KGI-Manager als Eingabegerät.

Bei Einsatz der EZDK und einem X-Server in L^{4}Linux ist diese Struktur nicht ohne weiteres nutzbar, da sich der KGI-Manager in der EZDK befindet und keinen Zugriff auf einen Treiber für eine serielle Schnittstelle hat. Der GGI-Maustreiber ist aber als *Line Discipline* für die serielle Schnittstelle nur in L^{4}Linux unverändert einsetzbar.

Die implementierte Lösung des Problems ist aus diesen Gründen etwas unübersichtlich, ermöglicht aber den Einsatz des unveränderten GGI Maustreibers ohne Implementierung eines neuen, nur auf L4 basierenden Treibers für die serielle Schnittstelle. Für diesen wäre dann wieder ein Stub-Treiber in L^{4}Linux notwendig, um im L^{4}Linux auf die serielle Schnittstelle zugreifen zu können.

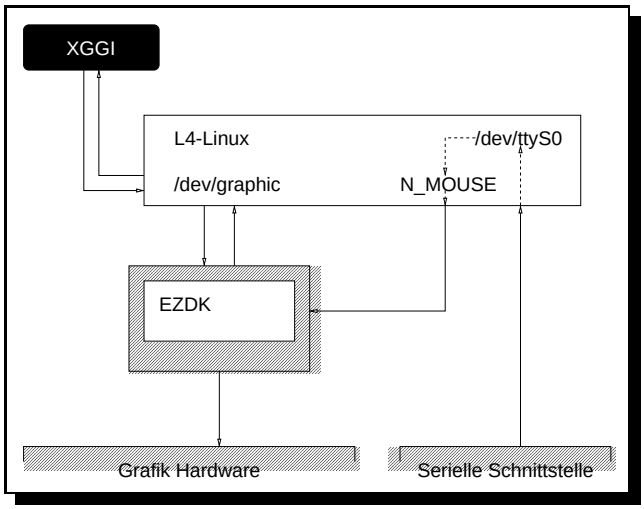


Abbildung 4.5: Unterstützung des GGI-X-Server

Der *Line Discipline*-Modul nutzt lediglich drei Funktionen des KGI-Managers. Alle anderen vom Modul genutzten Funktionen sind im L^{4}Linux-Kern vorhanden:

`input_register()` zum Registrieren eines Eingabegerätes,

`input_unregister()` zum Freigeben aller Ressourcen, die für das Eingabegerät angefordert wurden und

`input_handle_event()` zum Übergeben von Ereignissen, die vom Eingabegerät erzeugt werden, an den KGI-Manager.

Diese Funktionen sind so implementiert, daß sie ihre Parameter über IPC an die EZDK weiterleiten und zum *Line Discipline*-Modul hinzugebunden werden können. Dadurch kann dieser Modul den im L^{4}Linux-Kern enthaltenen seriellen Treiber nutzen und die erzeugten Ereignisse an die EZDK weiterleiten. Die EZDK stellt diese Ereignisse dann dem X-Server über die `/dev/graphic`-Schnittstelle bereit (Abbildung 4.5).

Kapitel 5

Leistungsbewertung

Zur Untersuchung, ob die Zielstellung der echtzeitfähigen Videodarstellung erreicht werden konnte, dient die in Abschnitt 4.2 vorgestellte Beispielimplementierung mit drei Komponenten, die über die entwickelte Echtzeit-Schnittstelle gekoppelt sind (Abbildung 4.4).

5.1 Systemvoraussetzungen

Die Messungen erfolgten auf einem Standard-PC mit einem 133 MHz Pentium Prozessor und einer Grafikkarte miro 40 SV ergo, die 4 MB Grafikspeicher und ein PCI-Interface besitzt.

Alle Zeitmessungen wurden mit dem im Prozessor integrierten *Time-Stamp-Counter* (*TSC*) durchgeführt, was eine hohe Zuverlässigkeit für die gemessenen Werte ergibt. Der *TSC* wird vom Prozessor bei jedem Takt inkrementiert, was beim genutzten Testsystem bedeutet, daß er pro Mikrosekunde 133 mal erhöht wird. Zum Auslesen des *TSC* kann der Maschinenbefehl `rdtsc` verwendet werden, der den aktuellen Wert des *TSC* als 64 bit Zahl liefert.

5.1.1 Bibliothek für Messungen

Die Messungen erfolgten einheitlich über eine Bibliothek, die die folgende Schnittstelle anbietet:

```
rt_time_init(int nr, long loops, char *name);
rt_time_start_region(int nr);
rt_time_end_region(int nr);
```

Die bei `rt_time_init()` angegebene Anzahl von Durchläufen definiert, nach wieviel Aufrufen von `rt_time_end_region()` die gemessenen Werte angezeigt bzw. gespeichert werden.

Weiterhin ist es möglich, die gemessenen Werte an einen speziellen Thread weiterzuleiten, statt sie auf dem Bildschirm ausgeben zu lassen. Davon wurde für die im folgenden beschriebenen Messungen Gebrauch gemacht.

Dazu läuft in L^{4}Linux ein Programm, das sich selbst beim Nameserver als `"time-log"` registriert und die von der Bibliothek gemessenen Werte per IPC empfängt, speichert und in einer Datei ablegt (Abbildung 5.1).

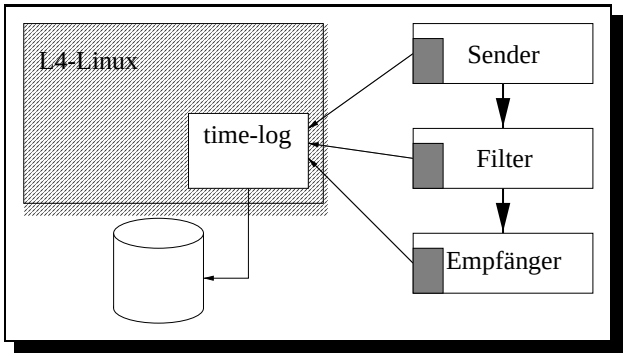


Abbildung 5.1: Meßanordnung

Um die Messungen nicht durch den Zugriffe des Log-Threads auf die Festplatte zu verfälschen, werden alle empfangenen Daten vorerst im Speicher gehalten und erst beim Beenden des Programmes ausgegeben.

5.1.2 Scheduler

Der eingesetzte User-Level Scheduler basiert auf dem *Programmierbaren Intervall Timer* (*PIT*), der Interrupts mit einer einstellbaren Rate auslösen kann. Diese werden vom L4-Kern in IPC-Nachrichten umgewandelt und dem Scheduler zugestellt. Dieser aktiviert dann den Thread, der die aktuelle Zeitscheibe reserviert hat.

Im Beispiel erzeugt der *PIT* Interrupts mit einer Rate von 1200 Hz. Die Periodendauer für das Scheduling ist für alle Threads auf 16.6 ms eingestellt, von der sich jeder Thread ein Vielfaches von 833 μs (d.h. $\frac{1}{20}$ der Periode) reservieren kann.

5.1.3 Meßwerte

Durch die Bibliothek werden pro Thread acht Werte gemessen, die am Beispiel der (um Fehlerabfragen vereinfachten) Bearbeitungsfunktion des Filters erläutert werden sollen.

```
rt_time_init(7, 1000, "Filter");

while (242) {
```

```
  begin_region();          /* warten auf Aktivierung durch Scheduler */

  rt_time_start_region(7);                      /* Beginn der Messung */

  rt_istream_get_buffer(...);          /* Holen des Empfangs-Puffers */
  rt_ostream_get_buffer(...);           /* Holen des Ausgabe-Puffers */
  do_fill_buffer(...);                          /* Filtern der Daten */
  rt_ostream_commit_buffer(...); /* Gültigmachen des Ausgabe-Puffers */

  rt_time_end_region(7);                         /* Ende der Messung */
}
```

Im Normalfall wird die Funktion `begin_region()` in konstanten Zeitabständen verlassen, die der beim Scheduler angeforderten Rate entsprechen. Im Beispiel sollen dafür 60 Hz angenommen werden. Das bedeutet, daß die Schleife alle 1/60 s bzw. 16.6 ms einmal durchlaufen wird.

Gemessen werden die Abstände zwischen aufeinanderfolgenden Aufrufen von `rt_time_start_region()` wodurch die korrekte Aktivierung durch den Scheduler kontrolliert werden kann. Zusätzlich werden für jeweils die in `rt_time_init()` angegebene Anzahl von Durchläufen der Maximal- und Minimal-Wert gespeichert sowie der Durchschnitt über alle Aufrufe berechnet.

Ab der zweiten Gruppe von Messungen wird der beim vorherigen Durchlauf bestimmte Mittelwert für eine Zählung der auftretenden Scheduling-Fehler benutzt. Dafür wird der eineinhalbfache Mittelwert gespeichert und alle neuen Meßwerte gezählt, die über diesem Wert liegen.

Ausgegeben werden nach der spezifizierten Anzahl von Durchläufen der Minimal-, Maximal- und Durchschnittswert sowie die Anzahl der Fehler.

Analog dazu werden die entsprechenden Werte für die Zeit zwischen dem Aufruf von `rt_time_start_region()` und `rt_time_end_region()` gemessen, Minimal- und Maximalwert bestimmt und Durchschnittswert sowie Fehleranzahl berechnet. Über diese Meßwerte kann überprüft werden, ob der entsprechende Thread die für ihn reservierte CPU-Zeit überschreitet und wenn ja, wie oft.

5.1.4 Benchmarks

Um die Messungen unter Last auf dem Timesharing System durchzuführen, kamen fünf Teilbenchmarks der `lmbench` Benchmark-Suite [MS96] zum Einsatz. Die Version der Benchmark-Suite ist `lmbench-2alpha11`.

Die Benchmarks bestimmen Bandbreiten und Latenzzeiten für die Übertragung von Daten zwischen zwei Prozessen über Standard-UNIX-Mechanismen bzw. die Latenzzeit für die Erzeugung von Prozessen.

Benchmark	Mechanismus	Einheit
`bw_pipe`	Pipe	MB/s
`lat_pipe`	Pipe	μs
`bw_unix`	UNIX-Domain-Sockets	MB/s
`lat_unix`	UNIX-Domain-Sockets	μs
`lat_proc`	`fork` und `execve`	μs

Pipe – Bandbreite

`bw_pipe` erzeugt eine UNIX-Pipe zwischen zwei Prozessen und transferiert 10 MB durch diese Pipe. Die Blockgröße für die Übertragung beträgt 64 Kb. Dabei ist zu beachten, daß die internen Puffer einer Pipe normalerweise kleiner als 64 Kb sind.

RCS-Id: `$Id: s.bw_pipe.c 1.10 97/06/25 10:25:01-07:00 lm $`

Pipe – Latenzzeit

`lat_pipe` nutzt zwei Prozesse, die über eine UNIX-Pipe kommunizieren, um die Latenzzeiten der Interprozeß-Kommunikation zu messen. Der Benchmark übergibt ein Zeichen von einem Prozeß zum anderen und zurück. Die Prozesse führen keine weiteren Operationen aus.

RCS-Id: `$Id: s.lat_pipe.c 1.8 97/06/15 22:38:58-07:00 lm $`

UNIX-Domain-Sockets – Bandbreite

`bw_unix` entspricht `bw_pipe` mit dem Unterschied, daß zur Kommunikation UNIX-Domain-Sockets verwendet werden.

RCS-Id: `$Id: s.bw_unix.c 1.9 97/06/25 10:25:01-07:00 lm $`

UNIX-Domain-Sockets – Latenzzeit

`lat_unix` entspricht `lat_pipe` mit dem Unterschied, daß die Latenzzeit für UNIX-Domain-Sockets bestimmt wird.

RCS-Id: `$Id: s.lat_unix.c 1.6 97/06/15 22:38:58-07:00 lm $`

Prozeßerzeugung – Latenzzeit

`lat_proc` mißt die Latenzzeit für das Erzeugen neuer Prozesse. Zum Einsatz kommt hier die Messung von `fork+execve`. Dabei wird die Zeit gemessen, die benötigt wird, um einen neuen Prozeß zu erzeugen (`fork`) und ein neues Programm zu starten (`execve`). Diese Befehlsfolge bildet die innere Schleife in Kommandozeilen-Interpretern.

RCS-Id: `$Id: s.lat_proc.c 1.13 97/06/15 22:38:58-07:00 lm $`

5.1.5 Datenstrom

Der Beispiel-Datenstrom besteht aus 128x128 Bildpunkten großen Einzelbildern mit je 1 Byte pro Bildpunkt, d.h. ein Bild beansprucht 16384 Bytes. Damit kann jedes Bild in exakt 4 Speicherseiten untergebracht werden. Diese können zusammen als eine *Flexpage* per IPC über Task-Grenzen transferiert werden.

Um die in den einzelnen Tasks durchgeführten Operationen zu verdeutlichen, sollen die inneren Schleifen der am Echtzeit-Datenstrom beteiligten Tasks kurz vorgestellt werden.

Sender

Hier erfolgt das Löschen des Speicherbereiches für das jeweils aktuelle Einzelbild sowie das Erzeugen des Bildes über eine einfache Formel (ähnlich den Lissajous-Figuren [OR98]). Die benötigte Rechenzeit beträgt im Mittel 279 μs (Abbildung 5.2, Diagramm 1). Beim Scheduler wurden 833 μs CPU-Zeit reserviert.

```
/*
 *  löschen des bildes
 */
for (a = 0;a < (width * height) / 4;a++) {
  ((dword_t *)buffer_pointer)[a] = 0;
}

/*
 *  erzeugen des bildes
 */
for (a = 0;a < 1024;a += 6) {
  x = x_scale * desc->x_table[(desc->x_inc + a) % MAX_POINTS];
  y = y_scale * desc->y_table[(desc->y_inc + a) % MAX_POINTS];
  *(addr + desc->width * y + x) = col;
}
```

Filter

Der Filter bearbeitet die Bildpunkte so, daß der Wertebereich für die Farbwerte eingeschränkt wird. Da der Sender nur schwarze (Farbwert = 0) und weiße (Farbwert = 255) Punkte erzeugt, wird dadurch praktisch der Kontrast vermindert. Die benötigte Rechenzeit beträgt im Mittel 301 μs (Abbildung 5.3, Diagramm 1). Beim Scheduler wurden 833 μs CPU-Zeit reserviert.

```
/*
 *  filtern des bildes
 */
for (a = 0;a < (width * height) / 4;a++) {
  *(daddr + a) = *(saddr + a) & 0x7f7f7f7f | 0x03030303;
}
```

Empfänger

Im Empfänger werden die Bildpunkte von 8 bit auf 24 bit expandiert, indem einfach der 8-Bit-Wert für alle Farbanteile (Rot, Grün und Blau) auf diesen Wert gesetzt werden. Daraus resultiert ein Graustufenbild mit theoretisch 256 Graustufen. Die benötigte Rechenzeit beträgt im Mittel 2011 μs (Abbildung 5.4, Diagramm 1). Beim Scheduler wurden 2499 μs CPU-Zeit reserviert.

```
/*
 *  expandieren auf 24 bit farbtiefe (graustufen) und anzeigen
 */
for (a = 0;a < width * height;a++) {
  *(daddr + a) = *(saddr + a) * 0x010101;
}
```

5.2 Durchführung

Die Messungen erfolgten in einem Zeitraum von 14 Stunden mit unterschiedlicher Systemlast sowohl im Timesharing- als auch im Echtzeit-Teil (Tabelle 5.1). Die in Abschnitt 5.3 angegebenen Werte beziehen sich immer auf den ersten Datenstrom. Dieser entspricht Abbildung 5.1, d.h. es existieren eine Sende-, eine Filter- und eine Empfänger-Task. Die Übertragung der Daten erfolgt über die in Abschnitt 3.5 definierte Echtzeit-Schnittstelle. Der zweite Datenstrom, der den Einfluß mehrerer Echtzeit-Datenströme auf die Stabilität der Ergebnisse zeigen soll, besteht nur aus Sende- und Empfangs-Task. Alle anderen Parameter entsprechen dem ersten Datenstrom.

Zeit (in h)	Benchmark	Strom 1	Strom 2
0	`bw_unix`		
1	`bw_pipe`		
2	`lat_unix`		
3	`lat_pipe`		
4	-	√	
5	`bw_unix`	√	
6	`bw_pipe`	√	
7	`lat_unix`	√	
8	`lat_pipe`	√	
9	-	√	√
10	`bw_unix`	√	√
11	`bw_pipe`	√	√
12	`lat_unix`	√	√
13	`lat_pipe`	√	√

Tabelle 5.1: Zeitablauf der Messungen (1)

Messungen mit der gesamten Benchmark-Suite konnten aufgrund eines Problems der derzeitigen Implementierung des μ-Kernes nicht erfolgen. Beim Löschen einer Task muß auch der komplette Adreßraum der Task gelöscht werden. Der L4-Kern sperrt während dieser Zeit alle Interrupts, daher erfolgt für diese Zeit im L4-Kern kein Scheduling und der für die Steuerung des Echtzeit-Systems notwendige User-Level-Scheduler kann nicht korrekt arbeiten. Um dieses Problem deutlich zu machen, wurden zusätzlich Messungen durchgeführt, bei denen parallel zu einem Echtzeit-Datenstrom der Benchmark `lat_proc` lief (Tabelle 5.2).

Zeit (in h)	Benchmark	Strom 1
0	`lat_proc`	
1	-	$\surd$
2	`lat_proc`	$\surd$

Tabelle 5.2: Zeitablauf der Messungen (2)

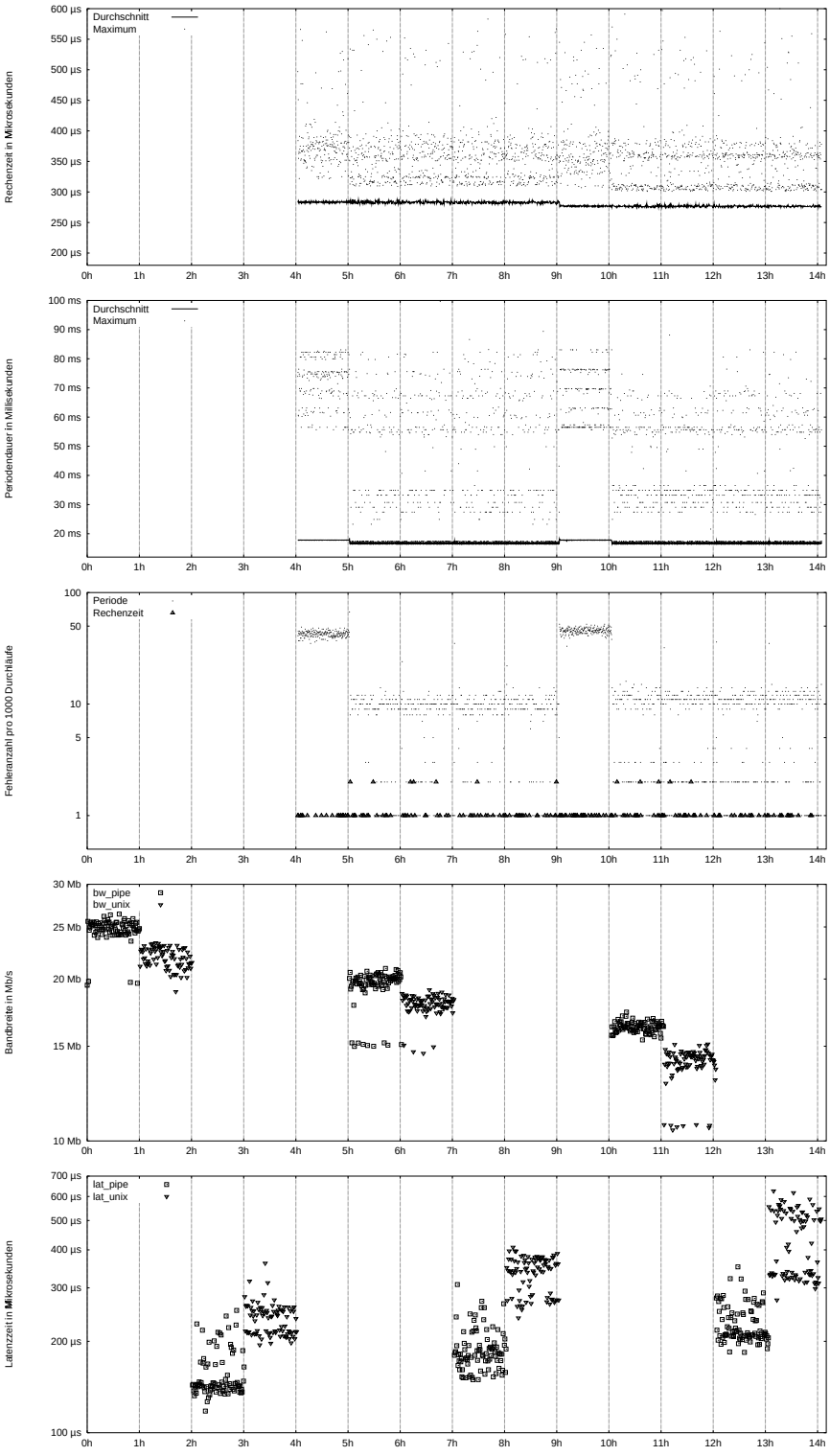
5.3 Ergebnisse

Die Ergebnisse der Messungen sind so zusammengestellt, daß ein einfacher Vergleich von Resultaten jeweils einer Echtzeit-Task und der Werte der Benchmarks im Timesharing-System möglich ist. Die Diagramme stellen folgende Werte dar (von oben nach unten):

1. Die von der Echtzeit-Task benötigte Rechenzeit pro Aktivierung durch den Scheduler. (Durchschnittswerte über 1000 Messungen und Maximalwert während dieser 1000 Messungen)
2. Die Zeitdifferenzen zwischen zwei Aktivierungen der Echtzeit-Task durch den Scheduler (Normalwert: 16.6 ms).
3. Die Anzahl der Zeitüberschreitungen (Zeitdauer größer dem 1.5-fachen des Normalwertes) bei Rechenzeit und der reservierten Periodendauer.
4. Die Werte der `lmbench`-Benchmarks zur Messung der Bandbreite (`bw_pipe` und `bw_unix`).
5. Die Werte der `lmbench`-Benchmarks zur Bestimmung der Latenzzeit der Interprozeß-Kommunikation (`lat_pipe` und `lat_unix`).

5.4 Analyse

Die Betrachtung der Meßwerte für die Rechenzeiten der Echtzeit-Tasks zeigt, daß in nahezu allen Fällen die Zeitschranken eingehalten werden. Insbesondere



(1)

(2)

(3)

(4)

(5)

Abbildung 5.2: Sender (1)

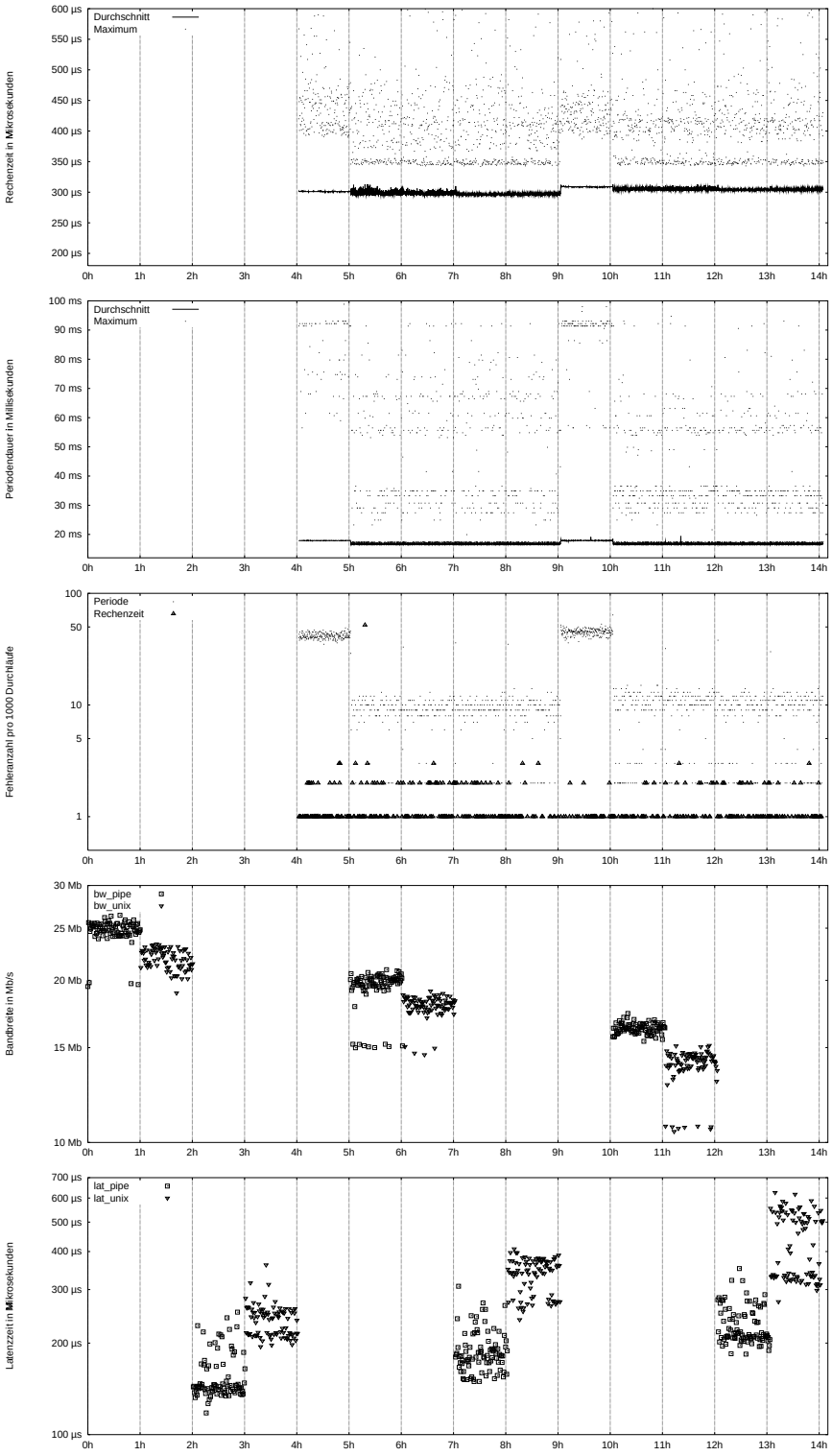


Abbildung 5.3: Filter (1)

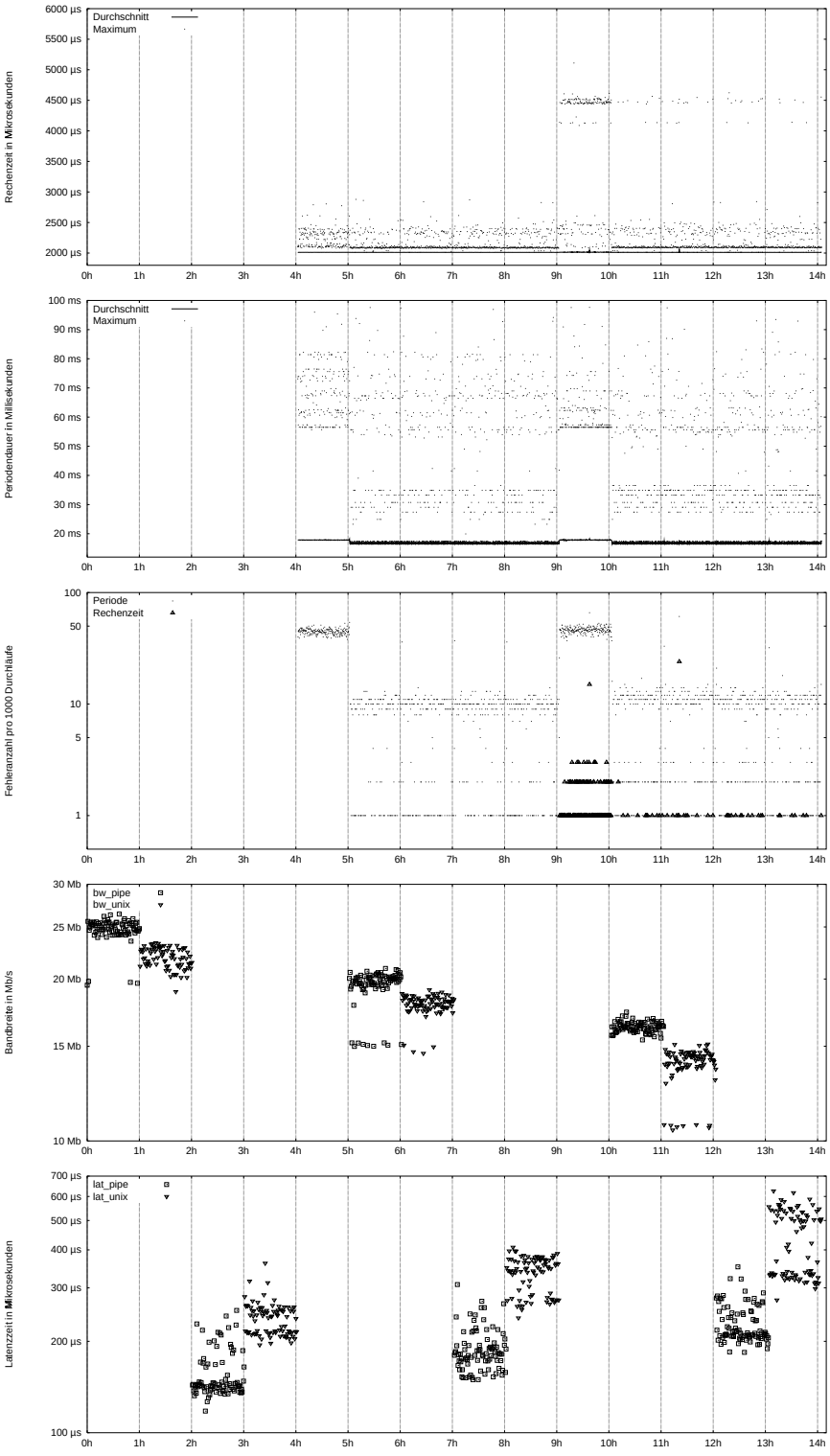


Abbildung 5.4: Empfänger (1)

der direkte Vergleich mit den Werten für gleichzeitig laufende Benchmarks im Timesharing-System macht deutlich, daß durch die Reservierungen das Echtzeit-System stabil läuft, auch wenn sich die Gesamtlast im Systems verändert.

Bei näherer Untersuchung der Ergebnisse des `bw_pipe`-Benchmarks (Start bei Stunde 0, 5 und 10) zeigt sich, daß die Bandbreite der Übertragung ohne Last im Echtzeit-System im Mittel bei 24.7 MB liegt. Nach Starten des ersten Echtzeit-Datenstromes sinkt die Bandbreite auf 19.4 MB und sinkt nach dem Starten des zweiten Echtzeit-Datenstromes weiter ab auf 16.3 MB.

Die Werte für die Filter-Echtzeit-Task schwanken in diesen Zeiträumen (ab Stunde 5 und 10; Dauer 1h) jedoch praktisch nicht. Die Mittelwerte für diese Zeitabschnitte betragen 301 μs bzw. 305 μs für die Rechenzeit und 17.9 ms bzw. 16.8 ms für die Periodendauer.

Durch die Reservierung von CPU-Zeit kann das Echtzeit-System die geforderten Zeitschranken einhalten, wobei eine höhere Prozessorauslastung nur das Timesharing-System beeinflußt.

Nicht erklärbar sind die hohen Fehleranzahlen beim Scheduling für die Zeiten, während denen im Timesharing-System kein Benchmark lief. Zu bemerken ist jedoch, daß das Problem nicht durch Überlastung des Prozessors erklärt werden kann. Die Anzahl der Überschreitungen der Rechenzeit liegt auch in diesen Zeiträumen bei maximal 2. Daraus kann geschlußfolgert werden, daß die Aktivierung der Echtzeit-Tasks nicht zu den vorgesehenen Zeitpunkten erfolgt. Wurde die Task jedoch aktiviert, dann erhält sie auch die reservierte CPU-Zeit und kann ihre Zeitschranke für die Rechenzeit einhalten.

Eine Erhöhung der Last im Timesharing-System hat also im Normalfall keinen Einfluß auf das Echtzeit-System. Das ist jedoch nicht in allen Fällen so. Aufgrund der Eigenschaft des L4-Kernes, beim Löschen von Tasks alle Interrupts zu sperren, kommt es zu massiven Auswirkungen des Timesharing-Systems auf Echtzeit-Tasks. Das kann z.B. beobachtet werden, wenn der L^{4}Linux-Kern häufig Prozesse erzeugt und beendet. Durch die Interruptsperre erhält der Scheduler keine Interrupts des PIT und kann die Echtzeit-Tasks nicht zum korrekten Zeitpunkt aktivieren. Deutlich wird das in Abbildung 5.5. Diese zeigt den Einfluß des Benchmarks `lat_proc` auf die Filter-Task. Die Parameter und die Diagramme 1-3 des Echtzeit-Datenstrom entsprechen denen des ersten Datenstromes der vorherigen Messungen. Diagramm A stellt die Latenzzeit für die Prozeßerzeugung dar. Diagramm B ist eine Ausschnittsvergrößerung von Diagramm 2 und verdeutlicht, daß sogar die durchschnittliche Periodendauer über 1000 Einzelmessungen extrem schwankt, wenn im Timesharing-System Prozesse erzeugt und beendet werden. Die Mittelwerte für die beiden interessanten Zeitabschnitte betragen jeweils 301 μs für die Rechenzeit und 17.9 ms bzw. 25.2 ms für die Periodendauer. Das entspricht einer Vergrößerung der Periodendauer um über 40%. Auch die Maximalwerte für die Periodendauer über jeweils 1000 Einzelmessungen sind mit Werten größer als 1 Sekunde zu hoch, um eine flüssige Anzeige von Videos zu gewährleisten. Diese Werte bedeuten, daß die Filter-Task mehrfach für länger als 1 Sekunde nicht aktiviert wurde.

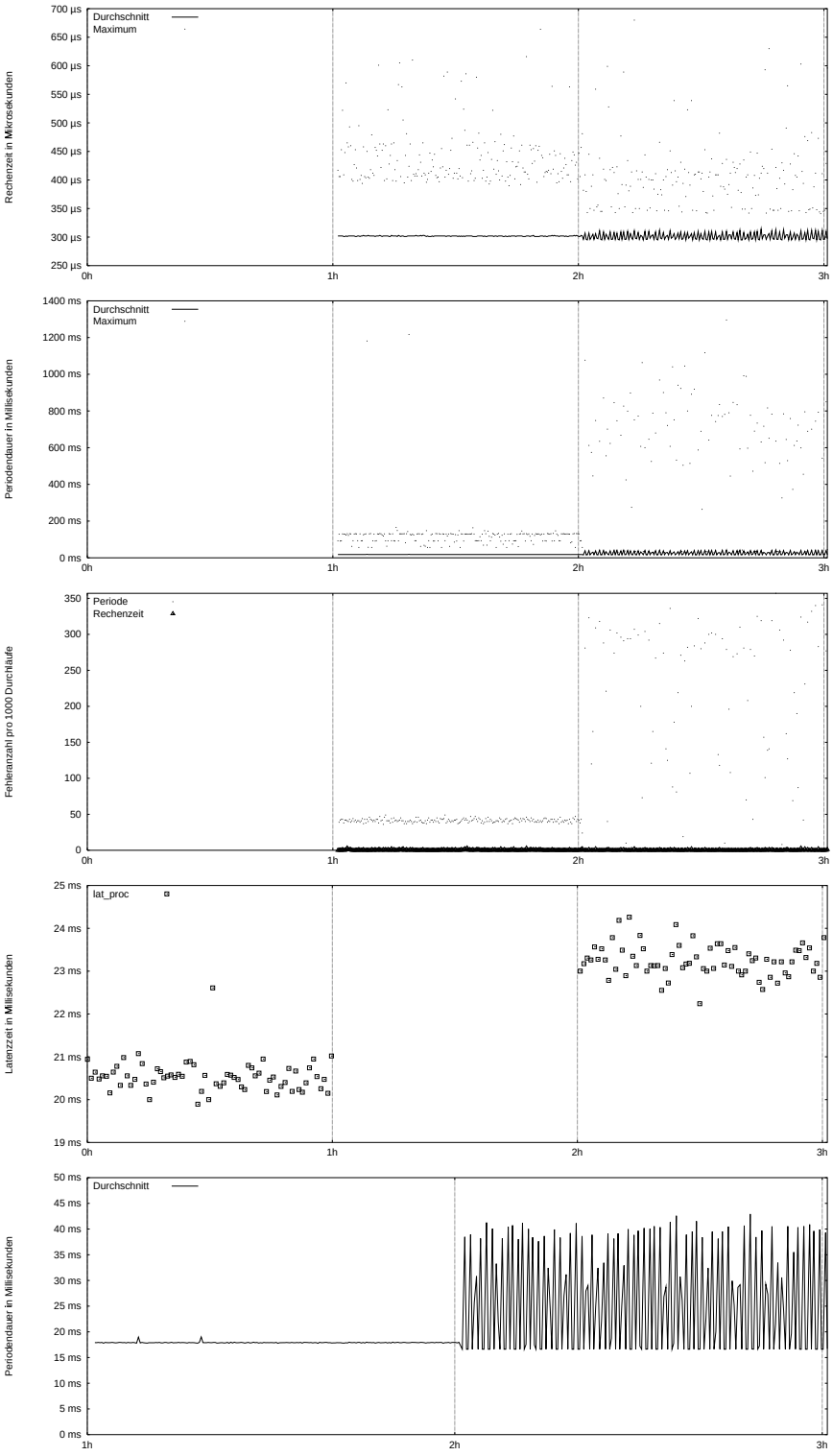


Abbildung 5.5: Filter (2)

Kapitel 6

Zusammenfassung und Ausblick

Ziel der Arbeit war der Entwurf und die Implementierung einer echtzeitfähigen Darstellungskomponente für DROPS. Nachdem verschiedene Modelle für eine Darstellungskomponente diskutiert wurden, konnte eines dieser Modelle implementiert werden.

Die dabei entstandene Darstellungskomponente für DROPS ermöglicht die gleichzeitige Nutzung eines X-Servers, der in L^4Linux läuft, und die Anzeige eines oder mehrerer Echtzeit-Datenströme. Das Ziel der Arbeit konnte damit erreicht werden.

Erweiterungen der Darstellungskomponente sind jedoch an einigen Stellen noch notwendig. So erfolgt zur Zeit kein Clipping der X-Server-Ausgaben an den Bildschirmbereichen für Echtzeit-Datenströme. Dadurch kann der X-Server auch auf diese Bereiche zugreifen und die Anzeige der Echtzeit-Datenströme beeinflussen. Außerdem erfolgte auch parallel zu dieser Arbeit eine Weiterentwicklung der GGI-Treiber, auf denen die Darstellungskomponente basiert. Insbesondere wurden Veränderungen an den Schnittstellen der Treiber vorgenommen, um eine einfache, direkte Integration in den Linux-Kern zu ermöglichen. Die dadurch entstandenen Inkompatibilitäten der aktuellen Treiber zur Darstellungskomponente müssen beseitigt werden, um weiterhin die Neuentwicklungen im Rahmen des GGI-Projektes nutzen zu können.

Für eine sinnvolle Nutzung der Darstellungskomponente ist es jetzt wichtig, einen echten Videodekoder für ein Standardformat in die Echtzeit-Umgebung, die von DROPS angeboten wird, zu integrieren. Mit einer derartigen Dekoderkomponente wären dann Untersuchungen für einen praktischen Einsatz der Darstellungskomponente möglich.

Anhang A

Glossar

Adreßraum: Menge gültiger Speicheradressen, auf die eine Task zugreifen kann.

BIOS: (Basic Input Output System) Im Festwertspeicher eines PCs enthaltenes Steuerprogramm für die Hardware.

Buffer Cache: Bereich des Hauptspeichers, der in Linux als Cache für den Zugriff auf Block-Geräte benutzt wird.

Console Parser: Modul für die Umsetzung von Steuerzeichen in vom Terminal abhängige Operationen wie Bildschirmlöschen oder Cursor-Positionierung.

Continuous-Media: Datenströme mit in der Regel hohen Bandbreiten wie z.B. Video- oder Audio-Daten. Diese Daten müssen kontinuierlich transportiert bzw. verarbeitet werden, um eine fehlerfreie Nutzung der Daten zu ermöglichen.

DMA: (Direct Memory Access) Übertragung von Daten zwischen externen Geräten und dem Hauptspeicher ohne Verwendung des Prozessors.

Flexpage: L4-Einheit zur Speichermanipulation.

Framebuffer: Direkt in den Adreßraum einer Task eingeblendeter Grafik-Speicher.

GGI: General Graphics Interface.

GLIDE: Eine Programmierschnittstelle für Grafikkarten der Firma 3dfx.

GMD: GMD – Forschungszentrum Informationstechnik GmbH, früher „Gesellschaft für Mathematik und Datenverarbeitung“.

Interrupt: Unterbrechungsanforderung durch ein externes Gerät, das dadurch die asynchrone Abarbeitung einer Interrupt-Behandlungsroutine auslöst.

IPC: (Inter Process Communication) Mechanismus zur Kommunikation zwischen Tasks (bzw. Threads).

KGI: Kernel Graphics Interface.

L4: Ein von Jochen Liedke an der GMD entwickelter Mikrokern.

Line Discipline: Teil des TTY-Treibers, der für allgemeine Bearbeitung des Zeichenstromes zuständig ist.

Linux: Ein frei verfügbarer UNIX-ähnlicher Betriebssystemkern.

Major-Nummer: Identifiziert ein Gerät im UNIX-Kern.

MDA: Monochrome Display Adapter.

Minor-Nummer: Identifiziert ein Subgerät eines Gerätetreibers.

PCI: (Peripherial Component Interconnect) Standard für den Anschluß von Geräten an ein Computersystem. Definiert die elektrischen und mechanischen Eigenschaften eines Bussystems und die Verwaltung von Konfigurationsinformationen dieser Geräte.

RAMDAC: Ein Digital-Analog-Umsetzer, der die digitalen Daten der Grafikkarte in analoge Signale für den Monitor umsetzt.

Scheduler: Teil des Betriebsystems, der festlegt, welcher Prozeß wann aktiviert wird.

Task: Ein oder mehrere Threads mit einem zugehörigem Adreßraum.

Terminal: Als Terminal (oder Console) wird ganz allgemein die Kombination eines Bildschirms mit einer Tastatur bezeichnet. Linux simuliert mehrere vt102-Terminals gleichzeitig (virtuelle Terminals).

Thread: Ein Kontrollfluß, der i.a. durch einen Befehlszähler und einen Stackzeiger gekennzeichnet ist.

VGA: Video Graphics Array, ein Grafikstandard, der 1987 von IBM für die PS/2-Systeme geschaffen wurde.

Literaturverzeichnis

[Anv96] H. Peter Anvin. Linux Allocated Devices. `devices.txt` in Linux Source Archive, 1996.

[AOG91] David P. Anderson, Yoshitomo Osawa, and Ramesh Govindan. Real-Time Disk Storage and Retrieval of Digital Audio/Video Data. Technical Report CSD-91-646, CS Division, EECS Department, University of California at Berkeley, 1991.

[B+96] William J. Bolosky et al. The Tiger Video Fileserver. Technical Report MSR-TR-96-09, Microsoft Research, Advanced Technology Division, 1996.

[BBH+98] 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.

[BH96] Robert Baumgartl and Hermann Härtig. On the Integration of DSP Hardware into a Microkernel-based Operating System. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'96)*, pages 867–872, Boston, MA, October 1996.

[BH97] Robert Baumgartl and Hermann Härtig. Efficient Communication Mechanisms for DSP-Based Multimedia Accelerators. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'97)*, pages 459–463, San Diego, CA, September 1997.

[BH98a] Robert Baumgartl and Hermann Härtig. DSPs as flexible multimedia accelerators. In *Proceedings of the 2nd European DSP Education and Research Conference (EDRC'98)*, Paris, September 1998.

[BH98b] Martin Borriss and Hermann Härtig. Design and Implementation of a Real-Time ATM-Based Protocol Server. In *19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, December 1998.

[CL94] Huang-Jen Chen and T.D.C Little. Storage Allocation Policies for Time-Dependent Multimedia Data. Technical report, Multimedia Communications Laboratory at Boston University, 1994.

[CLV95] Huang-Jen Chen, T.D.C. Little, and D. Venkatesh. A Storage and Retrieval Technique for Scalable Delivery of MPEG-Encoded Video. Technical report, Multimedia Communications Laboratory at Boston University, 1995.

[Dan97] Uwe Dannowski. Portierung des Linux ATM-Treibers für FORE PCA-200E auf den Mikrokern L4. Term paper, TU Dresden, 1997. In German. Available from URL: `http://os.inf.tu-dresden.de/project/atm/`.

[DP93] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 189–202, Asheville, NC, December 1993.

[GDFR90] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of Summer USENIX Conference*, June 1990.

[Ham97] Cl.-J. Hamann. On the quantitative specification of jitter constrained periodic streams. In *MASCOTS*, Haifa, Israel, January 1997.

[HBM+98] H. Härtig, R. Baumgartl, M.Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS - OS Support for Distributed Multimedia Applications. In *Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[Hoh96] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Master's thesis, TU Dresden, August 1996. In German; with English slides. Available from URL: `http://www.inf.tu-dresden.de/~mh1/prj/linux-on-l4/`.

[JBIF+96] Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, Daniela Rosu, and Marcel-Catalin Rosu. An Overview of the Rialto Real-Time Architecture. Technical report, Microsoft Research, Microsoft Corporation, October 1996.

[JR98] Michael B. Jones and John Regehr. Issues in Using Commodity Operating Systems for Time-Dependent Tasks: Experiences from a Study of Windows NT. In *Proceedings of the Eighth International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 98)*, pages 107–110, One Microsoft Way Redmond, WA 98052, July 1998. Microsoft Research Microsoft Corporation.

[JRR97] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. CPU Reservation and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. Technical report, Microsoft Research, Microsoft Corporation/ College of Computing, Georgia Institute of Technology, October 1997.

[Lew97] David B. Lewis. X Window System – Frequently Asked Questions (FAQ). Usenet News: comp.windows.x, 1997.

[Lie95] J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[Lie96] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, Sep 1996; available from URL: `ftp://borneo.gmd.de/pub/rs/L4/l4refx86.ps`.

[Lin] Linux website. URL: `http://www.linux.org`.

[LRM96] C. Lee, R. Rajkumar, and C. Mercer. Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. *In the proceedings of Multimedia Japan 96*, April 1996.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison-Wesley, 1996.

[Meh96] Frank Mehnert. Portierung des SCSI-Gerätetreibers von Linux auf L3. Term paper, TU Dresden, November 1996. In German. Available from URL: `http://os.inf.tu-dresden.de/L4/`.

[Meh98] Frank Mehnert. Ein zusagenfähiges scsi-subsystem für drops. Master's thesis, TU Dresden, 1998. In German.

[MR94] C. Mercer and R. Rajkumar. An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management. Technical report, Carngie Mellon University, School of Computer Science, Pittsburgh, PA, 1994.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 96 technical conference*, pages 279–284. Silicon Graphics, Inc./ Hewlett-Packard Laboratories, January 1996. Available from URL: `http://www.bitmover.com/`.

[MST93] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. Technical Report CMU-CS-93-157, School of Computer Science, Carnegie Mellon University, May 1993.

[Nie98] Hartmut Niemann. What is GGI? Documentation of the GGI-Project, 1998. Available from URL: `http://www.ggi-project.org/`.

[NKT93] T. Nakajima, T. Kitayama, and H. Tokuda. Experiments with Real-Time Servers in Real-Time Mach. In *Proceedings of the USENIX Mach III Symposium*, pages 1–19, April 1993.

[OR98] John J. O'Connor and Edmund F. Robertson. MacTutor – History of Mathematics archive. `http://www-groups.dcs.st-and.ac.uk/~history/Curves/Lissajous.html`, 1998. School of Mathematical and Computational Sciences, University of St Andrews.

[Pau97] Torsten Paul. Videodarstellung mit Echtzeitsystemen. Term paper, TU Dresden, 1997. In German.

[Reu98] Lars Reuther. Entwicklung eines echtzeitfähigen Dateisystems. Master's thesis, TU Dresden, 1998. In German.

[Rud98] Sven Rudolph. Admission Control für ein echtzeitfähiges Dateisystem. Master's thesis, TU Dresden, May 1998. In German. Available from URL: `http://os.inf.tu-dresden.de/pubs/`.

[Rus98] David A. Rusling. *The Linux Kernel.* david.rusling@digital.com, 3 Foxglove Close, Wokingham, Berkshire RG41 3NF, UK, 0.8-2 edition, 1998. Available from URL: `http://sunsite.unc.edu/mdw/linux.html`.

[Sim92] William Allen Simpson. The Point-to-Point Protocol (PPP) for the Transmission of Multi-protocol Datagrams over Point-to-Point Links. RFC 1331, Network Working Group, May 1992.

[Sta96] 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/`.

[TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, pages 73–82, October 1990.

[WJB97] John R. Douceur William J. Bolosky, Robert P. Fitzgerald. Distributed Schedule Management in the Tiger Video Fileserver. In *Proceedings of the 16th Symposium of Operating Systems Principles*, 1997.