Seminar

User Level Device Driver am Beispiel von TCP

Christian Hennrich

Universität des Saarlandes

Inhaltsverzeichnis

1 Einleitung

Gerätetreiber sind von besonderer Bedeutung für Betriebssysteme. Sie sind nicht nur für I/O Operationen zuständig, sondern gehören auch dem priviligierten Kernelmode an, wo sie durch Fehlfunktionen grossen Schaden für das laufende System anrichten können. Laut [CYC+01] sind 70% aller Kernelpanics und Systemabstürze im Betriebsystemkern auf Fehler in den Gerätetreibern zurückzuführen. Durch Aufkommen der Mikrokernel und der damit verbundenen Client-Server Architektur, hat schon eine Auslagerung aus dem Kernel stattgefunden. In diesem Seminar soll nun auch eine Auslagerung aus dem priviligierten Kernelbereich in den Userlevel-Bereich betrachtet werden. Eine solche Auslagerung soll vor allem die Stabilität und Modularität des gesamten Systems steigern. Die Benutzung von IPC sichert ein System stärker wie die Verwendung eines Shared-Memorys. Die Ansiedlung im Userlevel lässt einen einfachen Neustart des Subsystems, ohne Neustart des Geesamtsystems, zu. Dadurch ist der Austausch einzelner Systemkomponenten leichter zu bewerkstelligen. Die Subsysteme sind untereinander abgeschottet und beinflussen sich nicht gegenseitig.

Die Auslagerung bringt aber auch Probleme mit sich: die Performance leidet sehr stark und der direkte Speicherzugriff (DMA) ist nicht mehr auf einfache Art und Weise möglich. Folglich muss man Gründe und Lösungen für diese Probleme finden. In diesem Teil des Seminars soll vor allem auf ein Netzwerk Subsystem im Userlevel eingegangen werden. In diesem Fall gibt es Probleme mit der Speicherallokation und zusätzliche Latenzzeiten bei der Laufzeit der Pakete. Durch die Benutzung der IPC gibt es mehr Kommunikation und auch die Daten müssen häufiger kopiert werden (Crossing of Protection Boundaries).

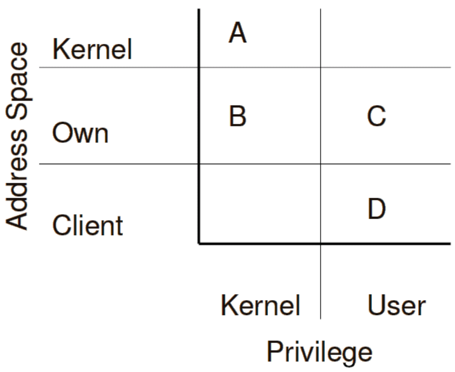

Bild 1.1: Ansiedlung der Device Driver und Privilegien

Bild 1.1 zeigt wo Treiber angesiedelt sind und welche Rechte sie besitzen:

- A normaler Linux Device Driver
- B seperater Kerneladressspace aber Kernelprivilegien
- C linked Library im Userprozess
- D seperater Prozess, Kommunikation über IPC (X-Server)

Auf die einzelnen Ideen wird später in Kapitel 2 eingegangen.

Die Implementierung eines Vergleichssystem wurde an der Universität Karlsruhe [Hupfeld02] auf dem L4Ka Mikrokernel vorgenommen. Dieses System wird in Kapitel 3 genauer untersucht werden und in Kapitel 4 wird das Design vorgestellt

2 Systembeschreibungen

Es gibt verschiedene Methoden um den Network Sublayer (hier besonders den TCP-Stack) aus dem Kernel auszulagern. Die verschiedenen Ansätze werden untergliedert in monolithische Kernel und Mikrokernel.

2.1 Monolithische Kernel

2.1.1 Shared-Memory

Die U-Net Architektur nach [Eicken95] stellt ein Highspeed Network Interface Controller (NIC) mit geringer Verzögerung für kleine Nachrichten dar. Den Anwendungen wird ein virtueller NIC zur Verfügung gestellt. U-NET Anwendungen tauschen über einen virtuellen Kommunikationsendpunkt mit dem virtuellen NIC Daten aus. Die Endpunkte bestehen aus einem Shared-Memory Segment mit einerSende- und Empfangsqueue und anderen noch benötigten Buffern. Der Paketempfang wird durch einen Upcall signalisiert (UNIX Signalhandler, Thread oder Userlevel-Interrupthandler).

Die Endpunkte können direkt in der Firmware realisiert werden (direct U-Net), wenn der NIC einen eigenen Prozessor besitzt. Probleme treten beim Memory-Management auf (Allokierung von Speicher, Umwandlung von virtuellen auf physikalische Adressen, Pagefaults). Zero-Copying, um unnötiges Kopieren zu vermeiden, wird durch das direkte Schreiben der Pakete in die Client-Datenstrukturen übergangen. Zur Performancesteigerung wird im NIC ein Software Translation-look-aside-Buffer (TLB) benutzt, in der die Umwandlung von virtuellen auf physikalische Adressen gespeichert wird und somit eine Art Cache für den Zugriff vorhanden ist. TLB Fehler werden durch den Kernel aufgefangen. Nicht vorhandene Pages werden dem NIC mitgeteilt und er kann andere Clients bearbeiten, während der Kernel die angeforderte Page nachlädt. Zusätzlich behandelt er Pagefaults und reserviert den Speicher.

Bei der Base-level U-Net wird der Buffer fest im physikalischen Speicher angelegt und die Queues liegen im user-accesible Oncard-Speicher. Der Prozessor des NICs pollt dann die Sendqueue ab. Beide Möglichkeiten erreichen eine minimale Verzögerung und eine maximale Bandbreite.

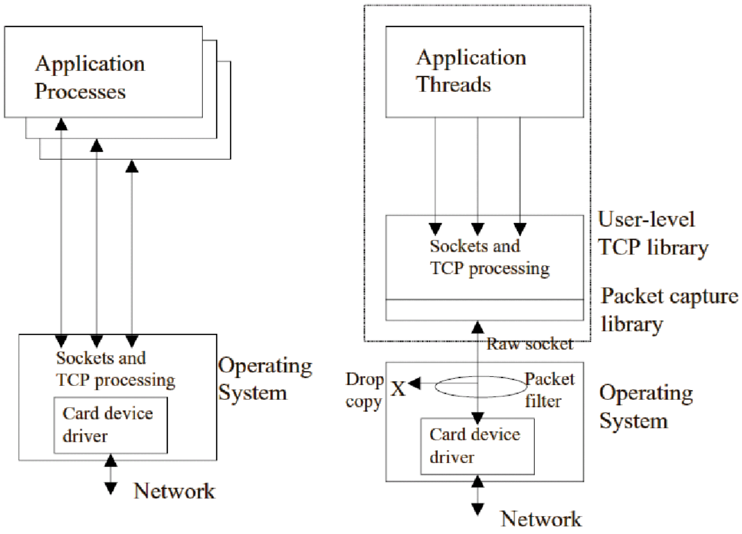

Bild 2.1: Daytona im Vergleich zum Kernel TCP Stack [Daytona]

2.1.2 Linked Library im Userspace

[Edwards95] entwickelte ein Network-Subsystem, um im Userlevel auf einen NIC mit großer Bandbreite und Performance zuzugreifen (HP-UX). Zur Zeitersparnis wurden der existierende TCP-Stack aus dem Kernel entfernt und in 2 Teile aufgesplittet. Ein Prozess übernimmt den Bottom-Half Teil des Kernels (sogenannter Child Prozess). Der andere Prozess wird mit dem Anwendungsprozess verlinkt. Der gemeinsame Zustand der beiden Prozesse wird im Shared-Memory gespeichert, Synchronisation findet über die Linux pthread Semaphore statt. Die Kommunikation übernehmen UNIX Domain-Sockets. Der Gerätetreiber verbleibt im Kernel. Die Paketbuffer liegen bei UX-HP im Speicher des NIC. Die Perfomance dieser Lösung liegt bei ca 80% der Kernelimplementierung.

[Daytona], ein anderer Userlevel TCP-Stack, baut ebenfalls auf dieser Idee auf. Bild 2.1 zeigt den Aufbau von Daytona, der dem selben Schema wie der HP-UX Implementierung entspricht. Dieser Stack wurde entwickelt um Performance Messungen vorzunehmen und TCP Status Meldungen leichter auszugeben. Normalerweise, werden diese Daten nur intern im Kernel verwendet.

2.2 Multi-Server Betriebssystem

Die bisher vorgestellten Ideen und Lösungen dienten dem Performancegewinn oder der leichteren Kontrolle und Überwachung. Die Idee von Multi-Server Betriebssystemen liegt aber weniger darin, als in der Modularisierung und der Schaffung von einfachen und klaren Schnittstellen. Dadurch wird der Austausch von Komponenten leichter und auch die Stabilität kann erhöht werden. Bei allen folgenden Untersuchungen wurde nur die Performance (Latenz und Bandbreite), aber nicht die CPU Auslastung untersucht, Ausnahme Sawmill.

2.2.1 Userlevel-Networking bei MACH

In dem Artikel von [Thekkath93] wird auch wieder der Protokollstack dynamisch zu den Applikationen gelinkt. Bei diesem Aufbau ist ein Network I/O-Modul und ein Registryserver im Mikrokernel vorhanden. Die Anwendung mit dem Protokollstack fragt beim Registryserver nach einenm Connection-Endpoint an. Dieser enthält auch einen Shared-Memory Bereich über den mit dem Netwok I/O-System kommuniziert wird. Übertragungen werden über Systemcalls dem Networkmodul mitgeteilt, die Daten werden dann in den Shared-Memory geschrieben, wo sie auch der Clientapplikation zur Verfügung stehen. Der Zugriffschutz geschieht über Semaphore. Dieses Beispiel zeigt, dass die Latenzzeiten und Performance, auch bei ausgelagerten Gerätetreibern, ähnlich wie bei kernelinternen Lösungen sind. Allerdings sind die Modularität und Sicherheit in diesem Fall nicht optimal erfüllt.

2.2.2 Userlevel-TCP/IP bei MACH

Dieser Ansatz von [Maeda93] kommt den Designgedanken für Multiserver Betriebssystemen am nächsten. Hier wird das Applikationsinterface zum Betriebssystem aus dem Kernel ausgelagert, um unötiges Kopieren zu vermeiden. In diesem Artikel wird der TCP/IP Stack auf dem Mikrokernel implementiert. Bild 2.2 zeigt den Aufbau. Er wird unterteilt in einen Operating-System Server (Protokollserver), der den Zustand und die Verbindungen behandelt. Der andere Teil, eine Library wird dynamisch zu dem Applikationsprozess gelinkt. Über diese Library wird dann direkt mit dem Treiber kommuniziert. Im einfachsten, leider aber auch schlechtesten Fall, wird pro Paket ein MACH Interprocess-Call (IPC) aufgerufen. Eine Verbesserung stellt das Zusammenfassen von mehreren Paketen pro IPC dar. Zusätzlich kann noch ein Shared-Memory Bereich benutzt werden. Als letzte Performancesteigerung kann man noch die Benutzung von Shared-Buffern, bei den BSD Sockets, anwenden. Dadurch wird das Kopieren bei der BSD Socket Semantik vermieden. Auch hiermit kommt man an kernelinterne Lösungen in Sachen Performance heran. Allerdings ist die Benutzung der IPC zur Übergabe der Daten wesentlich sicherer, wie ein Shared-Memory System.

2.2.3 I/O oriented IPC

Im Artikel von [Brustoloni98] wird ein Mischung dieser vorher beschriebenen Ansätze, MACH's Kernel-level Treibern und dem User-level Protokollserver, benutzt. Hier soll die BSD-Socket-Semantik in Bezug auf Kopieren erhalten bleiben. Es wird argumentiert, dass man nicht immer den Treiber in einen Protokollserver und eine Treiberlibrary unterteilen

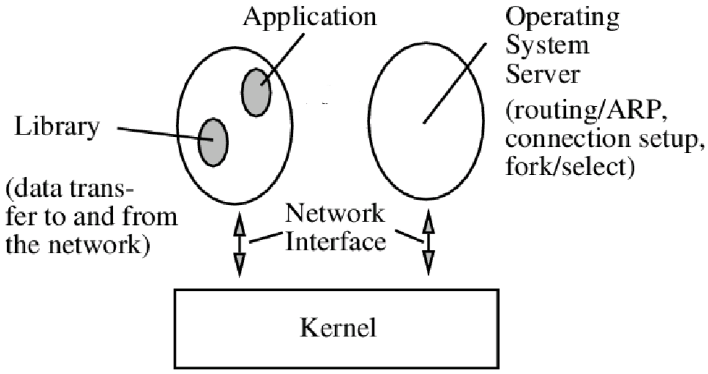

Bild 2.2: Systemaufbau bei MACH Userlevel-TCP/IP [Maeda93]

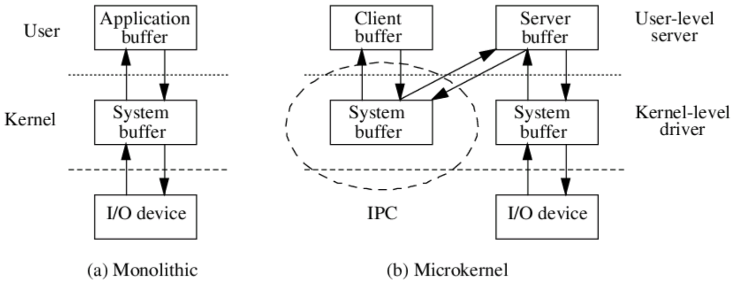

Bild 2.3: User- und Kernellevel Boundaries [Brustoloni98]

kann. Die Clients kommunizieren über, vom Kernel kontrollierte, Buffer mit den Protokollservern. Dabei werden die Buffer einfach in den Client oder Server gemappt, anstatt die Daten zu kopieren (Emulated Copy). Die Notwendigkeit des Kopierens der Buffer und der Erhalt der Unix-Copy Semantik wird über copy-on-write erhalten. Dies ist besonders in Bezug auf Kompatibilität zu den bisherigen API's wichtig. Bild 2.3 zeigt das Kopieren, das auftritt, wenn ein Mikrokernelsystem benutzt wird. Beim Mikrokernelansatz sieht man, dass die Daten häufig die Userlevel- und Kernellevel-Boundaries kreuzen müssen, was zu großen Performanceeinbussen führt.

Das hierbei verwendete Emulated Copy ist im Userbereich genauso gut, in einigen Spezialfällen sogar leicht besser, wie der Datenaustausch über Buffer im Kernelbereich.

2.2.4 Sawmill

Sawmill [HINDS] ist ein Multiserver Betriebssystem (L4Ka Mikrokernel), das aus Basisdiensten besteht, wie Memorymanagement, Naming, Emulation Librarys. Diese ermöglichen die Benutzung von Linuxsubsystemen als eigenständige Server. Es soll so viel wie

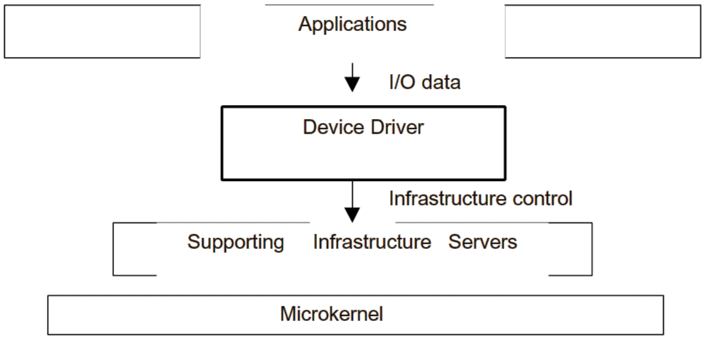

Bild 2.4: Multiserveraufbau bei Sawmill[Hupfeld02]

möglich linuxeigener Code verwendet werden. Der Network-Driver Server und der TCP-Stack Server werden von einem Network Server unterstützt, in Bild 2.4 werden sie Supporting Infrastructure Server genannt. Zusätzlich gibt es noch einen Device Driver Manager (DDM), der die Systemressourcen wie Speicher und IRQ's verwaltet. Sawmill und sein Aufbau wir im folgenden noch näher beschrieben.

Als TCP Stack wird der original Linux TCP Stack aus 2.2.5 verwendet. Zur Kommunikation wird ein Shared-Memory benutzt, in dem sich die Paketbuffer und Header befinden. Die Referenzen dieser Buffer werden mittels L4 IPC übergeben. Es gibt nun 2 Möglichkeiten diesen Shared-Memory Bereich zu verwalten. Zum einen, jeder Server verwaltet seinen eigenen Speicher, freier Speicher wird mittels IPC wieder freigegeben. Zum anderen wird eine gemeinsame Datenstruktur benutzt, die den Speicher verwaltet. Das ganze System ist so aufgebaut, dass anstatt der IPC auch Remote Procedure Calls benutzt werden können.

Sawmill hat nur 30% freie Prozessorzeit bei Übertragungen, währenddessen der Linux 2.2.5 Kernel ca 70% hat. Um diesesn miserablen Wert, der auf der Benutzung von IPC's und IRQ's mit den dazu erforderlichen Kontextwechseln beruht zu verbessern, werden wir uns im folgenden genauer mit diesem System beschäftigen.

2.2.5 Resumee

Bei allen Systemen stellt sich heraus, dass der Datentransfer nicht zu den Perfomanceeinbussen führt. Vielmehr sind der Austausch der Kontrolldaten das vordringliche Problem, was es zu lösen gilt.

3 Inter-Server Communication

3.1 Kosten der Kommunikation

In der Diplomarbeit von [Hupfeld02] wurde die CPU-Auslastung von Sawmill mit Userlevel-TCP-Stack genauer untersucht. Die Performance dieses unmodifizierten TCP-Stacks im Userbereich liegt bei ca 30% CPU Idle Time, wohingegen Linux eine Idle Time von ca 70% hat. Bild 3.1 zeigt, jeweils den minimalen und maximalen Idle Wert. Die Bandbreite ist laut den Messungen bei beiden System ungefähr identisch. Alle Messungen dauern im folgenden genau 5 Sekunden, bis die Counter resetet werden und eine neue Messung beginnt.

Versuche in [Hupfeld02] zeigen, dass weder der Treiber noch Protokollstack für diese große CPU Benutzung verantworlich ist. Die Kontextwechsel auf der CPU, also die Kommunikation der Kontrolldaten zwischen den Servern verursacht diese hohe Belastung. Es gibt nun verschiedene Ursachen für die Kontextwechsel, die jeweils unterschiedliche Häufigkeit und Kosten besitzen. Es gibt normale Hardware Interrupts, Timer Interrupts, explizite Kontextwechsel und Kontextwechsel aufgrund von IPC-Calls. Dazu wurden folgende Fälle auf IPC und IRQ untersucht:

1. TCP Transmitting mit IPC Saving Methode (mehrer Pakete pro IPC)
2. TCP Receiving mit IPC Saving Methode
3. TCP Transmitting mit 1 Paket pro IPC
4. TCP Receiving mit 1 Paket pro IPC

Results

Setup	Linux idle time confidence interval		Throughput	Sawmill idle time Confidence interval		Throughput
Unit	*[idle cycles]*		*[Mbit/s]*	*[idle cycles]*		*[Mbit/s]*
TCP Transmit	68.5 %	68.7 %	82.9	29.8 %	30.0 %	83.1
TCP Receive	70.6 %	71.0 %	62.7	26.9 %	27.3 %	63.2

Bild 3.1: CPU Idle Time [Hupfeld02]

Setup	# packets	# IPCs	# IRQs	Busy time
Unit	*/ 5 [s]*	*/ 5 [s]*	*/ 5 [s]*	*[s] / 5 [s]*
Setup 1 (Tx)	57100	3550	24250	1.98
Setup 2 (Rx)	43420	12950	37490	2.75
Setup 3 (Tx)	57880	24510	25540	2.83
Setup 4 (Rx)	42600	42300	32960	3.5

Bild 3.2: Auftrittsraten und ihre absoluten Kosten [Hupfeld02]

Die Messungen ergaben:

- 14 μs processing time per packet
- 34 μs time per IPC
- 44 μs time per IRQ

Die Zeitmesswerte sind natürlich nicht representativ für andere Systeme, aber sie zeigen einen Trend für die unterschiedlichen Größenwerte an. Bild 3.2 zeigt, dass IRQs eine dominante Rolle spielen in ihrem Auftreten und der CPU Belastung. Es lässt sich auch erkennen warum Senden teuerer als Empfangen ist, da hier mehr Kontextwechsel durch IRQs auftreten und dadurch größere Kosten verursacht werden.

Dann wurde untersucht ob der Overhead für das IRQ-Handling des L4Ka Kernels Schuld an der schlechten Performance ist. Dabei stellte sich heraus, das der Overhead lediglich für 2,8% der CPU-Belastung zuständig ist. Als nächstes hat man die TLB Missrate und die L2 Cache Line Removals gemessen. Dabei stellte sich heraus, dass beim Empfangen wesentlich mehr Cachelines entfernt werden und auch die TLB Missrate höher ist, wie beim Senden. Das kann daher rühren, dass beim Zurückschreiben der Cachelines in den Mainmemory öfters geschrieben werden muss, wie beim Senden, da dort ja auch bekannt ist, was als nächstes an Daten anliegt. Folglich sind die L2 Cache Events für den Unterschied beim Senden und Empfangen verantwortlich.

Das Ergebnis dieser Untersuchungen erklärt die Kosten für die Inter-Server Kommunikation, hervorgerufen durch die Hardware Interrupts und die Benachrichtigungs IPCs. Die Kontextwechsel mit den L2 Cache Events sind zusätzlich für die Kosten verantwortlich. Das gilt für den Sende- genau wie den Empfangsvorgang.

4 Design eines User-Level Gerättreibers

Propietäre Hardwaredevices erhalten durch Gerätetreiber eine allgemeine programmierbare Schnittstelle (API). Die Treiber, in diesem Fall die Server stellen den Clients im Userbereich Dienste, unabhängig von der darunterliegenden Hardware, zur Verfügung. Sie kommunizieren direkt mit der Hardware mittels der I/O Schnittstelle und den Interrupts (IRQ). Clients für diese Network Interface-Controller Treiber (NIC) sind Protokollstacks, user-level Kommunikationsprogramme oder Überwachungssoftware für das Netzwerk. Die Gerätetreiber nehmen von diesen Clients die Informationsströme an und schicken sie gemultiplexed an die darunterliegende Hardware.

Damit die Gerätetreiber untereinander kommunizieren und Clients den richtigen Treiber ansprechen gibt es den sogenannten Device Driver Manager (DDM). Er stellt den Clients einen Namingdienst zur Verfügung, über den der benötigte Treiber im System gefunden werden kann. Außerdem stellt der DDM den Shared Memory zur Verfügung über den der Client und Treiber kommunizieren können.

Der Geätetreiber kann in zwei Teile zerlegt werden. Zum in einen Managementteil und zum anderen in einen Kommunikationsteil, wobei wir hir nur zweiteren betrachten werden, da er für den grössten Teil der CPU-Auslastung zuständig ist. Der Managementteil dient dem Verbindungsaufbau und der Initilaisierung. Der Kommunikationsteil ist für die korrekte Übertragung der Networkpakete zuständig. Daher wird durch ihn auch die Bandbreite, Verzögerung, Jitter, CPU-Auslastung und Speicherbelegung bestimmt, also die Performance des Treibers. Genau diese Werte gilt es bei einem Userlevel-Treiber zu optimieren, daher wird bei der L4Ka Implementierung von [Hupfeld02] hier angesetzt.

Der Mikrokernelansatz besitzt besondere Probleme, wie die Aufteilung des Adressraumes und das unvorhersehbare Verhalten bei dem zusätzlich entstehenden Kommunikationsoverhead. Hinzukommt dass die Verwaltung der Ressourcen dezentral abläuft, daher besitzt die Kommunikation eine ganz besondere Bedeutung.

Die bisherigen Betrachtungen haben die Kontextwechsel auf der CPU, beim Übergang vom monolithischen Ansatz zum Multiserveransatz, als Grund für die geringe CPU-Idle Time ausgemacht. Neben den Kontextwechseln, aufgrund von Preemption, gibt es diese auch wegen den IRQ's, IPC's und IPC Timeouts. Wobei alle für unterscheideliche Kosten verantworlich sind. Daher sollen IPC's nur dann benutzt werden, wenn sie unerlässlich sind. Der Datenaustausch findet über Shared Memory und nicht über IPC's statt. Für IPC's bleiben damit nur noch Aufgaben zur Kommunikation und Synchronisation übrig. Benachrichtigungen sollen soweit wie möglich zusammengefasst werden. Dazu müssen aber Vorsichtsmassnahmen ergriffen werden um die Server zu schützen.

4.1 Vorrausetzungen

Die Performancesteigerung die nun erreicht werden soll darf nicht zu Lasten der Sicherheit und Fehlertoleranz des Mikrokernelansatzes gehen. Die Sicherheitrichtlinien dürfen nur nach Anwendung der folgenden Richtlinien geschwächt werden:

- Objekte des Designs dürfen nicht in der Lage sein sich gegenseitig zu beschädigen, weder zufällig noch gewollt
- Clients dieses Designs dürfen nicht auf die Daten von anderen Clients zugreifen, weder direkt noch indirekt durch Benutzung des Treibers

Das bedeutet, die Clients können weder den Treiber, noch der Treiber die Clients zum Absturz bringen. Falls einer der beiden Vorsätze verletzt wird, dürfen die Objekte die Kommunikation sofort abbrechen. Das bedeutet auch, dass sich der Treiber und nur genau ein Client einen Shared-Memory Bereich teilen. So kann es nicht zur Kompromittierung anderer Clients kommen. Die Buffer sind nicht in diesen Schutzmechanismus integriert. Die Fehler die dadurch entstehen können, liegen dann im fehlerhaften Treiber. Der Client muss sich um die Korrektheit der Daten in den Buffern kümmern.

Diese Einschränkungen und Schutzmechanismen dienen dazu die Vorteile des Multiserverbetriebssystems gegenüber dem Monolithischen OS beizubehalten und kaum Einbussen in der Performance zu erhalten.

4.2 Communication Interface

4.2.1 Kommunikation und Synchronisation

Die Kommunikation zwischen Client und Server findet über zwei FIFO-Queues statt, einer Empfangs- und einer Sendequeue. In der Sendequeue, speist der Client die zu sendenden Daten ein und der Server leitet sie dann an den NIC weiter. Für die Empfangsqueue gilt das gleiche Prinzip in die andere Richtung. Da weder Client noch Server ständig die Queues

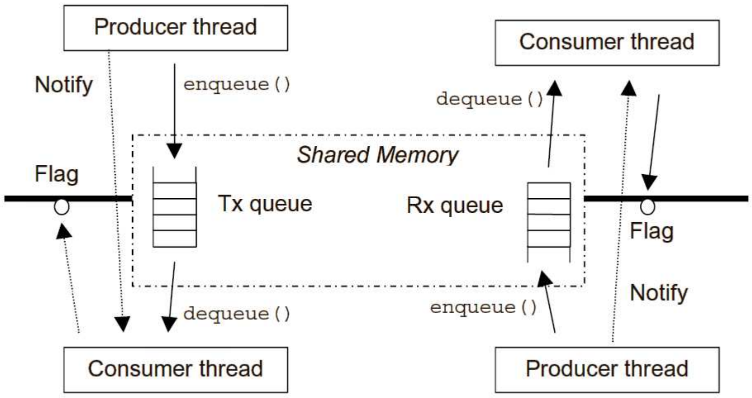

Bild 4.1: Communication Interface [Hupfeld02]

abfragen sollen, wird der Consumer vom Producer benachrichtigt. Bild 4.1 zeigt den Aufbau der Queues und die Benachrichtigungen. Client und Server können sowohl Producer, als auch Consumer sein, das kommt darauf an, ob gesendet oder empfangen wird. Wie das Bild zeigt, liegen die Queues im Shared Memory von einem Client und dem Server, so dass nur sie beide zugreifen können. Die Flags können vom Producer und Consumer gelesen und beschrieben werden. Sie heben die Blockade der Consumer auf, wenn der Producer neue Daten enqueued hat. Das Blockiern von Threads in L4 findet über die IPC Mechanismen statt. Dabei ruft der Thread receive auf, der sich schlafen legen will, bis zu einem bestimmten Timeout. Um diesen Thread aufzwecken, muss ein anderer Thread ein IPC an diesen schicken, darauf erwacht er wieder und fährt mit der Ausführung fort. Bei diesen Betrachtungen muss besonders auf Race-Conditions geachtet werden, zum Beispiel durch Preemption.

4.2.2 Buffer

Nun ist klar, wie Consumer und Producer Buffer Deskriptoren übergeben, aber wie sehen diese Deskriptoren nun aus? Logische Buffer bestehen aus den Buffer Metadaten und den Buffern selbst. Buffer Metadaten sind ein Tupel aus der Referenz auf den Buffer und die Länge des Buffers, wobei jedes Tupel auf einen Teil des logischen Buffers zeigt. Buffer Deskriptoren sind daher selbt ein logischer Buffer oder eine Referenz darauf.

Ein logischer Buffer erscheint im Virtual Memory als ein zusammenhängender Bereich. Wie Bild 4.2 zeigt kann dieser Bereich aber ziemlich verstreut im Physical-Memory liegen. Es gibt verschiedene Szenarien, wo der Treiber direkt auf den physikalischen Speicher des Buffers zugreifen muss.

- Das Gerät unterstützt kein DMA, der Treiber muss die Daten selbst hinkopieren

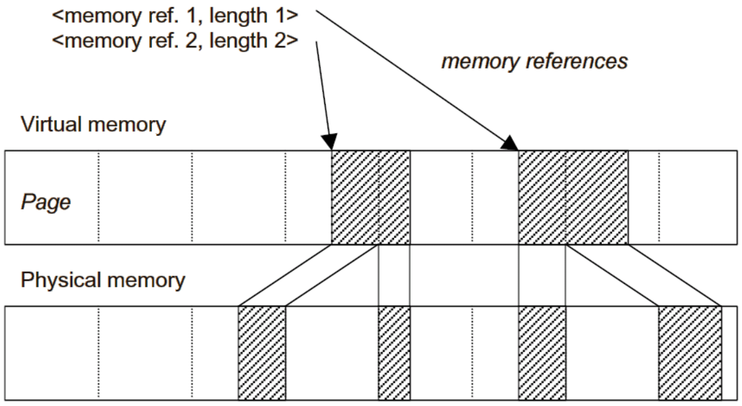

Bild 4.2: Buffer in virtual und physical Memory [Hupfeld02]

- Das Gerät unterstützt kein Scatter Buffer wie in Bild 4.2 und der Treiber muss die einzelnen Teile in einen Buffer kopieren
- Empfangene Daten müssen gedemultiplext werden und zu verschieden Clients kopiert werden
- Es soll eine Checksumme der Bufferdaten gemacht werden
- Der Treiber muss die Daten wegen eines speziellen Hardwareformats konvertieren

Jetzt muss nur noch beschrieben werden wie die Buffer angelegt werden und zerstört werden. Wir unterscheiden ob der Buffer im Shared-Memory liegt oder nicht.

Mit einer Shared Memory Umgebung

Wenn ein Client senden möchte, muss er Speicher allokieren, der Server freed ihn wieder, wenn er die Daten zur Weiterverarbeitung bekommt. Gleiches gilt für den Weg vom Server zum Client. Die Blöcke die allokiert werden gehören immer dem, der sie allokiert hat. Daher gibt es 3 Arten von Speicherblöcken: Client, Server und freie. Die Allokierung und Freigabe von Speicherbereichen wird mit `alloc` und `free` vorgenommen. Damit der Server Bereiche freigeben kann, die der Client angelegt hat, wurde eine Vereinbarung getroffen. Alle Bereiche, über die mit einer von beiden FIFOs kommuniziert wird, gehören der korrespondierden Seite.

Ohne eine Shared Memory Umgebung

Wird kein Shared-Memory benutz, muss es die Möglichkeit geben dem Treiber zu erlauben im Client Speicher zu allokieren. Dies muss bei jedem eingehenden Paket gemacht werden, die Kommunikation über IPC wird als zu teuer angesehen. Im Client muss es also eine Instanz geben, die auf Anfrage Speicher allokiert und dem Server die Referenz und Größe zurückliefert. Das muss nun für jedes Paket durchgeführt werden, das stellt jedoch einen zu grossen Aufwand dar, sogar wenn die Implemtierung ohne IPC stattfinden sollte. Ansätze hierfür wären ein Pool für bereits angelegte Speicherbereiche oder das Exportieren der Speicher-Datenstrukturen im Shared Memory.

Wie bereits vorher beschrieben muss der Treiber direkt auf den Physical-Memory zugreigen. Dafür wird nun ein System vorgestellt, das die Umwandlung von virtuellen auf physikalische Adressen vornimmt. Viele Geräte unterstützen den direkten Zugriff auf den Speicher (DMA). Die Benutzung von Virtual Memory steht ihm nicht zur Verfügung, da dieser von der CPU verwaltet wird und nur über die CPU verfügbar ist. Die Hardware greift immer direkt auf den Speicher zu, ohne Adresstranslation. Um den Zugriff des Gerätes korekt ablaufen zu lassen, muss der physikalische Speicherort valid sein, bis der Transfer beendet ist. In Sawmill Linux gibt es einen Data Space Manager (DSM). Der Treiber spricht den DSM an und versichert sich, dass die entsprechenden Speicherseiten gemappt sind und bis zum Ende gemappt bleiben. Diesen Vorgang nennt man *Pinning*. Auch die Adresstranslation übernimmt der DSM und teilt sie dem Treiber mit.

4.2.3 Schutz

Als Vorrausetzung wurde gemacht, dass die Clients vor Fehlern in den Treibern und der Treiber vor böshaftigen Clients gschützt wird. Die Benutzung eines ungeschützten Shared-Memory Bereichs, ist eine enorme Verletzung dieser Regeln. Der Zugriff auf die Shared Memory Region ist so lange in Ordnung, wie man dem DSM vertrauen kann, er also korrekt arbeitet. Davon wird jetzt einfach ausgegangen. Der Inhalt des Shared-Memory kann hingegen inkositent sein, durch falsche Bearbeitung oder Vorsatz. Beides hat die gleiche Folge, die Beendigung der Zusammenarbeit zwischen Client und Server. Fehler in einem Shared-Memory dürfen sich nicht auf andere Shared-Memorys übertragen oder andere Clients und Treiber. Es gibt zwei Klassen an Fehlern. Bei Datenfehlern liegen falsche Daten vor. Bei Progessfehlern liegt eine Endlosschleife vor und es gibt kein vorankommen mehr.

Datenfehler

Das Problem beim Überprüfen von Daten im Shared Memory ist, dass sie nach der Überprüfung nochmals verändert werden können. Daher Kopieren wir die Daten und überprüfen sie, dann wird auf den kopierten Daten gearbeitet. Daten im Shared-Memory können in zwei Klassen unterteilt werden: Pointer und normale Daten. Ein falscher Pointer kann auf Bereiche zeigen die nicht gemappt sind oder Daten enthalten die nicht für den Client bestimmt sind. Die Überprüfung, ob die Daten im zugewiesenen Shared-Memory liegen übernimmt der DSM mit der Funktion `isOwner()`. Wenn diese Bedingung nicht verletzt ist, kann es immer noch vorkommen, dass auf falsche Daten verwiesen wird. Das sollte aber kein Problem sein, wenn die Daten wiederum Pointer enthalten, zum Beispiel bei Queues, so würde der falsche Pointer entdeckt werden. Wenn dort auf wichtige Daten zugegriffen wird, werden sie auf Korrektheit überprüft. Fehlerhafte Daten werden durch Untersuchung der Metadaten überprüft.

Progressfehler

Diese Art der Fehler enden in einer Endlosschleife, bei der es keinen weiteren Fortschritt mehr gibt. Zählschleifen, kann man dadurch vermeiden, in dem man die Abbruchbedingung kopiert, so dass sie im nachhinein nicht mehr geändert werden kann. Logische Endlosschleifen sind nicht so leicht auszuschalten. Verkette Listen zum Beispiel haben keine maximale Größe. Ob man an das Ende der Schleife gekommen ist, kann man überprüfen indem man alle bereite besuchten Elemente markiert. Besucht man ein bereits markiertes Element, so befindet man sich in einer Schleife. Einfacher kann man die Größe des Shared-Memorys mit der Summer der bereits besuchten Elementen vergleichen.

5 Design einer effizienten Kommunikation

5.1 IPC Saving

Kontextwechsel treten bei folgenden Umständen auf:

- Preemption
- send IPC Systemaufruf
- receive IPC Systemaufruf
- receive IPC Systemaufruf mit Timeout (delay)
- IRQs

Jeder dieser Kontextwechsel hat individuelle Kosten in Bezug auf CPU-Zeiten, wobei ein delay IPC wesentlich weniger verzögert, wie ein IRQ der inmitten eines gerade ablaufenden Prozesses aufgerufen wird. In den vorangegangen Messungen in Kapitel 3.1, hat sich folgende Reihenfolge (absteigend) ergeben:

- IRQ IPCs
- send IPCs
- delay IPC, then send IPC
- delay IPCs

Daher müssen wir nun versuchen, so wenige Kontextwechsel wie möglich zu verursachen. Und die teuren durch günstigere zu ersetzen.

5.1.1 Consumer IPC Saving

Queue Überprüfung durch IRQ

Viele Geräte benachrichtigen den Treiber mittels IRQ, dass sie bereit sind neue Aufgaben zu übernehmen. Bevor sich das Gerät nicht zurückmeldet, muss der Treiber den Client nicht über neue Daten benachrichtigen. Stehen mehr als 2 Interrupts aus, muss der Treiber kein Benachrichtigungsflag setzen. Das Flag muss aber vor dem letzten IRQ gesetzt werden, also beim zweitletzten, da es sonst durch Preemption zu Race Conditions kommen kann. Der Consumer (hier der Client) kann nun erkennen ob neue Daten vorliegen.

Periodisches Queue Polling

Clients besitzen keinen Mechanismus wie im vorherigen Beispiel, der anzeigt ob sie in den Queues nachschauen sollen. Daher muss es etwas ähnliches geben, dass das Nachschauen veranlasst. Die Systemtimer übernehmen die Aufgabe den Consumer(Client) von Zeit zu Zeit aufzuwecken und nachzuschauen, ob neue Daten vorliegen. Das grösste Problem besteht bei der Wahl der Intervalllänge. Zu kurz und man hat keinen Gewinn, gegenüber der Benachrichtigung durch IPC bei jedem einzelnen Paket. Zu lange und die Laufzeiten, Jitter und Pfadlängen werden unnötig lang. Das periodische Polling lohnt sich, wenn der Aufwand, des Aufweckens und Nachschauen in der Queue, geringer der eines IPC Calls ist.

5.1.2 Producer IPC Saving

verzögerte Benachrichtigung

Wir wollen IPC Calls sparen. Wird ein neues Paket enqueued und man weiss wann das nächste Paket folgt, dann kann mit der Benachrichtigung gewartet werden. Folgt nicht direkt ein weiteres Paket wird benachrichtigt. Das Warten dauert bis eine bestimmte Menge an Daten in der Queue vorhanden ist. Dann wird die Benachrichtigung geschickt. Für den Client ist so eine Voraussage möglich. Treiber hingegen können nicht wissen wann das nächste Paket ankommt. Für Clients die diese Verhersage nicht treffen können, wird ein Timer gesetzt, danach wird erst benachrichtigt. Wobei das Setzen der Timer in L4 aufwendig ist.

Periodische Benachrichtigung

Periodisches Queue Polling gilt für den Consumer, für den Producer findet man die gleichen Mechanismen. Anstatt den Consumer regelmäßig die Queues abfragen zu lassen, können wir den Producer regelmäßig Benachrichtigungs-IPCs senden lassen. Die Verzögerungen haben die gleichen Auswirkungen wie beim Consumer. Der Break-even ist hier etwas anders, da wir hier einen zusätzlichen IPC pro Pollintervall haben. Der Aufwand für das Aufwecken

	Interval Queue Polling & Imprecise Timers		Linux 2.2.5
Benchmark	CPU Idle	Context sw.	CPU Idle
Transmit	63.30 %	53000	72 %
Receive	59.50 %	175000	70 %

Bild 5.1: Erreichbare Performance [Hupfeld02]

plus ein IPC muss kleiner sein, wie das Benachrichtigen per IPC nach jedem Paket.

5.2 Unpräzise Timer

Das Setzen von Timern in L4 geht mit einem Neustart des Timersthreads einher, wenn das neue Event vor dem liegt, auf das aktuell gewartet wurde. Um das zu vermeiden führen wir ein Modell ein, das dieses Einfügen beherrscht. Nun kann der Timeoutcounter ohne Neustart auf den neuen Wert gesetzt werden. Liegt der Zeitpunkt nach bereits eingefügten Timeouts, ändert sich nichts.

5.3 Resumee

Nach dem Einbau aller hier vorgestellten Performanceverbessungen hat sich der Userlevel TCP-Stack an den von Linux in Sachen CPU-Idle Time angenähert, wie 5.1 zeigt. Zwar ist die Performance immer noch 10% niedriger wie die des Linux 2.2.5 TCP-Stacks, allerdings nun fast doppelt so groß wie zu Beginn.

A Literaturverzeichnis

[CYC+01] Chou, Andy; Yang, Jun-Feng;Chelf, Benjamin; Hallem, Seth; engler, Dawson: *An empirical study of opertaing system Errors*. in Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP), pages 140-153, 1999

[Eicken95] von Eicken, T.; Basu, A.;Buch, V.;Vogels, W.: *U-Net:A User-Level Network Interface for Parallel and Distributed Computing*. In Proceedings of the Symposium on Operating System Principles (SOSP), 1995

[Edwards95] Edwards, A.; Muir, S.: *experiences implementing a high performance TCP in user-space*. In Proceedings of ACM SIGCOMM, 1995

[Daytona] Pradhan, P.; Kandula, S. ; Xu, W. ; Nahum, E.: *Daytona: A User Level TCP Stack*. Report, 2002

[Maeda93] Maeda, C.; Bershad, B.: *Protocol Service Decomposition for High Performance Networking*. In Proceedings of the Symposium on Operating System Principles, 1993

[Brustoloni98] Brustoloni, J.C.; Steenkiste, P.: *User-Level Protocol Servers with Kernel-Level Performance*. In IEEE INFOCOMM'98: Proceeding of the Conference of IEEE Computer and Communications Societies, 1998

[Thekkath93] Thekkath, C.; Nguyen, T.;Moyy, E.;Lazowska, E.: *Implementing Network Protocola at User Level*. IEEE/ACM Transaction on Networking. Vol. Nr.5, 1993

[HINDS] Hinds, N.; Gefflaut, A.;Jaeger, T.;Häberlein, A.;Götz, S.: *Performance Analysis of a Multiserver Network System*. Internal Report

[Hupfeld02] Hupfeld, Feliy: *Design and Performance of a User-level Network Driver in a Multi-Server Operating System*. Diplomarbeit Universität Karlsruhe Institut für Betriebs- und Dialogsysteme, 2002