Design and Performance of a User-level Network Driver in a Multi-Server Operating System

cand. inform. Felix Hupfeld

Betreuer: Dr. Kevin Elphinstone

Institut für Betriebs- und Dialogsysteme
System Architecture Group
Fakultät für Informatik
Universität Karlsruhe (TH)
Karlsruhe, Germany

Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, den 27. Juni 2002

Felix Hupfeld

Acknowledgements

I would like to thank my advisor, Dr. Kevin Elphinstone for guiding me through this thesis with his lucid analyses and critical comments.

Furthermore, I would like to thank my room mates (a.k.a. the „usual back room“), Stefan Götz, Christian Schwarz, Jan Stöß, Stephan Wagner, Horst Wenske, Christian Ceelen, Andreas Häberlen, and Markus Völp for many hours of work and fun, and the Ph.D. students Uwe Dannowski, Joshua LeVassueur, Espen Skoglund, and Volkmar Uhlig, and our system administrator James McCuller. Each of them provided help which saved hours or even days.

Finally, I would like to thank my parents, Ilse and Dr. Peter Hupfeld, and my girl-friend Juliane Reichenbach for giving me the support to able to focus on my studies.

Contents

1	**Introduction**	**6**
1.1	From monolithic kernels to multi-server operating systems	6
1.2	Motivation	8
1.3	Thesis outline	8
1.4	Issues in user-level network driver design	9
2	**Related Work**	**12**
2.1	The network subsystem of Sawmill Linux	12
2.2	The architecture of non-monolithic network subsystems	15
2.2.1	High-performance user-level networking	15
2.2.2	Multi-server OS's	18
2.2.3	Virtual Machine Monitors	20
2.3	Inter-Process Communication (IPC)	21
2.4	Protection while using shared memory	21
2.5	Packet buffer formats	23
2.6	Summary	24
3	**Costs in inter-server communication**	**25**
3.1	Measurement environment	25
3.1.1	Benchmarking network subsystems	25
3.1.2	Performance measurement	26
3.1.3	Measurement environment	27
3.2	Sawmill Linux vs Linux 2.2.5, consistency of idle time measurement	29
3.3	Context switches – The main cost factor in inter-server communication	30
3.3.2	Busy time can be accounted to packet processing, IPCs, IRQs	31
3.3.3	Context switches are dominated by their associated costs	33
3.3.4	Overhead of L4Ka IRQ handling	34
3.3.5	High costs of receive IRQs	35
3.3.6	Attributing context switch costs to cache refills	37
3.4	Summary	38
4	**Interfaces of a user-level device driver**	**39**
4.1	Introduction	39
4.2	Requirements	40
4.3	The Communication Interface	42
4.3.1	Communication	42
4.3.2	Synchronization – The wake-up protocol	43
4.3.3	Synchronization – API	46
4.3.4	The wake-up protocol – Comparison to literature	47
4.4	Buffer format	49
4.5	The driver's access to buffer data	51
4.6	Buffer memory management	51
4.7	The interface to the supporting infrastructure	53
4.7.1	Physical requirements of hardware devices	53
4.7.2	Model of page states	54

4.7.3 Driver-side interface 54
4.7.4 Client-side interface 55
4.7.5 Implementation issues 56
4.8 Protection 56
4.8.1 Data failures 57
4.8.2 Progress failures 57
4.9 Summary 58

5 Efficient inter-server communication 59
5.1 Consumer IPC saving 60
5.1.1 Interrupt-Driven Transmit Queue Checking 60
5.1.2 Periodic Queue Polling 61
5.2 Producer IPC saving 61
5.2.1 Deferred Notification 61
5.2.2 Periodic Notification 62
5.3 Evaluation of IPC saving techniques 62
5.3.1 Methodology 62
5.3.2 Periodic Queue Polling (PIC machine) – Results and Discussion 63
5.3.3 Periodic Queue Polling (APIC machine) – Results and Discussion 63
5.3.4 APIC Periodic Notification – Results and Discussion 67
5.3.5 Analysis of context switch costs 68
5.4 Imprecise software timers 69
5.5 Evaluation of imprecise software timers 70
5.5.1 Methodology 70
5.5.2 Result 70
5.5.3 Discussion 70
5.6 Achievable performance 70
5.6.1 Results 71
5.6.2 Discussion 71
5.7 Summary 71

6 Conclusion 72
6.1 Future Work 73

7 References 74

Abstract

Multi-server operating systems have many advantages like increased robustness and flexibility over monolithic kernel designs. However, the introduction of protection boundaries between the operating system's components gives them an inherent performance penalty.

This thesis investigates the performance issues of inter-server communication and identifies context switches as the main source of costs. It analyzes the requirements of a network device-driver server and presents a flexible design for its communication and infrastructure interfaces while ensuring full protection.

The results from the performance investigation are used to design and implement techniques that allow inter-server communication at high rates with a processor utilization that is competitive with Linux.

1 Introduction

1.1 From monolithic kernels to multi-server operating systems

The classic way to structure operating systems is the monolithic kernel architecture (figure 1). The operating system along with hardware device drivers, file systems, and network protocol implementations reside in the so-called kernel. This kernel is protected from the applications via memory management techniques. Communication from the applications to the kernel is via system calls, which are a well-defined protected way to enter the kernel.

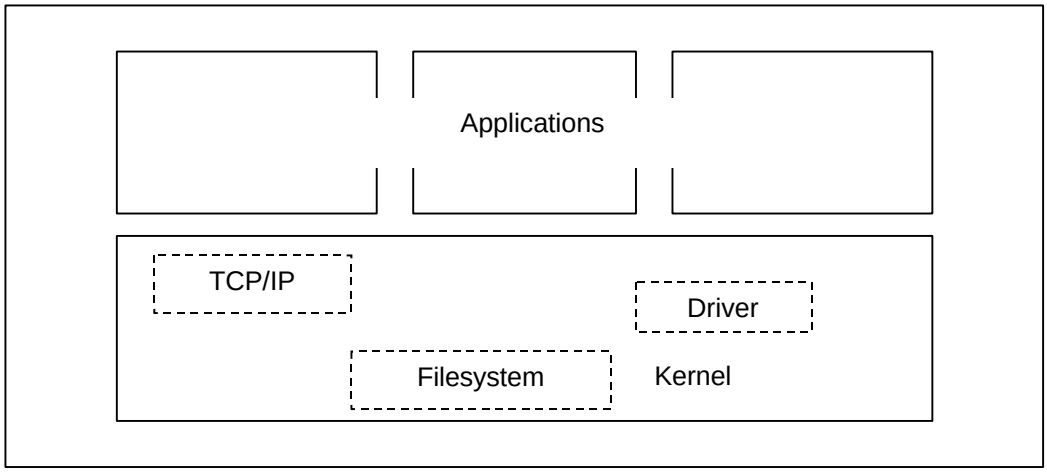

Figure 1.1 The monolithic kernel design

Inside the kernel, subsystems are not protected from each other. Therefore, they are able to corrupt each other's code and data, either by programming errors or voluntarily in the case of compromised subsystems.

The microkernel approach (figure 2) proposes a solution to this problem. Microkernels are stripped-down kernels which only have support for address spaces, threads and interprocess communication (IPC). Drivers and subsystems like disc I/O or network I/O are implemented as separate entities, so called servers, which are like normal applications processes.

In this way, all servers are protected from each other and can only communicate via well-defined interfaces. If a server crashes due to programming errors, or is compromised through a security leak, it can not influence other servers in an uncontrolled manner. If the system is well designed, this server can even be restarted during runtime without having to restart the whole system. This increases security, fault tolerance and robustness.

Furthermore, an operating system composed of servers allows flexibility in configuring the system: one can specialize the set of servers to tailor the system to specific applications. For example, a router appliance does not need disk driver servers and filesystem servers.

The decomposition of the system into a set of isolated entities has also advantages from a software engineering perspective. The problem of kernel development is reduced to the problem of developing individual isolated servers with well-defined interfaces which can be developed separately. These servers are run like user-level applications and can thus be tested and debugged with normal methods and tools.

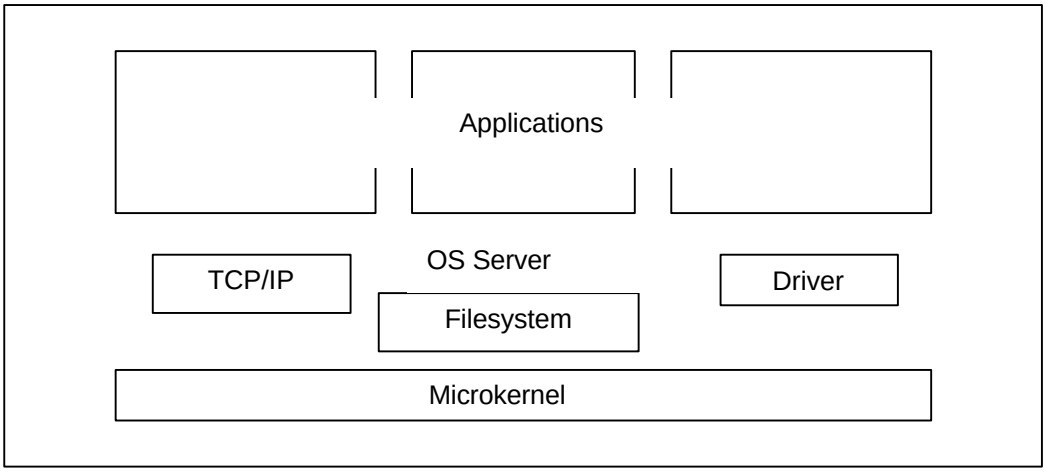

Figure 1.2 The multi-server approach

However, the separation of kernel components into separate servers comes with higher costs for communication. A multi-server system needs to share all data explicitly and communicate via the mediation of system calls, whereas in monolithic design, the data was shared and therefore accessible by the whole kernel and control could be transferred by function calls or lightweight synchronization primitives. This disadvantage of the multi-server approach especially becomes visible in scenarios where servers have to communicate at a high rate.

One of the most communication intensive parts of a multi-server OS is the I/O subsystem where the data sent or received by client applications has to cross multiple protection boundaries until it travels a path between the hardware device and the application. In standard multi-server architectures, the OS would have a device driver server and a protocol server (which implements a filesystem or a network protocol stack) inbetween.

1.2 Motivation

This work will propose how to structure a user-level driver in a multi-server environment, helps understanding the performance impacts of this approach and presents techniques that allow the operation of a user-level driver at competitive costs when compared to a monolithic design.

While there has been some work on the problem of device drivers in a multi-server environment, this work was done on first-generation microkernels where the driver remained inside the kernel and thus there were no user-level device drivers.

When the driver is in the kernel, many of the problems a user-level driver is faced with do not occur. The driver in the kernel has full access to the kernels internal information, full access and control of the communication data is given, and the transfer of control between the driver and a client consists only of kernel enter and exit.

Furthermore, a microkernel architecture that has drivers in the kernels exhibits none of the multi-server advantages, it has the same characteristics in this aspect as a monolithic architecture.

The microkernel driver work described in the literature concentrates on structuring issues and on comparable speed without looking at the costs which this performance has.

An experiment in this thesis reveals that a user-level driver on the L4 microkernel, which is currently the fastest microkernel around reaches the same throughput and latency as its Linux counterpart, but does so while consuming much more CPU cycles.

1.3 Thesis Outline

During this thesis, we will examine how the interfaces of a server-contained hardware device driver to other parts of the system should be structured so that they are flexible enough to provide service for a wide range of applications and hardware device types, that they preserve the protection advantages of the multi-server design and that they do not lose too much performance relative to a comparable monolithic kernel architecture.

We will investigate these problems by looking at the example of the interface between a network protocol stack and a network interface driver in the Sawmill Linux operating system. We picked this special example because network I/O is the most performance demanding I/O type due to the lack of real physical constraints (like platter speed of hard disks), it does not need much surrounding infrastructure to work properly (like for example disk I/O needs) and it has well measurable, relevant characteristics (throughput, latency and jitter).

To achieve our goals, we will first investigate how much performance Sawmill Linux loses when compared to Linux, where it loses that performance and what is the cause for the loss. The results of this investigation will help us in designing a better interface and in developing techniques that use the interface in a high-performance manner.

A hardware device driver has two main interfaces to the system: the service interface for driver clients and the interface to the driver‘s supporting infrastructure in the I/O subsystem.

One important aspect of the design of these interfaces is the way they dictate their own implementation and the implementation of other parts of the system. The problem is to find a set of interfaces that are general and policy-free enough to allow unforeseen implementation techniques for achieving high performance communication which do not restrict the design of the supporting infrastructure. They must also take into account the introduced protection domains and the efficient translation between them.

We will then use the results of our performance investigation to develop techniques that allow high-performance communication over the newly defined interfaces.

The follow section overviews the issues relevant to this work. The thesis continues with an overview of related work (chapter 2). We will then investigate the performance aspects of Sawmill Linux (chapter 3), present a new interface design (chapter 4), and describe the performance-enhancing techniques and evaluate them (chapter 5). The thesis finishes with a conclusion.

1.4 Issues in user-level network driver design

As just described, the goal of this thesis is defining a high-performance interface between the driver and its clients (the “driver interface”) and an interface of the driver to supporting functionality of the I/O subsystem, both while ensuring protection and performance. These two interfaces are the only communication channels of a user-level driver to other parts of the operating system.

Via the driver interface, the user-level driver provides an abstracted interface to the hardware it is responsible for. In the context of I/O-oriented hardware devices, this interface exports a service for clients to send and receive data and to manage the resources for this data.

One of the issues we will investigate is how to design this interface, i.e. which primitives to put in it to have a generally applicable and policy-free interface that maximizes the range of implementation techniques. In addition to the transfer of the actual data, an important issue that we will tackle here is the transfer of control, i.e. the way one party can notify the other party of events at this interface like the arrival of new data.

Data itself is transferred in units of logical buffers which are an ordered set of memory regions. These logical buffers have to be freed by the driver after it is finished with the data. For reception of new data, the driver has to allocate new

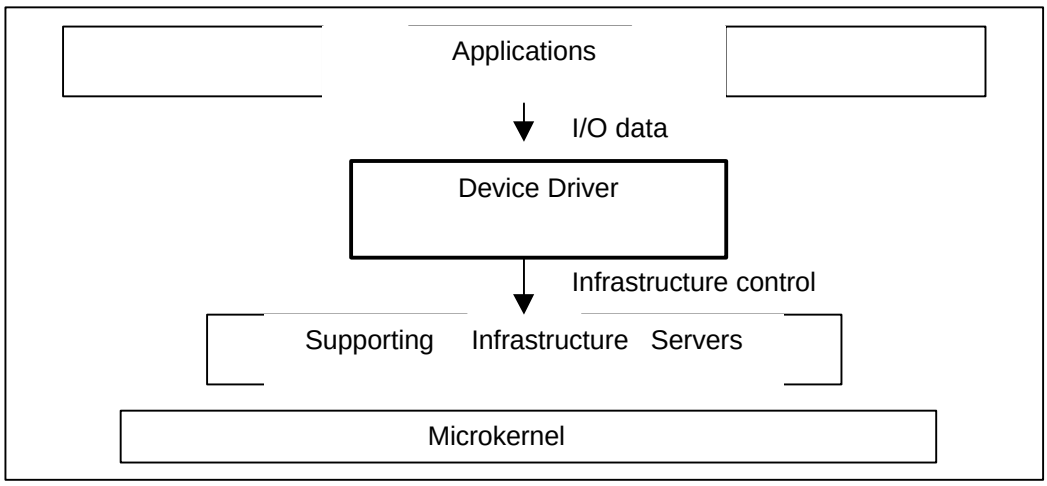

Figure 1.3 The driver's interfaces

logical buffers to have space to put in the just received data. This thesis will investigate the question of how to structure the logical buffers and how to manage their memory.

The driver's interface to the I/O subsystem ideally defines a minimal set of functions which the underlying I/O subsystem infrastructure has to provide to the driver so that it can work correctly. Again we will try to define an interface which is policy-free, does not dictate a particular way of implementation and makes an minimum of assumptions about the way the I/O subsystem provides its service.

One of the main tasks of the driver is to bridge the gap between the virtual nature of the logical buffers and the physical demands of the hardware device. Where logical buffers live in virtual memory that is not necessarily backed by physical memory all the time, the hardware devices still need to directly access the physical memory. This means that the driver has to provide the hardware with physical memory locations of the logical buffer and it has to make sure that this relationship stays constant during the time the hardware access this data.

We will show how the driver can manage these requirements and describe which support it needs from the infrastructure of the network subsystem. Here, we will only detail the interface to the supporting infrastructure without describing how the infrastructure can provide this functionality.

As the driver and its clients are located in different protection domains, references to logical buffers are only valid locally in a server but are communicated over protection boundaries. We will show how to translate these references from one protection domain to the other.

Before we design the interfaces, we analyze an existing system to find out which parts of the system we can attribute the consumption of CPU cycles in order to be able to lead the design to an acceptable performance. We will have to understand where the sources of high CPU utilization of the current multi-server architecture are and how they affect it.

Furthermore, when we decide to use shared memory in our design and thus weaken the protection, we will have take countermeasures to ensure the protection property of the multi-server approach and show that these countermeasures are sufficient.

2 Related Work

2.1 The network subsystem of Sawmill Linux

This thesis is based on work done at IBM Research under the project name Sawmill Linux [Hinds]. Sawmill Linux is a multi-server operating system that consists of a set of servers that provide basic services like memory management and naming, and a set of emulation libraries that allow subsystems of Linux kernel to be compiled and run as separate servers. The main goal was to be able to reuse as much unmodified Linux code as possible.

The network subsystem of Sawmill Linux consists of the original Linux 2.2.5 TCP/IP stack and the original Linux driver as separated servers (see figure 2.1). For communication, packet buffers and their headers live in a shared memory region and are passed via reference with L4 IPC calls. Two approaches have been tried to manage this shared memory region. The first one lets each server manage its own memory, free buffers are returned (via one or less IPC per buffer) to their allocating servers to put them back in the pool of free memory. The second approach used a shared data structure to manage the memory.

The driver and the stack server are supported by a network manager server which maintains the network system name space and state and a device driver manager (DDM) server which handles system resources like IRQs and memory.

The implementation is a port of the Linux kernel network subsystem including the whole supporting kernel infrastructure. All adaptations were made with the minimal effort at the most bottom (and thus) simple layer. The supporting infrastructure has been ported in a way that the emulation comes into play when the real Linux kernel would use the hardware for the functionality. Thus, memory management interfaces with Sawmill at page granularity, timers at the internal kernel clock.

The componentization boundary has been chosen so that a simple local function call could be replaced with a remote one between the two servers. Where the original bottom half handler calls `pt->func()` to inject a packet in the registered packet handler for that type of the stack Sawmill has replaced this with a (cached) lookup of the registered packet type handler registered in the DDM and then an IDL IPC to call this handler along with the reference to the packet. The whole bottom half handler infrastructure is emulated with full functionality, though its original purpose of decoupling interrupt handling from packet processing is no longer necessary.

In the opposite direction, calls to `dev_queue_xmit()` cause an IDL IPC to the driver which carries a reference to the packet, too. On the driver's side, the emulated kernel subsystem of Linux comes into play, which puts the packet in the so-called qdisc and immediately removes it from the queue and puts it in the NIC.

In normal Linux, the qdisc is used for QoS scheduling, but it has no use in Sawmill as QoS scheduling is not supported by the driver's interface.

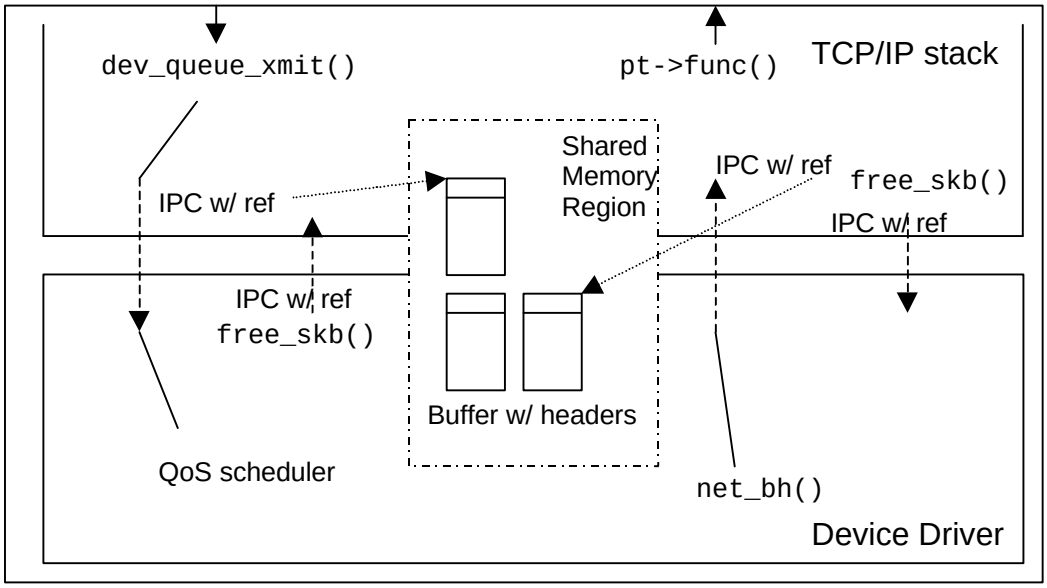

Figure 2.1 The TCP/IP stack – driver interface in Sawmill Linux

As each server of the network subsystem manages its own shared memory and the data structures are private to the servers, memory blocks, that were allocated in an other server must be returned to that one to get freed. This needs additional communication and makes the „recycling" of memory from transmit to receive in the driver impossible.

The interface between the servers has not been designed for protection and is tied to the communication with network I/O adapters. As we will see in the next chapters, the performance is far away from what is possible in a comparable multi-server environment

Our driver architecture will be embedded in Sawmill Linux. This fact has implications on the assumptions we can make about the environment as we want our architecture both to fit nicely into Sawmill Linux but also be a general architecture for communication between drivers and clients in a Multiserver OS.

As described, Sawmill Linux consists of a set of system servers, which provide a VM framework and loading and a set of linker libraries, that allow Linux code to run unmodified. These linker libraries emulate the infrastructure and APIs which are available in the Linux Kernel. Thus, one can take components like a network driver or the TCP/IP stack, link them against the emulation libraries and run them in Sawmill Linux.

Some of these interfaces are relevant to our work. We need a cutting point in the Linux code where we can plug in our infrastructure without having to modify

Linux code. As our infrastructure deals with the transport of packets, we need to control the allocation and deallocation of packet memory and injection points to get packets in and out of the Linux code.

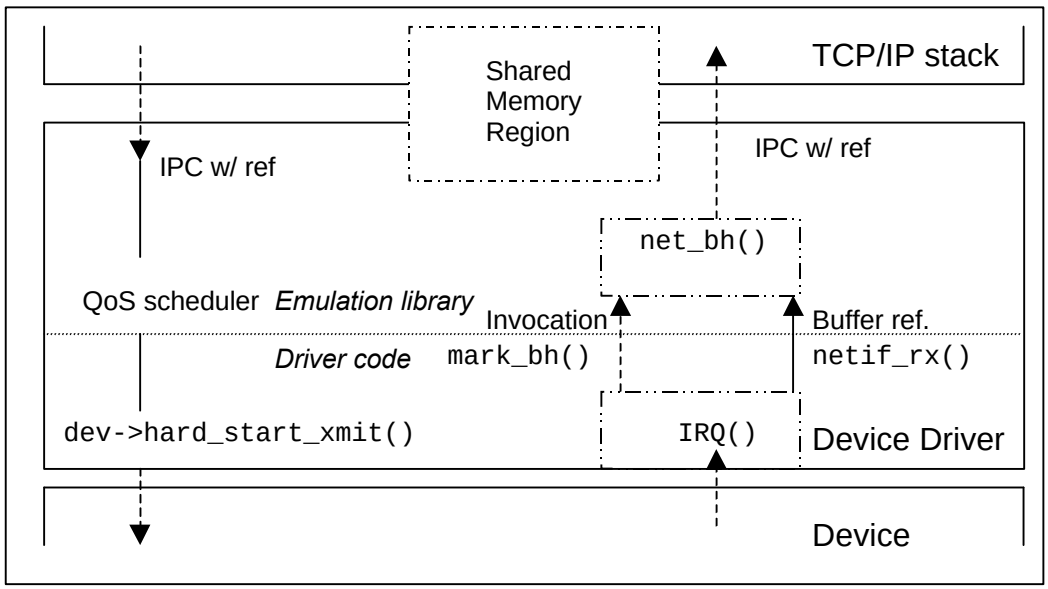

Figure 2.2 Internal structure of Sawmill's NIC driver

Linux provides an API for packet memory management, it can be found in `skbuff.(c|h)`. The main interface consists of two functions:

- `alloc_skb()`
- `free_skb()`

which allocate memory for both the skbuff header and the buffer itself and initialize those structures. The original skbuff code uses the kernel's slab allocator (`slab.(c|h)`) to manage the memory.

As we see in figure 2.2, a Linux network driver itself interfaces to the kernel's packet transport subsystem which is packed into the emulation library with three functions:

- `netif_rx()`, for receiving a packet,
- `dev->hard_start_xmit()` for sending a packet, and
- `mark_bh()` for starting the bottom half handler, ie. for doing work after an interrupt.

We can plug in our infrastructure directly here. The second important packet transportation interface is the one between the kernel's packet transport subsystem and the TCP/IP stack (see figure 2.1). Here, two functions are important:

- `pt->func()`, the packet handler for the respective packet type, and

- `dev_queue_xmit()`, the stack's interface for sending a packet over a given device.

Note that the Linux code of the driver and the stack depend on using `sk_buffs`. If we chose to use an other packet buffer model, we have to adapt the code accordingly or provide conversion interfaces. We will describe the consequences of this in a later section of this chapter.

As we will describe in detail in a later section, this Sawmill Linux structure has huge performance problems in terms of used processor cycles. While normal Linux 2.2.5 has about 70% of the processor idle during normal TCP transmission, Sawmill Linux has only about 30%. We will see that this can be traced to the fact, that Sawmill Linux uses IPC for both buffer transmission between the driver and a client and for management of the shared memory region.

Furthermore, in Sawmill Linux, the management of the shared memory regions is tightly integrated with the Linux code. It makes assumptions on where the shared memory region is located in the virtual address space, how it is layed out and that it is linearly mapped to physical memory. Sawmill Linux shared memory region is always pinned, which restricts flexible management of the resource memory.

2.1 The architecture of non-monolithic network subsystems

2.1.1 High-performance user-level networking

Most highly computation-intensive applications like simulations are run today on parallel computer systems consisting of thousands of nodes. The nature of those applications does not only demand high computing speeds but also high communication performance between the parallel nodes as the results of the computation of one node depends on the intermediate results of the computation on other nodes.

The communication load here is characterized by small packets with low-latency demands at high rates with application-tailored contents. As current kernel network subsystems is not able to provide this, one needs a communication mechanism which gives the same protection across applications as a kernel-resident one, but is located in user-space to allow custom communication mechanisms with minimal overhead [Damianakis98]. The targeted hardware here are high-speed NICs with on-board processor.

With a kernel network subsystem, packets have to cross the kernel boundary and multiple levels of abstraction, when traveling between the application and the network device and are copied at least once when using normal BSD socket semantics. Furthermore, current kernel implementations are optimized for throughput whereas many of the computation-intensive applications depend on a

low round-trip time. The driver's interface and the kernel's protocol stack are tightly integrated which makes an implementation of new protocols difficult [Eicken95].

By placing the entire protocol stack at user-level, they remove the kernel along with its overhead and specific semantics completely from the critical path. Thereby, unnecessary software layers, copies and the kernel boundary crossing are eliminated and each application can implement its own tailored protocol stack.

Their solutions are very similar to a multi-server architecture and thus face the same issues: Where they decompose a formerly kernel-resident network subsystem into a part which is integrated in the NIC hardware and a part which is put into the application in user-land, we divide it into user-level servers while having the same separation border. So they have the problem of defining an interface between the entities, too, and they have to solve the same problems that come along with the introduction of separate protection domains. There must be a way to convert addresses between them and entities must be able to control the physical representation of virtual memory. They also have to find a way to efficiently transfer packet data and to notify the other party of the advent of new data.

U-Net

The U-Net architecture [Eicken95] provides a virtualized high-bandwidth low-latency interface for small message sizes from the NIC to user-level applications. Their primary goal is to have low latency while having a general interface to the NIC. U-Net applications communicate via a virtual communication endpoint with the virtualized NIC. These communication endpoints are a shared memory segment consisting of a receive and a send queue, the buffers and a free queue, which holds pointers to free buffers. Packet reception is signaled to the applications either via an upcall which could be a UNIX signal handler, a thread or a user-level interrupt handler.

The communication endpoints can be implemented in several ways. When an network interface card with on-board processor is available, the U-Net interface can be implemented completely in firmware (direct-access U-Net architecture). The main problem is how to give the NIC access to the memory management to be able to pin memory, issue pagefaults and map virtual to physical addresses. However, true zero-copy can be achieved this way as the card writes received packets directly into the clients data structures.

A less sophisticated design is the Base-level U-Net architecture. Here, the shared buffer region is in pinned physical memory and the queues are in user-accessible on-card memory. The NIC's processor polls the queue to detect new packets in the send queue. For less demanding applications, there is a kernel emulation of an

U-Net endpoint which demultiplexes a normal U-Net endpoint to multiple applications.

The direct-access U-Net architecture uses a custom software TLB, which is implemented in the NIC's firmware to provide virtual to physical address translation, to pin memory and to handle page faults [Welsh97]. TLB misses are handled by the kernel through a translation-request queue, whose requests include the page to unpin instead. For resident pages, the kernel pins the page and answers with the corresponding physical address, non-resident pages are signaled to the NIC immediately so that it can serve other clients while the kernel loads the page's contents from disk.

A U-Net implementation for standard fast-ethernet hardware [Welsh96] uses a pinned shared memory region for the whole connection endpoint. The user process uses a fast trap to the kernel to notify the kernel to process the send queue. This trap is said to need about 1µs on a 133 MHz Pentium for entry/exit. On reception of a packet, the kernel determines correct destination endpoint and copies the packet in free buffer there. The user process is notified via an interrupt.

They achieve both a minimal latency and maximum network bandwidth with small message sizes.

User-level networking on HP-UX

[Edwards95] designed a user-level network subsystem for custom ATM gigabit cards on top of HP-UX to be able to access the NIC from user level with high TCP throughput. As TCP is not easy to implement, they moved the existing TCP/IP implementation out of the kernel and splitted it into two separate processes. One does the bottom-half part of the kernel (called the child process), the other one is linked with the application and does the top-half handling (called application process). They identified the sharing of signals and timers and the TCP's connection semantics to be problems when linking parts of the TCP/IP stack with the application.

The shared state of the two processes is located in the shared memory region managed by both processes and synchronized by semaphores. The two processes communicate via UNIX domain sockets.

The NIC driver remained in the kernel and is controlled via ioctl() calls. The packet buffers are located in the NIC's on-board memory, demultiplexing is done via ATM channel identifier to the buffer pools on the card.

Their experiments show that they are able to achieve 80 % of single-copy kernel implementation and outperform the normal kernel implementation in terms of throughput. While the single-copy in-kernel version uses about 72 % CPU time, their user-level implementations are at about 85 %.

VMMC

In the context of the VMMC project, [Damianakis98] evaluated different methods for detecting the reception of new messages on one side of the interface between a driver and an application. Traditionally, this event is communicated by a device interrupt. However, if the reception routine is in user-space, every interrupt implies an expensive kernel boundary crossing.

The thesis describes hybrids of polling by the user and interruption through the kernel, evaluates them and comes to the conclusion that a polling/interrupt hybrid is the best performing solution.

Discussion

U-Net and [Edwards95] both use queues to communicate logical buffers from one party to the other. Where the HP-UX architecture has the buffer data in the NIC's on-board memory and thus no problems with pinning or translation, U-Net uses normal memory. Pinning and translation is achieved here with a software loaded TLB on the NIC which gets filled by an asynchronous communication mechanism with the host system.

In all three systems, transfer of control (or notification) is done with using software or hardware interrupts. VMMC extends this mechanism with the usage of polling and proposes a interrupt/polling hybrid as the best solution.

2.1.2 Multi-server OS's

Whereas the user-level networking community is primarily focused on creating a low-latency high-throughput interface to high-speed NICs, the goal of the multi-server OS work is to identify separate entities which can be accessed with clean interfaces.

Apart from the work done with Sawmill Linux, there are few other multi-server projects that tackle the issues of I/O.

User-level networking on Mach

[Thekkath93] writes that "considerations of code maintenance, ease of debugging, customization, and the existence of multiple protocols argue for separating the implementations [of network software] into more manageable user-level libraries". They propose an architecture where the protocol stack is linked dynamically to the application.

Their system runs on a Mach microkernel and consists of a registry server and a network I/O module inside the microkernel. The client application (with its protocol stack) requests a connection endpoint from the registry server which includes a region of shared memory to communicate directly with the network I/O

module. This shared memory is managed by both the network I/O module and the client application.

Transmissions are initiated via a system call into the kernel network module, the received packets are written into the shared memory region and the client application is notified with a semaphore. They show that a decomposition of the network subsystem into user-level servers can be done while maintaining a competitive performance in terms of throughput and latency. Processor utilization was not examined.

User-level TCP/IP on Mach

[Maeda93] argue that the application's interface to the network can be separated from the interface to the operating system and thus unnecessary kernel entries, copies and software layers can be avoided in the critical communication path. They describe their user-level TCP/IP protocol stack implementation on top of the Mach microkernel. It is split into an operating system server, which manages state and handles connections and a dynamic library in the application process which communicates directly with the network driver.

In their simplest implementation, communication between the application's protocol library and the NIC driver inside the kernel used one Mach IPC per packet. Improvements included combining multiple packets into one Mach IPC and finally to use a shared memory region and a condition variable. A further change in the socket interface allowed shared buffers which eliminated the copy involved with BSD socket semantics.

They show that their implementation achieves competitive throughput and latency when compared to a Mach and Ultrix in-kernel implementation and a mach-based Unix single-server system. Processor utilization was not examined.

I/O-oriented IPC

I/O-oriented IPC [Brustoloni98] is a communication facility between Mach's kernel-level drivers and user-level protocol servers. In this work, they want to preserve BSD socket copy semantics and argue that it is not always possible to separate a driver client into an application-linked component and a server.

System buffers are always resident in physical memory and thus not pageable. Their three-tierd architecture assume protocol clients and protocol-implementing servers to be at user-level and communicate with kernel-managed buffers. The need for copying these buffers while preserving UNIX-copy semantic is minimized with copy-on-write techniques.

In their experiments, they examine round-trip latencies with varying datagram lengths. Their user-level implementation has minimal lower latency. However the latency was mostly dominated by network latency. The overhead is said to be

insensitive to data lengths, thus it is a control passing overhead, in contrast to the data passing overhead of copying IPC. The control passing overhead is said to be resulting from rescheduling and context switching.

Discussion

The research in these multi-server projects is primarily focused on the structure of the system. Performance is only examined in terms of throughput and latency, the processor utilization is not yet examined. By putting the emphasis on the overall structure, the literature does not describe very detailed.

All three systems, however, identify more or less clearly the transfer of control as an important factor. While the first system uses the kernel entry, the second one starts with using Mach IPC as a packet transfer mechanism and tries to optimize with putting more packets into one IPC or even with omitting the IPC by using a condition variable. I/O-oriented IPC does not mention the way of control transfer explicitly but they infer from their benchmarks that communication in a multi-server environment is a control transfer and no data transfer problem.

2.1.3 Virtual Machine Monitors

Like user-level networking and multi-server OS designs, Virtual Machine Monitors also face the problem of having the hardware drivers and the driver's clients located in separate protection domains. They also have to find efficient and clean interfaces for buffer transfer and reception notification between them and deal with the problem of domain translation.

VMWare

VMware [Sugerman01] is a virtual machine monitor which emulates a complete IA-32 machine including a virtual network interface controller (NIC). The virtual NIC is connected to an virtual internal network inside the virtual machine, which in turn connects also to a virtual NIC in the host OS and the physical NIC. This way, the virtual machine is able to communicate both with the host OS and the outside world. The virtual NIC is implemented as normal hardware device which communicates over the I/O address space and IRQs.

To achieve a completely virtualized environment, the virtual machine has to emulate all instructions which communicate outside the virtual machine. This emulation has a major performance impact as every boundary crossing between the virtual machine and the virtual system is costly. Thus, to get better performance, the number of boundary crossings has to be minimized.

As a first optimization, only I/O operations which affect the outside world and not just the NIC's state are implemented in the virtual machine monitor. Their second improvement is to combine multiple packet transmissions into one boundary

crossing, if the packet rate is high enough. This is done by measuring the packet rate and deciding by this number, whether to wait with the boundary crossing until further packets arrive or to do it immediately.

By applying these techniques they achieve a CPU utilization which is well below the one of the original implementation.

VMWare also identifies the transfer of control from the virtual machine to the host system as the primary source for CPU cycle consumption. They try to minimize it with transferring multiple packets per control transfer.

2.2 Inter-Process Communication (IPC)

In a multi-server system like in any system with separate processes, there must be a mechanism for the entities to communicate with each other. This mechanism is generally called inter-process communication (IPC).

As the processor is a valuable resource, processes that have nothing to do, like an idle server, have to be able to block their execution until there is new work for them to do. For the processes to be able to communicate synchronously, there must be a way to synchronize their behavior with an IPC protocol.

The work of Unrau and Krieger [Unrau97, Unrau98] deals with the problem of a synchronous client-server IPC protocol via queues. The client sends a request to an normally idle server and blocks to wait for the reply. The server runs, processes the request, sends the reply to the client, unblocks the client and blocks to wait for the next request. The protocol uses a flag variable called „awake“ which says whether the target entity of a queue is about to sleep. An unset awake flag implies that the server wants to be woken up.

Like this thesis, the work of Unrau and Krieger also examines the problem on how to notify the other party of new data when communicating over queues. We will use the described protocol to compare our protocol to a published solution in a later chapter.

2.3 Protection while using shared memory

The quest for high performance communication inevitably leads to the usage of shared memory to communicate packet data between entities as otherwise data would have to be copied. Copying adds additional latency to the communication path and consumes precious CPU cycles which could be used somewhere else.

However, the usage of shared memory for communication weakens the protection compared to the level provided by completely isolated protection domains. These protection levels can range from no protection over protection of integrity of shared data structures and the flows of control to the complete protection of all data and control.

As we will detail later, our design chooses to weaken the protection so that only the buffer data itself is unprotected.

Fbufs/IO-Lite

Fbufs [Druschel93] is an architecture to allow high-bandwidth communication across protection domains. The architecture allows the application to make an explicit choice about the level of protection of the communicated buffer data itself. When it chooses to have no protection of the actual buffer data, the architecture does not impose a performance penalty.

Buffer data lives in a shared memory region which is in the same virtual region in all address spaces.

Fbufs are buffer aggregates which consist of a list of <address, length> pairs which point to the virtual buffer region. The buffer aggregates are passed by value whereas its pointers to the actual buffers are passed by reference. Every fbuf spans one or more pages, a page contains at maximum one complete buffer. This is valid choice as high-speed networking uses buffer sizes well above a page size. Buffers are only written by the allocating entity, further protection domains may only read or copy the buffer. They argue that this is no problem as most applications either do change the whole buffer, so that a copy introduces no new costs or make isolated changes which are cheap by rebuilding a new buffer aggregate.

Protection works by access control lists, „incoming“ buffers have to be linked to a protection domain. This is a problem only for network data, as the packet demultiplexer has to find the protection domain and relate the packet to it. As protection domains are completely protected from each other, transferring an fbuf from one protection domain to another includes changing page protection or mappings.

This is not a cheap operation on all processor architectures as it implies flushing the whole TLB on architectures like IA-32. However, in most cases there is no need for mapping changes due to various assumptions made about the usage of the buffer. For example, the user may lower the level of protection by specifying a fbuf to be volatile, meaning that the contents of the fbuf may be changed by others. Furthermore, the fbuf’s pages may be cached so that it does not have to be cleared and remapped in the sender.

While we explicitly do not allow the driver clients to chose a level of protection, we have seen how this choice is made possible in other architectures by using the mechanism provided by the virtual memory system. The data in our logical buffers won’t be protected from the other party and, furthermore, the choice of implementing such a protection scheme would have to be made in the supporting infrastructure and not inside the driver and is thus out of scope of this thesis.

However, we see that fbufs uses logical buffer aggregates to which are transferred by copy over the protection boundary and which refer to buffer data that live in shared memory.

2.4 Packet buffer formats

Linux model – skbuffs

Linux uses a format called skbuff for its network buffers. It consists basically of a header structure, the struct skbuff and the actual packet data buffer. The skbuff has a pointer to the start of the buffer in memory and its length along with a second pointer that point to actual packet data of a layer. These second pointer is used to allow each layer to put packet data before or after data of an upper layer.

BSD model – mbufs

BSD [McKusick96] Unices use a buffer format which is called mbufs. Mbufs is a scatter-gather buffer format, a logical buffer there consists of a linked list of single mbuf structures.

```
struct mbuf {
    m_next
    m_next_pkt
    m_dat
    m_len
    m_type
    [ then up to 108 bytes of data ]
    };
```

Each mbuf structure contains a pointer to the actual buffer data, then length of this buffer data and a type field. The buffer data itself can be located just after the mbuf structure when it is small enough or be located in a separate block of memory.

IOLite

IOLite [Pai99] separates buffer data from buffer structure. Buffer data is contained in immutable pages. Buffers are logical objects, which consist of an ordered set of <pointer, length> pairs. These pairs point into a slice of the buffer data pages.

Uniform Driver Interface - UDI

UDI [UDI] is a portable driver development framework that allows drivers to be written once and then run on many platforms with UDI support. UDI's logical buffer are a name along with size information. There is no way to introspect the buffer's data.

Discussion

Linux a buffer format where each buffer only consists of one data buffer. BSD and IOLite chose to use logical buffers or buffer aggregates as their way to structure buffers. Each logical buffer consists of a ordered set of buffer descriptors of which each refers to a data region in memory. While IOLite completely separates buffer metadata and data, BSD buffer metadata can contain up to 108 bytes of data. The UDI framework does not define the internal structure of logical buffers. They are only used as a reference, assumably for portability reasons.

2.5 Summary

We have seen in this chapter, which drawbacks the current Sawmill Linux architecture has and how various aspects of a user-level driver architecture were solved by other systems.

Sawmill Linux uses unprotected shared memory for packet buffers and their accompanying data structures. Thus, failures in one party can easily spread to other parties and there is no protection. Furthermore, the stack-driver interface is tied to Linux's code and infrastructure semantics and is thus not general. The TCP benchmark showed, that Sawmill Linux uses much more processor cycles for doing the same work as its Linux ancestor.

We have looked at various other system designs, where the driver and its client are located in separate protection domains. We have compared the issues and extracted some of their solutions. We have found solutions for communication of data, transfer of control, and handling of address translations and page faults in software. Several publications identified the transfer of control as the main performance problem in this architecture class.

Finally, we have looked at published solutions to inter-process communication, protection of shared memory regions and packet buffer formats.

3 Costs in inter-server communication

In this chapter, we show will how big the performance gap between the multi-server OS Sawmill Linux and the comparable Linux 2.2.5 kernel in terms of CPU idle time is and that the receive path of Sawmill Linux consumes much more CPU cycles than the send path for the same packet rate.

The main part of this chapter will give evidence to the fact that transfer of control is the main cost source in inter-server communication. Each transfer of control implies a kernel entry, a change in the execution context (the so-called context switch), and the kernel exit to the new execution context.

We will attribute the busy CPU cycles to context switch events and their included costs like cache footprint changes and give evidence that the cost of a context switch can vary greatly and is dominated by its associated cost of cache footprint changes.

We will see that IRQs are the most expensive type of context switch. Along with the fact that the receive path has a much higher interrupt rate, we will use this observation to explain the relatively high costs of the receive path.

3.1 Measurement environment

3.1.1 Benchmarking network subsystems

In most network applications, latency is the most important property of a system as it gives an upper bound on the bandwidth (due to the acknowledgement window of the stream connection) or is an important factor in the performance of the whole system (like in parallel computing). Furthermore it is not easy make it better by adding more hardware like it is possible with bandwidth in many cases.

Another parameter of network systems is jitter, the latency variance. It is important in real-time or Quality-of-Service communication as a large jitter needs large buffers to smooth it which results in a higher latency. Whereas all systems in the literature where measured concerning the bandwidth, only some measure latency in real-world scenarios and none measure jitter.

Benchmarks of network subsystems can be divided into two groups. Microbenchmarks measure the timing of single operations. They provide insights in where the time goes during single isolated events. A standard microbenchmark for network subsystems is to measure the time needed to send and receive single packets, probably as a combined operation like sending ping packets to the system and account the used time in steps in that process like interrupt handling, checksum calculation, copies.

In contrast, macrobenchmarks allow us to see the system's performance from an application's perspective. They use operations which are assumed to be typical for

standard applications and provide a view on the performance under realistic conditions. A standard macrobenchmark is TCP bandwidth measurement, in some cases along with processor utilization when the bandwidth approaches the hardware's maximum. Additionally, single papers measure TCP latency, too.

All the measurements are done with varying message sizes up to hundreds of kilobytes, which are supported by high-speed network technologies like gigabit Ethernet and ATM. However, [Eicken95] argues that due to the fact that a low latency is central for parallel computing, the bandwidth while using small message sizes (a few hundred bytes) is important. Because parallel computing is very communication intensive as the result of one node depends on previous results of other nodes, performance of the communication system in this context can be measured through running standard parallel computing benchmarks like Linpack.

3.1.2 Performance measurement

A multi-server architecture should be competitive in terms of performance with other designs. Currently, the most optimized and inherently fast architecture is the one of a monolithic kernel. Therefore, we will compare the performance of the current Sawmill Linux architecture with the one of a monolithic kernel design, the Linux kernel.

Furthermore, the used Ethernet driver and TCP/IP protocol stack are extracted from the monolithic Linux 2.2.5 kernel. This way, the measured numbers give us more insight as all performance differences can be accounted to architectural differences as the platform and most of the driver's and stack's code are the same.

Like other multi-server architectures mentioned in the previous chapter, the current Sawmill Linux is able to saturate a fast Ethernet link and has a competitive number for latency. However, as we will see in the next sections, Sawmill Linux uses much more CPU cycles when compared to Linux 2.2.5 to achieve this result.

To be able to find the cause for this difference, we have to investigate the system's behavior with a workload where the structural differences come into play. This is the case when we transfer packets at high rates over the multi-server specific protection boundary between the driver and the protocol stack. Note that this high packet rate is not equivalent to an high data rate. As both Sawmill Linux and Linux 2.2.5 have the packet data in shared memory, the interesting event is here the number of packets transferred and not their size as the packet data itself does not move.

To get a high packet rate along with a realistic system workload, we will run the standard TCP throughput benchmark while measuring the CPU's idle time for both Sawmill Linux and an instrumented Linux 2.2.5 kernel.

Idle Time Measurement

CPU idle time is the span of time during which the processor has nothing to do. Modern processors like the IA-32 architecture allow the OS to put the processor in a halt state if there's no more work to do. The processor is put back in operation by an interrupt which is an event that signals the processor that there is some work to do. With this background, CPU idle time is the span of time from entering the halt state until an interrupt occurs. Time measurement is done with the CPU's time stamp counter, which allows a very precise measurement of time.

CPU idle time gets most interesting when set in relation to its measurement time span. This ratio of idle time and absolute time passed gives a number which tells how much work the processor had to do during the measurement interval. When the ratio approaches 100 %, the system spent most of its time in the halt state, when it approaches 0 %, the CPU was completely saturated. For the sake of simplicity, we will speak of idle time when we are referring to this ratio.

3.1.3 Measurement environment

Platform

We used two different computers for doing the measurements. The first one was a Pentium II machine with modifiable CPU speed and a PIC controller for interrupt handling. When not declared, the clock speed used was 300 MHz. The second one was a dual Pentium II 400 MHz machine with an APIC controller.

Kernel

The system ran on two different kernels. The first one, the „normal" kernel, was an L4Ka RC2 kernel, which only differed in two small bugfixes from the CVS version. The second one, the „precise" kernel, was a modified L4Ka kernel which used the APIC to be able to provide precise timeouts.

Both kernels have been instrumented with code to allow measurement of idle CPU cycles and the number of context switches, during a specific period of time. The numbers were measured by adding single measurements to an accumulative sum along with a counter which counts the number of single measurements in a field in the kernel info page. The timer interrupt handler is extended with code that stores the accumulative sum and its counter in another field in the kernel info page and resets the working counters then. This is done every five seconds.

A single measurement of idle CPU cycles was done by calculating the number of CPU cycles between the time when the CPU went idle and the next interrupt occurred and the CPU went on with its work. We read out the CPU's time stamp counter with `rdtsc` instruction just before the halt instruction and stored this value in a field in the kernel info page. After the halt instruction, the CPU sleeps; it comes back to work when receiving an interrupt. In L4Ka, this interrupt calls

one of two C functions via a short assembler stub, `handle_interrupt()` and `handle_timer_interrupt()`. The first thing we do in these functions is to read out the time stamp counter again, compute the difference to the value it had just before the halt instruction and add this difference to the accumulative sum in the kernel info page and increment the associated counter.

The number of context switches is measured in the `switch_to()` function of the kernel. Each time it is called, the „accumulative counter“ in the kernel info page is incremented.

As we mentioned in the previous section, CPU idle time gets its real meaning when compared to the real time passed. So we have to measure idle time as well as the real time in the kernel. Measurement of the passed real time is done by adding the programmed waiting time of each timer event to the real time counter when that event occurs. This form of real time measurement misses the time between the start of the timer event handling and the timer reprogramming.

While doing measurements with the precise kernel, it emerged that due to the frequent timer interrupts, the kernel clock went out of sync as the measurement errors due to this measurement method accumulated. We worked around this by measuring the actual time span of the accumulative sums with the CPU’s time stamp counter and normalizing the measured values to the five second interval.

Performance Benchmark

To put the system under a realistic workload, we used a macrobenchmark that transferred data via TCP/IP. This benchmark consists of a server which accepts a connections, and either sends or receives a given number of bytes, and a client which provides the server with the number of bytes that will be transferred and then sends or receives the specified amount of data. Furthermore, the client measures the time it took to transfer the data and calculates the bandwidth in Mbit/s from this.

The server was implemented so that it runs under both Linux and Sawmill. It uses a buffer of four pages or 16kbytes to do the socket operations. All measurements were done while transferring an amount of data which let the benchmark run well above 30 seconds.

We used the TCP benchmark in both directions, from and to the multi-server system. The two cases differ in the way Sawmill Linux is put under load. The TCP benchmark has a much higher packet rate in its send than its receive direction.

Due to the design of the used Ethernet hardware, each received packet implies an IRQ event. Thus, the IRQ event rate during the TCP receive benchmark is much higher.

3.2 Sawmill Linux vs. Linux 2.2.5, Consistency of idle time measurement

This first experiment shows the aforementioned performance gap between Sawmill Linux and Linux 2.2.5 and the relatively high costs of the receive path. Furthermore, we show that our idle time measurement is consistent.

Methodology

We measured the idle CPU time on the PIC machine in both the unmodified Sawmill system and a Linux 2.2.5 kernel using the TCP benchmark for both transmitting and receiving data.

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

Figure 3.1 Linux 2.2.5 vs. Sawmill Linux

The idle time columns show the lower and upper bound of the 99 % confidence interval of the measurement. As we described previously, the numbers themselves are the ratio of idle time and passed real time in a five second measurement interval.

Discussion

While having the same throughput, the numbers show clearly that Sawmill uses much more processor cycles to achieve this performance than Linux 2.2.5.

This experiment also reveals that the receive benchmark of Sawmill Linux consumes more CPU cycles than the transmit benchmark while running at a lower packet rate.

Additionally, this experiment shows the consistency of our measurement. Even with a 99% confidence interval, the lower and upper bound have never a difference of more than 0.4 % points. To make things easier, we will therefore only give the mean numbers in the next result presentations.

3.3 Context switches – The main cost factor in inter-server communication

The remaining part of this chapter will give evidence to the statement that context switches along with their associated costs are the main cost factor in inter-server communication. Furthermore, we will try to explain that the their associated costs makes them so costly.

3.3.1 Busy time is dominated by number of context switches

In our first experiment, we will show that the number of context switches is directly proportional to busy CPU time and thus the main cost factor in intra-server network communication is not driver or protocol code but the transfer of control between the servers.

Methodology

While running the TCP benchmark we measured the number of cycles the CPU was in the halt state. Each data point is the measurement of a different configuration to get a varying number of context switches.

We varied the configuration in a way so that there were more or less hardware interrupts in the driver and a varying number of control transfers per communicated packet between the driver and the stack.

By doing this the workload of the system due to code which handles packets in the protocol stack and in the driver remained constant because of the constant throughput. Only the number of context switches due to control transfer and interrupts changes.

We ran both the Transmit and the Receive TCP benchmark with these configurations. Note that the lower packet rate of the receive benchmark results both in less context switches and in smaller busy time due to less packet processing costs.

Results

The resulting diagram is shown in figure 3.1.

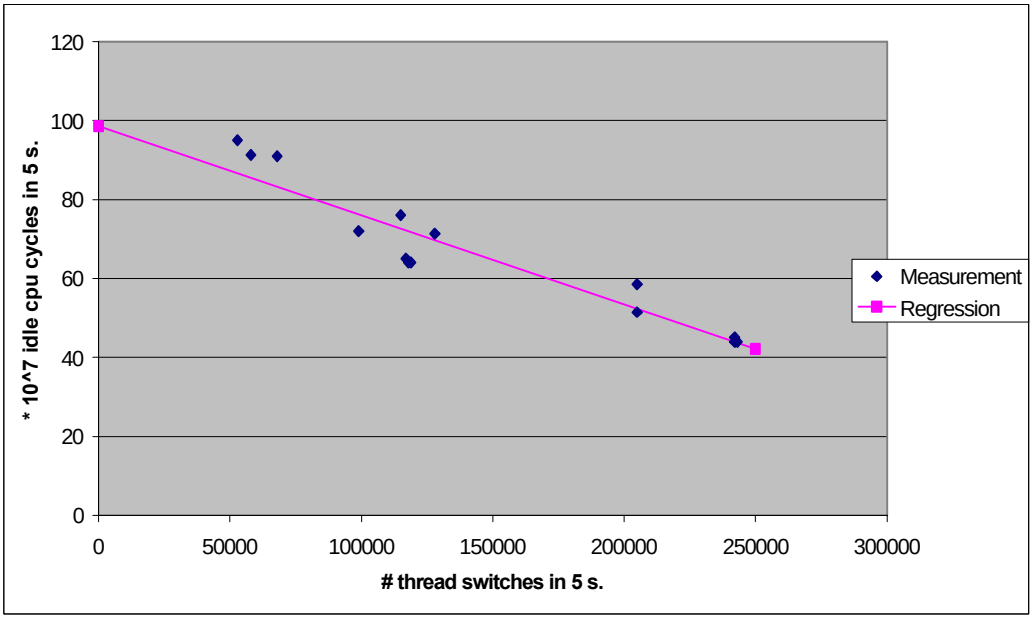

Figure 3.1 The correlation between context switches and idle CPU cycles

The correlation coefficient is –0.95.

Discussion

The resulting correlation shows clearly that the used processor cycles are mainly dependent on the number of context switches the system makes in a period of time.

While the code executed and the packet rate is different between the TCP transmit and the TCP receive benchmark, the costs of context switches dominate the varying costs of packet processing.

The variation of the data points relative to the regression line comes from the fact, that the numbers stem from various kinds of experiments, with different mixes of context switch types and thus different associated costs for those context switches.

3.3.2 Busy time can be accounted to packet processing, IPCs, IRQs

Under workload, the system has many different types of context switches. There are normal hardware interrupts, timer interrupts, explicit context switches with yield and associated context switches of IPC calls.

We will now show that different types of context switches have different associated costs and that IRQs a have measurable higher cost than IPCs. We will

give further evidence that the busy CPU cycles can mainly be attributed to the named events.

Methodology

We assume that the system does three things during TCP communication,

- packet processing,
- sending IPCs, and
- handling IRQs.

We measure the absolute frequency of these events and the idle time during a period of time and try to account the overall busy time to single events using a linear model. When this linear model fits well, we have a first hint for what the processor time is used.

We measured the absolute frequency of the events and the idle time in four setups:

1. TCP transmitting with an IPC saving method (multiple packets per IPC)
2. TCP receiving with an IPC saving method
3. TCP transmitting with the normal „one packet per IPC“ method.
4. TCP receiving with the normal „one packet per IPC“ method.

Results

The measured values of the experiment.

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

Table 3.2 Event rates and their absolute costs

Using the least squares method, we get:

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

Multiplying the numbers back gives:

Setup	Measured busy time	Accounted busy time	Absolute error	Relative error
Unit	*[s] / 5 [s]*	*[s] / 5 [s]*	-	-
Setup 1 (Tx)	1.98	2.01	0.03	1.5 %
Setup 2 (Rx)	2.75	2.73	-0.02	1 %
Setup 3 (Tx)	2.83	2.79	-0.04	1.4 %
Setup 4 (Rx)	3.5	3.52	0.02	0.6 %

Table 3.3 Difference between accounted event costs and the actual costs

Discussion

This experiment along with its analysis accounts the busy CPU time to the three mentioned event classes proportional to the frequency of the events. This implies that each event time includes time for executing code which is related to that event. However, as all processor time is related to these three event classes, each of the event classes includes to some extent processor time of code whose execution is not related to the actual event. Thus, the resulting numbers do not give a measure of how much each event exactly costs but show the relative importance of the events to the spent processor time.

The small relative error shows that the linear model fits well, both to the transmit and the receive benchmark. Therefore we can state that the used processor time can be explained mainly by IPCs, IRQs and general packet processing costs, and that there are no other unknown unrelated events which use processor time. The main influence are IRQs, followed by IPC and then by raw packet processing.

This experiment also explains why the TCP receive benchmark has higher costs than the TCP transmit benchmark at a lower packet rate. When we look at the measured numbers, we observe that the receive benchmark has a higher IRQ rate. Along with the fact that IRQs have higher costs than other types of context switches, this explains why the receive path of Sawmill Linux is more expensive.

3.3.3 Context switch costs are dominated by their associated costs

To get a better understanding of what aspect of a context switch causes the costs, we increased their number by adding context switches which have nearly no associated costs in terms of cache footprint changes.

Methodology

We ran a version of the system with a setup called „interval polling“ in transmit direction with the precise kernel on the APIC machine which allows us to directly influence the number of packets per control transfer. We measured idle CPU

cycles and the number of context switches for a number of interval lengths. Then we repeated this experiment in presence of a low-priority worker thread. This worker thread incremented a counter in a loop. We read that counter and reset it every five seconds. The resulting counter value is a measure for the idle CPU time.

Results

Setup	Normal		With worker thread	
Interval length	Context switches	Idle time ratio	Context switches	Idle time ratio
	/ 5 [s]		*/ 5 [s]*	
73 us	158000	50 %	222000	48.5 %
240 us	110500	56.5 %	134000	53.6 %
1010 us	93000	59.5 %	119000	59 %
4980 us	90000	61 %	112000	60.6 %

Table 3.4 Comparison between different context switch rates and their costs

Discussion

The numbers show that the presence of a worker thread has a clear influence on the number of context switches. However, while they are negligible at longer polling intervals, they have a small but noticeable influence at high polling rates on the CPU idle time.

At a first glance, this experiment seems to contradict the previous results. When we recall, however, what the worker thread does, incrementing a counter in a loop, we see that it has a very tiny cache footprint. Because of this, the context switches to and from it have only a small amount of associated costs and thus the context switches are cheap in terms of used processor cycles.

After we have seen in the previous section, that there are context switches which dominate the whole benchmark costs, we have found here a type of context switch which as nearly no costs due to the obvious lack of associated costs.

3.3.4 Overhead of L4Ka IRQ handling

As IRQ handling seems to be a expensive operation in our environment, we try to quantify its cost relative to Linux's.

Methodology

To learn how the costs for interrupt processing differ between L4Ka and Linux 2.2.5, we measured the number of cycles it takes for L4Ka interrupt to come from the interrupt handling routine to the L4Ka user-level IPC reception.

We read out the processor time stamp counter at the start of the C function which is linked to the IDT and save it in a field in the kernel info page. We read out the time stamp counter a second time just after the reception of the IPC at user level. We calculate the difference between those two values and accumulate it in a field in the kernel info page along with a counter of the number of values accumulated.

Results

The accumulated value divided by the number of values as noted in the kernel info page is between 1200 and 1300 cycles with an L4Ka version which uses the PIC running on an Pentium II @ 300 MHz. On an 400 MHz Pentium II with an APIC kernel, it takes about 6000 cycles.

Discussion

As Linux has nothing to do between the actual hardware interrupt and the call of the associated interrupt handler, the measured number of processor cycles is the overhead of L4Ka to begin processing interrupts compared to Linux. Note that it does not include the time it takes to establish the full context (in terms of cache footprint) of the interrupt handler and to reestablish the full context of the interrupted thread afterwards.

At a rate of about 32000 receive IRQs in 5 seconds, this overhead accounts for 2.8 % points of the CPU utilization for the PIC measurement at 300 MHz. However, this fact alone can not explain the huge differences in CPU utilization of the transmit and the receive path.

3.3.5 High costs of receive IRQs

We will now look on a number of CPU counters to find evidence for what might cause the high costs of IRQ events.

Methodology

We measured two driver-stack configurations which are called „IRQ polling“ (setup 1) and „Always Notify“ (setup 2) for both the TCP transmit and the TCP receive benchmark on the precise kernel. These configurations result in a different numbers of context switches while running the benchmark. During the benchmark we measured the TLB miss rate, the L2 cache line removal rate with the CPU’s performance counters and the context switch rate, and the packet and IRQ rates.

Results

Setup		Idle CPU cycles	Context switch rate	TLB miss rate	L2 line out rate	Tx path packet rate	Rx path packet rate	IRQ rate
Unit		* 10^6 [cycles] / 5 [s]	/ 5 [s]	* 1024 / 5 [s]	* 1024 / 5 [s]	/ 5 [s]	/ 5 [s]	/ 5 [s]
1	Tx	122.5	97500	9970	163	38000	19000	24000
	Rx	100	199000	6800	682	19000	29000	37000
2	Tx	116.5	173000	10000	182	38000	19000	24000
	Rx	100	199000	6700	690	19000	29000	37000

Table 3.5 Investigation of performance counters during the benchmark

Discussion

The linear model of section 3.3.2 showed that the high relative costs of the TCP receive benchmark directly result from the high number of costly IRQs during the receive benchmark.

The measured L2 line out rate is the number of L2 cache lines removed from the cache for any reason. Most of the removals are caused by cache line replacements, so their number includes a transfer of a line from memory to the L2 cache. When a modified cache line is removed, it has to written back first. The measured performance counter however does not allow to say whether this had to be done and thus we can not give an exact number of how much each L2 line out event costs. We will now calculate the lower and upper bound for the measured event.

The cache lines of a Pentium II are 32 bytes wide, its memory bus is 8 bytes wide. Each L2 cache line load thus stalls the execution for a minimum of one memory transaction (for “critical word first” behavior) which needs 4 cycles for a 4-1-1-1 memory cycle. On the used 400 MHz system with a 100 MHz memory bus, this equals to 4 * 400/100 = 16 processor cycles for the L2 cache line load alone. If the line has to be written back in advance and the processor stalls for the full cache reload, the processor would stalled for 2 * 7 * 400/100 = 56 processor cycles.

This measurement shows that during the receive benchmark, there is a much higher number of L2 cache line removals compared to the transmit benchmarks. The difference of the L2 line out rates in the two setups would need a L2 line out cost of 42 and 31 cycles, respectively, to fully account for the cycle difference. While we do not know the exact costs for an L2 cache line out event, the calculated interval is close enough so that we have evidence that the difference in idle CPU cycles between the transmit and the receive benchmark is caused mainly by L2 cache events.

3.3.6 Attributing context switch costs to cache refills

In the experiment with the low-cache-footprint worker thread in section 3.3.3, we gave evidence that context switch costs are dominated by their associated costs. While the lack of time did not permit to investigate this issue in greater detail in this thesis, we did another experiment to give further evidence.

In the experiment of section 3.3.5, we used the performance counters to find evidence for why the IRQs and thus the receive benchmark is relatively more expensive when compared to the transmit benchmark. We will now try to find evidence for the general cause of associated costs of context switches.

Methodology

We measured the number of context switches, the idle CPU time, the TLB miss rate and the L2 cache line removal rate on the precise kernel with three setups:

- „Always Notify“, where we have one notification per packet on the transmit path
- „Interval Notification“, where we send one notification per time interval
- „Interval Polling“

The receive path ran with „Always Notify“ in all three experiments; we only measured the TCP transmit benchmark

We chose the parameters for Interval setups in a way that two experiments have the same context switch rate. Hereby we are able to compare CPU idle time and cache behavior.

Results

Setup	Interv.	CPU idle ratio	Idle CPU cycles	Context switch rate	TLB miss rate	L2 rem. rate
Unit	*[ìs]*		** 10^6 [cycles] / 5 [s]*	*/ 5 [s]*	** 1024 / 5 [s]*	** 1024 / 5 [s]*
Always Notify	-	58.5 %	116.5	173000	10050	122
Interval Notification	230	56.0 %	112.0	151000	10050	100
	170	54.5 %	109.0	170000	10200	178
Interval Polling	74	50.0 %	100.0	158000	10400	115

Table 3.6 Context switch rates and their impact on cache change rates

Discussion

The first observation is that over experiments with a difference in CPU idle time of about 9 % points, the TLB miss rate does not change much.

If we then compare the numbers of „Always Notify“ and „Interval Notification“ of 170
„Always Notify“ has 7 % more idle CPU cycles, but 32 % less L2 cache line removals.

We then compare „Interval Notification“ with „Normal Polling“ at the same context switch rate of about 155000 context switches/5 sec. „Interval Polling“ has 12 % more idle CPU cycles while having 13 % less L2 cache line removals.

While a rising number of context switches has no influence on the TLB miss rate, there is a noticeable inverse relationship between the L2 cache line removal rate and the number of idle CPU cycles.

3.4 Summary

The experiments of this chapter have shown that context switches which are caused by hardware interrupts and notification IPCs explain the costs of inter-server communication for both the transmit and the receive path at various data rates. Furthermore they gave evidence that the associated L2 cache footprint changes of each context switch could be responsible for the high costs.

We will now use these observations to design the communication interface of an user-level hardware device driver so that it is capable of operating with a good CPU cycle consumption when compared to a monolithic system.

4 Interfaces of a user-level device driver

This chapter presents the design of the communication interface for providing the device driver service to clients and the interface to the supporting I/O infrastructure. These interfaces will provide the functionality which is needed to allow high-performance communication between the driver and its clients and to manage physical requirements of the device hardware.

After an introduction, we will present a design for the communication interface and compare it to the literature. Then we will describe the interface to the system's I/O infrastructure. This chapter will close with the investigation of protection and security issues implied by our interface design.

4.1 Introduction

Hardware device drivers, or short drivers, provide an adaptation of the proprietary interface of the hardware device to a generalized interface that abstracts from the implementation choice of various hardware vendors and is the same among all devices of a hardware device class. This work deals with the device class of local area network (LAN) network interface cards (NICs).

The device driver clients are components or programs that need the service of a device driver to do their task. Because of the generalized interface to the device drivers, they can do this without needing to be aware of special issues of particular hardware devices. Here, usual clients to the NIC drivers are protocol stacks, monitoring programs or user-level communication software.

Due to the roles the entities play in the I/O subsystem, we differentiate between drivers and driver clients (or clients). Drivers are responsible for providing an abstracted multiplexed interface to hardware devices. The driver itself is communicating directly with the hardware via the IO region and interrupts (IRQs). Driver clients need the service offered by the driver to do their job.

Device drivers and their clients need a supporting infrastructure to be able to communicate. A special role in this infrastructure is played by the device driver manager (DDM). It is responsible for providing an environment, where driver clients can associate themselves with device drivers and communicate with each other.

Specifically, the device driver manager provides a naming system that allows clients to find drivers, it manages the association between a driver and a client and handles the shared memory region which is used for communication between a driver and its clients. The DDM only provides a service for the device drivers and is not part of this thesis.

The interface to a driver can be separated into two parts, management and communication. The management part of the interface deals with initialization,

configuration and graceful shutdown of the driver. It has no performance requirements, it is only expected that it provides a general interface that allows management tasks to be done efficiently and in a straightforward way. There are no demands for the management side of the driver's interface that are special to a microkernel environment as the interface's function are rarely used.

The communication part of the device driver interface is responsible for the operation proper of the driver, the transmission and reception of network packets. This interface is subject to performance and resource constraints that affect bandwidth, latency and jitter as well as available processor cycles and memory.

A microkernel environment poses a special problem here as the separation into separate address spaces and scheduling units creates undeterminable behavior and non-linear communication overhead. Furthermore, resources in multi-server operating systems are not managed centrally anymore. Due to these observation, this work concentrates on the communication part of the device driver's interface.

In our experiments, we identified context switches as the major cost of separating a monolithic driver-client bundle into single servers. Apart from normal preemption, context switches can be differentiated into interrupts, IPCs, and IPC timeouts, each with different costs in terms of used processor cycles.

To achieve maximum performance in terms of idle processor cycles, we use IPCs only when they are absolutely necessary. Buffer transfers are handled completely via a shared memory region, the only application left for IPC are notification or synchronization. We make our shared data-structures non-blocking so that there is no need for synchronization primitives using IPC. Notifications will are kept at absolute minimum by using advanced notification techniques and special hardware support of the NIC.

In this design, the only way for servers to compromise each other is via this shared memory region. We will take countermeasures so that servers are not able to influence each other by employing carefully designed data structures and using the minimum required protection provided by the virtual memory system.

4.2 Requirements

We will now deduce a set of requirements for the interfaces by looking at design implications of the multi-server environment and the driver client's demands.

The advantages of the microkernel/multi-server OS design, which are the ease of configurability of the system and the mutual protection of the OS components with its implied fault tolerance and security must be preserved by any interface design. This means in particular that if we weaken the protection boundaries of the servers to allow more efficient communication, we have to make sure that this shared data is protected in the following sense:

Entities of the architecture should not be able to corrupt each other's correct operation, neither maliciously nor by accident.

Clients in the architecture should not be able gain illicit access to data of other clients, neither directly nor indirectly by using the driver .

In the driver scenario, that means that neither the driver is able to influence the correct operation of its clients, nor are the clients able to corrupt the driver and other clients or access the data of other clients. If one of these rules is violated, the entities are allowed to stop the communication relationship immediately.

Note that we do not include the buffers' contents in this protection requirement. By using the driver's service, a client trusts it in the sense that it will perform the requested operation without maliciously changing the data. That leaves the issue of data changes that occur through bugs in the driver.

If the client can't tolerate wrong buffer contents it will have protect itself as can not be sure whether the service provided by the driver in cooperation with the hardware is free of bugs. The driver itself does not depend on the protection of the buffer data as it does not depend on the buffers' contents in its operation.

Experiences with previous designs of a driver interface and the driver's resource management have shown that a naive approach to this problem leads to an inefficient implementation which consumes much of the processor's idle time during communication.

Furthermore, though the multi-server design has advantages over the monolithic design, the costs paid for this headstart should be kept at a minimum to keep the attractiveness of the multi-server design approach. Thus, the driver interface should be competitive in terms of *performance* with other designs, especially the monolithic one:

Minimize the difference when compared monolithic architectures in performance measures while complying to the other requirements

Beside the hardware device itself, the main resources the driver has to manage is buffer memory. This management involves interaction with the I/O subsystem which needs a well defined-interface that allows a general and efficient way to handle the memory. In particular, memory management means the handling of important hardware features of physical memory.

A subsystem is here in a natural tension: for maximum performance it would like to claim as much memory as available but that memory is then missed in other subsystems. The policy of how to distributed the available physical memory among components is managed by the resource management servers. The driver has to support any policy of these servers with its interface to them:

The involved memory should be able to be backed in a flexible way by physical memory.

As a multi-server operating system has generally the same application areas as a monolithic one, we do not want to restrict its applicability artificially by introducing special demands in the I/O subsystem. Therefore the architecture must be general to satisfy all flavors of network I/O demands for various application areas.

Generality (which includes configurability) in particular means:

> *Allow multiple drivers with each multiple clients in the system.*
>
> *Design for many possible application areas, like WAN, LAN, SAN and other network I/O.*

4.3 The Communication Interface

The first issue to which we will propose a solution for is the interface to the driver itself. As we showed in the previous section, only the communication part of this interface is relevant due to its high communication rate and its special demands in a microkernel environment where it crosses protection boundary.

The communication interface between the driver and a client is designed for high-throughput low CPU-utilization communication of buffer descriptors. The communication runs over two first-in first-out (FIFO) queues, the transmit (tx) queue for communication from the client to the driver, the receive (rx) queue for communication from the driver to the client. The queues are in pinned shared memory. This is possible because they do not need much space as they only hold buffer references.

Like all communication, the driver's interface can be separated into a raw communication part which provides data flow and a part which provides flow of control, the "synchronization" part. In the following sections, we propose an interface where both parts are kept separate.

This interface is minimal in the sense that none if its elements is redundant, and policy free in that it does not dictate a way to implement it.

These implementations are described at the end of this chapter. The interface design allows naive implementations with minimal latency and high consumption of CPU cycles as well as sophisticated ones which provide control over the latency-CPU consumption tradeoff.

4.3.1 Communication

While communicating at a high data rate, the queues are a central point. The driver interface uses two FIFO queues, the transmit and receive queues. Each buffer has to put into the queue with the enqueue() operation and then taken out by the other party with the dequeue() operation:

```
void enqueue( buffer_desc bdesc );
```

```
buffer_desc dequeue();
```

To decouple the progress of the consumer and the producer of these queues and to be able to operate at high data rates with minimal amount of synchronization between the two parties, we implement the queues as non-blocking data structure.

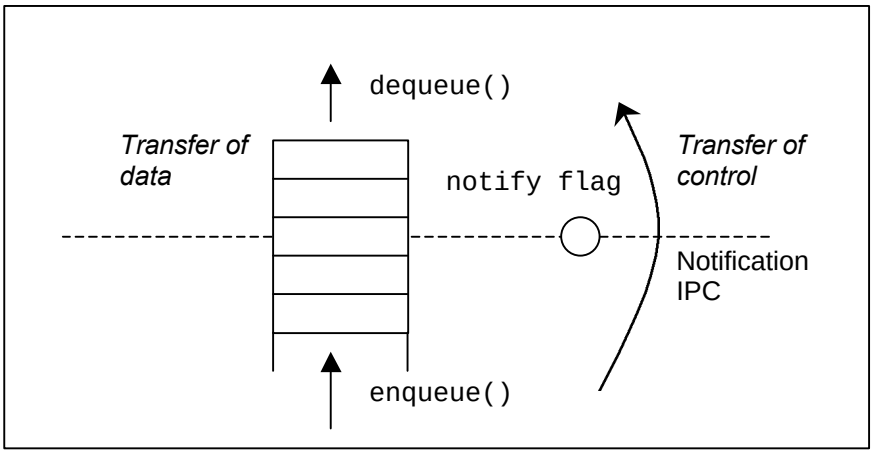

Figure 4.1 The communication interface

For this task, two algorithms are described in the literature. [Michael96] presents a queue implementation based on a list structure which is lock-free and non-blocking. It uses a compare-and-exchange operation which is available on many processor architectures. Tsigas and Zhang [Tsigas00] have a non-blocking FIFO queue which works on a ring buffer data structure. It uses the compare-and-exchange and outperforms every other FIFO algorithm including spinlock based-ones under any level of multiprogramming on a ccNUMA architecture.

The usage of the processor's atomic operation has one implication. It only works on words, thus, the contents of the queue, the queue's elements may not extend the size of one word, i.e. 4 bytes. If we want to transfer objects that extend this size we have to work with references to these objects.

As described, non-blocking FIFO queues are available as linked-list structures and as ring-buffers. As opposed to ring-buffers, linked-list structures have the advantage that they have no inherent limit on the number of elements that the queue may contain. This is, however, also a disadvantage as operations on unlimited data structures are subject to certain protection problems as iteration operations are potentially unbounded. We will look at this issue in detail in a later section.

4.3.2 Synchronization – The wake-up protocol

After having described how data flows bidirectionaly through the queues, we have to answer the question how the consumer is notified of the fact that it has to look in the queue. In a naive implementation without any further mechanisms other

than the `en-/dequeue()` operations, the consumer would have to spin on the queue until it finds new data in it. This needs an unnecessary amount of CPU cycles and thus we would like to have a mechanism which provides a mean for the consumer to block until new data is in the queue.

The driver and its client are in a producer-consumer relationship. The producer puts new work into the queue, the consumer dequeues this work packet and processes it. If the queue is empty and the consumer has done its work, it would like to block until new work is available, otherwise it would have to spin on the `dequeue()` operation until an element of the FIFO queue can be dequeued.

The entity responsible for unblocking a waiting consumer is the producer. It should unblock the consumer if it enqueues a new element into an empty queue and the consumer is blocked. In our scenario, however, the situation is more complex because of the separation into address spaces and the choice of non-blocking data structures. A the producer and the consumer live in their own respective black boxes, the producer can't detect whether the consumer is blocked.

Therefore we introduce a new communication channel which is from the consumer to the producer in opposite queue direction. We use a flag variable which is readable and writable by both the producer and the consumer to request an unblock of the consumer if the producer enqueues new work.

On the L4 kernel, blocking and unblocking of threads is implemented via L4's IPC mechanisms. To suspend its execution, a thread has to call the IPC receive system call, which allows the thread to wait for an IPC with a specified timeout. To unblock a suspended thread, the notifying thread has to send the suspended thread an IPC. On reception, the unblocked thread wakes up and continues.

The consumer uses this IPC reception system call to wait for IPC from anyone. If the notification flag is set, the producer sends an IPC to the consumer thread of the queue in enqueue the work packet. This IPC can either have an unlimited timeout to block until the consumer is really notified or have no timeout to return immediately. Both options have their pros and cons.

If one chooses to use an unlimited timeout, the producer can be sure that the notification went through when the system call returns. However, if the consumer does not wait for the notification IPC, this system call can be blocked for a long time. This scenario can even be used for a denial-of-service attack against a driver. If the driver notifies a client with an unlimited timeout, the client may just have specified a consumer thread that does not wait and the driver's notificating thread would be blocked forever.

A no-timeout notification IPC does not have this problem, but one must ensure that the notification IPC actually succeeded in unblocking the consumer. This is an issue as the consumer might have decided to block and has already requested

the notification but was preempted before being able to block. The producer would then detect the notification request and send a notification IPC to an unblocked consumer thread. The IPC would return with an error, the producer would continue doing its work but the consumer would remain blocked even there is some work in the queue.

To account for this race condition and thus to be able to use no-timeout notification IPCs, we first have to check the notification IPC's error code to detect whether the notification IPC went through. If the IPC did not succeed, we have to retry it until the error code signals success. Furthermore, we have to switch to the consumer thread during this retry loop so that it can make progress, reach the blocking IPC reception call and thus be able to receive the notification IPC.

If we don't trust the consumer thread here, we can limit the maximal number of notification retries or do some other work in-between.

We have now shown how to request a notification, how to wait for that notification and how to send the notification properly. The remaining issue is where and how to reset the notification flag.

There are two thinkable choices for where to reset the notification flag: In the producer, after he has sent the notification or in the consumer after he has received the notification.

If we reset the notification flag in the consumer we are prone to a race-condition. The consumer might receive the notification but gets immediately preempted before being able to reset the flag. The producer wants to notify him again and finds a set notification flag and sends a new notification without one being requested or awaited. This can not happen when we unset the flag in the producer because every delivered notification implies an unset notification flag.

In pseudo-code, for no-timeout notification the notify function would be:

```
if( queue.notify == true )
      queue.notify = false;
      while( error == true )
            error = send_IPC( consumer );
            switch_to( consumer );
```

A question that immediately arises now is whether this interface weakens the protection boundary. A possible attack is to send a notification IPCs to drivers which would result in queue lookup of the consumer. Done at a high rate, this could influence the working of the system. There are two ways to argue against this attack.

First, we could force the driver client to register all possible notification sending threads with the driver. This way, the driver could start a thread per possible

sending thread and restrict its notification IPC to the respective thread on the sender side. Thus only threads of the client could send notification IPCs.

However, a driver implementation might chose to use only on consumer thread which would then have to be notified by a number of producer threads of different clients. On an L4 kernel, this implies that we have to use an open wait IPC reception which basically allows then everybody to wake up the receiver thread as there is no possibility to restrict the possible sender threads to a number of threads identities.

The impact of this, however, is very small. If a malicious thread chooses to flood the consumer with notification IPCs, the only impact would be that the driver checks all the queues and goes then back to sleep. Furthermore, in current L4 implementations, the thread that sends an IPC donates its timeslice to the receiver, so the consumer would do this using the timeslice of the malicious thread.

4.3.3 Synchronization – API

The described mechanism is covered under the following API:

```
void wait_for_notification( Queue );
```

This function is called by the consumer to block until it gets notified by the producer. A return from this function usually implies the presence of new work in the queue.

```
void request_notification( Queue );
```

When the consumer detects that it will inevitably block and wait for an notification, it calls this function to request a notification from the producer.

```
void notify( Queue );
```

This function is called by the producer just after its call to the enqueue operation. It checks for the notification flag, and notifies the consumer if it found it set.

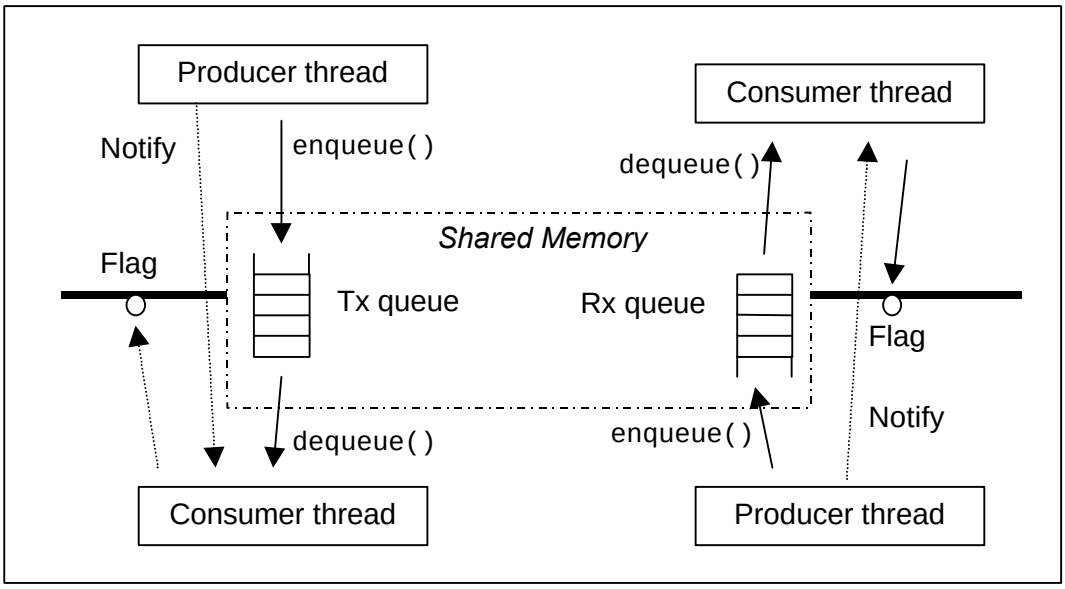

Figure 4.2 The communication interface

Figure 4.2 shows the communication interface in summary. Data is transferred via the synchronization-free queue. When the notification flag is set, the enqueueing party has to send an notification IPC.

4.3.4 The wake-up protocol – Comparison to literature

The design space for wake-up protocols on the L4 microkernel is restricted to emulation of synchronization primitives using the given L4 IPC mechanism and the processor's atomic instructions. Well-defined synchronization primitives are counting semaphores, binary semaphores or locks and monitors.

Description of the Unrau-Krieger protocol

As sketched in the Related Work section, the protocol of Unrau and Krieger [Unrau97, Unrau98] is made for synchronous client-server communication over FIFO queues with counting semaphore synchronization using atomic operations. It consists of two symmetric parts for enqueueing and signaling and waiting and dequeuing. These two parts make up the atoms of a synchronous request-process-reply communication. We will concentrate on the communication in one direction. The `tas()` function here is an atomic test-and-set instruction which sets the given variable and returns its previous state.

The *send* operation:

```
while( !enqueue() )     // queue full?
  sleep( 1 );
if( !tas( awake ) )     // Server about to sleep? Tell him not to
  V();                  //   notify
```

The *receive* operation:

```
while( !dequeue() )     // queue empty?
  awake = 0;            //   „about to go to sleep“
  if( !dequeue() )      //   still empty? Client had chance to
                        check awake
    P();                //     wait for wake-up
    awake = 1;          //     „I’m awake“
  else                  //   queue got entries inbetween
    if( tas( awake ) )  //     sync. unblock-block behavior with
                          client
      P();              //     block only if client is about to
                          block
```

There are three relations in which these two algorithms can interact:

- *send* enqueueing before first *receive* dequeuing. Sender enqueues, receiver dequeues with no further interaction.
- Both *receive* dequeues before *send* enqueue. Receiver has set awake to 0 and will call `P()` or already has blocked. Sender enqueues, sets awake to 1 and unblocks the receiver if it has already has blocked
- *Receive* sees the queue empty in the first dequeue, then *send* enqueues. The *send* then checks whether the receiver already set awake to 0 and resets it to 1 to communicate that it will send not call unblock and thus the receiver must not call block. The *receiver* checks for that message and reacts accordingly. So if the *receiver* already set the awake to 0, both parties will go through their unblock and block path, respectively, otherwise they communicate via the awake flag not to do so.

The protocol uses a flag variable called „awake“ to communicate two things:

- The *receive* party signals that it will block soon or that it already has blocked, and
- the *send* party uses it as a message to notify the *receive* party that it should call the blocking function as the send party will call *unblock*

Unrau and Krieger use the kernel’s counting semaphore implementation and its implied block and unblock functionality to send the server to sleep and to wake it up again. This implies that block and unblock operations are memorized in the sense that two unblock calls have effect on two block calls and are not consumed by the first block call and that an unblock operation affects a later block operation.

The counting semaphore is used in a strict way: only one server thread uses the `P()` (wait) operation, the `V()` (signal) operation is only used by the client threads. This implies that there are no consecutive block calls without an unblock operation in-between as the only server thread is blocked after the first `P()` call and needs an `V()` call to get unblocked to be able to issue the next `P()` operation. Thus, the semaphore counter is never below –1.

Comparison

Both the Unrau-Krieger protocol and ours are made for asynchronous communication over queues, trying to not to use the kernel's synchronization calls, if possible.

However, Unrau-Krieger's receiver blocks, if there's no data the queue while our detection of whether to block or not is determined by the whole state of the receiving communication party.

The *send* operation of our protocol when using semaphores instead of IPC looks like:

```
while( !enqueue() )
    sleep( 1 );

if( notify )              // notification requested?
    notify = false;       //   signal „I will notify“
    V();                  //   notify
```

When interpreting the notify flag as the inverse awake flag, there's no difference to the Unrau-Krieger protocol, apart from the atomicity of the test-and-set function, which relates to the unatomic functionality of our if and notify unsetting line.

However, our notify flag has not exactly the semantic of an inverse awake flag. For example, it is not possible to use a *receive* function like:

```
if( !dequeue() )
    notify = true;
    P();
```

This receive function would be prone to a race condition when the sender enqueues in an empty queue after the receiver has checked that queue.

The semantic of our notify flag implies that the receive has set it when it will inevitably block. This is not the case with this receive function, an empty queue implies that it will go to sleep, but the notify flag is not yet set.

If one would extend our notification scheme to be usable for the same purpose as Unrau-Krieger's, one would end up with their algorithm with a notify flag which is exactly their inverted awake flag.

4.4 Buffer format

We now know how a driver client and the driver itself can transfer buffer descriptors to each other. What do these descriptors look like? As we saw in the Related Work chapter, there are a number of buffer formats who have two things in common. A logical buffer in general consists of buffer metadata and the buffer data itself. Buffer metadata is usually an ordered set of <buffer data reference, buffer data length> pairs, of which each refers to a part of the logical buffer. Thus, a buffer descriptor is either the whole metadata set itself or is a reference to it.

The contents of the buffer data references have to be derived from the structure of the virtual memory system. Each buffer data reference describes a reference to a location in virtual memory in the producer's address space. Depending on how this address space is structured, the reference can be either a plain virtual memory address or a combination of a memory region descriptor and an offset therein.

While we have not restricted ourselves to a specific buffer format, we know now how the buffer descriptors look like and which information they carry. The first observation is that they are small, usually only one or a few words. However, as soon as they are larger than 4 bytes, we can not put them directly in our FIFO queues but have to work with references on them.

The other important thing to notice is that the location of buffer parts is described by references to the virtual memory locations in the producer's address space. To be able to supply the hardware with the physical translation of these locations, we have to be supported by the I/O infrastructure as this information is located in the virtual memory subsystem.

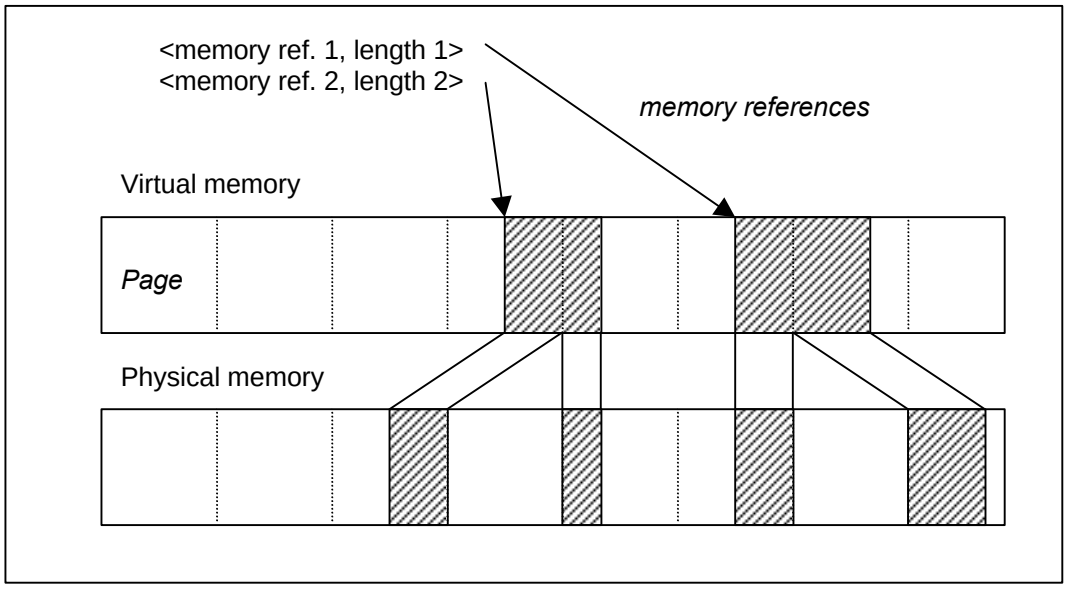

Figure 4.3 A scatter-gather buffer and its virtual and physical layout

Furthermore, if the driver itself needs access to these locations the memory has to be mapped to the driver's virtual address space. This is the case when the driver has to do checksums, copy buffer data to simulate scatter-gather transmission or has to copy from private memory to client's memory for the demultiplexing of received packets to clients. To sum up, we note that the nature of buffers require supporting functionality from the virtual memory system to convert virtual to physical addresses and to map the data to the driver's address space.

When the hardware-device supports scatter-gather transmission, the driver has to provide it with a list of tupels which describe contiguous physical memory

regions. These are derived from the buffer's metadata by converting virtual memory references into physical memory addresses and by describing buffer parts that are non-contiguous only in virtual but not in physical memory by multiple physical regions. In Figure 4.3, you see two buffer parts that are contiguous in virtual but not in physical memory. Thus, the device hardware has to be provided with the descriptions of four physical buffer parts.

4.5 The driver's access to buffer data

There are at least five scenarios where the driver needs access to the physical representation of the buffer data itself:

1. The device does not support DMA, the driver has to copy the data to the device manually.
2. The device does not support scatter-gather DMA and the driver has thus to copy the buffer parts into one buffer.
3. Received data must be demultiplexed inside the driver to several driver clients and thus copied from a private receive buffer to a client's buffer.
4. The driver has to do checksumming of buffer data.
5. The driver has to convert the data from and to a device specific hardware format.

Performance reasons do not allow to map buffer data on demand to the driver in these cases. Thus, all buffer data must be located in a shared memory region that is accessible by the driver and a client. The demand for this shared memory region is independent from the demands formulated in the previous section. The requirements of pinning, mapping and virtual-physical translation remain unchanged.

4.6 Buffer memory management

One question that is yet unanswered is how logical buffers and their buffer data are destroyed and created. The way this can be done depends on whether the buffer data is in a shared memory region or not.

With a shared memory region

Memory management in general is the task of bookkeeping which parts of the memory are in use and which are free. Here, the memory to manage is the shared memory region.

Everytime the client wants to send data, it has to allocate buffer memory (i.e. mark a piece of memory as "used") which then has to be freed again by the driver; on reception of data, the driver has to allocate memory which is freed then by its client once its done with the data.

The shared virtual memory region consists of a set of pages. Both entities must be able to allocate and free memory blocks from the set of pages. The notion of ownership of memory is defined implicitly. All memory that a party has allocated and that the party has pointers to is owned by the party. Thus, we have three pools of memory blocks: free ones, ones that are owned by the driver and ones that are owned by the client.

The transfer of memory blocks from and to the free pool is handled with the `alloc` and `free` functions. The exchange of memory blocks between the driver and a client is defined by the special agreement, that all memory blocks that were communicated over one of the two FIFO queues are then owned by respective other party.

We chose to implement a shared slab allocator for managing the shared memory region. Slab allocators embed their management data structures in the memory the have to manage. This type of allocator allows `alloc` and the `free` function to be implemented in way that no direct communication between the client and the driver is needed.

Without a shared memory region

If we chose to have no shared memory region, we have to provide a method for the driver to allocate memory in the client's address space. This is an operation that has to be done per received packet, and, as we have seen, normal communication via IPC is an expensive method to implement that.

Furthermore, buffer allocation is a synchronous function of at least one parameter, the buffer size. We have to provide the allocator with at least the size parameter, wait until it allocated a memory region for us and receive a descriptor for that allocated region.

Even when we succeed in implementing this mechanism with an IPC-less technique, it is no operation that can be done once per received packet. Each received packet would need a full communication round-trip with the allocator before it can be processed any further.

There are two solutions to this problem. First, we can try to remove the "size" parameter from this allocation communication and thus make it unidirectional. We can implement this by providing the driver with queues with pre-allocated packet buffers of various fixed sizes. Second, we could export the memory management structures of the client via shared memory to the driver. The driver can then manipulate these structures allocate or deallocate memory in the driver's address space.

4.7 The interface to the supporting infrastructure

The previous sections of this chapter described how a driver client and the driver itself can transfer buffer descriptors to each other, how they notify each other of the arrival of new data, how these buffers look like and how their memory is managed. Some of this functionality could not be provided by the driver alone, it needs support from the I/O infrastructure with the virtual memory management:

1. The driver must be able to convert memory descriptors of the client's address space to physical addresses
2. In some scenarios, the driver must be able to access the buffer data itself and thus map it into its own address space.

We will sum up the needed functionality into one interface. However, as we dictate in no way the implementation of it, its functionality could be provided by one or several components.

We will proceed with a detailed investigation of the functionality that the driver needs from external entities. Afterwards, we will use this information to design the interface.

4.7.1 Physical requirements of hardware devices

Most hardware devices are able to fetch data directly from the computer's memory via a mechanism called Direct Memory Access (DMA). The hardware device is given a physical memory location where it can get the data from or where it can copy data to.

The hardware devices only work with physical memory addresses because the virtual memory model is implemented by the CPU and is thus only accessible there. The hardware, however, communicates directly with the memory with no memory translation in-between.

In order to enable the hardware device to process the contents of a memory region, the driver has to be able to make sure that this region is backed by physical memory and remains unchanged until the hardware device is finished with it.

In Sawmill Linux, the virtual memory regions (there called Dataspaces) and thus the contents of the virtual memory are managed by a so called Data Space Manager (DSM), the driver has to communicate with the responsible DSM to make sure that these pages are mapped and request that they remain mapped. This communication process is called *pinning*.

To be able to provide the hardware with physical addresses, the driver must be able to convert its virtual addresses to physical ones. As the DSM defines the mapping between virtual and physical memory, it is the entity to get this information from.

This functionality is required to support a scenario where the driver client drives the mapping of buffer data pages. The driver pins buffer memory when dequeuing it from the transmit queue but does not send it until they are mapped. Here, the driver has to request the pinning as its client is not allowed to.

4.7.3 Driver-side interface

A driver needs the following interface to the DSM subsystem. Not that the page descriptors describe pages in the client's address space as in general, the driver has no access to theses pages and therefore they are not in its address space.

```
void pin( client_t client, fpage_t page )
```

Pin the given page.

```
void unpin( client_t client, fpage_t page )
```

Unpin the given page.

```
bool isMapped( client_t client, fpage_t page )
```

Check whether the given page is backed by physical memory.

```
int mapAsync( client_t client, fpage_t page )
```

Back the given page with physical memory asynchronously.

```
int mapSync( client_t client, fpage_t page )
```

Back the given page with physical memory synchronously, i.e. this operation blocks.

```
void* virtual2physical( client_t client, void* virtual )
```

Convert the given virtual address into a physical address.

```
bool isOwner( client_t client, fpage_t page )
```

Check whether the client is allowed to use the given page for communication.

In case we operate on a shared memory region we need additional functionality to change the size of the shared memory region and have a method to convert memory references form the clients address space to the address which the data is mapped in the driver's address space.

```
page_t allocPage( client_t client )
```

Get a new page from the DSM subsystem.

```
void freePage( client_t client, page_t page )
```

Give a page back.

```
void* client2driver( void* virtual )
```

Convert the given virtual address of the client's address space into a virtual address of the driver's address space.

4.7.4 Client-side interface

If buffer data is in a shared memory region the client also needs functionality from the DSM subsystem to be able change the size of the shared memory region and to convert memory references from the driver's address space to its own virtual memory layout.

```
page_t allocPage( client_t client )
```

Get a new page from the DSM subsystem.

```
void freePage( client_t client, page_t page )
```

Give a page back.

```
void* driver2client( void* virtual )
```

Convert the given virtual address of the driver‘s address space into a virtual address of the client‘s address space.

4.7.5 Implementation issues

Some of these interface functions are used at the same rate as the enqueue or dequeue operations for transmitting buffers between the driver and the client. These are:

- `pin` and `unpin`.
- The `isMapped` function.
- The `isOwner` function.
- The translation functions between the virtual address spaces and the physical address space.

They would suffer from the same problem of high context switch rates if they would be implemented using IPC.

Fortunately, these functions require no synchronous or asynchronous operation. They can be implemented by reading or writing data structures that are exported by the DSM subsystem and thus need no IPC.

4.8 Protection

The introduction of interfaces to the driver weakens its protection. As we have stated in our requirements, we want a driver to be protected from malicious clients and a client to be protected from failures in the driver. The usage of an unprotected shared memory region is obviously a huge violation of the protection criterion. Thus we have to provide a rational on how to use that memory region while still being protected.

The access to the shared memory region proper is no problem as the DSM of this memory region is trusted by both the client and the driver. This trust relationship means that they are sure that the physical memory mapped to a certain virtual page contains the data that is supposed to be there, page faults are handled as fast as possible and the interface to the DSM works as specified.

The contents of the shared memory region, however, can be inconsistent due to incorrect operation of one of the two parties or due to explicit corruption. While both failures have different causes, they have the same result: the termination of the client-driver association. We have to make sure that inconsistencies in one shared memory region do no propagate to other shared memory regions or into the driver and client itself.

There are two major classes of failures in the operation between the driver and a client:

- Data failures.
- Progress failures.

Data failures are failures where data has a wrong value. Data failures result in an inconsistent state and thus in incorrect operation. Progress failures are failures where a thread is affected to make no progress anymore.

4.8.1 Data failures

The only connections between the driver and its client are the notification interface and the shared memory region. We already showed how the notification interface can be protected.

The driver and the client in their respective operation depend on various data in the shared memory region. All this data is subject to a problem that is known as „time of check vs. time of use attack“. That is, shared data can be changed and made invalid after it has been confirmed as valid. The countermeasure here is to copy data, check it for validity and work on that copied data then.

Data in the shared memory region can be separated in two classes:

- Pointers, and
- normal data.

A wrong pointer let’s the driver access a memory region which is either not mapped or which contains data that does not belong to the client. We can test the validity of pointers by checking whether they lie in the associated shared memory region. This is done with the DSM function `isOwner()`.

If that condition is not violated, they could point to invalid data. This no problem: if the invalid data contains other pointers, like it is the case with queues, we would likely detect an invalid pointer there. If we access important data there, we check it for validity.

The only normal data the parties access are:

- Allocation data structures, and
- buffer metadata

Corrupted data of the first class results in other corrupted data, which is caught later, corrupted data of the second class can be caught by checking the correctness of the buffer metadata.

4.8.2 Progress failures

The second major failure class is the one of progress failures, where a thread is caught in some kind of infinite loop. There are a number of means to prevent this. Loops are either counting loops or loops that depend on a boolean criterion to exit.

Counting loops can be protected by copying the termination criterion if it lives in shared memory (so that it can't be changed after we checked it for validity) and check it for validity then. We can ensure this way that the loop exits after a sane number of iterations.

Boolean loops that operate on the shared memory region are not that easy to protect. They usually follow data structures like linked lists that have no implicit limitation. No progress would be made for example when we follow a linked list that is connected in a loop while looking for the last element. Here, we could remember visited elements and thus detect when we visit an element twice. This, however, is obviously an expensive solution.

As the shared memory region has a finite size, all data structures inside have a finite size. We can assume that every loop has only a limited number of iterations that it can make while being in correct operation. Thus, we can add a counting loop to every boolean loop to make sure that it has an upper bound in the number of iterations. Choosing this upper bound is an easy task.

The only „infinite data structures" we have in the shared memory regions are slab lists and scatter-gather buffer lists. The number of linked slabs in a slab list can not exceed the number of pages of the whole shared memory region. Setting this number as an upper iteration bound is probably a bit to high but won't cost too much as every page has to be only visited once. The result of a violation of this bound is the termination of the association with the client. A malicious client could only let the driver do a few times more iterations that it would expect. Then the association would be immediately terminated.

For the scatter-gather buffer lists, we can chose an arbitrary number of buffer parts, one that is much higher than what would be expected. Thus, this rule won't be violated by correct operation, and an incorrect operation would only cause a few more loop iterations.

4.9 Summary

This chapter analyzed the hardware's requirements and the driver's tasks and proposed a flexible interface for communication between the driver and its clients. It detailed the need of the driver for support from the I/O infrastructure and summarized the required functionality into one interface. Its last section argued how protection can be ensured while using shared memory for communication.

Using the performance analysis of chapter three, we will now show how the presented communication interface can be used to be able to do high-performance communication in a multi-server environment with a processor utilization that is competitive with monolithic architectures.

5 Efficient inter-server communication

One goal of this thesis is to show how servers of a multi-server operating system can communicate at a high data rate without needing much more processor resources compared to a monolithic kernel. During the course of the work for this theses, we learned where processor cycles go during the operation of the driver.

During our experiments, we used one macro-benchmark to evaluate the effects of various changes on the system. We measured CPU utilization while trying to sustain the maximum transfer rate of the hardware device. Thereby we learned that context switches are the main consumer of processor cycles during operation. Here, the term context switch implies all the effects that come with it, like kernel entry/exit, kernel operations and cache and TLB reloads.

Context switches can be caused implicitly by the end of a thread's time slice, the thread preemption or explicitly by calling one of the microkernel's send or receive system calls. Each system call can be used with or without a timeout. When a receive call is used with a specific timeout and an invalid sender restriction, it can be used for delaying the calling thread. In summary, we have the following causes for context switches:

- Thread preemption,
- send IPC system call,
- receive IPC system call,
- receive IPC with timeout system call (delay).

Each of these context switches has its own associated costs in terms of processor cycles. Furthermore, these costs depend on the situation the context switch occurs. A thread that calls only „delay" in a loop does not cause many changes in the processor caches as its working set is very small. But an interrupt handler, that starts to run in the middle of the timeslice of another thread and then copies packets and calculates a checksum comes with high associated costs.

If we know which context switches are more expensive then others, we can try to replace the more expensive ones which cheaper ones. In our experiments we have learned that the costs decrease in the following order:

1. IRQ IPCs
2. send IPCs
3. delay IPC, then send IPC
4. delay IPCs

Thus, in general, we have to try to create as few context switches as possible. When this is done to a maximum extent, we can try to trade one type of context switch with another.

The definition of our device driver interface is flexible enough to allow various usage methods which save processor cycles compared to the naive usage. Most of the costs in terms of busy processor cycles are due to context switches. While some types of context switches are inevitable, others can be omitted. We can't get around the receive interrupts, but as we will see, many of the notification IPCs are superfluous.

We will now present four methods to save context switches and thus processor cycles. We differentiate them by the side of the communication path they are implemented, that is either the producer side or the consumer side.

5.1 Consumer IPC saving

5.1.1 Interrupt-Driven Transmit Queue Checking

Many of current devices notify the host processor with an interrupt that they are able to do new work. While they are busy, the driver can be sure that at one point of time in future, it will be notified to provide the device with new work. During that time span, the driver does not have to be notified when there is new work in the transmit queue. The device is busy then and the driver will be notified by the device in the future.

This fact can be exploited to save notifications for the transmit queue. If the driver client knew that the driver is „alive" in the sense that it will look in the queue for further packets, it would not have to notify him of the arrival of new packets. This state, however, can not be detected by the client as the driver is a black box to him. The only information communicated in direction to the client is the notification flag of the transmit queue. Thus, the driver itself has to detect its liveliness and set the flag accordingly.

A driver with IRQ notification can check the queue every time it received an IRQ. As it supplies the hardware with work, it knows how many interrupts will happen in the future. Such a device driver is alive as long as there is more than one outstanding interrupt. If there is only one interrupt left in the device, it may happen that it gets triggered, the interrupt handler starts running, empties the queue gets preempted and a client enqueues new work and finds that the driver wants no notification. The interrupt handler proceeds and requests a notification and goes to sleep. Now there is new work in the queue and the device won't be notified of that. If there are two or more interrupts outstanding, this situation can not occur.

In summary, if there is more than one outstanding interrupt, there is no need for device to request a notification for the transmit queue. The device hardware is

already busy, there's no point in supplying it with extra work and the queue will be checked in future anyway.

5.1.2 Periodic Queue Polling

The clients of a driver usually don't have an external activity like a hardware device with the IRQ which could enable them to look in the queue for new buffers. For them, we have to create an activity which allows them to look in the queue from time to time.

We use the system's timer facility to wake-up our consumer thread from time to time to look into the queue. The problem now is to chose the interval to visit the queue. If we make it too short, we gain no processor cycles compared to the naive usage, where we have one packet per IPC, if we chose a too long interval, we increase the latency of the path.

The length of the interval influences the average and maximum latency of the path directly. This is no surprise as average packet has to stay in the queue a maximum time of the interval length. The changed average and maximum latency has implications on other measures, too. The variance of the latency, the jitter, increases, too, as the packets remain a time in the queue which goes from zero to the polling interval length. When using TCP, the throughput is not affected for sane choices of polling interval lengths as TCP is very tolerant to increased round-trip latencies due to it's large acknowledgement window.

Periodic Queue Polling pays, when the costs of the periodic thread wake-up and look in the queue is less than the cost of an IPC per packet. This break-even point is approximately reached, when the polling rate is equal to the IPC rate.

Here it is also the consumer who uses its knowledge about the system to reduce the number of IPCs on the path.

5.2 Producer IPC saving

5.2.1 Deferred Notification

While some hardware devices have no transmit acknowledgement facility, we would like to save notification IPCs on the transmit path, too. We could use Periodic Queue Polling, but that implies putting the choice of the polling interval into the driver or to provide the driver client with an interface to set this interval. In both cases we would put policy about how the queue interface is used into the driver. This would be against our design requirement, to have the most general interface.

Our general goal is save processor cycles by reducing the number of notification IPCs. If a producer knew for each enqueued packet when the next packet is to be sent, it could decide whether to notify the driver immediately or whether to defer

that notification until the next packet. Device drivers will never be in this position as they can not know when they receive the next packet. Some driver clients, however, might have that information.

If a driver client does not have that information, he could speculate that in the near future a new packet will be available for sending and he could then defer the notification until then. He would set a timer to the maximum tolerable latency for the first packet and wait until then to notify the consumer. If subsequent packet arrive in that time frame, their notification IPCs would be saved. This, however, implies that we would frequently have to set up timers, which is a costly operation on L4 if done precisely.

5.2.2 Periodic Notification

Setting maximum latency timers for packets has about the same effect as Periodic Queue Polling. We only send notification IPCs from time to time, depending on the wanted maximum latency. Whereas interval queue polling was implemented in the consumer, we can have a similar mechanism in the producer. Instead of having the consumer poll the queue in equidistant points of time, we can let the producer send notification IPCs in intervals.

Compared to Periodic Queue Polling, this mechanism has the same influence on latency, jitter, and throughput. However, the break-even point is somewhat different as we have an additional IPC per polling interval.

5.3 Evaluation of the IPC saving techniques

The just introduced techniques reduce the CPU utilization by reducing the number of notification IPCs and their implied context switches. However, the periodic techniques come with new costs as they replace notification IPCs with delaying IPCs and introduce additional latency on the path where they are used.

We will now investigate whether the presented techniques really save context switches and CPU cycles and which trade-offs are implied by their usage.

5.3.1 Methodology

We ran our TCP benchmark for an implementation of each IPC saving techniques on both the PIC and the APIC machine. In each experiment, we measured TCP throughput, processor idle time and the number of context switches.

The changes were done only in one direction. While doing transmit benchmarks, the receive side transmitted one packet per IPC, while doing the receive benchmark, the transmit side used the Interrupt-Driven Transmit Queue Checking when not indicated otherwise.

5.3.2 Periodic Queue Polling (PIC machine) - Results and Discussion

Table 5.1 shows the result of the TCP transmit benchmark on the normal kernel for the “Interrupt-Driven Queue Transmit Checking” and for the “Periodic Queue

Setup	Idle cycles	Context switches	Throughput	CPU idle
Unit	** 10^6 [cycles] / 5 [s]*	** 1/5 [s]*	*[Mbit/s]*	*% CPU time*
Interrupt check.	122.00	97000	81.9	61.00
Periodic Queue Polling 3.9 ms	121.00	90000	82.0	60.50

Table 5.1 Transmit path, normal kernel

Table 5.2 shows the result of the experiment for normal notification (one notification for each enqueued packet) and for “Periodic Queue Polling” on the receive path.

Setup	Idle cycles	Context sw.	Throughp.	CPU idle
Unit	** 10^6 [cycles] / 5 [s]*	** 1/5 [s]*	*[Mbit/s]*	*% CPU time*
Always Not.	101.0	201000	62	50.50
Periodic Queue Polling 2.7 ms	119.0	175000	64.5	59.50

Table 5.2 Receive path, normal kernel

Both polling intervals were set to 2 ms in the PIC experiment. However, due to the inaccuracy of the timeouts in the standard kernel, the interval length rose to 2.7 ms and 3.9 ms, respectively.

As we can see by comparing the numbers in table 5.2, polling with the PIC gives a significant gain in idle processor cycles when used in the receive direction. The idle processor cycles rise from 50.5 % to 59.5 %. We already reduced the number of IPCs in the send direction by polling the transmit queue with the interrupt routine. Therefore, polling in the send direction do gives no gain (table 5.1). The idle processor cycles do not rise as the freed processor cycles by omitting IPCs are used for polling context switches..

5.3.3 Periodic Queue Polling (APIC machine) - Results and Discussion

We ran the same experiment on the precise kernel on the APIC machine. The usage of the APIC allows a fine tuning of the interval length of the polling interval. This allows a detailed investigation of the question at which interval

length the Periodic Queue Polling techniques pay and how much additional latency the path gets at this point. First, we measured the transmit performance with „Interrupt-Driven Transmit Queue Checking“ as a base number.

Setup	Idle cycles	Context switches	Throughp.	CPU idle
Unit	** 10^6 [cycles] / 5 [s]*	** 1/5 [s]*	*[Mbit/s]*	*% CPU time*
Interrupt check.	122.00	97000	81.9	61.00

Table 5.3 Interrupt-Driven Transmit Queue Checking

Then we implemented “Periodic Queue Polling” on the transmit path and measured the performance numbers for various interval lengths.

Polling Interval	Idle cycles	Context sw.	Throughp.	CPU idle
[ì s]	** 10^6 [cycles] / 5 [s]*	** 1/5 [s]*	*[Mbit/s]*	*% CPU time*
73	100.00	158000	81.8	50.00
104	108.00	128000	82.3	54.00
241	113.00	110500	81.8	56.50
1010	119.00	93000	82.4	59.50
2000	121.00	91100	81.9	60.50
4950	122.00	90000	81.9	61.00

Table 5.4 Periodic Queue Polling, transmit path

When we compare table 5.3 and 5.4, we observe that Periodic Queue Polling does not pay here as the IPC rate is already low due to the Interrupt-Driven Transmit Queue Checking technique. Note however, that not every device driver has the possibility of using Interrupt-Driven Transmit Queue Checking.

Afterwards, we did the equivalent experiments for the receive benchmark on the receive path. First we measured the performance numbers when a notification is sent for every enqueued packet.

Setup	Idle cycles	Context switches	CPU idle
Unit	** 10^6 [cycles] / 5 [s]*	** 1/5 [s]*	*% CPU time*
Always notify	100.0	200000	50.00%

Table 5.5 Base numbers, precise kernel, receive path

Then we implemented “Periodic Queue Polling” on the receive path and measured it for various interval lengths.

Polling Interval	Idle cycles	Context switches	Throughput	CPU idle
[ì s]	** 10^6 [cycles] / 5 [s]*	** 1/5 [s]*	*[Mbit/s]*	*% CPU time*
106	91.5	266000	62.5	45.75
205	105.0	225000	62.5	52.50
500	112.0	205000	64.5	56.00
965	114.0	197000	66.3	57.00
1800	114.0	194000	69.4	57.00

Table 5.6 Periodic Queue Polling, receive path

In the receive direction, polling starts to save processor cycles at an polling interval length of about 0.15 ms and above. This point is at the approximate IPC rate in the „always notify“ case: We have a about 38000 packets in 5 seconds which gives an between-IPC time of about 0.13 ms.

Polling trades processor cycles for latency: The maximum and average latency of a path increases with the polling interval. Jitter gets worse, too, as the packet is dequeued anytime in the polling interval.

We will now try to find an analytical description of the mechanisms. In each measurement interval T, we have N polling actions in time steps of t. Thus:

$$N = T/t \qquad (1)$$

When we do not poll, or poll very seldom, we can assume that our polling costs are nearly null. We then reach the upper bound of the idle CPU time, whose value we will call U. Now when we start polling more frequently, the costs for this polling go away from the idle CPU time. Assume every polling iteration costs C cycles. Then all our polling actions in one measurement interval T cost:

$$c = C * N \text{ [cycles]} \qquad (2)$$

If we subtract that from our upper bound U, we get the function of idle CPU cycles for a polling interval of t:

$$c(t) = U – C * T/t \text{ [cycles]} \qquad (3)$$

If we apply our analysis to measurement of the polling on the transmit path, we have an upper bound of around U = 122*10^6 cycles. T was 5 seconds, t gets the X axis, c(t) the Y axis. With a choice of C = 3400 [cycles], we get a good fit to the measured curve as we can see in the figure 5.2 which compares the numbers of table 5.4 with the results of our formula (3). With the given CPU, a difference of 5*10^6 [cycles] is equivalent to 2.5 % points in CPU utilization.

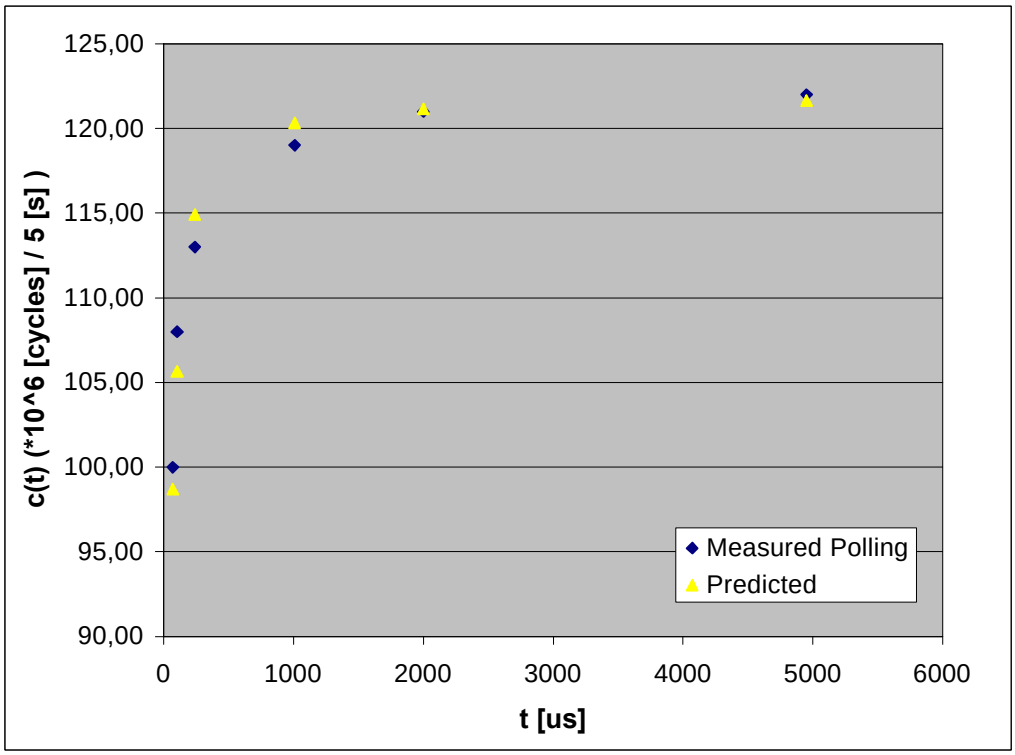

Figure 5.2 Measured vs. predicted polling cycles on the transmit path

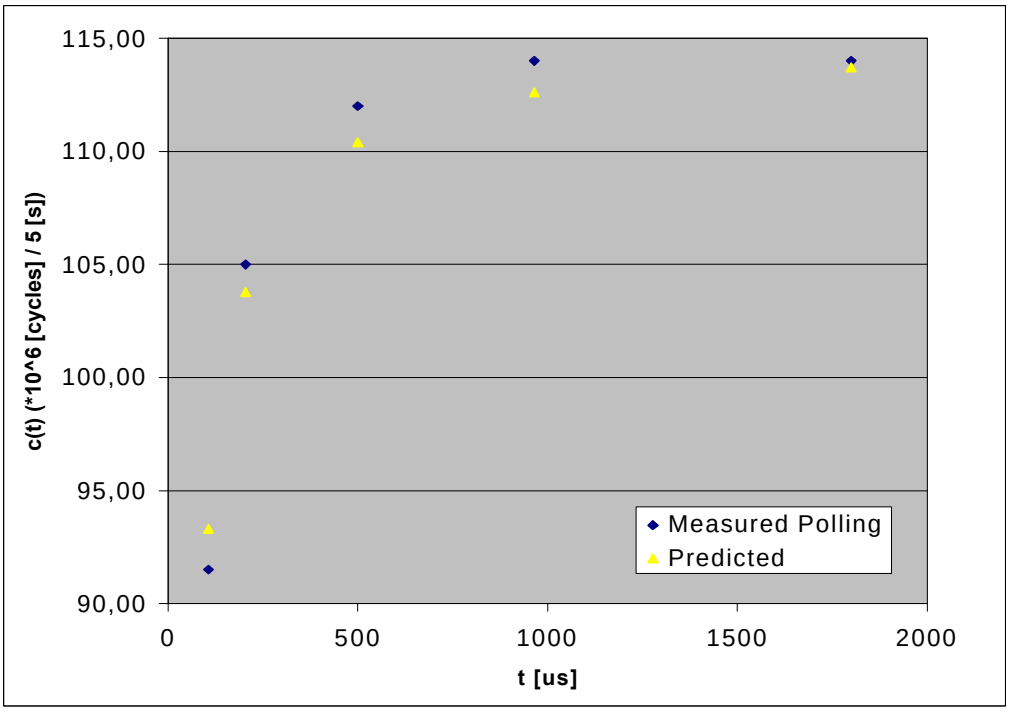

Figure 5.3 Measured vs. predicted polling cycles on the receive path

When applied to the receive path, we have an upper bound of about U = 115*10^6 [cycles]. If we chose C = 4600 [cycles], we get the graph in figure 5.3, which also includes the measured numbers of table 5.6 for reference.

In both cases, choosing intervals above 1000 ìs does not give significant gain in idle processor cycles. Even an interval 500 ìs does no cost much. This additional latency for example is not relevant for Ethernet networks used as with IP as even fast internet traffic has latency in the order of milliseconds.

5.3.4 APIC Periodic Notification – Results and Discussion

In our last experiment, we used Periodic Queue Polling on the transmit path for comparison reasons. Periodic Queue Polling is a consumer-side technique and has to be implemented therefore in the driver on the transmit path. Doing so in a real system would not be a good design decision as it moves the choice of the polling interval length (implying the path latency) and thus client policy into the driver.

However, with using Periodic Notification, we can move the choice of the polling interval length into the client. Periodic Notification uses twice as many IPC system calls as Periodic Queue Polling.

With measuring Periodic Notification on the transmit path for the transmit benchmark, we want to see how the additional IPC call affects the number of context switches and the idle processor cycles.

Polling Interval	Idle time cycles	Context sw.	Throughput	CPU idle
[ì s]	** 10^6 [cycles] / 5 [s]*	** 1/5 [s]*	*[Mbit/s]*	*% CPU time*
120	104	202000	82.0	52.00
224	111	151000		55.50
516	116	118000		58.00
1010	119	104000	81.9	59.50
2000	121	96000		60.50

Table 5.7 Periodic Notification

Setup	Idle cycles	Context sw.	Throughp.	CPU idle
Unit	** 10^6 [cycles] / 5 [s]*	** 1/5 [s]*	*[Mbit/s]*	*% CPU time*
Always notify	116.0	137000	81.9	58.00

Table 5.8 Base number, precise kernel, transmit path

When we compare table 5.7 and table 5.8, we see that Periodic Notification pays

setup.

Periodic Notification doubles the number of IPCs per notification when compared to the Interval Queue Polling with the same interval length. By comparing the numbers of table 5.4 and table 5.7, we see that the additional IPC has either no (above interval lengths of 1 s) or only low measurable costs.

5.3.5 Analysis of context switch costs

These numbers also give insights into the costs of the various context switch types. In figure 5.4, we see the number of context switches plotted against a number proportional to the number of idle CPU cycles.

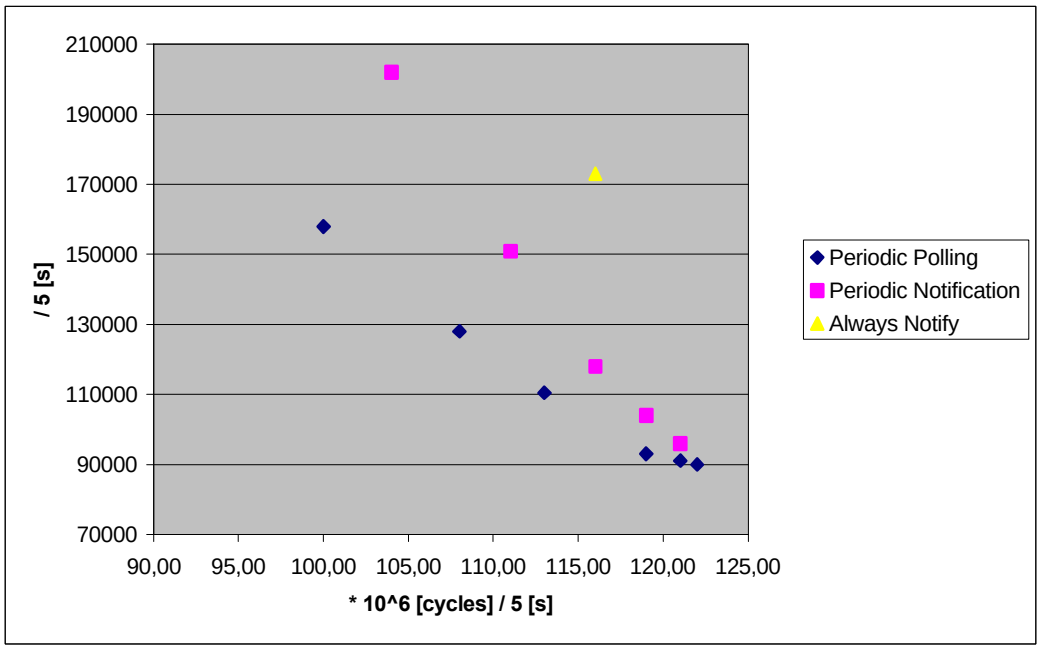

Figure 5.4 Context switches and their impact on idle CPU cycles

In terms of equivalent number of queue checking events, we have to compare the data points at the rate of 38000 packets in 5 seconds for „Always Notify“, which gives an between-IPC time of about 130 ìs. This is approximately the second data point of Periodic Queue Polling at 128000 context switches with 104 ìs and the first data point of „Periodic Notification“ at about 200000 context switches with 120 ìs.

At this rate, we see that, for the same rate of events, „Periodic Notification“ has about twice the number of context switches compared to „Periodic Queue Polling“. „Always Notify“ lies in the in the middle between both. This rate of events, however, does cost the most for „Always Notify“, about 3 % points in

CPU utilization more than „Periodic Notification“, which in turn costs 3 % more

As we have seen in a previous chapter, there is evidence that this difference of context switch costs stems from the implied cache refill costs of each context switch.

5.4 Imprecise software timers

Sawmill’s implementation of Linux software timers consists of a thread that waits for a timeout and a list of associated timers. If a timer event is added, which is before the event the thread currently waits for, the thread has to be restarted as there is now way it’s waiting time could be simply shortened. This restart routine calls `l4_myself` once and `ex_regs` twice, which results in three kernel entries.

We replaced the timer addition routine with one that adjusts the timeout time of newly added timeouts that would have to occur before the current earliest timeout to the timeout time of this current earliest event. Thus their timeout is deferred until the current earliest event occurs and the timer thread does not have to be restarted. Added timeouts whose timeout time is after this current earliest event are not influenced.

However, when the current earliest event is too far in the future, we restart the time thread, otherwise the timer system would be out of order for small time values. This change helps most at a high timer addition rate of small timer values. Each of them gets a small delay but the timer thread won’t have to be restarted.

This change does not influence the behavior of the TCP/IP stack. Neither the throughput nor the packet rate changed.

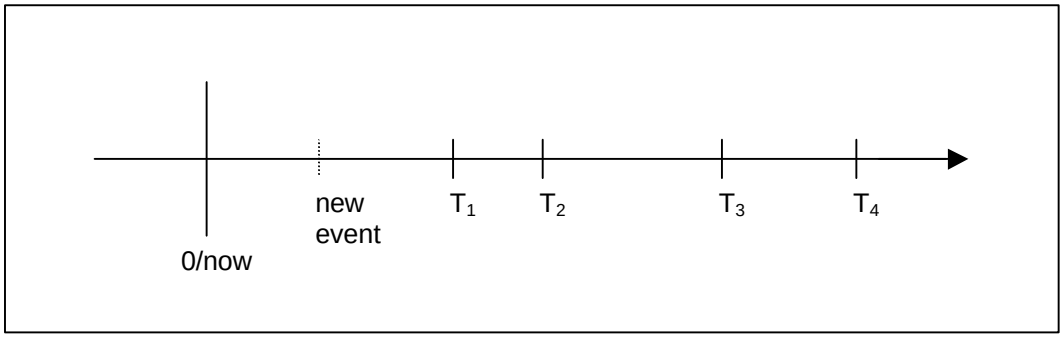

Figure 5.1 Insertion of a new timer requiring restart of the timer thread

5.5 Evaluation of imprecise software timers

5.5.1 Methodology

We replaced the Sawmill timer implementation with the imprecise one. We measured the number of context switches for two interface configurations (with and without interval polling) for both transmit and receive with the standard Sawmill timer implementation and the imprecise timer implementation.

5.5.2 Results

Benchmark	*Unit*	Tx poll.	Tx w/o poll	Rx poll.	Rx w/o poll.
Std. packet rate	*/ 5 [s]*	57100	57800	43420	42600
Std. context sw. Rate	*/ 5 [s]*	58000	117000	128000	242000
CS / packet	-	1.02	2.02	2.94	5.68
Imp. Packet rate	*/ 5 [s]*	56000	57500	48000	42000
Imp context sw. Rate	*/ 5 [s]*	53000	99000	115000	205000
CS / packet	-	0.94	1.72	2.40	4.88
Ratio CS/p.		*93%*	*85%*	*81%*	*86%*

Table 5.9 Influence of the imprecise timer on context switches per packet

The changes had no impact on the workings of the TCP/IP stack.

Benchmark	Idle CPU imp. Timer	Idle CPU normal timer
Tx Interval Polling	63.30 %	60.80 %
Tx Normal	48 %	43.33 %
Rx Interval Polling	50.67 %	47.53 %
Rx Normal	34.33 %	29.33 %

Table 5.10 Influence of the imprecise timer on idle CPU time

5.5.3 Discussion

We see that an imprecise timer implementation saves about 7% to 19% percent of context switches per packet. This results in a saving of 3% points to 5% points of CPU cycles.

5.6 Achievable performance

We will now combine „Periodic Queue Polling“ with the imprecise timer implementation to achieve the maximum performance.

5.6.1 Results

	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 %

Table 5.11 Achievable performance

5.6.2 Discussion

This is the maximum percentage of idle CPU cycles which are gainable with the presented methods. Most of the context switches for communicating packets between the servers are saved. The worse difference of performance for the receive benchmark is a direct result of the high IRQ costs on the receive path and the higher IRQ rate with the receive benchmark.

5.7 Summary

This chapter presented four implementation techniques that allow efficient inter-server communication at high rates with a processor utilization which is competitive with monolithic architectures.

The techniques are based on the result of chapter three, which identified context switches as the main source of costs in inter-server communication and the design of chapter four, which presented a communication interface that allows to decouple transfer of data from transfer of control in a flexible way.

We have shown that the techniques really save context switches and processor cycles. For two techniques that use a periodic checking of the queues, we have identified a trade-off between idle processor cycles and path latency. The resulting additional path latencies, however, are not longer than a few milliseconds, which is no problem for LAN and WAN traffic.

6 Conclusion

During the course of this thesis, we have shown how to design the communication interface between a network interface driver and its clients. The communication interface has been split into a pure synchronization-free data communication and a pure policy-free control part. The flexible design of the control part enables the implementation to have the most freedom for choosing implementation techniques that fit to its requirements.

The usage of DMA (direct memory access) techniques in the hardware device requires the driver to be able to know and to fix the physical representation of data in virtual address spaces of clients. As the information and the control of this mapping lies not within the driver, it needs support from external entities to perform these operations. These requirements have been distilled into the design of the interface between the network interface driver and the supporting operating system infrastructure.

We have measured and analyzed the costs during high-speed communication in terms of consumed processor cycles and were able to attribute these costs to the transfer of control during communication. We have identified a correlation between context switches and idle processor cycles and measured an order of the costs for the different types of context switches. We gave evidence that the costs for context switches could be caused by L2 cache footprint changes.

Supported by these measurement results, we have developed techniques to enhance the performance of the running system. These techniques free processor cycles by saving context switches during communication. They either use the device interrupt to check the queues or they use timers to be able to check the queues in well-defined intervals. Both techniques save notification context switch costs but are a source of context switches by themselves. In sum though, there are less context switches and thus more idle processor cycles. The usage of the timer techniques increases the path latency and jitter and therefore introduces a trade-off between path latency and idle processor cycles.

This work has shown that multi-server operating system designs are capable of achieving a comparable I/O performance relative to monolithic design without needing to give up their advantages of protection and modularity. There is no need to trade protection against performance. However, we have identified a trade-off between the path latency and jitter and the processor utilization.

The prototypical implementation of the presented design achieves a processor utilization which is only 9 % points (TCP transmit) and 11 % points (TCP receive) below the comparable Linux implementation.

6.1 Future Work

The limited amount of time for this thesis did not allow to investigate the cause for the context switch costs in detail. If a deeper analysis would clearly identify a distinct source for them, one could develop techniques to lower the costs of context switches instead of just avoiding them.

We have focused on the design and implementation of an Ethernet device driver, the TCP/IP stack of Linux and the uIP stack. The application to different device classes could probably bring up new issues.

This thesis did not describe how the supporting infrastructure is structured and implemented. We have defined only an interface to this infrastructure, but their design and implementation especially in multiple servers will be a huge task. This will discover many further issues that probably have impact on the driver's interface design.

7 References

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

[Damianakis98] Stefanos Nektarios Damianakis. *Efficient Connection-Oriented Communication on High-Performance Networks*. Ph.D. thesis. Princeton 1998.

[Druschel93] Peter Druschel and Larry L. Peterson. *Fbufs: A High-Bandwidth Cross-Domain Transfer Facility*. In Proceedings of the Symposium on Operating Systems Principles (SOSP) 1993.

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

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

[Hinds] Nigel Hinds, Alain Gefflaut, Trent Jaeger, Jochen Liedtke, Andreas Häberlen, Stefan Götz. *Performance Analysis of a Multiserver Network System*. Internal Report.

[Intel] *21143 PCI/CardBus 10/100Mb/s Ethernet LAN Controller.* Intel Corporation.

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

[McKusick96] M.K. McKusick. K. Bostic. M.J. Karels. J.S. Quarterman. *The Design and Implementation of th 4.4BSD Operating System.* Addison-Wesley Publishing Company. 1996.

[Michael96] Maged M. Michael, Michael L. Scott. *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.* In Proceedings of the Symposium on Principles of Distributed Computing (PODC) 1996.

[Pai99] V. Pai. P. Druschel. W. Zwaenepoel. *IO-Lite: A Unified I/O Buffering and Caching System*. Usenix Symposium on Operating System Design and Implementation (OSDI) 1999

[Sugerman01] Jeremy Sugerman, Ganesh Venkitachalam and Beng-Hong Lim. *Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor*, USENIX 2001.

[Thekkath93] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moyy, and Edward D. Lazowska. *Implementing Network Protocols at User Level*. IEEE/ACM Transactions on Networking. Vol. 1, Nr. 5, 1993.

[Tsigas00] P. Tsigas, Yi Zhang. *A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems*. Technical Report TR-2000-01. Department of Computer Science, Chalmers University of Technology. 2000.

[UDI] *The Uniform Driver Interface Specification Version 1.0.* http://www.projectudi.org/

[Unrau97] Ronald C. Unrau, Orran Krieger. *Efficient Sleep/Wake-up Protocols for User-Level IPC*. Technical Report TR97-08, Dept. of Computer Science, University of Alberta, Edmonton, Candada. 1997.

[Unrau98] Ronald C. Unrau, Orran Krieger. *Efficient Sleep/Wake-up Protocols for User-Level IPC*. In Proceedings of the International Conference on Parallel Processing (ICPP) 1998.

[Welsh96] Matt Welsh, Anindya Basu, and Thorsten von Eicken. *Low-Latency Communication over Fast Ethernet*. Proceedings of the Euro-Par 1996.

[Welsh97] Matt Welsh, Anindya Basu, Thorsten von Eicken. *Incorporating Memory Management into User-Level Network Interfaces*. In Proceedings of Hot Interconnects V 1997.