Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

WAMOS 2014

First Wiesbaden Workshop on Advanced Microkernel Operating Systems

Editor / Program Chair: Robert Kaiser

RheinMain University of Applied Sciences
Information Science
Unter den Eichen 5
65195 Wiesbaden
Germany

Technical Report Februrary 2014

Workshop Program

Date:	Thursday, 13 February 2014
Location:	UdE C Building, C405
09:30	Beginning
09:30	Invited Talk:
	WINGERT – A Thread Migrating OS for Real-Time Applications Alexander Züpke
10:00	Coffee Break
	Session 1: I/O Support Session Chair: Thomas Frase
10:15	Supporting USB in a Microkernel Framework Alexander Aring and Timon Link
10:45	A design proposal for a shareable USB server in a microkernel environment Daniel Ernst and Matthias H. F. Jurisch
11:15	Coffee Break
	Session 2: IPC Paradigms Session Chair: Richard Petri
11:30	Feasibility to replace Interprocess Communication by the Message Passing Interface in microkernel contexts René Drolshagen and Lasse Löffler
12:00	Interprocess Communication (IPC) in comparison to the Message Passing Interface (MPI) in a microkernel context Manuel Hermenau and Janos Zweifel
12:30	Lunch Break
	Session 3: Scheduling Session Chair: Matthias H. F. Jurisch
13:30	User-Level CPU Inheritance Scheduling Marcel Kneib and Jonas Reininger
14:00	User-level scheduling mechanisms Andreas Zoor and Nikolai Nagibin
14:30	Coffee Break
	Session 4: Performance and Isolation Session Chair: René Drolshagen
14:45	The long way towards usable IPC Performance in Microkernels Richard Petri
15:15	Side channel attacks in a microkernel environment Thomas Frase and Fabian Seiberling
15:45	Closing

Table of Contents

Foreword

The First Wiesbaden Workshop on Advanced Microkernel Operating Systems was conceived to provide a forum for students of the advanced operating systems course at Wiesbaden University of Applied Sciences to present the results of their work.
Besides submitting papers themselves, students also served as members of the program comittee and were involved in the peer-reviewiewing process. The intention, besides the presentation of interesing operating system papers, was to provide hands-on experience in organizing and running a workshop.
The authors were given the opportunity to revise and re-submit final versions of their papers based on the reviews. The papers contained herein are the final versions submitted just before the workshop.
I'd like to thank all participants for their enthusiasm.
I'd like also to thank our guest speaker Alex Züpke who provided an interesting insight into novel concepts of microkernel design.

Robert Kaiser

Program Chair

Program Committee

Date: 2014-02-09

Author:
Alexander Züpke
Hochschule RheinMain University of Applied Sciences
Campus Unter den Eichen 5
65195 Wiesbaden
Germany

alexander.zuepke@hs-rm.de

Title:
WINGERT -- A Thread Migrating OS for Real-Time Applications

Abstract:

The talk presents the WINGERT research operating system which aims to explore novel OS design patterns for real-time and mixed-criticality systems. Strongly influenced by existing micro kernel designs like L4, we think that a kernel providing a minimal set of abstractions and fast context switches is a key element to achieve high performance and deterministic system behavior.

We present the WINGERT architecture and discuss the use cases of "thread migration" for cross-address space RPCs (remote procedure calls) in detail to build hierarchical systems on top of this design concept.

Another strong research focus is on resource sharing in such systems: WINGERT provides Futexes (fast user space mutexes) for resource sharing between trusting parties. Untrusting parties utilize thread migration RPC to trusted servers instead. We discuss the benefits of the locking architecture in detail.

Supporting USB in a Microkernel Framework

[Extended Abstract]

Alexander Aring*
Hochschule Rhein-Main
University of Applied Sciences
Kurt-Schumacher-Ring 18
Wiesbaden, Germany
alex.aring@gmail.com

Timon Link†
Hochschule Rhein-Main
University of Applied Sciences
Kurt-Schumacher-Ring 18
Wiesbaden, Germany
timon.link@gmail.com

ABSTRACT

This paper provides an introduction in supporting USB in a Microkernel-Framework. In order to understand the main thoughts, a quick overview about the USB-specification is provided. Microkernels have a small trusted code-base, so the major part of the USB-Framework has to been implemented in userspace which is the USB-drivers themselves and the UHCI. The several USB-drivers has to cooperate with each other or the UHCI by exchanging messages. The task of sending and receiving messages is handled via IPC. This leads to a bottleneck by communicating to the kernel, because each IPC is realized as a system call. In case of *HelenOS*, a microkernel operating system we choose to use as example, a solution is provided by reducing the communication overhead.

Keywords

USB-Stack, USB, USB-Framework, Microkernel, HelenOS

1. INTRODUCTION

20 years of microkernel and 19 years of universal serial bus. It's time to marry these two technologies together. This paper describes an example implementation of an USB framework inside microkernels.

2. PURPOSE

This paper describes an USB framework for microkernel systems. Additional the USB framework will keep the microkernel philosophy.

2.1 Idea

We use the idea of microkernel and build an USB framework on top. An USB microkernel framework has several service applications. These services will handle the USB framework and abstract a generic USB interface to interact with applications which use the framework.

2.2 Problem

The USB framework sevices need a complex mechanism to handle the necessary USB functionality. This paper describes the necessary complex mechanism to solve this problem.

3. USB SPECIFICATION

USB stands for Universal Serial Bus and is developed by Intel Corporation. Today (in year 2014) this bus system is the most common used peripheral bus system for desktop pcs. The term "peripheral bus" describes a bus system to provide a wide support for a wide variety of devices. These devices can be a keyboard, mouse, etc. USB supports hotplugging. This means an USB device can be added or removed during runtime. In this paper we use the USB specification 2.0 [Cor00] to get a basic information about USB.

There are several kinds of USB port connectors to connect an USB device into USB. This connectors provide the same physically background inside the USB cable. The only difference is the form factor of an USB connector. USB uses a differential voltage for signaling. The USB Cable on USB 2.0 and backwards has four wires.

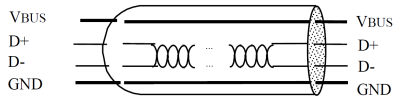


Figure 1: Wires of an USB cable. [Cor00]

Figure 1 shows the USB cable wires. The data wires are twisted to reduce signaling failures from electromagnetic fields. The other two wires are there to provide a five volt power supply and ground. Newer USB 3.x specifications have more than one twisted data wire to provide a higher bandwidth. Each USB specification describes how an USB cable needs to be build to provide the USB specification conditions.

The first USB 1.0 specification was released in November 1995. Since then Intel released USB 1.1, 2.0, 3.0 and 3.1

*B.Sc. in Computer Science
†B.Sc. in Computer Science

specifications. Each of them is backwards compatible to the previous USB specification.

3.1 USB Device

An USB device is usually a peripheral device like a keyboard, mouse, printer, video, storage, etc. This device offers an USB connector and the internal firmware of the device follows an USB specification.

Figure 2: Certification logo of USB 3.0 Standard. [Wik14c]

Figure 2 shows an example of the USB 3.0 certification logo. Only USB 3.0 certification devices can use this logo.

3.1.1 Classes

The USB specification describes a wide area of USB device classes. In this paper we handle with the following USB classes: HID *(human interface device)*, UVC *(USB video device class)* and MSC *(mass storage)*.

Each USB class has an unique identifier in the USB specification. If an USB Device doesn't fit in any USB class like a fancy USB lamp, there also exists a vendor specific USB class. An USB class provides a generic interface to interoperate with a generic USB device class driver. For example a BIOS does provide a generic HID driver to interoperate with a connected USB keyboard. The generic HID interface is specified by the used USB specification.

3.1.2 Descriptors

An USB device knows what functionality it offers. Besides the functionality it knows the consume of his power supply. These information stands in the descriptor table of an USB device. The firmware contains the descriptor table and provides this information. It's necessary that an USB device provides a descriptor table. The descriptor table contains the following kinds of descriptors:

Device Descriptor Contains general information about the connected USB device like the device class. Every USB device has only one device descriptor.

Configuration Descriptor Contains information about the configuration of the connected USB device. There also exist a way to configure an USB device at runtime if necessary.

Interface Descriptor Describes a specific interface with a configuration. An interface provides zero or more endpoint descriptors.

Endpoint Descriptor An endpoint is a communication channel to send or receive data to or from an USB device. Each endpoint has a number to identify the endpoint.

The device descriptor contains two important information about the USB device.

VID Stands for Vendor ID. A company which will produce an USB device needs a registered ID. The VIDs are managed by Intel.

PID Stands for Product ID. Each different USB device which is produced by a specific company needs a Product ID.

A combination of VID and PID provides an unique identifier for any USB device.

3.2 Endpoint Transfer-Types

All transfer messages on USB has little endian encoding. An endpoint descriptor has a transfer type to describe the kind of message which is provided at the specific endpoint. The USB specification has four kinds of transfer types, which are:

Control Control transfers are used to get or set configurations, commands or statuses of an USB device. This type of transfer is used in all kind of USB devices.

Bulk A bulk transfer provides a large data transfer support. This is usually used in mass storage devices which require a high bandwidth. Common used in MSC class devices.

Isochronous Like a bulk transfer type but provides a guarantee of required bandwidth. Common used in UVC class devices.

Interrupt This kind of interrupt isn't an interrupt in an usually case. An interrupt transfer type provides a periodically request of small data. Common used in HID class devices.

There exists a special endpoint with the number 0. An USB device must have this endpoint. This endpoint offers a control transfer type. Over this endpoint the descriptor table is accessible to enumerate devices on the USB system.

4. USB HOST CONTROLLER

An USB host controller is an electrical hardware chip. Each pc which supports USB has an USB host controller inside the northbridge on a desktop pc. This chip handles some USB operations on hardware. These operations are some time critical operations which cannot be handled by software. The operations and interfaces are specified by a host controller interface standard. There are four different host controller interface standards. These are OHCI, UHCI, EHCI and XHCI. The USB host controller doesn't implement the full USB specification. On top of an USB host controller driver must run an USB implementation. This implementation is usually known as USB stack.

Figure 3 shows the architecture model of an USB host controller, USB stack (described as *USB System SW*) and an USB driver (described as *Client SW*). These three layers are usually involved to run an USB system.

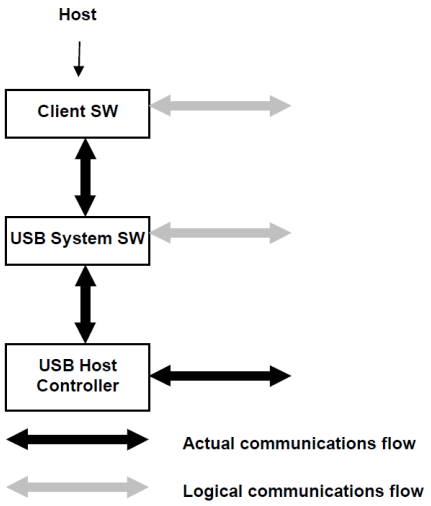


Figure 3: Host Composition. [Cor00]

5. MICROKERNEL

A microkernel *(µ-kernel)* aims the goal, as against the monolithic kernels, to store the minimum amount of functionality in the privileged mode, which are needed to run an operating system. The functionality is reduced to *inter process communication (IPC)*, *thread management* and *memory management*. As the monolithic kernels also implement the device drivers, filesystems and other components, microkernels only implements these software as userspace programs. This reduces the *Lines Of Code (LOC)* up to 10.000 [KE13, sec. 2] in the seL4 kernel. The advantage of the small amount of *LOC* is the reduced error-proneness. This prevents the microkernel to crash as often as a monolithic kernel, because most of the functionality is implemented in userspace. When an userspace program crashes, e.g. a device driver, it will be simply restarted without taking effect on the kernel runtime. Figure 4 shows the principles of a monolithic kernel vs. microkernels.

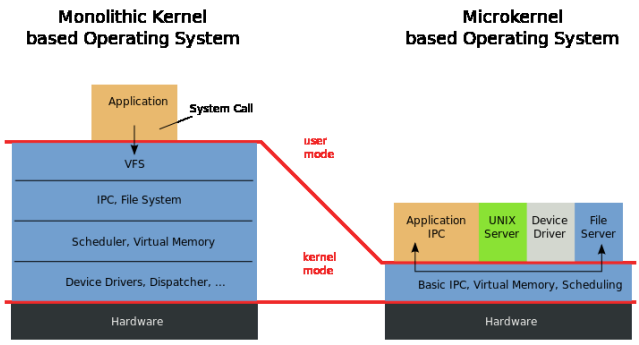


Figure 4: Structure of monolithic kernel vs. microkernel [Wik14b]

In both approaches an userspace program invokes a *system call* to get access to a needed functionality from the kernel. In case of *Unix*, the amount of system calls is 40 [hl14], whereas the L4 gets along with only 10 system calls [hcl14].

The main concept goal of microkernels is to store the most functionality in userspace. This leads to the concept of *client-server-communication*. Both, the server and client, are running in userspace. A server can be imagined as a daemon process, which offers a special service, e.g. a filesystem service, to consumers which are the clients. The server handles the communication with the microkernel by using system calls. The communication between server and client is handled by *IPC*, which will be described in detail in chapter 5.3 *Inter Process Communication*.

5.1 Related Microkernels

This section gives a short overview of existing microkernels with the main concepts and ideas in their chronological order.

Mach This Microkernel was developed by Carnegie Mellon University at 1985 [CMU14]. It's often mentioned as the pioneer of microkernels and so it's seen as the first generation. Mach abstracts the *pipe* of Unix by using IPC. The developer introduced the concepts of *tasks* which are sets of system resources, *threads* - a single unit of execution, *ports* as a message queue for IPC between tasks and *messages* which are a collection of data objects to send to the different ports. But IPC is also the bottleneck of Mach. It slows down the kernel up to 50% compared to a native Unix implementation.

Minix 3 Minix 3 [Min14] was developed by Andrew S. Tanenbaum and released in 2005. The motivation behind this microkernel is security, flexibility and high reliability. It's the second generation of microkernels.

L4 This Kernel was developed by Jochen Liedtke as replacement of L3. The main idea is a well designed, reduced IPC layer to eliminate the bottleneck of IPC known in Mach. Liedkte has written the source code in pure i386 assembly language. With this implementation Liedtke revolutionised the family of microkernels. Many research centres and universities began to implement the L4 Kernel in high level programming languages. Popular results are the *L4Ka::Hazelnut*, *L4ka::Pistachio* (both developed at the University of Karlsruhe), *L4/Fiasco* from Dresden University of technology, *OKL4* and *seL4* both impelled by *Gernot Heiser*.

5.2 Processes

In the context of operating systems (*OS*) there exist two kinds of processes. The processes in kernelspace and processes in userspace. Userspace processes don't have the permission to interact directly with the hardware of the underlaying platform. As mentioned before in section 5, microkernels only implement the minimum amount of functionality as processes in kernelspace, which are needed to run an operating system. But there are a lot of services of an OS, which need to interact with the underlaying hardware, e.g. a filesystem or a systemtimer. These services don't run in the privileged kernelspace, which would hurt the principles of a microkernel. So they have to be implemented in userspace. This causes us to take a closer look to the userspace processes.

Userspace processes are typically split into two parts, server and clients. Servers offers a special functionality, by interacting with the kernel via system calls, to the clients. The communication between servers and clients is handled by a *RPC-type* Inter Process Communication *IPC*, which is described in detail in section 5.3 Inter Process Communication.

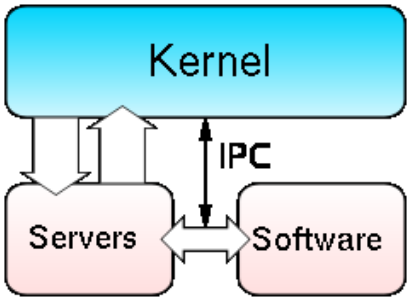


Figure 5: Principles of client-server [Wik14a]

Figure 5 shows the concept of client and server communication. In this case the client is simply called *software*. In fact there is no technical difference between servers and clients, so servers are software too.

5.3 Inter Process Communication

Inter Process Communication *IPC* allows separate Processes to communicate via messages. This concept is the premise for the client-server model described in section 5.2. Without IPC it would not be possible to strictly divorce the clients from servers and the most hardware near software like device drivers had to been implemented in privileged kernelspace.

There are two concepts of IPC - synchronous and asynchronous.

synchronous Synchronous IPC forces the sender and receiver to block the execution of the program and wait for the other side to perform the IPC. When the IPC was performed the execution can be continued. There is only one decision for the programmer, either to implement an *infinity-timeout* or a *zero-timeout* for the waiting, if a deadlock occurred and the IPC can't be performed.

asynchronous Asynchronous IPC is analogous to known concepts of network communication. The sender sends a message and continues executing. The receiver waits for the message by polling the sender. But this forces the kernel to maintain buffers for messages. This is the bottleneck of older microkernels which Liedkte has discovered. So he chooses to ban asynchronous IPC from his L4 implementation.

In most microkernels is only support for synchronous IPC integrated, to prevent the performance issues. But this is going to cost the comfort of the programmer, because he has to handle the rendezvous of both sides manually.

6. MICROKERNEL USB-FRAMEWORK

In this section we give a detailed overview about the required features to support *USB* in microkernel-based operating systems. The main challenge is the design of an USB framework inside the microkernel-architecture. This chapter discusses a working USB framework based on the example of *HelenOS* [hel14].

6.1 Related Microkernel USB-Stacks

In the following section we will show the example of an existing operating systems, based on a microkernel, which supports *USB*.

6.1.1 HelenOS

HelenOS aims the goal to be a very portable OS. At the moment HelenOS only supports USB 1.1 with keyboards and mice, but it worth to take a closer look to the principles of implementation. The *USB subsystem* [hel14] in HelenOS consists of these thoughts:

- drivers for host controllers
- drivers for USB devices
- mechanism to start device drivers when hotplugging occurred
- allow client programs to use the plugged in devices
- drivers have to communicate with each other

HelenOS offers a great feature to achieve the last three goals, the HelenOS *Device Driver Framework DDF* [hel14]. This generic framework is used by developers to implement specific device drivers. In fact that the framework supports the features

- start drivers automatically,
- driver to driver communication and
- offer a layer for exposing device interfaces to client programs,

it's a good idea to use it as backend for building USB device drivers. This offers a great advantage for implementing USB device drivers, because each driver is a standalone task that communicates with other tasks (drivers or clients) through IPC. The meaning is, that complicated drivers could be built modulary by splitting the driver in multiple tasks which cooperate with each other. So the effort of maintaining the driver is reduced and also the number of bugs of each partition.

The cooperation of multiple drivers is used in HelenOS for the implementation of the *UHCI (host controller driver)*. The developers decided to split this driver in two tasks. One driver for the UHCI itself and a second driver as *OHCI (hub controller) driver* and for the root hub. The *UHCI* and *OHCI* are strictly coupled. That means, if one driver is not started or has a failure, the other doesn't work at all.

Another meaningful usage of the coorperation are *multiple interface devices (MID)*. These (USB) devices support several interfaces to access the functionality of the device. A good example for a MID are digital cameras. One way to access the photographs is through a specific vendor interface, which needs a special driver. As fallback solution, when the vendor interface crashes or is simply not supported, digital cameras can be accessed via a mass storage driver. These two capabilities would be implemented in two drivers, which cooperate via DFF to support one MID.

The next step to understand the USB subsystem in HelenOS is to take a closer look to the USB devicetree.

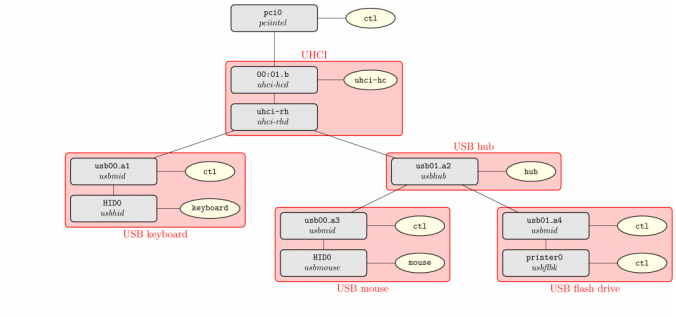


Figure 6: Example of devicetree in HelenOS [hel14]

Figure 6 shows an example of a possible scenario, when an USB hub, an USB keyboard, an USB mouse and a printer are connected to the host controller *UHCI*. In order to explain the concepts of the devicetree it's only necessary to describe some parts of it, because all connected devices work in a similar way. The first component to look at ist the host controller. The UHCI offers a function called *uhci-hc* which is invoked by all connected drivers to register the device. The host controller then connects all registered devices directly to the PCI-bus.

The leaf *USB keyboard* represents a connected keyboard device, which is obviously a MID. As mentioned before MID drivers are split into tasks - two in this case. When the device is plugged in a generic *Human Interface Driver (HID)* starts up. The main task of this driver is to capture key events and send them to connected clients. Clients in this context are processes which consumes the key-inputs from the keyboard driver.

The second driver is the MID-driver. It handles the communication between the device and his parent- and children-leafs by offering a special function *ctl*. Below the communication inside the devicetree is described in more detail.

The concept of the communication inside the tree is leaning on *parent-only* communication, which means that drivers can only communicate with their direct parents. The advantage of the parent-only communication is the exact adaption of the physical topology to the virtual device tree. But a big disadvantage is the performance. Let's examine a scenario, when the mouse is plugged in to the above devicetree (see Figure 6). The following Figure 7 shows the principles of the communication in this scenario.

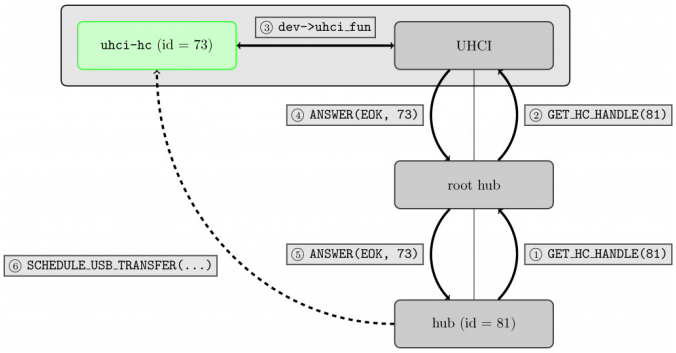


Figure 7: Communication scenario [hel14]

At first the mouse is plugged in and the MID driver starts. At this point the driver tries to register at the UHCI, but it only has the possibility to contact it's direct parent, which is in this case the *root hub*. So the MID driver asks the root hub for an *UHCI-handle*. But the root hub isn't the UHCI and has to forward the request to his direct parent. This scenario repeats until the UHCI is reached. The UHCI then answers the request and sends his answer over all leafs to the MID driver of the plugged in mouse. Obviously an *USB transfer* within the same concept is an enormous overhead and would slow down the transfer itself. But since the MID driver and the UHCI has exchanged their ids, the transfer is handled by a function *SCHEDULE_USB_TRANSFER*, offered by UHCI, directly called by the MID.

7. ADDITIONAL AUTHORS

8. REFERENCES

[CMU14] CMU. Overview of the mach project, Janurary 2014.

[Cor00] Intel Corporation. *Universal Serial Bus Revision 2.0 specification*. Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips, 2.0 edition, April 2000.

[hcl14] http://www.cse.unsw.edu.au/ cs9242/07/lectures/02 l4x6.pdf. L4 programming introduction, Janurary 2014.

[hel14] helenOS. Usb subsystem in helenos, Janurary 2014.

[hl14] http://www.di.uevora.pt/ lmr/syscalls.html. Unix system calls, Janurary 2014.

[KE13] Gernot Heiser Kevin Elphinstone. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? *NICTA, UNSW*, April 2013.

[Min14] Minix. Minix 3, Janurary 2014.

[Wik14a] Wikipedia. Kernel (computing) — Wikipedia, the free encyclopedia, 2014. [Online; accessed 20-Janurary-2014].

[Wik14b] Wikipedia. Microkernel — Wikipedia, the free encyclopedia, 2014. [Online; accessed 19-Janurary-2014].

[Wik14c] Wikipedia. Universal serial bus — Wikipedia, the free encyclopedia, 2014. [Online; accessed 23-Janurary-2014].

A design proposal for a shareable USB server in a microkernel environment

Daniel Ernst
Hochschule RheinMain
Fachbereich Design Informatik Medien
Unter den Eichen 5
D-65195 Wiesbaden
daniel.ernst01@gmail.com

Matthias H. F. Jurisch
Hochschule RheinMain
Fachbereich Design Informatik Medien
Unter den Eichen 5
D-65195 Wiesbaden
matthias.jurisch@gmail.com

ABSTRACT

The Universal Serial Bus has gained importance on the desktop market in the last years. Nowadays, even systems like the Raspberry Pi use them for tethering different peripherals. When building a shared USB server in a microkernel environment problems like controlling access-restrictions and scheduling bus access are important. It is also unclear, how such a system should encapsulate the USB access.

In this paper we will compare different approaches for solving these problems. Due to the universal purpose of this bus-system, we will illustrate an application- scenario which will address both real-time- and a multimedia-applications simultaneously. We will focus on the main-problems in such an environment. The first will be how the access to the bus can be restricted and which patterns are suitable for our application-scenario. The second will be how the bus can schedule the communication while assuring the real-time-ability. Different algorithms will be discussed. Finally, different design ideas for the server interface will be compared while targeting to retain existing implementations. From this discussion we will propose a design for a USB driver server and discuss its strengths and weaknesses.

Keywords

Microkernel, Universal Serial Bus, Shared Resource, Real-time, libUSB, Scheduling, Access Rights Management

1. INTRODUCTION

Microkernels in general allow a secure isolation of different applications running on them [5]. The separation of policy and mechanism makes it possible to decrease the size of trusted code which makes it easier to build safety-critical applications.

When mixing safety-critical applications with conventional applications without security implications, the isolation of microkernels allows these applications to run on the same computing node. When there is no need for sharing any resources, a reasonable configuration of the microkernel will usually even allow the safety-critical application to work properly when the conventional application misbehaves.

If the applications share a resource, some kind of mechanism has to be provided for sharing this resource and protecting it against denial-of-service-attacks (DoS). Simply allowing all applications to access the resource will not be a sufficient solution. This can be done by implementing another application that acts as a server for the other applications when they try to access the shared resource. For making the resource accessible, the server will send and receive messages through the inter-process-communication-mechanism of the microkernel.

The *Universal Serial Bus* (USB) is a very popular bus system for accessing peripherals. It is quite common that a lot of peripherals are attached this way. This will turn the USB into a shared resource, when different applications need to access different peripherals that are made available through the bus.

Since it is very common to use the USB as a shared resource, a sufficient mechanism for sharing is needed. In this paper, a design for this kind of mechanism will be provided. The sharing itself will lead to two questions:

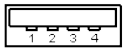
1. Which USB transaction can be excecuted when?
2. How will access permissions for devices be checked?

In chapter 2 a quick overview of USB is given. Chapter 3 will provide a short application scenario. A general overview of implementing USB support in microkernels is given in chapter 4. Ideas on how to solve the problem 1 are discussed in chapter 5. Chapter 6 describes solutions for problem 2. A design overview over the server is given in chapter 7.

2. USB

The Universal Serial Bus [10] is a serial bus system widely used on the desktop market. It is used for different tasks such as tethering peripherals in PCs, as powering gadgets like a fondue pot. USB is standardized by the *USB Implementers Forum*; the current version of the standard is V3.1, published in 2013.

The bus is strictly hierarchical and host-centric, this means every information exchange will be routed through the host. This can be seen in the design of the USB connectors. The connector type A (see figure 1a) always faces towards the *host* and the connector type B (see figure 1b) always faces towards the *device*.



(a) USB connector type A [3]

(b) USB connector type B [3]

Figure 1: USB connectors

The descriptor hierarchy in the USB specification is a very important concept from a software point of view. Descriptors allow to get information from a USB device about what kind of device is the current one, what configurations are available and what configuration endpoints can be used. Therefore, there are *device desriptors* containing *configuration descriptors* that contain *endpoint descriptors* themselfs that contain *interface descriptors*.

The device descriptors contain the information describing the device, for example the *Vendor IDs* and *Product IDs*. These device descriptors also contain information about the device's *device class*. A device class specifies the type of device, for example keyboard, storage device and other device types are possible. By providing a default driver for these device types, they can be supported without a special driver provided by the manufacturer, given, the device follows the device specification properly.

For the configuration definition, the configuration descriptors are used. Endpoint descriptors describe the endpoints, that can be used as endpoints for communication. Every endpoint is the end of a unidirectional pipe to the USB host. *IN*-endpoints are used to send information to the host and *OUT*-endpoints are used to obtain messages from the host.

For the data transfer there are four different transfer types:

Control transfer: This transfer type is used to exchange status information or similar data.

Interrupt transfer: This transfer type should be used to send urgent data, since it has a guaranteed latency.

Isochronous transfer: This should be used for periodic data transfer, for example for video streams. This transfer type guarantees a certain bandwidth.

Bulk transfer: When transfering large amounts of data without time restrictions, this transfer type should be used.

3. APPLICATION SCENARIO

For a better understanding of the problems that arise from using the USB as a shared resource, we will provide a short application example.

In the automotive industry, the trend is going towards Android-based media centers in cars. If we want to use this Android media center as a computing node for real time data processing of a sensor communicating via USB, we need some kind of architecture that supports the sharing of the USB.

We consider the following example: A media-center in a car is running Android. The installation of user-provided apps is allowed. It is possible to connect mp3-players via an USB slot and use them as storage devices to play music. Additionally, a tachometer is attached via USB to the media-center, to gather the mileage information.

To prevent malicious apps to change the mileage data and to interfere with the data from the tachometer, we need a microkernel based system, that runs the Android system as one task and the real-time application as the other. In this way, changing the mileage data from the Android system is not possible. The only problem that remains is how to share the USB between the Android system and the real time mileage application.

4. THE USB-SERVER

To approach this problem we first have to take a look at what is needed to provide USB support in a microkernel environment. Sharing access to the USB hardware for all applications running in the microkernel environment is considered a bad idea, because there could be conflicts between the several applications that access the bus. Therefore, implementing a server that encapsulates the required USB functionality is a better choice. To get a short overview of what is generally required, we will briefly discuss the USB-driver approach in the Linux kernel.

In the Linux kernel (see figure 2) there are three layers of USB drivers:

1. The host controller driver, which provides access to the host controller hardware that controls the USB
2. The USB core, that helps higher level drivers access the bus system and
3. USB device drivers, that provide support to several USB devices.

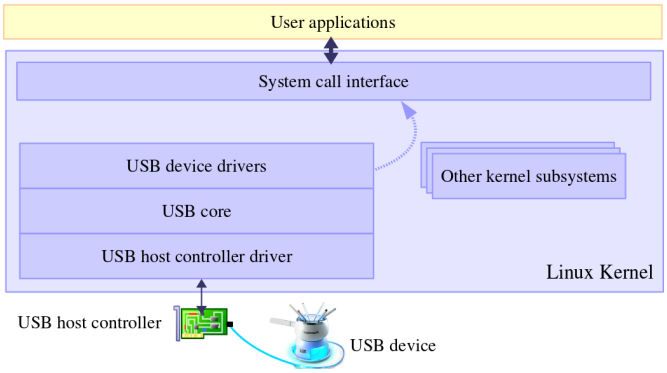


Figure 2: USB drivers in Linux[8]

When an USB device driver tries to send data to the device it handles, this is done through *USB Request Blocks* (URB) [6]. An URB encapsulates all information that is required to do a complete USB transfer.

The straightforward approach to supporting USB in a microkernel would be to implement a server containing the layers 1 and 2 of the Linux USB stack. The server can receive URBs from different client applications and forward these transfers to the USB. Layer 3 would be implemented by different client applications that have access to the server and send him URBs.

If we follow this architecture, we can identify two problems:

1. How can we decide, which URB should be processed next, so that no DoS attacks against the USB server are possible?
2. How can we restrict access to the bus so that only trusted applications can read and write from safety/security-relevant devices?

If we reconsider the application example, we can see that when no sufficient solution for the first problem is provided, it will be possible for an Android app to spam the USB so that no mileage data can be received by the real-time application. If no solution for the second problem is provided, it might even be possible for a malicious android app to send commands to the mileage data sensors and disable them or bring them to an inconsistent state. In the following chapters we will discuss solutions for these two problems.

5. BUS ACCESS SCHEDULING

For using shared resources, we have to define who will be the next to be granted access to this resource. This is called scheduling. The easiest solution for this problem is to simply use a first-come-first-serve algorithm for all clients. This could be implemented through a simple FIFO-datastrucure. If we think of the application example, this is problematic. A malicious Android app could submit lots of URBs and therefore block the real-time-applications access to the bus. This could result in the real-time application not receiving the mileage data. Even the real-time application itself could be threatened. Urgent URBs might be submitted later than non-urgent URBs. This could result in the urgent URBs being sent later than when they are due.

Therefore, we will compare several scheduling algorithms for real time scheduling. A very popular scheduling algorithm in this domain is *earliest deadline first* (EDF) [7]. The idea of this algorithm is to add a *deadline* to each task. The deadline describes at what time a certain task has to be finished. The next task to be started is the task with the closest deadline. It can be proven that preemptible EDF is an optimal algorithm when scheduling for a single resource. It can even be proven that if an EDF schedule fails to comply with a deadline, no solution that complies with all deadlines is possible. Since not all tasks have a deadline, finding sufficient deadlines can be problematic.

Another quite important algorithm is *least laxity first* (LLF) [4]. This scheduling algorithm calculates the laxity l with deadline d, remaining task execution time c and current time t, determined by

$$l = (d - t) - c$$

which can be seen as the time that can pass until the task has to start. The task with the smallest laxity l is the task, that is executed first. LLF can be proven to be optimal for scheduling for more than one ressource. It is not applicable in our case, because we don't quite know the execution time c. Also, for tasks with equal execution time LLF behaves exactly the same as EDF. It is reasonable to assume, that processing URBs has a quite similiar execution time.

Fixed priority scheduling[1] assigns each task a fixed priority. The task with the highest priority is scheduled first. Scheduling algorithms for periodic tasks like RMS [7] are not covered here, since we are not dealing with periodic tasks.

For our problem, we will regard the USB host controller as the resource that is scheduled. The tasks to be scheduled will be the single URBs to be sent. A possible solution for efficient host controller scheduling would be to use EDF for scheduling the URBs. Every URB has an attached deadline. Non-urgent packages could be attached a deadline infinitely in the future. In this case, a misbehaving client could still block the bus by submitting URBs with very close deadlines. This could be fixed by a static configuration that is read by the USB server at startup. In this configuration, the applications that have to use real-time features could be listed. These could be the only applications that are allowed to set the deadline of an URB. All URBs sent by other applications will have a deadline infinitely in the future, these would be scheduled with a FIFO algorithm.

For providing a different scheduling algorithm for the non-real-time applications, two queues could be provided. One (with a high priority) would be scheduled with an EDF algorithm and conatins only URBs by privileged real-time applications. The other queue (with low priority) would contain a queue of non-real-time URBs; this queue could be scheduled with any non-real-time scheduling algorithm. This design can be seen in image 3. The numbers in the orange squares represent the deadline of the URBs.

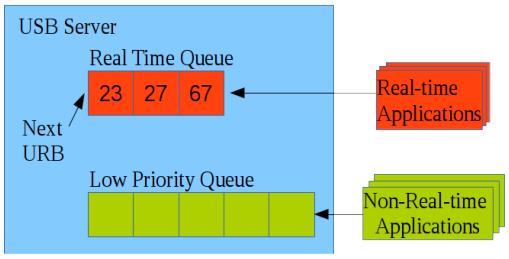


Figure 3: Scheduling with two queues

6. ACCESS RIGHTS MANAGEMENT

The application scenarios in chapter 3 show that having multiple applications and multiple USB-devices can be dangerous, especially if a sensitive environment is concerned. The access of applications to USB-devices need to be restricted

and managed by the system in order to prevent a failure or even a malicious threat to harm the sensitive environment. The proposed USB-server as the only module to access the USB-host of the system stands between the potential misuse of applications on USB-devices. Therefore it needs to manage which applications may make use of which devices.

Let us say that the USB-devices to be protected are objects, then pairs of objects and rights can be called domains. The system needs to know any domain combined with any object to make a decision. The result can be seen as a table, where e.g. the rows are domains and the columns are the objects. Each box in the table then holds the rights between the specific domain and the corresponding object. This table is redundant, since many combinations of domains and objects do not have any entries. By declaring that every empty cell of that table means that there are no access-rights between this domain and the object (*white-listing*), the redundancy can be minimized. Also, the *principle of least privilege* is maintained. Finally this table can then also be seen as a list of domains, where each domain holds a list of object-right-pairs (called *capabilities*), or as a list of objects, where each object holds a list of domain-right-pairs (called *Access Control Lists* (ACL)) [9].

In an ACL each object (the USB-devices) holds a list of domains (the applications) and which rights they provide, like read and write on the devices. This list will be queried each time a domain wants to access an object. Unlike the ACLs, capabilities are domain-driven. That means that each domain will hold a list of objects and the information on how it may access these objects. Therefore, the concepts of ACLs and capabilities can be seen as complementary.

The ACL and the capabilities both have their advantages and disadvantages. The capabilities are efficient; using generic rights, like copying and inheriting a domain's capability-list, provide a top-down-structure where access-rights can be controlled for subdomains. On the other hand, if an object's rights need to be revoked system-wide, e.g. because the object is removed, each domain's capability-list concerning that object must be reconsidered. If then an object is being removed without revoking all domain's rights beforehand, orphan-rights may occur and lead to even more problems. Since in an ACL any object holds their own list of domain-right-pairs, this problem doesn't occur there. But having many domains will cause the ACLs to grow very long, thus making finding, inserting and deleting items in this list less efficient [9].

But even though the capabilities provide some very useful features, there occurs a major problem for our case. The concept of capabilities is that the access to an object is granted purely on the mere possession of a capability. An application must therefore obtain and use their capability on accessing an USB-device, and therefore the application's code has to be adjusted. Practically, many changes on the APIs would be needed. ACLs on the other hand can be used in a centric and isolated USB-server-module. Every time an application wants to access a device via the USB-server, the server will query its ACL and decide whether or not this query will be granted.

In our use-case it is stated that there are a few, very distinct applications but multiple, also very distinct USB-devices. But USB-devices like thumbdrives can be added to the system on runtime (*hotplugging*) and they may be completely unknown to the system beforehand. On the one hand, the system needs to react to an arbitrary USB-device and give the applications access-rights. Following the principle of least privilege mentioned above, any newly added device which was unknown a priori will therefore not be accessable by applications at all. This leads to the question, how the access-rights are created and provided for hotplugged devices. Obviously there is a wide range of different USB-devices which can't all be known beforehand. Furthermore, providing basic rights in general is a security threat, while not providing any rights to hotplugged devices renders them useless and therefore isn't an option either. Our use-case states that both types of applications need to be handled at best; e.g. a real-time-safety device as well as a multimedia-application based on Android. While the safety-devices have our priority in this paper, the usability of an USB-thumbdrive containing music-files still needs to be as good as possible.

While there are plenty of possibilities to authenticate USB-devices and applications, like a key-system, there will always be the need to adjust the system more or less, e.g. that every USB-device needs an authenticating key through which basic access-rights can be determined by the USB-server. This will render any *unprepared* USB-device useless, thus making a plug-and-play of an arbitrary USB-thumbdrive containing music-files impossible. Therefore another option is to create a configuration which will be read on boot. It will determine which applications have which rights on which USB-device. While not all different devices can be covered, groups of devices can. Using the USB device descriptors (see chapter 2), classes and subclasses (like *mass-storage-device*) can be used to group unknown devices and provide basic access-rights to the applications. Additionally the vendor- and product-IDs can be used as well. This way, common USB-devices like thumbdrives can still be used when hotplugged, while critical devices can still run safely. Adding group-rights to the concept of ACLs basicially equals a minimal Role Based Access Control model ($RBAC$) [2].

7. SERVER DESGIN

The implementation of USB support in a diversified and heterogeneous environment can be realized with a server that solely receives URBs. The desgign can be seen in figure 4. The URBs are processed by the server and forwarded to the USB controller hardware. This has the advantage, that porting e.g. linux drivers for USB devices can be quite easy. When running a linux kernel on top of the microkernel, a pseudo driver that simply sends URBs to the USB server can be implemented. All existing USB device drivers for the linux kernel can be reused.

After an URB has been sent to the server, it is checked whether the client sending this URBs has sufficient rights for performing this operation. The rights are realized based on the concept of ACLs described in chapter 6. On boot, for each plugged-in USB-device the respective Access Control Entries (ACE) will be created based on the definitions in the static configuration. These will most likely include critical USB-devices who e.g. need to run in real-time. When

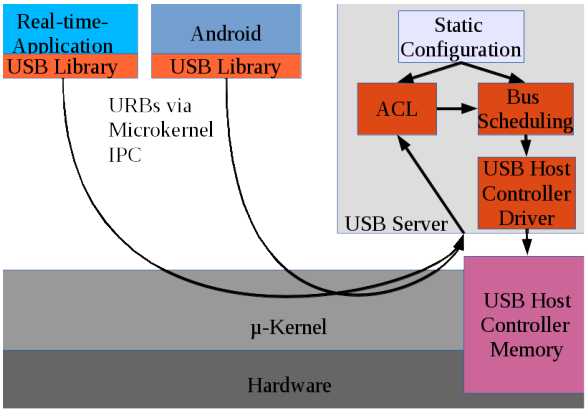


Figure 4: Architecture of the proposed server

an USB-device is hotplugged — e.g. USB-thumbdrives and the like — the server will determine the respective ACE by reading the static configuration again. The configuration can contain rights for explicitly stated devices by device-ID, or group-like rights for multiple unknown devices through using device descriptors. If then the application is sufficient to access the device, it will be granted. If no sufficient rights can be determined for the application, the access will be denied and any URB sent to the server will be ignored.

After having passed the ACL, the URBs can be submitted to the bus scheduling module, that applies a scheduling algorithm described in chapter 5. This component then decides, which URBs should be processed next and accesses the USB master controller hardware. This can be done by accessing the USB device controller io hardware provided by the microkernel.

8. CONCLUSIONS

In this paper we have discussed a design-pattern for a shareable USB-server based on microkernel-architecture. The application scenario is focused on a heterogenous and diversified environment, where critical, possibly real-time USB-devices and user-friendly mulimedia devices both need to be addressed. The critical real-time devices are granted a priority here, whereas the less critical ones need to be treated as good as possible.

The main issue is that non-real-time components can interfere with the real-time-components maliciously. Therefore we figured that the biggest threats on this shared USB-host concern the bus-scheduling as well as the access-management. Additionally, we focused on making the least possible changes to the environment, so that no changes on neither the USB-devices nor the applications (e.g. USB-drivers) are needed.

Providing access to the bus is done by submitting URBs to a USB server, that then forwards the requests to the bus. The provided solution will allow the leveraging of existing Linux driver code, since this approach uses URBs, just as linux USB device drivers. Also, a host controller driver for a Linux VM that accesses the server can be easily implemented, because this driver would simply need to forward URBs.

Providing a real-time-ability without being harmed by non-real-time components is reached by mixing priority based and deadline scheduling. The solution is working well, when there is only few real-time activity. Intense activity of real-time based applications will basically exclude non-real-time applications from the bus.

By using the concept of ACLs and the principle of least privilege, the access to USB-devices can be restricted. Due to the wide range between real-time-ability and consumer-usabilty, tradeoffs had to be made. While whitelisting cancels out unknown and therefore possibly malicious devices in general, the static configuration allows an administrator to explicitly grant access to hotplugged devices. Using USB-device descriptors for device-classifications adds to the usability. But granting general access-rights to the system lowers the security-level, e.g. by using adjusted USB-devices and applications. This ultimatively means that the administrator of this implementation needs to adjust the security-level to his needs, starting at highest security and lowest usability as default configuration.

9. REFERENCES

[1] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2-3):173–198, 1995.

[2] J. Barkley. Comparing simple role based access control models and access control lists. In *In Proceedings of the second ACM workshop on Role-based access control*, pages 127–132. ACM Press, 1997.

[3] Beyond Logic. USB in a Nutshell. *http://www.beyondlogic.org/usbnutshell/usb1.shtml*, 2014.

[4] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

[5] P. Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4), 1970.

[6] Linux Kernel. Usb request blocks. *http://free-electrons.com/kerneldoc/latest/usb/URB.txt*, 2009.

[7] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[8] M. Opdenacker. Linux usb drivers. *http://free-electrons.com/doc/linux-usb.pdf*, 2009.

[9] A. S. Tanenbaum. Modern operating systems. *Prentice Hall PTR*, 2, 2001.

[10] USB Implementers Forum, Inc. USB Standard Version 3.1. *http://www.usb.org/developers/docs/*, 2013.

Feasibility to replace Interprocess Communication by the Message Passing Interface in microkernel contexts

B. Sc. René Drolshagen
Wiesbaden University of Applied Science
Department DCSM
Unter den Eichen 5
Wiesbaden, Germany
info@rene-drolshagen.de

B. Sc. Lasse Löffler
Wiesbaden University of Applied Science
Department DCSM
Unter den Eichen 5
Wiesbaden, Germany
lasse.loeffer@gmail.com

ABSTRACT

The dominant male in the last 35 years of microkernel development was the Interprocess Communication (IPC). Since the day microkernel came up, using IPC as the message passing model was never changed. Only the message latency was reduced by a factor of 250 times between 1993 and 2014, caused by kernel improvements and hardware development. Maybe this decision should be reviewed in favor of the also long known Message Passing Interface (MPI). This paper will give an introduction to the two message passing models to name theoretical problems which will come up in the case of using MPI as a IPC replacement.

1. INTRODUCTION

Since the microkernels came up, the message passing was always done by the Interprocess Communication (IPC). Now that the microkernels are about 35 years old [Mic14] and the ways to pass messages inside a kernel have developed, it could be time to check whether a replacement is advisable. This paper will introduce the Message Passing Interface (MPI) and the actual dominant male Interprocess Communication (IPC) in the way it is implemented in the actual L4 microkernel family. A comparison of these technologies and a feasibility study with all detected problems and possible solutions for them is shown at the end of this paper. This list could possibly be incomplete and was not tested by a real implementation. They are generated out of theoretical work and should be proved practically.

2. MESSAGE PASSING INTERFACE (MPI)

The Message-Passing-Interface (MPI) is a specification which tries to solve the problem between portability, efficiency and functionality. The development of MPI began in the year 1992 based on existing ideas and libraries for communication. Today the most recent MPI version is MPI-3 [Mes12]. MPI should not be recognized as an implementation, more as a library which defines return values, functions and parameter. Based on this library, there are many existing open- and closed source implementation for different programming languages. The fact that all are implemented along this specification makes them compatible. Originally MPI was designed for distributed shared memory (DSM) architectures, which were becoming popular at the time when MPI was developed. As the architecture trends changed towards Non-Uniform Memory Access (NUMA) and No Remote Memory Access (NoRMA), MPI was adapted. Since this day MPI was able to handle both architectures seamlessly and transparently to the developer. Today MPI runs on virtually any available hardware platform. This is solved by maintaining a distributed shared memory architecture regardless of the underlying physical circumstances of the machine [Wil07][Mic]. A reason for using MPI could be one of the following:

- Portability: Only a little or no work to do if a application is ported from one MPI conform platform to another.
- Standardization: MPI is a message passing library which could be declared as a standard. MPI is supported by the most current available platforms and has replaced many other competitors over the years.
- Performance: Most of the MPI implementations are well developed and tested over many years. MPI is used in the context of parallel programming, so it was optimized to gain a high throughput of messages.
- Functionality: There are about 435 routines defined in the current MPI-3 standard, which includes the majority of the older MPI standards.

2.1 Point-to-Point Communication

The MPI point-to-point functionality typically invokes message passing between two different MPI tasks. Two tasks running on the same host (or even inside same process) do not know anything of the other side process. MPI communication has one explicit sender and another task which is performing the receive operation. MPI has different routines for sending and receiving messages [Bla14].

WAMOS '2014 Wiesbaden, Germany

These are for example:

- Blocking send / blocking receive
- Non-blocking send / non-blocking receive
- Combined send and receive

The perfect case of communicating is that both sides of communication are present at one time. This is often called a rendezvous. But this is rarely the case in the real world. Somehow the implementation of MPI must handle the general case of being not present when the other side wants to send or deliver a message. For this case MPI manages a system buffer to meanwhile save the messages. This MPI buffer is handled by the implementation and is not accessible by the user or any programmer. This buffer could be generated on both sides (sender and receiver). As said before, MPI has got two types of communication, the blocking and non-blocking operations. Most of the MPI routines could be used either one or the other way [Bla14]. These two types will be described in the following points:

Blocking Operations:

- Will only "return" if it is safe to possibly change the application buffer (where the send data is located) and the receiver's data is not influenced by continuing computation.
- A blocking send can be a rendezvous between the two threads, which means that there is some kind of handshaking between the two processes.
- A blocking send operation could possibly be asynchronous if the system buffer of the receiving thread could be used to hold the messages.
- A blocking receive only "returns" after the data is transferred and available for this processes.

Non-blocking:

- Non-blocking operations return to the caller immediately, because it is not necessary to wait on communication events to complete.
- These operations only request the MPI library to perform a send operation to the receiving partner when it is available.
- Non-blocking operations are mostly used to avoid loosing computation time while waiting for a synchronous operation.
- MPI provides functions to check whether a message was delivered or is still waiting in the system buffer.

Another feature of MPI is that messages are transmitted in the right order by guarantee. It is not possible that a message overtakes another. Messages which are sent first will be received first.

2.2 Group and Communicator

There are two control mechanisms inside the MPI specification to guarantee a collision free communication. A group is an abstract to combine MPI processes which are performing the same computational operation. A process could belong to more than one group, but has a unique identifier in every group.

The other mechanism is called the communicator, which could be separated into intra-communicator and inter-communicator. A communicator is responsible for handling communication operations for the processes of a group. As simply could be imagined, the intra-communicator handles communication of processes inside the group, while the inter-communicator handles the communication with processes of other groups. A communicator which is always available is the MPI_COMM_WORLD, where all processes belong to. The whole group and communicator design of MPI is shown on figure 1.

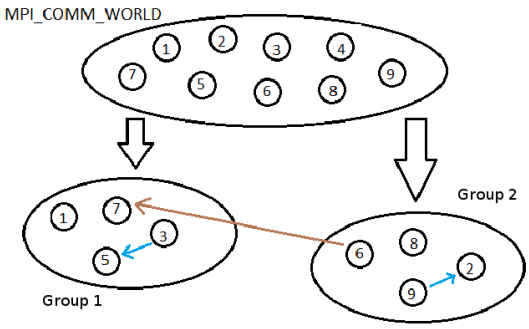


Figure 1: Overview of the MPI group and communicator mechanism.

3. INTERPROCESS COMMUNICATION (IPC)

In an operating system the kernel is the most important object, which is normally separated in two parts: the kernel space (privileged mode) and the user space (unprivileged mode). The early monolithic kernel concepts all the basic system services, like file system, memory management, I/O communication and interrupt handling are being executed in the privileged mode. Today's monolithic kernels have got a layered design, which can be found from the basic process management up to the high level interfaces of the kernel space. The issue is that all the basic services are running in privileged mode causing serious problems, i.e lack of extensibility, large kernel size and bad maintainability [J. 06].

At this point the microkernel concept was born. The idea of the microkernel is to increase the reliability by providing only the basic process communication and I/O control inside the kernel space and moving the other system services into the user space. The system services inside the user space are a form of normal processes, so called servers. On the other hand the basic servers are not longer inside the kernel space and therefore the microkernel needs a concept to allow communication between the servers by entering the privileged mode with a context switch. The servers are separated processes with their living environment. To open a communication channel, the microkernel needs a mechanism to allow an Interprocess Communication. Therefore the so called IPC mechanism is used [Raf]. IPC is not only a part of a microkernel and can also be found in a monolithic kernel like Linux. Normally in a monolithic kernel a process is the representation of a program in memory. In the microkernel context a process is called as task. The smallest section of virtual parallel execution of a program is called thread in both environments. The IPC mechanism differs between

tasks and processes. In the process context the communication is often done without the kernel via a pipe. In the task context a IPC via the kernel is always necessary [Dav99].

The IPC communication in the microkernel is done via messages. A microkernel has smaller code than a monolithic kernel, because basic services are moved into the user space. That means for the microkernel that it has to pass around more messages and the IPC performance has a very significant role in execution time. This is one of the issues of the microkernel against a monolithic kernel, because a monolithic kernel has no boundaries for the address space and can push easily pointers around. Microkernels use the message queue and several other techniques to communicate between servers, tasks and the kernel itself [Dav99]. The idea of IPC inside a microkernel is to create communication between threads and tasks in different address spaces. Each message in a microkernel has got a message tag and an optional list of message data. The message is passed from the sender's MessageData register to the kernel and the kernel copies the message unbuffered to the receiver's MessageData register. The communication is normally synchronized which means that the microkernel only delivers messages when the receiving thread is ready for receiving. The sending thread is blocked until the message is successfully delivered or a defined timeout has been reached [Ste].

The two fundamental IPC operations in a microkernel are send and receive. The send operation delivers messages from the calling thread to a destination thread, while the receive operation requests a message from another thread. The IPC messages can be marked with a flag whether the operation is blocking or non-blocking. If an IPC message is blocking, the IPC and the thread is blocked until the destination thread has performed the receive request. On the other hand, a non-blocking operation fails immediately if the target thread is currently not ready for receiving the IPC message. To reduce the IPC overhead on the microkernel the IPC operations can be combined, which means that sending and receiving is done in one operation [Raf]. In this case the thread sends the message data to the destination and waits for a reply, which saves time by saving one context switch.

To identify a thread or task in a microkernel environment, the kernel holds lists with capabilities. Capabilities represent an address of the destination thread or task. It is comparable with IP addresses at a TCP/IP network. Capabilities are not only an address of a destination, they can also represent permissions to access services. Each server inside the user space has got a capability. For example if a task needs to access the USB driver which is running in user space, than the task will be able to request the capability from the kernel. The kernel looks up if the task has the permission to access the USB driver and sends the capability to access the USB driver to the task. Now the task can start the IPC communication with the USB driver. If a task tries to send IPC messages to a capability without the permission, the kernel would terminate this task because this IPC communication is transferred and checked inside the kernel. IPC Messages in a microkernel environment are not only byte transfers. A message can also contain a capability of a server or interrupt notification from the kernel. The IPC communication is used for nearly every event, which happens in the kernel or user space. A IPC message can transfer address space or regions and is used to handle page faults or exceptions from a thread.

4. COMPARISON

Generally said the Message Passing Interface (MPI) and the Interprocess Communication (IPC) are both ways to pass messages between processes. The first impression of analogousness is changed the more detailed this topic is considered.

MPI could be considered as a kind of API library, whereas IPC is more a single system call. The communication of IPC works with an unbuffered bidirectional queue, which could be filled with information by tasks. The whole communication of IPC is managed inside the microkernel in privileged mode. These interactions could be explained with copying messages over a shared memory segment. The MPI is able to interact over a network or with a shared memory concept. The message passing inside MPI is made transparent to the developer, whether it is done via network or inside the memory of a local machine. Both technologies support a simple one-to-one communication in a blocking and non-blocking way. In addition, MPI has got more functions which allow more comfort to the developer. MPI is able to perform 1-to-n and n-to-1 operations via one simple function call.

The two message passing ways differ in handling the non-blocking communication. While MPI stores the message in a buffer until the receiver will pick it up, IPC fails and returns after a defined timeout. So it could be possible that a MPI communication will sleep and wait forever for the receiver to get the message, whether IPC would return with a failure.

The data structures of MPI are defined by the kind of data which is passed around. MPI has got no clear data structure which must be used by the developer of an application. The default MPI specification includes all primitive types such as int, char, double. But with a simple data-type definition, MPI can transfer own created structs. If a MPI message is sent, the data is copied into a MPI message structure and then is passed to the target.

IPC has got a fixed message register structure, which is stored in the UTCB to transfer data between services. Normally the IPC message contains a tag and an array with pointers into the memory. The IPC tag defines the meta data with flags, source and destination and the array with the payload of the message.

MPI comes with the ability to handle groups of processes. A MPI process could generate new processes and organize them in new groups. Each process could theoretically communicate with every other MPI process in the MPI environment via a communicator.

The speed of MPI was measured on an eight cores (2.53 GHz dual quad-core) and 12 GB main memory machine. The CPUs are based on Westmere architecture and run in 64 bit mode. The nodes support 16x PCI Express Gen2 interfaces and are equipped with Mellanox ConnectX-2 QDR HCAs with PCI Express interfaces. The nodes are connected using a 36 port Mellanox QDR InfiniBand switch. The op-

erating system used was RedHat Enterprise Linux Server release 5.4 (Tikanga). The processes were bound to core 1 on both nodes. The speed was measured with 1.6 μs for a 1 byte message. The execution time is nearly constant until the message reaches a size of 64 byte. After this point the time rises until a message with 1k byte takes 4 μs to be delivered [Dha11]. The speed parameters of the IPC are looking very different. Back in the year 1993 Liedtke [Joc93] described the "IPC dilemma" which was characterized by an IPC message delivery time of about 100 μs. Also he described a way to get a twentyfold improvement, which means that an IPC took about 5 μs. These two values were measured on a 50 MHz machine with a L3 microkernel running on it. Today's time values for IPC communication are about 185 CPU cycles on a 532 MHz (= 532,000,000 cycles per second) machine. This means an approximately time of a about 0.34 μs. This time was measured with the UNSW/NICTA L4 Kernel seL4 [Kev13]. So without calculating the cycle time of the MPI measurement it is obviously that the IPC calls are much faster than the MPI ones.

5. FEASIBILITY OF SUBSTITUTION

Depending on section 4 both concepts look very different. But as we have already seen, there are some issues that need to be fixed or worked around to be able to use MPI as the new IPC in the microkernel context.

The first issue which has to be taken care of is the fact that MPI doesn't have a capability feature. As the MPI section has shown, MPI allows per default every process to send a message to another. This is a security issue in the microkernel context and is managed via capabilities in the IPC context by the kernel. This issue could be worked around by programming the communicator of MPI that only processes of the same group could communicate with each other. So let's for example assume having three processes. Process A is the memory manager which has to send messages to every other process. Process B and C are user processes. To prevent process B from sending messages to process C the simply group structures shown below are implemented:

- Group 1: process A and process B
- Group 2: process A and process C

With the pre-condition from above, it would be possible for process B and C to send a message to process A (our memory manager) and receive a reply. It wouldn't be possible for process B to contact C or the other way around. With this condition it is possible to build a specific kind of capabilities. Also the fact of accessing resources could be managed with this strategy. Let's consider resources as a communication partner outside e.g. group 1. So if process B would access a resource outside the group there has to be a message sent to the communicator of group 1. At this point (the communicator) can be implemented as a kind of security management which will hold some something like access lists. Another way to solve this problem is to design resources as a communication partner which could be added to a group like our memory manager process A. Designing this fact in this way would also drown the problem of broadcasts or 1-to-n messages. The communicator of a group could also check if there is a permission to do so.

Another fact that has to be thought about and solved is the whole MPI message handling. In the non-blocking case there is a huge issue caused by the fact, that memory space is allocated to save the message until the receiver is available. This could be used by a malicious user-level program to perform a denial of service attack against the microkernel with the ability to send an unlimited amount of messages. This lesson was first learned by the developers of the Mach microkernel and was mentioned by Liedtke on many conferences. This was one of the reasons why Liedtke was anxious to only allow unbuffered messages.

In the blocking case there are some more facts to be taken care of. The first is that a process which is sending with a blocking system call could get stuck forever if the receiver is never concerns to appear to the rendezvous for message transferring. A real solution for this problem could lead to dramatic overhead of administrate some kind of timers or something else to quit the send operation. Also this maybe handled in the communicator, but the communicator does not have enough power to cancel a sent operation. This could be a possible solution, if the communicator is able to answer every blocking message operation which has been sent to him. As already said in the MPI chapter every message sending must be done via the communicator. So the communicator is implemented with the ability to answer the blocking messages with a failure after a defined time x (some kind of time out) which will cancel the process blocking.

The last two "issues" which have to be named is the performance of the MPI calls. As shown in the last chapter IPC is definitely faster than a MPI call. This would slow down the microkernel dramatically. A real workaround for this could not be named because the MPI implementations were tuned very hard in the last years, so there may be not more potential to optimise them. The other fact is, that the implementation of MPI has got much more code than the implementation of IPC. So the code in the microkernel would rise definitely and also there has to be more trusted and checked source code.

6. CONCLUSIONS

As the comparison of this paper showed, the difference between MPI and IPC are not that serious as the first impression after the presentation of the different technologies could have indicated. Some of the differences like the buffering problem for MPI could easily be solved in the microkernel context. The bigger problems like the missing capabilities in MPI could lead to real problems in the implementation of MPI for microkernels. Also the fact, that a denial of service attack is possible is a very critical disadvantage of MPI. In the feasibly chapter some ideas are shown to handle these problems in a, more or less, easy way. For some problems no solutions could be named, so that they have to be accepted in the case of replacement.

To summarize, it can be said that there could be a possible chance of replacing IPC with a implementation of MPI. That this should be possible is showed by a MPI based microkernel called PARAM9000[1]. The only fact that would make this idea not practicable is that the MPI calls are not

[1]http://www.cloudbus.org/∼raj/

as efficient as the IPC ones and that the microkernel could be killed by an user-level program. Maybe MPI would be more useful if this issues are solved somehow. Up to now the replacing IPC by MPI in the microkernel context is not advised and meaningful.

7. REFERENCES

[Bla14] Blaise Barney. Message Passing Interface (MPI), 2014. Available online at `https://computing.lln.gov/tutorials/mpi/`; visited on January 10th 2014.

[Dav99] David A. Rusling. Inter process communication mechanism, 1996-1999. Available online at `http://tldp.org/LDP/tlk/ipc/ipc.html/`; visited on January 11th 2014.

[Dha11] Dhabaleswar K. Panda. Performance numbers of MVAPICH2 on Westmere Architecture, 2011. Available online at `http://mvapich.cse.ohio-state.edu/performance/mvapich2/em64t/MVAPICH2-em64t-gen2-ConnectX-QDR.shtml`; visited on January 15th 2014.

[J. 06] J. Soltys. *Linux Kernel 2.6 documentation*. PhD thesis, November 2006.

[Joc93] Jochen Liedtke. Improving IPC by Kernel Design. *ACM Symposium on Operating System Principles (SOSP)*, 1993.

[Kev13] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 - What Have We Learnt in 20 Years of L4 Microkernels? *NICTA and UNSW, Sydney*, 2013.

[Mes12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 2012. Available online at `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`; visited on January 11th 2014.

[Mic] Michael Grobe. An introduction to the Message Passing Interface (MPI) using C. Available online at `http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml`; visited on January 9th 2014.

[Mic14] Microkernel, 2014. Available online at `https://en.wikipedia.org/wiki/Microkernel`; visited on January 16th 2014.

[Raf] Rafika Ida Mutia. Inter-Process Communication Mechanism in Monolithic Kernel and Microkernel. Available online at `http://eit.lth.se/fileadmin/project/142/IPC_Report.pdf/`; visited on January 11th 2014.

[Ste] Stevens, W.R. and Fenner, B. and Rudoff, A.M. *UNIX Network Programming*. Number v. 1.

[Wil07] William Gropp and Ewing Lusk and Anthony Skjellum. *MPI - eine Einführung: Portable parallele Programmierung mit dem Message-Passing Interface*. Oldenbourg Wissenschaftsverlag GmbH, München, 2007.

Interprocess Communication (IPC) in comparison to the Message Passing Interface (MPI) in a microkernel context

Manuel Hermenau
Hochschule RheinMain - Fachbereich DCSM
Unter den Eichen 5
Wiesbaden, Germany
manuel.hermenau@gmail.com

Janos Zweifel
Hochschule RheinMain - Fachbereich DCSM
Unter den Eichen 5
Wiesbaden, Germany
js.zweifel@tele2.de

ABSTRACT

Interprocess communication is one of the most important parts of every microkernel. The Message Passing Interface is a specification for communication between processes. This paper examines similarities and differences between these two mechanisms and discusses the question of the practicability to replace IPC with MPI in a microkernel context.

1. INTRODUCTION

The Message Passing Interface (MPI) and Interprocess Communication (IPC) are both trying to achieve the same goal of providing processes a means to communicate in some way. Communication in that context could mean either transmitting data from one process to another or making a service available for other processes, which is provided by a process. For that purpose MPI sends messages through a network. IPC on the other hand uses cross-address space. So MPI is normally used for paralell computing while the IPC in this paper will be considered as the communication of processes in the context of microkernels.

In this paper we will take a look at both concepts and compare them to each other. Referring to that comparison we will discuss the possibility of implementing the process communication with MPI instead of a more common standard IPC-mechanism.

2. THE MESSAGE PASSING INTERFACE

The Message Passing Interface (MPI) is a specification, which tries to achieve portability, functionality and efficiency. Although it is mostly used in the context of parallelization, the concept of MPI is based on the communication of processes via messages and can thus be used on a single machine.

The work on the MPI standard started 1992 based on already existing approaches and libraries. Subsequently there have been several versions of MPI, peaking in the most recent realease MPI-3 [Mes12]. However, rather than being an implementation, MPI is more like a library, that provides return values, functions and parameters for use in a program. MPI as a library makes it compatible across several languages and platforms.

In the beginning MPI was designed for distributed memory architectures, which were gaining popularity at the time MPI was developed. However, the trend changed into hybrids between distributed memory and shared memory systems which were created by combining shared memory through networks. Therefore developers changed their libraries to handle both types of architecture. By today any hardware platform is supported: distributed, shared and hybrid architectures. Admittedly, the program model is still a distributed memory model, regardless of the physical architecture of the machine [Bar].

There are several reasons for using MPI:

- **Standardization**: MPI is currently the only message passing library that can be considered a standard. It is supported on virtually all hardware architecture platforms. Practically, it has replaced all previous message passing libraries.
- **Portability**: There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- **Performance**: Most MPI implementations are well developed, tested and adjusted to gain a high flow of messages, since it is often used in the context of parallelization. Additionally, vendor implementations should be able to exploit native hardware features to optimize performance.
- **Functionality**: There are over 440 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
- **Availability**: A variety of implementations are available, both vendor and public domain.

Using MPI inside a program requires the MPI environment to be initialized. After that was done, MPI functions may be called to send or receive messages. Once all communication has ended, the environment needs to be terminated

as well. All communication using MPI should be completed within one instance of the environment. Everything else is to be considered unnecessary overhead created by the repeated initialising and terminating of the MPI environment and could lead to unintended side-effects.

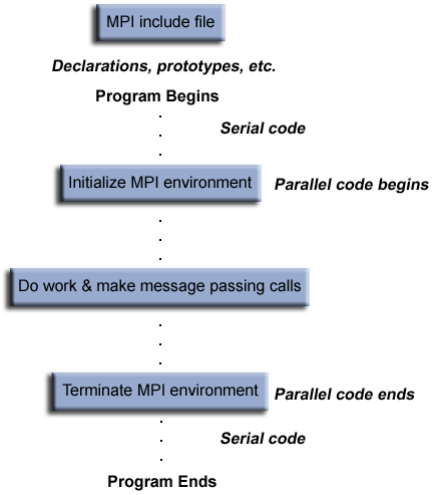


Figure 1: The MPI environment

MPI uses objects known as communicators and groups to coordinate which processes can communicate with each other. However, there is a communicator that holds all processes and can be accessed by calling **MPI_COMM_WORLD**. As soon as other communicators are defined, messages are to be routed through these, if the processes communicating with each other are not in the same communicator-group. Every communicator assigns a unique ID or rank to its respected processes. This is used to identify the sender and receiver, when handling messages and controlling program execution.

Normally every call returns zero as value coded as **MPI_SUCCESS**. If an error occures though, the default behaviour of an MPI call is to abort. Still, there are ways to overwrite that default error handling and implement an own handling.

When sending or receiving messages, several options are available on how the message shall be sent:

- **Synchronous Send:** The receive operation is synchronized with its matching send operation.
- **Blocking Send/Receive:** The send/receive operation blocks the thread for as long as there is no corresponding receive/send operation in another thread.
- **Non-blocking Send/Receive:** The message is saved in a buffer and is transmitted as soon as the receiver is ready. The thread is not blocked during the delay. Functions provide information whether the message was received already or is still inside the buffer.

MPI also guarantees that no message overtakes another, meaning that all messages will be received in the same order they were sent. On the other hand it does not grant fairness. If two tasks each send a message, that corresponds with a receive operation of a third task, only one message will be received and it can not be determined which one will arrive.

3. INTERPROCESS COMMUNICATION

Interprocess Communication, or IPC, is a mechanism for cross-address space communication. Modern microkernels provide an infrastructure to implement the operating system on top of it in the user mode, while the microkernel runs in kernel mode, as shown in figure 3. IPC is used for sharing information or memory between processes in strictly seperated address spaces, which means sender and receiver of an IPC call are always different threads.

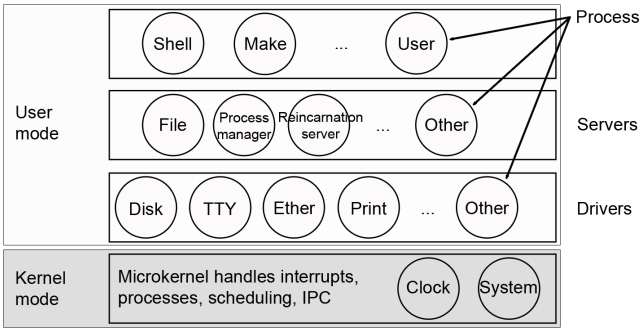


Figure 2: Division of kernel and usermode in a microkernel, by the example of Minix 3. Adapted from [Tan06]

Since applications use cross-address space IPC to interact with traditional operating system services [Ber92], IPC is one of the essential parts of every microkernel. This means that the design and implementation of the IPC mechanism which is used has an enourmous impact on the performance of the microkernel [EH92]. Therefore, over the years microkernel designers tried several different approaches to provide reliability and performance for cross-adress space communication. Also, some concepts come with a bunch of implementation tricks to optimize performance, like the L3 [Lie94] or L4 [EH92].

Traditionally, an IPC mechanism offers the following methods:

- Send
- Reply
- Receive from a specific sender
- Wait for a message from any sender

To reduce the number of system calls, newer IPC concepts encaspulate common usecases into

- Send and receive a message from a certain thread (=Remote Precedure Call, RPC), as shown in fig. 3
- Reply and wait for the next message

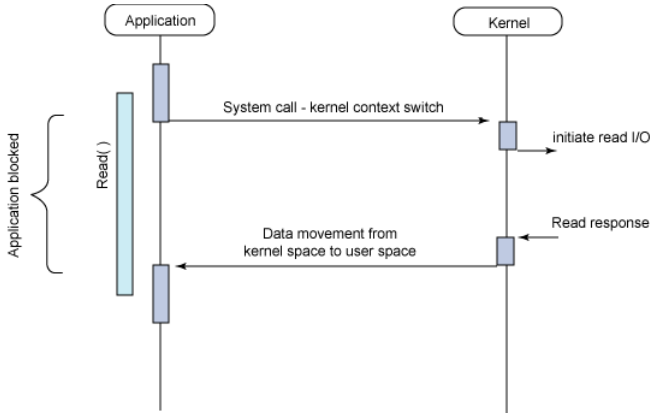


Figure 3: Sequence diagram for a Remote Procedure Call. Retrieved from `http://www.ibm.com/developerworks/library/l-asyncl`

You can break down any IPC concept to being either *synchronous* or *asynchronous*.

- **Synchronous IPC**: The sender and receiver need to meet (*rendezvous style*). When both are ready, the message is copied directly from the sender to the receiver. If one thread has not arrived yet at the rendezvous point, the other one will wait blocking. Therefore, this method needs no buffering.
- **Asynchronous IPC**: Asynchronous IPC calls do not need to meet at a specific point. Typically, the *send* procedure returns immediately, so that the sender does not have to block. At the receiver's side, one can either wait or poll. To avoid heavy copy overhead, some asynchronous IPC concepts, like the seL4, restrict the payload to one word.

Furthermore, there were various concepts introduced on how to address the targets and control who can interact:

- **Clans & Chiefs**: A thread can only communicate with its siblings ("clan") or its parent ("chief"). Outgoing communication is routed through the chief. [Lie92]
- **Thread IDs**: The target is specified by its unique thread ID
- **Capabilities**: Endpoint capabilities define IPC targets

4. COMPARISON

MPI is a specification while IPC can be implemented in several ways. However, there are several points where both techniques either collide with each other or drift apart. Especially considering both are trying to achieve the same thing: Communication between processes. Still, while working with these, you have to keep in mind that one is originally meant to be used for parallel computing, while the other one is used in a microkernel context. For this section we will to look at the Fiasco.OC microkernel([Fia14]) regarding the IPC and at OpenMPI([Ope13]) as an implementaion of MPI.

The Fiasco.OC only uses synchronous IPC, so both communication partners have to meet for the message transfer and one partner blocks while the other one is not ready yet. Once both partners are ready, they use *capabilities* to achieve the transmission of the message. MPI threads communicate via their IDs, which are distributed by their *parents*. These parents are the communicators. Each process can communicate with another process that is within the same communicator. If a message has to be transmitted beyond that, it has to be routed through the sender's and receiver's corresponding communicators. While MPI can as well do the transmission synchronously, it also provides methods to do it asynchonously. In that case the message is stored in a buffer until the receiving partner of the communication picks it up and deletes it from the buffer. The receiver can occasionally check if there are any messages to be picked up, and the sender can check if the messages has been received.

To achieve synchronization in MPI, so called *barriers* have been implemented. These barriers are being called via methods and block the calling process until every process of the group (or clan) has reached the same barrier and has called it itself. One reason why the Fiasco.OC relenquished this is because a denial-of-service attack could cause the kernel to deny any more messages to be written into the buffer (because of a bufferoverflow), thus denying the IPC. In addition, MPI allows the message not only to be sent via a one-to-one communication, but to be broadcasted to multiple processes at once. This way, the same data can be distributed to all processes for computing. In almost the same manner a process can receive data from multiple processes. That can be used to collect the results of a parallel calculation.

Another difference between IPC and MPI is that MPI does not have a permission-like concept like IPC with its *capabilities*. A process in an IPC environment needs to have a capability of the receiver to be able to send a message to that process. In contrast to that, an MPI process can send a message to any other process that is within the same group (clan) of processes.

As mentioned before MPI offers several options on how a message can be sent. These options include methods to transmit all primitive types of data (int, char, double, etc.). Additionally, MPI provides the possibility to create user-based data-types to be sent to the target process. Of course that process also needs to know that data-type. IPC on the other hand uses a virtual register with a fixed structure in the user-level thread control block (UTCB) to transfer data from sender to receiver.

5. REPLACING IPC WITH MPI

As stated in section 4, MPI is a specification for an IPC mechanism which aims specifically at parallel computing. So the real question is: Would it be possible to model microkernel IPC after the MPI standard and what would be the advantages and disadvantages?

Since the original L4 did not include an asynchronous IPC mechanism, most microkernels from the L4 family *do not*

support this communication method. If one would want to replace the microkernel IPC by MPI, he would need to either choose a kernel which supports the methods defined in the Message Passing Interface or to redesign the IPC concept of the chosen microkernel.

Similarly, the microkernel would need to use the *clans & chiefs* model for message distribution, because this is similar to the *groups* concept specified by MPI. The original L4 used this model, but some newer implementations abandoned it again, for example the seL4, which on the other hand supports asynchronous IPC. One reason for the abandonment was an increase in performance when not using clans & chiefs, because some messages do not need to be transferred via the chiefs. Again, you would need to choose a microkernel which implements this model, or redesign an existing microkernel for the use with the clans & chiefs model.

The biggest problem would be, that the MPI specification includes many features which are not needed in the microkernel context, like for example an I/O interface to work with files. The MPI standard specification consists of 822 pages[mpi12], whereas the complete L4 specification consists of only 218 pages[Tea06] with only 20 pages covering IPC. Implementing MPI in an microkernel would blow up the kernel code unnecessarily and, which is much worse, infringe the most essential paradigm of microkernel developing: keeping the kernel as small and simple as possible[Lie95].

The only real advantage of replacing microkernel IPC with MPI would be, that the IPC bindings would be standardised, which could achive better compability between different microkernels and their corresponding userspace-environments. But standardising microkernel IPC bindings by implementing the Message Passing Interface is, as shown in this paper, not a resonable decision. The better approach would be to design a new and clean API which fits the needs of microkernel IPC.

6. CONCLUSION

In this paper we examined the differences between the Message Passing Interface and microkernel IPC. For that purpose we first looked at what MPI and IPC are. Afterwards we analyzed the differences. By doing that we showed how both of them are designed for their special purposes and then discussed the possibility of implementing IPC with MPI in a microkernel context. The analysis concluded in the realization, that this would not gain any real advantages and the better approach would be to define an API standard for the programming language bindings.

7. REFERENCES

[Bar] Blaise Barney. Message Passing Interface (MPI), available at: *https://computing.llnl.gov/tutorials/mpi/*.

[Ber92] Brian N Bershad. The increasing irrelevance of IPC Performance for Micro-kernel-Based Operating Systems. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 205–212. Citeseer, 1992.

[EH92] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels? 1992.

[Fia14] Fiasco.OC & L4 Runtime Environment(L4Re), 2014.

[Lie92] Jochen Liedtke. *Clans & chiefs*. Springer, 1992.

[Lie94] Jochen Liedtke. Improving IPC by kernel design. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 175–188. ACM, 1994.

[Lie95] Jochen Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[Mes12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 3.0, available at: *http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf*, 2012.

[mpi12] MPI: A Message-Passing Interface Standard, Version 3.0, 2012.

[Ope13] Open MPI: Open Source High Performance Computing, 2013.

[Tan06] Tanenbaum, Andrew S. and Herder, Jorrit N. and Bos, Herbert. Can We Make Operating Systems Reliable and Secure? *Computer*, 39(5):44–51, May 2006.

[Tea06] L4Ka Team. L4 experimental kernel reference manual - Version X.2 - r6, 2006.

User-Level CPU Inheritance Scheduling

Marcel Kneib
Hochschule RheinMain
University of Applied Sciences
Schubertstrasse 10
Schwabenheim, Germany
marcel.kneib@posteo.de

Jonas Reininger
Hochschule RheinMain
University of Applied Sciences
Hellmundstrasse 17
Wiesbaden, Germany
jonas.reininger@gmail.de

ABSTRACT
This paper presents a scheduling mechanism known as CPU Inheritance Scheduling [2] with focus on microkernel operating systems. In traditional operating systems scheduling is performed by a scheduler inside the kernel and scheduling policies are typically predefined, which isn't flexible enough for some applications. The main objective of CPU Inheritance Scheduling [2] is to support a more flexible way for even user applications to create scheduling policies in form of scheduler hierarchies in which threads can work as schedulers for other threads. This design principle of a scheduling mechanism provides the option of bringing environments with different scheduling requirements together on one single system with negligible scheduling overhead, even in microkernel operating systems where context switches are more expensive.

General Terms
Theory

Keywords
Scheduling, Microkernel Operating System, User-Level Scheduling Mechanism

1. INTRODUCTION
A scheduler is often implemented as a component in the operating system's kernel space and is generally transparent to user applications. In addition, scheduling mechanisms in todays common operating systems are usually designed to follow a combination of fixed scheduling disciplines. Which techniques in detail are used depends on the consideration of which use in particular a system is intended to and how exacly processes should be scheduled. Unfortunately there is no "best strategy". User application threads should respond as fast as possible. In batch processing threads high latency is acceptable but thoughput is of interest, whereas real-time environments require a scheduler that threads can meet deadlines. To satisfy the increasing requirements of todays applications, it should be possible to combine different scheduling approaches as needed in a single system.

This paper presents a design concept for a hierarchy of threads which can act as schedulers for other threads – known as CPU Inheritance Scheduling [2] – in a microkernel operating system. Since keeping the TCB (Trusted Code Base) as small as possible in such systems is a major goal and to provide sufficient scheduling flexibility for applications, the scheduling mechanism is almost entirely located in user space (see Aegis Exokernel [1]). Following these design principles, combining environments with different scheduling requirements (e.g. interactive and real-time applications) can coexist in a single system with an acceptable loss of performance, since context switching is more expensive due to a higher scheduling overhead.

Chapter 2 describes the functional principle of CPU Inheritance Scheduling [2] in a microkernel operating system, gives an overview of an example scheduling hierarchy and covers problems that come with scheduling mechanisms in general (e.g. priority inversion) and accounting of CPU resources.

Chapter 3 presents an evaluation of CPU Inheritance Scheduling [2] in a microkernel operating system regarding performance and usefulness and Chapter 4 gives a conclusion about this work.

2. CONCEPT
In traditional operating systems, the scheduler is included in the kernel. The major benefit is that there is no additional communication needed. As sequel you incur a static system without the opportunity to combine several uses. For example it is not possible to merge some real time (e.g. car safety system) with general purpose applications (e.g. entertainment features). In this section we suggest a way to get the scheduler out of the kernel, and how its possible to run different mechanisms in one single system.

The main scheduler is the root of all the schedulers you want to combine, which has the control about the CPU. A scheduler can be seen as a thread that spends most of its CPU resources to other threads, which are called client threads. These clients can also act as other schedulers which have their own client threads. A scheduler can spend its CPU time that it gets from its scheduler. If the given CPU time of a client thread expires, the scheduler preempts the current thread and runs another client. If a client blocks because of an expected event, it returns its control back to the scheduler, which can then choose another thread or relinquish the control to its scheduler. This happens until some scheduler finds a client that is requesting time.

In Figure 1 an exemplary configuration is shown. The scheduler S0 is the root scheduler which has one client thread that consumes its resource for its own use. The other client of S0 is the second scheduler which donates its resources to its client threads T1 and T2. The root scheduler S0 could be

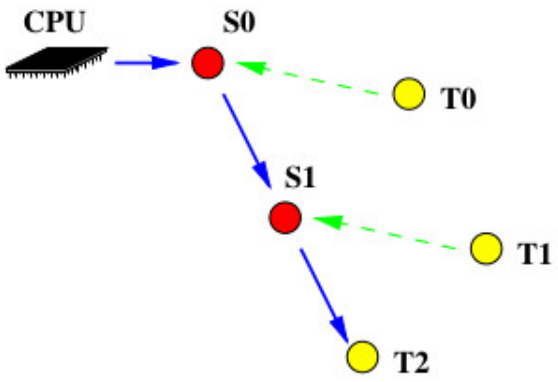


Figure 1: Scheduling hirarchy.

a fixed priority scheduler with a second scheduler S1 and a general purpose applications with a low priority. The second scheduler S1 could use a real-time scheduling mechanism and spend its time to real time applications.

Some low-level mechanisms, like thread blocking, unblocking and CPU donation, still have to be in the kernel. These low-level mechanisms are done by "the dispatcher" which works without making any scheduling decisions. It is deticated for accepting events (e.g. timer interrupt) and directing these to certain threads. Through its access to the hardware it is the only part that has to be in the kernel space [2].

If a client thread wakes up (e.g. a resource is available) or is created it requests CPU time from its scheduler for its execution. This information is delivered to the responsible scheduler by the dispatcher. If the notified scheduler is waiting for such a message it wakes up and requests execution time from its own scheduler and so on. If a notified scheduler is already donating its CPU time then the currently running thread will be preempted and the control is given back to its scheduler. Then it can make a new decision how it donates its CPU time. If the notified scheduler is awake but actually preempted, then the dispatcher knows that the event is irrelevant at the moment and the currently running thread is resumed immediately.

If the scheduler is not activated through a delivered message or a blocking thread within a fixed time the dispatcher preempts the running thread and passes control to the scheduler. The scheduler is now capable to account the expired time and choose another thread to run.

2.1 Priority Inheritance

Sometimes it is an advantage, if a client thread donates its CPU time to a low-priority thread. If a client thread is waiting on an event that is dependent from another client with lower priority it could donate its CPU time to this thread (priority inheritance [2]). This is necessary when a third thread with a middle-priority exists, because it has the ability to prevent the execution of the high-priority thread (priority inversion [5]). This happens when a high-priority thread blocks at a resource held by a low-priority thread. The middle-priority thread is now capable to preempt the low priority thread. Since it holds the resource, the medium-priority thread prevents the execution of the high-priority thread indirectly.

This problem occured on a mars mission with the rover "Mars Pathfinder[3]". A high-priority management client requested a communication channel that was already held by a low-priority data gathering client. The management client blocked and the gathering client continued. Very infrequently an interrupt occured and caused the execution of a medium-priority and long running thread. This thread prevented the low-priority thread from running and consequently the blocked management thread, too.

This technique can also be used for a RPC call[4]. A high priority client can donate its CPU time for the duration of the request to the server. It would be feasible if the dispatcher automatically perform voluntary donation appropriately when the running thread blocks.

It is possible that a single thread inherits execution time from more than one source. For example if two threads running on a multicore system donating its current CPU time to another thread holding the required lock. In practice a thread can not use more than one CPU at once. It only makes sense to inherit CPU time from more than one source if the duration to release the lock takes longer than the time given from the first source. For example two threads donating their time to a thread that is using the requested resource. The thread can now use the inherited time from the first thread to work on. If the given time expires, it can use the time given from the second thread to finish its work and release the lock. A technique for this is called bandwidth inheritance[5] and can be seen as a extension of the priority inheritance protocol. Through this it is possible to donate the time from all threads, that attempt to acquire the resource, to the holding thread.

2.2 Accounting

Schedulers often have to account the CPU resources are consumed by its client threads to decide which to start next. These information are often used for a variety of applications (e.g. usage statistics) and can be measured by many possible accounting mechanisms. In this document we suggest a statistical and a timestamp based type which have to be part of each scheduler in the system. Its possible to use a timer which activates an accounting function in equal intervals which assigns the expired time to the current thread. This method is very efficient if the used scheduler mechanism is based on a periodical interval. An alternative high accuracy method is to account the expired time between each context switch. A drawback is the fact that this kind of accounting extends the context switch time significant because it can be expensive to read the current time in some systems.

The CPU accounting becomes a little more complicated when the scheduler is stacked on other schedulers. If the root scheduler preempts a client scheduler thread to donate time to its clients then the client scheduler has no information about this and assigns the expired time when it is resumed

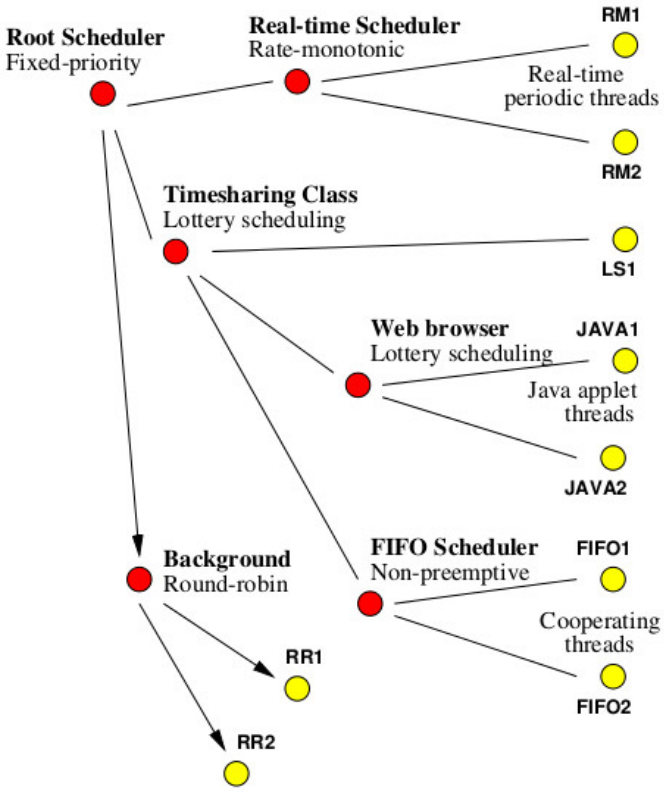


Figure 2: Scheduling structure.

to its client. For example if S1 has donated its time to one of its clients then the root scheduler S0 preempts S1 and in this way the current running client thread. When S0's client finishes, the preemted thread returns and its scheduler assigns the expired time from S0's client to its own client.

If these inaccuracies can not be ignored it is possible to eliminate this problem through the creation of a virtual time for each scheduler [2]. When a context switch or timer interrupt occurs the current active scheduler gets its time increased. With this method it is possible to account the consumed time correctly.

2.3 Multiprocessor Support

It is possible to run a seperate instance of the root scheduler on each processor. This brings the disadvantage, that a client thread is bounded to its scheduler and consequently to its CPU. In the most cases this constellation is not needed, except each CPU is dedicated to a specific usage.

If real multiprocessor scheduling is needed a multi-threaded scheduler is assumed, so that different threads can be distributed to the existing processors.

3. EVALUATION

The results of experimental tests in [2] based on a prototype implementation in a user-level threads package with a sample multilevel scheduling hierarchy show that the concept described in Chapter 2 works as expected. Schedulers arranged in a hierarchy supporting different scheduling policies coexist in one single system without interfering with each other. For example, it is ensured that client threads scheduled by a real-time scheduler running with high priority don't get interrupted by background threads which are scheduled by a low-priority Round-Robin scheduler (Figure 2). Performance analysis in [2] shows that CPU inheritance scheduling is applicable and practical, even in microkernel operating systems where system calls commonly require more context switches.

3.1 Test Environment

Bryan Ford, Sai Susarla and their team implemented a prototype in a user-level threads package. This package provides the common techniques like mutexes, semaphores and inter-thread communication. Furthermore separate thread stacks and a virtual CPU timer were used for preemption and timer interrupts. The dispatcher itself is isolated from the system and will be executed in the currently running context. It supports unlimited scheduling depth and complexity. The dispatcher is written in 158 lines (semicolons) and the scheduler in about 100 lines. All measements were taken on a 100 MHz Pentium with 32 MB RAM and FreeBSD 2.1.5.

In the following, the used scheduling hierarchy is shown. The root scheduler is a nonpreemptive fixed priority scheduler which arbitrates between three client schedulers (Real-time, Timesharing and Background). The first client scheduler with the highest priority is a real time ratemonotonic scheduler with two client threads. The second client scheduler is a lottery scheduler which manages another lottery- and a fifo scheduler. The third, with the lowest priority, is a simple round robin scheduler. The full structure is shown in figure 2.

3.2 Performance

CPU inheritance scheduling [2] in comparison to traditional scheduling techniques causes additional context switch overheads: dispatcher costs and context switch costs.

Dispatcher Costs

Dispatcher Costs are caused by the dispatcher itself. Depending on the depth of the scheduling hierarchy, the dispatcher has to iterate through trees and linked lists in order to compute the next thread to run. However, in practice, a limited depth of the scheduling hierarchy to four or eight levels should be suitable for almost any purpose. This provides a flexible way for CPU inheritance scheduling and acceptable computational overhead.

Context Switch Costs

Costs of additional context switches to and from scheduler threads occur when using CPU inheritance scheduling. Due to the fact that the scheduling overhead for a single context switch varies widely in different environments, this overhead is not considered. Instead, additional context switches that occur when using CPU inheritance scheduling are counted. While in the test environment (Section 3) for example, there is little overhead for switching because it is running in usermode, in a kernel environment it is more expensive.

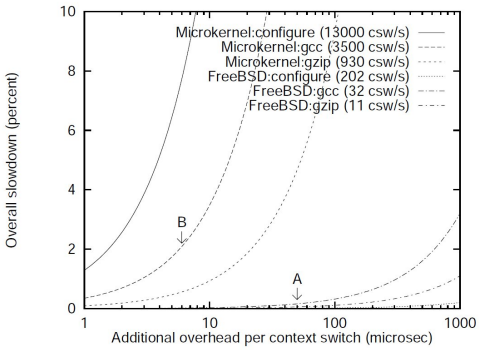


Figure 3: Scheduling overhead and overall slowdown.

> So comparing costs for a single context switch is not really useful to get information about performance issues of CPU inheritance scheduling. However, counting the number of additional context switches shows that for each ordinary context switch on average, approximately one additional scheduler thread invocation can be expected.

Figure 3 shows the scheduling overhead related to the overall slowdown in microkernel and monolithic kernel operating systems. Because microkernels commonly require more context switches, it is desirable to keep per context switch costs as small as possible. If scheduling is mainly implemented in user-space, where per context switch costs are almost free, CPU inheritance can be used in such environments to achieve scheduling flexibility with negligible overhead as the test results and performance analysis in [2] show.

4. CONCLUSION

In this paper a possibility for a multi-level scheduling hierarchy was shown. It allows the existence of varying scheduling policies in a single system. The framework from [2] prevents priority inversion by priority inheritance and shows that the presented concept can even be used in environments in which context switches happen more often, because the additional per context switch overhead is small.

5. REFERENCES

[1] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev.*, 29(5):251–266, Dec. 1995.

[2] B. Ford and S. Susarla. Cpu inheritance scheduling. In *IN PROCEEDINGS OF THE SECOND SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION*, pages 91–105, 1996.

[3] M. Jones. What really happened on Mars Rover Pathfinder. `http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html`, 1997. [Online; accessed 31-01-2014].

[4] J. Liedtke. Improving ipc by kernel design. *SIGOPS Oper. Syst. Rev.*, 27(5):175–188, Dec. 1993.

[5] U. Steinberg, A. Boettcher, and B. Kauer. Timeslice donation in component-based systems, 2010.

User-level scheduling mechanisms

Andreas Zoor
University of Applied Sciences
Hochschule RheinMain
Kurt-Schumacher-Ring 18, 65197 Wiesbaden
Email: andreas.b.zoor@student.hs-rm.de

Nikolai Nagibin
University of Applied Sciences
Hochschule RheinMain
Kurt-Schumacher-Ring 18, 65197 Wiesbaden
Email: nikolai.b.nagibin@student.hs-rm.de

***Abstract*—"The most processor scheduling mechanisms in operating systems are fairly rigid, often supporting only one fixed scheduling policy, or at most a few scheduling classes whose implementations are closely tied together in the kernel part. This paper introduces a solution for user-level scheduling based on CPU inheritance, a novel processor scheduling framework, developed by Bryan Ford Sai Susarla, in which arbitrary threads can act as schedulers for other threads. With user-level scheduling, operating systems can support scheduling policies which can be adapted to the needs of individual applications. Therefore operating systems that support user-level scheduling are more flexible than others. Conclusively we give a little outlook to Exokernels which carry out user-level scheduling."[1]**

I. Introduction

The idea of an microkernel based operating system is to minimize the kernel part of the operating system, in order to permit modularity, flexibility and a small "Trusted Computing Base". Modern microkernels just include a messaging service for inter-process communication (IPC) and the scheduling mechanism. So all resource-management policies of the operating system have to be implemented at user-level as a server application. If an application wants to use a service of these policies, it must communicate via IPC messages to the specific server.

Traditionally, operating systems use a fixed scheduling scheme to share the CPU resources, typically based on priorities. Mostly a few variants of the basic policies are provided, such as support of several "scheduling classes" to which threads with different purposes can be assigned to (e.g. real-time) or fixed-priority threads. Normally these variants are hard-coded into the the system implementation and cannot be easily adapted to the needs of individual applications. So if we could move the scheduling mechanism into the user-level the operating system would be even more flexible.[1]

In this paper we report about one solution to implement scheduler threads in the user-level, which is used in a framework for inheritance scheduling. We also show the introduced overhead of this solution.

The idea of inheritance scheduling is that threads can act as schedulers for other threads themselves. The scheduler threads temporarily donate their CPU time to a selected thread while waiting on events of interest such as timer interrupts. The thread which gets the time from the scheduler can also act as a scheduler thread. This allows to build a logical hierarchy of schedulers as illustrated in Figure 1.[1]

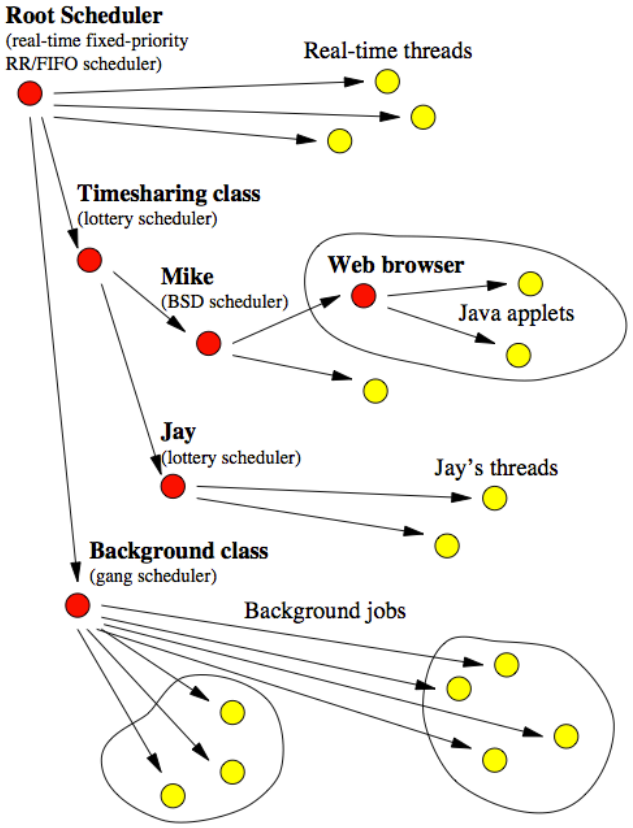


Fig. 1. Example for a scheduler hierarchy[1]

II. Related Work

In their paper "CPU Inheritance Scheduling", Bryan Ford and Sai Susarla present a new processor scheduling framework in which user-level threads can interact as schedulers. These scheduler threads can implement different scheduling policies, so that one single system supports multiple policies at once. The goal of this framework easily adapts scheduling techniques to the needs of individual applications. In their paper Ford and Susarla demonstrate that such flexibility can only be provided with minimal overhead, depending on factors such as context switch speed and frequency.

III. User Level Scheduling

Generally, threads are scheduled on a low level, either by the kernel scheduler or by some user-level thread packages.

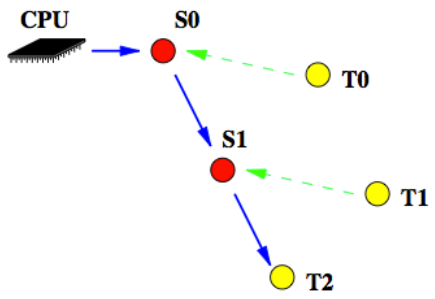


Fig. 2. Example for a scheduler donation[1]

With the concept of CPU inheritance scheduling, inroduced in the work of Bryan Ford and Sai Susarla, higher-level threads gain the ability to voluntarily donate and request CPU time to and from other threads. The sole thread that actually owns real CPU time is the root scheduler that can transfer its available portion of actual CPU time temporarily to other client threads (which can be scheduler threads). On multicore systems, there exists one root scheduler thread for each real CPU. It is technically not possible to implement one sole root scheduler that manages all CPUs at once. Scheduler threads primarily spend their time donating its CPU resources to client threads, that inherit this allocated portion as their virtual CPU.

For CPU inheritance scheduling a few low-level mechanisms still have to be in the kernel. One of these mechanisms is called ”dispatcher”. The dispatcher implements the functionality of thread blocking, unblocking, and CPU donation. Also the dispatcher receives events (e.g. an interrupt or wake-up of a thread) and redirects them to the threads that are waiting for those events. The real scheduling decisions are made by the scheduler threads, so the dispatcher is not an own thread, it runs under the context of a scheduler thread in the kernel space. Figure 2 shows an example for scheduler donation, where the root scheduler S0 donates its CPU time to S1 which is a scheduler thread that spends its CPU time further to T2.

Following this principle a tree-like hierarchy can be evolved by all participating clients distributing their CPU time among a chain of threads. Thus, recently created threads cannot be run until they are provided with CPU time from their responsible schedulers. Hence CPU time has always to be requested as soon as a thread needs to be processed. In that case the dispatcher sends a notification through an IPC to a Mach-like message port of the scheduler thread to wake it up. This mechanism can lead to a chain reaction where different scheduler threads are woken up. While in traditional systems priority inheritance would be used, CPU inheritance allows running threads to voluntarily donate their time rather than block and wait for an event to occur. As for instance one thread is holding a lock, another thread that needs to obtain that locked resource donates its CPU time to the former thread (more in section III-A2). As soon as the resource lock is released the donation ends and the CPU is passed along to the donator thread again. But it is also feasible for a sole thread to receive CPU time from more than one other thread simultaneously.

While in the implementation of inheritance scheduling the dispatcher is automatically invoked by the IPC primitives to perform voluntary donation, a voluntary donation or an explicit dispatcher call could be implemented as well. But one eventual problem arises from threads that consume CPU time from multiple donators an hence will always produce an ”avalanche effect” when they are woken or preemted because the dispatcher sends multiple scheduling requests at once, which on their own could result in even more requests by woken up intermediate-level schedulers. But in reality this is hardly probable due to the fact that a thread inherits from more than one or two different threads contemporary. It is also possible that threads relinquish the CPU as soon as all their work is done. In this case the dispatcher hands back the CPU to its scheduler. This procedure can go up the hierarchy until some thread is found that is willing to work.

A. *Timing*

For preemptive scheduling it is important to have a measurable knowledge of time, therefore a periodic interrupt is accurate enough in the most cases. This means, that for user-lever scheduling all what is needed is a way for a scheduler thread to be woken up after an amount of time has elapsed. A solution to support this, is that every scheduler thread can register timeouts with an own interrupt handler. So when a timeout occurs, an IPC message is sent to the corresponding schedulers port, waking up the scheduler.[1]

1) CPU Usage Accounting: Most schedulers have to account the usage of the CPU to decide which thread to run next. As with scheduling policies, there are many possible CPU accounting mechanisms, each with different cost/benefit tradeoffs. But there are two well-known approaches to CPU usage ac- counting: statistical and time-stamp based.[1]

Statistical accounting
By statistical CPU usage accounting mechanisms the scheduler wakes up on every clock tick to change the quantum of the running thread. This method is only efficient when the scheduler generally wakes up on every clock tick anyway.[1]

Time stamp-based accounting
In case of time-stamp based CPU usage accounting mechanisms the scheduler reads the current time on every context switch and elevates time since the last context switch, to change the quantum of the corresponding thread. This method provides extremely high accuracy, but also increases the time for context switch times, especially on systems where reading the current time is expensive. Thus the costs are much higher than on other systems.[1]

For the root schedulers one of these methods can be implemented directly. For other schedulers which get their CPU time from other schedulers (e.g. root schedulers) CPU accounting becomes a little more complicated. Because the CPU time which is donated to the schedulers are already ”virtual” and cannot be measured accurately in a time-stamp based accounting mechanism. For example, in Figure 2, if scheduler S1 measures the CPU usage time from thread T2 they add the CPU time from the high-priority thread T0 when S0 preempts T1 and donate the CPU time to T0 and S1 don’t recognize this. In most cases, this inaccuracy may be ignored because of the assumption that high-priority threads

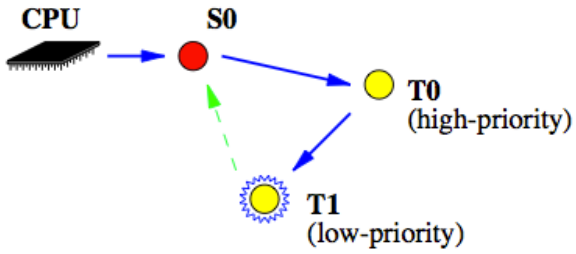


Fig. 3. Example of priority inheritance[1]

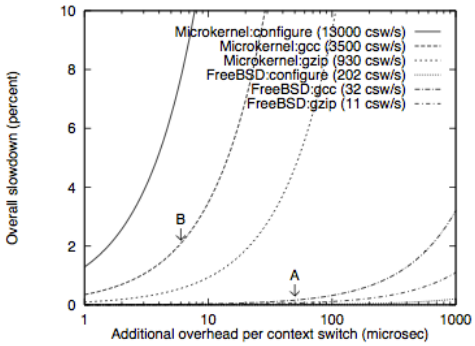


Fig. 4. Over all slow down related on scheduling overhead [1]

consumes relatively little CPU time. Otherwise, in the case these inaccuracies can't be ignored each scheduler thread need to take virtual time informations for his clients.[1]

2) Priority inversion: Priority inversion is the effect when a low-priority thread holds a lock on a shared resource while the high-priority thread is waiting for a message from the resource (e.g. answer of an I/O request). One solution to avoid this effect for user-level scheduling is to implement a priority inheritance mechanism.

Priority inheritance means that a high-priority thread can implicitly donate it's CPU time to a low-priority thread when this thread holds a resource which the high-priority thread needs. So the low-priority thread implicitly runs under the priority of the high-priority thread. Actually, you can say the thread inherits this priority. Figure 3 shows an Example for priority inheritance, S0 donates its CPU time to the thread T0 with a high priority. The low-priority T1 holds an resource which is needed from T0. T0 recognizes this and donates its CPU time to T1, so the thread can work with the resource under the priority from TO and release it after that. Increasing the priority of thread T1 to the priority of TO (inheritance of priority) is necessary to avoid that a middle-priority thread prevent or interrupt the execution of T1. After releasing the resource T1 also release the CPU time from T0, so T0 can continue its work.[1]

B. Scheduling Overhead

Up to this point we revealed the new potentials of user-level scheduling, but still they need to be surveyed to the extent to which they may be applicable in real world situations. In particular we will look at the efficiency and the implicit overhead produced by user-level scheduling compared to well-known traditional scheduling algorithms.[1]

There are two sources of overhead that need to be accounted for: in the first place there is the additional processing time inroduced by the dispatcher while it is processing the thread that will be switched to next after an event occurred. The overhead is produced due to iterating through linked lists and trees whose lenght depends solely on the scheduling hierarchies depth. Even if in theory the dispatcher can administrate an unlimited hierarchy depth of scheduler threads, in practice there is no need to apply/adopt so many scheduling levels since it creates a "source of unbounded priority inversion", as stated in section III-A2. To minimize computational overhead in hard real-time systems, for example, could limit the depth to eight or even four levels. Compared to just the root scheduler, 4-level scheduling would cause about twice as much processing overhead, 8-level scheduling even three times as much.[1]

The second source of overhead to be accounted for are the extra context switches between the several scheduler threads. As the impact of context switches heavily differs from system to system the design concept of the CPU inheritence model can be more or less expensive through various applications. The costs of context switches in user-level packages will be very subtle, but multiple times more significant in monolithic kernels for example.[1]

To get an idea of the real-world-performance that could be achieved by an implementation of the CPU inheritance framework, Figure 4 [1] shows some statistics that have been collected by Ford and Susarla during the measurement of different applications running in FreeBSD 2.1.5. The plot actually shows the tolerance of a system to scheduling overhead for a hypothetical L4-like microkernel. The statistics contain the compute- intensive application gzip compressing an 8MB file, the GNU C compiler gcc building a 20k-line programm, the I/O-intensive application tar copying 8MB of source files, and finally an extremely I/O and fork-intensive 3000-line Unix shell script configure.[1]

It is obvious that microkernels must be much less permissive to scheduling overhead due to the quantity of context switches that they perform. Even the L4 with its user-level device drivers would add two additional context switches per device interrupt. Assumed that all scheduling in FreeBSD was done in user mode, this would add roughly one more context switch for the scheduler invocation and hence keep the overhead negligible (shown in Figure 4 by arrow A). Taken gcc as an example (arrow B) the context switches introduced by the gcc must stay under 6 microseconds to keep the overhead under 2%.[1]

IV. View: Exokernel

Exokernels develop the idea of microkernels (Figuer 5 illustrates the architecture of mircokernels) one step further by outsourcing more mechanism, such as the scheduling mechanism, from kernel space to user space. From application perspective they have the goal to reduce several abstractions of the hardware. They try to avoid too many abstraction levels between hardware and the applications itself, making

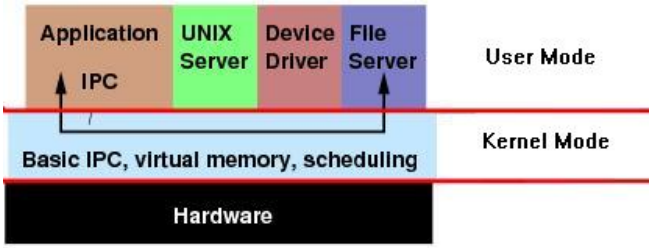


Fig. 5. Architecture of a mircokernel[2]

it possible to choose the best suited alternative without having unnecessary overhead. Speaking of abstractions, a typical example is the illusion of a filesystem that is created by regular operating systems to hide the real structure of the hard disc that, in reality, is build up from sectors and not just files. At the same level as the abstraction layer of the filesystem and the memory management, usually the security layer, suchlike Unix-style permission and ACLs (Access Control Lists), is located here also. To improve efficiency and meet the requirements, the scheduling of processor time is shifted into user-space as well, giving more flexibility and allowing adapted scheduling algorithms. Exokernels are structured in a way to provide application-specific customization. So it is easy to change the scheduling policy or to provide further concurrency models[6] such as workers, actors, or futures [5, 3, 4].

V. Conclusion

In this paper we present the idea of CPU inheritance scheduling as one possible solution for user-level scheduling. CPU inheritance scheduling is a simple way to implement user-level scheduling with low overhead. One positive ability of CPU inheritance scheduling is the possibility to create a tree-like hierarchy of schedulers. With this ability it is possible to implement individual scheduler policies for applications.

One of the reasons why CPU inheritance scheduling has low overhead is that the whole hierarchy of schedulers and their clients run in the same address space so the switching of client threads is cheap. But this is also a negative point because every thread of an application has access to the memory of all other applications. This is a critical security problem which is not acceptable.

To use the concept of CPU inheritance scheduling for more than one address space, one possibility is to define the hierarchy of schedulers and assign applications to one of them before building the operating system (e.g. in a XML-file or with a Lua script). Every thread which an application creates has to assign to the scheduler of the application. The schedulers are all using the same address space. So every application can use more address spaces. The switching of a client thread can invoke an address space switch. Switching the address space is very expensive (on some processors the TLB must be flushed and so on). Every time a scheduler gets CPU time an address space switch is necessary. This is the point why the concept of CPU inheritance scheduling respectively user-level scheduling is not used in operating systems yet.

References

[1] Bryan Ford and Sai Susarla, *CPU Inheritance Scheduling.* Department of Computer Science University of Utah Salt Lake City, 1996.

[2] http://www.brokenthorn.com/Resources/images/Microkernel.jpg from 26.01.2014

[3] AGHA, G. Actors, *A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass. 1986.

[4] HALSTEAD, R. Multilisp, *A language for concurrent symbolic computation.* ACM Trans. Program. Lang. Syst. 7, 4 (Oct. 1985), 501538.

[5] MOELLER-NIELSEN, P., AND STAUNSTRUP, J. Problem-heap, *A paradigm for multiprocessor algorithms.* Parallel Comput. 4, 1 (Feb. 1987), 6374.

[6] THOMAS E. ANDERSON, BRIAN N. BERSHAD, EDWARD D. LAZOWSKA, and HENRY M. LEVY, *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*, University of Washington,Oct 20, 1998

The long way towards usable IPC Performance in Microkernels

Richard Petri
University of Applied Sciences RheinMain
paper@rpls.de

ABSTRACT

IPC performance of first-generation microkernels used to be too poor to create fast systems which adhere to Liedtkes minimality priciple. In fact, commercial implementations of microkernels like Mach and early L3 versions started to deviate from this principle and incorporated the most performance critical servers/drivers into the kernel again. It was a long held believe, that IPC performance of microkernels was already at its optimum at approx. 100 microseconds (5000 CPU cycles at the time) for an empty message. But radical new approaches and tight hardware optimization all the way down to cache showed, that the real limit should have been 5 microseconds. This finally enabled completely new application scenarios.

1. INTRODUCTION

Microkernels began to pop up in the early 1980s. They were the first to achieve the separation of traditional operating system functions like drivers or filesystems from the common monolithic kernels. This enabled not only easier work on programming and experimenting with these components, since they were securely isolated from the rest of the system. Microkernels would also enable completely new applications, by allowing competing implementations of common operating system facilities like memory management running simultaneously on a single system. Probably the most commercially attractive new possibility of mikrokernels was the ability to let them serve as an easily portable machine abstraction on which other operating systems are built upon – not unlike modern virtualization techniques. This would allow the creation of new computer systems, which are compatible to common ones like UNIX or MSDOS – in this context called an "OS personality" – but simultaneously enabling engineers the usage of the completely new paradigms.

The main primitive used in interacting with microkernels and tasks running on them, a simple but powerfull interprocess communication (IPC) mechanism, quickly turned out to be a painfull bottleneck. IPC messages to drivers and other services were an order of magnitude slower than their systemcall counterparts in monolithic kernels, which severly degraded performance of common workloads. Due to this pressure, most OS developers saw their hand forced to deviate from most of the microkernel approach and reincorporated performance critical compontants into the kernel.

This paper shows the development-timeline of the IPC performance of microkernels. Section 2 compiles the first attempts at microkernel-based operating systems and how they deviated from an "ideal" microkernel environment. The subsequent section takes a look at the second generation of microkernels, detailing the techniques used to shrink the IPC latencies. This spans the first early attempts all the way to modern implementations. Section 4 then shows how a modern OS emulation works, using L^4Linux – a L4-based linux kernel – as an example. Section 5 concludes the paper with a discussion of new applications made possible by systems like L^4Linux and an outlook on modern mikrokernel applications.

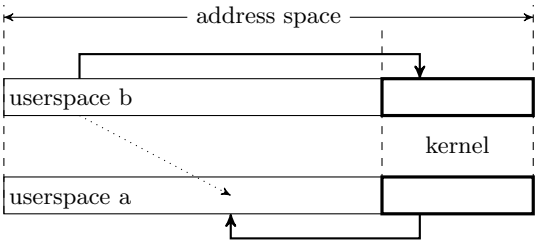
2. FIRST GENERATION MICROKERNELS

CHORUS [5] was an early experimental operating system kernel which explored the idea of a distributed, message-based, modular kernel in its first iterations V0 and V1. Beginning with its third iteration, V2, the authors wanted to explore the idea of adding a binary compatible UNIX emulation using a microkernel with a few isolated servers implementing device, file, network, and process management. While succeeding partially in their approach, the authors abandoned the idea of secure isolated servers in version V3. The authors argued, that while the concept seems elegant, the overhead of message passing between these most performance critical services prohibits Chorus to become commercially competative UNIX implementation. While the services were still handled similiar to normal user processes (e.g. paged, scheduled, ...), they were moved into a single address space and are always executed in the priviledged machine state.

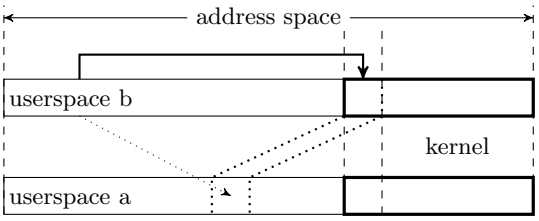
Mach [2] has a very similiar history to Chorus. The aim was to create an extensible kernel using simple abstractions with network transparency, and the possibility to emulate an OS personality like UNIX. The authors succeeded and [14] even went as far as implementing external memory management support, a technique self-evident in modern microkernels. However [7] takes a closer look at the performance of this approach by comparing the commercial UNIX implementation

OSF/1 with an integrated kernel to an OSF/1 implementation based on Mach. The authors meassured a performance degradation of up to 40% using the multi-user benchmark "AIM III". Previous observations, like [6], came to similiar conclusions by comparing Ultrix and Mach, with degradations depending on workload from 4% to 66%. As a countermeassure, [7] combined the servers responsible for the OS personality together in kernel-mode – a process coined "colocation". The process fixed the degradations but turned the system away from a clean mikrokernel approach.

Another promising approach was the L3 microkernel. The system described in [13] is built on top of the L3 microkernel and emulates a MSDOS personality. The system applies most of the concepts of microkernels: drivers and resource management are handled in processes outside of the kernel. Solely some specialized drivers are part of the kernel, like a SCSI driver or a driver emulating a RS232 port as a virtual DMA (which in turn is handled by user-space software). Memory management however was still part of the kernel, but plans already existed to implement this feature externally. But most importantly, the L3-kernel already partially featured the iconic IPC interface of the famous L3 descendant L4.



(a) Twofold copy through buffer.



(b) Direct copy through mapped communication window.

Figure 1: Direct message copy eliminates one copy process. Graphic adapted from [10]

3. THE BREAKTHROUGH

Even though the success of microkernels was stalled by the setbacks presented in section 2, work never stopped. Many microkernels brought isolated techniques for improved IPC performance, but it seemed as though a $100\mu s$ latency was a practical limit, which could only be beaten by faster hardware. Some analysis like [3] even suggested, that further work is futile because IPC performance is becomming irrelevant in the face of common workloads. The work of [10] effectively proved both hypothesises wrong. An improved version of L3 is presented, which is the first to combine most IPC improvements together with new radical restructurings of the microkernel.

Most of the changes concern the IPC mechanism itself. The smallest change was the expansion of IPC systemcalls send, reply, receive and wait by combined versions like "call" (send and receive) and "reply&wait". Since these are a very common use-cases, they save two systementry/exit calls (and address space changes) per call. And since the calling thread will immediately block anyway, a direct process switch is performed, donating the rest of the timeslice to the receiver. This direct switch however is not performed on the way back via reply&wait, if another thread is already waiting to deliver a message. The handling of the next message is prioritized, to raise the responsiveness of servers. The last change to the IPC mechanism itself was the way messages are copied to the receiver. Traditionally, messages were first copied from the senders address space into a buffer known to the kernel, then the address space was switched to the receiver, and the message copied into the receiving window. Simply mapping the data from the senders address space into the receiving window would have eliminated both copy-processes, but this would have security implications: the sender might change the message before the receiver was done, enabling a covert communication channel. One copy process however can be eliminated, by temporarily mapping the target region into a communication window, and copying the message into this window. The message will appear in the right place for the receiver due to the mapping. The process is illustrated in figure 1.

Other changes were less aimed at the concept of IPC, but on their technical implementation. Two big changes involved the bookkeeping of the kernel. One technique, coined "Lazy Scheduling", changed the way the process queues were maintained. Normally, a kernel would keep blocked/waiting and ready processes in queues and move communicating processes accordingly among theses list. But handling theses queues would (a) cost time, (b) cause cache misses, or even worse (c) cause TLB misses. Instead of keeping theses queues in perfect order, modern kernels just flag a calling thread waiting in the thread control block (TCB) and do the bookkeeping at the next time the queues have to be parsed (e.g. at the end of a timeslice during scheduling). This change works extremely well together with the direct process switch method. Other bookkeeping was also done lazily: the state of the 486-coprocessor is not saved on process switch. Instead it's just locked until another process tries to use it. The state is then saved for the previous process and unlocked for the current.

The rest of the changes concern the overall implementation. Most stalls were attributed to frequent cache- or TLB-misses. To lower the probability of these misses, the kernel working data was kept as small as possible (saving cache misses) and consolidated on as few pages as necessary (saving TLB misses). The authors even went as far as comming up with complicated schemes to store wakeup-times for sleeping processes. The same authors advocated in [11] to aim for small code and datasize of the kernel, citing the size of the L4-kernel code as slightly less than 12Kb (about 300Kb were common for first-generation kernels). Other technical changes aimed at raising processor utilisation and optimising for the common case. The IPC code is preceded by what in later publications would be called a "fastpath",

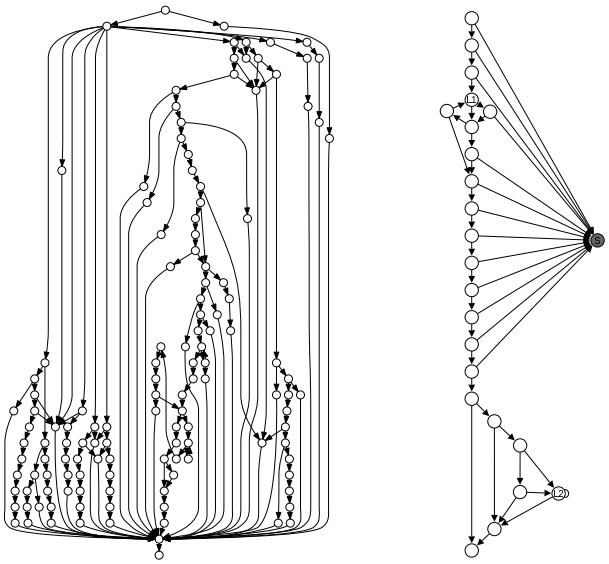


Figure 2: The flow-graph for IPC code. Left side shows the "slowpath". Right side shows the "fastpath", with the grey node leading to the "slowpath". Graphic taken from [4]

which contains optimised code for common IPC cases – for example a simple message by send&receive to another process. This code is very small, and mostly checks if the current IPC is indeed the common case, jumping to the full IPC code (called "slowpath") if it is not the case. Figure 2 shows a flow-graph for common IPC code. The slowpath contains a complicated checks with a lot of branches and long jumps, while the fastpath almost only contains branches which jump to the slowpath. If the fastpath is successful, the processor never branched and its pipeline stayed near full utilisation during the whole call.

Some early versions of the L4-kernel were even completely implemented using hand-written assembly language. This code was better than code generated by contemporary compilers, and didn't need to adhere to any calling conventions. Liedtke, the author of L4, argued in [12], that kernels are just something inherently hardware-dependant. This approach however was quickly abandoned, as modern incarnations of microkernels are usually implemented in C/C++. Only the most performance-critical sections, like the IPC fastpaths, stayed as hand-written assembly for a long time. Modern compilers however, allow the replacement of even this code. In [4], this approach is evaluated. Fastpaths are implemented in C, with the code-generation of compilers in mind, e.g. by moving loads or annotating code with compiler specific functions like "`__builtin_expect`"[1]. Unavoidable assembly, which cannot be expressed in C (e.g. toggling hardware interrupts), is placed into inline-assembly sections, which most compilers support.

The combination of all the named changes to the L3-kernel, caused a very drastic improvement in IPC performance. Table 1 lists the timings for messages of various lengths. The final speed for a simple "ping-pong" message was $5\mu s$ using an Intel 486 DX-50 processor. The $100\mu s$ barrier, which seemed unbreakable at the time, was distinctively broken. In [10] some insight is offered into how these numbers are accomplished. Lazy scheduling and the shrinking and optimising of data structures have the most noticable impact for smaller messages. It's not clear how much effect each trick has on the IPC performance, since a lot of these tricks have synergetic effects. For larger messages however, the overhead of any IPC work becomes very small relative to the copy-process of the message. Due to this, the direct copy optimisation has the biggest impact on performance when sending big messages.

	Size (Byte)			
Kernel	8b	12b	128b	512b
Mach	115	115	124	172
L3	5.2 (4.5%)	7.6 (6.6%)	9.8 (8%)	18.1 (10%)

Table 1: Time consumed in μs for IPC of different message sizes. Results taken from [10].

4. EMULATING AN OS ON L4

Section 3 showed, how propper IPC performance can be achieved. The shown numbers finally enable a fully microkernel based OS personality emulation. [8] offers a detailed look at just how this can be accomplished. The authors developed a special version of the Linux kernel called "L^4Linux", which can turn a L4-kernel into a Linux binary-compatible operating system.

A pure microkernel like L4 doesn't need any special facilities to emulate a system like Linux (or any other POSIX system). The IPC facilities play the most important role in this. Memory management (e.g. paging), interrupt- and exception handling are all represented by IPC messages. Any L4 task can be set as the pager or exception handler. The IPC system is enough to serve as a virtual platform for a Linux-kernel. Linux is an optimal candicate for porting, as most of its code is architecture-independent. Process- and resource management, file systems, networking and even device drivers[2] are all completely independent. The architecture-dependent part encapsulates the address space construction mechanisms, process interaction (especially the system call interface) and some very low-level drivers (e.g. DMA). All these mechanisms are however already implemented by the L4-kernel, so the architecture-dependent part of Linux for L4 contains code which accesses these mechanisms on L4. The Linux-kernel then is simply a task runnning on L4. The regular system-call interface of Linux is replaced with code using the L4 IPC interface.

To understand how binary-compatibility for between a regular Linux and the L4 kernel can be achieved, a look at the system-call mechanism of Linux/POSIX systems is neccessary. All system-calls have an assigned number/ID and a number of arguments. When a process wants to perform a system-call, it places the system-call number in a designated register, places the arguments on the stack[3] and performs

[1]This function allows the compiler to optimise pipeline/branch-prediction utilisation of the target processor

[2]Appart maybe from the fact, that some devices only exist on some architectures

[3]On Linux the arguments are placed in registers too to speed

a privileged instruction (most processors have a designated TRAP instruction). This causes an interrupt to be fired, which the kernel needs to handle. The interrupt handler in this case examines the system-call registers and performs the call. If a L4-thread performs a privileged instruction, an exception is raised. This in turn will cause the creation of an IPC message, which is sent to the registered exception handler of the thread. So all a L^4Linux has to do to run normal Linux binaries, is to simply run them as normal L4-tasks and register an exception handler which checks the exceptions that these processes cause. If the exception looks like normal system-call behaviour, the exception handler translates the call to a regular IPC-message which is sent to the L4-linux kernel. In literature, this exception handler is called a "trampoline". This trampoline however is only required for strict binary-compatibility. Linux binaries which are linked against a dynamic library version of libc, can simply be linked against an adapted version, which instead of Linux system-calls creates direct IPC messages to the L^4Linux kernel, bypassing the trampoline.

The authors of L^4Linux performed a microbenchmark to meassure the overhead of such a trampoline and the adapted libc-library. The benchmark involved meassuring the time a `getpid`[4] system-call needs. While a regular Linux kernel needed $1.68\mu s$, the a L4-version required $3.95\mu s$ using the adapted libc, and $5.66\mu s$ using the trampoline mechanism. While this 2.4x markup seems big at first, it quickly vanishes in macrobenchmarks, which reveal the true cost to be a 6-7% performance regression.

5. POSSIBLE APPLICATIONS

While it seems unproductive at first, to emulate an OS personality, or porting a whole monolithic kernel to a microkernel, all while suffering (albeit nowadays small) performance regressions, this approach opens up interesting new applications. One of the early reasons was raising the portability of operating systems. A microkernel encapsulates most of the hardware-dependant parts of an OS, while the rest (e.g. L^4Linux, Linux-applications, etc.) is mostly hardware-independent, apart maybe the need for recompilation if the processor architecture is changed. Due to this, a hardware change only requires the retargeting of the microkernel, which is fairly small. The rest of the system just needs to target the microkernel and can be ported straight away. Another early application of microkernels was transparent distribution of the systems to multiple computers. An application can't see, if it's exchanging IPC messages with its target directly, or if it's sending these to a proxy-task, which sends messages back and forth over a network to another computer.

These elegant software engineering tricks are not the only "pros" on the list for microkernels. Since the codebase for microkernels is very small, the code of such a kernel isn't typically larger than 10000 lines of C/C++. This codebase is much easier to check, and – as [9] showed – can also be formally proven correct. This opens up a lot of possibilities for security applications. One possible application for example are cryptographic modules running isolated from the rest of the system. A L4-task could perform encryption and keep the keys stored securely, providing this service to existing applications running on a L^4Linux kernel. Assuming the correctness of the microkernel, this encryption-service could never be compromised by applications, even if the Linux-kernel is completely exploitable. A similar approach is used by [1] to keep the security functions of an encrypting cellphone isolated from the Android OS running on a L^4Linux.

up the call, if the system-call has less than 6 arguments

[4]One of the simplest system-calls.

6. CONCLUSION

This paper showed a timeline of the development of microkernel systems. Section 2 listed three early examples of deployed UNIX/MS-DOS systems, which used microkernels as a foundation. Most of these systems used a relatively "pure" microkernel before before abandoning their efforts due the to performance problems exhibited by contemporary microkernels. Section 3 showed the necessary changes to a microkernel, which caused an unexpected breakthrough in performance. Section 4 showed how a common operating system can be ported to use a microkernel as a virtual platform, while maintaining binary-compatibility. While the performance penalty of emulating an OS personality on modern microkernels is fairly limited, systems like these remain a rare sight, usually limited to specialised security applications, as discussed in Section 5.

7. REFERENCES

[1] Hochsicherheitshandy der Telekom erhält BSI-Zulassung. URL: `http://www.telekom.com/medien/loesungen-fuer-unternehmen/200140` [cited January 2014].

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings Usenix Summer'86 Conference*, pages 93–113, Georgia, June 1986. Alaxta.

[3] B. N. Bershad. The increasing irrelevance of ipc performance for micro-kernel-based operating systems. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 205–212, Berkeley, CA, USA, 1992. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=646405.692226`.

[4] B. Blackham and G. Heiser. Correct, fast, maintainable: Choose any three! In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 13:1–13:7, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2349896.2349909`, `doi:10.1145/2349896.2349909`.

[5] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. A new look at microkernel-based unix operating systems: Lessons in performance and compatibility. Technical report, Chorus systemes, February 1991.

[6] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 120–133, New York, NY, USA, 1993. ACM. URL: `http://doi.acm.org/10.1145/168619.168629`, `doi:10.1145/168619.168629`.

[7] M. Condict, D. Bolinger, D. Mitchell, and M. Eamonn. Microkernel modularity with integrated

kernel performance. Technical report, Open Software Foundation, June 1994.

[8] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 66–77, New York, NY, USA, 1997. ACM. URL: http://doi.acm.org/10.1145/268998.266660, doi:10.1145/268998.266660.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1629575.1629596, doi:10.1145/1629575.1629596.

[10] J. Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM. URL: http://doi.acm.org/10.1145/168619.168633, doi:10.1145/168619.168633.

[11] J. Liedtke. Microkernels must and can be small. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*, IWOOOS '96, pages 152–, Washington, DC, USA, 1996. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=851041.856946.

[12] J. Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, Sept. 1996. URL: http://doi.acm.org/10.1145/234215.234473, doi:10.1145/234215.234473.

[13] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two years of experience with a μ-kernel based os. *SIGOPS Oper. Syst. Rev.*, 25(2):51–62, Apr. 1991. URL: http://doi.acm.org/10.1145/122120.122124, doi:10.1145/122120.122124.

[14] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 63–76, New York, NY, USA, 1987. ACM. URL: http://doi.acm.org/10.1145/41457.37507, doi:10.1145/41457.37507.

Side channel attacks in a microkernel environment

Fabian Seiberling
Hochschule RheinMain
Wiesbaden, Hessen
Fabian.b.Seiberling@student.hs-rm.de

Thomas Frase
Hochschule RheinMain
Wiesbaden, Hessen
Thomas.b.Frase@student.hs-rm.de

ABSTRACT

This paper gives an introduction to the topic of side- channel attacks. After an explanation of basic cryptographic primitives, we show two successful side-channel attacks that might also be used against cryptography in a microkernel environment. The first attack works in a virtual machine environment. While the second attack uses sound recorded from a smart phone.

1. INTRODUCTION

Side-channel attacks are a common problem everyone faces when implementing cryptographic algorithms. This paper gives an introduction to the topic of side-channel attacks and their application to virtual machines (and by extension, microkernels).

To understand what side-channel attacks are and why they work, you need to know the basic principles of public key cryptography and their underlying mathematical implementations.

The second section of this paper explains the RSA algorithm for public key cryptography and the square-and-multiply algorithm used to quickly calculate exponentiation of large numbers modulo a prime number. The third section explains what a side channel attack is, and how it relates to the implementation of the square-and-multiply algorithm. The last section shows two successful side-channel attacks that might also work in a microkernel environment.

2. PUBLIC KEY CRYPTOGRAPHY

The main feature of public key cryptography is the public/private key pair.

The public key, as the name suggests, can be published for everyone to see. The public key is required to either send messages to the owner of the private key, or to check signatures made with the private key.

The private key must be kept private. It is usually stored in encrypted form (using a symmetric cipher such as AES). The private key is used to decrypt data that was encrypted with the public key. It can also be used to sign messages by encrypting a suitable hash value.

Leaking the private key (or parts of it) compromises the public key encryption. Everyone in possession of the correct private key can decrypt and sign messages. Knowing parts of the private key might significantly reduce the search space for finding the private key.

The RSA algorithm large numbers as key pairs. The minimal recommended key size is 1024-bits. The basic encryption operation is as follows:

$$c \equiv m^e \pmod{n}$$

where m is the message, e is the public key exponent, n is the modulus (a product of two large prime numbers), and c is the ciphertext.

The decryption is as follows:

$$m \equiv c^d \pmod{n}$$

where c is a previously created ciphertext. d is the private key exponent (the multiplicative inverse of $e \pmod{n}$), and n is the modulus.

While the public key exponent is usually small (a popular value is 65537), the private key exponent is usually about the same size as n. One efficient way to quickly compute the exponentiation is the so called square-and-multiply algorithm.

The square-and-multiply algorithm is based on the following observation:

$$x^n = \begin{cases} x \cdot (x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even} \end{cases}$$

For example the exponentiation 2^{13} might be rewritten as $(((2)^2)^6 \cdot 2)$ (because the exponent 13 is odd). The result can be finally reduced to $((((2)^2 \cdot 2)^2)^2 \cdot 2)$. The result is a series of exponentiation and multiplication. The intermediate results can be reduced mod n if required.

As an additional example, the square-and-multiply algorithm

is shown in Algorithm 1.

Data: x, e, N
Result: y
let $e_n, ..., e_1$ be the bits of e;
$y \leftarrow 1$;
for $i = n$ *down to* 1 **do**
  $y \leftarrow$ Square(y);
  $y \leftarrow$ ModReduce(y, N);
  if $e_i = 1$ **then**
    $y \leftarrow$ Mult(y, x);
    $y \leftarrow$ ModReduce(y, N);
  end
end

Algorithm 1: The square-and-multiply algorithm

For more details, see [2, 5, 3].

3. SIDE-CHANNEL ATTACKS

Side-channel attacks use the physical implementation of a cryptographic function to gain information about the secret. It does not use a weakness of the algorithm, but uses only the information from other sources of the system, on which the cryptosystem is running.
The most common types of side-channel attacks are:

- Acoustic cryptanalysis - attacks which use the noise emitted by the computer while using the cryptographic function.
- Data remanence - attacks which gain information about the secret from the data which was used by a cryptographic function. It does not matter whether the data resides in memory, on the hard disk or on another storage medium. The data can be restored after the cryptographic function delete them.
- Differential fault analysis - this attack creates a fault in the cryptographic function to gain information about the current state of the function. A fault can be created with high temperature, to high or low voltage or with electric or magnetic fields.
- Electromagnetic attacks - attacks which use the electromagnetic field to gain information about the secret of the cryptographic function.
- Power monitoring attack - this attack used the characteristic of the power consumption for each instruction of the CPU.
- Timing attack - attacks which measure the execution time of parts of the cryptographic function to gain information.

For example the Power monitoring attacks can be used to gain the secret out of the square-and-multiply algorithm. The Power Monitoring Attacks can be divided into two categories: Simple Power Analysis (SPA) and Differential Power Analysis (DPA). To explain the attack, we use the same example as in the previous section.
2^{13} which can also be written as $((((2)^2 \cdot 2)^2)^2 \cdot 2)$.
The processor consume a different amount of power for the square operation than for the multiply times 2 operation. With a digital oscilloscope the secret can be read from the difference of the power consumption of the processor. In this example it show a short, long, short, short consumption which is represented binary as a 1011. If we want the decimal value of the secret, we must read the binary value from right to left, which will be a 1101_2 or 13_{10}. See figure 1. This was the Simple power analysis attack.

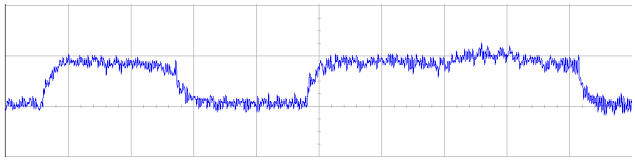


Figure 1: Power variations on an embedded processor. Source: [4]

> Power variations, observed during work of the embedded processor, computing RSA signatures. The left (short) peak represents iteration without multiplication (key bit is cleared), and the right represents iteration with multiplication (key bit is set). The low power pause between iterations has been artificially implemented to make key decoding trivial. This would be more complex on the real world devices that, differently, try to obfuscate it. See world devices that, differently, try to obfuscate it. See figure 1.

The Differential Power Analysis attack uses the same method as the Simple Power Analysis to gain information about the secret. The DPA use statistical methods to filter out noise, which can be created by other processes. To use the DPA attack, it is necessary to receive the power consume of the cryptographic function with the same secret several times.

4. REAL WORLD EXAMPLES AND APPLICATION TO MICROKERNELS

Now that we've looked at the basics of cryptography and the theory behind side-channel attacks, it's time to examine two recent attacks that affect virtual machines (and eventually microkernels).

The articel "Rsa key extraction via low-bandwidth acoustic cryptanalysis" [1] shows how it is possible to extract the private key with just a microphone recording in the vicinity of the computer that does the decryption. It is a form of acoustic cryptanalysis and to some extend also a power monitoring attack. The power consumption is the main factor of the acoustics that result in leaking the private key.

One interesting attack scenario shown in [1] is "self-eavesdropping". With "self-eavesdropping", the attacking application is listening to the device that it is running on. The goal is to obtain the private key of another application running on the same device.

Self-eavesdropping can be used on virtual machines and

microkernels. The only privilege that a process needs is access to the microphone. Because both attacker and victim are running on the same device, there is no time constraint to the execution of the attack. If decryption and encryption is provided by a service process in a microkernel, it is also possible that the attacker listens on the microphone while sending chosen ciphertexts to the decryption service.

What can be learned from this attack? The microphone should be considered as a security critical hardware and as such should only be accessible by trusted processes. Other than that, there probably isn't much a kernel can do to prevent the attack. Countermeasures must be implemented at the algorithm level of cryptographic software.

The second article "Cross-VM side channels and their use to extract private keys" details an "access driven" attack running inside a virtual machine, targeting another virtual machine running in parallel. The attack is targeted at a specific version of GnuPG and the Xen hypervisor. The attack works by allocating a continuous segment of memory and accessing that memory in a specific pattern to fill the processor cache lines. Observing timings, the attacking process can figure out which cache lines were evicted. This information is then used to gain knowledge from the GnuPG process.

Apart from the specific targeting of a cryptographic software and hypervisor, this attack might also work in a microkernel environment. The attack doesn't need any special privileges. It just needs a virtual machine (or microkernel process) running on the same machine as the other virtual machines or processes. In a hypothetical attack, this could be used to extract keys in a cloud based virtualization scenario.

The cross vm side-channel attack, conducted by Zhang, et al. [6] is the first published (so called) "access-driven" attack that is accurate enough to extract data that can be used to reconstruct a private key.

The attacking virtual machine measures level 1 cache timings in a so-called "prime-probe protocol" where continuous memory pages are allocated and divided into blocks each corresponding to a cache line. Issuing jump instructions to each page forces the addresses into the cache.

Depending on the instructions that the victim executes, some of the cache lines are replaced. Once the attacker is rescheduled, the time to access each allocated page changes, which in turn can be used to determine which instructions were executed.

There were several obstacles that the authors faced:

- The Xen hypervisor periodically reschedules virtual CPUs (VCPUs) to different physical CPUs (PCPUs).
- Processes or VMs other than the victim might be scheduled to execute.
- "Noise" introduced by task switches, other VMs, etc.

The authors found ways to circumvent or diminish those problems. The Xen hypervisor has a high priority for so-called Interprocess Interrupts (IPIs). A second VM effectively spams the attack process with IPIs, with the effect that Xen preempts the victim and schedules the attacker. This allows the attacker the frequent measurement of the cache timings.

The authors chose a support vector machine (SVM) to classify the cache timings. The SVM was trained with a carefully crafted environment that mirrored the targeted attack scenario.

The SVM output was further filtered using a hidden markov model. In the end, the private key fragments from the hidden markov model were stitched together with custom algorithms based on those that are used to reconstruct DNA data.

After the stitching process, the amount of uncertain bits was low enough to allow a brute-force attack to reveal the missing bits. According to the authors, the search space was only 9,832 keys.

While the attack was targeted specifically at Xen and GnuPG, the basic principle can be applied to various systems and software. Zhang, et al. suggest some countermeasures to avoid this kind of side-channel attack, none of which are new:

- Avoiding co-residency. Use a dedicated computer for high-security tasks, isolated from other tasks.
- Use side-channel resistant algorithms.
- Core scheduling. Sacrifice low latency scheduling for improved security.

What can we learn from this attack? How does it relate to microkernels? Even though the attack is targeted at a virtual machine / cloud environment, it could be applied to a microkernel environment.

Many microkernels might still be running in a single-core setting. This mostly eliminates the need to manipulate process scheduling. Compared to a virtualized cloud environment, a modern microkernel probably pollutes the L1 cache much less when switching tasks, which makes the timing measurements more accurate.

One solution might be to flush the cache on each context switch, at the cost of a performance. Instead, security critical services could be tagged as such. The microkernel only flushes the cache when a context switch occurs to or from such a critical process. The countermeasures suggested in [6] also apply.

5. CONCLUSION

There are many side-channel attacks that can be used to attack computer systems. Many of them apply to microkernels as well as every other type of kernel. Some might be mitigated by kernel-level measures, such as flushing the cache or restricting access to resources such as the microphone. For other attacks, it is probably best to fix the security critical software, that is the crypto-algorithms.

6. REFERENCES

[1] D. Genkin, A. Shamir, and E. Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. Cryptology ePrint Archive, Report 2013/857, 2013. http://eprint.iacr.org/.

[2] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2000.

[3] Wiki. Exponentiation by squaring. Online, January 2014. http://en.wikipedia.org/wiki/Exponentiation_by_squaring.

[4] Wikipedia. Power attack. Online, March 2010. http://en.wikipedia.org/wiki/File:Power_attack.png. Colors inverted for print.

[5] Wikipedia. Rsa algorithm. Online, January 2014. http://en.wikipedia.org/wiki/RSA_%28algorithm%29.

[6] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.